Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms

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

dem

Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg

vorgelegt von

Oleg Lobachev

aus Gießen

Marburg/Lahn, 2011

Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms

Oleg Lobachev

4th November 2011

Vom Fachbereich Mathematik und Informatik der Philipps-Universität Marburg
als Dissertation am *17.8.2011* angenommen.
Tag der mündlichen Prüfung: *11.10.2011*.

Erstgutachterin: Prof. Dr. Rita Loogen
Zweitgutachterin: Prof. Dr. Yolanda Ortega Mallén

Abstract

This thesis presents design and implementation approaches for the parallel algorithms of computer algebra. We use algorithmic skeletons and also further approaches, like data parallel arithmetic and actors. We have implemented skeletons for divide and conquer algorithms and some special parallel loops, that we call 'repeated computation with a possibility of premature termination'. We introduce in this thesis a rational data parallel arithmetic. We focus on parallel symbolic computation algorithms, for these algorithms our arithmetic provides a generic parallelisation approach.

The implementation is carried out in Eden, a parallel functional programming language based on Haskell. This choice enables us to encode both the skeletons and the programs in the same language. Moreover, it allows us to refrain from using two different languages—one for the implementation and one for the interface—for our implementation of computer algebra algorithms.

Further, this thesis presents methods for evaluation and estimation of parallel execution times. We partition the parallel execution time into two components. One of them accounts for the quality of the parallelisation, we call it the 'parallel penalty'. The other is the sequential execution time. For the estimation, we predict both components separately, using statistical methods. This enables very confident estimations, although using drastically less measurement points than other methods. We have applied both our evaluation and estimation approaches to the parallel programs presented in this thesis. We haven also used existing estimation methods.

We developed divide and conquer skeletons for the implementation of fast parallel multiplication. We have implemented the Karatsuba algorithm, Strassen's matrix multiplication algorithm and the fast Fourier transform. The latter was used to implement polynomial convolution that leads to a further fast multiplication algorithm. Specially for our implementation of Strassen algorithm we have designed and implemented a divide and conquer skeleton basing on actors. We have implemented the parallel fast Fourier transform, and not only did we use new divide and conquer skeletons, but also developed a `map-and-transpose` skeleton. It enables good parallelisation of the Fourier transform. The parallelisation of Karatsuba multiplication shows a very good performance. We have analysed the parallel penalty of our programs and compared it to the serial fraction—an approach, known from literature. We also performed execution time estimations of our divide and conquer programs.

This thesis presents a parallel `map+reduce` skeleton scheme. It allows us to combine the usual parallel `map` skeletons, like `parMap`, `farm`, `workpool`, with a premature termination property. We use this to implement the so-called 'parallel repeated computation', a special form of a speculative parallel loop. We have implemented two probabilistic primality tests: the Rabin–Miller test and the Jacobi sum test. We parallelised both with our approach. We analysed the task distribution and stated the fitting configurations of the Jacobi sum test. We have shown formally that the Jacobi sum test can be implemented in parallel. Subsequently, we parallelised it, analysed the load balancing issues, and produced an optimisation. The latter enabled a good implementation, as verified using the parallel penalty. We have also estimated the performance of the tests for further input sizes and numbers of processing elements. Parallelisation of the Jacobi sum test and our generic parallelisation scheme for the repeated computation is our original contribution.

The data parallel arithmetic was defined not only for integers, which is already known, but also for rationals. We handled the common factors of the numerator or denominator of the fraction with the modulus in a novel manner. This is required to obtain a true multiple-residue arithmetic, a novel result of our research. Using these mathematical advances, we have parallelised the determinant computation using the Gauß elimination. As always, we have performed task distribution analysis and estimation of the parallel execution time of our implementation. A similar computation in Maple emphasised the potential of our approach. Data parallel arithmetic enables parallelisation of entire classes of computer algebra algorithms.

Summarising, this thesis presents and thoroughly evaluates new and existing design decisions for high-level parallelisations of computer algebra algorithms.

Abstract (Deutsch)

Die vorliegende Arbeit beschreibt den Entwurf und die Implementierung paralleler Computeralgebraalgorithmen mithilfe von algorithmischen Skeletten und weiteren Ansätzen, wie datenparallele Arithmetik und Actors. Wir implementieren algorithmische Skelette für Divide and Conquer und spezielle, spekulative parallele Schleifen. Ähnlich zu den Skeletten ist die rationale datenparallele Arithmetik ebenfalls ein Parallelisierungschema, allerdings ist dieses speziell für Algorithmen der Computeralgebra ausgelegt.

Die Parallelisierung wird in der parallelen funktionalen Sprache Eden vorgenommen. Eden ist eine Erweiterung von Haskell. Mit Eden ist es möglich, sowohl die Skelette als auch die Programme in einer Sprache zu definieren. Ein ähnlicher Ansatz erlaubt uns, die „innere“ und die „äußere“ Sprache einer Computeralgebra-Implementierung zu vereinigen.

Desweiteren schlägt diese Arbeit eine Methode zur Auswertung und Abschätzung der parallelen Laufzeiten vor. Im Gegensatz zu den gängigen Ansätzen werden statistische Verfahren verwendet. Dadurch können wir sehr überzeugende Vorhersagen liefern. Im Vergleich zu den anderen Verfahren verwenden unsere Methoden deutlich weniger Datenpunkte. Eine von uns eingeführte Komponente der parallelen Laufzeit erlaubt eine strikte Kontrolle der Güte der Parallelität. Wir führen sowohl die Qualitätsmerkmale als auch die Abschätzung der parallelen Laufzeit für die in der Arbeit vorgestellte Implementierungen aus.

Mit den Divide and Conquer Skeletten werden die Algorithmen für schnelle Multiplikation, nämlich Karatsuba Algorithmus und Strassen Matrixmultiplikation sowie die schnelle Fouriertransformation, entwickelt. Letzteres wird auch mit einem `map-and-transpose` Skelett umgesetzt. Damit haben wir eine gute Parallelisierungmerkmale beobachtet. Wir verwenden die schnelle Fouriertransformation, um eine schnelle Faltung von Polynomen umzusetzen, was zu einem weiteren Algorithmus zur schnellen Multiplikation führt. Der parallele Algorithmus von Karatsuba zeigt sehr gute Parallelisierungmerkmale. Für den Algorithmus von Strassen implementieren wir ein Actors-basiertes Divide and Conquer Skelett. Wir analysieren die Messungen ausführlich, um die Güte der Parallelisierung sicherzustellen.

Wir stellen ein `map+reduce` Schema vor. Es erlaubt, gängige parallele Skelette wie `parMap`, `farm`, `workpool` mit einer Abbruchsklausel zu versehen. Wir setzen mit diesem Skelettschema die Algorithmen zu probabilistischen Primzahlprüfung um. Wir implementieren den Rabin–Miller-Test und den Jacobi-Summen-Test. Bei dem ersten optimieren wir die Aufgaben- und Prozessorenanzahl. Bei dem Jacobi-Summen-Test beweisen wir zuerst, dass es überhaupt möglich ist, unseren Ansatz zu verwenden. Danach wird die Implementierung erzeugt und parallelisiert. Dabei nehmen wir eine Optimierung der Aufgabenreihenfolge vor, was die Speedup Werte deutlich verbessert. Ebenfalls werden auch hier die beiden Implementierungen ausführlich analysiert. Die Laufzeiten werden für weitere Problemgrößen und Prozessorenzahlen abgeschätzt.

Die datenparallele Arithmetik wurde nicht nur für ganze Zahlen, sondern auch für Brüche umgesetzt. Wir stellen eine neuartige Lösung vor, die die gemeinsamen Faktoren des Zähler oder Nenner des Bruches mit der Zahl eliminiert, modulo der man rechnet. Das ist notwendig um eine hinreichende Skalierung der multimodularen Arithmetik zu ermöglichen. Unter Verwendung dieser Erkenntnisse haben wir die Determinantenberechnung mittels Gauß Elimination parallelisiert. Eine vergleichbare Berechnung in Maple zeigte die großen Vorteile unseres Ansatzes.

Die vorliegende Dissertation präsentiert Implementierung und gründliche Auswertung von high-level Parallelisierungen verschiedener Computeralgebraalgorithmen.

To my mother

ACKNOWLEDGEMENTS

FIRST of all I would like to thank my thesis adviser Prof. Dr. Rita Loogen for the invaluable collaboration and support. Her supervision and encouragement made this work possible in the first place. I am very grateful to Prof. Dr. Yolanda Ortega-Mallén for her deep and helpful comments. Further, I would like to thank my colleagues Dr. Jost Berthold, Mischa Dieterle, Prof. Dr. Michael Guthe and Thomas Horstmeyer for collaboration and discussions. Dominik Krappel, Kent Kwee, Bernhard Pickenbrock, Florian Pfeiffer and Steffen Vaupel supported the implementation of the codebase, which is used in this thesis. I was also lucky to have fruitful discussions with Paolo Giarrusso, Egbert Fohry, Kathi Haselhorst and Tillmann Rendel. I want to specially thank Tillmann for the discussion on monoids and many other very interesting conversations. I benefited from the collaboration with my colleagues in Scotland: I thank Prof. Dr. Phil Trinder and Dr. Abyd Al Zain for the possibility to use the Beowulf cluster of the Heriot-Watt University. Further, I thank Dr. Alexander Konovalov (University of St. Andrews) for his support. Dr. Siarhei Khirevich and Anton Daneyko from the Faculty of Chemistry at Philipps-Universität Marburg kindly provided me their run time measurements on a supercomputer. These helped me a lot.

A loud 'thank you' goes to Mark Schäfer and all my former colleagues at Identass GmbH & Co. KG for the possibility to work together in the first phase of this project. The material for this work emerged under the support of Deutsche Forschungsgemeinschaft under the grant number LO 630-3/1 and Hessian Ministry for Science and the Arts' (HMWK) LOEWE KMU programme.

I am very grateful to my family for all support and patience I enjoy. I deeply appreciate all and every kind of assistance, my parents give to me.

Oleg Lobachev

CONTENTS

Chapter 1

INTRODUCTION

HISTORICALLY seen, computing methods were numerical first. Plenty of tasks in scientific computing require representation of real or complex values and software-technically advanced data structures like polynomials or matrices. So the initial approaches were to use various *approximations* to $\mathbb{R}$. The most popular one since at least 50 years ago has been the *floating point* representation. With it we represent the first few most significant digits of a number—the mantissa—and their position as a power of the number system base—the exponent. This approach is inexact to the extent it gave birth to a whole branch of science: the numeric error analysis. One of the questions the latter strives to answer is: Given the largest initial error δ, i.e., the difference between the actual and the represented value, and performing a particular sequence of arithmetical operations, what is the figure for the final error, i.e., what is the difference between the result we should have obtained and the result we have actually computed in terms of δ? Numeric analysis was motivated by an example by Wilkinson (James Hardy Wilkinson, *27.9.1919, †5.10.1986).

Example (Wilkinson monster [1963]). *We perform a simple test of a polynomial equation solver. Consider a polynomial*

$$w(x) = \prod_{j=1}^{20} (x - j)$$

in a closed form—we computed the product—and in single-precision floating point representation. Then, if we search for the zeros of $w(x)$ with the classical Newton method, we obtain complex-valued *solutions. Two plots for the $w(x)$ are shown in Figure 1.1.*

The reason for such a behaviour of the Wilkinson monster is the inexact floating point representation and the numerical instability of the polynomials. This particular polynomial $w(x)$ has both very large and very small coefficients. Thus, its representation in floating point system becomes more and more distorted as the computation progresses. The usage of a single-precision arithmetic makes this process more outrageous. Let the distorted polynomial be $\widetilde{w}(x)$. The polynomials are very instable numerically, and the zeros of $\widetilde{w}(x)$ differ from the zeros of $w(x)$ to the extent that the former are complex and the latter are not—by construction.

What we in fact desire to obtain are methods for the exact computation. We call them from now on '*symbolic methods*' and consider them as an opposite of the numeric methods. However, the latter can and will serve as a 'guide' (in many senses) for our approaches. Still, numeric and symbolic computing methods differ. To give an example: it is very hard to compute Gröbner bases [Buchberger, 1965, Becker et al., 1993] with floating point numbers [Sasaki and Kako, 2007]. As symbolic methods are much more resource-demanding than numeric ones, *parallel* symbolic computing is a highly important research direction.

Let us perform a practical experiment. Start Maple [Redfern, 1995, Monagan et al., 2005], we took the version 13, available at Faculty of Mathematics and Computer Science at Philipps-Universität Marburg. Generate a large random matrix over rational numbers

```
with(LinearAlgebra):
A := RandomMatrix(100, 100, generator = rand(-99..99) / rand(1..99));
```

and compute its determinant

```
Determinant(A);
```

It took us 84.66 seconds on an 8-core machine `sakania`, until the full result was computed. Admittedly, it is a quite large fraction, as we demanded the *exact* result. However, we observed that while the computation was in progress, only one core of our mighty 8-core machine was in fact computing the

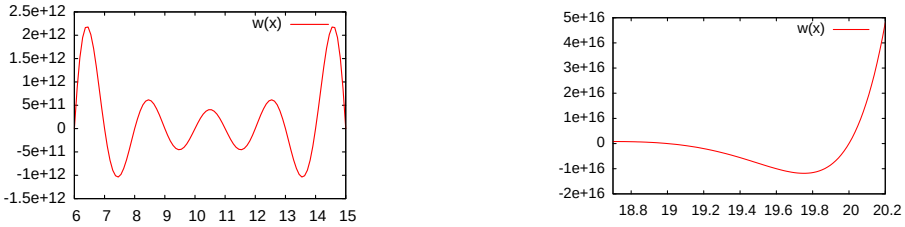


Figure 1.1: *The Wilkinson monster: the polynomial $w(x)$. Note the amplitude of the values.*

determinant, the remaining 7 cores were idle. So, if we could occupy these idle cores too, we would not need to wait almost one and a half minute for the result. We have seen that the parallelisation is very important for computer algebra methods.

We will consider a radically different method of computation, namely a residue-based rational arithmetic, in Chapter 7. We will revisit the Maple determinant computation example on pages 172–174. We will also consider existing algorithms of computer algebra and search for a generalisable high-level parallelisation techniques for these algorithms.

We work in the research parallel language Eden [Loogen et al., 2005], which enables us to use Haskell [Peyton-Jones, 2003] for specifying both the parallelisation methods and the code to parallelise. We will use ad-hoc polymorphism, higher-order functions, futures and further features of our target language. Our high-level approach will add such benefits as good abstraction from unneeded details and high productivity of the developer to the performance increase drawn from the parallelisation. Our goal is to uncover parallel processing approaches and techniques. We hope that some day, a parallel computer algebra system will emerge which will have utilised this knowledge to become faster. This would serve for the benefit of anyone who has ever been staring in the monitor, waiting for the computation to finish.

Related Computer Algebra Systems

Various (sequential) computer algebra systems exist. An overview is in [Grabmeier et al., 2003]. First systems include Macsyma [Martin and Fateman, 1971], Reduce [Hearn, 2004], Axiom [Bronstein et al., 2003], see also [Davenport, 1994]. Standard commercial systems nowadays are Maple [Redfern, 1995, Monagan et al., 2005] and Mathematica [Wolfram, 1991, 2000]. More special open source systems include CoCoA [Capani and Niesi, 1995, CoCoA, 2009], GAP [GAP, 2008], GiNaC [Bauer et al., 2002, GiNaC], KANT V4 [Daberkow et al., 1997], PARI [Batut et al., 2010], Singular [Greuel et al., 2001, Decker et al., 2011]. An open source bundle of available free systems with a Python-based 'glue' to hold them together is SAGE [Stein and Joyner, 2005, Stein et al., 2011].

Approaches towards *parallel* computer algebra as a complete system include PARSAC [Kuechlin, 1995] and PACLIB [Schreiner and Hong, 1993, Schreiner, 1994]. MuPAD [Naundorf, 1995], Macaulay2 [Eisenbud, 2002, Grayson and Stillman, 2010] and Cannes/Parcan [Gloor and Muller, 1997] feature SMP support using threads in an otherwise sequential system. This is a low-level approach. An early approach to parallel Maple is [Watt, 1986]. There are also techniques to separate parallel orchestration from computer algebra implementation, see, e. g., ‖Maple‖ [Siegl, 1993], Distributed Maple [Schreiner et al., 2003], SCIEnce Project [Hammond et al., 2007, SCIEnce, 2010].

1.1 Parallel Programming

Parallel programming, a large problem field in computer science, has gained much attention over the last few years. It is considered a new, important front line of the research due to the current trends of computer hardware development [Geer, 2005, Hill and Marty, 2008]. For a few years now chip makers have been not able to scale CPU performance by increasing speed of a single core. The only recourse is to maintain existing—and needed!—performance growth is to increase the number of processors on a single chip. This still reflects the Moore's law formulation for the number of the transistors on the same

die [Moore, 1998], but should the current development continue, we will face *thousands* of cores on a single die [Agarwal and Levy, 2007]. However, parallel programming has some caveats [Skillicorn, 1994, Foster, 1995, Grama et al., 2003]. It is much harder than its sequential counterpart. We need to take care of data partitioning, communication and synchronisation issues. Deadlocks are possible. Good speedups depend on many factors. It is hard to foresee the behaviour of a parallel program for further input sizes or number of processing elements.

We want to combine complicated algorithms and (hard) parallel programming. This requires high-level, abstract, generic programming schemes [Charles et al., 2005, Chamberlain et al., 2007]. We aim for language-independent techniques. Possible techniques include i) algorithmic skeletons [Cole, 1989], ii) data parallel approach [Grama et al., 2003], iii) actors [Hewitt et al., 1973, Haller and Odersky, 2009], iv) a monad for deterministic parallelism [Marlow et al., 2011], v) software transactional memory [Harris et al., 2010], vi) communicating sequential processes [Hoare, 1978, Abdallah et al., 2005]. This list includes both approaches towards concurrency and parallelism. We do not assess all of them equally detailed. We concentrate our interest on the parallel methods. We aim to model mathematical algorithms with as few side effects as possible.

- The main focus of this work is set on *algorithmic skeletons*. Skeletons capture parallelisation schemes and communication patterns. They provide a more formal framework for program construction [Gorlatch, 1998b, Rabhi and Gorlatch, 2003]. With skeletons it is possible to reuse the most complex part of a parallel algorithm: its parallelisation. Skeletons are often used in parallel functional languages, as they can be encoded as higher-order functions in this setting.

  These properties of algorithmic skeletons make them best candidates for a high-level parallelisation of multiple (formally stated) algorithms, divided in groups, sharing the same paradigm—exactly the context of this thesis. With skeletons we aim to save much effort for the parallelisation of the algorithms in our application area.

- We will use the *data parallel paradigm* from the point of view of algorithm transformations. We search for a better suitable arithmetic, which would limit the intermediate expression swell and enable parallelisation. This is a mathematical technique, which is easily applicable for the problems in computational mathematics. However, it is not a general parallel processing method. Our data parallel approach—after the algorithm transformation is done—would still require a skeleton to implement the obtained parallelism.

  Though there are techniques to flatten nested data parallel programs [Blelloch, 1996, Chakravarty et al., 2007], and thus to make data parallel approach more widely usable, we do not see such techniques to be versatile enough. We search for *mathematical* techniques to transform algorithms to data parallel representation and do not use nested data parallelism.

- We will also briefly discuss *actors* [Hewitt et al., 1973, Haller and Odersky, 2009]. The typical actor approach is very different from ours. We will see in Chapter 6 how it is possible to model actors in a high-level manner in the setting of a lazy functional language. Actors are an approach to *concurrency*, however, we will model them in a parallel setting.

- The approach of [Marlow et al., 2011], the *deterministic `Par` monad*, confines all the implementation in a monad. Our algorithmic skeletons would require the sequential code to use the (sequential) higher-order functions, which will be then replaced by skeletons. `Par` requires all the code to be monadic, which is a larger transition. Further, the preliminary measurements show [Marlow et al., 2011] that the `Par` monad is in most cases significantly slower than traditional Multicore Haskell [Marlow et al., 2009]. Therefore, we do not consider this approach to parallelism in more detail.

- A rather high-level approach to concurrency is *software transactional memory* (STM). First publications on this topic are [Knight, 1986, Herlihy and Moss, 1993], see also [Shavit and Touitou, 1995, 1997, Harris et al., 2010, Herlihy, 2010]. The basic idea is that transactions for memory access either succeed or fail. In the latter case, the STM implementation just tries again, until

success is reached. It can be used for lock-free data structures. STM is quite accepted in the industry, *cf.* [Adl-Tabatabai and Shpeisman, 2009], furthermore it can be nicely implemented in Haskell [Peyton-Jones, 2007, Harris et al., 2008, O'Sullivan et al., 2008]. STM can also be used to fix some possible inconsistencies [Bieniusa et al., 2010].

In our view, using STM for our aims has multiple drawbacks. Firstly, STM can be used only in the shared memory setting. We aim for techniques usable both in distributed and in shared memory settings. Networks of multicore machines need both approaches [El-Rewini and Abd-El-Barr, 2005]. Secondly, it is not a very high-level approach. STM can free the programmer from locking, but it still enforces a quite low-level programming model, involving transactions and atomic blocks. The third reason is the same as with `Par` monad: Haskell's implementation of STM is monadic. Hence, we will need to majorly restructure sequential code to use STM. And the most important reason is: STM is an approach for concurrency, not for parallelism. Contrarily to actors, we see no easy way to implement STM in the parallel functional setting. We do not regard STM in further, focusing on *parallel* high-level methods.

- The *communicating sequential processes* [Hoare, 1978, Abdallah et al., 2005] is a quite formal approach. In fact, it is a calculus for concurrency. Implementations include [Jones and Jones, 1987, Brown and Welch, 2003, Welch and Barnes, 2005, Brown, 2008]. We do not consider this model here. The reasons are twofold. Firstly, it is an approach towards concurrency. Contrarily to this fact, we focus on the parallel computing, as detailed above. Secondly, available Haskell implementation is monadic, hence we have similar drawbacks as in `Par` monad and STM.

1.2 Goals of This Work

We aim to answer the following question.

> *Whether and how can we utilise high-level approaches for the efficient parallelisation of the core algorithms of computer algebra?* Can we extract skeletons from common computer algebra algorithms? How do we make the symbolic computation possible in the chosen setting? Which skeletons do we need to implement? How can we ensure portability and reuse of our work? Are we able to apply the above mathematical technique not only in theory? How can we model actors in our setting?

This thesis proceeds in two directions towards the answers to these questions. Firstly, we contribute high-level parallelisations for some popular problems in computer algebra. Secondly, we introduce further evaluation methods, see below.

1.2.1 Parallel Symbolic Computation

- We require basic support for symbolic computations, i.e., a way to express computer algebra algorithms independently from concrete underlying algebraic structure. To give an example: not 'polynomials over $\mathbb{Z}$', but 'polynomials over a unique factorisation domain'. This thesis will present our approach to this issue.

  We need to represent values of arbitrary size, and not only the ones fitting in some hardware registers: unsigned hardware integers are bounded with 2^{64} on 64 bit hardware. We require an implementation of fractions. It is possible to represent them as pairs of integers. However, we embark on a different route. We will present an alternative approach to the representation of fractions, which maps well to a parallel processing paradigm. We need to represent not only rationals, but also irrational values exactly. The algebraic method is *adjunction*. With it we add the needed irrational value symbolically to the number system we work in. We will present our implementation of adjunction in this work.

- We contribute skeletons for parallel *polynomial* and *integer multiplication*. Our approach is to parallelise fast divide and conquer methods. We are interested in both Karatsuba multiplication and in methods based on the fast Fourier transform. The latter account for the complexity of the multiplication being $\mathcal{O}(n \log n)$ for the polynomials of degree n, provided some limitations hold. The consequence is: we implement the parallel fast Fourier transform. As it is also a divide and conquer algorithm, we will investigate to what extent it is possible to share a single divide and conquer skeleton between such very different algorithms.

- We contribute approaches for parallelising *matrix multiplication* and *decomposition*. We parallelise the fast matrix multiplication algorithm by Strassen [1969] (again a divide and conquer algorithm!) and the **LU** decomposition of matrices (also known as Gauß elimination). An important application of the latter is the determinant computation.

- In some algorithms of computer algebra, e.g., in Gauß elimination, naive polynomial GCD, Buchberger's algorithm, a very fast increase of the rational entries in the mathematical structures occurs. Examples are coefficients of polynomials for GCD and Gröbner bases, and entries of the matrices for Gauß elimination. Such growth is called intermediate expression swell [Tobey, 1966, von zur Gathen and Gerhard, 2003, Brickenstein, 2010].

  We require a method to *prevent the intermediate expression swell*—and to do so in parallel. We contribute a residue arithmetic for this goal. We present approaches for both integers and rationals. Further, it should be possible to compute with the residue arithmetic *in parallel*. An apparent approach is to use *multiple* residue classes a in data parallel manner. Some questions arise. It it correct? Residue classes are well known for the integer residues. How can we tackle fractions? Is it possible to use multiple residue classes for fractional residues? We will answer the above questions in this thesis. The approach towards a fractional multiple-residue arithmetic is the result of our long-term research.

- For the residue arithmetic we implement methods to find quite large primes. Even leaving the residues aside, *primality testing* is a very interesting and required discipline. Not least this is so because of public key cryptographic systems like RSA [Rivest et al., 1978], which require large prime numbers. We will *abstract the parallelisation patterns* (i.e., the skeletons) for probabilistic primality tests. We will also discuss and evaluate the performance of these methods.

1.2.2 Evaluation Methods

Secondly, after obtaining some results for the performance of our new-crafted implementations on the hardware available to us, we contribute methods to to i) evaluate thoroughly these results, ii) find and apply an approach to generalise these results. This means, we contribute

- A method to predict the execution time of a given program for non-measured input sizes and for non-measured numbers of processors.

- An ability to verify the quality of a parallel implementation quantitatively. We apply it to the parallel programs we will present.

1.3 Structure of The Thesis

The remaining text is structured as follows. Chapter 2 discusses the choice of a lazy functional programming language for the implementations in question. We also introduce the language unity approach in Chapter 2. Chapter 3 presents the parallel functional language Eden [Loogen et al., 2005], which is well suited for the development of the skeletons in the further chapters. We will see in further chapters why some features of Eden are essential for this thesis. Chapter 4 presents the methods for the estimation of parallel programs' run times mentioned above. These methods are our novel work. We examine

the probabilistic primality testing methods and develop new skeletal approaches for their parallelisation in Chapter 5. We consider fast multiplication routines for polynomials, integers, and matrices in Chapter 6. All algorithms in this chapter are divide and conquer algorithms. We will see how we can reduce the implementation overhead with skeletons. The same chapter features actors. A throughout evaluation is also included. A novel parallel rational residue arithmetic is presented in Chapter 7. The same chapter also presents an integer residue arithmetic. These implementations form the data parallel approach. Further, we present there an example using matrix decomposition and evaluate its performance with the new arithmetic. Conclusions, future, and related work follow in Chapter 8.

Chapter 2

PROGRAMMING LANGUAGES AND SYMBOLIC COMPUTATION

> There is no mode of action, no form of emotion, that we do not share with the lower animals. It is only by language that we rise above them, or above each other—by language, which is the parent, and not the child, of thought.
>
> Oscar Wilde, *The Critic as Artist*

Between 810 and 833 A.D. Muhammad ibn Mūsā al-Khwārizmī, *c. 780, †c. 850, wrote *Al-Kitāb al-mukhtasar fī hīsāb al-ğabr wa'l-muqābala* (الكتاب المختصر في حساب الجبر والمقابلة, "The Compendious Book on Calculation by Completion and Balancing"). It features the concepts of *al-ğabr* and *muqābala*—the symbolic computation and term reduction. This book's content and its author's name are also good known for giving us the words 'algebra' and 'algorithm' respectively [Rāshid, 1994, Chabert, 1999, Grabmeier et al., 2003]. Rafael Bombelli (*1526, †1572) published *Algebra* in 1572, introducing rules for computation with complex numbers. John Pell, *1.3.1611, †12.12.1685, tabelised in 1668 the factors of all integers up to 100 000. Leonhard Euler, *15.4.1707, †18.9.1783, significantly advanced algebra, number theory, and various other fields of mathematics. Another mathematical genius, Johann Carl Friedrich Gauß, was born 30.4.1777. He proved the fundamental theorem of algebra in 1799. His *Disquisitiones Arithmeticae* appeared 1801. It features advances in number theory and algebraic constructions. The latter part of this book utilises the complex roots of unity. Gauß died 23.2.1855 [Eves, 1969, Chabert, 1999, MacTutor, 2011].

John Napier (also: Neper, Nepair) of Merchiston, *1550, †4.4.1617, introduced Napier's bones, a kind of mechanical calculator. Napier is one of the discoverers of logarithms, alongside with Briggs and Bürgi. Edmund Gunter (*1581, †10.12.1626) develops Gunter's scale, a mechanic device capable of multiplication using logarithms, in 1620. This device was the predecessor of a slide rule. John von Neumann (*28.12.1903, †8.2.1957) and Norbert Wiener (*26.11.1894, †18.3.1964) produced the foundations of contemporary computing hardware. James Hardy Wilkinson, *27.9.1919, †5.10.1986, was one of the first scientists to raise awareness towards approximate methods and numeric computing—the finesse of the computation, adapted to the hardware of the digital computers [MacTutor, 2011].

The symbolic computation in the modern sense, as opposed to the numeric computation, is merely a few dozen years old: various sources state either [Kahrimanian, 1953, Nolan, 1953] or [Birch and Swinnerton-Dyer, 1963] as emerging publications on this topic. We generally use the terms 'symbolic computation' and 'computer algebra' as synonyms, for the fine difference between these concepts see, e.g., [Watt, 2006]. Symbolic computing is computationally very intensive not only because of the symbolic representation of data, but also because of some negative effects, e.g., intermediate expression swell [Tobey, 1966, von zur Gathen and Gerhard, 2003]. These facts motivate the development of parallel computer algebra systems. We will focus on parallel programming languages in Chapter 3. Here we discuss the (sequential) features of a language for symbolic computation. This chapter will provide the programming language foundation of this thesis.

We chose Haskell for our implementation of the algorithms of computer algebra. It is a statically typed lazy functional programming language [Peyton-Jones, 2003]. We will see below which features of Haskell make it especially suitable for our tasks. The first section discusses the problems, an efficient implementation of a computer algebra system needs to solve. The remaining part of this chapter is organised as follows. Section 2.2 discusses the language unity concept. Section 2.3 explains the type problems, connected with algebraic domain construction. Section 2.4 provides an example of symbolic

computing with Haskell. It emphasises the importance of laziness. Section 2.5 concludes the chapter. We will discuss the details connected with the implementation of *parallel* algorithms in the next chapter. We assume a good knowledge of Haskell [Peyton-Jones, 2003], see, e.g., [Doets and van Eijck, 2004, Hutton, 2007, O'Sullivan et al., 2008].

2.1 Symbolic Computation

So, we want to perform a symbolic computation. Systems doing so are called 'computer algebra systems', abbreviated CAS. As we move away from floating point arithmetic (and overflowing hardware integer arithmetic), the first step is an arbitrary precision integer arithmetic. However, the cost of a single arithmetic operation with such arithmetic is no longer constant—it depends on the length of the integers. (See Chapter 6 for fast multiplication routines.) The implementations of arbitrary precision integers include the GNU multiprecision library [Granlund and Swox, 2011] and the CLN library [Haible and Kreckel, 2008]. Also noteworthy is the NTL [Shoup, 2009], which, focusing on number theory, begins with implementing the foundations for it. An increasing number of modern programming languages, including Haskell, provide access to an implementation of arbitrary precision integers [Peyton-Jones, 2003, van Rossum, 2006, Flanagan and Matsumoto, 2008, Odersky et al., 2008]. We can take arbitrary precision integers for granted in these languages. Haskell with its approach to polymorphism is especially convenient. In Haskell the arbitrary precision integers are called `Integer`, as opposed to the hardware integer type `Int`. We can use the usual arithmetic operations' signs:

```
f :: Integer → Integer
f x = 10^10000 + x
```

In the GHC implementation of Haskell the `Integer` type is implemented with the GNU multiprecision library.

In Haskell we can easily manipulate mathematical objects in a manner quite conventional to a mathematician. To give an example, we can define the factorial as `product [1..n]`, which maps very well to the usual mathematical notation.

Beyond integers. However, arbitrary precision integers are just a first step to symbolic computing. We require an implementation of fractions, which can be accomplished quite simply using pairs of arbitrary precision integers. Still this aspect is interesting, see the aforementioned packages for elaborate implementation.

To give an example, one of the important questions, arising from a traditional implementation of fractions, is 'when do we reduce?' Recall, a fraction a/b is called irreducible, when a and b have no common factors. A fraction ka/kb is not reduced, in other words: not in its lowest terms. The irreducibility test involves a computation of the greatest common divisor (GCD), which is quite costly to perform after each arithmetic operation. On the other hand, not reduced fractions are larger, so further arithmetic operations would be more expensive.

But there is a larger problem. We need to represent irrational values in our system. This issue is not present in floating point arithmetic, as it simply uses approximations. The key solution comes from algebra and is well known: it is called *adjunction*. For the sake of it we switch from rationals to rings of polynomials over rationals, as we will see below.

Example (Adjunction). *We want to adjunct $\sqrt{2}$ to $\mathbb{Q}$. The result is written formally as $\mathbb{Q}(\sqrt{2})$. The technical side is that we compute from now on in $\mathbb{Q} \times \mathbb{Q}$, where for the second component similar rules apply, as for the imaginary part of $\mathbb{C}$. For instance, $(a + \sqrt{2}b) \cdot (c + \sqrt{2}d) = (ac + 2bd, \sqrt{2}(bc + ad))$.*

A classic example for adjunction is $\mathbb{C}$, which is in fact $\mathbb{R}(\sqrt{-1})$. For the generic adjunction to be feasible on a computer, *factor rings* modulo the ideal, generated by specific polynomials, are used. These polynomials have roots, which are exactly the values to adjunct, and these polynomials are irreducible[1]. So, if we want to represent $\mathbb{Q}(\sqrt{2})$, we use in fact $\mathbb{Q}[x]/\langle x^2 - 2\rangle$. In Section 5.5.4 we will

[1]For the definition of irreducible polynomials see Definition A.23 on page 196 in the Appendix or any algebra book.

see more on how to work with such polynomials. Such an approach accounts for the infamous `RootOf` expressions in computations, using CAS.

We consider polynomials (as well as other mathematical structures) over an arbitrary field F. So, F could be $\mathbb{Q}(\sqrt{2})$, but we actually do not care, as we simply construct $F[x]$ over it. This is one of the meanings of *symbolic* computation: we abstract from the representation of base elements of our structures. This is our principal approach throughout the thesis. For the sake of completeness, the other meaning is the possibility to introduce symbols. We do not discuss the latter side of symbolic computing in detail. Our approach, elaborated below, provides these features for free.

As we have indicated before, symbolic computations take more resources into account than numeric ones. This lies partially in the nature of the algorithms used. Partially, the data representation is at fault. If we work in $\mathbb{Q}[x]/\langle f\rangle$ with an irreducible polynomial f of degree n, then we have up to n times more work, as compared to a computation directly in $\mathbb{Q}$. A parallelisation can provide capacity to deal with such an increasing workload. We will see in Chapter 5 how to implement adjunction in Haskell.

Other caveats. But even an efficient implementation of $\mathbb{Q}$ is not a silver bullet! Some algorithms, like naive polynomial euclidean algorithm, Gauß elimination, or Buchberger's algorithm, suffer from so-called *intermediate expression swell* [Tobey, 1966, von zur Gathen and Gerhard, 2003, Brickenstein, 2010]. This means that the intermediate expressions grow very fast. This growth is stimulated by the fact that small-valued fractions might still be large expressions. For instance,

$$\frac{9999999999}{10000000000}$$

is a very small fraction, near to 1. However its representation is quite large.

There are a few workarounds. We can divide them into two approaches. Firstly, we can use another arithmetic. This is a quite generic technique, Chapter 7 elaborates on our approach, the residue-based arithmetic. Further possible methods include some other residue-based approaches, arbitrary precision floating point numbers [Priest, 1991], and continued fractions [Perron, 1954, Khinchin, 1997]. While all residue classes pursue the same goal with likely methods, which method is better is a vast topic for research. However, any residue arithmetic requires an upper bound on the final result. The two other approaches are unfortunately unsuitable here. Arbitrary precision floating point number systems should be *prescaled* to a desired size. An expert is needed to choose the required size. There is some work on adaptive, self-scaling arbitrary-size floating point arithmetic [Shewchuk, 1997], but in practise problems with rounding in the hardware floating point numbers can arise. As for continued fractions, they have very suitable properties for *representing* rational values and even approximations to real values. But actually *computing* with continued fractions is a rather hard task.

Secondly, in some cases, e.g., in a polynomial euclidean algorithm, we do not particularly need exact representation of all coefficients of intermediate polynomials, an approach pioneered by Lehmer [1938]. Methods using a similar idea belong to the area of symbolic-numeric computing. Great success has been achieved, e.g., in the acceleration of the LLL algorithm [Koy and Schnorr, 2001, Nguyen and Stehlé, 2005, Stehlé, 2010]. We do not elaborate on this approach here.

Symbolic algorithms. Another feature our target programming language should support is an ability to express generic symbolic algorithms. To give an example, an implementation of polynomials $R[x]$ should work fine, regardless of which base unique factorisation domain R is currently used. We will see in Chapter 6 how Haskell's type system, especially type classes, help to implement symbolic computation.

2.2 Language Unity

In this section we generally follow [Lobachev and Loogen, 2008]. The problems the programmer of a computer algebra system (CAS) needs to solve are hard [Cohen, 2000, von zur Gathen and Gerhard,

language	
internal	external
implementation	interaction
efficient	comfortable
compiled	interpreted
static	dynamic

Table 2.1: *Two languages of a CAS.*

2003, Grabmeier et al., 2003, Ribenboim, 2004, Levandovskyy, 2005]. For that, an expressive programming language with high-level concepts is desired. But the actual situation with programming languages of a CAS is more difficult. A CAS is a large piece of software, and it is implemented efficiently in a particular programming language. We call this language an *internal* language of the system. It is desired to be fast, which comes not without a price. However, the users of the system would like to *program*. Even more, since the users want to express mathematics in their programs, they would be happy to find matrices, polynomials, symbolic integration etc. predefined. By 'predefined' we mean here either in a standard library or as a primitive. It is not important for us, which one. For these reasons, a second, *external* or interface language is introduced. The internal language should be safe and fast, we imply it has a good compiler and a static type system. The external language should be able to embrace all the mathematics the user may want to express. It seems desirable to choose a dynamic language. It will be embedded into the 'actual' CAS implementation.

The external language needs to express the whole spectre of mathematical structures and their relations. The drawbacks include:

- Some low-level features, e. g., file access, might be missing in the external language.
- The external (interpreted, dynamic) language might be orders of magnitude slower than the internal (compiled, statically typed) one.
- Two possibilities exist for the implementation of the external language.
  - It is implemented in the internal language. In this case we need to implement support for all the features of the interface language in the efficient, but low-level internal one. Think of implementing LISP in C as a by-project! Our aim is not to write a compiler or an interpreter.
  - The other option is to take an off-the-shelf embeddable language, like Groovy [Barclay and Savage, 2006] or any other small dynamically typed language, like Python [van Rossum, 2006] or Ruby [Thomas et al., 2004, Flanagan and Matsumoto, 2008].

In fact, Python is used as glue between the programs in SAGE [Stein and Joyner, 2005, Stein et al., 2011]. For both approaches towards the implementation of an external language, we need to provide access to the core functionality, implemented in the internal language, to the external language programmers, *cf.* R [Ihaka and Gentleman, 1996, R Development Core Team, 2009], where the internal language is C and the external one is a Scheme-like language. This is less of an issue. But it makes impossible for the user to 'look into' and to modify some core functionality: at some point the correct answer on the question 'how is *that* defined?' becomes 'it is a hook to a function in a bundled internal language library.' The requirements to both languages are briefly sketched in Table 2.1.

Fortunately, there is a solution. An interesting approach was done by the developers of GiNaC [Bauer et al., 2002]. This system is written in C++ and has C++ as its interface language.

This is exactly the approach we aim to implement. This way, not a computer algebra *system*, but rather a computer algebra *library* emerges. In other words, we do not create a new system with required features, both programming-language-wise and computer-algebra-wise, but rather *extend* a given programming language by computer algebra features. In this work we focus on algorithms. Hence, we

Language	C	Python	Haskell
Efficient	✓		✓
Compiled	✓	?	✓
Interpreted		✓	✓
High-level		✓	✓
ADT	✓	?	✓
HOF	?	?	✓
Statically typed	✓		✓
Type inference			✓

Table 2.2: *Features of CAS languages in the mainstream.*

would include in our library some types and functions for expressing certain computer algebra algorithms. We consider the types only as far as needed for the algorithm implementation. To give an example, we would implement a type for a univariate polynomial over another, given type. But we would not implement a type for an arbitrary skew field.

Getting back to GiNaC, its implementers had to overcome a rather large problem. The C++ language is a compiled one. But a typical CAS interaction is not only a batch job for some finely written code execution! It is also a long interactive session for finding the particular fine code for a given problem. Can GiNaC be used for that? The answer is yes. Although a C++ interpreter exists, the library implementers chose another way: GiNaC provides its own interactive interface. It is not complete, but it is usable. Are we back at external language design, despite all our efforts? No!

The conclusion we draw from the GiNaC case is: the desired implementation language should possess both a compiler for efficient execution of batch jobs and an interpreter for interactive programming sessions. Having these and adding the CAS-as-a-library idea, we obtain the *language unity* concept, elaborated in [Lobachev and Loogen, 2008]. This paper pushes the idea further: this common language for both implementation and interaction of the CAS library should be a statically typed functional language, *cf.* Table 2.2. The latter compares C [ISO/IEC 9899:1999], Python [van Rossum, 2006] and Haskell [Peyton-Jones, 2003]. The question mark in the table means that some approach exists, but it is not a mainstream implementation. Further, ADT stands for algebraic data types, and HOF stands for higher-order functions. We consider such features essential for a CAS development. Let us discuss the table in more detail. Results on efficiency of various languages' implementations and on programmer's productivity are available in, e. g., [Hudak and Jones, 1994, Fulgham and Gouy, 2011]. We are not aware of a full Python compiler which would produce machine code. All approaches known to us focus on just-in-time compilation. A mature interpreter for Haskell is GHCi, further implementations exist. There are some approaches to higher-order functions in Python, but they are not the main focus of the language. Similarly, it is possible to use functional pointers in C to represent higher-order functions, but we would refrain from such dangerous constructs. This is a common industry practise [Hatton, 2007]. C is statically typed, but it has no type inference.

Another benefit of a CAS library is the availability of language constructs for the user: name binding in the interface language provides the symbol manipulation for free. We will see in following, how we can implement the symbolic computation, i. e., an abstraction from the representation of base elements.

2.3 Dependent Typing

There is a known problem with expressing algebra in a typed functional language. If we define a separate type for each kind of algebraical structure, then we promptly find out that the type of some further algebraical structure depends on its properties. Or, in programming language terms: the type depends on the properties of the value of this very type.

Example 2.1. *Suppose we have some $n \in \mathbb{N}$ and build a residue ring $\mathbb{Z}/\mathfrak{n}$. The notation is $\mathfrak{n} = \langle n \rangle = n\mathbb{Z}$ for an ideal $\mathfrak{n}$, generated by n. We cannot state if the ring is a field or not, without examining the fact whether n is prime.*

However, in a Haskell definition of residue rings:

```
residueRing :: Ring → Ideal → Ring
residueField :: Ring → Ideal → Field
```

the second definition is wrong. As not every residue ring is a field, the matter whether it is one, depends on the properties of a specific value of `Ideal`.

Such problems were described in the works of Serge Mechveliani [Mechveliani, 2000, 2007a,b]. One possible solution is dependent typing. But as we focus on algorithms and their parallelisation, this problem is not acute for us. We do not actually want to represent the hierarchy of algebra constructs, but to implement some algorithms!

A full-grown implementation should rather use a dependently typed parallel functional language. However, in our prototype we resort to a Hindley–Milner typed (parallel) functional language. (After J. Roger Hindley, *1938 and Robin Milner (Arthur John Robin Gorell Milner), *13.1.1934, †20.3.2010.) See [Hindley, 1969, Milner, 1978, Damas and Milner, 1982] for the Hindley–Milner type inference algorithm. The only approach to a parallel dependently typed language known to us is parallel Aldor [Gautier and Mannhart, 1999, Maza et al., 2007].

2.4 An Example: Laziness

Why is laziness important in a programming language for symbolic computation? A lazy, non-eager evaluating programming language can be used to represent mathematical objects in a more natural manner, see [Karczmarczuk, 2000, Hinze, 2008]. We will see in the course of this work how lazy semantics can be used to express both computer algebra algorithms and parallelisation in a more easy manner. For an example of the latter see Chapter 5.

In this section we will present a very elegant 'toy' example for lazy evaluation in a mathematical context. Further cases will be presented in the course of this work.

Power series. Closely following [McIlroy, 1999, 2001], we present some operations on power series encoded in Haskell. We define power series as an infinite list of coefficients. The required basics are in Figure 2.1. It defines the `Num` and `Fractional` instances for (infinite) lists. We reiterate, we assume the reader knows Haskell.

Having these basics we can easily write Haskell code for some basic calculus operations, like derivation and integration.

```
deriv :: (Num a) ⇒ [a] → [a]
deriv (f:fs) = helper (*) fs 1
deriv _ = []
integral :: (Fractional a) ⇒ [a] → [a]
integral fs = 0 : (helper (/) fs 1)

helper :: Num a ⇒ (t → a → b) → [t] → a → [b]
helper op (g:gs) n = g`op`n : (helper op gs (n+1))
helper _ _ _ = []
```

Note, `helper` is a higher-order function. As soon as we have integration, we can define the series expansion of the exponential function.

```
-- exponential function
expx :: [Rational]
expx = 1 + (integral expx)
```

```
default (Integer, Rational, Double)
infixl 7 .*
(.*) :: (Num a) ⇒ a → [a] → [a]
x .* (y:ys) = x*y : x.*ys
_ .* _ = []

instance Num a ⇒ Num [a] where
  (+) = zipWith (+) -- all lists are infinite
  (-) = zipWith (-)
  (x:xs) * g@(y:ys) = x*y : (x.*ys + xs*g )
  _ * _ = []
  fromInteger x = (fromIntegral x) : repeat 0

instance Fractional a ⇒ Fractional [a] where
  (0:xs) / (0:ys) = xs/ys
  (x:xs) / (y:ys) = let q = x/y
                    in q : (xs - q.*ys)/(y:ys)
  _ / _ = []
```

Figure 2.1: *Some initial code for encoding power series in Haskell. We base our presentation on [McIlroy, 1999, 2001].*

We can also easily define some recursive number sequences, like Catalan numbers. Note the arbitrary precision integers.

```
catalan :: [Integer]
catalan = 1 : catalan^2
```

Mutually recursive definitions are also not a problem: we define sine and cosine expansions in a very concise manner.

```
sinx, cosx :: [Rational]
sinx = integral cosx
cosx = 1 - (integral sinx)
```

Of course, we cannot compute the complete series in a finite time, but these potentially infinite, mutually recursive and complex definitions result in the correct output—without a failure. We merely need to request *some*, but not all coefficients of the power series. Note, that the same definition of power series was used both for integers and rational numbers. This is the ad-hoc polymorphism. We will discuss it later, in Chapter 6.

We can verify the correctness of McIlroy's approach with the `QuickCheck` library. The definition of the test for the definition of cosine with the well-known identity $\sin^2 x + \cos^2 x = 1$ is below.

```
test :: Int → Bool
test n = let cs = take n $ sqrt $ 1 - sinx^2
         in cs == take n cosx
```

Powering roots of unity. Another example of laziness in our main codebase is the generation of the powers of roots of unity in Chapter 6. Let us not bother now, what exactly is a primitive root of unity. Let us assume, it is already defined as `w` somewhere in our program. Then we can generate all powers of it with `ws = map (w ^) [0..]`. This corresponds well with the mathematical notation $\boldsymbol{\omega} = [\omega^0, \ldots, \omega^n]$ for some $n \in \mathbb{N}$. In the contrast to the latter, we denote *all* powers of ω. But indeed only the values really required in the subsequent computation will be generated. So, if we need only $n/2$ powers of ω, we will generate exactly that many. No action is required to save work.

Conclusions. We can model some advanced mathematical structures in Haskell with ease and grace. We do not use power series in the further text, but the second example is a clear part of our work. We rate laziness as one of the most important Haskell's features. Lobachev and Loogen [2008] presented further examples and compared the speed of Haskell and C++ in one of the case studies.

2.5 Conclusions and Outlook

Haskell is very suitable for implementing computer algebra algorithms. Our 'language unity' approach tells us we should use Haskell as both implementation and interface language of a computer algebra system. We conclude that build-in arbitrary precision integers and lazy evaluation can be very useful for implementations of symbolic algorithms. We have seen how easy it is to define differentiation and integration in terms of infinite power series in Haskell. One more feature we will use in the next chapters for the implementation of computer algebra algorithms is the ad-hoc polymorphism. Haskell is a very expressive language, so we can use the abstraction techniques for a generic programming approach. One of the most important features of Haskell are higher-order functions. The functions are first class citizens in Haskell. We will use higher-order functions to represent algorithmic skeletons. In multiple aspects Haskell is a very 'mathematical' language. As mentioned above, list comprehension syntax in Haskell has an almost one-to-one correspondence with the set notation in mathematics. This is of a benefit for our implementation. We will consider the *parallel* lazy functional language Eden in the next chapter.

Chapter 3

PARALLEL PROGRAMMING WITH EDEN

> The tale is old as the Eden Tree—
> as new as the new-cut tooth.
>
> Rudyard Kipling, *The Conundrum of the Workshops*

As we have seen in the previous chapter, we need to use a statically typed functional programming language. We aim for a parallel computer algebra implementation, so we also need a support for parallelism. Section 2.4 has shown that laziness is of benefit to express mathematics in a programming language. We will see in this thesis that laziness is also of benefit to represent parallelisation concisely. In the following, we use a parallel lazy functional language with explicit process creation. Such a language, based on Haskell, is under development in Marburg and Madrid and is called Eden [Loogen et al., 2005]. With it we can focus on the development of the algorithmic skeletons for computer algebra algorithms, which is the actual subject of this thesis. Our choice of Eden is influenced by the following features:

- Eden is a language with explicit parallelisation. Thus we will always have exact control of what is exactly executed in parallel [Loogen et al., 2005].
- Eden has a large skeleton library with sophisticated skeletons, implemented in Eden itself [Loogen et al., 2003, Eden Skeletons, 2011].
- There is a possibility of dynamic communication in Eden [Breitinger and Loogen, 1997, Berthold and Loogen, 2005, Dieterle et al., 2010b].
- Eden features *futures* [Dieterle et al., 2010b] and process network construction tools [Horstmeyer and Loogen, 2010].
- Eden performs well both on distributed memory machines and on shared memory multicores, see e. g., [Berthold et al., 2009a].
- It is easy to express high-level parallelism in Eden [Loogen et al., 2003, Berthold and Loogen, 2006, Berthold et al., 2009b].

There are multiple approaches to a parallel Haskell. We chose Eden as it gives us the most *control* of the parallelisation, but without bothering us with unneeded, low-level details, in contrast to Haskell+MPI. We will present alternative approaches to parallel Haskell in Section 3.1.1.

Structure of this chapter. In Section 3.1, we describe Eden and determine its position in terms of classifications, which we will present below. Then we review existing approaches to parallel computing in the context of Haskell programming language in Section 3.1.1. We describe the process model of Eden and an important extension of the language in Sections 3.1.2–3.1.4. In Section 3.1.5 we review the drawbacks and benefits of lazy evaluation in a parallel setting. Section 3.2 presents a survey of existing algorithmic skeletons implemented in Eden. Some helper functions for list manipulation are also defined there. Further, Section 3.3 introduces the Eden TraceViewer and the whole concept of *tracing*, whereas Section 3.4 discusses hardware, conventions and approaches we use to measure the parallel execution time of our programs. Section 3.5 concludes the chapter.

We present two taxonomies of parallel processing next, Flynn's and Foster's approaches. We will apply them to Eden and other parallel Haskells next.

		Instruction	
		single	multiple
Data	single	SISD	MISD
	multiple	SIMD	MIMD

Table 3.1: *The Flynn taxonomy.*

Flynn taxonomy. The classic taxonomy of parallel computing by Flynn defines the matrix of single/multiple 'instructions' on separate nodes and of single/multiple data streams [Flynn, 1966]. These are abbreviated, e.g., 'single instruction, multiple data' becomes 'SIMD'. The Flynn taxonomy results in four possible combinations, as shown in Table 3.1 [Grama et al., 2003]. The classic sequential computing is SISD. The 'vector' machines realised the SIMD principle. Independent computers working on different data with various methods obey the MIMD scheme.

The extension of the taxonomy considers not the particular hardware instructions, but rather the complete program codes. An example would be SPMD.

PCAM Methodology. Foster [1995] defines four different stages of the development of an efficient parallel algorithm. These stages are noted after their first letters as PCAM: partitioning, communication, agglomeration and mapping.

The *partitioning* stage consists of domain decomposition and functional decomposition and designates separate tasks for the parallel processing. There are some recommendations on the number of tasks (e.g., at least an order of magnitude more than processing elements), granularity of tasks, scalability of the partition. In the *communication* stage locality and structure of the communication are considered. For instance, we might consider reordering the data so that one-to-one communications happen more often between neighbour processing elements (PEs). Furthermore, it is important that one-to-many (many-to-one) schemes do not exceed the bottleneck of the single sender (receiver). Another point of consideration is dynamic communication. Here the communication channels are not established beforehand, but are created while the computation is already in progress. Finally, asynchronous communication is possible. Foster [1995] underlines the importance of it in a distributed memory setting. *Agglomeration* stands for merging small tasks from the partitioning stage to larger tasks. An important aspect of agglomeration is to merge interdependent tasks together and to reduce the need in communication. Then, *mapping* designates where each task has to be executed. In this phase the tasks are assigned to processing elements, it is so called 'process placement'. The optimal mapping problem is NP-complete [Bokhari, 1981], but there are some specialised heuristics and strategies for particular cases. One of the approaches to mapping are various methods of *load balancing*, see, e.g., [Tantawi and Towsley, 1985, Foster, 1995, Kwok and Ahmad, 1999a,b]. The load balancing methods include both some partitioning approaches, like graph partitioning and round-robin balancing, and task scheduling approaches, like various master-worker schemes, including more sophisticated hierarchical and distributed master-worker implementations [Hamdi and Lee, 1995, Shao et al., 2000, Shao, 2001, Aida et al., 2003, Grama et al., 2003]. Eden-based master-worker schemes include [Peña and Rubio, 2001, Loogen et al., 2003, Priebe, 2006, Berthold et al., 2008, Dieterle et al., 2010a].

Language classification. We distinguish between the work done by the compiler or by the runtime system and the work done by the programmer. In Table 3.2 we show classification of parallel languages based on PCAM. We show a dash (–) if a stage is completed by a compiler or a runtime system (RTS). We show the letter if an appropriate stage has to be done manually.

3.1 Eden as Haskell Extension

Commonly developed at Philipps-Universität Marburg and Universidad Complutense de Madrid, the lazy parallel functional language Eden [Loogen et al., 2005] is a programming language with an explicit

Language	Implicit	Semi-explicit	'Control'	Explicit
Approach	Data Parallel	Annotations	Process control	Full
Stages	----	P---	PCA-	PCAM

Table 3.2: *Classification of parallel languages.*

process instantiation but implicit communication.

The classification of Eden after the extended Flynn taxonomy is SPMD. In the PCAM classification, Eden is PCA−. Basing on these classifications, we consider Eden to be a 'control' parallel language. As we shall see in following sections, we have explicit process control and an option for explicit communication. In contrast, GpH [Trinder et al., 1999] is an annotation-based language. We discuss details and further approaches in the next section. We assume the knowledge of Haskell. Still, we will detail on usage of library functions and important utilisations of laziness in the following.

Eden has been implemented as an extension of the Haskell compiler GHC. The extensions include minor modifications of the parser for dynamic channels (see below), a few additional 'ways' of compilation and a special parallel runtime. The parallelism is organised as follows. The runtime provides special *parallel primitives* [Berthold and Loogen, 2007a], mostly implemented as function calls to an underlying parallel middleware. Currently MPI [Snir et al., 1995, MPI, 2009] and PVM [Sunderam, 1990, PVM, 2009] are used. An option to use direct memory copy is in work [Pickenbrock, 2011]. The mentioned primitives form a basis for higher-level language constructs [Berthold, 2008]. We aim to use some low-level skeletons in the higher-level ones, *cf.* 'implementation skeletons' [Klusik et al., 2001]. As we regard Eden from the application programmer's point of view, it suffices to give denotational overview of higher-level Eden constructs.

Various aspects of the Eden semantics have been studied in, e. g., [Breitinger et al., 1996, Hidalgo-Herrero and Ortega-Mallén, 2002, 2003, Sánchez-Gil et al., 2011].

3.1.1 Flavours of Parallel Haskells

Pure functional languages are seen as quite a tempting base for a parallel language. The reason for this lies in the fact that the order of reductions is irrelevant in a pure functional language. For example, given two functions `f` and `g`, if a function `f` neither consumes output of the function `g` nor vice versa, then the applications of these two functions can be evaluated—*reduced to head normal form*—simultaneously. As Haskell is a mature pure functional language, quite a few attempts have been made for a parallel Haskell. An overview is in [Trinder et al., 2002]. A seemingly abandoned pioneer is pH [Aditya et al., 1995, Nikhil and Arvind, 2001]. At roughly the same time Haskell+MPI was researched [Breitinger et al., 1995, 1998]. A recent development is [Astapov et al., 2011]. The desire to explicitly control the process creation resulted in Eden, one of the first publications was [Breitinger and Loogen, 1995], the standard reference is [Loogen et al., 2005]. Eden has been designed as a distributed memory language; however a threaded simulation is available [Breitinger et al., 1997]. Even more, we found that the standard, distributed memory implementation of Eden performs surprisingly well on the multicores [Berthold et al., 2009a]. A version using directly communicating OS processes on multicores is being developed [Pickenbrock, 2011]. Eden is implemented basing on GHC, the Glasgow Haskell compiler.

Quite a different approach to parallelism is the implicit process control. It falls under the P--- category of the PCAM classification. In Haskell this approach is implemented in GpH language [Trinder et al., 1999]. It carries out parallelism with evaluation strategies and annotation combinators [Trinder et al., 1998a]. Two different implementations exist for the *language* GpH. GpH−GUM [Trinder et al., 1996b, 1998b, 1999] is a virtual shared heap implementation for the distributed memory machines, it is a fork of GHC. Multicore Haskell [Marlow et al., 2009] is implemented as an SMP language on top of GHC. Both Multicore Haskell and Eden threaded simulation [Breitinger et al., 1997] utilise for their implementation the same concurrency primitives of GHC—the Concurrent Haskell [Peyton-Jones et al., 1996]. At the same time, Eden and GpH−GUM share a part of their code base in the

implementation of a parallel runtime system [Berthold, 2004]. However, the explicit process control of Eden shows its benefits in comparison with both implementations of GpH [Loidl et al., 2003, Berthold et al., 2009a].

Recent developments include the `Par` monad for deterministic parallelism [Marlow et al., 2011] and Cloud Haskell [Epstein et al., 2011]. The latter is quite similar to Erlang [Armstrong, 2007]. We discussed `Par` briefly in Chapter 1.

A completely different approach is Data Parallel Haskell [Chakravarty et al., 2007]. It implements support for distributed arrays and utilises a special transformation layer to flatten nested data parallel loops. It this sense it is similar to NESL [Blelloch and Greiner, 1994, Blelloch, 1995].

3.1.2 Eden Processes

The parallelism model in Eden is implemented with *processes*. The latter are executed on remote machines, thus carrying out parallelism. Before a process can be executed, it needs to be defined. For the sake of this the following mechanism is introduced. The *process abstraction* in Eden is similar to the lambda abstraction from the lambda calculus [Church, 1941, Barendregt, 1984]. We can *define* a process abstraction and subsequently *instantiate* it with some parameters on another processing element (PE). Process abstraction is a mould from which multiple 'actual' processes can be obtained.

We define a process abstraction with a constructor function `process` of type `(a → b) → Process a b`. Given a function `f :: a → b` the call of `process f` results in a special type `Process` of kind $* \to * \to *$. Thus, `f` from above, captured in a process abstraction, has the type `Process a b`. Let us call this particular processes definition `p`.

When needed to *instantiate*—create—a predefined process with some input data, we use the 'hash operator' `#`. Having a process abstraction `p` of type `Process a b` and input data `x` of type `a`, we do the following: `p # x`. This expression creates a *new process*, which computes the result of application of `f :: a → b` (see above) to the input data `x`, producing the same result as `f x` would produce. For `x :: a` holds that `p # x` is of type `b`. The input is communicated to the computing process and the output is communicated back to the parent implicitly. The data is evaluated to the reduced normal form prior to communication. See Section 3.1.5 for a discussion of evaluation strategies. The application programmer does not have to do anything for this communication to happen. One could say that `#` is the `$!` operator with a side effect of a parallel application. Because of the evaluation of data to be sent, `#` is a strict operator.

Example 3.1 (Parallel binomial coefficients). *We compute the binomial coefficients*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad \text{where } n! = \prod_{i=1}^{n} i.$$

We spark three processes for subsequent computations of the factorial. This is not the most efficient way to compute binomial coefficients, but we aim to demonstrate Eden constructs. The sequential implementation is easy.

```
fac :: Integer → Integer
fac n = product [1..n]

binomSeq :: Integer → Integer → Integer
binomSeq n k = (fac n) `div` ( (fac k) * (fac (n-k)) )
```

For the parallelisation, we define the process abstraction

```
facProc :: Process Integer Integer
facProc = process fac
```

Using it, we can rewrite `binomSeq` in a parallel manner. We create particular instances of the process abstraction—the processes.

```
binomPar :: Integer → Integer → Integer
binomPar n k = (facProc # n) `div` ( (facProc # k) * (facProc # (n-k)) )
```

Note that we have instantiated the single `facProc` process abstraction with three input values, resulting in three processes created. The input and output communication is transparent.

In the current implementation the # operator is implemented with the function `instantiateAt`. It has the type `Int → Process a b → a → IO b`. The additional integer argument provides a possibility to specify the *placing* of a process—the number of the PE the process should be created on. The default placing is local round-robin: move to next PE every time; if it is not possible, go to the first one. A wrapper `instantiate` is defined with the default placing.

Example 3.2. *Imagine we have 5 PEs. We have 8 processes to create. We are already running on the first PE in a 'master' process. In a round-robin manner the new processes will be created on the following PEs: 2, 3, 4, 5, 1, 2, 3, 4.*

Further, the result of the instantiation will be an expression of type `IO b`, because process creation is a side effect captured in the `IO` monad. The current implementation makes at the end just `b` out of it with a simple and crude `unsafePerformIO` call.

3.1.3 Explicit Communication

The above approach enables us to create process trees, but not arbitrary process graphs. Two child processes cannot communicate directly, they are forced to talk via their common predecessor. However, such 'relaying' is not nice. We have an increased communication overhead leading to a major slowdown. A good example is a naive `map-transpose-map`, *cf.* Section 6.4.5. Eden has multiple solutions to this problem. All of them are based on creating a direct communication line between two child processes. Hence, we effectively make a process tree to a process graph. These approaches differ in usability and implementation level. The following newer approach towards explicit communication is implemented using an older one, so we need to introduce it first.

In order to connect two processes, we need to create a *communication channel* between them. This is a quite common approach in distributed computing. The child processes are already connected via implicit channels with their parents, such channels are already open when a child begins working. But it would be too costly to create in advance further channels for an all-to-all communication, especially if not all of them are required and if we have many processes. So, the programmer tells the system, she needs a channel between two particular processes and this channel will be created *dynamically*. Breitinger and Loogen [1997] have introduced dynamic communication channels to Eden. The static communication channels are essentially the same channels created automatically by the system at the time of process creation.

The channels in Eden are typed, essentially they have a type 'channel for type `a`'. We can create a channel to communicate channels for type `a`. Arbitrary nesting of such 'higher-order channels' is allowed. Channels are valid only for one-to-one communication. While using channels, a convention is important that restricts a channel to be used only once. With multiple communications over one channel the referential transparency would be broken immediately! An exception to this rule is an optimisation for list communication: a list is sent in a *stream*, where the same channel is reused for all list elements. However, this is an under-the-hood optimisation: at the receiving end the list is reconstructed incrementally. We elaborate on streams on page 23. Using dynamic channels directly is quite challenging.

Still, let us regard the direct communication channels in more detail. We follow [Loogen et al., 2005, Berthold, 2008]. The aforementioned type for 'channel of type `a`' is `ChanName a`. The two functions

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

are used in the context of dynamic channels. With `new (λchanName chanValue → res)` a channel handle `chanName` is created, "via which the values `chanValue` will eventually be received in the future" [Loogen et al., 2005]. The `chanName` is the channel handle, we also call it 'channel name'. The *pro forma*

result of the whole computation is `res`. Next, `parfill chanName res1 res2` results in following. Before `res2` is evaluated, `res1` is concurrently received via `chanName` and evaluated to reduced normal form. The formal final result is `res2`. As an optimisation, we define a list version of `parfill`.

```
multifill :: Trans x ⇒ [ChanName x] → [x] → b → b
multifill []  _  b = b
multifill _  []  b = b
multifill (c:cs) (x:xs) b = parfill c x (multifill cs xs b)
```

The communication is established for each channel name in the `cs` list. This function is used in Chapter 6.

3.1.4 Making Communication Implicit: Remote Data

One of the known approaches to communicating data between the processes is the concept of *futures*. A handle is used, it is a placeholder for an object to be computed, which is introduced in the receiver process. The latter can operate with it as with a normal object as long as the receiver does not need to look inside. As soon as the sender process computes the actual value for the object, the handle can be replaced with the actual object without any significant difference for the receiver [Halstead Jr., 1985].

In the context of the Eden supporting library, we can use a channel for transmitting channels as a future. We call it *remote data*. When we need the content of the handle, we use the remote data to request a new channel. This channel will transmit the desired value as soon as it will be computed. The details are presented in [Dieterle et al., 2010b]. A similar approach was developed independently by Alt and Gorlatch in [Alt and Gorlatch, 2003, 2005, Alt, 2007].

The important goal of the remote data is skeleton composition [Dieterle et al., 2010b]. If we use only remote data based input and output values, we can leave the data distributed across the PEs. This way the next skeleton will be able to pick the data off from where is now lies. Thus, we can simply use the usual composition combinator ∘ to connect skeletons. The data will be passed around automatically.

From the application programmer's view, remote data is a type `RD a` with two mutually inverse functions `fetch` and `release`. The function `fetch` of type `RD a → a` is used to obtain the data from the handle. However, from the denotational view it just corresponds to stripping the constructor. We create a handle to existing data, 'releasing' it into the remote data world with `release`. This function has the type `a → RD a`. From the denotational view it is just 'packing' the data into a constructor, but in fact a channel for transmitting channels for data of a given type `a` is created. In other words, it holds `fetch ∘ release = id` of type `RD a → RD a` and `release ∘ fetch = id :: a → a`. The functions `fetchAll` and `releaseAll` are optimised analogues of `fetch` and `release` for the lists of remote data objects. We will use remote data in Chapters 5 and 6.

Example 3.3. *The function `liftRD` converts a given function `f` to remote data.*

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

The input is firstly fetched from the remote data, then the parameter function `f` is applied. The result becomes remote data with the `release` function.

3.1.5 Demand and Evaluation Control

> So, wouldn't a Haskell robot just sit there and be lazy?
>
> A sceptic on functional reactive programming quoted in http://www.haskell.org/frob/

Haskell is defined as a lazy programming language, expressions are evaluated only if required. An optimisation step of any mature compiler of a lazy language is the so-called *strictness analysis,* when

subexpressions that might be required to be evaluated in the program run are identified. The compiler can then discard everything which is surely *not* evaluated. The actual GHC implementation imposes Haskell as a demand-driven language: expressions are evaluated at the moment when a *demand* on the result of the evaluation is present. A demand on an expression, depending on some other expression, will issue a demand on both expressions. To give a specific example: the output of the expression on the display forces a demand on the expression.

The need of a demand is seen as an obstacle for parallelism. If no expressions are evaluated before they are needed, then no parallelism will emerge. Hence, most parallel extensions for functional languages with demand-driven evaluation create artificial demand in expressions dealing with parallelism. GpH [Trinder et al., 1996a, 1998b, 1999], for instance, utilises demand- and precedence controlling combinators. The sequential precedence controlling combinator `pseq` is also used in Eden and other, even non-parallel Haskell implementations. We describe them below.

Demand control in GpH. The GpH uses so-called *evaluation strategies,* the annotations of the depth and precedence of evaluation [Trinder et al., 1998a]. The actual interfaces are subject of research, see, e.g., [Marlow et al., 2010]. GpH uses the evaluation strategies and two *precedence combinators* `pseq` and `par` to describe both sequential precedence of evaluation and parallel evaluation.

- `x ‘pseq‘ y` will execute `x` and return `y`.
- `x ‘par‘ y` will do the same in parallel.

The actual evaluation strategies are:

- `r0` is a strategy for no evaluation.
- `rwhnf` is a strategy for reduction to weak-head normal form. We evaluate ‘one level’ of depth.
- ‘spine’ is a traverse strategy for evaluating all constructors, but none of the constructor parameters. It is often used for lists. In this case it evaluates all the `Cons` constructors, but none of the list elements.
- `rdeepseq` is a strategy for reduced normal form. It leads to the complete evaluation. Such behaviour was known as `rnf` strategy, the actual relation between the two is:

```
rdeepseq a = rnf a ‘pseq‘ a
```

Common examples are:

- `rnf x ‘pseq‘ x`, forcing the evaluation of `x` and returning it.
- `putStrLn "working" ‘pseq‘ largeWork`.

As we see, we can use the precedence combinator `pseq` and evaluation strategies for controlling program execution also in absence of parallelism. Exactly these features belong to the regular Haskell programmer's repertoire and are used for demand control in Eden.

The need for demand control in Eden skeletons is motivated by the wish to enforce evaluation of a particular subexpression before it is evaluated in the case of default demand control. Such fine-tuning yields better performance, but is a rather non-trivial task. For example, in a skeleton computing and then communicating some results, it might not be wise to create communication channels on demand when the results are already computed. Instead, early preparation steps for the communication will ensure immediate transmission of the results as soon as they are computed.

```
-- a stream processing implementation
unshuffle :: Int → [a] → [[a]]
unshuffle p xs = [takeEach p (drop i xs) | i ∈ [0..p-1]]

takeEach :: Int → [a] → [a]
takeEach n [] = []
takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)

shuffle :: [[a]] → [a]
shuffle  = concat ∘ transpose
```

Figure 3.1: *Source code for* `shuffle` *and* `unshuffle`.

3.2 Skeletons Survey

The Eden application library has multiple algorithmic skeletons [Cole, 1989] defined. The skeletons are implemented in Eden as higher-order functions, *viz.* [Galán et al., 1996, Michaelson et al., 2001, Peña and Rubio, 2001, Klusik et al., 2001, Loogen et al., 2003]. Only selected skeletons are shortly presented here, we elaborate on them as needed. We can classify these into a few categories. Map-like skeletons are presented in Section 3.2.2 on page 25, divide and conquer skeletons can be found in Section 3.2.3 on page 31, and iteration skeletons are in Section 3.2.4 on page 33. We base our presentation on the current state of the Eden skeleton library [Eden Skeletons, 2011]; however, we have modified some definitions for the sake of better presentation. We need to regard commonly used helper functions first.

3.2.1 Helper Functions

Here we define some functions we will use throughout the whole thesis. In general, we use functions from `Prelude` and other standard GHC libraries like `Data.List` with little additional description. However, we annotate the type of each library function used. By simply saying '`foo` is a library function' we refer to the standard library `Prelude`.

Example 3.4. *The library function* `map` *of type* `(a → b) → [a] → [b]` *applies a given function to each element of a given list.*

For functions defined as a part of Eden's standard library, including skeletons, we choose a different approach. We list here the definitions we will need at several points in the thesis.

List distribution. A very common task is to distribute a list (of length n) equally among p PEs in a round-robin manner, with $n \gg p$. We use two complementary functions `shuffle` and `unshuffle` for this purpose. The implementation is in Figure 3.1. The function `drop k xs` of type `Int → [a] → [a]` omits the first `k` elements of the list `xs` and returns the result. As the name says, the library function `transpose` of type `[[a]] → [[a]]`, available from the `Data.List` library, transposes the nested list: the i^{th} elements of inner lists of the input become the i^{th} inner list of the output. Figure 3.2 gives an intuition. The function `concat` of type `[[a]] → [a]` flattens a nested list to an ordinary one.

Combined, the function `unshuffle` takes an integer and a list, and produces a list of lists in a round-robin manner. The outer list has as many entries as the integer parameter states, let it be n. The usual value of n is the number of processing elements. The inner lists consist of elements of the input list. The first element of the input list is the first element of the first inner list, the second element of the input list is the first element of the second inner list, and so on. Finally, the n^{th} element of the input list is the first element of the n^{th} inner list and the $n + 1^{\text{st}}$ element of the input list is the second element of the first inner list. To give an example: `unshuffle 2 [1..6]` is `[[1,3,5],[2,4,6]]` and

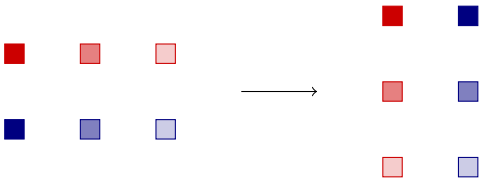


Figure 3.2: *A scheme for* `transpose`.

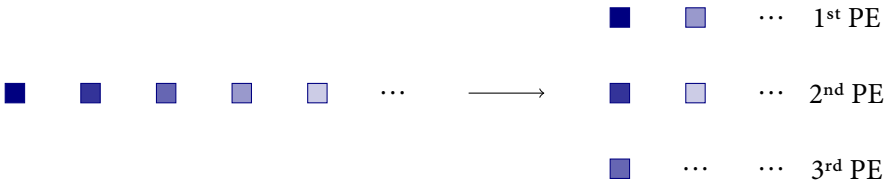


Figure 3.3: *A scheme for* `unshuffle` *for 3 PEs.*

`unshuffle 3 [1..7]` returns `[[1,4,7],[2,5],[3,6]]`. This behaviour is sketched in Figure 3.3. The function `shuffle` does the converse and builds the original list from the list of lists.

Streams. The implementation in Figure 3.1 is a bit different as one may deduce from its description. Normally one works with finite, completely existing lists. However, this is not the case here. As the transformation in `unshuffle` is used to distribute tasks to PEs (more on it in Section 3.2.2), we need to create new tasks as soon as first elements of the inner lists are available. This means also that `unshuffle` begins producing the output before the end of the input list is available. Such functions also work for infinite lists. We call functions fulfilling the above property *stream processing functions, cf.* [Wray and Fairbairn, 1989]. Not all functions have this property. An example of a non-stream processing function is `length` from the standard library. The latter function has the type `[a] → Int`.

All the *transmissible* types in Eden, i. e., types which can be communicated to other machines, must be instances of the `Trans` type class. The latter and its standard instance are shown in Figure 3.4. As Eden semantics forces the evaluation of data before sending, the `Trans` type class can be derived only for instances of `NFData`, i. e., the types which can be evaluated to the reduced head normal form.

The more in-depth background of using stream processing functions is that Eden communicates lists as *streams*. Before the list arrives completely, it is not clear how long it is and whether it is finite. However, a stream processing function can already begin working on the already received elements instead of waiting for the list end. The streaming behaviour for lists is defined in the `Trans` instance for lists, *viz.* the same Figure 3.4. It is possible to define a similar transfer strategy for arbitrary datatypes by stating a custom `Trans` instance for them.

The `unshuffle` function is often used to distribute tasks to PEs, thus it is important to implement it as a stream processing function. This is the reason for the implementation of `unshuffle` as in Figure 3.1.

Lifting and chunking. In some circumstances the communication of lists as streams is not desired. The major reason not to want list streaming is when the lists are very large. As each list element is transmitted separately, the communication overhead is very large in this case. There are two approaches to disable streaming at the application level.

The first one disables streaming completely. The default configuration is to send only lists as streams. A constructor of some arbitrary type with no custom `Trans` instance is sent as one message. So, the essence of *lifting* is to put the actual data into such a type.

```
data L a = L { fromL :: a }
```

We wrap the data into the type `L` before sending. When the data is received, we strip the constructor `L` with the access function `fromL`.

```
-- the Trans type class and its default implementation
class NFData a ⇒ Trans a where
    write :: a → IO ()
    write x = rdeepseq x `pseq` sendData Data x

    createComm :: IO (ChanName a, a)
    createComm = do (cx,x) ← createC
                    return (Comm (sendVia cx), x)

-- the instance for lists
instance (Trans a) ⇒ Trans [a]  where
    write l@[]   = sendData Data l
    write (x:xs) = (write' x) » write xs
      where write' x = rdeepseq x `pseq` sendData Stream x
```

Figure 3.4: *The* `Trans` *type class, its default instance and its instance for lists. This is the Eden system code not intended to be modified by an application programmer.*

```
chunk :: Int   -- ^ length of a single chunk
      → [a]    -- ^ list to be split
      → [[a]] -- ^ list of chunks
chunk _ [] = []
chunk n xs = ys : chunk n zs
    where (ys, zs) = splitAt n xs

unchunk :: [[a]] -- ^ list of chunks
        → [a]    -- ^ restored list
unchunk = concat

-- it holds: unchunk ∘ chunk n = id
```

Figure 3.5: *Functions chunk and unchunk.*

However, it might not be a good thing to disable streaming completely. Regard the previous use case of a very long list. In this case, we still want the receiver side to start processing before the whole list arrives. We merely do not want to send the list element-wise. Thus, we need to reduce the amount of streaming, but not to disable it completely. This is what our second approach does. It relies on the fact that only outer lists are streamed in the default Eden configuration. So, if we want to stream not single elements, but portions of a list, we merely need to pack them into the transmission of the outer list as lists. This is called *list chunking*. The fine-tuning is achieved by specifying the *chunking size*—the maximal length of the inner list. Best such length is determined experimentally. For the code of `chunk` and its inverse function `unchunk` see Figure 3.5.

Notably, for a particular class of functions on lists, it is possible to define the higher-order function `applyChunk`, displayed in Figure 3.6. This function receives an integer for the chunking length, a function on lists, and an input list. The input list is chunked into pieces of given length, forming a list of lists. The function from the second parameter is applied to each piece. Afterwards `unchunk` is applied to the list of lists, forming a list. We aim to describe all `f` such that for all positive `n` and any lists `xs` the call of `applyChunk n f xs` is equivalent to `f xs`. All such functions `f` in question can be expressed with `concat ∘ map g` for some `g :: a → b`. Note that, e. g., functions based on the `reduce` combinator cannot be expressed in the above manner. For such functions `applyChunk` cannot be used without modifications. A similar approach is the `macro` function from [Klusik et al., 2001], which led to an elegant formulation of the `farm` skeleton. See the next section for the latter.

```
applyChunk :: Int → ([a] → [b]) → [a] → [b]
applyChunk n f xs = unchunk $ map f $ chunk n xs
```

Figure 3.6: The function applyChunk.

```
spawn :: (Trans a, Trans b)
      ⇒ [a → b] -- ^ list of worker functions
      → [a]     -- ^ task list
      → [b]     -- ^ result list
spawn fs xs = unsafePerformIO $ zipWithM instantiate (map process fs) xs

type Map a b = (a →  b)       -- ^ worker function
             → [a]            -- ^ task list
             → [b]            -- ^ result list

parMap :: (Trans a, Trans b) ⇒ Map a b
parMap f xs = spawn (repeat f) xs
```

Figure 3.7: Implementation of parMap.

3.2.2 Map and Friends

Several of our skeletons have a type signature similar to that of a `map`. The most simple one is a `parMap`. It has exactly the `map` signature—up to the `Trans` context. So, let, as in Example 3.4,

```
type Map a b = (a → b) → [a] → [b]
```

then we can write both `map :: Map a b` and `parMap :: Trans a b ⇒ Map a b`. In the latter implementation, for the each element of the input list a process is created. This approach is not efficient in cases with a large quotient of list length to number of PEs. We will use the `map`-like skeletons throughout this thesis.

Let us regard `parMap` in more detail. We define the parallel map with the helper function `spawn`. It instantiates a *list* of functions over the list of input values. This function is defined in Figure 3.7. The definition includes a monadic version of `zipWith`, the function `zipWithM`,

```
zipWithM :: Monad m ⇒ (a → b → m c) → [a] → [b] → m [c]
```

from the `Control.Monad` library. In this case the monad is the `IO` monad. Then the monad is removed with the function `unsafePerformIO` from the `System.IO.Unsafe` library. The type of this function is `IO a → a`. The Eden primitives `process :: (a → b) → Process a b` and `instantiate` of type `Process a b → a → IO b` are defined in Section 3.1.2 on page 19. The current definition of `spawn` starts the next process *without waiting* for the previous process to be instantiated completely. Leaving this technical issue aside, we can write `spawn = zipWith (#)`.

There is a version `spawnAt` (not shown), taking an additional *placement list* as the first parameter. The latter list is a list of integers, mapping the consecutive processes onto machine numbers. To give an example, `spawnAt [1, 2, 2, 3] [f, g, h, h] [x, y, z, t]` will place processes to compute `f x` on the PE 1, `g y` and `h z` on the PE 2, and `h t` on the PE 3.

Now, the `parMap` accepts two arguments, the function `f` of type `a → b` and a task list `xs` of type `a`. `parMap` supplies `spawn` with an infinite list `repeat f` and the task list. Because of the laziness of Haskell and because `spawn` 'zips' the two input lists producing the result list with the length of the smaller input list, the infinite list of worker functions is not evaluated completely. The result of `parMap` is the list of the results. The semantic equivalent of `parMap f` is `zipWith (#) (repeat f)`. We see that—just as mentioned above—a process is created for each element of the input list. This approach is not efficient for long lists and few PEs. The next skeleton relaxes this drawback.

```
farmBase :: (Trans a, Trans b)
         ⇒ ([a] → [[a]]) -- ^ input distribution function
         → ([[b]] → [b]) -- ^ result combination function
         → Map a b       -- ^ map type signature

farmBase distr comb f tasks = comb $ parMap (map f) $ distr tasks

farm' :: (Trans a, Trans b)
      ⇒ Int     -- ^ number of child processes
      → Map a b -- ^ map type signature
farm' n = farmBase (unshuffle n) shuffle

farm :: (Trans a, Trans b) ⇒ Map a b -- compatible to map
farm = farm' (max 1 (noPe-1))
```

Figure 3.8: *The implementation of* `farm`*.*

Task farm. A classical approach from parallel computing is a *farm* [Hey, 1990, Klusik et al., 2001, Peña and Rubio, 2001]. We can implement it with a *statically task-balanced* parallel map: the input list is divided into sublists, as many as the number of PEs. Then the sequential `map f` for the worker function `f` is applied to these sublists locally. This approach works best if the processing time of separate elements is the same. For the implementation, a more generic `farmBase` skeleton is defined. It has the distribution and the combination functions as parameters. The type of `farmBase` is more generic than that of `map`. Please refer to Figure 3.8 for definitions of `farmBase` and `farm`. The latter is a drop-in replacement for the sequential `map`.

Let us discuss the code in Figure 3.8 in more detail. The function `farmBase` receives two special functions for input distribution (`distr`) and result combination (`comb`). The result of `distr` is a list of lists. The outer list is processed in parallel with `parMap`, which applies `map f` to the inner list. The result is combined with `comb` to a flat list, which is the result of the whole function. Now we instantiate the `farmBase` skeleton with the previously introduced functions `unshuffle` and `shuffle`. We need to partially apply `unshuffle` to the number of child processes `n`, yielding the type `[a] → [[a]]`. The function `shuffle` has the type `[[a]] → [a]`. The result is the definition of `farm'`. Yet the single additional integer parameter—the number of processes—can be figured out automatically, this happens in the definition of `farm`. As the constant `noPe` contains the number of PE for the current execution, we designate one PE as a processor for the master process and take care to have at least one worker by writing `max 1 (noPe-1)`. We have arrived at the definition of `farm`, which has the same type as `map`—up to the `Trans` context, namely `farm :: Trans a b ⇒ Map a b`.

Direct mapping. An interesting consequence of Eden's remote evaluation semantics is the 'direct mapping' trick. The *data* in Eden is to be evaluated to reduced normal form before transmission. However, the *process function*, i.e., the argument of `process` is not evaluated, but sent to the remote side 'as it is'. This allows to send unevaluated data to the remote process in the process abstraction. Assuming we want to send some value, let it be `x`, unevaluated. We define a process `process (const x)`, in this case `const :: a → () → a`, and instantiate it with `()`. In this case `x` will be evaluated remotely and sent back to the main process in its reduced normal form. The evaluation will happen as a part of the 'usual' communication of data back to the main process.

Of course, the most frequent usage of the 'direct mapping' method is to supply a working function (let it be `wf`) with data, without evaluating the data first. Assume, `wf :: a → b → c` and let `bigData` of type `a` be the input of `wf` which we do not want to evaluate on the sender side. Then we define a new function

```
wf' :: () → b → c
wf' () y = wf bigData y
```

```
ssfBase :: forall a b. (Trans a, Trans b)
        ⇒ ([a] → [[a]]) -- ^ input distribution function
        → ([[b]] → [b]) -- ^ result combination function
        → Map a b       -- ^ map signature
ssfBase  distribute combine f xs
   =  combine ( spawn [ pf (tasks i) | i ∈ [0..n-1] ] (replicate n ()) )
      where  tasks i  = taskss!!i
             taskss = distribute xs
             n = length taskss
             pf :: (Trans a, Trans b) ⇒ [a] → ( () → [b] )
             pf x = (λ_ → map f x)

ssf :: (Trans a, Trans b) ⇒ Map a b   -- pure map signature
ssf = ssfBase (unshuffle noPe) shuffle -- partial application

-- a version, which does not co-locate a task with master
ssf' :: (Trans a, Trans b) ⇒ Map a b                 -- pure map signature
ssf' = ssfBase (unshuffle (max 1 (noPe-1)) ) shuffle -- partial application
```

Figure 3.9: *The implementation of the self service farm.*

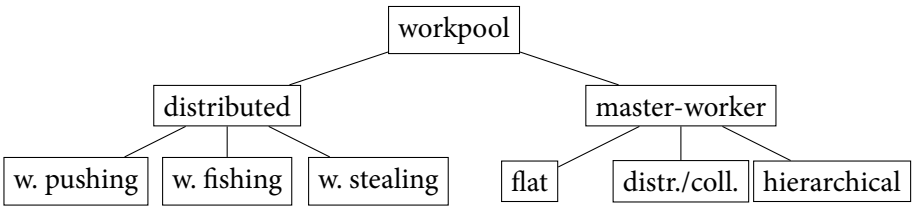


Figure 3.10: *A classification of workpools.*

If we now define a process with function `wf'`, the value of `bigData` will not be evaluated before transmission. It is crucial that `bigData` is not already evaluated, as in this case not the instruction for generating `bigData`—its unevaluated form, but the evaluated form of `bigData` is transmitted.

Using this method, we can build a `farm` version where the worker processes extract their tasks from unevaluated total task pool. We call it a 'self service' farm or `ssf`. An implementation of `ssf` has been sketched in [Klusik et al., 2001, Loogen et al., 2003]. The list of tasks is sent unevaluated to each worker. It is usually more efficient for large amount of input data. We show our variant in Figure 3.9. We place one worker task also at the first, 'master' machine: the number of workers is `noPe`, which differs from the `farm` implementation above. We have defined `ssf'` to mimic the default task placement of the `farm` skeleton.

Further, note the `forall` annotation in the type of `ssfBase`. It allows us to write the type of the `pf` helper function, whilst `pf` is not a top level function. Formally, it is an existential quantification of type variables `a` and `b`. The ∀ quantifier is implied by the type system. By putting a further ∀ in front of the type, we reverse the quantification, so the ∃ quantifier ensues. Thus, the types `a` and `b` are unique in the type of `ssfBase` and all underlying type annotations, i.e., in the type of `pf`.

The self service farm emulates data input from a network drive shared by all the processing elements. If *evaluating* the data is not costly, but *transmitting* is, the `ssf` is an approximation to the behaviour of the system with initially distributed input.

Workpool scheme, an overview. The *dynamically task-balanced* parallel map is an example of the master-worker computation scheme, the more generic name is `workpool`. The cost of processing the separate elements may be different. Hence it is better to do the task balancing at the execution time

in this case [Foster, 1995, Grama et al., 2003]. Approaches to an implementation of a workpool either designate a master process which distributes and collects tasks or use a distributed scheme.

Let us regard the first approach first, it is called a master-worker scheme. Grama et al. [2003] call it 'centralised'. Further names are 'master-slave' and 'manager-worker'. If we have a single master and all other processes are workers, it is called a *flat* master-worker scheme. Such scheme is the simplest master-worker implementation, *cf.* Figure 3.10 for a classification. More complicated versions include *hierarchical* master-worker schemes and a splitting of a master in a 'distributor' and 'receiver'. The first approach uses a hierarchy of masters instead of a single master. An Eden implementation is presented in [Priebe, 2006, Berthold et al., 2008]. The second approach uses a single master process for distributing the tasks, but another process for collecting the results. Hence, it is called a *distributor/collector workpool*. Such an approach has been directly implemented in [Dieterle, 2007]; however, it can be also straightforwardly implemented using the remote data concept [Dieterle et al., 2010b]. Another orthogonal classification of master-worker schemes is the *scheduling* type [Grama et al., 2003]. If a worker asks the master for a single task, such behaviour is called 'self-scheduling'. If a worker reports a shortage on tasks and obtains a bunch of tasks from the master, it is called 'chunk scheduling'. List chunking from Section 3.2.1 can be used to implement chunk scheduling. An option similar to the scheduling issues is the *prefetch*. It is the amount of tasks initially distributed to the workers. So for prefetch value of k each worker needs to process k tasks before it launches its first task request.

As for the distributed workpools, such schemes do not have a designated master. Because the workers process their tasks at different pace, one worker might have more tasks left than another. Hence, in the distributed workpool setting the problem of load balancing ensues. Various approaches to solve this problem include work pushing, work fishing, and work stealing. The common idea of all these methods is to distribute the remaining tasks to the workers more equally. A knowledge about the availability of the tasks at the neighbour worker processes is required for all these approaches. Let us call a worker 'idle' when it has few tasks or none left. We call a worker 'busy' if it is working on a task and has enough tasks left. Work pushing is a strategy when a busy worker *pushes* a few not yet processed tasks to another worker that seems to have fewer tasks, i. e., is idle. Work *fishing* is when an idle worker chooses randomly a busy worker to give the former its largest non-processed task, *cf.* [Janjic and Hammond, 2010]. Work stealing is when an idle worker *steals* not yet processed tasks from a busy worker [Blumofe and Leiserson, 1999]. Because of the dynamic load balancing, workpool is not deterministic *per se*. The reason is that the precedence in which the tasks are processed is determined at the runtime, a minor change in the execution time of a single task can affect the distribution of all further tasks. Eden implementation of a distributed workpool has been presented in [Dieterle, 2007, Dieterle et al., 2010a].

A further landmark in the workpool classification is the possibility of *dynamic task creation*. A problem arises for a working function not of the type `a → b`, but of the type `a → (b, [a])`. In other words, the working function might produce new tasks, but not necessarily does so in each run. This issue is tackled by workpools with dynamic task creation. Some further questions arise for the latter, e. g., how the task pool is transformed, given the old task pool and freshly created tasks. Possible strategies include appending new tasks to the beginning or the the end of the list of old tasks. Dynamic task creation is possible for both master-worker-based (for Eden implementations see [Priebe, 2006, 2007, Brown and Hammond, 2010]) and distributed workpools (*viz.* [Dieterle et al., 2010a]). In the latter setting, the problem of distributing tasks to workers becomes even more sharp in the context of dynamic task creation.

