8-2-2022 3:00 PM

The Design and Implementation of a High-Performance Polynomial System Solver

Alexander Brandt, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science

Abstract

This thesis examines the algorithmic and practical challenges of solving systems of polynomial equations. We discuss the design and implementation of triangular decomposition to solve polynomials systems exactly by means of symbolic computation.

Incremental triangular decomposition solves one equation from the input list of polynomials at a time. Each step may produce several different components (points, curves, surfaces, etc.) of the solution set. Independent components imply that the solving process may proceed on each component concurrently. This so-called component-level parallelism is a theoretical and practical challenge characterized by *irregular parallelism*. Parallelism is not an algorithmic property but rather a geometrical property of the particular input system's solution set.

Despite these challenges, we have effectively applied parallel computing to triangular decomposition through the layering and cooperation of many parallel code regions. This parallel computing is supported by our generic object-oriented framework based on the dynamic multithreading paradigm. Meanwhile, the required polynomial algebra is supported by an object-oriented framework for algebraic types which allows type safety and mathematical correctness to be determined at compile-time.

Our software is implemented in C/C++ and we have extensively tested the implementation for correctness and performance on over 3000 polynomial systems that have arisen in practice.

The parallel framework has been re-used in the implementation of Hensel factorization as a parallel pipeline to compute roots of a polynomial with multivariate power series coefficients. Hensel factorization is one step toward computing the non-trivial limit points of quasi-components.

Keywords: polynomial system solving, triangular decomposition, computer algebra software, irregular parallelism, dynamic multithreading, Hensel lifting, power series

Summary for Lay Audience

Solving systems of polynomial equations and inequations is a fundamental problem in scientific computing, required by practically all scientific disciplines. Expanding research in these disciplines demands solving larger and more complex problems than ever before. Only relatively recently has technological advances in computer hardware performance (processor speeds, computer memory, and computer architectures) and in algorithmic methods made it feasible to obtain exact solutions to such practical polynomial systems by solving them symbolically. In contrast, scientists have traditionally relied on numerical methods to obtain approximate solutions. Nonetheless, exact solutions are desirable, and often necessary, in many fields such as cryptography, theoretical physics, robotics, and signal processing.

Solving systems of polynomial equations symbolically is, by its very nature, a very hard problem. The algorithms involved are highly sophisticated, computationally expensive, and require numerous supporting sub-algorithms. In this thesis we examine the design and implementation of a symbolic polynomial system solver which is very efficient in practice. We consider implementation techniques including parallel computing to achieve performance. In particular, multiple areas of parallelism are combined cooperatively. We also consider so-called dynamic evaluation, which allows for the branches of a case discussion to dynamically evolve. Such case discussions evolve in solving systems where the geometry of the solution set splits into multiple pieces (points, curves, surfaces). The implementation has been extensively tested on over 3000 polynomial systems which have arisen in practice.

This solver is part of an open-source computer algebra library called BPAS (Basic Polynomial Algebra Subprograms). In this thesis we also discuss software design strategies for this library and the solver which look to achieve both high performance and high levels of software quality. We obtain desirable quality attributes including extensibility and maintainability. A major result in this direction is a computer implementation of a mathematical framework called the algebraic hierarchy. We implement an extensible hierarchy that has the added benefit that mathematical correctness is automatically enforced before the program is even executed (i.e. at compile-time).

Co-Authorship Statement

Some work in this thesis has been the result of collaboration with others.

- Conceptualization of the algebraic class hierarchy (Chapter 4) was joint work with Robert H.C. Moir.
- Initial implementations of triangular decomposition in BPAS were a joint effort with Robert H.C. Moir. These implementations formed the bases of the implementation presented in [12].
- Work on speculative subresultants (Section 6.2) is joint with Mohammadali Asadi; see [13].
- Lazy multivariate power series presented in Section 7.2 is joint work with Mahsa Kazemi [32]. Kazemi worked the theory and an initial implementation in Python. The implementation reported here is in C and by myself.

Acknowledgements

There are many people who require my thank for their selfless support throughout the completion of this thesis.

Firstly, I want to especially thank my supervisor Marc Moreno Maza for his patience, guidance, and humour. His suggestions, motivations, reassurances, and encouragements have made my time at Western fruitful and filled with learning.

My many co-authors and collaborators must be thanked for their hard work, creative thoughts, and rewarding discussions: Mohammadali Asadi, Robert Moir, Mahsa Kazemi, Davood Mohajerani, Erik Postma, Linxiao Wang, and Yuzhen Xie.

I would also like to thank Rob Corless and David Jeffrey; their informal advice has been invaluable. My work, and downtime, has also been bettered by the support of fellow students Duff Jones and Caroline Strickland.

Many thanks to my examiners Dr. Michael Monagan, Dr. Rob Corless, Dr. David Jeffrey, and Dr. Hanan Lutfiyya for their thoughful comments and careful reviews.

Special gratitude must be given to my family and friends for their unending support, affirmations, and confidence in me. Thank you to my husband for keeping me sane.

Contents

List of Figures

List of Algorithms

List of Code Listings

List of Tables

Chapter 1

Introduction

Solving systems of polynomial equations and inequations is a fundamental problem in scientific computing, required by practically all scientific disciplines. Expanding research in these areas demands solving larger and more complex problems than ever before. Only relatively recently have technological advances in processor speeds, computer memory, computer architectures, and algorithmic methods made it feasible to obtain exact solutions to such practical polynomial systems by solving them *symbolically*. In contrast, scientists have traditionally relied on numerical methods to obtain approximate solutions. Nonetheless, exact solutions are desirable, and often necessary, in many fields such as cryptography, theoretical physics, robotics, and signal processing [89].

The choice between using numeric and symbolic algorithms requires two considerations. First, what is the nature of the input system: are coefficients known exactly or only as approximations obtained from observation or experimentation? Second, what is the form of the expected answers: does one need a complete description of all the solutions, only the real solutions, or just one sample solution? If the former of both conditions are met, then symbolic methods are the tool of choice (although symbolic methods can also be used to find real solutions, see [50]).

Solving systems of polynomial equations symbolically is, by its very nature, a very hard problem. The algorithms involved are highly sophisticated, computationally expensive, and have vast dependencies. Solving systems of equations requires nearly the entire functionality of a general-purpose computer algebra system: linear algebra, arbitrary-precision numbers, finite fields, polynomial arithmetic, polynomial factorization, and Greatest Common Divisor (GCD) computations.

Methods for solving systems of equations have been under development since the 1800s. Mathematically speaking, finding solutions is a completely solved problem via primary decomposition and the Laskar-Noether theorem [150]. However, this decomposition

is prohibitively expensive to compute and is not well-suited to most applications where geometric information about the solution set is required. To that end, with the advent of computation, two main categories of algorithms arose: those based on characteristic sets and triangular decomposition, introduced by Ritt [153] and made computational by Wu [181, 185]; and those based on Gröbner bases, introduced by Buchberger [39]. Triangular decomposition, the characteristic set method, and the Gröbner basis method are introduced informally in the next section, and details are presented in Chapter 2.

Algorithms based on both methods have been under constant development for decades. Triangular decomposition and characteristic sets have continued to be studied by Wu [182–184, 186], Chou and Gao [55, 82, 84], and Wang [174–177]. Special kinds of characteristic sets called *regular chains* were introduced independently by Kalkbrener [107] and Yang and Zhang [188]. Regular chains have many useful algorithmic and geometric properties, improving the practicality and performance of triangular decomposition. The mathematical theory of regular chains has continued to be researched [15, 16, 30, 83] and algorithms employing them [47, 50, 51, 54, 142] have seen great success. Among Gröbner basis methods, the F4 and F5 algorithms by Faugere [72, 73], and the FGLM algorithm, named for its authors Faugere, Gianni, Lazard, and Mora [76], greatly improved upon the performance of Buchberger's original algorithm. This has allowed Gröbner bases to be among the most important tools in computer algebra, with countless applications beyond solving systems of equations; see, e.g., [2, 61] and references therein.

There are many software implementations of Gröbner bases, and they are commonplace in any standard computer algebra system, including: *Maple* [22], *Mathematica* [180], *Singular* [66], *Magma* [29], *FGb* [75], *Macaulay 2* [92], *SageMath* [154], and *SymPy* [132], among others. Implementations of triangular decomposition based on regular chains are available in: *Axiom*, a literate programming [113] computer algebra system designed to express rich and complex mathematical expressions [104]; *Aldor*, an extension of *Axiom* for high-performance and interoperability [35]; and *Maple* via the *RegularChains* library [125].

Implementations based on regular chains are more scarce, being mainly from a single research group including Lazard, Moreno Maza, Lemaire, and their students [17, 47, 125, 142]. However, that is not to say that Gröbner bases are the better method. Indeed, empirical and theoretical results indicate that methods based on regular chains perform better in practice compared to Gröbner bases [17, 47, 187] and have near-optimal output sizes [63, 65]. In particular, the Ritt-Wu characteristic set method—and its derivatives based on regular chains—is a singly exponential method [81]. Methods based on Gröbner bases are at least singly exponential [37], but doubly exponential in general [129, 130].

More formally, consider the computational complexity classes $\mathcal{NP} \subseteq \mathcal{PSPACE} \subseteq \mathcal{EXPTIME} \subseteq \mathcal{EXPSPACE}$, where $\mathcal{PSPACE} \subsetneq \mathcal{EXPSPACE}$ (see, e.g., [164]). The *existential theory of the reals* [18, Ch. 13] is a decision problem to determine whether a system of polynomial equations over the real numbers has a solution. This problem has been shown to be both $\mathcal{NP}$-hard and contained in $\mathcal{PSPACE}$ [42, 158]. Indeed, a singly exponential time algorithm exists for the existential theory of the reals [18, Ch. 13]. The closely related *radical membership problem* decides if a given polynomial shares a solution with a system of polynomials. Over any field of numbers (real or otherwise) the radical membership problem is in $\mathcal{PSPACE}$ [116]. Triangular decomposition is one method to solve the radical membership problem. The more general *ideal membership problem* has been shown to be $\mathcal{EXPSPACE}$-complete [129, 130]. Computing a reduced Gröbner basis, which can be used to solve the ideal membership problem, is also $\mathcal{EXPSPACE}$-complete [86, Ch. 21]. See Section 2.5 for more details on these problems.

This drastic increase in computational complexity can be attributed, in part, to *intermediate expression swell*—where the size of intermediate expressions grows drastically, even if the final solution is of moderate size [86, Ch. 5–6]. This phenomenon is pervasive throughout symbolic computation in general and continues to challenge researchers, requiring both theoretical and practical approaches. *Modular methods* [86, Ch. 5] based on *Chinese Remaindering Theorem* or *Hensel's lemma* have become indispensable in controlling expression swell and making certain problems tractable. In particular, GCD computations [102, 146, 179, 190] and polynomial factorization [136, 178] are both fundamental subroutines of system solving which have seen great practical improvements thanks to modular methods. For example, polynomial factorization over the integers has been reduced from exponential time [117] to polynomial time [162]. Modular methods have also been applied directly to triangular decomposition for particular kinds of input systems [64, 127].

Despite these exponential time and space costs, solving systems of polynomial equations symbolically remains a necessity in scientific computing. Unfortunately, existing implementations remain insufficient. There are many highly optimized implementations of Gröbner bases, however, they are not well suited to describe the solutions of polynomial systems with infinitely many solutions (i.e. for positive-dimensional systems). Further, their doubly exponential complexity remains a problem. Regular chains provide better time complexity and better geometrical properties in their description of a solution set. Yet, the small number of implementations employing regular chains do not exhibit the same degree of optimization as seen in Gröbner bases. The implementations of triangular decomposition in *Axiom* and *Aldor* are mainly of research interest. The *RegularChains*

library of *Maple* is more optimized for practical use. However, its implementation in the interpreted *Maple* language limits its ability to finely control hardware resources like memory and multi-core processors.

In this work we propose to close this gap by designing and implementing a high-performance and parallel polynomial system solver based on regular chains. This work has two high-level goals:

(*i*) to provide a high-performance, open-source, parallel implementation of regular chains for modern computer architectures; and

(*ii*) to examine how software design and software engineering can be used to manage the maintainability and usability of such highly optimized and complex code.

The need for research on the latter is clear from the state of academic and open-source libraries in mathematical and scientific computing. A major downfall of this so-called end-user programming is its negligence towards maintainability and usability. Broadly speaking, researchers are more interested in innovative algorithms than in maintainability and usability [43]. Within mathematical software this problem is exacerbated by scientists often lacking expertise in software engineering [115, 161]. The challenges of end-user programming have only recently been examined [115].

Regarding the former objective, the need for high-performance considerations is obvious from the previous paragraphs. Toward high-performance, we will examine both parallelization and memory usage. The processor-memory gap—where processor speeds are exponentially diverging from memory speeds [95, Section 2.1]—requires that we finely control memory usage and cache complexity (cache misses) [79] to achieve optimal performance on modern computer architectures with cache memory hierarchies. This idea has already been quite successful in computer algebra [8, 11, 13, 31, 48, 59].

Turning to parallelization, there has been a great deal of work on parallelizing high-level algebraic and geometric algorithms in the '80s and '90s; for example, see [14, 38, 40, 74, 157]. In recent years, parallelization has again seen attention, but as primarily fine-grained parallelism in low-level operations. Examples include multithreading in polynomial arithmetic [25, 85, 135], GCDs [101], and factorization [139]; and the use of vectorized instructions in modular integer arithmetic [100] and polynomial evaluation [78].

Parallelization of these low-level routines is more natural, where either the input data, or the algorithmic process, naturally decomposes into independent and similar pieces, or tasks, respectively. Such parallelism is known as *regular parallelism*, see Section 3.2, since the algorithm decomposes work in a static way into relatively consistently sized units. On

the other hand, taking advantage of the *irregular parallelism* found in high-level geometric computations is more challenging. Indeed, the scalability of such methods is typically limited by the inherent geometric structure of each particular problem instance [27, 28].

In triangular decomposition and other geometric algorithms, how the solution space decomposes into independent components (points, curves, surfaces) depends on the particular input system being solved. Indeed, a key issue in triangular decomposition is that finding splittings in the geometry is as difficult as solving the system itself. It is not possible to statically define parallelism via the organization of the algorithm. This implies that the decomposition of the work into independent tasks must be found dynamically as the algorithm progresses. Only then can so-called *component-level parallelism* be exploited. This technique has previously been explored in *Aldor* but was limited by inter-process communication and the available hardware [144].

To better understand the concepts of regular chains and component-level parallelism, we now present an informal introduction to polynomial system solving. We then review our objectives and contributions in Section 1.2.

1.1 An Informal Introduction to Polynomial System Solving

Let us begin by recalling what a polynomial is: a linear combination of products of variables. Polynomials consist only of addition, subtraction, and multiplication. A univariate polynomial has only one variable, for example $f = 3x^2 + 7x + 2$. A multivariate polynomial has several variables, for example $g = 7xy^2 + 2x + 3yz$. Where there are multiple variables, it is useful to define an ordering on the variables, resulting in a *main variable* and thus a *leading coefficient*, akin to the univariate case. Under the variable ordering $x > y > z$, g may be written in a *recursive* representation: $g = (7y^2 + 2)x + 3yz$. The main variable of g is x and its leading coefficient is another polynomial $7y^2 + 2$.

Consider now a first polynomial system F_1:

$$F_1 = \begin{cases} 2x + y + z - 1 = 0 \\ x + 2y + z - 1 = 0 \\ x + y + 2z - 1 = 0 \end{cases}$$

Notice that in this particular case the polynomial system is in fact a linear system. Hence, it is quite simple to solve. One may recall the elimination method from grade school mathematics, or *Gaussian elimination* or *Cramer's rule* from introductory linear

algebra [166]. From numerical linear algebra, there are many possible methods including QR factorization and singular value decomposition (SVD); see, e.g., [58, Ch. 4]. Applying any technique leads us to find the unique solution:

$$\begin{cases} x = \frac{1}{4} \\ y = \frac{1}{4} \\ z = \frac{1}{4} \end{cases}$$

Moving to the non-linear case, solving equations becomes much more challenging. Given a simple polynomial equation, e.g. $x^2 + 2x - 1 = 0$, the elementary *quadratic formula* tells us that its solutions are $x = -1 \pm \sqrt{2}$. Alternatively, one may say the roots of the polynomial x^2+2x-1 are $-1+\sqrt{2}$ and $-1-\sqrt{2}$. But, finding roots of polynomials with larger degrees is much more challenging and sometimes impossible. *Galois Theory* and the *Abel-Ruffini Theorem* tell us that there are no generic formulas in radicals for the roots of a polynomial whose degree is higher than four; see, e.g., [171, Ch. 7].

There is an inherent symmetry between finding the solutions of a polynomial equation and finding the roots of a polynomial. Indeed, any polynomial can be rewritten into the form $f = 0$ so that the roots of the polynomial f are precisely the solutions of the equation. For that reason, solving a system of polynomial equations and finding the common roots of a collection of polynomials is fundamentally the same problem. We henceforth omit the "$= 0$" for brevity.

Consider a non-linear system of equations F_2:

$$F_2 = \begin{cases} x^2 + y + z - 1 \\ x + y^2 + z - 1 \\ x + y + z^2 - 1 \end{cases}$$

Computing a Gröbner basis G_2 for F_2 produces an equivalent system in the sense that G_2 and F_2 have the same solutions. Under the lexicographical term ordering with $x > y > z$, a Gröbner basis (see [86, Ch. 21]) is:

$$G_2 = \begin{cases} x + y + z^2 - 1 \\ y^2 - z^2 - y + z \\ 2yz^2 + z^4 - z^2 \\ z^6 - 4z^4 + 4z^3 - z^2 \end{cases}$$

Notice that this set has a "triangular shape" in that there is one univariate polynomial in z, two bivariate polynomials in y and z, and one polynomial in all three variables. From

this Gröbner basis, much like Gaussian elimination, one can apply *back substitution* to find all of the solutions. First, we apply factorization to each polynomial:

$$G_2 = \begin{cases} x + y + z^2 - 1 \\ (y + z - 1)(y - z) \\ z^2(2y + z^2 - 1) \\ z^2(z-1)^2(z^2 + 2z - 1) \end{cases}.$$

Then, it is apparent that there are at most 4 unique values for z in the solution set: $0, 1, -1 \pm \sqrt{2}$. Evaluating the second-last equation at each point of z yields possible values for y: $0, -1 \pm \sqrt{2}$. Repeating in this way for each equation, dropping solutions which are inconsistent, we may enumerate all 5 solutions of F_2:

$$\begin{cases} x = 0 \\ y = 0 \\ z = 1 \end{cases}, \quad \begin{cases} x = 0 \\ y = 1 \\ z = 0 \end{cases}, \quad \begin{cases} x = 1 \\ y = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x = -1 + \sqrt{2} \\ y = -1 + \sqrt{2} \\ z = -1 + \sqrt{2} \end{cases}, \quad \begin{cases} x = -1 - \sqrt{2} \\ y = -1 - \sqrt{2} \\ z = -1 - \sqrt{2} \end{cases}$$

From this example, it is evident that merely computing a Gröbner basis is insufficient to solve a system of equations. Additional processing is required to interpret the results and obtain particular solutions. In contrast, a triangular decomposition of F_2 produces four regular chains:

$$\begin{cases} x = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x - 1 = 0 \\ y = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x - z = 0 \\ y - z = 0 \\ z^2 + 2z - 1 = 0 \end{cases}.$$

Notice that these four sets or *components* directly encode the points in the solution set (where the roots of $z^2 + 2z - 1$ are precisely $-1 \pm \sqrt{2}$).

The system of equations F_2 is said to be *zero-dimensional* since there are a finite number of solutions. A *positive-dimensional* system is one with infinitely many solutions. As an example, consider the system F_3:

$$F_3 = \begin{cases} 2xy - yw + z^2 \\ 2y^2 - yw - z^2 \\ z + w \end{cases}$$

Under the lexicographical term ordering with $x > y > z > w$, a Gröbner basis of F_3 is:

$$G_3 = \left\{ \begin{array}{r} 2xy - yw + w^2 \\ xw^2 + yw^2 - w^3 \\ 2y^2 - yw - w^2 \\ z + w \end{array} \right. = \left\{ \begin{array}{r} 2xy - yw + w^2 \\ w^2(x + y - w) \\ (y - w)(2y + w) \\ z + w \end{array} \right.$$

and its triangular decomposition is the set $\{T_1, T_2, T_3\}$:

$$T_1 = \left\{ \begin{array}{r} 2x - 3w \\ 2y + w \\ z + w \end{array} \right. , \quad T_2 = \left\{ \begin{array}{r} x \\ y - w \\ z + w \end{array} \right. , \quad T_3 = \left\{ \begin{array}{r} y \\ z \\ w \end{array} \right. .$$

From the triangular decomposition one may immediately recognize that this system has infinitely many solutions. Indeed, T_3 tells us that $y = z = w = 0$ while x is a free variable. Geometrically, T_3 represents the one-dimensional line in four-dimensional space parameterized by $(t, 0, 0, 0)$. Similarly, w is a free variable in T_1 and T_2. Each of the regular chains T_1, T_2, T_3 in the triangular decomposition correspond to a specific geometric component of the solution space. In this case, 3 different lines.

Comparatively, the Gröbner basis G_3 does not immediately present such geometric properties. Moreover, the basis appears to be more complicated than the system F_3, since degrees have increased. Meanwhile, the triangular decomposition appears to be more simple than the input system. Yet, both sufficiently "solve" the system.

This introduces one fundamental concern in the context of solving polynomial systems: what does it mean to solve a system of equations? In dimension zero, the answer is to simply list all points in the solution set. In positive dimension, the answer is more subtle, with no unique correct answer. Ultimately, the output of solving a positive-dimensional system should be a representation of the solutions from which useful information can be obtained.

Gröbner bases have a useful algorithmic property that, after a basis has been computed, it efficiently solves the *ideal membership problem*. That is, given any polynomial p and a Gröbner basis G of a polynomial system F, determining whether or not p shares a solution with F is algorithmic and very easy to compute. However, as we have just seen with the system F_3, Gröbner bases do not provide as much geometric information as triangular decompositions.

An alternative way to "solve" a system of equations is to give a description of each component in its solution set. Triangular decompositions do just that, as we have seen with T_1, T_2, and T_3 representing the three lines in the solution of the system F_3. We will

see later in Section 2.5 that using regular chains to encode a triangular decomposition also has algorithmic advantages similar to Gröbner bases. For now, we conclude this section with another useful algorithmic property of triangular decompositions which will serve as the basis of our *component-level parallelism*; see Chapter 6.

An important advantage of some methods for solving systems is that they proceed by *incremental solving*—equations from the input system is solved one at a time. Incremental solving has successfully been applied to both Gröbner bases [73] and triangular decomposition [121, 122]. For triangular decomposition, one equation of the input system is solved against each component of the current solution set, where the current solution set is the components produced by solving the previous equations. This method proceeds until there are no equations remaining in the input system.

An alternative method is to solve *by elimination*: removing one variable—rather than equation—at a time from the system. Elimination theory is strongly supported by Gröbner bases computation; see, e.g., [61, Ch. 3]. Wu's characteristic set method also proceeds by elimination [47]. However, incremental solving has several algorithmic advantages. Firstly, the size of the equations and components involved at each step is much smaller. Secondly, several algebraic properties are maintained, leading to theoretical and algorithms improvements [47, 142]. Thirdly, and a main focus in this research, concurrency arises by processing each component independently and in parallel. Indeed, solving an equation of the input system against one component of the current solution set may produce possibly many components, leading to further parallelism.

1.2 Objectives and Contributions

This thesis examines high-performance, software design, and implementation challenges within symbolic computation. The unifying motivation of this thesis is the development of an efficient polynomial system solver for modern computer architectures. In this work we examine solving systems of polynomial equations by means of triangular decomposition and regular chains. We develop practical algorithmic and implementation techniques for solving these systems. The final product of this work is an open-source, high-performance, parallel polynomial system solver. This will be the first multi-threaded system solver based on regular chains.

Towards that goal, we have designed and implemented an efficient polynomial algebra library: the Basic Polynomial Algebra Subprograms (BPAS) library [7]. Indeed, system solving requires a wide range of functionalities from a general-purpose computer algebra system. This library now contains over 600,000 lines of code, contributed by several au-

thors [7]. It supports univariate and multivariate polynomials over prime fields, integers, and rational numbers [11], subresultant chains [13], Fast Fourier Transforms (FFTs) [60], power series [32, 33], and, of course, polynomial system solving [9, 12].

The BPAS library, and our polynomial system solver, are implemented in the C/C++ languages. These low-level and powerful languages have allowed us to achieve high-performance and sophisticated parallelism in our implementation. The library's core is implemented in C, for performance, and wrapped in a C++ interface for usability. The object-oriented paradigm of C++ provides a suitable environment to apply good software design practices to produce well-structured and maintainable code. Further, the ability to encapsulate complex C code behind a C++ class interface improves end-user usability and reduces coupling for improved maintainability.

```
#include <bpas.h>

int main(int argc, char** argv) {
    SMQP p1("x^2 + y + z - 1");
    SMQP p2("x + y^2 + z - 1");
    SMQP p3("x + y + z^2 - 1");
    std::vector<SMQP> F = {p1, p2, p3};

    for (RegularChain<SMQP>& rc : triangularize(F)) {
        std::cout << rc << std::endl;
    }

    return 0;
}
```

Listing 1.1: A sample program to compute a triangular decomposition in BPAS.

Consider Listing 1.1, showing the source code of a possible user's program to solve the system of equations F_2 from the previous subsection. The object-oriented interface for sparse multivariate polynomials with rational number coefficients (`SMQP`) and for regular chains (`RegularChain<SMQP>`) makes calling these complex algebraic functions quite easy. The output of executing the program shown in Listing 1.1 is then:

```
{x, y - 1, z}
{x, y, z - 1}
{x - 1, y, z}
{x - z, y - z, z^2 + 2*z - 1}
```

Motivated by a high-performance polynomial system solver, this thesis provides four main contributions to computer algebra, high-performance computing, and software engineering:

1. the object-oriented design of a polynomial algebra library,
2. object-oriented support for multithreaded parallelism,
3. high-performance and parallel triangular decomposition, and
4. lazy and parallel power series and Hensel factorization.

Object-oriented design of a polynomial algebra library

Computer algebra systems (CAS) are used for a variety of purposes, ranging from education to prototyping research software to scientific computing. Their accessibility and ease of use for non-expert users are true challenges. Developers must consider the characteristics of the implementation environment, the level of expertise of the users, and the expectations of its user community.

In the world of computer algebra software there are two main categories. The first is computer algebra systems, self-contained environments providing an interactive user-interface, and usually their own programming language. Custom interpreters and languages yield powerful functionality and expressibility, however, obstacles remain. For a basic user, they must learn yet another programming language and environment. For an advanced user, interoperability and obtaining fine control of hardware resources is typically challenging. The second category is computer algebra libraries, which add support for symbolic computation to an existing programming environment. Since such libraries extend existing environments, they can have a lower barrier to entry and better accessibility. On the other hand, they often lack the expressibility of dedicated CAS.

The BPAS library looks to improve the efficiency of end-users through both usability and performance, providing high-performance code along with an interface which incorporates some of the expressibility of a custom computer algebra system. BPAS follows two driving principles in its design.

(D_1) Encapsulate as much complexity as possible on the developer's side, where the developer's intimacy with the code allows her to bear such a burden. This leaves the end-user's code as clean as possible.

(D_2) "Make it hard to do the wrong thing", a common phrase in *user experience (UX) design*.

The conventions of encapsulation and information hiding in object-oriented design naturally work toward the first design principle. So too does our high-performance implementation of foundational routines in the BPAS library. Toward the second design principle, we have employed *template metaprogramming* to enable two key features: (*i*) compile-time type safety for algebraic structures and elements of their underlying set, and (*ii*) mechanisms to "dynamically" create types from the composition of others. In the former, "algebraic type safety"—the compatibility of elements from two different algebraic structures—is traditionally only determined at runtime. Of course, determining errors and bugs at compile-time is much more desirable for code correctness. In the latter, this composition is best illustrated by the creation of a polynomial type from some coefficient ground ring. Our mechanisms allow for the polynomial type to enforce properties, at compile-type, of its coefficient ring. It also allows the polynomial type to modify its interface (its exposed methods) based on the actual type of the coefficient ring. This topic is explored further in Chapter 4.

Object-oriented and generic support for multithreaded parallelism

Further towards the first design principle, our second contribution is the object-oriented design and implementation of multithreaded parallelism. This module, part of the BPAS library, encapsulates many difficulties of parallel programming behind class interfaces. This includes synchronization, deadlock avoidance, load-balancing, and mitigating parallel overheads (see Section 3.2 for details on these terms).

Our work is motivated by implementing parallelism in triangular decomposition, and in particular, supporting its irregular component-level parallelism. Recall that irregular parallelism describes cases where workloads are imbalanced or where opportunities for parallelism are not guaranteed by the algorithm and must be found dynamically during execution. There are two main questions for irregular parallel applications. How can load-balancing be best achieved? How can these applications scale beyond the inherent limits imposed by the particular problem instance being solved?

This parallelism module extends the multithreading primitives (threads, locks, mutexes) of the C++11 *Thread Support Library* to provide generic and reusable support for more sophisticated parallelism. In particular, support for various *parallel patterns*—algorithmic skeletons for organizing parallel code for efficient execution. We have implemented map, workpile, producer-consumer, pipeline, and fork/join. We have discovered that one effective way to support irregular parallelism is through the composition and cooperation of parallel regions and dynamic scheduling. This allows for dynamic load-balancing between parallel regions: fine-grained parallelism can be exploited within

coarse-grained parallel tasks. Moreover, if the lower-level, fine-grained parallelism exhibits regular parallelism, it can improve the overall scalability of the irregular parallel application.

Our module follows the notions of *dynamic multithreading*: programmers specify the areas where parallelism is possible, but without requiring it. Then, the runtime dynamically decides whether or not to execute a code region in parallel based on current resource usage and competing code regions. This is enabled through the *functional* module of C++11, which is used to elevate functions to first-class objects. Our object-oriented parallelism support employs function objects as the basic object to capture functionality and pass it between code regions and threads. We discuss this work in Chapter 5. This module supports the parallel implementation found in our next two contributions: a polynomial system solver, and lazy power series.

A high-performance polynomial system solver

We have implemented a polynomial system solver based on triangular decomposition and regular chains. Many aspects work harmoniously to make this implementation high-performance. We must consider data structure design for memory consumption and data locality (cache complexity), state-of-the-art algorithms with improved computational complexity, implementation techniques for practical performance, and opportunities to exploit parallelism. The solver is part of the BPAS library.

The foundation of the solver is in the implementation of polynomial data types, arithmetic, and the conversion between types. We have previously implemented sparse multivariate polynomials, with integer or rational number coefficients, and their arithmetic [11, 31]. This has informed the implementation of univariate and bivariate polynomials over the integers and prime fields which, in turn, serve as foundations for efficient polynomial GCD and factorization implementations. Indeed, the computation of GCDs and factorization is fundamental for triangular decompositions to achieve practical performance. We have implemented GCD and factorization routines based on algorithmic advancements in *Hensel lifting* [136, 170], which has lead to significant performance improvements.

Another foundational routine toward triangular decomposition is computing resultants and subresultant chains (see Section 2.4 for a brief description). Essentially, a resultant gives conditions for two polynomials to share a solution. Repeated computations of resultants are thus used within triangular decomposition to solve a system of equations. We have investigated and developed effective practical schemes for computing subresultant chains which consider both cache complexity and parallelism [9, 13].

These data structures and algorithms support a highly efficient implementation of triangular decomposition based on regular chains. The implementation of regular chains and the triangular decomposition algorithm itself in C/C++, as part of the BPAS library, also provides a suitable compiled environment to obtain additional performance.

Towards additional performance, we have investigated and exploited the component-level parallelism of triangular decomposition. This begins with an investigation of the opportunities for concurrency available in the many sub-routines of triangular decomposition. These concurrency opportunities are non-trivial since they exhibit irregular parallelism, and may or may not provide parallelism depending on the particular problem being solved. By employing our object-oriented parallel support, our implementation is able to compose many regions of parallelism to both load-balance the parallel regions while also finding and exposing further parallelism. Chapter 6 details the algorithms and implementation techniques of our system solver.

Extensive experimentation complements these algorithms and implementation, providing many insights into the practical performance of these various techniques. We have assembled a test suite over 3000 systems of polynomial equations coming from the scientific literature as well as from user-data and bug reports provided by *MapleSoft* and the *RegularChains* library [125]. Many of these systems have also been collected into other test suites; see [24] and [173]. The entire collection of systems may be downloaded from the BPAS website [7].

In total, this test suite contains 3193 systems. Limiting serial computation time to 3 hours, we are able to solve 2815 of these systems. Specifically, 2815 can be solved in the sense of Kalkbrener, and 2793 can be solved in the sense of Lazard and Wu; see Definition 2.35. Those 2815 systems have the following properties.

- 1076 systems have more than one component in the final triangular decomposition.
- 1249 systems are inconsistent (have no solution).
- 360 systems are zero-dimensional.
- 1206 systems are positive-dimensional. The dimension of a system is the maximum dimension of any component in the triangular decomposition. These 1206 systems have the following distribution of dimension:

Dimension	1	2	3	4	5	6	7	8	9	10	12	18
Num. of Sys.	592	239	138	91	69	33	10	12	14	1	6	1

- The number of equations in each system ranges from 2 to 29. The 2815 systems we can solve have the following distribution of number of equations:

Num. Eqs.	2	3	4	5	6	7	8	9	10	11	12
Num. Sys.	1356	156	115	578	234	105	85	48	49	19	16
Num. Eqs.	13	14	15	16	17	18	19	20	24	26	29
Num. Sys.	13	7	9	5	6	7	0	1	2	2	1

- The number of components in the Kalkbrener decomposition of each system ranges from 0 to 96. The 2815 systems have the following distribution of number of components:

Num. Comps.	0	1	2	3	4	5	6	7	8	9
Num. of Sys.	1249	491	631	185	80	46	28	27	19	11
Num. Comps.	10	11	12	13	14	15	16	17	18	19
Num. of Sys.	5	4	5	7	1	8	2	1	2	3
Num. Comps.	23	24	29	35	37	40	48	56	83	96
Num. of Sys.	1	1	2	1	1	1	1	1	1	1

- The number of components in the Lazard-Wu decomposition of each system ranges from 0 to 119. The 2793 systems have the following distribution of number of components:

Num. Comps.	0	1	2	3	4	5	6	7	8	9	10
Num. of Sys.	1248	363	631	188	96	57	36	34	27	19	5
Num. Comps.	11	12	13	14	15	16	17	18	19	20	21
Num. of Sys.	8	4	9	5	11	4	4	3	5	1	0
Num. Comps.	22	23	24	25	26	27	29	33	36	37	39
Num. of Sys.	1	2	1	1	1	5	4	1	1	2	1
Num. Comps.	40	48	56	58	61	77	79	83	87	99	119
Num. of Sys.	1	1	3	1	2	1	2	1	1	1	1

Where examples have multiple components, parallel speed-up reaches up to 10.8× on a 12-core machine (24 physical threads with hyperthreading). Where examples do not have multiple components, our parallel schemes do not add considerable overhead, with very few non-trivial examples experiencing slowdown. Indeed, some parallelism can be exploited even when there is no available component-level parallelism. For example, in the computation of subresultant chains (see Section 6.3.3).

Experimental results are detailed later in Section 6.4. Here, we present a short summary. First, we consider serial execution and compare our implementation against the most modern triangular decomposition algorithm: *RegularChains* of *Maple 2020* [125]. Across the 2815 systems of our test suite which are solveable by our implementation, 1 is *not* solveable by *Maple*. To compute a Kalkbrener decomposition (resp. a Lazard-Wu decomposition), our implementation is on average 7.53× (7.54×) faster. Only 63 (58) systems are faster to solve in *Maple*. Of those 63 (53) systems, *Maple* is 1.91× (1.82×) faster on average, and up to 29.06× (34.16×) faster. Of the 2751 (2756) systems for which our implementation is faster, our implementation is 7.69× (7.68×) faster on average, and up to 36.43× (36.43×) faster.

Considering now parallelism, we have many different and simultaneous areas of parallelism in our implementation. Those parallel regions must cooperate and effectively share resources to achieve load-balance and optimal speed-up. The following reports on the configuration with parallel triangualrize tasks, parallel removal of redundant components, and parallel subresultant chains; see Sections 6.3 and 6.4 for details. On a compute node with 12 cores (24 physical threads with hyperthreading) we achieve the following parallel speed-ups. To compute a Kalkbrener decomposition (resp. a Lazard-Wu decomposition) the maximum speed-up is 10.22× (10.47×). For non-trivial systems which require at least 100ms to solve, of which there are 268 (293) such systems, the average parallel speed-up is 2.14× (2.12×).

Lazy, yet high-performance, power series and Hensel factorization

When solving systems of polynomial equations, some of the solutions are known as "generic zeros" and are much easier to compute than all solutions of the system. Solutions which are not generic may be called *non-trivial limit points* (see Section 2.6 for a more specific description). It has been observed [4, 5] that, given the generic zeros of a system of equations, one can compute their limit points using Puiseux series or the *Extended Hensel Construction*. The Extended Hensel Construction also has applications in finding limits of multivariate rational functions, and computing the real branches of space curves [4].

Towards an efficient implementation of the Extended Hensel Construction, and eventually to effectively computing non-trivial limit points, we investigate and develop an efficient implementation of *Hensel factorization*. Hensel factorization is a special case of the Extended Hensel Construction which computes the roots of univariate polynomials with multivariate power series coefficients. This, in turn, requires an efficient implementation of multivariate power series.

Power series are polynomial-like objects with, potentially, an infinite number of terms. The fact that they may have an infinite number of terms presents interesting implementation challenges. How can they be represented on a computer with finite resources? How can we perform arithmetic operations effectively and efficiently with them? One natural representation is to encode them as a function, or *generator*, which computes terms of the power series as needed. This point of view leads to arithmetic operations and an implementation based on *lazy evaluation*. Such a scheme makes intelligent use of previously computed terms to avoid recomputing them and to more efficiently compute new terms. To the best of our knowledge, this is the first implementation of multivariate power series in a compiled language.

These lazy power series are then employed in a parallel implementation of Hensel factorization and Weierstrass Preparation Theorem. Hensel factorization is another example of irregular parallelism in computer algebra. Our variation of Hensel factorization, applied to a polynomial with power series coefficients, computes roots of the polynomial one at a time, where each successive root depends on the previous, but is also more easy to compute. Nonetheless, we have implemented a *parallel pipeline* which simultaneously computes all of these successive roots, despite their dependencies. This work includes a complexity analysis which shows that our method improves previous methods by at least a log factor. Moreover, the complexity estimates are shown to be useful to help dynamically load-balance the irregular parallel implementation, exemplifying how complexity analyses can be used to improve practical parallel performance. Our implementation achieves a parallel speed-up of up to 9× on a 12-core machine.

1.3 Organization of this Thesis

This thesis is organized into several chapters, each examining a contribution detailed in the previous section. The logical dependencies between the chapters of this thesis are summarized by the flowchart in Figure 1.1

We begin, however, with two background chapters. The work presented in later chapters draws on distinct, and often disparate, areas of knowledge. On one hand we have algebraic geometry and commutative algebra, while on the other we have high-performance computing, software engineering, and template metaprogramming. In Chapter 2 we present the necessary mathematical background, giving definitions and notations for polynomials, ideals, varieties, and regular chain theory. In Chapter 3 we present the necessary computational background, including data locality, multithreading, and modern aspects of C++.