Simple workpool implementations. Below we will regard three implementations of a simple master-worker workpool, with flat hierarchy and no dynamic task creation. The difference between the first two versions lies in the way they deal with non-determinism, while the third one ignores this problem. We show the first skeleton, `workpoolSorted`, and a core implementation of a workpool, `workpoolAux`, in Figure 3.11. The second skeleton, called `workpoolSortedNonBlock`, is in Figure 3.12. The third option: `workpool'` is in Figure 3.13. The same figure aliases the default workpool implementation to the `workpoolSorted` skeleton. We use it when we generally need a deterministic workpool, without a

```
workpoolAux :: (Trans t, Trans r)
            ⇒ Int      -- ^ number of child processes (workers)
            → Int      -- ^ prefetch of tasks (for workers)
            → (t → r) -- ^ worker function
            → [t]      -- ^ inputs
            → IO ([Int], [[Int]], [[r]]) -- ^ non-combined output
-- result format: (input distribution, task positions, results)
-- input distribution format: input i is in i-th position in the list,
-- task positions format: element i of result-sub-list j is
-- in the input list at (poss!!j)!!i
workpoolAux np prefetch worker tasks
   = return (reqs, poss, fromWorkers)
    where   fromWorkers     = parMap (map worker) taskss
            (taskss, poss)  = distributeWithPos np reqs tasks
            -- generate only as many reqs as there are tasks
            reqs            = map snd $ zip tasks $
                              initialReqs ++ newReqs
            initialReqs     = concat (replicate prefetch [0..np-1])
            newReqs         = merge $ mkPids fromWorkers

-- the sorting, blocking variant
workpoolSorted :: (Trans a, Trans b)
               ⇒ Int        -- ^ number of workers
               → Int        -- ^ prefetch of tasks (for workers)
               → Map a b
workpoolSorted np prefetch f tasks = res
    where (_, poss, ress) = unsafePerformIO $ workpoolAux np prefetch f tasks
          res = map snd $ mergeByPos ress'
          ress' = map (uncurry zip) (zip poss ress)
```

Figure 3.11: *A simple, sorting, blocking workpool, based on* `workpoolAux`*.*

```
workpoolSortedNonBlock :: (Trans a, Trans b)
         ⇒ Int -- ^ number of child processes (workers)
         → Int -- ^ prefetch of tasks (for workers)
         → Map a b
workpoolSortedNonBlock np prefetch f tasks = orderBy fromWorkers reqs
   where (reqs, _ ,fromWorkers) = unsafePerformIO $ workpoolAux np prefetch f tasks
```

Figure 3.12: *The base, non-sorting* `workpoolAux` *implementation and a simple, sorting, non-blocking workpool. Helper functions are in Figure 3.14.*

```
-- this is the default workpool
workpool :: (Trans a, Trans b) ⇒ Int → Int → Map a b
workpool = workpoolSorted

-- this is the non-sorting workpool with usual interface
workpool' :: (Trans a, Trans b) ⇒ Int → Int → Map a b
workpool' = concat $ (λ(_, _, x) → x) $ unsafePerformIO $ workpoolAux
```

Figure 3.13: *The top-level interfaces for the workpools.*

```
-- helper functions for workpoolAux
-- task distribution according to worker requests
distributeWithPos ::  Int  -- ^number of workers
                  → [Int] -- ^ request stream (IDs from 0 to n-1)
                  → [t]   -- ^ task list
                  → ([[t]], [[Int]]) -- ^ result
-- result format: (task positions in original list, task distribution),
--                each inner list for one worker.
distributeWithPos np reqs tasks
   = unzip [unzip (taskList reqs tasks [0..] n) | n ∈ [0..np-1]]
    where taskList (r:rs) (t:ts) (tag:tags) pe
                        | pe == r   = (t,tag):(taskList rs ts tags pe)
                        | otherwise =  taskList rs ts tags pe
          taskList _ _ _ _ = []

-- replace each element of the inner list with the id of its inner list.
mkPids :: [[r]]   -- ^ workpool output lists
       → [[Int]] -- ^ IDs
mkPids rss = [[ i | j ∈ rs] | (i,rs) ∈ zip [0..] rss]

-- helper functions for workpoolSorted
-- join sorted lists into one sorted list, using a binary scheme
mergeByPos :: [[(Int, r)]] → [(Int, r)]
mergeByPos [] = []
mergeByPos [wOut] = wOut
mergeByPos [w1, w2] = merge2ByTag w1 w2
mergeByPos wOuts = merge2ByTag (mergeHalf wOuts) (mergeHalf (tail wOuts))
    where mergeHalf = mergeByPos ∘ (takeEach 2)

merge2ByTag :: (Ord i) ⇒ [(i, r)] → [(i, r)] → [(i, r)]
merge2ByTag [] w2 = w2
merge2ByTag w1 [] = w1
merge2ByTag w1@(r1@(i,_):w1s) w2@(r2@(j,_):w2s)
                        | i < j = r1: merge2ByTag w1s w2
                        | i > j = r2: merge2ByTag w1 w2s
                        | otherwise = error "found tags i == j"

-- helper function for workpoolSortedNonBlock
-- order a nested list by a given distribution
orderBy :: Integral idx
         ⇒ [[r]] -- ^ nested input list
         → [idx] -- ^ distribution
         → [r]   -- ^ ordered result list
orderBy rss (r:reqs)
  = let (rss1, (rs2:rss2)) = splitAt (fromIntegral r) rss
    in  (head rs2): orderBy ( rss1 ++ ((tail rs2):rss2) ) reqs
orderBy _ _ = []
```

Figure 3.14: *Helper functions for the workpool implementations.*

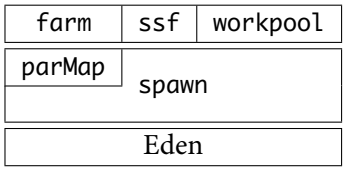


Figure 3.15: *The map-like skeletons in Eden.*

preference in the implementation of sorting. Finally, Figure 3.14 shows the helper functions for all of the above skeletons.

The technical reason for the non-determinism in this workpool implementation lies in the way the result streams from workers are combined into the final result stream, see the value `fromWorkers` in the function `workpoolAux`. The typical implementation puts an element into the result stream as soon as it arrives, using the function `merge`, thus completely destroying the order of elements of the input list. It is non-deterministic in the order of its results, which is signalled by the `IO` monadic type of the skeleton `workpoolAux` in Figure 3.11. It forms the core of a *non-sorting* workpool. However, it is quite easy to make it deterministic. The simplest approach is to 'tag' the input data with natural numbers and then to *sort* the results by their tags. It will be still non-deterministic in the order in which the results are computed from the input data, however, the order of resulting elements in the output list will always correspond to the input elements. Thus, a 'sorting workpool' is a deterministic interface to the dynamic task balancing. We choose a slightly more complicated approach.

The implementation of `workpoolAux` relies on functions `distributeWithPos` and `mkPids` from Figure 3.14. The skeleton `workpoolAux` saves the task positions on the workers. Surely, it is possible to implement a *sorting* workpool, which *blocks* on its output, see `workpoolSorted` in Figure 3.11. The helper function `mergeByPos` is defined in Figure 3.14. The task positions carry enough information to restore the correct order of the output list. Further, these positions are available locally and do not need to be communicated. Note that the base implementation of `workpoolAux` is shared by all three implementations.

We present two further workpool modifications. Both utilise the `workpoolAux` core. The first, `workpoolSortedNonBlock`, uses the precedence of the tasks to sort the results. The benefit of this version versus the simple tagging idea is that the tags do not need to be communicated in both directions. With this implementation we can access a particular element in the result (provided, it is already computed), even though all other elements are not yet computed. Using this information and a helper function `orderBy` (see Figure 3.14), the correct order of the inputs can be recovered. This is the *non-blocking, sorting* workpool implementation (`workpoolSortedNonBlock`), that we show in Figure 3.12. Note that the efficiency of a non-blocking workpool is damaged by a not quite effective implementation of the `orderBy` function.

The last skeleton, called `workpool'`, performs an `unsafePerformIO` call, takes the unsorted nested list of results from `workpoolAux`, flattens it, and returns the resulting unsorted list. The sorting is to be done in the application code. Thus, this is the *non-sorting, non-deterministic* workpool. Its user needs to ensure that the order of the tasks does not matter in the remaining program. We show this 'adaptor' skeleton in Figure 3.13.

A hierarchy of the `map`-like skeletons is shown in Figure 3.15. This figure can be interpreted as 'building blocks': the foundation, i. e., Eden primitives, is in the bottom and the `map`-like skeletons with task-balancing features are on the top.

3.2.3 Divide and Conquer

The divide and conquer scheme can be implemented as an algorithmic skeleton. The generic type of a divide and conquer skeleton, working from task of type `a` to result of type `b`, is

```
type DC a b = (a → Bool)          -- ^ is trivial?
            → (a → b)             -- ^ solve
            → (a → [a])           -- ^ divide
```

```
divConSeq_c :: DC a b
divConSeq_c isTrivial solve divide combine x
  | isTrivial x = solve x
  | otherwise = combine x $ map self $ divide x
  where self = divConSeq_c isTrivial solve divide combine
```

Figure 3.16: *Sequential divide and conquer skeleton.*

restriction	input with	input without
independent subproblems	dcA	
bounded depth of call tree	dcB	
fixed tree arity	dcC	
multiple block recursion	dcD	
elementwise operations	dcE	
correspondent communications		dcF

Table 3.3: *Classification of divide and conquer by [Herrmann, 2000].*

class	with input	without input
dcA	quicksort	
dcB		naive *n* queens
dcC	**Karatsuba multiplication**	schoolbook multiplication
dcD	triangular matrix inversion	
dcE	matrix-vector multiplication	
dcF		**fast Fourier transform** **matrix multiplication**

Table 3.4: *Examples for the particular divide and conquer classes. We still follow Herrmann [2000].*

```
  → (a → [b] → b)    -- ^ combine
  → a → b            -- ^ input and result
```

The sequential case is quite easy, see Figure 3.16. We will introduce new divide and conquer skeletons in Chapter 6.

Classification. A classification of divide and conquer skeletons has been presented in [Herrmann, 2000]. For example, notice the type `DC a b` to have the input `a` in the type of the `combine` function `a → [b] → b`. It is possible to implement some divide and conquer algorithms without using the input in the `combine` function, but a more generic variant requires such a parameter. The classification follows.

Herrmann [2000] classifies divide and conquer into six types, from dcA to dcF. Each of them is a subset of the previous one, e.g., dcD ⊂ dcC holds [Herrmann, 2000, Chapter 3]. The classification of divide and conquer schemes and the relations of the introduced divide and conquer classes is presented in Table 3.3. Their relation is emphasised in Figure 3.17. The classes dcA to dcE are defined for the divide and conquer with the `combine` function which requires the original problem. This is designated with 'with input' in the table. The 'with input' variants are more powerful than the 'without input' case, as the latter can be simulated with the former. But the class dcF is defined for problems in the 'without input' category. In Table 3.4 we show examples for divide and conquer classes. The algorithms discussed in this thesis are marked **bold**. The schoolbook multiplication is not marked bold, as we do not implement it as a divide and conquer instance.

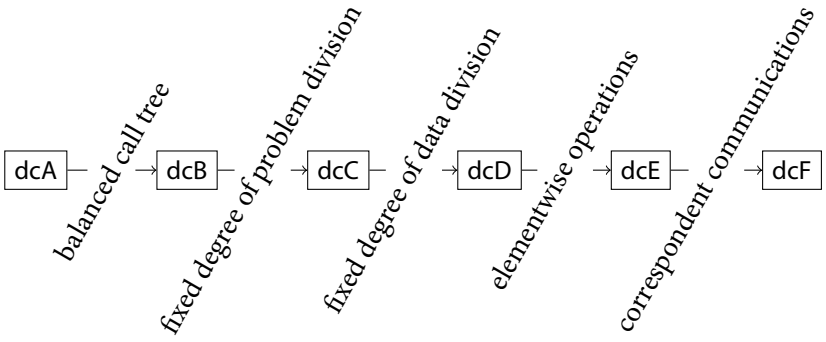


Figure 3.17: *Relation of the dcX classes following [Herrmann, 2000].*

```
divConPar :: (Trans a, Trans b)
      ⇒ Int → DC a b
divConPar d isTrivial divide solve combine x
  | d ≤ 0 = divConSeq_c isTrivial divide solve combine x
  | isTrivial x = solve x
  | otherwise = combine x $ parMap self $ divide x
  where self = divConPar (d-1) isTrivial divide solve combine x
```

Figure 3.18: *An adaptation of divide and conquer skeleton with depth control from [Loogen et al., 2003].*

Implementations. For a simple *parallel* divide and conquer implementation, replace in the sequential implementation (Figure 3.16) the `map` with `parMap`. Such divide and conquer skeleton has been implemented in [Loogen et al., 2003]. The source code is presented in Figure 3.18. This variant of the divide and conquer is *not regular*, i.e., no fix number of children in each step is assumed. However, such an approach is not optimal.

We have developed special divide and conquer skeletons for dcC class. Such skeletons are presented in Chapter 6. In the same chapter we present alternative approaches for specific dcF applications, namely the fast Fourier transform and Strassen matrix multiplication.

3.2.4 Iteration Skeleton

A further important pattern in the theory of computing, especially in scientific computation, is *iteration*. For example, a **for** or a **while** loop are extremely frequent patterns in the imperative programming languages. The purely functional counterpart of this pattern is often expressed with recursive calls or with a contraction function, e.g., `iterate`.

A classical imperative **for** loop can be implemented in parallel under the following conditions. Assume the loop is executed n times. If for some positive integer p the loop body can be partitioned into $\lceil n/p \rceil$ blocks, and in each of these blocks single loop iterations are not depending on each other, then the loop can be implemented in parallel on p processors. The scheme is very simple and is known as *speculative* parallelisation. The first p loop iterations are executed in parallel, then the second p loop iterations follow, until the $\lceil n/p \rceil^{\text{th}}$ block of at most p loop iterations concludes. In Chapter 5 we will consider the case when the computation can be aborted before the loop is over and when the blocks do not depend of each other—the parallel functional counterpart of a special **for** loop with a conditional **break**.

Consider a more generic case of dynamic task creation. So, not only do we want to abort the loop at some as-of-now unclear point, but also we would like to introduce further iterations *while* the loop is already being executed. This also means a present dependency between the blocks. A sequential imperative counterpart is a **do**–**while** loop. Let us inspect the requirements in more detail. We would like to start a few speculative tasks in parallel. As we need to produce more tasks during the computation,

```
parWhile :: (Trans local, Trans task, Trans subResult)
         ⇒ localM → [local] → dataIn → (dataIn → [task])
         → (local → task → (subResult, local))
         → (localM → [subResult] → Either result ([task], localM))
         → result
parWhile lm ls d divide fworker comb = result    where
  tasks                 = divide d
  outss                 = parMap (p fworker) (zip ls taskss)
  outssComb             = zipWith comb lms (transpose' outss)
  (more, ~[end])        = span isRight outssComb
  result                = fromLeft end
  moreTaskssLms         = map fromRight more
  (moreTaskss, morelms) = unzip moreTaskssLms
  taskss                = transpose' (tasks : moreTaskss)
  lms                   = lm : morelms

p :: (local → task → (subResult, local)) → (local, [task]) → [subResult]
p f (local, tasks) = subResults
        where results = zipWith f (local : moreLocals) tasks
              (subResults, moreLocals) = unzip results
```

Figure 3.19: *The `parWhile` skeleton from [Loogen et al., 2003] with minor modifications. The helper functions are in Figure 3.20.*

it is better to have a special function to produce the tasks from a single 'big task'. We call it `divide`. A working function `wf` to process the tasks is a straightforward requirement. Further we designate the `comb` function. It unites the check for termination and the generation of further tasks, possibly using the `divide` function for the latter purpose. Combined, the `parWhile` skeleton emerges, *viz.* Figure 3.19 with helper functions in Figure 3.20. It has been stated as `iterUntil` in [Loogen et al., 2003]. Our version includes some minor modifications. We define an alternative approach in Chapter 5.

Let us discuss Figure 3.19 in more detail. The skeleton shown there uses the `Either` type. It is available in Haskell library `Data.Either`. It is the *sum type* of two arbitrary types `a` and `b`, holding either a value of type `a` or a value of type `b`.

Example 3.5. *Consider a sum type of a string and an integer.*

```
stringOrInt :: Either String Int
stringOrInt | inOneCase = Left "failure"
stringOrInt | otherwise = Right 42
```

The value of `stringOrInt` is either 'failure' or '42', depending on the value of the boolean expression `inOneCase`. It has a different type than `intOrString`, being, for example, `Left 42`.

Some helper functions are also available from `Data.Either.Unwarp` library, we state some of them in Figure 3.20. Note that in our implementation the 'getter' functions are partial: `fromRight (Right 42) = 42`, but `fromRight (Left "failure") = ⊥`. A further helper function is `transpose'`. Semantically it is the same list transposition function `transpose` of type `[[a]] → [[a]]`, as available from `Data.List`, but `transpose'` is more lazy. Further, the library function `span` is used.

```
span :: (a → Bool) → [a] → ([a], [a])
```

Called with a predicate and a list, it returns a pair of lists. While the predicate holds, the first output list is filled. If the predicate is once violated, the second list obtains all the remaining elements of the input list.

```
transpose' :: [[a]] → [[a]] -- lazy transpose
transpose' (xs:xss) = mzipWith (:) xs (transpose' xss)
transpose' _        = repeat []

-- lazy zipWith
mzipWith f (x:xs) ~(y:ys) = f x y : mzipWith f xs ys
mzipWith f _ _ = []

isRight :: Either a b → Bool
isRight (Left _)   = False
isRight (Right _)  = True

fromLeft :: Either a b → a
fromLeft (Left a)   = a

fromRight :: Either a b → b
fromRight (Right b) = b
```

Figure 3.20: *Helper functions for Figure 3.19.*

3.3 Eden Tracing

For the purpose of evaluation of parallel behaviour, a deep insight in program execution is required. We would like to know how large the communication overhead is, whether some unneeded blocking arises, whether all PE are supplied with work, etc. For these purposes, the Eden executables can be instrumented with specific message output. The latter is called a *trace* or an activity profile. After program termination, these messages are collected and evaluated. For this, a special program, *Eden TraceViewer*, also called EdenTV, has been written. More details are provided in [Berthold and Loogen, 2007b]. Similar tools for other platforms include [Geimer et al., 2010] and [Wheeler and Thain, 2009]. Here we address the issue of interpreting the resulting diagram.

We call such diagrams *trace visualisations*. A more insightful term would be 'visual representation of a parallel post-mortem activity profile'. The x axis of a trace visualisation represents time. The y-axis shows PE numbers. The horizontal bars represent Eden processes, multiple processes can be placed on one PE, but normally only one of them will be executed at once. The colour of the bar corresponds to the traffic lights. Red stands for *blocked*—the process is waiting for input or is deferred from running by other means. Yellow is *runnable*, a process in this state *could* be running, but is not. Typical causes for this include communication in progress and garbage collection. Finally, green means *running*. Optionally, communication might be indicated with black arrows from one process to another. We state such matter explicitly when we include it in our traces. A typical trace picture is in Figure 3.21.

Let us interpret it! We have here a program execution in 0.45 seconds, running on 8 PEs, one process pro PE. We see that the processes wait quite long for their input. This is due to PE 1, which presumably is computing the tasks to distribute. Further, PE 2 has two phases of activity, while PEs 3–8 each have only one. Let the PE 1 be the 'master' and all other PEs be 'workers'. We see here three issues:

- Input data preparation takes too much time. It should be optimised or at least delegated to the 'worker' PEs, e. g., with the *direct mapping* technique.
- The worker tasks do not start concurrently: PEs 4 and 6 exhibit a delay compared with PE 2 and PE 8. An input distribution issue might be a reason for this.
- Load balancing is by far not perfect. PE 2 has two activity phases, while other workers have one. We interpret this as two tasks running on PE 2. We should either reduce the number of tasks

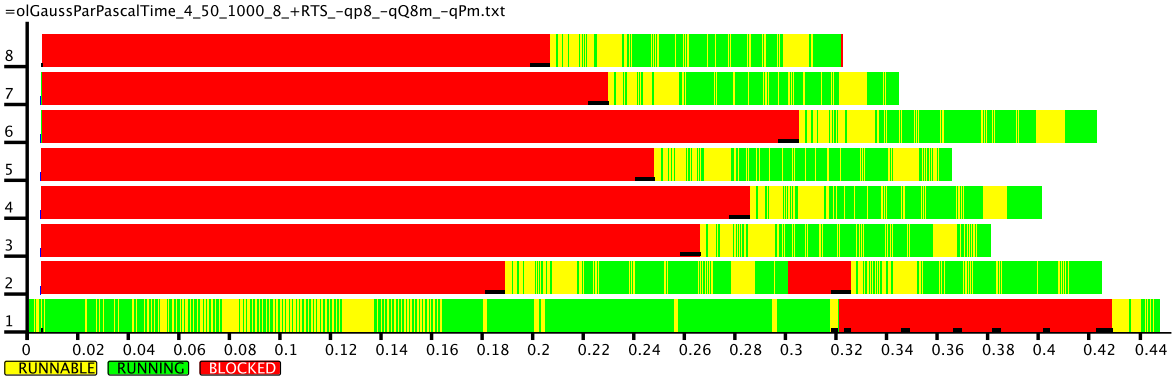


Figure 3.21: *Example of a trace.*

by one, or create a second process on PE 1. As the master process is idling while it waits for the workers to be done, a further worker process can be co-located on the PE 1.

Without tracing, obtaining the above insight would be not trivial.

3.4 Measurement Methodology

> Before Philosophy can teach by Experience, the Philosophy has to be in readiness, the Experience must be gathered and intelligibly recorded.
>
> Thomas Carlyle, *On History*

We perform a multitude of runtime experiments in this work. They serve not only the purpose of verifying that our approach actually works. The actual goal of such experiments is twofold.

- It enables us with a possibility to *compare* different skeletons, both sharing a similar approach and based on completely different ideas.

  An example of the latter are two different divide and conquer implementations, one based on direct task placement, and other creating many small tasks computed in parallel. An example for the first goal are two different underlying `map`-like skeletons for computing the aforementioned small tasks, say, `farm` and `workpool`.

- We can both visualise the program run with EdenTV [Berthold and Loogen, 2007b] and compute speedup and some numerical quality measures (e.g., serial fraction) of our parallel program, and hence we can detect and enquire into bottlenecks of our implementation. Thus we acquire precious information for the development of better skeletons.

Hardware. Before we go into the details of the measurement methodology, we need to discuss the hardware platforms our implementations are executed on. We use both SMP and distributed memory hardware on both 32 bit and 64 bit x86 architectures. We denote different hardware with shortcuts we will use in further text when we refer to it.

- The `sakania` is an SMP Intel Xeon machine with 8 cores running at 2.5 GHz of the Faculty of Mathematics and Computer Science at Philipps-Universität Marburg. The cores are aligned in two quadcore chips. The system has 16 Gb RAM and runs 64 bit Linux OS. A PE of this machine is a single core, hence we use `sakania` as a 8 PE parallel machine. The multicore machine has almost negligible communication cost. Moving the data from one core to another in this

setting is merely copying the data in memory. We still need to copy the data because Eden is a distributed memory language.

Originally our skeletons were designed for distributed memory systems. However, they were proved to work quite well also for SMP systems. So, in this case we simulate the communication within a shared memory system, without utilising the fact of having the shared memory. An advantage is mostly negligible communication time, which was the reason for the poor behaviour of many distributed memory systems. The benefit of having separate heaps is in natively distributed garbage collection, which is otherwise a larger issue in a parallel system, see [Berthold et al., 2009a, Marlow et al., 2009].

- The 'local workstations' means the workstation cluster of the Faculty of Mathematics and Computer Science at Philipps-Universität Marburg. These machines are dual-core workstations, with 2 Gb RAM each, interconnected with 100 MBit/s Ethernet. This is a genuinely distributed memory setting. We have two additional special cases for using these machines which we will always state separately. One is using *double* as much PEs as we have machines. In this case we use both cores as separate PEs. It will be seen on the traces for skeletons of a master-worker setting that a particular PE has much smaller communication overhead. This will be the *second* core of the master machine. The other case is when we deliberately add further slower and faster machines to the network. This is to mimic different load of the PEs and serves mainly to test the dynamic load balancing of specific skeletons.
- The 'Beowulf cluster' is a Beowulf cluster at Heriot-Watt-University in Edinburgh, UK. It consists of 32 Intel Pentium 4 SMP processors running at 3 GHz with 512 Mb RAM and a Fast Ethernet interconnection. The operating system is 32 bit Linux OS.

Methodology. One of the foundations of our performance analysis is execution time measurement. Unless otherwise stated, we measure the *complete* execution time of a given program with wall clock. We perform the measurement five times and use the mean of the obtained values as the actual execution time in a given setting. If we use some further runtime arguments, e.g., for tuning the memory allocation, we state it explicitly. We do not elaborate on the usage of the option `-qp`p for executing the program on p PEs. Furthermore, we sometimes do not elaborate on the option `-qQ`m for setting the message buffer to m bytes. We consider this option as rather technical and hence uninteresting in most cases.

The 'interesting' options include `-H`k and `-A`l for setting the initial heap size to k bytes and for instructing the runtime system to allocate further space in l byte pieces respectively. The correct usage of these options reduces the execution time of the programs slightly, as less time is spent on garbage collection (GC). However, abuse of too large values of k leads to increased execution time due to allocation of in fact unneeded memory. Further, the `-H`k option mostly reduces the *number* of GCs drastically, but does not really control the *time* of GCs. In most cases the correct usage of this option results in few moderately large GCs versus the abundant small GCs in the standard, untuned case. Depending on memory usage in the program, abuse of the `-H`k option may result in a handful of large GCs. However, these few large GCs typically occupy even more time than numerous small GCs. As we can see, it is a trade-off in both directions, hence we always annotate the usage of memory-tuning options in the runtime system. It is possible to use the `-t` option to let the RTS output the number and the total duration of garbage collections.

3.5 Conclusions

We presented Eden [Loogen et al., 2005] and existing algorithmic skeletons [Breitinger et al., 1995, Loogen et al., 2003, 2005, Eden Skeletons, 2011]. We included in our presentation a new language feature [Dieterle et al., 2010b], the map-like skeletons, a simple skeleton for divide and conquer, and an approach to iterative computation. The classification of divide and conquer skeletons presented after Herrmann [2000] will be used in Chapter 6. We also presented the EdenTV tracing visualisation tool

[Berthold and Loogen, 2007b] that would help us analyse our parallel programs. Finally, we devised the measurement methodology we will adhere to in the following.

Chapter 4

ESTIMATING PARALLEL PERFORMANCE

> Je vous reprochais tout à l'heure d'estimer la perfection des choses par votre capacité; et je pourrais vous accuser ici d'en mesurer la durée sur celle de vos jours.*
>
> Denis Diderot, *Lettre sur les aveugles*

Measures for the quality of parallelisation are essential for the systematic evaluation of the design and implementation of parallel software. The most simple and wide-used is the *speedup*, a measure for how much faster the program runs on p parallel processors compared to the sequential case. However, there are various reasons for good and bad speedup values. Finding out the possible causes for bad speedups would allow the programmer to detect the code sequences causing loss in performance. This is the first step towards optimisation. The measurement is repeated with the new parallel version, and the whole approach is iterated until a satisfying result emerges.

In the context of this thesis, a technique for prediction and generalisation of parallel execution times is especially useful. We can test our programs only on a limited number of hardware architectures. However, we aim for parallelisation methods which are usable in a broader setting. We want to find good parallelisation approaches which might be useful on other architectures. We want to detect bad parallelisations and, possibly, to obtain a hint how to improve them.

We suggest a model for a coarse division of the parallel runtime into 'good' and 'bad' parts. Contrary to the popular display of parallel runtime being the sequential one 'sped up' to some factor less than the number of processing elements, we envision parallel runtime as the sequential 'work' distributed over a number of (uniform) processing elements—the 'good' part—plus an additional penalty term—the 'bad' part. This partition is our original concept. The notion of total parallel overhead is related, but different from our approach. See below for discussion of it.

The first goal of this chapter is accurate prediction of parallel runtimes for new input sizes and for non-available numbers of processors. Our approach is to measure the sequential work and to obtain the parallel overhead for a set of sample input sizes or sample numbers of processors. Statistical techniques are then used to extrapolate and to estimate the values for further input sizes or other numbers of processors. As the parallel execution time is expressed in terms of the sequential work and of the parallel overhead, this approach enables the forecast of parallel runtime. Secondly, the estimated parallel overhead is a measure for the scalability of a given parallel program. It provides insight into the performance properties of a parallel program. If the 'bad' part is increasing, bottlenecks or similar problems in the code are likely. Hot-spot analysis is not the objective of this chapter.

In this chapter we show the practicality of our approach with two examples: a simple example program in Eden, and a large-scale C+MPI [ISO/IEC 9899:1999, MPI, 2009] program. Throughout the thesis further examples in Eden are discussed. The latter programs are parallelised using a skeleton library. Although our Eden system is used for most of the experiments, our approach is completely language-independent, as an example in this chapter shows. Our technique is applicable to any parallel system, ranging from a multicore machine to a supercomputer. The skeletons are also not a must for program *implementation*, they merely describe a particular pattern of parallel computation in the analysed program. This chapter is based on our collaborative work with Rita Loogen [Lobachev and Loogen, 2010c]. An extended version of this paper is [Lobachev et al., 2011].

*I reproached you just now with estimating the perfection of things by your own capacity; and I might accuse you here of measuring its duration by the length of your own days.

In the following text we firstly consider existing measures for parallel performance in Section 4.1, most prominent examples are efficiency and serial fraction. Secondly, we present our own model for a parallel computation in Section 4.2. From it, a further quality measure is later derived. On the basis of the measurement data from previous program executions, we can estimate the execution time for larger input sizes or larger amount of parallel processing elements (PEs). Section 4.3 explains how it is done. An example is in Section 4.4. We present another example, this time from a large-scale simulation, in Section 4.5. We discuss similarities and differences between our approach and the serial fraction in Section 4.6. Related work for the estimation of parallel runtime is discussed in Section 4.7. We conclude in Section 4.8. The relevant estimations of further parallel programs are presented in the corresponding chapters.

4.1 Related Work for Parallel Performance

Speedup and Co. A comprehensive summary of existing approaches is presented in [Kumar and Gupta, 1994, Grama et al., 2003], for a more recent overview see [El-Rewini and Abd-El-Barr, 2005]. As mentioned above, the most popular measure for the quality of parallelisation is the speedup. Recapitulate, the speedup $s(n,p)$ for task size n on p PE is defined as $T(n)/T(n,p)$. Here $T(n)$ is the best sequential runtime, and $T(n,p)$ is the parallel execution time on p PE. Ideally, $s(n,p) \geq p$ holds. Notice that $T(n,p) = T(n)/s(n,p)$ holds. *Efficiency* $e(n,p)$ is tightly connected to the notion of speedup. It is defined as $e(n,p) = s(n,p)/p$. However, some factors impact the reliability of speedup [Grama et al., 2003].

Firstly, superlinear speedup may occur. It is most of the time seen as a *non plus ultra* result in parallel computing. Superlinear speedup might be caused by positive effects of the parallel setting, e. g., better cache usage. But, speedup values are not reliable if the sequential program performs badly. This is the reason for demanding the best sequential implementation in the speedup definition. Naturally, bad sequential times can cause a superlinear speedup. So, the latter is definitely a reason to rise an eyebrow. This issue cannot be detected by inspecting the speedup alone, but can be figured out with the *serial fraction* defined by Karp and Flatt [1990]. We will discuss it in more detail later, on page 48.

Secondly, the canonical speedup definition demands the same input size for the sequential and for the parallel version. However, this is not practicable for large-scale computations. There are often cases when so many data are generated that they barely fit into the memory distributed over numerous PEs. A sequential execution is plainly not possible in this case. Other supporting issues are, on the one hand the long execution time of the sequential program if the task size is too large. A constant but proportionally large overhead for too small task size in the parallel case is also a reason for problems with conventional speedup computation, on the other hand. A method called *scaled speedup* [Gustafson et al., 1988] can be used in such cases. Execution time measured on a sequential machine for smaller inputs is compared with parallel time for larger inputs. *Isoefficiency* [Grama et al., 1993], as the name suggests, aims to keep the *efficiency* the same by choosing the appropriate number of PE *and* size of input.

Amdahl's law. The limits of parallelisation are a very old question. How far can one push the parallel program? Will it scale for a very large number of PE? The theoretic answer to this problem is given by the Amdahl's law [Amdahl, 1967, Hill and Marty, 2008]. It states the limits for the execution time of a parallel program in generic. Amdahl's law divides such program in two parts: the perfectly parallelisable part $\pi < 1$, which can be scaled to as many PE as desired, and the serial part $1 - \pi$, which cannot be divided onto multiple PE. So, the claim of Amdahl's law is that for p PEs the maximal speedup is

$$\frac{1}{1 - \pi + \pi/p}.$$

If we now aim for infinitely many PEs, the maximal speedup tends to be $1/(1-\pi)$. In other words: the size of the serial part $1 - \pi$ restrains the overall speedup. Again, an extension of this law similar to the case with speedup exists. Amdahl's law fixes the input size for all considered cases. Gustafson [1988]

relaxes the consequences of the Amdahl's law if the parallel program operates on arbitrarily large task sizes, which *scale* with number of PE.

4.2 Our View on Parallel Computation

We suggest another subdivision of the work in a parallel program. To do so, we need to agree, what such work is. For input size n, let $W(n)$ be the total work of the (sequential) program. Since it is hard to identify it, we assume that the sequential program does nothing unnecessary. Writing $W(n) = T(n)$, we identify the sequential work with the sequential execution time.

The common notation for execution time of a program with input size n on p PEs is $T(n,p)$. We denote this work done with p PEs by $W(n,p)$ and assume that $W(n,p) = pT(n,p)$. In a parallel execution, the sequential work is distributed over p PEs. Now in a parallel program, running in $T(n,p)$ time on p processors, more work than $W(n)$ is done. In a parallel execution, the sequential work is distributed over p PEs. But $W(n,p) \geq W(n)$! The distribution causes a total parallel overhead, denoted here with $A(n,p)$ (*cf.* [Grama et al., 2003]). Then

$$W(n,p) = W(n) + A(n,p).$$

But the total parallel overhead is also distributed over the parallel PEs. Let $B(n,p) = A(n,p)/p$, we define the distributed *parallel overhead* by splitting the total overhead equally onto PEs. This has not been introduced before. As we prefer to deal with time and not with amount of work, we can divide the formula for $W(n,p)$ by p, yielding

$$T(n,p) = T(n)/p + B(n,p). \tag{4.1}$$

The point of view on parallel computation, provided by (4.1), is our contribution. It differs from the speedup perspective. The latter envisions parallel time as sequential time divided by speedup, which is typically smaller than number of PE. Our approach defines the parallel time as the sequential time equally distributed onto full number of PE plus an addition term $B(n,p)$. We call this term a parallel penalty or a parallel overhead.

Now we have an expression where all values with the exception of the parallel overhead are known or can be measured. A trivial transformation provides us a formula for the parallel penalty:

$$B(n,p) = T(n,p) - T(p)/p. \tag{4.2}$$

As $B(n,p)$ depends on two parameters, we will investigate the behaviour of $B(n,p)$ depending on one of its parameters while the other one is fixed.

The distinction between sequential time $T(n)$ and 'parallel' time on a single PE, denoted $T(n,1)$, is essential for distinguishing between the *absolute speedup* $T(n)/T(n,p)$ and the *relative speedup* $T(n,1)/T(n,p)$, the latter usually being higher than the former because of the overhead of the parallel system on a single PE. Analogously we distinguish between an *absolute reference point* for our estimations, using sequential time $T(n)$ and a *relative reference point*, when $T(n,1)$ is used. In the latter case we consider the overhead for the transition from $T(n,1)$ to $T(n,p)$. Our approach bears—as everything on this topic—a certain grade of similarity to Amdahl's law [1967]. Exactly as Amdahl did, we assume a perfect parallelisation of the computation onto p PE, but consider also the unavoidable overhead. In a contrast to [Amdahl, 1967], we divide the parallel computation not into Amdahl's perfectly parallel and strictly sequential fractions, but into fractions of the actual parallel computation and of the parallel overhead. Our next contributions are threefold.

- **Estimation of run time.** Instead of estimating the execution time of a parallel program directly, i. e., using several values of $T(n,p)$ for the prediction, we separately find good approximations to $T(n)$ and $B(n,p)$. After the estimation we combine both values per (4.1) to obtain $T(n,p)$. We elaborate on this in Section 4.3.

- **Quality measure.** We discuss how usable $B(n, p)$ is as a measure of a parallelisation quality of a given program. We find some similarities to an approach, discussed briefly in Section 4.1. We consider this side of $B(n, p)$ in full detail in Section 4.6.

- The difference between the real, measured parallel time and a naive, zero-cost assumption of the perfect distribution of the total work is the parallel overhead. It is a measure of how bad the load balancing and other issues, like communication overhead, impact the ideal, 'Amdahlish' distribution of the total work across p PE. In the following, at few occasions we will deduce the optimal and non-optimal task balancing configurations theoretically. These will show a behaviour, strikingly corresponding to that of the parallel penalty.

4.3 Estimation

We estimate $T(n)$ and $B(n, p)$. Our aim is to find separate approximations for these two terms, as $T(n)$ and $B(n, p)$ are of a different nature. In order to do so, we measure several values of $T(n)$ and $B(n, p)$, then we use statistical techniques on the resulting data sets. We can compute $B(n, p)$ from $T(n, p)$ per (4.2). In other words, the parallel overhead per PE follows from the total parallel runtime for p PEs minus the sequential runtime divided by p. The latter values can be measured or estimated. Note however, that $B(n, p)$ depends on p.

Statistical methods. We use different methods to predict values of $T(n)$ and $B(n, p)$ for non-measured input sizes. We *could* have used straightforward polynomial interpolation, but for better results we sample more points and use one of the following methods. One approach is the cubic spline interpolation [Birkhoff and de Boor, 1965, Forsythe et al., 1977]. Splines curves are piecewise polynomials, seamlessly connected. Another method is local polynomial regression fitting, see [Chambers and Hastie, 1991, Chapter 8] and [Cleveland, 1979, Cleveland and Devlin, 1988]. We refer to these approaches using the R function names spline and loess [R Development Core Team, 2009]. Also we use linear model fitting with orthogonal polynomials constructed from the actual input [Chambers and Hastie, 1991, Chapter 4]. We denote this approach with lm(poly). A simple linear model fitting is just lm. Finally mean is not a real method, but the mean of the two best methods w. r. t. the relative error.

The spline method interpolates the measured data points exactly, while the other methods utilise regression fitting. The latter means that we do not attempt to fit all the input data points, but rather to capture the 'trend'. The method lm tries to fit a straight line, hence it is less appropriate for our purposes. Its generalisation lm(poly) uses orthogonal polynomials to weaken this drawback. The loess method is a modern statistical approach to polynomial regression. It is local, so distant data points have little influence on the shape of the fitted curve. The loess method is similar to spline with respect to this property.

Method choice. The decision on the best method *could* be done automatically. Given an $\varepsilon > 0$ and some existing runtime measurements, we predict a known(!) value with other ones using various methods.

- First, discard methods producing nonsense results, e. g., time estimation < 0.

- If some of the remaining methods produces a relative error $< \varepsilon$, we will pick one with the smallest relative error.

- If none of the methods produces a relative error $< \varepsilon$, but the relative error of the mean of the two nearest methods w. r. t. the actual value is $< \varepsilon$, we will pick the mean.

  If the real value is unknown ('real life' estimation) and we have to resort to the mean of two methods, two strategies exist on choosing the best two methods. Both require some 'training': we need to predict a few known values first. Then, for predicting an unknown value, we use the information from the training. Either we pick the mean of the two methods, which produced

best results in the training. Or we decide in the training phase on three best methods. In the 'real life' estimation we discard the more distant value and compute the mean of the two remaining values.

- If none of the methods yields a satisfying result, reconsider ε. Make further measurements. Otherwise fail.

Procedure of estimation. As mentioned before, both $T(n)$ and $B(n,p)$ are estimated separately. For each of them, a few values are measured. The values for $T(n)$ can be measured directly. The values for $B(n,p)$ are computed per (4.2). For $T(n)$ the aforementioned statistical methods are directly applied to the data. The result is an estimation $\hat{T}(n)$ for some not measured n. The values of $T(n)$ describe the total amount of work in the program, depending on n only.

The shape of $B(n,p)$ is the key to rating the parallel performance *quality*. As this penalty term depends on both the problem size n and the number p of PEs, it is important that it does not increase for growing p. Otherwise, the implementation does not scale well. We have two possibilities to estimate $B(n,p)$. We can look at it with the regard to n with fix p and vice versa. The values of $B(n,p)$ w.r.t. n represent the overhead of increasing the task size on the same p PE. The shape of $B(n,p)$ w.r.t. p shows the overhead for scalability of the program for increasing p. Analogously, we write $\hat{B}$ for an estimation of B. We always state, estimation w.r.t. which parameter of B we consider.

The estimates $\hat{T}(n)$ and $\hat{B}(n,p)$ w.r.t. n for some not measured n can be combined using (4.1) to an estimation $\hat{T}(n,p)$ of the parallel execution time for some *not measured* input size on *known* number of PEs p. We show an example of such estimation in Section 4.4. The estimate $\hat{B}(n,p)$ w.r.t. p and the measured value $T(n)$ for some known n and not measured p can be combined, again with (4.1), to an estimation of parallel run time for some *known* input size on *not measured* number of PEs. Section 4.5 shows an example of such estimation.

4.4 Example I: Hamming Numbers

In this example we perform an estimation of parallel run time w.r.t. the task size n. The Hamming numbers, also known as 5-smooth numbers, form an infinite set of numbers, which divide powers of 60. In other words these numbers have only 2, 3 and 5 as prime factors. The problem of computing such numbers is attributed to to Richard Hamming,*11.2.1915, †7.1.1998. It was popularised by Dijkstra (Edsger Wybe Dijkstra, *11.5.1930, †6.8.2002). Smooth numbers can be elegantly defined in a functional style, see, e.g., [Dijkstra, 1981]. The infinite list of 5-smooth numbers H is defined with three rules. Firstly, $1 \in H$. Secondly, for all elements $h \in H$ holds $2h$, $3h$, $5h$ are elements of H. Thirdly, each element of H occurs only once.

We see that in a lazy programming language it suffices to multiply existing entries in H by 2, 3 and 5, resulting in lists $2H$, $3H$ and $5H$, and then to merge these three lists to H. This is exactly what the Eden source code in Figure 4.1 does. It originates from Eden's test suite. Three `multp` processes carry out the multiplication. We introduce an extra merging process `sMerge`. Summarising, this program creates 4 process additionally to the master process. This means, that never more than 5 PE are required.

We want to detect suboptimal behaviour using our methods. This will happen in Section 4.6. We have executed the program in question five times and computed the average of execution times, exactly as our methodology Section 3.4 tells us to. The program was run on `sakania`. We used no memory tuning, e.g., we do not tweak initial heap size for our binary.

Prediction. We use the formula (4.1). We measure the sequential execution times (with absolute reference point) and parallel execution times on five PE (i.e., $T(n,5)$) of the aforementioned program computing Hamming numbers. We vary the task size n from 1 000 to 15 000 Hamming numbers, starting from the first one. We subsequently compute the values for $B(n,p)$ w.r.t. n. See Table 4.1 for the figures. To be able to reason about our prediction, we use the values with input sizes from 1 000 to 14 000 as input for our estimation methods. We want to estimate the values at 15 000.

```
hamming :: [Int]
hamming = 1:
  sm (sMerge # ((multp 2) # hamming,
                (multp 3) # hamming))
     ((multp 5) # hamming)

multp :: Int → Process [Int] [Int]
multp n = process (map (*n))

sMerge:: Process ([Int], [Int]) [Int]
sMerge = process (uncurry sm)

sm :: [Int] → [Int] → [Int]
sm [ ] ys = ys
sm xs [ ] = xs
sm (x:xs) (y:ys)
        | x < y = x : sm xs (y:ys)
        | x == y = x : sm xs ys
        | otherwise = y : sm (x:xs) ys
```

Figure 4.1: *Computing Hamming numbers.*

$n \times 1000$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**
$T(n)$	0.19	0.42	0.59	0.86	1.11	1.38	1.57	1.85	2.00	2.29	2.50	2.79	3.10	3.33	**3.64**
$T(n,5)$	0.13	0.28	0.41	0.59	0.73	0.90	1.09	1.29	1.39	1.58	1.93	2.05	2.35	2.56	**2.81**
$B(n,5)$	0.091	0.19	0.29	0.42	0.51	0.63	0.77	0.92	0.99	1.13	1.43	1.49	1.73	1.89	**2.08**

Table 4.1: *Execution times of Hamming numbers program.* ***Bold*** *values will be estimated.*

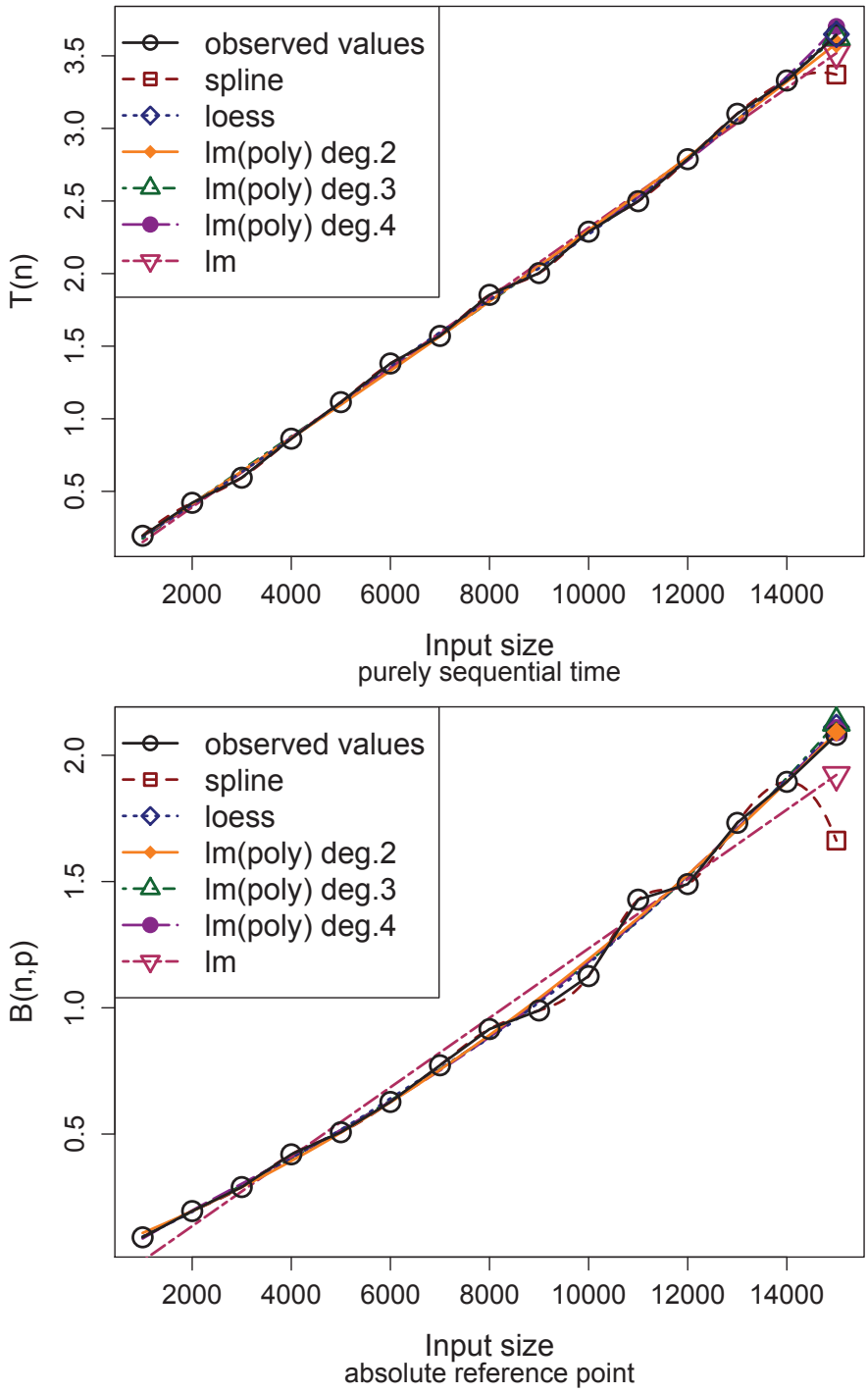
We use all six available estimation methods for both components of (4.1), an overview is in Figure 4.2. The best method for $\hat{T}(15000)$ is loess with 0.13% relative error. Also lm(poly) of degree 3 is very close, scoring −0.39% relative error. Linear model fitting (lm) is not quite successful, and spline is the worst method with −7.48% relative error. As for $\hat{B}(15000,5)$, spline misses completely with −20.11% relative error. The reason is the curvature between 12 000 and 14 000, which make spline believe, the values would decrease for increasing n. As for remaining methods, second worst is lm with −7.61% relative error. Two best methods are again lm(poly) of degree 2 and loess, with 0.57% and 1.48% relative error. The degrees 3 and 4 of lm(poly) result in 2.31% and 0.91% relative error. If we combine the both best estimations per (4.1), we obtain 2.82 seconds as an estimation for $\hat{T}(15000,5)$. This is correct up to 0.45% relative error.

4.5 Example II: Lattice-Boltzmann Method

Now we perform an estimation of parallel runtime w.r.t. the number of PE p. We use skeletons here as an abstract parallelisation paradigm description. The program to be analysed is not required to be implemented using skeletons, although all our following Eden programs are. To show that our method is applicable at a supercomputer scale, we discuss the following example.

We consider the lattice-Boltzmann method from fluid flow and mass transport simulation. The time measurement data in Table 4.2 originate from [Khirevich, 2010]. The measurements used here are conducted from a physical simulation on a supercomputer.

In this case the Jülich Blue Gene/P machine was used. It is built using a system-on-a-chip approach



Method		spline	loess	lm(poly) of degree 2	3	4	lm
Rel. err, %	T(15 000)	−7.479	0.1315	−1.612	−0.3884	1.567	−3.429
	B(15 000,5)	−20.11	1.475	0.5653	2.307	0.908	−7.608

Figure 4.2: *Predicting both components for Hamming numbers. Top: sequential time, middle: parallel overhead, bottom: relative errors.*

LBM							
PE, p	32 768	65 536	98 304	131 072	196 608	*262 144*	294 912
$T(n,p)$	16.285	9.99	6.82	6.80	5.284	*5.273*	3.675

Table 4.2: *LBM on a supercomputer, courtesy of [Khirevich, 2010]. The task size n is fixed. The boxed values will be estimated.*

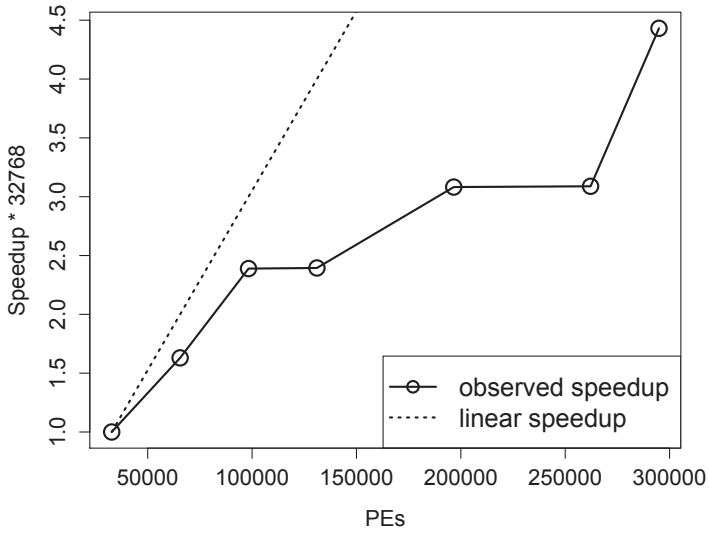


Figure 4.3: *The speedup plot for LBM.*

with quadcore PowerPC chips with 2 GB of RAM as the base. Each core is a 32 bit processor, running at 850 MHz. So, a single node is a traditional multicore processor. The nodes are assembled into racks with 1024 nodes in each. Up to 294 912 cores were used for the experiment. The peak performance of this supercomputer reaches 1 PetaFLOP/s. Jülich Blue Gene/P was the third fastest supercomputer in the world, when it was ranked in its current configuration for the first time in June 2009. As of Summer 2011, this supercomputer is ranked 12$^{\text{th}}$ [Meuer et al., 2011].

Overview. The aim of [Khirevich et al., 2009a,b, Khirevich and Daneyko, 2010] was to simulate the transport processes in porous media, e.g., in the chromatographic separations. Pumped into a long thin pipe, filled with some matter, what paths does an injected solution follow? The matter is modelled with spheres. The pumped solution is simulated in two steps: first fluid flow is simulated, subsequently the actual movement of the injected solution is studied. The simulation consists of several phases:

1. Random close-sphere packing and its spatial discretisation. We do not consider this phase.

2. Simulation of the fluid flow with lattice-Boltzmann method (LBM). This is the phase we focus on.

3. Simulation of the advective-diffusive mass transport. It is performed with the random-walk particle tracking method. We also do not discuss this phase here.

The latter two phases clearly dominate the computational complexity [Khirevich and Daneyko, 2010]. We chose to focus on the LBM phase. Due to the dimensions of the chosen lattice—$632 \times 632 \times 294\,912$—a one-dimensional decomposition is possible. Basically, the pipe is cut in length and each 'slice' is assigned to a PE. The amount of spheres in each slice varies, the maximal difference is 27%. Hence, we have a data parallel implementation. We identify slices with tasks in a `farm`.

PEs, p	32768	65536	98304	131072	196608	*262144*
$B(n,p)$	0	1.85	1.39	2.73	2.57	*3.24*

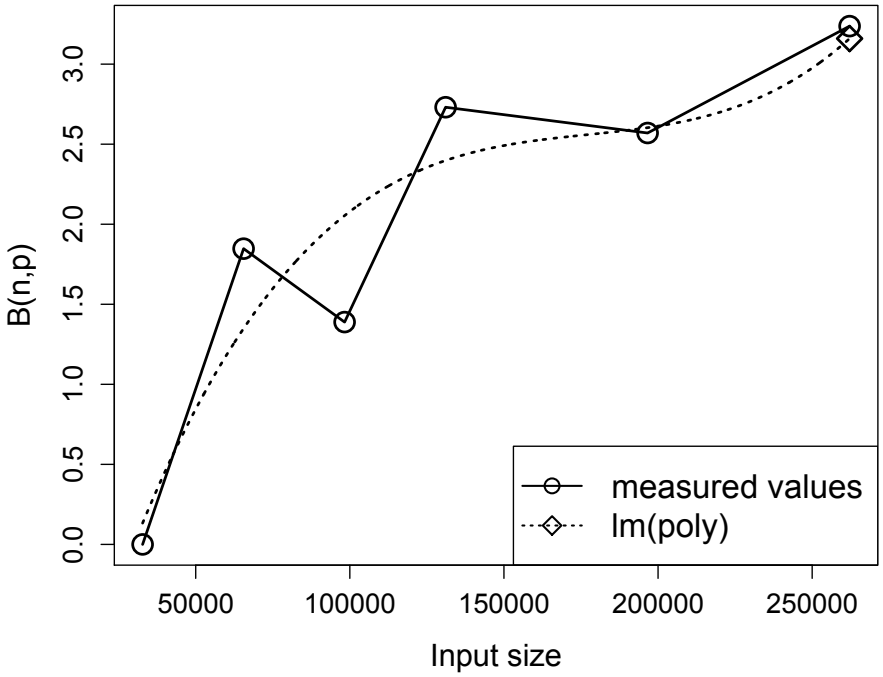


Figure 4.4: *Computing $B(n,p)$ for LBM. Top: computed values for B. The boxed value will be estimated on the bottom.*

The LBM phase generates the data, used later in the RWPT phase. Basically, the flow of the fluid around the sphere packing is computed, it is done with sophisticated variants of cellular automata, *cf.* [Chen and Doolen, 1998]. We show the speedup curve LBM in Figure 4.3.

The time measurement data for LBM on the Jülich Blue Gene/P supercomputer is presented in Table 4.2. These have been taken from [Khirevich, 2010]. The time required for 10 iterations is given. We assume $T(n)$ in our computations to be $T(n, 32\,786) \cdot 32\,768$. In other words: a perfect speedup for up to 32 768 PEs is assumed. This is rather a convenience convention than an assumption: we could as well 'downscale' the PE numbers by dividing them by 32 768. It is impossible to obtain the real sequential time, as the data to be processed does not fit into the memory of a single machine. This means, we use relative reference point.

Estimation. The task size is fixed, but we can estimate the scalability w. r. t. the PE number p. Note that we have neither the source code nor a binary version of the program we investigate. All we know is the scheme used for parallelisation and the time measurements in Table 4.2.

We present the values for $B(n,p)$ w. r. t. p in the top part of Figure 4.4. The lm(poly) method with polynomials of degree 3 produces the value 3.16 seconds for $B(n, 262\,144)$. We present it graphically in the bottom part of Figure 4.4. Using it and approximating $T(n)$ with $16.285 \cdot 32\,768$, we obtain the estimation for the execution time $\hat{T}(n, 262\,144) = 5.196$ seconds. This estimation is exact up to the relative error −1.47%. Notably, a direct runtime estimation—an attempt to predict the parallel time directly from the number of PEs, using the same data, but without using equation (4.1)—fails. The all-best relative error for direct estimation is −31%. This shows that our approach is applicable to large-scale production applications.

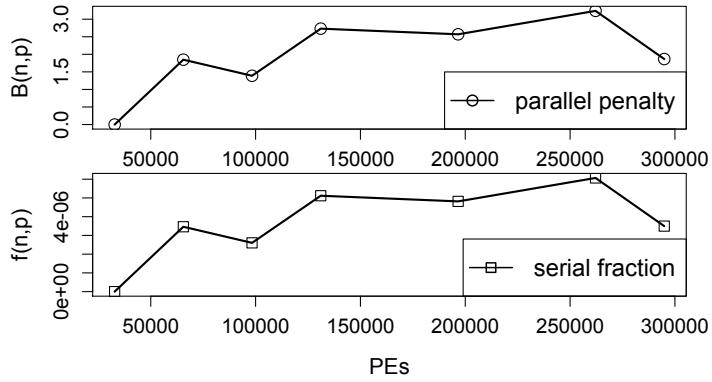


Figure 4.5: *Comparing parallel penalty (top), and serial fraction (bottom) for LBM.*

PEs, p	32 768	65 536	98 304	131 072	196 608	262 144	294 912
PEs, normed	1	2	3	4	6	8	9
full rounds	9	4	3	2	1	1	1
remaining part	0	1	0	1	3	1	0
total rounds	9	5	3	3	2	2	1
idle in last round	0	1	0	3	3	7	0
slack-off, %	0	50	0	75	50	87.5	0

Table 4.3: *Analysing the task distribution in LBM.*

4.6 Comparing Serial Fraction and Parallel Penalty

In their work *Measuring Parallel Processor Performance* Karp and Flatt [1990] introduced the notion of *serial fraction*. Given the parallel time $T(n, p)$ on p PEs and the sequential time $T(n)$, the absolute speedup is $T(n)/T(n, p)$. The serial fraction is

$$f(n, p) = \frac{T(n, p)/T(n) - 1/p}{1 - 1/p}.$$

Contrary to Karp and Flatt [1990] we used absolute reference point in the above formula. The serial fraction should be constant: if it increases, we have a parallelisation resulting in poor speedups. If the serial fraction decreases, this shows problems with the sequential implementation.

LBM. We compare the complete plots for parallel penalty w. r. t. p and for serial fraction. As our first case study we use the LBM supercomputer example from above. Figure 4.5 visualises both functions. We see in both cases a decrease at 65 536, 196 608 and 294 912 PE. Both functions have a maximum at 262 144 PE. As the relative reference point is used, both functions are zero at 32 768 PE.

To be able to reason on the meaning of the said plots, let us consider the following. We know that in total 294 912 tasks are issued. We show the theoretic task distribution over PE in Table 4.3. We assume here that all tasks need equal time to complete. The first row ('PEs') shows the total number of processing elements. The second row shows the number of PEs, divided by 32 768. The third and fourth rows show appropriately how many times each and every PE becomes a task ('full rounds') and how many of PEs have further tasks remaining for the last, incomplete round ('remaining part'). These two values originate from the division with remainder of the number of tasks by the number of PEs. The remaining part is normed by 32 768. The total amount of rounds is in the fifth row. It includes now the incomplete last round. The sixth row is the number of free PEs in the incomplete round, normed by 32 768. The last, 'slack-off' row displays the relation of the sixth row to the second row. It shows the percentage of idle PEs in the last task distribution round. A large figure in this row means poor task distribution in the last, incomplete round. To give an example, we plot in Figure 4.6 the task

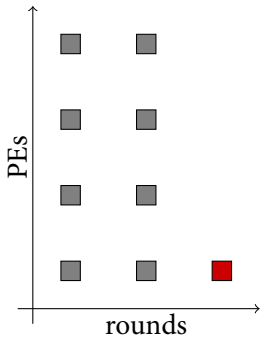


Figure 4.6: An example for task distribution. Imagine 4 PEs, running 9 tasks in 3 full rounds. The last round will have 75% slack-off.

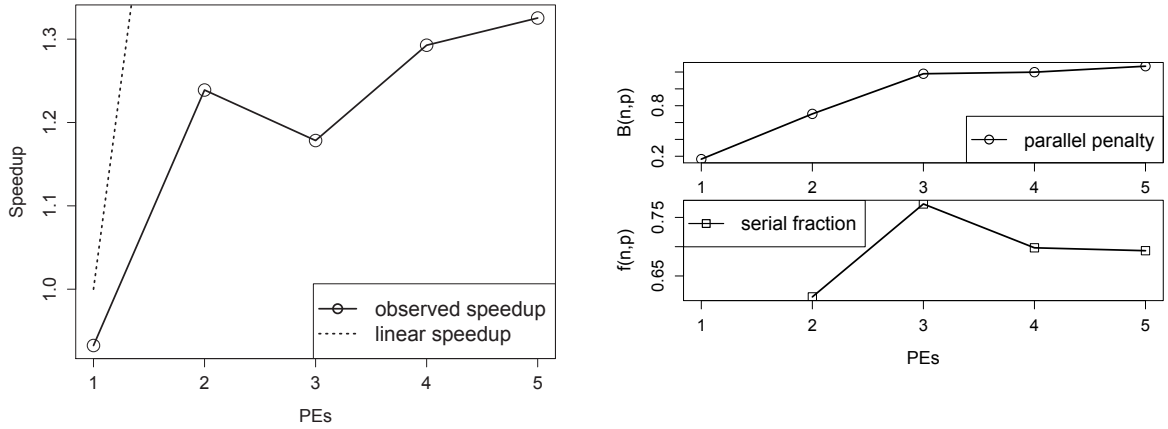


Figure 4.7: Computing the first ten thousand Hamming numbers: the speedup (left), parallel penalty (top right) and serial fraction (bottom right).

distribution for the normed 131 072 case: 4 PEs, two full rounds, one remaining part. This means that slack-off is 75% in this case.

The theoretical considerations above correspond to Figure 4.5. The two peaks at 131 072 and 262 144 fit the high percentages of slack-off. The two minima at 98 304 and 294 912 correspond to zero slack-off. The practical values in Figure 4.5 at same two points are not zero because of some task imbalance and communication overhead, which we ignored in our considerations in Table 4.3.

Note that despite the different scala both plots in Figure 4.5 look strikingly similar. However it is not always like this, as the next example shows. The zero slack-off at 294 912 is also the reason, why we did not consider this number of PE in our estimation. It has a different nature as 262 144 PE version.

Hamming numbers. Coming back to the Hamming numbers example of Section 4.4, we want to compute first ten thousand of them. We use the same program for computing the first n Hamming numbers as before. It was run on `sakania`, the usual measurement method applies. Absolute reference point is used. We see in Figure 4.7, left, that the speedup of the program is quite poor. Now for the actual goal of the example: we consider parallel penalty and serial fraction of the naive program computing Hamming numbers. The parallel penalty is presented in Figure 4.7, top right. The serial fraction is depicted in Figure 4.7, bottom right. We see that both plots are increasing in their left parts, indicating non-perfect parallelisation for smaller number of PEs p. However, the behaviour of the parallel penalty and of the serial fraction is different between three and five PE. The parallel penalty function is increasing from 4 to 5 PE and almost constant from 3 to 4 PE. Contrary to that, the serial fraction is almost constant between 4 and 5 PE, but is *decreasing* between 3 and 4 PE.

We want to stress that both plots of the parallel penalty and of serial fraction, although bearing similar *meaning*, are completely different in their *looks*. Thus we cannot think about parallel penalty as

just of serial fraction, multiplied with some other factor. The parallel penalty describes nicely the task balancing issues, like ones described in Table 4.3. We will see further examples in the next chapters.

4.7 Related Work on Performance Estimation

Related publications on performance forecasting include the book chapter on skeletons in Eden [Loogen et al., 2003] and the formal cost model of NESL [Blelloch, 1996]. However, our approach is different. We derive the time and work from time *measurements* for the runs on different numbers of PEs, while the skeleton analysis in [Loogen et al., 2003] is based on latency and message-passing costs. The NESL complexity model [Blelloch, 1996] takes a 'bottom-up' approach, trying to assign cost to single semantic operations. We look in a 'top-down' manner on the total runtime and divide it into very coarse blocks: the work and the parallel penalty.

An example of the sequential estimation of runtime is [Saavedra and Smith, 1996]. Ipek et al. [2005] use neuronal networks to (directly) predict execution times of a multigrid solver. The paper by Akioka and Muraoka [2004] focuses on the network load and uses Markov model-based meta-predictor. In a contrast, we used our decomposition of the parallel execution time in equation (4.1) and statistical methods. Kapadia et al. [1999] found locally weighted polynomial regression definitely superior than other instance-based learning methods (e.g., nearest neighbour) for the estimation of parallel execution time of real programs. However, Kapadia et al. [1999] did not use any decomposition of the execution time. Further, the number of program executions for the machine learning approaches is quite high: 8100 in [Kapadia et al., 1999], 10000 in [Ipek et al., 2005]. Our approach provided good results on orders of magnitude less data points: our Eden case study featured 14 data points, each of them constituted of an average of 5 program runs. The C+MPI case study used even less data points and still provided successful estimations. Our further estimations, presented in the next chapters also feature similarly many data points. Thus, our method can more easily be used in an adaptive runtime environment, which is future work.

As already mentioned in the previous chapters, skeletal approach to parallelism was introduced in [Cole, 1989]. Approaches on skeleton-based performance evaluation include [Cole and Hayashi, 2002, Benoit et al., 2004]. Cole and Hayashi [2002] consider a BSP-like cost model, assigning costs to basic elements of a parallel program. A similar approach to ours, but more fine-grain and still focusing on BSP is [Zavanella, 2001]. Beside BSP [Valiant, 1990], models like LogP [Culler et al., 1993] and message passing models exist. See [Roda et al., 1999] for a runtime prediction approach for the latter. A rather theoretical well-investigated performance model is the PRAM model [Fortune and Wyllie, 1978] and its variants. The classic PRAM model does not consider communication costs.

We do not relate the approach of this chapter to computer algebra algorithms, because it is applicable to an arbitrary parallel program.

4.8 Conclusions

We have presented a novel approach for parallel run time estimations. With it we can predict the execution times of parallel programs in an elegant and concise manner. Our method is different from previously known approaches. It does not depend on source code analysis or on special semantic rules. Instead, our method is empowered by computational statistics. We separate estimations for parallel computation and parallel overhead, hence different forecasting models can be used for each. This makes sense, given the different nature of these processes, and enables better estimations. We observed correlations of the parallel penalty with the theoretical values for task balancing.

We will use our technique to predict and evaluate execution times of the example instantiations of algorithmic skeleton abstractions, developed in this thesis. Thus we can ensure wider applicability of our results. In the following we will focus on evaluation of Eden programs. Still, on par with an Eden example, we included in this chapter an example computed on a peta-scale supercomputer, using C+MPI. Notably, in all presented cases we managed with relatively few measurement points.

Chapter 5

PRIMALITY TESTING — REPEATED COMPUTATION

Numero deus impari gaudet.*

Virgil, *Eclogues VIII*

Skeletal implementation of parallel repeated computation and the implementation of probabilistic primality tests is the focus of this chapter. We will see that these two goals are connected. An imperative way of thinking about the repeated computation is that of a particular parallel loop. We devise a classification of new and existing skeletons in Section 5.1. We develop there a more special approach, which fits exactly to our needs. The desired parallel behaviour has a well-known functional counterpart: a `map-reduce` with a possibility of premature termination. The premature abort part is new. We find some features of the Eden language useful in this context, as the premature termination is granted for free in our setting. We discuss it in Section 5.2.

Considering the second goal of this chapter, Section 5.3 discusses some common issues in the implementation of the two primality tests. In Sections 5.4 and 5.5 we implement two sophisticated algorithms for the primality testing using our approach. Combined, this chapter provides

- A skeleton for parallel repeated computation.
- A toolchain for parallel probabilistic primality testing.

We combine the two tests mentioned above into a single framework. At the end of each implementation section we provide some measurements and express the performance of this chapter's approach quantitatively. Section 5.6 concludes.

5.1 Repeated Computation Skeletons

The goal of this section is to survey parallel repeated computation and its features. Further, we will show a specialised version of our new approach.

Overview. The repeated computation is a quite typical task in computational mathematics. Given some data $x_1, x_2, \ldots$, we apply some function f to it, resulting in $f(x_1), f(x_2), \ldots$, until a predicate p is not satisfied for some $f(x_n)$. As we will see below, this behaviour corresponds to a particular **while**-loop.

The parallel case is more complicated. We consider the speculative parallelism, where multiple tasks are processed simultaneously. In some cases, after a few tasks are processed, more tasks need be created. Different tasks may need different time to complete, we need a way to balance them. If a computation may be aborted, this also needs to be implemented. Combined, we examine the combinations of three distinct features of the parallel repeated computation skeletons: task creation, task balancing and premature abort of the computation. An overview is available in Figure 5.1 on the following page.

We consider task creation first. One possible approach is the dynamic *internal task creation*. In this case we have the possibility to create new tasks while the computation progresses from within the skeleton. Essential for the internal task creation is the possibility to use intermediate results *inside* the skeleton and to *transform* the task pool of the skeleton. In absence of these, the ability to create tasks from the final results is available with a wrapper around standard map skeletons in a lazy functional language. We call it dynamic *external task creation*. This approach is similar to the non-monadic I/O

*Uneven numbers are the god's delight.

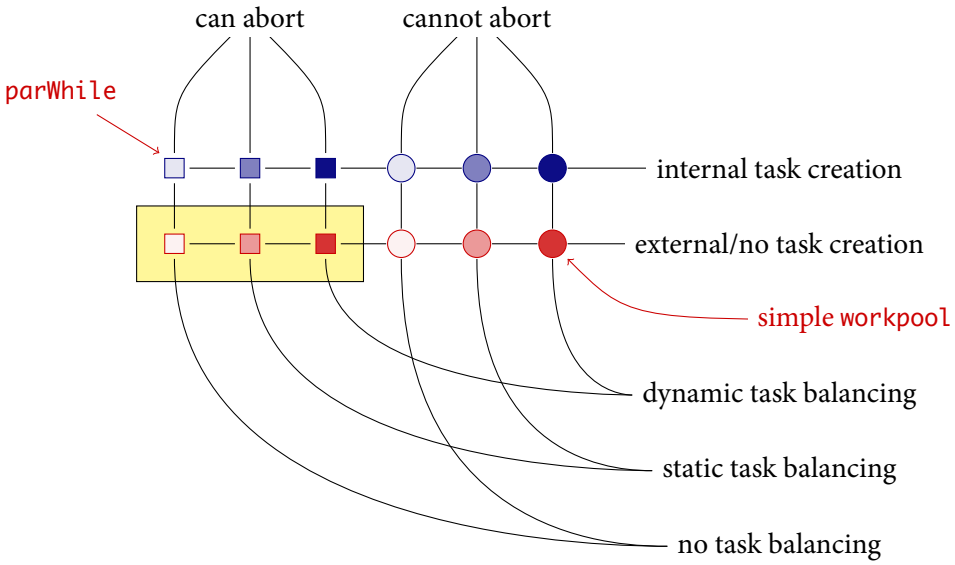


Figure 5.1: *Twelve possibilities for a combination of the three repeated computation skeleton features. The marked specimen highlight our area of interest: all three types of task balancing with no internal task creation and present premature abort feature.*

in Haskell. We will elaborate on it later. If all tasks are known beforehand and no new tasks emerge, *no task creation* takes place.

Task balancing is the way, how the tasks are balanced in the progress of computation in order to maintain similar load on all PEs. The *dynamic task balancing* is the prime feature of workpools (see also Chapter 3). It is named similar to *load* balancing, but only in case of dynamic task balancing we indeed balance the actual load. Here the decisions are made at runtime. In a contrast, the `farm` skeleton does task balancing, alas not while the computation progresses, but beforehand. We call this *static task balancing*. Finally, a simple parallel map does not task balancing, but simultaneously creates a process for each task.

The property of the premature abort will be discussed in detail later. Essentially, we want to be able to cancel a computation in progress, basing on already available results.

Combined, twelve possibilities emerge. The no task creating skeletons can be easily upgraded to dynamic external task creation with a wrapper. We perform this extension for a `farm` in Section 6.6. We consider only these six with premature abort property. The literature mainly studies the three internal task creation versions, as these are more generic—but also more complicated. We define and use herein a generic scheme for three *no* task creation versions. To do so, we need to analyse our starting point, the dynamic internal task creating, not task balancing skeleton `parWhile` with premature abort (*viz.* Figure 3.19 on page 34). Basing on the pseudocode specification of this approach we will deduce a simpler scheme. However, before we can do so, we need to define the pseudocode, we will use below. We will use this language solely in the current section to discuss the different classes of repeated computation in the imperative and functional context. This explains the limitations of our pseudocode language.

Pseudocode. We define a small, imperative parallel language for our ramblings. We choose a C-similar syntax [ISO/IEC 9899:1999] with a few functional features. With such a choice it is easier to see the exact moment of task creation. In the following we describe the details.

- The curly brackets combine statements to blocks, as in C. Also similarly, a semicolon (;) terminates the statement.

- Variable declaration is as in C.

```
int i = 0;
repeat {
  // update i
  i++;

  // compute the result
  results[i] = f(tasks[i]);
}
until(not p(results[i]));

return results;
```

Figure 5.2: *A typical sequential repeated computation in pseudocode.*

- A commentary is introduced with // and ends with line termination.
- **repeat** body **until**(condition); is a **while**-loop, where the body block is executed at least once, but as often as needed, until condition is true. It is similar to the construct of the same name in Pascal [Wirth, 1971]. It is also similar to C's **do**–**while**, but the termination condition is inverse.
- **for**(init; condition; step) body is a **for**-loop, as in C.
- Collections can be defined, similar to arrays in C and lists in Haskell. We access a particular, i^{th} element in a collection with collection[i]. The primitive **size** returns the size of the collection. We need it to organise a **for**-loop.
- **map**(function, collection) is a primitive, which applies the function to each element of the collection. It is similar to `map` in Haskell, but we chose to make it a primitive here for the simplicity.
- **par** is an annotation, signalling that the following loop or primitive is executed in parallel. We need only a parallel **for** and a parallel **map**. The latter could be implemented as a special **for** loop, but we chose to make it a language construct for the sake of simplicity.
- **not** negates a boolean expression.
- **all** is a reduction operation on collections of booleans, similar to Haskell's `all`. It is true, if all elements in the collection are true, and false otherwise.
- **takeWhile**(condition, collection) outputs collection elements as long as the condition is true. It is similar to the function of the same name from the Haskell standard library.
- **subset**(collection, range) returns a collection, being a subset of the input collection with indices in a given range.
- **return** expression designates the final result of the code snippet, namely expression. It should occur exactly once.
- Our pseudocode language is strict.

Now, after defining some foundations of the pseudocode language, let us discuss the details of the specification. Recall: given a collection tasks, we want to apply a function f to each element of it, yielding the collection results, but only as long as the predicate p holds for each of the results. This is the setting of Figure 5.2. But we transform this loop, changing its shape.

To specify our parallel repeated computation loop more exactly, we need some further functions. The function prepare creates a collection of tasks from an initial input, the function combine builds the collection of the results to a monolithic final result. Now we can partition the collection into several

```
repeat {
  tasks = prepare(input, results);

  // define subsets of tasks, the functions
  // startRange() and isInCurrentRange()
  // control the choice of i to cover all tasks
  par for(int i=startRange();
          isInCurrentRange(i); i++) {
    result[i] = f(task[i]);
  }
}
// the function getPrevRange() returns
// the just processed range of indices
until(not(all(map(p,
        subset(results, getPrevRange()) ))));

return combine(results);
```

Figure 5.3: Parallel repeated computation with dynamic internal task creation.

```
tasks = prepare(input); // notice the change

repeat {
  // as in previous figure, functions
  // startRange() and isInCurrentRange()
  // control the choice of i to cover all tasks
  par for(int i=startRange();
          isInCurrentRange(i); i++) {
    result[i] = f(task[i]);
  }
}
// the function getPrevRange() returns
// the just processed range of indices
until(not(all(map(p,
        subset(results, getPrevRange()) ))));

return combine(results);
```

Figure 5.4: Parallel repeated computation with no task creation, imperative view.

ranges. All the elements in a single range will be processed independently and in parallel. We call this an inner loop. The outer loop handles the range preparation and the termination control. We can no longer check, whether a predicate is fulfilled for each element of the collection. Instead, we check the new results in the currently processed range. If any value is false, we need to terminate the whole computation. This corresponds to Figure 5.3. In our terminology this is a parallel repeated computation with internal task creation and premature termination. The **par for** loop does no task balancing. Summarising, this is the semantics of the `parWhile` skeleton.

Now we observe, that we create a bunch of tasks before each round of parallel computation, i.e., before each **par for** loop. Notice, that in this case new tasks can depend on already processed ones. However, it is possible to designate a class of applications, where *all* tasks can be created beforehand. For such applications we can modify the above schema in a natural way. All tasks are created in advance, before both the outer loop—the termination control—and the inner loop—the speculative parallel computation—start. Finally, the results are combined to a final output. Obviously, new tasks do not depend on old tasks' results anymore. This is the content of Figure 5.4. Note that now we have *no task creation*—all tasks are available before the parallel computation starts. Additionally, the partitioning of data in the outer **repeat**–**until** loop and the inner **par for** loop can constitute a fitting form of task balancing, e.g., a `farm` in the Eden implementation.

Lazy setting. We can write the code in Figure 5.4 differently. Imagine another pseudocode language with same syntax, but lazy semantics. In this case, similar to the GHC implementation of Haskell, nothing is evaluated until the result is required. We do not demand this property in the other examples! In a contrast, the speculative loops in Figures 5.3 and 5.4 do parallel iterations with **par for** even if some of them are not required, as may be found out later in **until** clause.

In Figure 5.5 no (parallel) computation happens, if it is not demanded. Hence, the premature abort is merely a missing demand on the remaining repeated computations. A function **takeWhile**, doing exactly this, can be seen as an outer loop. The inner loop corresponds to a parallel map, which we express with **par map** in Figure 5.5. Note that the code in the latter figure works only in a lazy language, contrarily to Figure 5.4. We can use some `reduce` function instead of **takeWhile** and combine, as will be detailed in Section 5.2. As we need to drop the demand on the remaining elements of the results collection, we need to use some particular `reduce` functions, doing exactly this. We show below, which exactly `reduce`s qualify.

```
// lazy
tasks = prepare(input);

// in a lazy setting map can be aborted
results = takeWhile(p, par map(f, tasks));

return combine(results);
```

Figure 5.5: Parallel repeated computation with no task creation, lazy functional view.

```
farm+reduce :: Trans a ⇒ (a → Bool) → [a] → Bool
farm+reduce f xs = and $ farm f xs
```

Figure 5.6: A simple repeated computation skeleton without task creation.

```
(&&) :: Bool → Bool → Bool -- see Haskell Prelude

and :: [Bool] → Bool
and = foldr (&&) True
```

Figure 5.7: The and function from standard Haskell Prelude.

Implementation. We have just seen, that in case of using a functional lazy programming language and opting for no task creation, we can sketch a much simpler version of the `parWhile` skeleton. Our Eden implementation, called here `farm+reduce`, is shown in Figure 5.6. Note that strict Haskell syntax disallows such a function name, so we use `farmPlusReduce` in the actual implementation. This Eden skeleton is quite similar to the pseudocode in Figure 5.5. We emphasise, that this version *does* have task management and can abort the computation early, but it *disallows* internal task creation. The version shown is *very* simple and assumes worker functions of type `a → Bool`. It is straightforward to extend it to a more generic `a → Maybe b`. A worker function returning a `False` causes a collapse of the skeleton. This happens because of some properties of the `and` function [Peyton-Jones, 2003]. Namely, `and [True, False, ⊥] = False` holds: the `and` function stops traversing the list, once a `False` occurs. See also Figure 5.7. A call to the function `farm` of type `Map a b` does the actual parallel evaluation. A further generalisation happens in the next section. This generic scheme is our original research.

5.2 Map+Reduce

In this section we generalise our current approach. We discuss the concepts, related to `map`, `reduce` and their combination next. Afterwards we consider the `farm+reduce` skeleton and generalise it to the `map+reduce` scheme. Then we discuss the laziness of `reduce` and the implementation of the `map-reduce` scheme with `map+reduce`.

5.2.1 Related Work

The two popular higher-order functions for parallel processing are `map` and `reduce`. The function `map` with type `(a → b) → [a] → [b]` applies a worker function to each element of a list. We discussed parallel maps in Chapter 3. The function `reduce` with type, e. g., `(b → b → b) → [b] → b`, reduces the list to a single element with the given operation[1]. Note that `reduce` is a different name for a `fold`.

[1]The type shown here corresponds to the functions `foldl1` and `foldr1` from the Haskell's standard library. In a contrast to `foldl` and `foldr` from the same library, they require the input list to be not empty, thus eliminating the need in the

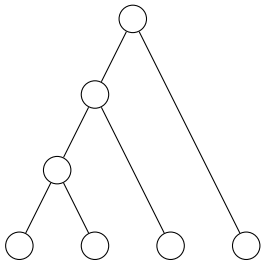


Figure 5.8: *A standard* `reduce`*.*

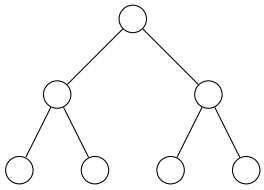


Figure 5.9: *A tree-shaped* `reduce`*.*

See, e. g., [Hutton, 1999] for an introduction. The `reduce` function has a few very interesting theoretic properties. For example, for a classic parallel implementation, the `reduce` parameter function should be commutative, i. e., the result should not depend on the order of single reductions. This enables us not only to compute `reduce` 'from the left' and 'from the right', as Haskell's `foldl` and `foldr` do. We discuss next i) a tree-shaped `reduce`, ii) local `reduce`s, followed by a global `reduce`.

The classical `reduce` builds a degraded tree, *cf.* Figure 5.8, be it completely left or right. But in the case the `reduce` is commutative, we can obtain a better shape of this tree. Figure 5.9 depicts a tree-shaped `reduce`, we might imagine the tree of a larger depth, however only two elements at once are reduced. In these figures the data flow is bottom-up.

Because of the same commutativity property of the `reduce` it is possible to 'stage' `reduce` calls. Imagine some kind of a parallel `map`, where each worker produces a list as the result. We can apply `reduce` locally at the workers, in parallel. Then we need to communicate just single values, not lists. The resulting list of the pre-reduced values, one value per worker, can be reduced again in the master process, resulting in the desired single value. The same Figure 5.9 can be interpreted as a distributed `reduce`, provided we have only four initial elements and reduce in two stages. Here the two layers of circles represent inputs for the global and for the local `reduce`s. We discuss it next.

A popular pattern in parallel computing is `map-reduce`. It is based on the parallel processing of data with a parallel (and probably: load-balancing) `map` with a following reduction. The latter happens in a distributed manner. Multiple results on each PE are reduced locally, afterwards a global `reduce` takes place. In other words, the data locality plays an important role in an efficient implementation of `map-reduce`. In the following, when we refer to `map-reduce`, we mean such an implementation. Eden implementations of `map-reduce` were presented in the works of Fernando Rubio, e. g., [Loogen et al., 2003]. It was also discussed in [Berthold et al., 2009d]. Figure 5.10 presents a graphical visualisation of the `map-reduce` pattern. The data flow is from left to right, boxes represent processing elements, nodes represent data, edges represent function applications. Keeping the data distributed is crucial for the efficient implementation of `map-reduce`.

The `map-reduce` pattern became much publicity because of Google MapReduce [Dean and Ghemawat, 2004, 2008, 2010]. However, Google's approach differs from the functional `map-reduce` pattern, see [Lämmel, 2008]. An Eden implementation of Google MapReduce is in [Berthold et al., 2009d].

Another approach to parallel functional repeated computation is to use an advanced `workpool` with dynamic internal task creation. As the name says, dynamic load balancing is already granted in this setting. Internal task creation is also well studied in the context of a `workpool`. Two possibilities

separately supplied neutral element. Note that `foldl1` and `foldr1` demand a commutative operation as their first parameter.

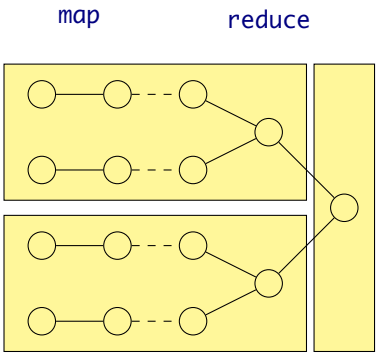


Figure 5.10: *The* `map-reduce` *scheme. The nodes symbolise data, the lines symbolise function applications, the boxes stand for processes. Here we have two 'mapper' processes, which also perform local* `reduce` *steps, and one global reducer. The dashed lines symbolise the data locality: there is no process boundary between* `map` *and local* `reduce` *phases.*

		Task creation	
		internal	external/no
Task balancing	dynamic	Priebe [2006] Dieterle [2007]	`workpool+reduce`
		Brown and Hammond [2010]	
	static	Brown and Hammond [2010]	`farm+reduce`
	no	`parWhile`	`parMap+reduce`
Task balancing	abstracted	not known to us	`map+reduce`

Table 5.1: *Classification of parallel repeated computation skeletons.*

for premature termination exit. One is to extend the skeleton with a state management. However, if a global task pool transformation is available, a variant without a state exists. Such transformations are often used as a generalisation for the placement of new tasks. It suffices to introduce a new, special task. The worker function, willing to terminate the whole computation, would create such a new task. The task pool transformation function would just empty the task pool upon seeing it. We stress however that the examples in this chapter will *not* need any dynamic task creation. So, we do not elaborate on this issue.

Note that the `parWhile` skeleton (Figure 3.19 on page 34) has no task management, but implements dynamic internal task creation and premature termination. We can implement the required subset of its functionality in a completely different and much simpler way: using a `farm` or a `workpool` skeleton and a fitting `reduce`.

We build up a classification of existing repeated computation skeletons in Eden in Table 5.1. We differentiate by the method of task creation and by the type of task balancing. The skeletons, shown here, are referenced by their names. The skeletons, known from the literature, are referenced by the appropriate citation. We see, the internal task creation skeletons with dynamic task balancing are quite well studied, as they form an extension to the well-known workpool scheme. The skeletons `parMap+reduce` and `workpool+reduce` are analogues of the `farm+reduce` scheme from Figure 5.6 with an obvious change. We will generalise these skeletons under a unified approach called `map+reduce`.

Popular skeleton libraries include various parallel `map`-like skeletons as well as distributed `reduce` implementations, see, e. g., [Bacci et al., 1999b, Matsuzaki et al., 2006, Ciechanowicz and Kuchen, 2010]. An approach to skeleton composition is [Alt and Gorlatch, 2003, Dieterle et al., 2010b]. Google's MapReduce is available for various platforms [Hadoop, 2011, Zaharia et al., 2008], including GPUs [He et al., 2008] and Cell BE processors [de Kruijf and Sankaralingam, 2009]. The Eden implementation is discussed in [Berthold et al., 2009d].

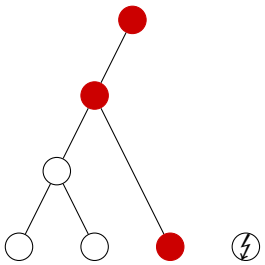


Figure 5.11: *A standard* `reduce` *with shortcutting. The data flow is bottom-up. The red node contributes all needed information, which is propagated to the root of the tree. The computation at ↯ has no demand.*

The *poison* concept of Hoare's CSP [Welch, 1989, Brown and Welch, 2003, Brown, 2008] is related to the our premature abort notion.

5.2.2 Discussion

Given the higher-order function `farm+reduce` from Figure 5.6, we aim to generalise it. This specific implementation of a parallel **for**-loop turns out to be nothing else than a naive `map` and `reduce` combination. We will see in Sections 5.4 and 5.5 that using it is not a limitation for our target problems. Still we can generalise more, as we will see below. In Figure 5.6 we perform the `reduce` sequentially, as it is a simple `and`. We stress that the actual work is to be performed in the `map` phase. However, we have further requirements for the `reduce` function. They are fulfilled by `and`, but not by an arbitrary `reduce`. (Figure 5.7 shows the definition of `and` in standard Haskell library.) We call our `map` and `reduce` combination with a special `reduce` a `map+reduce` skeleton scheme.

The key feature of our Eden implementation should also be granted in a generic `map+reduce` case: if our `reduce` function returns the result before evaluating the whole input list, e.g., `and [True, False, ⊥] = False`, then the parallel `map` instance, still computing the further list entries, should be terminated. In the following we seek for a formalisation of this issue and develop a generic skeleton scheme, using the gained knowledge.

The required property of the `reduce` is *shortcutting*. If some particular input element contributes all the information required for the final result, then no further input elements need to be evaluated. Figure 5.11 displays such a special input element and the information propagation in red. Note that it is impossible to encode shortcutting in Haskell's type system. It can be compared to the inability to encode laziness in the type system of Haskell.

We stress again that `map+reduce`, which we describe here, is not quite the `map-reduce` scheme. The latter assumes distributed `reduce`, as we already know. Also, the data flow in `map-reduce` leaves the data distributed between `map` and `reduce` phases. This is one of the main reasons for the success of the `map-reduce` scheme. However, in our case a special `reduce` is needed. We use sequential implementations of the `reduce` here and do not consider a distributed `reduce` with shortcutting. As we use `reduce` to control the termination, it is acceptable to use a sequential implementation. Still, we will see below that is it possible to implement `map-reduce` using our approach and an additional concept.

After seeing the `farm+reduce` example in Figure 5.6 on page 55, we aim to implement a more generic `map+reduce` skeleton scheme. We basically need to abstract the `reduce` function. It is done by adding a further parameter. The implementation is surprisingly simple, see Figure 5.12, but there is no way to ensure the shortcutting property in the Haskell implementation. The skeleton `farm+reduce` from Figure 5.6 can be implemented using this generic skeleton scheme, as seen in Figure 5.13. Note, the `map+reduce` scheme is building on further skeletons.

A different view on the `map+reduce` functionality is that of the streams, *viz.* page 23. To recapitulate, the intuition of a stream is that of a (potentially infinite) lazy list. A stream processing function can handle streams efficiently. The premature termination feature of the `map+reduce` scheme functions only if both of its components are stream processing functions.

```
-- the reduce parameter should fulfil the shortcutting property
map+reduce :: (Trans a, Trans b, Trans c)
           ⇒ (Map a b)                          -- ^ map
           → ((b → c → c) → c → [b] → c)  -- ^ reduce
           → (a → b)                            -- ^ map worker
           → (b → c → c)                        -- ^ reduce worker
           → c                                  -- ^ reduce zero
           → [a] → c                            -- ^ input and output
map+reduce amap areduce f g z xs = areduce g z $ amap f xs

-- a version with applied worker functions
-- again, the desired property of the reduce--the shortcutting--cannot
-- be encoded in the type system.
map+reduce' :: (Trans a, Trans b, Trans c)
            ⇒ ([a] → [b]) -- ^ partially applied map
            → ([b] → c)   -- ^ partially applied reduce
            → [a] → c     -- ^ input and output
map+reduce' amap areduce = areduce ∘ amap
```

Figure 5.12: *A generic* `map+reduce` *skeleton scheme*

```
farm+reduce' :: Trans a ⇒ (a → Bool) → [a] → Bool
farm+reduce' f = map+reduce' (farm f) and
```

Figure 5.13: *Implementing* `farm+reduce` *using the generic scheme.*

Laziness. Our `map+reduce` scheme works, because of some special properties of the `reduce` function. We consider these in a more detail. We define commutative semirings[2] in Figure 5.14. These suffice in our case. It is easy to encode commutative monoids in Haskell's type system.

We define `reduce`, or, to be more exact: a left `fold` for a multiplicative operation, but keep in mind the possibility of a `zero` occurrence. Our shortcutting property is nothing else than such occurrence of zero in a product-based `fold`.

```
myLeftFold :: (Semiring a, Eq a) ⇒ [a] → a
myLeftFold xs = myLeftFold' xs unity

myLeftFold' :: (Semiring a, Eq a) ⇒ [a] → a → a
myLeftFold' (x:xs) acc | x==zero = zero        -- sic!
                       | otherwise = myLeftFold' xs (mult acc x)
myLeftFold' _ acc = acc
```

Assuming

- The commutativity of `mult`
- The laziness of `mult`, i. e., the fact `mult zero ⊥ = zero`

we can write the equivalent right `fold`.

```
myRightFold :: Semiring a ⇒ [a] → a
myRightFold (x:xs) = mult x (myRightFold xs)
myRightFold [] = unity
```

[2] A commutative semiring $(R, +, \cdot)$ is a commutative monoid w. r. t. $+$ and a commutative monoid w. r. t. $\cdot$. Additionally, a distributive law should hold. In commutative semirings zero annihilates any element, i. e., for all $x \in R$ holds $0 \cdot x = 0$.

```haskell
-- slightly simplified
class AddMonoid a where
  zero :: a
  add :: a → a → a

class MultMonoid a where
  unity :: a
  mult :: a → a → a

class (AddMonoid a, MultMonoid a) ⇒ Semiring a
  -- distributive law holds: for all x, y, z:
  -- x `mult` (y `add` z) == (x `mult` y) `add` (x `mult` z)
  -- zero annihilates everything: for all x: zero `mult` x == zero

instance AddMonoid Int where
  zero = 0
  madd = (+)

instance MultMonoid Int where
  unity = 1
  mult = (*)

instance Semiring Int

instance AddMonoid Bool where
  zero = False
  add = (||)

instance MultMonoid Bool where
  unity = True
  mult = (&&)

instance Semiring Bool
```

Figure 5.14: *Commutative rings in Haskell. We show instances for* `Int` *and* `Bool`.

We have no special comparison with `zero` in the above code, however it relies now on `mult` to preserve zero and not to evaluate the recursive call if a `zero` occurs in the first parameter. We conclude that the premature termination in our `map`+`reduce` setting will function with `myRightFold` as a `reduce`, if its parameter is a zero-preserving lazy `mult` function.

Implementing `map-reduce` with `map`+`reduce`. We have seen above, how a sequential `reduce` with the premature abort property looks like. Under the assumption of commutativity, it is possible to construct a distributed `reduce` with the same properties. We do not focus on this construction and show how it is possible to implement `map-reduce` with our scheme, assuming distributed `reduce` as given.

We stress that the `map-reduce` skeleton is not merely a concatenation of `map` and `reduce`. The key issue is to keep the data distributed across the PEs after the `map` phase, such that `reduce` could operate on already distributed data. This is represented graphically in Figure 5.10 on page 57.

It is possible to keep the data distributed, even after the computation has ended. Using remote data [Dieterle et al., 2010b] we can accomplish this task and thus compose the skeletons elegantly. We implement the function `map-reduce1` in Figure 5.15, top, with fitting parallel implementations of the parallel map (here: `parMap`) and of the distributed reduce (`distReduce`, not elaborated here). The data locality

```
distReduce :: (c → d → d) → d → [c] → d

map-reduce1, map-reduce2 :: (a → [b])
                         → (b → c → c)
                         → (c → d → d)
                         → c
                         → d
                         → [a]
                         → d
map-reduce1 f r1 r2 z1 z2
  = map+reduce' (parMap ((foldr r1 z1) ∘ f)) (distReduce r2 z2)

map-reduce2 f r1 r2 z1 z2
  = map+reduce' (parMap (release ∘ (foldr r1 z1) ∘ f))
                ((distReduce r2 z2) ∘ fetchAll)
```

Figure 5.15: *Two implementations of* `map-reduce` *with* `map+reduce` *and remote data.*

is maintained for the first reduce call in this skeleton. However, we need to optimise the data flow in the transaction from the distributed reduce to the global reduce. Hence, we use remote data in function `map-reduce2`. We utilise the functions `fetchAll :: [RD a] → [a]` and `release :: a → RD a`, see Section 3.1.4. This results in Figure 5.15, bottom. We can implement the classic `map-reduce` skeleton with `map+reduce` and remote data. As we need an additional feature to implement `map-reduce` efficiently, `map+reduce` is a less powerful concept than `map-reduce`.

5.3 Case Studies

Below we present a parallel implementation of two probabilistic primality tests as large examples of our approach to repeated computation. These tests check if their input is prime in a probabilistic, random number based manner. Recapitulate the following definition.

Definition (Prime). A natural number > 1 is a *prime number* if its only divisors are unity and the said number.

For example, 5 is prime, because $2 \nmid 5$, $3 \nmid 5$, $4 \nmid 5$. The first and the oldest method to obtain prime numbers is the sieve of Eratosthenes. Details on Haskell implementation are in [O'Neill, 2008]. However, the contemporary mathematics embarks on a different way, and not without a reason. The problem with the sieve is: we need to precompute all the preceding primes. This is surely both time and space inefficient. But there is no closed formula for prime numbers! There is some work on expressing primes as non-negative values of large multivariate polynomials, *cf.* [Matiyasevich, 1971, 1981, Jones et al., 1976]. However it leads it pretty much astray: to Diophantine equations and Hilbert's Tenth Problem. So, the modern number theory moves from prime number generation to prime number detection. With this approach, if we need a prime in a certain range, e.g., a really large prime number, we test random integers for primality, until we succeed. This method does not involve enumerating all primes. Still, it has some other caveats.

Given a number, we need to decide whether it is prime. For another unfortunate condition, the most straightforward way to decide this for sure and for all cases is to perform a trial division from two to the square root of the prime candidate. This is again unacceptable. There exist deterministic polynomial-time primality tests [Agrawal et al., 2004], but their practical complexity is quite bad, thus we refrain from implementing them. There *are* more practical methods, but they provide less guaranties. The probabilistic tests provide a very high probability of a prime candidate being indeed a prime number. Some are even capable to decide whether the input is prime for almost all inputs, only an explicit failure is possible in this case, but not a false positive. So, we focus here on the state of the

Input	prime	not prime
Success	'yes'	'no'
Failure	'yes'	'yes'

Table 5.2: *The result matrix of Rabin–Miller test. The question is 'is input prime?'*

art: the probabilistic primality tests. We consider here two methods: Rabin–Miller test and Jacobi sum test. In case of the *Rabin–Miller primality test*, the *negative* outcome of the test means the input was not a prime. The *positive* outcome means not extraordinary much: the input *could* be a prime. We call such numbers 'strong pseudo-primes', exactly these are subjects of the further testing. The next test, the *Jacobi sum test*, also called APRCL, which stands for Adleman, Pomerance, and Rumely [1983], Cohen and Lenstra [1984]. This test gives us a *proof* of primality. So it either guarantees the input is prime, or guarantees the opposite, or fails. The uncertainty lies in the inability to detect *all* prime numbers with the test: if a number is accepted by the test, it is a prime. If it is deemed as not a prime, it is not a prime. Still, an input number might be rejected, although with an extremely little probability. We present these two algorithms following [Cohen, 2000]. The Rabin–Miller test is presented in Section 5.4, whereas Section 5.5 presents the Jacobi sum test.

5.4 Rabin–Miller Test

Here we describe the Rabin–Miller primality test. This result is due to Miller [1976] and Rabin [1980], we present it based on a book by Cohen [2000]. The test is probabilistic, it is a Monte Carlo algorithm. If the test answers 'not a prime', then the input is indeed *not* prime. However, if the test answers 'is a prime', this does not mean anything. Inputs exist, for which the test fails to detect that they are not prime. In other words, false positives are possible, but not false negatives. All four possible cases are depicted in Table 5.2.

Theory. The test relies on an old and fundamental result in number theory called Fermat's Little Theorem. The name of the Fermat's Little Theorem is opposed to the Fermat's Last Theorem. The theorem in question was discovered in 1640 by Pierre de Fermat, *1607 or 1608, †12.1.1665.

Theorem 5.1 (Fermat's Little)**.** *Let $N, p \in \mathbb{N}$ with p a prime. Then $N^p \equiv N \pmod{p}$.*

It is a special case of Euler's Totient Theorem, *viz.* [Hardy and Wright, 1975, Theorem 72], which extends the case of prime p to $N \perp p$, i.e., $\gcd(N, p) = 1$. This theorem is named after Leonhard Euler, *15.4.1707, †18.9.1783. Euler proved his theorem in 1736, while Leibniz produced his proof before 1683, but never published it.

The classic proof uses group theory. We prove the theorem using binomial coefficients, *cf.* [Hardy and Wright, 1975, von zur Gathen and Gerhard, 2003, Ribenboim, 2004]. The notion of $\mathbb{Z}/\mathfrak{p}$ with prime p describes a finite field modulo p: the numbers $\{0, \ldots, p-1\}$ with corresponding addition and multiplication modulo p. The notion $\mathfrak{p}$ denotes a principal ideal, generated by p. Recall, the value $\binom{n}{k}$ is the binomial coefficient $\frac{n!}{k!(n-k!)}$, where $n! = 1 \cdots n$.

Lemma 5.2 (Binomial coefficients in finite fields)**.** *In $\mathbb{Z}/\mathfrak{p}$ with a prime p holds $(a+b)^p \equiv a^p + b^p \pmod{p}$.*

Proof. Generally, $(a+b)^p = \sum_{k=0}^{p} \binom{p}{k} a^k b^{p-k}$ holds. However, for all $0 < k < p$ with a prime p holds $\binom{p}{k} \equiv 0 \pmod{p}$, as each $\binom{p}{k}$ has the factor p in its numerator. □

Proof of Fermat's Little Theorem. We use induction on $k \in \mathbb{N}$. Let $p \in \mathbb{N}$ be a prime. Trivially, $0^p \equiv 0 \pmod{p}$. Now, assuming the theorem holds for some k^p, consider $(k+1)^p$. Then by the previous lemma, $(k+1)^p \equiv k^p + 1 \pmod{p}$. But as $k^p \equiv k \pmod{p}$, we conclude $(k+1)^p \equiv k+1 \pmod{p}$. □

Algorithm 1 Rabin–Miller test.

Require: odd integer $N \geq 3$.

1: **function** Main Rabin-Miller(N)
2: Find such q and t that $N - 1 = 2^t q$ with odd q.
3: Generate 20 random numbers between 1 and N as a list as.
4: Let $bs \leftarrow$ **map** $(\lambda a \rightarrow a^q \pmod N)$ as. *// Power random numbers modulo N.*
5: Let $res \leftarrow$ **map** $(\lambda b \rightarrow$ Single Rabin–Miller$(N, t, 0, b))$ bs. *// Execute subtests.*
6: **return true** only if all elements in res are **true**. Else **return false**.
7: **end function**
8: **function** Single Rabin–Miller(N, t, e, b)
9: **if** $b = \pm 1 \pmod N$ **then return true**
10: **else if** $e \leq t - 2$ **then return** Single Rabin–Miller$(N, t, e + 1, b^2 \pmod N)$
11: **else return false**
12: **end if**
13: **end function**

Ensure: Either **false** for 'N is composite' or **true** for 'N is probably prime'.

It is essential for p to be a prime, because else some inner binomial coefficients in Lemma 5.2 could be not zero modulo p. To give an example: $\binom{6}{3} \equiv 2 \pmod 6$. So the aforementioned proof does not work when p is not a prime. But in fact the whole theorem is broken in this case: $8^{12} \equiv 4 \pmod{12}$. The premise of the Euler's Totient Theorem does not hold: $\gcd(8, 12) \neq 1$.

Definition 5.3. Let $N \in \mathbb{N}$ a odd natural number and $a \in \mathbb{Z}$. We can write $N - 1 = 2^t q$ with an odd q. We call N a *strong pseudo-prime* in a base a if

- Either $a^q = 1 \pmod N$
- Or there exists such e that $0 \leq e \leq t$ with $a^{2^e q} = -1 \pmod N$

Naturally, all odd primes are strong pseudo-primes in arbitrary bases. However, for a composite number, less than ¼ of all possible bases evidence it as a strong pseudo-prime. For a proof of this statement see e. g., [Koblitz, 1994, Knuth, 1998, Yan and Hellman, 2002]. We can now devise Algorithm 1. Its imperative version is stated in [Cohen, 2000, Algorithm 8.2.2].

Implementation. We have a few implementation-specific remarks. Firstly, note that the test might abort with 'N is composite' after any 'subtest'—an individual call of Single Rabin–Miller. Secondly, it is not required to generate the random numbers while the test is in progress. It suffices to pregenerate sufficiently many random numbers in a given range. Hence, we have a test case for our new skeletons. Further, Rabin–Miller test is frequently used as a first stage of more complicated tests. We will elaborate on this in Section 5.6.

We consider the Rabin–Miller primality test in a more detail. It is is a probabilistic test. The input of the test is a prime number candidate. It consists of 20 runs of the same subtest with different parameters. If one of the subtests fails, the whole test has failed. This case is always correct in its output: the number is not prime. Each successful subtest increases the probability of the input being prime, but we are never sure.

As we know, the probability for a single subtest to fail is less than ¼. When choosing the bases randomly (*cf.* basis a from Definition 5.3), the probability of a false negative for k tests is 4^{-k}. We sketch the parallel implementation of Algorithm 1 in Figure 5.16. It is a slightly simplified version. We could have made use of `farm`+`reduce` higher-order function from Figure 5.6. However, in the presented implementation we instantiate the generic `map`+`reduce` skeleton scheme seen in Figure 5.12. The implementation is straightforward. The omitted function `singleRabinMiller` corresponds to the procedure of the same name in Algorithm 1. We show it and a few more helper functions in the Appendix, in Section B.1.

```
singleRabinMiller :: Integer       -- ^ n is the prime candidate
                  → Integer        -- ^ t from n=2^t*q
                  → Integer        -- ^ current e from a^((2^e)q)
                  → Integer        -- ^ b = a^q
                  → Bool           -- ^ result
singleRabinMiller = ... -- implementation omitted

separate :: (Integer, Integer) → (Integer, Integer)
separate ... =  -- implementation omitted, finds q and t

listRabinMiller :: Int → Integer → [Integer] → Maybe Integer
listRabinMiller k n as = let worker :: (Integer, Integer) → Bool
                             worker (n, a) = singleRabinMiller n t 0 b
                                 where (q, t) = separate (n-1, 0)
                                       b = powermod a q n
                             tasks = take k [(n, a) | a ∈ as]
                             reduce :: Integer → [Bool] → Maybe Integer
                             reduce n bs | and bs = Just n
                                         | otherwise = Nothing
                         in map+reduce' (farm worker) (reduce n) tasks

randomBaseList :: Integer → IO [Integer]
randomBaseList = ... -- generate random bases for given n.
                     -- Implementation omitted

rabinMillerIO :: Integer -- ^ n is the prime candidate
              → IO (Maybe Integer) -- ^ Nothing or Just n
rabinMillerIO n = do
  ls ← randomBaseList n
  return $ listRabinMiller 20 n ls
```

Figure 5.16: *The main loop of Rabin–Miller test in parallel.*

Parallelisations of the Rabin–Miller test were discussed in [Hahnel, 1998, Cheung et al., 2004, Schmidt et al., 2004]. These works focus on hardware design. In a contrast, we will use our `map+reduce` approach.

Discussion. We evaluate the parallel behaviour of our implementation of the Rabin–Miller test on `sakania`. An earlier version of our implementation was used as a test program in [Lobachev and Loogen, 2010c]. We present a trace visualisation of a test run in Figure 5.17 on the facing page. We have used up to 8 cores in parallel, the input was $2^{9689} - 1$. Numbers of a form $2^p - 1$ for a prime p are called Mersenne numbers, after Father Marin Mersenne, *8.9.1588, †1.9.1648. They are often prime. The number $2^{9689} - 1$ is prime. Hence, the test makes all 20 runs of the subtests. We can see it in the trace diagram: the thin bars at the first PE—once at 2.3 seconds mark and once at 4.6 seconds mark—delimit the rounds of subtest executions. We see also that we could have done the one more subtest 'for free', as one PE is not used in the third round. We will discuss this issue in more detail below.

The speedup is displayed in Figure 5.18, left. It is 6.61 on 7 PE with input size $2^{9689} - 1$. The best speedup value is 6.63 for the input size $2^{11213} - 1$ on `sakania`. The efficiency on 7 PE is 0.94 and 0.95 appropriately. Please notice the linear speedup for up to 5 PEs and a strange behaviour for 6–8 PEs. The reason for such an effect is not the bad parallelism, but problems with task placement. Let us discuss the latter in more detail.

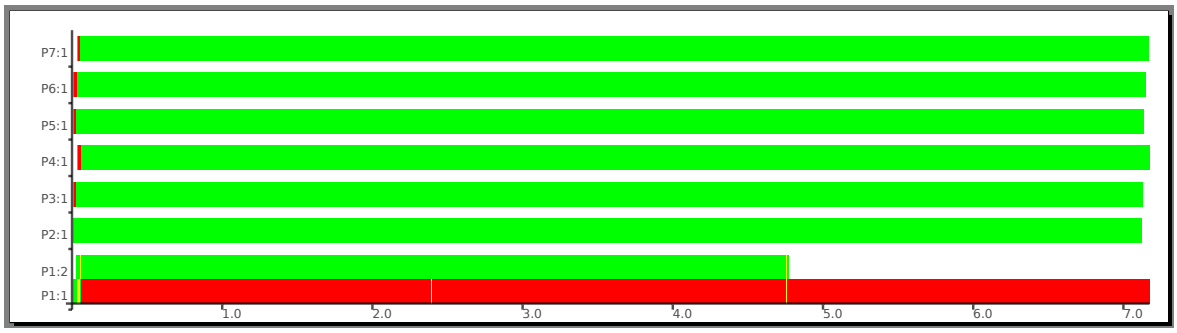


Figure 5.17: *Trace diagram of parallel Rabin–Miller test on 7 PEs. The input size is* $2^{9689} - 1$*. Note the task balancing issue.*

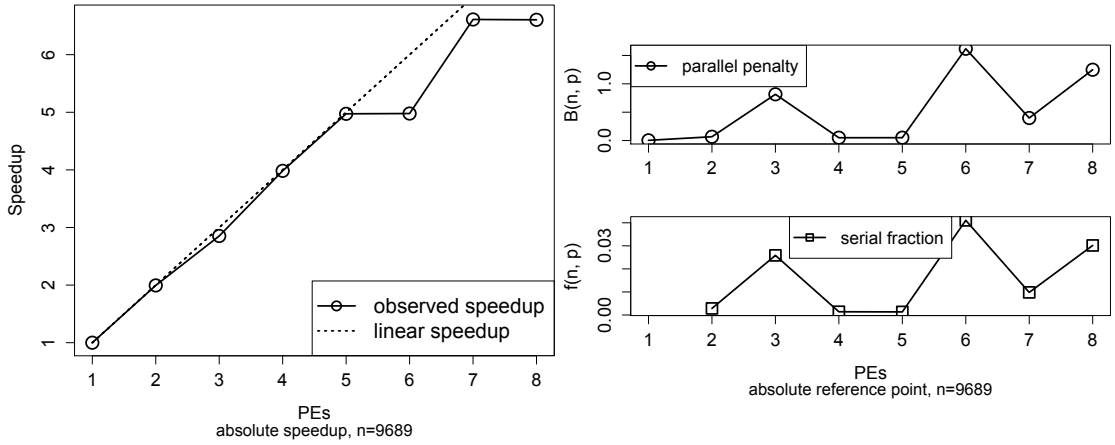


Figure 5.18: *Speedup (left), parallel penalty (top right) and serial fraction (bottom right) for Rabin–Miller test on* `sakania` *with absolute reference point. We use* $2^{9689} - 1$ *as input.*

The parallel penalty function and serial fraction are depicted in Figure 5.18, right. These plots are similar to the ones from [Lobachev and Loogen, 2010c]. Both curves indicate a problem with the task distribution. As of now we always request 20 tasks. However, as we also see in Table 5.3, not all numbers of PEs are of a benefit for an equally-balanced parallel computation. Consider the latter table in more detail. The first row ('PEs') shows the total number of processing elements. The second and third rows appropriately show how many times the outer loop is fully saturated ('full rounds') and how many tasks remain for the last, incomplete round ('remaining tasks'). These two values correspond to the result of the division with remainder of the number of tasks (20) by the number of PEs. Revisiting these data for the parallel repeated computation, we need to increase the number of rounds by one to obtain the total amount of rounds in the fourth row. The fifth row of the table shows the number of free PEs in the incomplete round. The last row ('slack-off'), shows the relation of the fifth row to the first row. It describes how many PEs are idle in the last round. These figures are easily deduced from the full rounds and the remaining tasks, e. g., the number of the free PEs is the total amount of PEs minus the remaining tasks. Every time we see a large figure in the fifth row, poor task balancing ensues in the last, incomplete round of the Rabin–Miller test. These theoretical considerations correspond nicely to the picture we see in both parts of Figure 5.18. Both plots of parallel penalty and of serial fraction describe the 'bad' parts of a parallel program. The larger the values are, the worse does the parallel Rabin–Miller test run on this number of PEs. We see a flat line at 4–5 and peaks at 3, 6 and 8 PEs, which is compatible to Table 5.3. Note a small decline in the speedup curve for 3 PE. It corresponds to the entry in the last row of Table 5.3 and to the peak at 3 PE in Figure 5.18. We chose 7 PEs for Figure 5.17 as a compromise between larger number of processes and smaller overhead in task balancing. Another proper decision would be to adapt the number of tests, e. g., use 21 for 7 PEs. This would make a better (and stronger!) Rabin–Miller test, but would render versions on different PEs incomparable.

PEs	1	2	3	4	5	6	7	8	9	10
Full rounds	20	10	6	5	4	3	2	2	2	2
Remaining tasks	0	0	2	0	0	2	6	4	2	0
Total rounds	20	10	7	5	4	4	3	3	3	2
Unused PEs	0	0	1	0	0	4	1	4	7	0
Slack-off, %	0	0	33.3	0	0	66.6	14.3	50	77.78	0

Table 5.3: *Overheads for process placement: 20 tasks at 1 to 10 PEs.*

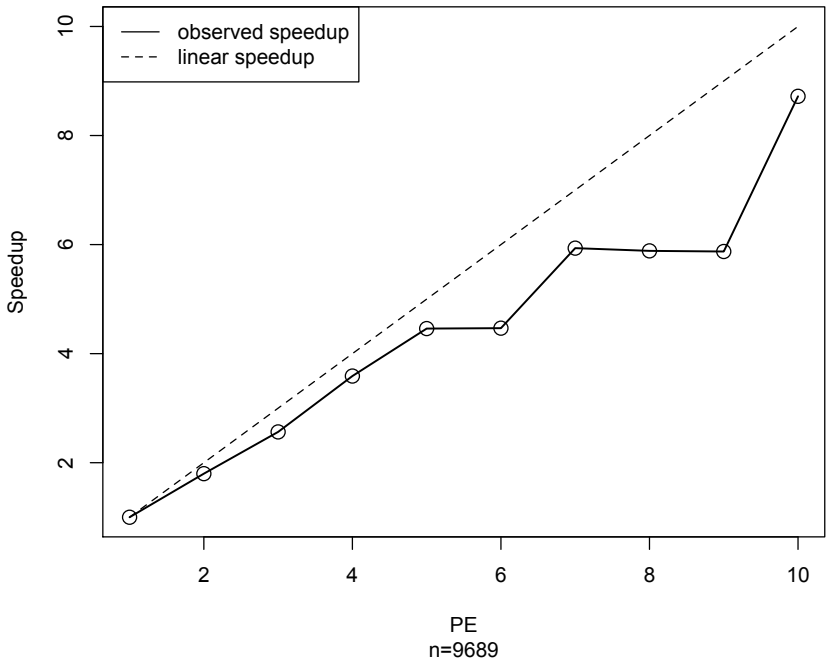


Figure 5.19: *Speedups for Rabin–Miller test on local workstations. We use* $2^{9689} - 1$ *as input.*

Note that ²%₂₁ is 0.9523... This corresponds well with the efficiency values at 7 PE, e.g., 0.95 for the input size $2^{11213} - 1$, as we had exactly one 'slot' free.

Summarising, we would assume the same program to perform significantly better on 10 PEs, than it does on 8. We cannot test this assumption on `sakania`, as it is a 8 core machine. Still we performed the test run in question on local workstations. We used 10 Intel Core2Duo machines, running 64 bit Linux OS. The CPU frequencies of these machines differed, so some tasks were processed faster than others. Still we obtained the speedup of 8.72 on 10 PE. The speedup plot is shown in Figure 5.19. Our assumption was correct: on 10 machines all tasks can be computed in two full rounds. This is the reason for the 'jump' of the speedup curve in the right part of the plot.

As for the speedup in the range 1–5 PE in Figure 5.19, the curve is worse than in Figure 5.18. This is because of not perfect load balancing. The machines in local workstations have similar, but not identical performance, thus some tasks are computed faster on certain machines, causing a minor speedup deficiency. We confirm this fact by examining a detail of the trace of Rabin–Miller test on the local workstations in Figure 5.20. We see different PEs terminate at different times, although the task size is approximately the same for all workers—see Figure 5.17.

Estimation of the execution time. To conclude the discussion of Rabin–Miller test, we examine an estimation of the execution time for this test. We use the measurements, performed on `sakania`.

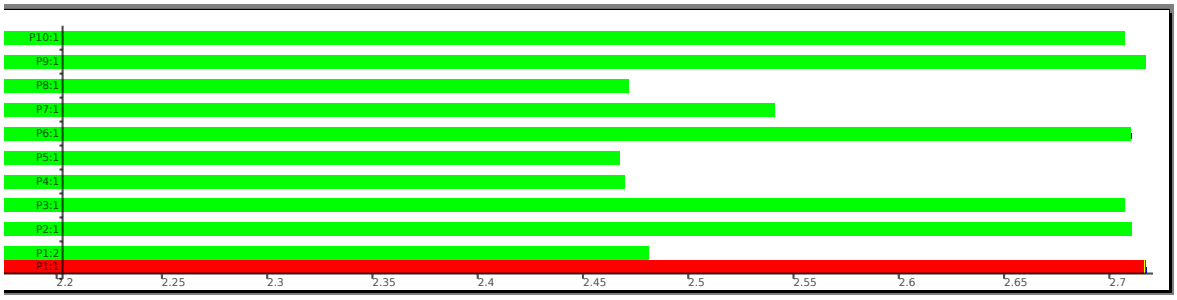
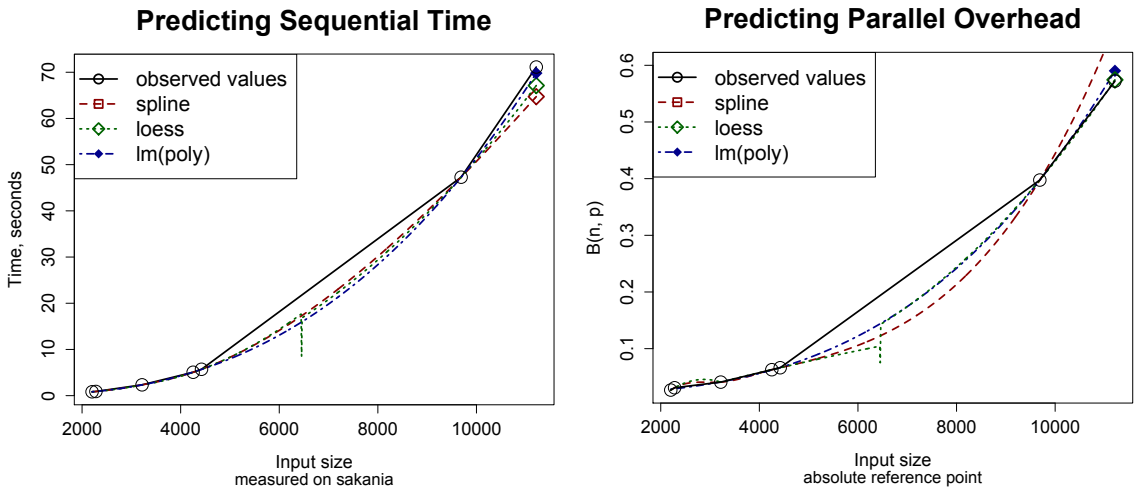


Figure 5.20: The zoom into the trace on local workstations for Rabin–Miller test.



Method	spline	loess	lm(poly)
Rel. err, %	−7.334	−5.632	−1.859

Method	spline	loess	lm(poly)
Rel. err, %	17.21	0.4007	3.180

Figure 5.21: Predicting the execution time of Rabin–Miller test. Left: sequential run time, right: parallel penalty.

Mersenne primes are used as input, we denote the input sizes by the exponent n in $2^n - 1$. The estimation is performed w.r.t. input size, we assume all execution times up to $n \leq 9689$ as given and aim to predict the execution time for $n = 11213$ at 7 PE. Figure 5.21 shows the both components. We are able to predict the sequential execution time of Rabin–Miller test with −1.86% relative error with the lm(poly) method (*viz.* Figure 5.21, left). The loess method results in the value 0.574 for the parallel penalty $\hat{B}(11213, 7)$. The observed value is 0.572, *cf.* Figure 5.21, right. Combined, these methods predict the observed run time $T(11213, 7)$ up to the relative error of −1.739%.

Overall, we see very good parallel performance of the instantiation of the `map+reduce` skeleton scheme with `farm` and Rabin–Miller test. We consider a much more complex case in the next section.

5.5 Jacobi Sum Test

The Jacobi Sum test, also called the APRCL test [Adleman et al., 1983, Cohen and Lenstra, 1984], is a sophisticated primality test. Contrarily to Rabin–Miller test, this test issues a *proof* of primality. So, it assures primality in cases it is successful. In a seldom case of failure, this test also does not provide a wrong statement. For the detailed matrix of possible test results see Table 5.4. Jacobi sum test is a Las Vegas algorithm. We will require a deeper insight into the theory to understand how the Jacobi sum test performs its task.

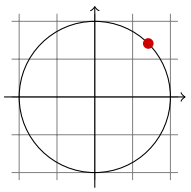


Figure 5.22: *A primitive root of unity of order eight: ζ_8.*

5.5.1 Theoretical Background

We generally follow [Cohen, 2000, 2007] and [Lang, 2002] in this section. The key idea of the APRCL test is to test for Fermat-like congruences in finely chosen cyclotomic fields. We seldom give proofs in this section. Instead, we refer to the aforementioned books for all missing proofs and a more rigorous treatment of the following matter.

Cyclotomic fields and cyclotomic polynomials. Let $\zeta_n \in \mathbb{C}$ be a primitive root of unity of n^{th} order. It means $\zeta_n^n = 1$ and no ζ_n^k for any $0 < k < n$ is unity. We may understand ζ_n as an abstract symbol, but one could envision $\zeta_n = e^{2i\pi/n}$. Here, and everywhere else, when talking about complex numbers, i denotes the imaginary unit, $\sqrt{-1}$. Naturally, the symbol π has its usual value, 3.141592653589793 ... We find primitive roots of unity also in the fast Fourier transform of Section 6.4.

The roots of unity are points on the complex unity circle. They are called primitive, if none of the powers less than n of the n^{th} root of unity is the actual unity. Figure 5.22 shows an example of a primitive root of unity of order eight. The shown root of unity is the primitive one of order eight, as none of its powers less than eight equals unity.

For an arbitrary (commutative) ring R with unity we denote its multiplicative group with R^*. Given a field F, a root a of a minimal polynomial p of degree k in $F[x]$ with $a \notin F$ can be *adjuncted* to F. We write $F(a)$ for it. It holds $F(a) \cong F/\langle p\rangle$. Here $\langle p\rangle$ is an ideal, generated by p. Is an ideal *principal*, i. e., generated by a single p, then we also write $\mathfrak{p}$ for it. So, $F/\langle p\rangle$ is a factor ring. We call $F(a)$ an algebraic extension of F. A classic example is $\mathbb{C} = \mathbb{R}(i)$. In this case $i = \sqrt{-1}$ and $p(x) = x^2 + 1$. Further, on page 78 we will see how to represent $F(a)$ efficiently.

Definition 5.4 (Cohen [2000], Definition 9.1.1)**.** For a positive integer n, the n^{th} *cyclotomic field* is $\mathbb{Q}(\zeta_n)$ for ζ_n the n^{th} primitive root of unity.

Definition 5.5 (Cohen [2007], Definition 3.5.1)**.** Define the *cyclotomic polynomials* $\Phi_n(x)$ as unique rational functions with

$$\prod_{d|n} \Phi_d(x) = x^n - 1.$$

There is an alternative definition of cyclotomic polynomials [Lang, 2002]: begin inductively with $\Phi_1(x) = x - 1$ and apply

$$\Phi_n(x) = (x^n - 1) / \prod_{\substack{d|n \\ 1 \le d < n}} \Phi_d(x).$$

It can be easily seen, that cyclotomic polynomials are well-defined. The first cyclotomic polynomials are $x - 1$, $x + 1$, $x^2 + x + 1$, $x^2 + 1$, $x^4 + x^3 + x^2 + x + 1$, $x^2 - x + 1$. However, not always the coefficients

Input	prime	not prime
Success	'yes'	'no'
Failure	'failure'	'failure'

Table 5.4: *The result matrix for Jacobi sum test. The question is 'is the input prime?'*

of cyclotomic polynomials are –1, 0 and 1, see also [Migotti, 1883, Bungers, 1934, Lehmer, 1936]. The author of the latter publication is Emma Lehmer, nèe Emma Markowna Trotskaja (Эмма Марковна Троцкая), *6.11.1906, †7.5.2007, the wife of Derrick Henry Lehmer of Lucas–Lehmer test and Lehmer's GCD algorithm.

We write $\mathfrak{q}$ for an ideal, generated by $q \in \mathbb{Z}$.

Proposition 5.6 (Cohen [2007], Proposition 3.5.10)**.** *It holds that*

1. *The polynomial $\Phi_n(x)$ is irreducible in $\mathbb{Q}[x]$. It is the minimal polynomial of ζ_n in $\mathbb{Q}$.*
2. *The extension $\mathbb{Q}(\zeta_n)/\mathbb{Q}$ is a Galois extension, its Galois group is canonically isomorphic to the group $(\mathbb{Z}/\mathfrak{n})^*$.*

Why are cyclotomic fields so important? The following theorem answers this question. See [Cohen, 2007, Theorem 3.5.13] for the proof.

Theorem 5.7 (Kronecker–Weber)**.** *Any abelian extension of $\mathbb{Q}$ is a* subfield *of some cyclotomic field.*

Group rings. We follow [Cohen, 2000] also in notation. Cohen uses the notion of group rings to express the further content. The following definition works for infinite groups if only a finite number of maps are not zero.

Definition 5.8 (Cohen [2000], Definition 9.1.3)**.** Let G be a finite group. The set of maps from G to $\mathbb{Z}$ builds a *group ring* $\mathbb{Z}[G]$. Let f and g be in $\mathbb{Z}[G]$ The addition in $\mathbb{Z}[G]$ is defined as $(f+g)(\sigma) = f(\sigma) + g(\sigma)$. The multiplication is more complicated:

$$f \cdot g(\sigma) = \sum_{\tau \in G} f(\tau) g(\tau^{-1}\sigma).$$

These two operations give a ring structure to $\mathbb{Z}[G]$, which justifies the name. Formally we can write

$$f = \sum_{\sigma \in G} f(\sigma)[\sigma],$$

hence we establish direct connection of operations in $\mathbb{Z}[G]$ to $\mathbb{Z}$-algebra laws. Further, we identify $\mathbb{Z}$ as a subring in $\mathbb{Z}[G]$: a $n \in \mathbb{Z}$ can be identified with $n[1] \in \mathbb{Z}[G]$. Here $[1]$ is a unity in $\mathbb{Z}[G]$.

The *group action* $\bullet$ for a set X and a group G is defined as follows. It has the type $\bullet : G \times X \to X$ and fulfils the properties

1. $1 \bullet x = x$ for 1 the neutral element of G
2. $g \bullet (h \bullet x) = (gh) \bullet x$ for all $g, h \in G$

for all $x \in X$. In a special case $G = \mathrm{Gal}(K/\mathbb{Q})$, for an algebraic extension K of $\mathbb{Q}$, the group G acts on K. The latter action can be extended to $\mathbb{Z}[G]$ as a multiplicative extension. Let $f \in \mathbb{Z}[G]$ with

$$f = \sum_{\sigma \in G} n_\sigma [\sigma]$$

and $x \in K$. We write

$$x^f = \prod_{\sigma \in G} \sigma(x)^{n_\sigma}.$$

Definition 5.9. Let p be a prime number, let further be $k \in \mathbb{Z}$ and $n = p^k$. Let K be n^{th} cyclotomic field. Let G be its Galois group. It holds $G = \{\sigma_a : a \in (\mathbb{Z}/\mathfrak{n})^*\}$. Let $\zeta_p = e^{2\pi i/p}$. In contrast to $\mathfrak{p} = \langle p \rangle \subset \mathbb{Z}$, define

$$\tilde{\mathfrak{p}} := \{f \in \mathbb{Z}[G] : \zeta_p^f = 1\}.$$

The following originates from [Cohen, 2000, pp. 446–447], see there for the detailed proof.

Lemma 5.10. *The above ideal $\bar{\mathfrak{p}}$ is a prime ideal in $\mathbb{Z}[G]$.*

Idea of the proof. Let

$$f = \sum_{a \in (\mathbb{Z}/\mathfrak{n})^*} n_a[\sigma_a],$$

then $f \in \bar{\mathfrak{p}}$ if and only if $\sum_a n_a a \equiv 0 \pmod{p}$. The number of equivalence classes of $\mathbb{Z}[G]/\bar{\mathfrak{p}}$ is p. †

Characters and sums. We need to build up some more theory, *cf.* [Cohen, 2007, Definition 2.1.15] and [Cohen, 2000, Section 9.1.2]. Recall: the group $\mathbb{C}^*$ is the multiplicative group of non-zero complex numbers. Analogue, $(\mathbb{Z}/\mathfrak{q})^*$ is the multiplicative group of invertible residues modulo q. We denote the residue equivalence class of a modulo q as $|a|_q \in \mathbb{Z}/\mathfrak{q}$. For a group G we denote its dual group with $\widehat{G}$, it is a group of homomorphisms [Lang, 2002].

Definition 5.11. Let G be a finite abelian group.

1. A generic *character* of G is a group homomorphism from G to $\mathbb{C}^*$. All characters build a group $\widehat{G}$, the dual group of G.
2. The *Dirichlet character* modulo q is a map χ from $\mathbb{Z}$ to $\mathbb{C}$, such that a character $\psi \in \widehat{(\mathbb{Z}/\mathfrak{q})^*}$ exists with
   - $\chi(n) = 0$ if $\gcd(q, n) > 1$ and
   - $\chi(n) = \psi(|n|_q)$ otherwise.

   In the further text we use only Dirichlet characters. So, we call them simply *characters (modulo q)*.

The group $(\mathbb{Z}/\mathfrak{q})^*$ is (non-canonically) isomorphic to its dual group $\widehat{(\mathbb{Z}/\mathfrak{q})^*}$. Note that χ modulo q is multiplicative and periodic in q, i.e., $\chi(n+q) = \chi(n)$ for all n. The unit element of the group of characters is denoted with χ_0. The following proposition tells us a bit more about characters.

Proposition 5.12. *Let χ be a character and $\chi \neq \chi_0$. Then*

$$\sum_{x \in (\mathbb{Z}/\mathfrak{q})^*} \chi(x) = 0.$$

This proposition is one direction of [Cohen, 2000, Proposition 9.1.4]. Look there for the proof. For the next two definitions see also [Cohen, 2000, Definition 9.1.5.1 and Definition 9.1.5.2].

Definition 5.13 (Gauß sum)**.** Let χ be a character modulo q. The *Gauß sum* $\tau(\chi)$ is

$$\tau(\chi) = \sum_{x \in (\mathbb{Z}/\mathfrak{q})^*} \chi(x)\zeta_q^x,$$

with ζ_q the q^{th} root of unity.

Definition 5.14 (Jacobi sum)**.** Let χ_1 and χ_2 be characters modulo q. The *Jacobi sum* of them, in sign $j(\chi_1, \chi_2)$, is

$$j(\chi_1, \chi_2) = \sum_{x \in (\mathbb{Z}/\mathfrak{q})^*} \chi_1(x)\chi_2(1-x).$$

It is possible to replace $(\mathbb{Z}/\mathfrak{q})^*$ with $\mathbb{Z}/\mathfrak{q}$ and to exclude $x = 1$. The following key property of the Jacobi sums allows us the feasible implementation of the test. Crandall and Pomerance [2005] explain the computational benefits of it. Computing with a Gauß sum $\tau(\chi_{p,q})$ in $\mathbb{Z}[\zeta_p, \zeta_q]$ would require doing arithmetic with lists of length $(p-1)(q-1)$, each list element is modulo N. The primes p might be of order $\ln N$. But the primes q are "as large as $(\ln N)^{c \ln\ln\ln N}$" for some $c > 0$ [Crandall and Pomerance, 2005]. With Jacobi sums we can remain in $\mathbb{Z}[\zeta_p]$. See [Cohen, 2000, Section 9.1.2] and specifically [Cohen, 2000, Proposition 9.1.6.2] for more details and a rigorous proof.

Proposition 5.15. *Let χ be a character modulo q of order n. It holds $n \mid \phi(q)$. Here is ϕ the Euler's totient function:*

$$\phi(k) = \#\{1 \le l \le k : \gcd(l,k) = 1\}.$$

Further

$$\tau(\chi) \in \mathbb{Z}[\zeta_n, \zeta_q],$$

but for two characters χ_1, χ_2 modulo q of orders m_1 and m_2 with $m_1, m_2 \mid n$ holds

$$j(\chi_1, \chi_2) \in \mathbb{Z}[\zeta_n].$$

For the above characters χ_1, χ_2 with additional property $\chi_1\chi_2 \neq \chi_0$ also holds

$$j(\chi_1, \chi_2) = \frac{\tau(\chi_1)\tau(\chi_2)}{\tau(\chi_1\chi_2)}.$$

Sketch of the proof. We follow [Cohen, 2000] in the proof of the last statement. Let the indices in all sums be over elements of $(\mathbb{Z}/\mathfrak{q})^*$. With $x = ty$ follows

$$\tau(\chi_1)\tau(\chi_2) = \sum_x \sum_y \chi_1(x)\chi_2(y)\zeta_q^{x+y} = \sum_t \sum_y \chi_1(t)\chi_1\chi_2(y)\zeta_q^{y(1+t)}.$$

The $\overline{\ }$ denotes complex conjugation. The sum $\sum_y \chi(y)\zeta_q^{ay}$ is for any $\chi \neq \chi_0$ either $\overline{\chi}(a)\tau(\chi)$ or zero. The latter is the case if $a \equiv 0 \pmod{q}$. Now for $\chi_1\chi_2 \neq \chi_0$ holds

$$\tau(\chi_1\chi_2) \sum_{t \neq -1} \chi_1(t)\overline{\chi_1\chi_2}(1+t) = \tau(\chi_1\chi_2) \sum_u \chi_1(u)\chi_2(1-u) = \tau(\chi_1\chi_2) j(\chi_1, \chi_2)$$

with a bijective mapping $u = t/(1+t)$ from $(\mathbb{Z}/\mathfrak{q}) \setminus \{0, -1\}$ to $(\mathbb{Z}/\mathfrak{q}) \setminus \{0, 1\}$. †

Now Proposition 5.16 and Theorem 5.18 on page 74 provide the insight to the core of the Jacobi sum test. But before we state them, we need to take care of some precomputations.

5.5.2 Helper Algorithms

Primitive roots modulo p. A corollary of Fermat's Little Theorem (Theorem 5.1) is that for any $n \in \mathbb{Z}/\mathfrak{p}$ with prime p a power e exists, such that $n^e \equiv 1 \pmod{p}$. This power is called the *order* of n modulo p. Consider $(\mathbb{Z}/\mathfrak{p})^*$, the multiplicative group of $\mathbb{Z}/\mathfrak{p}$ for an odd prime p. Some element g therein exists, with order of g equal $\phi(p) = p - 1$. Such element g is called a *primitive root* modulo p or a *generator*, see [Hardy and Wright, 1975]. In some cases primitive roots modulo composite numbers exist.

Primitive roots modulo p are not to be confused with (primitive) roots of unity. The first are generators of finite groups, while the latter are points on the complex unity circle. However, as the name suggests, they bear something in common. A primitive root of unity of order n is the generator of a cyclic group of order n. The relation between the two is emphasised on page 74.

Hardy and Wright [1975] explain primitive roots in quite a detail. We follow Stein [2009] and state an algorithm for computing a primitive root modulo a prime p as Algorithm 2. Cohen [2000] states a similar algorithm. This algorithm terminates for all prime p, as in this case all $(\mathbb{Z}/\mathfrak{p})^*$ *have* primitive roots [Hardy and Wright, 1975, Theorem 111]. Further, let d be a divisor of $p - 1$, such as $dn = p - 1$. Then for some a, which is *not* a generator of $(\mathbb{Z}/\mathfrak{p})^*$, a factor p_i of $p - 1$ exists with $p_i \mid n$. Now $a^{(p-1)/p_i} \equiv (a^{(p-1)/n})^{n/p_i} \equiv 1 \pmod{p}$. A converse also holds. See [Cohen, 2000, Stein, 2009] for details.

Algorithm 2 Primitive roots modulo p.

Require: prime p

function MAIN PRIMITIVE ROOT(p)
  if $p = 2$ **then return** 1 // *we are done!*
  end if
  Factor $p-1$ to $\{p_1, \ldots, p_k\}$.
  return HELPER PRIMITIVE ROOT($p, 2, \{p_1, \ldots, p_k\}, \{p_1, \ldots, p_k\}$)
end function
function HELPER PRIMITIVE ROOT(p, a, qs, rs)
  if qs list is empty **then return** a.
  end if
  Let q be the head of qs and qs' be the tail of qs.
  if $a^{(p-1)/q} \equiv 1 \pmod{p}$ **then return** HELPER PRIMITIVE ROOT($p, a+1, rs, rs$)
  else return HELPER PRIMITIVE ROOT(p, a, qs', rs).
  end if
end function

Ensure: a is a primitive root modulo p.

Algorithm 3 Precomputations for Jacobi sum test.

Require: upper bound B.

Look up in Table 5.5 a such t, that $e^2(t) > B$.
for every prime q with $q \geq 3$ **and** $q \mid e(t)$ **do**
  use Algorithm 2 to compute the primitive root g_q modulo q.
Tabulate the discrete logarithm function $f(x)$.
  for every prime p dividing $q-1$ **do**
    let $k \leftarrow v_p(q-1)$ and define the character $\chi_{p,q}$ with $\chi_{p,q}(g_q^x) = \zeta_{p^k}^x$.
    Compute the Jacobi sums $J(p,q)$.
  end for
end for

Ensure: precomputations for Algorithm 8.

Precomputation. The following helper algorithm (see Algorithm 3) performs precomputations, depending only on the upper bound B for the test, but not on the future input N of the test [Cohen, 2000, Algorithm 9.1.27]. The idea is that the stored precomputations for some B allow efficient processing of an arbitrary $N < B$. The stored data include the precomputed Jacobi sums. The table of values for t, which is used in Step 1 of Algorithm 3, is given in Table 5.5 on the facing page. The stated value of t is always given for all integers smaller than N. An improvement by Lenstra [1984] allows to replace $e^2(t) > B$ with $e^3(t) > B$ in the precomputation. For details on the choice of t see [Cohen and Lenstra, 1984, Lenstra, 1984, Cohen, 2000]. The discrete logarithm $f(x)$ to tabulate in Step 4 is defined for $1 \leq x \leq q-2$ is defined by $g_q^{f(x)} = 1 - g_q^x \pmod{q}$ with $1 \leq f(x) \leq q-2$. The value of g_q is found in Algorithm 3 in Step 3. Further, ζ_p is the p^{th} primitive root of unity. The function $v_q(t)$ denotes the multiplicity: $t = q^{v_q(t)}s$ for some integer s with $q \nmid s$. Refer to Figure 5.23 on the next page for the Haskell implementation.

The formulae for the Jacobi sums $J(p,q)$ (Step 7) are defined as follows. If $p \geq 3$ or both $p = 2$ and $k = v_p(q-1) = 2$, then

$$J(p,q) = j(\chi_{p,q}, \chi_{p,q}) = \sum_{x=1}^{q-2} \zeta_{p^k}^{x+f(x)}$$

```
nu q t | (r == 0)  = 1 + nu k q
       | otherwise = 0
  where (k, r) = t `divMod` q
```

Figure 5.23: *Computing multiplicity in Haskell.*

N	t
4292870400	12
2^{101}	180
2^{152}	720
2^{204}	1260
2^{268}	2520
2^{344}	5040
2^{525}	27720
2^{774}	98280
2^{1035}	166320
2^{1566}	720720
2^{2082}	1663200
2^{3491}	8648640

Table 5.5: *Values of t for Algorithm 3.*

holds. The other cases are

$$J(3,q) = j_3(\chi_{2,q},\chi_{2,q},\chi_{2,q}) = J(2,q)(\chi_{2,q}^2,\chi_{2,q}) = J(2,q)\sum_{x=1}^{q-2}\zeta_{2^k}^{2x+f(x)},$$

$$J(2,q) = j^2(\chi_{2,q}^{2^{k-3}},\chi_{2,q}^{2^{k-3}}) = \left(\sum_{x=1}^{q-2}\zeta_8^{3x+f(x)}\right)^2.$$

5.5.3 Main Algorithm

Theory. The key idea of the algorithm is to check the following condition. It is [Cohen, 2000, Proposition 9.1.7], see there for the detailed proof.

Proposition 5.16 (Generalised Fermat). *Let $\beta \in \mathbb{Z}[G]$. If N is prime, then there exists $\eta(\chi) \in \langle\zeta_n\rangle$ such that*

$$\tau(\chi)^{\beta(N-\sigma_N)} \equiv \eta(\chi)^{-\beta N} \pmod N \tag{5.1}$$

for $\sigma_N \in G$. We understand $\eta(\chi)$ as $\chi(N)$. The congruences modulo N are in fact modulo $N\mathbb{Z}[\zeta_n,\zeta_q]$. We consider $\mathbb{Z}[G]$ acting not only on $\mathbb{Q}(\zeta_n)$, but also on $\mathbb{Q}(\zeta_n,\zeta_q)$.

Idea of the Proof. With a generalisation of Lemma 5.2 holds $(\sum a_k)^N \equiv \sum a_k^N \pmod N$: all inner coefficients are divisible by N. Then we can write modulo N

$$\tau(\chi)^N = \sum_x \tau(x)^N\zeta_q^{Nx} = \sum_x \tau(N^{-1}x)^N\zeta_q^x = \chi(N)^{-N}\tau(\chi^N). \tag{5.2}$$

The required result follows from raising (5.2) to the power β. See also [Cohen and Lenstra, 1984] for details. †

Algorithms 4, 5, 6 and 7 test for (5.1) in various circumstances. Theorem 5.18 states that with that requirement and some further checks we can ensure primality. Following [Cohen, 2000, Definition 9.1.9] we can define the l_p set condition.

Definition 5.17 (l_p set condition)**.** We say that the l_p set condition w. r. t. N is satisfied, if for all prime divisors r of N and all integers $a > 0$ such integer $l_p(r, a)$ exists, that

$$r^{p-1} \equiv N^{(p-1)l_p(r,a)} \pmod{p^a}$$

holds.

Note that for prime N the l_p set condition holds trivially with $l_p(r, a) = 1$.

We introduce a new notation: $a \parallel b$ means that $a \mid b$, but $a \nmid (b/a)$. The symbol $c \perp d$ means that c and d have no common factors. In other words: $\gcd(c, d) = 1$. Of course $q - 1 \mid t$ could be written more strictly as $(q-1) \mid t$. The following central theorem is derived from [Cohen, 2000, Theorem 9.1.12 and Corollary 9.1.13]. Look there for the proof.

Theorem 5.18. *Let t be an integer, let*

$$e(t) = 2 \prod_{\substack{q \text{ prime} \\ q-1 \mid t}} q^{\nu_q(t)+1},$$

assume $N \perp t \cdot e(t)$. For every pair of primes (p, q) with $q - 1 \mid t$ and $p^k \parallel q - 1$, let $\chi_{p,q}$ be a character modulo q of order p^k. Assuming that

1. *For all pairs (p, q) per above, the character $\chi := \chi_{p,q}$ fulfils (5.1) for some $\beta \notin \bar{\mathfrak{p}}$*
2. *For all primes $p \mid t$ the l_p set condition is satisfied*

then for every divisor r of N there exists an integer j with $0 \le j \le t$ and

$$r \equiv N^j \pmod{e(t)}.$$

Further, let $r_j := N^j \mod e(t)$ with $0 < r_j < e(t)$ for all j. Such r_j build all possible divisors of N. If $e(t) > \sqrt{N}$ and if for each j with $0 < j < t$ none r_j divides N non-trivially, then N is prime.

A typical example of χ is $\chi_{p,q}(g_q^a) = \zeta_q^a$ if g_q is a primitive root modulo q. This is the connection between primitive roots modulo q and primitive roots of unity.

Example 5.19 (Trial division)**.** *Assume, $N = 23$. Then $t = 12$, $e(t) = 2 \cdot 2^3 \cdot 3^2 = 144$. It holds $N \perp t \cdot e(t)$. Assume further that both conditions 1 and 2 from the above theorem are satisfied. Possible divisors of N are then $0 < r_j < e(t)$ with $r_j \equiv N^j \pmod{e(t)}$. The possible values are 1, 23, 97, 71, 49, and 119. Up to 23 itself, all other r_j have no common factors with $N = 23$. Thus 23 is indeed prime.*

The value $\beta \in \mathbb{Z}[G]$ from condition 1 in Theorem 5.18 depends on p, as we will see below. A more deep insight into the Jacobi sum test is provided by the following statements.

Proposition 5.20 (Cohen [2000], Proposition 9.1.17)**.** *Assuming, we find a character χ modulo q of order p^k for which (5.1) is satisfied with $\eta(x)$ a* primitive $p^{k\,th}$ *root of unity. Then, given one of the following conditions is true, the l_p set condition is satisfied:*

- *If $p \ge 3$*
- *If $p = 2$, $k = 1$ and $N \equiv 1 \pmod 4$*
- *If $p = 2$, $k \ge 2$ and $q^{(N-1)/2} \equiv -1 \pmod N$*

The next proposition also originates from [Cohen, 2000]. Look there for the proof. The multiplication with σ_x^{-1} in the following statements signals that the computation happens in $\mathbb{Z}[G]$.

Proposition 5.21 (Cohen [2000], Proposition 9.1.18)**.** *Let χ be a character modulo q of order p^k and let a and b be integer numbers with $p \nmid ab(a+b)$. Let E be the set of all integers x, with $1 \le x < p^k$ and $p \nmid x$. Further, define*

$$\alpha = \sum_{x \in E} \left\lfloor \frac{Nx}{p^k} \right\rfloor \sigma_x^{-1}$$

and

$$\beta = -\sum_{x \in E} \left(\left\lfloor \frac{xa}{p^k} \right\rfloor + \left\lfloor \frac{xb}{p^k} \right\rfloor - \left\lfloor \frac{x(a+b)}{p^k} \right\rfloor \right) \sigma_x^{-1}.$$

Then

$$\tau(\chi)^{\beta(N-\sigma_N)} = j(\chi^a, \chi^b)^\alpha.$$

With this proposition, we can replace powering the Gauß sum $\tau(\chi)$ with powering the Jacobi sum $j(\chi^a, \chi^b)$. Then we work in a much smaller ring $\mathbb{Z}[\zeta_{p^k}]$. This result makes the test practical. However, we have excluded the case $p = 2$ with the condition $p \nmid ab(a+b)$. Cohen [2000] shows in his statements 9.1.21–9.1.26 that there are ways to handle it. We see an almost direct application of these results in Algorithms 5–7.

Remark (Cohen [2000], Lemma 9.1.19). The above proposition is related to Lemma 5.2:

$$\beta \notin \tilde{\mathfrak{p}} \iff (a+b)^p \not\equiv a^p + b^p \pmod{p^2}.$$

Let us focus on the case of Algorithm 4. The following result is [Cohen, 2000, Proposition 9.1.20]. Look there for the proof. It is connected to the Wieferich congruence $2^p \not\equiv 2 \pmod{p^2}$ of the first case of the Fermat's Last Theorem. The only known solutions for $p < 4 \cdot 10^{12}$ are the two excluded values $p = 1093$ and $p = 3511$, *viz.* [Crandall et al., 1997].

Proposition 5.22. *For $3 \le p < 6 \cdot 10^9$, $p \ne 1093, 3511$ we can take $a = b = 1$. In other words,* (5.1) *is equivalent to*

$$j(\chi, \chi)^\alpha \equiv \eta(\chi)^{-cN} \pmod{N}$$

with α as above,

$$\beta = \sum_{\substack{p^k/2 < x < p^k \\ p \nmid x}} \sigma_x^{-1}$$

and

$$c = 2\frac{2^{(p-1)p^{k-1}} - 1}{p^k}.$$

Theorem 5.18 provides us with the the main part of Jacobi sum test. The preliminary part of the test, independent from N, was stated in Section 5.5.2 as Algorithm 3, see page 72. We state the main part of the Jacobi sum test as Algorithm 8 on page 78, with the checks of (p, q) pairs in Algorithms 4, 5, 6 and 7. The input number N should have passed the Rabin–Miller test, i. e., Algorithm 1 of Section 5.4. We discuss the implementation details of the Jacobi sum test next. The parallelisation is discussed in Section 5.5.5 on page 79.

5.5.4 Implementation

Representation of residues. We need an implementation of a single-residue integer arithmetic, as one seen in Chapter 7. However, the main focus here lies not in the possibilities for the parallelism, but in the fast powering algorithm. The current implementation uses high-level Haskell code[3], corresponding to the right-left binary powering algorithm [Cohen, 2000, Algorithm 1.2.1], see Figure 5.24. There are more complicated powering algorithms, like the left-right base 2^k algorithm [Cohen, 2000, Algorithm 1.2.4]. These should be faster.

[3] A Haskell binding to the low-level GNU MP code for residual powering algorithms has been suggested, but is not yet incorporated in GHC, see http://hackage.haskell.org/trac/igtest/ticket/3489.

Algorithm 4 Jacobi sum test. Case 1: check (5.1) for $p \geq 3$.

Require: A pair (p, q), the power k of p, a pseudo-prime N, the value l_p.

1: Let $E \leftarrow \{n : n \in \{0, \ldots, p^k\}$ with $p \nmid n\}$.

2: Let

$$\Theta \leftarrow \sum_{x \in E} x\sigma_x^{-1} \quad (\in \mathbb{Z}[G]).$$

Let $r \leftarrow N \pmod{p^k}$ and

$$\alpha \leftarrow \sum_{x \in E} \left\lfloor \frac{rx}{p^k} \right\rfloor \sigma_x^{-1} \quad (\in \mathbb{Z}[G])$$

3: Compute $s_1 \leftarrow J(p, q)^{\Theta} \pmod{N}$, and $s_2 \leftarrow s_1^{\lfloor N/p^k \rfloor} \pmod{N}$.

4: Let

$$S(p, q) \leftarrow s_2 J(p, q)^{\alpha} \pmod{N}.$$

5: **if** no $p^{k\text{th}}$ root of unity η exists such that $S(p, q) \equiv \eta \pmod{N}$ **then**

6: **return** 'N is composite' and terminate the main algorithm.

7: **else if** η is a primitive $p^{k\text{th}}$ root of unity **then** set $l_p \leftarrow 1$

8: **end if**

9: **return** l_p

Ensure: Either a signal for the termination of the main algorithm with 'N is composite' or a possibly new value l_p.

```
powermod :: Integer → Integer → Integer → Integer
powermod b e m -- produces b^e mod m
  | (e==0)           = 1
  | (e `mod` 2 == 0) = (temp * temp) `mod` m
  | otherwise        = b * (powermod b (e-1) m) `mod` m
 where temp = (powermod b (e `div` 2) m)
```

Figure 5.24: *The powering algorithm for residue classes.*

```
type Poly a = [a]

unshuffle :: Integer → [a] → [[a]] -- see Chapter 3

easysimp :: Poly Integer -- ^ polynomial to simplify
         → Integer       -- ^ the order of the root of unity
         → Poly Integer  -- ^ simplified polynomial
easysimp l n = foldr (+) [] $ unshuffle n l
```

Figure 5.25: *Simplifying polynomials Φ_n, the easy way.*

Optimisation. The profiling of the sequential version with GHC 6.12.3 showed for the input $2^{607} - 1$ that 99% of time is spent in the main loop of the algorithm (*viz.* Figure 5.26). Further, pretty much time was used by the implementations of the powering algorithm and of the polynomial arithmetic. To be more specific: the execution time of five top cost centres was 1 452 ticks. In order to reduce the total program execution time, especially in the mentioned above part, an implementation of univariate polynomials was optimised. The key optimisation was to replace primitive recursions with the wider usage of higher-order functions. As the result, the same functions require only 696 ticks to compute. We benefited from the optimised implementation of list processing functions functions in GHC, see, e.g., [Coutts et al., 2007]. We conducted all our measurements with the optimised implementation.

Algorithm 5 Jacobi sum test. Case 2: check (5.1) for $p = 2$ and $k \geq 3$

Require: A pair (p, q), the power k of p, a pseudo-prime N, the value l_p.

1: Let $E \leftarrow \{n : n \in \{0 \dots 2^k\}$ with $n = 1 \pmod 8$ or $n = 3 \pmod 8\}$.

2: Let

$$\Theta \leftarrow \sum_{x \in E} x\sigma_x^{-1} \quad (\in \mathbb{Z}[G])$$

Let $r \leftarrow N \pmod{2^k}$ and

$$\alpha \leftarrow \sum_{x \in E} \left\lfloor \frac{rx}{2^k} \right\rfloor \sigma_x^{-1} \quad (\in \mathbb{Z}[G])$$

3: **if** $r \in E$ **then** $\delta_N \leftarrow 0$

4: **else** $\delta_N \leftarrow 1$.

5: **end if**

6: Compute $s_1 \leftarrow J_3(p, q)^{\Theta} \pmod N$ and $s_2 \leftarrow s_1^{\lfloor N/p^k \rfloor} \pmod N$.

7: Let

$$S(2, q) \leftarrow s_2 J_3(q)^{\alpha} J_2(q)^{\delta_N} \pmod N.$$

8: **if** no 2^kth root of unity η exists such that $S(2, q) \equiv \eta \pmod N$ **then**

9: **return** 'N is composite' and terminate the main algorithm.

10: **else if** η is a primitive p^kth root of unity **and** $q^{(N-2)/2} = -1 \ (N)$ **then** $l_2 \leftarrow 1$

11: **end if**

12: **return** l_2

Ensure: Either a signal for the termination of the main algorithm with 'N is composite' or a possibly new value l_p.

Algorithm 6 Jacobi sum test. Case 3: check (5.1) for $p = 2$ and $k = 2$

Require: A pair (p, q), the power k of p, a pseudo-prime N, the value l_p.

1: Let $s_1 \leftarrow J(2, q)^2 \cdot q \pmod N$ and $s_2 \leftarrow s_1^{\lfloor N/4 \rfloor} \pmod N$.

2: **if** $N = 1 \pmod 4$ **then** set $S(2, q) \leftarrow s_2$

3: **else if** $N = 3 \pmod 4$ **then** set $S(2, q) \leftarrow s_2 J(2, q)^2$

4: **end if**

5: **if** no fourth root of unity η exists with $S(2, q) \equiv \eta \pmod N$ **then**

6: **return** 'N is composite' and terminate the main algorithm.

7: **else if** such η exists **and** $q^{(N-1)/2} \equiv -1 \pmod N$ **then** set $l_2 \leftarrow 1$.

8: **end if**

9: **return** l_2

Ensure: Either a signal for the termination of the main algorithm with 'N is composite' or a possibly new value l_p.

Algorithm 7 Jacobi sum test. Case 4: check (5.1) for $p = 2$ and $k = 1$.

Require: A pair (p, q), the power k of p, a pseudo-prime N, the value l_p.

1: Let $S(2, q) \leftarrow (-q)^{(N-1)/2} \pmod N$.

2: **if** $S(2, q) \not\equiv \pm 1 \pmod N$ **then**

3: **return** 'N is composite' and terminate the main algorithm.

4: **end if**

5: **if** $S(2, q) \equiv -1 \pmod N$ **and** $N \equiv 1 \pmod 4$ **then** let $l_2 \leftarrow 1$.

6: **end if**

7: **return** l_2

Ensure: Either signal for the termination of the main algorithm with 'N is composite', or a possibly new value l_p.

Algorithm 8 Jacobi sum test. Main algorithm.

Require: A strong pseudo-prime N. It holds $N \leq B$.

procedure CHOOSE HELPER(p, q, k) // *For these parameters holds* $p^k \parallel (q-1) \mid t$
  if $p \geq 3$ **then** execute Algorithm 4, eventually updating the l_p value.
  else if $p = 2$ **and** $k \geq 3$ **then** execute Algorithm 5, eventually updating the l_p value.
  else if $p = 2$ **and** $k = 2$ **then** execute Algorithm 6, eventually updating the l_p value.
  else if $p = 2$ **and** $k = 1$ **then** execute Algorithm 7, eventually updating the l_p value.
  end if
  return l_p.
end procedure
procedure MAIN ALGORITHM(N, B)
  if $\gcd(t \cdot e(t), N) > 1$ **then return** 'N is composite'.
  end if
  for every prime $p \mid t$ **do**
    if $p \geq 3$ **and** $N^{p-1} \neq 1 \pmod{p^2}$ **then** $l_p \leftarrow 1$
    else $l_p \leftarrow 0$
    end if
  end for
  for all pairs of primes (p, q) such that $p^k \parallel (q-1) \mid t$ **do** // *Loop to parallelise*
    $l_p \leftarrow$CHOOSE HELPER(p, q, k)
  end for // *Either we have terminated via one of the helper algorithms with 'N is composite' or we have a possibly updated list of* l_p *values for all* p.
  for all $p \mid t$ with $l_p = 0$ **do**
    Choose a random prime q with $q \neq e(t)$, **and** $q \equiv 1 \pmod{p}$, **and** $q \perp N$.
    $l_p \leftarrow$CHOOSE HELPER(p, q, k) // *We will have to compute some new Jacobi sums, Algorithm 3 tells us how. In each step we obtain possibly updated list elements* l_p.
    if after 30 attempts some l_p are still zero **then return** 'failed' // *improbable!*
    end if
  end for
  for $i \in \{1, \ldots, t-1\}$ **do**
    compute by induction $r_i \leftarrow N^i \pmod{e(t)}$.
    if $r_i \mid N$ **and** $r_i \neq 1$ **and** $r_i \neq N$ **then return** 'N is composite'
    end if
  end for
  return 'N is prime'
end procedure

Ensure: Either 'N is composite' or 'N is prime' or 'failed'.

Representation of cyclotomic fields. Further, we need to represent an n^{th} cyclotomic field on a computer. The straightforward design decision was to represent $\mathbb{Q}(\zeta_n)$ *symbolically* as $\mathbb{Q}[x]/\langle \Phi_n(x) \rangle$ with n^{th} cyclotomic polynomial $\Phi_n(x)$. We know the latter is the minimal polynomial in our case. The algorithmic optimisations concerning this representation were to

- Normalise ζ_n^n to 1 in each arithmetic operation.
- Reduce the result afterwards.

The current implementation represents (univariate) polynomials as dense lists. Experiments have shown, that the usage of sparse lists does not produce substantial difference. A simple approach for the simplification of the field extension polynomial is presented in Figure 5.25, the more advanced approach was also implemented, see Appendix.

```
jacobiSumTestMainLoop :: [(Integer, Integer)]
                         -- ^ (p, q) list, p^k || q-1 | t
                      -> Integer   -- ^ n, the prime candidate
                      -> Integer   -- ^ t from precomputation
                      -> [Integer] -- ^ list of primes with l_p=0
                      -> Maybe [Integer]
                         -- ^ the result: Nothing: not a prime
                         --   Just _: possibly updated list of primes with l_p=0
jacobiSumTestMainLoop [] n t lps = Just lps
jacobiSumTestMainLoop ((p, q):xs) n t lps
  | (res == Nothing) = Nothing
  | otherwise = jacobiSumTestMainLoop xs n t lps'
          where k = nu (q-1) p  -- multiplicity of p in (q-1)
                res = jacobiSumTestSelector p k q n lps
                lps' = fromJust res -- strip the Just constructor

jacobiSumTestSelector :: Integer -> Integer -> Integer
                         -- ^ p, k and q from p^k || q-1
                      -> Integer   -- ^ prime candidate n
                      -> [Integer] -- ^ list of primes with l_p=0
                      -> Maybe [Integer]
                         -- ^ the result: Nothing: not a prime
                         --   Just _: possibly updated list
                         --           of primes with l_p=0
jacobiSumTestSelector p k q n lps
                  | (p ≥ 3) = jacobiSumTestCase1 p k q n lps fps
                  | (k ≥ 3) = jacobiSumTestCase2 p k q n lps fps
                  | (k==2) = jacobiSumTestCase3 p k q n lps fps
                  | (k==1) = jacobiSumTestCase4 p k q n lps fps
                  | otherwise = error "failed to select the right case"
                                where fps = fpairlist q
```

Figure 5.26: *Jacobi sum: Main loop. This is the sequential version of the Jacobi sum test. We show only the central part of Algorithm 8 (main loop in Step 17 and decision for the choice of the helper algorithm).*

Return value. The main loop of the test is in Figure 5.26. The remaining parts of the test's implementation are not very interesting—with one exception. The simplest approach is to return a `Bool` value for 'is prime' or 'not a prime', and a runtime error for the failure of the test. But it makes more sense to return `Maybe Integer`. Then the interface of the top-level function is

```
jacobi :: Integer → Maybe Integer
```

Then we can use a monadic approach, detailed in Section 5.6.

5.5.5 Parallelisation

Implementation. If we carefully inspect Algorithms 4–7, we see an important fact.

Theorem 5.23. *Algorithms 4–7 can be called for different entries of the list of* (p, q) *pairs simultaneously.*

Proof. We need to show that one call of each of the aforementioned algorithms does not depend on the result of any another one, and that these algorithms do not mutually destroy their result. It suffices to observe that neither of Algorithms 4–7 actually needs the entries in the list of l_p values, except for the purpose of possibly changing a l_p value. The l_p list is later used in step 20 of Algorithm 8. So, the behaviour of these algorithms does not change, if we call each of them as appropriate with initial value

```
unifyMaybe :: (Eq a) ⇒ [Maybe [a]] → Maybe [a]
unifyMaybe xss | any isNothing xss = Nothing
               | otherwise = Just $ unify $ catMaybes xss

jacobiSumTestMainLoop
  :: Map a b             -- ^ parallel map instance to use
  → [(Integer,Integer)] -- ^ (p, q) list, p^k || q-1 | t
  → Integer              -- ^ n, the prime candidate
  → Integer              -- ^ t from precomputation
  → [Integer]            -- ^ list of primes with l_p = 0
  → Maybe [Integer]      -- ^ the result:
  --   Nothing: not a prime
  --   Just _: possibly updated list of primes with l_p = 0
jacobiSumTestMainLoop amap xs n t lps
  = let worker (p, q) = jacobiSumTestSelector p k q n lps
          where k = nu (q-1) p -- multiplicity of p
    in map+reduce' (amap worker) unifyMaybe xs
```

Figure 5.27: *Parallel implementation of Jacobi sum test. We use* `farm` *or* `workpool` *as amap.*

short name	properties		implementation
workpool A	sorting	blocking	`workpoolSorted`
workpool B	sorting	non-blocking	`workpoolSortedNonBlock`
workpool C	non-sorting		`workpool'`

Table 5.6: *Naming of workpools in this chapter.*

$l_p = 0$ for all p in question. We merely need to *merge* the list of lists of l_p to the single list of l_p for the further processing. □

This means for us, that we can execute the step 17 of Algorithm 8 in parallel. The calls to Algorithms 4–7 will provide us with their versions of the l_p list. All we need to do then is to `reduce` the list of the l_p lists to a single list. We can do it in the following way:

```
unify :: (Eq a) ⇒ [[a]] → [a]
unify = foldl1 intersect ∘ nub
```

The actual function demands a bit more care, as it processes the lists of type `[Maybe [a]]` and should stop consuming the list, if a `Nothing` occurs. Further, one needs to handle empty lists correctly. A full implementation of the `reduce` function, called `unifyMaybe`, is shown in Figure 5.27, top. Now we can replace the main loop of the implementation, i.e., the function `jacobiSumTestMainLoop` in Figure 5.26, with the code in Figure 5.27, bottom. The resulting code runs in parallel. This code uses the `map+reduce` skeleton scheme, shown in Figure 5.12 on page 59.

Note that we are able to change the order of the result list in the implementation of `unifyMaybe`. This allows us to conclude the following result.

Corollary 5.24. *The order of evaluation of separate calls to Algorithms 4–7 is irrelevant, as long as the correct order of the resulting l_p list, i.e., the order of list elements, expected in the remaining program, is restored after the calls to helper algorithms.*

Performance. Let us observe the performance of our parallel code. We take Mersenne primes as input, meaning $N = 2^k - 1$. The time is stated in seconds. As always (*cf.* methodology in Section 3.4) each program is run five times, then an average is built. We performed some experiments on `sakania`,

n	skeleton	PE 1	2	3	4	5	6	7	8
512	farm	3.82	1.98	1.51	1.08	0.99	0.82	0.79	0.71
	workpool A	3.82	2.00	1.36	1.12	0.94	0.83	0.72	0.69
	workpool B	3.82	1.99	1.37	1.12	0.95	0.82	0.73	0.70
	workpool C	3.82	2.00	1.37	1.11	0.93	0.83	0.74	0.71
607	farm	16.0	8.53	8.11	4.72	4.25	4.76	3.83	3.29
	workpool A	16.0	8.49	5.97	4.72	4.05	3.70	3.37	3.14
	workpool B	16.0	8.41	6.00	4.72	4.05	3.72	3.47	3.31
	workpool C	16.1	8.30	6.04	4.70	4.06	3.78	3.32	3.14

Table 5.7: *Time measurement for Jacobi sum test on* `sakania`*. This is the version with original task order. The value n in the table corresponds to the input value* $2^n - 1$*. Workpool A is* `workpoolSorted`*, workpool B is* `workpoolSortedNonBlock`*, workpool C is a non-sorting* `workpool`*'. The latter is explained on page 83.*

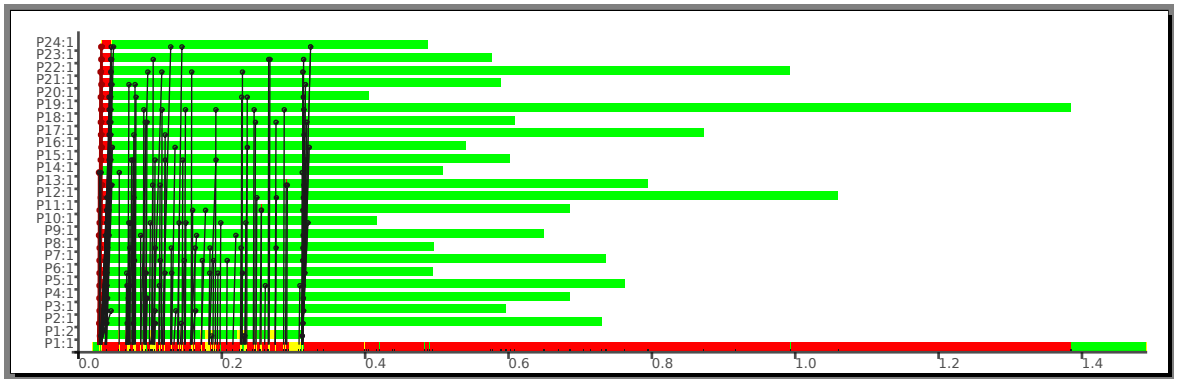


Figure 5.28: *Parallel Jacobi sum trace diagram with workpool A on 24 PE. Input is* $2^{607} - 1$*. Note the task granularity. Arrows indicate messages from the master.*

we used all eight cores for worker processes. We used no memory tuning. In the remaining part of this chapter we abbreviate the names of different workpool implementations as shown in Table 5.6. We use workpool A and workpool B. Our starting point are figures in Table 5.7.

How good is the load balancing? To answer this question we perform some measurements on local workstations. In Figure 5.28 we see a trace diagram of our initial parallelisation of Jacobi sum test. The input was $2^{607} - 1$, a Mersenne prime. The executable, which has produced the visualised trace, was run on local workstations. We used 24 PE, this corresponds to twelve physical dual-core machines. We observe that despite using dynamic task balancing with various sorting workpool implementations, we still have an issue with unbalanced tasks near the end of the computation. The small sequential phase at the end of the diagram represents the remaining steps of the algorithm. In the showed case, none of the l_p values were zero in Step 20 of Algorithm 8 on page 78. The inspection of messages in the same diagram reveals that the last tasks are issued simultaneously, just some tasks take much longer time than other tasks. We see this at 0.3 seconds: PE 19 and PE 20 obtain one task each. PE 19 is busy until 1.4 second, PE 20 is done after 0.1 seconds of work. However, we see that earlier tasks have this problem to a less extent. Further, it seems that some small tasks are issued first, see PE 2 and first tasks on PE 19. We discuss this issue below.

We investigate into the task size for a typical parallel execution. To do so, we execute the program with input $2^{607} - 1$ on as many virtual PEs, as tasks are available. In this particular case that were 145 PE, which were executed on 12 dual-core machines, i.e., on 24 real processors. The initial setting is depicted in Figure 5.29, left. The horizontal axis shows the precedence of the tasks, left are the first tasks, right are the last ones. In a typical workpool setting there are more tasks than PEs, the tasks

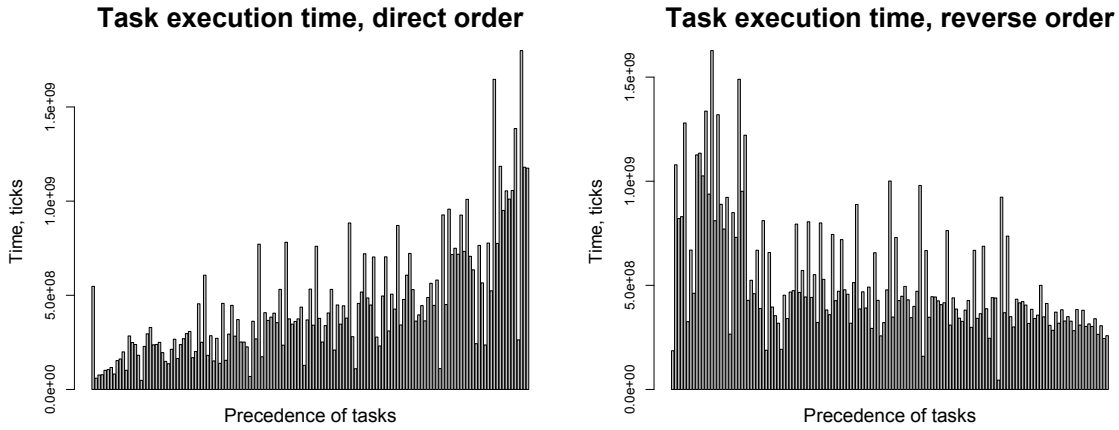


Figure 5.29: *Distribution of tasks in a Jacobi sum program. Left: without a task pool transformation. Right: with said transformation.*

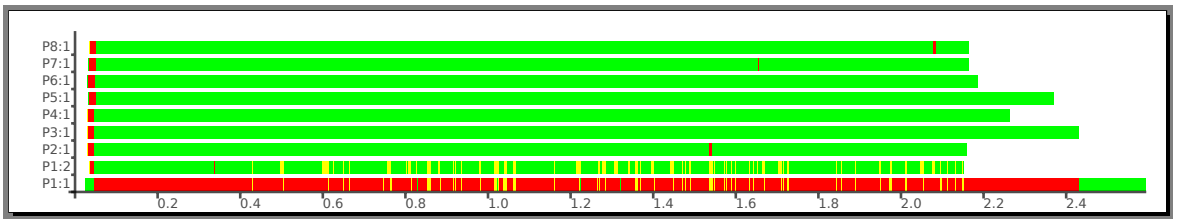


Figure 5.30: *Trace diagram for Jacobi sum test on* `sakania`*, input* $2^{607}-1$*, reversed task order, workpool C.*

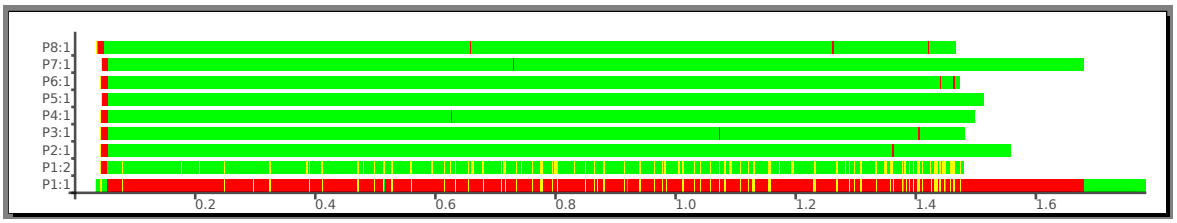


Figure 5.31: *Trace diagram for Jacobi sum test on 8 local workstations, input* $2^{607}-1$*, reversed task order, workpool C.*

on the left will be consumed first. Only when these are computed, the next tasks will be issued. The vertical axis shows the time to compute each task, measured in internal ticks of the Haskell runtime system. We see that many 'heavy', large tasks appear at the right hand side of said diagram. These tasks will be computed last. As all other tasks are already computed, bad load balancing ensues. The reason for what we see in the trace visualisations, e. g., in Figure 5.28, is relatively small amount of remaining tasks and the no possibility for the skeleton to know beforehand how hard each task is. We call the collection of all tasks a *task pool*.

However, as single tasks are independent from each other (see Theorem 5.23), Corollary 5.24 tells us, we can permute the tasks in the task pool in any way suitable for us. The aim is to start as many large tasks as early as possible. To achieve this we reverse the list of the tasks (i. e., the order of the (p, q) pairs) and reverse the precedence of the l_p list. This action, applied to our test program with input size $2^{607}-1$, results in the order of the task pool depicted in Figure 5.29, right. The axes and units are the same as before. We see that most of the 'hard' time-consuming tasks move to the left part of the plot. This means that these tasks are computed before the tasks to the right. Hence, a better load balancing is to be assumed. We support our claim by inspecting the traces of the test executions of the two approaches in question. The time measurements for this approach are presented in Table 5.8.

n	skeleton	PE							
		1	2	3	4	5	6	7	8
512	farm	3.75	1.95	1.48	1.06	0.99	0.81	0.78	0.71
	workpool A	3.76	1.92	1.34	1.05	0.89	0.78	0.69	0.63
	workpool B	3.77	1.93	1.33	1.05	0.88	0.77	0.69	0.64
	workpool C	3.75	1.93	1.34	1.05	0.88	0.78	0.69	0.62
607	farm	15.51	8.32	7.89	4.56	4.07	4.50	3.68	3.22
	workpool A	15.58	7.97	5.50	4.29	3.71	3.22	2.87	2.64
	workpool B	15.58	7.98	5.47	4.23	3.63	3.16	2.91	2.58
	workpool C	15.57	7.99	5.45	4.19	3.61	3.14	2.84	2.68

Table 5.8: *Time measurement for Jacobi sum test on* `sakania`*. This is the version with reversed task order. The value n in the table corresponds to the input value* $2^n - 1$*. Workpool A is* `workpoolSorted`*, workpool B is* `workpoolSortedNonBlock`*, workpool C is non-sorting* `workpool'`*.*

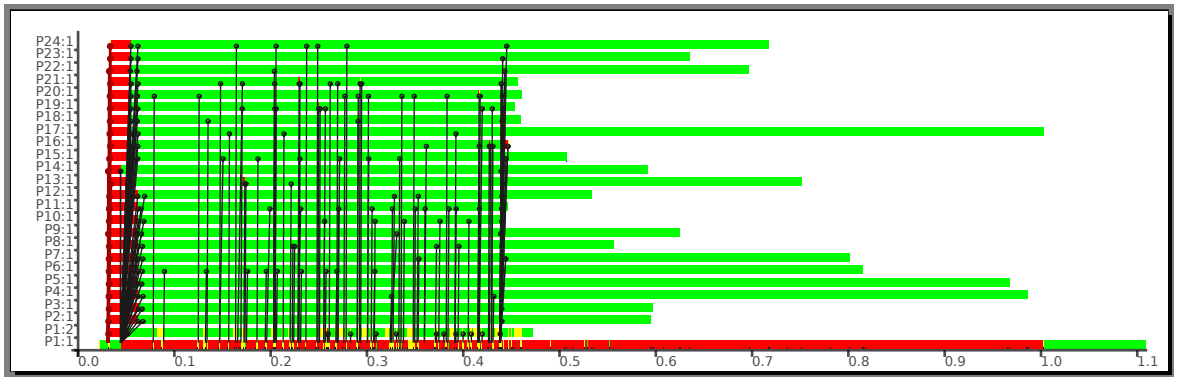


Figure 5.32: *Trace diagram for Jacobi sum test with reversed task order on 24 local workstations, input* $2^{607} - 1$*, reversed task order, workpool C. Arrows indicate messages from the master.*

A further optimisation originates from the same observation. As we maintain the order of the result elements manually, we do not need a sorting workpool. Workpool C in Table 5.8 is a non-sorting workpool implementation, the `workpool'` to be more precise. The order of the elements is secured by sequential sorting in the post-processing phase after the `reduce`. The length of the final, combined result list is at most the total length of all the individual result lists from single tasks. Hence, such sorting is more efficient in a combination with a non-sorting workpool than a direct implementation of a sorting workpool. To facilitate a better comparison we also performed measurements of the program execution time with original task order and non-sorting workpool with a sequential sort afterwards. These results are stated in Table 5.7 on page 81 in the lines labelled 'workpool C'.

Let us call the unaltered task pool program an 'original' one, while the program with a modified task pool will be a 'reversed' one. The trace diagrams for the both cases are presented in Figure 5.28 on page 81 for the original and in Figures 5.30, 5.31 and 5.32 for the reversed task pool. The message arrows in the latter image indicate that most of larger tasks are performed at the beginning of the computation. The last issued tasks (at 0.43 seconds) are issued to the workers simultaneously. Just some workers (like PEs 4 and 17) take much more time for the last task than other workers (like PE 2 and 10). Our task pool transformation resulted in the trace diagram shown Figure 5.32 as opposed to Figure 5.28 for an unaltered task pool.

The best speedup on `sakania`, at eight PE, is 5.00 for the original version with workpool A and $2^{521} - 1$ as input. It is 5.54 for the reversed version with the same input and workpool C. However, the best speedup for the reversed version with larger input $2^{607} - 1$ is 5.17, now for workpool B. For even larger input $2^{1279} - 1$ the best workpool is 'A' with speedup 5.78 on eight PE for the reversed version.

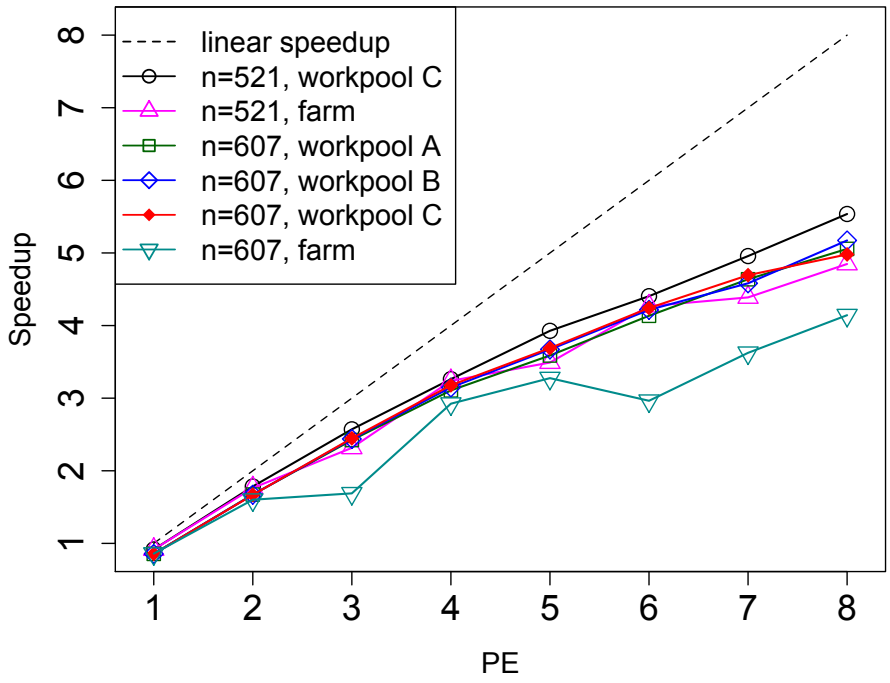


Figure 5.33: *Absolute speedup for Jacobi sum test. We use the reversed implementation for our measurements. The two inputs were* $2^{521} - 1$ *and* $2^{607} - 1$*. The tests were performed on* `sakania`*.*

We show the execution times of various program versions at multiple inputs in Tables 5.7 and 5.8, the speedup curves for different workpool implementations of the better, reversed case is in Figure 5.33. The latter figure shows clearly that we do not need to consider a `farm`: static task balancing without information on task size is not of a benefit. For comparison, we visualised in Figure 5.31 a trace of workpool C with input $2^{607} - 1$ on eight local workstations. The corresponding image for `sakania` is in Figure 5.30.

We see a better performance of the implementation with reversed task pool. The maximal absolute speedup achieved was 5.78 for the input $2^{1279} - 1$ on eight PE. An optimisation of the polynomial arithmetic (see page 76, included in both sequential and parallel versions regarded here) and the task pool optimisations from this section enabled a satisfying parallelisation of Jacobi sum test.

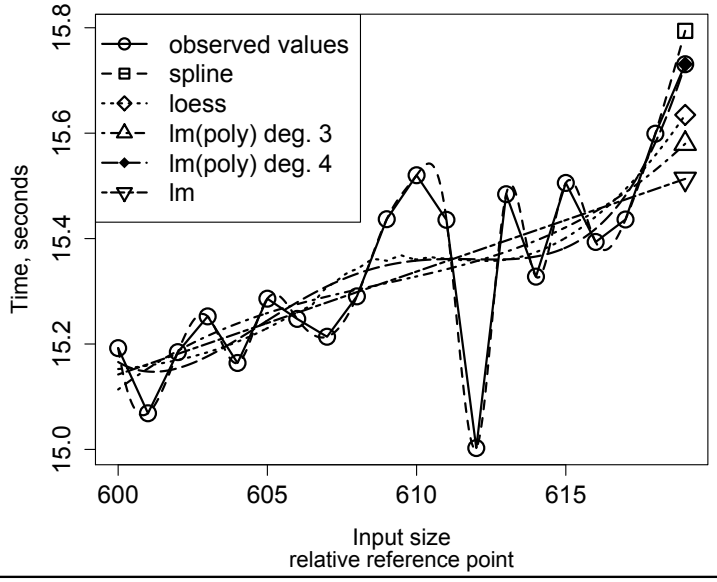
Estimating the execution time. For the estimation of the run time of Jacobi sum test, we used primes of size $\approx 2^{600}$–2^{619} as the input. We designate the exponents as the input sizes, i. e., they range from 600 to 619. The aim is to estimate the execution time for the input size 619 from smaller inputs. We use the relative speedup and, correspondingly, relative reference point. The runtimes, used to compute the quality measures, are the mean times of five consecutive program runs. This approach should have captured the general trend. We need the averaging because of the non-determinism of the workpool, a particular single program execution might behave in a different manner.

The choice of the input has three reasons. Firstly, our standard inputs, the Mersenne primes in the suitable range from $2^{521} - 1$ to $2^{1279} - 1$, are too few and too far apart to obtain a reliable input size to time relation. Secondly, we wanted to test our prediction methods on non-monotonously increasing data. Thirdly, we wanted to perform Jacobi sum test on 'generic' primes, as Mersenne primes are of a special form.

We were able to predict the execution time w. r. t. input size quite accurately. We show the initial data in Table 5.9. Note that the increasing input size does not always results in increased execution

n	600	601	602	603	604	605	606	607	608	609
$T(n,1)$	15.19	15.07	15.18	15.25	15.16	15.27	15.25	15.21	15.29	15.44
$T(n,8)$	2.59	2.58	2.66	2.62	2.68	2.60	2.55	2.59	2.58	2.62
n	610	611	612	613	614	615	616	617	618	619
$T(n,1)$	15.52	15.44	15.00	15.48	15.33	15.51	15.39	15.44	15.60	15.73
$T(n,8)$	2.76	2.66	2.53	2.71	2.59	2.65	2.59	2.69	2.64	2.78

Table 5.9: *Timings for Jacobi sum test.*



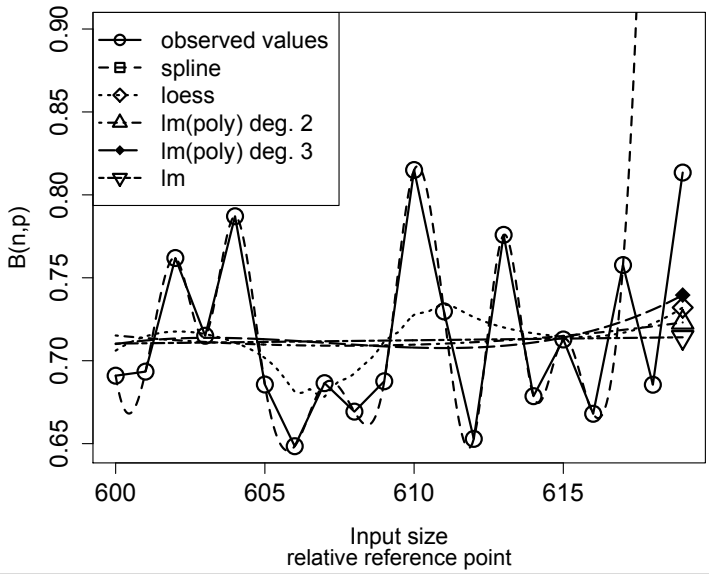
Method	spline	loess	lm(poly) of degree 2	3	4	lm
Rel. err., %	0.40	−0.61	−1.35	−0.959	$4.3298 \cdot 10^{-5}$	−1.383

Figure 5.34: *Predicting sequential execution time of Jacobi sum test.*

time. The plots of our prediction approaches are in Figures 5.34 and 5.35. We see, that we were able to predict the sequential time very well. Indeed, the relative error for the lm(poly) method with orthogonal polynomials of degree 4 was merely $4 \cdot 10^{-5}$%. The spline and loess methods were also quite good, with 0.40% and −0.61% relative error appropriately. On the other hand, the estimation of the parallel overhead w.r.t. n was disappointing, as we obtained the relative error of −9.08% with lm(poly) of degree 3. To do so, we dismissed the value at $n = 618$. Still, with both predicted values for $T(n,1)$ and $B(n,p)$ combined, we obtain the time for $T(619,8)$ up to −2.66% relative error.

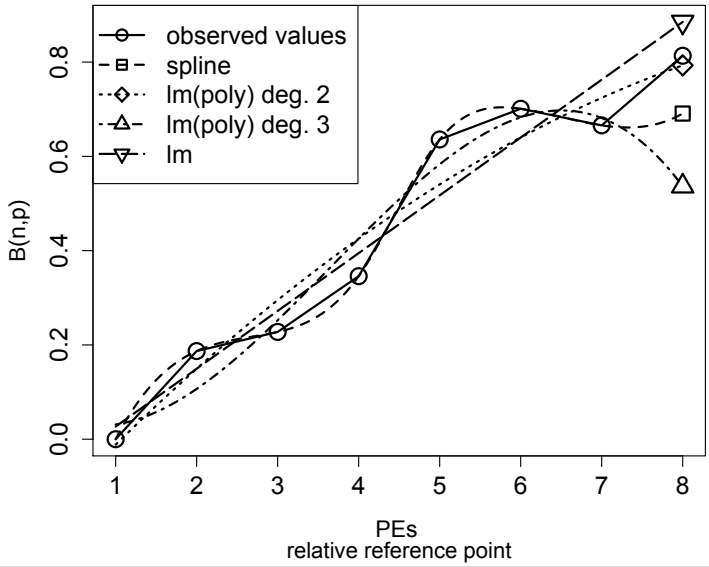
Even more interesting is the prediction of $B(n,p)$ w.r.t. p. We assume the values on $p \leq 7$ as given. The values for $n = 619$ are used. The estimation is not easy, but we have obtained the value 0.79 with lm(poly) of degree 2, *viz.* Figure 5.36. This corresponds with the measured value for $B(619,8)$ up to −2.5% relative error. Using it for the estimation of $T(619,8)$ in a combination with a given $T(619,1)$, we obtain the execution time up to −0.73% relative error.

Parallel overhead and serial fraction. We evaluate the plots of parallel quality measures for our Jacobi sum test implementation. We used the time measurements, obtained on `sakania` for the Mersenne prime $2^{607} - 1$ with reversed task pool. Figure 5.37 shows both plots, with parallel penalty $B(n,p)$ on the left and serial fraction $f(n,p)$ on the right. We show the values for the `farm` and for three `workpool` implementations, abbreviated 'wp' in the figures. The handles of these implementations originate from Table 5.6. We use the absolute reference point.