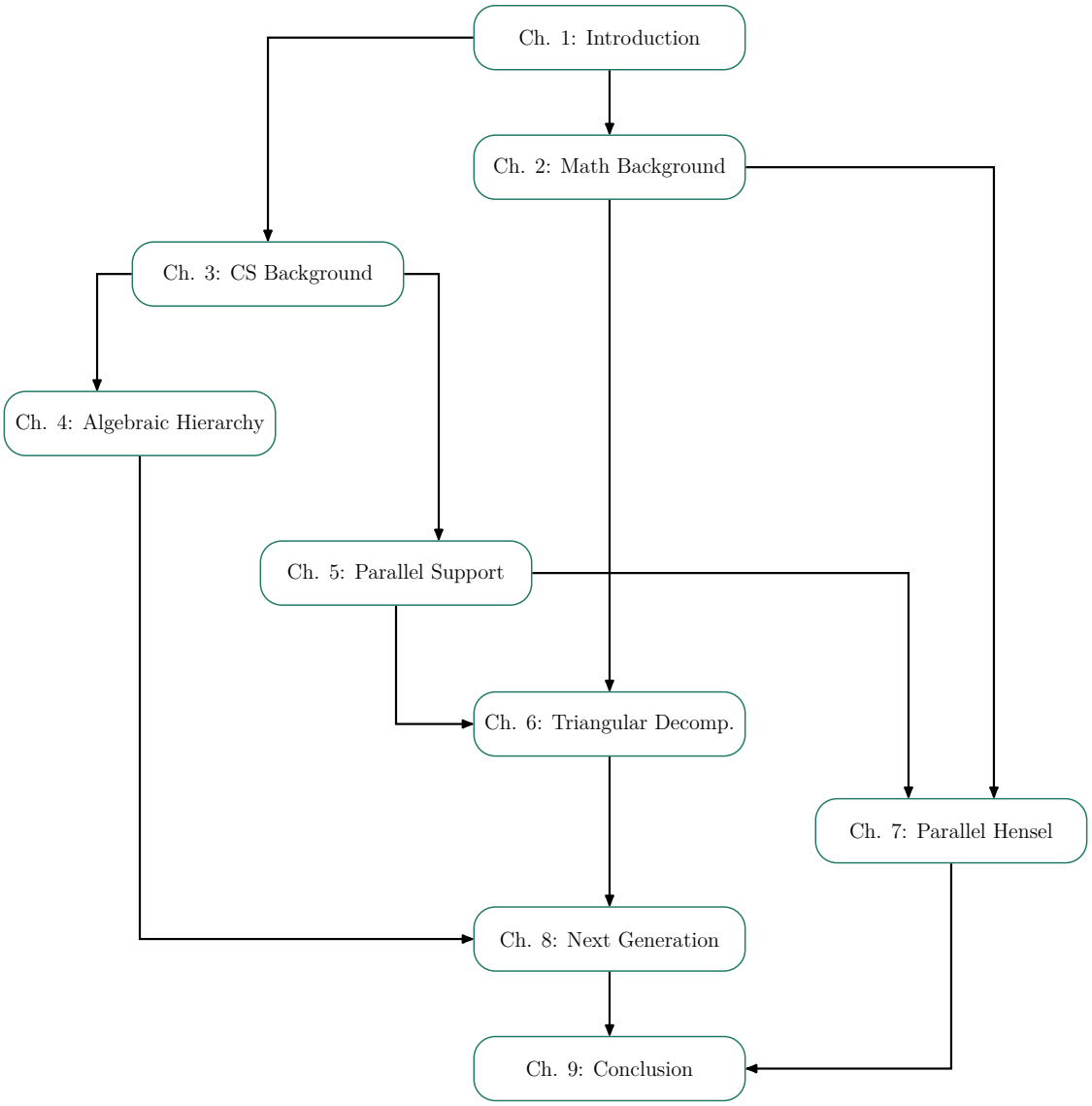

Figure 1.1: The logical dependence of thesis chapters.

The body of this thesis is composed of five chapters. In Chapter 4, we discuss the design of the Basic Polynomial Algebra Subprograms library and compare it against other computer algebra libraries. In particular, we discuss the compile-time improvements, namely type safety, made possible using template metaprogramming. We examine the design of algebraic structures as an object-oriented class hierarchy.

In Chapter 5 we explore parallel patterns and their application to irregular parallelism. These parallel patterns are implemented in a object-oriented manner to support ease-of-use of developers who wish to integrate parallelism into their own applications. Moreover, the module is designed with cooperation and composition of parallel regions in mind to better dynamic load-balance.

Dynamic load-balance is exemplified as one practical aspect of a high-performance polynomial system solver. The implementation of this solver is detailed in Chapter 6. We describe the design and implementation of low-level data structures and algorithms for polynomials, polynomial GCDs and polynomial factorization. From this, we establish efficient and parallel routines for computing subresultants. These methods together form the foundation of our regular chains and triangular decomposition implementation. Applying the parallel support described in Chapter 5, we are able to exploit the irregular component-level parallelism of triangular decomposition. Extensive experimentation demonstrates that this parallelization can provide excellent performance increases.

Next, we examine another application of our parallel support in multivariate power series and Hensel factorization. Chapter 7 describes the design and implementation of multivariate power series based on lazy evaluation. Algorithms for lazy Hensel factorization and lazy Weierstrass preparation are then derived, and their complexity analyzed. The complexity estimates are applied to a parallel implementation of both algorithms to dynamically load-balance them.

The experiences gained in implementing parallel triangular decomposition and parallel Hensel factorization have provided an better informed view on the difficulties of redundant computations and irregular parallelism in the context of computer algebra. We reflect on these challenges in Chapter 8. Therein, we discuss the design of a next-generation triangular decomposition implementation which incorporates the ideas of dynamic evaluation to avoid redundant computation. We have designed a data structure and framework for a regular chain "universe" to support dynamic evaluation, splitting trees, and further parallelism in triangular decomposition.

Finally, we summarize the accomplishments of this thesis in Chapter 9. There, we make concluding remarks and discuss areas for future work, unsolved problems, and new research directions we have discovered during this work.

Chapter 2

Mathematical Background

In this chapter we review mathematical foundations relevant to solving systems of polynomial equations via triangular decomposition. We begin in Section 2.1 reviewing definitions and theorems in commutative algebra concerning rings, ideals, and polynomials. Section 2.2 reviews an important result known as the *D5 Principle* and the related notion of dynamic evaluation. Then, Section 2.3 relates polynomials ideals with their corresponding geometric object: varieties. Section 2.4 reviews subresultant theory, a crucial operation in triangular decomposition. Then, Section 2.5 defines solving systems of polynomial equations more formally, including triangular decomposition, regular chains, and the main theorems and results regarding those objects. Finally, in preparation for Chapter 7, limits points are defined, and the basics of power series described in Section 2.6.

2.1 Commutative Rings

A *commutative ring with identity* is a set R endowed with two binary operations, denoted $+$ and $\times$. For clarity, let us call these operations addition and multiplication, respectively, but they need not be addition and multiplication in the usual sense. In particular, these operations need only satisfy the following conditions.

Definition 2.1 (Commutative Ring)**.** *A set R endowed with two operations $+$ and $\times$ is a* commutative ring with identity *if:*

(i) *R is a commutative group under $+$ with identity 0 (i.e. addition is associative, commutative, and every element has an additive inverse);*

(ii) *$\times$ is associative and commutative, with identity 1; and*

(iii) *$\times$ is distributive over $+$: $\forall\, a, b, c \in R\ a \times (b + c) = (a \times b) + (a \times c)$.*

In this work we assume that all rings are commutative with a multiplicative identity. Henceforth we call them simply *rings*. Moreover, as is common with multiplication, the operation $\times$ is often implicit, such as: $a \times b = ab$.

A *ring homomorphism* is a map between rings which preserves ring operations. Let $\Phi : R \to S$ be a such a map. Then, Φ is an homomorphism if, for all $r, s \in R$, $\Phi(r+s) = \Phi(r) + \Phi(s)$, $\Phi(r\,s) = \Phi(r)\,\Phi(s)$, and $\Phi(1) = 1$. If Φ is a bijection, then its inverse is also a homomorphism and Φ is a *ring isomorphism*. The two rings R and S are then said to be *isomorphic*, and we write $R \cong S$. Isomorphic rings are *algebraically indistinguishable*.

Rings are very general objects. By extending the definition of a ring and imposing additional constraints, we obtain different (ring-like) *algebraic structures* or *algebraic types* with different properties. As we will see, these algebraic structures form a hierarchy called *the hierarchy of rings*. In the general context of a ring, one might encounter a *zero divisor*. A non-zero element $a \in R$ is a zero divisor if, for some $b \neq 0 \in R$, $a \times b = 0$. Another kind of element is a *unit*, or *invertible element*: an element $a \in R$ such that there exists $b \in R$ with $ab = 1$. Zero divisors are necessarily not units.

Two elements $a, b \in R$ are said to be *associate* if $a = ub$ for a unit $u \in R$. It is then useful to distinguish some normal form $\text{normal}(a) \in R$ such that for every $a \in R$ $\text{normal}(a)$ and a are associates. That is, $a = u\,\text{normal}(a)$ for some unit $u \in R$. Two elements of R are associate if and only if they have the same normal form and the normal form of a product is equal to the product of the normal forms. Hence, for any unit $u \in R$, $\text{normal}(u) = 1$. For example, the integers have a natural form, the absolute value.

Yet another kind of element is a *regular element*: an element $a \in R$ which is neither zero nor a zero-divisor. Notice that regular elements need not be invertible. The existence of zero divisors poses great difficulty in theory and practice. We thus give a special name to rings without zero divisors.

Definition 2.2 (Integral domain)**.** *A set $\mathbb{D}$ is an* integral domain*, or simply* domain*, if it is a ring and:*

(iv) $\mathbb{D}$ contains no zero-divisors.

Integral domains are natural algebraic structures for studying divisibility and exact division without worrying about zero divisors. Indeed, if $\mathbb{D}$ is an integral domain and $a, b, q, q' \in \mathbb{D}$ such that $b \neq 0$, $a = b\,q$, and $a = b\,q'$, then we must have $q = q'$. That is, if $b \mid a$ then the quotient is uniquely defined.

One may extend divisibility slightly with the notion of a *greatest common divisor* (GCD). A GCD of two elements $a, b \in \mathbb{D}$ is a third element $c \in \mathbb{D}$ such that $c \mid a$, $c \mid b$, and any other common divisor of a and b divides c. With this property in mind, we arrive at a *GCD domain*.

Definition 2.3 (GCD domain)**.** *A set* $\mathbb{D}$ *is a* GCD domain *if it is an integral domain and:*

(*v*) *any two non-zero elements of* $\mathbb{D}$ *have a* greatest common divisor*.*

Notice that GCDs may not be unique. Indeed, the GCD of 4 and 6 over the integers $\mathbb{Z}$ is ± 2. If we have a normal form, then $\mathrm{GCD}(a, b)$ for $a, b \in \mathbb{D}$ can be defined as the unique normalized associate of all GCDs for a and b.

We can once more extend the idea of a GCD to that of an *irreducible factorization*. A non-unit a of a ring R is said to be reducible if there exists non-units $b, c \in R$ such that $a = b\,c$. Indeed, irreducible elements correspond to *prime elements* in GCD domains [86, Theorem 25.3]. Then, an irreducible factorization of an element $a \in R$ is the equivalent product of irreducible elements (up to reordering and multiplication by units). This leads us to a *unique factorization domain* (UFD).

Definition 2.4 (Unique Factorization Domain)**.** *A set* $\mathbb{D}$ *is a* unique factorization domain *if it is a GCD Domain and:*

(*vi*) *every element of* $\mathbb{D}$ *has a unique* irreducible factorization

A fundamental example of UFDs where GCDs can be effectively computed are *Euclidean domains*.

Definition 2.5 (Euclidean domain)**.** *A set* $\mathbb{D}$ *is a* Euclidean domain *if it is a UFD where a* Euclidean function $f : \mathbb{D} \to \mathbb{N}$ *exists satisfying:*

(*vii*) *for all* $a, b \in \mathbb{D}$*,* $b \neq 0$*, there exists* $q, r \in \mathbb{D}$ *such that* $a = bq + r$ *and either* $r = 0$ *or* $f(r) < f(b)$*.*

This property states that computing a *division-with-remainder* or *Euclidean division* is always possible in a Euclidean domain. We say that $q = a$ quo b is the quotient, and $r = a$ rem b is the remainder. Moreover, the classic *Euclidean algorithm* (see Section 2.1.2 and, e.g., [112, Ch. 4]) may be executed in a Euclidean domain to effectively compute GCDs. As we will see later in Sections 2.1.2 and 2.2, computing GCDs in non-Euclidean domains is a challenging but important problem.

Finally, we arrive at our last algebraic structure of interest, the *field*.

Definition 2.6 (Field)**.** *A set* $\mathbb{K}$ *is a* field *if it is a Euclidean domain and:*

(*viii*) *every non-zero element* $a \in \mathbb{K}$ *is a* unit*.*

This is a powerful property and it means that every element is divisible by every non-zero element in $\mathbb{K}$. Equivalently, all divisions result in a 0 remainder. Throughout this work we only consider *perfect fields* (see [86, Section 14.6]), fields like the rational numbers

$\mathbb{Q}$, the complex numbers $\mathbb{C}$, or any finite field, as well as any algebraically closed field or any extension field of $\mathbb{Q}$. These latter two terms are defined in the following subsections.

A fundamental example of field construction is the field of fractions of an integral domain $\mathbb{D}$. It is the smallest field which contains $\mathbb{D}$; it may be defined as the set $\{a \,/\, b \mid a, b \in \mathbb{D}, b \neq 0\}$ and is denoted $\mathrm{Frac}(\mathbb{D})$. This is a natural generalization of the construction of the field $\mathbb{Q}$ of rational numbers from the ring $\mathbb{Z}$ of integer numbers.

Notice that we have built the definitions of these algebraic structures through continuously adding properties (*i*) through (*viii*). Indeed, these algebraic types form a strict class inclusion:

$$\text{ring} \supset \text{integral domain} \supset \text{GCD domain} \supset \text{UFD} \supset \text{Euclidean domain} \supset \text{field}.$$

One subject to be studied in this work is the encoding of this class inclusion as an object-oriented class hierarchy. This is discussed later in Chapter 4. For now, we will continue our discussion on commutative rings with the introduction of the fundamental concept of ideals.

2.1.1 Ideals

Ideals are foundational algebraic objects which give us a language to discuss zeros and elements which are sufficiently similar to zero. This becomes evident considering quotient rings and, later in Section 2.3, polynomial ideals and varieties.

Definition 2.7 (Ideal)**.** *An* ideal $\mathcal{I}$ *of a ring* R *is a subset of the ring satisfying:*

(i) $0 \in \mathcal{I}$*;*

(ii) For every $f, g \in \mathcal{I}$*,* $f + g \in \mathcal{I}$*; and*

(iii) For every $r \in R$ *and every* $f \in \mathcal{I}$*,* $r \times f \in \mathcal{I}$*.*

For $f_1, \ldots, f_k \in R$, the *ideal generated by* $f_1, \ldots, f_k$ is the set:

$$\langle f_1, \ldots, f_k \rangle = \left\{ \sum_{i=1}^{k} r_i f_i \;\middle|\; r_1, \ldots, r_k \in R \right\}$$

We say that $\{f_1, \ldots, f_k\}$ is the *generating set* of the ideal. Note that the generating set of an ideal is not unique.

Any ideal which contains the multiplicative identity 1 is the *unit ideal* and is equal to the ring itself. A *proper ideal* is an ideal which does not contain 1 and is thus a proper subset of its ring.

Ideals admit certain useful operations, in particular, sum, product, intersection, and radical. Let $\mathcal{I}$ and $\mathcal{J}$ be two ideals of the same ring.

The sum of $\mathcal{I}$ and $\mathcal{J}$ is the ideal:

$$\mathcal{I} + \mathcal{J} = \{f + g \mid f \in \mathcal{I} \text{ and } g \in \mathcal{J}\}.$$

The product of $\mathcal{I}$ and $\mathcal{J}$ is the ideal:

$$\mathcal{I}\,\mathcal{J} = \left\{ \sum_{i=1}^{k} f_i g_i \;\middle|\; k \in \mathbb{Z}^+,\ f_i \in \mathcal{I},\ g_i \in \mathcal{J} \right\}.$$

The intersection of $\mathcal{I}$ and $\mathcal{J}$ is the ideal:

$$\mathcal{I} \cap \mathcal{J} = \{f \mid f \in \mathcal{I} \text{ and } f \in \mathcal{J}\}.$$

The radical of $\mathcal{I}$ is the ideal:

$$\sqrt{\mathcal{I}} = \left\{ f \mid \exists\, e \in \mathbb{Z}^+,\ f^e \in \mathcal{I} \right\}.$$

An ideal $\mathcal{I}$ is said to be radical if $\mathcal{I} = \sqrt{\mathcal{I}}$. Moreover, the radical of the intersection of ideals is the intersection of their radicals: $\sqrt{\mathcal{I} \cap \mathcal{J}} = \sqrt{\mathcal{I}} \cap \sqrt{\mathcal{J}}$.

An important consequence of an ideal $\mathcal{I} \subseteq R$ is that it induces an equivalence relation $\sim$ on R. For $a, b \in R$, $a \sim b \iff a - b \in \mathcal{I}$. For $a \sim b$ we say that a and b are *congruent modulo* $\mathcal{I}$. For any $a \in R$, its equivalence class under this relation is the set $[a] = \{a + f \mid f \in \mathcal{I}\}$. One may write $[a] = a + \mathcal{I}$. This equivalence class may also be called the *residue class of* a *modulo* $\mathcal{I}$.

The set of all such equivalence classes itself forms a ring, called the *quotient ring* or *residue class ring*, and is denoted by $R/\mathcal{I}$. In a quotient ring, the additive identity (zero element) is $[0] = \mathcal{I}$ and the multiplicative identity is $[1] = 1 + \mathcal{I}$. Addition and multiplication are defined respectively as $[a] + [b] = [a + b]$ and $[a]\,[b] = [a\,b]$.

More generally, a quotient ring may form an integral domain or a field depending on the properties of the ideal. A proper ideal $\mathcal{I} \subsetneq R$ is a *prime ideal* if for any $a, b \in R$, $a\,b \in \mathcal{I}$ implies $a \in \mathcal{I}$ or $b \in \mathcal{I}$. $R/\mathcal{I}$ is an integral domain if and only if $\mathcal{I}$ is a prime ideal. A proper ideal $\mathcal{I} \subsetneq R$ is a *primary ideal* if for any $a, b \in R$, $a\,b \in \mathcal{I}$ implies $a \in \mathcal{I}$ or $b \in \mathcal{I}$ or $a, b \in \sqrt{\mathcal{I}}$. Clearly, the radical of a primary ideal is a prime ideal. A proper ideal $\mathcal{I} \subsetneq R$ is a *maximal ideal* if, for any proper ideal $\mathcal{J} \subsetneq R$, $\mathcal{I} \subseteq \mathcal{J} \subsetneq R$ implies that $\mathcal{I} = \mathcal{J}$. $R/\mathcal{I}$ is a field if and only if $\mathcal{I}$ is a maximal ideal [171, Ch. 3].

Example 2.8 (Integers modulo 5)**.** The quotient ring $\mathbb{Z}/\langle 5\rangle$ (often denoted $\mathbb{Z}/5\mathbb{Z}$ or $\mathbb{Z}_5$) is a *finite field* consisting of the equivalence classes of integers modulo 5: $0, 1, 2, 3,$ and 4. It is easy to check that $\mathbb{Z}/\langle 5\rangle$ is a field since the non-zero elements $1, 2, 3, 4$ have respective multiplicative inverses $1, 3, 2, 4$. Thus, $\langle 5\rangle$ is a maximal ideal.

A *Noetherian* ring is a ring R in which every proper ideal $\mathcal{I}$ is the intersection of finitely many primary ideals. Such a primary decomposition is a *minimal primary decomposition* if the radicals of the primary ideals are all unique. Every proper ideal in a Noetherian ring has a minimal primary decomposition [150]. The radicals of those primary ideals are called the *associated primes* of $\mathcal{I}$. A property which will become useful later with respect to regular chains, is that an element $a \in R$ is regular in $R/\mathcal{I}$ if and only if a does not belong to any of the associated prime ideals of $\mathcal{I}$; this is exemplified in Example 2.14.

In order to explore more interesting ideals and residue class rings for the purpose of solving systems of equations, we must first introduce polynomial rings.

2.1.2 Polynomial Rings

A (univariate) *polynomial ring* is formed by taking arbitrary length vectors of some *base ring*, *ground ring*, or *coefficient ring*, say R. Elements of a polynomial ring, *polynomials*, are vectors of the form $(a_0, a_1, \ldots)$ where $a_0, a_1, \ldots \in R$ and all but a finite number of a_i are zero. A polynomial ring is itself a ring since we can define addition as $(a_0, a_1, \ldots) + (b_0, b_1, \ldots) = (a_0{+}b_0, a_1{+}b_1, \ldots)$, and multiplication as $(a_0, a_1, \ldots)\,(b_0, b_1, \ldots) = (c_0, c_1, \ldots)$, where $c_i = \sum_{j=0}^{i} a_j\, b_{i-j}$.

More typically, we write these vectors as a sum of products of some *variable*, say x, $a_0{+}a_1x{+}a_2x^2{+}\ldots$, giving us the usual notation of a polynomial. We denote a polynomial ring as a combination of its ground ring and its variable: $R[x]$. Some key aspects of a polynomial include: its *terms*, the products $a_0, a_1x, a_2x^2, \ldots$; its *coefficients*, the elements $a_0, a_1, \ldots$; its *degree*, the maximum d such that $a_d \neq 0$; and its *leading coefficient* and *leading term*, a_d and a_dx^d, where d is its degree. The *reductum* of a polynomial p is the polynomial resulting from removing the leading term of p. A *root* of a polynomial is a value on its variable which makes the polynomial evaluate to 0.

The *content* of a polynomial p, denoted $\mathrm{cont}(p)$, is the (unique normalized) GCD of all of its coefficients in R. Then, the *primitive part* of p is $p/\mathrm{cont}(p)$.

Polynomial rings can be used to construct *extension fields*—fields which are supersets of another field—through quotient rings. Let $\mathbb{K}$ be a field. Given an irreducible polynomial $p \in \mathbb{K}[x]$, $\mathbb{K}[x]/\langle p\rangle$ is isomorphic to the field which adjoins the roots of p to $\mathbb{K}$. For example, if $p = x^2 + 1 \in \mathbb{R}[x]$, then $\mathbb{R}[x]/\langle p\rangle$ is the set of real numbers adjoined with $\pm i$.

This is, of course, isomorphic to the complex numbers $\mathbb{C}$. Extension fields of this kind which extend the rational numbers $\mathbb{Q}$ are called *algebraic number fields*.

For polynomials over a field $\mathbb{K}$, we say that $\mathbb{K}$ is *algebraically closed* if every polynomial in $\mathbb{K}[x]$ has a root in $\mathbb{K}$. The *algebraic closure* of $\mathbb{K}$, denoted $\overline{\mathbb{K}}$, is an extension field of $\mathbb{K}$ that is algebraically closed. For example, the set of real numbers $\mathbb{R}$ is not algebraically closed since $x^2 + 1$ has no root in $\mathbb{R}$; the algebraic closure of $\mathbb{R}$ is $\mathbb{C}$.

One may also define a polynomial ring with two or more variables, for example $R[x, y]$. Elements of this polynomial ring are seen as a linear combination between elements of R and *monomials*—some product of the variables. For example, $R[w, z, y, x]$ is a polynomial ring with four variables. $p = 4x^2z + x^2y + 3xzw + 5z^2 + 3w$ is one element of this ring with monomials x^2z, x^2y, xzw, z^2, w and coefficients $4, 1, 3, 5, 3$. Generally, we may define a polynomial ring with a finite number n of variables as $R[x_1, \ldots, x_n]$. Throughout this work we will assume all polynomial rings have a finite number of variables.

In some contexts, it is important to give an ordering to the variables and view the polynomial *recursively*. Since polynomial rings are themselves rings, they admit a recursive construction where the ground ring itself is a polynomial ring. Under the variable ordering $w < z < y < x$, we can view the previous polynomial ring as univariate in the *main variable* x and having the variables w, z, y in the coefficient ring. We make this view explicit with the notation $R[w, z, y][x]$. Then, p may be re-written as: $(4z + y)x^2 + (3zw)x + (5z^2 + 3w)$.

In this recursive view, the degree in the main variable is called the *main degree*, the main variable raised to the main degree is the *rank*, the leading coefficient (which is itself a polynomial) is called the *initial*, and the *tail* is the difference between the polynomial and its initial times its rank. The main degree, rank, initial, and tail of p are, respectively, 2, x^2, $4z + y$, and $(3zw)x + (5z^2 + 3w)$. It is also useful to define the *main primitive part* as the primitive part of the polynomial when viewed as univariate in its main variable. For example, $(y + 2)x^2 + (y^2 + 3x + 2)$ has main primitive part $x^2 + (y + 1)$.

The properties of a polynomial ring vary depending on the properties of its ground ring. For example, division in the polynomial ring $\mathbb{Z}[x]$ is not always possible; $3x+1$ does not divide x^2+2. This is obvious considering the schoolbook long division of polynomials, where one must divide coefficients of the dividend by the leading coefficient of the divisor (i.e. 3 does not divide 1). More formally, $\mathbb{Z}[x]$ is not a Euclidean domain, despite the integers obviously being one. However, if the ground ring of a univariate polynomial ring is a field, then division is always possible since the leading coefficient of the divisor polynomial has a multiplicative inverse.

The relations between a polynomial ring and its ground ring may be summarized as follows:

1. if R is a field, $R[x]$ is Euclidean domain;
2. if R is a UFD, $R[x]$ is a UFD (this is Gauss's lemma [86, Theorem 6.8]);
3. if R is a GCD domain, $R[x]$ is a GCD domain;
4. if R is an integral domain, $R[x]$ is an integral domain; and
5. if R is a ring, $R[x]$ is a ring.

These relations have important consequences when determining the properties of polynomial rings, particularly where they are constructed dynamically from different ground rings and with different numbers of variables. In terms of a computer algebra library, this leads to challenges in producing flexible and correct code which is simultaneously adaptable and extensible for end-users; we explore this challenge in Section 4.1. These relations also indicate many mathematical and algorithmic challenges. In particular, the computability of division and GCDs. If a polynomial ring is not a Euclidean domain, then division-with-remainder may not always be possible, nor can the Euclidean algorithm be used to effectively compute GCDs.

Let us consider the Euclidean algorithm more formally. For two elements a, b in a Euclidean domain $\mathbb{D}$, not both 0, the Euclidean algorithm computes their GCD via a *remainder sequence*. Successive remainders are computed and the GCD is the last non-zero remainder, as shown in Algorithm 2.1.

Algorithm 2.1 The Euclidean Algorithm

Input: a, b in a Euclidean domain $\mathbb{D}$ with Euclidean function f such that $f(a) \geq f(b)$.
Output: $g \in \mathbb{D}$, a greatest common divisor of a and b.

```
r_0 := a ; r_1 := b ; i := 1
while r_i ≠ 0 do
    r_{i+1} := r_{i-1} rem r_i
    i := i + 1
return r_{i-1}
```

To address the challenge of division in polynomial rings that are not Euclidean domains, we may use *pseudo-division*—a fraction-free division which ensures division can occur regardless of limitations in the ground ring. Pseudo-division is defined for polynomials over any ground ring [112] and is particularly used when polynomials are univariate or viewed recursively.

Definition 2.9 (Pseudo-Division)**.** *Let R be any ring. Let $a, b \in R[x]$ have degrees m and n, respectively, with $m \geq n \geq 0$. Let h be the leading coefficient of b. Then, $q, r \in R[x]$ are, respectively, the pseudo-quotient and pseudo-remainder, satisfying the equation*

$$h^e a \;=\; bq + r,$$

where the degree of r is less than n. The multiplication by h ensures that division can always be performed in the ground ring; we have that $e = \min(0, n - m + 1)$. We write $\operatorname{pquo}(a, b) = q$ *and* $\operatorname{prem}(a, b) = r$.

On the other hand, effectively computing GCDs in a non-Euclidean domain has been a problem for decades, particularly in the case of polynomials over the integers and multivariate polynomials. In the polynomial case, one can replace Euclidean division in the Euclidean algorithm with pseudo-division to obtain a *polynomial remainder sequence* (PRS); see, e.g., [86, Algorithm 6.61].

However, the expression swell experienced while computing a PRS is prohibitive. Much work has looked towards practical improvements; see, e.g., [36, 45, 57, 146, 179, 190], and further discussion in Section 6.1. The pioneering works of Brown [36] and Collins [57] employed the *Chinese Remainder Theorem*, an important concept worthy of its own section. Meanwhile, a generalization of the Euclidean algorithm and its remainder sequence is described later in Section 2.4 as subresultants.

2.1.3 Chinese Remainder Theorem and Direct Products

The Chinese Remainder Theorem (CRT) has been known for centuries, particularly in the case of the integers. The CRT states that if one knows the remainders (by Euclidean division) of a number by several different divisors, then one may determine the original number. One may view this as the "interpolation" of a number from its value modulo different moduli. For a polynomial $p \in R[x]$, computing the remainder of p by $x - r$ for some $r \in R$ is the same as evaluating p at r. Hence, the remainder of a number by some divisor r can be thought of as "evaluating the number at r". In its more modern form, the CRT is stated in terms of these moduli and congruence classes.

Theorem 2.10 (Chinese Remainder Theorem)**.**
Let $\mathbb{D}$ be a Euclidean domain with Euclidean function f, $m_1, \ldots, m_k \in \mathbb{D}$ be pairwise co-prime, and $a_1, \ldots, a_k \in \mathbb{D}$ such that $a_i = 0$ or $f(a_i) < f(m_i)$. Let $m = \prod_{i=1}^{k} m_i$. Then, the system

$$\begin{aligned} x &\equiv a_1 \bmod m_1 \\ x &\equiv a_2 \bmod m_2 \\ &\vdots \\ x &\equiv a_k \bmod m_k \end{aligned}$$

has a unique solution modulo m.

The proof of this theorem is constructive and yields the *Chinese Remainder Algorithm* (CRA), which relies only on the Euclidean algorithm; see [86, Theorem 5.2].

More generally, one can state the Chinese Remainder Theorem as a *ring isomorphism*. To be more concrete, let us consider the Euclidean domain of the integers, $\mathbb{Z}$, although the following holds for any Euclidean domain. Let $\mathbb{Z}/m := \mathbb{Z}/m\mathbb{Z}$ be the quotient ring of integers modulo m, where m is defined as in Theorem 2.10. Then, the CRT implies the following ring isomorphism:

$$\mathbb{Z}/m \;\cong\; \mathbb{Z}/m_1 \otimes \mathbb{Z}/m_2 \otimes \cdots \otimes \mathbb{Z}/m_k, \tag{2.1}$$

where the right-hand side is a *direct product of rings*.

A direct product of rings is itself a ring, where ring operations are defined component-wise. For two rings R and S, $R \otimes S$ is a ring with elements $\{(r, s) \mid r \in R,\ s \in S\}$ where $(0, 0)$ and $(1, 1)$ are the additive and multiplicative identities, respectively. In $R \otimes S$, addition and multiplication are defined respectively as $(r_1, s_1) + (r_2, s_2) = (r_1 + r_2, s_1 + s_2)$ and $(r_1, s_1)\,(r_2, s_2) = (r_1\, r_2, s_1\, s_2)$.

Since ring isomorphisms preserve ring operations, the CRT implies that any computation in $\mathbb{Z}/m$ can be mapped to an equivalent computation in $\mathbb{Z}/m_1 \otimes \cdots \otimes \mathbb{Z}/m_k$, and vice versa; see, e.g., [86, Ch. 25]

Example 2.11 (Integers modulo 6)**.** $\mathbb{Z}/6 \cong \mathbb{Z}/2 \otimes \mathbb{Z}/3$ defines a ring isomorphism between the integers modulo 6 and the integers modulo 2 and 3. $3 \cong (3 \bmod 2, 3 \bmod 3) = (1, 0)$. Arithmetic proceeds component-wise in $\mathbb{Z}/2 \otimes \mathbb{Z}/3$, and the bijective mapping is maintained even after arithmetic: $(1, 0) + (0, 1) = (1, 1) \cong 3 + 4 = 1$.

An important result of the CRT is that one can avoid direct computation in $\mathbb{Z}$ or $\mathbb{Z}/m$ and thus avoid *intermediate expression swell*. Since each $\mathbb{Z}/m_i$ is a finite ring, the size of elements of that ring is limited. One proceeds by a *modular computation*, working in $\mathbb{Z}/m_1 \otimes \cdots \otimes \mathbb{Z}/m_k$, and computing *modular images* of the solution modulo each of $m_1, \ldots, m_k$. Then, the solution in $\mathbb{Z}/m$ is reconstructed using the CRT and the ring isomorphism. The key observation of modular methods is that if an upper bound B of the absolute value of the solution is known, then we can choose moduli such that $m > 2B$ and therefore identify the solution in $\mathbb{Z}/m$ with the solution in $\mathbb{Z}$; see [86, Ch. 5] for more details.

Referring back to the problem of computing GCDs in a non-Euclidean domain, modular algorithms for GCDs are a key application of the CRT. Let B bound the largest integer appearing in the polynomial remainder sequence of two polynomials $a, b \in \mathbb{Z}[x]$. Then, consider prime numbers $p_1, p_2, \ldots, p_k$ such that their product is larger than $2B$. We can easily compute polynomial GCDs between a and b modulo each p_i, $1 \leq i \leq k$, as $g_1, \ldots, g_k$, since $\mathbb{Z}/p_i$ is in fact a field [86, Theorem 4.1]. Then, we can use the CRT to reconstruct the GCD of a and b from $g_1, \ldots, g_k$ and $p_1, \ldots, p_k$.

An important practical consequence of this technique in general is that each modular image can be computed independently. Indeed, arithmetic in a direct product of rings is only ever component-wise. This naturally leads to opportunities for parallelism. We will examine this further in Chapter 6.

The Chinese Remainder Theorem can easily be generalized beyond integers. Consider polynomials over the rational numbers $\mathbb{Q}[x]$. For some non-zero $p \in \mathbb{Q}[x]$, let $\mathbb{Q}[x]/p := \mathbb{Q}[x]/\langle p \rangle$ be the quotient ring of polynomials in $\mathbb{Q}[x]$ modulo p. Let $p_1, \ldots, p_k$ be pairwise co-prime polynomials in $\mathbb{Q}[x]$. Then, the CRT infers the ring isomorphism:

$$\mathbb{Q}[x]/(p_1 p_2 \cdots p_k) \;\cong\; \mathbb{Q}[x]/p_1 \otimes \mathbb{Q}[x]/p_2 \otimes \cdots \otimes \mathbb{Q}[x]/p_k. \tag{2.2}$$

If the polynomials $p_1, \ldots, p_k$ are more than relatively prime, and actually prime (i.e. irreducible) in $\mathbb{Q}[x]$, then each $\mathbb{Q}[x]/p_1, \ldots, \mathbb{Q}[x]/p_k$ is a field. Indeed, these are precisely algebraic number fields. Therefore, $\mathbb{Q}[x]/(p_1 \cdots p_k)$ is a *direct product of fields* (DPF), an algebraic structure with interesting computational properties. Note that a direct product of fields need not be a field itself, as we will see next in Section 2.2.

If the polynomials $p_1, \ldots, p_k$ are not necessarily prime but *square-free* (a polynomial p is square-free if for any non-constant polynomial q, q^2 does not divide p) then *each* of $\mathbb{Q}[x]/p_1, \ldots, \mathbb{Q}[x]/p_k$ is a direct product of fields. This follows by noticing that the factors in an irreducible factorization of a square-free polynomial are all relatively prime. Then, one recursively applies the construction of the previous paragraph.

Example 2.12 (Direct product of quotient rings)**.** Let $p_1 = x^2 - 4$, $p_2 = x^2 - 3 \in \mathbb{Q}[x]$ and let $p = p_1\, p_2 = x^4 - 7x^2 + 12$. Then, we have the direct product

$$\mathbb{Q}[x]/p \;\cong\; \mathbb{Q}[x]/(x^2-4) \otimes \mathbb{Q}[x]/(x^2-3).$$

$\mathbb{Q}[x]/(x^2-3)$ is a field, since x^2-3 is irreducible in $\mathbb{Q}[x]$. Meanwhile, $\mathbb{Q}[x]/(x^2-4)$ is itself a DPF since p_1 is square-free but not prime. This is easy to see by factoring p_1 as $(x-2)(x+2)$ and applying a second ring isomorphism to obtain:

$$\mathbb{Q}[x]/(x^2-4) \;\cong\; \mathbb{Q}[x]/(x-2) \otimes \mathbb{Q}[x]/(x+2).$$

This, in turn, tells us that $\mathbb{Q}[x]/p$ is a DPF, which can be obtained explicitly by combining the two previous isomorphisms, since direct products are associative:

$$\mathbb{Q}[x]/p \;\cong\; \mathbb{Q}[x]/(x-2) \otimes \mathbb{Q}[x]/(x+2) \otimes \mathbb{Q}[x]/(x^2-3).$$

One more generalization of the CRT states that if $\mathbb{K}$ is a DPF, and $f_1, \ldots, f_k \in \mathbb{K}[y]$ are relatively prime, then we have another ring isomorphism:

$$\mathbb{K}[y]/(f_1 f_2 \cdots f_k) \;\cong\; \mathbb{K}[y]/f_1 \otimes \mathbb{K}[y]/f_2 \otimes \cdots \otimes \mathbb{K}[y]/f_k. \tag{2.3}$$

This isomorphism tells us that we can construct direct products of polynomial rings where the ground ring is itself a direct product. This case arises naturally when considering (recursive) multivariate polynomial rings; for example, $\mathbb{K}$ may be $\mathbb{Q}[x]/(x^2-4)$. Therefore, the CRT and its generalizations allows for the *splitting* of rings into smaller and smaller components. In the same vein as modular methods, computation in each of these more simple rings is much more efficient and brings many practical benefits. For example, working directly in $\mathbb{Q}[x]/(x^2-4)$ is more expensive than the total work required to work in both $\mathbb{Q}[x]/(x-2)$ and $\mathbb{Q}[x]/(x+2)$. This notion, combined with the following *D5 Principle*, is crucial in the theory and practice of triangular decomposition.

2.2 The D5 Principle

From the above discussion and examples, we have seen that direct products of fields are a generalization of fields. Moreover, we can use square-free and irreducible polynomials to build extensions of DPFs as shown in Equations (2.2) and (2.3). Therefore, one can *almost* work in DPFs as if they were fields, except for the fact that some zero divisors exist in a DPF.

Example 2.13 (Zero divisors in a DPF)**.** Let $p_1 = x^2 - 2$, $p_2 = x^2 - 3 \in \mathbb{Q}[x]$. We have the direct product of fields and isomorphism:

$$\mathbb{Q}[x]/(p_1 p_2) \;\cong\; \mathbb{Q}[x]/(x^2-2) \otimes \mathbb{Q}[x]/(x^2-3).$$

Notice that we have $p_1 = x^2 - 2 \;\cong\; (x^2 - 2 \bmod p_1, x^2 - 2 \bmod p_2) = (0, 1)$, since $p_1 = p_2 + 1$. Then, as elements of $\mathbb{Q}[x]/p_1 \otimes \mathbb{Q}[x]/p_2$, we have $(0, 1) \times (1, 0) = (0, 0)$. That is, $(0, 1)$ and $(1, 0)$ are both zero divisors. By the ring isomorphism, p_1 is also a zero divisor in $\mathbb{Q}[x]/(p_1 p_2)$ and therefore it is not a field.

Zero divisors necessarily do not have multiplicative inverses. We have already seen that the lack of multiplicative inverses leads to computational challenges. For example, division in the polynomial ring $\mathbb{Z}[x]$. However, DPFs have the great benefit that every non-zero element is either a zero divisor or a unit. This follows easily from the structure of a DPF and the fact that every non-zero element in a field is a unit.

The celebrated *D5 principle* [67] states that one can work in a DPF as if it were a field, as long as computations *split* when a zero divisor is encountered. Due to the nature of DPFs, computations will split such that in one component or *branch* the encountered zero divisor becomes zero, meanwhile in the other branch the element is invertible.

Example 2.14 (Splitting a DPF)**.** Let $p = x^2 - 5x + 6 = (x - 2)(x - 3) \in \mathbb{Q}[x]$. Notice that p is square-free and thus $\mathbb{K} := \mathbb{Q}[x]/p$ is a DPF. Yet, let us assume we do not know the factorization of p nor the direct product isomorphism of $\mathbb{K}$.

Let $q = x^2 - 3x + 2 = (x - 1)(x - 2)$. It is easy to see that q is a zero divisor in $\mathbb{K}$ since $(x - 3)\, q = 0$ in $\mathbb{K}$. However, let us again assume we do not know the factorization of q. To discover whether or not q is a a zero-divisor, one can compute a GCD between p and q in $\mathbb{Q}[x]$ (which is easy since $\mathbb{Q}[x]$ is a Euclidean domain). Let this GCD be $g = (x - 2)$. Since p and q have a non-trivial GCD, q is a zero divisor in $\mathbb{K}$.

Now, since p is square-free, g and $p/g = (x - 3)$ are necessarily relatively prime in $\mathbb{Q}[x]$, and we can apply CRT to obtain:

$$\mathbb{Q}[x]/(x^2 - 5x + 6) \;\cong\; \mathbb{Q}[x]/(x-2) \otimes \mathbb{Q}[x]/(x-3).$$

Following this isomorphism, q is zero in $\mathbb{Q}[x]/(x - 2)$ and a unit in $\mathbb{Q}[x]/(x - 3)$.

In this example, we assumed that we did not know the factorization of the modulus p. Factorization is a very costly operation, particularly when working in extension fields. Indeed, the original motivation of the D5 principle was a practical way to perform elementary operations in algebraic number fields without relying on factorization. The key

idea behind the D5 principle is the following remark, attributed to Daniel Lazard [67].

Remark 2.15. *Let $p, q \in \mathbb{Q}[x]$ with p square-free and $g = GCD(p, q)$. Then, q is zero modulo g, and invertible modulo p/g. That is, q is zero in $\mathbb{Q}[x]/g$ and a unit in $\mathbb{Q}[x]/(p/g)$.*

Therefore, one does not compute the entire direct product of fields making up $\mathbb{Q}[x]/p$, nor the entire irreducible factorization of p. One only needs to compute a GCD and, if a zero divisor is found, an exact quotient to sufficiently split the ring so that elements are zero or invertible and computations may proceed. This has an immediate consequence in computing triangular decompositions, as we will see in Section 2.5 and Example 2.38, where a core operation is to compute polynomial GCDs over DPFs (which are not Euclidean domains). In particular, in the multivariate case implied by Equation (2.3), computing GCDs in *towers of algebraic extensions* is very costly without the use of the D5 principle [143].

2.2.1 Dynamic Evaluation

The D5 principle has many applications beyond field extensions and $\mathbb{Q}[x]$, as noted in its original presentation [67]. Among them, it was realized that the D5 principle is one instance of *dynamic evaluation* or "automatic case discussion" [70]. Dynamic evaluation is concerned with case discussions according to parameters, where there may be several branches or solutions depending on the particular values of the parameters. The dynamic part of dynamic evaluation is that the case discussion evolves during program execution.

Consider again Example 2.14, where the question was whether or not $q = (x-1)(x-2)$ was invertible in $\mathbb{Q}[x]/p$ for $p = (x-2)(x-3)$. This can be formulated as applying the statement "if $q = 0$ then 0 else $\frac{1}{q}$" modulo $(x-2)(x-3)$. The D5 principle can be applied to give the solution by dynamic evaluation as: 0, if $x = 2$ (i.e. modulo $x-2$), and $1/2$ otherwise (i.e. modulo $x-3$).[1] In the original implementation of the D5 principle [67], the internal representation of elements of the DPF (such as q) evolve during the computations to split and include such case discussions as needed.

More generally, the implementation of dynamic evaluation is often through *splitting trees* [34, 70]. A splitting tree is a tree structure where the root is the beginning of the computation, nodes represent points in the computation where information can change (e.g. the introduction of a case discussion), and edges represent a particular choice of constraints on a parameter. Nodes with more than one child represent a split. Several implementations of splitting trees [34, 70] are available in Axiom [104].

[1] Finding the multiplicative inverse of q in $\mathbb{Q}[x]/(x-3)$ is computed easily by the extended Euclidean algorithm between $x-3$ and q in $\mathbb{Q}[x]$.

The implementation in [34] realizes one crucial application of dynamic evaluation: avoiding redundant computations. Given a list of conditions on a parameter, one could repeatedly traverse the splitting tree down to a leaf node for each possible condition. However, this leads to redundant computation where many of the traversals would overlap. The authors in [34] propose two solutions. First, using *continuations* (see, e.g., [160, Ch. 6]) where a sort of "checkpoint" is made at each split and back-tracking avoids redundant computation. Second, using parallelism to fork the computation and traverse multiple branches of the splitting tree simultaneously.

Dynamic evaluation and the idea of avoiding redundant computation has more recently been revisited and expanded in [53]. Therein, the authors attempt to avoid redundant computation *between different* splitting trees during the computation of a cylindrical algebraic decomposition. When a split in a DPF is discovered, say through a GCD computation as in Example 2.14, this split is immediately shared between all trees. The advantage is twofold. First, redundant computation in computing the GCD and the splitting of the DPF is avoided. Second, as explained at the end of Section 2.1.3, splitting a DPF allows for computations in smaller components which is less expensive than working in the direct product itself.

Example 2.16. Let $\mathbb{K} = \mathbb{Q}[x]/(x^2-5x+6)$ be a DPF. Then, $\mathbb{K}[y]/(y-5)$ and $\mathbb{K}[y]/(y-7)$ are both DPFs. Working in the former, the following splitting may be discovered:

$$\mathbb{K}[y]/(y-5) \;\cong\; (\mathbb{Q}[x]/(x-2))\,[y]/(y-5) \otimes (\mathbb{Q}[x]/(x-3))\,[y]/(y-5).$$

This information may be shared with the latter to immediately obtain a similar splitting:

$$\mathbb{K}[y]/(y-7) \;\cong\; (\mathbb{Q}[x]/(x-2))\,[y]/(y-7) \otimes (\mathbb{Q}[x]/(x-3))\,[y]/(y-7).$$

The splitting $\mathbb{Q}[x]/(x^2-5x+6) \;\cong\; \mathbb{Q}[x]/(x-2) \otimes \mathbb{Q}[x]/(x-3)$ is computed only once but applied in two different contexts.

One difficulty noted in [53] is that its implementation in *Maple* lacks fine control over multithreaded execution and memory resources. Multithreaded execution and parallelism are obvious practical considerations for applications employing the D5 principle, as highlighted in [34]. Memory resources are highly important, particularly in a multithreaded implementation, where attempting to share data and information may lead to race conditions. Therefore, sharing information between splitting trees in a parallel implementation is a difficult problem. In this thesis, we will investigate both of these challenges in the context of triangular decompositions; see Chapters 6 and 8.

2.3 Polynomial Ideals and Varieties

The theory underlying triangular decomposition is heavily rooted in polynomial ideals and varieties. Here, we extend the general notion of ideals presented in Section 2.1.1 to ideals of polynomial rings. In particular, highlighting the correspondence between algebraic and geometric concepts. For a detailed presentation we suggest the work of Cox, Little, and O'Shea [61].

Let $\mathbb{K}$ be a field and let $f_1, \ldots, f_k \in \mathbb{K}[x_1, \ldots, x_n]$ be multivariate polynomials. We can construct a system of equations F from these polynomials:

$$F = \begin{cases} f_1 = 0 \\ f_2 = 0 \\ \quad \vdots \\ f_k = 0 \end{cases}.$$

Notice that we can derive other equations from this system using algebra. In particular, for any $g_1, \ldots, g_k \in \mathbb{K}[x_1, \ldots, x_n]$, we have the equation $f_1\, g_1 + f_2\, g_2 + \cdots + f_k\, g_k = 0$ as a consequence of F. This equation is precisely of the form which defines elements of the ideal $\langle f_1, \ldots, f_k \rangle \subseteq \mathbb{K}[x_1, \ldots, x_n]$. Indeed, the correspondence between ideals generated by a set of polynomials and a system of polynomial equations is quite strong. The ideal generated by $f_1, \ldots, f_k$ are precisely all the "polynomial consequences" of the equations $f_1 = f_2 = \cdots = f_k = 0$.

We know that the solution to a set of polynomial equations is all of the common roots of those polynomials. Geometrically, this may be a collection of points, curves, or surfaces. The set of all solutions to a system of polynomial equations is called an *algebraic variety*.[2]

Definition 2.17. *A set $V \subset \overline{\mathbb{K}}^n$ is an* (affine) algebraic variety *over $\mathbb{K}$ if there exists polynomials $f_1, \ldots, f_k \in \mathbb{K}[x_1, \ldots, x_n]$ whose zero set*

$$V(f_1, \ldots, f_k) = \left\{ (a_1, \ldots, a_n) \in \overline{\mathbb{K}}^n \mid f_i(a_1, \ldots, a_n) = 0 \text{ for all } 1 \leq i \leq k \right\}$$

equals V. $V(f_1, \ldots, f_k)$ is the algebraic variety defined by $f_1, \ldots, f_k$.

Conversely, for any subset S of the affine space $\overline{\mathbb{K}}^n$, we can define its *vanishing ideal*.

[2]Some authors require that algebraic varieties be irreducible, and non-irreducible varieties be called algebraic sets. Following [61], we do not make this distinction.

Definition 2.18. *For $S \subseteq \overline{\mathbb{K}}^n$ (not necessarily a variety), its* vanishing ideal *is:*

$$I(S) = \{f \in \mathbb{K}[x_1, \ldots, x_n] \mid f(a_1, \ldots, a_n) = 0 \text{ for all } (a_1, \ldots, a_n) \in S\}.$$

The correspondence between polynomial systems, ideals, and varieties grows stronger with *Hilbert's basis theorem* [61, Section 2.5, Theorem 4].

Theorem 2.19 (Hilbert's basis theorem)**.**
Every ideal $\mathcal{I} \subseteq \mathbb{K}[x_1, \ldots, x_n]$ is generated by a finite set of polynomials. That is, $\mathcal{I} = \langle f_1, \ldots, f_k \rangle$ for some $f_1, \ldots, f_k \in \mathcal{I}$.

This theorem has some immediate consequences. For $\mathcal{I} = \langle f_1, \ldots, f_k \rangle$, $V(\mathcal{I}) = V(f_1, \ldots, f_k)$. Moreover, if $\langle f_1, \ldots, f_k \rangle = \langle g_1, \ldots, g_\ell \rangle$ then $V(f_1, \ldots, f_k) = V(g_1, \ldots, g_\ell)$. This tells us that varieties are really determined by ideals.

Consider again a vanishing ideal of some subset $S \subseteq \overline{\mathbb{K}}^n$. The variety $V(I(S))$ is the smallest variety which contains S. In fact, the *Zariski closure* of S, denoted $\overline{S}$, is precisely $V(I(S))$. The *Zariski topology* is a topology over $\overline{\mathbb{K}}^n$ where its closed sets are the algebraic varieties over $\overline{\mathbb{K}}^n$. Thus, the closure operation "fills in the gaps" of any subset S to make it a variety; $\overline{S}$ is the intersection of all varieties V containing S. Moreover, we have that $I(\overline{S}) = I(S)$.

The correspondence between ideals and varieties is made more exact by the *Nullstellensatz* [61, Section 4.2, Theorem 6].

Theorem 2.20 (The Nullstellensatz)**.**
For an ideal $\mathcal{I} \subseteq \mathbb{K}[x_1, \ldots, x_n]$, $I(V(\mathcal{I})) = \sqrt{\mathcal{I}}$.

Proof. [61, Section 4.2, Theorem 6] □

The Nullstellensatz has several important geometric consequences. We list them below; see [61, Ch. 4] for further details and proofs. Let $\mathcal{I}_1, \mathcal{I}_2 \subseteq \mathbb{K}[x_1, \ldots, x_n]$ be ideals and $V_1, V_2 \subseteq \overline{\mathbb{K}}^n$ be algebraic varieties.

1. If $\mathcal{I}_1 \subseteq \mathcal{I}_2$, then $V(\mathcal{I}_1) \supseteq V(\mathcal{I}_2)$.
2. If $V_1 \subseteq V_2$, then $I(V_1) \supseteq I(V_2)$.
3. $V(I(V_1)) = V_1$
4. $V(\sqrt{\mathcal{I}}) = V(\mathcal{I})$
5. $I(V_1)$ is always radical.

Now that the Nullstellensatz has given us the *ideal-variety correspondence*, we may look at how ideal operations (see Section 2.1.1) correspond geometrically. Let $\mathcal{I} = \langle f_1, \ldots, f_k \rangle$ and $\mathcal{J} = \langle g_1, \ldots, g_\ell \rangle$ be ideals of $\mathbb{K}[x_1, \ldots, x_n]$. We have the following.

- $\mathcal{I} + \mathcal{J} = \langle f_1, \ldots, f_k, g_1, \ldots, g_\ell \rangle$
- $\mathcal{I} = \langle f_1, \ldots, f_k \rangle = \langle f_1 \rangle + \cdots + \langle f_k \rangle$
- $V(\mathcal{I} + \mathcal{J}) = V(\mathcal{I}) \cap V(\mathcal{J})$
- $\mathcal{I}\,\mathcal{J} = \langle f_i g_j \mid 1 \leq i \leq k, 1 \leq j \leq \ell \rangle$
- $\mathcal{I}\,\mathcal{J} \subseteq \mathcal{I} \cap \mathcal{J}$
- $V(\mathcal{I}\,\mathcal{J}) = V(\mathcal{I} \cap \mathcal{J}) = V(\mathcal{I}) \cup V(\mathcal{J})$

Two more ideal operations which have not yet been explored are *ideal quotient* and *ideal saturation*. The ideal quotient of $\mathcal{I}_1$ by $\mathcal{I}_2$ is the ideal:

$$\mathcal{I}_1 : \mathcal{I}_2 = \{f \in \mathbb{K}[x_1, \ldots, x_n] \mid \text{for all } g \in \mathcal{I}_2,\ fg \in \mathcal{I}_1\}$$

The saturation of $\mathcal{I}_1$ by $\mathcal{I}_2$ is the ideal:

$$\mathcal{I}_1 : \mathcal{I}_2^\infty = \{f \in \mathbb{K}[x_1, \ldots, x_n] \mid \text{for all } g \in \mathcal{I}_2,\ \exists\, e \in \mathbb{N},\ g^e f \in \mathcal{I}_1\}$$

For an ideal quotient or ideal saturation by an ideal with a single element in its generating set, say $\mathcal{I}_2 = \langle g \rangle$, then $\mathcal{I}_1 : \mathcal{I}_2$ may be written as simply $\mathcal{I}_1 : g$.

Example 2.21. Let $\mathcal{I} = \langle (y^2 + 2y + 1)(x^2 - 1) \rangle$ and $\mathcal{J} = \langle y + 1 \rangle$ be ideals of $\mathbb{K}[x, y]$. The ideal quotient and ideal saturation of $\mathcal{I}$ by $\mathcal{J}$ is then:

$$\begin{aligned} \mathcal{I} : \mathcal{J} &= \langle (y + 1)(x^2 - 1) \rangle \\ \mathcal{I} : \mathcal{J}^\infty &= \langle x^2 - 1 \rangle \end{aligned}$$

The operations of ideal quotient and ideal saturation are intimately tied. We have the following properties:

(*i*) $\mathcal{I}_1 \subseteq \mathcal{I}_1 : \mathcal{I}_2 \subseteq \mathcal{I}_1 : \mathcal{I}_2^\infty$,

(*ii*) for n large enough, $\mathcal{I}_1 : \mathcal{I}_2^n = \mathcal{I}_1 : \mathcal{I}_2^\infty$,

(*iii*) $\sqrt{\mathcal{I}_1 : \mathcal{I}_2^\infty} = \sqrt{\mathcal{I}_1} : \mathcal{I}_2$,

(*iv*) if $\mathcal{I}_1$ is radical, $\mathcal{I}_1 : \mathcal{I}_2^\infty = \mathcal{I}_1 : \mathcal{I}_2$.

Geometrically, ideal quotient and saturation also have related consequences, notably about set differences:

(i) $V(\mathcal{I}_1 : \mathcal{I}_2^\infty) = \overline{V(\mathcal{I}_1) \setminus V(\ I_2)}$,

(ii) $I(V_1) : I(V_2) = I(V_1 \setminus V_2)$,

(iii) $V(\mathcal{I}_1) = V(\mathcal{I}_1 + \mathcal{I}_2) \cup V(\mathcal{I}_1 : \mathcal{I}_2^\infty)$
$= (V(\mathcal{I}_1) \cap V(\mathcal{I}_2)) \cup \overline{V(\mathcal{I}_1) \setminus V(\mathcal{I}_2)}$.

The next theorem relates varieties with a potential meaning of "solving" a system of polynomial equations. An algebraic variety V is said to be *irreducible* if, for any other algebraic varieties V_1, V_2, $V = V_1 \cup V_2$ implies $V = V_1$ or $V = V_2$. A variety is irreducible if and only if $I(V)$ is a prime ideal [61, Section 4.5, Proposition 3]. Then, the following theorem states the existence of a *minimal irreducible decomposition* of any algebraic variety. Note that the statement of this theorem is simply the geometric dual of the minimal primary decomposition of ideals in Noetherian rings.

Theorem 2.22 (Lasker-Noether Theorem)**.**
Let $V \subseteq \overline{\mathbb{K}}^n$ be an algebraic variety. V has a minimal irreducible decomposition *into a unique set of irreducible varieties $\{V_1, \ldots, V_k\}$ such that*

$$V = V_1 \cup V_2 \cup \cdots \cup V_k$$

and $V_i \not\subseteq V_j$ for all $i \neq j$.

Proof. [61, Section 4.6, Theorem 4] □

This theorem gives the language to discuss the dimension of a variety or of an ideal. Intuitively, the dimension of a finite set of points is 0, the dimension of a curve is 1, the dimension of a surface is 2, etc. Specifically, the dimension of a variety V is the maximal length d of a chain of distinct irreducible subvarieties of V: $V_0 \subsetneq V_1 \subsetneq \cdots \subsetneq V_d = V$. Algebraically, for V with $I(V) = \mathcal{I} \subseteq R$, the dimension of V equals the dimension of $\mathcal{I}$, and the dimension of $\mathcal{I}$ equals the Krull dimension of the quotient ring $R/\mathcal{I}$. The *Krull dimension* of a ring is the maximal number d of strict inclusions in a chain of prime ideals of that ring. For an ideal $\mathcal{I}$ of a polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$, the *height* of $\mathcal{I}$ is defined as n minus the dimension of $\mathcal{I}$.

An ideal $\mathcal{I}$ is called *unmixed* if the dimensions of all of its associated primes are equal. Consequently, the variety $V(\mathcal{I})$ of an unmixed ideal $\mathcal{I}$ is *equidimensional*—it can be decomposed into irreducible varieties which all have the same dimension. Unmixed ideals

have useful properties. One useful property is that the associated primes an unmixed ideal are the same as the associated primes of its radical.

Generally, computing the dimension of an arbitrary variety or ideal is difficult. However, we will see in Section 2.5 that computing the dimension of a regular chain, and its corresponding geometric object, is very easy. The Lasker-Noether Theorem also suggests a natural output for the solution of a system of polynomial equations F: a description of each of the irreducible components of the variety $V(F)$. However, this may not necessarily be useful in practice. Instead, we suggest a method based on triangular decomposition to decompose $V(F)$ into components of unmixed dimension. This method is described later in Section 2.5. But first, a crucial operation within triangular decomposition is the computation of resultants and subresultants.

2.4 Subresultant Theory and Regular GCDs

We have already seen in Section 2.1.2 that the Euclidean Algorithm produces a sequence of remainders towards computing a GCD. The theory of subresultants is strongly related. For polynomials f, g in some polynomial ring $R[y]$, with $\deg(f) \geq \deg(g)$, their subresultant of degree k (for $k < deg(g)$) is a scalar multiple of the polynomial of degree k in the Euclidean remainder sequence.[3]

Much like the Euclidean algorithm, where the last remainder is 0, the "last" subresultant, known as the *resultant*, has a special property. The resultant of two polynomials is zero if and only if they have a common root, that is, if they have a non-trivial GCD. Loosely speaking, the GCD can be computed as the last non-zero subresultant.

We take this section to give a brief review of subresultant theory to see how we can compute "regular GCDs", a crucial step in triangular decomposition. In this description of subresultants, we follow the presentation of [69], [106], and [86].

Determinantal polynomial

Let $\mathbb{A}$ be a commutative ring with identity and let $m \leq n$ be positive integers. Let M be a $m \times n$ matrix with coefficients in $\mathbb{A}$. Let M_i be the square submatrix of M consisting of the first $m-1$ columns of M and the i-th column of M, for $m \leq i \leq n$; let $\det(M_i)$ be the determinant of M_i. The *determinantal polynomial* of M denoted by $\mathrm{dpol}(M)$ is a polynomial in $\mathbb{A}[y]$, given by

$$\mathrm{dpol}(M) = \det(M_m)y^{n-m} + \det(M_{m+1})y^{n-m-1} + \cdots + \det(M_n).$$

[3] If a such a polynomial with degree k exists in the sequence, otherwise the subresultant is 0.

Note that if $\mathrm{dpol}(M)$ is not zero then its degree is at most $n - m$. Let $f_1, \ldots, f_m$ be polynomials of $\mathbb{A}[y]$ of degree less than n. We denote by $\mathrm{mat}(f_1, \ldots, f_m)$ the $m \times n$ matrix whose i-th row contains the coefficients of f_i, sorted in order of decreasing degree, and such that each $f_1, \ldots, f_m$ is treated as a polynomial of degree $n - 1$. We denote by $\mathrm{dpol}(f_1, \ldots, f_m)$ the determinantal polynomial of $\mathrm{mat}(f_1, \ldots, f_m)$.

Example 2.23. Let $a = a_3y^3 + a_2y^2 + a_1y + a_0$ and $b = b_2y^2 + b_1y + b_0$ be polynomials in $\mathbb{A}[y]$. We have:

$$\mathrm{mat}(a, b) = \begin{bmatrix} a_3 & a_2 & a_1 & a_0 \\ 0 & b_2 & b_1 & b_0 \end{bmatrix},$$

with,

$$M_2 = \begin{bmatrix} a_3 & a_2 \\ 0 & b_2 \end{bmatrix}, M_3 = \begin{bmatrix} a_3 & a_1 \\ 0 & b_1 \end{bmatrix}, \text{ and } M_4 = \begin{bmatrix} a_3 & a_0 \\ 0 & b_0 \end{bmatrix}$$

and consequently $\mathrm{dpol}(a, b) = a_3b_2y^2 + a_3b_1y + a_3b_0$.

Subresultants

Let $a, b \in \mathbb{A}[y]$ be non-constant polynomials of respective degrees $m = \deg(a), n = \deg(b)$ with $m \geq n$. Let the leading coefficient of a w.r.t. y be defined as $\mathrm{lc}(a)$. Let k be an integer with $0 \leq k < n$. Then the k-th *subresultant* of a and b (also known as the *subresultant of index* k of a and b), denoted by $S_k(a, b)$, is

$$S_k(a, b) = \mathrm{dpol}(y^{n-k-1}a, y^{n-k-2}a, \ldots, a, y^{m-k-1}b, \ldots, b).$$

This is a polynomial which belongs to the ideal generated by a and b in $\mathbb{A}[y]$. In particular, $S_0(a, b)$ is the resultant of a and b denoted by $\mathrm{res}(a, b)$. In the case of $k = 0$, $\mathrm{mat}(y^{n-1}a, y^{n-2}a, \ldots, a, y^{m-1}b, \ldots, b)$ is in fact the *Sylvester matrix* of a and b with respect to y, and its determinant is the resultant.

Observe that if $S_k(a, b)$ is not zero then its degree is at most k. When $S_k(a, b)$ has degree k, it is said *non-defective* or *regular*[4]; when $S_k(a, b) \neq 0$ and $\deg(S_k(a, b)) < k$, $S_k(a, b)$ is said *defective*. The coefficient of $S_k(a, b)$ in y^k is denoted by s_k and is called the *nominal leading coefficient* (sometimes called the *principal leading coefficient*). If $S_k(a, b)$ is defective then $s_k = 0$.

For convenience, we extend the definition of subresultants to include b itself, letting $S_n = b$. Then, the collection of all subresultants forms a *subresultant chain*:

$$\mathrm{subres}(a, b) = \{S_n(a, b), S_{n-1}(a, b), S_{n-2}(a, b), \ldots, S_0(a, b)\},$$

[4]We refer to it as non-defective to avoid conflicting with regular elements of a ring.

Example 2.24. Let $a = y^3 - y^2$ and $b = y^2 - 3y$ be two polynomials in $\mathbb{Z}[y]$ with $\gcd(a, b) = y$. Then:

$$S_1(a,b) = \mathrm{dpol}(ya, a, b) = \mathrm{dpol}(\begin{pmatrix} 1 & -3 & 0 & \\ & 1 & -3 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix}) = 6y.$$

$$S_0(a,b) = \mathrm{dpol}(y^2a, ya, a, yb, b) = \mathrm{dpol}(\begin{pmatrix} 1 & -3 & 0 & & \\ & 1 & -3 & 0 & \\ & & 1 & -3 & 0 \\ 1 & -1 & 0 & 0 & \\ & 1 & -1 & 0 & 0 \end{pmatrix}) = 0,$$

Computing subresultants and the specialization property

The specialization property of subresultants is a crucial property for triangular decomposition based on regular chains. Informally, this property states that, under some specialization, the subresultant of the specializations is the specialization of the subresultant. Formally, it is stated as follows.

Let $\mathbb{A}$ and $\mathbb{B}$ be commutative rings with identity. Let Φ be a homomorphism from $\mathbb{A}$ to $\mathbb{B}$ which naturally induces a homomorphism from $\mathbb{A}[y]$ to $\mathbb{B}[y]$. Assume $\Phi(\mathrm{lc}(a)) \neq 0$ and $\Phi(\mathrm{lc}(b)) \neq 0$. Then, for $0 \leq k \leq n$, we have:

$$\Phi(S_k(a,b)) = S_k(\Phi(a), \Phi(b)),$$

see [49, Theorem 3.1]. This property has many practical consequences. In particular, it implies that subresultants may be computed by modular methods or by evaluation-interpolation schemes.

- When $\mathbb{A}$ is a prime field, say $\mathbb{Z}_p$ for an odd prime p, a Euclidean-like procedure can be derived to compute subres(a, b) for $a, b \in \mathbb{A}[y]$ from simple manipulations of the Sylvester matrix; see [86, Ch. 6].

- When $\mathbb{A}$ is the ring of integers $\mathbb{Z}$, we can proceed by a modular method. By selecting sufficiently many prime numbers $p_1, \ldots, p_e$, where $lc(a) \not\equiv 0 \bmod p_i$ and $lc(b) \not\equiv 0 \bmod p_i$, one computes subres$(a \bmod p_i, b \bmod p_i)$ in $Z_{p_i}[y]$ and then uses the CRT to recover subres(a, b) over $\mathbb{Z}[y]$.

- When $\mathbb{A}$ is a polynomial ring over $\mathbb{Z}$ or over a prime field, one can reduce to one

of the two previous cases via evaluation-interpolation. For example, if $\mathbb{A} = \mathbb{Z}[x]$, for some values $x_1, \ldots, x_e \in \mathbb{Z}$, we can compute subres$(a\,|_{x=x_i}, b\,|_{x=x_i})$ for each $1 \leq i \leq e$ and then recover subres(a, b) via interpolation.

The latter two methods are described later in Section 6.3. For multivariate polynomial rings one can repeatedly apply evaluation-interpolation. For multivariate polynomial rings over $\mathbb{Z}$, one can apply evaluation-interpolation and the CRT.

Yet, it is also feasible to directly compute a subresultant chain in a polynomial ring, say $\mathbb{Z}[x_1, \ldots, x_n][y]$. A Euclidean-like method can be derived from the following *divisibility relations of subresultants.* Let $S_k := S_k(a, b)$ for $0 \leq k \leq n$, where a, b continue to have respective degrees $m \geq n$. We write $f \sim g$ whenever f and g are associates in $\mathrm{Frac}(\mathbb{A})[y]$. Then, the following relations hold:

(*i*) if $S_{n-1} = \mathrm{prem}(a, -b) \neq 0$, and $\deg(S_{n-1}) = e$, then

$$\mathrm{prem}(b, -S_{n-1}) = \mathrm{lc}(b)^{(m-n)(n-e)+1} S_{e-1};$$

(*ii*) if $S_{k-1} \neq 0$ and $\deg(S_{k-1}) = e < k-1$ (i.e. S_{k-1} is defective), then

(*a*) $\deg(S_k) = k$ and S_k is non-defective,

(*b*) $S_{k-1} \sim S_e$, $\mathrm{lc}(S_{k-1})^{k-e-1} S_{k-1} = s_k^{k-e-1} S_e$, and S_e is non-defective,

(*c*) $S_{k-2} = S_{k-3} = \cdots = S_{e+1} = 0$;

(*iii*) if $S_k \neq 0$ and $S_{k-1} \neq 0$, with respective degrees k and e, then

$$\mathrm{prem}(S_k, -S_{k-1}) = \mathrm{lc}(S_k)^{k-e+1} S_{e-1}.$$

These divisibility relations bring about Algorithm 2.2, a well-known method [68, 69]. Note that this algorithm can easily be modified for the case where $\mathbb{A}$ is a field to use Euclidean remainders rather than pseudo-remainders. Moreover, note that Algorithm 2.2 can easily be modified (as a post-processing step) to include the zero subresultants as well. The computability of subresultants, in view of the specialization property and these divisibility relations, is summarized in Figure 2.1.

Algorithm 2.2 Subresultant (a, b, y)

Input: $a, b \in \mathbb{A}[y]$ with $m = \deg(a) \geq n = \deg(b) > 0$ and $\mathbb{A}$ an integral domain.
Output: the non-zero subresultants of $\text{subres}(a, b) = (S_n, S_{n-1}, S_{n-2}, \ldots, S_0)$

$s := \text{lc}(b)^{m-n}$
$A := b;\ B := \text{prem}(a, -b)$
$S := (b)$
while true **do**
  if $B = 0$ **then**
    return S
  $d := \deg(A);\ e := \deg(B)$
  $\delta := d - e$
  $S := S, B$
  if $\delta > 1$ **then**
    $C := \frac{\text{lc}(B)^{\delta-1} B}{s^{\delta-1}}$
    $S := S, C$
  else
    $C := B$
  if $e = 0$ **then**
    return S
  $B := \frac{\text{prem}(A, -B)}{s^{\delta} \text{lc}(A)}$
  $A := C;\ s := \text{lc}(A)$
end while

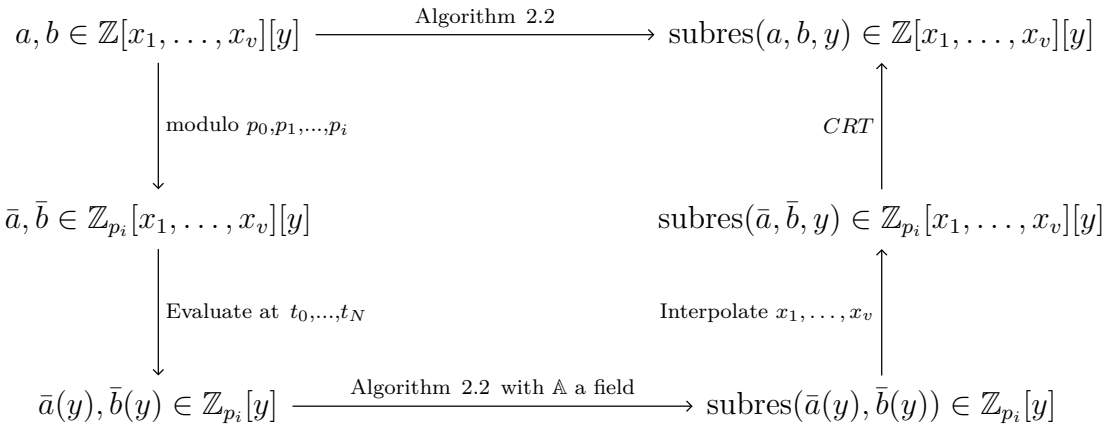

Figure 2.1: Computing the subresultant chain of $a, b \in \mathbb{Z}[x_1, \ldots, x_v][y]$ using modular arithmetic, evaluation-interpolation, and CRT, where $(t_0, \ldots, t_N) \subset {\mathbb{Z}_{p_i}}^v$ is the list of evaluation points and $(p_0, \ldots, p_i)$ is the list of distinct primes.

Regular GCDs

Algorithm 2.2 and the previous divisibility relations show that subresultants form a Euclidean-like sequence. The following theorem now relates subresultants explicitly to GCDs.

Theorem 2.25. *Let $\mathbb{A}$ be a commutative ring with identity, $\mathbb{B}$ be a UFD, and Φ be a homomorphism from $\mathbb{A}$ to $\mathbb{B}$. Let $a, b \in \mathbb{A}[y]$ such that $\deg(a) = m \geq \deg(b) = n$, $\Phi(\mathrm{lc}(a)) \neq 0$, and $\Phi(\mathrm{lc}(b)) \neq 0$. The following relations hold between subresultants and the GCD of $\Phi(a)$ and $\Phi(b)$.*

(i) Let $0 < k < n$ be an integer such that $\Phi(s_k) \neq 0$ and for all $0 \leq i < k$ $\Phi(s_i) = 0$. Then, the GCD of $\Phi(a)$ and $\Phi(b)$ is $\Phi(S_k)$

(ii) If $\Phi(s_i) = 0$ for all $0 \leq i < n$, then the GCD of $\Phi(a)$ and $\Phi(b)$ is $\Phi(b)$.

Proof. [49, Theorem 3.2] □

For rings which are not necessarily UFDs, it is useful to extend the notion of a GCD to a regular GCD. These will be particularly useful over direct products of fields, where regularity of elements can easily be computed and enforced via splitting the DPF.

Definition 2.26 (Regular GCD)**.** *Let $\mathbb{A}$ be a commutative ring. Let $a, b \in \mathbb{A}[y]$ be non-zero. We say $g \in \mathbb{A}[y]$ is a* regular gcd *of a and b if:*

(i) the leading coefficient of g in y is a regular element of $\mathbb{A}$;

(ii) $g \in \langle a, b \rangle \subseteq \mathbb{A}[y]$; and

(iii) degree of $g > 0 \implies \mathrm{prem}(a, g) = \mathrm{prem}(b, g) = 0$

There is an obvious connection between regular GCDs and subresultants. Indeed, the subresultants of a and b belong to the ideal $\langle a, b \rangle$ and, Theorem 2.25, suggests a way to compute regular GCDs using subresultant chains. In particular, given a subresultant chain, say computed over $\mathbb{Q}[x_1, \ldots, x_n]$, one proceeds "bottom-up", for $k = 0, 1, \ldots$, searching for the smallest k such that $\Phi(s_k)$ is *regular*. In practice, Φ represents working modulo a regular chain, an idea which is formalized in the next section.

2.5 Solving Polynomial Systems

We have already been made familiar with the idea of polynomial system solving in Section 1.1. Therein, Gröbner bases and triangular decompositions were introduced informally. We will not formally define Gröbner bases (rather, see [61, Ch. 2]). But, we will discuss the kind of problems that they solve, to show the contrast with what triangular decompositions are able to solve. Then, we will formally define triangular decomposition and regular chains, highlighting their useful geometric properties.

Mathematically speaking, solving systems of polynomial equations is a completely solved problem via the Lasker-Noether Theorem, Theorem 2.22. For an input system of polynomials F, one can find the complete decomposition of its variety $V(F)$ into irreducible varieties (or the decomposition of the ideal $\langle F \rangle$ into primary ideals). But, this is not particularly meaningful in practice, and is very computationally expensive to compute. For some applications, it may be sufficient to simply find a single sample solution. In other applications, it may be sufficient to find the *generic zeros* (in the sense of van der Waerden [171]) of the irreducible components of $V(F)$. This is precisely the work of Kalkbrener in introducing regular chains [107]. Further still, perhaps all solutions of the system are interesting, but they do not necessarily need to be separated completely into irreducible components, rather only into components of unmixed dimension. This is the case for the work of Wu [181] (although only proven to do so afterwards by Gao and Chou [83]) and Lazard [121]. The more recent work of Moreno Maza and his collaborators also follow this idea [17, 47, 125, 142]. Importantly, this latter work also retains important algebraic properties [16] which makes computation much more feasible.

Consider some algebraic problems which are related to polynomial system solving.

Problem 2.27 (Ideal Membership Problem)**.** Given $\mathcal{I} \subseteq R$ and $f \in R$, is $f \in \mathcal{I}$?

If we let $R = \mathbb{K}[x_1, \ldots, x_n]$, then this problem is related to whether or not a particular polynomial shares a common solution with a set of polynomials. That is, it determines if $V(f) \supseteq V(\mathcal{I})$. In view of the Nullstellensatz (Theorem 2.20), a similar question arises naturally regarding the radical of an ideal.

Problem 2.28 (Radical Membership Problem)**.** Given $\mathcal{I} \subseteq R$ and $f \in R$, is $f \in \sqrt{\mathcal{I}}$?

While this problem is algebraically different from the previous, it is geometrically the same. Indeed, $f \in \sqrt{I}$ also implies $V(f) \supseteq V(\sqrt{\mathcal{I}}) = V(\mathcal{I})$.

Gröbner bases provide a solution for solving both of these problems in the case of polynomial rings over a field. A Gröbner basis is a particular generating set for an ideal

that has useful algorithmic properties. Indeed, Hilbert's basis theorem (Theorem 2.19) implies that every ideal admits a Gröbner basis [86, Corollary 21.26].

Let $G \subseteq \mathbb{K}[x_1, \ldots, x_n]$ be a Gröbner basis for an ideal $\mathcal{I}$. One fundamental algebraic and algorithmic property of Gröbner bases is that, for any polynomial $f \in \mathbb{K}[x_1, \ldots, x_n]$, *multivariate polynomial division* (see, e.g., [61, Ch. 2]) of f by all $g \in G$ results in a unique remainder r. We say that f *reduces* by G to r. There are two immediate consequences. First, if f reduces to r by G, then $f - r \in \mathcal{I}$. Second, and obviously, if f reduces to 0 by G, then $f \in \mathcal{I}$. Hence, this solves the ideal membership problem.

Given any ideal of a polynomial ring over a field (and some term ordering), *Buchberger's algorithm* [39] computes a Gröbner basis for that ideal (w.r.t. to that ordering). Therefore, most problems proceed by first computing a Gröbner basis of an input collection of polynomials and then using that Gröbner basis to answer certain questions. For example, one may compute a so-called *reduced Gröbner basis* to solve the radical membership problem [61, Section 4.2].

For the problem of solving a system of polynomial equations, we have already seen in Section 1.1 that a Gröbner basis may be used to solve a system with a finite number of solutions. That is, a zero-dimensional system. Turning to the positive-dimensional case, Gröbner bases are less effective. A Gröbner basis can easily be used to answer the ideal membership problem in the positive-dimensional case, but that basis may be very complicated and, moreover, will not describe well the curves, surfaces, etc. contained in the corresponding variety. Rather, we saw that the use of triangular decomposition was much more effective. In the remainder of this section, we will formalize solving systems of equations by means of triangular decomposition and regular chains.

2.5.1 Triangular Decomposition

Triangular decomposition and triangular sets have a long history. Triangular decompositions were introduced by Ritt in [153] for solving systems of partial differential equations using special kinds of triangular sets known as characteristic sets. Following Ritt's work, Wu [185] proposed a method for solving systems of algebraic equations. However, his method may fail to detect when a system has no solution. This problem was solved by Kalkbrener [107] using regular chains, a strengthened notion of characteristic sets with remarkable algorithmic properties.

Denote the ring multivariate polynomials over a field $\mathbb{K}$ as $\mathbb{K}[x_1, \ldots, x_n] := \mathbb{K}[\underline{X}]$. Let us also fix a variable ordering $x_1 < x_2 < \cdots < x_n$ for the remainder of this section. Under this ordering we may view a non-constant polynomial $p \in \mathbb{K}[\underline{X}]$ recursively with respect

to its *main variable*: the greatest variable appearing in p. Then, its leading coefficient, degree, and reductum with respect to that main variable are called, respectively, the *initial*, *main degree*, and *tail*. We denote the main variable, main degree, initial, and tail, of a polynomial p, respectively, as $\mathrm{mvar}(p)$, $\mathrm{mdeg}(p)$, $\mathrm{init}(p)$, and $\mathrm{tail}(p)$. For a polynomial with main variable x_i, this is akin to working in the subring $\mathbb{K}[x_1, \ldots, x_{i-1}][x_i]$.

Definition 2.29 (Triangular set)**.** *A set $T \subset \mathbb{K}[\underline{X}] \setminus \mathbb{K}$ is a* triangular set *if the polynomials in T have pairwise distinct main variables.*

Denote by $\mathrm{mvar}(T)$ the set of main variables of the polynomials in T. A variable $v \in \underline{X}$ is called *algebraic* with respect to T if $v \in \mathrm{mvar}(T)$ and is called *free* otherwise. For $v \in \mathrm{mvar}(T)$, denote by T_v the polynomial $p \in T$ with $\mathrm{mvar}(p) = v$ and denote by T_v^- (resp. T_v^+) the set of polynomials $p \in T$ with $\mathrm{mvar}(p) < v$ (resp. $\mathrm{mvar}(p) > v$). For a polynomial $p \in \mathbb{K}[\underline{X}]$, denote by $\mathrm{pquo}(p, T)$ and $\mathrm{prem}(p, T)$ the pseudo-quotient and pseudo-remainder, respectively, of p by T. $\mathrm{prem}(p, T) = p$ if $T = \varnothing$. Otherwise, let v be the largest variable appearing in $\mathrm{mvar}(T)$, then $\mathrm{prem}(p, T) = \mathrm{prem}(\mathrm{prem}(p, T_v), T_v^-)$. $\mathrm{pquo}(p, T)$ is defined similarly.