Method	spline	loess	lm(poly) deg. 2	lm(poly) deg. 3	lm
Rel. err., %	183	−10.01	−11.11	−9.081	−12.21

Figure 5.35: *Predicting parallel overhead w.r.t. n of Jacobi sum test.*



Method	spline	lm(poly) deg. 2	lm(poly) deg. 3	lm
Rel. err., %	−15.07	−2.508	−34.04	8.823

Figure 5.36: *Jacobi sum test. Estimating $B(n, p)$ w.r.t. p.*

Both plots are decreasing in the interval from PE 1 to PE 4. This would signal poor performance of the sequential version. However, from PE 4 to PE 8 both measures do not show a significant decrease. The occasional 'waves' in both plots from 4 to 8 PE indicate slightly better and worse task distributions across PEs. We have addressed this issue in previous sections. We see that neither workpool implementation can be called a clear winner: the plots of both quality functions for the workpool implementations are quite interleaved at 6–8 PE. We conclude that the final grain of the better performance of a particular workpool for a particular number of PEs depends of the task distribution in the particular case. However, the task distribution is of course runtime-dependent. We find all three workpools similarly good, with a slight preference for one or another method for a particular number

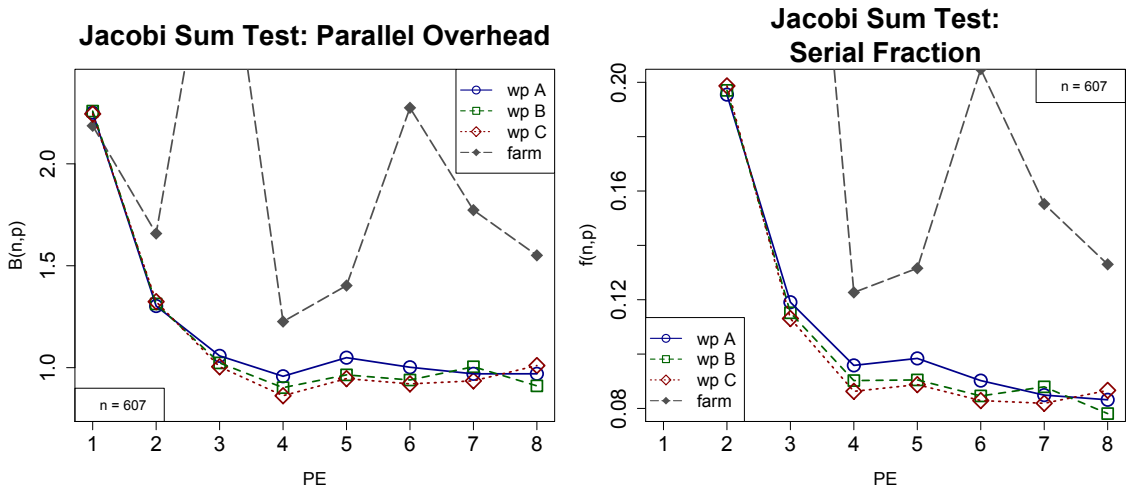


Figure 5.37: Parallel overhead (left) and serial fraction (right) for Jacobi sum test. The input is $2^{607} - 1$*, data originate from* `sakania` *(viz. Figure 5.33).*

of PEs.

As for 1–4 PE, we see the parallel overhead for `farm` falling from 1 to 2 PE, increasing at 3 PE, and falling again for 4 PE. The 'hill' at 3 PE can be explained with bad task distribution. Note that up to 1 PE the parallel penalty for the `farm` version is always larger than that for all workpools. A possible interpretation would be: the dynamic task management of a workpool creates more overhead at 1 PE than a more simple `farm` implementation. The large value of both performance measures at small number of PEs can be also explained with the absolute reference point: the pure sequential implementation does not need to manage all the tasks. Additionally, the sequential implementation does not need to sort and to reduce the results in the sequential version, as it implements the premature abort property differently.

As for the differences between parallel penalty and serial fraction, in this case they both are similar to other observations we made. On the one side, we can see the effects from above more clear in the serial fraction, the 'waves' have a larger amplitude in the right plot. On the other side, the mentioned above effect at 1–2 PE cannot be observed in serial fraction because it begins only at 2 PE.

Future work. The bottleneck for the better scaling of parallel Jacobi sum test is the size of the largest single task. Figures 5.28 and 5.32 show the trace diagrams of the original and reversed task pools on 24 PE with input size $2^{607} - 1$. It was measured on local workstations with twelve dual-core machines. We see clearly that with increasing number of PE larger tasks dominate the computation time. Ingenious splitting of current large tasks to multiple smaller ones could help. Further, we observe that with larger inputs more tasks are created, thus enabling the current approach on a larger scale. This claim is supported by the number of tasks, i. e., of (p, q) pairs. There are 145 of them for input $2^{607} - 1$, but their number rises to 287 already for input $2^{1279} - 1$ and to 560 for $2^{2203} - 1$. Hence, the size of the input is in a particular relation to the number of PEs, similarly to the scaled speedup or isoefficiency [Grama et al., 2003].

In other words: the increased number of tasks accounts for better task balancing on larger number of PE. To give an example we show trace visualisations for Jacobi sum test with input $2^{1270} - 1$ on twenty-four (*cf.* Figure 5.38, see also Figure 5.32 for input $2^{607} - 1$ on the same number of PE) and forty local workstations (Figure 5.39). In the latter program run we used not only Intel Core2Duo machines, but also other available machines, including older Intel Pentium D dual-cores, a newer Intel i5 machine and three various Intel Xeon machines, including the eight core `sakania`. Still we were careful not to use more (virtual) PE than cores available. Dynamic workpool implementations were quite able to manage the different computation times for the tasks of the same size. The imbalances, discussed here, originate from different task sizes, not from bad processor performance on different PEs. The imbalance was

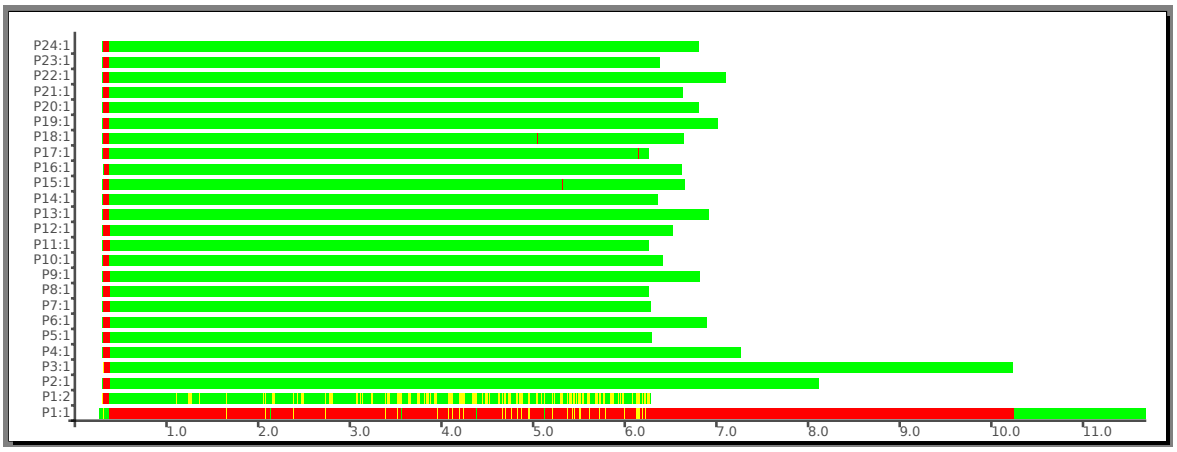


Figure 5.38: *Trace for Jacobi sum test on local workstations, input size* $2^{1279} - 1$*, 24 PE.*

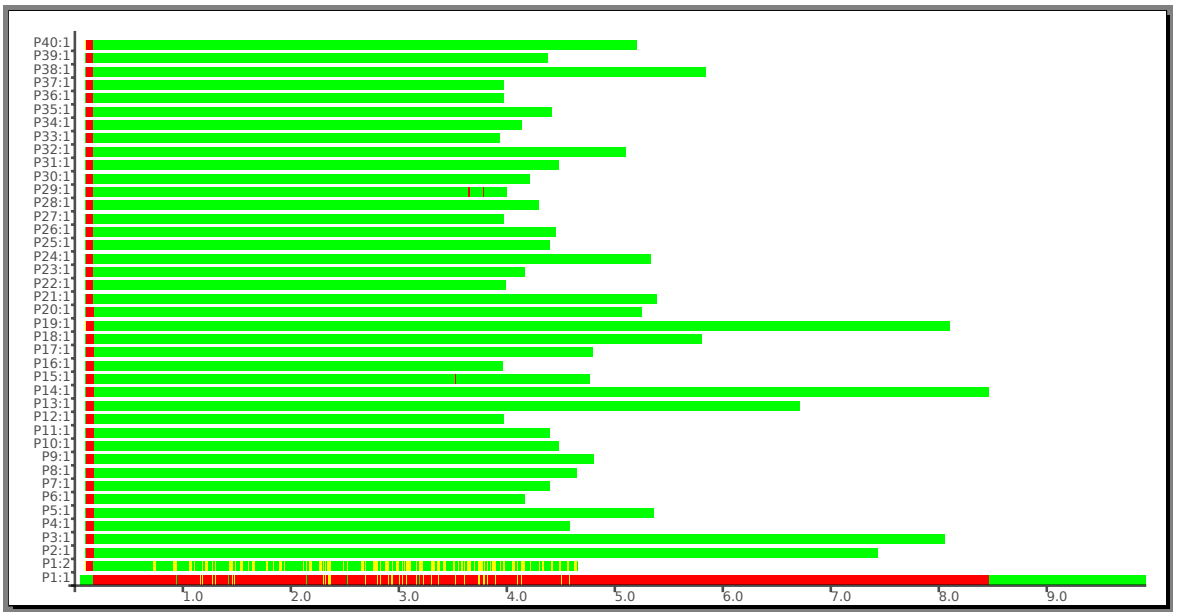


Figure 5.39: *Trace for Jacobi sum test on local workstations, input size* $2^{1279} - 1$*, 40 PE.*

also present in the differently composed set of workstations. We use consistently workpool C. We see some support for the conjecture about better task balancing for increased number of tasks. If we compare the 24 PE version with 40 PE version, we see larger final tasks on more PEs. In the twenty-four PE diagram (Figure 5.38) only one PE works for 4 seconds longer than average. Contrarily to that, in the forty PE visualisation, five tasks compute for 3–4 seconds longer than average, four additional tasks compute for 1–2 seconds longer than average. Nine tasks are done at least 0.5 seconds faster than average. This shows further development potential of the task pool transformations for the parallel implementations of the Jacobi sum test.

Another issue for the future work would be further improvement of the implementation, using some further optimisations and theoretical advances. Issues discussed by [Lenstra, 1984, Bosma and van der Hulst, 1990, Mihăilescu, 1998] are of interest. Better, but, probably, more low-level powering routines would account for better execution times. For these, consult, e.g., [Cohen, 2000, Section 1.2] or [Crandall and Pomerance, 2005, Chapter 9]. For generic fast multiplication routines see next chapter. Special tricks for fast multiplication in residue classes are also reviewed in [Crandall and Pomerance, 2005, Chapter 9]. A classic one is [Montgomery, 1985].

```
rabinMiller, jacobi :: Integer → Maybe Integer

isPrime :: Integer → Maybe Integer -- ensures primality
isPrime x = rabinMiller x »= jacobi
```

Figure 5.40: *Composition of primality tests in Haskell.*

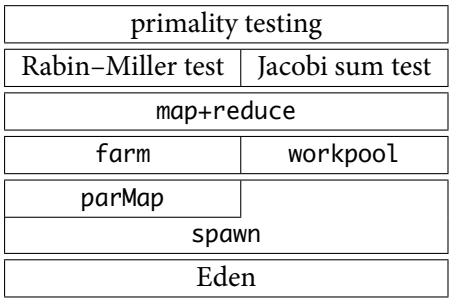


Figure 5.41: *Overview of this chapter.*

5.6 Conclusions

Outgoing from a wish to implement a skeleton for parallel repeated computation we arrived at the `map+reduce` implementation. We found it sufficient for the task of implementing two primality tests. For that purpose the possibility of a premature abort was important, but granted for free in a parallel functional setting. The Rabin–Miller test enables fast primality testing of large numbers. With the Jacobi sum test we can issue guaranties of primality. An integer accepted by the test is definitively prime, despite using probabilistic primality tests to obtain such a guaranty. The parallel performance of the first stage, the Rabin–Miller test, is very well. The parallel quality of Jacobi sum test is well in the multicore environment and satisfactory in the local workstations environment. A better insight into the task load of the test would improve this performance.

As we hinted in Sections 5.4, Rabin–Miller test is often used as a first stage for more complicated primality tests, like the Jacobi sum test. Having both we can build a chain of tests, actually *assuring* primality. As `Maybe` is a monad, we can write the code in Figure 5.40. This code will run the Jacobi sum test only if the previous test (i.e., Rabin–Miller test) has returned `Just n`, a `Nothing` would be passed through immediately. As the Jacobi sum test either ensures primality, or ensures compositeness, or fails, the output `Just n` of the function `isPrime` will ensure that `n` is a prime number. Thus we have completed our toolchain for the probabilistic parallel primality testing.

An overview of this chapter's contribution is presented in Figure 5.41. The boxes symbolise 'building blocks', i.e., the components of the implementation. The abstraction level grows from bottom to top. Building on Eden primitives and `map+reduce` scheme, we arrive at the complete toolchain for primality testing.

Rabin–Miller test has been parallelised in [Hahnel, 1998, Cheung et al., 2004, Schmidt et al., 2004]. These works focus on (parallel) hardware design. In a contrast, we suggested a high-level parallelisation with the `map+reduce` skeleton scheme. Our approach supports the premature termination of the computation. The parallelisation of the Jacobi sum test and the generic `map+reduce` skeleton scheme form our original contribution. We are not aware of these results being presented in the literature.

Chapter 6

FAST MULTIPLICATION — DIVIDE AND CONQUER

> …and then the different branches of Arithmetic—Ambition, Distraction, Uglification, and Derision.
>
> Lewis Carroll, *Alice's Adventures in Wonderland*

Mathematical software needs various kinds of multiplication. Even if we limit ourselves to basic structures and to two factors of the same kind, we need to multiply integers, polynomials, matrices. But before we learn how do it fast and in parallel, we need to realise why we focus specially on multiplication. The addition and subtraction can be done in a linear time in the number of the elements of a polynomial or of a matrix. This corresponds straightforwardly to the bit-length of an integer. But the multiplication is different! Consider polynomials or matrices over some unique factorisation domain. We call a single operation in this domain a 'base operation'. As we will see below, we can represent integers as polynomials. We need a quadratic number of base operations for the naive integer and polynomial multiplication. The matrix product needs the cubic number of base operations. So, the multiplication is truly a not trivial operation. As for the division, its complexity depends on the complexity of the multiplication. There are some division-related problems, *viz.* [von zur Gathen and Gerhard, 2003, Chapter 9]. But from the complexity-theoretic point of view, it suffices to master the fast multiplication to obtain an upper bound on all four base arithmetic operations [von zur Gathen and Gerhard, 2003].

Let us consider the multiplication of polynomials in a single variable. If we assume the same degree n and the full density of both polynomials, then it takes $\mathcal{O}(n^2)$ operations to compute the product, which is not quite acceptable. Still, we consider the basic method in Section 6.1. As the advance of computing made calculations with large polynomials not only feasible, but also needed, a number of more advanced approaches were stated. The first one is the Karatsuba multiplication. It uses a ternary divide and conquer scheme. We present the divide and conquer skeletons in Section 6.2. The polynomials are discussed in Section 6.3, the parallel Karatsuba multiplication for polynomials is discussed in Section 6.3.2. In the history of the fast multiplication further improvements followed the Karatsuba multiplication, e.g., the Toom–Cook multiplication [Toom, 1963, Cook, 1969]. The crown of the multiplication quest was the method by Schönhage and Strassen [1971] utilising the fast Fourier transform, see also [Gentleman and Sande, 1966, Pollard, 1971, Schönhage, 1982, Cantor and Kaltofen, 1991, Yap and Li, 2000, Fürer, 2007]. We present parallel skeletal implementations of the said transform (which we also call FFT from now on) in Section 6.4 and apply it to polynomial multiplication in Section 6.4.7.

It is easy to map the approach of the polynomial multiplication to integers. The latter are seen as polynomials in their base. The resulting integer is then obtained with Horner scheme evaluation of the result polynomial at the base. We will see, how to model the fast integer multiplication with fast polynomial multiplication on page 126. Hence, we focus on the core of the fast polynomial multiplication in this chapter. In order to implement both Karatsuba multiplication and FFT-based multiplication—and thus: the FFT—we need an implementation of the divide and conquer scheme.

The matrix-matrix product of $n \times n$ matrices takes $\mathcal{O}(n^3)$ base operations in the account. An idea, similar to Karatsuba multiplication, is the basis of the fast matrix-matrix multiplication, i.e., the Strassen [1969] method. It computes the same product in $\mathcal{O}(n^{\log_2 7})$ operations. We consider matrices in Section 6.5 and deal with the Strassen method in Section 6.5.2. We present an alternative, actor-like implementation of a dcF divide and conquer skeleton in Section 6.6. (See Table 3.3 on page 32 for the classification of the divide and conquer skeletons.) Conclusions follow in Section 6.7.

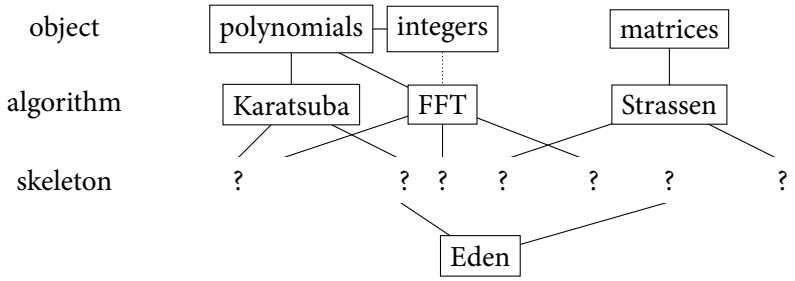


Figure 6.1: *Questions of this chapter.*

We base this chapter on our results from [Lobachev and Loogen, 2008, Berthold et al., 2009a,b,c]. Our mission statement is in Figure 6.1, we need to pave ways from the aforementioned multiplication algorithms for mathematical objects to their parallelisation in Eden. Figure 6.41 on page 142 presents the complete version of the above figure, we will fill the gaps in this chapter.

The contributions of this chapter are

- A case study of quadratic dense and sparse matrix multiplication in Haskell.
- We present the Eden implementation of the divide and conquer skeletons `dcNtickets`, `divConFlat` and `dcFarm`.
- Case study of parallel Karatsuba multiplication with our skeletons.
- The implementation of the `map-and-transpose` skeleton in Eden. It has been suggested before by Gorlatch and Bischof [1998]. However, Gorlatch and Bischof implemented it in C+MPI.
- A skeleton-based implementation of the fast Fourier transform using above divide and conquer skeletons and the `map-and-transpose` skeleton. Such implementations are not new, *cf.* [Grama et al., 2003], but up to [Gorlatch and Bischof, 1998] we are not aware of skeleton-based implementations. Contrarily to [Gorlatch and Bischof, 1998], we compare both approaches to the parallel fast Fourier transform.
- Strassen multiplication has been implemented in Haskell, see, e. g., [Rainey and Wise, 2004]. We base a parallel implementation on our divide and conquer skeletons. Contrarily to [Poldner and Kuchen, 2008a,b], we present a parallel *functional* approach.
- For the `dcFarm` skeleton we have implemented the actor model [Hewitt et al., 1973] in Eden. This was not done before.
- We include a throughout analysis of our results, using parallel penalty and serial fraction. The former is our new-developed method.

6.1 History

The definition of multiplication is as long as the written history of mankind. In the heritage of Ancient Egypt, algorithms for addition, subtraction and multiplication can be found, e. g., in the Rhind papyrus. We briefly sketch the historic development of multiplication, following [Chabert, 1999, Eves, 1969]. The number system was additive, so addition and subtraction were quite easy. But, in order to implement multiplication, division and reduction to the common denominator, more complicated operations were required: *duplication* and *mediation*. We implicitly work in a binary positional number system. Ancient Egypt mathematicians worked with particular vulgar fractions, however we aim for integers or polynomials.

Algorithm 9 Russian peasant multiplication.

Require: two integers, a and b.
1: Let $a^{(0)} \leftarrow a$ and $b^{(0)} \leftarrow b$.
2: **repeat**
3: write down $a^{(i+1)} \leftarrow \lfloor a^{(i)}/2 \rfloor$ and $b^{(i+1)} \leftarrow 2b^{(i)}$.
4: Set $i \leftarrow i + 1$.
5: **until** $a^{(i)} = 0$.
6: Sum all $b^{(i)}$ where corresponding $a^{(i)}$ odd.
Ensure: product of a and b.

A method similar to the aforementioned is known as 'Russian peasant multiplication.' We show it for integers, although all the multiplication algorithms in this section can be applied both for polynomials and for integers, the only difference lies in the carry bits. There is also a FFT-based multiplication approach for integers. But it has some caveats and special cases, the polynomials do not have. For more information see [Schönhage and Strassen, 1971, von zur Gathen and Gerhard, 2003, Emiris and Pan, 2010]. We begin at a low pace and present the Russian peasant multiplication in Algorithm 9. We understand the division in Algorithm 9 as integer division with *flooring*, i.e., $5/2 = 2$. We denote flooring by $\lfloor \cdot \rfloor$. In this algorithm we look for the unities in the binary representation of a and add together the products of b with the corresponding powers of two. In other words: for $a = \sum_i a_i 2^i$, we compute

$$ab = \sum_{i \text{ with } a_i \neq 0} b2^i.$$

Example 6.1. *We compute* $57 \cdot 48$ *with Algorithm 9. We obtain*

✓	57	48
	28	96
	14	192
✓	7	384
✓	3	768
✓	1	1536

Now, if we sum all the second columns of the marked lines, we obtain $48 + 384 + 768 + 1536 = 2736$. *This is the correct product of* 57 *and* 48. ✓

The multiplication became more feasible and easy with the introduction of a positional number system. It is a rather old invention, used already by Babylonian astronomers. They utilised a positional system to the base of 60, we still see its remains in our measures of time and angle. The corresponding multiplication algorithm is widely known as the 'schoolbook multiplication', it is exactly the way the multiplication is taught in schools today. In [Chabert, 1999] we see a bit different method, called 'tableau', 'net' or 'grid' multiplication, for the form of writing down the intermediate results, when executing it on paper. Algorithm 10 shows the schoolbook multiplication method in an algorithmic description. Under $[a_n, a_{n-1}, \ldots, a_0]$ we understand the *digits* of an integer $a = [a_n, \ldots, a_0]$. So, 1235 would be $[1, 2, 3, 5]$. However, it also can be a polynomial $x^3 + 2x^2 + 3x + 5$, we do not go into the details of carrying, hence Algorithm 10 is suitable for both. Knuth shows an algorithm where the addition is interleaved with elementary multiplication [Knuth, 1998, Algorithm M from Section 4.3.1]. It is immediately obvious, that Algorithm 9, Algorithm 10 and Knuth's version all have the complexity of $\mathcal{O}(n^2)$ elementary multiplications.

If we represent the integers as lists, we can implement Algorithm 10 quite easily. The following code for polynomial multiplication originates from the Numeric Prelude [Thurston et al., 2010].

```
mul :: (...) ⇒ [a] → [a] → [a]
mul xs = foldr (λy zs → add (scale y xs) (shift zs)) []
```

Algorithm 10 Schoolbook multiplication.

Require: $a = [a_n, a_{n-1}, \ldots, a_0]$ and $b = [b_m, b_{m-1}, \ldots, b_0]$.

1: **for** $i = 0$ to m **do**
2: compute $c^{(i)} = [a_n b_i, a_{n-1} b_i, \ldots a_0 b_i]$.
3: **end for**
4: Add together $c^{(0)}, \ldots c^{(m)}$ whereas i^{th} entry $c^{(i)}$ is shifted i steps to the left. *// Such a shift corresponds to the multiplication by B^i, where B is the base of the number system used for a and b.*

Ensure: product of a and b.

Here `scale y xs` is defined as `map (y*) xs`, and `shift` prepends a zero, symbolising the multiplication by the variable. We have omitted the type context of the `mul` function.

The further development of the quadratic multiplication includes Napier's rods, which were improved by Lucas and Genaille. The latter rods were commercially printed in Berlin up to 1911 [Chabert, 1999]. John Napier (also: Neper, Nepair) of Merchiston, *1550, †4.4.1617, also called 'marvellous Merchiston', is remembered for discovering logarithms. Further, he stimulated the usage of decimal point. The binary arithmetic was introduced by Gottfried Wilhelm Leibniz, *1.7.1646, †14.11.1716, in *L'explication de l'arithmétique binaire* (1703). However, he was interested in it already in 1666. Leibniz related the problem of Chinese king Fohy to the binary arithmetic.

It the remaining part of this chapter we focus on two aspects, that is, firstly, on non-integer, e.g., univariate polynomial or matrix-matrix multiplication, secondly, on fast multiplication routines. (The only exception is the comparison of two representations of the polynomials in Figure 6.7 on page 100.) The methods for polynomials and integers are essentially the same, we focus on the first ones. See page 126 for a short treatment of integer multiplication.

6.2 Divide and Conquer Skeletons

Before we continue with multiplication algorithms, we need to consider algorithmic skeletons for a specific kind of algorithms, namely the divide and conquer ones. The essence of any divide and conquer algorithm is:

1. Given a task `x`, decide, whether it *is trivial* or not.
   a) If not: *divide* it into a few subtasks `xs`.
   b) If it is trivial: *solve* the task `x` and return the result.
2. Call the algorithm recursively for each subtask from `xs`.
3. Given the subsolutions `ys` of subtasks `xs`, *combine* them to a single solution. Return it.

The generic class of divide and conquer algorithms allows a different amount of subtasks, depending of the recursion depth. However, we consider here only particular divide and conquer algorithms, where the amount of the subtasks is fixed for each algorithm. We call such divide and conquer algorithms *regular*. Further, according to the divide and conquer classification (see Section 3.2.3 and [Herrmann, 2000]), FFT and Strassen matrix multiplication belong to the most restricted `dcF` class of divide and conquer schemes, while the Karatsuba multiplication belongs to the `dcC` class, which requires the original problem to be present in the `combine` function.

Before we can continue with faster methods for polynomial multiplication, we need to introduce some divide and conquer skeletons. These will be used not only for Karatsuba multiplication, which we will consider in Section 6.3, but also for FFT (Section 6.4) and Strassen matrix multiplication (Section 6.5). In this chapter we use the type `DC a b`, which we restate below.

```
type DC a b = (a → Bool)      -- ^ trivial?
            → (a → b)         -- ^ solve
            → (a → [a])       -- ^ divide
```

```
divConX_f k t trivial solve divide combine x
  = divConX_c k t trivial solve divide
              (λ _ parts → combine parts) x
```

Figure 6.2: *Interfacing dcF from dcC.*

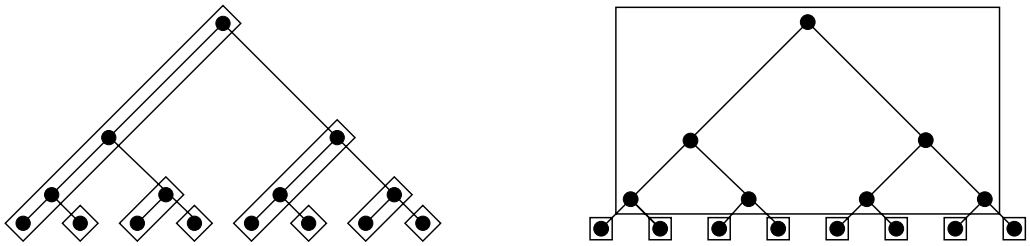


Figure 6.3: *Divide and conquer expansion schemes binary trees, depth 3. Left: distributed expansion, right: flat expansion. Boxes symbolise processes.*

```
→ (a → [b] → b) -- ^ combine
→ a             -- ^ input
→ b             -- ^ result
```

This type corresponds to the divide and conquer class dcC. We presented the complete classification in Chapter 3, see Table 3.3 on page 32 for an overview. We have defined the sequential divide and conquer skeleton in Figure 3.16 on page 32. Each divide and conquer skeleton takes four parameter functions for the actual implementation of a divide and conquer algorithm: to decide whether the input is trivial (the first parameter function, `isTrivial`) and, in this case, to solve it (using parameter function 2, `solve`), or else to decompose non-trivial input into a number of sub-problems, which are solved recursively (with parameter function 3, `divide`), and to combine the output (parameter function 4, `combine`).

Interface to dcF skeletons. The difference between dcC and dcF classes lies in the additional parameter for the `combine` function. Thus, it is possible to implement the dcF skeletons in terms of dcC skeletons. The essential idea is to change the signature of the `combine` function. So, the interface for dcF skeleton `divConX_f`, based on the above dcC implementation `divConX_c` is shown in Figure 6.2. We denote the dcF skeletons in Haskell with the type `DC' a b`.

```
type DC' a b = (a → Bool)
             → (a → b)
             → (a → [a])
             → ([b] → b)
             → a → b
```

However, because of different types of `k` and `t` in different versions, one needs to write three such interface functions: each with a correct type for corresponding implementation of dcF skeleton.

6.2.1 Distributed Expansion Skeleton

We see two basic approaches for the parallel regular divide and conquer. The first one is presented here, the other one is detailed in the next section. We discussed these two approaches in [Berthold et al., 2009a,b], see also [Cesari and Maeder, 1996b] for a special case with focus on Karatsuba multiplication. Imagine a divide and conquer tree. The very first idea is to spawn a parallel task for each branch of the tree. However, in this case the father task is locked until all (supposedly equal) spawned tasks return results of their computation. So we give only $r-1$ tasks away for r-ary divide and conquer tree. These $r-1$ tasks are instantiated as new processes, as long as PEs are available. These branches will recursively produce new parallel subtasks again, resulting in a *distributed expansion* of the computation. We show

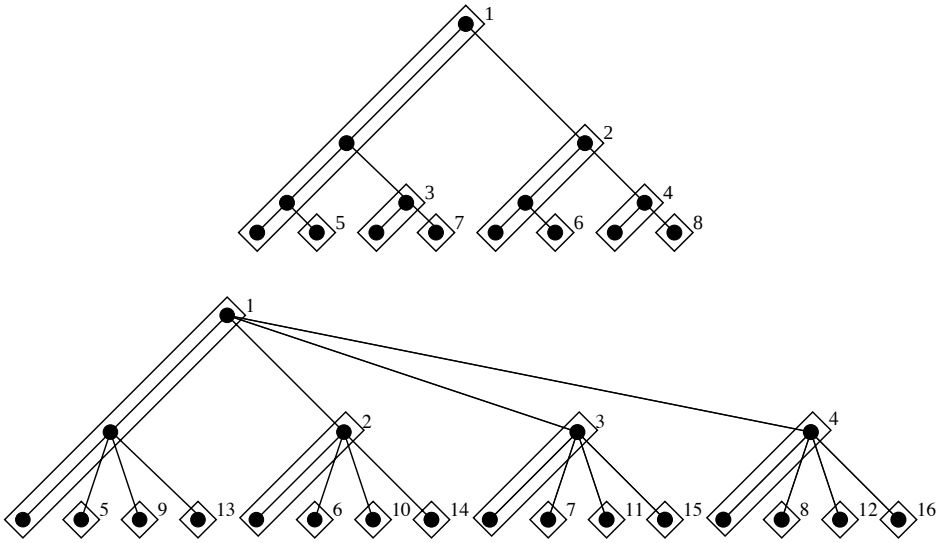


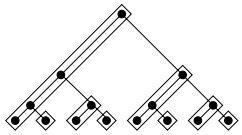
Figure 6.4: *Ticket placement with* `dcNtickets` *skeleton for binary and quaternary divide and conquer trees. Images from [Berthold et al., 2009c].*

it in Figure 6.3, left, for the binary tree. Another approach is to unfold the divide and conquer tree to a larger depth and then to process the many small tasks in a workpool. It is called *flat expansion* and is depicted in Figure 6.3, right, again for a binary scheme. The main problem of the flat expansion is to control the combine step. We elaborate on it in the next section and focus on distributed expansion here.

A *ticket list* is used to control the placement of newly defined processes in Figure 6.4. At first, the PE numbers for placing the immediate child processes are taken from the ticket list. Then, the remaining tickets are distributed to the children in a round-robin manner using the function `unshuffle`. The explicit process placement with the ticket lists is a simple and flexible way to control the distribution of processes and the recursive unfolding of the task tree. If too few tickets are available, computations are performed locally. Duplicate tickets can be used to place several child processes on the same PE. The numbers in Figure 6.4, top and bottom, give the PE numbers used for placement with the ticket lists `[2..8]` and `[2..16]` respectively.

The implementation of the `dcNtickets_c` skeleton is in Figure 6.5. The skeleton receives two additional parameters: the arity of the divide and conquer tree k and a list of the *tickets*. If the latter list is empty, we use the sequential skeleton. This is the first equation of `dcNtickets_c`. The second equation handles two cases. In the first guard expression the trivial case is `solved`. The second guard expression is the interesting, parallel case. In the beginning the demand control is established. The evaluation of the list of children is demanded early, then the evaluation of `myRes` and `localRess` to the reduced normal form is forced, before the result of the application of `combine` can be used. The latter is the outcome of the skeleton invocation.

In the `where`-clause of the `dcNtickets_c` skeleton, the ticket list is split into the first $k-1$ elements and the remaining list. Then, the position of the children and of the further ancestors (i. e., grandchildren, etc.) is determined. The initial tasks are divided into two parts, the head of the resulting list is `myIn`, which is processed by a recursive call—by an application of the binding `ticketF` in the definition of `myRes`. The tail of the divided tasks is `theirIn`. The part of the computation, deemed to be local, is performed by mapping the sequential skeleton on `localIns`—that part of the list of the divided tasks, for which we have no free tickets. This results in `localRess`. The parallel child tasks are computed in the binding of `childRes`: the functions `childProcs`, i. e., the parallel recursive call of the skeleton, are applied to at most k-many `procIns`. The PE numbers for these processes reside in the `childTickets` part of the `tickets` list.



```
dcNtickets_c :: (Trans a, Trans b)
             ⇒ Int   -- ^ n (expect n children)
             → [Int] -- ^ Tickets (machine ids to use)
             → DC a b
dcNtickets_c k [] trivial solve divide combine x
  = divConSeq_c trivial solve divide combine x
dcNtickets_c k tickets trivial solve divide combine x
  | trivial x = solve x
  | otherwise =  childRes `pseq` -- early demand on the list of children
                       rnf myRes `pseq` rnf localRess `pseq`
                       combine x (myRes:childRes ++ localRess)
        where
          -- splitting computation into processes
          (childTickets, restTickets) = splitAt (k-1) tickets
          -- denote position of (children, further ancestors)
          (myTs:theirTs) = unshuffle k restTickets
          ticketF ts = dcNtickets_c k ts trivial solve divide combine
          insts = length childTickets
          (procIns, localIns) = splitAt insts theirIn
          childProcs = map ticketF theirTs
          childRes  = spawnAt childTickets childProcs procIns
          -- local computation:
          myRes = ticketF myTs myIn
          (myIn:theirIn) = divide x
          localRess = map (divConSeq_c trivial solve divide combine) localIns
```

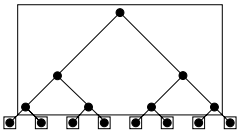
Figure 6.5: *Expansion-based divide and conquer with tickets.*

6.2.2 Flat Expansion Skeleton

Contrary to the previous approach, the flat expansion skeleton performs several steps of `divide` locally on the master PE in order to generate a large amount of smaller tasks. When these tasks are done, they are again `combine`d locally on the master. As `combine` is connected with a large volume of received data, it should be generally more trivial than `divide`. As for the many small tasks, we have generated, they are processed in a task balanced parallel `map` implementation—in other words: in a `farm` or a `workpool`.

The flat expansion divide and conquer skeleton given in Figure 6.6 is a specialisation of the 'divide and conquer by replicated workers' (`dcrw`) skeleton, displayed in [Peña and Rubio, 2001, Loogen et al., 2003]. The 'replicated workers' is another term for 'a master-worker scheme'. The skeleton exploits the fact that our divide and conquer scheme has the fixed branching degree k. The divide and conquer tree is unfolded up to a given depth d and each process selects one of the resulting sub-trees (with function `taskNr`). Instead of a `workpool`—a skeleton with dynamic task balancing—we can use a simple `farm` with static task distribution. This suffices for the regular parallelism of our applications. It is possible to specify the parallel `map` implementation as a parameter. The function `combineTopMaster` combines the results level-wise. The advantage of this skeleton is that the divide and conquer tree can be unfolded to produce much more tasks than available processor elements. This helps to achieve a balanced parallel computation.

It is important that the *unevaluated* task information can be passed to the child processes which select their own parts of it. This technique is called direct mapping [Klusik et al., 2000], *viz.* Chapter 3. It reduces the computation time in master PE for `divide`. A real application would use *future handles*, also known in Eden environment as remote data [Dieterle et al., 2010b], to pass the result of previous distributed computation to the workers. This is allowed, as in the production use the input data *should* be distributed along the PEs anyway. We will come back to this issue in Section 6.4.5. As all processes



```
divConFlat_c :: (Trans a, Trans b)
             ⇒ Map a b -- ^ a custom map implementation
             → Int     -- ^ depth of parallel DC tree
             → DC a b  -- ^ DC type
divConFlat_c myParMap depth trivial solve divide combine x
  = combineTopMaster combine levels results
  where (tasks, levels) = generateTasks depth trivial divide x
        results = myParMap (divConSeq_c trivial solve divide combine) tasks

data Tree a = Tree a [Tree a] | Leaf a

-- define, how Tree a can be evaluated to rnf
instance NFData a ⇒ NFData (Tree a)
  where rnf (Tree a ls) = rnf a `pseq` rnf ls
        rnf (Leaf a)    = rnf a

-- the helper functions for the combine in dcC
combineTopMaster :: (NFData b) ⇒ (a → [b] → b) → (Tree a) → [b] → b
combineTopMaster c t bs = fst (combineTopRnf c t bs)

combineTopRnf :: (NFData b) ⇒ (a → [b] → b) → (Tree a) → [b] → (b, [b])
combineTopRnf _ (Leaf a) (b:bs) = (b,bs)
combineTopRnf combine (Tree a ts) bs
  = (rnf res `pseq` combine a res, bs')
  where (bs', res) = foldl f (bs, []) ts
        f (olds, news) t = (remaining, news ++ [b])
          where (b, remaining) = combineTopRnf combine t olds

generateTasks :: Int → (a → Bool) → (a → [a]) → a → ([a], Tree a)
generateTasks 0 _ _ a = ([a], Leaf a)
generateTasks n trivial divide a
  | trivial a = ([a],Leaf a)
  | otherwise = (concat ass, Tree a ts)
  where (ass, ts) = unzip $ map (generateTasks (n-1) trivial divide) (divide a)
```

Figure 6.6: *Flat expansion divide and conquer skeleton for k-ary task trees.*

are created by the same 'master' process, a simple round robin process placement—which is the default—is sufficient. The only explicit demand control is the enforced evaluation of children's results prior to their combination.

6.3 Univariate Polynomials

One of the ubiquitous structures in mathematics are polynomials. We focus here on polynomials in one variable. We define the univariate polynomials and present some thoughts on the representation of polynomials in Section 6.3.1. The Karatsuba multiplication follows in Section 6.3.2. The same section devises the complexity of Karatsuba's algorithm. Sections 6.3.3, 6.3.4 and 6.3.5 describe the performance of our parallelisation, estimate the execution time w. r. t. input size and conclude correspondingly.

6.3.1 Preliminaries

An univariate polynomial is a finite sequence of values $a_n, \ldots, a_0$ in a single variable x, i. e., $a_n x^n + \cdots + a_0$. We need a precise definition first.

Definition 6.2 (Polynomial)**.** Given a finite index set $I \subset \mathbb{N}$, a sum of terms in a variable x

$$f(x) = \sum_{i \in I} a_i x^i$$

is a *polynomial* of *degree* $\max I$. Herein for all $i \in I$ the value a_i is called the coefficient, i is called an exponent because of x^i. A *monomial* is a polynomial, consisting of a single term in the sum.

To store a polynomial we store its coefficients. We have not told yet, what type the coefficients a_i have. Indeed, this is various. We can have polynomials over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, finite fields. We could use more complicated constructs, e. g., polynomials over matrices, if we would like to. So, the mathematical theory speaks of polynomials over some unique factorisation domain[1], abbreviated UFD. So, given such a UFD K, we denote the *ring* of polynomials with $K[x]$. It is a UFD, if K is. (See Theorem A.25 in the Appendix.) Even more: $K[x]$ is a K-*algebra* of polynomials, see [Grove, 2004, p. 163]. If $I = \varnothing$, we have a zero polynomial. Its degree is $-\infty$. A polynomial with $I = \{0\}$ is called a constant polynomial. If K has no zero divisors, then a constant polynomial $u \in K[x]$ is a unit in $K[x]$ if and only if u is a unit in K.

An alternative to the 'coefficient' representation is the 'point–value' representation of polynomials [Cormen et al., 2001]. In this case, for a polynomial $f(x)$ of degree n, we store a set of pairs $\{(x_i, y_i) : i \in \{0, \ldots, n\}\}$ such that $y_i = f(x_i)$ with $x_i \neq x_j$ for all $i, j \in \{0, \ldots, n\}$ and $i \neq j$. To convert the 'point–value' representation to the 'coefficient' representation from above, we need to interpolate the polynomial. To obtain a 'point–value' representation from the 'coefficient' one, we evaluate the polynomial at $n + 1$ distinct points x_i. The 'point–value' representation allows a faster multiplication, but the 'coefficient' representation is more convenient for evaluation at an arbitrary point. We consider only the 'coefficient' representation below.

The shape of the set I is important. It should be finite and contain only non-negative integers. If we allow negative exponents, we obtain Laurent polynomials. If we allow infinite index set I, we obtain series. We briefly discussed them in Chapter 2. With both extensions we get Laurent series. All of them are *very* different from the polynomials. For instance, for polynomials over rationals, the set of the multiplicatively invertible polynomials consists only of constants. For Laurent polynomials over the same field, the latter set includes all monomials.

Basic Representation. We have at least two possibilities for the 'coefficient' representation of polynomials. Either we store a *list* of coefficients—we shall call this approach a *dense* representation. Or we store a *list of pairs* of coefficient and exponent. Such representation is called a *sparse* one. We use both approaches to represent polynomials in Haskell. We use `Int` as an index set. The rationale behind this is: limiting the maximal degree of polynomials to 2^{32} or 2^{64} is not an actual limitation.

```
type PolyD a = [a]
type PolyS a = [(Int, a)]
```

But what is `a`? It represents the UFD K, we can safely constrain `Num a ⇒ a`. A more algebraic implementation would introduce its own type class for UFDs and constrain `a` to be its instance. But even in our representation, the above types describe polynomials over various possible types. In other words, this is a symbolic representation of polynomials, which abstracts from the base UFD. The algorithms, we present below, do not change with the change of the base type.

Concerning the data structure choice, it is arguable that for large polynomials arrays might be a better choice. In a sequential case: they definitely are. But as we need to introduce some overhead to transmit arrays in Eden and because of our FFT implementation, it might be justified to use lists. The reason is: our implementation of FFT will also use lists as its input and output. Now, the choice between sparse and dense representations of polynomials depends on the input. If we have very sparse polynomials, then a dense representation is no good. However, if the polynomials are not very sparse, we could win with a dense representation, as we can use the FFT-based multiplication method. We

[1] See Section A.6, esp. Definition A.23.3 on page 197 for the definition of a unique factorisation domain.

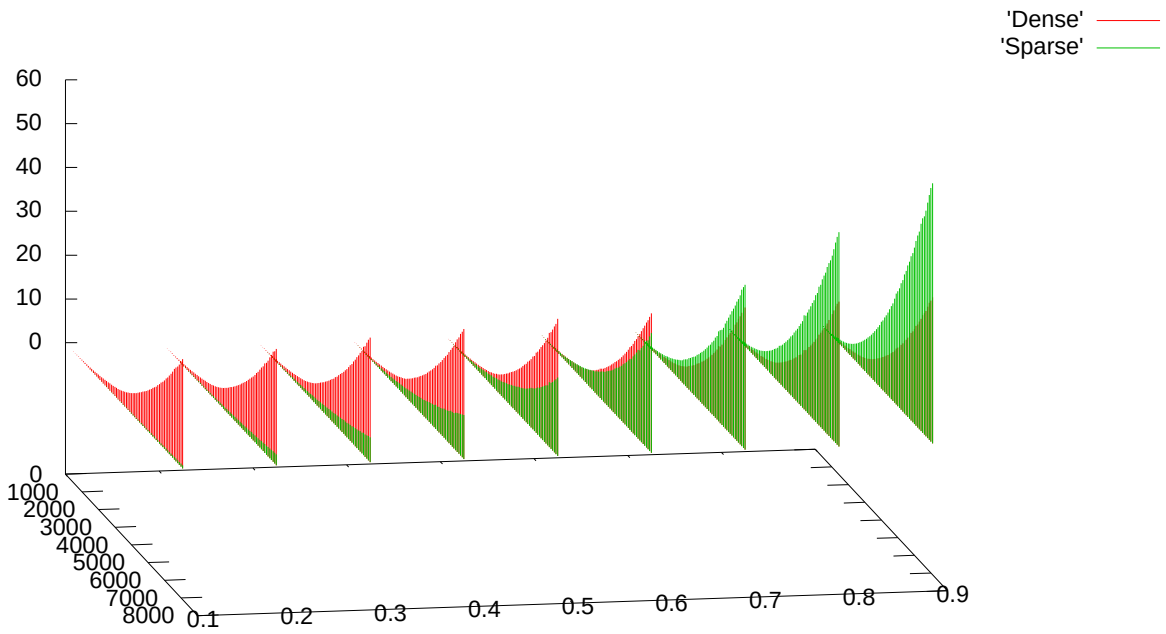


Figure 6.7: *Comparing naive dense and naive sparse polynomial multiplication.*

describe the FFT-based multiplication in detail in Section 6.4.7. A full-fledged implementation would not use a type alias for lists, but define an own type, possibly with named fields. To give an example:

```
data Poly a = {coeffs :: [a], leadCoeff :: a, degree :: Int,
  numRealRoots :: Int, roots :: [Maybe (Closure a)], approxRoot ::
  [Complex Double], ... }
```

Here `Closure a` denotes the type of the elements in the algebraic closure.

Thus it will be possible for some function in a complicated program to store some extended information about a given polynomial fully transparent for the rest of the program. Examples of such information are degree and roots of the polynomial. However, let us not consider this advanced technical features any longer. We compare a dense and a sparse polynomial multiplication next.

In order to compare the feasibility of sparse and dense polynomials, we have implemented naive polynomial multiplication for both storage types. The non-naive methods are presented in the next section. We compare the execution time of the multiplication for different *density* of polynomials and for various degree. We compute the product of different polynomials of the same degree. The result is a 3D plot, which provides us with thresholds for a hybrid algorithm. Figure 6.7 compares the runtime of dense and sparse polynomial multiplication, obtained with GHC 6.8 on `sakania`. The degree of the polynomials varies from 500 up to 8000 at the first horizontal axis, the density varies from 0.1 to 0.9 at the second horizontal axis. The vertical axis denotes the execution time. The coefficients are randomly generated small integers. We see: for density ≥ 0.6 for small degrees and for density ≥ 0.7 for larger degrees the dense polynomial multiplication is faster.

6.3.2 Karatsuba Method

We shall present the Karatsuba algorithm now, basing on [Karatsuba and Ofman, 1962, Karatsuba, 1995, Bernstein, 2001, von zur Gathen and Gerhard, 2003]. We formulate the algorithm for the polynomials here, although it works for both integers and polynomials. See [Berthold et al., 2009a] and Table 6.1 on page 103 for the results of the parallel integer version. The only difference of the latter is that it has to care about carrying, when reconstructing the result after the actual multiplication.

History. This paragraph on history of Karatsuba multiplication follows [Karatsuba, 1995]. Andrey Kolmogorov (Андрей Николаевич Колмогоров), *25.4.1903, †20.10.1987, is well known for his other

Algorithm 11 Karatsuba multiplication.

Require: polynomials u and v of degree $n = 2^k$.

1: Separate u and v in lower and upper parts with $f = f_1 x^{n/2} + f_0$ for $f \in \{u, v\}$.

2: Compute three products

$$\begin{aligned} r_2 &\leftarrow u_1 v_1 \\ t &\leftarrow (u_1 + u_0)(v_1 + v_0) \\ r_0 &\leftarrow u_0 v_0 \end{aligned}$$

using recursive calls.

3: Set $r_1 \leftarrow t - r_2 - r_0$.

Ensure: The product $r_2 x^n + r_1 x^{n/2} + r_0$ of u and v.

contributions, including the Kolmogorov complexity and the solution of the Hilbert's Thirteenth Problem in a collaboration with V. Arnold (Владимир Игоревич Арнольд). The Kolmogorov complexity handles the complexity of representing some amount of information, contrary to the algorithmic complexity, which we simply reference as the complexity in this work. Before 1956 Kolmogorov conjectured the classical multiplication algorithm to be optimal. By conjecture any multiplication algorithm would have the complexity $\Omega(n^2)$. We call it the Kolmogorov n^2 conjecture, following [Karatsuba, 1995]. This conjecture definitely existed in 1956, as it was discussed at one of the meetings of Moscow Mathematical Society.

The first breakthrough happened in autumn 1960. A young mathematics student Anatolii Karatsuba (Анатолий Алексеевич Карацуба), *31.1.1937, †28.9.2008, took part in a seminar on the complexity of computations. Among other topics the Kolmogorov n^2 conjecture was presented there personally by Kolmogorov. In one week the student came up with a divide and conquer algorithm, which had a lower bound: $\mathcal{O}(n^{\log_2 3})$. Thus Karatsuba has disproved the Kolmogorov n^2 conjecture. Kolmogorov reported this in the further seminar session and terminated the seminar. Two years later Kolmogorov (probably in collaboration with Yu. Ofman) wrote an article about this new method. We know this article as [Karatsuba and Ofman, 1962], however Karatsuba [1995] states to have learnt about the article only upon seeing the reprints.

The algorithm. The essence of the Karatsuba algorithm lies in a tricky partitioning of the multiplicands with the subsequent usage of the divide and conquer scheme. We consider the version of the Karatsuba algorithm for polynomials. Suppose, we compute a product of two polynomials u and v, both of degree $n = 2^k$. We denote with u_1 and u_0 the upper and lower 'halves' of the polynomial u. In a more formal way, $u = u_1 x^{n/2} + u_0$ and $v = v_1 x^{n/2} + v_0$. The conventional product is uv. Let us transform it to

$$uv = (u_1 x^{n/2} + u_0)(v_1 x^{n/2} + v_0) = u_1 v_1 x^n + (u_1 v_0 + u_0 v_1) x^{n/2} + u_0 v_0.$$

But we could have obtained these factors differently! It holds

$$(u_1 + u_0)(v_1 + v_0) = u_1 v_1 + u_1 v_0 + u_0 v_1 + u_0 v_0$$

and we have to compute $u_1 v_1$ and $u_0 v_0$ anyway. This leads to the instructions of Algorithm 11. Per [Bernstein, 2001] this algorithm corresponds to evaluation and interpolation.

To derive a statement on the complexity of Karatsuba multiplication, we show a more general version of Lemma 8.2 from [von zur Gathen and Gerhard, 2003]. We follow the addenda of the latter book. We will use this proposition also for the complexity statement of Strassen multiplication, see Theorem 6.7 on page 131. We write $\operatorname{lb} x$ for the binary logarithm of x, also denoted as $\log_2 x$. The graphical representation of this complexity for the depth five is in Figure 6.8.

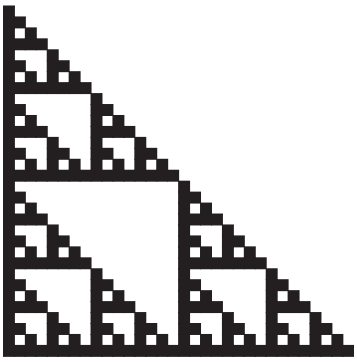


Figure 6.8: *The complexity of the Karatsuba algorithm corresponds to the surface of this fractal. We show it for the depth five. Redrawn from [von zur Gathen and Gerhard, 2003].*

Proposition 6.3. *Let a, k be in $\mathbb{R}^+$ and $T(n)$, $S(n) : \mathbb{N} \to \mathbb{N}$. Additionally, $S(2n) \leq kS(n)$ for all n in $\mathbb{N}$. It holds*

$$T(n) \leq aT(n/2) + S(n)$$

for $n = 2^i$ with natural $i > 1$. Then for $i \in \mathbb{N}$ and n as above holds

- *In case '$a = k$': $T(n) \leq T(1)n^{\operatorname{lb} a} + S(n) \operatorname{lb} n$.*
- *In case '$a \neq k$': $T(n) \leq T(1)n^{\operatorname{lb} a} + \frac{k}{a-k}(n^{\operatorname{lb}(a/k)} - 1)S(n)$.*

If additionally $n^{\operatorname{lb} k} \in \mathcal{O}(S(n))$ holds, then

- *In case '$a = k$': $T(n) \in \mathcal{O}(S(n) \operatorname{lb} n)$ holds.*
- *In case '$a \neq k$': $T(n) \in \mathcal{O}(S(n)n^{\operatorname{lb}(a/k)})$ holds.*

Proof. We unfold the recursive calls in $T(n) = T(2^i) \leq aT(2^{i-1}) + S(n) \leq \ldots$ This results in

$$T(n) \leq a^i T(1) + \sum_{j=0}^{i-1} a^j S(2^{i-j}) \leq T(1)2^{i \operatorname{lb} a} + S(2^i) \sum_{j=0}^{i-1} (a/k)^j.$$

The last inequality holds because $S(2^{i-j}) \leq k^{-j} S(2^i)$. If $a = k$, then the fraction a/k is 1 and $T(n) \leq T(1)2^{i \operatorname{lb} a} + iS(2^i)$ results. Else

$$\sum_{j=0}^{i-1} (a/k)^j = \frac{(a/k)^i - 1}{a/k - 1} = \frac{k(2^{i \operatorname{lb}(a/k)} - 1)}{a - k}$$

holds. The latter equality results from multiplying both numerator and denominator with k. □

A similar statement was shown in [Cormen et al., 2001, Theorem 4.1] under the name of a 'master theorem' for solving recursions, see also [Manber, 1989]. Now let us apply the above proposition to Karatsuba multiplication. The initial consideration is clear: to multiply two polynomials of degree $n-1$ (i. e., with n coefficients), we need to multiply *three* factors of *half* length. So $T(n) = 3T(n/2) + \mathcal{O}(n)$, i. e., $a = 3$. Of course, we compute each $T(n/2)$ as a recursive call. The more exact shape of $S(n)$ is $4n$, as we do n additions, $2n$ subtractions and again n additions in Algorithm 11. The trivial multiplication has a cost of unity, in other words: $T(1) = 1$. It holds $S(2n) = 8n$, i. e., $k = 2$. Hence, per Proposition 6.3

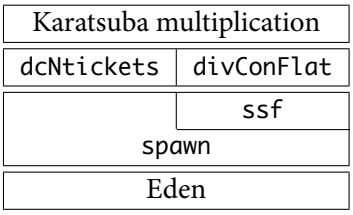


Figure 6.9: *The skeletal implementation of Karatsuba multiplication.*

	PEs	1	2	4	8
Multicore Haskell 6.11	distributed expansion	7.75	4.10	2.44	1.96
	flat expansion	5.74	3.27	1.79	1.55
Eden 6.8	distributed expansion	6.29	4.17	2.15	1.51
	flat expansion	6.24	5.52	1.89	0.88

Table 6.1: *Run time comparison of Karatsuba multiplication with Eden and Multicore Haskell (GpH) on* `sakania`*. Time is in seconds. Data originates from [Berthold et al., 2009a].*

it follows

$$\begin{aligned} T(n) &\leq T(1)n^{\mathrm{lb}\,3} + \frac{2}{3-2}S(n)(n^{\mathrm{lb}(3/2)} - 1) \\ &= n^{\mathrm{lb}\,3} + 2S(n)(n^{\mathrm{lb}(3/2)} - 1) \\ &= n^{\mathrm{lb}\,3} + 8n \cdot n^{\mathrm{lb}(3/2)} - 8n \\ &= n^{\mathrm{lb}\,3} + 8n^{1+\mathrm{lb}(3/2)} - 8n \\ &= n^{\mathrm{lb}\,3} + 8n^{\mathrm{lb}(2\cdot 3/2)} - 8n \\ &= 9n^{\mathrm{lb}\,3} - 8n \in \mathcal{O}(n^{\mathrm{lb}\,3}). \end{aligned}$$

We have shown the following theorem.

Theorem 6.4. *The Karatsuba multiplication method for two input polynomials of degree* $\leq n = 2^k$ *(for some* $k \in \mathbb{N}$*) over a ring has the complexity of* $\mathcal{O}(n^{\log_2 3})$ *ring operations.*

It is possible to generalise Karatsuba's formula to a larger arity of the divide and conquer tree. Such approaches and further modifications where made in [Toom, 1963, Cook, 1969], more on the implementation side is told in [Weimerskirch and Paar, 2003, Montgomery, 2005]. The method by Strassen [1969] is an analogue of Karatsuba multiplication for matrices. We handle Strassen matrix multiplication in Section 6.5.2.

A Haskell implementation of polynomial Karatsuba multiplication was suggested e.g., in [Hermann, 2000], the parallel Eden implementation [Loogen et al., 2003] is easy tunable with a fitting divide and conquer skeleton. We have instantiated the divide and conquer skeletons from above.

Implementation. A skeletal implementation of the Karatsuba multiplication was available to us in the form of an Eden test program. We have extended the implementation to new divide and conquer skeletons. An overview is in Figure 6.9. This modified implementation was used to compare Multicore Haskell and Eden in [Berthold et al., 2009a]. The latter work considered a 32 bit Eden compiler. Table 6.1 shows the results of Karatsuba multiplication from this paper. Below we present the Eden implementation using the new 64 bit compiler.

We show snippets of the program code, as it is a well-known algorithm, where we are interested majorly in the skeletal parallelisation. We stress it is not ours! The Haskell source code implementation of Algorithm 11 is shown in Figure 6.10.

An interesting point in the shown implementation of Karatsuba multiplication is the usage of the so-called 'lift trick'. To discuss it we need to recall how data communication works in Eden. Every data structure, which might be transmitted to another machine, should be an instance of the `Trans` type class. It specifies how the variables of a given type are send and received. There is a special `Trans` instance for lists (see Figure 3.4 on page 24), which enables list streaming. In this case the list elements are sent separately and incrementally, stream processing functions can start processing first list elements without waiting for the remaining ones. However, this induces an overhead of sending a message pro list element. In a case, when the lists are large and the cost for processing one element is small, this approach is not acceptable.

However, as of now the `Trans` instance for the lists is specified in the Eden supporting module, it is not possible for the application programmer to modify this instance. Instead, the 'lift trick' is used. Namely, besides special instances of `Trans`, a generic instance exists. It is used as a default implementation for a non-specialised instance derivation. This instance puts all the data of a value to-be-sent into a single message. This instance is defined in the same system module, where the `Trans` instance for lists resides. Now, in an application, a new type is artificially created. The actual data, in this case: a list, is wrapped into the said type. As it has no special `Trans` instance, the default, all-packing instance is used. The new data type `L` in Figure 6.10 implements the above idea.

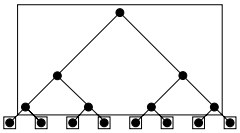
Aside from this issue, the implementations of `divide` and `combine` are very straightforward. The function `karatsuba` is the top-level implementation. It receives an additional integer parameter for the depth of the parallel divide and conquer recursion. We show the implementation with flat divide and conquer skeleton, which in its turn, uses the `ssf` implementation of the parallel map. As in practise Karatsuba method performs better than the schoolbook multiplication only from a particular input size onwards, the `solve` function uses the quadratic method, akin to Algorithm 10.

6.3.3 Performance

The performance of our implementation is very good. We obtained on eight PE `sakania` the *relative* speedups of 5.54 for input size 32 000 and 8.84 for the input size 128 000. In both cases this is a relative speedup, we used the flat expansion skeleton with divide and conquer tree depth 5 and `ssf'`—a `farm` with direct mapping, which does not co-locate tasks with the first PE. We call the latter 'master PE'. Nevertheless we want to stress the achieved super linear speedup. The speedup curves are plotted for both input sizes in Figure 6.11. We have also measured the *absolute* speedup of 5.27 for the input size 32 000. We were unable to execute the sequential version of Karatsuba multiplication for the input size 128 000. The probable reason is the excessive memory usage. Next we will consider the values of the parallel penalty w. r. t. number of processors and of the serial fraction to evaluate the above results.

We have obtained super linear speedups for some particular input size of Karatsuba multiplication, namely for the largest input we measured. We need to investigate the reason for such a phenomenal result. Some positive reasons—like cache effects—aside, a poor performance of the sequential version can also lead to super linear speedup. A possible reason is a memory problem. So, we consider the two usual parallel quality measures: parallel penalty w. r. t. number of processors and serial fraction with relative reference point. These are depicted appropriately in the top and in the bottom part of Figure 6.12. The left half of this figure shows both measures for input size 32 000, where no super linear speedup occurs. The right hand side displays the values for the input size 128 000, the case of super linear speedup.

Both quality measures decrease slowly for input size 32 000, while at 5–8 PEs both are almost constant. The increase at two PE is discussed below. As for the quality measures in case of the input size 128 000, when we consider 3–8 PEs, we see a decline at 3–5 PEs and an increase at 5–8. Interestingly, the values at 3 and 8 PE are approximately the same. Further, it is noteworthy that said values for 3–8 PEs are smaller than zero (dashed horizontal line in the plots). These values are also quite close. This indicates a good parallelisation quality on this side with a possibly bad sequential implementation. As we consider the values with the relative reference points, this means problems with our 1–2 PEs versions. This is a hint for a probably unfitting speedup value.



```
type Poly a = [a] -- Num instances are not shown

-- Lift trick
data L a = L {fromL :: a}
instance NFData a ⇒ NFData (L a) where
  rnf (L a) = rnf a
instance Trans a ⇒ Trans (L a)

divide :: L (Poly Int, Poly Int) → [L (Poly Int, Poly Int)]
divide (L (is1, is2)) = [L (is1a, is2a), L (is1b, is2b),
                         L (is1a + is1b, is2a + is2b) ]
  where ldiv = (max (length is1) (length is2) ) `div` 2
        (is1b, is1a) = splitAt ldiv is1
        (is2b, is2a) = splitAt ldiv is2

solve :: L (Poly Int, Poly Int) → L Poly Int
solve = ... -- sequential, naive multiplication

-- combine uses the initial task for the size information
combine :: L (Poly Int, Poly Int) → [L Poly Int] → L Poly Int
combine (L (is1, is2)) [L u, L v, L w] = L result
  where ldiv = (max (length is1) (length is2) ) `div` 2
        u0s = replicate (2*ldiv) 0 ++ u    -- u * base^(2*ldiv)
        wuv = w - (u + v)                  -- calculate w-u-v (= w-(u+v))
        wuv0s = replicate ldiv 0 ++ wuv    -- multiply  w-u-v by base^ldiv
        result = u0s + v + wuv0s           -- sum of subresults

karatsuba :: Int → Poly Int → Poly Int → Poly Int
karatsuba depth is1 is2
  = fromL $ divConFlat_c ssf' depth trivial solve divide combine $ L (is1,is2)
-- ssf' does not co-locate a task with master PE
```

Figure 6.10: *Implementation of Karatsuba multiplication.*

Discussion. At 2 PE there is a large value of both parallel penalty w.r.t. number of processors (Figure 6.12 on the next page, top) and serial fraction (same figure, bottom). There are two possible reasons. Firstly, two PE might be too few for an efficient parallelisation to happen. We will consider a trace visualisation to decide on this. Secondly, as we use relative reference point and relative speedup, the single PE version has zero parallel overhead. However, the problem size might be too large for a small number of PEs. Here, a notion of a scaled speedup [Gustafson et al., 1988] might have more sense.

Let us consider in Figure 6.13 the trace diagram, produced by Karatsuba multiplication program with input size 32 000 on 2 PE. We see that in fact only the second PE is working, while the master PE is idle most of the time. This explains the almost non-existent speedup at two PE for input size 32 000, which is also reflected in high penalty values for 2 PE in Figure 6.12, left. However, the trace visualisation for eight PE with input size 128 000 reveals some work in the master PE, *viz.* Figure 6.14. The same figure shows that the worker processes are somewhat reluctant in their startup and have a corresponding lag when terminating. This image is disappointing, as such balancing issues should have prohibited super linear speedups. In this particular case we deem the overall 'lost' time to be 13% of the total work (i.e., of $p \cdot T(n, p)$). We calculate this simply: we see in the trace diagram that each next process takes a second more for startup and needs this very second more for termination. Further, we have seven worker PEs, the total execution time is 53.3 seconds. Thus, our expectation for the 'real' speedup would be ≈ 7.

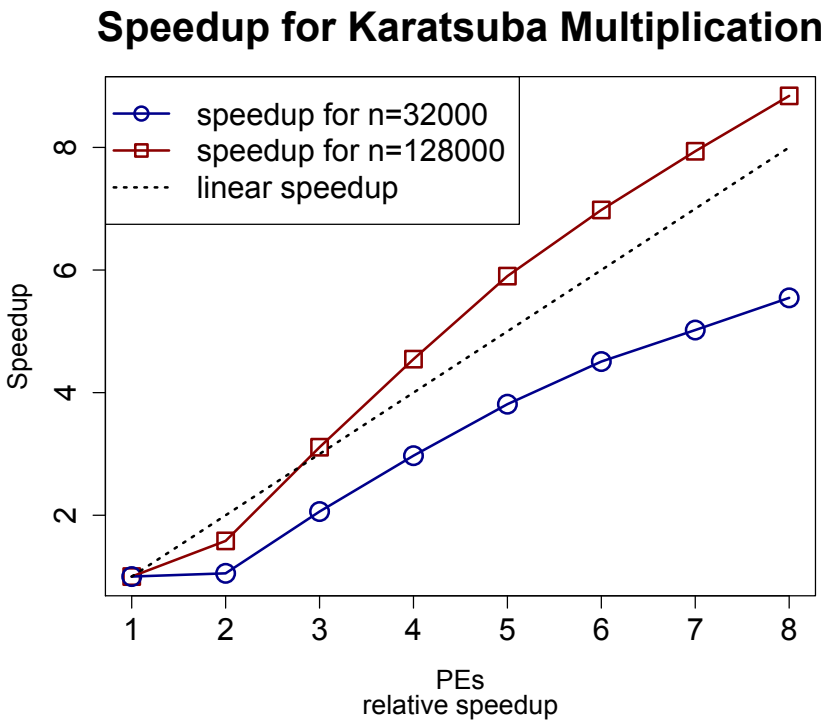


Figure 6.11: *Speedups of Karatsuba multiplication on* `sakania`.

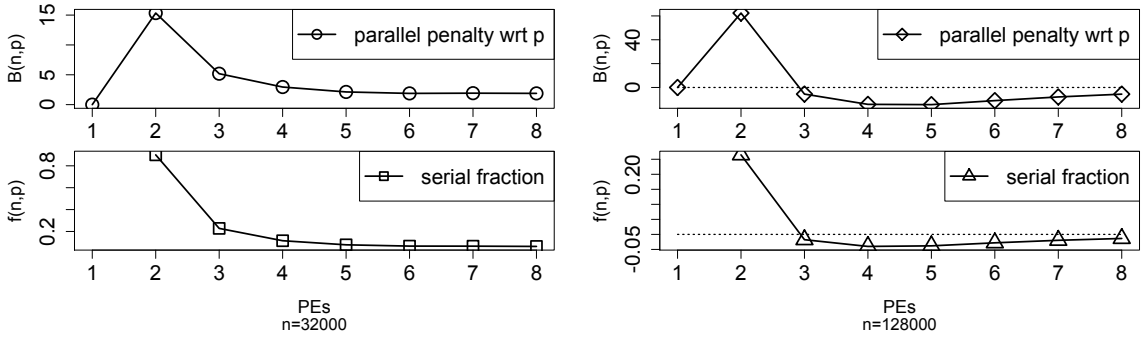


Figure 6.12: *Quality measures for Karatsuba multiplication. We use relative reference point.*

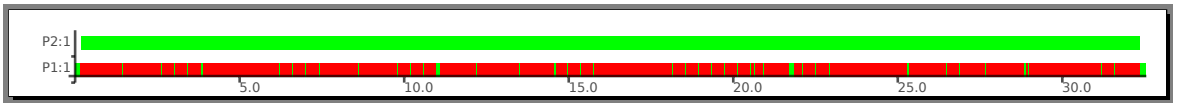


Figure 6.13: *Trace diagram for Karatsuba multiplication for input size* 32 000 *on two cores.*

6.3.4 Estimation of the Runtime

We present the estimation of the execution time of Karatsuba multiplication basing on [Lobachev and Loogen, 2010c] and Chapter 4. We estimate the sequential time and the parallel penalty w.r.t. task size n. Combining them, we obtain an estimation of the execution time for a not measured input size.

With GHC 6.12 we were unable to execute the sequential program for the input sizes from 84 000 onward. We used the sequential execution time in the range from 16 000 to 80 000 to predict the sequential execution times for larger input sizes. We measured the parallel execution time up to input size 128 000. We use these values as a reference for our estimation. The parallel execution time on 8 PE is predicted in the range 84 000–128 000 from the data in range 16 000–80 000.

We found various combinations of our usual methods more useful in different ranges of the input sizes. An overview is in Figure 6.15. We see that for each range of the task size to predict at least one

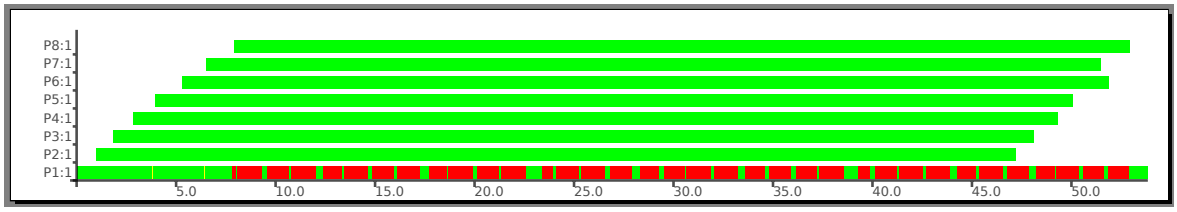


Figure 6.14: *Trace diagram for Karatsuba multiplication for input size* 128 000 *on eight processors.*

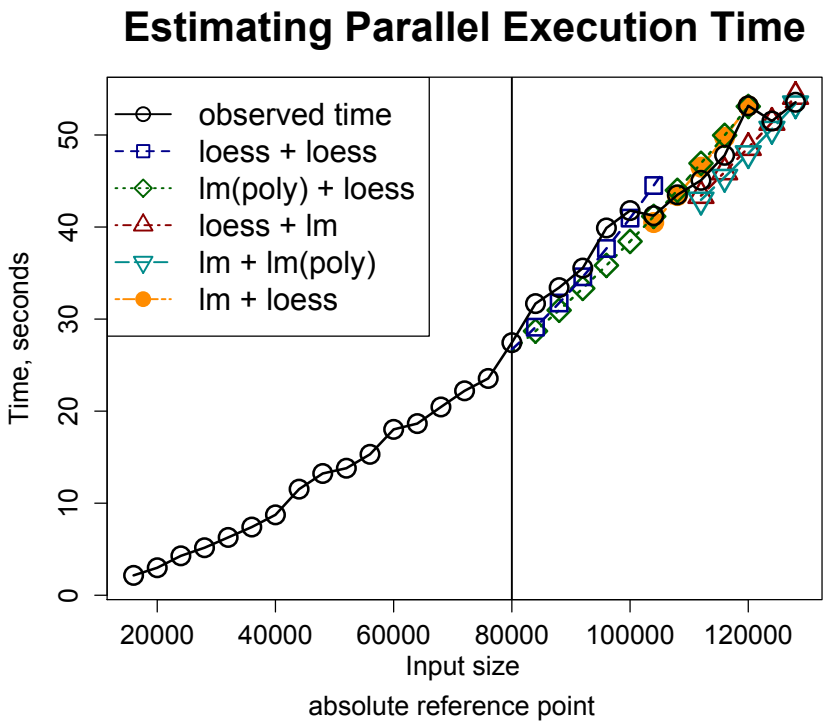


Figure 6.15: *Predicting the parallel execution time of Karatsuba multiplication.*

good combination of the prediction methods exists. The method lm for the sequential time, combined with loess for the parallel penalty, results a relative error of −0.0253% for an estimation of the execution time for the input size 120 000. Other remarkable values and methods are summarised in Table 6.2.

6.3.5 Conclusions

We have presented a parallel implementation of Karatsuba multiplication. We focused on `divConFlat` skeleton, which produced very good results, the relative speedup for the input size 128 000 was clearly superlinear. We investigated reasons for such a behaviour of our code. We performed the estimation of execution time of our implementation w. r. t. the input size. The results were very good, with remarkably small relative errors.

6.4 Fast Fourier Transform

The Fourier transform is a very significant procedure. An application of the Fourier transform to an input vector $\mathbf{x}$ results in an output vector $\hat{\mathbf{x}}$, such that the 'time' component in it is replaced with the 'frequency' component. To give an example: for a vector $[1, 2, 4, 2]$ the transform corresponds to a change from '2 is at positions 2 and 4' to '2 appears in the vector twice'. In digital signal processing, the input and output vectors are considered as signals, indexed with $\mathbb{Z}$, in special cases we consider only a window—a finite part of the signal [Lyons, 2004]. We are interested in the discrete transform,

n	statistical method for $T(n)$	statistical method for $B(n,8)$	rel. error, %
100 000	loess	loess	−1.958
104 000	lm(poly)	loess	−0.166
104 000	lm	loess	−1.792
108 000	lm	loess	−0.188
108 000	lm(poly)	loess	1.128
120 000	lm	loess	−0.025
120 000	lm(poly)	loess	−0.137
124 000	loess	lm	−0.306
124 000	lm	lm(poly)	−1.696
128 000	lm	lm(poly)	−0.292

Table 6.2: *An overview of good prediction methods for Karatsuba multiplication.*

the formal treatment of the continuous Fourier transform is far beyond the scope of this work. Here i denotes the imaginary unit, $\sqrt{-1}$. The Fourier transform commonly operates on complex-valued inputs and outputs, although there are real-valued variants. After an overview of the related work in Section 6.4.1 and some theory in Section 6.4.2 we consider the fast Fourier transform in Section 6.4.3. We abbreviate the latter name to FFT. Some practical results are in Section 6.4.4, and a better approach is in Section 6.4.5. The performance of these two FFT implementations is discussed in Section 6.4.6. We will show an application of the FFT for the polynomial multiplication in Section 6.4.7. We envision our FFT implementation as a divide and conquer one, although there are also other approaches to it, the keyword is 'filters and pipes' [Arjona et al., 1998]. The conclusions are in Section 6.4.8.

6.4.1 Related Work

The fast Fourier transform itself, and its parallelisation in particular, form a very large topic, which has been rigorously researched—and still is. There have been implications that already Gauß knew of the fast way to compute the transform [Heideman et al., 1984]. The modern history of the FFT begins with the publication by Cooley and Tukey [1965]. Since then, a lot of research on FFT has been done. Nussbaumer [1981] presents an overview of FFT and accompanying concepts. The transform is heavily used in signal processing, see, e.g., [Lyons, 2004]. There are also important applications in computer algebra. We cannot cover everything, but aim to show the research directions, essential for this work. A landmark is the paper by Schönhage and Strassen [1971] on using the FFT for multiplication, see also [Gentleman and Sande, 1966, Schönhage, 1982, Cantor and Kaltofen, 1991, Knuth, 1998, Yap and Li, 2000, von zur Gathen and Gerhard, 2003, Emiris and Pan, 2010]. For other than Cooley–Tukey approach to FFT, consult [Winograd, 1976]. Winograd defined closed formulae for the particular input sizes of the transform, while the composition of these formulae is similar to the 'classic' FFT. The state of the art of the FFT implementation is `fftw` [Frigo and Johnson, 2005].

The interest in a parallel FFT implementation was always high, *cf.* [Pease, 1962]. For the overview of the approaches on parallel FFT, consult [Duhamel and Vetterli, 1990, Grama et al., 2003]. Various implementations exist, e.g., [Gupta et al., 1994, Agarwal et al., 1994, Hammes et al., 1997, Bailey et al., 1991, Dmitruk et al., 2001, Grelck and Scholz, 2003, Crandall et al., 2004], alternative approaches are [Arjona et al., 1998, Jackson et al., 2004]. Grama et al. [2003] show two different ways of implementing the parallel FFT. The skeleton-based approach, which we will present in Section 6.4.5, is based on papers by Gorlatch, partially in a collaboration with Bischof [Gorlatch, 1996, 1998a, Gorlatch and Bischof, 1998].

The *convolution* is closely related to FFT. It accounts for the possibility of the fast multiplication, see Section 6.4.7. There are some subtle details, which account to the additional term in the complexity. The latter is $\mathcal{O}(n \log n \log\log n)$ for the fast integer multiplication. The pioneers were Schönhage and Strassen [1971]. A much more generic variant, working for polynomials with coefficients from an

arbitrary algebra was introduced by Cantor and Kaltofen [1991]. If we limit the coefficients to fields, supporting FFT, we can use the $\mathcal{O}(n \log n)$ method [Emiris and Pan, 2010].

6.4.2 The Theory of FFT: The 'Slow' Fourier Transform

Notation. We denote vectors in bold lowercase, e.g., $\mathbf{x}$. The indices of a vector of length n are $[1, \ldots, n]$—vectors' indices begin with a unity. Sometimes we write $\#\mathbf{x}$ for the length of the vector $\mathbf{x}$. We denote some subset of a vector $\mathbf{x}$ with elements with indices $[k, \ldots, l]$ with $\mathbf{x}_{[k,\ldots,l]}$. For example, a vector with all odd elements of $\mathbf{x}$ with $\#\mathbf{x} = n$ is $\mathbf{x}_{[1,3,\ldots,2k+1]}$ where $2k + 3 > n$. For a single, j^{th} element of $\mathbf{x}$ we write simply x_j.

Definition. We define the Fourier transformed values $\hat{\mathbf{x}}$, of an vector $\mathbf{x}$ of length n as

$$\hat{x}_k = \sum_{j=1}^{n} x_k \underbrace{e^{\frac{-2\pi i j k}{n}}}_{=:\omega^{jk}}. \tag{6.1}$$

The underbraced part of the above equation defines the n^{th} primitive *root of unity* to the power jk. See also Section 5.5.1 on page 68. Both $e^{2\pi i/n}$ and $e^{-2\pi i/n}$ are primitive roots of unity of order n.

The typical *naive* way to compute the Fourier transform is through a matrix-vector product of $\mathbf{x}$ with the so-called *Fourier matrix*. We denote here the n^{th} such matrix with $\mathbf{F_n}$. The few first Fourier matrices are [Strang, 1993]:

$$\mathbf{F_2} = \frac{1}{\sqrt{2}} \begin{pmatrix} i^0 & i^0 \\ i^0 & i^2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\mathbf{F_4} = \frac{1}{\sqrt{4}} \begin{pmatrix} i^0 & i^0 & i^0 & i^0 \\ i^0 & i^1 & i^2 & i^3 \\ i^0 & i^2 & i^4 & i^6 \\ i^0 & i^3 & i^6 & i^9 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} 1 & & 1 & \\ & 1 & & i \\ 1 & & -1 & \\ & 1 & & -i \end{pmatrix} \begin{pmatrix} \mathbf{F_2} & \\ & \mathbf{F_2} \end{pmatrix} \begin{pmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{pmatrix}$$

So, the pattern is easy graspable, the next matrix $\mathbf{F_{2n}}$ consists of a combination of $\mathbf{F_n}$, successively multiplied with powers of the primitive n^{th} root of unity ω and even-odd shuffle. We formalise it as

$$\mathbf{F_{2n}} = \begin{pmatrix} \mathbf{I_n} & \mathbf{D_n} \\ \mathbf{I_n} & -\mathbf{D_n} \end{pmatrix} \begin{pmatrix} \mathbf{F_n} & \\ & \mathbf{F_n} \end{pmatrix} \mathbf{S_{2n}},$$

where $\mathbf{I_n} = \operatorname{diag} \mathbf{1_n}$, $\mathbf{D_n} = \operatorname{diag} \boldsymbol{\omega} = \operatorname{diag} [1, \omega, \omega^2, \ldots, \omega^{n-1}]$ and $\mathbf{S_{2n}}$ is an $2n \times 2n$ even-odd shuffle.

We see, the Fourier matrix is easy decomposable in a divide and conquer manner. Essentially, it is exactly what Cooley–Tukey *fast Fourier transform* algorithm is about. But it does not construct the matrix $\mathbf{F_n}$ explicitly. Still, the formal base for the FFT is the Danielson–Lanczos lemma, which we have informally presented above for the Fourier matrices. With it we can split an n-size discrete Fourier transform of $\mathbf{x}$ of length n to two $n/2$-size transforms plus some additional operations. These correspond to an FFT of size two! We denote here application of the transform to some vector $\mathbf{x}$ as $\hat{\mathbf{x}}$. We will handle the FFT rigorously in the next section.

The *inverse* FFT is equivalent to FFT with some other parameters, and generally has the same characteristics and complexity. Its computation utilises the fact, that if ω is a primitive n^{th} root of unity, then ω^{-1} is also one. See [von zur Gathen and Gerhard, 2003, Theorem 8.13] for more. A single additional caveat is the factor n in the resulting output of the inverse transform, but even it can be handled beforehand by a careful prescaling of the 'forward' transform. To be short: we do not need to care of computing the inverse FFT too much, if we can compute the 'forward' transform.

Decimation in Frequency	
divide	$(\mathbf{l}, \mathbf{r})$ from $\mathbf{x}$
	$\mathbf{s} = \mathbf{r} + \mathbf{l}$
	$\mathbf{d} = \mathbf{r} - \mathbf{l}$
	$\mathbf{t} = \mathbf{d} \cdot \boldsymbol{\omega}$
recur	$\hat{\mathbf{s}}$ from $\mathbf{s}$
	$\hat{\mathbf{t}}$ from $\mathbf{t}$
combine	$\hat{\mathbf{x}}$ from $\hat{\mathbf{s}}, \hat{\mathbf{t}}$

Decimation in Time	
divide	$(\mathbf{o}, \mathbf{e})$ from $\mathbf{x}$
recur	$\hat{\mathbf{o}}$ from $\mathbf{o}$
	$\hat{\mathbf{e}}$ from $\mathbf{e}$
combine	$\mathbf{l} = \hat{\mathbf{o}} + \hat{\mathbf{e}}$
	$\mathbf{t} = \hat{\mathbf{o}} - \hat{\mathbf{e}}$
	$\mathbf{r} = \mathbf{t} \cdot \boldsymbol{\omega}$
	$\hat{\mathbf{x}} = (\mathbf{l}, \mathbf{r})$

Table 6.3: *Variants of FFT, in a short form.*

6.4.3 The Theory of FFT: The Fast Fourier Transform

An overview of FFT is in [Duhamel and Vetterli, 1990]. The basics are nicely explained in [Strang, 1993]. We can easily represent the Fourier transform as a divide and conquer algorithm. This was done by Cooley and Tukey [1965]. The result bears the name of *fast* Fourier transform and has reduced the complexity of the transform to $\mathcal{O}(n \log n)$ for input vector of length n. We show the complexity result later in this section. The explicit matrix construction from the previous section is rather harmful for an efficient algorithm.

Variants. An overview of this section is available in Table 6.3. For a divide and conquer version of FFT we divide our input vector $\mathbf{x}$ of length n into two halves: $\mathbf{l} = \mathbf{x}_{[1,\ldots,n/2]}$ and $\mathbf{r} = \mathbf{x}_{[n/2+1,\ldots,n]}$. Then we compute the element-wise sum $\mathbf{s} = \mathbf{r} + \mathbf{l}$ and difference $\mathbf{d} = \mathbf{r} - \mathbf{l}$. Finally, we component-wise multiply $\mathbf{d}$ with *twiddle factors vector* $\boldsymbol{\omega}$, resulting in $\mathbf{t}$. The twiddle factors vector is defined as $\boldsymbol{\omega} = [\omega^0, \omega^1, \ldots, \omega^{n/2-1}]$ with ω being a primitive root of unity of order n, e.g., $\omega = e^{-2\pi i/n} \in \mathbb{C}$. Note that the vectors $\mathbf{l}$, $\mathbf{r}$, $\mathbf{s}$, $\mathbf{d}$, $\boldsymbol{\omega}$ and $\mathbf{t}$ all have the length $n/2$. By the way: we do not need much additional memory. There is a variant of FFT, running *in-place*, however, it requires destructive assignment. Now we recursively compute the FFT of $\mathbf{s}$ and $\mathbf{t}$ with the FFT-transformed vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ as the result. The recursion ends at the singleton vectors, which are returned unaltered. Now, having $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ of length $n/2$ each, we need to combine them to $\hat{\mathbf{x}}$. We do it with *interleaving*: we take the first element of $\hat{\mathbf{s}}$, then the first element of $\hat{\mathbf{t}}$, then the second element of $\hat{\mathbf{s}}$, then the second element of $\hat{\mathbf{t}}$, and so on. More formally we can write for $k \in \{1, \ldots, n\}$

$$\hat{\mathbf{x}} = [\hat{s}_1, \hat{t}_1, \hat{s}_2, \hat{t}_2, \ldots, \hat{s}_{n/2}, \hat{t}_{n/2}] = \begin{cases} \hat{s}_{\lceil k/2 \rceil} & k \text{ odd} \\ \hat{t}_{k/2} & k \text{ even.} \end{cases}$$

We have just obtained the FFT transformed vector of length n. The FFT described above, was one of the multiple ways to compute the transform. It is called *decimation in frequency*, abbreviated: DIF.

There is another approach, called *decimation in time* (DIT). It is, in fact, the dual algorithm to the decimation in frequency FFT. The input vector $\mathbf{x}$ of length n is divided into two vectors $\mathbf{o}$ and $\mathbf{e}$ of the half size by interleaving $\mathbf{x}$. So, the vector $\mathbf{o}$ holds only odd and $\mathbf{e}$ only even elements of $\mathbf{x}$. Then we compute recursively the FFT transform of $\mathbf{o}$ and $\mathbf{e}$, resulting in $\hat{\mathbf{o}}$ and $\hat{\mathbf{e}}$. The combine step is more complicated. The first half $\hat{\mathbf{x}}_{[1,\ldots,n/2]}$ of the final result is the component-wise sum of the results of the half size transform. The second half is the component-wise difference of the same vectors, multiplied with the twiddle factors $\boldsymbol{\omega}$. The latter are the same as in the DIF case.

The two versions presented above, are the so-called *radix two* versions: we use binary divide and conquer. There are versions of FFT with higher radix values, generally described as r-radix FFT. If we use different radices in different recursion depths, we obtain *mixed-radix* FFT. In [Berthold et al., 2009c] we paid attention to the parallel *four radix* divide and conquer FFT implementation. We describe the algorithm below, see also the above paper and [Nussbaumer, 1981] for more details. We implement DIF four radix FFT. Let us divide the input vector into four parts. We number the parts through: $\mathbf{x_1}$ to $\mathbf{x_4}$. Please note: x_1 is the first element of $\mathbf{x}$, but $\mathbf{x_1}$ is a vector called so. Then we compute

Algorithm 12 The fast Fourier transform [von zur Gathen and Gerhard, 2003, Algorithm 8.14].

Require: For $n = 2^k \in \mathbb{N}$ with $k \in \mathbb{N}$, a polynomial $f \in R[x]$ of shape $f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + \cdots + f_0$ in form of its coefficients $[f_{n-1}, \ldots, f_0]$, a primitive root of unity ω in R of order n with corresponding powers $\omega, \omega^2, \ldots, \omega^{n-1}$.

1: **if** $n = 1$ **then return** f_0.
2: **end if**
3: Set

$$r_0 \leftarrow \sum_{j=0}^{n/2} (f_j + f_{j+n/2})x^j, \qquad r_1^* \leftarrow \sum_{j=0}^{n/2} (f_j - f_{j+n/2})\omega^j x^j.$$

4: Call the algorithm recursively to evaluate r_0 and r_1^* at the powers of ω^2.
5: **return** $[r_0(1), r_1^*(1), r_0(\omega^2), \ldots, r_0(\omega^{n-2}), r_1^*(\omega^{n-2})]$.

Ensure: The vector $\hat{f}$, the Fourier-transformed of f. It holds $\hat{f} = [f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{n-1})]$.

the FFT of them, resulting in $\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_4$. The results are combined to

$$\hat{\mathbf{x}}_1 = (\hat{\mathbf{y}}_1 + \hat{\mathbf{y}}_3) + (\hat{\mathbf{y}}_2 + \hat{\mathbf{y}}_4), \qquad \hat{\mathbf{x}}_2 = (\hat{\mathbf{y}}_1 - \hat{\mathbf{y}}_3) - i(\hat{\mathbf{y}}_2 - \hat{\mathbf{y}}_4),$$
$$\hat{\mathbf{x}}_3 = (\hat{\mathbf{y}}_1 - \hat{\mathbf{y}}_3) + (\hat{\mathbf{y}}_2 + \hat{\mathbf{y}}_4), \qquad \hat{\mathbf{x}}_1 = (\hat{\mathbf{y}}_1 - \hat{\mathbf{y}}_3) + i(\hat{\mathbf{y}}_2 - \hat{\mathbf{y}}_4).$$

The final result $\hat{\mathbf{x}}$ is just the concatenation of $\hat{\mathbf{x}}_\mathbf{j}$ for $j \in \{1, \ldots, 4\}$. Please observe that if we substitute for $\mathbf{x}_\mathbf{j}$ and $\hat{\mathbf{y}}_\mathbf{j}$ single elements (resulting in $\mathbf{x}_\mathbf{j} = \hat{\mathbf{y}}_\mathbf{j}$) for all $j \in \{1, \ldots, 4\}$, then we obtain the FFT for the vectors of length four. We reiterate, that this forms the essence of the FFT: we can divide a $k \cdot l$-sized Fourier transform into l-many k-sized Fourier transforms, which are divided and combined with l-sized transform.

Complexity. Based on [von zur Gathen and Gerhard, 2003], we state an algorithm for computing the FFT and show its complexity is $\mathcal{O}(n \log n)$ in the length of input. As we know, $R[x]$ denotes the algebra of univariate polynomials over R, where R is a commutative ring with unity, wherein primitive roots of unity ω of sufficient degree n exist. An intuition is: $R = \mathbb{C}$. For any positive $n \in \mathbb{N}$ primitive roots of unity of degree n exist in $\mathbb{C}$. We show the formal description of FFT, formulated for polynomials, in Algorithm 12. We identify the input vector with a polynomial with the coefficients being the elements of the input vector. In a similar manner, Wilf [2002] sees FFT as a change of the 'description' of a polynomial. Borodin and Munro [1975] present FFT of a polynomial as a multipoint evaluation and the inverse FFT as interpolation. The following statement is derived from [von zur Gathen and Gerhard, 2003, Theorem 8.15] and [Nussbaumer, 1981, Section 4.2.1]. Recall that lb is the binary logarithm.

Theorem 6.5 (Complexity of FFT)**.** *Given an integer n, being a power of 2, a ring R, and $\omega \in R$ a primitive root of unity of order n, Algorithm 12 computes the FFT of an input vector of size n over R using $n \operatorname{lb} n$ additions in R and $\frac{1}{2} n \operatorname{lb} n$ multiplications by powers of ω. So, we can compute FFT of input of length n in $\frac{3}{2} n \operatorname{lb} n \in \mathcal{O}(n \log n)$ operations in R.*

Proof. We leave the correctness proof to [von zur Gathen and Gerhard, 2003]. As for complexity: let $s(n)$ be the number of additions and $m(n)$ be the number of multiplications in R for the input size n. The costs for steps of the algorithm are shown in Table 6.4. We have $s(n) = m(n) = 0$, $s(n) = 2s(n/2) + n$, $m(n) = 2m(n/2) + n/2$. Unfolding the recursion we obtain $s(n) = n \operatorname{lb} n$ and $m(n) = (n/2) \operatorname{lb} n$.

Alternatively, using Proposition 6.3 twice, with $a = 2$, $k = 2$ both times, we can see that $s(n), m(n) \in \mathcal{O}(n \operatorname{lb} n) = \mathcal{O}(n \log n)$ for the logarithm in any base > 1. □

In [Nussbaumer, 1981] we find complexity statements for base field F, on top of which the complex number field $F(i)$ is constructed, wherein the actual FFT computation happens. Further, [Nussbaumer, 1981, Sections 4.2.1–4.2.2] devises the complexity of two radix FFT separate for both DIF and DIT cases and of four radix FFT. Some observations on the constants are made, but the complexity remains in $\mathcal{O}(n \log n)$.

Expr.	Step	Cost
$s(n)$	1	0
$m(n)$	1	0
$s(n)$	2	0
$m(n)$	2	0
$s(n)$	3	n
$m(n)$	3	$n/2$
$s(n)$	4	$2s(n/2)$
$m(n)$	4	$2m(n/2)$
$s(n)$	5	0
$m(n)$	5	0

Table 6.4: *Complexity of separate steps of FFT.*

```
fftDIF :: [Complex Double] → [Complex Double]
fftDIF [x] = [x]
fftDIF xs  = shuffle [fftDIF (ls +→ rs), fftDIF ((ls −→ rs) ·→ ws)]
             where (ls, rs) = splitAt (length xs `div` 2) xs
                   ws = map (w ^ ) [0..] -- list of powers of a root of unity
                   w = ... -- a fitting primitive root of unity

(+→) :: [Complex Double] → [Complex Double] → [Complex Double]
xs +→ ys = zipWith (+) xs ys -- invariant: xs and ys have the same length
-- −→, ·→ are defined similarly
```

Figure 6.16: *A non-skeleton based radix two FFT implementation (DIF).*

A simple, non-skeleton-based Haskell implementation of the FFT is in Figure 6.16. Vectors are represented as lists. The library function `splitAt` of type `Int → [a] → ([a], [a])` divides a list at the given position into two parts. The function `transpose` of type `[[a]] → [[a]]` from the module `Data.List` transposes a matrix, stored as nested lists. The library function `concat :: [[a]] → [a]` flattens the latter to a single list. The library function `zipWith` of type `(a → b → c) → [a] → [b] → [c]` applies a given binary function to corresponding elements of its two parameter lists and builds the list of the results of such applications.