Let h_T be the product of the initials of the polynomials in the triangular set T. The *saturated ideal* $\mathrm{sat}(T)$ of T is $\langle 0 \rangle$ if $T = \varnothing$ and $\langle T \rangle : h_T^\infty$ otherwise. The *quasi-component* $W(T)$ of a triangular set T is the set $V(T) \setminus V(h_T)$. Naturally, a quasi-component may not be a variety. We may get a variety by taking the closure of the quasi-component: $\overline{W(T)}$. As a property of triangular sets, we have that $\overline{W(T)} = V(\mathrm{sat}(T))$.

Example 2.30. Let $T = \{(y + 1)x,\ z - 1\}$ be a triangular set of $\mathbb{K}[x > y > z]$. $W(T) = V(T) \setminus V(z+1)$ is the curve parameterized by $p(t) = (0, t, 1)$ for $t \in \overline{\mathbb{K}}$ *excluding* the point $(0, -1, 1)$. The closure of $W(T)$ fills in this missing point. In particular, $\mathrm{sat}(T) = \langle x, z - 1 \rangle$ and $V(\mathrm{sat}(T))$ is the entire curve parameterized by $p(t)$.

Definition 2.31 (Regular chain)**.** *A triangular set T is a* regular chain *if either $T = \varnothing$ or, letting v be the greatest variable in* $\mathrm{mvar}(T)$*, the set T_v^- is a regular chain and the initial of T_v is regular modulo* $\mathrm{sat}(T_v^-)$ *(i.e.* $\mathrm{init}(T_v)$ *is a regular element of* $\mathbb{K}[\underline{X}]/\mathrm{sat}(T_v^-)$*).*

The useful algorithmic properties of a regular chain T are numerous (see [16, 30]). Some notable properties are summarized in the following proposition and proved in [49].

Proposition 2.32. *The following properties hold for a regular chain T:*

(i) for any regular chain T, $W(T) \neq \varnothing$;

(ii) the dimension of T is $d = \dim(T) = \dim(\mathrm{sat}(T)) = n - |T|$;

(iii) letting $u_1, \ldots, u_d$ be the free variables of T, then $\mathrm{sat}(T) \cap \mathbb{K}[u_1, \ldots, u_d] = \varnothing$;

(iv) $\mathrm{sat}(T)$ is unmixed of dimension d; and

(v) $p \in \mathrm{sat}(T) \iff \mathrm{prem}(p, T) = 0$

Notice that the last property is very similar to the ideal membership problem except that it determines inclusion in $sat(T)$ rather than $\langle T \rangle$. Let us now see some examples of a triangular set and a regular chain.

Example 2.33. Let $\mathbb{K}[\underline{X}]$ be $\mathbb{Q}[z < y < x]$.

$$T_1 = \begin{cases} yx + 1 \\ y \\ z - 1 \end{cases} \qquad T_2 = \begin{cases} (y+1)x^2 - x \\ y^2 - 1 \\ z - 1 \end{cases} \qquad T_3 = \begin{cases} yx^2 - x \\ y^2 - 1 \\ z - 1 \end{cases}$$

T_1, T_2, and T_3 are all zero-dimensional triangular sets; they each contain only one polynomial whose main variable is x, y, or z. However, only T_3 is a regular chain. T_1 is not a regular chain since $W(T_1) = \varnothing$. The second polynomial says $y = 0$ and substituting that into the first polynomial says $(0)x + 1 = 0 \implies 1 = 0$; the set is inconsistent. T_2 is not a regular chain since $(y + 1)$, the initial of the first polynomial, is a zero-divisor modulo the second polynomial $y^2 - 1 = (y+1)(y-1)$. Notice that the quasi-component $W(T_3)$ is in fact a variety encoding the points $(0, 1, 1), (0, -1, 1), (1, 1, 1), (-1, -1, 1)$; it is not an irreducible variety, but it is equidimensional.

Example 2.34. Let $\mathbb{K}[\underline{X}]$ be $\mathbb{Q}[b < a < y < x]$.

$$T_4 = \begin{cases} (2y + ba)x - by + a^2 \\ 2y^2 - by - a^2 \\ a + b \end{cases}$$

T_4 is a regular chain of dimension $1 = 4 - |T_4|$. Indeed, $\overline{W(T_4)}$ has 4 irreducible components of dimension 1: three lines and one curve. This can be seen through back-

substitution and factorization to yield the regular chains:

$$T_{4,1} = \left\{ \begin{array}{r} (2a-2)x - 3a \\ 2y - a \\ a + b \end{array} \right. , \quad T_{4,2} = \left\{ \begin{array}{r} x \\ y + a \\ a + b \end{array} \right. , \quad T_{4,3} = \left\{ \begin{array}{r} y - 2 \\ b - 2 \\ a + 2 \end{array} \right. , \quad T_{4,4} = \left\{ \begin{array}{r} y \\ b \\ a \end{array} \right. .$$

Finally, we can define the triangular decomposition (into regular chains) of a system of polynomials.

Definition 2.35 (Triangular Decomposition)**.** *Let $F \subset \mathbb{K}[\underline{X}]$ be a finite subset. A set of regular chains $T_1, \ldots, T_e$ is a* triangular decomposition *of F if:*

(Kalkbrener Decomposition) $V(F) = \overline{W(T_1)} \cup \overline{W(T_2)} \cup \cdots \cup \overline{W(T_e)}$*, or*

(Lazard-Wu Decomposition) $V(F) = W(T_1) \cup W(T_2) \cup \cdots \cup W(T_e)$.

In a triangular decomposition in the sense of Kalkbrener, the output regular chains only represent the generic zeros of the irreducible components of $V(F)$. This is much easier to compute than a decomposition in the sense of Lazard and Wu. In the latter, the quasi-components of the output regular chains exactly decompose the variety $V(F)$. Notice that a Lazard-Wu decomposition is necessarily a Kalkbrener decomposition, but the converse is not true. In either case, *redundant components* are possible. That is, there may exist T_i and T_j such that $W(T_i) \subseteq W(T_j)$. Efficiently discovering and removing those redundancies is detailed in Section 6.3.4.

An important algorithmic aspect of a Lazard-Wu decomposition is that it can be computed incrementally. Incremental triangular decomposition was proposed by Lazard [121] and extended by Moreno Maza [142] and by Chen and Moreno Maza [47, 52]. Moreover, Moreno Maza in [142] shows how to compute a Kalkbrener decomposition from an incremental algorithm which computes a Lazard-Wu decomposition.

Incremental triangular decomposition relies on a fundamental operation for computing the intersection between a hypersurface (a variety defined by a single polynomial) and a quasi-component. See Algorithm 6.18 in Section 6.1. This core routine has well-defined geometric inputs and outputs. We call this routine Intersect(p, T). Its inputs are a polynomial p and a regular chain T and returns regular chains $T_1, \ldots, T_e$ such that:

$$V(p) \cap W(T) \ \subseteq \ W(T_1) \cup \cdots \cup W(T_e) \ \subseteq \ V(p) \cap \overline{W(T)}$$

We denote by $Z(p, T) := V(p) \cap W(T)$. This relation says that $W(T_1) \cup \cdots \cup W(T_e)$ is a "sharp" approximation of $V(p) \cap W(T)$. In particular, the relation implies the following properties, which we call a *regular split*.

Definition 2.36 (Regular split)**.** *Let $p \in \mathbb{K}[\underline{X}]$ and $T \subset \mathbb{K}[\underline{X}]$ be a regular chain. If $T_1, \ldots, T_e$ are regular chains of $\mathbb{K}[\underline{X}]$, we call $T_1, \ldots, T_e$ a* regular split *of (p, T), and we write $(p, T) \to T_1, \ldots, T_e$, if, for all $1 \leq i \leq e$, we have:*

(i) $\sqrt{\text{sat}(T)} \subseteq \sqrt{\text{sat}(T_i)}$*;*

(ii) $W(T_i) \subseteq V(p)$ *(i.e.* $p \in \sqrt{\text{sat}(T_i)}$*); and*

(iii) $V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e)$.

When $p = 0$ we write simply $T \to T_1, \ldots, T_e$.

A crucial operation within INTERSECT is to compute polynomial GCDs modulo a regular chain. Recall that Definition 2.26 gave a definition for a regular GCD over $R[y]$ where R was not necessarily a UFD. In practice, the ring R is of the form $\mathbb{K}[\underline{X}]/\sqrt{\text{sat}(T)}$ for some regular chain T. Thus, a regular GCD of p and q in $(\mathbb{K}[\underline{X}]/\sqrt{\text{sat}(T)})[y]$ is also called a regular GCD of p and q modulo $\sqrt{\text{sat}(T)}$.

The ability to effectively compute regular GCDs relies on several properties. First, recall that $\text{sat}(T)$ is unmixed. Therefore, $\sqrt{\text{sat}(T)}$ is a finite intersection of prime ideals which are also the associated prime ideals of $\text{sat}(T)$. This implies that any regular element of $\mathbb{K}[\underline{X}]/\text{sat}(T)$ is also a regular element of $\mathbb{K}[\underline{X}]/\sqrt{\text{sat}(T)}$.

Regular GCDs modulo a regular chain have many useful properties, summarized by the following proposition. In the following, for a regular chain T and a polynomial p, let $T \cup p$ be shorthand for $T \cup \{p\}$.

Proposition 2.37. *Let $T \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$. Let $p, t, g \in \mathbb{K}[x_1, \ldots, x_i]$ all have main variable x_i. Assume $T \cup t$ is a regular chain and that g is a regular GCD of p, t modulo $\sqrt{\text{sat}(T)}$. Then, we have:*

(i) if $\text{mdeg}(g) = \text{mdeg}(t)$*, then* $W(T \cup t) \subseteq Z(\text{init}(g), T \cup t) \cup W(T \cup g) \subseteq \overline{W(T)}$ *and* $\sqrt{\text{sat}(T \cup t)} = \sqrt{\text{sat}(T \cup g)}$ *both hold;*

(ii) if $\text{mdeg}(g) < \text{mdeg}(t)$*, let* $q = \text{pquo}(t, g)$*, then $T \cup q$ is a regular chain,*

(ii.a) $\sqrt{\text{sat}(T \cup t)} = \sqrt{\text{sat}(T \cup g)} \cap \sqrt{\text{sat}(T \cup q)}$*, and*

(ii.b) $W(T \cup t) \subseteq Z(\text{init}(g), T \cup t) \cup W(T \cup g) \cup W(T \cup q) \subseteq \overline{W(T \cup t)}$*;*

(iii) $W(T \cup g) \subseteq V(p)$*; and*

(iv) $V(p) \cap W(T \cup t) \subseteq W(T \cup g) \cup (V(p, \text{init}(g)) \cap W(T \cup t)) \subseteq V(p) \cap \overline{W(T \cup t)}$

Proof. [49, Proposition 3.2] □

Notice that the splittings caused by a regular GCD, namely $W(T \cup t)$ into $W(T \cup g)$ and $W(T \cup q)$, follows closely with the idea of the D5 principle and splitting a DPF with a GCD computation. If we assume that $T \cup t$ is a square-free regular chain (see [17]) then $W(T \cup g)$ represents everywhere that p is 0 and $W(T \cup q)$ represents everywhere that p is regular.

Example 2.38. Let $T = \{z^2 + z - 2, y^2 - 1\}$, $t = x^2 - xy$, and $p = x^2 - 1$. Notice that p is a zero divisor modulo $\text{sat}(T \cup t)$. Indeed, computing a regular GCD of p and t modulo $\sqrt{\text{sat}(T)}$ confirms this by giving $g = -xy + 1$. Then, $q = \text{pquo}(t, g) = -xy + y^2 - 1$, which simplifies to x modulo T.[5] $T \cup g$ and $T \cup q$ are

$$T \cup g = \begin{cases} -xy + 1 \\ y^2 - 1 \\ z^2 + z - 2 \end{cases} \quad \text{and} \quad T \cup q = \begin{cases} x \\ y^2 - 1 \\ z^2 + z - 2 \end{cases}$$

and we easily find that p is not regular modulo $\text{sat}(T \cup g)$ (since either $y = 1, x = 1$ or $y = -1, x = -1$ and thus p reduces to 0) but is regular modulo $\text{sat}(T \cup q)$ (since $x = 0$ and p reduces to 1).

Recall that Intersect(p, T) is a function to compute (an approximation of) $V(p) \cap W(T)$. This example highlights one very important subroutine of Intersect, namely regularity testing, which we call Regularize. Splitting a regular chain in this way to determine where polynomials are regular or not is the foundation of all *component-level parallelism* in triangular decomposition. Indeed, such a split will eventually allow for Intersect to return multiple components.

There are many subroutines for computing an incremental triangular decomposition, including Intersect, RegularGCD, and Regularize. These routines are detailed in [47]. We will review these routines, and examine some of them closely, in Chapter 6 in order to discuss their opportunities for parallel computation.

2.6 Limit Points and Power Series

The previous section defined triangular decompositions in the sense of Lazard-Wu and in the sense of Kalkbrener. To illustrate the difference between these two, and the concept of limit points, consider the following example.

Example 2.39. Let $F = \{ax + b,\ bx + y\} \subset \mathbb{Q}[a < b < y < x]$. A Kalkbrener decomposition of F produces the regular chain $R_1 = \{bx + y,\ ay - b^2\}$. Indeed, one may

[5] T fixes $y = \pm 1$, hence $y^2 - 1 = 0$ and $y \neq 0$, reducing q to x.

verify that $V(F) = \overline{W(R_1)}$. Working in the field of fractions, R_1 describes the solutions to F of the form:

$$\begin{cases} x = {}^{-y}/_{b} \\ y = {}^{b^2}/_{a} \end{cases}.$$

This excludes solutions where $a = 0$ or $b = 0$; therefore $W(R_1) \subsetneq \overline{W(R_1)} = V(F)$. It is easy to check that the missing solutions are:

$$R_2 = \begin{cases} x = 0 \\ y = 0 \\ b = 0 \end{cases}, \text{ and } R_3 = \begin{cases} y = 0 \\ a = 0 \\ b = 0 \end{cases}.$$

Therefore, $\{R_1, R_2, R_3\}$ form a Lazard-Wu decomposition of $V(F)$, and, we have:

$$V(F) = (V(R_1) \setminus V(ab)) \cup V(R_2) \cup V(R_3), \text{ and}$$

$$\overline{W(R_1)} \setminus W(R_1) = V(R_2) \cup V(R_3).$$

Recalling the Zariski topology defined in Section 2.3, we can see that the "missing pieces" to make $W(R_1)$ a variety are the two lines given by $V(R_2)$ and $V(R_3)$. Those missing pieces are also the difference between a Kalkbrener decomposition and a Lazard-Wu decomposition. Since the former is much easier to compute, having an additional method to compute these missing pieces from a Kalkbrener decomposition is useful in practice. These missing pieces are described more formally as limit points.

Definition 2.40 (Limit point)**.** *Let (X, τ) be a topological space, and $S \subset X$ be a subset. A point $p \in X$ is a* limit point *of S if every neighbourhood of p contains at least one point of S different from p itself.*

In the context of the affine space $\overline{\mathbb{K}}^n$ and the Zariski topology, we may define the *non-trivial limit points* of the quasi-component $W(R)$, for some regular chain R, as the set $\overline{W(R)} \setminus W(R)$. Intuitively, this describes the closure of $W(R)$ as the union of $W(R)$ and all of its limit points.

It it worth noting that, when $\mathbb{K} = \mathbb{C}$, the affine space $\overline{\mathbb{K}}^n$ is endowed with both the Zariski topology and the usual Euclidean topology. In particular we have the following:

Lemma 2.41. *Let $S \subseteq V$ be a Zariski open subset of some irreducible variety V in $\mathbb{C}^n$. The closure of S in the Zariski topology and the closure of S in the Euclidean topology are both equal to V.*

Proof. [147, Corollary 1, Section 1.10]. □

Lemma 2.41 has useful consequences. In particular, one can compute the limit points of a quasi-component using limits in the usual sense of the Euclidean topology. When a quasi-component is given by a regular chain of dimension one, its closure is some algebraic curve. Its limit points can therefore be computed as limits (in the usual sense) of sequences of points along "branches" of the algebraic curve. In turn, these branches can be computed as the terms of Puiseux series—generalized power series with the possibility of negative and fractional exponents.

Details and algorithms for computing such limit points as Puiseux series are described in [4] and [3]. In this thesis, we do not directly compute limit points, but rather implement efficient power series and Hensel factorization (see Chapter 7). These tools, in turn, may be used to implement the *Extended Hensel Construction*. The Extended Hensel Construction can then be used to compute the limit points of quasi-components or the limits of multivariate rational functions [4].

In the classical Newton–Puiseux theorem (see, e.g., [77]), any bivariate polynomial with complex coefficients $F(X_1, Y)$ can have its solutions in Y expressed as Puiseux series in X_1. Equivalently, the polynomial F can be factored into linear factors in Y. The *Hensel–Sasaki Construction* or *Extended Hensel Construction* (EHC) was proposed in [155] as an efficient alternative to the Newton–Puiseux method. In the same paper, an extension of the Hensel–Sasaki construction for multivariate coefficients was suggested. That is, to compute the solutions in Y of a polynomial $F(X_1, \ldots, X_n, Y)$. This method was later extended in, e.g., [103] and [156]. EHC was further improved in terms of algebraic complexity and practical implementation in [4]. When the polynomial to be factored is monic, its solutions in Y are actually power series in $X_1, \ldots, X_n$, rather than Puiseux series. We conclude this section with a basic review of formal (i.e. not necessarily convergent) power series.

2.6.1 Formal Power Series

Power series are polynomial-like objects with potentially infinite terms. Let $\mathbb{K}$ be an algebraic number field and $\overline{\mathbb{K}}$ be its algebraic closure. The ring of formal power series with coefficients in $\mathbb{K}$ and variables in $X_1, \ldots, X_n$ is denoted $\mathbb{K}[[X_1, \ldots, X_n]]$. Let $f = \sum_{e \in \mathbb{N}^n} a_e X^e$ be a formal power series where $a_e \in \mathbb{K}$, $e = (e_1, \ldots, e_n)$ is a multi-index with $|e| = e_1 + \cdots + e_n$, and X^e stands for $X_1^{e_1} \cdots X_n^{e_n}$.

Let $k \in \mathbb{N}$, where we take $0 \in \mathbb{N}$. The *homogeneous part* and *polynomial part* of f in degree k are denoted by $f_{(k)}$ and $f^{(k)}$, and defined by:

$$f_{(k)} = \sum_{|e|=k} a_e X^e \quad \text{and} \quad f^{(k)} = \sum_{i \leq k} f_{(i)}. \tag{2.4}$$

Let $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$ be two power series. The *sum*, *difference*, and *product* of f and g are given by

$$\begin{aligned} f \pm g &= \sum_{k \in \mathbb{N}} (f_{(k)} \pm g_{(k)}) \\ fg &= \sum_{k \in \mathbb{N}} \left(\sum_{i+j=k} (f_{(i)}\, g_{(j)}) \right). \end{aligned}$$

Consider the following example to clarify this definition.

Example 2.42. Let

$$f = 1 + 3X_1 + \frac{5}{2}X_2 + 2X_1X_2 + 5X_1^2 + \cdots, \qquad g = 3 + 2X_1X_2 - 4X_1X_2^2 - \frac{1}{2}X_1^3X_2 + \cdots$$

It follows from (2.4) that:

$$\begin{gathered} f_{(0)} = 1, \quad f_{(1)} = 3X_1 + \frac{5}{2}X_2, \quad f_{(2)} = 2X_1X_2 + 5X_1^2, \\ g_{(0)} = 3, \quad g_{(1)} = 0, \quad g_{(2)} = 2X_1X_2. \end{gathered}$$

Hence, we have the following arithmetic results:

$$\begin{aligned} f + g &= \big(f_{(0)} + g_{(0)}\big) + \big(f_{(1)} + g_{(1)}\big) + \big(f_{(2)} + g_{(2)}\big) + \cdots \\ &= 4 + 3X_1 + \frac{5}{2}X_2 + 4X_1X_2 + 5X_1^2 + \cdots \\ f - g &= \big(f_{(0)} - g_{(0)}\big) + \big(f_{(1)} - g_{(1)}\big) + \big(f_{(2)} - g_{(2)}\big) + \cdots \\ &= -2 + 3X_1 + \frac{5}{2}X_2 - 5X_1^2 + \cdots \\ f\, g &= (f_{(0)}g_{(0)}) + (f_{(0)}g_{(1)} + f_{(1)}g_{(0)}) + (f_{(0)}g_{(2)} + f_{(1)}g_{(1)} + f_{(2)}g_{(0)}) + \cdots \\ &= 3 + 9X_1 + \frac{15}{2}X_2 + 8X_1X_2 + 15X_1^2 \end{aligned}$$

In this previous example we say that f and g are known to *precision* 2, since terms up to total degree 2 are known.

The *order* of a formal power series $f \in \mathbb{K}[[X_1, \ldots, X_n]]$ is defined as:

$$\operatorname{ord}(f) = \begin{cases} \min\{k \mid f_{(k)} \neq 0\} & \text{if } f \neq 0 \\ \infty & \text{if } f = 0 \end{cases}.$$

By extension, we have the following:

$$\operatorname{ord}(f+g) \geq \min\{\operatorname{ord}(f), \operatorname{ord}(g)\} \quad \text{and} \quad \operatorname{ord}(fg) = \operatorname{ord}(f) + \operatorname{ord}(g).$$

We now recall several properties of the ring of formal power series which will be useful later in Chapter 7.

(*i*) $\mathbb{K}[[X_1, \ldots, X_n]]$ is an integral domain,

(*ii*) the set $\mathcal{M} = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \operatorname{ord}(f) \geq 1\}$ is the only maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$,

(*iii*) for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \operatorname{ord}(f) \geq k\}$.

Notice that the maximal ideal $\mathcal{M}$ is the set of all power series whose constant term is 0. Moreover, $\mathcal{M}^k \setminus \mathcal{M}^{k-1}$ is the set of all terms (and sums of terms) with total degree k.

Although the ring of formal power series is an integral domain, it is not a field since some elements do not have a multiplicative inverse. To see this, we begin with the notion of convergence under the Krull topology.

Definition 2.43. Let $(f_n)_{n\in\mathbb{N}}$ be a sequence of elements of $\mathbb{K}[[X_1, \ldots, X_n]]$ and let $f \in \mathbb{K}[[X_1, \ldots, X_n]]$. We say that:

(*i*) $(f_n)_{n\in\mathbb{N}}$ *converges* to f if for all $k \in \mathbb{N}$ there exists $N \in \mathbb{N}$ such that for all $n \in \mathbb{N}$ we have $n \geq N \quad \Rightarrow \quad f - f_n \in \mathcal{M}^k$,

(*ii*) $(f_n)_{n\in\mathbb{N}}$ is a *Cauchy sequence* if for all $k \in \mathbb{N}$ there exists $N \in \mathbb{N}$ such that for all $n, m \in \mathbb{N}$ we have $n, m \geq N \quad \Rightarrow \quad f_m - f_n \in \mathcal{M}^k$.

As a consequence, the maximal ideal $\mathcal{M}$ of $\mathbb{K}[[X_1, \ldots, X_n]]$ has the additional property that $\bigcap_{k\in\mathbb{N}} \mathcal{M}^k = \langle 0 \rangle$. Moreover, if every Cauchy sequence in $\mathbb{K}$ converges, then so too does every Cauchy sequence in $\mathbb{K}[[X_1, \ldots, X_n]]$.

Proposition 2.44. For all $f \in \mathbb{K}[[X_1, \ldots, X_n]]$, the following properties are equivalent

(*i*) f is a unit

(*ii*) $\operatorname{ord}(f) = 0$

(*iii*) $f \notin \mathcal{M}$.

This proposition follows from the classical observation that for $f \in \mathbb{K}[[X_1, \ldots, X_n]]$, with $\operatorname{ord}(f) > 0$, the sequence $(u_n)_{n\in\mathbb{N}}$, where $u_n = 1 + g + g^2 + \cdots + g^n$ and $g = 1 - {}^{f}/_{f_{(0)}}$, converges to the inverse of ${}^{f}/_{f_{(0)}}$.

2.7 Symbols and Notations

In this section we summarize all of the symbols and notations used throughout this thesis.

Rings, Ideals, and Varieties

- R is a commutative ring with identity
- $\mathbb{D}$ is an integral domain, GCD domain, Euclidean domain, or Unique Factorization Domain by context.
- $\mathbb{K}$ is a field and $\overline{\mathbb{K}}$ its algebraic closure.
- $\mathbb{Z}/n\mathbb{Z} := \mathbb{Z}/n := \mathbb{Z}_n$ all denote the finite ring of integers modulo n.
- $\mathcal{I}, \mathcal{J}, \langle p \rangle$ denote ideals.
- $\mathbb{K}[x]$ is a univariate polynomial ring, $\mathbb{K}[x_1, \ldots, x_n] := \mathbb{K}[\underline{X}]$ is a multivariate polynomial ring.
- $\overline{\mathbb{K}}^n$ is an n-dimensional affine space.
- V is an affine algebraic variety.
- $V(f_1, \ldots, f_k) = V(\langle f_1, \ldots, f_k \rangle)$ is the variety defined by $f_1, \ldots, f_k$.
- $I(S)$ is the vanishing ideal of some affine subset $S \subseteq \overline{\mathbb{K}}^n$.
- $S_k(a, b) := S_k$ denotes the kth subresultant between a and b, and s_k denotes the coefficient of degree k of S_k.
- In an ordered polynomial ring $\mathbb{K}[x_1 < x_2 < \cdots < x_n]$:
  - $\text{mvar}(p)$ is the main variable of p, the largest variable appearing in p;
  - $\text{mdeg}(p)$ is the degree of p in its main variable;
  - $\text{init}(p)$ is the leading coefficient of p with respect to its main variable;
  - $\text{tail}(p)$ is the reductum of p with respect to its main variable.

Triangular Sets and Regular Chains

- $T \subset \mathbb{K}[\underline{X}]$ is a triangular set or regular chain.
- $\mathrm{mvar}(T)$ is the set of main variables of polynomials in T.
- T_v is the polynomial in T with main variable v.
- T_v^- (resp. T_v^+) is the set of polynomials in T with main variable less (resp. greater) than v.
- h_T is the product of initials of polynomials in T.
- $\mathrm{sat}(T) = \langle T \rangle : h_T^\infty$ is the saturated ideal of T.
- $\dim(T)$ is the dimension of $\mathrm{sat}(T)$.
- $W(T) = V(T) \setminus V(h_T)$ is the quasi-component of T.
- $\overline{W(T)}$ is the Zariski closure of $W(T)$.
- $Z(p, T) := V(p) \cap W(T)$.
- $T \to T_1, \ldots, T_e$ is a *regular split* of T.

Power Series

- $\mathbb{K}[[X_1, \ldots, X_n]]$ is the ring of formal multivariate power series over $\mathbb{K}$.
- $f_{(k)}$ is the homogeneous part of degree k of f.
- $f^{(k)}$ is the polynomial part of degree k of f.
- $\mathrm{ord}(f)$ is the order of f.

Chapter 3

Computational Background

In this chapter we discuss various computational aspects related to our goals of designing high-performance and parallel computer algebra routines. We begin in Section 3.1 with a discussion of data locality and cache complexity, recalling the importance of locality on modern computer architectures with cache memory hierarchies. Next, we review fundamental aspects of parallel computing and multithreaded programming in Section 3.2. We also review important aspects of so-called "Modern C++" in Section 3.3. These modern language constructs enable the design and implementation of our polynomial system solver and the entire BPAS library.

3.1 Data Locality and Cache Complexity

Since the 1980s, processor speeds and memory speeds have diverged exponentially. Today, processors are roughly 3 orders of magnitude faster than main memory. This difference is called the *processor-memory gap* and is a key contributor to the *memory wall*—the point at which a program's performance is completely determined by the speed of memory. To combat this gap, cache memory hierarchies were introduced. The memory hierarchy is designed using the *principle of locality*—programs tend to reuse data and instructions which they have used recently [95, Section 1.9]. Recently accessed data, and data which is adjacent to recently accessed data, is transferred from main memory and stored in a cache. Repeated accesses to the same data is known as *temporal locality* while accessing data adjacent to recently accessed data is known as *spatial locality*.

When a programmer adheres to the principle of locality in their own programs (i.e. most often accesses the cache rather than main memory), the program's performance may improve significantly. Indeed, cache latency is on the order of a processor's clock cycle, and thus significantly faster than accessing main memory. If the memory hierarchy

is not considered, the cost of accessing main memory and transferring data into the cache can dominate the running time of an algorithm's execution. This idea can be formalized using *cache complexity* and the *ideal cache model*.

The cache complexity of an algorithm estimates the negative impact on performance caused by data transfers between the cache and the main memory of a computer executing that algorithm. This observation motivated the introduction of the *ideal-cache model*, by Frigo, Leiserson, Prokop, and Ramachandran in 1999; see the extended journal version [79].

Despite the strong assumptions of the ideal-cache model (optimal replacement, exactly two levels of memory, full associativity), designing algorithms that minimize costs in that model is rewarding in practice. The authors of [79] show that algorithms designed in the ideal-cache model can be efficiently simulated by weaker models. They show how to simulate the optimal replacement assumption with the typical *least-recently used* (LRU) policy used by modern processors.

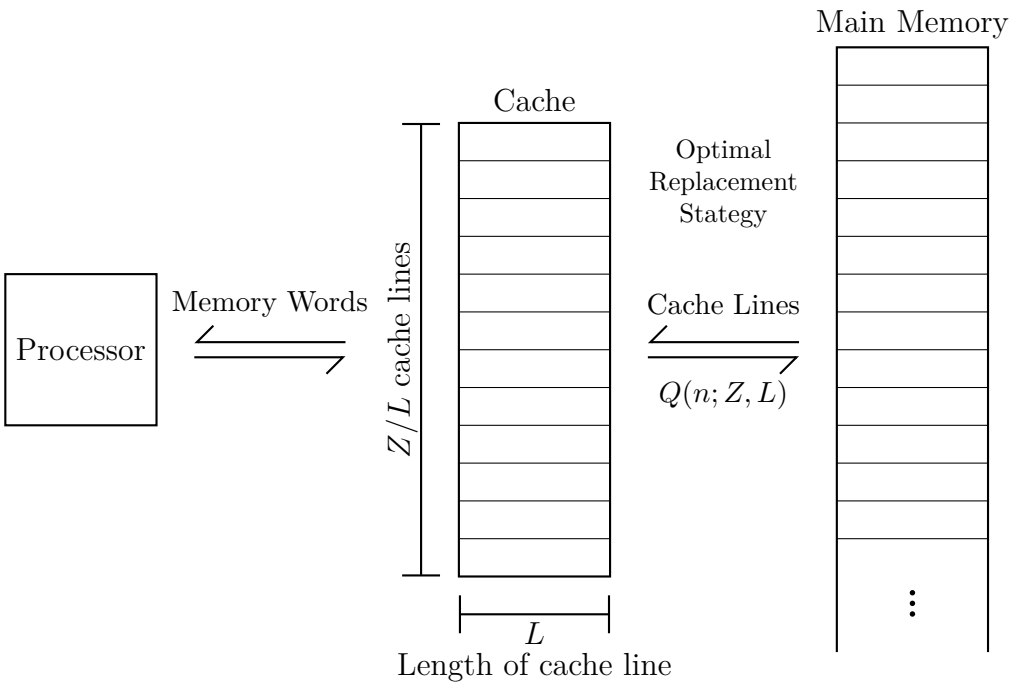

Figure 3.1: The ideal cache model has one cache and one backing memory. The processor can only access memory words stored in the cache whose capacity is Z. Memory words are transferred L words at a time between main memory and cache.

In this model, the processor has a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. The cache is partitioned

into Z/L *cache-lines*, where L is the length of each cache-line; see Figure 3.1. A cache line represents the consecutive memory words that are always moved as a group between the cache and the main memory. Thus, accessing a single word in main memory actually retrieves that word and its neighbours as a cache line. In order to achieve spatial locality, one assumes $L > 1$, which eventually mitigates the overhead of moving the cache-line from the main memory to the cache. To achieve *temporal locality*, one also assumes that the cache is *tall*, that is, that there are significantly more cache lines L than number of bytes per cache line Z. In the ideal-cache model, it is assumed that $Z \in \Omega(L^2)$ [79].

In the ideal-cache model, the processor can only read and write to memory words that reside in the cache. If the line of a referenced word is found in cache, then that word is delivered to the processor for processing. This situation is called a *cache hit*. Otherwise, a *cache miss* occurs, and the line is first fetched from main memory and *installed* in the cache before transferring the requested word to the processor. In this model, a cache-line may be installed anywhere in the cache; this particular mapping from memory to cache is called *fully associative*. If the cache is full, a cache-line must be *evicted* before a new one can be installed. The ideal cache uses the optimal off-line (statically determined before runtime) cache replacement policy to perfectly exploit temporal locality. In this policy, a cache-line which is never accessed again in the future is evicted; if no such cache-line exists, then the line whose next access is furthest in the future is evicted [20].

The ideal cache model measures the cost of memory accesses through *cache complexity*, $Q(n; Z, L)$, representing the number of cache misses the algorithm incurs as a function of the input data size n, the cache size Z, and the cache line size L. When Z and L are clear from context, the cache complexity can be denoted simply by $Q(n)$.

Optimizing cache complexity is a generally difficult problem. A general strategy is to improve the locality of a program. This may take the form of minimizing "working memory" of the algorithm, minimizing memory usage of data structures, avoiding data copies (e.g. pass-by-reference, move constructors), or improving loop design. The latter is particularly important. Multi-dimensional data structures must be *linearized* into main memory. This may be *row-major order* or *column-major order*, depending on the programming language. For example, given a matrix, either adjacent elements in a row are adjacent in memory, or adjacent elements in a column are adjacent in memory, but not both.

A typical optimization strategy is *blocking*, which is typically derived from a divide-and-conquer mechanism; see, e.g., [119]. In this strategy, the data is separated into blocks and each block processed recursively.[1] However, blocking typically relies on some

[1] One can always "unwind" the recursion into an iterative scheme based on loops, where the innermost

property of the algorithm or data which allows it to be divided. Loosely speaking, the data or algorithm must be somehow "regular". This notion is similar to the idea of regular and irregular parallelism, described later in Section 3.2.

The performance decreases caused by cache misses are only worsened by parallel programs. When a computer has multiple processors, or a processor has multiple cores (i.e. in multithreaded parallelism), individual cores have individual caches but share a single backing memory. This creates the *cache coherence problem*, where a processor reading a memory word (from its cache or otherwise) must obtain the data most recently written to that word. Since each cache is distinct, there must be additional mechanisms to ensure that a write occurring in one cache is "seen" by the others. Cache coherence protocols (see [95, Ch. 5]) which maintain cache coherence may add significant overheads to a parallel program. In particular, they cause addition types of cache misses called *sharing misses*.

In a *true sharing cache miss*, one cache is attempting to access a memory word which was written to by another cache. In a *false sharing cache miss*, one cache is attempting to access a memory word whose *cache line*, but a different memory word, was written to be another cache. The former occurs when two or more cores are accessing the same data. The latter occurs when two or more cores are accessing data which is too close together. True sharing is inherently avoided in parallel programming as one looks to avoid all *data races*. On the other hand, false sharing is not as easy to recognize nor avoid. Yet, cache complexity can still be defined in this case. The ideal cache model has been adapted to algorithms on multi-core architectures in [80]. Generally, the more *finely grained* the parallelism the more likely it is for false sharing to occur. Parallel granularity is defined in the next section.

3.2 Parallel Programming Basics

Parallel programming is concerned with writing programs and algorithms which employ multiple processors to execute multiple calculations simultaneously, that is, parallel computation. This contrasts with concurrent computing, where parts of a program may be executed during overlapping time periods, but not necessarily at the exact same moment; e.g., via multitasking, coroutines, or futures (see [160, Ch. 8] and [96, Ch. 16]).

Concurrency may be made explicitly parallel through the hardware support of multi-core processors and (shared memory) multiprocessors [151, Ch. 6]. Such parallelism is typical of, and best utilized by, a single workstation or single compute node. We are

loop processes individual blocks.

concerned with this type of parallelism rather than distributed computing which employs multiple interconnected but independent computers.[2] The relation between concurrent and parallel computing suggests that an algorithm or program should first be analyzed to find opportunities for concurrency and then programmed to exploit that concurrency.

Parallelism in parallel computation may take several forms; see [131, Ch. 1–2] for an introduction and informative discussion. Parallelism is typically grouped into two categories: *data parallelism* or *task parallelism*. In data parallelism, the data on which calculations are to be executed is partitioned and different processors execute (the same) operations over each partition. This may include executing vector (SIMD) instructions over a number of data elements simultaneously, or executing an operation over each element in a collection. In task parallelism, the operations or tasks to execute are themselves distributed across processors and executed simultaneously. Task parallelism is often realized as *functional decomposition*, where different program functions are executed simultaneously.

Data parallelism is often seen as *scalable* parallelism, where the amount of concurrency that can be exploited increases with increasing hardware resources (e.g. cores). On the other hand, task parallelism is often viewed as not being scalable, despite, for example, divide-and-conquer task parallelism being scalable. A less ambiguous categorization of parallelism is thus *regular parallelism* and *irregular parallelism* [131, Section 2.2]

Regular parallelism describes when the work to be executed in parallel can be easily and evenly decomposed into units with predictable data and control dependencies. In regular parallelism, the decomposition of work is often an algorithmic property known *a priori*, such as in a divide-and-conquer algorithm or a block-wise matrix computation. Irregular parallelism is the opposite, where the decomposition results in unbalanced work, dissimilar tasks, and unpredictable dependencies.

In either case, the decomposition of the overall work into smaller units is measured qualitatively though *granularity*. In *coarse-grained parallelism* the amount of work associated with each task (or data partition) is high but the total number of tasks may be low, and thus may not fully utilize available hardware resources. In *fine-grained parallelism* the amount of work associated with each task is low, possibly allowing for *parallel overheads* to dominate execution time. Generally, larger numbers of coarser-grained tasks are preferred, since this allows for good resource utilization, and less dominant overheads.

Parallel overhead describes time spent managing the parallel execution of a program

[2] With current algorithms and hardware technologies, there is not much to gain in triangular decomposition by moving to distributed computing. The concurrency opportunities are too irregular. Moreover, very hard problems are limited by expression swell rather than computing power.

that would not otherwise exist if the same program was executed serially. We have already discussed false sharing cache misses as one example. Sources of parallel overhead differ depending on the programming mechanisms enabling parallelism. In this work we are concerned with *thread-level parallelism*.

3.2.1 Thread-level Parallelism

Thread-level parallelism is the application of *threads*—independent control flows of execution within a single process—to execute work concurrently. For threads to actually enable parallelism, the executing hardware must support them through shared memory multiprocessors (multiple independent processors within the same computer), or multi-core processors (multiple processors/"cores" in a single integrated circuit) [151, Ch. 6]. Each processor or core is able to execute an independent thread. We thus differentiate between *hardware threads*—a hardware entity capable of independently executing program code—and *software threads*—a virtual abstraction of a hardware thread.

A program is capable of creating as many software threads as it wishes (up to some large limit defined by the operating system). However, it is generally advised to not have more active software threads than the number of available hardware threads. Firstly, doing so necessitates *multithreading*, where a single processor handles multiple threads concurrently but not necessarily in parallel; see [151, Section 6.4]. Secondly, and in all cases, programmers must be concerned with the parallel overheads of thread parallelism.

(*i*) *Spawning*, i.e. creating and initializing, a thread is not an insignificant amount of work and can become costly if many threads are spawned throughout a program's lifetime. *Joining* a thread is the act of terminating a thread and merging its control flow into another. This operation can also be costly.

(*ii*) *Over-subscription*, the case when a program spawns more software threads than hardware threads, causes threads to share hardware resources (as in multithreading) and thus penalizes performance through the cost of repetitive *context switching* between threads.

(*iii*) *Load-balance* refers to evenly distributing work between threads so that none are idle while there is still work to do. Load-*imbalance* hinders parallel performance.

(*iv*) *Inter-thread communication*, often implemented through access to a shared piece of data, and *synchronization* between threads, should be minimized. Inter-thread communication requires costly mechanisms to serialize access to the shared data.

Similarly, synchronization requires at least one of the synchronizing threads to be idle and left waiting for the other(s).

The cost of spawning and the penalties of over-subscription can be mitigated through the use of a *thread pool*. This structure spawns and maintains many long-running threads to serve as "workers". Spawning a small number of threads only once at the beginning of the program, and keeping them active throughout the program's lifetime, minimizes spawning overheads. Moreover, by limiting the number of threads spawned by a thread pool to be less than or equal to the number of threads supported by the current hardware, over-subscription is easily avoided. When this limited number of worker threads all become busy, there are several options. Control flow may stall until a thread becomes idle, the targeted code segment may be executed serially, or the code segment may be added to a queue of tasks for a worker to execute once it becomes idle.

Where inter-thread communication is absolutely necessary, it should be organized and implemented efficiently. For example, synchronization is required to avoid *data races*—where the non-deterministic order of access to data by threads may cause inconsistent runtime behaviour [131, Section 2.6]. Effectively organizing parallel code is precisely the goal of *parallel patterns* or *algorithmic skeletons* [131, Ch. 3]. We will explore some examples of parallel patterns in Section 5.1, and their application to computer algebra in Section 6.3 and Chapter 7.

It is worthwhile to note that over-subscription may be avoided only if the *total* number of active threads does not exceed the number of hardware threads. If a program is to combine the use of a thread pool with, for example, additional explicitly parallel regions or parallel external libraries, the total number of active threads should not exceed the hardware-imposed limit.

Therefore, a notable challenge is the composition of parallel regions of a program. In particular, how to effectively handle the cooperation between threads executing those parallel regions. One possible solution is *dynamic multithreading* where programmers specify opportunities for concurrency while a runtime scheduler or *concurrency platform* dynamically chooses which regions to execute in parallel. For example, the *Cilk* concurrency platform [124] is based on the *fork-join model* of parallel computation and a shared thread pool.

3.2.2 Fork-Join Model

The fork-join model of computation is one technique for organizing and analyzing parallel computation. The model is based on two operations: the *fork* branches the computation

into multiple parallel executions, and the *join* merges two or more branches back into a single serial execution. Forking occurs recursively to allow for a greater number of active branches of computation.

This exercise of forking and joining organizes the computation into a rooted directed acyclic graph (DAG), often called a *spawn tree* [26]. In this DAG, nodes are a maximal length sequence of instructions between any fork or join, and edges are a fork or a join.

The fork-join model readily maps to an implementation. In its simplest form, a fork corresponds to spawning a new thread and a join corresponds to joining a thread. However, this can quickly lead to over-subscription and excessive overheads in spawning and joining. Rather, a fork is usually just a suggestion to execute a region in parallel. In the *Cilk* concurrency platform, a pool of *worker threads* is initialized at the beginning of the program, each with a task queue. A fork corresponds to enqueuing a task to a worker's task queue. Then, a *work-stealing scheduler* [26] allows idle workers to steal tasks and achieve load-balance between workers. Similar ideas are employed by our cooperative threading implementation detailed later in Chapter 5.

The fork-join model is also very useful for analyzing the performance of a parallel computation; see [131, Section 2.5]. Representing the computation as a DAG allows for simple performance measures, namely work and span.

- The *work* of a parallel program is the sum of instructions in all the nodes of a spawn tree. It is denoted by T_1 and represents the running time of the parallel program if executed serially on a single processor.
- The *span* of a parallel program is the total number of instructions along the longest path (the critical path) of the DAG. It is denoted by T_∞ and represents the running of the parallel program if executed on infinitely many processors.

From work and span we derive several other important metrics and properties.

- The *parallelism* of the program is $T_1 \,/\, T_\infty$ and represents the average amount of work executed for each step along the span.
- If the spawn tree is executed in parallel by p processors, the maximum number of instructions executed by any one processor (assuming optimized load-balance) is denoted by T_p. It represents the running time of the program on p processors.
- The *work law* says that $T_p \geq T_1 \,/\, p$.
- The *spawn law* says that $T_p \geq T_\infty$.
- The *(theoretical) parallel speed-up* on p processors is $T_1 \,/\, T_p$.

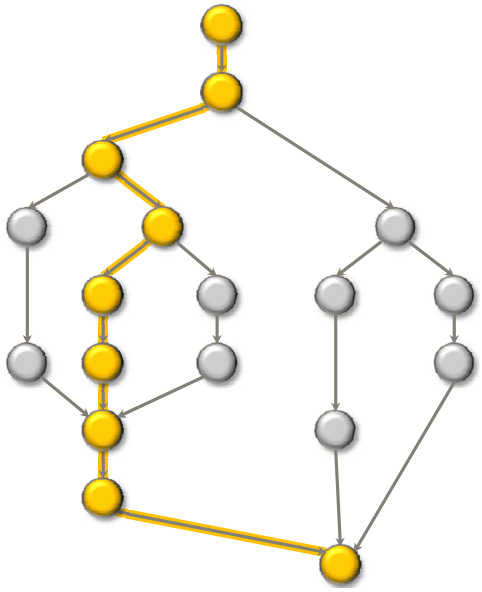

Figure 3.2: A spawn tree whose highlighted nodes show its span.

The fork-join model has many practical consequences. By modeling a program and its concurrency opportunities as a spawn tree, we can determine an upper bound on the program's possible parallel speed-up. We want the span T_∞ to be low, the parallelism $T_1 \,/\, T_p$ to be high, and the parallel speed-up $T_1 \,/\, T_p$ to be nearly equal to p (to achieve *linear speed-up*).

In practice, overheads will reduce the actual parallel speed-up achieved. Nonetheless, disciplined methods exist to minimize overheads and efficiently organize parallel code. One may analyze algorithms for concurrency opportunities, and then implement the parallelism using parallel patterns. Section 6.3 describes this for the case of triangular decomposition.

3.3 Modern C++

Modern C++ refers to the C++ language since the C++11 specification. Compared with the previous C++98 and C++03 specifications, C++11 may feel like an entirely new language. There have been significant library additions and idiomatic changes which may elude veteran C++ programmers. We take this section to highlight some notable changes which we eventually employ in our design and implementation; see Chapters 4 and 5. Common knowledge of basic C++ is assumed, otherwise, see [167]. In this section, we review the *Type Support Library*, the *Function Objects Library*, and the *Thread Support Library*. For further details on modern C++, see the book of Meyers [133].

3.3.1 Type Support Library and Template Metaprogramming

The *Type Support Library* provides definitions of types, macros, and other expressions that can be evaluated at compile-time, for obtaining type information and making type modifications. We are particularly concerned with *type traits*—expressions which perform *compile-time introspection* of types. Type traits are a particular kind of *template metaprogramming* (TMP) which allows code to evaluate expressions at compile-time and then accordingly modify itself during the compilation process. Using templates for metaprogramming has been known for quite some time; see [172, Ch. 16–17]. However, with the introduction of type traits in C++11, TMP became ingrained in the language.

Compile-time introspection is used to determine truth values (among other things) about a type at compile-time. The resulting Boolean can then be used in other template metaprogramming features to conditionally change the code and the way it is compiled. This introspection is based on the *Substitution Failure Is Not An Error* (SFINAE) principle, coined by Vandevoorde in [172]. The invalid substitution of a type as a template parameter is itself not an error; where two or more template specializations exist, only one is required to be correct. This principle, combined with compile-time *function overload resolution*, provides template metaprogramming its power.

Consider the typical example, adapted from [172, Section 8.3], shown in Listing 3.1. `type_has_X` determines if a type has a member `X` by checking the size of the return type of a function. By function overload resolution, if `T` has a member `X` the `test<T>` function chosen will be the first, whose return type has size 1. Otherwise the second function is chosen with return type of size (at least) 2.

```
1 template<typename T> char test(typename T::X const*);
2 template<typename T> int  test(...);
3 #define type_has_X(T) (sizeof(test<T>(NULL)) == 1);
```

Listing 3.1: A simple compile-time introspection to determine if type T has member X.

The use of templates for this kind of compile-time introspection underlies all type traits. Some important examples implemented in the *Type Support Library* are defined below.[3] Of course, all evaluations occur at compile-time.

- `is_base_of<Base, Derived>` defines a struct whose `value` variable evaluates to true if and only if `Base` is the superclass of `Derived` or they are the same type.

[3] `Derived_from` is only included in the library since C++20. But, it has been a well-known trick for many years [165].

- `conditional<B, T, F>` defines a struct whose `type` variable evaluates to the type `T` if `B` is true, and `F` otherwise.

- `Derived_from<T, Base>` defines a struct which produces a compiler error if the type `T` is *not* `Base` or a subclass of `Base`, otherwise, defines a do-nothing method.

As we will see in Chapter 4, the use of type traits allows for interesting, adaptive, and dynamic type constructs. In particular, we use them to ensure compile-time type safety, and to perform so-called "dynamic" type creation and "conditional export".

3.3.2 Function Objects Library

The *Function Objects Library* defines objects supporting a function call operator and provides support for creating and manipulating such objects. Functions as first-class objects are common in scripting languages and functional languages. This library allows for similar capabilities in C++, beyond the typical function pointers.

The `std::function` class template defines function objects, and are templated by the underlying function's return type and the number and type of arguments. A function object is most often created by passing the constructor a function name or function pointer. This class overloads the function call operator, the suffix `()` applied to an object, to pass arguments to and call the underlying function. An important related construct is `std::bind`, a function template which *binds* arguments to a function or function object, and returns a new function object with lower arity. Listing 3.2 shows an example of constructing and binding a function.

While not explicitly part of the Function Objects Library, *lambda expressions* were also introduced in C++11 and fulfill similar tasks. Lambda expressions are special kinds of anonymous functions which define in-line, unnamed functions and create a *closure* [160, Section 3.6]. Closures are able to *capture* variables in their enclosing scope, either by value (copy) or by reference. Lambda expressions return a function object. Listing 3.3 shows two example lambda expressions, the latter of which captures variables by reference.

3.3.3 Thread Support Library

The *Thread Support Library* provides classes and types supporting multithreaded programming and thread synchronization. The basic object is a `std::thread`, which provides an implementation-agnostic object representing an independent thread of execution. Typically, this is a POSIX thread [149]. `thread` objects are created by passing them a

```
void printInteger(int a) { std::cout << a << std::endl; }

//Function object from function name
std::function<void(int)> f_printInt(printInteger);
f_printInt(12);

//Function object binding arguments to function name
std::function<void()> f_print42( std::bind(printInteger,42) );
f_print42();
```

Listing 3.2: The use of `std::bind` to bind arguments to a function.

```
//Lambda expression with two parameters
std::function<int(int,int)> f_addInts( [](int a, int b) -> int {
    return a + b;
});
f_addInts(4, 6);

int x = 12, y = 27;
//Lambda expression capturing variables in scope by reference
std::function<void()> f_printXY( [&]() -> void {
    std::cout << "x: " << x << ", y: " << y << std::endl;
});
f_printXY();
```

Listing 3.3: Creating function objects from lambda expressions. The function `f_printXY` captures the variables x and y from its parent scope.

function or function object to execute. This creates a software thread which is immediately scheduled to execute the passed function. The software threads, through the `thread` object's interface, must be *joined* or *detached* (allowed to run to completion and then self-destruct) before the `thread` object can be destroyed.

The Thread Support Library also provides many helpful synchronization primitives; for detailed foundations see [96, Ch. 2, 3, 8]. The fundamental piece is a *mutual exclusion* (mutex) object, the `std::mutex`, which prevents multiple threads from accessing the same resource simultaneously. Hence, avoiding data races. A mutex is *owned* or *locked* by at most one thread at a time. Various types of locks are provided by the standard to achieve this. A `std::unique_lock` object is created by passing a `mutex` to its constructor. The constructor blocks until the mutex is owned. Methods are provided to unlock, and later re-lock, the mutex through `unique_lock::unlock()` and `unique_lock::lock()`. A `std::lock_guard` provides a scoped lock; its creation semantics are the same as `unique_lock`, but it automatically unlocks the mutex when the lock itself goes out of scope and is destructed.

Lastly, a `std::condition_variable` allows for explicit and synchronized communication between threads via notifications. A condition variable is a single object which is shared between two or more threads. Its basic operations are *wait* and *notify*. A mutex is first locked by a thread and then that lock passed to the condition variable via the `wait()` method. The wait method first unlocks the lock and then blocks the calling thread. The condition variable will block the thread until a different thread notifies the condition variable to unblock a waiting thread. The lock is then re-locked before the wait method returns. When more than one thread is notified, the lock and underlying mutex ensure that only one waiting thread is active at a time, again avoiding data races caused by the notification.

Chapter 4

The Design of a Polynomial Algebra Library

In the world of computer algebra software there are two main categories. The first is computer algebra systems, self-contained environments providing an interactive user-interface and usually their own programming language. Custom interpreters and languages yield powerful functionality and expressibility, however, obstacles remain. For a basic user, they must learn yet another programming language. For an advanced user, interoperability and obtaining fine control of hardware resources is challenging. AXIOM [104] is a classic example of such a system. Moreover, these problems are exacerbated by systems being proprietary and closed-source, such as *Maple* [128], *Magma* [29], and *Mathematica* [180]. The second category is computer algebra libraries, which add support for symbolic computation to an existing programming environment. Since such libraries extend existing environments, and are often free (as in free software), they can have a lower barrier to entry and better accessibility. Some examples are *NTL* [163], *FLINT* [94], and *CoCoALib* [1].

The Basic Polynomial Algebra Subprograms (BPAS) library [7] is a free and open-source computer algebra library for polynomial algebra. We first introduced this library in Section 1.2. In this chapter, we examine the design of this library to improve its ease of use and extensibility. The BPAS library looks to improve the efficiency of end-users through both usability and performance, providing high-performance code along with an interface which incorporates some of the expressibility of a custom computer algebra system. Like any computer algebra software, functionality is highly important, yet usability makes the software practical.

The implementation of BPAS is focused on performance for modern computer architectures by optimizing for data locality and through the effective use of parallelization.

These techniques have been applied to the implementations of multi-dimensional FFTs, real root isolation, dense modular polynomial arithmetic, and dense integer polynomial multiplications; see [46] and references therein. Recent works have extended BPAS to include arithmetic over large prime fields [59], sparse multivariate polynomial arithmetic [11], power series and polynomials with power series coefficients (see Chapter 7 and [32, 33]), subresultant chains (see Section 6.2 and [13]), and a polynomial system solver based on triangular decomposition (see Chapter 6 and [9])

In this chapter, however, we look to describe our efforts to make these existing high-performance implementations accessible and practical through user-interface design and improved usability. Usability includes many things: ease of use in interfaces, syntax, and semantics; mathematical correctness; accessibility and extensibility for end-users; and maintainability for developers. The interplay between performance and usability, and the difficulty to balance both, has long been a concern. In 1996, John R. Rice presented many open questions and "barriers to progress" for scientific software [152]. In particular, he questions "what should be the relationship between problem solving environment performance and ease-of-use" identifying the need for "inclusion of techniques such as object orientation and program interface specifications for developing reusable, evolutionary software". Nearly twenty-five years later, software engineering issues still plague scientific computing [44].

The BPAS library follows two driving principles in its design. The first is to encapsulate as much complexity as possible on the developer's side, where the developer's intimacy with the code allows her to bear such a burden, in order to leave the end-user's code as clean as possible. The second can be described by a common phrase in user experience design: "make it hard to do the wrong thing."

The object-oriented nature of C++, along with its automatic memory management, provides a very natural environment for a user-interface. While C++ is notoriously difficult to learn, it remains ubiquitous in industry and scientific computing making it reasonably accessible, and particularly so, if complexity can be well-encapsualted. Moreover, C++ being a compiled, statically- and strongly-typed language, further aids the end-user. The compilation process itself provides the user with checks on their code before it even runs. Meanwhile, statically-typed languages have been shown to be beneficial to usability, decrease development time, and provide high-quality, self-documenting code, compared to dynamic languages [71].

While it can be argued that dynamic languages are more flexible and easy to use, this is only the case for prototyping and short-term needs. In terms of computer algebra software, this says that general-purpose computer algebra systems are very useful for

experimentation and proofs of concept. But, for more maintainable and high-performance software, a compiled and strongly-typed environment is preferred. One can see this, for example, when comparing the lazy power series in the `PowerSeries` of *Maple*, and our newer version in C; see Section 7.2.2.

In the present chapter, we discuss our efforts to use C++ metaprogramming to aid in the usability of the library, in particular making the library adaptable and easily adoptable by practitioners. Our discussion focuses on two particular aspects relating to type safety and expressibility. First, encoding the algebraic hierarchy as an object-oriented class hierarchy is discussed in Section 4.1. Doing so while maintaining type safety is difficult; syntactically valid operations may yield mathematically invalid operations between incompatible rings. Secondly, we examine a mechanism to automatically adjust the definition of a class created from the composition of other classes. In particular, we look at polynomials adapting to different ground rings in Section 4.2. Our techniques are discussed and contrasted with existing works in Section 4.3. We conclude and present future work in Section 4.4.

We note that our techniques are not entirely new; the underlying template metaprogramming constructs have been adopted into the C++ standard since as early as C++11. Nevertheless, it remains useful to explore how these advanced concepts can be employed in the context of computer algebra. For details on C++, templates, and their capabilities, see Section 3.3 and [172].

4.1 Algebraic Hierarchy as a Class Hierarchy

In object-oriented programming (OOP) classes form a fundamental part of software design. A class defines a type and how all instances of that type should behave. Through a class hierarchy, or a tree of inheritance, classes have increasing specialization while maintaining all of the functionality of their superclasses. The benefits of a class hierarchy are numerous, including providing a common interface to which all objects should adhere, minimizing code duplication, facilitating incremental design, and of course, polymorphism. All of this provides the software with better maintainability and a more natural use of the classes themselves since they directly model their real-world counterparts.

For algebraic structures, the chain of class inclusions naturally admits an encoding as a class hierarchy. For example, the class inclusions of some rings

$$\text{field} \subset \text{Euclidean domain} \subset \text{GCD domain} \subset \text{integral domain} \subset \text{ring},$$

would allow rings as the topmost superclass with an incremental design down to fields; see Section 2.1.

Let us call such an encoding of algebraic types as a class hierarchy the *algebraic class hierarchy*. Particularly, we look to implement this hierarchy as a collection of abstract classes for the benefits of code re-use and enforcing a uniform interface across all concrete types (e.g. integers, rational numbers).

Unfortunately, an encoding of algebraic structures as classes in this way yields incorrect type safety. Through polymorphism, two objects sharing a superclass interact and behave in a uniform way, without regard to if they are mathematically compatible.

4.1.1 A Motivating Example

```
1  class EuclidDomain : GCDDomain {
2      // Assign this to the remainder of a by b.
3      virtual EuclidDomain& rem(EuclidDomain& a, EuclidDomain& b);
4  };
5
6  void EuclideanAlg (EuclidDomain& g, EuclidDomain& a, EuclidDomain& b) {
7      while (b != 0) {
8          g.rem(a, b);
9          ...
10     }
11 }
12
13 class RationalNumber : public EuclidDomain { ... };
14 class UnivarPolyModSeven : public EuclidDomain { ... };
15
16 RationalNumber a, g;
17 UnivarPolyModSeven b;
18 EuclideanAlg(g, a, b); // Rutime Error!
```

Listing 4.1: Incorrect polymorphism between two concrete algebraic types of the same interface but mathematically incompatible rings.

Consider a simple C++ abstract class for a a Euclidean domain: `EuclidDomain`. Keeping in line with a class hierarchy of algebraic types, it is a subclass of another class `GCDDomain`, which is not included for brevity. `EuclidDomain` defines the function `rem` to get the remainder of one element by another. We then have a generic implementation of the Euclidean algorithm which operates on `EuclidDomain` objects via polymorphism. Further, we have two concrete Euclidean domains, the rational

numbers as `RationalNumber` and univariate polynomials over the finite field $\mathbb{Z}/7\mathbb{Z}$, `UnivarPolyModSeven`.

Listing 4.1 shows the definition of the abstract class and a possible sequence of instructions to call this Euclidean algorithm with one `RationalNumber` object and one `UnivarPolyModSeven` object as inputs. This will produce valid compiled code through polymorphism, but will certainly lead to runtime errors as these two Euclidean domains are mathematically incompatible.

This example is certainly contrived, but highlights a key issue we wish to address. How can we encode the fact that different algebraic types adhere to the same interface and yet are mutually incompatible? The underlying issue is the `rem` function accepting any two `EuclidDomain` objects, regardless of which particular Euclidean domain they belong.

4.1.2 The Algebraic Class Hierarchy

Consider the C++ function declaration which could appear in the topmost `Ring` class: `Ring add(Ring x, Ring y)`. By polymorphism, any two Ring objects could be passed to this function to produce valid code, but, if those objects are from mathematically incompatible rings, this will certainly lead to errors. A more robust system is needed to facilitate strict type safety.

Some libraries (see Section 4.3) solve this by checking runtime values, throwing an error if incompatible. Reliance on runtime type safety, and inflating an object's size with unnecessary data, are both undesirable. Instead, our main idea is to define the interface of a ring (or a particular subclass of a ring) in such a way where a function declaration itself restricts its parameters to be from compatible rings.

In our algebraic class hierarchy, function declarations do this through the use of template parameters. Particularly, our algebraic class hierarchy is a hierarchy of class templates with the template parameter `Derived`. This template parameter identifies the concrete ring(s) with which the one being defined is compatible. In this design, all abstract classes in the hierarchy have the template parameter `Derived` while the concrete classes instantiate this template parameter of their superclass with that concrete class itself being defined. This yields the C++ idiom, the *Curiously Recurring Template Pattern* (CRTP); see [172, Ch. 16].

While CRTP has several functions, it is used here to facilitate *static polymorphism*. That is to say, it forces function resolution to occur at compile-time, instead of dynamically at runtime via virtual tables. This provides compile-time checks and error-detection

for incompatibility. For example, the previous `EuclidDomain` abstract class would become a class template `EuclidDomain<Derived>` and the `rem` function would become `Derived rem(Derived& a, Derived& b)`.[1]

This process works from a key observation when considering simultaneously templates and class inheritance: different template parameter specializations produce distinct classes and thus distinct inheritance hierarchies. Recall that template instantiation in fact causes code generation at compile-time. Thus, each concrete ring defined via CRTP exists in its own class hierarchy, and dynamic dispatch via polymorphism cannot cause runtime inconsistencies. This concept is illustrated in Listing 4.2 where the abstract classes for ring and Euclidean domain are shown, as well as the concrete class for the ring of integers. The `Integer` class uses template instantiation where it defines its superclass, specializing the `Derived` parameter of `BPASEuclideanDomain` to be `Integer`, following CRTP. The same is done for `UnivarPolyModSeven`, but this time with a different specialization of the `Derived` template parameter.

While this design provides the desired compile-time type safety, it may be viewed as too strict, since each concrete ring exists in an independent class hierarchy. For example, arithmetic between integers and rational numbers would be restricted. More generally, natural ring embeddings are neglected. However, we can make use of *implicit conversion* in C++. Where a constructor exists for type `A` taking an object of type `B` as input, an object of type `B` can be implicitly converted to an object of type `A`, and used anywhere type `A` is expected.

A `RationalNumber` constructor taking an `Integer` parameter thus allows for automatic and implicit conversion, allowing integers to be used as rational numbers. Indeed, all integers are valid rational numbers. However, the opposite conversion may also be useful in practice. Should conversion from rational numbers to integers be supported be default? To strike a balance between flexibility, ease-of-use, and type safety, we have decided that the answer is yes, but with a caveat. The caveat is to add dynamic runtime checks for correctness. Declaring an `Integer` constructor which takes a `RationalNumber` allows for flexibility via implicit conversion. However, some conversions are ill-formed, like when the denominator of a rational number is anything but 1. Therefore, we add runtime checks to ensure type safety. Indeed, a runtime solution is required since we cannot know the value of every object to be converted at compile-time. Listing 4.3 shows these conversions between `Integer` and `RationalNumber` types.

This design via implicit conversion can be seen as giving permission for compatibility between rings by defining such a constructor. Errors are discovered at compile-time where

[1] We can omit the return reference since `Derived` is now a concrete type and not an abstract type.

```
template <class Derived>
class BPASRing;

//... more abstract algebraic classes, e.g. BPASGCDDomain, BPASField

template <class Derived>
class BPASEuclideanDomain : BPASGCDDomain<Derived> {
Derived rem(Derived& a, Derived& b);
};

class Integer : BPASEuclideanDomain<Integer> {
//Through template instantiation, declares the function:
//Integer rem(Integer& a, Integer& b);
};

class UnivarPolyModSeven : BPASEuclideanDomain<UnivarPolyModSeven> {
//Through template instantiation, declares the function:
//UnivarPolyModSeven rem(UnivarPolyModSeven& a, UnivarPolyModSeven& b);
};
```

Listing 4.2: A subset of the algebraic class hierarchy, using CRTP to declare an Integer class and a UnivarPolyModSeven class.

implicit conversion fails and two rings are completely incompatible. In the case where conversion is sometimes possible (such as from rational numbers to integers), errors, if any, are discovered at runtime. This is in opposition to other libraries with strictly runtime type safety, which act in a restrictive manner. Those libraries allow everything at compile-time and then throw errors at runtime if two rings are incompatible. We discuss this further in Section 4.3.

4.1.3 Polynomials in the Algebraic Class Hierarchy

We now look to extend the abstract algebraic class hierarchy to include polynomials. For genericity and a common structured interface, we wish to parameterize polynomials by their ground ring. This can be accomplished with a secondary template parameter in addition to the `Derived` parameter already included by virtue of polynomials existing in the algebraic class hierarchy (see Listing 4.4).

However, this is not fully sufficient, and two issues arise.

1. First, while polynomials do form a ring, they often form more specialized algebraic structures, e.g. a GCD domain.

2. Secondly, there is no restriction on the types which can be used as template parameter specializations of the ground ring.

```
class Integer : BPASEuclideanDomain<Integer> {
    //Integer constructor taking RationalNumber for implicit conversion
    Integer (RationalNumber& r) {
        if (r.denominator() == 1) {
            *this = r.numerator();
        } else {
            throw invalid_argument("Rational number is not an integer");
        }
    }
};

class RationalNumber : BPASField<RationalNumber> {
    //RationalNumber constructor taking Integer for implicit conversion
    RationalNumber (Integer& i) {
        numerator = i;
        denominator = 1;
    }
}
```

Listing 4.3: Implicit conversion between algebraic types is explicitly allowed by providing a constructor whose parameter is the type being converted from.

We leave the discussion of the first issue to Section 4.2. For the second issue, we want to ensure that a polynomial's coefficient ring is indeed a ring and not any other nonsense type. Recall, our design goal is to make it hard to do the wrong thing.

Leveraging another template trick along with multiple inheritance, this can be solved with the so-called `Derived_from` class[2] which determines at compile-time if one class is the subclass of another. `Derived_from` is a template class with two parameters: one a potential subclass, and the other a superclass. This class defines a function converting the apparent subclass type to the superclass. If the conversion is valid via implicit up-casting, then the function is well-formed, otherwise, a compiler error occurs.

Generically, one makes use of `Derived_from` by defining a class template which inherits from it. The new class passes its template parameter to `Derived_from` as the potential subclass. The superclass to enforce will be statically defined in the inheritance declaration. This enforces that a template parameter be a subclass of (or the same class as) the statically-defined superclass. In our implementation, shown in Listing 4.4, polynomial classes are a class template with the coefficient ground ring as template parameter. They inherit from `Derived_from` to enforce that their ground ring is a some subclass of `BPASRing`, our abstract class for rings (recall the declaration of `BPASRing` from Listing 4.2).

[2] `Derived_from` is a long-known trick, but is now adopted into the C++20 standard.

```
// If T is not Base, nor a subclass of Base, a compiler error occurs
template <class T, class Base> class Derived_from {
    static void constraints(T* p) { Base* pb = p; }
    Derived_from() { void(*p)(T*) = constraints; }
};

//abstract polynomial class template
template <class Ring, class Derived>
class BPASPoly : BPASRing<Derived>, Derived_from<Ring, BPASRing<Ring>>;

//concrete polynomial type with parameterized coefficient ring
template <class Ring>
class SparseUnivariatePoly : BPASPoly<Ring, SparseUnivariatePoly<Ring>>
```

Listing 4.4: An implementation of a abstract polynomial class and concrete polynomial class using CRTP and `Derived_from`.

Listing 4.4 shows some interesting features and, on first sight, confusing semantics. The combination of CRTP with `Derived_from` and with a second template parameter requires some parsing. Let us begin with the abstract class template `BPASPoly`. Following the design of our algebraic class hierarchy, it has a `Derived` template parameter in order to implement CRTP. This is used in the inheritance from `BPASRing<Derived>`; indeed, polynomials themselves form a ring (and more specific types, to be discussed later in Section 4.2). Second, the coefficient ring is given as the `Ring` template parameter. We enforce that type to be an actual ring type using `Derived_from`. Yet, because of CTRP within our algebraic class hierarchy, the "static" superclass which we want to enforce, is itself a class template. Therefore, we apply a second instance of CRTP within `Derived_from` to enforce the `Ring` parameter to be (a subclass of) `BPASRing<Ring>`.

Listing 4.4 also shows how a concrete polynomial class template may be defined. Here, we declare a univariate polynomial class with a sparse representation. It has a `Ring` template parameter for the coefficient ring. Notice that it is sufficient for this concrete class to inherit from only `BPASPoly`. Indeed, by the declaration of `BPASPoly`, the `Ring` parameter of `SparseUnivariatePoly` will automatically be passed to `Derived_from`. Moreover, this concrete class fulfills CRTP by passing itself to the the `Derived` parameter of `BPASPoly`.

The use of CRTP, `Derived_from`, and implicit conversion, all work together to create an algebraic hierarchy as a class hierarchy which maintains strict compile-time type safety. Yet, our scheme remains flexible where obvious conversions, such as natural ring embeddings, are allowed by implicit conversion via explicit constructor definition. Meanwhile, this class hierarchy remains flexible enough to allow for polynomial classes

to exist over user-defined coefficient rings, so long as those coefficient rings inherit from `BPASRing`. What remains now is to address the issue of polynomial rings sometimes forming different algebraic types depending on their particular ground ring.

4.2 "Dynamic" Type Creation, Conditional Export

In object-oriented design, the combination of types to create another type is known as composition. In this section, let us consider univariate polynomial rings; one can always work recursively for multivariate polynomials. Viewing a polynomial ring as a ring extension of its ground ring, polynomials can be seen as the composition of some finite number of elements of that ground ring. Moreover, we know that the properties of a polynomial ring depend on the properties of the ground ring. For example, the ring of univariate polynomials over a field is a Euclidean domain while the ring of polynomials over a ring is itself only a ring. Recall from the previous section that our implementation of polynomials are templated by their ground ring. Our goal then is to capture the idea that the position of a polynomial ring in the algebraic class hierarchy changes depending on the particular specialization of this template parameter.

More generally, we would like that the type resulting from the composition of another type depends on the type being composed. Hence, a sort-of "dynamic" type creation. This is not truly dynamic, since it will be a compile-time operation, but it nonetheless feels dynamic since it is an automatic process by the compiler via template instantiation. In fact, having this occur at compile-time is actually a benefit where errors can be determined preemptively, much like the type safety aspect described in the previous section. One can also view this mechanism as a way of controlling the methods which the newly created type exports. That is, conditionally exposing methods (or other attributes) in its interface depending on the particular template parameter specialization. This technique relies on compile-time introspection and SFINAE.

4.2.1 SFINAE and Compile-Time Introspection

Recall from Section 3.3.1 that the Substitution Failure Is Not An Error (SFINAE) principle is fundamental to template metaprogramming and compile-time introspection. SFINAE says that the invalid substitution of a type as a template parameter is itself not an error; see Listing 3.1. Where two or more template specializations exist, it is not required that the substitution of the template parameter fit all of the specializations, but only one. This principle, combined with compile-time function overload resolution,

yields *compile-time introspection*. Using templates, truth values about a type can be determined and then made use of within the program, at compile-time.

For example, `is_base_of`, a standard feature in C++11, is much like `Derived_from` which was defined in the previous section. However, instead of creating a compiler error, `is_base_of` determines a Boolean value representing if one type is derived from another (or is the same type).

Using introspection, one may think that `enable_if`, another standard C++11 template construct, is sufficient. The `enable_if` struct template conditionally compiles and exposes a function template based on the value of a Boolean known at compile-time. This Boolean value can of course be determined by introspection. Unfortunately, function templates cannot be virtual. This potential solution, therefore, cannot be used within a class hierarchy. Conditionally exposing methods in our algebraic class hierarchy requires a slightly more complex solution.

4.2.2 Conditional Inheritance for Polynomials

Defining new types depending on another type, as well as conditionally exposing member functions, can both be fulfilled by *conditional inheritance*. Specifically, we implement a compile-time case discussion for inheritance based on introspective values. In the context of polynomials in our algebraic class hierarchy, this case discussion works as a cascade of type checks on the ground ring, say R, when forming the polynomial ring $R[x]$. For example: if R is a field, then $R[x]$ is a Euclidean domain; else if R is a GCD domain, so is $R[x]$; else if R is an integral domain, so is $R[x]$; else $R[x]$ is a ring. This case discussion can be extended to include as much granularity as needed.

To perform this case discussion, we use the class template `conditional<B, T, F>` which is part of the C++11 *Type Support Library*. `conditional<B, T, F>` has three template parameters which act as a compile-time ternary conditional operator. It uses a compile-time Boolean value to choose between two template parameters, choosing `T` if true, and `F` is not. In our case, we use `is_base_of` to determine the Boolean value while `conditional` chooses the superclass for conditional inheritance.

As a simple example, consider Listing 4.5. The definition of `BPASPoly` tests if the `Ring` template parameter is a subclass of `BPASField`. If so, `conditional` chooses `BPASEuclideanDomain` as the the superclass of `BPASPoly`. Otherwise, `BPASRing` is chosen. Additionally, the concrete class `SparseUnivariatePoly` is shown again, still parameterized by its coefficient ring. Notice that conditional inheritance will apply to it once `Ring` is instantiated to a particular type at compile-time and the `conditional` in

its superclass gets evaluated. Finally, the concrete class `RationalNumberPoly` is also declared, with its coefficient ring statically defined to be the `RationalNumber` type. This time, conditional inheritance is applied immediately since the template specialization is already defined.

```
1  template <class Ring, class Derived>
2  class BPASPoly : conditional< is_base_of<Ring, BPASField<Ring>>::value,
3                                BPASEuclideanDomain<Derived>,
4                                BPASRing<Derived> >::type,
5                   Derived_from< Ring, BPASRing<Ring> >;
6
7  template <class Ring>
8  class SparseUnivariatePoly : BPASPoly<Ring,SparseUnivariatePoly<Ring>>;
9
10 class RationalNumberPoly : BPASPoly<RationalNumber, RationalNumberPoly>;
```

Listing 4.5: A simple use of `conditional` to choose between Euclidean domain or ring as the algebraic type of a polynomial based on its template parameter.

The presented code for `BPASPoly` in Listing 4.5 is rather simple in that it only checks if the coefficient ring is a field of not. To implement a chain of type checks, the "else" branch of a `conditional` should be yet another `conditional`. To improve the readability of this case discussion, we avoid directly implementing nested if-else chains, and thus avoid using one `conditional` inside another. Instead, we create two symmetric class hierarchies, one representing the true algebraic class inclusions while the other is a "tester" hierarchy.

This tester hierarchy uses one `conditional` to determine if a property holds and, if so, chooses the corresponding class from the algebraic hierarchy as superclass. Otherwise, the next tester in the hierarchy is chosen as superclass to trigger the evaluation of the next `conditional`. Finally, all concrete polynomial classes inherit from `BPASPolynomial` to automatically determine their correct interface based on their ground ring. This structure is shown in Figure 4.1, with the algebraic hierarchy on the left, and the tester hierarchy on the right.

This technique of conditional inheritance is a powerful tool in any class template hierarchy. By understanding the properties of a type via introspection, it can automatically and dynamically change its position in the class hierarchy, for example, based on the specialization of a template parameter. Not only does this enforce a proper class interface, but it allows the possibility of choosing between several different abstract implementations in order to best support the new type (i.e. the result of a composition).

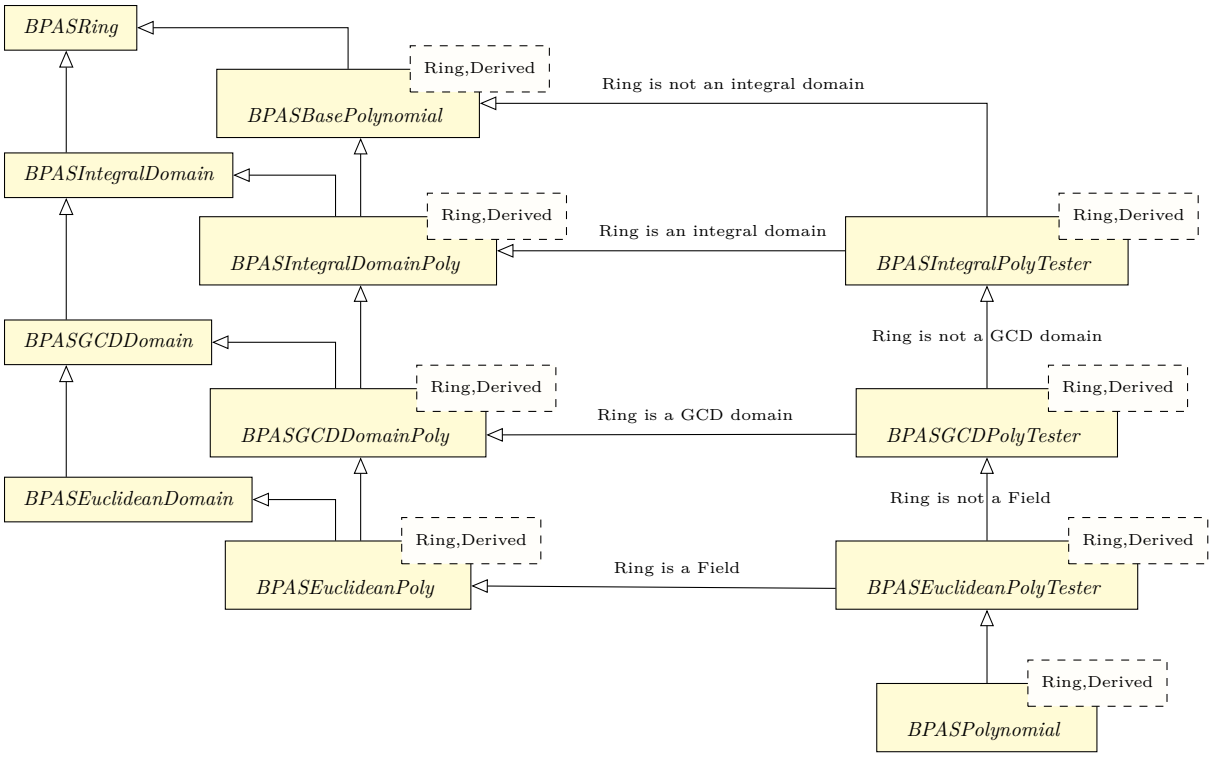

Figure 4.1: UML diagram for a subset of the polynomial abstract class hierarchy. Recall in UML that template parameters are shown in dashed boxes. Template parameters for non-polynomial classes are omitted for clarity. Note also that the multiple inheritance diamond problem is easily solved using virtual inheritance.

4.3 Discussion and Related Work

For decades, computer algebra systems have worked towards type safety. Axiom [104] is a pioneering work on that front, but has grown out of popularity. Functional languages, like Scala and Haskell, have seen some progress in developing computer algebra systems thanks to type classes (see, e.g., [105] and references therein). These languages and their type classes provide a very suitable environment to define algebraic structures. However, while powerful, functional languages can be seen as an obscure and inaccessible programming paradigm compared to the mainstream imperative paradigm.

Considering other C/C++ computer algebra libraries, there are many examples with interesting mechanisms for handling algebraic structures. The *Singular* library [66] perhaps has the most simple mechanism: a single class represents all rings, using a number of enum and Boolean variables to determine properties of instances at runtime. In *CoCoALib* [1] an abstract base class `RingBase` declares many functions returning Boolean values, for example, `IamField`. Concrete subclasses define and override these functions so that, when called at runtime, properties of the class can be determined. While rings

are subclasses of `RingBase`, elements of a ring are an entirely different class. Elements have pointers to the ring they belong, which are then compared at runtime to ensure compatibility in arithmetic between two elements. *LinBox* [168] also has separate classes for rings and their elements. There, ring properties are encoded as class templates where concrete rings use explicit template specialization to define properties.

Much like the previous cases, the *Mathemagix* system requires instances (i.e. elements) of a ring to have a specific reference to a separate entity encoding the ring itself. Notably, *Mathemagix* also includes a scheme to import and export C++ code to and from the *Mathemagix* language [99]. This uses templates to allow, for example, a ring specified in the *Mathemagix* language to be used as the coefficient ring for polynomials defined in C++.

In all of these cases there is some limiting factor. Most often, mathematical type safety is only a runtime property maintained by checking values. In some cases, this is implemented by separating rings themselves from elements of a ring, a process counterintuitive to object-oriented design where one class should define the behaviour of all instances of that type.