6.4.4 Performance of the Divide and Conquer Skeletons

The performance results of the parallel divide and conquer FFT implementations, reported in [Berthold et al., 2009a,b,c], were not completely satisfying. For a better approach, see Section 6.4.5. In the current section we will report the aforementioned results and discuss reasons for them.

We needed to use list chunking (*viz.* Chapter 3) to reduce the communication overhead. Instead of modifying the parameter functions at skeleton instantiation, we designed a wrapper skeleton `chunkDC`. It is depicted in Figure 6.17 for the skeletons in the `dcF` class, i.e., of the type `DC' a b`. All parameter functions of a divide and conquer skeleton are wrapped with a pair of `unchunk`/`chunk` applications. The parameter functions `unchunk` and `chunk` should be inverse to each other.

The following run time experiments have been performed on the local network of 8 Linux workstations with Intel Core2Duo processors and 2 GB RAM connected by Fast Ethernet. We used the Eden implementation on top of GHC 6.8. We tested the standard Cooley–Tukey 2-radix FFT algorithm variants decimation in frequency and decimation in time with the distributed expansion and flat ex-

```
chunkDC :: Int                      -- ^ chunk size
        → (Int → [a] → [[a]])       -- ^ chunking function
        → ([[b]] → [b])             -- ^ unchunking function
        → DC' a b                   -- ^ input skeleton
        → DC' a b                   -- ^ result
chunkDC c chunk unchunk dcSkel trivial solve divide combine xs
 =  unchunk $  dcSkel (trivial  ∘ unchunk)
               ((chunk c)       ∘ solve    ∘ unchunk)
               ((map (chunk c)) ∘ divide   ∘ unchunk)
               ((chunk c)       ∘ combine  ∘ (map unchunk))
               (chunk c xs)
```

Figure 6.17: *Implementing chunking in a divide and conquer skeleton.*

pansion skeletons. Figure 6.18 shows typical trace visualisations and the run times obtained with the following parameters: input size 2^{20} (double precision complex numbers), chunk size 1500, recursion depth 4, and heap size 1500 Mb. Note that in this case we tuned the memory size, available to the program, with the command line option `-H`. Our experiments have shown that list chunking is essential for the performance of the distributed expansion skeleton (see left part of Figure 6.18 on the next page), but varying the chunk size did not have a great impact on the run times. We chose the chunk size to be 1500 in the run time experiments here.

The trace diagrams in Figure 6.18 reveal that the flat expansion skeleton (see right part of the figure) leads to a much better run time behaviour than the distributed expansion skeleton (left part). This is due to the good task balance in the worker processes which start immediately. Note that the skeleton co-locates one worker process with the main process on machine one (lowest bars). The communication overhead is low—only 80 messages were sent in both versions. With the flat expansion skeleton and the direct mapping trick, we can drastically reduce the input communication, i. e., an overhead of distributing tasks to the worker processes. Each worker receives the whole unevaluated task specification and evaluates its own part on demand. Contrarily, work distribution is slower with the distributed expansion skeleton, because the main process distributes the tasks to all worker processes. These are initially blocked, waiting for their tasks, and start working at different points in time. This leads to an inhomogeneous run time behaviour.

The decimation in frequency flat expansion 2-radix version (top right part of Figure 6.18) was the fastest of all 2-radix versions with 6.92 seconds. The reason is that the post processing in the master can be done very fast, because combining the results is a trivial shuffle, while the top level combining phase of the decimation in time version takes almost three quarters of the overall run time.

2-Radix vs. 4-radix. In theory, the 4-radix algorithm should be faster than 2-radix, as it reduces the number of multiplications and enables the sharing of some sub-results. In Table 6.5, we show the run times of the four possible combinations of 4-radix algorithm in comparison to 2-radix, with the same input size $2^{20} = 4^{10}$. The shape of the trace visualisations is very similar. Hence, we do not show the visualisations for the 4-radix programs. The run time measurements show that the 4-radix FFT behaves much better with the distributed expansion scheme than 2-radix, but the run times are bad in comparison to the flat expansion scheme. When using 4-radix with the flat expansion scheme, we get a modest improvement of the overall run time. The (non-shown) traces reveal that the worker times are almost halved when using 4-radix instead of 2-radix, but the postprocessing in the main process still dominates the overall run time. It is possible to further increase r and to counter the growing gap in acceptable input length with *mixed radix FFT*. In this case we would utilise large values for r as long as possible and then switch to smaller values, mostly 2 or 4. However, we have chosen a completely different approach, which is called 'distributable homomorphism'.

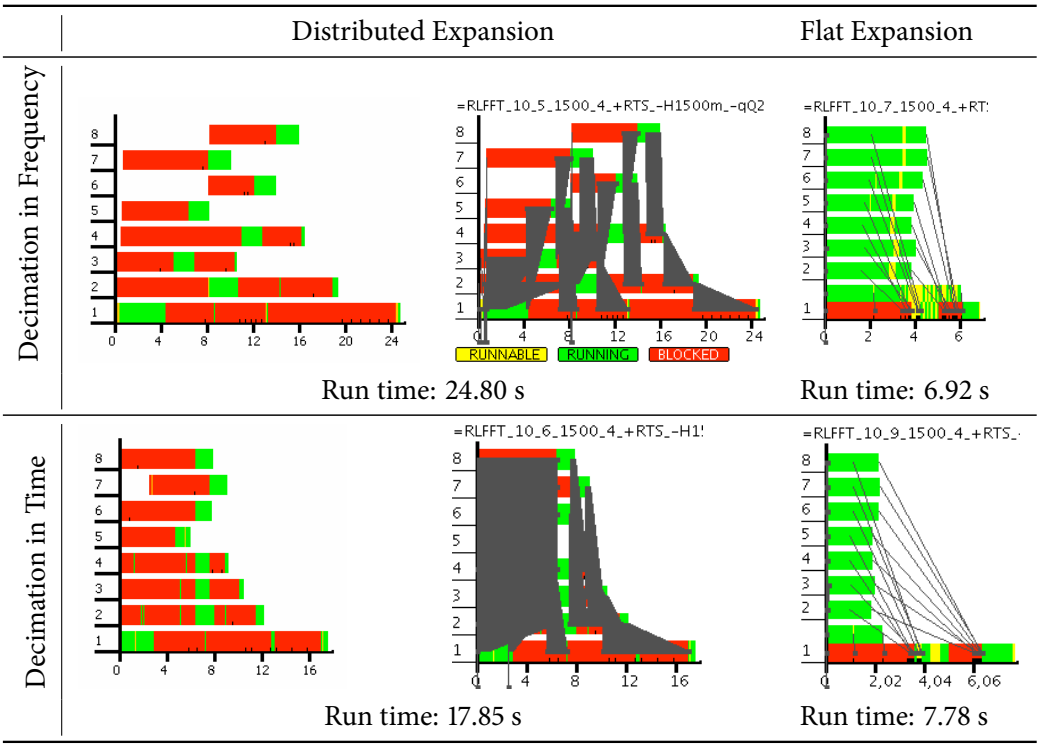


Figure 6.18: *Traces and run times of divide and conquer FFT approaches, without/with messages on local workstations. Trace visualisations are from [Berthold et al., 2009b].*

	2-radix	
	Distrib.	Flat
DIF	24.80	6.92
DIT	17.85	7.78

	4-radix	
	Distrib.	Flat
DIF	12.71	5.77
DIT	10.77	6.28

Table 6.5: *Run times of 2-radix vs 4-radix on eight local workstations. Time is in seconds. Data originates from [Berthold et al., 2009c].*

6.4.5 Distributable Homomorphism

In our FFT example, the distributed expansion skeleton does not scale well. Flat expansion skeleton scales better, but it has a drawback in the sequential `divide` and `combine`, where the first can be solved with various techniques, but the latter becomes the larger problem, the more tasks—and hence: PEs—we use. The generic reason for this problem is the large communication complexity of the FFT. A skeleton-based solution to relax the one-to-all communication patterns was suggested by Sergei Gorlatch in a sequence of papers of *distributable homomorphism* [Gorlatch, 1996] in the context of FFT [Gorlatch, 1998a, Gorlatch and Bischof, 1998]. The actual approach is widely known as the 'transpose FFT', *cf.* [Grama et al., 2003], alas it was presented there in a non-skeleton-based manner.

Recapitulate the key idea of Cooley–Tukey FFT scheme. Let us call two elements *k-distant*, if between which another $k - 1$ elements lie. So, in FFT we process 1-distant element pairs first, then 2-distant, then 4-distant, and so on. We see the distance is increasing, and the locality of data is fading out. Most FFT implementations perform a data recombination before the actual transformation to maintain better data locality. We *could* do such permutation in each step to maintain the 1-distance through the computation, but it is too costly. So, we place our data in a m-dimensional hypercube and do a permutation each few steps. Such permutation corresponds to the rotation of the hypercube. The latter brings the distant elements back to 1-distance. If we restrict our dimension to two, then we have to do only one rotation in-between, which corresponds to matrix transposition. A permutation in the beginning of the computation is recommended to establish data locality, but as we operate on unmod-

ified data, we can resort to the direct mapping trick. In a some sense, we perform an m-dimensional FFT, which simulates the single-dimensional transform.

We have implemented this approach in Eden. It resulted in a much better scalability than our previous approaches, alas for the price of higher memory usage. We had to store the positions of the permuted elements for the global FFT transformation.

How it works. We present the details, basing on [Berthold et al., 2009b] and its extended version [Berthold et al., 2009c]. The desired core functionality is

```
parMap (fft) ∘ transpose ∘ parMap (fft)
```

We abstract a bit and imagine two distinct unknown worker functions `f1` and `f2` in the parallel maps.

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

The input vector is divided into rows on a m-dimensional grid with side lengths $l = 2^k$. We consider the twodimensional case, so the grid is a matrix ($m = 2$) and the input vector length is $n = l^2 = 4^k$. We can separate three phases of the algorithm.

1. Preprocessing
   a) Permute the input
   b) Tag input elements with their position. This ensures the 'virtual global' length
   c) Split into rows
2. Central processing: 'local `fft3` ∘ global transpose ∘ local `fft3`'
3. Postprocessing
   a) Remove tags
   b) `shuffle`, i.e., `concat ∘ transpose`

In the step 1a the i^{th} input element is now at the position bitreverse(i). The latter function produces the bit-reversed binary representation, e.g., 1101_2 becomes 1011_2. Such bit-reversals can be also used in the Cooley–Tukey FFT implementations to improve data locality. Our application of the FFT does not insist on the bit-reversal. As for `fft3` from step 2, it will be defined subsequently. It is essentially the standard FFT, using the tagging information.

This approach is *very* similar to a multidimensional—in this case: twodimensional—FFT. The key difference is in the tagging. While for multidimensional FFT the twiddle factors are based on actual row lengths and positions, the local FFT worker function for distributable homomorphisms, called `fft3` here, applies the very same FFT transform as ever. But the twiddle factors are calculated, based on the *global* length and position information, which we include in preprocessing step 1b. We form at this point a triple `(position, length, data element)`. Hence, we achieve the computation of singledimensional FFT with an approach for a m-dimensional transform.

So, `fft3` is (almost) the usual sequential FFT. The `divide` function is not spectacular, it is a trivial list division. However, the `combine` function needs to use the global length and positioning information from the triples. In a contrast to that, the multidimensional FFT applies standard transform to the each of the rows as if it were a singledimensional transform for the current row only. In other words: an implementation of *multidimensional* FFT is a byproduct of the discussed approach. It is briefly discussed on page 117.

The skeleton. We have managed to map the singledimensional FFT problem (solvable with divide and conquer) to a multidimensional-like FFT problem, being a sequence of `map`s and hypercube rotations. In our case '$m = 2$', we have a scheme `map ∘ transpose ∘ map`. In [Berthold et al., 2009b] we have devised a skeleton for such tasks.

```
parMapTranspose :: Int → ([a] → [b]) → ([b] → [c]) → [[a]] → [c]
parMapTranspose np f1 f2 matrix = shuffle res
  where
   myProcs css   = spawn [ distr2d_f np f1 f2 rows
                         | rows ∈ unshuffle np matrix ] css
   (res, chanss) = myProcs $ transpose chanss

distr2d_fs ::  Int → ([a] → [b]) → ([b] → [c]) → [[a]]
           → [ChanName[b]] → ([[c]], [ChanName [b]])
distr2d_fs np f1 f2 rows theirChanNs
  = let (myChanNs, theirFstRes) = createChans np
        intermediateRes = map f1 rows
        myFstRes = unshuffle np $ transpose intermediateRes
        res = map f2 $ shuffleMatrixFracs theirFstRes
    in  (multifill theirChanNs myFstRes $ res, myChanNs)

-- combine n matrix fragments into one matrix
shuffleMatrixFracs :: [[[a]]] → [[a]]
shuffleMatrixFracs = ... -- implementation omitted
```

Figure 6.19: *Parallel map-and-transpose skeleton from [Berthold et al., 2009b].*

It is immediately obvious that a sequential `transpose` bears too much overhead. All the data have to be sent to the master PE, computing `transpose`, and then sent back to the workers. We need a distributed `transpose`. In [Berthold et al., 2009b] we have implemented this functionality with a dynamic channel communication *in* the skeleton. We utilised the primitives `new` and `parfill`, *cf.* Section 3.1.3. The core idea is to sequentially transpose the *channels* for obtaining the data. Figure 6.19 shows the implementation.

Details of implementation. Let us consider the implementation of the skeleton in Figure 6.19. The parallel `map` is easily defined. Let `np` be the number of PEs. The matrix rows are familiarly divided into `np` contiguous blocks with `unshuffle`. At the end the shape of the final result is reconstructed with `shuffle`. The processes are created with `spawn`, see Chapter 3. We call the process, creating all other processes, the 'master' or the parent process. In each process the function `distr2d_fs np f1 f2` consumes its portion of the rows and of the lazily received input (a row of `css`). This row is a list of channel names to establish a direct link to all processes in an all-to-all manner. Recall, a *channel name* is a not-yet-open channel. The channel list features a channel for the current process to talk to self for the sake of simplicity. Each process evaluates the function `distr2D_fs`. Thus the first `np` input channel names `myChanNs` are created. These are returned to the parent process in the second component of the result tuple. The parent process receives a matrix `chanss :: [[ChanName a]]` of channel names: `np` channel names from `np` processes. The matrix is transposed by the parent process and sent back row-wise. Hence, each process receives lazily `np` channel names called `theirChanNs` for sending data to all processes. In this manner, we can transpose the data matrix without sending and receiving it again from parent process.

After `map f1`, a process locally `unshuffle`s the columns of the locally transposed result rows into `np` lists. These are sent via the received input channels of the other processes by the `multifill` call. The latter function is an improved `parfill` for lists. The input for the second `map` is received via the initially created own input channels. The column fragments build rows of the transposed intermediate result matrix. The second `map f2` call provides us the final result of the child processes.

A derivative skeleton. The skeleton from above is quite large and not modular. A major improvement is possible with the remote data concept, *cf.* [Dieterle et al., 2010b] and Section 3.1.4. Using it

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

parTransposeRD :: (Trans b) ⇒ [RD [[b]]] → [RD [[b]]]

mapTranspose :: (Trans a, Trans b, Trans c)
             ⇒ ([a] → [[b]])
             → ([[b]]→ [c])
             → [[a]] → [[c]]
mapTranspose f g = fetchAll       ∘ (farmRD g)
                 ∘ parTransposeRD ∘ (farmRD f)
                 ∘ releaseAll
```

Figure 6.20: *A possible implementation of the map-and-transpose skeleton with remote data.*

and a plausible parallel `map`, say, `farmRD`, we would write the code in Figure 6.20 for the implementation of distributable homomorphisms with worker function `fft3`. Recapitulate: `releaseAll` maps `[a]` to `[RD a]` and `fetchAll` brings the data back to `[a]` from `[RD a]`. As always, `RD` stands for the 'remote data'. The function `farmRD` is a remote data version of a `farm`, `parTransposeRD` is a parallel transposition. We show their types in Figure 6.20. A further important point is that a proper implementation should obtain a distributed input and produce a distributed result. However, it suffices to remove in the code in Figure 6.20 the leading `fetchAll` and the trailing `releaseAll` to obtain the desired behaviour.

Unfortunately, we observe that the remote data version creates more processes than needed: the first parallel map and the second have distinct processes, all of them live quite long and start early. The actual fact of creating too many processes is not surprising: we have two `parMap`s and the processes, forming the first one, need to live on in order to deliver the processed data to the second group of the processes, forming the second `parMap`. What we discuss is the process life duration. We have not the expected behaviour: the first group of processes should die soon after the second group launches. The reasons are: firstly, the communication is streamed, i.e., it takes some time. Thus, the primary processes do not die early. Secondly, a demand exists on the weak head normal form of the result of the computation, thus forcing the instantiation of the second process group, as the `RD` constructor is always immediately available. Thus the secondary processes start early.

We conclude that the hand-crafted implementation from Figure 6.19 is more optimal that the remote data version from Figures 6.20. On the contrary, the latter version is much easier to implement. Note also: the code in Figure 6.20 is a skeleton, using another skeleton.

Note that both in the remote data case and in the monolithic skeleton case from [Berthold et al., 2009b], we envision the input data to be already distributed along the PEs and the result *remains* distributed. This solves a further major problem of our direct divide and conquer implementation: the enormous overhead for data collection in master PE. The current implementation leaves the FFT transformed data in a column-wise manner. In other words: we omit the postprocessing step 3b—`shuffle` of the results with subsequent result collection.

Multidimensional FFT. It is possible to represent a singledimensional discrete FFT with input length $n \cdot m$ as a twodimensional discrete FFT of size $n \times m$ if $n \perp m$. It is done with the Ruritanian mapping, which is also called Good's mapping [Nussbaumer, 1981, p. 125–127]. Another approach is the Winograd's nesting algorithm [Nussbaumer, 1981, p. 141]. Generalisations of these allow to map a one-dimensional transform of length $\prod_{i=1}^{d} p_i$ to a d-dimensional transform. However, the factors p_i should be coprime.

Multidimensional FFT is also important for applications, e.g., image processing. There are many different ways to perform a multidimensional FFT. The simplest case for two dimensions is the *column-row* method. In this case, a singledimensional FFT is applied to the columns and then to the rows of the two dimensional matrix. This method can be generalised to d dimensions of length n each, assuming a

Figure 6.21: *Trace of parallel FFT using map-and-transpose skeleton on a Beowulf cluster. Image from [Berthold et al., 2009b].*

grid of regular basis, the cost corresponds to dn^{d-1} times the cost of singledimensional FFT of length n. Even if we stay in two dimensions, we trade the overhead to compute a single FFT of length 2^{10} against the overhead to compute $2 \cdot 2^5$ FFTs of length 2^5. The formal representation of twodimensional FFT of length $n_1 \times n_2$ is

$$\hat{x}_{k_1,k_2} = \sum_{l=1}^{n_1} \sum_{m=1}^{n_2} x_{l,m} \omega_1^{(l-1)(k_1-1)} \omega_2^{(m-1)(k_2-1)},$$

with $\omega_1 = e^{-2i\pi/n_1}$, $\omega_2 = e^{-2i\pi/n_2}$, $k_1 \in \{1, \ldots, n_1\}$, $k_2 \in \{1, \ldots, n_2\}$. We see, the computation scheme for the twodimensional FFT is similar to the map-and-transpose FFT computation, i.e., it requires two `map`s of onedimensional transforms and a transpose in each dimension. We use the above parallel map-and-transpose skeleton also to implement the twodimensional FFT. Our implementation, started with the input size 2^{10}, produces the relative speedup 10.2 at 10 worker PE at local workstations with Intel Core2Duo CPUs. This corresponds to 11 full PE. The heap size was 700 Mb, the message queue size was 80 Mb, the chunking size was 4. The efficiency is 92.7%, calculated with full PEs. We evaluate the singledimensional FFT implementations next.

6.4.6 Performance

The trace visualisation (*cf.* Section 3.3) of our implementation of the transpose FFT is shown in Figure 6.21. This image was produced from a trace, generated on a Beowulf cluster, see pages 121–124 for details. The frequent yellow phases correspond to garbage collections—this implementation has an increased memory usage due to the extended location information for `fft3`. The communication phase is at 2.2–3.7 seconds. We need to further investigate, whether memory shortage has influence on the performance of our program.

We evaluate the program execution time measurements, we have obtained in [Berthold et al., 2009b] for the flat expansion divide and conquer skeleton and the monolithic map-and-transpose skeleton, both instantiated with a corresponding implementation of the FFT.

Evaluation: Networked Workstations

In this section we consider the results on local workstations, the next section handles the results, obtained on the Beowulf cluster. See page 36 for the specifications of the hardware used. In this section we use the relative speedup and the relative reference point for both the parallel penalty w. r. t. number of processors and for the serial fraction.

Flat expansion. Consider the 'normal' divide and conquer skeleton first. All the plots for it are shown in the left half of Figure 6.22. We used 4-radix FFT, the input size was 2^{20}, the depth was 3. In other words, 64 tasks were created. The heap size (`-H` parameter) was 750 Mb, chunking size was 512, the message buffer (`-qQ` parameter) was set to 20 Mb. The best speedup is 3.90 at 6 PE, increasing number of PEs does not improve the result, see Figure 6.22, top left. The vertical line marks the 6 PE mark, where the speedup is the best. We need to admit, that we used eight dual-core machines, i. e., 8 PEs are the maximum with single process placement. From this mark onward we have double process placement—each dual-core machine has two OS processes on it, each with its own Eden RTS. This is justified, as we still never have more than one OS process pro core, but this approach seems to affect the network communication, see Figure 6.22. A strange effect is that the maximal speedup is at 6, not 8 (full) PE. The probable reason lies in the communication overhead, which dominates the time in the master process and thus slows down the whole computation. We see that the speedup degrades at 6–9 PEs and not really recovers at 10–15 PEs. The growth of the speedup at 1–6 PEs is satisfying. We will see a better behaviour below, when we will consider the map-and-transpose FFT implementation on the same local workstations.

Interestingly, the shapes of the parallel penalty $B(n, p)$ w. r. t. p and of the serial fraction $f(n, p)$ are not quite the same. We present these two quality measures for the flat expansion skeleton in the lower left part of Figure 6.22. The parallel penalty is in the left middle part of the figure, the serial fraction is below it, in the left bottom part. We use the relative reference point. The parallel penalty increases slowly from 1 to 4 PE and is nearly constant for 4–6 PEs. From 6 to 9 PE we have fast growth, from 9 PE onwards the curve jitters and very slowly increases. The overall growth of the parallel penalty in the rage 1–6 PEs corresponds to the *slowdown* of the speedup, likewise, the slight decrease in 4–5 PEs matches the *increase* of the speedup in the same range. The fast growth maps to the decreasing speedup, while the unsteadiness of the parallel penalty curve at 9–15 PEs is reflected in the similar behaviour of the speedup curve. To give an example: a local minimum at 13 PE in the parallel penalty means a local peak at the same 13 PE in the speedup. The serial fraction plot in Figure 6.22, bottom left, looks differently, but tells us quite the same. Notably, it decreases a bit at 2–3 PEs and increases again at 4 PE. This is different to the ongoing increase of the parallel penalty at 1–4 PEs. We mark 5 and 6 PEs as a very good values, since the serial fraction there is small. We see a small peak at 4 PE, which corresponds to a minor speedup deficiency there. The decline of the serial fraction at 4–5 PEs is more visible than for parallel penalty and the increase at 6–9 PEs is more steep. Still, this behaviour gives us the same information, as the corresponding parts of the parallel penalty plot above. The minima at 10 and 13 PEs are more clearly visible with serial fraction. To interpret the serial fraction in the way Karp and Flatt [1990] meant it, the 5–6 PEs versions have a smaller serial fraction than the initial value at 2 PE, which is a hint for the 1 PE version to be slightly not optimal. Still, much more remarkable is the increase at 6–9 PEs and the jitter, but anyway high values at 9–15 PEs. These signal us, that the parallelisation, which optimal for up to 6 PE is no longer optimal for larger number of PEs. In other words, our flat expansion FFT implementation does not scale on networked local workstations. This confirms the impression, we have from observing the speedup curve in Figure 6.22, top left.

Map-and-transpose skeleton. Next we consider the map-and-transpose FFT implementation on the local workstations. The result is kept distributed, the input size is the same, meaning that 2^{10} tasks

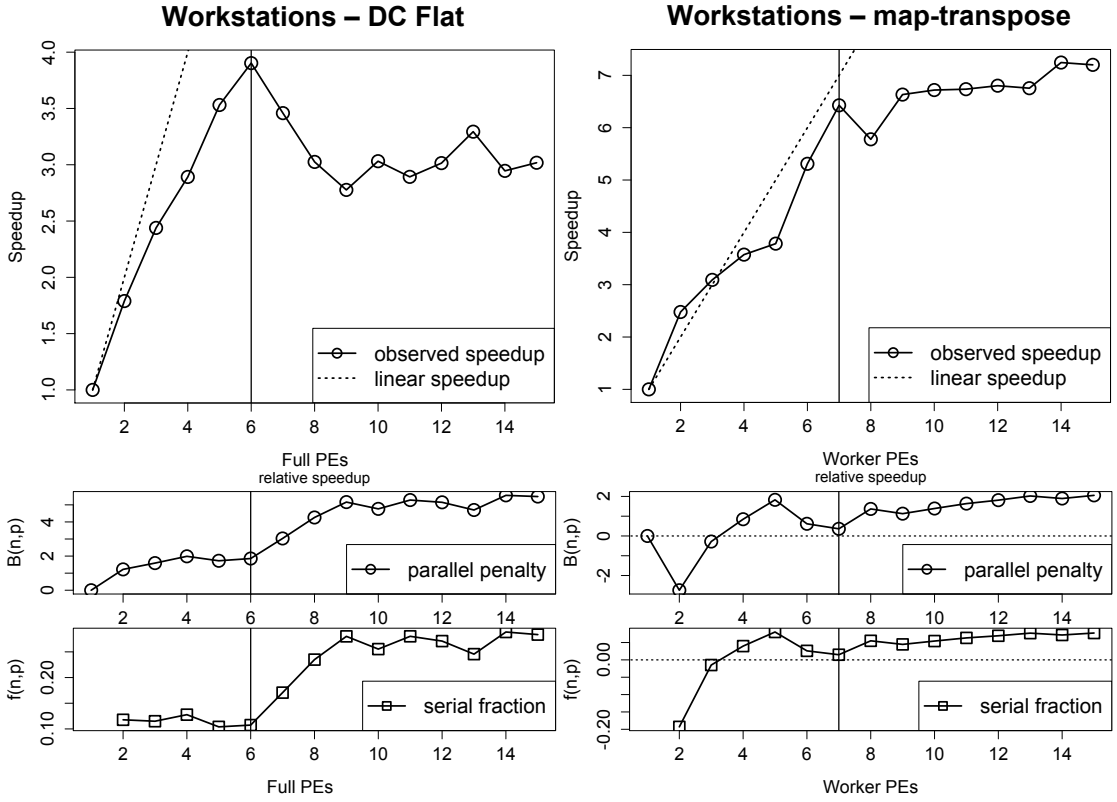


Figure 6.22: *Evaluation of FFT on local workstations.*

are processed. The heap size is 750 Mb. In this section under 'PE' we mean 'worker PE'. One PE is designated as the 'master'. The plots are shown in Figure 6.22, right. The top right part of this figure shows the speedup curve. We see the super linear speedup at 1–3 PEs, a speedup decline at 3–5 PEs, a fast climb up to 7 PE, and slowly increasing, but almost not improving values at 9–15 PE. At 8 PE the speedup degrades. The speedup values are 6.43 at 7 PE and 7.24 at 14 PE. The former is very satisfying, the efficiency is 91.82%, while the latter is not. Note, we use relative speedup w.r.t. the worker PEs.

To obtain some clues about the behaviour of our parallel program, we consider the usual two quality measures in the lower right part of Figure 6.22. The parallel penalty $B(n, p)$ w.r.t. p is negative between 1 and 3 PE, rises from 2 to 5 PE and falls again to a relatively small positive value at 7 PE. The negative part corresponds to the super linear speedup at 1–3 PE, the rising part matches the degrading speedup values and the decreasing positive past at 5–7 PEs stands for the very quickly recovering speedup in the same range. The vertical line in the speedup plot (Figure 6.22, top left) marks the still almost-linear speedup values at 7 PE. The same vertical line is in both quality measures' plots. We interpret this border as the full amount of single-placed workers: we used 8 dual-core machines, one is busy as the master, 7 workers are left. In a further development, we have a rapid decline of the speedup at 8 PE, as one worker is co-located. This means the increase of the parallel penalty w.r.t. p. We have a slightly improving speedup, which corresponds to slowly increasing parallel penalty at 9–15 PE. The decline of the parallel penalty at 8–9 corresponds to the recovered speedup behaviour. Note that aside the missing value at 1 PE, the serial fraction plot (i.e., Figure 6.22, bottom right) mimics the behaviour of the parallel penalty graph. The horizontal dashed line in both quality measures' plots marks the zero value. We see that the negative values correspond to the super linear speedup.

An interpretation of the serial fraction plot follows. We connect the negative values at 2–3 PEs with super linear speedup. From 2 to 5 PE the serial fraction curve is increasing, thus symbolising some increasing overhead. But at 6–7 PE it falls again to almost zero, hence we consider the parallelisation for 6–7 PEs as very satisfying. This corresponds to the good speedup value of 6.43 for 7 PE. The initial decline is probably related with too much work for too few PEs, another possible reason is the communication overhead between the workers. This is different from the usual suspect, the communication

overhead in master for sending (or receiving) the tasks (or the results). In this case the workers need to perform a distributed transposition and communicate in an all-to-all manner. As the volume of the communication is the same, the workers seem to have less overhead, when they communicate the same amount of information not to, e.g., three other workers, but to, e.g., six other workers. We would assume that such a behaviour should continue with increasing number of PEs, i.e., that the map-and-transpose skeleton scales well. However, it is not so in our observations on the local workstations, at 7–15 PEs. Still, we think, that the reason for such behaviour is our setup and not the drawbacks of the skeleton or of the implementation. Namely, from eight workers onward, we have double placement—at least one dual-core machine hosts two processes in this case. As both of them share the same network connection, the communication on the doubly-placed machine is affected. This could explain the speedup decrease we see at 8 worker PE. Further adding more workers helps a bit to cope with the communication overhead, but does not produce as good speedups as before. The slow overall increase of the parallel penalty from 7 to 15 PEs confirms our assumption. The minor declines of the parallel penalty at 9 and at 14 PE are probably related with some positive effects of scaling, a short-term victory of the increased number of workers against the inter-worker communication overhead.

Overall, we see a quite good behaviour of the skeleton for low number of PE, which is not so nice for larger numbers of PE. However, we consider this drawback to be an artifact of the regarded setup. To look into this problem and to observe scaling at larger number of PE we consider experiments on a Beowulf cluster next.

Evaluation: Beowulf Cluster

The following experiments were performed on the Beowulf cluster of the Heriot-Watt University.

Flat expansion. Contrary to the above experiment on the local workstations, we use here the flat expansion 4-radix divide and conquer FFT with parallel depth two and not three. The depth 3 produces a mediocre speedup of 1.63 on 25 PE with best overall speedup being 2.41 on 4 PE. So, we consider the depth 2 experiment in a more detail. As only 16 tasks are available, no observations beyond 16 worker PEs are performed. The heap size (`-H` parameter) was 330 Mb, the message buffer (`-qQ` parameter) was 20 Mb. The chunking size was 512.

Consider the speedup graph in Figure 6.23, top left. We show the relative speedup. It is almost linear at 1–3 PE, a small setback occurs at 3–6 PE, the speedup *decreases* at 6–8 PE. We see the best overall speedup of 5.57 at 9 PE, then another decline occurs. As only 16 tasks are issued, one possible reason for this lies in the task distribution across the PEs. We consider it in Table 6.6. 'Full PEs' means the total number of processing elements, 'worker PEs' stands for the PEs actually doing some work and not overseeing the task distribution. As in previous chapters, 'full rounds' and 'remaining part' stand appropriately for how often the worker PEs have one task each and for the amount of the tasks remaining in the last, incomplete round (provided that all tasks need the same time to compute). The 'total rounds' row is self-explanatory. 'Idle in last round' shows the amount of worker PEs, which are not working in the incomplete round. The quotient of this figure and of the number of worker PEs is the 'slack-off'. We see that, e.g., after some significant slack-off at 5–7 worker PEs, eight worker PE—naturally—have zero slack-off. This explains the 'jump' in the speedup at 9 full PE, which we see in Figure 6.23, top left. According to Table 6.6, the next zero slack-off configuration is 16 worker PE. At this mark (17 full PE) we see a modest recover from all the problems faced at 10–16 full PEs.

The theoretic considerations from above correspond to the issues we observe in both quality measures for the flat expansion FFT computation on the Beowulf cluster. We show the parallel penalty $B(n,p)$ w.r.t. p and the serial fraction $f(n,p)$ in the bottom left part of Figure 6.23. The relative reference point is used. There, small slack-offs at 3 and 9 full PEs correspond to the local minima in both plots. Further, it is very interesting to observe the correspondence of the increase both of the parallel penalty and of the serial fraction at 9–16 full PEs with the ever-increasing slack-off values in Table 6.6. We interpret the values of the serial fraction at points 3 PE and 9 PE as a sign of absent problems in the parallel implementation *at said configurations*. Otherwise, as the increase at 9–16 full PEs can be explained with bad task distribution, the only real problem which can be read from the serial fraction

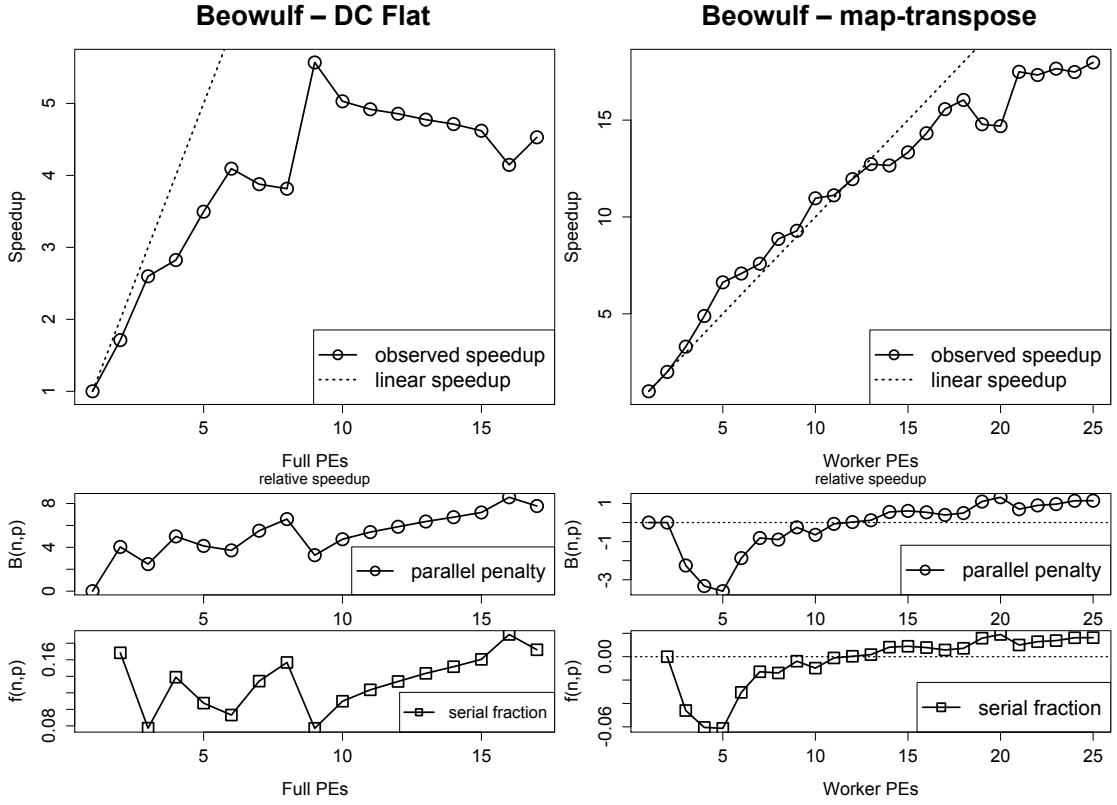


Figure 6.23: *Evaluation of FFT on Beowulf cluster.*

full PEs	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
worker PEs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
full rounds	16	8	5	4	3	2	2	2	1	1	1	1	1	1	1	1
remaining part	0	0	1	0	1	4	2	0	7	6	5	4	3	2	1	0
total rounds	16	8	6	4	4	3	3	2	2	2	2	2	2	2	2	1
idle in last round	0	0	2	0	4	2	5	0	2	4	6	8	10	12	14	0
slack-off, %	0	0	66.7	0	80	33.3	71.4	0	22.2	40	54.5	66.7	76.9	85.7	93.3	0

Table 6.6: *Task distribution for depth two flat expansion FFT computation.*

plot is its high value at 17 full PE, where the task distribution should not harm anymore. It is not clear to us, why the speedup curve does not make any high jumps at 17 full PE, as it does at 9 full PE. A possible reason is the communication overhead in the master process; however, we cannot see such a problem in the serial fraction plot. Still, we have seen above that other our implementations of the flat divide and conquer FFT also have scalability problems, so let us consider a better way to implement the parallel FFT.

Map-and-transpose. We discuss here the parallel map-and-transpose implementation of FFT on a Beowulf cluster. The input size is again 2^{20}, we used the memory tuning parameter `-H` to set the heap size to 330 Mb. All the plots for this implementation are in Figure 6.23, right. Below 'PE' stands for 'worker PE'. Because of large memory footprint and because of communication issues with self we did not produce the single worker PE version of the map-and-transpose FFT time measurement. Instead, we ran the program on two worker PE and assumed the perfect speedup. In other words, we took the time on two PE and doubled it to obtain the 'sequential' time against which we perform all other computations. We can confirm the choice of the linear speedup value for 1–2 PE with a look to Figure 6.22, top right. It shows a super linear speedup between 1 and 2 worker PE on local workstations.

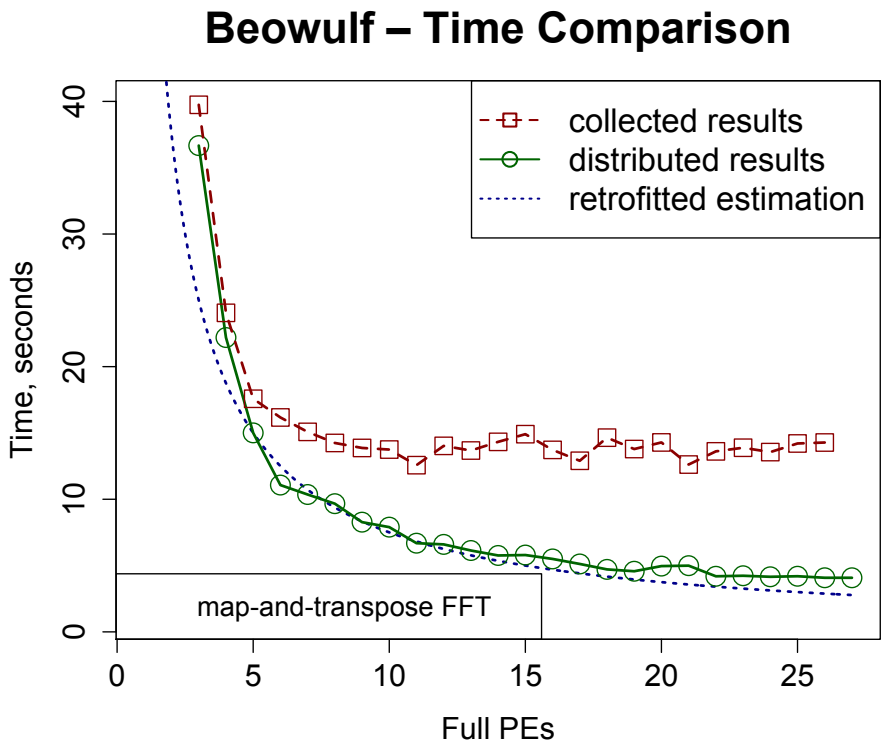


Figure 6.24: *Distributed result map-and-transpose vs. gathered result. Time is in seconds, measured on Beowulf cluster. Data from [Berthold et al., 2009b].*

We will consider the values of speedup, parallel penalty w. r. t. number of worker PEs and serial fraction, obtained in a manner, discussed above. The 'pure' PE–time relationship, without any fictional data points, is shown in Figure 6.24 as the distributed result method. Additionally this figure shows the times for the same computation with the result collection in master PE. We shall discuss this figure, especially the 'retrofitted' line later, after we have considered the both quality measures.

The speedup plot on Beowulf cluster, seen in Figure 6.23, top right, is good. We see that a larger part of the speedup curve is super linear, but there are also some setbacks in the right part of the curve. The map-and-transpose implementation processes large amount of tasks (2^{10} to be exact), so we do not expect issues with task balancing. We have the speedup 6.62 at 5 PE, 10.97 at 10 PE, a crash to 14.69 at 20 PE, and the total maximum of 17.98 at 25 PE. However, we need to investigate whether these results emerge due to our choice of the numerator in the speedup formula or because of some other positive side effects. The efficiency is 99.7% at 11 full PE.

The parallel penalty plot w. r. t. number of PE in the lower right part of Figure 6.23, is constantly zero at 1–2 worker PEs (due to the choice of 1 PE time), drops rapidly at 2–5 PEs, then rises again slowly with few small setbacks at 5–13 PEs. At 12 PE the parallel penalty is positive again, and the speedup is almost linear: 11.95. The increase of the curve at 14–15 PEs corresponds to the point where the speedup degrades. The parallel penalty decreases slowly at 15–19 PEs only to see a large peak at 19–20 PEs. At these values the speedup has a major setback, which we cannot explain. The parallel penalty value at 21 PE decreases again, compared to 20 PE, which is a slight increase compared to 15 PE. The values at 21–25 worker PEs increase slowly, which corresponds to slowly-than-linear increasing speedup at these points. We think that the decline at 2–5 worker PEs can be explained by the memory shortage in the workers. The nodes of the Beowulf cluster did not have much memory, hence the amount of work pro PE was probably too large for these configurations. From 5 PE onward, the parallel penalty generally increases. The derivative of the speedup declines fast at 5–7 PEs. The zero mark at the parallel penalty corresponds to the linear speedup, we see this clearly at 1–2 PEs and 11–13 PEs.

The serial fraction plot in Figure 6.23, bottom right, is very similar to the parallel penalty graph above it. The sole exception is the absent value at 1 PE due to the nature of the serial fraction. The serial fraction values have a similar relation to the zero as the values of the parallel penalty—serial fraction is

negative if the speedup is super linear. Let us discuss the serial fraction plot in a more detail. The falling values at 2–5 PE signal a bad sequential version. We conjectured this above with the probable culprit being the memory shortage. An inspection of the trace diagram in Figure 6.21 on page 118 supports this assumption. We see four major garbage collections there. From 5 worker PE onward the serial fraction is majorly increasing. We disregard the few minor exceptions here. Said behaviour of the serial fraction indicates an increasing serial part in the computation. It could be the communication from master to the workers. Such problems in the context of the master-worker schemes were considered, e. g., in [Berthold et al., 2008].

Now after we have seen the parallel penalty w. r. t. number of PE and the serial fraction plots in Figure 6.23, bottom right, let us consider the PE–time relationship in Figure 6.24. It shows the same execution times of the map-and-transpose FFT program we discussed before, but also some further data. The plots are shown for the full number of PEs, not for the worker PEs as above. The green straight line shows the times for the distributed result variant. In this case the final results remain distributed across the PE, `shuffled`. This is the version, we considered above. The dashed red line shows the times for the same program in a collected result variant, here the final results are communicated back to the master. As we talk about 2^{20} complex doubles, the communication overhead shows its effect and this version is clearly handicapped. Finally, the blue dotted line shows an attempt to fit values for an ideal speedup to the shown PE–time relations. As the decreasing serial fraction at 2–5 worker PEs—i. e., 3–6 full PEs—indicates the drawbacks of the sequential reference time, used in Figure 6.24, right, we used the time at 5 full PE as a base for this retrofitting. The reason for the choice of 5 full PE is: the serial fraction at 3–4 full PEs is falling, whereas for 5–6 full PEs it is nearly constant and from 6 PE onward the increase begins. We aim for the smallest PE number with non-falling serial fraction. Of course, said curve does not correspond to the real speedup plot of Figure 6.23, top right. It is rather a vision of the execution time values for an ideal, linear speedup.

Estimation of runtime. We estimated the execution time of map-and-transpose FFT on 25 worker PE of the Beowulf cluster with our approach from Chapter 4. We obtained the parallel execution time up to −2.33% exact, using the loess method for $B(n, p)$ w. r. t. p and the equation (4.1).

Conclusions. We have discussed two different implementations of the parallel FFT on two different hardware platforms. We have seen that the flat expansion skeleton does not scale well in case of its FFT instantiation. We have also seen that the map-and-transpose FFT implementation produces good and reassuring performance results. However, as both Figure 6.22, right, and Figure 6.24 reveal, an increased communication overhead can hinder the optimal performance of this approach. This supports the need in keeping the data distributed. In said two cases, two issues, correspondingly: the double PE placement and the complete collection of the results, were reasons for the degrading performance. If none of these happen, the performance of the skeleton is as good as seen in Figure 6.23, right. Hence, a scheme with the distributed transposition, where data may stay distributed at the beginning and at the end of the transform, benefits the parallel FFT significantly.

6.4.7 FFT-based Polynomial Multiplication Method

The best known to us multiplication method is based on the fast Fourier transform and utilises the fact, that polynomial multiplication is in fact a convolution. An answer to the question ‘What do FFT and multiplication have in common?’, the *convolution* is a term from functional analysis, describing an ‘overlapping degree’ of two functions. In a more common for us way, for two vectors $\mathbf{x}$ and $\mathbf{y}$, a convolution $\mathbf{x} * \mathbf{y}$ is a product of each element of the first vector with each element of the second one—*not* element-wise! With $n = \#\mathbf{x} = \#\mathbf{y}$ it holds

$$(\mathbf{x} * \mathbf{y})_k = \sum_j x_j y_{n-j+1}$$

Algorithm 13 FFT-based multiplication.

Require: vectors $\mathbf{x}$, $\mathbf{y}$ of length n.

1: Compute the FFT transformed of the input vectors, resulting in $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$.
2: Multiply them component-wise, resulting in $\hat{\mathbf{z}} = \hat{\mathbf{x}} \cdot \hat{\mathbf{y}}$.
3: Complete the inverse FFT of $\hat{\mathbf{z}}$, resulting in $\mathbf{z}$.

Ensure: product $\mathbf{z} = \mathbf{x} * \mathbf{y}$.

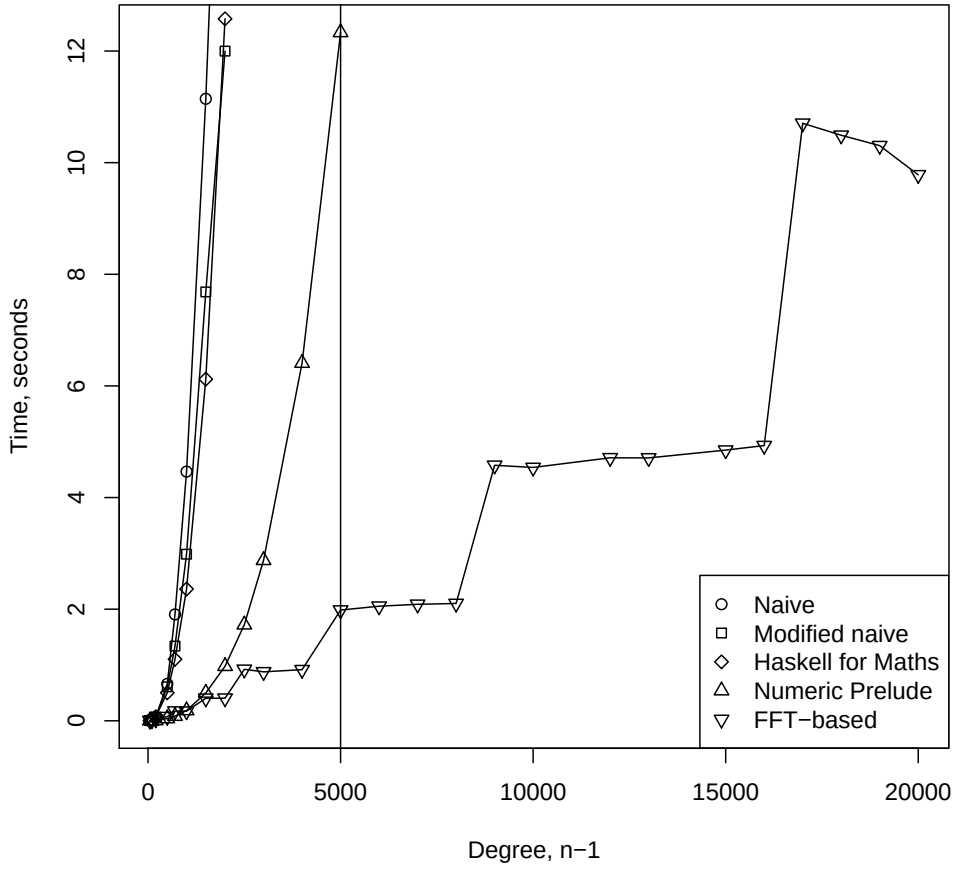


Figure 6.25: *Multiplication of dense univariate polynomials. Plot from [Lobachev and Loogen, 2008].*

for all $k \in \{1, \ldots, n\}$. This is the polynomial multiplication of two polynomials with coefficients $\mathbf{x}$ and $\mathbf{y}$ [Knuth, 1998]. The next theorem states, that the Fourier transform 'untangles' the convolution. See [Nussbaumer, 1981, Section 4.6] or [von zur Gathen and Gerhard, 2003, Lemma 8.11] for the proof.

Theorem 6.6 (Convolution theorem)**.** *For two complex-valued vectors of an equal size, the Fourier transformation of their convolution is the point-wise product of their Fourier transformed. In sign:*

$$\widehat{\mathbf{x} * \mathbf{y}} = \hat{\mathbf{x}} \cdot \hat{\mathbf{y}}.$$

This enables us to reach the fastest multiplication algorithm currently known. As the element-wise product is $\mathcal{O}(n)$ for input length n, all we need to do for a very fast multiplication is in Algorithm 13. We need to apply the FFT three times, all subsequent operations are dominated by the asymptotic complexity of the FFT, thus the asymptotic complexity of Algorithm 13 is $\mathcal{O}(n \log n)$.

A subtle detail of the implementation is that FFT-based fast convolution, transforming polynomials in $R[x]$ of degree $< n$, produces a result in $R[x]/\langle x^n - 1 \rangle$. To relax this drawback it suffices to use

some fitting transform of size $\geq 2n$. A more important and hideous detail is the requirement to the base ring R to have primitive roots of unity of any order n. After von zur Gathen and Gerhard [2003] such rings 'support FFT'. Only in such rings the complexity of the fast convolution is $\mathcal{O}(n \log n)$ for the input vectors of size n [von zur Gathen and Gerhard, 2003, Emiris and Pan, 2010]. It is possible to find *some* roots of unity in fitting residue rings [Geddes et al., 1992]. There is a way to cope with this drawback in an arbitrary commutative ring, see [Cantor and Kaltofen, 1991]. It adds a factor of $\log \log n$ to the complexity of the multiplication [Schönhage and Strassen, 1971, Borodin and Munro, 1975, von zur Gathen and Gerhard, 2003, Emiris and Pan, 2010]. The computation happens not modulo $x^{2n} - 1$, as decided above, but modulo $x^{2n} + 1$ for $2n > \deg fg$ for factors f and g. This approach is called a 'negative wrapped convolution'. We do not focus on it further and consider polynomials over $\mathbb{C}$, where naturally all primitive roots of unity exist. We disregard hereby the problem of the representation of any integer in the finite arithmetic of $\mathbb{C}$ with satisfying precision.

Although there are plenty more generic FFT algorithms, for instance, Rader's algorithm (which works for prime sizes) or Winograd's FFT algorithm (which is defined for many small transform sizes), the sizes, with which the classical Cooley–Tukey implementation can cope, are bound by a power of r. We considered $r = 2, 4$. As we need to fill up the input vectors with zeroes from length n to $2n$ to obtain the full result, we choose a simpler method: we fill the vectors up to the next power of two larger than $2n$. Thus we can use the standard two-radix Cooley–Tukey FFT to obtain the full result. We focus on algorithms and their parallelisation in this thesis, hence we decided not to consider advanced domain construction, needed for the methods of [Schönhage and Strassen, 1971, Cantor and Kaltofen, 1991].

We still need to be cautious. The current implementation of the FFT works for complex doubles, so not for all possible coefficients of the result a fitting approximation exists. However, we stick to our prototype implementation. We use it on polynomials in $\mathcal{Z}[x]$ with $\mathcal{Z} \subset \mathbb{Z}$, i.e., with coefficients limited not to extend the exact integer representation in the mantissa of a double precision float. Thus we model the polynomial multiplication over a field, which supports FFT.

Integers. The multiplication of large integers is similar to the polynomial multiplication. With a representation of integers in base p, we have a simple mapping between integers and polynomials in variable p, where p is the base of the chosen integer representation. Using polynomial evaluation with Horner scheme, it is possible to evaluate these artificial polynomials at p, obtaining an integer. Summarising, it is possible to multiply integers with an algorithm similar to Algorithm 13. However, there are also better approaches. Von zur Gathen and Gerhard [2003] describe a 'three primes' residual FFT multiplication algorithm, capable of processing very large integers. This variant is theoretically limited by an upper bound, but it is much larger than available memory. Essentially, this algorithm uses nine FFT computations with roots of unity *modulo said primes*. Then, the result needs to be recovered from the residual representation. For our approach towards such reconstruction see Chapter 7. For further discussion of three primes FFT multiplication see [von zur Gathen and Gerhard, 2003] and [Cesari and Maeder, 1996a].

Thus, the practical computational complexity of the multiplication of two large (but bounded) integers is $\mathcal{O}(n \log n)$ in their maximal bit-length n [von zur Gathen and Gerhard, 2003].

Relevance. Based on [Lobachev and Loogen, 2008], we show some results on the performance of the FFT-based polynomial multiplication. We used AMD Athlon 64 X2 4000+ CPU with 1 Gb RAM running Linux and GHC 6.8. For the same degree, each test was run ten times and the mean value of measured execution time has been determined. We utilised standard Haskell lists for representing the polynomials. We tested four different $\mathcal{O}(n^2)$ implementations:

1. Our own naive implementation with lists of integers
2. Our naive implementation, modified à la Numeric Prelude
3. The implementation from *Haskell for Math* [Amos, 2007]
4. The implementation from *Numeric Prelude* [Thurston et al., 2010]

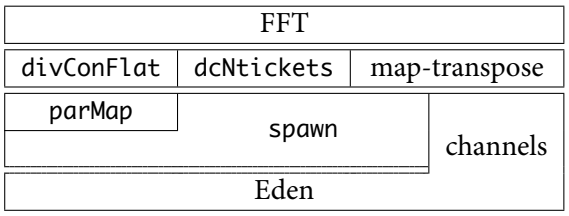


Figure 6.26: *The skeletal implementation of the FFT.*

and our sequential FFT-based implementation. We multiply two dense univariate $(n-1)$-degree polynomials with random coefficients. The coefficients are random signed integers in $\mathcal{Z}$. The result is shown in Figure 6.25. The Numeric Prelude implementation is much better than other naive implementations, which show similar run times. The probable reasons are the some clever inlining and usage of standard higher-order function on lists, which are optimised using stream fusion [Coutts et al., 2007].

We see, a sub-quadratic method for polynomial multiplication is definitely superior. It is clearly visible in Figure 6.25. Our current FFT algorithm is relying on the fact, that the length of its input is a power of two, see the same figure. The rapidly ascending lines correspond to the times of quadratic methods. Unfortunately, we have no explanation for the decreasing execution times of the FFT-based algorithm for $16\,000 \leq n \leq 20\,000$. The vertical line at 5 000 designates the *de facto* limit of the naive methods. This concludes our presentation of the FFT.

6.4.8 Conclusions

We have developed and presented parallel implementations of the fast Fourier transform, abbreviated FFT. In the beginning, we utilised the divide and conquer skeletons with distributed and flat expansion. However, we saw the communication overhead to be very significant. Thus, we have implemented a special map-and-transpose scheme for the parallel FFT [Gorlatch, 1998a, Grama et al., 2003]. We developed a monolithic skeleton and tested it on a Beowulf cluster. We have also briefly considered a composable implementation of the same map-and-transpose skeleton and an approach to multidimensional FFT. We presented four extensive evaluations of the quality measures of two our FFT implementations on two different platforms. This way we obtained significant insights into said implementations. We have also discussed how to use FFT to implement multiplication.

An overview is available in Figure 6.26. It is the 'building blocks' metaphor, the abstraction level increases from bottom to top. The global view on this chapter's results is in Figure 6.41 on page 142.

6.5 Matrices

Matrices are a further very popular structure in mathematics. Intuitively perceived as a vector of vectors, a $n \times m$ matrix $\mathbf{M}$ over a field F is denoted with $\mathbf{M} \in F^{n \times m}$. We consider briefly the approaches to matrix representation on a computer next. Section 6.5.2 discusses the fast matrix multiplication. Section 6.5.3 presents some performance results of the skeleton instantiation with the Strassen algorithm. Section 6.6 presents an alternative approach towards the parallelisation of Strassen matrix multiplication. Combined, these sections provide two implementations of the fast parallel matrix multiplication.

6.5.1 Representations

Multiindexed list. We can represent a matrix as a multiindexed vector. This is the approach known to us from Haskell's `Data.Array` library. We simply use pairs of integers as indices and store the matrix in a *single* list, be in dense or sparse, e. g.,

```
type SparseMatList a = SparseList (Int, Int) a
```

However, this approach has a quite drawback. Most matrix algorithms are not laid out for operations on single matrix elements, but rather on some subsets of the matrix. Options for natural subsets of a matrix include rows or columns and smaller matrices. More on that is in Section 6.5.2. The well-known classification of matrix algorithms is presented in [Golub and Van Loan, 1996]. A matrix algorithm is called

first level	if it operates on	complete vectors;
second level	if it operates on	rows or columns of matrices, i.e., matrix-vector multiplication;
third level	if it operates on	complete matrices, i.e., matrix-matrix multiplication.

Two natural choices for the level two representation are rows and columns. However, the actual multiindexed list representation does not maintain the data locality for the column format. Further, it will be quite inefficient to extract a particular row, as we need to seek through all the previous elements in all previous rows.

Nested lists. The naivest representation of a matrix in Haskell is a list of lists. Let

```
type NaiveMatrix a = [[a]]
```

Depending on the partitioning scheme (rows or columns in the outer list), we can gain access to the desired subset (either row or column) quite easy. So we can conclude

```
type NaiveMatrix a = List Int (List Int a)
```

If we were to apply some row-based algorithm to a matrix in a row-based representation, where the actions on the separate columns are independent, we could very easily compute in parallel with a `parMap`. The transformation from a row-based to a column-based representation and vice versa is the transposition, as implemented sequentially in the function `transpose` of type `[[a]] → [[a]]` from `Data.List` library. As we have learnt in Section 6.4.5, the distributed transposition requires an all-to-all communication.

Matrix as an array. This approach essentially doubles the multiindexed lists, but the access time of a single matrix element is constant. For the sake of Gauß elimination we implement matrices as arrays in Eden. This data type is not quite native for pure functional language, but we did not want to use nested lists for matrix representation. We write

```
type MatArr a b = Array (a, a) b
```

e.g., `MatArr Int (Fraction Integer)`. We will use this matrix representation in Section 7.8.1. Note that in any of the above representations the type `b` is not limited in any form. To be able to compute with such a representation, the type `b` should be an instance of `Num` and `Fractional`. Recall, this is our implementation of the symbolic computation: we model it with polymorphism.

6.5.2 Strassen Method

As before with polynomials, we focus on the most important operation of the matrix arithmetic: the matrix multiplication. An overview of our implementation efforts is in Figure 6.27. We will implement Strassen matrix multiplication using two algorithmic skeletons. The one would be the `divConFlat` skeleton from Section 6.2.2. The other one will be defined later, in Section 6.6. But consider the Strassen algorithm first. The computational complexity of the naive matrix-matrix multiplication[2] of two $n \times n$ matrices is $\mathcal{O}(n^3)$ operations in the base field. The complexity result is very easy to see, if we regard the naive matrix multiplication as three nested **for**-loops, each processing n elements for the product of two $n \times n$ matrices. A well-known approach to reduce this complexity is the Strassen matrix

[2]For the definition of the matrix-matrix product see Section A.7 on page 197 in the Appendix.

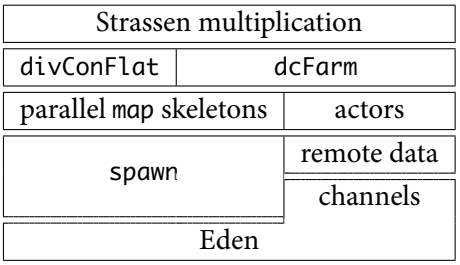


Figure 6.27: *Implementing parallel Strassen matrix multiplication.*

multiplication algorithm with the complexity of $\mathcal{O}(n^{\log_2 7})$ base operations for the two $n \times n$ matrices. There are some advances, see, e.g., [Brent, 1970, Winograd, 1971, Coppersmith and Winograd, 1982, Pan, 1984a,b, Laderman et al., 1992, Huang and Pan, 1998, Li et al., 2002, Cohn et al., 2005, Huang et al., 1990]. However, we consider here the approach of Strassen [1969]. Its communication complexity is $\mathcal{O}(n^2)$. Strassen's method is often used as a first stage of a hybrid parallel matrix multiplication implementation. See, e.g., [Luo and Drake, 1995, Grayson and Geijn, 1996, Huss-Lederman et al., 1996, Thottethodi et al., 1998, Nguyen et al., 2005]. A related Haskell implementation is [Rainey and Wise, 2004]. Other approaches include the method of Gentleman [1978] and the method, presented in [van de Geijn and Watts, 1997]. Gentleman's approach is also referred to as method of Cannon [1969], its implementation and improvements are considered, e.g., in [Hendrickson et al., 1994, Gupta and Sadayappan, 1996, Agarwal et al., 2010]. An Eden implementation of the Gentleman algorithm has been suggested in [Peña and Rubio, 2001].

The algorithm. For products of matrices of odd dimensions various techniques can be used. One of them is *padding*, the addition of a further zero row or column. Another is *dynamic peeling*, the removal of an 'extra' row or column. The 'peeled off' computation is performed separately. Details can be found, e.g., in [Huss-Lederman et al., 1996]. These techniques can be applied in each recursion step, thus we can constrain ourselves to matrices with dimensions being the powers of two.

We assume the input matrices have dimension of $2^l \times 2^l$. Now for $\mathbf{C} = \mathbf{AB}$ we define

$$\begin{aligned}
\mathbf{M_1} &= (\mathbf{A_{1,1}} + \mathbf{A_{2,2}})(\mathbf{B_{1,1}} + \mathbf{B_{2,2}}) \\
\mathbf{M_2} &= (\mathbf{A_{2,1}} + \mathbf{A_{2,2}})\mathbf{B_{1,1}} \\
\mathbf{M_3} &= \mathbf{A_{1,1}}(\mathbf{B_{1,2}} - \mathbf{B_{2,2}}) \\
\mathbf{M_4} &= \mathbf{A_{2,2}}(\mathbf{B_{2,1}} - \mathbf{B_{1,1}}) \\
\mathbf{M_5} &= (\mathbf{A_{1,1}} + \mathbf{A_{1,2}})\mathbf{B_{2,2}} \\
\mathbf{M_6} &= (\mathbf{A_{2,1}} - \mathbf{A_{1,1}})(\mathbf{B_{1,1}} + \mathbf{B_{1,2}}) \\
\mathbf{M_7} &= (\mathbf{A_{1,2}} - \mathbf{A_{2,2}})(\mathbf{B_{2,1}} + \mathbf{B_{2,2}}),
\end{aligned} \tag{6.2}$$

where the notion $\mathbf{X_{i,j}}$ means the j^{th} block in i^{th} row, while each matrix $\mathbf{X}$ is divided into four blocks in total. All the multiplications in (6.2) are done with recursive calls. The recursion ends at the single-element matrices. In practise one would quite quickly switch to the naive algorithm after passing a certain threshold. Then the result is reassembled:

$$\begin{aligned}
\mathbf{C_{1,1}} &= \mathbf{M_1} + \mathbf{M_4} - \mathbf{M_5} + \mathbf{M_7} \\
\mathbf{C_{1,2}} &= \mathbf{M_3} + \mathbf{M_5} \\
\mathbf{C_{2,1}} &= \mathbf{M_2} + \mathbf{M_4} \\
\mathbf{C_{2,2}} &= \mathbf{M_1} - \mathbf{M_2} + \mathbf{M_3} + \mathbf{M_6}.
\end{aligned} \tag{6.3}$$

We summarise the above method in Algorithm 14.

Quadtrees. Strassen matrix multiplication operates in a divide and conquer manner 'cutting' one operation of eight in a recursive block-wise representation. This is similar to the Karatsuba integer

Algorithm 14 Strassen matrix multiplication.

Require: input matrices **X** and **Y** of dimensions $m \times k$ and $k \times l$ appropriately.

1: **procedure** PADDING(**X**, **Y**)
2: Let $\hat{n}$ be the maximum of m, k, l, let $n \geq \hat{n}$ be the next power of two.
3: Fill the matrices **X** and **Y** with zeros, resulting in **A** and **B** of dimensions $n \times n$.
4: **return A**, **B**
5: **end procedure**
6: **procedure** STRASSEN MULTIPLICATION(**A**, **B**) *// both parameters have the dimension* $n \times n$
// Each multiplication is computed with a recursive call of STRASSEN MULTIPLICATION.
7: Partition **A** and **B** per (6.2), resulting in matrices $\mathbf{M_1}, \ldots, \mathbf{M_7}$, each of dimension $n/2 \times n/2$.
8: Combine $\mathbf{M_1}, \ldots, \mathbf{M_7}$ per (6.3), resulting in matrices $\mathbf{C_{1,1}}, \ldots, \mathbf{C_{2,2}}$. These are the four quadrants of the matrix **C**.
9: **return** combined matrix **C** of dimension $n \times n$
10: **end procedure**
11: **procedure** MAIN STRASSEN(arbitrary matrices **X**, **Y**)
12: Call PADDING(**X**, **Y**), resulting in padded matrices **A**, **B**.
13: Call STRASSEN MULTIPLICATION(**A**, **B**), resulting in the product matrix **C**.
14: **end procedure**

Ensure: **C** = **AB** is the product of the matrices **X** and **Y** with sides padded to the next power of two.

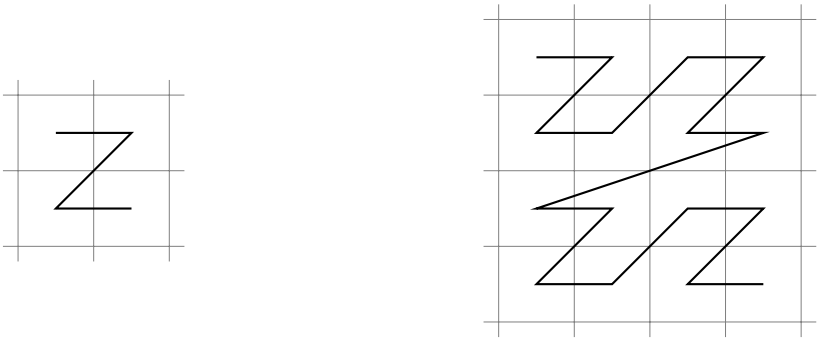


Figure 6.28: *The z-curve. Left: recursion depth one, right: depth two.*

multiplication. In this case each matrix is divided into four submatrices. A quaternary tree is very suitable for a clean representation of such block-wise operations. Such trees are called quadtrees.

The quadtrees are quaternary trees with no information in the nodes. They are easily defined in Haskell, as we will see below. A matrix is recursively divided into quadrants, like this:

$$\begin{array}{c|c} * & * \\ \hline * & * \end{array},$$

and stored in the quadtree structure. This implies the traversal of the matrix in the so-called z-order. The latter is deeply related to the z-curve, which is a space-filling curve. It is recursively defined as the curve, going from the top-left quadrant to the top-right, then to bottom-left and finally to bottom right quadrant. This shape is similar to the shape of the letter z, hence the name. We show a rendition of the first two iterations of the z-curve in Figure 6.28. See [Abdali and Wise, 1989, Rainey and Wise, 2004] for more details.

Algorithm 14 implies the partition of the input matrices into four blocks each and a recursive descent. The following code in Haskell outlines the data structure for the quadtrees.

```
data QTree a = QEmpty
             | QNode (QTree a) (QTree a) (QTree a) (QTree a)
             | QLeaf a a a a
```

The first constructor is reserved for the empty sub-trees. We decided against implementing the tree node as `QNode QTree [a]` to enforce the 'four-arity' in the type system. The `NFData` and `Trans` instances were added to the definition in the real code to ensure that the data can be reduced to normal form and that the data is transferable between the PEs.

Complexity. We need to prove the complexity statements from above. We base our presentation on [Wilf, 2002] and some results from [von zur Gathen and Gerhard, 2003]. A slight improvement in the number of additions is the Winograd algorithm [1971]. Among others, Coppersmith, Pan and Winograd have produced numerous improvements of the exponent w in the $\mathcal{O}(n^w)$ complexity of matrix multiplication, see, e.g., [Winograd, 1971, Coppersmith and Winograd, 1982, 1990, Pan, 1984a,b, Huang and Pan, 1998]. See also [Bürgisser et al., 1997]. Other publications on a similar topic include [Laderman et al., 1992, Li et al., 2002].

Consider the Strassen multiplication of two $2^l \times 2^l$ matrices over a UFD R. We will need $m(l)$ multiplications and $s(l)$ additions to accomplish it. We count subtractions as additions. As we need seven recursive calls to compute (6.2), set $m(l) = 7m(l-1)$ and $m(0) = 1$. It follows $m(l) = 7^l$ for $l \geq 0$. In other words,

$$m(n) = 7^{\log n/\log 2} = n^{\log 7/\log 2} = n^{\log_2 7} = n^{2.81\ldots}, \quad m(n) \in \mathcal{O}(n^{\log_2 7}).$$

As for additions, in each of the recursive calls of (6.2) we perform $s(l-1)$ additions in the base UFD R. Each of the eighteen calls to addition (or subtraction) of the matrices does square number of the matrix operations in R. In other words, in the first recursion level, which corresponds to the size of the matrix side 2^{l-1}, each of the calls does 2^{2l-2} additions in the base UFD. It follows with $n = 2^l$

$$s(n) = 7s(n/2) + 9/2n^2.$$

In terms of Proposition 6.3 we have $a = 7$, $k = 4$. Again, lb is the base-two logarithm. Hence

$$s(n) \leq s(1)n^{\mathrm{lb}\, a} + \frac{k}{a-k}(n^{\mathrm{lb}(a/k)} - 1) \cdot 9/2n^2 = n^{\mathrm{lb}\, 7} + 6n^{\mathrm{lb}(7/4)+2} - 6n^2 = 7n^{\mathrm{lb}\, 7} - 6n^2 \in \mathcal{O}(n^{\log_2 7}).$$

So, the $s(n)$ is in the same complexity class as $m(n)$. For n being not the power of two, the padding maximally doubles the dimension. We have just shown the next theorem.

Theorem 6.7. *The complexity of Algorithm 14 for the product of two $n \times n$ matrices over a UFD R is $\mathcal{O}(n^{\log_2 7})$ arithmetic operations in R.*

We can visualise the complexity of a single step of the Strassen multiplication as a volume of a cube with side n, which has one eighth of it removed. The removed fraction corresponds to a cube with sides $n/2$, removed from the original cube. The complexity of the naive matrix multiplication algorithm takes the whole volume of the cube, without cutting anything out. We show a corresponding structure for two steps of Algorithm 14 in Figure 6.29. It is the threedimensional analogue of Figure 6.8 on page 102.

6.5.3 Parallel Performance

The instantiation of a divide and conquer skeleton with Strassen multiplication is straightforward. Figure 6.30 shows the code. We introduce another constructor for `QTree`, `QFull` contains a matrix in a non-quadtree representation. Thus we can switch to the naive $\mathcal{O}(n^3)$ matrix multiplication as soon as we need to. The functions `divide` and `combine` closely follow Algorithm 14 and equations (6.2) and (6.3). The helper functions `isTrivial` and `solve` are straightforward. The parallel skeleton-based implementation is very simple:

```
strassen :: Num a ⇒ Int → QTree a → QTree a → QTree a
strassen d x y = divConFlat (farm' noPe) d isTrivial solve divide combine (x,y)
```

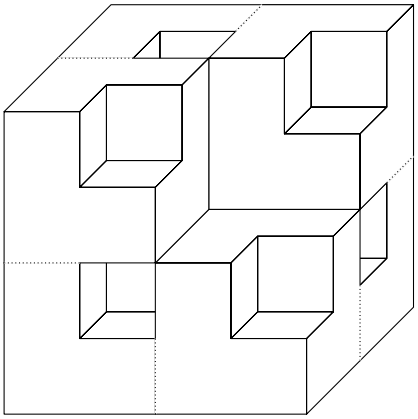


Figure 6.29: *The complexity of the first two steps of Strassen multiplication corresponds to the volume of this fractal.*

We merely replaced the skeleton.

All our tests are conducted on `sakania`, on random integer matrices with entries between ±1000. We use a hybrid implementation: few steps of the Strassen method, followed by the naive matrix multiplication. The test input consists of two $2^9 \times 2^9$ randomly generated integer matrices, which means 2^{19} entries in total. Each multiplication is performed five times, we take the mean time. We do not measure the time needed to generate the test input. We utilise the existing divide and conquer skeleton `divConFlat`. It produces the absolute speedup 3.54 on 8 PE with GHC 6.12 implementation. For input size $2^{10} \times 2^{10}$ our program produces an absolute speedup of 3.85. These results are acceptable. The parallel version was written very fast: we needed to change the skeleton invocation in the `strassen` function—a single line of code.

Figure 6.31 shows the trace diagram of the parallel Strassen multiplication on `sakania`. The parallel depth was one. We used no memory tuning runtime options. The message buffer was set to 80 Mb. A bit more than two seconds are spent in beginning of the program for the generation of the input matrices. We computed the product of two $2^9 \times 2^9$ matrices, generated as discussed above. We see the worker processes start simultaneously at 2.37 seconds, but begin working at different time, from 2.87 to 5.2, seconds. In other words, the worker PEs start computing few seconds apart. This accounts for the relatively low speedups. The time from 2.37 seconds to the begin of the computation is used for the communication of the input matrices to the workers. At 5.2 seconds all workers are green, i.e., working. At 7.07 seconds the first worker is done. The last worker finishes at 9.4 seconds. Shortly before 10 seconds the program terminates.

We plot the speedup curves for the Strassen-based multiplication with `divConFlat` skeleton in Figure 6.32. We show the performance for the depths 1–4 of parallel divide and conquer tree. Depth 1 shows best results. The depth 2 results in a slightly worse, but more smooth curve because of less issues with task distribution. We will discuss these in detail below. The depth 3 and 4 versions are even worse, so we do not focus on these. The additional, 'adaptive' depth is the program version, which creates as many tasks, as needed to saturate all PEs, namely, for arity r and p PEs the depth is $\lceil \log_r p \rceil$. In the setting of `sakania` this means 7 tasks for $1 < p \leq 7$ and 49 tasks for $p = 8$. At said points we see a close correspondence with depths 1 and 2 appropriately.

We see some strange symptoms—like a degrading curve—in the speedup plot for depth 1 in Figure 6.32. To understand this behaviour we will consider the theoretical task distribution of the relevant instance of the `divConFlat`. It uses one PE as a master, tasks will be co-located, if needed. Further, we aim to create only one level of parallel divide and conquer expansion. This means seven worker tasks are placed. We display our thought experiment in Table 6.7. We assume all tasks are of the same size, i.e., they take equal time to process. We have already explained all rows of such a table on page 121, with a notable exception of the second row. The entry 'co-located with master' of Table 6.7 shows the number of tasks, which land at the same PE as the master. The notion $1^\star$ means that in the default place-

```
data QTree a = QEmpty
             | QFull [[a]]
             | QNode (QTree a) (QTree a) (QTree a) (QTree a)
             | QLeaf a a a a
-- corresponding arithmetic operations are defined in an omitted Num instance

divide :: Num a ⇒ (QTree a, QTree a) → [(QTree a, QTree a)]
divide ((QNode a11 a12 a21 a22), (QNode b11 b12 b21 b22)) =
    [((a11 + a22), (b11 + b22)),
     ((a21 + a22), b11),
     (a11, (b12 - b22)),
     (a22, (b21 - b11)),
     ((a11 + a12), b22),
     ((a21 - a11), (b11 + b12)),
     ((a12 - a22), (b21 + b22))]

combine :: Num a ⇒ [QTree a] → QTree a
combine ms = QNode ((ms!!0)  +  (ms!!3)  -  (ms!!4)  +  (ms!!6))
                   ((ms!!2)  +  (ms!!4))
                   ((ms!!1)  +  (ms!!3))
                   ((ms!!0)  -  (ms!!1)  +  (ms!!2)  +  (ms!!5))

isTrivial :: (QTree a, QTree a) → Bool
isTrivial (QNode _ _ _ _, QNode _ _ _ _) = False
isTrivial _ = True

solve :: Num a ⇒ (QTree a, QTree a) → QTree a
solve = uncurry *

strassenSeq :: Num a ⇒ QTree a → QTree a → QTree a
strassenSeq x y = divConSeq isTrivial solve divide combine (x, y)
```

Figure 6.30: *Strassen multiplication in Haskell.*

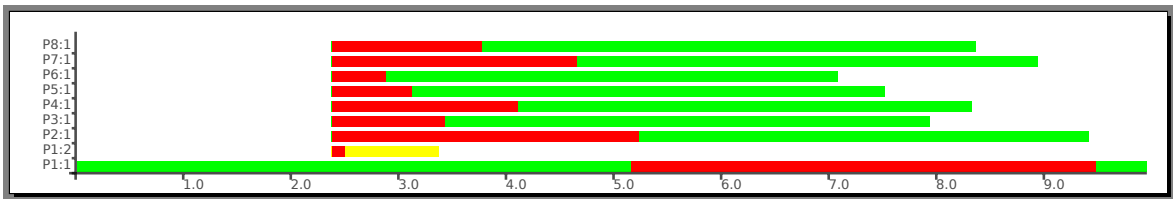


Figure 6.31: *Trace visualisation of Strassen multiplication with* `divConFlat` *skeleton, depth one.*

ment the task is co-located, but this could be avoided with a cleverer placement scheme. We see that 3 PE is not a very fortunate configuration and that 5–6 PEs are unlucky. The latter part of this theoretic prediction corresponds with the practical result of Figure 6.32, top left, we see bad performance at 5–6 PEs. We also see a minor decline of the speedup at 3 PE, but cannot be sure the reason is a task distribution problem. The recovering speedup at 7 PE is interesting. If we would use no tasks co-located with the master process, we would see a large slack-off at this number of PE. But the co-location with master makes it possible to finish the computation in a single round. We denote this with 0* value in the table. Another confirmation is the behaviour of the depth 2 speedup curve in Figure 6.32, top left. We see that the larger number of tasks reduces the bad issues with task distribution, but the overall performance of the depth 2 version does not reach the better values of the depth 1 version. Now, if we

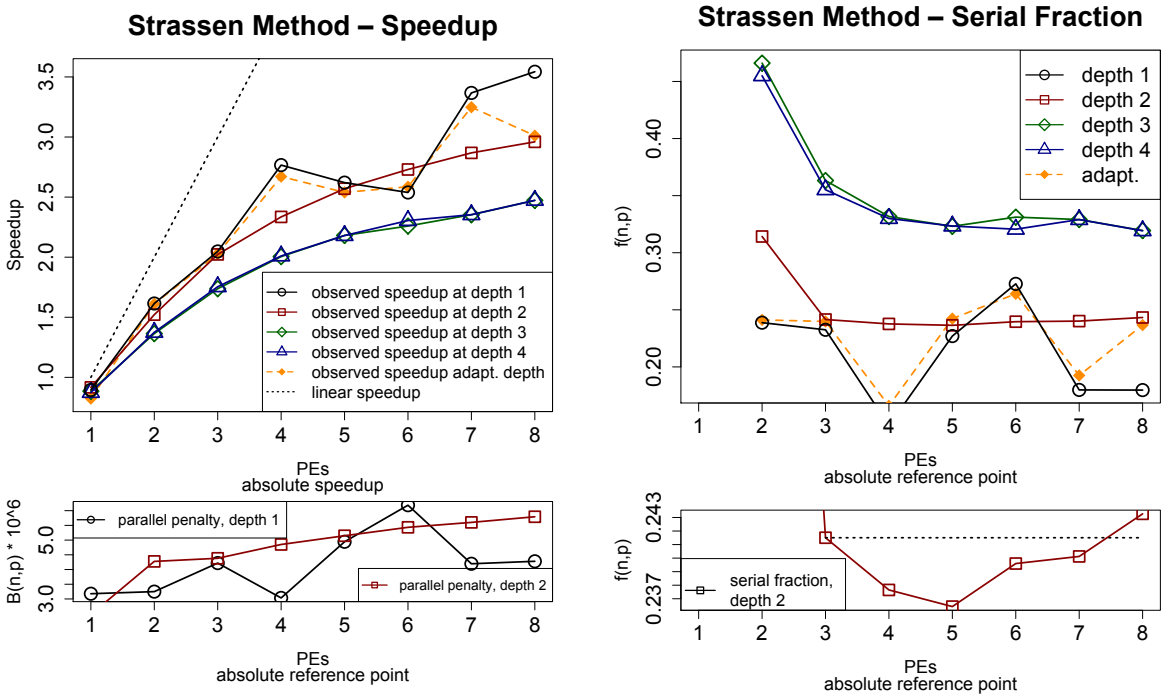


Figure 6.32: Speedup and quality measures for Strassen matrix multiplication. Input size $2^9 \times 2^9$. Top left: speedup, bottom left: parallel penalty, top right: serial fraction, an overview, bottom right: serial fraction for depth 2.

full PEs	1	2	3	4	5	6	7	8
co-located with master	7	3	2	1	1*	1*	1*	0
full rounds	8	4	2	2	1	1	1	1
remaining part	0	0	2	0	3	2	1	0
total rounds	8	4	3	2	2	2	1*	1
idle in last round	0	0	1	0	2	4	0*	0
slack-off, %	0	0	33.3	0	40	66.6	0*	0

***Table 6.7:** Theoretical task distribution in depth one parallel Strassen multiplication. A star (*) denotes a co-located task on the master PE.*

look at the values for 7 PE, we see a rapid increase of the speedup in depth 1 version, but nothing like that in depth 2 version. Summarising, we were able to find an explanation to the strange behaviour of the depth 1 speedup curve at 5–7 PEs.

Consider now the practical results on the parallelisation quality of Strassen multiplication. We look at the two usual quality measures—parallel penalty and serial fraction. The parallel penalty $B(n,p)$ w. r. t. p in Figure 6.32, bottom left, black line for depth 1, has a small peak at 3 PE, declines again at 4 PE, then rises quite sharply at 5–6 PEs, then declines and is nearly constant at 7–8 PEs. In other words it designates the 2, 4, 7 and 8 PE configurations as best ones in the given setting. This agrees with the theoretical results of Table 6.7. This interpretation of the parallel penalty matches the observations from the speedup plot. As for the parallel penalty plot for depth 2 (red line in Figure 6.32, bottom left), we see an almost steady increase, which fits the speedup curve in Figure 6.32, top left. The large absolute values of $B(n,p)$—around 10^6—are due to the time unit, which was microseconds in this particular case.

The serial fraction $f(n,p)$ for depths 1 to 4 and for the adaptive depth method is in Figure 6.32, top right. The serial fraction values for depths 3 and 4 are always much larger than their counterparts for depths 1 and 2. Further, the plots for higher depths almost coincide, similarly to the appropriate

speedup curves. Hence, they are of little interest for us. The 'adaptive' version matches well the depth 1 version up to the point at 8 PE, where it matches the depth 2 version. This is exactly the intended behaviour of the adaptive version. Let us focus on depths 1 and 2 below. In a contrast to the parallel penalty, the serial fraction plot for depth 1 skeleton decreases at 2–4 PEs. It slows down the decrease at 3 PE, but we see no peak as in the parallel penalty. An increase at 4–6 PEs matches the degrading speedup at the same number of PEs. The 7 PE value brings a decline, the 8 PE value is almost identical to 7 PE. This is similar to the parallel penalty and also matches our task distribution theory from Table 6.7.

Interestingly, the depth 2 parallel penalty is almost constant at 3–8 PEs. To be more exact: the difference between its values in this range is too small to be seen easily in Figure 6.32, top right. For this reason, we consider the depth 2 parallel penalty in Figure 6.32, bottom right. We can conjecture that 5 PE version is quite perfect, as 49 tasks and one master can be distributed perfectly well over 5 PE. Still, the serial fraction has almost the same value at 8 PE, as in 3 PE (dashed horizontal line in Figure 6.32, bottom right). We could not see these issues in the parallel penalty in Figure 6.32, bottom left. Overall, the serial fraction for depth 2 suggests a good parallelisation, but the speedup (Figure 6.32, top left) suffers from communication overhead.

We do not perform an estimation of the run time of our implementation of Strassen matrix multiplication w.r.t. the input size n, as we use only the powers of two for the matrix sides. We have too few measurable data points, which are too far aside. As for the estimation w.r.t. p, we performed it for depth 2 version. Depth 1 configuration depends too much on task balancing issues, which we addressed with other methods. We predict the execution time for $p = 8$ from times at $p < 8$. With loess we estimated $B(n, p)$ w.r.t. p with the relative error of −1.50%. The spline method was exact up to −4.98%. Combining the estimation of $B(n, p)$ with the sequential time per (4.1), we obtained the parallel run time with a relative error −0.945%, using loess-based estimation. We see a good performance of our prediction method.

6.6 Divide and Conquer with Actors

The *actors* [Hewitt et al., 1973, Agha et al., 1997] form a quite high-level model of concurrent computing. As Agha [1990] states, actors are "self-contained, interactive, independent components of a computing system". Modern implementations of actors include Erlang [Armstrong, 2007], Scala actors [Haller and Odersky, 2009], Haskell actors [Sulzmann et al., 2008, Sulzmann, 2008]. The last-mentioned library implements multi-headed receive patterns, an extension of the actors formalism, but is quite low-level. Also related is [Epstein et al., 2011]. We will present an actor-based implementation of a divide and conquer skeleton next. After the implementation is introduced, we will explain its relation to the actors in Section 6.6.2.

In the context of the actors emphasis is often set on the *open* distributed systems, where one part of the system can be modified without changing everything else. We cannot model the dynamic changes of the payload code at the runtime of the program with Eden. But we can still use the actor model. In [Haller and Odersky, 2007, 2009] the *receive* clause of an actor is modelled with a pattern matching on the events. The main problem is how to enable an 'Eden actor' to receive all the time and not only for the initial moment, as it would be a case with Eden processes, having obtained their input completely. We will see below, that we can solve this problem using streams [Wray and Fairbairn, 1989].

The structure of below code is: we will use `farm` as a 'task manager', on top of it the actor-like message passing will be implemented, which, again, will be used to implement divide and conquer. See Figure 6.33 for a visual presentation. Our approach is a generalisation of `divConFlat`, see also [Peña and Rubio, 2001, Loogen et al., 2003]. It is similar to [Poldner and Kuchen, 2008a,b] in the sense, that we have a *task pool*, wherein different types of tasks are injected.

The basic idea behind the `dcFarm` skeleton is a 'lazy' `farm` with external task creation. For more details on the classification of task creating skeletons see Chapter 5. Our approach is similar to [Priebe, 2006, Brown and Hammond, 2010]. The lazy input list of the parallel `farm` is concatenated with new

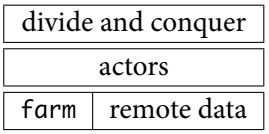


Figure 6.33: *Implementing divide and conquer with actors.*

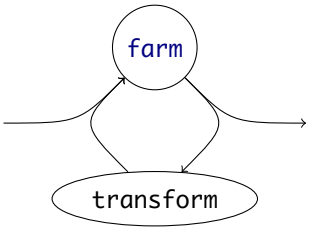


Figure 6.34: *The idea behind a task creating* `farm`*.*

tasks, which are created from the lazy list of the results by the `transform` function. We display the basic scheme in Figure 6.34.

6.6.1 Implementation

In order to submit new divide and conquer tasks for the parallel processing, we need to unify the types. Instead of the usual `DC' a b` type, we need some common type `c`, which is both the type of the problem and the type of the result. We solve this problem by wrapping the divide and conquer structure into a single datatype, which contains both the divide and combine tasks. Further, we need to distinguish the to-be-processed and already processed tasks. This adds up to four constructors. The function, sitting in a worker process, obtains the stream of tasks to process and returns already processed results. This function is the transition function of an actor. The actor itself is a combination of such a function, provided with the facilities for the remote execution, and stream communication.

The suggested scheme has a drawback. Let consider the divide phase. A divide task of depth i has been processed by a worker and was divided into r divide tasks of depth $i+1$. Each of these tasks is processed by a separate worker. This means that the data in each of the tasks travels to the master (forming the result of the depth i task) and back to the workers from the master, forming new, depth $i+1$ tasks. In other words, larger communication overhead is present. We solve this problem with the remote data, *viz.* Chapter 3. So, although the *handles* are communicated from the worker to the master and back to the workers in the suggested scenario, the actual data is communicated directly between workers. Earlier experiments have shown, that the evaluation order of the initial tasks might be not optimal. The reason is: the `release` function of remote data spawns a thread to evaluate the data. For large number of tasks, created directly in the master, it happens often that the first tasks, for which the workers are waiting, are not evaluated early enough. The programmer has no influence on the thread scheduling. But in fact at this, initial stage the remote data is meaningless, as the data needs to be communicated from the master to appropriate workers anyway. We designate a special constructor for such, initial tasks. It carries the data directly, without futures. Hence, a data type with five constructors emerges, see Figure 6.35, top. The `farm`, lazily consuming the list of its transformed results, is easily written, see function `dcFarmBodySimple` in Figure 6.35, middle. We need the function `dcFarmBody` in Figure 6.35, bottom, which is a bit more complex than its above analogue. In the 'simple' case we let the results through the `transform` function and inject them as tasks into the `farm` skeleton. The actual case (`dcFarmBody`) uses the additional parameter `postfork` to find the `ToCombine` tasks of too high level (i. e., of low depth in the divide and conquer tree) and processes them locally. Apart from this, `dcFarmBody` forms the same loop of the data flow, as `dcFarmBodySimple`. This was the easy part, more work is in the `transform` function and in the divide and conquer wrapper around `dcFarmBody`. We discuss them next. Essential is the definition of the single actor transition function from the four divide and conquer parameter functions. It happens in the same wrapper function `dcFarm`.