On the contrary, our scheme does not rely on runtime checks. Instead, a function declaration itself restricts its arguments to be mathematically compatible at compile-time via the use of template parameters and the Curiously Recurring Template Pattern. By using an abstract class hierarchy, many such function declarations are combined through consecutive inheritances to build up an interface incrementally. This closely follows the chain of class inclusions for algebraic types, where each type adds properties to the previous. The symmetry between the algebraic hierarchy and our class hierarchy hopes to make our interfaces natural and approachable to an end-user. This symmetry comes at the price of creating a deep class hierarchy, and thus strong coupling within the class hierarchy. Yet, this price is worth the symmetry and comprehensibility of the class hierarchy with the algebraic hierarchy.

In contrast with our class hierarchy solution to type safety, there is a different possible compile-time solution. Namely, the use of type traits (see, e.g., [172, Ch. 15, 17]). Type traits are template metaprogramming constructs for type introspection and modification, some of which have already been seen, such as `is_base_of`, and `conditional`. Type traits are arguably more flexible, but require an even deeper understanding of templates and template metaprogramming than what is used in our implementation. In particular, an end-user defining a new type would have to implement their own type traits. In contrast, our class hierarchy solution hides the template metaprogramming in the declaration of the superclasses. Moreover, type traits are essentially unique to

C++. Class hierarchies, on the other hand, are present in every object-oriented language and should therefore be more accessible to end-users who may know C++, but are not necessarily experts. The use of class hierarchies, in addition to encapsulating much of the template metaprogramming in our design, should allow for easy extensibility by end-users.

4.4 Conclusion and Future Work

In this chapter we have explored the design of the algebraic and polynomial hierarchy of the BPAS library. Through the use of template metaprogramming, we have devised a so-called algebraic class hierarchy which directly models the algebraic hierarchy while providing compile-time type safety. This hierarchy is type-safe both in the programming language sense and the mathematical sense.

Using inheritance throughout the algebraic abstract class hierarchy, the interface of algebraic types is constructed incrementally. Therefore, a concrete type's properties and interface is determined by its particular abstract superclass from this hierarchy. Through additional templating techniques (e.g. `is_base_of`) we can automatically infer, at compile-time, the correct superclass (and thus interface) of new types created by template parameter specialization (e.g. polynomials). The result is a consistent and enforced interface for all classes modelling algebraic types.

We are currently working to employ our abstract class hierarchy into our implementation of triangular decomposition; see Chapter 8. Further, we hope to extend the accessibility of BPAS and polynomial algebra routines with a Python interface to the BPAS library (i.e. an extension module). This will allow for a dynamic scripting environment in which BPAS may be used for rapid prototyping. We discuss additional considerations for accessibility and ease of use in Chapter 9.

Chapter 5

Object-Oriented Parallel Support

Towards our goal of implementing high-performance polynomial algebra routines, a natural direction is parallelization. As discussed in Chapter 1, we are motivated by the idea of component-level parallelism in triangular decomposition. Our goal is to solve polynomials systems incrementally, computing and refining solutions on each geometric component independently. Exploiting this kind of high-level parallelism in geometric algorithms in challenging since the independent components on which to operate are determined by the particular problem being solved, and must therefore be found dynamically at runtime.

The paradigm of *dynamic multithreading* offers a solution. In this paradigm, programmers specify where opportunities exist for concurrency, meanwhile a runtime scheduler dynamically decides which regions to actually execute in parallel. In this way, when concurrency is discovered, and hardware resources are available to exploit it, the dynamic scheduler can do so.

Concurrency platforms, like *Cilk* [124], follow this idea. However, *Cilk* is based on fork-join parallelism. In our implementation of triangular decomposition, we were motivated to also use pipeline parallelism; see Section 6.3. Pipeline parallelism was challenging within the semantics of *Cilk*, and so we implemented our own. However, in our early implementations, we found that mixing user-based parallelism with *Cilk* parallelism did not work for our purposes. Hence, we have designed and implemented our own dynamic multithreading platform, which is the topic of this chapter.

Following the general layered and object-oriented design of BPAS (see Chapter 4), we have designed this multithreading library to be object-oriented. This is in contrast with platforms like *Cilk* or *OpenMP* [62], which add new keywords to the C/C++ language and require additional compiler support. With the help of the *Function Objects Library* of C++11 (see Section 3.3), manipulating functions and code regions as objects is possible. By extension, one can implement an object-oriented concurrency platform. This platform

is implemented as a sub-library of the BPAS library. The objectives of our platform are threefold:

(*i*) to supply support for parallelism which is generic, reusable, object-oriented, and encapsulates much of the difficulty of parallel programming;

(*ii*) to support the usage of *parallel patterns* in practice; and

(*iii*) to provide mechanisms for cooperation between parallel regions.

This last objective becomes important under the consideration of irregular parallelism, dynamic load-balancing, and finite hardware resources (i.e. cores). When a new concurrency opportunity is discovered, how should existing parallel regions be affected? If all resources are occupied, should the new concurrency opportunity simply execute in serial? Should the new concurrency opportunity be queued and executed once resources become available? Should other parallel regions be paused and the new opportunity allowed to execute in parallel? Such questions, in general, are very difficulty to answer and can be very problem-specific. We discuss potential solutions as directions for future work in Chapter 9. A simplified solution to cooperation is presented in Section 5.4. First, however, we discuss our solutions to the previous two objectives.

The remainder of this chapter is organized as follows. Recall that some fundamental concepts from thread-level parallelism are discussed in Section 3.2. We begin in Section 5.1 by reviewing some particular parallel patterns which motivate the utilities provided by our object-oriented concurrency platform. The implementation of our concurrency platform is based on two key classes: an "asynchronous object stream" and a thread pool. We describe their design and implementation in Section 5.2. Next, the use of these tools to realize object-oriented parallel patterns is described in Section 5.3. Finally, Section 5.4 discusses the mechanisms provided for cooperation between parallel regions.

5.1 Parallel Patterns

Parallel patterns, or algorithmic skeletons, are meta-algorithms used to organize code for efficient parallel execution. They look to organize code to avoid the issues of parallel overheads, particularly inter-thread communication and synchronization. Like any multi-threaded program, one wishes to limit inter-thread dependencies and achieve load-balance between threads to maximize parallel speed-up. In the following subsections we examine different parallel patterns which we will later implement generically in our

object-oriented parallel support. We dicuss five particular patterns: *map* (Section 5.1.1); *workpile* (Section 5.1.2); *asynchronous generators* and *producer-consumer* (Section 5.1.3); *pipeline* (Section 5.1.4) and *fork-join* (Section 5.1.5). For further discussion on parallel patterns and their implementation, see [131].

5.1.1 Map

One of the fundamental patterns in parallel programming is *map* [131, Ch. 4]. Simply stated, the map pattern applies a function to each item in a collection, simultaneously executing the function on each independent data item. Often, the application of a map produces a new collection with the same shape as the input collection. Alternatively, the map may modify each data element in-place. When the number of data elements is greater than the number of available threads, the data elements are evenly partitioned, and one thread is responsible for executing the function on each item in one partition.

A **for** loop with independent iterations is a prime candidate for parallelization via the map pattern. Each thread simply executes one or more iterations of the loop. Due to this ubiquity, the map pattern is often implicit, and such parallel loops merely labeled **parallel_for**. In this way, the number of threads and the division of work evenly across a certain number of threads is left implicit. This is not only beneficial for clearly defining the algorithm in pseudo-code, but also leaves the exact number of threads to be used as a parameter to be decided at runtime. Algorithm 5.1 shows the map pattern in generic pseudo-code, mapping the function F over an array of data items A using the **parallel_for** keyword. A visual representation of the map pattern is shown in Figure 5.1.

Algorithm 5.1 MAPEXAMPLE(A, n, F)

Input: an array A of size n, and a function F
Output: an array B of size n where $B[i] = F(A[i])$

```
1: parallel_for i from 0 to n − 1 do
2: |   B[i] := F(A[i])
3: return B
```

Using the map pattern, threads operate in lockstep; if several maps are executed in a row, all operations in a previous map step must finish before the next map step may begin. This may be seen as multiple **parallel_for** loops in a row. Notice that the overall performance of a map step is limited by the slowest operation in the group. Hence, the map pattern works well with regular parallelism where individual data items or tasks are similar in size or execution cost. Where tasks have various or unknown sizes

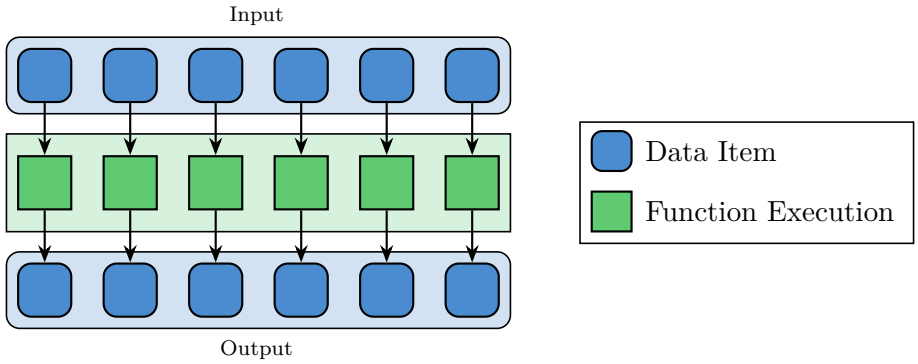

Figure 5.1: The map pattern concurrently executes a function over each item in a collection, producing a collection of outputs.

and, thus, irregular parallelism, a more flexible pattern is needed to allow for possible load-balancing.

5.1.2 Workpile

The *workpile* pattern generalizes map to handle both irregular tasks and an unknown number of tasks [131, Section 4.6]. In this pattern, the data elements on which to operate are collected into a queue or *pile*. One thread then takes one data element from the queue and executes the desired function on that data element. Once a thread is finished with a data element, it retrieves another from the pile and again executes the desired function, repeating until the pile is empty. See a visualization of this pattern in Figure 5.2.

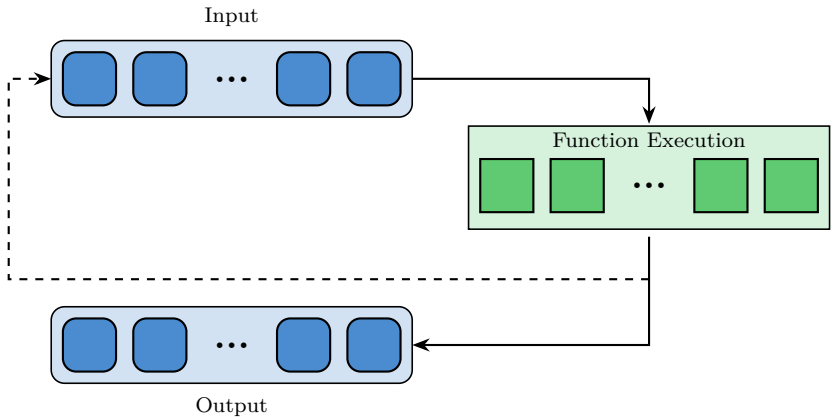

Figure 5.2: The workpile pattern concurrently executes a function over items in a queue, producing a queue of outputs, and repeating until the queue is empty. Optionally, the workpile pattern may add new items to the input queue throughout the execution. See the legend in Figure 5.1.

Load-balancing is easier to achieve with workpile, particularly in cases where the work associated with each data item is irregular. In the map pattern, the data elements in the collection are usually divided evenly across the available threads, and thus performance is limited by the slowest group. In workpile, rather, one thread is responsible for only a single data element at a time. If one data element requires more work than other elements, then other threads becoming idle sooner will simply, and dynamically, take additional data items from the pile. This is similar to the idea of a work-stealing scheduler, such as is used in *Cilk* and Intel's *Thread Building Blocks* [131].

The workpile pattern may be described in pseudo-code with a **while** loop whose iterations are executed in parallel and whose condition is that the queue of data items is not empty. We denote such a loop as **parallel_while**; see Algorithm 5.2.

Algorithm 5.2 WORKPILEEXAMPLE(A, F)

Input: a queue of data items A and a function to execute F
Output: a queue of output data items B resulting from applying F to each item of A

```
1: parallel_while A is not empty
2: |  a := pop(A)
3: |  b := F(a)
4: |  push(B, b)
5: return B
```

Workpile has the added benefit that tasks in-flight are able to add additional data items to the pile. Consider replacing Line 4 of Algorithm 5.2 with some **if** condition that decides to push the new data item b to the queue of tasks A or the queue of results B. (see, e.g., Algorithm 6.19 in Section 6.3.1). Such a newly created item is picked up immediately by an idle thread, if one exists, and begins executing. More sophisticated workpile implementations may consider ordering the tasks in the pile based on a problem-specific criterion so that tasks executed earlier in the computation are more likely to create new tasks and thus exploit further parallelism. It is also possible to adapt the workpile pattern to the situation where each function execution on a single data item produces multiple data items, thus replacing Line 4 of Algorithm 5.2 with a loop of push operations.[1]

[1]The same modification could also be applied to the map pattern, although not as easily.

5.1.3 Asynchronous Generators and Producer-Consumer

A *generator* function is a function yielding data elements one at a time rather than many together as a collection. A generator is also known as an *iterator*, a special kind of co-routine; see, e.g., [160, Ch. 8]. Concurrency arises with generators when the generation of data items themselves is a costly operation and, moreover, can be performed concurrently to the processing of previously yielded data items. This establishes a so-called *asynchronous generator*. Concurrent execution of a generator and its calling function is one instance of the classic *producer-consumer problem*; see [21, Ch. 6].

The producer-consumer problem/pattern describes two functions connected by a queue. The producer creates data items, pushing them to the queue, meanwhile the consumer processes data items, pulling them from the queue; see Figure 5.3. Producer and consumer may operate in parallel, with the queue providing inter-thread communication. This shared intermediary queue is one possible implementation of an asynchronous generator, where the generator is the producer and caller of the generator is the consumer.

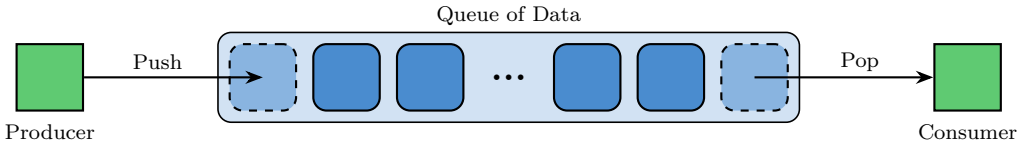

Figure 5.3: A producer-consumer pair uses a queue of data to pass data items between them. The producer pushes data to the queue and the consumer pops data from the queue. See the legend in Figure 5.1.

Iterators and generators are often described using the keyword **yield**. This keyword is used to pass a value from the generator back to its caller, but without the generator function returning completely. In serial, this entails continuing the function's control flow from the yield point when the generator function is next called. Generally, this may take the form of a *continuation*; see [160, Ch. 6]. In the case of an asynchronous generator, a yield is a push to that intermediary queue of the producer-consumer pair, and then continuing its execution concurrently. As an example, consider the simple producer-consumer (generator and caller) pair described in Algorithm 5.3.

5.1.4 Pipeline

The *pipeline* pattern—not to be confused with pipelining, a concept from instruction-level parallelism, see [95, Ch. 3]—is a sequence of stages, or function executions, where data

Algorithm 5.3 GENERATOREXAMPLE

Output: generates the sequence of Fibonacci numbers, 0, 1, 1, 2, 3, 5, ..., one at a time

```
1: function FibGenerator( ) :
2:     F_{n-1} := 0
3:     F_n := 1
4:     while true do
5:         yield F_{n-1}
6:         F_n := F_n + F_{n-1}
7:         F_{n-1} := F_n - F_{n-1}
```

Input: the number n of Fibonacci numbers to print

```
8: function FibConsumer(n) :
9:     for i from 1 to n do
10:        print( FibGenerator() )
```

flows from one stage to the next. This pattern can be seen as a sequence of producer-consumer pairs, with intermediary stages acting as both a producer and a consumer; see Figure 5.4. The individual stages of a pipeline all execute in parallel. Thus, the parallelism to be exploited is directly and positively proportional to the number of data items to process through the pipeline and the number of stages in the pipeline. Moreover, the pipeline pattern allows for earlier data items to immediately flow from one stage to the next without waiting for later items to become available. This is similar to the idea of asynchronous generators.

Note that a pipeline need not be a linear sequence of stages. More generally, each consumer may be the consumer of several different producers. Thus, the pipeline pattern can be generalized to a tree of producer-consumer pairs.

Often, the organization of producer-consumer pairs into a pipeline is statically defined by the organization of the code. However, by modeling each producer-consumer pair as an asynchronous generator—where the asynchronicity is hidden behind a function call to the generator—a pipeline can be created dynamically and implicitly via the function call stack. That call stack may actually be a tree of function calls where any function may call several asynchronous generators. That is, the pipeline may form a direct acyclic graph (DAG). To be discussed further in Section 5.2, the dynamic realization of a pipeline using asynchronous generators allows for dynamic scheduling and load-balancing when coupled with an underlying thread pool.

In terms of the latency of processing a single data item, a pipeline does not improve upon its serial counterpart. Rather, a parallel pipeline improves throughput, the amount of data that can be processed in a given amount of time. Throughput is limited by the

slowest stage of a pipeline, and thus special care must be given to ensure each stage of the pipeline runs in nearly equal time. In a naive implementation of the pipeline, all stages may be synchronized by some central scheduler which triggers all stages to pass data from one to the next. If, rather, the pipeline stages are implemented as producer-consumer pairs with intermediary queues, then each stage can really operate independently. But, extra care is needed to ensure that no one stage occupies all hardware resources, over-produces, and "clogs" the pipeline.

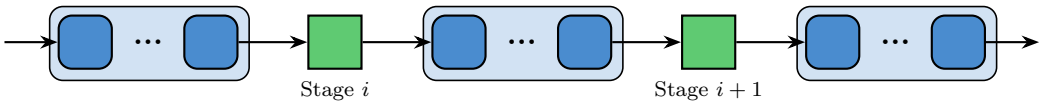

Figure 5.4: The pipeline pattern concurrently executes its individual stages to gain parallelism and increased throughout. Pictured here are two stages of a pipeline, stage i and stage $i+1$, connected by data queues as in producer-consumer pairs. Each stage is both a producer and a consumer. See the legend in Figure 5.1.

5.1.5 Fork-Join

In the *fork-join* pattern, a separate control flow is *forked* or *spawned* and executed concurrently to the control flow of the spawning thread. This can be seen as the direct realization of the fork-join model (see Section 3.2.2). Most often, a function call is forked and then the caller is allowed to continue its execution without waiting for the forked function to return. These separate control flows are then *joined* before continuing serially. This join step is a synchronization point where the forking code must wait for the forked function to finish executing and to allow safe access to the output of that function.

The ubiquitous divide-and-conquer algorithmic technique, when parallelized, is the most simple and obvious example of the fork-join pattern. In divide-and-conquer, a problem is divided into sub-problems, each solved (conquered) recursively, and then sub-solutions combined to provide a solution to the original problem. The recursion continues until a trivial base case is reached which can be solved directly. When a divide-and-conquer algorithm has more than one recursive call, it can easily be parallelized using the fork-join pattern by *forking* one or more recursive calls, and executing them all concurrently; see Figure 5.5. This idea is also exemplified in Algorithm 5.4 where a divide-and-conquer approach to mergesort is parallelized using the fork-join pattern.

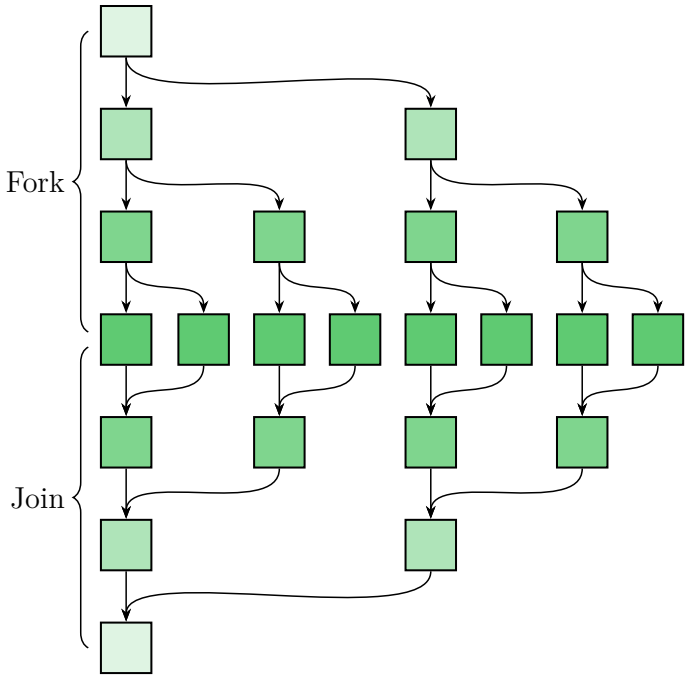

Figure 5.5: Fork-join parallelism applied to divide-and-conquer. Where there are multiple recursive calls, each executed in parallel, with the join acting as a synchronization point between the recursive calls.

Algorithm 5.4 FORKJOINEXAMPLE

Input: an array of items A, an inclusive lower bound index i, and an exclusive upper bound index j

Output: the elements of A in the index range $[i, j)$ are modified to be in sorted order

```
1: function MergeSort(A, i, j) :
2:     if j − i ≤ 1 then
3:         return
4:     k := i + ⌊(j − i)/2⌋                          ▷ the mid-point
5:     spawn MergeSort(A, i, k)
6:     MergeSort(A, k, j)
7:     join
8:     Merge(A, i, k, j)     ▷ merge the two sorted partitions [i, k) and [k, j) in-place
```

One should note that special precautions must be taken when parallelizing a divide-and-conquer algorithm. There are (at least) two factors to consider. First, one must ensure that the work to be executed by each spawn is not too small. A separate *parallel base case* is often needed where, after reaching a small enough problem size, further recursive calls are executed serially until the true base case it reached. Second, one must ensure that the number of spawns does not outnumber the hardware threads; that is to say, over-subscription must be avoided. Much like dynamic multithreading, a **spawn** in

pseudo-code is usually only a suggestion to spawn a thread, and proceeds serially if there are no more available hardware resources (or a thread pool is empty).

5.2 Implementation

To implement object-oriented parallel support, and eventually the ability to implement parallel patterns, we begin with 3 foundational objects:

(*i*) a `AsyncObjectStream`, an asynchronous queue or stream of data objects;

(*ii*) a `FunctionExecutorThead`, a long-running thread which can dynamically receive and execute new code regions; and

(*iii*) a thread pool, which we call an `ExecutorThreadPool`.

The implementation of a thread pool is a natural choice for a concurrency platform. It avoids oversubscription and the overheads of spawning many threads. However, there are some difficulties. Consider the typical thread of C++11's *Thread Support Library*. A `thread` object is constructed by being passed a function or function object. The thread is immediately scheduled to execute that function and, once the function returns, the thread is terminated. Therefore, one must overcome two obstacles: creating a long-running thread, and a thread which is able to accept new code regions to execute one after the other. This is the design and implementation of our `FunctionExecutorThread`, which we discuss in Section 5.2.2. To keep the thread alive, and able to execute many code regions, we first define the asynchronous object stream `AsyncObjectStream` in Section 5.2.1. These object streams will also serve as the basis of our asynchronous generators described in Section 5.3.3

Our entire implementation relies only on the standard library of C++11. In particular, we employ fundamental threading primitives like `std::mutex`, `std::unique_lock`, and `std::condition_variable`. See Section 3.3 for a review of C++11.

5.2.1 Asynchronous Object Streams

The asynchronous object stream facilitates the transfer of objects, asynchronously, from one thread to another. It is one possible implementation of the data queue connecting a producer-consumer pair. But, as we will see, it is capable of more than just that.

The asynchronous object stream fulfills three key goals:

(*i*) a queue to store intermediary data;

(*ii*) a mechanism to *block* the consumer until data is available to be consumed;

(*iii*) an ability for the producer to close the stream once all data has been produced.

Our class template `AsyncObjectStream` provides an object-oriented solution to these goals while encapsulating all of the multithreading requirements; see Listing 5.1. The usage of the class is simple. A single `AsyncObjectStream` object is shared between the producer and consumer, with some methods of the object used by the producer to push data to the stream, and other methods of the object used by the consumer to pull data from the stream. For genericity, the `AsyncObjectStream` class is templated by the type of object to be passed between threads. Internally, the `AsyncObjectStream` is really just a queue of data with the necessary `mutex` and `condition_variable` primitives to enable synchronization, blocking, and the eventual wake-up of consumer, once data becomes available.

The producer has two options: (*i*) to push a newly created piece of data to the stream, or (*ii*) to "close" the producer's end of the stream to declare that it has finished producing all possible data. These operations are given by the methods `addResult()` and `resultsFinished()`. The consumer has two options. The method `streamEmpty()` is a non-blocking method to query if the stream currently has any data available to be consumed. The method `getNextObject()` is a blocking method which has three possible behaviours:

1. if data is available, obtains an object from the stream, and returns true;
2. if data is not available and the producer's end of the stream is still open, blocks until data becomes available, obtains the new object, and returns true; and
3. if data is not available and the producer's end of the stream is closed (by calling `resultsFinished()`), then no object is obtained and the method returns false.

Synchronization is needed in the implementation of `AsyncObjectStream` to avoid data races where producer and consumer are simultaneously trying to push and pop data from the stream. A `std::mutex` solves this easily. Any method call to `AsyncObjectStream` begins by locking the `mutex`, then modifying (or querying) the data queue, and then unlocking the `mutex` before returning. Slightly more challenging is the mechanism to block the consumer when no data is available, and waken/notify them once it has become available. This is provided by a `condition_variable`. Recall the general scheme for using a condition variable: (*i*) lock a mutex; (*ii*) have the condition variable "wait" on the lock, temporarily unlocking it and blocking the caller; (*iii*) another thread notifies

```
template <class Object>
class AsyncObjectStream {

    std::queue<Object> retObjs;
    std::mutex m_mutex;
    std::condition_variable m_cv;
    bool finished; //is the stream still open?

    // Producer: add an object to the queue
    void addResult(Object&& res);

    // Producer: close the producer end of stream,
    //           signalling that no more objects will be produced
    void resultsFinished();

    // Consumer, non-blocking: determine if queue is currently empty
    void streamEmpty();

    // Consumer, blocking: pop an object from the queue and return true if
    // successful, return false if no data available and stream closed
    bool getNextObject(Object& res);
};
```

Listing 5.1: The `AsyncObjectStream` class interface.

the condition variable; and (*iv*) the waiting thread re-locks the mutex before returning from the wait method call.

Notice that the `getNextObject()` method takes an `Object` reference, meanwhile the `addResult()` methods takes an rvalue `Object` reference. This suggests one small, but useful, runtime optimization. Since the `AsyncObjectStream` is implemented with multithreading in mind, the data to be streamed exists in a shared-memory system. That is to say, the data itself does not need to be transferred between threads, but rather just the object's ownership. One could facilitate this by passing pointers between threads. However, this can be tedious and is not exactly object-oriented. Rather, we use C++ *move semantics* (see [133, Ch. 5]) to transfer the underlying data from the object passed to `addResult()`, to the data queue, and then moved again into into the object passed to `getNextObject()`. The use of mutex, condition variable, and move semantics can be seen in the implementation of `getNextObject()` shown in Listing 5.2.

Notice that this implementation of `AsyncObjectStream` is generic enough to support many consumers. Although many consumers may be waiting (and "fighting") for data in `getNextObject()`, the `mutex` ensures only one at a time is able to access the data queue. A simple modification could also be made to support multiple producers. Rather than the `finished` variable being a Boolean, it could be a counter, and the `resultsFinished()`

```
bool getNextObject(Object& res) {
    std::unique_lock<std::mutex> lk(m_mutex);
    if (finished && retObjs.empty()) {
        lk.unlock();
        return false;
    }

    //Wait in a loop in case of spurious (accidental) wake ups
    while (!finished && retObjs.empty() {
        m_cv.wait(lk);
    }

    if (finished && retObjs.empty()) {
        lk.unlock();
        return false;
    } else {
        res = std::move(retObjs.front());
        retObjs.pop();
        lk.unlock();
        return true;
    }
}
```

Listing 5.2: The implementation of `getNextObject()` in `AsyncObjectStream`

method increments the finished counter to indicate how many producers have so far finished. We have not implemented this behaviour since multi-producer patterns are rather rare.

5.2.2 `FunctionExecutorThread`: a long-running executor thread

The next piece towards object-oriented multithreading support is a specialized, long-running thread object. Recall that we look to overcome two challenges: keeping a thread object alive beyond the execution of a single function, and having a thread execute multiple different functions.

To overcome both of these challenges, our `FunctionExecutorThread` internally uses the `AsyncObjectStream` to receive `std::function` objects to execute asynchronously. The design of the `FunctionExecutorThread` follows a straightforward design, again encapsulating the multi-threading aspects.

(*i*) On construction, the `FunctionExecutorThread` spawns an internal `std::thread`.

(*ii*) The `std::thread` enters an *event loop*, waiting on the `AsyncObjectStream`'s `getNextObject()` method.

(*iii*) Functions and code regions ("tasks") to execute are passed to the thread as `std::function` objects.

(*iv*) On destruction, the `FunctionExecutorThread` joins the `std::thread`.

The `FunctionExecutorThread` interface provides two methods for the user; see Listing 5.3. The method `sendRequest()` receives a `std:function<void()>`, a function object which encompasses a void function taking no parameters. The `waitForThread()` method is a blocking function call which waits until the `FunctionExecutorThread` has completed executing all of its queued functions. In this way, the `FunctionExecutorThread` (and callers of its `sendRequest()` method) is the producer, meanwhile the internal `std::thread` is the consumer of function objects.

Recall that the `AsyncObjectStream` is templated by the object type to pass along the stream. For the `FunctionExecutorThread`, that type is `std::function<void()>`. Function objects with different return types and different numbers (and types) of parameters are actually different types because of template instantiation. We therefore require that the functions passed through the `AsyncObjectStream` are of a single type: having a void return and no parameters. However, this is not a restriction on the kind of functions which the `FunctionExecutorThread` can handle. First, any explicit return value can instead be returned by reference through a function parameter to create a void function. Second, with the help of `std::bind`, any function can have its arity reduced by binding particular objects (or fundamental data types) to function arguments; see Section 3.3.2. By binding an object to every function argument, we obtain a function object with no explicit parameters.

The `eventLoop()` method, alongside `AsyncObjectStream`'s blocking `getNextObject()` method is the key to keeping the worker thread alive. By entering a `while` loop whose condition is the return of `getNextObject()`, the thread will remain active until the stream is both closed and emptied of all data items; recall this method's implementation in Listing 5.2.

The `waitForThread()` method uses a condition variable to notify waiting threads that the worker has completed executing all of its previously queued tasks. The worker thread sets the `FunctionExecutorThread`'s `isIdle` variable upon competing a task in the event loop and finding that the queue of functions is empty. At the same time, it triggers a notification of the condition variable to wake up any threads waiting on it.

```
class FunctionExecutorThread {

    AsyncObjectStream<std::function<void()>> requestQueue;
    std::thread m_worker;
    std::mutex m_mutex;
    std::condition_variable m_cv;
    bool isIdle;

    FunctionExecutorThread() {
        //member functions have implicit pointer as first parameter
        m_worker = std::thread(&FunctionExecutorThread::eventLoop, this);
    }

    void eventLoop() {
        std::function<void()> task;
        while(requestQueue.getNextObject(task)) {
            task();
            std::unique_lock<std::mutex> lk(m_mutex);
            isIdle = requestQueue.streamEmpty();
            bool notify = isIdle;
            lk.unlock();
            if (notify) m_cv.notify_all();
        }
    }

    void sendRequest(std::function<void()>& f) {
        isIdle = false;
        requestQueue.addResult(f);
    }

    void waitForThread() {
        std::unique_lock<std::mutex> lk(m_mutex);
        while (!isIdle) {
            m_cv.wait(lk);
        }
    }
}
```

Listing 5.3: The implementation of the `FunctionExecutorThread` class.

5.2.3 ExecutorThreadPool

A fundamental structure of most parallel systems is a thread pool. Thread pools maintain a collection of long-running threads which wait to be given a task, execute that task, and then return to the pool. This avoids the overhead of repeatedly spawning threads and limits the number of threads to avoid over-subscription. Thread pools are also usually adjoined with a task queue. The purpose of the queue is to hold tasks in waiting when active and pending tasks outnumber threads created by the pool.

We have already seen the implementation of `FunctionExecutorThread` and how it is able to keep a thread alive and execute many tasks using an `AsyncObjectStream`. Our thread pool implementation, `ExecutorThreadPool`, is really just a collection of `FunctionExecutorThreads` combined with a task queue.

When a task is passed to the `ExecutorThreadPool` to run in parallel, it either adds the task to the pool's task queue, if all threads are busy, or passes the function object to an idle `FunctionExecutorThread` and (temporarily) removes that thread from the pool. Removing the thread from the pool signifies that it is busy and unable to process any new incoming tasks. Once an `FunctionExecutorThread` finishes executing its current task, it attempts to return itself to the pool. At this point, the `ExecutorThreadPool` gives a new task to the thread to execute, if any is available in its task queue, or adds the idle thread back to the pool.

The interface for `ExecutorThreadPool` has two simple methods which are symmetrical to the underlying `FunctionExecutorThread`. The `addTask()` method accepts a `std::function<void()>` object to be passed to a worker thread. The `waitForThreads()` behaves as `FunctionExecutorThread::waitForThread()`, except waits for all threads in the pool to become idle. This simple interface is shown in Listing 5.4.

One complication arises in this implementation: how does a `FunctionExecutorThread` notify the thread pool that it has become idle and is ready to receive another task? A naive solution would be for the `ExecutorThreadPool` to act as a scheduler, and simply push all incoming tasks to the `FunctionExecutorThread`'s queues. However, one cannot know in advance how long it will take to execute each incoming task. Statically assigning tasks to the `FunctionExecutorThreads` would result in poor load-balancing if the tasks are not nearly identical. This would be sufficient to implement, for example, the map pattern (Section 5.1.1), but would be insufficient in the general case where each task requires a different or unknown amount of time.

Therefore, our design is for each thread in the thread pool to only have one task assigned to it at a time, and to return to the pool to obtain new tasks. This inversion of responsibility, from `FunctionExecutorThread`s to the `ExecutorThreadPool`, can be implemented generically as a *callback* function. In particular, the callback function object is of type `std::function<void(FunctionExecutorThread*)>`. This additional parameter is useful for receivers of the callback to know which function executor thread is making the callback.

This callback function is executed by the `FunctionExecutorThread` each time it finishes executing a task, and its task queue is empty. This addition requires minimal modification. First, a `callback` instance variable is added to store the callback function

```
class ExecutorThreadPool {

private:
    std::deque<FunctionExecutorThread*> threadPool;
    std::deque<std::function<void()>> taskPool;
    std::mutex m_mutex;
    std::condition_variable m_cv; //used in waitForThreads

    void tryPullTask() {
        std::lock_guard<std::mutex> lk(m_mutex);
        if (!taskPool.empty() && !threadPool.empty()) {
            FunctionExecutorThread* worker = threadPool.front();
            threadPool.pop_front();
            std::function<void()> f = taskPool.front();
            taskPool.pop_front();
            worker->sendRequest(f);
        }
    }

public:
    ExecutorThreadPools(int nthreads);

    void addTask(std::function<void()> f) {
        std::unique_lock<std::mutex> lk(m_mutex);
        taskPool.push_back(f);
        lk.unlock();
        tryPullTask();
    }

    void waitForThreads() {...}
```

Listing 5.4: The `ExecutorThreadPool` class interface.

object. Second, a few lines are added to the `eventLoop()` method to call the callback. The updated `eventLoop()` method is shown in Listing 5.5. One may notice that the `FunctionExecutorThread`'s mutex is locked before obtaining the callback. As we will soon see in Section 5.3, this design allows the callback methods to be changed dynamically for more flexible usage of the thread.

The last feature of the `ExecutorThreadPool`, before we see its usage in implementing parallel patterns in the next section, is its implementation as a *singleton* (see [126, Section 6.5]). To avoid over-subscription, it is useful to limit the number of threads available to a user. We also want to enable cooperation between parallel regions and, again, avoid over-subscription where multiple parallel regions want to simultaneously use a thread pool. We have thus implemented the thread pool to be, by default, a singleton. Every code region in a program shares a single thread pool so that there are never too many

```
void FunctionExecutorThread::eventLoop() {
    std::function<void()> task;
    while(requestQueue.getNextObject(task)) {
        task();

        std::unique_lock<std::mutex> lk(m_mutex);
        std::function<void(FunctionExecutorThread*)> localCB = callback;
        if (localCB) {
            lk.unlock();
            localCB(this);
            lk.lock();
        }

        isIdle = requestQueue.streamEmpty();
        bool notify = isIdle;
        lk.unlock();
        if (notify) m_cv.notify_all();
    }
}
```

Listing 5.5: `FunctionExecutorThread` event loop with a callback function.

active threads at one time. Moreover, the default size of the thread pool is dynamically set to be equal to the number of hardware threads available on the computer executing the program minus one. The minus one arises as the program's main thread has already been created and occupies hardware resources. The thread pool's singleton pattern is shown in Listing 5.6.

```
class ExecutorThreadPool {

private:
  //pool size defaults to 1 less than the number of hardware threads
  ExecutorThreadPool(int nthreads = std::thread::hardware_concurrency()-1);

public:
  static ExecutorThreadPool& getThreadPool() {
    static ExecutorTreadPool pool;
    return pool;
  }
}
```

Listing 5.6: The `ExecutorThreadPool` as a singleton.

5.3 Support for Parallel Patterns

In this section we discuss how the classes described in the previous section may be used to implement parallel patterns. In some cases, such as asynchronous generators, a new class is needed. In other cases, slight modifications of the `ExecutorThreadPool` are needed. We begin in Section 5.3.1, showing how the workpile pattern can be implemented with no modifications to the original design. Next, Section 5.3.2 shows our additions to the `ExecutorThreadPool`'s interface to support the map pattern and fork-join parallelism. Finally, Section 5.3.3 shows our implementation of asynchronous generators and pipelines using asynchronous object streams.

5.3.1 Support for Workpile

As discussed in Section 5.2.3, our thread pool uses callback functions for worker threads to notify the thread pool when they are idle and ready to execute another task. In this scheme, the default behaviour of the thread pool implements the workpile pattern. The thread pool's task queue is the workpile, and the worker threads take items from the queue until it is empty. Tasks themselves can add more tasks to the pool's task queue as needed.

Listing 5.7 shows a simple example implementing the workpile pattern. The function `processInt` is the task to execute. It receives a reference to the results queue and a data item `a` to process. Based on some condition (in this case if a is positive), another task is created. Otherwise, the result is added to the results queue. The function `WorkpileExample` initializes the computation by creating tasks from every item in the input queue and adding them to the thread pool's task queue. Notice that `std::bind` is used to create a function with arity 0. The function then waits for all threads to finish processing before returning, by calling the pool's `waitForAllThreads()` method.

5.3.2 Support for Map and Fork-Join

As we have just seen, the thread pool's default behaviour follows the workpile pattern. A map pattern *could* be implemented from this without any modification. If the number of available threads in the pool is more than the number of tasks (data elements to process) in the map pattern, then one could proceed by simply calling the pool's `addTask()` method a number of times.

However, the following observation suggests a different design is needed. The map pattern is often applied several times in a row, where one function after another is applied

```
void processInt(std::queue<int>& B, int a) {
    a -= 10; //do some computation using the data item a
    if (a > 0) {
        getThreadPool().addTask(std::bind(processInt, B, a));
    } else {
        B.push(a);
    }
}

void WorkpileExample(std::queue<int>& B, std::queue<int>& A) {
    ExecutorThreadPool& pool = getThreadPool();
    while (!A.empty()) {
        pool.addTask( std::bind(processInt, B, A.front()) );
        A.pop();
    }
    pool.waitForAllThreads();
}
```

Listing 5.7: Implementing the workpile pattern with `ExecutorThreadPool`.

to an ongoing collection of data items. This can take the form of, for example, a sequence of **parallel_for** loops. This has two implications. First, for data locality, it is preferable that, say, thread t_1 always operates on the data at index 1 in the collection, and thread t_2 always operates on the data at index 2. But, with the non-determinism of threads and the thread pool's task queue, the `addTask()` method cannot guarantee which thread will execute which task. Second, in the view of multiple code regions simultaneously using the thread pool, the number of idle threads in the thread pool may decrease between the first application of map and the second.