```
data DCTask a b = InitialToDivide Depth a
                | ToDivide Depth (RD a)
                | Divided Depth [RD a]
                | Combined Depth (RD b)
                | ToCombine Depth [RD b]
type Depth = Int
type Arity = Int

dcFarmBodySimple :: (Trans a, Trans b)
                 ⇒ Arity → Depth
                 → (Arity → [DCTask a b] → [DCTask a b]) -- ^ task pool transf.
                 → (DCTask a b → DCTask a b)             -- ^ working function
                 → [DCTask a b] → [DCTask a b]           -- ^ input to result
dcFarmBodySimple k d transform f initTasks = farm' noPe f allTasks
    where newTasks = transform k res
          allTasks = initTasks ++ newTasks

dcFarmBody :: (Trans a, Trans b)
       ⇒ Arity
       → Depth -- ^ parallel divide depth
       → Depth -- ^ postfork combine depth
       → (Arity → [DCTask a b] → [DCTask a b]) -- ^ task pool transform
       → (DCTask a b → DCTask a b)             -- ^ working function
       → [DCTask a b] → [DCTask a b]           -- ^ input to result
dcFarmBody k d postfork ttf f initTasks = localRes
    where -- parallel part
          remoteRes = farm' noPe f (initTasks ++ newRemoteTasks)
          newRemoteTasks = ttf k putInPool
          -- select parallel and local, sequential parts
          (putInPool, stayLocal) = span (not ∘ p_stayLocal) remoteRes
               where p_stayLocal (Combined d' _) | d' ≤ postfork = True
                     p_stayLocal _ = False
          -- local work
          localRes = stayLocal ++ map f newLocalTasks -- stayLocal already computed
          newLocalTasks = ttf k localRes
```

Figure 6.35: *The `DCTask` declaration and the `farm` with external task creation.*

The task pool transform function. We need to know how the results are *transformed*, before they are injected back into `dcFarmBody`. One of the issues to take care of is the termination of the skeleton. So, let us consider the function `transform` in Figure 6.36 in a more detail. Firstly, if we see the final result, i.e., `Combined 0 x`, we terminate the list. The zero level means the top-most position in the divide and conquer tree, if this position is reached in the combine step, then the computation is finished. The second equation of `transform` makes a list of `ToDivide` tasks from a single `Divided` result. This represents another step of the divide step. The return value consists of the said task and a result of the recursive call of `transform` on the remaining list. The depth parameter is updated in the working function, which we present below. The third equation of `transform` handles the termination of the input stream. The fourth equation uses the helper function `catchNewToCombineTask`. This function looks whether the arity-many next elements of its input list are `Combined` results of the same depth. If it is the case, these elements are combined into a single `ToCombine` task of a higher level. In this case, the helper function returns a pair of the new combine task and the remaining list elements. In the opposite case, `Nothing` is returned. The function `transform` checks the result of the helper function and throws an error in the negative case. If the result is sane, the new task is returned and the function

```
catchNewToCombineTask :: Arity → [DCTask a b] → Maybe (DCTask a b, [DCTask a b])
-- implementation omitted

transform :: Arity → [DCTask a b] → [DCTask a b] -- ^ task pool in and out
transform k ((Combined 0 x):r) = [] -- done!
transform k ((Divided d' xs):r) = let ys = zipWith ToDivide (repeat d') xs
                                  in ys ++ transform k r -- flatten the list!
transform k [] = [] -- done! We have reached the postfork depth
transform k xs = case catchNewToCombineTask k xs of
                   Just (newTCTask, rest) → newTCTask : transform k rest
                   Nothing → error "transform: DC structure is broken!"
```

Figure 6.36: *The `transform` function of the `dcFarm` implementation.*

`transform` is called recursively on the remaining list elements.

The divide and conquer wrapper and the definition of the working function. Let us consider the `dcFarm` wrapper function, which gives our skeleton the `DC' a b` type. We also need to consider the definition of the working function, which is defined in the same place, namely in Figure 6.37. The skeleton takes three additional parameters: the arity k of the divide and conquer tree (it is 7 in case of Strassen multiplication), the depth d up to which new parallel tasks are issued, and the depth m up to which the divide and conquer tree is processed locally in the master PE. It should hold $d > m$. The case $d = m$ corresponds to the `divConFlat` implementation in the divide phase. Contrarily to the flat expansion, all the combine operations from the depth d onward are performed in a distributed manner. To put it short, the skeleton below may perform distributed divide steps, then it solves the distributed tasks locally in the workers, and may perform distributed combine steps of a given divide and conquer algorithm.

At the first glance the most distributed divide phase, i.e., the $m = 0$ setting, results in the best parallelism. But this is not so. If we launch the skeleton with one to-be-parallel divide task of depth zero, it is processed in the first worker, yielding k divide tasks of depth 1. If the number of PE is larger than $d + 1$ then the PEs are not even saturated after two first divide steps. We need to add the communication penalty for the larger tasks. This problem is illustrated in Figure 6.38 for $k = 3$, $m = 0$ and 9 PE. Thus we have introduced the parameter m to 'prefork' the fitting amount of tasks. This parameter can easily be precomputed. Let p be the number of PE and k the arity of the divide and conquer tree. Then $m = \lceil \log_k p \rceil$ will surely saturate all PEs.

Hence, we do not state the initial task as `ToDivide 0 (release x)`, but rather create k^m initial tasks locally. This happens in the definition of `initTasks`. These tasks are then processed by the `dcFarmBody` function, from their results the `transformed`, secondary tasks are created. The helper function `tryNtimes` is used to compute `initTasks`. It becomes a parameter function `f`, a predicate `p`, a number of applications, and the initial input. As long as `p` holds, and if the number of applications is smaller than requested, the function `f` is applied again to the result of previous applications. The final result is a tuple of the last function application and the total number of applications.

The worker function `wf` in Figure 6.37, bottom, is the final part of the transformation of a divide and conquer problem into a map problem. It pattern matches on three 'input' constructors of `DCTask`—`InitialToDivide`, `ToDivide` and `ToCombine`. The first two equations of `wf` handle the divide tasks. Eventually, the actual data needs to be communicated with `fetch`. The remaining part of these two cases is in the `helper` function. If the predicate `isTrivial` is satisfied, then the task is solved locally. The same happens in the recursion depth parameter `d'` in the input task is larger or equal to the maximal parallel recursion depth `d`. The only difference in this case is that the further processing is done not with the function `solve`, but with the correspondingly instantiated sequential divide and conquer skeleton. The third case in the guard expression of the `helper` function is a further parallel divide step. Here the `divide` function is applied to `y`, the resulting list is released to remote data, and

```
-- helper function. Applies f n times, if p holds
-- return value is a pair (final result, number of applications)
tryNtimes :: (a → a) → (a → Bool) → a → Int → (a, Int)
tryNtimes f p x n = ... -- implementation omitted

-- helper function. Evaluates the input to rnf and applies a function afterwards
rnfApply :: NFData a ⇒ (a → b) → a → b
rnfApply f x = rnf x `seq` f x

dcFarm :: (Trans a, Trans b)
       ⇒ Arity    -- ^ Arity of the DC tree
       → Depth    -- ^ parallelLevels: depth to unfold DC tree in parallel
       → Depth    -- ^ prefork: depth up to which only the master unfolds
       → Depth    -- ^ postfork: number of combine levels to do sequentially
       → DC' a b
dcFarm k d prefork postfork isTrivial divide solve combine x
    = fetch $ fromCombined $ last $
        dcFarmBody k d postfork transform (wf d) initTasks
      where -- compute initial tasks, as prefork tells
            initTasks = map (InitialToDivide splitDepth) initRaw
            (initRaw, splitDepth)
              = tryNtimes (concatMap divide) (all $ not ∘ isTrivial) [x] prefork
            -- worker function, this is the actual actor's transition function
            wf d (InitialToDivide d' y) = helper d' y
            wf d (ToDivide d' x) = helper d' (fetch x)
            wf d (ToCombine d' ys) = Combined (d'-1)
                                     $ (release ∘ combine ∘ fetchAll) ys
            -- both divide branches do almost the same:
            helper d' y
                | isTrivial y = rnfApply ((Combined d') ∘ release) (solve y)
                | d' ≥ d = rnfApply ((Combined d') ∘ release)
                                    (divConSeq isTrivial divide solve combine y)
                | otherwise = rnfApply (Divided (d'+1) ∘ releaseAll) (divide y)
```

Figure 6.37: *The wrapper function dcFarm and the definition of the working function.*

the depth parameter is incremented. In all three cases before wrapping the corresponding results into the appropriate constructor and possibly applying `release`, the helper function `rnfApply` is used to force the evaluation of the content. The third, 'combine' equation of `wf` is simpler. The input list is fetched from remote data, combined to a single value, which in its turn is released again to remote data. Putting the `Combined` constructor around this result and decrementing the depth counter finalises the second equation of `wf`.

6.6.2 Actors

With the presented framework we have modelled actors in a pure functional style. The crucial observation is that issuing a new divide or a new conquer task to the working function on a remote process is in fact sending a message to divide or to combine the attached data to a (remote) actor. Admittedly, this is a very primitive actor model, but note the high-level implementation, not bothering with channels and low-level message passing. The channels for communication with an actor are the implicit streams, connecting the worker processes with the master. When a new process is created, an instance of the working function is placed on it. This is a new actor. Process termination, correspondingly, kills

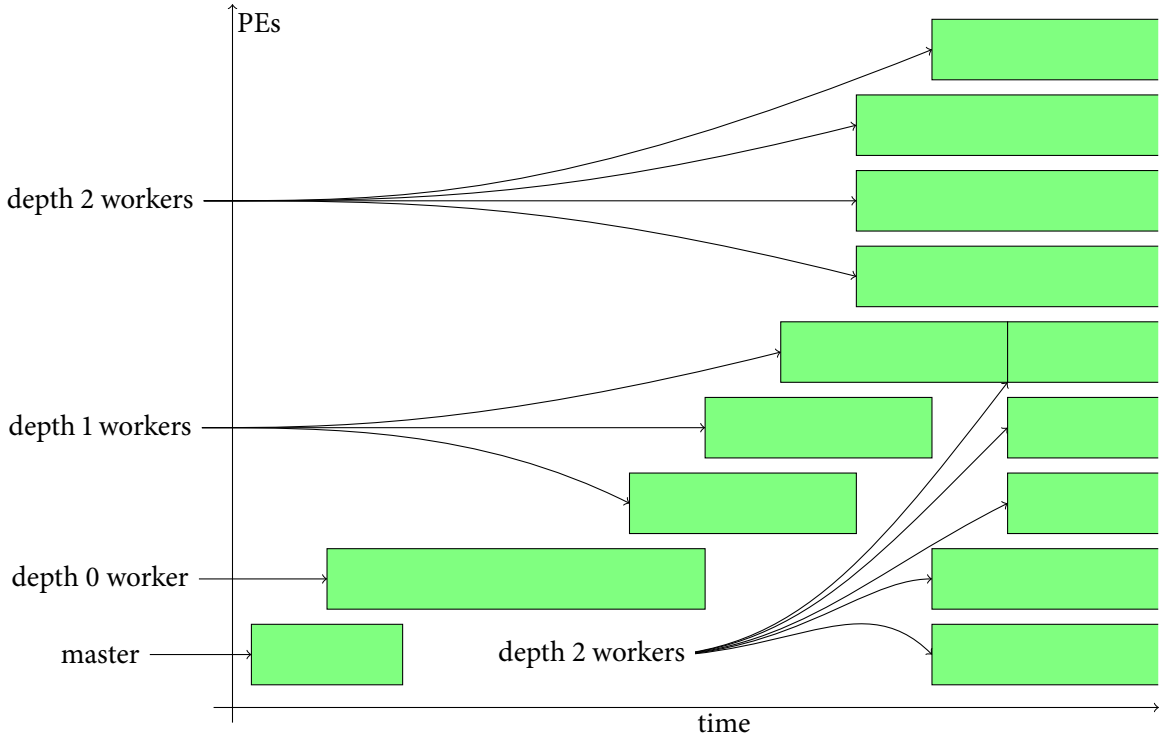


Figure 6.38: *The completely distributed divide with arity three and depth two at 9 PE. If a PE has no task at the moment, it is blocked.*

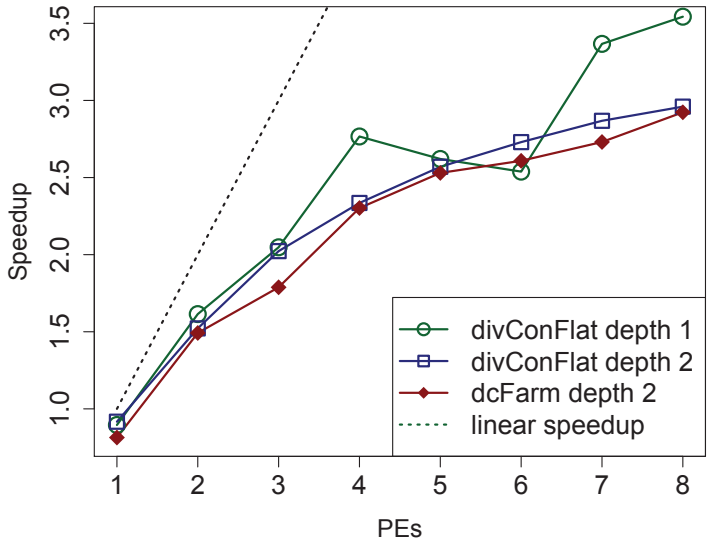


Figure 6.39: *Comparing the speedups for `divConFlat` and `dcFarm` on `sakania`.*

an actor. The non-initial messages, i. e., the output values of the worker function, are sent to the master per the `farm` scheme. However, after the post-processing with `transform`, the messages are sent to the further actors. Although the communication formally goes over the master, the actual data flows over a shortcut using the remote data: we model direct communication of actors.

Contrarily to the traditional actor scheme, in this implementation the master decides, which message is sent to which actor. The tasks are distributed according to the round-robin laws of the `farm` skeleton. In our implementation an actor never discards a message.

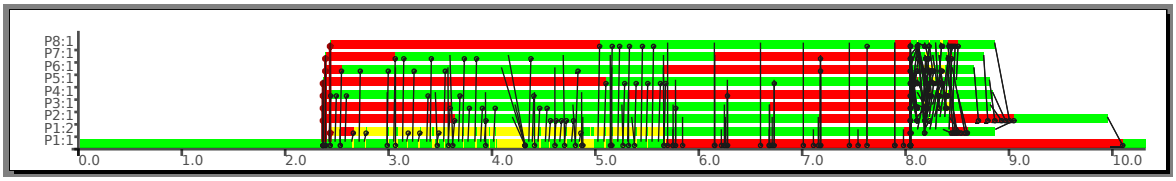


Figure 6.40: *Trace visualisation of Strassen multiplication with `dcFarm` skeleton. Black arrows symbolise messages. Parallel depth 2, prefork depth 2, postfork depth 0.*

Performance. We instantiate the `dcFarm` skeleton with Strassen multiplication. We use the same input as for the `divConFlat` version: $2^9 \times 2^9$ integer matrices. The executable was produced with GHC 6.12. We used no manual memory management. The best `dcFarm` approach results in relative speedup 3.59 on 8 PE on `sakania`. The absolute speedup is even more modest: 2.92. We used the depth 2 version. The depth 1 version has even worse speedup of 2.77 in the same setting. The `divConFlat` version resulted in the absolute speedup 3.54 for depth 1 and 2.96 for depth 2 on the same hardware for the same input size. These quantitative differences between the two approaches are highlighted in Figure 6.39.

We see that with the exception of 6 PE, where `divConFlat` with depth 1 has task balancing problems, the `dcFarm` with depth 2 produces worse speedup than both `divConFlat` with depth 1 and 2. We need to admit, that both skeletons with divide and conquer depth 2 have similar performance.

Figure 6.40 shows the trace diagram of the depth 2 parallel Strassen multiplication on `sakania`. A bit more than two seconds are spent in beginning of the program for the generation of the input matrices. The `combine` phase is clearly seen at 8–10 seconds. It shows nicely the actor approach of the `dcFarm` skeleton. Note the heavy worker-to-worker communication in the combine phase.

We do not present the plots for the parallel penalty and the serial fraction for the `dcFarm` implementation. Our goal was to compare the performance with the 'standard' `divConFlat`.

6.6.3 Bottom Line

We have presented a straightforward implementation of Strassen fast matrix multiplication. We instantiated the `divConFlat` skeleton with it and obtained acceptable speedups. We also experimented with a new `dcFarm` skeleton, which was implemented in an actor-like manner. We found it slightly worse than the traditional skeletal approach.

6.7 Conclusions

An overview of our contributions is in Figure 6.41. Boxes show entities, straight lines show dependencies. The dotted lines show possible dependencies of marginal or prototyped implementations.

We have implemented two parallel multiplication routines for univariate polynomials (which are also usable for large integers) and two for matrices. We presented Karatsuba multiplication, FFT-based multiplication (including some approaches to the parallel FFT for that sake) and Strassen matrix multiplication. These three methods belong to the algorithmic base of a modern parallel computer algebra system. They also share a structural principle: these three algorithms are divide and conquer algorithms. With our approach to algorithmic skeletons we were able to extract common parallelisation principles of said three algorithms. This way we were able to shorten the development time of the implementation of the core algorithms of computer algebra. We were able to switch fast between various parallelisation approaches, see various divide and conquer implementations of the FFT.

We experimented with the divide and conquer expansion schemes. The transition from a sequential program to a parallel one was extremely easy, given an existing fitting skeleton, as the examples of Karatsuba multiplication and Strassen multiplication show.

The Strassen matrix multiplication was also implemented using two skeletons. The one is the `divConFlat` skeleton. The other is the 'actor-like' approach of `dcFarm`. In the latter case, the divide and

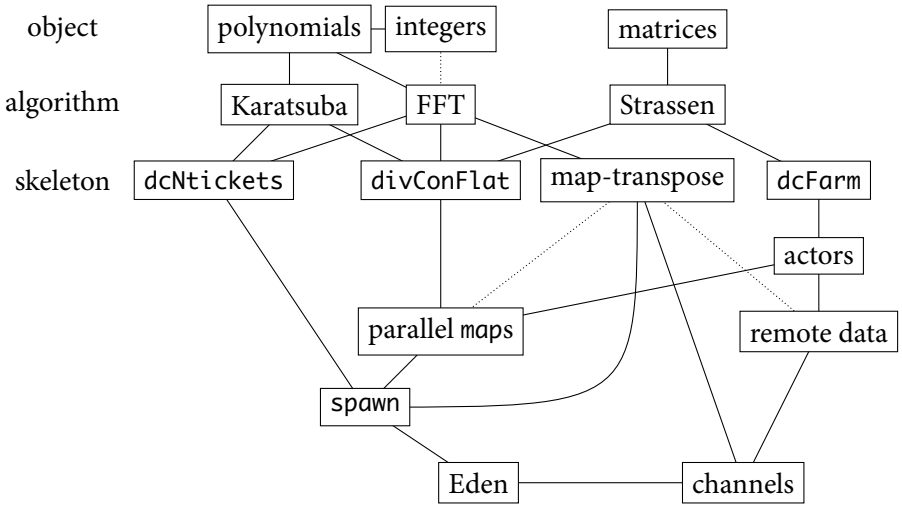


Figure 6.41: *An overview of this chapter.*

conquer structure was modelled with a `farm` skeleton and streams. This case study emphasises again the flexibility of the skeleton-based approach in a parallel functional setting.

Chapter 7

DATA PARALLEL ARITHMETIC

> The purpose of computing is insight, not numbers.
>
> Richard Hamming, *Introduction to Applied Numerical Analysis*

In many cases the wish to perform some computation *exactly* leads to an overhead not present in the computation with floating point approximations. While the numerical analysis expects floating point operations—FLOPs—to have the same cost and focuses on the error analysis [Golub and Van Loan, 1996], in the domain of the exact operations this is not so. Most of the exact operations depend on the cost of (integer) multiplication $M(n)$, which itself depends on the *length* n of the integers in question [von zur Gathen and Gerhard, 2003]. But these lengths are not constant, while the computation progresses. The intermediate expressions are often even larger than the final result is. This is known as the *intermediate expression swell* [Tobey, 1966, von zur Gathen and Gerhard, 2003].

To give an example of a dependency of exact operations on $M(n)$, consider an example. Imagine a computation with a matrix over $\mathbb{Q}(\sqrt{3})$, the cost for the computation in $\mathbb{Q}[x]/\langle x^2 - 3\rangle$ bears the cost for computing in $\mathbb{Q}$, while the latter depends on the cost of computing in $\mathbb{Z} \times \mathbb{Z}$. Hence, we have tracked a dependency of a complicated algebraic computation to the computation cost of single integer operations. Of these the cost for integer multiplication $M(n)$ is the most interesting one. We considered fast multiplication algorithms (i.e., algorithms with improved $M(n)$) in Chapter 6. As we already know, integer multiplication cost dominates the integer addition cost. The integer division cost depends on $M(n)$. We discuss further issues, emerging from this fact, in Section 7.1.

A common approach to reduce the computational complexity of symbolic computations, i.e., to reduce the intermediate expression swell, is a residue arithmetic. We consider residues of integers in this chapter in Section 7.2. A residue system w.r.t. some prime m is the field $\mathbb{Z}/\mathfrak{m}$ where $\mathfrak{m}$ is an ideal, generated by m. The addition and multiplication are the usual operations in $\mathbb{Z}$, combined with the residue computation modulo m. The latter is an integer division by m with the actual result being the residue, the remainder of the division. The focus of this chapter lies on systems i) using multiple residues at the same time and ii) capable of representing fractions. The latter problem is tackled for a single residue case in Section 7.3. It is possible to represent certain subsets of *rational* numbers as integers in a residue class—and to recover the rational numbers from integers. This property holds for a bound, connecting the input rational numbers and output residue class. It is stated in Theorem 7.7 on page 147.

To scale the residue class' size correctly, an upper bound on the final result is required. This bound needs to be known beforehand, *a priori*. Fortunately, closed formulae for such bounds are known for many algorithms. If we have such an *a priori* upper bound on the result of our computation, then we can perform it in a correspondingly scaled residue class with the result remaining exact. The benefit is the reduced computation time, especially for intermediate expressions, which might be significantly larger than the bound [von zur Gathen and Gerhard, 2003]. Using the Chinese Residue Theorem we can split a *single* large residue class into multiple smaller residue classes. The latter can be designed to fit into a machine word, where a single operation in a small residue class requires merely a constant time. Thus we can obtain arbitrary precision by increasing the number of small residue classes. The computation itself needs to be sufficiently long to dominate the costs for mapping to and from residue classes. It has no sense to perform these operations before and after each single arithmetic operation. Instead, the whole computation is 'lifted' to the residue classes.

A further effect of introducing *multiple* residue classes is that the computation in each of them can be performed independently. Once the computation has started, each of the residue classes does not require a connection with any of its neighbours. Only when collecting the final result we would again consider multiple residue classes as a whole. Thus, a multiple residue classes arithmetic *is* a parallelisation technique. As the parallelisation occurs on the payload data, with no communication in-between, we consider such arithmetic as a tool for data parallelism. A data parallel arithmetic can be applied to whole classes of algorithms, thus it is in fact a *parallelisation scheme*.

Such methods are traditionally implemented for an *integer* arithmetic. A typical example is the multiple-residue integer arithmetic of Section 7.4. However its rational counterpart is not quite straightforward. It is important to take care of common factors of the numerator and denominator of the fraction with the moduli of the residue classes. Even more important is *how* these common factors are separated. We consider two approaches to a rational multiple-residue arithmetic in Section 7.5. One of these approaches has been introduced in [Gregory and Krishnamurthy, 1984], the other one is our own work. We demonstrate a counterexample for the first method in Section 7.6. The same section shows that our approach is unaffected. A more formal foundation is provided in Section 7.7.

At this point we are done defining the rational multiple-residue arithmetic and showing its properties. We turn to more practical issues. Using an implementation of **LU** factorisation, we present and evaluate a data parallel program in Section 7.8. Finally, Section 7.9 presents some related work, including a comparison with Maple. Section 7.10 concludes and lists up directions for future work.

This chapter is based on [Lobachev and Loogen, 2010a,b, Lobachev, 2011a], our first work on this topic is [Lobachev, 2007]. It is an alternative development of an arithmetic, introduced in [Gregory and Krishnamurthy, 1984]. A new textbook [Kornerup and Matula, 2010] corrects the original presentation of Gregory and Krishnamurthy, yielding the same definition of the common factors extraction as the matter of discussion here. However, Kornerup and Matula [2010] present only a brief sketch of arithmetic operations. Our pioneer work, presenting the arithmetic in detail, appeared 2007. Further, [Lobachev and Loogen, 2010a] appeared on 7$^{\text{th}}$ of September 2010, whereas [Kornerup and Matula, 2010] appeared on 30$^{\text{th}}$ of September. Priority aside, we learnt of [Kornerup and Matula, 2010] only in March 2011, in the process of the preparation of this work. It is genuinely an independent discovery. Such attention to the rational multiple-residue arithmetic only emphasises its importance. For example, [Ebert, 1983, Buchberger et al., 1985, Winkler, 1988, Arnold, 2003, Idrees et al., 2011] search for a residue-based approach to Gröbner bases computation. We leave this particular topic for the future work.

7.1 Why We Cannot Use Vulgar Fractions

Recapitulate the definition of rational numbers as equivalence classes on pairs of integers. We can straightforwardly encode it in Haskell:

```
data (Integral a) ⇒ Fraction a = Fraction {
  numerator :: Integral a ⇒ a
  denominator :: Integral a ⇒ a
}
```

The first element of the pair is called numerator and the second is a denominator. Of course this representation is not unique, but using a *greatest common divisor* (GCD) and limiting the sign makes it unique. The standard, *reduced* representation of a/b is $\frac{a/g}{b/g}$, where $g = \gcd(a, b)$. We know that following holds:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}, \qquad \frac{a}{b}\frac{c}{d} = \frac{ac}{bd}.$$

When computing with fractions, we can observe the increasing values of numerator and denominator, whilst the 'absolute value' of the fraction is still small enough. An example is $^{99999999}/_{100000000}$, which has a small absolute value ≈ 1. Thus we need a measure for the components of the fraction.

Definition 7.1 (Farey fractions). All vulgar fractions a/b satisfying $|a| \leq N$, $|b| \leq N$ for some $N \in \mathbb{N}$ are called *Farey fractions* of order N. We sometimes denote them as F_N.

John Farey, Sr., *1766, †6.1.1826, was a geologist. He introduced the *Farey sequence* from the definition above. Interestingly, he was curious about some properties of this sequence, but the key property of the Farey fractions was not proved by him [Hardy, 1992, MacTutor, 2011].

Definition 7.2. We call $\|a/b\|_F = \max\{|a|, |b|\}$ a *Farey measure* of the fraction a/b.

Lemma 7.3. *The Farey measure is not a norm.*

Proof. A norm is positive scalable, subadditive and positive definite. The mapping $\|\cdot\|_F$ is not positive scalable: $\|x\frac{a}{b}\|_F = \|\frac{xa}{b}\|_F = \max\{|xa|, |b|\} \neq |x| \max\{|a|, |b|\} = x\|\frac{a}{b}\|_F$. □

The Farey measure is a tool to estimate a complexity of a particular fraction arithmetic operation applied to given fractions. Unfortunately it grows pretty much along the operations. If we assume $|a| \geq |b| \geq |c| \geq |d|$, then

$$\left\|\frac{a}{b} + \frac{c}{d}\right\|_F = \left\|\frac{ad + bc}{bd}\right\|_F \leq 4|a| \qquad \left\|\frac{a}{b}\frac{c}{d}\right\|_F = \left\|\frac{ac}{bd}\right\|_F \leq 2|a|. \tag{7.1}$$

We return to fractions in Sections 7.3 and 7.5. Before we do so, we need to discuss integer residue classes next.

7.2 Single Integer Residue Class

The classic residue classes, we call here *single integer* residue classes modulo m are integers from 0 to $m-1$, i. e., residues of integral division by m, with corresponding mathematical operations. More generally, such residue classes can be described as residues modulo some ideals. Considering some ring R and an ideal $\mathfrak{m}$ we can build a factor ring $R/\mathfrak{m}$. A known result is that if $\mathfrak{m}$ is a maximal ideal[1], then $R/\mathfrak{m}$ is a field. Every maximal ideal is also a prime ideal [Lang, 2002, Grove, 2004]. Getting back to our setting $R = \mathbb{Z}$, given the residue ring $\mathbb{Z}/\mathfrak{m}$, if the number m is prime, then $\mathbb{Z}/\mathfrak{m}$ is a field. Such fields are often called Galois fields. These are named after Évariste Galois, *25.10.1811, †31.5.1832, one of the most tragic figures in the history of mathematics. Galois died with 20 in a duel. In the night before the duel Galois prepared the final version of the manuscript with his ideas. It was published only in 1846. Most known because of the Galois theory, Galois is regarded to be one of the fathers of the group theory.

So $\mathbb{Z}/\mathfrak{m} = \mathbb{Z}/m\mathbb{Z}$ is the set of equivalence classes of residues of division by m with corresponding operations. Clearly, this set is $\{0, \ldots, m-1\}$. But what about the arithmetic operations? We denote the elements of the residue class with $a \pmod m$, in a contrast to some $a \in \mathbb{Z}$. Then, let $\circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$ be arithmetic operations in $\mathbb{Z}/\mathfrak{m}$ and let $\circ \in \{+, -, \cdot, /\}$ denote arithmetic operations in $\mathbb{Z}$. Then $(a \bmod m) \circledcirc (b \bmod m) = (a \circ b) \bmod m$ holds. Let us summarise.

Definition 7.4 (Residues and division). We denote integer division with $a = bm + r$ as $b = a \operatorname{div} m$ and $r = a \bmod m$. The latter forms a *residue class* for given m. We define

$$(\mathbb{Z}_m, \oplus, \odot) := (\mathbb{Z}, +, \cdot)/\mathfrak{m} = (\mathbb{Z}, +, \cdot)/\langle m \rangle.$$

It holds $x \circledcirc y := (x \circ y) \bmod m$ for $x, y \in \mathbb{Z}_m$ and $\circ \in \{+, -, \cdot, /\}$. An element $a \bmod m$ of $\mathbb{Z}_m = \{0, \ldots, m-1\}$ is denoted with $|a|_m$. With a small abuse of notation, we will write simply $\mathbb{Z}_m$ for $(\mathbb{Z}_m, \oplus, \odot)$. A residue class modulo *multiple* residues $\boldsymbol{\beta} = [m_1, \ldots, m_n]$ is defined as $\mathbb{Z}_{\boldsymbol{\beta}} := \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$. The corresponding arithmetic operations will be defined later.

[1] See Definition A.20 on page 196 in the Appendix for the definition of maximal and prime ideals. See also Section A.6 for the algebraic background.

Algorithm 15 Standard extended euclidean algorithm.

Require: $\mathbf{A} = \begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix}$.

1: **procedure** EEA($\mathbf{A}$)
2: **if** $a_2 = 0$ **then return** $[a_1, b_1]$.
3: **else**
4: Let $c \leftarrow a_1 \operatorname{div} a_2$ and $\mathbf{B} \leftarrow \begin{pmatrix} a_2 & b_2 \\ a_1 - ca_2 & b_1 - cb_2 \end{pmatrix}$
5: **return** EEA($\mathbf{B}$)
6: **end if**
7: **end procedure**

Ensure: $[a, b]$ with $a = \gcd(a_1, a_2)$.

Algorithm 16 Rational-to-integer mapping.

Require: A fraction a/b, an integer m with $a \perp m$, $b \perp m$, $a \perp b$

1: Start Algorithm 15 with input matrix

$$\begin{pmatrix} m & 0 \\ b & a \end{pmatrix},$$

resulting in $[x, y]$
2: **if** $x \neq 1$ **then return failure**.
3: **else return** y, the second element of the output vector.
4: **end if**

Ensure: an integer y, representing $|a/b|_m$ or a **failure** if $\gcd(b, m) \neq 1$.

Algorithm 17 Mapping integer to Farey fraction.

Require: an integer $x = a/b \pmod m$, m.

1: Compute n from m per (7.2).
2: Start Algorithm 15 with seed matrix

$$\begin{pmatrix} m & 0 \\ x & 1 \end{pmatrix}.$$

3: In each call to the procedure EEA check, whether $|a_1|$ and $|b_1|$ are both $\leq N$. If so, return the fraction a_1/b_1. If Algorithm 15 terminates without producing such a pair of numbers, **return failure**.

Ensure: either a Farey fraction a/b or a **failure**.

7.3 Mapping a Fraction to Integer and Back

The well-known notion of an *extended euclidean algorithm* (EEA) in a matrix-vector form can be defined as in Algorithm 15. The extended euclidean algorithm is the 'normal' euclidean algorithm with additional columns. We implement Algorithm 15 in Figure 7.1, see top of the figure. The 2×2 matrix is represented by a nested pair of pairs. The `Integral` type class is a Haskell notion for ring $\mathbb{Z}$, which abstracts from the integer representation.

Definition 7.5. We define the residue-based representation of the rationals $|a/b|_m$ as elements of $\hat{I}_m$. The elements of $\hat{I}_m$ are *integers* in the residue arithmetic modulo m. The notion of $|a/b|_m$ stands for an integer modulo m, congruent to $|a|_m \odot |b^{-1}|_m$. Here $\odot$ denotes the multiplication modulo m.

The residue-based representation of a fraction in $\hat{I}_m$ for some m is an *integer* modulo m, see [Gregory, 1981, Gregory and Krishnamurthy, 1984]. So $\hat{I}_m$ is not a field of Hensel's p-adic numbers, for which please refer to [Grove, 2004] or further literature. It holds $\hat{I}_m \subsetneq \mathbb{Z}_m$, i.e., not each value modulo m can be identified with above representation of fractions. We can compute $|a/b|_m$ efficiently, using EEA.

```
eeaStep :: (Integral a) ⇒ ((a, a), (a, a)) → ((a, a), (a, a))
eeaStep ((a1, a2), (b1, b2)) = ((a2, a3), (b2, b3))
    where t = a1 `div` a2
          a3 = a1 `mod` a2
          b3 = b1 - t*b2

eea :: (Integral a) ⇒ ((a, a), (a, a)) → ((a, a), (a, a))
eea ((a1, a2), (b1, b2))
  | a2 == 0 = ((a1, a2), (b1, b2))
  | otherwise = eea $ eeaStep ((a1, a2), (b1, b2))

convertFraction :: (Integral i) ⇒ Fraction i → i → Mod i
convertFraction (F x y) p
  = let ((d,_),(r,_)) = eea ((p,y),(0,x))
    in if d ≠ 1 then error "convertFraction" else makeZ r p
-- Type Mod i and function makeZ are explained in the next figure.
convertFraction' :: (Integral i) ⇒ i → i → i → Mod i
-- uncurried version, omitted

eeaSearch :: (Integral a) ⇒ ((a, a), (a, a)) → a → Maybe (a, a)
eeaSearch ((a1, a2), (b1, b2)) n
  | a2==0 = Nothing
  | a2 ≠ 0 && not (criteria a2 b2 n)
    = flip eeaSearch n $ eeaStep ((a1, a2), (b1, b2))
  | otherwise = Just (a2, b2)
  where criteria x y n = abs x < n && abs y < n

restoreFraction :: (Integral i) ⇒ i → i → i → Maybe (i, i)
restoreFraction a m n = eeaSearch ((m, a), (0, 1)) n
  where n = nFromM m -- converts m to n per Wang
```

Figure 7.1: *A generic (Algorithm 15) and two special (Algorithms 16 and 17) implementations of extended euclidean algorithm in Haskell.*

Lemma 7.6. *Algorithm 16 does not fail, if $b \perp m$, i.e.,* $\gcd(b, m) = 1$.

Proof. The only reason of failure of Algorithm 16 is $x = \gcd(b, m) \neq 1$. □

Algorithm 16 returns the desired *recoverable* result if a *bound* on m and a/b holds. This recovery happens in Algorithm 17. It is rigorously discussed [Gregory, 1981, Wang, 1981, Wang et al., 1982, Kornerup and Gregory, 1983, Gregory and Krishnamurthy, 1984, Sasaki and Sasaki, 1992, Collins and Encarnación, 1995, Sasaki et al., 2002, Monagan, 2004, Lobachev, 2007, 2011a]. We summarise.

Theorem 7.7. *If*

$$N \leq \sqrt{\frac{m}{2}} \tag{7.2}$$

holds, it is possible to recover the original Farey fraction a/b of order N from integer $|a/b|_m$.

Note that above theorem essentially relates $\|a/b\|_F$ and the scale m of the residue system. We want to designate [Wang et al., 1982] as the first proof of Theorem 7.7 known to us. We name this mapping 'rational reconstruction' [von zur Gathen and Gerhard, 2003]. We have shown which criteria the input of the algorithm needs to fulfil, for the algorithm to succeed. The correctness of Algorithm 17 and the uniqueness of the fraction a_1/b_1 are shown in the next theorem. We follow [Kornerup and Gregory,

```haskell
data Mod a = Z a a

makeZ :: Integral a ⇒ a → a → Mod a
makeZ a m = ... -- essentially: apply the Z constructor

lift2z :: Integral a ⇒ (a → a → a) → Mod a → Mod a → Mod a
lift2z f (Z a p) (Z b q)
  | p ≠ q = error "Different residue classes!"
  | otherwise = makeZ (f a b) p

-- P.+ denotes (+) instances for Integral type class
-- from the Prelude. It corresponds to the ring ℤ.
instance (Integral a) ⇒ Num (Mod a) where
    (+) = lift2z (P.+)
    (-) = lift2z (P.-)
    (*) = lift2z (P.*)
instance (Integral a) ⇒ Fractional (Mod a) where
    (/) (Z a p) (Z b q) = ... -- use EEA
```

Figure 7.2: *Required function types for single-residue arithmetic in Haskell.*

1983, Gregory and Krishnamurthy, 1984] and state it without a proof. The proof technique relates the execution tableau of the EEA with *convergents* of continued fractions, see above citations.

Theorem 7.8 (Gregory and Krishnamurthy [1984], Theorem 6.39). *If a/b is a Farey fraction of order N with $|a/b|_m = |k|_m$, $k \in \hat{I}_m$ and $0 < a \leq N$, $0 < |b| \leq N$, with the relation of N and m as above, then there exists an integer i such that*

$$(a, b) = (a_i, b_i),$$

where $\{(a_i, b_i) : i \in \{1, \ldots, l\}\}$ is the sequence of the values, generated in step 4 of Algorithm 15, started with the matrix

$$\begin{pmatrix} m & 0 \\ k & 1 \end{pmatrix}.$$

We show Haskell implementations of Algorithms 16 and 17 in Figure 7.1. Further, we need a Haskell implementation of the single residue arithmetic in $\mathbb{Z}/\mathfrak{m}$. The details are well-known, we present the source code briefly in Figure 7.2. A relevant optimisation, currently not implemented, is the Montgomery multiplication [Montgomery, 1985].

7.4 An Integral Multiple-Residue Arithmetic

> We should not underestimate [Qin's] revolutionary advance, because from [Sun Zi's] single remainder problem, we come at once to the general procedure for solving the remainder problem [...], and there is not the slightest indication of gradual evolution.
>
> Ulrich Libbrecht, *Chinese mathematics in the 13th century*, quoted by MacTutor's *History of Mathematics* on Qin Jiushao.

We have a next hurdle: the bound on the final result, and thus the size of our residue class, grows with the problem size. Further, we want to obtain parallelism. How we master it? The centuries-old *Chinese Residue Theorem*, which we call CRT from now on, tells us, how to reconstruct a value modulo

```
type IMods a = [Mod a]
makeIZ' :: (Integral a, Integral b) ⇒ a → [a] → IMods b
makeIZ' value primes = map (makeZ value) primes
instance (Integral a) ⇒ Num (IMods a) where
    (+) = zipWith (+)
    -- and so on...
```

Figure 7.3: *Implementing the multiple-residue integer arithmetic.*

a large number from several values modulo smaller numbers. The CRT was discovered by Qin Jiushao (秦九韶), *1202, †1261. He was a violent Chinese bureaucrat, his only mathematical legacy is the book *Shushu Jiuzhang* (数书九章, "Mathematical Treatise in Nine Sections"), published in 1247. Still exactly it made Qin Jiushao immortal. Chinese mathematician Sun Zi (孙子), mentioned by U. Libbrecht above, lived between third and fifth century A.D. He authored the book *Sun Zi Suan Jing* (孙子算经, "Sun Zi's Calculation Classic"), mentioning a special case of the CRT. Sun Zi is not the same person as the famous Chinese strategist of the same name.

Let us consider a *multiple*-residue system $\mathbb{Z}_{\boldsymbol{\beta}}$ with more than one residue. Hence, $\boldsymbol{\beta}$ is a vector. Assuming $\boldsymbol{\beta} = [m_1, \ldots, m_n]$, we often write $M = \prod \boldsymbol{\beta}$ or $\prod m_i$ for the product of all m_i.

Theorem 7.9 (Chinese Residue Theorem). *Given residue representations* $[a_1, \ldots, a_n]$ *of some unknown a modulo* $[m_1, \ldots, m_n]$ *with* $m_i \perp m_j$ *for* $1 \leq i, j \leq n$, $i \neq j$, *we can reconstruct a* mod M *with* $M = \prod m_i$.

There is also a formulation of this theorem for principal ideal domains, but the integer version suffices here. There are various proofs to this theorem, both constructive and not, whereas the constructive ones state the possible implementation [Knuth, 1998, Cohen, 2000]. For more research on CRT refer for instance to [Ore, 1952, Fraenkel, 1963, Baker and Pixley, 1975]. But now, having the CRT, we can map our computation onto multiple residue rings, the so called *multiple residue system*. The formulation of the CRT allows us to map integers, we will come to fractions later. Known operations over a single residue ring are performed *independently* at each element of the system. Thus, we have an approach for the data parallelism. More technical details on this are in Section 7.8. We can scatter our system along the PEs and let them do the whole lengthy computation separately. At the end we only need to collect the result and to convert it back with the CRT. This method is widely used, von zur Gathen and Gerhard [2003] call it the 'small primes' approach and use it to drastically reduce the complexity of Gaußian elimination and of polynomial GCD computation. Note that all $a_1, \ldots, a_n$ represent *the same* value a.

Let us take a more abstract view. For the sake of simplicity we consider elements of $\boldsymbol{\beta}$ to be prime numbers. Then for $\boldsymbol{\beta} = [m_1, \ldots, m_n]$ the single residue classes are $\mathbb{Z}_{m_1}, \ldots, \mathbb{Z}_{m_n}$. With $M = \prod m_i$, it holds that

$$\mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n} = \mathbb{Z}_{\boldsymbol{\beta}} \cong \mathbb{Z}_M. \tag{7.3}$$

The equation (7.3), read from left to right, is the CRT, i.e., Theorem 7.9. Some of the constructive proofs of this theorem allow the algorithmic construction of the 'large' residue. We call such proofs *implementations* of CRT, the other name in the literature is 'Chinese Residue Algorithm', *cf.* [von zur Gathen and Gerhard, 2003]. We show a known approach to it below.

Further, equation (7.3) facilitates a background for forth and backwards mappings between $\mathbb{Z}_{\boldsymbol{\beta}}$ and $\mathbb{Z}_M$ as well as for defining the arithmetic. We will present this known result with a notion from functional programming. Recapitulate the following definition.

Definition 7.10 (Map function). For all functions operating on single elements: `f :: a → b`, we define a function `map`, which takes as its arguments such `f` and a *collection* of type `[a]` of elements of type `a`. The function `map` applies `f` to each element of its input collection and combines the results of each such application to its output collection of type `[b]`. Hence, `map` has the type `(a → b) → [a] → [b]` and a partial application `map f` has the type `[a] → [b]`. So we write

Algorithm 18 Mapping an integer to integral multiple-residue.

Require: a vector of primes $\boldsymbol{\beta}$, integer x with no common factors with elements of $\boldsymbol{\beta}$.

1: Compute $|x|_{m_i}$ for all elements m_i of $\boldsymbol{\beta}$.
2: **return** $[|x|_{m_i} : i \in \{1, \ldots, \#\boldsymbol{\beta}\}]$.

Ensure: $|x|_{\boldsymbol{\beta}}$

Algorithm 19 From integral multiple-residue to an integer. Part 1: convert an integer multiple-residue value to mixed-radix representation.

Require: $|x|_{\boldsymbol{\beta}}$.

1: **procedure** MIXED RADIX($|x|_{\boldsymbol{\beta}}$)
2: Set $\mathbf{t}^{(1)} \leftarrow |x|_{\boldsymbol{\beta}}$ and $i \leftarrow 1$.
3: Let $d_0 \leftarrow t_1^{(1)}$ and n is the length of $\boldsymbol{\beta}$.
4: **for** $1 < i < n$ **do**
5: let

$$\mathbf{t}^{(\mathbf{i+1})} \leftarrow \left| \frac{\mathbf{t}^{(\mathbf{i})} - |d_{i-1}|_{\boldsymbol{\beta}_{\{i+1,\ldots,n\}}}}{m_i} \right|_{\boldsymbol{\beta}_{\{i+1,\ldots,n\}}}$$

$$d_i \leftarrow t_{i+1}^{(i+1)}$$

$$i \leftarrow i + 1$$

6: **end for**
7: **return** $\langle d_0, \ldots, d_{n-1} \rangle_{\boldsymbol{\beta}}$
8: **end procedure**

Ensure: $\langle x \rangle_{\boldsymbol{\beta}} = \langle d_0, \ldots, d_{n-1} \rangle_{\boldsymbol{\beta}}$.

```
map :: (a → b) → [a] → [b]
map f xs = [ f x | x ∈ xs ]
```

Corollary 7.11 (ZipWith function). *We define a binary version of* `map`.

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f xs ys = [ f x y | x ∈ xs | y ∈ ys ]
```

Now we can define all four integral multiple-residue arithmetic operation on $\mathbb{Z}_{\boldsymbol{\beta}}$ as a kind of `map` of their single-residue counterparts. Because `map` applies `f` to each single element in an independent manner, such definition of a multiple-residue arithmetic underlines its strength for vectorisation. All of the computation within a single 'residue element' can be done independently from other residue elements. This will be the basis for the parallel implementation in Section 7.8. The implementation of the arithmetic falls back to `zipWith`—a variant of `map` for binary functions. Hence, the type for the multiple-residue system is just a list of single-residues. Now we can sketch Algorithm 18. See Figure 7.3 for implementation details.

Mixed-Radix Representation. We use the *mixed-radix* representation to convert the entries from $\mathbb{Z}_{\boldsymbol{\beta}}$ to the $\mathbb{Z}_M$. This approach is described in detail in [Gregory and Krishnamurthy, 1984]. Also see [Szabo and Tanaka, 1967, Fraenkel, 1985], a parallelisation of this algorithm is described in [Huang, 2006].

Definition 7.12 (Mixed-radix representation). For a representation of an integer x w. r. t. a base vector $\boldsymbol{\rho} = [r_1, \ldots, r_{k-1}]$ we write $\langle x \rangle_{\boldsymbol{\rho}} = [d_0, \ldots, d_{k-1}]$ with $x = d_0 + d_1 r_1 + d_2 r_1 r_2 + \cdots + d_{k-1} r_1 \cdots r_{k-1}$. Naturally, $0 \leq d_i < r_{i+1}$.

It is immediately clear how to compute $s = |x|_M$ from $\langle x \rangle_{\boldsymbol{\beta}}$. It is easy to prove that a mixed-

Algorithm 20 From integral multiple-residue to an integer. Part 2: from mixed-radix representation to an integer.

Require: $\langle x \rangle_{\boldsymbol{\beta}}$

procedure CONVERT MIXED RADIX($\langle x \rangle_{\boldsymbol{\beta}}$)
  Set $s_1 \leftarrow d_{n-1}$.
  Compute the list $S \leftarrow [s_i \leftarrow d_{n-i} + s_{i-1} m_{n-i+1} : \text{ with } i \in \{2, \ldots, n\}]$.
  return the last element of the list S.
end procedure

Ensure: $|x|_M = s_n$.

Algorithm 21 From integral multiple-residue to an integer. Final part.

Require: $|x|_{\boldsymbol{\beta}}$.

Let $\langle x \rangle_{\boldsymbol{\beta}} \leftarrow$ MIXED RADIX($|x|_{\boldsymbol{\beta}}$). // *Compute* $\langle x \rangle_{\boldsymbol{\beta}}$ *with Algorithm 19.*
Let $M \leftarrow \prod \boldsymbol{\beta}$.
Let $|x|_M \leftarrow$ CONVERT MIXED RADIX($\langle x \rangle_{\boldsymbol{\beta}}$) // *Compute* $|x|_M$ *from* $\langle x \rangle_{\boldsymbol{\beta}}$ *with Algorithm 20.*
return $|x|_M$.

Ensure: $|x|_M$, the integer representation of $|x|_{\boldsymbol{\beta}}$ modulo M.

radix representation is unique using repeated division with a remainder. The intriguing case is of course $\boldsymbol{\rho} = \boldsymbol{\beta}$. We can obtain a mixed-radix representation $\langle x \rangle_{\boldsymbol{\beta}}$ from a given multiple-residue representation $|x|_{\boldsymbol{\beta}}$ with Algorithm 19. Then we can convert the mixed-radix representation to $|x|_M$ with $M = \prod \boldsymbol{\beta}$ with Algorithm 20. Combined, we come from $|x|_{\boldsymbol{\beta}}$ to $|x|_M$ in Algorithm 21. But we need to find the mixed-radix representation first. Algorithm 19 shows how to do it. Denote with $\boldsymbol{\beta}_{\{i,\ldots,n\}}$ a reduced vector $[m_i, \ldots, m_n]$, when originally $\boldsymbol{\beta} = [m_1, \ldots, m_n]$. Further notation: $\mathbf{t}_k$ is the k^{th} element of the vector $\mathbf{t}$, in a contrast $\mathbf{t}^{(\mathbf{k})}$ is the whole k^{th} vector $\mathbf{t}$, hence $t_k^{(k)}$ is the k^{th} element of the vector $\mathbf{t}^{(\mathbf{k})}$. The length of $\mathbf{t}^{(\mathbf{i})}$ is reducing in each iteration step. Our implementation is presented in Figure 7.4. Algorithm 21 is a proof of the CRT. With it we can convert integer multiple-residue values to a mixed-radix representation of a single integer. This value, recovered to its standard integer representation, is the desired result of the constructive CRT proof: the reconstructed value. Other implementations of the CRT are widely known and can be found in [Knuth, 1998, Cohen, 2000, von zur Gathen and Gerhard, 2003].

7.5 Rational Multiple-Residue System

Before we begin with the actual content of this section, we would like to clarify some wording. In further we would write something like 'a fraction a/b has (no) common factors with an integer m'. This a bit cryptic message is an abbreviation of 'a numerator a or a denominator b of the reduced fraction a/b has common factors with the integer m', or, in a negative case: 'neither numerator a nor denominator b of the same fraction have common factors with an integer m'.

Note that for more than two rational residues modulo $m_1, m_2, \ldots$, obeying (7.2), the smallest integer m_1 *will* be smaller than the size of the maximal fitting Farey fraction N, *cf.* [Gregory and Krishnamurthy, 1984, Theorem 7.17]. This means, that there are fractions, fulfilling the bound (7.2), but having a common factor with at least m_1 in numerator or denominator.

Such common factors cause the computation of the inverse to fail or lead to zeros found in wrong places. This happens, if a common factor occurs in denominator or numerator appropriately. Clearly, we need to deal with this 'common factor' problem. It is related to the 'unlucky primes' problematics. The latter occurs, e.g., in the modular Gröbner bases computation [Ebert, 1983, Arnold, 2003, Idrees et al., 2011]. This problem was considered from the workpool perspective in [Loidl, 1997].

```
data SingleRadix a = MR a a
                     deriving Eq
type MixedRadix a = [SingleRadix a]

-- omitted because of simplicity
valuesPrimes :: Integral a ⇒ IMods a → [(a, a)]
valuesRadices :: Integral a ⇒ MixedRadix a → [(a, a)]
getPrimes :: Integral a ⇒ IMods a → [a]
takeFirstValue :: (Integral a) ⇒ IMods a → a

restoreIZ :: Integral a ⇒ IMods a → Z a
-- Wrapper around restoreIZ'. In fact we restore Z a!
restoreIZ' :: Integral a ⇒ IMods a → a
restoreIZ' input = let (values, primes) = unzip $ valuesPrimes input
                   in convertMixedRadix $ mixedRadix input

inverses :: (Integral i) ⇒ i → [i] → IMods i
inverses k ps = flip makeIZ ps $ map (inverse k) ps

-- a single step of mixed radix algorithm
mixedRadixStep :: Integral a ⇒ IMods a → IMods a
mixedRadixStep input
  = let (values, primes) = unzip $ valuesPrimes input
        h = head values
        diffs = (tail input) - (makeIZ' h $ tail primes)
        lagrangians = inverses (head primes) (tail primes)
    in diffs * lagrangians

mixedRadix :: Integral a ⇒ IMods a → MixedRadix a
mixedRadix input = zipWith (flip MR) (getPrimes input)
                   $ map takeFirstValue
                   $ takeWhile (λx → length x > 0)
                   $ iterate mixedRadixStep input

convertMixedRadix :: Integral a ⇒ MixedRadix a → Mod a
convertMixedRadix mixed
  = let residue = product primes
        (values, primes) = unzip $ valuesRadices mixed
        primes' = 1:primes
        -- (P.+) is (+) from standard Prelude, same with (*)
        conv (v, p) acc = (P.*) ((P.+) acc v) p  -- (acc + v) * p
    in flip makeZ residue $ foldr conv 0 $ zip values primes'
```

Figure 7.4: *Converting from $\mathbb{Z}_{\boldsymbol{\beta}}$ to $\mathbb{Z}_M$ with mixed-radix algorithm.*

7.5.1 The Mappings

Definition 7.13 (Elements)**.** We define an element of a rational multiple-residue system $\mathbb{W}_{\boldsymbol{\beta}}$ as follows. Let n be the length of prime list $\boldsymbol{\beta}$. Then such element is a list of pairs

$$[(u_i, v_i) : i = 1, \ldots, n].$$

Here the components u_i are the residues and v_i are the powers of corresponding elements of $\boldsymbol{\beta}$.

The implementation is in Figure 7.5. Gregory and Krishnamurthy [1984] define a similar residue sys-

```
data FSingleMod a = FM (Mod a) a
type FMods a = [FSingleMod a]

nFromM, mFromN :: Integral i ⇒ i → i
-- convert M to N
makeFZ :: Integral i ⇒ Fraction i → FMods i
-- forth mapping, see next figure
restoreFZ :: Integral i ⇒ FMods i → Maybe (Fraction i)
-- backwards mapping, see second next figure
```

Figure 7.5: *Basic outline of rational multiple-residue implementation in Haskell.*

Algorithm 22 Common outline of forth rational multiple-residue mapping.

Require: a fraction a/b, residues $\boldsymbol{\beta} = [m_1, \ldots, m_n]$.

1: Extract common factors v_i of all m_i and a/b. Remember $v_1, \ldots, v_n$.
2: Convert the resulting fraction to an integer modulo $M = \prod \boldsymbol{\beta}$ with Algorithm 16.
3: Convert the resulting integer to a multiple-residue system modulo $\boldsymbol{\beta}$ with Algorithm 18. Store the results in a list $[u_1, \ldots, u_n]$.
4: **return** list of pairs $[(u_1, v_1), \ldots, (u_n, v_n)]$.

Ensure: rational multiple-residue representation of a/b being $[(u_1, v_1), \ldots, (u_n, v_n)]$.

tem, we call $\mathbb{M}_{\boldsymbol{\beta}}$ here. Its elements are similar to those of $\mathbb{W}_{\boldsymbol{\beta}}$, and the arithmetical operations definitions coincide. The major difference lies in how the forth and backwards mappings are defined. Noteworthy, [Gregory, 1981, Kornerup and Gregory, 1983] define a yet another residue system, which differs from both $\mathbb{M}_{\boldsymbol{\beta}}$ and $\mathbb{W}_{\boldsymbol{\beta}}$ in not separating out the powers of primes v_i from the residues u_i. It is the predecessor of $\mathbb{M}_{\boldsymbol{\beta}}$. We focus here on $\mathbb{W}_{\boldsymbol{\beta}}$ and handle $\mathbb{M}_{\boldsymbol{\beta}}$ in a more detail in Section 7.6.

Forth mapping. Given a fraction a/b and $\boldsymbol{\beta} = [m_1, m_2, \ldots, m_n]$, satisfying (7.2) with some $N \in \mathbb{N}$ such that $a/b \in F_N$ (see Definition 7.1), we define iteratively $a^{(i)}/b^{(i)}$ and $v_1, \ldots, v_n$ as shown in Algorithm 22. We assume $m_i \perp m_j$ for all pairs (m_i, m_j) from $\boldsymbol{\beta}$ with $i \neq j$. Said Algorithm 22 consists of $\varphi_1 \circ \varphi_2 \circ \varphi_3$, where φ_i is essentially the i^{th} step of the algorithm. We need to be careful of φ_1, the common factor extraction. Exactly the difference between (7.4 WRONG) and (7.4 RIGHT) is the difference between the approach from [Gregory and Krishnamurthy, 1984] and our approach.

$$\begin{aligned} \frac{a^{(1)}}{b^{(1)}} &= \frac{a}{b} m_1^{v_1} \\ \frac{a^{(2)}}{b^{(2)}} &= \frac{a^{(1)}}{b^{(1)}} m_2^{v_2} \\ &\vdots \\ \frac{a^{(n)}}{b^{(n)}} &= \frac{a^{(n-1)}}{b^{(n-1)}} m_n^{v_n} \end{aligned} \tag{7.4 WRONG}$$

The forth mapping for $\mathbb{M}_{\boldsymbol{\beta}}$, defined in [Gregory and Krishnamurthy, 1984], takes in step 2 of Algorithm 22 the value $a^{(n)}/b^{(n)}$ from (7.4 WRONG) for *all* residue classes in the system. Our approach, following [Lobachev, 2007, 2011a], is to remove each time only the factors, we need to remove. With the following equation, the forth mapping for $\mathbb{W}_{\boldsymbol{\beta}}$ takes in the same step 2 the value $a^{(i)}/b^{(i)}$ for i^{th} residue class in the system from (7.4 RIGHT). Hence, the key difference between our approach and

Algorithm 23 Forth rational multiple-residue mapping to $\mathbb{W}_{\boldsymbol{\beta}}$.

Require: fraction a/b, residues $\boldsymbol{\beta} = [m_1, \ldots, m_n]$.

1: Extract common factors v_i from a/b per (7.4 RIGHT).
This results in lists $[v_1, \ldots, v_n]$ and $[a^{(1)}/b^{(1)}, \ldots, a^{(n)}/b^{(n)}]$.

2: Convert each $a^{(i)}/b^{(i)}$ to a value u_i modulo m_i with Algorithm 16 for all $i \in \{1, \ldots, n\}$. *// i.e., assign to* $[u_1, \ldots, u_n]$ *the result of* `zipWith` *with Algorithm 16 of the lists* $[a^{(1)}/b^{(1)}, \ldots, a^{(n)}/b^{(n)}]$ *and* $\boldsymbol{\beta}$.

3: **return** an element $[(u_1, v_1), \ldots, (u_n, v_n)]$ of $\mathbb{W}_{\boldsymbol{\beta}}$

Ensure: a rational multiple-residue representation of a/b.

Algorithm 24 Backward mapping from $\mathbb{W}_{\boldsymbol{\beta}}$ to $\mathbb{Q}$.

Require: $[(u_1, v_1), \ldots, (u_n, v_n)] \in \mathbb{W}_{\boldsymbol{\beta}}$, $\boldsymbol{\beta} = [m_1, \ldots, m_n]$

1: Compute $M = \prod \boldsymbol{\beta}$ and $N = \sqrt{M/2}$. *// Can be precomputed.*

2: Compute $a'/b' = m_1^{v_1} \cdots m_n^{v_n}$.

3: For each $i \in \{1, \ldots, n\}$ distort the values of u_i. Let

$$\hat{u}_i \leftarrow u_i / \prod_{j \neq i} m_j^{v_j}.$$

4: Consider $[\hat{u}_1, \ldots, \hat{u}_n]$ as an integer multiple-residue value in $\mathbb{Z}_{\boldsymbol{\beta}}$. Find its representation q in $\mathbb{Z}_M$ with an implementation of CRT (e.g., with Algorithm 21).

5: Find a Farey fraction a/b of order N, such that $|a/b|_M \equiv q \pmod{M}$ with Algorithm 17. If this does not succeed **return** a **failure**.

6: **return** the fraction aa'/bb'.

Ensure: recovered fraction aa'/bb' or a **failure**.

$\mathbb{M}_{\boldsymbol{\beta}}$ is in

$$\begin{aligned} \frac{a^{(1)}}{b^{(1)}} &= \frac{a}{b} m_1^{v_1} \\ \frac{a^{(2)}}{b^{(2)}} &= \frac{a}{b} m_2^{v_2} \\ &\vdots \\ \frac{a^{(n)}}{b^{(n)}} &= \frac{a}{b} m_n^{v_n} \end{aligned} \qquad \text{(7.4 RIGHT)}$$

With these equations, the forth mapping for $\mathbb{W}_{\boldsymbol{\beta}}$ takes in step 2 of Algorithm 22 the value $a^{(i)}/b^{(i)}$ for i^{th} residue class in the system. The forth mapping for $\mathbb{M}_{\boldsymbol{\beta}}$ extracts *all* factors from the input fraction for all residue classes. Unfortunately, this leads to an instable addition. Our approach does not have such a problem. We will show an example, underlining the difference of both approaches in Section 7.6, after the definition of arithmetic operations.

The idea of the forth mapping algorithm for $\mathbb{W}_{\boldsymbol{\beta}}$ is to take a fraction a/b and a list of primes $\boldsymbol{\beta} = [m_1, \ldots, m_n]$; then to execute Algorithm 22 with (7.4 RIGHT) in step 2. The result is the rational multiple-residue representation of a/b as an element of $\mathbb{W}_{\boldsymbol{\beta}}$. Algorithm 23 shows the forward mapping in full detail. The implementation of this algorithm is in Figure 7.6. Note that Algorithm 23 converts each fraction $a^{(i)}/b^{(i)}$ separately, resulting in n calls of Algorithm 16. Thus, among $u_1, \ldots, u_n$ some represent *different* values in $\mathbb{Q}$ if at least one $v_i \neq 0$ for some $i \in \{1, \ldots, n\}$.

Backward mapping. The backward mapping is defined in Algorithm 24. Denoting ψ_I for multiple-residue integer reconstruction, ψ_R for fraction reconstruction, ψ_1 for factoring out, and ψ_2 for restoring the factored out values, Algorithm 24 is essentially $\psi_1 \circ \psi_I \circ \psi_R \circ \psi_2$. The implementation of the latter

```
detectPower :: (Integral i, Num n) ⇒ i → i → (n, i)
-- Code omitted. Example: detectPower 40 2 = (3, 5)

convertFraction' :: (Integral i) ⇒ i → i → i → Mod i
-- uncurried EEA implementation from above

extractFactors :: (Integral i, Num n) ⇒ i → [i] → [(n, i)]
extractFactors x ps = map (detectPower x) ps

makeFZ' :: Integral i ⇒ i → i → [i] → FMods i
makeFZ' a b ps | gcd a b == 1 = let (ws, ys) = unzip $ extractFactors a ps
                                    (qs, zs) = unzip $ extractFactors b ps
                                    vs = zipWith (-) ws qs -- well-defined
                                    cs = zipWith3 convertFraction' ys zs ps
                                in zipWith FM cs vs
               | otherwise = ... -- recursive call

makeFZ :: Integral i ⇒ Fraction i → FMods i
makeFZ = ... -- a trivial constructor expansion
```

Figure 7.6: *Forward mapping (Algorithm 23).*

```
getM :: Integral i ⇒ FMods i → Integer
-- returns the product of all primes in the system

stripPowers :: Integral i ⇒ FMods i → (i, i, FMods i)
-- set v_i = 0 for all i and compensate

restoreFZ' :: Integral i ⇒ FMods i → (Maybe (i,i), (i,i))
restoreFZ' x = let m = getM x
                   n = nFromM m
                   (nom, denom, hatUs) = stripPowers x
                   z = convertToIntResidues hatUs
                   r = toIntegral $ restoreIZ' z
                   e = restoreFraction r m n
               in (e, (nom, denom))

restoreFZ :: Integral i ⇒ FMods i → Maybe (Fraction i)
restoreFZ = ... -- compute in Maybe monad the product of fraction e with nom/denom
```

Figure 7.7: *The outline of an implementation of the backwards mapping (Algorithm 24).*

algorithm is presented in Figure 7.7. We show the correctness of Algorithms 23 and 24 in Section 7.7, basing on [Lobachev, 2007, 2011a, Lobachev and Loogen, 2010b].

Image of ψ. How does the image set of Algorithm 23 look like? This set consists of Farey fractions of corresponding order N *and* their products with powers $v_1, \ldots, v_n$ of $m_1, \ldots, m_n$. For the restricted values of v_i the shape of this set is shortly discussed in [Lobachev, 2007]. Figure 7.8 on the next page gives an intuition. Its top part depicts all Farey fractions of order four. The middle and bottom parts of Figure 7.8 show the very same Farey fractions, combined with small powers of 3 and 5. Note the different scale of these two plots. If we do not restrict the values of v_i, then the said set is infinite. Further questions on the shape of this set are open.

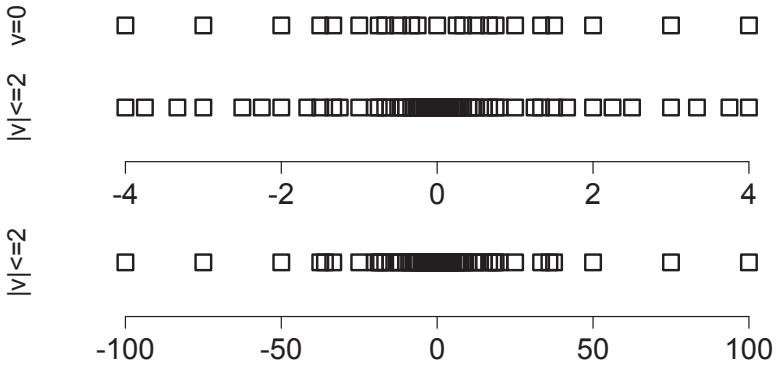


Figure 7.8: *Farey fractions with powers of small integers. Top: Farey fractions of order 4. Middle and bottom: Farey fractions of order 4 multiplied with fitting powers of* 3 *and* 5 *such that* $v_i \in \{-2, -1, 0, 1, 2\}$. *Note, this includes the factors of* 3, *already present in numerator or denominator of fractions in* F_4. *In the middle we show only the values between* -3 *and* 3. *The bottom plot shows the complete set.*

Our implementation defines v_i as types of an `Integral` type class. Hence, it is possible to use both hardware integers and arbitrary precision integers. The choice of the first should increase the speed of the computation. It limits the absolute size of v_i to 2^{63} on a 64 bit machine. This should be not an issue for any reasonable input sizes, as we discuss the *powers* of primes, which itself have the magnitude of 2^{64}. Still for the theoretical purposes it is possible to use arbitrary precision integers for values v_i. Combined with arbitrary precision integers for $\hat{I}_{m_i}$, this makes the set $\mathbb{W}_{\boldsymbol{\beta}}$ capable of representing any fractions, which will fit into the memory of a machine. We can also achieve very good scaling of our methods with hardware integers in both components by simply increasing the number of residue classes.

7.5.2 The Arithmetic

We need to define the actual arithmetic on $\mathbb{W}_{\boldsymbol{\beta}}$. We stress again that *all* arithmetic operations are independent in separate components. Each operation is defined for a single residue (use `zipWith`!). These definitions coincide with ones for $\mathbb{M}_{\boldsymbol{\beta}}$ from [Gregory and Krishnamurthy, 1984], but not with those from [Gregory, 1981]. We begin with the definition of multiplication, since it is the simplest operation in the system.

Definition 7.14. The *product* of (u, v) and (μ, v) modulo m is defined as $(|u\mu|_m, v + v)$.

The implementation is straightforward: `(FM u1 v1) * (FM u2 v2) = FM (u1*u2) (v1+v2)`.

Definition 7.15 (Multiplicative Inverse)**.** The inverse of (u, v) is $(|u^{-1}|_m, -v)$.

Note, it is easy and well-known, how to compute $|u^{-1}|_m$, the multiplicative inverse of u modulo m with EEA, Algorithm 15, for such u and m, that $u \perp m$ [Knuth, 1998]. It coincides with computing an integer representation of a fraction $1/u$ with Algorithm 16, a standard approach in residue rings.

The sum of (u, v) and (μ, v) modulo m is $(|u + \mu|_m, v)$ if $v = v$ and just (u, v) for $|v| < |v|$ with a single exception for sum of something with zero being the non-zero summand, regardless of the power of m. A more formal definition follows.

```
instance (Integral a) ⇒ Num (FMods a) where
    (+) = zipWith (+)
    (-) = zipWith (-)
    (*) = zipWith (*)
instance (Integral a) ⇒ Fractional (FMods a) where
    (/) = zipWith (/)
```

Figure 7.9: *Instances of* `Num` *and* `Fractional` *for* $\mathbb{W}_\beta$.

Definition 7.16 (Addition)**.** The sum of (u, v) and (μ, v) modulo m is defined as follows. Recall that $u \oplus \mu = |u + \mu|_m$. We write in this table v for positive values, $-v$ for negative and 0 for zero.

$+$	$(0, z)$	(u, v)	$(u, 0)$	$(u, -v)$
$(0, \zeta)$	$(0, 0)$	(u, v)	$(u, 0)$	$(u, -v)$
(μ, v)	(μ, v)	A	$(u, 0)$	$(u, -v)$
$(\mu, 0)$	$(\mu, 0)$	$(\mu, 0)$	$(u \oplus \mu, 0)$	$(u, -v)$
$(\mu, -v)$	$(\mu, -v)$	$(\mu, -v)$	$(\mu, -v)$	B

The two subcases are:

$$A = \begin{cases} (u, v) & \text{if } v < v \\ (u \oplus \mu, v) & \text{if } v = v \\ (\mu, v) & \text{if } v > v \end{cases} \qquad B = \begin{cases} (u, -v) & \text{if } -v < -v \\ (u \oplus \mu, v) & \text{if } v = v \\ (\mu, v) & \text{if } -v > -v \end{cases}$$

Further, z and ζ are in $\mathbb{Z}$. The zero element is not unique because of $(0, z)$ with $z \neq 0$, but we *norm* it to the standard representation $(0, 0)$.

Definition 7.17 (Additive Inverse)**.** The additive inverse of (u, v) modulo m is $(|-u|_m, v)$.

The actual arithmetic operations on $\mathbb{W}_\beta$ are defined by lifting the above single-element operations with `zipWith` to lists, as shown in Figure 7.9. The code for addition, the most complicated operation even for single-element inputs, is presented in Figure 7.10.

7.6 Counterexample for $\mathbb{M}_\beta$

Now, having the definition of the arithmetic operations, we can show that our approach is better than $\mathbb{M}_\beta$ from [Gregory and Krishnamurthy, 1984]. The arithmetic operations coincide with $\mathbb{W}_\beta$. Consider an example computation [Lobachev, 2007]. Let $a = 1/21$ and $b = 1/3$. We compute the sum $a + b$ in $\mathbb{M}_\beta$ modulo $\boldsymbol{\beta} = [5, 7, 11, 13]$. Per (7.2), all fractions a/b with $\|a/b\|_F \leq 50$ are on the safe side, as $N = 50 = \left\lfloor \sqrt{M/2} \right\rfloor = \left\lfloor \sqrt{\prod \boldsymbol{\beta}/2} \right\rfloor$.

Constructing the inputs in $\mathbb{M}_\beta$. As $\mathbb{M}_\beta$ needs to extract all factors of elements of $\boldsymbol{\beta}$ from all elements of the residue system, we obtain the following representations. The value $a = 1/21$ has a common factor with 7 in its denominator. So, we need to extract it: $v_2 = -1$. All other v_i are zero. The resulting value is 1/3, which luckily coincides with b. The latter has no common factors with any m_i. We map b modulo $\boldsymbol{\beta}$. The inverses of 3 modulo $[5, 7, 11, 13]$ are respectively $[2, 5, 4, 9]$. So, the result is $[(2, 0), (5, -1), (4, 0), (9, 0)]$ for a and $[(2, 0), (5, 0), (4, 0), (9, 0)]$ for b.

Mapping back from $\mathbb{M}_\beta$. The sum of a and b is $c := [(4, 0), (5, -1), (8, 0), (5, 0)]$, we simply add the components in the case '$v = v$', which occurs always up to $m_2 = 7$. Here we need to add $(5, -1)$ and $(5, 0)$. From the definition of the addition $(5, -1)$ ensues.

```
instance (Integral a) ⇒ Num (FSingleMod a) where
    (+) x y = addSingle x y
    (-) x y = x + (additiveInverseSingle y)
    -- etc.

addSingle :: (Integral a) ⇒ FSingleMod a → FSingleMod a → FSingleMod a
addSingle (FM (Z 0 p) _) (FM (Z 0 p') _) | p==p' = FM (Z 0 p) 0
addSingle (FM (Z 0 _) _) y = y
addSingle x (FM (Z 0 _) _) = x
addSingle (FM u 0) (FM u' 0) = FM (u+u') 0
addSingle (FM u v) (FM u' 0)  | v >0  = FM u' 0
                              | v <0  = FM u v
addSingle (FM u 0) (FM u' v') | v'>0  = FM u 0
                              | v'<0  = FM u' v'
addSingle (FM u v) (FM u' v') | v<v'  = FM u v
                              | v>v'  = FM u' v'
                              | v==v' = FM (u+u') v
addSingle _ _ = error "Bad case!" -- never happened

additiveInverseSingle (FM (Z u p) v) = FM (Z (p-u) p) v
```

Figure 7.10: *Additive operations in a single fractional residue class.*

$\boldsymbol{\beta}$	5	7	11	13
$\lvert t^{(1)}\rvert_{\boldsymbol{\beta}_{\{1,\ldots,4\}}} = \lvert\hat{c}\rvert_{\boldsymbol{\beta}}$	4	5	8	5
$\lvert d_0\rvert_{\boldsymbol{\beta}_{\{1,\ldots,4\}}}$	$\boxed{4}$	4	4	4
$\lvert t^{(1)} - d_0\rvert_{\boldsymbol{\beta}_{\{2,\ldots,4\}}}$		1	4	1
$m_1^{-1} \pmod{\boldsymbol{\beta}_{\{2,\ldots,4\}}}$		3	9	8
$\lvert t^{(2)}\rvert_{\boldsymbol{\beta}_{\{2,\ldots,4\}}}$		3	3	8
$\lvert d_1\rvert_{\boldsymbol{\beta}_{\{2,\ldots,4\}}}$		$\boxed{3}$	3	3
$\lvert t^{(2)} - d_0\rvert_{\boldsymbol{\beta}_{\{3,4\}}}$			0	5
$m_2^{-1} \pmod{\boldsymbol{\beta}_{\{3,4\}}}$			8	2
$\lvert t^{(3)}\rvert_{\boldsymbol{\beta}_{\{3,4\}}}$			0	10
$\lvert d_2\rvert_{\boldsymbol{\beta}_{\{3,4\}}}$			$\boxed{0}$	0
$\lvert t^{(3)} - d_2\rvert_{\boldsymbol{\beta}_{\{4\}}}$				10
$m_3^{-1} \pmod{\boldsymbol{\beta}_{\{4\}}}$				6
$\lvert t^{(4)}\rvert_{\boldsymbol{\beta}} = \lvert d_3\rvert_{\boldsymbol{\beta}_{\{4\}}}$				$\boxed{8}$

Table 7.1: *The execution of Algorithm 19 for the $\mathbb{M}_{\boldsymbol{\beta}}$ counterexample.*

Let us map this result back to $\mathbb{Q}$. Using the backward mapping for $\mathbb{M}_{\boldsymbol{\beta}}$ from [Gregory and Krishnamurthy, 1984], we extract the factor, residing in $v \neq 0$. So we map back $\hat{c} = [(4,0),(5,0),(8,0),(5,0)]$ and remember the factor $1/7$. The tableau of the execution of the mixed-radix algorithm (i.e., Algorithm 19) is in Table 7.1. The $\boxed{\text{boxed}}$ values are the results of the algorithm execution. As before, $\boldsymbol{\beta}_{\{i,\ldots,n\}}$ denotes a reduced vector $[m_i,\ldots,m_n]$, for $\boldsymbol{\beta} = [m_1,\ldots,m_n]$. We see, $\langle 4,3,0,8\rangle_{\boldsymbol{\beta}}$ is the mixed-radix representation of the *integer* multiple-residue value $[4,5,8,5] \in \mathbb{Z}_{\boldsymbol{\beta}}$. We can easily recover the value in $\mathbb{Z}_M$, namely $\lvert\hat{c}\rvert_M = 4 + 3\cdot 5 + 0\cdot 5\cdot 7 + 8\cdot 5\cdot 7\cdot 11 = 3099$, from the mixed-radix representation.

Next we need to map $3099 \pmod{M}$ to a fraction. We do it with Algorithm 17, its execution

5005	0
3099	1
1906	−1
1193	2
713	−3
480	5
233	−8
14	21
9	−344
5	365
4	−709
1	1074
0	−5005

Table 7.2: *Execution tableau of Algorithm 17 for the $\mathbb{M}_\beta$ counterexample.*

5005	0
1671	1
1663	−2
8	3
7	−623
1	626
0	−5005

Table 7.3: *Execution tableau of Algorithm 17 for the $\mathbb{W}_\beta$ example.*

tableau is in Table 7.2. The framed values represent the only possible result. The recovered fraction for $\hat{c}$ is 14/21, which is 2/3 in a reduced form. We need to mix 1/7 in, which we removed earlier. So, we obtain 2/21 as the result, contrary to the correct result $a + b = 8/21$. ↯

The same with $\mathbb{W}_\beta$. The same example with $\mathbb{W}_\beta$ of the same scale results in the following. We map forth with Algorithm 23. We do not extract the factor 1/7 where it does no harm. So, 1/21 is mapped 'as is' modulo 5, 11 and 13: $|21^{-1}|_{[5,11,13]} = [1, 10, 5]$. As 21 has common factors with 7, this residue needs a special treatment: we extract 1/7 from a, remember it as $v_2 = -1$, and map $|1/3|_7$ to $|5|_7$. We obtain the representation $[(1, 0), (5, -1), (10, 0), (5, 0)]$ for a. Similarly, $[(2, 0), (5, 0), (4, 0), (9, 0)]$ stands in $\mathbb{W}_\beta$ for b.

The sum is $[(3, 0), (5, -1), (3, 0), (1, 0)]$ per Definition 7.16, we need to map it back. We follow closely Algorithm 24. The fraction a'/b' is quite obviously 1/7. The distorted values $[\hat{u}_i : i \in \{1, \ldots, 4\}]$ are $[1, 5, 10, 7]$. We consider these as elements of $\mathbb{Z}_\beta$. The corresponding mixed-radix representation is $\langle 1, 5, 3, 4\rangle$, which corresponds to $|1671|_M$. We found these values with an implementation of Algorithm 21, similar to Table 7.1. Now we need to restore the fraction from the value in $\mathbb{Z}_M$. Again, we execute Algorithm 17, the execution tableau is in Table 7.3. The only result below the bound $N = 50$ is 8/3. Finally, we need to multiply 8/3 with 1/7, yielding 8/21. This is the correct result. ✓

7.7 Correctness of $\mathbb{W}_\beta$

In this section we show two statements on the behaviour of the arithmetic in $\mathbb{W}_\beta$ and of the mappings to and from $\mathbb{W}_\beta$.

Theorem 7.18 (Well-definiteness). *The arithmetic operations in $\mathbb{W}_\beta$ produce correct results.*

Proof. We consider the element-wise operations modulo a single m.

1. The addition works, despite looking somewhat strange. Let $(|p/q|_m, v)$ and $(|r/s|_m, v)$ be the summands. The trivial case for summation with zero is clear. The case of $v = v$ is also not endearing. All left is the complicated case $v \neq v$. Without loss of generality, let $v > v$. Now we have three non-trivial sub-cases for different signs of v and v, all other cases can be seen as one of those with places swapped. Let us consider all of them.

   Case '$v > v > 0$'. It holds

   $$\frac{pm^v}{q} + \frac{rm^v}{s} = \frac{psm^v + qrm^v}{qs} = \frac{psm^{v-v} + qr}{qs} m^v.$$

   We can factor out m^v. Now with $|(psm^{v-v} + qr)(qs)^{-1}|_m = |qr(qs)^{-1}|_m = |rs^{-1}|_m$ and the separated factor m^v we obtain exactly $(|r/s|_m, v)$.

   Case '$v > 0, v < 0$'. It is

   $$\frac{pm^v}{q} + \frac{r}{sm^{-v}} = \frac{psm^{v-v} + rq}{qsm^{-v}} = \frac{psm^{v-v} + rq}{qs} m^v.$$

   We separate m^v and look at the remaining part: $|(psm^{v-v} + rq)(qs)^{-1}|_m = |rq(qs)^{-1}|_m = |rs^{-1}|_m$. As we remember the factor m^v, the final result is $(|r/s|_m, v)$.

   Case '$v < v < 0$'. In this case holds

   $$\frac{p}{qm^{-v}} + \frac{r}{sm^{-v}} = \frac{psm^{-v} + rqm^{-v}}{qsm^{-v-v}} = \frac{psm^{-v+v} + rq}{sqm^{-v}} = \frac{psm^{-v+v} + rq}{sq} m^v,$$

   so both numerator and denominator have the common factor m^v. We separate the factor m^v, yielding $|(psm^{-v+v} + rq)(sq)^{-1}|_m = |rq(sq)^{-1}|_m = |rs^{-1}|_m$. Combined with m^v we obtain $(|r/s|_m, v)$.

2. The additive inverse is correct. Changing the sign does not change the factors, thus no change at v. The addition of (u, v) and $(|-u|_m, v)$ returns $(0, v)$, which is zero.

3. The multiplication is straightforward. For the product of $(|p/q|_m, v)$. and $(|r/s|_m, v)$. It follows

   $$\frac{p}{q} m^v \cdot \frac{r}{s} m^v = \frac{pr}{qs} m^{v+v},$$

   which is exactly what we see.

4. The multiplicative inverse is also correct. For some (u, v) and its inverse $(|u^{-1}|_m, -v)$ holds

   $$(u, v) \cdot (|u^{-1}|_m, -v) = (|u \cdot u^{-1}|_m, v - v) = (1, 0).$$ □

Remark. Note that $\mathbb{W}_{\boldsymbol{\beta}}$ does not give any guaranties on the correctness of the result, if the latter does not satisfy the bound (7.2). Theorem 7.18 shows that the representation of the result *in* $\mathbb{W}_{\boldsymbol{\beta}}$ is correct, but this does not mean we can always map this result correctly to $\mathbb{Q}$. This issue is tackled in Theorem 7.11.

Corollary 7.19. *The algorithm 23 is correct.*

Sketch of the proof. Call φ the mapping, defined by Algorithm 23. Let F_N be Farey fractions of order N, let $X \subset \mathbb{Q}$ be the domain of φ with codomain $\mathbb{W}_{\boldsymbol{\beta}}$. It holds $F_N \subset X$, if (7.2) holds for N and $M = \prod \boldsymbol{\beta}$. Theorem 7.18 essentially shows $\varphi(a \circ b) = \varphi(a) \circ \varphi(b)$ for $\circ \in \{+, -, \cdot, /\}$. It is easy to show that zero and unity are preserved. Hence, φ is a homomorphism. †

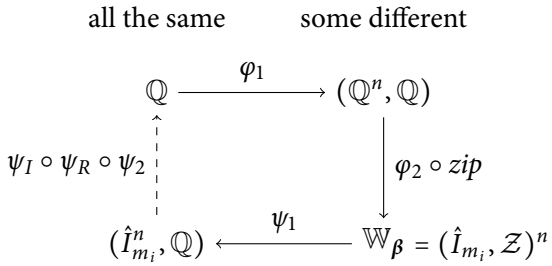


Figure 7.11: *Visualising φ_1 and ψ_1.*

Theorem 7.20. *Let φ be the mapping, defined by Algorithm 23 and ψ be the mapping, defined by Algorithm 24. Let $\boldsymbol{\beta} = [m_1, \ldots, m_n]$ for some $n \in \mathbb{N}$. It holds $M = \prod \boldsymbol{\beta}$ and N is the order of Farey fractions, which can be represented in $\mathbb{W}_{\boldsymbol{\beta}}$, per* (7.2)*. Then $\varphi \circ \psi = \mathrm{id}$ on the set of the Farey fractions F_N.*

Proof. We see φ as $\varphi_1 \circ \varphi_2 \circ zip$ with $\varphi_1 : \mathbb{Q} \to (\mathbb{Q}^n, \mathcal{Z}^n)$, $\varphi_2 : (\mathbb{Q}^n, \mathcal{Z}^n) \to (\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}, \mathcal{Z}^n)$ and zip in this case is $(\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}, \mathcal{Z}^n) \to ((\hat{I}_{m_1}, \mathcal{Z}) \times \cdots \times (\hat{I}_{m_n}, \mathcal{Z})) = \mathbb{W}_{\boldsymbol{\beta}}$ for some $\mathcal{Z} \subset \mathbb{Z}$. These functions correspond to the steps of the Algorithm 23. Recall that $\hat{I}_M \subset \mathbb{Z}_M$ holds. Note that our usual pairs (u_i, v_i) are elements of $(\hat{I}_{m_i}, \mathcal{Z})$ for $i \in \{1, \ldots, n\}$. It is essential for the functionality of $\mathbb{W}_{\boldsymbol{\beta}}$ that the values $\hat{I}_{m_i}$ for $i \in \{1, \ldots, n\}$ represent *different* values in $\mathbb{Q}$. This can be easily seen in corresponding values in $\mathcal{Z}$. It holds that all u_i for $[(u_1, v_1), \ldots, (u_n, v_n)] \in \mathbb{W}_{\boldsymbol{\beta}}$ represent the *same* value in $\mathbb{Q}$ if and only if all v_i for $i \in \{1, \ldots, n\}$ are zero. However, all u_i for $i \in \{1, \ldots, n\}$ always need to represent the same value, if we want to apply the CRT in the backwards mapping. This explains the 'distortion' step in Algorithm 7.7: it is rather the 'correction' step.

Similarly to φ, let $\psi = \psi_1 \circ \psi_I \circ \psi_R \circ \psi_2$ be the mapping, defined by Algorithm 24. It holds $\psi_1 : \mathbb{W}_{\boldsymbol{\beta}} = (\hat{I}_{m_1}, \mathcal{Z}) \times \cdots \times (\hat{I}_{m_n}, \mathcal{Z}) \to (\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}, \mathbb{Q})$. It is more than *unzip*! The function ψ_1 'corrects' the values in $\hat{I}_{m_i}$ for $i \in \{1, \ldots, n\}$ as encoded in corresponding values in $\mathcal{Z}$. The resulting values in $\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}$ would have zeros in all v_i values, if we would construct them. The values in $\mathcal{Z}$ in the input pairs are the powers of m_i for $i \in \{1, \ldots, n\}$. We can convert them to $\mathbb{Q}$. This is the second component of the target set of ψ_1. At this point we have the uniformity back: the values in $\hat{I}_{m_i}$ for all $i \in \{1, \ldots, n\}$ represent the same value. This means, we can use the CRT, i.e., Theorem 7.9. We denote it as $\psi_I : (\mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}, \mathbb{Q}) \to (\mathbb{Z}_M, \mathbb{Q})$. Both ψ_1 and ψ_I are total, they never fail. The function ψ_I does not change the value in $\mathbb{Q}$ of its second component, it merely passes it through for the function ψ_2. A visualisation of the 'uniformity' issue is in Figure 7.11. Note that in our case the input of the CRT is a subset of $\mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$, namely $\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}$.

We need to show that $\psi_I(\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}, \mathbb{Q}) = (\hat{I}_M, \mathbb{Q})$, i.e., that the first component of the result is not in $\mathbb{Z}_M \setminus \hat{I}_M$. Let $a/b \in F_N$ with $a \perp b$ and $a \perp m_i$, $b \perp m_i$ for all $i \in \{1, \ldots, n\}$. We denote with $eea_M(a/b)$ the call of Algorithm 16 with the fraction a/b and integer M. Then we can obtain a value $x := eea_M(a/b)$ in $\hat{I}_M$, as $M = \prod m_i$. Similarly, $[y_1, \ldots, y_n] := [eea_{m_i}(a/b) : i \in \{1, \ldots, n\}]$ in $\hat{I}_{m_1} \times \cdots \times \hat{I}_{m_n}$. We can map from x to $[y_1, \ldots, y_n]$ with the CRT, i.e., with $|\cdot|_{\boldsymbol{\beta}}$. But the CRT is an isomorphism on $\mathbb{Z}_{\boldsymbol{\beta}}$. The inverse mapping is exactly the projection of ψ_I to the first component. The second component is trivial.

The next step is ψ_R, the rational reconstruction with the extended euclidean algorithm. This mapping is partial! It holds $\psi_R : (\hat{I}_M, \mathbb{Q}) \dashrightarrow (\mathbb{Q}, \mathbb{Q})$. Again, the second component is passed through. We will show below that ψ_R is total on the target set of $\varphi \circ \psi_1 \circ \psi_I$. Assuming, ψ_R provides the result, the last component of ψ is ψ_2. It has the type $\psi_2 : (\mathbb{Q}, \mathbb{Q}) \to \mathbb{Q}$ and is essentially $uncurry(*)$. It multiplies the result of the rational reconstruction (the 'first' $\mathbb{Q}$) with the passed through product of powers of m_i for $i \in \{1, \ldots, n\}$ (that is: the 'second' $\mathbb{Q}$). Combined, the type $\psi : \mathbb{W}_{\boldsymbol{\beta}} \dashrightarrow \mathbb{Q}$ emerges.

We need to show that ψ_R does not fail on the image of $\varphi \circ \psi_1 \circ \psi_I$. For $(\varphi \circ \psi_1 \circ \psi_I)(F_N)$ this is quite trivial. All the factor extraction will never increase the order of a Farey fraction. Hence, φ_1 results in n Farey fractions of order N and some powers of factors $v_i \in \mathcal{Z}$ for $i \in \{1, \ldots, n\}$. Each of the Farey fractions is mapped to an integer modulo m_i with φ_2. These values are changed to represent the same

value with ψ_1 and converted to $\hat{I}_M$ with ψ_I. We claim that $(\varphi \circ \psi_1 \circ \psi_I)(a/b) = eea_M(a/b)$ for $a \perp M$, $b \perp M$, $a \perp b$ and $a/b \in F_N$. Now, $eea_M(a/b)$ results in $|a/b|_M \in \hat{I}_M$.

Let $\hat{F}_N^{\boldsymbol{\beta}} := \{a/b \in F_N : a \perp m, b \perp m, a \perp b \text{ for all } m \in \boldsymbol{\beta}\}$. We consider $(\varphi \circ \psi_1 \circ \psi_I)(a/b)$ for $a/b \in \hat{F}_N^{\boldsymbol{\beta}}$ in a more detail. The mapping φ_1, applied to a/b results in $([a/b, \ldots, a/b], [0, \ldots, 0])$. Then, $\varphi_2(\varphi_1(a/b)) = ([eea_{m_1}(a/b), \ldots, eea_{m_n}(a/b)], [0, \ldots, 0]) =: (eea_{\boldsymbol{\beta}}(a/b), \mathbf{0_n})$. Now, ψ_1 has no factors to extract and correct. If ψ_I maps $(eea_{\boldsymbol{\beta}}(a/b), 1)$ to $(eea_M(a/b), 1)$ for some $a/b \in \hat{F}_N^{\boldsymbol{\beta}}$, we are on the safe side. In other words, the first $\hat{I}_M$ component of the domain of ψ_R contains the mapped-to-integer value of some $a/b \in \hat{F}_N^{\boldsymbol{\beta}}$. Per Theorem 7.7 ψ_R is total in this case. If the input fraction $a/b \in F_N$ of φ_1 has some common factors with m_i, then these are extracted and φ results in mappings of different fractions modulo $\boldsymbol{\beta}$. The function ψ_1 makes then the representations of (different) values $a^{(i)}/b^{(i)}$ from equation (7.4 RIGHT) to $a^{(n)}/b^{(n)}$ with $a/b = a^{(n)}/b^{(n)} \prod m_i^{v_i}$. Thus, the first component of the argument of ψ_I contains n representations of the same value, namely $a^{(n)}/b^{(n)}$. The second component is the fraction $\prod m_i^{v_i}$. It will be used by ψ_2 to restore original value a/b.

In fact, ψ_R also does not fail on all products of the Farey fractions of order N with powers v_i of m_i, for $v_i \in \mathcal{Z}$ and $i \in \{1, \ldots, n\}$. In this case, just as in the above case with common factors, the aforementioned powers would be extracted in φ_1 and saved in the $\mathcal{Z}^n$ component of $\mathbb{W}_{\boldsymbol{\beta}}$. The function ψ_1 would convert all v_i for $i \in \{1, \ldots, n\}$ to the 'second' $\mathbb{Q}$ in the pair, which is not considered by ψ_R. The value in $\hat{I}_M$, the function ψ_R would operate on, is the encoded Farey fraction from $\hat{F}_N^{\boldsymbol{\beta}}$. We have already shown that ψ_R is total on such inputs. □

7.8 Parallelism

Multiple-residue arithmetic approaches are known for their data parallelism potential. We compute with different residues in a fully independent manner, without a need for the communication in-between. As our implementation of rational multiple-residue arithmetic conforms to this principle, we can immediately make a step from a (sequential) Haskell implementation to the (parallel) Eden code. Suppose, we have some function `f :: FMods Int → FMods Int`. This function could be implemented in Haskell as `f = map g`, where `g :: FSingleMod Int → FSingleMod Int`. It suffices to write `f = farm g` to obtain a parallel Eden implementation.

A further advantage is provided by Haskell type system. As both `FMods` and `FSingleMod` are instances of the standard `Num` and `Fractional` type classes, we could use the standard arithmetical notation of $+, -, \cdot, /$ in the implementation of the function `g` from above. Even more: the generalised type of `g` is `g :: (Num a, Fractional a) ⇒ a → a`. This means, that we can use `g` for *any* arithmetic of our choice: be it the standard one, or the one presented above. In terms of computer algebra, one says that `g` is symbolic.

In order to have a large enough task, we use matrix computations for testing the arithmetic. We choose the **LU** decomposition of matrices as our test problem. It is also known as Gauß elimination. We will discuss it next. The sequential implementation is shown in Figure 7.13 on page 165. To obtain a parallel implementation, we use the higher-order function `lift1`, described below in Section 7.8.3. The actual parallel implementation of the Gauß elimination and a subsequent determinant computation is presented in Figure 7.14 on page 166. We present the Gauß elimination first and then handle the determinant computation.

7.8.1 Gauß Elimination

A quite common problem in linear algebra is the solution of the systems of linear equations. From input of the form

$$\mathbf{Ax} = \mathbf{b}$$

we need to obtain the vector $\mathbf{x}$. A naive way to do it is to invert the matrix $\mathbf{A}$. The more specific, robust and classical approach is not a direct inversion, but a method called *Gauß elimination*. Figure 7.12

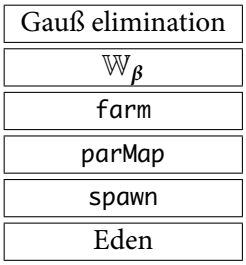


Figure 7.12: *Implementation hierarchy of the residue arithmetic example.*

visualises the relation of Gauß elimination, $\mathbb{W}_\beta$ and Eden skeletons. Our parallel implementation of the Gauß eliminationdepends on $\mathbb{W}_\beta$, which depends on the `farm` skeleton.

The theory. This method lies in successive multiplication of the input matrix **A** by so called *transformation matrices* $\mathbf{Y_i}$, each time eliminating all the entries in the matrix column below the diagonal, hence the name: Gauß elimination. This method is also called **LU** factorisation. The elimination is done for each column of the matrix, resulting an *upper triagonal* matrix **U** as the output. The product of all matrices $\mathbf{Y_i}$ results in a *lower triagonal* matrix **L**. It holds

$$\mathbf{A} = \mathbf{LU}.$$

Now with simple techniques like backward substitution we can obtain the solution of the initial equation $\mathbf{Ax} = \mathbf{b}$. Our sources on Gauß elimination are [Golub and Van Loan, 1996, Sauer, 2002, Cheney and Kincaid, 2007, Quarteroni et al., 2007]. However, while the *method* and *presentation* are the same, our *goals* and *arithmetic* differ. Golub and Van Loan present a classic work in numerical linear algebra. They compute in floating point arithmetic, which is inexact. Our approach is to use a residue class arithmetic. It is exact—up to a certain bound. (The bound is not a problem, if we can estimate it *a priori* and scale the arithmetic accordingly.) Hence, the theory of numerical computing, like error analysis for floating point arithmetic, is not applicable here. Even the complexity analysis is different and more complicated, as we infer from [von zur Gathen and Gerhard, 2003]. The Gauß elimination is cubic in its elementary operations. Unfortunately, it is not given, that the complexity of an elementary operation is constant in time.

The implementation idea of Gauß elimination is simple: we subtract multiples of one matrix column from another one to obtain zeroes beneath the matrix diagonal. We use determinant computation via Gauß elimination as a test for our arithmetic. However, contrary to common approaches, we perform the *exact* computation. The input matrix is filled with fractions, the determinant is also fractional.

The implementation. We implement Gauß elimination in pure Haskell, without any parallel constructs. It is implemented symbolically, using Haskell type classes, as

```
gauss :: (Num a, Fractional a) ⇒ MatArr Int a → MatArr Int a
```

Recapitulate from Chapter 6 that `MatArr i x` is the array-based matrix representation with indices of type `i` and matrix elements of type `x`. With such implementation, it is *completely irrelevant* over *what* the matrix operates, as long as we can perform divisions in `x` plausibly. We show the full source code of the sequential Gauß elimination in Figure 7.13.