Our solution is to allow users to *reserve* a number of threads from the thread pool for their temporary, but exclusive use. The client code can then assign a particular function or code region to be executed by a particular thread, thus maintaining locality, as just discussed. Moreover, the threads which are reserved by the client continue to be reserved until they are returned to the thread pool. That is to say, reserved threads will not be used to execute tasks in the thread pool's task queue.

To abstract away the threads themselves, again encapsulating the parallel computing aspects, reserving threads from the thread pool returns a list of thread IDs rather than actual threads. Then, tasks are executed by those reserved threads through the pool's `executeTask()` method, which takes a thread ID and a task to execute.

The last step in this design is to provide a *barrier* mechanism, where the code must wait for all threads executing the map tasks to complete before continuing. Of course, the client code must ensure that all tasks have completed before continuing. A simple

```
1  class ExecutorThreadPool {
2      //Step 1: obtain threadIDs, removing them from the pool,
3      //        returns the number of threads actually obtained
4      int  obtainThreads(int numThreads, std::vector<threadID>& ids);
5
6      //Step 2: execute a task on a particular thread
7      void executeTask(threadID id, std::function<void()>& f);
8
9      //Step 3 (optional): wait for threads to become idle
10     void waitForThreads(std::vector<threadID>& ids);
11
12     //Step 4: return threads to pool (waits before returning)
13     void returnThreads(std::vector<threadID>& ids);
14 }
```

Listing 5.8: `ExecutorThreadPool` interface for reserving threads.

overloading of the thread pool's `waitForThreads()` method takes a list of thread IDs to block the caller until this subset of threads has completed their tasks and have become idle. Note that returning reserved threads to the thread pool also implicitly waits for them all to become idle.

The entire process of: (*i*) reserving a thread, (*ii*) executing a task using that thread, (*iii*) waiting on that thread to finish, and (*iv*) returning threads to the pool, is shown in the modified interface of `ExecutorThreadPool` (Listing 5.8), and exemplified as a generic parallel map function in Listing 5.9. Notice that, unlike the previous workpile example in Listing 5.7, the workpile pattern could also be implemented by reserving a subset of the pool's threads and then only waiting on that subset of threads.

Implementing the fork-join pattern from this design is straightforward. Indeed, one can implement fork-join using the methods of Listing 5.8, passing 1 for `numThreads` in `obtainThreads()`. However, using `vectors` of size 1 is rather clumsy. Hence, we also provide a symmetric set of methods: `obtainThread()` returns a single thread ID, and `waitForThread()` and `returnThread()` take a single threadID as a parameter. Implementing fork-join parallelism, while also keeping the multithreading details out of the client code, is simple with these new methods. An example of a divide-and-conquer fork-join merge sort is shown in Listing 5.10 using these new methods.

A key element to this design is that the thread pool may or may not return a thread ID from `obtainThread()` (hence "possibly" fork, on Line 11 of Listing 5.10). Of course, one should not fork so many threads that oversubscription occurs. We describe this "optional and cooperative" parallelism later in Section 5.4. We conclude this section by looking at the implementation of asynchronous generators and the pipeline pattern.

```
/**
 * Execute function f on each element of A and return the results in B.
 * The arrays have length n.
 */
void MapExample(int* B, int* A, int n, std::function<void(int*,int*)> f) {

    ExecutorThreadPool& pool = getThreadPool();
    std::vector<threadID> ids;
    pool.obtainThreads(n-1, ids); //assume n-1 threads avail.

    for (int i = 0; i < n-1; ++i) {
        pool.executeTask(ids[i], std::bind(f, &B[i], &A[i]));
    }
    f(&B[n-1], &A[n-1]); //use main thread for one call

    pool.returnThreads(ids); //also waits for threads
}
```

Listing 5.9: Implementing the map pattern with `ExecutorThreadPool`.

```
/**
 * A parallel divide-and-conquer mergesort algorithm.
 * Sorts the array A over the index range [i,j).
 */
void mergeSort(int* A, int i, int j) {
    if (j - i <= 1) {
        return;
    }
    int k = i + (j-i) / 2;

    threadID id = getThreadPool().obtainThread();
    //possibly fork a recursive call if a thread obtained
    getThreadPool().executeTask(id, std::bind(mergeSort, A, i, k));
    mergeSort(A, k, j);

    //join by waiting for the thread and returning it to the pool
    getThreadPool().returnThread(id);

    merge(A, i, k, j); //merge the two partitions
}
```

Listing 5.10: Implementing fork-join parallelism with `ExecutorThreadPool`.

5.3.3 Asynchronous Generators and Pipelines

Following the object-oriented design of our thread pool and the standard `std::function` function objects, we look to encapsulate the functionality of a generator or iterator as objects. We have created a generic `AsyncGenerator` class template, where a generator is constructed by passing it a function and its arguments. This function would typically return a collection of items, but will now be modified slightly to yield data items one a time. This class encapsulates the generation and queuing of objects, providing an object-oriented interface for producers to produce objects and consumers to consume objects. The creator of the generator (i.e. the consumer) requests data from the generator object rather than directly calling the function.

```
template <class Object>
class AsyncGenerator {

    // Consumer: create generator to encapsulate a function call.
    template<class Func, class... Args>
    AsyncGenerator(Func&& f, Args&&... args) {
        std::function<void()> F = std::bind(std::forward<Func>(f),
                                    std::forward<Args>(args)...,
                                    std::ref(*this));

        if (getThreadPool().allThreadsBusy()) {
            F();
        } else {
            getThreadPool().addTask(F);
        }
    }

    // Producer: add a new Object to be retrieved later.
    void generateObject(Object& obj);

    // Producer: finalize the AsyncGenerator by declaring that it has
    //           finished generating all possible objects.
    void setComplete();

    // Consumer: obtain the next generated Object by reference,
    // returns false iff no more objects available and setComplete()
    bool getNextObject(Object& obj);
};
```

Listing 5.11: The `AsyncGenerator` class interface. `class... Args` signifies variadic template parameters. Note that `std::forward` is used to capture const-ness, lvalue, or rvalue information from the parameters, since template pararmeters do not contain such information.

The `AsyncGenerator` is essentially an `AsyncObjectStream` combined with a worker thread to execute the producer. The `AsyncGenerator` also handles the complexity of getting a reference to the object stream to both the producer and consumer. For the consumer to behave as closely as possible to its non-generator counterpart (a consumer which simply receives a list of returned values), we want the generator itself to handle the creation of the object stream and the communication with the producer. Moreover, we wish for `AsyncGenerator` to *allow* for asynchronous generation, but without *requiring* it. Recall we are motivated by the dynamic multithreading paradigm.

Serially, a generator object could be implemented by collecting the objects returned by the function in a queue and yielding them one at a time to the caller. In fact, the `AsyncGenerator` behaves in this way when no worker threads are available in the `ExecutorThreadPool`. For true parallelism, the `AsyncGenerator` passes the producing function (which it was given in its constructor) to the `ExecutorThreadPool` via the pool's `addTask()` method.[2]

The `AsyncGenerator` interface is shown in Listing 5.11. Much like `AsyncObjectStream`, the `AsyncGenerator` is a class template which is templated by the type of object to be generated. The interfaces of the two class are also very similar. `AsyncGenerator` has methods `generateObject()` and `setComplete()` which simply delegate to object stream's `addResult()` and `resultsFinished()` (see Listing 5.1). Similarly, the generator's `getNextObject()` method behaves identical to, and delegates to, the object stream's `getNextObject()` method. Therefore, sleeping the consumer when no object is available to consume is automatic via the `AsyncObjectStream`.

The constructor of an `AsyncGenerator` is worthy of some discussion. The connection between producer and consumer begins with the producing function being passed to the constructor. The `AsyncGenerator` then inserts a reference to itself into the function's argument list. This allows the producer to have a reference to the `AsyncGenerator` object being constructed. Then, the `AsyncGenerator` queries the `ExecutorThreadPool` to determine if all threads are currently busy. If so, the generator serially executes the function on the existing calling (consumer's) thread. Otherwise, the generator passes the function object to the thread pool to be executed asynchronously.

In this design, the one restriction of possible generating functions is that they must take an `AsyncGenerator` object as their last parameter. Indeed, this is required as the generating function must somehow obtain a reference to the generator itself. An example

[2]Note that it is also possible for the generator to spawn its own `FunctionExecutorThread` for explicit parallelism rather than relying on the thread pool. We have implemented both, the former being called `AsyncGeneratorThread` and the latter called `AsyncGeneratorPool`.

```
/**
 * A producer function making use of AsyncGenerator
 * to generate the Fibonacci sequence up to index n.
 */
void GenerateFib(int n, AsyncGenerator<int> gen) {
    int fn1 = 0;
    int fn  = 1;
    for (int i = 0; i < n; ++i) {
        gen.generateObject(fn1); //generate an object
        fn  = fn + fn1;
        fn1 = fn - fn1;
    }
    gen.setComplete();
}

/**
 * A consumer function making use of AsyncGenerator to
 * execute GenerateFib in parallel.
 */
void ConsumeFib() {
    int n;
    std::cin >> n;

    AsyncGenerator<int> gen(GenerateFib, n);
    int fib;
    while(gen.getNextObject(fib)) { //consume an object
        std::cout << fib << " ";
    }
    std::cout << "\n"
}
```

Listing 5.12: An example of using an `AsyncGenerator` to implement a generator function for the Fibonacci sequence.

of the `AsyncGenerator`'s use in a client code is shown in Listing 5.12. Notice that all of the multithreading and parallelism is encapsulated by the generator's object interface.

With asynchronous generators, implementing the pipeline pattern is easy and immediate. Each stage of the pipeline can be implemented as a generator which also consumes data from a generator. The only special case is the first stage of the pipeline, which is only a producer and not a consumer. However, this special case is simple as this is just the normal usage of an `AsyncGenerator`. A simple example is shown in Listing 5.13. This example is very simple in what it computes—just a sequence of integers from 1 to 100—but, the flexibility exists where each pipeline stage may perform some computation with the data item it receives from the previous stage.

```
/**
 * The first stage of a pipeline is strictly a producer.
 */
void firstStage(int n, AsyncGenerator<int>& gen) {
    for (int i = 1; i <= n; ++i) {
        gen.generateObject(i);
    }
    gen.setComplete();
}

/**
 * Every other stage in a pipeline is both a consumer and a producer,
 * it consumes from prev and produces to next.
 */
void pipelineStage(AsyncGenerator<int>& prev, AsyncGenerator<int>& next) {
    int i;
    while (prev.getNextObject(i)) {
        //compute with and modify the previous stage's object
        next.generateObject(i);
    }
    next.setComplete();
}

/**
 * An example pipeline created from AsyncGenerator objects.
 */
void PipelineExample() {
    AsyncGenerator<int>   stageOne(firstStage, 100);
    AsyncGenerator<int>   stageTwo(pipelineStage, stageOne);
    AsyncGenerator<int> stageThree(pipelineStage, stageTwo);
    AsyncGenerator<int>  stageFour(pipelineStage, stageThree);

    //consume from last stage of pipeline
    int i;
    while(stageFour.getNextObjext()) {
        std::cout << i << std::endl;
    }
}
```

Listing 5.13: Implementing the pipeline pattern with `AsyncGenerators`.

5.4 Optional and Cooperative Parallelism

Throughout Section 5.2 and Section 5.3 we have seen the design and implementation of an object-oriented library in support of multithreading and the implementation of parallel patterns. With this object-oriented design, we were able to encapsulate and hide all of the concurrency and parallel computing details. Spawning and joining threads, synchronization via mutexes, and notification of threads via conditional variables, were all hidden behind an object-oriented interface.

Recall that this design was motivated by the dynamic multithreading paradigm. Alongside encapsulating all the difficulties of parallel programming, our classes and interfaces also allow for *optional parallelism*. That is, client codes describe where concurrency is possible by using our various classes, and the classes internally decide between serial and parallel execution. All of our classes and interfaces merely *suggest* concurrency, but did not explicitly require it.

The basis of all of our classes is the shared thread pool `ExecutorThreadPool`. The decision between executing a code region in parallel or not essentially comes down to whether or not the thread pool has idle threads available. In our design of the thread pool and generators, a large benefit is that the client code does not need to know if the requested code region actually runs in parallel or not; the client code remains identical regardless.

The definition of `AsyncGenerator` makes it obvious that parallelism is optional. The generator's constructor (see Listing 5.11), will execute the producer function serially on the calling thread if all of the thread pool's threads are busy. The optional parallelism in the implementation of the fork-join and map patterns is less obvious. Consider again Listing 5.10, where a parallel implementation of merge sort uses fork-join parallelism. In that example, we used the `ExecutorThreadPool` methods `obtainThread()`, `executeTask()`, and `returnThread()`. It seems to the client that `executeTask()` would execute the task in parallel and `returnThread()` would wait for the task to finish. However, it is very possible that there are no threads available to be reserved and returned by `obtainThread()`. If that is the case, `obtainThread()` returns a special identifier to say that no thread was available. However, the client code does not need to handle this special case. Indeed, `executeTask()`, upon receiving this special thread ID, will execute the task serially on the calling thread. Since the task was executed serially, the task finishes before `executeTask()` returns, and the call to `waitForThread()` or `returnThread()` returns immediately without any synchronization.

A similar design is applied to the symmetric methods which are used to implement

the map pattern. In the `obtainThreads()` method, the thread pool will only reserve a number of threads which is the maximum between the number of currently idle threads and the number requested. The method returns the actual number of threads obtained. Much like the fork-join example in Listing 5.10, the client code needs to do very little when fewer threads are obtained than requested. Indeed, it is most common that the map pattern will be applied to a collection of data items whose size exceeds the number of threads. Therefore, unlike the example shown previously in Listing 5.9, a client code would typically partition the index range over the number of threads and have each thread execute a certain number of function calls. This partitioning scheme, and the fact that the client code needs no special considerations if `obtainThreads()` return less threads than expected, is shown in Listing 5.14.

```
/**
 * Execute function f on each element of A and return the results in B.
 * The arrays have length n.
 */
void MapExample2(int* B, int* A, int n, std::function<void(int*,int*)> f) {

    int nWorkers = std::thread::hardware_concurrency() - 1;
    ExecutorThreadPool& pool = getThreadPool();
    std::vector<threadID> ids;
    nWorkers = pool.obtainThreads(nWorkers, ids);

    int tasksPerThread = n / (nWorkers+1); //+1 as main thread will do work
    for (int t = 0; t < nWorkers; ++t) {
        for (int i = 0; i < tasksPerThread; ++i) {
            pool.executeTask(ids[t], std::bind(f,
                                               &B[t * tasksPerThread + i],
                                               &A[t * tasksPerThread + i]);
        }
    }
    for (int i = tasksPerThread*nWorkers; i < n; ++i) {
        f(&B[i], &A[i]); //use main thread for the final partition
    }

    pool.returnThreads(ids); //wait for and return threads
}
```

Listing 5.14: Implementing the map pattern when the number of data items exceeds the number of threads.

The current design of our optional parallelism is to simply "give up" if no threads are available in the thread pool, and execute the task serially. However, sometimes it is worthwhile to temporarily allow over-subscription and (almost) enforce parallelism. For

example, if a large, coarse-grained parallel task is discovered, it is likely worthwhile to begin executing as soon as possible. Our solution to this use case is "high priority tasks". Moreover, these priority tasks enable a sort of *cooperative parallelism*. Tasks which are deemed to be high-priority get priority access to hardware resources to begin executing as soon as possible. "Normal" tasks, rather, must either wait their turn or execute serially (as in the case of `AsyncGenerator` and the implementation of the map pattern).

Priority tasks have a special effect with respect to resource allocation, resource usage, and task scheduling. There are three possible scenarios for a priority task.

(*i*) If there is an idle thread in the thread pool, it executes the priority task immediately just as a normal task.

(*ii*) If all threads in the pool are busy when a priority task is added, then the `ExecutorThreadPool` temporarily expands the size of the pool by spawning a new so-called *priority thread* which immediately executes the priority task.

(*iii*) If a maximum number of priority threads have been spawned and they are all busy, then the priority task is inserted in the front of the queue, and is thus the next task to be started by the next available thread.

In case (*ii*), over-subscription is temporarily allowed because it is assumed that the existing tasks occupying the existing threads are finer-grained tasks and smaller amounts of work than the incoming priority task. Thus, the currently executing normal tasks should finish quickly. To return to a state without over-subscription, we "retire" a thread pool thread for every priority thread spawned. This retirement occurs once a thread becomes idle and wishes to be put back into the pool. To avoid over-subscription even among priority tasks, we limit the total number of spawned priority threads, as in case (*iii*). This limit is typically equal to the thread pool's original size, thus avoiding the case that the priority threads alone cause over-subscription.

It is future work to further investigate the ideas of cooperative parallelism beyond just high-priority and low-priority threads. Indeed, general questions arise. How can a programmer decide—or a program dynamically decide for itself—which code regions to execute in parallel and how to execute them in parallel? Which regions should be considered high priority? How can a program mediate the competition for hardware resources from many different parallel regions? These open questions, and possible solutions, are discussed later in Chapter 9.

Now, we explore the usage of this generic parallel support applied to the specific problem of triangular decomposition. The next chapter discusses high-performance considerations for triangular decomposition. Parallelism specifically is explored in Section 6.3.

Chapter 6

High-Performance Triangular Decomposition

Solving a polynomial system by means of triangular decomposition entails computing a collection of regular chains which together encode the zero set of the input system. Where triangular decomposition proceeds incrementally, that is, by solving one equation after the other, a splitting of the quasi-component of a regular chain may be discovered when intersecting the next polynomial of the input system and the current partial solution. Concurrency is possible as the decomposition proceeds independently on each branch, or component, of the solution set. We call this *component-level parallelism*. This parallelism is our main motivator behind implementing a high-performance polynomial system solver.

Since solving systems of equations is a foundational problem across disciplines, developing more optimized system solvers is a natural direction for research. In this chapter, we examine the many directions we have taken to improve the performance of our triangular decomposition implementation.

As discussed in Section 1.1 and Section 2.5, triangular decomposition based on regular chains offers theoretical and practical advantages over methods based on Gröbner bases. Unfortunately, the performance of triangular decomposition to date remains limited. The most modern implementation may be found in the *RegularChains* library of *Maple* [125]. Yet, its implementation in the interpreted *Maple* language leaves many opportunities for performance gains. Using a compiled language is an obvious choice. Not only would this improve upon the inherent performance limitations of an interpreted environment, but a low-level language like C/C++ would also offer fine control over hardware resources for improved performance via data locality and parallelism.

Component-level parallelism in triangular decomposition was first examined in [144]. However, this implementation used only multi-processing for parallelism, and was lim-

ited by inter-processor communication. Moreover, the implementation relied on solving polynomial systems modulo a prime number. Working over a prime field generates extra splittings (via polynomial factorization, compared to polynomial factorization over the integers) and thus provides extra opportunities for parallelism. Solving systems instead over the rational numbers provides less opportunity for parallelism but is of more practical importance. Despite these challenges, we investigated opportunities for thread-level parallelism in triangular decomposition algorithms over the rational numbers.

Our implementation of triangular decomposition is for systems with rational number coefficients. It is part of the open-source Basic Polynomial Algebra Subprograms (BPAS) library [7]. As discussed in Chapter 4, this library is implemented primarily in C, with an object-oriented wrapper in C++. As we will see throughout this chapter, we have leveraged the tools available in this environment to significantly improve upon the performance of triangular decomposition. Core operations are implemented in C, like subresultants and pseudo-division, meanwhile regular chains and component-level parallelism are implemented as objects in C++.

We organize our discussion as follows. Section 6.1 presents the algorithms, routines, and supporting operations required of triangular decomposition based on regular chains. This includes sparse and dense polynomial arithmetic, polynomial GCDs and factorization, and subresultants. Section 6.2 presents an overview of our schemes for computing subresultants which consider data locality, parallelism, and algorithmic techniques to avoid unnecessary computation. With these core operations and data structures implemented well, our implementation of triangular decomposition is already formidable. Experimentation against the *RegularChains* library confirms this. However, as suggested throughout this thesis, component-level parallelism is possible for additional performance. We explore these concurrency opportunities in Section 6.3, where the main algorithms of triangular decomposition are examined in light of parallel patterns and the support for their implementation provided by BPAS (see Chapter 5).

Finally, extensive experimentation is discussed in Section 6.4. With help from the developers of *Maple*, we have accumulated a suite of over 3000 polynomial systems coming from the literature, real-world *Maple* user data, and bug reports. Our implementation of triangular decomposition, and its many different configurations (e.g. *Kalkbrener decomposition* vs. *Lazard-Wu decomposition*, serial vs. parallel), are systematically tested against this test suite. This data provides a large amount of insight into the runtime characteristics of triangular decomposition in general, insight into component-level parallelism, and insight into our particular implementation.

6.1 Triangular decomposition based on regular chains

We take this section to review the key operations and algorithms used within triangular decomposition. Recall that definitions and notations of regular chains and triangular sets were presented in Section 2.5.

In this section let us take $\mathbb{K}$ to be a perfect field and the polynomial ring $\mathbb{K}[\underline{X}] := \mathbb{K}[x_1, \ldots, x_n]$ to have ordered variables $x_1 < \cdots < x_n$. Let $T \subset \mathbb{K}[\underline{X}]$ be a regular chain, and $F \subset \mathbb{K}[\underline{X}]$ be a set of polynomials for which we want to find $V(F)$, the algebraic set consisting of the common roots of the polynomials of F.

Triangularize

The goal of triangular decomposition is to represent $V(F)$ as a set of regular chains $T_1, \ldots, T_e$ whose union of quasi-components equals $V(F)$. That is, $V(F) = \bigcup_{i=1}^{e} W(T_i)$. This is a *Lazard-Wu decomposition*. A Kalkbrener decomposition aims to find only the generic zeros of $V(F)$, thus obtaining regular chains $T_1, \ldots, T_e$ such that $V(F) = \bigcup_{i=1}^{e} \overline{W(T_i)}$. In either case, the triangular decomposition can be computed *incrementally* through repeated *intersection*. Namely, given a $p \in F$, and a regular chain T, one computes $V(p) \cap W(T)$. This process is repeated for each polynomial in the input system, resulting in a triangular decomposition. We present the incremental algorithm Triangularize to compute a Lazard-Wu triangular decomposition; see Algorithm 6.1. The order in which polynomials from the input system are intersected does not change the correctness of the algorithm. For efficiency purposes, one typically processes the "simplest" polynomial first, the polynomial with smallest rank.

Unfortunately, this process may result in *redundant components*, where there exist regular chains T_i and T_j in the output list such that $W(T_i) \subseteq W(T_j)$. Deciding whether or not such set inclusions hold can be performed quickly in practice with a heuristic algorithm. The function IsNotIncluded(T_i, T_j) returns true if $W(T_i) \not\subseteq W(T_j)$; see [187, Ch. 8]. Typically, the output of a triangular decomposition should be pair-wise irredundant to facilitate easier processing of the solutions. Later, in Section 6.3, we will examine effective parallel methods for computing an irredundant triangular decomposition. We will also see the effect that intermediately removing redundant components has on the performance of Triangularize.

Notice that Algorithm 6.1 creates a rooted tree of regular chains. The root is the empty regular chain $\varnothing$ and the leaf nodes are the final output regular chains $T_1, \ldots, T_e$. An edge connects node T to node T' whenever T' is produced by the intersection between p and T for some $p \in F$. Notice that, for any edge (T, T') in this tree, we have $|T| \leq |T'|$,

Algorithm 6.1 TRIANGULARIZE(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
1: 𝒯 := {∅}
2: for p ∈ F do
3: |  𝒯' := ∅
4: |  for T ∈ 𝒯 do do
5: |  |  𝒯' := 𝒯' ∪ INTERSECT(p, T)
6: |  𝒯 := 𝒯'
7: return REMOVEREDUNDANTCOMPONENTS(𝒯)
```

implying that the dimension of $sat(T')$ is at most the dimension of $sat(T)$. This implies Algorithm 6.1 can be modified to incrementally compute a Kalkbrener decomposition by simply *pruning branches*, or discarding any regular chains T found where $|T| > |F|$.

This observation is a consequence of *Krull's height theorem* (see [49, Theorem 4.4]), which states that the height of any prime ideal associated with $\langle F \rangle$ is less than or equal to $|F|$. Formally, the following corollary proves that discarding regular chains with this property does not change the Kalkbrener decomposition.

Corollary 6.1. *Let $\mathcal{T}$ form a Kalkbrener decomposition of $V(F)$, where F generates a proper ideal of $\mathbb{K}[x_1, \ldots, x_n]$. Let $T \in \mathcal{T}$ be a regular chain such that $|T| > |F|$, then $\mathcal{T} \setminus \{T\}$ is also a Kalkbrener decomposition of $V(F)$.*

Proof. We have $V(F) = \bigcup_i V(\mathcal{P}_i)$, where $\mathcal{P}_i$ are the minimal prime ideals associated with $\langle F \rangle$. By Krull's height theorem, the height of each $\mathcal{P}_i$ is less than or equal to $|F|$. This implies that the dimension of $V(\mathcal{P}_i)$ for each $\mathcal{P}_i$ is at least $n - |F|$. Since a regular chain's saturated ideal is unmixed, we have that the minimal prime ideals associated with T must all have height $|T| > |F|$ and thus the dimension of $\overline{W(T)}$ is less than $n - |F|$. Therefore, we must have that, for $\mathcal{T}' = \mathcal{T} \setminus \{T\}$, $V(F) = \bigcup_i V(\mathcal{P}_i) = \bigcup_{T' \in \mathcal{T}'} \overline{W(T')}$. □

Note that this corollary also implies that a Lazard-Wu decomposition and a Kalkbrener decomposition coincide when $V(F)$ is zero-dimensional. For the sake of clarity in the following sections and algorithms, we ignore this *height bound*, but the algorithms can easily be modified to simply check every regular chain produced and discard it if its height exceeds $|F|$. Indeed, our implementation does just that to support incremental solving for both Kalkbrener decompositions and Lazard-Wu decompositions.

Note that, while the organization of Algorithm 6.1 follows most easily from the ideas of incremental decomposition, it is not the only organization. Using the terminology of a rooted tree of regular chains, Algorithm 6.1 follows a breadth-first organization.

Later, Algorithm 6.6 will show an equivalent algorithm which proceeds recursively in a depth-first manner.

Intersect

Moving to the Intersect operation, we recall its specification from Section 2.5. Given a polynomial p and a regular chain T, Intersect returns a *regular split* of (p, T). That is, a collection of regular chains $T_1, \ldots, T_e$ satisfying the property:

$$V(p) \cap W(T) \subseteq W(T_1) \cup \ldots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}.$$

For now, let us make some assumptions on p to describe the typical case of Intersect. The full case is detailed next in Section 6.1.1. Assume $v = \text{mvar}(p)$, $v \in \text{mvar}(T)$, $T_v^+ = \varnothing$, and $\text{init}(p)$ is regular with respect to $\text{sat}(T)$. Then, Intersect reduces to computing a regular GCD between p and T_v, and some recursive calls; see Algorithm 6.2. The regular GCD encodes places where p and T share common roots meanwhile the recursive calls ensure no solution is missing under special circumstances. For example, Lines 9–11 consider the case where both the input polynomial p and the leading coefficient of the regular GCD are zero. The justification of this algorithm follows directly from Proposition 2.37.

Algorithm 6.2 IntersectAlgebraicTypical(p, T)

Input: $p \in \mathbb{K}[\underline{X}]$, $p \notin \mathbb{K}$, $v := \text{mvar}(p)$, a regular chain $T \subseteq \mathbb{K}[\underline{X}]$ such that $v \in \text{mvar}(T)$, $T_v^+ = \varnothing$, and $\text{init}(p)$ is regular w.r.t. $\text{sat}(T)$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}$

```
1:  T := ∅
2:  for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
3:      if dim(T_i) ≠ dim(T_v^-) then
4:          for T_{i,j} ∈ Intersect(p, T_i) do
5:              T := T ∪ {T_{i,j}}
6:      else
7:          if g_i ∉ K and mvar(g_i) = v then
8:              T := T ∪ {T_i ∪ g_i}
9:          for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
10:             for T_{i,j,k} ∈ Intersect(p, T_{i,j}) do
11:                 T := T ∪ {T_{i,j,k}}
12: return T
```

RegularGCD

The REGULARGCD algorithm computes the common solutions between two polynomials modulo a regular chain. As suggested in Section 2.4, one can compute a regular GCD by using the specialization property of subresultants.

Recall the properties we expect of a regular GCD g, coming from its definition, Definition 2.26, specialized to the case of a polynomial ring modulo a regular chain. Let p and t have main variable x_i. Let $T \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$ be a regular chain. $g \in \mathbb{K}[x_1, \ldots, x_i]$ is a regular gcd of p and t in $\mathbb{A} := \mathbb{K}[x_1, \ldots, x_{i-1}]/\sqrt{\text{sat}(T)}$ if:

(G_1) $\text{lc}(g, x_i)$ is a regular element of $\mathbb{A}$;

(G_2) $g \in \langle p, t \rangle \subset \mathbb{A}[x_i]$; and

(G_3) if $\deg(g, x_i) > 0$, then $\text{prem}(p, g) \in \sqrt{\text{sat}(T)}$ and $\text{prem}(t, g) \in \sqrt{\text{sat}(T)}$.

The following theorem forms the basis of the regular GCD algorithm.

Theorem 6.2. *Let $p, t, T, \mathbb{A}$ be as in the previous paragraph. Let $\{S_n, S_{n-1}, \ldots, S_1, S_0\}$ be the subresultant chain of p, t in $\mathbb{K}[x_1, \ldots, x_{i-1}][x_i]$, where $n := \min(\text{mdeg}(p), \text{mdeg}(t))$. If there exists some integer j such that for all $0 \leq i < j$, $s_i = 0$ in $\mathbb{A}$ and s_j is a regular element of $\mathbb{A}$, then S_j is a regular GCD of p and t in $\mathbb{A}[x_i]$.*

Proof. [47, Theorem 6] □

This theorem tells us that we can compute a subresultant chain over $\mathbb{K}[x_1, \ldots, x_{i-1}][x_i]$ and use it to compute a regular GCD in the ring $\mathbb{K}[x_1, \ldots, x_{i-1}]/\sqrt{\text{sat}(T)}[x_i]$. Algorithm 6.3 shows how to compute a regular GCD by processing a subresultant chain "bottom-up". Notice that on Line 8, asking whether $s_i \in \text{sat}(D)$ is the same as asking whether $s_i = 0$ in $\mathbb{A}$.

Algorithm 6.3 RegularGcd(p, q, v, T)

Input: $p, t \in \mathbb{K}[x_1, \ldots, x_i]$, $x_i = \text{mvar}(p) = \text{mvar}(t)$, a regular chain $T \subseteq \mathbb{K}[x_1, \ldots, x_{i-1}]$ such that init(p), init() are regular w.r.t. sat(T)

Output: a set of pairs $\{(g_1, T_1), \ldots, (g_e, T_e)\}$ such that $T \to T_1, \ldots, T_e$ and, if $\dim(T_k) = \dim(T)$, then g_k is a regular gcd of p, t w.r.t. T_k

```
1: if mdeg(p) > mdeg(t) then S := subres(p, t) else S := subres(t, p)
2: 𝒯 := ∅; Tasks := {(T, 0)}
3: while Tasks ≠ ∅ do
4:     Choose a pair (C, i) ∈ Tasks; Tasks := Tasks \ {(C, i)}
5:     for D in Regularize(lc(S_i, v), C) do
6:         if dim(D) < dim(C) then
7:             𝒯 := 𝒯 ∪ (0, D)
8:         else if lc(S_i, v) ∈ sat(D) then
9:             Tasks := Tasks ∪ {(D, i + 1)}
10:        else
11:            𝒯 := 𝒯 ∪ (S_i, D)
12: end while
13: return 𝒯
```

Regularize

RegularGCD relies on the Regularize algorithm to determine whether a polynomial is regular in the ring $\mathbb{K}[\underline{X}]/\sqrt{\text{sat}(T)}$ for some regular chain T. If not, Regularize computes a regular split of T to find components where the polynomial is regular and where it is not. This process, in turn, relies on RegularGCD, since a polynomial p will not be regular anywhere that it has common roots with a polynomial $T_v \in T$. A simplified Regularize procedure is shown in Algorithm 6.4 where it is assumed that the initial of the input polynomial is already regular.

While previous algorithms could return multiple components via recursive calls, Regularize explicitly computes a regular GCD and a pseudo-quotient to split the computation into multiple components. This is precisely property $(ii.b)$ of Proposition 2.37.

As we will see in Section 6.3, this splitting is the main source of component-level parallelism. Moreover, every sub-routine of Triangularize eventually calls Regularize via recursive calls, suggesting that component-level parallelism is possible more than just in the high-level Triangularize algorithm. The inter-dependency of Intersect, Regularize, RegularGCD, and other sub-routines is discussed in the next section.

Algorithm 6.4 REGULARIZE(p, T)

Input: $p \in \mathbb{K}[\underline{X}]$, $p \notin \mathbb{K}$, $v := \text{mvar}(p)$, a regular chain $T \subseteq \mathbb{K}[\underline{X}]$ such that $\text{init}(p)$ regular w.r.t. $\text{sat}(T_v^-)$ and $T_v^+ = \emptyset$

Output: regular chains $T_1, \ldots, T_e$ such that $T \to T_1, \ldots, T_e$

```
1: T = ∅
2: if v ∉ mvar(T) then return {T}
3: for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
4:     if dim(T_i) < dim(T_v^-) then
5:         for T_{i,j} ∈ Regularize(p, T_i) do
6:             T := T ∪ T_{i,j}
7:     else
8:         if g_i ∈ K or mvar(g_i) < v or mdeg(g_i) = mdeg(T_v) then
9:             T := T ∪ T_i
10:        else
11:            T := T ∪ {T_i ∪ g_i}
12:            q_i := pquo(T_v, g_i, v)
13:            for T_{i,j} ∈ Regularize(p, T_i ∪ {q_i}) do
14:                T := T ∪ T_{i,j}
15:            for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
16:                for T_{i,j,k} ∈ Regularize(p, T_{i,j}) do
17:                    T := T ∪ T_{i,j,k}
18: return T
```

Practical Considerations

Before giving the details of the many subroutines of TRIANGULARIZE in the next section, we conclude this section with some additional considerations for the algorithms which greatly improve their practical performance. For clarity in the algorithms' descriptions, and the fact that correctness of the algorithms do not rely on these optional considerations, the descriptions of the algorithms in the following section do not include these modifications. However, they are included in our implementation and play a vital role in its practical efficiency.

First, consider the order in which polynomials from the input system should be intersected. To illustrate this fact, consider a sequence of $n+1$ polynomials $f_1, \ldots, f_n, f_{n+1}$ so that $\langle f_1, \ldots, f_n \rangle$ is a proper ideal but $f_{n+1} = 1 - f_1 \cdots f_n$ holds. That is, $\langle f_1, \ldots, f_{n+1} \rangle = \langle 1 \rangle$ and the system $\{f_1, \ldots, f_{n+1}\}$ has no solution. If f_{n+1} is processed last, the inconsistency will not be discovered until the end. Processing f_{n+1} sooner allows computations to complete much faster.

Experience from developing the regular chains library [125] suggests that polyno-

mials should be processed in increasing *Ritt-Wu order*, that is, in order of increasing "complexity". This order is an extension of the classic *Ritt order* [16, Definition 2.3].

Definition 6.3 (Ritt-Wu Order)**.** *For two polynomials $p, q \in \mathbb{K}[\underline{X}]$ p is less than q in the* Ritt-Wu order*, and we write $p <_r q$ if:*

(i) $\operatorname{mvar}(p) < \operatorname{mvar}(q)$*; or*

(ii) $\operatorname{mvar}(p) = \operatorname{mvar}(q)$ *and* $\operatorname{mdeg}(p) < \operatorname{mdeg}(q)$*; or*

(iii) $\operatorname{mvar}(p) = \operatorname{mvar}(q), \operatorname{mdeg}(p) = \operatorname{mdeg}(q)$ *and* $\operatorname{init}(p) <_r \operatorname{init}(q)$*; or*

(iv) $\operatorname{mvar}(p) = \operatorname{mvar}(q), \operatorname{mdeg}(p) = \operatorname{mdeg}(q)$,
$\operatorname{init}(p) \not<_r \operatorname{init}(q)$, $\operatorname{init}(q) \not<_r \operatorname{init}(q)$ *and* $\operatorname{tail}(p) <_r \operatorname{tail}(q)$.

Second, we must consider expression swell. While the correctness of the triangular decomposition algorithms presented later in Section 6.1.1 (originally from [47]) does not rely on any simplification or reduction of intermediate polynomials, it is very important in practice. Reduce(p, T) returns a polynomial r such that $r \equiv p \bmod \operatorname{sat}(T)$ (i.e. $r \equiv h_T p \bmod \langle T \rangle$, where $h_T = \prod_{t \in T} \operatorname{init}(t)$). We say r is *reduced* with respect to T. We can reduce a polynomial p with respect to T by means of pseudo-remainders. Letting $T = \{t_1, \ldots, t_k\}$ we have:

$$\begin{aligned} & h_T p = q_1 t_1 + \cdots + q_k t_k + r, \quad \deg(r, \operatorname{mvar}(t_i)) < \operatorname{mdeg}(t_i) \text{ for } 1 \le i \le k \\ \Longrightarrow\ & h_T p \equiv r \bmod \langle T \rangle \\ \Longrightarrow\ & p \equiv r \bmod \operatorname{sat}(T). \end{aligned}$$

In dimension zero (i.e. when $k = n = |\underline{X}|$) reduction via pseudo-remainders should always occur since T gives a bound on the degrees of every variable which could appear in the reduced polynomial r. However, when T is positive dimensional, a pseudo-remainder computation may increase partial degrees. Let $\underline{V}$ be the algebraic variables of T and $\underline{U}$ be the free variables of T. Then, $r = \operatorname{prem}(p, T)$ will have $\deg(r, v) < \operatorname{mdeg}(T_v)$ for all $v \in \underline{V}$. However, r may possibly have *higher* partial degrees than p with respect to $u \in \underline{U}$.

An alternative simplification can occur which we call ReduceMinimal(p, T). This function does not necessarily fully reduce p with respect to T, but does compute a polynomial $r \equiv p \bmod \operatorname{sat}(T)$ where the partial degrees of r are less than or equal to p. We say that r is "cleaned" with respect to T. This simplification is much more practical in positive dimension and avoids expression swell in the free variables of T.

REDUCEMINIMAL first computes $\tilde{p} = \text{prem}(p, \tilde{T})$ for $\tilde{T} = \{t \mid \text{init}(t) \in \mathbb{K} \text{ for } t \in T\}$. Since the initials of every polynomial in $\tilde{T}$ are constant, there is no expression swell in the variables $\underline{X} \setminus \text{mvar}(T)$ and we still have $\tilde{p} \equiv p \bmod \text{sat}(T)$. Then, REDUCEMINIMAL calls REMOVEZERO$(\tilde{p}, T)$, shown in Algorithm 6.5. The goal of REMOVEZERO is to remove the parts of the polynomial p which are zero modulo $\text{sat}(T)$. If $v = \text{mvar}(p) \in \text{mvar}(T)$ a pseudo-remainder is computed but not necessarily returned. Only if the pseudo-remainder is 0 is it returned as the reduced polynomial. This ensures no expression swell occurs in the other variables. Otherwise, the function recurses on each coefficient of p where p is viewed as univariate in v. Notice that the final recursive call on Line 10 is the coefficient of p for v^0.

Algorithm 6.5 REMOVEZERO(p, T)

Input: a polynomial $p \in \mathbb{K}[\underline{X}]$ and a regular chain $T \subset \mathbb{K}[\underline{X}]$
Output: a "cleaned" polynomial r such that $r \equiv p \bmod \text{sat}(T)$