We need to use some kind of a residue arithmetic anyway. The problem of the symbolical Gauß elimination is the *intermediate expression swell* [von zur Gathen and Gerhard, 2003]. Imagine, we transform a matrix **A**. We are in i^{th} step and process currently the j^{th} row. So, we divide a_{ij} by a_{ii}. Let the former be x/y and the latter z/t. Then the new value at a_{ij} is $a'_{ij} = \frac{xt}{yz}$. The Farey measure (*cf.* Definition 7.2) of the new value can be larger, than the sum of the measures for both input elements.

The actual growth of the intermediate expressions is polynomial [Edmonds, 1967, Bareiss, 1968], but still quite large, see, e.g., [Lobachev, 2011a] for some numeric examples.

The parallelism arises through the usage of multiple-residue arithmetic. Gauß elimination as such is quite indifferent to whether we use *one* residue class, or *multiple*. The only change is in the complexity. Von zur Gathen and Gerhard [2003] state the complexity of the determinant computation for $n \times n$ integer matrix with largest entry $\leq B$ using correspondingly prescaled single-residue class as $\mathcal{O}(n^3 \cdot n^2(\log n + \log B)^2)$. With the fast integer multiplication (see Chapter 6) this bound can be relaxed to power of four with some logarithmic factors. These results differ from our customary $\mathcal{O}(n^3)$ for the constant-time arithmetic! The authors of *Modern Computer Algebra* imply here the case, where a single operation in the arithmetic is more complex than that—as the 'big' residue class' implementation has to deal with arbitrary precision integers. The complexity is also different for multiple larger primes, see [von zur Gathen and Gerhard, 2003]. However, for p 'small' primes, fitting in the hardware integers, the single operation complexity is *constant*, because the hardware integers are bounded by 2^{64} for 64 bit hardware. With this approach the complexity is reduced to $\mathcal{O}(pn^3)$.

Summarising, p residue classes need to work simultaneously. They do not depend on each other, until the computation is finished. So, we can just use a `parMap` to obtain a parallel version of the same computation.

7.8.2 Determinant Computation

Definition. Aside from solving linear equation systems, the most prominent use of Gauß elimination is the computation of the determinant. Such computation can be seen as a mapping $\det : R^{n\times n} \to R$, for some commutative ring with unity R. We call the result of applying det to some matrix $\mathbf{A}$ the *determinant* of $\mathbf{A}$, in sign $\det \mathbf{A}$. The determinant can be defined basing on its properties, see e.g., [Lang, 1987, Cullen, 1990]. We embark the other way and present a recursive definition. It is called Laplace's determinant expansion by minors, after Pierre-Simon de Laplace, *23.3.1749, †5.3.1827. The i^{th} *minor* of a square matrix $\mathbf{A}$ is a submatrix, consisting of all rows of $\mathbf{A}$ except the first one and all columns of $\mathbf{A}$ except the i^{th}. To be more precise, for the $n \times n$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1i} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2i} & \dots & a_{2n} \\ \vdots & & \ddots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{ni} & \dots & a_{nn} \end{pmatrix},$$

the i^{th} minor, in sign: $\mathbf{A}^{(\mathbf{i})}$, is

$$\mathbf{A}^{(\mathbf{i})} = \begin{pmatrix} a_{21} & a_{22} & \dots & a_{2,i-1} & a_{2,i+1} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{n,i-1} & a_{n,i+1} & \dots & a_{nn} \end{pmatrix}.$$

Now, we can write

$$\det \mathbf{A} = \sum_{i=1}^{n} (-1)^{i-1} \det \mathbf{A}^{(\mathbf{i})}.$$

With the determinant of 1×1 matrix being its sole element, we have obtained the recursive definition of the determinant.

Special names. A square matrix with integer elements is called *unimodular* if its determinant is ± 1. An example with especially small entries would be

$$\begin{pmatrix} 2 & 3 & 2 \\ 4 & 2 & 3 \\ 9 & 7 & 7 \end{pmatrix},$$

see [Guy, 2004]. Any square matrix is called *singular* if its determinant is zero and non-singular otherwise.

```
-- matrix!(j, k) selects a single element of the matrix

type Bound = ((Int, Int), (Int, Int))
gaussWorker :: (Num x, Fractional x)
            ⇒ (Int, Bound, MatArr Int x)
            → (Int, Bound, MatArr Int x)
gaussWorker (i, bound, matrix)
  = let ((ln, lm), (n, m)) = bound
        zs = [ ((j, k), makeEl i j k) | j ∈ [i+1..n], k ∈ [i..n] ]
        makeY (i, j) = matrix!(j, i) / matrix!(i, i)
        makeEl i j k = matrix!(j, k) - makeY(i, j) * matrix!(i, k)
    in (i+1, bound, matrix//zs)

gauss :: (Num x, Fractional x) ⇒ MatArr Int x → MatArr Int x
gauss matrix
  = let b@((ln, lm), (n, m)) = bounds matrix
        (_, _, result) = last $ take (n-ln+1)
          $ iterate gaussWorker (ln, b, matrix)
    in result
```

Figure 7.13: *Gauß elimination in Haskell.*

Computing the determinant. Because of complexity issues, the determinants of larger matrices are *computed* with a different method, namely with Gauß elimination. It produces two triagonal matrices **L** and **U**, we can require the diagonal of **L** to consist of unities. It suffices to show two facts. Firstly, the determinant of a product of the two compatible matrices is the product of their determinants. Secondly, the determinant of a triagonal matrix is the product of its diagonal elements. We do not go into detail here, but refer to [Lang, 1987, Cullen, 1990]. The first is connected to the multilinearity of the determinant, one of its basic properties. We show a special version of the second fact below.

Lemma 7.21. *The determinant of an upper triagonal matrix* **U** *is the product of its diagonal elements.*

Proof. The Laplace expansion produces the product in question already in the first term. All other terms in the sum are products with determinants of singular matrices. □

Thus, it suffices to compute the Gauß elimination and to multiply the diagonal elements of the **U** matrix, to compute the determinant of the initial input. The sequential determinant computation can be written as

```
det :: (Num x, Fractional x) ⇒ MatArr Int x → x
det = product ∘ diag ∘ gauss
```

with a trivial implementation of `diag :: (Ix a, Num b) ⇒ MatArr a b → [b]`. Our implementation of `gauss` is depicted in Figure 7.13, it is really straightforward. The function `gauss` has the type `(Num x, Fractional x) ⇒ MatArr Int x → MatArr Int x`. In the definition of `gaussWorker`, the binding `zs` is defined with a list comprehension. Basically, it can be read as a corresponding mathematical set notation: 'the list of pairs of a pair `(j,k)` and `makeEl i j k` for all $i \in \{i+1,\ldots,n\}$ and $k \in \{i,\ldots,n\}$'. The library function `iterate` of type `(a → a) → a → [a]` produces lazily an infinite list of applications of its parameter function to the previous result. In other words, `iterate f x` results in `[x, f x, f (f x), f (f (f x)),...]`. We do not use pivoting in our implementation of Gauß elimination.

Residue arithmetic and intermediate expression swell. The residue arithmetic fits our needs. If we can supply an *a priori* bound on our final result, we can scale the residue size to be larger then the

```
gaussResidue :: (Integral a, Integral b)
             ⇒ Map a b                          -- ^ a map implementation
             → [a]                              -- ^ list of primes
             → MatArr Int (Fraction a)          -- ^ input
             → MatArr Int (Maybe (Fraction a)) -- ^ output
gaussResidue myMap = lift1 myMap gauss
detResidue = product ∘ diag ∘ gaussResidue      -- simplified

gaussResidueDiag :: (Integral a, Integral b)
                 ⇒ Map a b                      -- ^ a map implementation
                 → [a]                          -- ^ list of primes
                 → MatArr Int (Fraction a)      -- ^ input
                 → [Maybe (Fraction a)]         -- ^ diagonal of output
gaussResidueDiag myMap = lift1' myMap gauss
-- this is a more complex implementation with even more lift1 magic

-- parallel implementation, two parallel invocations
detParMap, detFarm :: (Trans a, Trans b, Integral a, Integral b)
                   ⇒ [a]                        -- ^ list of primes
                   → MatArr Int (Fraction a)    -- ^ input
                   → [Maybe (Fraction a)]       -- ^ diagonal of output
-- the instantiation is simple
detParMap = detResidueDiag parMap
detFarm = detResidueDiag farm
```

Figure 7.14: *Residue-based invocation of parallel Gauß elimination. Implementation of* `detResidue` *is slightly simplified.*

bound. Thus we guarantee our result to be exact. In many cases we can indeed supply such a bound. An example is the determinant computation of a matrix. Looking at determinant computation via Gauß elimination, we observe that the intermediate results grow faster, than the bound on the final result does. Such growth is precisely the intermediate expression swell. A bound is provided by the Hadamard inequality [Hadamard, 1893, Brenner and Cummings, 1972] for the matrices in $\mathbb{R}^{n\times n}$ and $\mathbb{Z}^{n\times n}$. So, we can scale the arithmetic to compute the *final* result in an exact manner and do not care that the intermediate results, which cannot be represented exactly.

In special cases, *almost all* of the intermediate results might be in the overflow zone. For instance, if we test whether a matrix is *not* unimodular, we compute the determinant modulo 2. All unimodular matrices have determinant of 1. But of course not every integer matrix with odd determinant is unimodular! So, if the result is 0 mod 2, then the matrix is for sure *not* unimodular. If it is 1 mod 2, we don't know. So, we have obtained a correct result in a not unimodular case, without computing exact (i.e., possibly large) values for any of the intermediate results absolutely larger than 1.

As for matrices over $\mathbb{Q}$, we use here some special matrices with known values of the determinant.

7.8.3 Technical Details of Implementation

In order to be able to distribute the data parallel tasks across the PEs, we need to rotate the hypercube, being matrix over a list of the fractional residue classes—we need a list of matrices. Another issue, arising in the actual implementation was connected with arrays. They are not very suitable for data transmission in the current Eden implementation. Thus, we needed to convert all matrices to a special transport type `TransMat`, being essentially a stream. Future Eden implementations might enable this by allowing special, user-crafted instances of `Trans` type class. Then we would specify the '`send`' and '`receive`' operations there and the transformations of a matrix to a stream and vice versa would be hidden from the application programmer. Currently we base our work on a more stable specification

```
type TransMat i n = ((i, i), [n])
type SparseDiagMat i n = ((i, i), [((i, i), n)])

toL :: (Ix i, Num n) ⇒ MatArr i n → TransMat i n
fromL :: (Ix i, Num i, Num n) ⇒ TransMat i n → MatArr i n

toSD :: (Ix i, Num n) ⇒ MatArr i n → SparseDiagMat i n
fromSD :: (Ix i, Num i, Num n) ⇒ SparseDiagMat i n → MatArr i (Maybe n)

liftL :: (Ix i, Num i, Num n)
      ⇒ (MatArr i n → MatArr i n ) → TransMat i n → TransMat i n
liftL f = toL ∘ f ∘ fromL

liftLS :: (Ix i, Num i, Num n)
       ⇒ (MatArr i n → MatArr i n ) → TransMat i n → SparseDiagMat i n
liftLS = toSD ∘ f ∘ fromL

toResiduePrimes :: (Ix i, Integral n)
                ⇒ [n] → MatArr i (Fraction n) → [MatArr i (FSingleMod n)]
fromResidueMaybe :: (Ix i, Integral n)
                 ⇒ [MatArr i (FSingleMod n)] → MatArr i (Maybe (Fraction n))

-- a map implementation | working function | primes | input value
-- lift1 :: Map a b → (c → d) → [n] → m1 → m2
lift1 mymap f = fromResidueMaybe ∘ map fromL ∘ mymap (liftL f)
              ∘ map toL ∘ toResiduePrimes
```

Figure 7.15: *The function* `lift1` *and supporting code signatures. We omit technical details here.*

of `Trans` (*cf.* Figure 3.4 on page 24), which firstly, makes the definition of `receive` quite cumbersome, and secondly effectively disallows definitions of user `Trans` instances, i.e., of `Trans` instances outside Eden module. Hence, we stick with the older approach here. These are the main reasons to designate the function `lift1` and its derivatives. As we would like to specify different parallel `map` implementations as a parameter, the same higher-order function `lift1` is used also for this purpose. A quite performance boost resulted from the two following design decisions.

- *Do not send unneeded data.* We have implemented a special transmission data type for diagonal matrices. We take the diagonal of the transformed matrix *before* the communication takes place, thus majorly reducing communication bottleneck in the direction 'workers to master'.

- *List chunking.* This issue is very technical. Supporting Eden library defines list communication in form of streams, where each list element is sent separately. This is inefficient for large lists. A common solution to this problem is list chunking—we reduce a list to a nested list for the sake of communication, thus we send multiple list elements in a single message. Our experiments have shown that chunking size of 100 elements provides best results in this particular case.

We have implemented the higher-order function `lift1'`, an improved version of `lift1` with these requirements in mind. We show an abridged implementation of `lift1` in Figure 7.15. The full implementation of `lift1'` is is the Appendix, see Section B.4. The special transport type `TransMat` serves the transport issue of arrays, discussed above. The type `SparseDiagMat` is a special type for transmitting only a diagonal of a matrix as a sparse list. The function `lift1'` uses the latter type. The parallelisation of $\mathbb{Z}_\beta$ is similar. It is just as data parallel as $\mathbb{W}_\beta$ is.

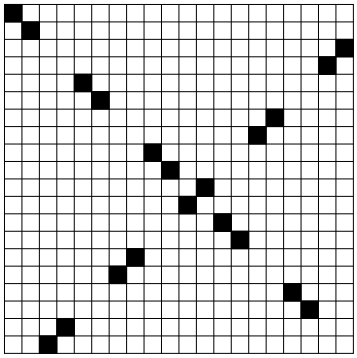


Figure 7.16: *Plot of the* 20 × 20 *permutation matrix.*

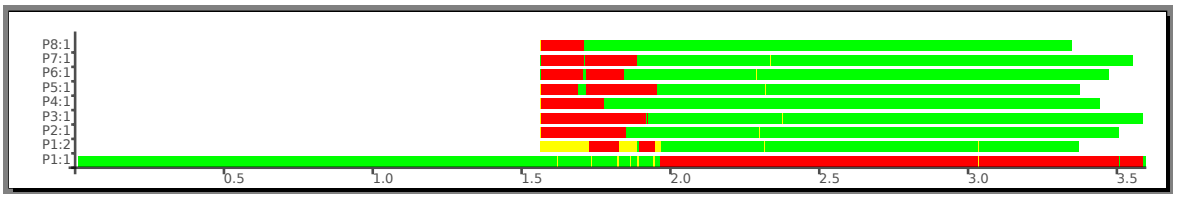


Figure 7.17: *The trace diagram for Gauß elimination.*

7.8.4 Test Results

An n^{th} Pascal matrix **P** of dimension $n \times n$ is defined as a $p_{ij} = \binom{i+j}{i}$ for $1 \leq i, j \leq n$. We have implemented distributed determinant computation of permuted, scaled with ⅓ Pascal matrices, using the above approach. The permutation matrix is shown in Figure 7.16 for $n = 20$. A black pixel represents a unity, a white pixel represents zero. The determinant of a permutation matrix is ±1, a Pascal matrix is unimodular. Thus the final result of the computation is always known: $\pm 1/3^n$. The arithmetic of a right scale always performed correctly in our tests. The visualisation of the parallel program execution trace is depicted in Figure 7.17. The program was run on `sakania`, it was compiled with GHC 6.12. The input was the 100 × 100 permuted scaled Pascal matrix, as discussed above. We used an optimised `lift1'` implementation, sending back only a diagonal. Eight primes of size 10^4 were used. The chunking size was 100, i.e., the size of a matrix side. The first 1.5 seconds of the program execution are taken for the generation of the input matrix. We do not consider this time in our speedup measurements, as we assume the matrix as given in our tests. We see that processes are created at the same time, but the processes begin *working* at slightly different times: 1.6–2.0 seconds. The reason is the communication overhead. We need to communicate ten thousand matrix elements to each of the workers. Even in chunks of 100, this counts up to 800 messages to be sent from the master. Thanks to the residue arithmetic, each process has to perform roughly the same amount of work. But because of some inconsistency with startup time, the workers have the same inconsistency with termination time. This unsteadiness is seen at 3.3–3.5 seconds. Overall we see a typical data parallel computing pattern. The data communication is quite a bottleneck, especially the input part. However, when having all the data, the processes work steadily and independently, until done.

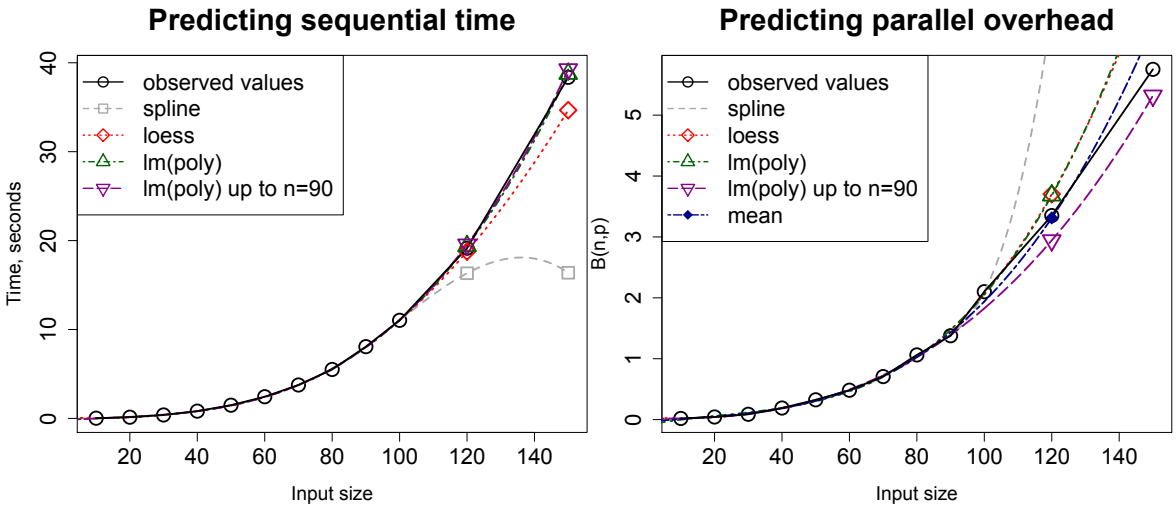
7.8.5 Performance Estimation

Basing on our approach from Chapter 4, we perform an estimation of the execution times of our parallel residual implementation of Gauß elimination. This is similar to what we have shown in [Lobachev and Loogen, 2010c] for an earlier version of the implementation. One of the major differences between the following and said work is that current implementation uses a new 64 bit Eden compiler.

We measure the time needed to compute the **LU** decomposition of a permuted scaled $n \times n$ Pascal matrix modulo r primes. The program has been parallelised using the simple `farm` skeleton with the

n	10	20	30	40	50	60	70	80	90	100	**120**	**150**
$T(n)$	0.02	0.13	0.39	0.82	1.48	2.44	3.76	5.50	8.07	11.03	**19.14**	**38.36**
$T(n,8)$	0.02	0.06	0.14	0.29	0.51	0.79	1.18	1.75	2.39	*3.48*	**5.74**	**10.55**

Table 7.4: *Execution times for Gauß elimination of $n \times n$ permuted scaled Pascal matrices on* `sakania`*. The* ***bold*** *values will be estimated w.r.t. input size n, the* *framed* *value will be estimated w.r.t. number of PE p.*



Rel. error for $n = 120$

Method	Rel. err., %
spline	−14.71
loess	−1.90
lm(poly)	1.28
lm(poly) w. $n \leq 90$	2.13

Rel. error for $n = 120$

Method	Rel. err., %
spline	99.38
loess	10.52
lm(poly)	9.88
mean	−1.13
lm(poly) w. $n \leq 90$	−12.14

Figure 7.18: *Gauß elimination. Predicting sequential run time (top left) and parallel penalty values w.r.t. n (top right) for $n = 120, 150$. The bottom parts show the corresponding relative errors.*

input list of size r, as the map is done over the different residue classes [Lobachev and Loogen, 2010a,b]. This means $\#\boldsymbol{\beta} = r$.

In our setting, r corresponds to the total number of PEs, here: $r = 8$. Note that this is not the optimal way to parallelise this program for $p < 8$. When we have more processes than PEs, multiple processes will be executed by the same PE. This causes an imbalance when the processes cannot be evenly distributed to PEs, i.e., when 8 is not a multiple of the number of PEs. Thus, the 2, 4 and 8 PE configurations should perform best. In this and next section we will see more details on this, including how to obtain the same information with our approach.

Estimating the execution time w.r.t. n. Table 7.4 shows the measured times. The values are stated in seconds, rounded up to two digits after decimal dot. We have measured the execution times on eight PE and estimate the parallel time also on eight PE, but for larger input sizes. The data points for $n = 120$ and 150 are not known to the estimation routines. Figure 7.18, left, shows the estimation of the sequential runtime $T(n)$. The same figure, right, shows the estimation of $B(n,p)$ w.r.t. n. With mean we denote the mean of lm(poly) and of lm(poly), restricted to $n \leq 90$. The reason for this decision is a

PEs, p	1	2	3	4	5	6	7	8
Full rounds	8	4	2	2	1	1	1	1
Remaining tasks	0	0	2	0	3	2	1	0
Total rounds	8	4	3	2	2	2	2	1
Unused PEs	0	0	1	0	2	4	6	0
Slack-off, %	0	0	33.3	0	40	66.6	85.7	0

Table 7.5: *Task distribution in the discussed implementation of Gauß elimination.*

small increase of $B(n,p)$ at $n = 100$, which misleads multiple methods. We also used the restricted version of lm(poly) for $T(n)$, but with no reassuring result.

Now, as can be seen in the figures, we have the best method for estimating B—mean—and the best method for estimating $T(n)$—lm(poly). Note that we can disregard spline in both cases for its very poor performance. Combined, we can apply equation (4.1) and obtain the complete time estimation. We obtain an estimate $T(120,8) = 5.736$ seconds, which corresponds to the appropriate value 5.743 up to the relative error −0.125%.

Estimating the execution time w.r.t. p. In the previous example we have assumed, there was the possibility to measure time on a eight PE machine, but no one had run the test program for the input size 120 or 150. Now we do the converse: assume, we have measurements for task size 100 on smaller PE numbers, but do not have a machine with seven PEs to measure time there. We choose 7, not 8 PE because of the task distribution issues in our program. We have 8 tasks, which are distributed evenly to PEs. For $p = 8$ this special case is not connected with 6 and 7 PE configurations. We *could* use only the special cases, but then we would not have enough data points for most of our methods, as we perform our measurements on an 8 PE machine. Still, see 'lm w. special' estimation in Figure 7.19 for this approach.

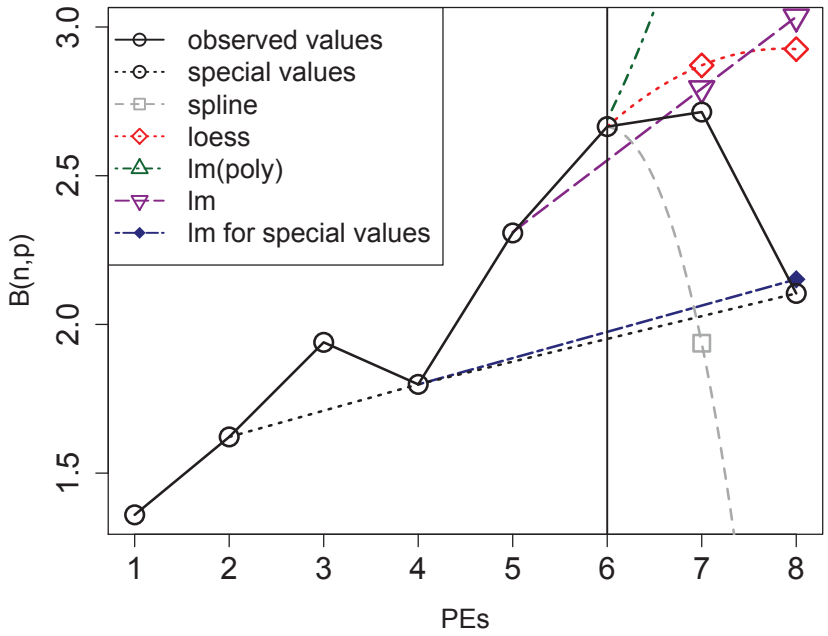
All other approaches target $p = 7$. In Figure 7.19 we see, that it is quite hard to predict $B(n,p)$ w. r.t. p correctly. To enable a better insight, we have separated the available values ($p \leq 6$) from values-to-estimate ($p = 7, 8$) with a straight line. Somewhat bearable results were produced by loess. The best method was lm. We used it with all the values from 1 to 6 as 'lm w. all'. So we use these data from the estimation of $B(n,p)$ w.r.t. p and the already measured $T(100)$ to predict the parallel execution time. We obtain an estimate $\hat{T}(100,7) = 4.369$ seconds, which is 1.838% accurate. The measured value is 4.290. Now, using lm for $\hat{B}(100,8)$, based only on data for $p = 2, 4$ ('lm w. special'), we obtain $\hat{T}(100,8) = 3.531$. This result shows 1.361% relative error, compared to the measured value.

We see, that it is possible to estimate the parallel runtime both w.r.t. task size and w.r.t. the number of PEs with significant accuracy.

7.8.6 Parallelisation Quality

The plot of $B(n,p)$ w.r.t. p in Figure 7.19 reveals some information on the performance quality of our implementation. We demand eight residue classes, regardless the number of PEs. Thus, some task imbalance ensues. To see it more clearly, we plot $B(n,p)$ w.r.t. p alongside with serial fraction $f(n,p)$, w.r.t. p in Figure 7.20. The corresponding speedup graph (*not* the best speedup!) is shown in Figure 7.21 The latter plot displays speedup for $n = 100$, while best measured speedup of 3.67 was for $n = 150$ on 8 PEs.

Let us discuss the both quality measures in Figure 7.20. The corresponding speedup plot is in Figure 7.21. We see that both the parallel penalty w.r.t. number of processors (in the top part of the figure) and the serial fraction (bottom part) decrease at four and eight PE. Both approaches agree at 5–7 PE: these configurations are regarded as quite bad. A quick glimpse on the speedup in Figure 7.21 confirms this. However, parallel overhead graph deems seven PE version as worst, while serial fraction shows the 6 PE version as such. Also, the value of the serial fraction is large at two PE, but this is a



Relative error

Method	spline	loess	lm(poly)	lm w. all	lm w. special
PEs, p	7	7	7	7	8
Rel. err., %	−28.64	5.78	30.24	2.90	2.25

Figure 7.19: *Gauß elimination. Estimation of penalty values w.r.t. p. We fix $n = 100$ and predict the values for $p = 7, 8$ using the values for $p \leq 6$.*

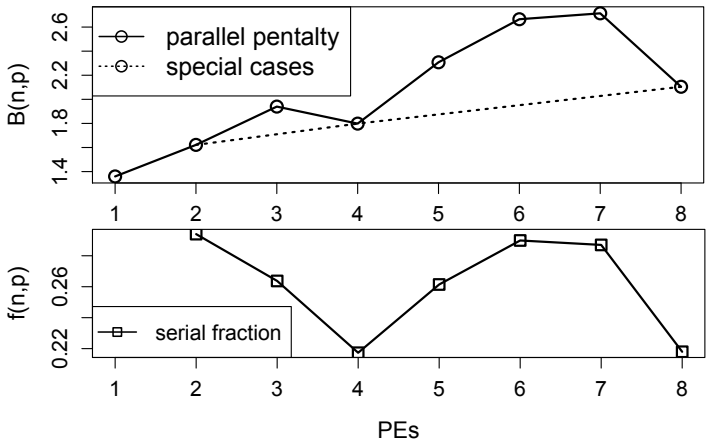


Figure 7.20: *Parallel penalty (top) and serial fraction (bottom) for Gauß elimination.*

borderline effect, as serial fraction is not defined for a single PE. Continuing with the serial fraction: it decreases between 2 and 4 PE, which would be a sign for a bad sequential version, but the function increases again between 4 and 6 PE. This means that serial fraction designates the four PE version as preferable for a particular reason. We see the same for eight PE. As for the parallel penalty, it shows the same issues for 4 and 8 PE, but in the range between one and four PE the shape of this function is very different from serial fraction. The parallel overhead increases slightly in said interval, but it decreases again at four PE. Our interpretation: 2 and 4 PE versions are better than the 3 PE version. From 4 to 8 PE, the shape of parallel penalty function is similar to the serial fraction, with the exception of the maximum, which is here at seven PE. Note, however, that the values for 2, 4 and 8 PE are almost on a straight line. This can also be confirmed by 'lm w. special' estimation in Figure 7.19.

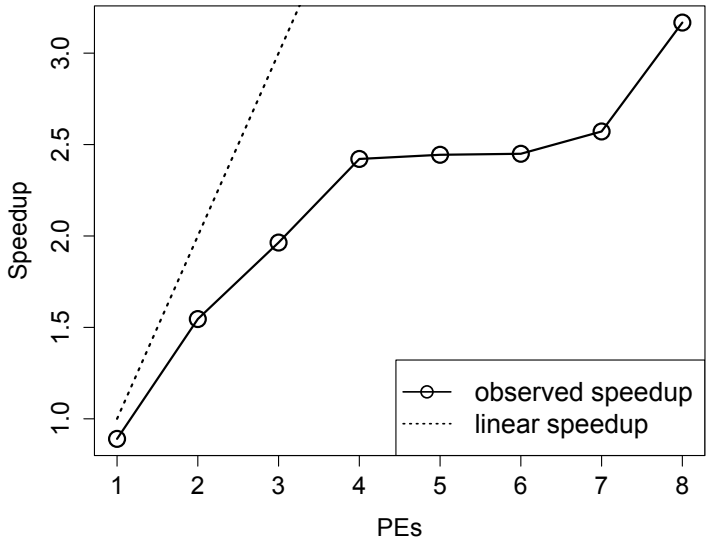


Figure 7.21: *A corresponding speedup plot for Figure 7.20.*

Some reasons for the shape of the curves in Figure 7.20 are uncovered in Table 7.5. As we already did before, we compute the task distribution across the PEs. Important for us here is the last row of the table. The 'slack-off' is the coefficient of unused PEs in the last round and total number of PEs. Note that this table is similar to Table 6.7, as we have here 8 equal tasks on the same number of PE. We see that for 1, 2, 4 and 8 PE the task balancing is perfect. This was the reason in separating them, with the exception of the single PE case, as 'special cases'. As Figure 7.19 reveals, these data points are too few for a non-linear estimation, but provide a good result (2.25% relative error) when using lm estimation method. Further, 'slack-offs' for 6 and 7 PE are especially high, with 7 PE having the maximal 'slack-off'. This corresponds to our interpretation of Figure 7.20. Note that only the parallel penalty correctly identified the configuration with maximal 'slack-off'. The serial fraction pointed out the wrong configuration. For $p > 2$ the configurations with minimal 'slack-off' values were clearly visible with both performance measures. The fact, that two PE version also has zero 'slack-off', was not visible, when using serial fraction, because of the construction properties of the latter. Additionally, another reason for bad parallel performance is the communication overhead from master to workers. It is quite high: we need to communicate 10^4 matrix elements to each worker.

We conclude that in this case our 'parallel penalty' approach was more exact and informative than the serial fraction.

7.9 Related Work

7.9.1 Comparison with Maple

We compare our parallel implementation from above with the reference software: Maple 13 [Redfern, 1995, Monagan et al., 2005]. We neither implemented a *rational* single- or multiple-residue arithmetic in Maple, nor are we aware of a such implementation. Instead, we compute the determinant of the same matrix using exact rational arithmetic and the determinant of the unscaled[2] integer matrix using small single-residue arithmetic, available in Maple. This is actually very helpful for Maple, as the integer case is much 'easier'. We will return to the computational difference between the determinant of matrices over integers and over fractions in the next section. Like in the Eden case, the time measurement is incorporated in the software itself, we do not measure the time to generate the matrix. We run all applications on the same hardware, namely `sakania`.

[2]The reason to scale the input matrix with ⅓ was to obtain a matrix over fractions with sustainably large known determinant. Lacking a rational residual implementation in Maple, we use an integer matrix for our tests. Still, up to the scaling it is the same matrix.

n	10	50	70	100
our approach, $\mathbb{W}_\beta$	0.02	0.51	1.18	3.48
Maple, $\mathbb{Q}$	0.002	0.175	0.60	2.0

Table 7.6: *Comparison of our approach with Maple using permuted Pascal matrices. The computation in $\mathbb{W}_\beta$ uses scaled matrices for the input to be rational. Time is in seconds.*

	Maple method				*Speedup*		
	n	default	$\mathbb{Q}$	$\mathbb{Z}_m$	n	$\bar{s}$	$\hat{s}$
Integers	100	0.07	4.9	0.012	100	5.8	408
	300	2.2	494	0.018	300	122	27 378
	1000	325	–	0.75	1000	433	–

Table 7.7: *Determinants of integer random sparse matrices. Comparing Maple methods and stating upper bound on the speedups. Time is in seconds.*

The results are depicted in Table 7.6. We see that our Eden implementation is less than a factor of two slower than optimised Maple implementation with standard fractions. Still, we would like to emphasise further qualities of our implementation. We implemented **LU** factorisation in a very straightforward manner. The high-level features of Haskell allowed us to write down an implementation, which is quite similar to the algorithmic definition of the **LU** factorisation. It was quite efficient. Thus we see, that high-level programming is a reasonable engineering technique. It is still possible to tune and to improve our implementation. In a contrast, Maple implementation is highly optimised. The linear algebra package in Maple was recently completely re-engineered. It is not possible to us to 'look inside' the closed-sourced implementation. Despite all the community efforts, e.g., [Siegl, 1993, Martínez and Peña, 2004, SCIEnce, 2010], it is very hard to utilise many cores for such tasks in Maple. The Maple implementation, available to us, used only one core for the determinant computation. Note that the time for the *single* residue-based computation in Maple is very low. Hence, solely the ability to compute the same result with a residue arithmetic would be a major advantage for Maple. This issue, combined with a possibility to execute the arithmetic in parallel on many cores, would produce very high speedups, compared with the traditional implementations of the rational arithmetic. The reason for such improvement is reduced computational complexity, as described in [von zur Gathen and Gerhard, 2003]. We consider this issue next.

7.9.2 Speedups

For the current Eden implementation we obtain absolute speedup of 3.67 on `sakania`, using eight cores and $n = 150$. The previous, 32 bit version, described in [Lobachev and Loogen, 2010a,b], resulted in the relative speedup of 4.31 and the absolute speedup 3.34 on the same machine. A parallel sparse Gauß elimination with parallel pivoting [Demmel et al., 1999] produces for dense matrix input the speedup of $\approx$ 3.5 on 8 PE Cray C90 and $\approx$ 5 on 8 PE DEC AlphaServer 8400. Still for special sparse inputs the same algorithm on the latter machine achieves speedup of $\approx$ 7. Noteworthy, Demmel et al. [1999] describe an SMP implementation. Our approach works in a distributed memory setting.

As a further indicator of possible benefit, we compute in Maple 13

- Determinants of larger sparse random-generated matrices with density ½ over small (absolute value < 1000) integers using default, rational and residue-based implementations of built-in Maple function `Determinant`. We show the result in Table 7.7.

- Determinants of middle-sized dense random-generated matrices over small fractions. The maximal absolute value of numerator and denominator is < 100. Again, we used the methods of the

Fractions	*Maple method*					*Speedup*		
	n	default ($\mathbb{Q}$)	`algnum`	$\mathbb{Z}_m$	`float`	n	$\bar{s}$	$\tilde{s}$
	50	2.03	5.98	–	0.001	50	–	–
	100	84.66	98.38	0.002	0.003	100	42 330	49 190
	120	176.21	229.19	0.004	0.004	120	44 052	57 297

Table 7.8: *Determinants of rational random dense matrices. Comparing Maple methods and stating upper bound on the speedups. Time is in seconds.*

`Determinant` function. We show the result in Table 7.8. We have shown this experiment for $n = 100$ in the introduction.

The residue-based approach measures time for one residue class at $m = 1013$. Note that the timings for the same matrix size differ for vulgar fraction computation in Tables 7.6, 7.7 and 7.8. The reason is: we compute over different rings with different matrices. Table 7.6 features permuted Pascal matrices, they are dense, in the Maple version these matrices are over integers. Maple seems to figure out the structure of these matrices. In Table 7.7 we present timings for random sparse integer matrices. Table 7.8 features dense random rational matrices. The method `algnum` computes over the field of algebraic numbers. The method `float` uses floating point approximations to the rationals. Contrarily to the methods, we present here, the `float` method is the classical result from numeric computing. It does not provide exact results. We see its performance is comparable with the performance of the residue-based method. The values at $\bar{s}$ are the best possible speedup of default method, values at $\hat{s}$ show the best possible speedup of the rational method for integer matrices, and values at $\tilde{s}$ show the best possible speedup of the `algnum` method for the rational matrices, all provided ideal scalability of the multiple-residue arithmetic. Note that residue-based computation promises exceptional speedup values! These values assume that computing p residues at p PEs takes as long as computing one residue at a single PE and that we have as many PEs as we need. Hence, this is the maximal possible speedup in the particular case. The enormous speedups of the rational determinant computation (Table 7.8) are explained with the intermediate expression swell. Of course, practically measured values may and will vary. We have only modelled a proper multiple-residue rational arithmetic with its single-residue integral counterpart. Still, we see a huge potential for using a multiple-residue arithmetic for rational computations.

7.9.3 Further Related Work

Any decent computer algebra system implements residue-based approach for reducing the intermediate expression swell. A lot of work on rational residue systems was done in [Gregory and Krishnamurthy, 1984] and preceding papers [Gregory, 1981, Wang, 1981, Wang et al., 1982, Kornerup and Gregory, 1983, Rao et al., 1984]. Another proof of Theorem 7.7 is in [Sasaki and Sasaki, 1992], Sasaki et al. [2002] state an alternative divide and conquer algorithm. The earliest approaches to rational residues known to us are [Svoboda and Valach, 1957] and [Borosh and Fraenkel, 1966]. Our own works on this topic are [Lobachev, 2007, 2011a, Lobachev and Loogen, 2010a,b]. We discussed [Kornerup and Matula, 2010] in the introduction to this chapter. It is a new book, mentioning the topic of this chapter among others. The connections of the integer case with polynomials are shown in [von zur Gathen and Gerhard, 2003]. These include the link between rational reconstruction and Padè approximation and the correspondence of CRT and polynomial interpolation. The latter leads to a further proof of the CRT, which we learnt in [Sauer, 2002]. A parallel Haskell implementation of integer multiple-residue arithmetic is [Loidl, 1997].

Another approach to the same goal is the residual representation of rational numbers with Hensel codes, see, e.g., [Kornerup and Gregory, 1983, Rao et al., 1984, Koblitz, 1984, Buchberger et al., 1985, Zarowski and Card, 1990, Limongelli, 1993]. Morrison [1988] features a parallel rational arithmetic based on Hensel codes. For Hensel lifting of polynomial equations, where representations modulo p^k, for prime p, are used, see [Koblitz, 1984, von zur Gathen and Gerhard, 2003, Grabmeier et al.,

2003]. It is often called the 'p-adic Newton lemma'. Contrary to this approach, we remain in *multiple* representations modulo several distinct m_i and do not compute in residual powers. Both Hensel codes and Hensel lifting are called so after Kurt Hensel, *29.12.1861, †1.6.1941.

7.10 Conclusions and Future Work

We have presented an integer and a rational multiple-residue arithmetic approaches and their parallel implementation in Eden. We constructed a test case, provided the visualisation and the performance estimation of the test program execution. We have presented concise source code for almost all operations. The omitted parts are straightforward.

Our approach to the rational multiple-residue arithmetic removes in an innovative manner the common factors of input fraction's numerator and denominator with the moduli, i.e., the primes, modulo which the computation is performed. We deduced this method in 2006 and presented it in [Lobachev, 2007], See [Lobachev and Loogen, 2010a,b, Lobachev, 2011a] for details.

The transformations to the integral and rational data parallel arithmetic were deduced mathematically. The presented methods form a generic data parallelisation scheme, suitable for symbolic computing algorithms. The only additional requirement is an *a priori* upper bound on the final result. The presented scheme can be applied to an arbitrary method, requiring an exact computation with a known bound on the final result, be it with integers or with fractions. In other words: we provided an alternative high-level approach to parallelism.

Possible future work would be an attempt for an adaptive computation. It would be interesting to see a Maple implementation of our approach. Another research direction would be to optimise the hypercube rotation from Section 7.8.3. A distributed implementation, based on the remote data concept [Dieterle et al., 2010b] should improve the speedup values of our implementation. Although it is possible to parallelise Algorithm 21, we currently refrain from doing so. The reconstruction of integer values modulo M occupies only a tiny fraction of the execution time in our setting.

It would be interesting to apply the parallelisation approach of this chapter to further algorithms of computer algebra. As we already know, the GCD computation for particular polynomials can be heavily optimised with multiple-residue integer arithmetic, see, e.g., [von zur Gathen and Gerhard, 2003]. We plan to apply this scheme to Gröbner bases computation, related approaches are [Ebert, 1983, Buchberger et al., 1985, Winkler, 1988, Arnold, 2003, Idrees et al., 2011].

Chapter 8

CONCLUSIONS, FUTURE, AND RELATED WORK

Let us close this work with some final remarks. We begin with enumerating our contributions. Below we will also present the related and future work. Computer algebra is a relatively new field of computational mathematics. It utilises computationally intensive algorithms that benefit greatly from parallelisation. We performed high-level parallelisation with algorithmic skeletons and transformations to data parallel algorithms, and we parallelised a meaningful selection of computer algebra algorithms. We chose the Eden language for the parallel implementation. We focused on core algorithms of symbolic computation. We have presented suitable algorithmic skeletons (and further techniques) and have found a way to ensure portability of our results to larger scales and input sizes. We used existing and developed new measures to verify efficiency of the programs developed in this thesis. Based on our results we can conclude that *high-level parallelisation of core computer algebra algorithms is indeed effective*. We summarise below the main contributions of this thesis.

8.1 Contributions

We have used type classes to express ad-hoc polymorphism of symbolic computation. We developed algorithmic skeletons for high-level parallelisation of important computer algebra algorithms. We were the first to develop alternative rational arithmetic for generic high-level parallelisation of computations. We made a test study of high-level actors implementation in Eden: a first actors' implementation in Eden to our knowledge. Novel techniques using statistical methods for performance estimation and evaluation were introduced in this thesis. To be more elaborate:

- We have discussed some language design aspects of computer algebra systems in Chapter 2. We introduced the term 'language unity' for computer algebra systems having the same language in their implementation and interface parts. This chapter shares material with [Lobachev and Loogen, 2008] and facilitates the work in the remaining chapters. To our knowledge we are the first ones to emphasise the importance of language unity. [Bauer et al., 2002] and [Mechveliani, 2007a] made similar decisions, but did not highlight the language unity and did not emphasise its importance.

- Chapter 3 presented the programming language foundation of this work: the parallel functional programming language Eden [Loogen et al., 2005]. We have also presented existing algorithmic skeletons and tools for the analysis of parallel program execution.

- In Chapter 4 we have deduced a generic approach to estimate execution time of parallel programs. We employed statistical methods and an ingenious separation of the specific parts of the parallel computing time to be able to predict the execution time for non-measured input sizes and for non-measured numbers of processors. We have used the estimation and evaluation methods from Chapter 4 throughout this thesis. We have estimated execution times of our parallel programs and compared the results with the observed values. Additionally, we found the measure 'parallel overhead w.r.t. the number of processing elements' to show effectively the quality of the parallel implementation. We have used our new quality measure and the serial fraction [Karp and Flatt, 1990] to ensure parallelisation quality of our programs. Our approach provided us with some very interesting insights into the details of the parallelisations in question. Possible reasons for particular values of the parallel penalty ranged from non-perfect task distribution to communication overhead. We applied our methods to some examples in Chapter 4. We presented an example in Eden, and an example using time measurements on Jülich Blue

Gene/P supercomputer. We have additionally performed theoretical analysis of task distribution for computation on a supercomputer. The approach of this chapter has been first presented in [Lobachev and Loogen, 2010c]. A journal publication on the same topic was submitted to the *International Journal of Parallel Programming*.

The usage of statistical methods and the suggested division of parallel program time are our original research.

- We have developed a generic `map+reduce` skeleton scheme for repeated computations with a possibility of premature termination. It was especially easy to implement this scheme in Eden. Laziness played an important role in our elegant implementation. Using the instantiations of `map+reduce`: the skeletons `farm+reduce` and `workpool+reduce`, we have implemented the parallel Rabin–Miller and the parallel Jacobi sum tests in Chapter 5. This resulted in a complete toolchain for fast (i. e., probabilistic) primality testing. We have performed estimations of execution time of both parallel tests. We analysed task balancing issues in the parallel Rabin–Miller test and made successful predictions for optimal configurations. We introduced optimisations for task balancing in the parallel implementation of the Jacobi sum test. We predicted the run times of both tests with low relative error.

  We are not aware neither of the `map+reduce` scheme nor of the parallelisation of Jacobi sum test having been presented before. Parallelisation of the Rabin–Miller test is quite straightforward, however, we have not seen a parallel high-level approach towards such a parallelisation in literature. We have submitted an article on our parallelisation scheme to the special issue on multicore programming in *International Journal of of High Performance Computing and Networking*.

- Chapter 6 dealt with fast multiplication in its various forms. We have implemented the divide and conquer skeletons for both distributed and flat expansion of the divide and conquer tree. We instantiated these skeletons with Karatsuba multiplication, the fast Fourier transform (FFT), and Strassen matrix multiplication [von zur Gathen and Gerhard, 2003]. The FFT was also implemented with the distributable homomorphism approach [Gorlatch, 1998a]. We experimented with numerous parallelisation approaches for divide and conquer algorithms with very low expense on the programmer's side because of the modularity of the skeleton-based approach. Existing skeletons in the same class are drop-in replacements for one another.

  We made an observation concerning applicability of the direct divide and conquer implementation of the parallel FFT. Thus we have seen both the interchangeability of algorithmic skeletons ('the skeleton `divConFlat` works everywhere!') and their limits. The quality of our parallel implementations was quantitatively evaluated in this thesis. We explained the performance of our implementation of Strassen multiplication with theoretical task distribution analysis. Its results were supported by practical observation of the parallel penalty values.

  In Section 6.6 we implemented a divide and conquer skeleton using *actors* [Hewitt et al., 1973, Agha, 1985, Haller and Odersky, 2006, 2009]. We modelled them in Eden with lazy streams [Wray and Fairbairn, 1989], a `farm` skeleton and pattern matching. In a sense, such an approach is similar to task creating `farm`, *viz.* [Priebe, 2006, Brown and Hammond, 2010]. Note, we utilised a skeleton to implement this approach, resulting in another skeleton.

  We used the fast Fourier transform to implement fast multiplication of polynomials with bounded coefficients in $\mathbb{Z}$. The latter approach and the Karatsuba multiplication of polynomials correspond directly with approaches for integer multiplication. Chapter 6 shares material with [Berthold et al., 2009a,b,c]. These papers introduced new, more special divide and conquer skeletons to Eden. Summarising, we have shown approaches to the fast multiplication of three different basic mathematical structures. We utilised our prediction methods to estimate the execution times. The results had small relative errors for Strassen multiplication and FFT and extremely small for Karatsuba method: 0.025%.

- Chapter 7 presented our novel approach to data parallel rational arithmetic. We used multiple residue classes. Special attention was paid to the common factors of numerator or denominator

with the moduli—the numbers modulo which the congruence relation is constructed. We have introduced a method to remove all such common factors and to ensure the feasibility of a rational multiple-residue arithmetic. We parallelised it naturally. The parallel integer arithmetic was implemented as a byproduct. These two arithmetic implementations can be applied to various algorithms. They enable easy parallelisation of various symbolic computation approaches, provided an upper bound on the result can be stated. The same chapter presented test results for parallelising Gauß elimination with the rational arithmetic. We used a `farm` skeleton in the implementation of parallel Gauß elimination. We analysed the task balancing of this implementation. We predicted its execution time using our methods with very small relative error. Our presentation shares material with [Lobachev, 2007, 2011a, Lobachev and Loogen, 2010a,b].

This chapter is different from the others in two aspects. Firstly, we introduced new algorithms instead of the parallelisation of existing methods. Secondly, the parallelisation approach presented in Chapter 7 forms a scheme for parallelisation of computer algebra algorithms. It is not an algorithmic skeleton, but a further reusable approach towards easy parallelisation of computational algorithms in a data parallel manner.

The presented algorithms and suggested parallelisation approach of Chapter 7 are results of our long-term research that begun 2006. Recent publications on similar topics, like [Arnold, 2003, Idrees et al., 2011], only emphasise the importance of this research.

- We found some features of the parallel functional programming language Eden [Loogen et al., 2005] very useful in the context of this thesis. We used laziness to express actors in Chapter 6, as mentioned above. We also used it in the implementation of the FFT and of the `map+reduce` scheme. These implementations can function also without laziness, but laziness accounts for their elegance. Some further applications of laziness are, e. g., in [Loidl, 1997, Hinze, 2008]. We presented an example, based on [McIlroy, 1999, 2001], in Chapter 2.

  The expressiveness of Haskell, which is the sequential base for Eden, was particularly useful in the implementation of mathematical algorithms. To give an example, we were able to express the FFT in just a few lines of code in Chapter 6; the implementation of Rabin–Miller test was an almost direct translation of the corresponding algorithm, see Chapter 5. Ad-hoc polymorphism and type classes elegantly made symbolic computing possible in our framework. We emphasised these features in Chapters 6 and 7. We made a use of Eden's futures [Dieterle et al., 2010b] in Chapter 6.

  Using the parallel functional lazy programming language Eden, we were able to express high-level parallelisations in a concise and efficient manner. The language choice contributed significantly to the programmer's productivity that was required to produce results presented in this work.

Summarising, we firstly developed new high-level parallelisation techniques applicable to multiple computer algebra problems; and secondly introduced and maintained a certain quality measure for the performance of programs in question. We were able to predict the performance of our test programs using a novel estimation method. Symbolic methods were the case studies for these results. Thus we have shown, *we can parallelise selected computer algebra algorithms effectively on a high level of abstraction*. The *gained knowledge* on the parallelisation *is reusable* through our estimation technique.

List of Publications

[**Lobachev and Loogen, 2008**] O. Lobachev and R. Loogen. Towards an implementation of a computer algebra system in a functional language. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Intelligent Computer Mathematics*, LNAI 5144, pages 141–154. AISC 2008: 9th International Conference on Artificial Intelligence and Symbolic Computation, Springer-Verlag, 2008

[**Berthold et al., 2009a**] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. In K.-E. Großpitsch, A. Henkersdorf, S. Uhrig, T. Ungerer, and J. Hähner, editors, *Workshop on Many-Cores at ARCS '09: 22nd International Conference on Architecture of Computing Systems 2009*, pages 47–55. VDE-Verlag, 2009a

[**Berthold et al., 2009b**] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. In V. Malyshkin, editor, *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83. Springer-Verlag, 2009b. Extended version in [Berthold et al., 2009c]

[**Berthold et al., 2009c**] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. Technical Report bi2009-2, Philipps-Universität Marburg, Fachbereich 12 – Mathematik und Informatik, 2009c

[**Lobachev and Loogen, 2010a**] O. Lobachev and R. Loogen. Implementing data parallel rational multiple-residue arithmetic in Eden. In V. P. Gerdt, W. Koepf, E. W. Mayr, and E. H. Vorozhtsov, editors, *CASC'2010: Computer Algebra in Scientific Computing*, LNCS 6244, pages 178–193. Springer-Verlag, 2010a. Extended and revised version in [Lobachev and Loogen, 2010b]

[**Lobachev and Loogen, 2010b**] O. Lobachev and R. Loogen. Implementing data parallel rational multiple-residue arithmetic in Eden. Technical Report bi2010-3, Fachbereich Mathematik und Informatik der Philipps-Universität Marburg, 2010b

[**Lobachev and Loogen, 2010c**] O. Lobachev and R. Loogen. Estimating parallel performance, a skeleton-based approach. In *Proceedings of 4th International Workshop on High-level Parallel Programming and Applications*, pages 25–34. ACM Press, 2010c

[**Lobachev, 2011a**] O. Lobachev. *Multimodulare Arithmetik*. Logos, Berlin, Germany, 2011a. ISBN 978-3-8325-2881-2

[**Lobachev et al., 2011**] O. Lobachev, M. Guthe, and R. Loogen. Estimating parallel performance. *International Journal of Parallel Programming*, 2011. Submitted

[**Lobachev, 2011b**] O. Lobachev. On an implementation of parallel computation skeletons with premature termination property. *International Journal of High Performance Computing and Networking*, 2011b. Submitted to the Special Issue on Multicore Programming

8.2 Related Work

We mention here the publications that are related to the whole concept of this thesis. Please see for the related work sections in the individual chapters for a more detailed related work on an appropriate topic.

Computer algebra systems. Any computer algebra system is related to our approach. The most popular general purpose ones are Maple [Redfern, 1995, Monagan et al., 2005] and Mathematica [Wolfram, 1991, 2000]. More special systems are GAP [GAP, 2008], CoCoA [Capani and Niesi, 1995, CoCoA, 2009], GiNaC [Bauer et al., 2002], Singular [Greuel et al., 2001, Decker et al., 2011], PARI [Batut et al., 2010], NTL [Shoup, 2009], DISCRETA [Betten et al., 1997]. We also would like to mention LiDIA [Biehl et al., 1995], KANT V4 [Daberkow et al., 1997], Macaulay2 [Eisenbud, 2002, Grayson and Stillman, 2010], BiPolar [von zur Gathen and Gerhard, 2002], see also [Grabmeier et al., 2003]. One of the first computational approaches was [Williams, 1962]. First *systems* were Macsyma (Maxima in the open source variant) [Martin and Fateman, 1971], Scratchpad (later: Axiom) [Griesmer and Jenks, 1971], Reduce [Hearn, 1966, 2004], SAC-I (later SACLIB) [Buchberger et al., 1992], MuMath (later: Derive), see [Grabmeier et al., 2003]. Later, Axiom/Aldor [Jenks and Sutor, 1992, Bronstein et al., 2003], Magma [Bosma et al., 1997], MuPAD [Fuchssteiner, 1994] emerged.

8.2.1 Parallel Computer Algebra

We can divide existing parallel implementations of algorithms of computer algebra into few categories. The first are SMP-based systems, typically using threads. They feature a typical low-level parallelism on multiprocessors. The second feature coarse-grained distributed memory parallelism. The third interconnect existing systems. All of them focus on parallel computer algebra *systems*. The fourth option is to examine particular algorithms. See below for related work on this issue.

Threads and Co. MuPAD [Fuchssteiner, 1994, Naundorf, 1995], Macaulay2 [Eisenbud, 2002], Cannes/Parcan [Gloor and Muller, 1997], and PARSAC [Kuechlin, 1995] included some parallel computing features. In the first two cases, parallel processing relies completely on the shared memory setting. For example, Macaulay2 features threads. PARSAC has threads and green threads, but it was also extended to distributed memory systems. PARSAC evolved from an SMP implementation to support of distributed memory. Our approach functions both on distributed and shared memory machines. We paid special attention to its very good performance in an SMP setting. In the PARSAC case, much focus was also on algorithms, see, e. g., [Amrhein et al., 1996]. Cannes/Parcan was developed in C/C++ [Grabmeier et al., 2003]. It suffered from memory management issues. The differences of MuPAD, Parcan, Macaulay2, and PARSAC with our approach are twofold. Firstly, these are complete systems, while we focus on a bundle of core algorithms. Secondly, these approaches focus on low-level parallelisation with threads. In a contrast, we consider high-level parallelisation approaches.

Distributed systems and special languages. KANT V4 is capable of running on a network of workstations [Grabmeier et al., 2003]. Independent computations can be managed in a master-worker manner. We have used a similar approach in some parallelisations in this thesis. However, methods like FFT or fast multiplication algorithms required a much more fine-grain handling of parallelism, see Chapter 6. We have also seen in Chapter 5 how large the differences can be in the duration of a single task in a complicated number theoretical algorithm parallelised with a master-worker skeleton.

PACLIB [Schreiner and Hong, 1993, Schreiner, 1994] aimed for a parallel functional language for computer algebra. This is surely related to our approach. However, Schreiner focuses much more on a parallel language with constructs suitable for computer algebra implementation. We take such for granted with Eden, and focus on parallelisation of computer algebra algorithms.

Orchestration. A quite recent move to forge an interface between various systems also having a parallelism option is SCIEnce Project [Hammond et al., 2007, SCIEnce, 2010, Brown and Hammond, 2010, Brown et al., 2010], see also ‖Maple‖ [Siegl, 1993], Maple-Eden interface [Martínez and Peña, 2004], Distributed Maple [Schreiner et al., 2003]. These are heterogeneous systems where the coordination happens in a different language, than computation. Both [Martínez and Peña, 2004] and [Brown and Hammond, 2010] use Eden as a coordination language. Siegl [1993] also used a functional language for coordination purposes. We use Eden for both coordination and computation.

Algorithms. Parallel algorithms of computer algebra have been studied since for quite a long time. Works on this topic include fast multiplication (and the fast Fourier transform) [Morrison, 1988, Cesari and Maeder, 1996a,b, Hammes et al., 1997, Gorlatch, 1998a, Gorlatch and Bischof, 1998, Jebelean, 1997, Dmitruk et al., 2001, Crandall et al., 2004, Marti-Puig et al., 2008], matrix manipulation [Luo and Drake, 1995, Gupta and Sadayappan, 1996, Demmel et al., 1999, 2008, Li et al., 2002, Nguyen et al., 2005, Agarwal et al., 2010], GCD computation (and related topics) [Borodin et al., 1982, von zur Gathen, 1984, Sedjelmaci, 2008], Gröbner bases computation [Ebert, 1983, Watt, 1986, Buchberger and Jebelean, 1992, Pauer, 1992, Winkler, 1988, Arnold, 2003, Kredel, 2010, Idrees et al., 2011] (not directly parallel, but definitely of benefit is [Sasaki and Kako, 2007]), and other topics, like the LLL algorithm [Wetzel, 1998, Backes and Wetzel, 2009]. We handle related work in a more detail in appropriate chapters when discussing concrete algorithms. Sequential methods serving as foundation for the parallel implementation include [Karatsuba and Ofman, 1962, Cooley and Tukey, 1965, Schönhage and Strassen, 1971, Strassen, 1969] for the fast multiplication methods, [Miller, 1976, Rabin, 1980, Adleman et al., 1983,

Cohen and Lenstra, 1984] for primality testing, and [Borosh and Fraenkel, 1966, Szabo and Tanaka, 1967, Gregory and Krishnamurthy, 1984, Lobachev, 2007, 2011a] for rational residue classes.

Known publications in this area either focus on *algorithms* or use *low-level* parallelism, e.g., the one based on message passing. In a contrast, we focus on high-level parallelism, e.g., algorithmic skeletons.

8.2.2 Parallel Computing in General

Parallel computing as such is often based on low-level libraries for threads or message passing [Grama et al., 2003, Nichols et al., 1996, Snir et al., 1995, MPI, 2009, Sunderam, 1990, PVM, 2009] (for Haskell implementations of MPI see [Breitinger et al., 1998, Astapov et al., 2011]). Higher-level concurrency approaches include communicating sequential processes by Hoare [1978] (see also [Abdallah et al., 2005], refer to [Brown, 2008] for Haskell implementation) and actors [Hewitt et al., 1973, Agha, 1985, Armstrong, 2007, Haller and Odersky, 2006, 2009, Sulzmann et al., 2008, Sulzmann, 2008, Epstein et al., 2011]. We discussed actors briefly in Chapter 6. An interesting approach to concurrency is software transactional memory. Please refer to [Knight, 1986, Herlihy and Moss, 1993, Shavit and Touitou, 1995, 1997, Welc et al., 2008, Adl-Tabatabai and Shpeisman, 2009, Harris et al., 2010] for more information on this topic. Haskell implementation is discussed, e.g., in [Discolo et al., 2006, Peyton-Jones, 2007, Harris et al., 2008]. Concurrent Haskell approaches include [Scholz, 1995, Peyton-Jones et al., 1996]. We mostly focused on genuinely parallel and not concurrent approaches.

Parallel Haskells

We chose Eden [Loogen et al., 2005] (see also [Breitinger, 1998, Berthold, 2008]) as our implementation language. Further parallel or distributed Haskell approaches include pH [Nikhil et al., 1995, Nikhil and Arvind, 2001], GpH [Trinder et al., 1996b, 1999, Marlow et al., 2009], Data Parallel Haskell [Chakravarty et al., 2007], Cloud Haskell [Epstein et al., 2011] and the `Par` monad [Marlow et al., 2011]. See discussion in Chapter 3 for more information. A typical research direction for a parallel Haskell is implicit parallelism, in a contrast to explicit parallelism model in Eden. We made some use of the *futures* [Dieterle et al., 2010b], see also [Alt and Gorlatch, 2003].

8.2.3 Skeletons

The algorithmic skeletons were introduced by Cole [1989]. An overview of existing skeleton libraries is in [González-Vélez and Leyton, 2010]. One of the first skeleton libraries was [Darlington et al., 1993]. Kesseler [1995] presented a skeleton approach in Clean [Brus et al., 1987]. Further development was made by Botorog and Kuchen [1996, 1998] with the Skil language. It is imperative, but features such interesting aspects as higher-order function and polymorphism. Sérot et al. [1999] presented a special skeleton library in CAML. PAS (Parallel Architectural Skeletons) [Goswami et al., 2002] is a C++ library, relying on MPI for message passing. Its extension EPAS [Akon et al., 2005] allows to specify new skeletons in a special extension language. P^3L (Pisa Parallel Programming Language) [Bacci et al., 1995, Danelutto et al., 1997, Pelagatti, 2003] and its derivative SKiE [Bacci et al., 1999a] are skeleton-based coordination languages. The generic 'templates' could have multiple implementations, e.g., for different platforms. SKELib is an associated C library [Danelutto and Stigliani, 2000]. Muesli [Kuchen, 2011, Kuchen and Striegnitz, 2005, Poldner and Kuchen, 2008a,b] is a further development of Skil ideas in C++ templates. Skeleton composition in Muesli is similar to P^3L. Data parallel skeletons in Muesli feature automatic scaling across the PEs, using both OpenMP [Dagum and Menon, 1998] and MPI [Snir et al., 1995]. Alba et al. [2002] presented a C++ library for combinatorial optimisation. The search strategies are encoded as skeletons. SkeTo [Matsuzaki et al., 2006] is also a C++ skeleton library using MPI. The eSkel library [Benoit and Cole, 2002, Benoit et al., 2005] is a C skeleton framework using MPI. SBASCO (Skeleton-BAsed Scientific COmponents) [Díaz et al., 2004] is a large framework for numeric computing. It combines functional and non-functional components. HOC-SA [Dunnweber and Gorlatch, 2004] uses higher-order components for grid infrastructure. It includes such features as abstraction from Globus toolkit and usage of web services. JaSkel is a Java-based skeleton library

[Ferreira et al., 2006]. Lithium/Muskel [Aldinucci et al., 2003] and Calcium/Skandium [Leyton and Piquer, 2010] are also Java skeleton libraries.

Contrarily to most aforementioned approaches, in Eden the skeleton implementation language is the same as the skeleton instantiation language, see, e.g., [Galán et al., 1996, Klusik et al., 2001, Hammond et al., 2003, Loogen et al., 2003, Berthold and Loogen, 2005, Priebe, 2007]. New Eden approach to skeleton composition is [Dieterle et al., 2010b]. A classic publication on Haskell-related skeleton approach is [Michaelson et al., 2001]. Herrmann [2000] considered divide and conquer skeletons in a Haskell dialect. In GpH [Trinder et al., 1998b] parallelism can be expressed with evaluation strategies [Trinder et al., 1998a] that can be capsuled in algorithmic skeletons. The deterministic `Par` monad [Marlow et al., 2011] can also be used to implement skeletons.

Particular skeleton implementations in Eden include various workpools [Priebe, 2006, Berthold et al., 2008, Dieterle et al., 2010a, Brown and Hammond, 2010], topological skeletons [Berthold and Loogen, 2006], and divide and conquer skeletons [Loogen et al., 2003, Berthold et al., 2009a,b,c]. Further Eden skeletons include `map-reduce` [Berthold et al., 2009d] and systolic skeletons [Loogen et al., 2003].

Skeletons are sometimes used as an aid for the performance estimation, like in, e.g., [Zavanella, 2001, Cole and Hayashi, 2002, Benoit et al., 2004]. See [Lobachev and Loogen, 2010c] for discussion of skeleton-based, specialised versions of our performance estimation method.

8.2.4 Further Related Work

Our approach of uniting the implementation and interface languages of a CAS (see Chapter 2) is supported by [Mechveliani, 2000, Bauer et al., 2002]. Haskell libraries featuring some aspects of computer algebra are [Mechveliani, 2007a,b, Amos, 2007, Thurston et al., 2010, Brown et al., 2010]. An interesting bridge between Haskell and mathematics is formed by infinite data structures [Wray and Fairbairn, 1989, Bird et al., 1997], see [Karczmarczuk, 1999, McIlroy, 1999, 2001] and also [Hinze, 2008]. These works focus on the laziness of Haskell. We discussed an example in Chapter 2 and used laziness throughout this thesis. The gain was in a more concise representation of complex processes.

8.3 Future Work

In a close connection to this work, further topics can be designated. An adaptation of the rational multiple-residue arithmetic of Chapter 7 to the needs of Gröbner bases computation—another essential method of computer algebra—is a future vision. An implementation of the negative wrapped convolution (see Chapter 6) will relax the bound on the coefficient size for FFT-based polynomial multiplication. In the context of [SCIEnce, 2010] an interface to other systems is required.

This thesis has a conceptional focus on parallel algorithms. A similar work focused on data structures and considering parallel domain construction would be a very interesting reading. For the reasons we saw in Chapter 2, it would need dependent typing. A good candidate is Agda [Bove et al., 2009]. A sequential Haskell approach for domain construction is [Mechveliani, 2000, 2007a,b].

Symbolic computation is a very large field, so we can easily name topics we chose not to consider in this thesis, but which might constitute an interesting material, having the basics that this work provides. *Division* is a large interesting topic for which both advanced domain construction (for Hensel lifting) and advanced parallelisation techniques are required [von zur Gathen and Gerhard, 2003, Grabmeier et al., 2003]. *Symbolic addition and integration* is another area of symbolic data manipulation where it would be interesting to consider the parallel case [von zur Gathen and Gerhard, 2003, Bronstein, 2005]. Note, we briefly handled differentiation and integration of power series, and hence also: of polynomials, in Chapter 2. The *factorisation of polynomials* is very interesting, not least because of the challenges connected with Hensel lifting [Hensel, 1908, Koblitz, 1984, Grabmeier et al., 2003], and because of the LLL algorithm with its potential for symbolic-numeric computing [von zur Gathen and Gerhard, 2003, Nguyen and Stehlé, 2005].

Appendix

Appendix A

FOUNDATIONS

> A negative minus zero is negative, a positive [minus zero] is positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.
>
> Brahmagupta,
> *Brahmasphutasiddhanta*, 628 A. D.
> Considered the first mention of zero.

We need to define some background concepts reoccurring through the complete thesis or serving as a premise for some of its contributions. We handle some abstract algebra first, then switch to an elementary number theory—so elementary, it can be called arithmetic!—and continue defining some domains from the abstract algebra, which integers, univariate and multivariate polynomials share. Matrices also fulfil some of the below laws, but not all, as matrix multiplication is not commutative.

The next section discusses the issues of notation. Section A.2 provides some background in abstract algebra. Then we arrive at the definition of numbers in Section A.3, a short treatment of the euclidean algorithm in Section A.4 and the definition of prime numbers in Section A.5. After these sections we can return to an abstract treatment in Section A.6. A very brief introduction to what we will need from the notion of vectors and matrices is in Section A.7.

A.1 Notation

A.1.1 Mathematical Notation

We write vectors and matrices in **bold**, when referring to an entity, but no more, if we refer to a single element of it. For instance, $\mathbf{v} = [v_1, v_3, v_3]$ is a vector of length three and

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

is a 2×2 matrix. Squared braces '[ ]' denote vectors and curly braces '{ }' denote sets. If a fact of something being a set is not important and we merely need a collection of items of the same type, we often call it a vector, with lists or arrays in mind. However, we do not use these two words interchangeably. In our notation, we omit parenthesis whenever possible, but care not to be ambiguous. We follow a chapter-based numbering style: a lemma, theorem, example and so on have a number of the chapter, following a running incremental number. Tables and figures also obey to this scheme, but have their own counters. We have one exception to this rule: algorithms are thoroughly counted.

We sometimes write $\Rightarrow$ for an implication. Thus, $\Longleftrightarrow$ is equivalence. We assume set theory as given. Also, we write sets in Latin uppercase letters and set elements in Latin lowercase, e. g., X is a set and a, b are its elements if $a, b \in X$ holds. The subset and superset relations are denoted with $\subset$, e. g., $X \subset Y$ for some sets X and Y. We sometimes write $\subsetneq$ if we want to stress the inequality. The elements of the set can be enumerated within curly brackets: $\{0, 1, 2, \dots\}$. We also write conditionals in a similar way. The symbol $\varnothing$ denotes an empty set. To give an example for two last issues: for integers a, b, c and n holds $\{a^n + b^n = c^n : n > 2\} = \varnothing$ per Last Fermat's Theorem. The number sets $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ are assumed to be known to the reader. Nevertheless, we will introduce the first three with a bit more formalism in this chapter. If we want to stress that $\mathbb{N}$ begins with a zero, we write $\mathbb{N}_0$. A sign $\mathbb{R}_0^+$ stands

for non-negative real numbers, i.e., $\mathbb{R}_0^+ := \{x \in \mathbb{R} : x \geq 0\}$. We assume the reader is familiar with the concept of relations. Occasionally we write 'B is defined as A' as $B := A$. Further, $\lfloor x \rfloor$ for $x \in \mathbb{R}^+$ means such $a \in \mathbb{N}$ that $a = \max\{a \in \mathbb{N} : a \leq x\}$. Analogously, $\lceil y \rceil = \min\{b \in \mathbb{N} : b \geq y\}$ for $y \in \mathbb{R}^+$ and $b \in \mathbb{N}$. Vertical bars $|\cdot|$ denote an absolute value of a number, i.e., the number without a sign. But $|a|_m$ denotes the residue of a modulo m, see Chapter 7 for more on this. The double vertical bars $\|\cdot\|$ denote some norm. We index the norm with some identifier, to tell various norms apart.

As usual, we write $\sum_{i=0}^{n} a_i$ for $a_0 + \cdots + a_n$ and $\prod_{i=0}^{n} a_i$ for $a_0 \cdots a_n$. We may omit the coefficients at the sum and product symbols, if they are very clear from the context. We use the common symbols for factorial and binomial coefficients: for $n, k \in \mathbb{N}$ we write $n! := \prod_{i=1}^{n} i$ and $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

We write log for a logarithm, when a base does not really matter. We write $\log_b$ for a logarithm in basis b. Similar to $\ln = \log_e$ and $\lg = \log_{10}$, we define $\mathrm{lb} = \log_2$. Here e is the Euler constant, 2.718281828459...

A.1.2 Algorithms

> A programming language is low level when its programs require attention to the irrelevant.
>
> Alan J. Perlis, *Epigrams on programming*

We often use pseudocode to describe algorithms used in this work. We chose the common imperative style of the pseudocode. The functional implementation is stated later in terms of Haskell source code. We use the usual language constructs, familiar from C [ISO/IEC 9899:1999] and Pascal [Wirth, 1971]. We try to keep the side effects at minimum.

We describe briefly the structure of an algorithm description in pseudocode. Each algorithm begins with an 'require' and ends with 'ensure' clauses, stating the input and output of the algorithm. The actual actions to be performed are described with plain text and mathematical notation. The left arrow denotes assignment, e.g., $x \leftarrow 42$. Following keywords are used. The keywords **for**, **while**, **if**, **then**, **else**, **return** have their usual meaning. See Algorithm 25 for an example. A **repeat–until** loop has the semantics of **do–while not**. Its body is executed at least once. With **map** we designate an application of a given function to a list. Note that the **return** statement breaks up all loops.

Helper algorithms can be introduced. In this case the pseudocode either refers to the other algorithm explicitly or performs a *procedure call*. For an example of the latter see the call of the Is Prime procedure in Algorithm 25, line 6. The assumption here is that the procedure Is Prime exists and has the type 'integer → boolean'. A declaration of a procedure is shown in line 1 of Algorithm 25. Two slashes introduce a comment, like one in line 2.

Algorithm 25 An example algorithm

Require: an integer n

```
 1: procedure First Example(n)
 2:     set n ← n + 1. // For what reason?
 3:     for all i > 0 and i < n do
 4:         for all odd j with j < i do
 5:             if n · j = 42 then return false
 6:             else if Is Prime(i + j) then return true
 7:             end if
 8:         end for
 9:     end for
10:     return true
11: end procedure
```

Ensure: **true** or **false**.

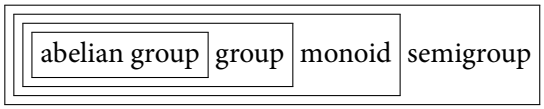


Figure A.1: *The relations between several group concepts. We denote the subset relations, e.g., all groups are also monoids.*

A.1.3 Complexity

We use the standard 'big Oh' complexity notation. A function $f : \mathbb{N} \to \mathbb{R}_0^+$ is in a $\mathcal{O}(g)$ class of some $g : \mathbb{N} \to \mathbb{R}_0^+$, if some $N \in \mathbb{N}$ exists, that for all $n > N$ in $\mathbb{N}$ holds $f(n) \leq cg(n)$ for some constant $c > 0$. Note that both f and g have no negative values. We often write in sign $f \in \mathcal{O}(g)$ for the preceding definition. Other related notations follow. As above, we assume $f, g : \mathbb{N} \to \mathbb{R}_0^+$.

- *'Little oh'*: $f \in o(g) \iff$ some $N \in \mathbb{N}$ exists, that for all $n > N$ in $\mathbb{N}$ holds $f(n) \leq cg(n)$ *for all* constants $c > 0$.
- *'Big omega'*: $f \in \Omega(g) \iff$ some $N \in \mathbb{N}$ exists, that for all $n > N$ in $\mathbb{N}$ holds $f(n) \geq cg(n)$ for some $c > 0$.
- *'Big theta'*: $f \in \Theta(g) \iff$ some $N \in \mathbb{N}$ exists, that for all $n > N$ in $\mathbb{N}$ holds $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for some constants $c_1, c_2 > 0$.
- *'Little omega'*: $f \in \omega(g) \iff$ some $N \in \mathbb{N}$ exists, that for all $n > N$ in $\mathbb{N}$ holds $f(n) \geq cg(n)$ *for all* constants $c > 0$.

We need to note, that originally the 'ohs' in the names were omicrons. This notation is also known as Bachmann–Landau notation, after Paul Gustav Heinrich Bachmann *22.6.1837, †31.3.1920 and Edmund Georg Hermann Landau *14.2.1877, †19.2.1938, number theorists.

A.2 Groups, Rings, Fields I: The Definitions

The classic sources for abstract algebra are [van der Waerden, 1971, Lang, 2002]. However, here we follow [Grove, 2004], which provides an overview at a graduate course pace.

Definition A.1. A *group* $G = (G, \cdot)$ is some non-empty set G and a corresponding binary operation $\cdot$ on it, which fulfils the following axioms. For all $a, b, c \in G$:

1. *Closure:* a and $b \in G \Rightarrow a \cdot b \in G$.
2. *Associativity:* $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
3. *Identity:* a neutral element $1 \in G$ exists with $a1 = a = 1a$ for all $a \in G$.
4. *Inverse:* For each $a \in G$ an inverse $a^{-1} \in G$ exists, with $aa^{-1} = 1 = a^{-1}a$.

The group G in the above definition is a *multiplicative* group, nothing which is different from an additive group up to notation. A neutral element of a multiplicative group is also called a *unity*. Do Axioms 3 and 4 not hold, then such an object is called a *semigroup*. Does only 4 not hold, then it is a *monoid*. An example for a monoid is $\mathbb{N}_0$. Such a group G, where for all $a, b \in G$ holds $a \cdot b = b \cdot a$, is also called an *abelian group*. An example for latter is $(\mathbb{Z}, +)$. We denote explicitly if a group should not be abelian. An overview of the relations of the above concepts is presented in Figure A.1.

A ring is a non-empty set R with *two* operations $+$ and $\cdot$, where $(R, +)$ is a full-fledged abelian group, and $(R, \cdot)$ is just a semigroup. For non-trivial rings, also called rings with unity, holds that $(R, \cdot)$ is a monoid. If $(R, +)$ is merely a commutative monoid, we call R a semiring. A subset of a ring, that is invertible under $\cdot$, is denoted with R^*. It is a multiplicative group. Elements of R^* are called *units*. Note that a unity is also a unit, but not all units are unities, see also Lemma A.4 on page 191.

We regard commutative non-trivial rings in this thesis, we call them 'commutative rings with unity' or simply 'rings'.

A ring closed under multiplicative inversion is a *field*. In other words: a set F, where both $(F,+)$ and $(F,\cdot)$ are groups, is a field $(F,+,\cdot)$. Naturally, for a field F holds $F^* = F \setminus \{0\}$. An example is $\mathbb{Q}$, the set of all rational numbers. But in most cases we do not need a whole ring. For additive groups the inverse is often denoted with (unary) $-$, and the neutral element with 0. So $a+(-a) = 0$, for $a \in (G,+)$. Further, given $a, b \in (R,+,\cdot)$, the common notion for $a + (-b)$ is $a - b$ and that for $a \cdot b^{-1}$ is a/b (for $b \in R^*$). Let R be a ring. We denote with $R[x]$ the ring of univariate polynomials over R. Similarly, $R[x_1,\ldots,x_n]$ denotes the ring of multivariate polynomials over R. The next proposition is [Grove, 2004, Theorem 4.1, p. 58]. See there for the proof.

Proposition A.2. *If R is a ring, then also $R[x_1,\ldots,x_n]$ is.*

In fact it's not just a ring [Grove, 2004]. For a (commutative) ring R an *R-module* is an additive abelian group A, where *scalar multiplication* of an element in R and an element in A with the result in A is defined and is additive in both components and multiplicative. If additionally, for $a, b \in A$ and $r \in R$ holds $r(ab) = (ra)b = a(rb)$, then A is a *R-algebra*. If R is a commutative ring with unity, then any $R[x_1,\ldots,x_n]$ is an R-algebra. By the way, an extension of a field is also an algebra over that field. We show a stronger statement on structure of $R[x_1,\ldots,x_n]$, depending on structure of R, as Theorem A.25 on page 197.

A.3 The Numbers

In this section we will define the well-known notions of natural, integral and rational numbers as well as the concepts of integer division and of associates. We follow [Ebbinghaus et al., 1992] below.

Definition A.3 (Natural numbers)**.** Let $\mathbb{N}$ be a set of *natural numbers*. It holds that a neutral element, the zero, in sign: 0, is an element of $\mathbb{N}$. Further, a mapping $\sigma : \mathbb{N} \to \mathbb{N}$, a *successor mapping* is defined, such that

1. The mapping σ is one-to-one (injective).
2. Zero is not in the domain of σ.
3. If some set M with $0 \in M$ exists, and for all $m \in M$ holds $\sigma(m) \in M$, then $M \supset \mathbb{N}$.

The famous Peano axioms, named after Giuseppe Peano, *27.8.1858, †20.4.1932, correspond exactly to the above definition. The original Peano axioms can be formulated as the following definition. In fact, we assume here that the equality = is a reflexive, symmetric and transitive relation.

Definition (Peano)**.** We can define $\mathbb{N}$, σ and 0 with the following statements.

1. It holds $0 \in \mathbb{N}$.
2. Is $n \in \mathbb{N}$, then $\sigma(n) \in \mathbb{N}$.
3. Is $n \in \mathbb{N}$, then $\sigma(n) \neq 0$.
4. Is $0 \in M$ and does it hold for all $n \in M$ that $\sigma(n) \in M$, then $\mathbb{N} \subset M$.
5. For n and m in $\mathbb{N}$ the fact $\sigma(n) = \sigma(m)$ means that $n = m$.

Strictly, one needs to show that $\mathbb{N}$ is a monoid and that $\mathbb{N}$ is unique. However, we omit these proofs here and refer to [Ebbinghaus et al., 1992] for a more rigorous treatment of this topic.

As for integer numbers, we define them as pairs of natural numbers. To be more precise, *viz.* [Ebbinghaus et al., 1992, Chapter 1, §3], define an equivalence relation $\sim$ on pairs of natural numbers. For a, b, c, d in $\mathbb{N}$ let

$$(a,b) \sim (c,d) \quad \Longleftrightarrow \quad a + d = b + c.$$

Figure A.2: *Natural numbers and beyond. The subset relation is denoted, e.g., $\mathbb{Z} \subset \mathbb{Q}$.*

We omit here the proof, that $\sim$ is in fact an equivalence relation. Let $[\cdot]$ denote here the equivalence class of the above relation:

$$[(a,b)] = \{(a,b) : (a,b) \sim (c,d) \text{ for all } a,b,c,d \in \mathbb{N}\}.$$

Then $\mathbb{Z}$ is the set of all such equivalence classes. For the convenience we write the specimen of the equivalence class $[(0,a)]$ for some $a \in \mathbb{N}$ as $-a$, an *unary minus*.

One can define the addition and multiplication in $\mathbb{Z}$ basing on the equivalence class representation. For the latter: $[(a,b)] \cdot [(c,d)] := [(ac+bd, ad+bc)]$. We can extend the notion of unary minus to $\mathbb{Z}$, hence defining the additive inverse in $\mathbb{Z}$.

Lemma A.4. *The set of all integers $\mathbb{Z}$ is a group under $+$. But $\mathbb{Z}$ is only a monoid w.r.t. $\cdot$.*

Proof. We can easily verify this fact: $(\mathbb{Z}, +)$ is naturally closed and associative. The neutral element is $0 \in \mathbb{Z}$. The negative of an integer is its additive inverse. Although $(\mathbb{Z}, \cdot)$ is closed and associative, and a neutral element w.r.t. multiplication, the unity, exists, no reciprocal a^{-1} of $a \in \mathbb{Z}$ with $|a| \neq 1$ is again in $\mathbb{Z}$. So, for the set of units of integers holds $\mathbb{Z}^* = \{-1, 1\}$. □

A similar approach is possible with the rational numbers. We define them as pairs of integers. To be more precise [Ebbinghaus et al., 1992, Chapter 1, §4]:

Definition A.5. Consider the relation $(a,b) \approx (c,d) \iff ad = bc$ for a and $b \in \mathbb{Z}$, c and $d \in \mathbb{Z} \setminus \{0\}$. The equivalence classes of this relation are denoted with a/b or $\frac{a}{b}$ and are called *fractions*, the set of all fractions is denoted with $\mathbb{Q}$.

It is easy to verify that $\mathbb{Q}$ together with addition and multiplication

$$\frac{a}{b} + \frac{c}{d} := \frac{ad+bc}{bd} \quad \text{and} \quad \frac{a}{b} \cdot \frac{c}{d} := \frac{ac}{bd} \quad \text{for all } a,c \in \mathbb{Z} \text{ and } b,d \in \mathbb{Z} \setminus \{0\}$$

forms a field. We do not consider real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$ in detail here. A hierarchy of numbers in shown in Figure A.2.

A.4 Euclidean Algorithm

The division for all non-zero elements is defined for $\mathbb{Q}$. But it is possible to introduce a special division for integral numbers.

Definition A.6 (Integer division)**.** For any integers a and b such c and r exist that $a = bc + r$. We write $c = a \operatorname{div} b$ and $r = a \bmod b$. Further, we write $b \mid a$, in words 'b divides a', if $r = 0$. Is it not the case, we write $b \nmid a$, 'b does not divide a'.

Definition A.7 ([Grove, 2004], p. 61)**.** Two elements a and b of a ring R are called *associates*, if some unit $u \in R^*$ exists with $a = ub$.

Lemma A.8. *If two elements a and b of a ring are associates, then $a \mid b$ and $b \mid a$.*

Proof. Follows trivially from $a = ub \iff u^{-1}a = b$ for a unit u. □

Algorithm 26 Euclidean algorithm

Require: integers a and b
1: **repeat**
2: Let $r \leftarrow a \bmod b$.
3: Set $a \leftarrow b$
4: and $b \leftarrow r$.
5: **until** $r \neq 0$
6: **return** b.
Ensure: $\gcd(a, b)$

The classic notion of the euclidean algorithm serves exactly for determining the *greatest common divisor*, which we abbreviate with GCD. If the $\gcd(a, b)$ is unity for some integers a and b, the input values have no common factors, i.e., $a \nmid b$ and $b \nmid a$.

Following [Graham et al., 1994], we often write $a \perp b$ for $\gcd(a, b) = 1$. Note that in theoretical computer science the orthogonality symbol $\perp$ often denotes the 'bottom'. It stands for a diverging program (e.g., `f 42` $= \perp$) or for the least element of a partially ordered set. However in the first case $\perp$ is a binary relation and in the second case $\perp$ is a constant. So, it is possible to tell the two different usages apart from the context. We consider the euclidean algorithm next.

History. Euclid (Εὐκλείδης) of Alexandria lived around 300 A.C. and is referred to as 'Father of Geometry'. He is most famous for his book *Elements*. Euclid's name means good glory. His *Elements* (Στοιχεῖα) is the most successful textbook ever. It was used up to the end of 19th century for teaching geometry. The *Elements* defines mathematics with deduction from small set of axioms. One can say, that this book was the very first and still extremely influential foundation of mathematics we know. It was printed first in Venice in 1482 and has survived over thousand editions. Only the Bible has more.

Donald Knuth considers the euclidean algorithm to be the oldest non-trivial algorithm in the history of mathematics [Knuth, 1998]. It contains a **while** loop, so the termination condition of the algorithm is not immediately clear from its description. However a modern notion of *euclidean rings* tells us, that division with remainder always produces a 'smaller' value than its largest input. Hence a 'division chain', which is essentially the euclidean algorithm, terminates with a zero in finite many steps. Curiously enough, the notion of 'smaller' might be quite general, it is not limited to a smaller absolute value, *cf.* Section A.6. The algorithm appears in Euclid's *Elements*, but historians suppose it predates Euclid. Coming apparently from the pythagoreans, the algorithm is supposedly was known to Eudoxus (Εὔδοξος) of Cnidus, *410 or 408 A.C., †355 or 347 A.C., and to Aristarchus (Ἀρίσταρχος) of Samos, *310 A.C., †c. 230 A.C. The euclidean algorithm was independently discovered in India and China centuries after Euclid. It took approx. 200 years to comprehend the importance of the euclidean algorithm for the number theory [Stillwell, 1998]. The algorithm was then utilised in Europe for the efficient solution of Diophantine equations. Later, its application in the context of continued fractions emerged [Brezinski, 1981]. Gauß mentions the euclidean algorithm in this context in 1801. The notion of the extended euclidean algorithm (*cf.* Algorithm 15 on page 146) is connected with names of Saunderson and Cotes. It provides us with Bèzout identity: for any integers a and b there exist such x and y that $\gcd(a, b) = xa + yb$ holds. Dirichlet sees the euclidean algorithm as a base for number theory. The algorithm is related to Sturm chains. Please refer [MacTutor, 2011, Chabert, 1999] for more information on Euclid's biography and work. Books on number theory also sometimes include some historic notices, like [Stillwell, 1998] does. Further sources are [van der Waerden and Habicht, 1966, Eves, 1969, Chabert, 1999, Fowler, 1999].

The algorithm. The euclidean algorithm computes the GCD of two integers, see 26. The polynomial variant could be similarly straightforward, but requires much finesse have good performance [von zur Gathen and Gerhard, 2003]. For a variant of the algorithm, which operates with subtraction and not division, see Algorithm 27 on the facing page. Notably, the euclidean algorithm is a base for computer

Algorithm 27 Subtraction-based euclidean algorithm

Require: integers a and b
1: Let $r \leftarrow a - b$.
2: **while** $r \neq 0$ **do**
3: set $a \leftarrow b$ and $b \leftarrow r$.
4: **end while**
5: **return** b
Ensure: $\gcd(a, b)$

algebra. We have brought this thought from Tomas Sauer's lectures [Sauer, 2002]:

> The subtitle of a half of [*Computer Algebra*] lecture could be 'variations of the euclidean algorithm'.

Indeed, the representation of rationals in residue rings, we present in Chapter 7 is nothing else than an utilisation of the extended euclidean algorithm. A more generic case is Padé reconstruction of rational functions. Further, the famous Buchberger algorithm for computing Gröbner bases is a very wide generalisation of the same euclidean algorithm.

A.5 Primes

In Section A.6 we will define primes in terms of abstract algebra. What we consider here, are old good prime *numbers*.

Definition A.9 (Prime number)**.** A natural number > 1 is a *prime number* if its only divisors are unity and the number itself.

To give an example, 5 is prime, because $2 \nmid 5$, $3 \nmid 5$, $4 \nmid 5$. The 'dark ages' of number theory know multiple methods of prime search, but they all are not quite efficient. At the same time, these methods are deterministic. The most ancient and still quite able one is the sieve of Eratosthenes. The idea of the sieve is very simple. Lay out a list of natural numbers. We begin with a prime and strike out all multiples of it. Thus we arrive at a next prime and repeat. There are optimisations considering the amount of striking out and 'wheeling'—pre-filtering of *a priori* unfitting input. The first steps of the sieve are presented in Table A.1 on the next page. We do not use any optimisations there. Note that some numbers, like 15 for 3 and 5, have been stroked out multiple times.

The person the sieve is called after, Eratosthenes (Ἐρατοσθένης) of Cyrene (*c. 276 A.C., †c. 195 A.C.) was a prominent Ancient Greek scientist. His contributions included the measurements of the circumference of the Earth and of the tilt of the Earth's axis. Him is attributed the calculation of the distance from the Earth to the Sun and the invention of the leap day. Further, Eratosthenes was the head of the Library of Alexandria [Eves, 1969, MacTutor, 2011]. As for the sieve, there is an opinion that Eratosthenes had not discovered the sieve, but merely gave a name to the already known method, *viz.* [Waschkies, 1989].

A very interesting summary with underlying functional implementations of the sieve was done by O'Neill [2008]. We presented modern approaches to primality testing in Chapter 5. Primes as such are used in Chapter 7.

Primes are important because every natural number can be expressed as a product of powers of primes. This makes primes the universal 'building blocks' of the natural numbers. This result has it own name: Fundamental Theorem of Arithmetic. We show it elegantly as Corollary A.26.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	…
[2]	3	*	5	*	7	*	9	*	11	*	13	*	15	*	17	*	19	*	21	*	…
2	[3]		5	*	7		*		11		13		*		17		19		*		…
2	**3**		[5]		7			*	11		13		*		17		19	*			…
2	**3**		**5**		[7]				11		13	*			17		19		*		…
2	**3**		**5**		7				[11]		13				17		19		*		…
2	**3**		**5**		7				**11**		[13]				17		19				…
⋮																					⋱

Table A.1: *First steps of the sieve of Eratosthenes.*

A.6 Groups, Rings, Fields II: Domains and Ideals

So, let us get back to abstract algebra. It is well possible to define the euclidean algorithm on some particular rings and there is an abstraction of notion of primes too! We refer to [Becker et al., 1993] and [Grove, 2004] as bases for our presentation.

Euclidean Domains. In order to abstract the euclidean algorithm to generic fields, we need to look at the proof of termination of the euclidean algorithm. It depends on the definition of division. To be more precise: the residue r of the division $a = bc + r$ in a ring R should be smaller than b. For the next definition, *viz.* [Becker et al., 1993, Definition 2.23] or [Sauer, 2002, Section 2.2]

Definition A.10 (Euclidean function)**.** Let R be a commutative ring with unity. A function $\varphi : R \to \mathbb{N} \cup \{-\infty\}$, with $\varphi(a) \leq \varphi(ab)$ for all $a, b \in R$, is called an *euclidean function,* if for all $a, b \in R$ with $\varphi(a) \geq \varphi(b)$ such $c, r \in R$ exist, that $a = bc + r$ and $\varphi(b) > \varphi(r)$. We agree that if for $x \in R$ holds $\varphi(x) = -\infty$, then $x = 0$.

Definition A.11 (Domains)**.** Let R be a commutative ring with unity.

1. R is an *integral domain*, if it contains no zero divisors, i. e., if for all $a, b \in R$ with $a \neq 0 \neq b$, holds $ab \neq 0$.

2. An integral domain R is an *euclidean ring* or euclidean domain, if a corresponding euclidean function exists (see Definition A.10).

Corollary A.12. *$\mathbb{Z}$ is an integral domain.*

To give a counterexample, residue classes in general are not integral domains: $2 \cdot 3 = 0 \bmod 6$.

Remark ([Grove, 2004], p. 49). Another interesting fact: R is an integral domain if and only if $R \setminus \{0\}$ is a multiplicative semigroup.

Example A.13.

1. *Let $R = \mathbb{Z}$. Then a possible φ is the absolute value $|\cdot|$. With it, $\mathbb{Z}$ is an euclidean domain. For a more detailed proof see [Becker et al., 1993, Proposition 2.25].*

2. *Let R be a field. Then a 'Dirac' function*

$$\delta(x) = \begin{cases} 1 & \text{if } x \neq 0 \\ 0 & \text{for } x = 0 \end{cases}$$

is an euclidean function.

For an euclidean ring R holds $R^* = \{x \in R \setminus \{0\} : \varphi(x) = \varphi(1)\}$. This is [Grove, 2004, Proposition 5.5, p. 65].

Definition A.14 ([von zur Gathen and Gerhard, 2003], Definition 3.3)**.** We generalise | straightforwardly to euclidean domains.

1. We denote *greatest common divisor* d of two elements a and b of an euclidean domain R with euclidean function φ as $d = \gcd(a, b) \in R$ with
   - $d \mid a$ and $d \mid b$.
   - For all $x \in R$ fulfilling the above property, holds $\varphi(x) \leq \varphi(d)$.
2. Let R be as above. We denote the *least common multiple* $l \in R$ of two elements a and b in R, $l = \operatorname{lcm}(a, b)$ with
   - $a \mid l$ and $b \mid l$.
   - For all other $x \in R$ with above property, holds $\varphi(l) \leq \varphi(x)$.

It is possible to compute GCD not merely for integers, but also for univariate polynomials. We can *define* the GCD in more generic rings, but we need an euclidean ring to compute it! The next lemma provides us with a 'reality check'. We know that $1 \in \mathbb{Z}$ is a unit. But also $1/2 \in \mathbb{Q}[x]$ is!

Lemma A.15 (An intuition)**.** *Let F be a field. Then $(F[x])^* = F^*$.*

Proof. Non-constant polynomials have no inverses, already ax^{-1} is not a polynomial anymore. But constant polynomials have inverses, as our polynomial ring is over a field. The zero polynomial has no inverse. □

Ideals. The ideals are an essential instrument of the ring theory. Let us define them first. For more information see [Becker et al., 1993, Definition 1.33] and [Grove, 2004, p. 50] or any algebra book, like [van der Waerden, 1971].

Definition A.16. Let R be a ring and let $\mathfrak{I}$ be an non-empty subset of R. Then $\mathfrak{I}$ is an *ideal* of R if

- $\mathfrak{I}$ is additive: for all $a, b \in \mathfrak{I}$ holds $a + b \in \mathfrak{I}$.
- For any $k \in R$ and all $a \in \mathfrak{I}$ holds $ka \in \mathfrak{I}$.

For any subset X of R, an ideal *generated* by X, denoted $\langle X \rangle$ is the smallest ideal, which contains X. An ideal $\mathfrak{I} \subset R$ is *trivial*, if $\mathfrak{I} = \{0\}$ and *proper* if $\mathfrak{I} \subsetneq R$. An ideal, generated by a single element, is called *principal*.

We typeset ideals in *fraktur*, a black-letter font. The convention is that an uppercase $\mathfrak{I}$ is some ideal, a lowercase $\mathfrak{n}$ is a principal ideal, generated by the element n.

Example A.17.

1. *The set $n\mathbb{Z}$ of all multiples of a $n \in \mathbb{Z}$ is an ideal $\mathfrak{n}$. For a more concrete example: $2\mathbb{Z} = \mathfrak{2} = \{\ldots, -4, -2, 0, 2, 4, \ldots\}$.*
2. *For an euclidean domain R, the notion $d = \gcd(a, b) = xa + yb$ is equivalent to $\mathfrak{a} + \mathfrak{b} = \mathfrak{d}$.*

The second part of the above example is shown in [Becker et al., 1993, Lemma 1.70]. Proposition A.21 gives a more formal treatment of this.

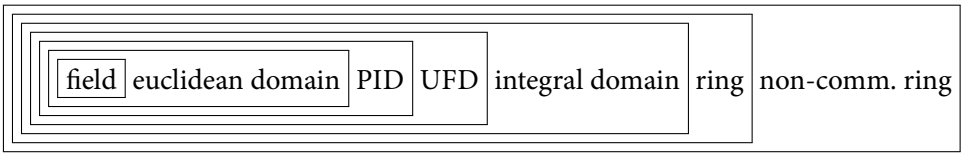


Figure A.3: *The relation between several ring concepts. We denote subset relation, e.g., every UFD is a PID.*

Two further domains. Now we can define two further domains, already shown in Figure A.3: the PID and the UFD.

Definition A.18. A *principle ideal domain*, abbreviated PID, is a ring, where each ideal is principle.

The next statement is [Grove, 2004, Proposition 5.6, p. 65]. The converse does not hold.

Lemma A.19. *Every euclidean ring is a PID.*

Proof. Let $\mathfrak{I}$ be an arbitrary non-trivial ideal in an euclidean domain R. Choose such non-zero $b \in \mathfrak{I}$, that $\varphi(b)$ is minimal. We can write $a = bc + r$ for some $a \in \mathfrak{I}$. It holds $\varphi(r) \leq \varphi(b)$. As $r = a - bc \in \mathfrak{I}$, $r = 0$ per choice of b. Because b is the minimal non-zero element in $\mathfrak{I}$ w.r.t. φ, the sharp inequality $\varphi(r) < \varphi(b)$ cannot hold. So $a = bc$, meaning $\mathfrak{I} = \mathfrak{b}$. □

Definition A.20. Let R be a commutative ring. An ideal $\mathfrak{m} \subsetneq R$ is *maximal* if no ideal $\mathfrak{p} \subsetneq R$ exists, with $\mathfrak{m} \subsetneq \mathfrak{p}$. The *prime* ideal $\mathfrak{p} \subset R$ is such an ideal, that for any $x, y \in R$ with $xy \in \mathfrak{p}$ either $x \in \mathfrak{p}$ or $y \in \mathfrak{p}$ holds.

The GCD is unique in a PID. More precisely stated:

Proposition A.21 ([Grove, 2004], Proposition 5.3, p. 62)**.** *Let R be a PID and let a and b be non-zero elements of R. Then a and b* have *a GCD, it values to d with*

$$d = xa + yb$$

for some $x, y \in R$. The GCD is unique up to associates.

Note the emphasis on 'have' in the above proposition. It is about GCD *existing*, not about *computing it*.

Proof. Let $\mathfrak{I} = \langle a, b \rangle = \{ua + tb : u, t \in R\}$. As R is a PID, $\mathfrak{I} = \mathfrak{d}$. For that d also $d = xa + yb$ with some particular x and $y \in R$ holds. Clearly: d is a common divisor of a and b.

Remains to show maximality and uniqueness. As for the first: let $c \in \mathfrak{I}$ be another common divisor of a and b. As $c \mid a$ and $c \mid b$, then $c \mid a + b$, so $c \mid xa + yb$, hence $c \mid d$. This means that d is maximal, i.e., d is the GCD of a and b. As for uniqueness, let d' be another GCD. Then $d \mid d'$ and $d' \mid d$. So, d and d' are associates. □

Proposition A.22 ([Grove, 2004], Corollary 2, p. 55)**.** *If R is an integral domain, then also $R[x]$.*

The following definition can be found in [Grove, 2004, p. 61 and p. 66] and [Becker et al., 1993, Definition 2.52].

Definition A.23. Let R be an integral domain.

1. A *prime* element $p \in R$ is not a zero and not a unit, such that for a and $b \in R$ holds: if $p \mid ab$ then either $p \mid a$ or $p \mid b$.

2. An *irreducible* element has only units and its own associates as divisors.

3. A *unique factorisation domain*, abbreviated UFD, is an integral domain, where every element, being not zero and not unit, is a product of irreducible elements and all irreducible elements are prime elements.

Remark. Primes are irreducible in every integral domain.

The following theorem summarises few classic algebra results. Basing on [Grove, 2004], item 1 follows immediately from item 3, our Lemma A.19 and our Example A.13. Item 2 is a non-inductive version of Grove's Theorem 5.16, page 69. Item 3 is Grove's Proposition 5.10 on page 66, and item 4 is a corollary to the aforementioned theorem from the same book. Look there for the proof.

Theorem A.24 (A bit of intuition)**.**

1. *$\mathbb{Z}$ is a UFD.*
2. *Generally, if R is a UFD, then $R[x]$ is a UFD. For example, $\mathbb{Z}[x]$ is a UFD.*
3. *Every PID is a UFD.*
4. *If R is a UFD, then $R[x, y]$ is a UFD, but not a PID anymore.*

Theorem A.25 ([Grove, 2004], Theorem 5.11, p. 67)**.** *The factorisation in a UFD is unique. More precisely: let R be an UFD and $x \in R$ be not a unit and not zero. Then a unit $u \in R^*$, powers $\nu_1, \ldots, \nu_n \in \mathbb{Z}$ and primes $p_1, \ldots, p_n \in R$ exist, such that*

$$x = u p_1^{\nu_1} p_2^{\nu_2} \cdots p_n^{\nu_n}.$$

Corollary A.26 (Fundamental Theorem of Arithmetic)**.** *Any integer can be (up to the sign) uniquely decomposed into a product of primes.*

Proof. The ring of integers $\mathbb{Z}$ is a UFD. □

Remark. The statement of Proposition A.21 holds also if R is a UFD [Grove, 2004, Proposition 5.12, p. 67]. However, to *compute* GCD, we need the euclidean algorithm, so R should be an euclidean ring for that sake.

A.7 Matrices and Vectors

In a contrast to scalar values, we denote vectors and matrices in a bold font, lowercase and uppercase correspondingly. So, x is a scalar, $\mathbf{x}$ is a vector and $\mathbf{X}$ is a matrix. The uppercase 'T' denotes transposition, it turns the row vector into a column vector and vice versa. For matrices, transposition replaces rows and columns. Vectors and matrices of fitting dimensions can be multiplied. Any vector or matrix can be scaled with a scalar. There are different vector-vector products, which we do not consider in more detail. For the remaining products holds with $a \in K$ a scalar, $\mathbf{v} \in K^n$ a vector and two matrices $\mathbf{X}, \mathbf{Y} \in K^{n \times m}$, for an UFD K:

$$a\mathbf{v} = a[\nu_1, \ldots, \nu_n]^T = [a\nu_1, \ldots, a\nu_n]^T,$$

$$a\mathbf{X} = a \begin{pmatrix} x_{11} & \ldots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \ldots & x_{mn} \end{pmatrix} = \begin{pmatrix} ax_{11} & \ldots & ax_{1n} \\ \vdots & \ddots & \vdots \\ ax_{m1} & \ldots & ax_{mn} \end{pmatrix},$$

$$\mathbf{X}\mathbf{v} = \begin{pmatrix} x_{11} & \ldots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \ldots & x_{mn} \end{pmatrix} \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_n \end{bmatrix} = \begin{pmatrix} x_{11}\nu_1 & \ldots & x_{1n}\nu_n \\ \vdots & \ddots & \vdots \\ x_{m1}\nu_1 & \ldots & x_{mn}\nu_n \end{pmatrix},$$

$$\mathbf{X}\mathbf{Y} = \begin{pmatrix} x_{11} & \ldots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \ldots & x_{mn} \end{pmatrix} \begin{pmatrix} y_{11} & \ldots & y_{1l} \\ \vdots & \ddots & \vdots \\ y_{m1} & \ldots & y_{ml} \end{pmatrix} = \left(\sum_{j=1}^{m} x_{ij} y_{jk} \right)_{ik} \quad \text{for all } 1 \le i \le n, 1 \le k \le l.$$

Here we write, e.g., $(x)_{ij}$ for the entry at (i, j) of the matrix $\mathbf{X}$ and, e.g., v_i for the i^{th} entry of the vector $\mathbf{v}$. Matrix product is defined only if the dimensions of both matrices correspond. It is not commutative, i.e., in general $\mathbf{XY} \neq \mathbf{YX}$ holds.

Appendix B

CODE

B.1 Helper Functions for Rabin–Miller Test

In Figure 5.16 we omitted the code for the function `singleRabinMiller` and two further helper functions. We amend this omission hereby. Figures B.1, B.2 and B.3 show the Haskell implementation of `powermod`, `separate` and `singleRabinMiller` respectively.

```
-- | computes powers in residue rings, i.e. b^e mod m
powermod :: Integer    -- ^ the base ...
         → Integer    -- ^ the exponent ...
         → Integer    -- ^ ... modulo m
         → Integer    -- ^ the result b^e mod m
powermod b e m
  | (e==0)           = 1
  | (e `mod` 2 == 0) = (temp * temp) `mod` m
  | otherwise        = b * (powermod b (e-1) m) `mod` m
  where temp = (powermod b (e `div` 2) m)
```

Figure B.1: *Implementation of* `powermod`.

```
separate :: (Integer, Integer)  -- ^ input (n,0)
         → (Integer, Integer)  -- ^ output (q,t) w. n=2^t*q
separate (q, t)
  | (q `mod` 2 == 0) = separate (q `div` 2, t+1)
  | otherwise        = (q, t)
```

Figure B.2: *Implementation of* `separate`.

```
-- | Rabin-Miller test for a given basis a
singleRabinMiller :: Integer  -- ^ n is prime candidate
                  → Integer  -- ^ t of n=2^t*q
                  → Integer  -- ^ e is the current value of a^((2^e)q)
                  → Integer  -- ^ b = a^q
                  → Bool     -- ^ strong pseudoprime w.r.t. a or not?
singleRabinMillerBool n t e b
     | (b==1)     = True
     | (b==n-1)   = True
     | (e ≤ t-2)  = singleRabinMiller n t (e+1) (b*b `mod` n)
     | otherwise  = False
```

Figure B.3: *Implementation of* `singleRabilMiller`.

B.2 Helper functions for Jacobi sum test

We show the signatures of some helper functions in Figure B.4. The code to simplify cyclotomic polynomials is in Figure B.5, the function `easysimp` was defined in Figure 5.25 on page 76. The sequential implementation of the Jacobi sum test is in Figure B.6.

```
type Poly a = [a]
```

```
-- implementation details omitted, we show only selected type signatures
multInverse :: Integer → Integer → Integer
prodsimmod :: Poly Integer → Poly Integer → Integer
           → Integer → Integer → Poly Integer
gpotsimmod :: Poly Integer → [(Integer, Integer)]
           → Integer → Integer → Integer → Poly Integer
isrootofunity, isprrootofunity :: Poly Integer → Integer
                                → Integer → Integer → Bool
binpolypotsimmod :: Poly Integer → Integer → Integer
                 → Integer → Integer → Poly Integer
```

Figure B.4: *Types of helper functions for Jacobi sum test*

```
-- simplifies a cyclotomic polynomial
-- uses the fact ζ_n^n=1 and the minimal polynomial
simpol :: Poly Integer    -- ^ polynomial to simplify
       → Integer          -- ^ prime p
       → Integer          -- ^ power k of p
       → Poly Integer     -- ^ simplified polynomial
simpol poly p k = begin - remain * (minPoly p k)
  where (begin, remain) = separate (easysimp poly (p^k)) ((p-1)*p^(k-1))
  -- a Num instance for polynomials is defined

-- produces the minimal polynomial for the given root of unity
minPoly :: Integer       -- ^ prime p
        → Integer        -- ^ power k of p
        → Poly Integer   -- ^ minimal polynomial
minPoly p k = foldr (+) [] [ nrootpot (p^k) (i*p^(k-1)) | i ∈ [0..(p-2)] ]

-- produces the power of p-th root of unity in an extension of ℤ
nrootpot :: Integer       -- ^ number p
         → Integer        -- ^ power of the root of unity
         → Poly Integer   -- ^ result: the element of the extension
nrootpot = ... -- implementation omitted
```

Figure B.5: *Simplifying polynomials Φ_n, the hard way.*

```
jacobiSumTestCase1 :: Integer → Integer → Integer
                      -- ^ p, k and q from p^k || q-1
                   → Integer   -- ^ prime candidate n
                   → [Integer] -- ^ list of primes with l_p=0
                   → [(Integer,Integer)] -- ^ tabled function f
                   → Maybe [Integer] -- ^ the result:
                                     -- Nothing: not a prime
                                     -- Just _: possibly updated list
jacobiSumTestCase1 p k q n lps fps
  | (not $ isRootOfUnity spq p k n) = Nothing
  | (isPrRootOfUnity spq p k n) = Just (delete p lps)
    -- remove the prime p from the list
  | otherwise = Just lps
    -- keep the list as is
    where eset = [l | l ∈ [1..n], (l `mod` p) ≠ 0]
          theta = [(x, multInverse x n) | x ∈ eset ]
          alpha = [((r*x) `div` n, multInverse x n) | x ∈ eset ]
          r = n `mod` l
```

```
l = p^k
s1 = wgpow (j p q fps) theta
s2 = wpow s1 (n `div` l)
spq = wprod s2 jpot
jpot = wgpow (j p q fps) alpha
wpow b e = binpolypotsimmod b e p k n
wprod w1 w2 = prodsimmod w1 w2 p k n
wgpow b e = gpotsimmod b e p k n
```

Figure B.6: *Jacobi sum test: sequential implementation of helper Algorithm 4.*

B.3 Skeletons of dcF Class

As discussed in Chapter 6, it is possible to express the skeletons of the dcF class with skeletons of dcC class. Here we show the exact interfaces.

```
divConSeq :: (Trans a, Trans b)
          ⇒ DC' a b
divConSeq trivial solve divide combine x
   = divConSeq_c trivial solve divide (λ_ parts → combine parts) x
```

Figure B.7: *The sequential divide and conquer skeleton of the dcF class.*

```
dcNTickets :: (Trans a, Trans b)
           ⇒ Int      -- ^ n (expect n children)
           → [Int]    -- ^ tickets (machine ids to use)
           → DC' a b  -- ^ the dcF divide-and-conquer type
dcNTickets k ts trivial solve divide combine x
  = dcNTickets_c k ts trivial solve divide (λ_ parts → combine parts) x
```

Figure B.8: *The `dcNTickets` skeleton of dcF class.*

```
divConFlat :: (Trans a,Trans b)
           ⇒ Map a b  -- ^ custom map implementation
           → Int      -- ^ depth
           → DC' a b  -- ^ the dcF divide-and-conquer type
divConFlat myParMap depth trivial solve divide combine x
  = divConFlat_c myParMap depth trivial solve divide
                 (λ_ parts → combine parts) x
```

Figure B.9: *The `divConFlat` skeleton of dcF class.*

B.4 The Optimised Boilerplate Code for Gauß Elimination

Here we show the source code for the function `lift1'`, the optimised version of `lift`, shown in Figure 7.15 on page 167. Helper functions are shown in Figure B.10, the actual implementation of `lift1'` is in Figure B.11. The functions `toL` and `fromL` implement the signatures from Figure 7.15 on page 167, `liftLSdiag` is an enhanced variant of `liftL` from that figure.

```
type MatArr i n = ... -- array-based matrix representation
type TransMat i n = ((i, i), [[n]])

toL :: (Num n, NFData n) ⇒ Int → MatArr Int n → TransMat Int n
toL shuff arr = let ((_, _), (n, m)) = bounds arr
```

```
                    ll = unshuffleN shuff $ elems arr
                    lm = rnf ll `pseq` ll
                in ((n, m), lm)
fromL :: (Num n) ⇒ TransMat Int n → MatArr Int n
fromL ((n, m), xss) = listArray ((1,1), (n, m)) $ shuffleN xss

type SparseMatEl i n = ((i, i), n)
type SparseMatDiag i n = ((i, i), [[SparseMatEl i n]])

-- adapt the chunking size to the diagonal
diagC :: Int → Int
diagC = round ∘ sqrt ∘ fromIntegral

toSd :: Num n ⇒ Int → MatArr Int n → SparseMatDiag Int n
toSd c a = let b@((mn,mm), (xn,xm)) = bounds a
               xs = [((i,i), a!(i,i)) | i ∈ [mn..xn] ]
               xss = unshuffleN (diagC c) xs
           in ((xn-mn+1, xm-mm+1), xss)

fromSd :: Num n ⇒ SparseMatDiag Int n → [n]
fromSd ((n, m), xss) = map snd $ shuffleN xss
```

Figure B.10: *Helper functions for* `lift1'`.

```
-- convert only the diagonal
liftLSdiag :: (Num x, Num y)
           ⇒ Int -- ^ chunking size
           → (MatArr Int x → MatArr Int y)
             -- ^ working function, e.g. gauss
           → TransMat Int x -- ^ input
           → SparseMatDiag Int y -- ^ output
liftLSdiag c f = (toSd c) ∘ f ∘ fromL

-- chunk size | a map implementation | working function
-- | primes | input matrix | output, matrix diagonal
-- lift1' :: Int → Map a b → (c → d)
--        → [n] → MatArr Int x → [x]
lift1' c mymap f ps x
  = let xs' = map (toL c) $ toResiduePrimes x ps
        xs = rnf xs' `pseq` xs'
        ys' = mymap (liftLSdiag c f) xs
        ys = rnf ys' `pseq` ys'
    in map (restoreFZ) $ transpose $ map fromSd ys
```

Figure B.11: *The implementation of* `lift1'`.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

BIBLIOGRAPHY

> Archimedes will be remembered when Aeschylus is forgotten, because languages die and mathematical ideas do not.
>
> G. H. Hardy, *A Mathematician's Apology*

S. Abdali and D. Wise. Experiments with quadtree representation of matrices. In *Symbolic and Algebraic Computation*, pages 96–108. Springer-Verlag, 1989. [130]

A. Abdallah, C. Jones, and J. Sanders. *Communicating sequential processes: the first 25 years*. LNCS 3525. Springer-Verlag, 2005. DOI http://dx.doi.org/10.1007/b136154. [3, 4, 182]

S. Aditya, L. A. Arvind, L. Augustsson, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Haskell Workshop*, 1995. [17]

A. R. Adl-Tabatabai and T. Shpeisman. Draft specification of transactional language constructs for C++. Transactional Memory Specification Drafting Group, Intel, IBM, and Sun, 2009. URL http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/. Retrieved 24.5.2011. [4, 182]

L. M. Adleman, C. Pomerance, and R. S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117(1):173–206, 1983. [62, 67, 181]

A. Agarwal and M. Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, pages 750–753. ACM, 2007. [3]

R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-D FFT. In *Proc. Supercomputing*, page 34. IEEE Computer Society, 1994. [108]

R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 2010. [129, 181]

G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33:125–141, 1990. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/83880.84528. [135]

G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, University of Michigan, 1985. URL http://hdl.handle.net/1721.1/6952. [178, 182]

G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997. DOI http://dx.doi.org/10.1017/S095679689700261X. [135]

M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics–Second Series*, 160(2): 781, 2004. [61]

K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *IEEE International Symposium on Cluster Computing and the Grid*, pages 156–163, 2003. DOI http://doi.ieeecomputersociety.org/10.1109/CCGRID.2003.1199364. [16]

S. Akioka and Y. Muraoka. Extended forecast of CPU and network load on computational grid. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 765–772. IEEE Computer Society, 2004. ISBN 0-7803-8430-X. DOI http://doi.ieeecomputersociety.org/10.1109/CCGrid.2004.1336711. [50]

M. Akon, A. Singh, D. Goswami, and H. Li. Extensible parallel architectural skeletons. In *HiPC 2005 — High Performance Computing*, pages 290–301. Springer-Verlag, 2005. [182]

E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, et al. MALLBA: A library of skeletons for combinatorial optimisation. In *Euro-Par 2002 Parallel Processing*, pages 63–73. Springer-Verlag, 2002. [182]

M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. [183]

M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, July 2007. URL http://wwwmath.uni-muenster.de/pvs/publikationen/papers/AltDiss.pdf. [20]

M. Alt and S. Gorlatch. Future-Based RMI: Optimizing compositions of remote method calls on the Grid. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003*, LNCS 2790, pages 682–693. Springer-Verlag, Aug. 2003. [20, 57, 182]

M. Alt and S. Gorlatch. Adapting java rmi for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2005. ISSN 0167-739X. DOI http://dx.doi.org/10.1016/j.future.2004.05.010. [20]

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS 1967 Spring Joint Computer Conference*, pages 483–485. ACM Press, 1967. [40, 41, 42]

D. Amos. Haskell for Math program, 2007. URL http://www.polynomino.f2s.com/david/haskell/codeindex.html. Retrieved 2.8.2007. [126, 183]

B. Amrhein, O. Gloor, and W. Küchlin. A case study of multi-threaded Gröbner basis completion. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, page 102ff. ACM Press, 1996. [181]

J. L. O. Arjona, G. Roberts, and G. Street. Architectural patterns for parallel programming. In *Proceedings of EuroPLoP*, 1998. [108]

J. Armstrong. *Programming Erlang*. The Pragmatic Programmers, LLC, 2007. [18, 135, 182]

E. A. Arnold. Modular algorithms for computing Gröbner bases. *Journal of Symbolic Computation*, 35 (4):403–419, 2003. [144, 151, 175, 179, 181]

D. Astapov, B. Pope, et al. Haskell-MPI: Haskell bindings to the MPI library, 2011. URL https://github.com/bjpop/haskell-mpi. Retrieved 30.3.2011. [17, 182]

B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995. DOI http://dx.doi.org/10.1002/cpe.4330070305. [182]

B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SKiE: A heterogeneous environment for HPC applications. *Parallel Computing*, 25(13-14):1827–1852, 1999a. [182]

B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. *Parallel Computing Technologies*, pages 760–760, 1999b. [57]

W. Backes and S. Wetzel. Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm. *Euro-Par 2009 Parallel Processing*, pages 960–973, 2009. [181]

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. ISSN 1094-3420. [108]

K. A. Baker and A. F. Pixley. Polynomial interpolation and the Chinese remainder theorem for algebraic systems. *Mathematische Zeitschrift*, 143:165–174, 1975. ISSN 0025-5874. DOI http://dx.doi.org/10.1007/BF01187059. [149]

K. Barclay and J. Savage. *Groovy Programming: An Introduction for Java Developers*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006. [10]

E. H. Bareiss. Sylvester's identity and multistep integer-preserving gaussian elimination. *Mathematics of Computation*, 22(103):565–578, 1968. ISSN 00255718. URL http://www.jstor.org/stable/2004533. [164]

H. P. Barendregt. *The lambda calculus: its syntax and semantics*. North Holland, 1984. ISBN 0444875085. [18]

C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI/GP*. Universite Bordeaux I, version 2.3.5 edition, 2010. URL http://pari.math.u-bordeaux.fr/dochtml/html.stable/. Retrieved 12.06.2011. [2, 180]

C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–12, 2002. ISSN 0747-7171. [2, 10, 177, 180, 183]

T. Becker, V. Weispfenning, and H. Kredel. *Gröbner bases: a computational approach to commutative algebra*. Springer-Verlag, 1993. [1, 194, 195, 196]

A. Benoit and M. Cole. eSkel – The Edinburgh Skeleton Library, 2002. URL http://homepages.inf.ed.ac.uk/abenoit1/eSkel/. [182]

A. Benoit, M. I. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *ICCS 2004 — The International Conference on Computational Science, Part III*, LNCS 3038, pages 299–306. Springer-Verlag, 2004. [50, 183]

A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eskel. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, LNCS 3648, pages 613–613. Springer-Verlag, 2005. DOI http://dx.doi.org/10.1007/11549468_83. [182]

D. J. Bernstein. Multidigit multiplication for mathematicians, 2001. URL http://cr.yp.to/papers/m3.pdf. Accepted to *Advances In Applied Mathematics*, but withdrawn by author. Retrieved online 24.08.2010. [100, 101]

J. Berthold. Towards a generalised runtime environment for parallel Haskells. In M. Bubak et al., editors, *Computational Science*, number 3 in LNCS 3038, page 297ff. ICCS'04—PAPP'04, Springer-Verlag, 2004. [18]

J. Berthold. *Explicit and Implicit Parallel Functional Programming — Concepts and Implementation*. PhD thesis, Philipps-Universität Marburg, 2008. [17, 19, 182]

J. Berthold and R. Loogen. The impact of dynamic channels on functional topology skeletons. *Parallel Processing Letters*, 2005. [15, 183]

J. Berthold and R. Loogen. Skeletons for recursively unfolding process topologies. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, ParCo 2005, Malaga, Spain*, NIC Series 33. Central Institute for Applied Mathematics, Jülich, Germany, 2006. [15, 183]

J. Berthold and R. Loogen. Parallel coordination made explicit in a functional setting. In Z. Horváth and V. Zsók, editors, *IFL 2006 — 18th Intl. Symposium on the Implementation of Functional Languages*, LNCS 4449, Budapest, Hungary, 2007a. Springer-Verlag. [17]

J. Berthold and R. Loogen. Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. In *ParCo '07. Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, 2007b. [35, 36, 38]

J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical master-worker skeletons. In D. S. Warren and P. Hudak, editors, *Practical Aspects of Declarative Languages (PADL'08)*, LNCS 4902, San Francisco (CA), USA, January 2008. Springer-Verlag. [16, 28, 124, 183]

J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. In K.-E. Großpitsch, A. Henkersdorf, S. Uhrig, T. Ungerer, and J. Hähner, editors, *Workshop on Many-Cores at ARCS '09: 22nd International Conference on Architecture of Computing Systems 2009*, pages 47–55. VDE-Verlag, 2009a. [15, 17, 18, 37, 92, 95, 100, 103, 112, 178, 180, 183]

J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. In V. Malyshkin, editor, *PaCT 2009: 10th International Conference on Parallel Computing Technologies*, LNCS 5698, pages 73–83. Springer-Verlag, 2009b. Extended version in [Berthold et al., 2009c]. [B, 15, 92, 95, 112, 114, 115, 116, 117, 118, 119, 123, 178, 180, 183]

J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel FFT with Eden skeletons. Technical Report bi2009-2, Philipps-Universität Marburg, Fachbereich 12 – Mathematik und Informatik, 2009c. [92, 96, 110, 112, 114, 115, 178, 180, 183, 214]

J. Berthold, M. Dieterle, and R. Loogen. Implementing parallel Google map-reduce in Eden. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, LNCS 5704, pages 990–1002. Springer-Verlag, 2009d. DOI http://dx.doi.org/10.1007/978-3-642-03869-3_91. [56, 57, 183]

A. Betten, R. Laue, and A. Wassermann. Discreta — a tool for constructing t-designs. *Lehrstuhl II für Mathematik, Universität Bayreuth*, 1997. [180]

I. Biehl, J. Buchmann, and T. Papanikolaou. LiDIA — a library for computational number theory. Technical report, SFB 124-C1, Fachbereich Informatik, Universität des Saarlandes, 1995. [180]

A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief announcement: actions in the twilight — concurrent irrevocable transactions and inconsistency repair. In *Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 71–72. ACM Press, 2010. ISBN 978-1-60558-888-9. DOI http://doi.acm.org/10.1145/1835698.1835714. [4]

B. J. Birch and H. P. F. Swinnerton-Dyer. Notes on elliptic curves I. *Journal für die reine und angewandte Mathematik*, 1963(212):7–25, 1963. ISSN 0075-4102. DOI http://www.reference-global.com/doi/abs/10.1515/crll.1963.212.7. [7]

R. S. Bird, G. Jones, and O. De Moor. More haste, less speed: lazy versus eager evaluation. *J. of Functional Programming*, 7(5):541–547, 1997. [183]

G. Birkhoff and C. R. de Boor. Piecewise polynomial interpolation and approximation. *Approximation of functions*, pages 164–190, 1965. [42]

G. E. Blelloch. NESL: A Nested Data-Parallel Language. (Version 3.1). Storming Media, 1995. [18]

G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996. URL http://www.cs.cmu.edu/~scandal/cacm.html. [3, 50]

G. E. Blelloch and J. Greiner. A parallel complexity model for functional languages. Technical report, Carnegie Mellon University, 1994. [18]

R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. ISSN 0004-5411. [28]

S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, pages 207–214, 1981. ISSN 0018-9340. [16]

A. Borodin and I. Munro. *The computational complexity of algebraic and numeric problems*. Theory of Computation 1. American Elsevier, 1975. [111, 126]

A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and GCD computations. *Information and control*, 52(3):241–256, 1982. DOI http://dx.doi.org/10.1016/S0019-9958(82)90766-5. [181]

I. Borosh and A. S. Fraenkel. Exact solutions of linear equations with rational coefficients by congruence techniques. *Mathematics of Computation*, 20(93):107–112, 1966. [174, 182]

W. Bosma and M. P. van der Hulst. *Primality proving with cyclotomy*. PhD thesis, University of Amsterdam, 1990. [88]

W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system i: The user language. *Journal of Symbolic Computation*, 24(3/4):235–265, 1997. [180]

G. H. Botorog and H. Kuchen. *Euro-Par'96 Parallel Processing*, volume 1123 of *LNCS*, chapter Efficient parallel programming with algorithmic skeletons, pages 718–731. Springer-Verlag, 1996. [182]

G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1-2):71–107, 1998. ISSN 0304-3975. [182]

A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda – a functional language with dependent types. *Theorem Proving in Higher Order Logics*, pages 73–78, 2009. [183]

S. Breitinger. *Design and Implementation of the Parallel Functional Language Eden*. PhD thesis, Philipps-Universität Marburg, 1998. URL http://archiv.ub.uni-marburg.de/diss/z1999/0142/. [182]

S. Breitinger and R. Loogen. Concurrency in Functional and Logic Programming. In *Fuji Intl. Workshop of Functional and Logic Programming*, 1995. [17]

S. Breitinger and R. Loogen. Channel Structures in the Parallel Functional Language Eden. In *Glasgow Workshop on Funct. Prg.*, 1997. [15, 19]

S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for parallel and concurrent programming. In D. Turner, editor, *Functional Programming*, Workshops in Computing, Glasgow, 1995. Springer-Verlag. [17, 37]

S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report 10, Philipps-University of Marburg, 1996. URL http://www.mathematik.uni-marburg.de/fb12/bfi/bfi10.ps. [17]

S. Breitinger, U. Klusik, and R. Loogen. An Implementation of Eden on Top of Concurrent Haskell. In W. Kluge, editor, *Implementation of Functional Languages '96*, LNCS 1268. Springer-Verlag, 1997. [17]

S. Breitinger, R. Loogen, and S. Priebe. Parallel programming with Haskell and MPI. In *Implementation of Functional Languages*, 1998. [17, 182]

J. Brenner and L. Cummings. The Hadamard maximum determinant problem. *The American Mathematical Monthly*, 79(6):626–630, 1972. [166]

R. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numerische Mathematik*, 16:145–156, 1970. ISSN 0029-599X. DOI http://dx.doi.org/10.1007/BF02308867. [129]

C. Brezinski. The long history of continued fractions and Padé approximants. In M. de Bruin and H. van Rossum, editors, *Padé Approximation and its Applications*, LNM 888, pages 1–27. Springer-Verlag, 1981. DOI http://dx.doi.org/10.1007/BFb0095574. [192]

M. Brickenstein. Slimgb: Gröbner bases with slim polynomials. *Revista Matemática Complutense*, 23: 453–466, 2010. ISSN 1139-1138. DOI http://dx.doi.org/10.1007/s13163-009-0020-0. [5, 9]

M. Bronstein. *Symbolic integration I: transcendental functions*, volume 1. Springer-Verlag, 2005. [183]

M. Bronstein, J. Davenport, A. Fortenbacher, et al. Axiom – the 30 year horizon, 2003. URL http://portal.axiom-developer.org/public/book2.pdf. [2, 180]

C. Brown and K. Hammond. Ever-decreasing circles: a skeleton for parallel orbit calculations in Eden. In *TFP'2010 — Draft Proceedings of the Symposium on Trends in Functional Programming*, 2010. [28, 57, 135, 178, 181, 183]

C. Brown, H.-W. Loidl, J. Berthold, and K. Hammond. Improving your CASH flow: The computer algebra shell. In *IFL 2010 — 22nd International Symposium on Implementation and Application of Functional Languages*, 2010. [181, 183]

N. C. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. *Communicating Process Architectures*, pages 67–84, 2008. WoTUG-31. [4, 58, 182]

N. C. C. Brown and P. H. Welch. An introduction to the Kent C++ CSP library. *Communicating Process Architectures*, 61:139–156, 2003. [4, 58]

T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer. Clean — a language for functional graph rewriting. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, LNCS 274, pages 364–384. Springer-Verlag, 1987. DOI http://dx.doi.org/10.1007/3-540-18317-5_20. [182]

B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. [1]

B. Buchberger and T. Jebelean. Parallel rational arithmetic for computer algebra systems: Motivating experiments. RISC Report Series 92-29, Research Institute for Symbolic Computation (RISC), Johannes Kepler University of Linz, 1992. [181]

B. Buchberger, E. V. Krishnamurthy, and F. Winkler. Gröbner bases, polynomial remainder sequences and decoding of multivariate codes. In *Recent Trends in Multidimensional Systems Theory*, pages 252–256. D. Reidel Publ. Comp., 1985. ISBN 90-277-1764-8, 1-4020-1623-9. [144, 174, 175]

B. Buchberger, G. Collins, H. Hong, J. Johnson, W. Krandick, R. Loos, and A. Neubacher. A SACLIB Primer. Technical Report 92–34, Johannes Kepler University, Linz, Austria, 1992. [180]

R. Bungers. *Über die Koeffizienten von Kreisteilungspolynomen*. PhD thesis, Georg-August-Universität Göttingen, 1934. [69]

P. Bürgisser, M. Clausen, and M. Shokrollahi. *Algebraic complexity theory*, volume 315. Springer-Verlag, 1997. [131]

L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969. [129]

D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991. ISSN 0001-5903. [91, 108, 109, 126]

A. Capani and G. Niesi. *CoCoA 3.0 User's Manual*. Dipartimento di Matematica, Università di Genova, Via Dodecaneso, 35, I-16146 Genova (Italy), 1995. [2, 180]

G. Cesari and R. Maeder. Parallel 3-primes FFT algorithm. In J. Calmet and C. Limongelli, editors, *Design and Implementation of Symbolic Computation Systems*, LNCS 1128, pages 174–182. Springer-Verlag, 1996a. [126, 181]

G. Cesari and R. Maeder. Performance analysis of the parallel Karatsuba multiplication algorithm for distributed memory architectures. *Journal of Symbolic Computation*, 21(4-6):467–473, 1996b. DOI 10.1006/jsco.1996.0026. [95, 181]

J.-L. Chabert, editor. *A history of algorithms: from the pebble to the microchip*. Springer-Verlag, 1999. [7, 92, 93, 94, 192]

M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP '07*, pages 10–18. ACM Press, 2007. [3, 18, 182]

B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. URL http://hpc.sagepub.com/content/21/3/291.abstract. [3]

J. M. Chambers and T. J. Hastie. *Statistical models in S*. CRC Press, 1991. [42]

P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM Press, 2005. ISBN 1-59593-031-0. DOI http://doi.acm.org/10.1145/1094811.1094852. [3]

S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998. [47]

E. W. Cheney and D. R. Kincaid. *Numerical mathematics and computing*. Brooks/Cole, 2007. [163]

R. C. C. Cheung, A. Brown, W. Luk, and P. Cheung. A scalable hardware architecture for prime number validation. In *FPT 2004 — Proceedings of IEEE International Conference on Field Programmable Technology*, pages 177–184, 2004. [64, 89]

A. Church. The calculi of lambda-conversion. *Annals of Mathematics Studies*, 1941. [18]

P. Ciechanowicz and H. Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *12th IEEE International Conference on High Performance Computing and Communications*, pages 108–113. IEEE, 2010. [57]

W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979. ISSN 01621459. URL http://www.jstor.org/stable/2286407. [42]

W. S. Cleveland and S. J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988. ISSN 01621459. URL http://www.jstor.org/stable/2289282. [42]

CoCoA, 2009. CoCoA: a system for doing computations in commutative algebra, 2009. URL http://cocoa.dima.unige.it. Retrieved 10.6.2011. [2, 180]

H. Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer-Verlag, forth edition, 2000. [9, 62, 63, 68, 69, 70, 71, 72, 73, 74, 75, 88, 149, 151]

H. Cohen. *Number Theory: Tools and Diophantine Equations*, volume 1 of *Graduate Texts in Mathematics 239*. Springer-Verlag, 2007. ISBN 978-0-387-49922-2. [68, 69, 70]

H. Cohen and H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42 (165):297–330, 1984. [62, 67, 72, 73, 182]

H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Annual IEEE Symposium on Foundations of Computer Science*, pages 379–388. IEEE Computer Society, 2005. ISBN 0-7695-2468-0. DOI http://doi.ieeecomputersociety.org/10.1109/SFCS.2005.39. [129]

M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989. [3, 22, 50, 182]

M. I. Cole and Y. Hayashi. Static performance prediction of skeletal programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002. [50, 183]

G. E. Collins and M. J. Encarnación. Efficient rational number reconstruction. *Journal of Symbolic Computation*, 20(3):287–297, 1995. ISSN 0747-7171. DOI http://dx.doi.org/10.1006/jsco.1995.1051. [147]

S. Cook. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969. [91, 103]

J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965. [108, 110, 181]

D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. In *SFCS'81 — 22nd Annual Symposium on Foundations of Computer Science*, pages 82–90. IEEE, 1982. [129, 131]

D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990. ISSN 0747-7171. [131]

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001. [99, 102]

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP 2007 — Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326. ACM Press, 2007. [76, 127]

R. Crandall, K. Dilcher, and C. Pomerance. A search for Wieferich and Wilson primes. *Mathematics of Computation*, 66(217):pp. 433–449, 1997. ISSN 00255718. URL http://www.jstor.org/stable/2153665. [75]

R. Crandall, E. Jones, J. Klivington, and D. Kramer. Gigaelement FFTs on Apple G5 clusters. Technical report, Advanced Computation Group, Apple Computer, August 2004. [108, 181]

R. E. Crandall and C. Pomerance. *Prime numbers: a computational perspective*. Springer-Verlag, second edition, 2005. ISBN 0387252827. [70, 88]

C. G. Cullen. *Matrices and Linear Transformations*. Dover, Mineola, New York, 1990. Republication of second edition by Addison–Wesley Publishing Company, 1972. [164, 165]

D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):12, 1993. [50]

M. Daberkow, C. Fieker, J. Klüners, M. Pohst, K. Roegner, M. Schörnig, and K. Wildanger. KANT V4. *Journal of Symbolic Computation*, 24(3-4):267–283, 1997. ISSN 0747-7171. DOI http://dx.doi.org/10.1006/jsco.1996.0126. Special Issue on Computational Algebra and Number Theory. [2, 180]

L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, pages 46–55, 1998. [182]

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM Press, 1982. ISBN 0-89791-065-6. DOI http://doi.acm.org/10.1145/582153.582176. [12]

M. Danelutto and M. Stigliani. SKElib: Parallel programming with skeletons in C. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS 1900, pages 1175–1184. Springer-Verlag, 2000. DOI http://dx.doi.org/10.1007/3-540-44520-X_166. [182]

M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P^3L. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97*, LNCS 1300, pages 619–628. Springer-Verlag, 1997. [182]

J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *PARLE'93 — Parallel Architectures and Languages Europe*, pages 146–160. Springer-Verlag, 1993. [182]

J. H. Davenport. Computer algebra – past, present and future. *Euromath Bulletin*, 1(2):25–44, 1994. [2]

M. de Kruijf and K. Sankaralingam. MapReduce for the Cell Broadband Engine architecture. *IBM Journal of Research and Development*, 53(5):10–1, 2009. [57]

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI 2004 — 6th Symposium on Operating Systems Design And Implementation*, volume 6, page 10, Berkeley, CA, USA, 2004. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1251254.1251264. [56]

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/1327452.1327492. [56]

J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53:72–77, 2010. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/1629175.1629198. [56]

W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-3 — A computer algebra system for polynomial computations, 2011. URL http://www.singular.uni-kl.de. Retrieved 10.6.2011. [2, 180]

J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *Siam Journal On Matrix Analysis And Applications*, 1999. [173, 181]

J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 2:111–197, 2008. [181]

M. Díaz, B. Rubio, E. Soler, and J. M. Troya. SBASCO: Skeleton-based scientific components. In *EuroMicro Conference on Parallel, Distributed, and Network-Based Processing*, page 318. IEEE Computer Society, 2004. DOI http://doi.ieeecomputersociety.org/10.1109/EMPDP.2004.1271461. [182]

M. Dieterle. *Parallele funktionale Implementierung von Master-Worker-Skeletten*. Diplomarbeit, Philipps-Universität Marburg, 2007. [28, 57]

M. Dieterle, J. Berthold, and R. Loogen. A skeleton for distributed work pools in Eden. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, LNCS 6009, pages 337–353. Springer-Verlag, 2010a. DOI http://dx.doi.org/10.1007/978-3-642-12251-4_24. [16, 28, 183]

M. Dieterle, T. Horstmeyer, and R. Loogen. Skeleton composition using remote data. In M. Carro and R. Peña, editors, *PADL 2010: 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *LNCS*, pages 73–87. Springer-Verlag, 2010b. [15, 20, 28, 37, 57, 60, 97, 116, 175, 179, 182, 183]

E. W. Dijkstra. Hamming's exercise in SASL. EWD792, 1981. [43]

A. Discolo, T. Harris, S. Marlow, S. Jones, and S. Singh. Lock free data structures using STM in Haskell. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming*, LNCS 3945, pages 65–80. Springer-Verlag, 2006. DOI http://dx.doi.org/10.1007/11737414_6. [182]

P. Dmitruk, L. P. Wang, W. H. Matthaeus, R. Zhang, and D. Seckel. Scalable parallel fft for spectral simulations on a beowulf cluster. *Parallel Computing*, 27(14), 2001. [108, 181]

K. Doets and J. van Eijck. The haskell road to logic, math and programming. 2004. [8]

P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, 1990. [108, 110]

J. Dunnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *SCC 2004 — Proceedings of the IEEE International Conference on Services Computing*, pages 288–294. IEEE, 2004. ISBN 0769522254. [182]

H.-D. Ebbinghaus, H. Hermes, F. Hirzebruch, M. Koecher, K. Mainzer, J. Neukirch, A. Prestel, R. Remmert, and K. Lamotke. *Zahlen*. Springer-Verlag, third edition, 1992. ISBN 3540556540. [190, 191]

G. L. Ebert. Some comments on the modular approach to Gröbner-bases. *SIGSAM Bull.*, 17(2):28–32, 1983. ISSN 0163-5824. [144, 151, 175, 181]

Eden Skeletons, 2011. Eden skeleton library. Software package, 2011. URL http://www.mathematik.uni-marburg.de/~eden/doc/edenskel-1.0.1.1/index.html. Retrieved 15.6.2011. [15, 22, 37]

J. Edmonds. Systems of distinct representatives and linear algebra. *J. Res. Natl. Bur. Stand., Sect. B*, 71: 241–245, 1967. [164]

D. Eisenbud. *Computations in algebraic geometry with Macaulay 2*. Springer-Verlag, 2002. [2, 180, 181]

H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2005. ISBN 0471467405. [4, 40]

I. Z. Emiris and V. Y. Pan. Applications of FFT and structured matrices. In M. J. Atallah and M. Blanton, editors, *Algorithms and theory of computation handbook: general concepts and techniques*, volume 1. Chapman & Hall/CRC, 2010. ISBN 1584888229. [93, 108, 109, 126]

J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129. ACM Press, 2011. ISBN 978-1-4503-0860-1. DOI http://doi.acm.org/10.1145/2034675.2034690. [18, 135, 182]

H. W. Eves. *An introduction to the history of mathematics*. Saunders College Publishers, 1969. [7, 92, 192, 193]

J. F. Ferreira, J. L. Sobral, and A. J. Proenca. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, pages 301–304. IEEE Computer Society, 2006. ISBN 0769525857. [183]

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008. [8, 10]

M. J. Flynn. Very high-speed computing systems. *Proc IEEE*, 54(12):1901–1909, 1966. [16]

G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer methods for mathematical computations*. Prentice Hall Professional Technical Reference, 1977. [42]

S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th annual ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978. [50]

I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison–Wesley, 1995. [3, 16, 28]

D. Fowler. *The mathematics of Plato's academy*. Oxford Clarendon, 1999. [192]

A. S. Fraenkel. New proof of the generalized Chinese remainder theorem. *Proceedings of the American Mathematical Society*, 14(5):790–791, 1963. ISSN 00029939. URL http://www.jstor.org/stable/2034995. [149]

A. S. Fraenkel. Systems of numeration. *The American Mathematical Monthly*, 92(2):105–114, 1985. ISSN 00029890. URL http://www.jstor.org/stable/2322638. [150]

M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc IEEE*, 93(2), 2005. URL http://www.fftw.org/fftw-paper-ieee.pdf. [108]

B. Fuchssteiner. *MuPAD*. Birkhäuser, 1994. [180, 181]

B. Fulgham and I. Gouy. The computer language benchmarks game, 2011. URL http://shootout.alioth.debian.org. Retrieved 16.5.2011. [11]

M. Fürer. Faster integer multiplication. In *Proceedings of the 39th annual ACM Symposium on Theory of Computing*, STOC '07, pages 57–66. ACM Press, 2007. ISBN 978-1-59593-631-8. DOI http://doi.acm.org/10.1145/1250790.1250800. [91]

L. A. Galán, C. Pareja, and R. Peña. Functional skeletons generate process topologies in Eden. In *PLILP'96 — Programming Languages: Implementations, Logics, and Programs*, LNCS 1140, pages 289–303. Springer-Verlag, 1996. [22, 183]

GAP, 2008. *GAP – Groups, Algorithms, and Programming, Version 4.4.10*. The GAP Group, 2008. URL http://www.gap-system.org. [2, 180]

T. Gautier and N. Mannhart. Parallelism in aldor—the communication library πit for parallel, distributed computation. *Euro-Par'99 Parallel Processing*, pages 1466–1475, 1999. [12]

K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for computer algebra*. Kluwer, 1992. [126]

D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. ISSN 0018-9162. [2]

M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6), 2010. [35]

W. M. Gentleman. Some complexity results for matrix computations on parallel processors. *Journal of the ACM*, 25(1):112–115, 1978. ISSN 0004-5411. DOI http://doi.acm.org/10.1145/322047.322057. [129]

W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. ACM Press, 1966. DOI http://doi.acm.org/10.1145/1464291.1464352. [91, 108]

GiNaC. GiNaC program. URL http://www.ginac.de. [2]

O. Gloor and S. Muller. Parcan — a parallel computer algebra nucleus, 1997. [2, 181]

G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996. [128, 143, 163]

H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010. ISSN 1097-024X. DOI 10.1002/spe.1026. [182]

S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, LNCS 1140, pages 274–288. Springer-Verlag, 1996. DOI http://dx.doi.org/10.1007/3-540-61756-6_91. [108, 114]

S. Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *Journal of Supercomputing*, 12(1-2):85–97, 1998a. [108, 114, 127, 178, 181]

S. Gorlatch. *Abstraction and Performance in the Design of Parallel Programs*. Habilitation thesis, Universität Passau, 1998b. URL http://wwwmath.uni-muenster.de/pvs/publikationen/papers/Gor98a.ps.gz. MIP-9802. Retrieved 4.6.2011. [3]

S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4), 1998. [92, 108, 114, 181]

D. Goswami, A. Singh, and B. R. Preiss. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669–695, 2002. ISSN 0743-7315. [182]

J. Grabmeier, E. Kaltofen, and V. Weispfenning. *Computer algebra handbook: foundations, applications, systems*. Springer-Verlag, 2003. ISBN 3540654666. [2, 7, 10, 174, 180, 181, 183]

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*. Addison–Wesley, 1994. [192]

A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Concurrency*, 1(3):12–21, 1993. ISSN 1063-6552. DOI http://doi.ieeecomputersociety.org/10.1109/88.242438. [40]

A. Y. Grama, V. Kumar, A. Gupta, and G. Karypis. *Introduction to parallel computing*. Addison–Wesley, 2003. [3, 16, 28, 40, 41, 87, 92, 108, 114, 127, 182]

T. Granlund and A. B. Swox. The GNU multiple precision arithmetic library, 2011. URL http://gmplib.org/. Retrieved 30.05.2011. [8]

B. Grayson and R. Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996. DOI http://dx.doi.org/10.1142/S0129626496000029. [129]

D. R. Grayson and M. E. Stillman. Macaulay2, a software system for research in algebraic geometry, 2010. URL http://www.math.uiuc.edu/Macaulay2/. Retrieved 10.6.2011. [2, 180]

R. T. Gregory. Error-free computation with rational numbers. *BIT Numerical Mathematics*, 21(2):194–202, 1981. [146, 147, 153, 156, 174]

R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error–Free Computation*. Springer-Verlag, 1984. ISBN 0-387-90967-2, 3-540-90967-2. [144, 146, 147, 148, 150, 151, 152, 153, 156, 157, 158, 174, 182]

C. Grelck and S.-B. Scholz. Towards an efficient functional implementation of the NAS benchmark FT. In *PaCT*, LNCS 2763, pages 230–235. Springer-Verlag, 2003. [108]

G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 2.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2001. URL http://www.singular.uni-kl.de. Retrieved 10.6.2011. [2, 180]

J. H. Griesmer and R. D. Jenks. SCRATCHPAD/1: An interactive facility for symbolic mathematics. In *Proceedings of the 2nd ACM Symposium on Symbolic and Algebraic Manipulation*, pages 42–58. ACM Press, 1971. [180]

L. C. Grove. *Algebra*. Dover, 2004. Republication of an edition by Academic Press, 1983. [99, 145, 146, 189, 190, 191, 194, 195, 196, 197]

H. Gupta and P. Sadayappan. Communication-efficient matrix multiplication on hypercubes. *Parallel Computing*, 22(1):75–99, 1996. ISSN 0167-8191. DOI http://dx.doi.org/10.1016/0167-8191(95)00058-5. [129, 181]

S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, 4(4): 477–488, 1994. [108]

J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988. [40]

J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988. [40, 105]

R. K. Guy. *Unsolved problems in number theory*. Springer-Verlag, 2004. ISBN 0387208607. [164]

J. S. Hadamard. Résolution d'une question relative aux déterminants. *Bulletin des Sciences Mathématiques*, 17(1):240–246, 1893. [166]

Hadoop, 2011. Apache hadoop. Website, 2011. URL http://hadoop.apache.org/. Retrieved 9.7.2011. [57]

T. Hahnel. *The Rabin–Miller Prime Number Test on Systola 1024 on the Background of Cryptography*. Master's thesis, University of Karlsruhe, 1998. [64, 89]

B. Haible and R. Kreckel. CLN, a class library for numbers manual, 2008. URL http://www.ginac.de/CLN/cln.ps. Retrieved 18.05.2011. [8]

P. Haller and M. Odersky. Event-based programming without inversion of control. *Modular Programming Languages*, pages 4–22, 2006. [178, 182]

P. Haller and M. Odersky. Actors that unify threads and events. In A. Murphy and J. Vitek, editors, *Coordination Models and Languages*, LNCS 4467, pages 171–190. Springer Berlin / Heidelberg, 2007. DOI http://dx.doi.org/10.1007/978-3-540-72794-1_10. [135]

P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009. ISSN 0304-3975. DOI http://dx.doi.org/10.1016/j.tcs.2008.09.019. Distributed Computing Techniques. [3, 135, 178, 182]

R. H. Halstead Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7:501–538, October 1985. ISSN 0164-0925. DOI http://doi.acm.org/10.1145/4472.4478. [20]

M. Hamdi and C.-K. Lee. Dynamic load balancing of data parallel applications on a distributed network. In *ICS 1995 — Proceedings of the 9th international conference on Supercomputing*, pages 170–179. ACM, 1995. ISBN 0-89791-728-6. DOI http://doi.acm.org/10.1145/224538.224557. [16]

J. Hammes, S. Sur, and A. P. W. Böhm. On the effectiveness of functional language features: NAS benchmark FT. *J. Funct. Program.*, 7(1):103–123, 1997. [108, 181]

K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003. [183]

K. Hammond, D. Petcu, P. Trinder, A. A. Zain, S. Linton, and G. Michaelson. Project paper: the SCIEnce joint research activity symbolic computing on the grid. In *TFP'07: International Symposium on Trends in Functional Programming*. Intellect, 2007. Draft Proceedings. [2, 181]

G. H. Hardy. *A mathematician's apology*. Cambridge University Press, 1992. ISBN 978-0-521-42706-7. Reprint. [145, 211]

G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford Clarendon, fourth edition, 1975. [62, 71]

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Communications of the ACM*, 51:91–100, 2008. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/1378704.1378725. [4, 182]

T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010. URL http://www.morganclaypool.com/doi/abs/10.2200/S00272ED1V01Y201006CAC011. [3, 182]

L. Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49(5):475–482, 2007. [11]

B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269. ACM Press, 2008. ISBN 978-1-60558-282-5. DOI http://doi.acm.org/10.1145/1454115.1454152. [57]

A. C. Hearn. Computation of algebraic properties of elementary particle reactions using a digital computer. *Communications of the ACM*, 9:573–577, 1966. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/365758.365766. [180]

A. C. Hearn. *REDUCE — User's Manual Version 3.8*, 2004. reduce@rand.org. [2, 180]

M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the FFT. *IEEE Acoustics, Speech, and Signal Processing Magazine*, 1:14–21, 1984. [108]

B. Hendrickson, David, and E. Womble. The torus–wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput*, 15:1201–1226, 1994. [129]

K. Hensel. *Theorie der algebraischen Zahlen*. BG Teubner, 1908. [183]

M. Herlihy. Transactional memory today. In T. Janowski and H. Mohanty, editors, *Distributed Computing and Internet Technology*, volume LNCS 5966, pages 1–12. Springer-Verlag, 2010. DOI http://dx.doi.org/10.1007/978-3-642-11659-9_1. stm. [3]

M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300. ACM Press, 1993. ISBN 0-8186-3810-9. DOI http://doi.acm.org/10.1145/165123.165164. [3, 182]

C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Universität Passau, 2000. ISBN 3-89722-556-5. [32, 33, 37, 94, 103, 183]

C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. Morgan Kaufmann, 1973. URL http://portal.acm.org/citation.cfm?id=1624775.1624804. [3, 92, 135, 178, 182]

A. J. G. Hey. Experiments in MIMD parallelism. *Future Generation Computer Systems*, 6(3):185–196, 1990. ISSN 0167-739X. DOI 10.1016/0167-739X(90)90018-9. [26]

M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language eden. *Parallel Processing Letters*, 12(2):211–228, 2002. [17]

M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel haskell dialects. In A. Ohori, editor, *Programming Languages and Systems*, LNCS 2895, pages 303–321. Springer-Verlag, 2003. DOI http://dx.doi.org/10.1007/978-3-540-40018-9_20. [17]

M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 2008. [2, 40]

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. URL http://www.jstor.org/stable/1995158. [12]

R. Hinze. Functional pearl: streams and unique fixed points. In *ICFP '08 — Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 189–200, 2008. [12, 179, 183]

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/359576.359585. [3, 4, 58, 182]

T. Horstmeyer and R. Loogen. Graph-based comminication in Eden. In Z. Horváth, V. Zsók, P. Achten, and P. Koopman, editors, *TFP'2010 — Trends in Fuctional Programming*, volume 10, pages 1–16. Intellect, 2010. [15]

C.-H. Huang. A fully parallel mixed-radix conversion algorithm for residue number applications. *IEEE Transactions on Computers*, 100(4):398–402, 2006. ISSN 0018-9340. [150]

C.-H. Huang, J. R. Johnson, and R. W. Johnson. A tensor product formulation of strassen's matrix multiplication algorithm. *Applied Mathematics Letters*, 3(3):67–71, 1990. ISSN 0893-9659. DOI http://dx.doi.org/10.1016/0893-9659(90)90139-3. [129]

X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *Journal of complexity*, 14:257–299, 1998. ISSN 0885-064X. [129, 131]

P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. awk vs. …: An experiment in software prototyping productivity, 1994. URL http://haskell.org/papers/NSWC/jfp.ps. [11]

S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 32–32, 1996. [129]

G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9:355–372, July 1999. ISSN 0956-7968. DOI 10.1017/S0956796899003500. [56]

G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 978-0521692694. [8]

N. Idrees, G. Pfister, and S. Steidel. Parallelization of modular algorithms. *Journal of Symbolic Computation*, 46(6):672 – 684, 2011. ISSN 0747-7171. DOI http://dx.doi.org/10.1016/j.jsc.2011.01.003. [144, 151, 175, 179, 181]

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *J. of computational and graphical statistics*, pages 299–314, 1996. [10]

E. Ipek, B. de Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, LNCS 3648, pages 627–628. Springer-Verlag, 2005. DOI http://dx.doi.org/10.1007/11549468_24. [50]

ISO/IEC 9899:1999. Programming languages – C, 1999. [11, 39, 52, 188]

P. A. Jackson, C. P. Chan, J. E. Scalera, C. M. Rader, and M. M. Vai. A systolic FFT architecture for real time FPGA systems. In *Proceedings of the High Performance Embedded Computing Conference*, 2004. [108]

V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform grids. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 123–134. IEEE, 2010. [28]

T. Jebelean. Using the parallel Karatsuba algorithm for long integer multiplication and division. In *Euro-Par'97 Parallel Processing*, pages 1169–1172. Springer-Verlag, 1997. [181]

R. D. Jenks and R. S. Sutor. *Axiom — The Scientific Computation System*. Springer-Verlag, 1992. [180]

G. Jones and G. Jones. *Programming in occam*. Prentice-Hall International, 1987. [4]

J. P. Jones, D. Sato, H. Wada, and D. Wiens. Diophantine representation of the set of prime numbers. *American Mathematical Monthly*, pages 449–464, 1976. [61]

H. G. Kahrimanian. *Analytical differentiation by a digital computer*. Master's thesis, Temple University, 1953. [7]

N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, pages 47–54. IEEE, 1999. [50]

A. Karatsuba. The complexity of computations. *Proc. Steklov Inst. Math*, 211:169–183, 1995. [100, 101]

A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics–Doklady 7, 595–596, 1963. [100, 101, 181]

J. Karczmarczuk. Scientific computation and functional programming. *Computing in Science & Engineering*, 1(3):64–72, 1999. [183]

J. Karczmarczuk. Lazy processing and optimization of discrete sequences. In *Proceedings of the JFLA*, 2000. [12]

A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Comm. ACM*, 33(5):539–543, 1990. [40, 48, 119, 177]

M. Kesseler. Constructing skeletons in Clean: the bare bones. In *HPFC'95 — Proceedings of High Performance Functional Computing*, pages 182–192, 1995. [182]

A. Y. Khinchin. *Continued Fractions*. Dover Publications, Inc., 1997. Translation originally published by University of Chicago Press, 1964. Russian edition in year 1960. [9]

S. Khirevich. LBM and RWPT timings on Jülich Blue Gene. Private communication, May 2010. [44, 46, 47]

S. Khirevich and A. Daneyko. Simulation of fluid flow and mass transport at extreme scale. In B. Mohr and W. Frings, editors, *Jülich Blue Gene/P Extreme Scaling Workshop 2010*. Jülich Supercomputing Centre, 2010. [46]

S. Khirevich, A. Höltzel, S. Ehlert, A. Seidel-Morgenstern, and U. Tallarek. Large-scale simulation of flow and transport in reconstructed hplc-microchip packings. *Analytical Chemistry*, 81(12): 4937–4945, 2009a. DOI 10.1021/ac900631d. [46]

S. Khirevich, A. Höltzel, A. Seidel-Morgenstern, and U. Tallarek. Time and length scales of eddy dispersion in chromatographic beds. *Analytical Chemistry*, 81(16):7057–7066, 2009b. DOI 10.1021/ac901187d. [46]

U. Klusik, R. Loogen, and S. Priebe. Controlling parallelism and data distribution in Eden. In *SFP 2000 — Scottish Functional Programming Workshop*, volume 2 of *TFP*, pages 53–64. Intellect, 2000. [97]

U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden — low-effort parallel programming. In *IFL 2000 — Implementation of Functional Languages*, LNCS 2011, pages 71–88. Springer-Verlag, 2001. [17, 22, 24, 26, 27, 183]

T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112. ACM Press, 1986. ISBN 0-89791-200-4. DOI http://doi.acm.org/10.1145/319838.319854. [3, 182]

D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison–Wesley, third edition, 1998. [63, 93, 108, 125, 149, 151, 156, 192]

N. Koblitz. *p-adic Numbers, p-adic Analysis, and Zeta-Functions*. Springer-Verlag, 1984. ISBN 0387960171. [174, 183]

N. Koblitz. *A course in Number Theory and Cryptography*. Springer-Verlag, 1994. ISBN 0387942939. [63]

P. Kornerup and R. T. Gregory. Mapping integers and Hensel codes onto Farey fractions. *BIT Numerical Mathematics*, 23(1):9–20, 1983. [147, 153, 174]

P. Kornerup and D. W. Matula. *Finite Precision Number Systems and Arithmetic*. Cambridge Univeristy Press, 2010. ISBN 0521761352. [144, 174]

H. Koy and C. Schnorr. Segment LLL-reduction with floating point orthogonalization. *Cryptography and Lattices*, pages 81–96, 2001. [9]

H. Kredel. Distributed hybrid Gröbner bases computation. *Complex, Intelligent and Software Intensive Systems, International Conference*, pages 561–567, 2010. [181]

H. Kuchen. The Münster skeleton library Muesli, 2011. URL http://www.wi.uni-muenster.de/pi/forschung/Skeletons/index.html. Retrieved 20.7.2011. [182]

H. Kuchen and J. Striegnitz. Features from functional programming for a C++ skeleton library. *Concurrency and Computation: Practice and Experience*, 17(7-8):739–756, 2005. ISSN 1532-0634. [182]

W. Kuechlin. PARSAC-2: Parallel Computer Algebra on the Desk Top. In *Computer Algebra in Science and Engineering*. World Scientific Publishing Company, 1995. [2, 181]

V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994. [40]

Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999a. ISSN 0743-7315. DOI http://dx.doi.org/10.1006/jpdc.1999.1578. [16]

Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31:406–471, December 1999b. ISSN 0360-0300. DOI http://doi.acm.org/10.1145/344588.344618. [16]

J. Laderman, V. Pan, and X.-H. Sha. On practical algorithms for accelerated matrix multiplication. *Linear Algebra and its Applications*, 162-164:557–588, 1992. ISSN 0024-3795. DOI 10.1016/0024-3795(92)90393-O. [129, 131]

R. Lämmel. Google's mapreduce programming model — revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008. ISSN 0167-6423. DOI 10.1016/j.scico.2007.07.001. [56]

S. Lang. *Linear Algebra*. Springer-Verlag, third edition, 1987. [164, 165]

S. Lang. *Algebra*. Graduate Texts in Mathematics 221. Springer-Verlag, third edition, 2002. ISBN 038795385X. [68, 70, 145, 189]

D. H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, pages 227–233, 1938. [9]

E. Lehmer. On the magnitude of the coefficients of the cyclotomic polynomial. *Bull. Amer. Math. Soc*, 42:389–392, 1936. [69]

H. W. Lenstra, Jr. Divisors in residue classes. *Mathematics of Computation*, 42(165):331–340, 1984. URL http://www.jstor.org/stable/2007582. [72, 88]

V. Levandovskyy. *Non–commutative computer algebra for polynomial algebras: Gröbner bases, applications and implementation*. PhD thesis, Universität Kaiserslautern, 2005. [10]

M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 289–296. IEEE, 2010. [183]

K. Li, Y. Pan, and S. Q. Zheng. Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):705–720, 2002. ISSN 1045-9219. [129, 131, 181]

C. Limongelli. On an efficient algorithm for big rational number computations by parallel p-adics. *Journal of Symbolic Computation*, 15:181–181, 1993. ISSN 0747-7171. [174]

O. Lobachev. *Multimodulare Arithmetik*. Diplomarbeit, Justis-Liebig-Universität Gießen, Mar. 2007. [144, 147, 153, 155, 157, 174, 175, 179, 182]

O. Lobachev. *Multimodulare Arithmetik*. Logos, Berlin, Germany, 2011a. ISBN 978-3-8325-2881-2. [144, 147, 153, 155, 164, 174, 175, 179, 180, 182]

O. Lobachev. On an implementation of parallel computation skeletons with premature termination property. *International Journal of High Performance Computing and Networking*, 2011b. Submitted to the Special Issue on Multicore Programming. [180]

O. Lobachev and R. Loogen. Towards an implementation of a computer algebra system in a functional language. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Intelligent Computer Mathematics*, LNAI 5144, pages 141–154. AISC 2008: 9th International Conference on Artificial Intelligence and Symbolic Computation, Springer-Verlag, 2008. [B, 9, 11, 14, 92, 125, 126, 177, 179]

O. Lobachev and R. Loogen. Implementing data parallel rational multiple-residue arithmetic in Eden. In V. P. Gerdt, W. Koepf, E. W. Mayr, and E. H. Vorozhtsov, editors, *CASC'2010: Computer Algebra in Scientific Computing*, LNCS 6244, pages 178–193. Springer-Verlag, 2010a. Extended and revised version in [Lobachev and Loogen, 2010b]. [B, 144, 169, 173, 174, 175, 179, 180]

O. Lobachev and R. Loogen. Implementing data parallel rational multiple-residue arithmetic in Eden. Technical Report bi2010-3, Fachbereich Mathematik und Informatik der Philipps-Universität Marburg, 2010b. [144, 155, 169, 173, 174, 175, 179, 180, 228]

O. Lobachev and R. Loogen. Estimating parallel performance, a skeleton-based approach. In *Proceedings of 4th International Workshop on High-level Parallel Programming and Applications*, pages 25–34. ACM Press, 2010c. [B, 39, 64, 65, 106, 168, 178, 180, 183]

O. Lobachev, M. Guthe, and R. Loogen. Estimating parallel performance. *International Journal of Parallel Programming*, 2011. Submitted. [39, 180]

H.-W. Loidl. Linsolv: A case study in strategic parallelism. In *Glasgow Workshop on Functional Programming*, 1997. [151, 174, 179]

H.-W. Loidl, F. Rubio Diez, N. Scaife, et al. Comparing parallel functional languages: Programming and performance. *HOSC*, 16(3), 2003. [18]

R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 71–88. Springer-Verlag, 2003. ISBN 1-85233-506-8. [15, 16, 22, 27, 33, 34, 37, 50, 56, 97, 103, 135, 183]

R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005. [2, 5, 15, 16, 17, 19, 37, 177, 179, 182]

Q. Luo and J. B. Drake. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *SAC '95: Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 221–226. ACM Press, 1995. ISBN 0-89791-658-1. DOI http://doi.acm.org/10.1145/315891.315965. [129, 181]

R. G. Lyons. *Understanding digital signal processing*. Prentice Hall, 2004. ISBN 0131089897. [107, 108]

MacTutor. History of mathematics archive, 2011. URL http://www-history.mcs.st-and.ac.uk/. Retrieved 25.03.2011. [7, 145, 192, 193]

U. Manber. *Introduction to algorithms: a creative approach*. Addison–Wesley, 1989. ISBN 0201120372. [102]

S. Marlow, S. Peyton-Jones, and S. Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 44(9):65–78, 2009. [3, 17, 37, 182]

S. Marlow, P. Maier, H. W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Proceedings of the 3rd ACM Symposium on Haskell*, Haskell '10, pages 91–102. ACM Press, 2010. [21]

S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82. ACM Press, 2011. ISBN 978-1-4503-0860-1. DOI http://doi.acm.org/10.1145/2034675.2034685. [3, 18, 182, 183]

P. Marti-Puig, R. R. Bolaño, and V. P. Baradad. Radix-R FFT and IFFT factorizations for parallel implementation. In *International Symposium on Distributed Computing and Artificial Intelligence*, DCAIA '08, pages 152–160. Springer-Verlag, 2008. [181]

W. A. Martin and R. J. Fateman. The MACSYMA system. In *Proceedings of the 2nd ACM Symposium on Symbolic And Algebraic Manipulation*, SYMSAC '71, pages 59–75. ACM Press, 1971. DOI http://doi.acm.org/10.1145/800204.806267. [2, 180]

R. Martínez and R. Peña. Building an interface between Eden and Maple: A way of parallelizing computer algebra algorithms. In P. W. Trinder, G. Michaelson, and R. Peña, editors, *IFL 2003 — 15th International Workshop on Implementation of Functional Languages, Revised Papers*, LNCS 3145, pages 135–151. Springer-Verlag, 2004. [173, 181]

Yu. V. Matiyasevich. Diophantine representation of the set of prime numbers. In *Dokl. Akad. Nauk SSSR*, volume 196, pages 770–773, 1971. [61]

Yu. V. Matiyasevich. Primes are nonnegative values of a polynomial in 10 variables. *Journal of Mathematical Sciences*, 15(1):33–44, 1981. DOI http://dx.doi.org/10.1007/BF01404106. Translation of 1977 publication in *Theoretical application of methods of mathematical logic. Part II*, Zap. Nauchn. Sem. LOMI. URL http://www.ams.org/mathscinet-getitem?mr=505376. [61]

K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06. ACM Press, 2006. ISBN 1-59593-428-6. DOI http://doi.acm.org/10.1145/1146847.1146860. [57, 182]

M. M. Maza, B. Stephenson, S. M. Watt, and Y. Xie. Multiprocessed parallelism support in ALDOR on SMPs and multicores. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 60–68. ACM Press, 2007. ISBN 978-1-59593-741-4. DOI http://doi.acm.org/10.1145/1278177.1278188. [12]

M. D. McIlroy. Power series, power serious. *Journal of Functional Programming*, 9(3):325–337, 1999. DOI http://dx.doi.org/10.1017/S0956796899003299. Functional pearl. [12, 13, 179, 183]

M. D. McIlroy. The music of streams. *Information Processing Letters*, 77(2-4):189–195, 2001. DOI http://dx.doi.org/10.1016/S0020-0190(00)00201-5. [12, 13, 179, 183]

S. D. Mechveliani. Haskell and computer algebra. Manuscript, 2000. URL http://www.botik.ru/pub/local/Mechveliani/basAlgPropos/haskellInCA2.ps.zip. Pereslavl-Zalessky, Russia. [12, 183]

S. D. Mechveliani. DoCon the algebraic domain constructor program, 2007a. URL http://www.haskell.org/docon/. Retrieved 6.5.2011. [12, 177, 183]

S. D. Mechveliani. *DoCon. The Algebraic Domain Constructor Manual*. Program Systems Institute, Pereslavl–Zalessky, Russia, 2007b. Version 2.11. [12, 183]

H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra. Top500 supercomputing sites, 2011. URL http://www.top500.org. Retrieved 22.6.2011. [46]

G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.*, 16:181–206, 2001. [22, 183]

A. Migotti. Zur Theorie der Kreisteilungsgleichung. In *Sitzber. der Classe der Kaiserlichen Akademie der Wissenschaften*, volume 87, pages 7–14, Wien, 1883. [69]

P. Mihăilescu. Cyclotomy primality proving – recent developments. In J. Buhler, editor, *Algorithmic Number Theory*, LNCS 1423, pages 95–110. Springer-Verlag, 1998. DOI http://dx.doi.org/10.1007/BFb0054854. [88]

G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13 (3):300–317, 1976. [62, 181]

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. DOI http://dx.doi.org/10.1016/0022-0000(78)90014-4. [12]

M. Monagan. Maximal quotient rational reconstruction: an almost optimal algorithm for rational reconstruction. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 243–249. ACM Press, 2004. ISBN 1-58113-827-X. DOI http://doi.acm.org/10.1145/1005285.1005321. [147]

M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005. [1, 2, 172, 180]

P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44 (170):519–521, 1985. [88, 148]

P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, pages 362–369, 2005. [103]

G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1): 82–85, 1998. ISSN 0018-9219. Original publication was in April 19, 1965, in *Electronics*. [3]

J. F. Morrison. Parallel p-adic computation. *Information Processing Letters*, 28(3):137–140, 1988. ISSN 0020-0190. DOI 10.1016/0020-0190(88)90159-7. [174, 181]

MPI, 2009. *MPI: A Message-Passing Interface Standard — Version 2.2*. High Performance Computing Center Stuttgart, 2009. [17, 39, 182]

H. Naundorf. Parallelism in MuPAD. In M. Wester, S. Steinberg, and M. Jahn, editors, *Electronic Proceedings of the 1st International IMACS Conference on Applications of Computer Algebra*, May 1995. [2, 181]

D. K. Nguyen, I. Lavallèe, M. Bui, and Q. T. Ha. A general scalable parallelizing of Strassen's algorithm for matrix multiplication on distributed memory computers. In *ICIS '05 — Fourth ACIS Intl. Conf. Computer and Information Science*, pages 294–299. IEEE Computer Society, 2005. ISBN 0-7695-2296-3. DOI http://doi.ieeecomputersociety.org/10.1109/ICIS.2005.8. [129, 181]

P. Q. Nguyen and D. Stehlé. Floating-point LLL revisited. In *Advances in Cryptology – EUROCRYPT 2005*, pages 215–233. Springer-Verlag, 2005. [9, 183]

B. Nichols, D. Buttlar, and J. Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. *Reilly, California*, 1996. [182]

R. Nikhil and L. A. Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, 2001. [17, 182]

R. S. Nikhil, L. A. Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen, and Y. Zhou. *pH Language Reference Manual, Version 1.0*. Massachusetts Institute of Technology, 1995. URL http://csg.csail.mit.edu/pubs/memos/Memo-369/memo-369.pdf. Computation Structures Group Memo No. 396. [182]

J. F. Nolan. *Analytical differentiation on a digital computer*. Master's thesis, Massachusetts Institute of Technology, 1953. [7]

H. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, 1981. [108, 110, 111, 117, 125]

M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008. ISBN 0981531601. [8]

M. E. O'Neill. The genuine sieve of Eratosthenes. *Journal of Functional Programming*, 19(01):95–106, 2008. [61, 193]

O. Ore. The general Chinese remainder theorem. *The American Mathematical Monthly*, 59(6):365–370, 1952. ISSN 00029890. URL http://www.jstor.org/stable/2306804. [149]

B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2008. ISBN 978-0-596-51498-3. [4, 8]

V. Y. Pan. *How to multiply matrices faster*. LNCS 179. Springer-Verlag, 1984a. ISBN 3-387-13866-8. [129, 131]

V. Y. Pan. How can we speed up matrix multiplication? *SIAM review*, 26(3):393–415, 1984b. ISSN 0036-1445. [129, 131]

F. Pauer. On lucky ideals for Gröbner basis computations. *Journal of Symbolic Computation*, 14(5): 471–482, 1992. [181]

M. C. Pease. An adaptation of the fast fourier transform for parallel processing. *Journal of the ACM*, 15(2):252–264, April 1962. [108]

S. Pelagatti. Task and data parallelism in P3L. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and skeletons for parallel and distributed computing*, pages 155–186. Springer-Verlag, 2003. ISBN 1-85233-506-8. [182]

R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *PPDP'01 — Intl. Conf. on Principles and Practice of Declarative Programming*, pages 187–198, Firenze, Italy, September 5–7, 2001. [16, 22, 26, 97, 129, 135]

A. J. Perlis. Special feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17:7–13, September 1982. ISSN 0362-1340. DOI http://doi.acm.org/10.1145/947955.1083808. [188]

O. Perron. *Die Lehre von den Kettenbrüchen*, volume I. Teubner, third edition, 1954. [9]

S. Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003. [2, 7, 8, 11, 55]

S. Peyton-Jones. Beautiful concurrency. In G. Wilson, editor, *Beautiful Code*, pages 385–406. O'Reilly, 2007. [4, 182]

S. Peyton-Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of POPL '96*, pages 295–308, New York, NY, USA, 1996. ACM Press. [17, 182]

B. Pickenbrock. *Optimierung der Eden-Kommunikation auf Multicores*. Bachelor's thesis, Philipps-Universität Marburg, 2011. In German. To appear. [17]

M. Poldner and H. Kuchen. Skeletons for divide and conquer algorithms. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN '08, pages 181–188, Anaheim, CA, USA, 2008a. ACTA Press. ISBN 978-0-88986-714-7. URL http://portal.acm.org/citation.cfm?id=1722252.1722286. [92, 135, 182]

M. Poldner and H. Kuchen. Optimizing skeletal stream processing for divide and conquer. In *Proceedings of the 3rd International Conference on Software and Data Technologies*, ICSOFT '08, pages 181–189, 2008b. [92, 135, 182]

J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971. [91]

S. Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. In *Euro-Par*, LNCS 4128, 2006. [16, 28, 57, 135, 178, 183]

S. Priebe. *Structured Generic Programming in Eden*. PhD thesis, Philipps-Universität Marburg, 2007. [28, 183]

D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proc. 10th Symposium on Computer Arithmetic*, pages 132–145. IEEE, 1991. [9]

PVM, 2009. Parallel Virtual Machine, 2009. URL http://www.csm.ornl.gov/pvm/. Retrieved 7.5.2011. [17, 182]

A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*. Springer-Verlag, 2007. [163]

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. URL http://www.R-project.org. [10, 42]

F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003. [3]

M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. [62, 181]

M. A. Rainey and D. S. Wise. Embedding quadtree matrices in a functional language. Technical report, Opie Research Group, Department of Computer Science, Indiana Univeristy, USA, 2004. [92, 129, 130]

M. Rao et al. Conversion of Hensel codes to rational numbers. *Computers & Mathematics with Applications*, 10(2):185–189, 1984. ISSN 0898-1221. [174]

R. Rāshid. *The development of Arabic mathematics: between arithmetic and algebra*. Springer-Verlag, 1994. ISBN 0792325656. [7]

D. Redfern. *The Maple Handbook: Maple V Release 4*. Springer-Verlag, December 1995. [1, 2, 172, 180]

P. Ribenboim. *The little book of bigger primes*. Springer-Verlag, 2004. ISBN 0387201696. [10, 62]

R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/359340.359342. [5]

J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999. [50]

R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14:344–384, 1996. ISSN 0734-2071. DOI http://doi.acm.org/10.1145/235543.235545. [50]

L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. *Trends in Functional Programming*, volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pages 65–80. Intellect, 2011. Final Proceesings of TFP 2009. [17]

T. Sasaki and F. Kako. Computing floating-point Gröbner bases stably. In *SNC '07: Proceedings of the 2007 international workshop on Symbolic-numeric computation*, pages 180–189. ACM Press, 2007. ISBN 978-1-59593-744-5. DOI http://doi.acm.org/10.1145/1277500.1277526. [1, 181]

T. Sasaki and M. Sasaki. On integer-to-rational conversion algorithm. *ACM SIGSAM Bulletin*, 26(2): 19–21, 1992. [147, 174]

T. Sasaki, Y. Takahashi, and T. Sugimoto. A divide-and-conquer method for integer-to-rational conversion. In *Symposium in Honor of Bruno Buchberger's 60th Birthday*, page 231, 2002. [147, 174]

T. Sauer. Computeralgebra. Lecture notes, Justus-Liebig-Universität Gießen, 2002. [174, 193, 194]

T. Sauer. Numerische Mathematik I. Lecture notes, Justus-Liebig-Universität Gießen, 2002. [163]

B. Schmidt, M. Schimmler, and H. Schroeder. *Embedded Cryptographic Hardware: Methodologies & Architectures*, chapter High-Speed Cryptography. Nova Science Publishers, 2004. ISBN 1-59454-012-8. [64, 89]

E. Scholz. Four concurrency primitives for Haskell. In *ACM/IFIP Haskell Workshop*, 1995. [182]

A. Schönhage. Asymptotically fast algorithms for the numerical muitiplication and division of polynomials with complex coefficients. In J. Calmet, editor, *Computer Algebra*, volume 144 of *LNCS*, pages 3–15. Springer-Verlag, 1982. DOI http://dx.doi.org/10.1007/3-540-11607-9_1. [91, 108]

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4):281–292, 1971. [91, 93, 108, 126, 181]

W. Schreiner. *Parallel Functional Programming for Computer Algebra*. PhD thesis, Johannes Kepler University, Research Institute for Symbolic Computation (RISC-Linz), 1994. [2, 181]

W. Schreiner and H. Hong. The design of the PACLIB kernel for parallel algebraic computation. In J. Volkert, editor, *Parallel Computation*, LNCS 734, pages 204–218. Springer-Verlag, 1993. DOI http://dx.doi.org/10.1007/3-540-57314-3_17. [2, 181]

W. Schreiner, C. Mittermaier, and K. Bosa. Distributed Maple: Parallel computer algebra in networked environments. *Journal of Symbolic Computation*, 35(3):305–347, 2003. [2, 181]

SCIEnce, 2010. Symbolic Computation Infrastructure for Europe project, 2010. URL http://www.symbolic-computation.org/. Retrieved 05.09.2010. [2, 173, 181, 183]

S. M. Sedjelmaci. A parallel extended GCD algorithm. *Journal of Discrete Algorithms*, 6:526–538, 2008. ISSN 1570-8667. DOI http://dx.doi.org/10.1016/j.jda.2006.12.009. [181]

J. Sérot, D. Ginhac, and J. P. Dérutin. SKiPPER: a skeleton-based parallel programming environment for real-time image processing applications. In *Parallel Computing Technologies*, LNCS 1662, pages 767–767. Springer-Verlag, 1999. ISBN 978-3-540-66363-8. [182]

G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, University of California, San Diego, CA, USA, 2001. AAI3022200. [16]

G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. *Heterogeneous Computing Workshop*, pages 3–16, 2000. ISSN 1097-5209. DOI http://doi.ieeecomputersociety.org/10.1109/HCW.2000.843728. [16]

N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213. ACM Press, 1995. ISBN 0-89791-710-3. DOI http://doi.acm.org/10.1145/224964.224987. [3, 182]

N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. ISSN 0178-2770. DOI http://dx.doi.org/10.1007/s004460050028. [3, 182]

J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997. [9]

V. Shoup. NTL: A library for doing number theory, 2009. URL http://www.shoup.net/ntl/. Retrieved 27.4.2010. [8, 180]

K. Siegl. Parallelizing algorithms for symbolic computation using ‖Maple‖. In *In 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186. ACM Press, 1993. [2, 173, 181]

D. B. Skillicorn. *Foundations of parallel programming*. Cambridge University Press, 1994. ISBN 0521455111. [3]

M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995. [17, 182]

D. Stehlé. Floating-point LLL: theoretical and practical aspects. In P. Q. Nguyen and B. Vallée, editors, *The LLL Algorithm*, Information Security and Cryptography, pages 179–213. Springer-Verlag, 2010. [9]

W. Stein. *Elementary number theory: primes, congruences, and secrets. A computational approach*. Springer-Verlag, 2009. ISBN 0387855246. [71]

W. Stein and D. Joyner. SAGE: system for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39:61–64, 2005. ISSN 0163-5824. DOI http://doi.acm.org/10.1145/1101884.1101889. [2, 10]

W. A. Stein et al. *Sage Mathematics Software (Version 4.6.2)*. The Sage Development Team, 2011. URL http://www.sagemath.org. [2, 10]

J. Stillwell. *Numbers and geometry*. Springer-Verlag, 1998. ISBN 978-0387982892. [192]

G. Strang. Wavelet transforms versus Fourier transforms. *Bulletin of the American Mathematical Society*, 28(2):288–305, 1993. [109, 110]

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969. [5, 91, 103, 129, 181]

M. Sulzmann. Haskell actors library, 2008. URL http://hackage.haskell.org/package/actor. Retrieved 7.5.2011. [135, 182]

M. Sulzmann, E. S. L. Lam, and P. Van Weert. Actors with multi-headed message receive patterns. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, pages 315–330. Springer-Verlag, 2008. ISBN 3540682643. [135, 182]

V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990. [17, 182]

A. Svoboda and M. Valach. Rational system of residue classes. *Stroje na Zpraccorani Informaci, Sbornik, Nakl. CSZV, Prague*, pages 9–37, 1957. [174]

N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967. [150, 182]

A. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32(2):445–465, 1985. ISSN 0004-5411. [16]

B. L. van der Waerden. *Algebra*, volume I. Springer-Verlag, 1971. [189, 195]

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, second edition, 2004. [10]

M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. IEEE Computer Society Washington, DC, USA, 1998. [129]

D. Thurston, H. Thielemann, and M. Johansson. Haskell Numeric Prelude program, 2010. URL http://code.haskell.org/numeric-prelude/. Retrieved 24.08.2010. [93, 126, 183]

R. G. Tobey. Experience with FORMAC algorithm design. *Communications of the ACM*, 9(8):589–597, 1966. ISSN 0001-0782. DOI http://doi.acm.org/10.1145/365758.365773. [5, 7, 9, 143]

A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963. [91, 103]

P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton-Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998a. DOI http://dx.doi.org/10.1017/S0956796897002967. URL http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/strategies.ps.gz. [17, 21, 183]

P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. Peyton-Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96*. ACM Press, 1996a. [21]

P. W. Trinder, K. Hammond, et al. GUM: a portable parallel implementation of Haskell. In *PLDI'96*. ACM Press, 1996b. [17, 182]

P. W. Trinder, E. Barry Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. Peyton-Jones. GpH: An Architecture–Independent Functional Language. In *Glasgow Functional Programming Workshop*, Pitlochry, Scotland, September 1998b. [17, 21, 183]

P. W. Trinder, E. Barry Jr, et al. GPH: an architecture-independent functional language. *IEEE Trans. Software Engineering*, 1999. [17, 21, 182]

P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4, 5):469–510, 2002. [17]

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):111, 1990. [50]

R. A. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997. [129]

B. L. van der Waerden and H. Habicht. *Erwachende Wissenschaft*. Birkhäuser, 1966. [192]

G. van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2006. [8, 10, 11]

J. von zur Gathen. Parallel algorithms for algebraic problems. *SIAM Journal on Computing*, 13(4): 802–824, 1984. DOI 10.1137/0213050. URL http://link.aip.org/link/?SMJ/13/802/1. [181]

J. von zur Gathen and J. Gerhard. Polynomial factorization over $\mathbb{F}_2$. *Mathematics of Computation*, 71 (240):1677–1698, 2002. [180]

J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, second edition, 2003. [B, 5, 7, 9, 62, 91, 93, 100, 101, 102, 108, 109, 111, 125, 126, 131, 143, 147, 149, 151, 163, 164, 173, 174, 175, 178, 183, 192, 195]

P. S. Wang. A p-adic algorithm for univariate partial fractions. In *Proceedings of the 4th ACM Symposium on Symbolic and Algebraic Computation*, pages 212–217. ACM Press, 1981. [147, 174]

P. S. Wang, M. J. T. Guy, and J. H. Davenport. p-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin*, 16(2):3, 1982. [147, 174]

H.-J. Waschkies. *Anfänge der Arithmetik im alten Orient und bei den Griechen*. John Benjamins, 1989. [193]

S. M. Watt. *Bounded parallelism in computer algebra*. PhD thesis, University of Waterloo, 1986. [2, 181]

S. M. Watt. Making computer algebra more symbolic. In *In Proceedings of Transgressive Computing 2006: A conference in honor of Jean Della Dora*, 2006. DOI http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.126.5233. [7]

A. Weimerskirch and C. Paar. Generalizations of the Karatsuba algorithm for efficient implementations. Cryptology ePrint Archive 2006/224, Ruhr Universität Bochum, Lehrstuhl für Embedded Security, 2003. [103]

A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 285–296. ACM Press, 2008. ISBN 978-1-59593-973-9. DOI http://doi.acm.org/10.1145/1378533.1378584. [182]

P. H. Welch. Graceful termination–graceful resetting. pages 310–317, Netherlands, 1989. IOS Press. [58]

P. H. Welch and F. R. M. Barnes. Communicating mobile processes: introducing occam-pi. LNCS 3525. Springer-Verlag, 2005. [4]

S. Wetzel. An efficient parallel block-reduction algorithm. In *Algorithmic Number Theory*, pages 323–337. Springer-Verlag, 1998. [181]

K. B. Wheeler and D. Thain. Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, 2009. [35]

H. S. Wilf. *Algorithms and complexity*. AK Peters, second edition, 2002. ISBN 1568811780. [111, 131]

J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice Hall, 1963. [1]

L. H. Williams. Algebra of polynomials in several variables for a digital computer. *Journal of the ACM*, 9:29–40, 1962. ISSN 0004-5411. DOI http://doi.acm.org/10.1145/321105.321109. [180]

F. Winkler. A p-adic approach to the computation of Gröbner bases. *Journal of Symbolic Computation*, 6(2-3):304, 1988. [144, 175, 181]

S. Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4(4):381–388, 1971. ISSN 0024-3795. DOI http://dx.doi.org/10.1016/0024-3795(71)90009-7. [129, 131]

S. Winograd. On computing the discrete Fourier transform. *Proceedings of the National Academy of Sciences of the United States of America*, 73(4):1005, 1976. [108]

N. Wirth. The programming language Pascal. *Acta informatica*, 1(1):35–63, 1971. [53, 188]

S. Wolfram. *Mathematica: a system for doing mathematics by computer*. Wolfram Research, Inc., 1991. [2, 180]

S. Wolfram. *The Mathematica Book*. Wolfram Research Inc., 2000. [2, 180]

S. C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *Computer Journal*, 32(2):142–151, 1989. ISSN 0010-4620. [23, 135, 178, 183]

S. Y. Yan and M. E. Hellman. *Number theory for computing*. Springer-Verlag, 2002. ISBN 3540430725. [63]

C. Yap and C. Li. QuickMul: Practical FFT-based integer multiplication. Technical report, Department of Computer Science Courant Institute, New York, 2000. [91, 108]

M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8thUSENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42. USENIX Association, 2008. URL http://portal.acm.org/citation.cfm?id=1855741.1855744. [57]

C. J. Zarowski and H. C. Card. On addition and multiplication with hensel codes. *IEEE Transactions on Computers*, 39:1417–1423, 1990. DOI http://dx.doi.org/10.1109/12.61062. [174]

A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–407, 2001. [50, 183]

INDEX OF PERSONALITIES

INDEX

LEBENSLAUF

Oleg Lobachev
Hangelsteinring 15
D–35396 Gießen
Tel.: 0641–690244
E-Mail: oleg.lobachev@gmail.com

Persönliche Daten

geboren am	26.11.1982 in Kiew, Ukraine
Seit 04.1997	in Deutschland
Staatsangeh.	deutsch
Familienstand	ledig

Studium

2001–2007	Studium an der Justus-Liebig-Universität Gießen Diplomstudiengang Mathematik mit Nebenfach Informatik Schwerpunkt: Computeralgebra, Diplomnote: 1,0.
seit 09.2007	Promotionsvorhaben an der Philipps-Universität Marburg in theoretischer Informatik

Berufserfahrung

02.2002–09.2006	Systemadministrator an der Justus–Liebig–Universität Gießen
04.2007–03.2009	Softwareentwickler für Embedded– und PC–Systeme bei Identass GmbH & Co. KG, Gießen
04.2009–03.2011	Wissenschaftlicher Mitarbeiter im Drittmittelprojekt an der Philipps-Universität Marburg
seit 06.2011	Wissenschaftlicher Mitarbeiter im Drittmittelprojekt an der Philipps-Universität Marburg

Schulbildung

09.1989–04.1997	Gymnasium Nr. 57 mit Tiefenfach Englisch in Kiew, Ukraine
09.1997–06.2001	Herderschule Gießen, englischer Bilingualzweig. Abschluß: Abitur

Sprachen	Verhandlungssicheres Deutsch
	Verhandlungssicheres Englisch
	Russisch und Ukrainisch als Muttersprachen

Gießen, 15. August 2011

ERKLÄRUNG

Ich versichere, dass ich meine Dissertation »Implementation and Evaluation of Algorithmic Skeletons: Parallelisation of Computer Algebra Algorithms« selbstständig und ohne fremde Hilfe verfasst, nicht andere als die in ihr angegebenen Quellen oder Hilfsmittel benutzt, alle vollständig oder sinngemäß übernommenen Zitate als solche gekennzeichnet sowie die Dissertation in der vorliegenden oder einer ähnlichen Form noch bei keiner anderen in- oder ausländischen Hochschule anlässlich eines Promotionsgesuchs oder zu anderen Prüfungszwecken eingereicht habe.

Gießen, 15. August 2011 .....