```
1:  v := mvar(p)
2:  if v ∈ mvar(T) then
3:  |   if prem(prem(p, T_v), T_v^-) = 0 then return 0
4:  if T_v^- = ∅ then return p
5:  r := 0
6:  while deg(p, v) > 0 do
7:  |   h = RemoveZero(init(p), T_v^-)
8:  |   r := r + h rank(p)
9:  |   p := tail(p)
10: p := RemoveZero(p, T_v^-)
11: return p + r
```

Simplification of polynomials with respect to a regular chain T occurs throughout all the subroutines of TRIANGULARIZE. To keep the pseudo-code clear, we do not show simplification or reduction in the pseudo-code presented later in Section 6.1.1. We describe here where reduction occurs in our implementation. Note that the choice of where to apply simplification and where not to apply it has been driven by experience and observation and is not necessarily optimal in general or for specific systems. Recall that when T is zero-dimensional, reduction of p may occur as $\text{prem}(p, T)$. Otherwise, p should be cleaned using REDUCEMINIMAL(p, T).

1. Any time a regular chain $T \cup \{p\}$ is created, p should first be simplified: INTERSECT, REGULARIZE, CLEANCHAIN, EXTEND.

2. When T is zero-dimensional, the polynomials output by *RegularGCD* and *Regularize* should be reduced.

3. When T is zero-dimensional, the input polynomial to *Intersect* should be reduced before beginning.

Finally, consider together simplification of polynomials and the order in which polynomials from the input set are intersected. As the algorithm progresses, there may be several active regular chains describing the components of the current solution space. The polynomials which remain to be intersected may simplify in different ways with respect to the different regular chains. Thus, it is advantageous to, for each independent component, simplify the polynomials which remain to be intersected, and order the simplified polynomials again with respect to the Ritt-Wu order. In the organization of Triangularize shown in Algorithm 6.1, this re-ordering and simplification is not possible since the order in which we iterate over F is fixed from the beginning of the algorithm. However, as we will see later in Algorithm 6.19, it is possible to assign a copy of the input set of polynomials to each active component, and then calls CleanSet on that copy before each incremental step. CleanSet(F, T) simplifies the polynomials of F with respect to T and then re-orders them with respect to the Ritt-Wu order.

Factorization in Triangular Decomposition

Although the Ritt-Wu characteristic set method was first created as a factorization-free routine, algorithms to compute factorizations have since become much more efficient in theory and in practice. While none of the algorithms presented so far have explicitly used factorization, we examine now some places where factorization can be used to reduce the degrees of intermediate polynomials and to introduce additional component-level parallelism.

The following obvious observation shows that factorization does not harm triangular decomposition.

Observation 6.4. Let $T_v = f^k g^\ell$ be a reducible polynomial in $\mathbb{K}[\underline{X}]$ for some $k, \ell \in \mathbb{Z}^+$ The Nullstellensatz and basic properties of ideals yields $V(T_v) = V(f^k) \cup V(g^\ell) = V(f) \cup V(g)$. Let T be a regular chain with $T_v \in T$ and, w.l.o.g., $T_v^+ = \varnothing$. Then we have:

$$W(T) \subseteq W(T_v^- \cup f) \cup W(T_v^- \cup g) \subseteq \overline{W(T)}.$$

Therefore, factoring T_v yields a valid regular split of T. Moreover, when the multiplicity of solutions is not required, one can ignore the multiplicity of the factors.

It is therefore useful to compute a factorization over $\mathbb{K}$ for any polynomial which is to be added to a regular chain. Let $p = p_1 \cdots p_k$. Where one would normally return $T \cup \{p\}$, instead return a list of regular chains $T \cup \{p_1\}, \ldots, T \cup \{p_k\}$. Consider REGULARIZE as shown in Algorithm 6.4. One can immediately factor the input polynomial p before computing a regular GCD. Moreover, one can factor each regular GCD on Line 11 and the pseudo-quotient on Line 12 to reduce degrees and obtain more components.

Note that irreducible factorization is not necessarily needed. A square-free factorization is easier to compute and would still produce more components. However, an irreducible factorization is even better towards those goals, particularly with recent algorithmic improvements in Hensel lifting [136, 140].

In our implementation we work over $\mathbb{Q}$. Over the rationals (or, equivalently, the integers), multivariate polynomial factorization typically proceeds via *Hensel lifting*. To factor a polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$, one chooses a suitable evaluation point $(\alpha_2, \ldots, \alpha_n) \in \mathbb{Z}^{n-1}$ and factors $f^{(1)} := f(x_1, \alpha_2, \ldots, \alpha_n)$ in $\mathbb{Z}[x_1]$. In our implementation, we choose α_i randomly from the set $[1, 200^{2^e}]$, where e ranges over $0, 1, 2, \ldots$, until a suitable point is found.

These factors are then passed to $\mathbb{Z}_{p^\ell}$, for some prime number p and suitably large ℓ, and then the factors are *lifted* to multivariate factors over $\mathbb{Z}_{p^\ell}[x_1, \ldots, x_n]$. One performs the lifting over $\mathbb{Z}_{p^\ell}$ rather than $\mathbb{Z}$ since the former is *almost* a field. Since ℓ is chosen to be sufficiently large, the factors in $\mathbb{Z}_{p^\ell}[\underline{X}]$ identify with the factors in $\mathbb{Z}[\underline{X}]$. Wang in [178] significantly improved multivariate polynomial factorization using an interative approach to Hensel lifting.

One can also view computing GCDs as a particular kind of factorization to which Hensel lifting can also be applied. Indeed, a polynomial can factor into a GCD and its co-factor. Classic GCD algorithms using Hensel lifting are the EZ-GCD algorithm [146], the EEZ-GCD algorithm [179].

While other methods for multivariate factorization have also been proposed [108, 189], the algorithm of Wang [178] remains the de facto standard implementation. It is used in the computer algebra systems *Maple* [128], *Singular* [66], and *Magma* [29].

Recent work by Monagan and Tuncer in [136, 140] show that one can improve Wang's method for multivariate factorization through an improved organization of the Hensel lifting steps, known as MTSHL. The MTSHL method is a probabilistic method where univariate factors are lifted to multivariate factors over the prime field $\mathbb{Z}_p$. Then, the coefficients of the multivariate factors are lifted from $\mathbb{Z}_p$ to $\mathbb{Z}_{p^\ell}$. The key observation of MTSHL is that most of the work comes from lifting the variables. Working over a prime field rather than $\mathbb{Z}_{p^\ell}$ significantly reduces the cost of Hensel lifting. The MTSHL method

significantly improves on the performance of multivariate factorization in its various implementations by Wang [178], Zippel [189], and Kaltofen [108]; see experimentation in [170, Ch. 6]. This new method has since been integrated into *Maple* and has become the default for multivariate polynomial factorization.

In BPAS, we have implemented Wang's multivariate factorization algorithm and EEZ-GCD algorithm. Both algorithms use the MTSHL method for the Hensel lifting steps. We note that, while BPAS does have an implementation of univariate factorization, it is not as high-performing as, for example, the implementation found in *NTL*. We thus use *NTL* to compute univariate factorization and then employ our Hensel lifting algorithms to lift the univariate factors to multivariate ones.

Our implementation of Hensel lifting has some differences from the original presentation [136]. First, it is completely implemented in C with a keen focus on data locality and data re-use. In the *Maple* implementation, only the bivariate base case is in compiled code. This has practical advantages when the polynomials to lift have high degree, many variables, or large coefficients. Second, we have applied the MTSHL method to also compute GCDs via the EEZ-GCD algorithm. In contrast, *Maple* currently uses the GCD method presented in [111] (an extension of Zippel's sparse interpolation method [190]) as the default, with Wang's EEZ-GCD used as a backup.

Without explaining the exact details of Hensel lifting and sparse interpolation (see [88, Ch. 6, Ch. 8] and [170] for a comprehensive discussion), we note some of our adaptations of MTSHL toward reduced memory usage and data locality.

- Multivariate polynomial arithmetic is performed *in-place* (without using any auxiliary memory).

- Lifting the coefficients of factors using p-adic updates is performed in-place, directly modifying the coefficients of the polynomial data structure (see [11]).

- Lifting the variables of factors is performed in-place. Indeed, our polynomial data structure supports *exponent vectors* of up to 32 variables within a fixed amount of space (64 bytes). Thus, adding variables requires only the traversal of an array and a few integer operations per monomial. However, this is not a limitation of the implementation. We also support "unpacked" exponents for polynomials with high partial degrees or large number of variables

- In solving a bivariate Diophantine equation (the base case of MTSHL), incremental Newton interpolation is performed in-place.

Consider REGULARGCD itself now. The algorithm proceeds "bottom-up" to find a regular GCD from the subresultant chain of two polynomials. In practice, the regular GCD is often of degree 1; this can be seen as consequence of the celebrated *Shape Lemma* [19]. In the context of REGULARGCD, this means that typically only the subresultants S_0 and S_1 are needed, and S_1 is typically the regular GCD. It is therefore often wasteful to compute the entire subresultant chain. As we will explore in Section 6.2, it is possible to compute subresultants *speculatively*, to compute a pair of subresultants S_{k-1} and S_k for some k, rather than computing the entire chain.

6.1.1 Specifications of Algorithms

In this section we list the specifications and pseudo-codes for TRIANGULARIZE and its subroutines. These algorithms for triangular decomposition were first presented in [52] by Changbo Chen as modifications of the algorithms presented in [142]. The new algorithms of [52] were later extended in [49] and [47].

As will become evident in the following pseudo-code, the key principle applied by Chen in [47, 52] is the idea of *recycling* a subresultant chain. For example, the subresultant chain computed in REGULARIZE to obtain the resultant of two polynomials can then be used in REGULARGCD. The subresultant chains computed in INTERSECT (Algorithm 6.7) are used to compute regular gcds in INTERSECTALGEBRAIC (Algorithm 6.9). The interdependence of the many subroutines of TRIANGULARIZE is summarized in Figure 6.1 as a flow graph.

To be explored in the remaining sections of this chapter, we have modified these algorithms to exploit parallelism, data locality, and the speculative computation of subresultants. However, the serial counterparts of these algorithms remain the same as those of [47, 49, 52]. We present these algorithms for reference and completeness Their original presentation, along with proofs of correctness, is [52]. More details on their underlying theory and organization can be found in [142], [49], and [47].

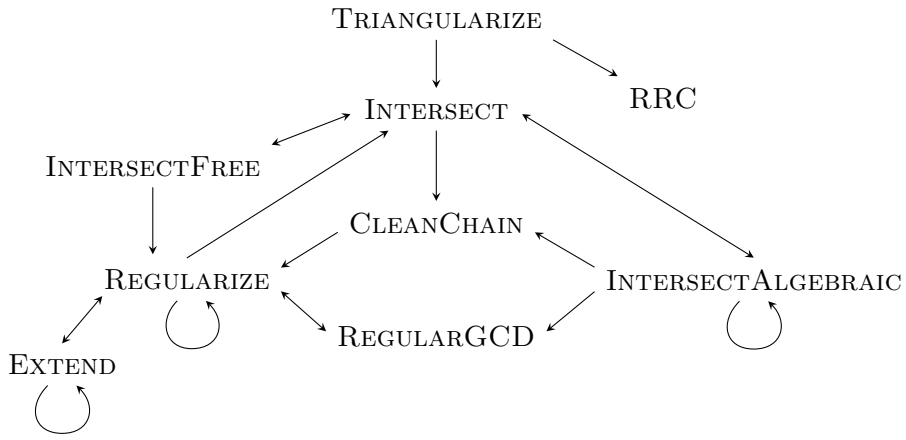

Figure 6.1: A flow graph of function calls within the Triangularize algorithm.

Specifications

Let $\mathbb{K}[\underline{X}] := \mathbb{K}[x_1 < \ldots < x_n]$ be a polynomial ring in n variables. Let $T \cup p$ be shorthand for $T \cup \{p\}$, for a triangular set T and a polynomial p.

Triangularize

Input:	$F \subset \mathbb{K}[\underline{X}]$, a finite set of polynomials.
Output:	A Lazard-Wu triangular decomposition of $V(F)$.
Description:	Computes a triangular decomposition of a set of polynomials.

Intersect(p, T)

Input:	A polynomial $p \in \mathbb{K}[\underline{X}]$, $T \subset \mathbb{K}[\underline{X}]$ a regular chain
Output:	A set of regular chains $T_1, \ldots, T_e$ such that $(p, T) \to T_1, \ldots, T_e$
Description:	Computes the common solutions of a polynomial and regular chain. This proceeds variable by variable through the variables of p, computing an *iterated resultant*.

Regularize(p, T)

Input:	A polynomial $p \in \mathbb{K}[\underline{X}]$, $T \subset \mathbb{K}[\underline{X}]$ a regular chain
Output:	A set of pairs $(p_1, T_1), \ldots, (p_e, T_e)$ such that for each $1 \leq i \leq e$, T_i is a regular chain, $p \equiv p_i \bmod \sqrt{\text{sat}(T_i)}$, and if $p_i = 0$ then $p \in \sqrt{\text{sat}(T_i)}$ otherwise p_i is regular modulo $\sqrt{\text{sat}(T_i)}$. Moreover, $T \to T_1, \ldots, T_e$.
Description:	Computes a splitting of a regular chain so that either p is 0 or regular on each component.

RegularGCD(p, q, v, S, T)

Input: Polynomials $p, q \in \mathbb{K}[\underline{X}]$, $\text{mvar}(p) = \text{mvar}(q) = v$, $T \subset \mathbb{K}[\underline{X}]$ a regular chain such that $v \notin \text{mvar}(T)$ and $T_v^+ = \varnothing$, $\text{init}(q)$ regular w.r.t. T, S the subresultant chain of p and q with respect to v, where the resultant $r \in \sqrt{\text{sat}(T)}$.

Output: A set of pairs $(g_1, T_1), \ldots, (g_e, T_e)$ such that $T \to T_1, \ldots, T_e$ and, for each $1 \leq i \leq e$, if $\dim(T) = \dim(T_i)$ then g_i is a regular GCD of p and q modulo $\sqrt{\text{sat}(T_i)}$.

Description: Computes a regular GCD with main variable v between p and q modulo $\text{sat}(T)$, using their subresultant chain S. This may cause T to split where the GCD must have a regular initial.

IntersectAlgebraic(p, T, x_i, S, C)

Input: A polynomial $p \in \mathbb{K}[\underline{X}]$ with $\text{mvar}(p) = x_i$, $T \subset \mathbb{K}[\underline{X}]$ a regular chain with $x_i \in \text{mvar}(T)$, S the subresultant chain of p and T_{x_i} with respect to x_i, $C \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$ a regular chain such that $\text{init}(T_{x_i})$ is regular modulo $\sqrt{\text{sat}(C)}$.

Output: A set of regular chains $T_1, \ldots, T_e$ such that $(p, C \cup T_{x_i}) \to T_1, \ldots, T_e$.

Description: Computes the common solutions a polynomial p and T_{x_i} modulo a regular chain C via a regular GCD computation. C represents $T_{x_i}^-$ or a component from a splitting of it and is the regular chain to be extended with the common solutions of p and T_{x_i}.

IntersectFree(p, x_i, T)

Input: A polynomial $p \in \mathbb{K}[\underline{X}]$ with $\text{mvar}(p) = x_i$, $T \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$ a regular chain.

Output: A set of regular chains $T_1, \ldots, T_e$ such that $(p, T) \to T_1, \ldots, T_e$.

Description: Computes the intersection of a polynomial with p with T where $\text{mvar}(p)$ is larger than every variable in T. This is "free" and p can be "put on top" of T, up to a splitting of T to ensure p is regular.

CleanChain(C, T, x_i)

Input: $T \subset \mathbb{K}[\underline{X}]$ a regular chain, $C \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$ a regular chain such that $\sqrt{\text{sat}(T_{x_i}^-)} \subseteq \sqrt{\text{sat}(C)}$.

Output: If $x_i \notin \text{mvar}(T)$, return $\{C\}$. Otherwise, a set of regular chains $T_1, \ldots, T_e$ such that for each $1 \leq j \leq e$, $\text{init}(T_{x_i})$ is regular modulo $\sqrt{\text{sat}(T_j)}$, $\sqrt{\text{sat}(C)} \subseteq \sqrt{\text{sat}(T_j)}$ and $W(C) \setminus V(\text{init}(T_{x_i})) \subseteq \bigcup_{j=1}^{e} W(T_j)$.

Description: Clean up a regular chain C with respect to T, where C is derived from an intersection of some polynomial and $T_{x_i}^-$. Computes a splitting of C so that $\text{init}(T_{x_i})$ is regular on each returned component.

Extend(C, T, x_i)

Input: $T \subset \mathbb{K}[\underline{X}]$ a regular chain, $C \subset \mathbb{K}[x_1, \ldots, x_{i-1}]$ a regular chain such that $\sqrt{\text{sat}(T_{x_i}^-)} \subseteq \sqrt{\text{sat}(C)}$.

Output: A set of regular chains $T_1, \ldots, T_e$ such that $W(C \cup T_{x_i} \cup T_{x_i}^+) \subseteq \bigcup_{j=1}^{e} W(T_j)$ and $\sqrt{\text{sat}(T)} \subseteq \sqrt{\text{sat}(T_j)}$.

Description: Extend the regular chain C by adding to it T_{x_i} and $T_{x_i}^+$. This may cause a splitting of C to keep the initials of the polynomials to add regular. This function is used in Regularize to "start over" when dimension drops to compute a new resultant and a new regularization of the resultant.

RemoveRedundantComponents$(\mathcal{T})$

Input: $\mathcal{T}$ a set of regular chains of $\mathbb{K}[\underline{X}]$

Output: A set of regular chains $\mathcal{T}'$ such that for any $T_i, T_j \in \mathcal{T}'$, $W(T_i) \not\subseteq W(T_j)$ and $\bigcup_{T \in \mathcal{T}} W(T) = \bigcup_{T' \in \mathcal{T}'} W(T')$

Pseudo-code

Algorithm 6.6 TRIANGULARIZE(F)

if $F = \varnothing$ **then return** $\{\varnothing\}$
$\mathcal{T} := \varnothing$
Choose a polynomial $p \in F$ with maximal rank
for $T \in$ TRIANGULARIZE($F \setminus \{p\}$) **do**
  $\mathcal{T} := \mathcal{T} \cup$ INTERSECT(p, T)
return REMOVEREDUNDANTCOMPONENTS($\mathcal{T}$)

Algorithm 6.7 INTERSECT(p, T)

if $\mathrm{prem}(p, T) = 0$ **then return** $\{T\}$
if $p \in \mathbb{K}$ **then return** $\varnothing$
$r := p$; $P := \{r\}$; $S := [\,]$ ▷ S is a map from variables to subresultant chains
while $\mathrm{mvar}(r) \in \mathrm{mvar}(T)$ **do**
  $v := \mathrm{mvar}(r)$; $src :=$ SUBRESULTANT(r, T_v, v)
  $S[x_i] := src$; $r := S_0 of src$, the subresultant of index 0
  if $r = 0$ **then break**
  if $r \in \mathbb{K}$ **then return** $\{\,\}$
  $P := P \cup \{r\}$
$\mathcal{T} := \{\varnothing\}$; $\mathcal{T}' := \varnothing$; $i := 1$
while $i \leq n$ **do** ▷ n from $\mathbb{K}[x_1, \ldots, x_n]$
  for $C \in \mathcal{T}$ **do**
    if $x_i \notin \mathrm{mvar}(P)$ **and** $x_i \notin \mathrm{mvar}(T)$ **then**
      $\mathcal{T}' := \mathcal{T}' \cup$ CLEANCHAIN(C, T, x_{i+1})
    else if $x_i \notin \mathrm{mvar}(P)$ **then**
      $\mathcal{T}' := \mathcal{T}' \cup$ CLEANCHAIN($C \cup T_{x_i}, T, x_{i+1}$)
    else if $x_i \notin \mathrm{mvar}(T)$ **then**
      for $D \in$ INTERSECTFREE(P_{x_i}, x_i, C) **do**
        $\mathcal{T}' := \mathcal{T}' \cup$ CLEANCHAIN(D, T, x_{i+1})
    else
      for $D \in$ INTERSECTALGEBRAIC($P_{x_i}, T, x_i, S[x_i], C$) **do**
        $\mathcal{T}' := \mathcal{T}' \cup$ CLEANCHAIN(D, T, x_{i+1})
  $\mathcal{T} := \mathcal{T}'$; $\mathcal{T}' := \{\,\}$; $i := i + 1$
return $\mathcal{T}$

Algorithm 6.8 RegularGcd(p, q, v, S, T)

```
1: 𝒯 := ∅; Tasks := {(T, 1)}
2: while Tasks ≠ ∅ do
3:     Choose a pair (C, i) ∈ Tasks; Tasks := Tasks \ {(C, i)}
4:     for (f, D) ∈ Regularize(lc(S_i, v), C) do
5:         if dim(D) < dim(C) then
6:             𝒯 := 𝒯 ∪ {(0, D)}
7:         else if f = 0 then
8:             Tasks := Tasks ∪ {(D, i + 1)}
9:         else
10:            𝒯 := 𝒯 ∪ {(S_i, D)}
11: return 𝒯
```

Algorithm 6.9 IntersectAlgebraic(p, T, x_i, S, C)

```
1: 𝒯 := ∅
2: for (g, D) ∈ RegularGcd(p, T_{x_i}, x_i, S, C) do
3:     if dim(D) < dim(C) then
4:         for E ∈ CleanChain(D, T, x_i) do
5:             𝒯 := 𝒯 ∪ IntersectAlgebraic(p, T, x_i, S, E)
6:     else
7:         𝒯 := 𝒯 ∪ {D ∪ g}
8:         for E ∈ Intersect(init(g), D) do
9:             for F ∈ CleanChain(E, T, x_i) do
10:                𝒯 := 𝒯 ∪ IntersectAlgebraic(p, T, x_i, S, F)
11: return 𝒯
```

Algorithm 6.10 IntersectFree(p, x_i, C)

$\mathcal{T} := \varnothing$
for $(f, D) \in$ Regularize(init$(p), C)$ **do**
  if $f = 0$ **then**
    $\mathcal{T} := \mathcal{T} \cup$ Intersect(tail$(p), D)$
  else
    $\mathcal{T} := \mathcal{T} \cup \{D \cup p\}$
    for $E \in$ Intersect(init$(p), D)$ **do**
      $\mathcal{T} := \mathcal{T} \cup$ Intersect(tail$(p), E)$
return $\mathcal{T}$

Algorithm 6.11 Extend(C, T, x_i)

if $x_i \notin \text{mvar}(T)$ **and** $T^+_{x_i} = \varnothing$ **then return** $\{C\}$
$\mathcal{T} := \varnothing$
Choose $p \in T$ with greatest main variable; $T' := T \setminus \{p\}$
for $D \in$ Extend(C, T', x_i) **do**
  for $(f, E) \in$ Regularize(init$(p), D)$ **do**
    if $f \neq 0$ **then** $\mathcal{T} := \mathcal{T} \cup \{E \cup p\}$
return $\mathcal{T}$

Algorithm 6.12 CleanChain(C, T, x_i)

if $x_i \notin \text{mvar}(T)$ **or** $\dim(C) = \dim(T^-_{x_i})$ **then return** $\{C\}$
$\mathcal{T} := \varnothing$
for $(f, D) \in$ Regularize(init$(T_{x_i}), C)$ **do**
  if $f \neq 0$ **then**
    $\mathcal{T} := \mathcal{T} \cup D$
return $\mathcal{T}$

Algorithm 6.13 Regularize(p, T)

```
1: if p ∈ 𝕂 or T = ∅ then return {(p, T)}
2: v := mvar(p); 𝒯 := ∅
3: if v ∉ mvar(T) then
4:     for (f, C) ∈ Regularize(init(p), T) do
5:         if f = 0 then
6:             𝒯 := 𝒯 ∪ Regularize(tail(p), C)
7:         else
8:             𝒯 := 𝒯 ∪ {(p, C)}
9:     return 𝒯
10: src := Subresultant(p, T_v, v); r := S_0 of src, the subresultant of index 0
11: for (f, C) ∈ Regularize(r, T_v^-) do
12:     if dim(C) < dim(T_v^-) then
13:         for D ∈ Extend(C, T, v) do
14:             𝒯 := 𝒯 ∪ Regularize(p, D)
15:     else if f ≠ 0 then
16:         𝒯 := 𝒯 ∪ {p, C ∪ T_v ∪ T_v^+}
17:     else
18:         for (g, D) ∈ RegularGcd(p, T_v, v, src, C) do
19:             if dim(D) < dim(C) then
20:                 for E ∈ Extend(D, T, v) do
21:                     𝒯 := 𝒯 ∪ Regularize(p, E)
22:                 continue
23:             if mdeg(g) = mdeg(T_v) then
24:                 𝒯 := 𝒯 ∪ {(0, D ∪ T_v ∪ T_v^+)}
25:             else
26:                 𝒯 := 𝒯 ∪ {(0, D ∪ g ∪ T_v^+)}
27:                 q := pquo(T_v, g)
28:                 𝒯 := 𝒯 ∪ Regularize(p, D ∪ q ∪ T_v^+)
29:                 for E ∈ Intersect(init(g), D) do
30:                     for F ∈ Extend(E, T, v) do
31:                         𝒯 := 𝒯 ∪ Regularize(p, F)
32: return 𝒯
```

Algorithm 6.14 RemoveRedundantComponents($\mathcal{T}$)

1: **if** $|\mathcal{T}| \leq 1$ **then return** $\mathcal{T}$
2: $\ell := \lceil |\mathcal{T}|/2 \rceil$; $\mathcal{T}_{\leq \ell} :=$ first ℓ elements of $\mathcal{T}$;
3: $\mathcal{T}_{>\ell} := \mathcal{T} \setminus \mathcal{T}_{\leq \ell}$
4: $\mathcal{T}_1 :=$ RemoveRedundantComponents($\mathcal{T}_{\leq \ell}$)
5: $\mathcal{T}_2 :=$ RemoveRedundantComponents($\mathcal{T}_{>\ell}$)
6: $\mathcal{T}_1' := \varnothing$; $\mathcal{T}_2' := \varnothing$
7: **for** T_1 **in** $\mathcal{T}_1$ **do**
8: | **if** $\forall T_2$ **in** $\mathcal{T}_2$ IsNotIncluded (T_1, T_2) **then**
9: | | $\mathcal{T}_1' := \mathcal{T}_1' \cup \{T_1\}$
10: **for** T_2 **in** $\mathcal{T}_2$ **do**
11: | **if** $\forall T_1$ **in** $\mathcal{T}_1'$ IsNotIncluded (T_2, T_1) **then**
12: | | $\mathcal{T}_2' := \mathcal{T}_2' \cup \{T_2\}$
13: **return** $\mathcal{T}_1' \cup \mathcal{T}_2'$

6.2 Computing Subresultants Speculatively

We saw in the previous section that RegularGCD computes a regular GCD of two polynomials by searching through their subresultants S_0, then S_1, etc., until a subresultant with a regular initial is found. Such a subresultant is equal to the regular GCD of the two polynomials, as shown by Theorem 6.2. Thanks to the *Shape lemma*, it is often the case that S_1 will be equal to the regular GCD. This suggests to avoid computing an entire subresultant chain, and rather compute only the resultant S_0 and the subresultant of index 1, S_1. We call this design the *speculative* computation of subresultants, where one *speculates* that the GCD will be S_1 and thus avoids explicitly computing the subresultants of higher index. As we will explore in this section, this computation is possible thanks to the *Half-GCD* algorithm. Refer to the notations and definitions of subresultants presented in Section 2.4.

Consider two non-zero univariate polynomials $a, b \in \mathbb{K}[y]$ with $n_0 := \deg(a)$, $n_1 := \deg(b)$ with $n_0 \geq n_1$. The extended Euclidean algorithm (EEA) computes the successive remainders $(r_0 := a, r_1 := b, r_2, \ldots, r_\ell = \gcd(a, b))$ with degree sequence $(n_0, n_1, n_2 := \deg(r_2) \ldots, n_\ell := \deg(r_\ell))$ and the corresponding quotients $(q_1, q_2, \ldots, q_\ell)$ defined by $r_{i+1} = \text{rem}(r_i, r_{i-1}) = r_{i-1} - q_i r_i$, for $1 \leq i \leq \ell$, $q_i = \text{quo}(r_i, r_{i-1})$ for $1 \leq i \leq \ell$, $n_{i+1} < n_i$ for $1 \leq i < \ell$, and $r_{\ell+1} = 0$ with $\deg(r_{l+1}) = -\infty$. This computation requires $O(n^2)$ operations in $\mathbb{K}$. We denote by Q_i the *quotient matrices* defined for $1 \leq i \leq \ell$, by

$Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}$, so that, for $1 \leq i < \ell$, we have

$$\begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix} = Q_i \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix} = Q_i \dots Q_1 \begin{bmatrix} r_0 \\ r_1 \end{bmatrix}. \tag{6.1}$$

We define $m_i := \deg(q_i)$, so that we have $m_i = n_{i-1} - n_i$ for $1 \leq i \leq \ell$. The degree sequence $(n_0, \dots, n_l)$ is said to be *normal* if $n_{i+1} = n_i - 1$ holds, for $1 \leq i < \ell$, or, equivalently if $\deg(q_i) = 1$ holds, for $1 \leq i \leq \ell$.

Using the remainder and degree sequences of non-zero polynomials $a, b \in \mathbb{K}[y]$, Proposition 6.5, known as the *fundamental theorem on subresultants*, introduces a procedure to compute the nominal leading coefficients of polynomials in the subresultant chain.

Proposition 6.5. For $k = 0, \dots, n_1$, the nominal leading coefficient of the k-th subresultant of (a, b) is either 0 or, if there exists $i \leq \ell$ such that $k = \deg(r_i)$, then it is equal to s_k and is given by:

$$s_k = (-1)^{\tau_i} \prod_{1 \leq j < i} \mathrm{lc}(r_j)^{n_{j-1} - n_{j+1}} \mathrm{lc}(r_i)^{n_{i-1} - n_i},$$

where $\tau_i = \sum_{1 \leq j < i} (n_{j-1} - n_i)(n_j - n_i)$

Proof. [86, Theorem 11.16]. □

The *Half-GCD* algorithm, also known as the *fast extended Euclidean algorithm*, is a divide-and-conquer algorithm for computing a single pair of remainders from the EEA sequence. For example, if wishing to compute the GCD, this can be interpreted as the computation of a 2×2 matrix Q over $\mathbb{K}[y]$ so that we have:

$$\begin{bmatrix} \gcd(a, b) \\ 0 \end{bmatrix} = Q \begin{bmatrix} a \\ b \end{bmatrix}.$$

The major difference between the classical EEA and the Half-GCD algorithm is that, while the EEA computes all the remainders $r_0, r_1, \dots, r_\ell = \gcd(r_0, r_1)$, the Half-GCD computes only two consecutive remainders. These two remainders are derived from the Q_i quotient matrices, which are, in turn, obtained from a sequence of "truncated remainders", instead of the complete r_i remainders.

This computation runs within $O(M(n) \log n)$ operations in $\mathbb{K}$, where $M(n)$ is a multiplication time, as defined in Chapter 8 of [86]. This *Half-GCD* originated in the ideas of [123], [114] and [159]. Yet, one of the earliest correct implementation came only 20 years later in [169].

We follow the presentation of the Half-GCD algorithm from [86, Chapter 11]. For a non-negative $k \leq n_0$, this algorithm computes the quotients $q_1, \ldots, q_{h_k}$ where h_k is defined as

$$h_k = \max\Big\{0 \leq j \leq \ell \mid \sum_{i=1}^{j} m_i \leq k\Big\}, \tag{6.2}$$

the maximum $j \in \mathbb{N}$ so that $\sum_{1 \leq i \leq j} \deg(q_i) \leq k$. This computation runs within $(22M(k) + O(k)) \log k$ operations in $\mathbb{K}$. From Equation 6.2, $h_k \leq \min(k, \ell)$, and

$$\sum_{i=1}^{h_k} m_i = \sum_{i=1}^{h_k} (n_{i-1} - n_i) = n_0 - n_{h_k} \leq k < \sum_{i=1}^{h_k+1} m_i = n_0 - n_{h_k+1}. \tag{6.3}$$

Thus, $n_{h_k+1} < n_0 - k \leq n_{h_k}$, and h_k is uniquely determined; see [86, Algorithm 11.6].

Due to the deep relation between subresultants and the remainders of the EEA, the Half-GCD technique can support computing subresultants. This approach is studied in [86]. Therein, the Half-GCD algorithm is used to compute the nominal leading coefficient of subresultants up to s_ρ for $\rho = n_{h_k}$. The method proceeds by first computing the quotients $q_1, \ldots, q_{h_k}$, then calculating the $\mathrm{lc}(r_i) = \mathrm{lc}(r_{i-1})/\mathrm{lc}(q_i)$ from $\mathrm{lc}(r_0)$ for $1 \leq i \leq h_k$, and finally applying Proposition 6.5. The resulting procedure runs within the same complexity as the Half-GCD algorithm itself.

However, for the purpose of computing two successive subresultants, rather than just their leading coefficient, more is needed. Consider computing the pair of subresultants $S_{n_v}, S_{n_{v+1}}$ given $0 \leq \rho < n_1$, for $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$. This requires computing the quotients $q_1, \ldots, q_{h_\rho}$ where h_ρ is defined as

$$h_\rho = \max\Big\{0 \leq j < \ell \mid n_j > \rho\Big\}, \tag{6.4}$$

using Half-GCD. Let $k = n_0 - \rho$, Equations 6.3 and 6.4 deduce $n_{h_\rho+1} \leq n_0 - k < n_{h_\rho}$, and $h_\rho \leq h_k$. So, to compute the array of quotients $q_1, \ldots, q_{h_\rho}$, we can use an adaptation of the Half-GCD algorithm of [86]. Algorithm 6.15 is this adaptation and runs within the same complexity as their algorithm.

Algorithm 6.15 receives as input two polynomials $r_0 := a, r_1 := b$ in $\mathbb{K}[y]$, with $n_0 \geq n_1$, $0 \leq k \in \mathbb{N}$, $\rho \leq n_0$ where ρ, by default, is $n_0 - k$. The algorithm also receives the array $\mathcal{A}$ of the leading coefficients of the remainders which have been computed so far. This array should be initialized to size $n_0 + 1$ with $\mathcal{A}[n_0] = \mathrm{lc}(r_0)$ and $\mathcal{A}[i] = 0$ for $0 \leq i < n_0$. $\mathcal{A}$ is updated (and therefore values returned) in-place by the algorithm and its recursive calls. The algorithm returns the array of quotients $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ and matrix $M := Q_{h_\rho} \cdots Q_1$.

Algorithm 6.15 AdaptedHGCD$(r_0, r_1, k, \rho, \mathcal{A})$

Input: $r_0, r_1 \in \mathbb{K}[y]$ with $n_0 = \deg(r_0) \geq n_1 = \deg(r_1)$, $0 \leq k \leq n_0$, $0 \leq \rho \leq n_0$ is an upper bound for the degree of the last computed remainder that, by default, is $n_0 - k$ and is fixed in recursive calls (See Algorithm 6.16), the array $\mathcal{A}$ of the leading coefficients of the remainders (in the Euclidean sequence) which have been computed so far

Output: $h_\rho \in \mathbb{N}$ so that $h_\rho = \max(\{)j \mid n_j > \rho\}$, the array $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ of the first h_ρ quotients associated with remainders in the Euclidean sequence and the matrix $M := Q_{h_\rho} \cdots Q_1$; the array $\mathcal{A}$ of leading coefficients is updated in-place

1: **if** $r_1 = 0$ **or** $\rho \geq n_1$ **then return** $\left(0, (), \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$

2: **if** $k = 0$ **and** $n_0 = n_1$ **then**

3: $\quad$ **return** $\left(1, (\mathrm{lc}(r_0)/\mathrm{lc}(r_1)), \begin{bmatrix} 0 & 1 \\ 1 & -\mathrm{lc}(r_0)/\mathrm{lc}(r_1) \end{bmatrix}\right)$

4: $m_1 := \lceil \frac{k}{2} \rceil$; $\delta_1 := \max(\deg(r_0) - 2\ (m_1 - 1), 0)$; $\lambda := \max(\deg(r_0) - 2k, 0)$

5: $\left(h', (q_1, \ldots, q_{h'}), R\right) := \text{AdaptedHGCD}(\mathrm{quo}(r_0, y^{\delta_1}), \mathrm{quo}(r_1, y^{\delta_1}), m_1 - 1, \rho, \mathcal{A})$

6: $\begin{bmatrix} c \\ d \end{bmatrix} := R \begin{bmatrix} \mathrm{quo}(r_0, y^\lambda) \\ \mathrm{quo}(r_1, y^\lambda) \end{bmatrix}$ where $R := \begin{bmatrix} R_{00} & R_{01} \\ R_{10} & R_{11} \end{bmatrix}$

7: $m_2 := \deg(c) + \deg(R_{11}) - k$

8: **if** $d = 0$ **or** $m_2 > \deg(d)$ **then return** $\left(h', (q_1, \ldots, q_{h'}), R\right)$

9: $r := \mathrm{rem}(c, d)$; $q := \mathrm{quo}(c, d)$; $Q := \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix}$

10: $n_{h'+1} := n_{h'} - \deg(q)$

11: **if** $n_{h'+1} \leq \rho$ **then return** $\left(h', (q_1, \ldots, q_{h'}, q), R\right)$

12: $\mathcal{A}[n_{h'+1}] := \mathcal{A}[n_{h'}]/\mathrm{lc}(q)$

13: $\delta_2 := \max(2m_2 - \deg(d), 0)$

14: $\left(h^*, (q_{h'+2}, \ldots, q_{h'+h^*+1}), S\right) :=$
AdaptedHGCD$(\mathrm{quo}(d, y^{\delta_2}), \mathrm{quo}(r, y^{\delta_2}), \deg(d) - m_2, \rho, \mathcal{A})$

15: **return** $\left(h_\rho := h' + h^* + 1, \mathcal{Q} := (q_1, \ldots, q_{h_\rho}), M := SQR\right)$

Algorithm 6.15 and *the fundamental theorem on subresultants* yield Algorithm 6.16. This algorithm is a *speculative* subresultant algorithm based on Half-GCD to calculate two successive subresultants without computing others in the chain. Moreover, this algorithm returns intermediate data that has been computed by the Half-GCD algorithm—the array $\mathcal{R}$ of the remainders, the array $\mathcal{Q}$ of the quotients and the array $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence—to later calculate higher subresultants in the chain without calling Half-GCD again. This *caching* scheme is shown in Algorithm 6.17.

Algorithm 6.16 SPECULATIVESUBRESULTANT(a, b, ρ)

Input: $a, b \in \mathbb{K}[x] \setminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$

Output: Subresultants $S_{n_v}(a, b)$, $S_{n_{v+1}}(a, b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$, the array $\mathcal{Q}$ of the quotients, the array $\mathcal{R}$ of the remainders, and the array $\mathcal{A}$ of the leading coefficients of the remainders (in the Euclidean sequence) that have been computed so far

1: $\mathcal{A} := (0, \ldots, 0, \mathrm{lc}(a))$ where $\mathcal{A}[n_0] = \mathrm{lc}(a)$ and $\mathcal{A}[i] = 0$ for $0 \leq i < n_0$
2: **if** $\rho \geq n_1$ **then**
3: | $\mathcal{A}[n_1] = \mathrm{lc}(b)$
4: | **return** $\Big((a, \mathrm{lc}(b)^{m-n-1}b), (), (), \mathcal{A}\Big)$
5: $(v, \mathcal{Q}, M) :=$ ADAPTEDHGCD$(a, b, n_0 - \rho, \rho, \mathcal{A})$
6: *deduce* $\Big(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_v = \deg(r_v)\Big)$ *from* a, b and $\mathcal{Q}$.
7: $\begin{bmatrix} r_v \\ r_{v+1} \end{bmatrix} := M \begin{bmatrix} a \\ b \end{bmatrix}$; $\mathcal{R} := (r_v, r_{v+1})$; $n_{v+1} := \deg(r_{v+1})$
8: $\tau_v := 0$; $\tau_{v+1} := 0$; $\alpha := 1$
9: **for** j **from** 1 **to** $v - 1$ **do**
10: | $\tau_v := \tau_v + (n_{j-1} - n_v)(n_j - n_v)$
11: | $\tau_{v+1} := \tau_{v+1} + (n_{j-1} - n_{v+1})(n_j - n_{v+1})$
12: | $\alpha := \alpha\ \mathcal{A}[n_j]^{n_{j-1}-n_{j+1}}$
13: $\tau_{v+1} := \tau_{v+1} + (n_{v-1} - n_{v+1})(n_v - n_{v+1})$
14: $S_{n_v} := (-1)^{\tau_v}\alpha\ r_v$
15: $S_{n_{v+1}} := (-1)^{\tau_{v+1}}\alpha\ \mathcal{A}[n_v]^{n_{v-1}-n_{v+1}}\ r_{v+1}$
16: **return** $\Big((S_{n_v}, S_{n_{v+1}}), \mathcal{Q}, \mathcal{R}, \mathcal{A}\Big)$

Let us explain this technique with an example. For non-zero polynomials $a, b \in \mathbb{K}[y]$ with $n_0 = \deg(a)$, $n_1 = \deg(b)$, so that we have $n_0 \geq n_1$. The function call SUBRESULTANT$(a, b, 0)$ returns $S_0(a, b)$ and $S_1(a, b)$ speculatively, without computing $(S_{n_1}, S_{n_1-1}, S_{n_1-2}, \ldots, S_2)$. This function also returns the arrays $\mathcal{Q} = (q_1, \ldots, q_\ell)$, $\mathcal{R} = (r_\ell, r_{\ell-1})$, and $\mathcal{A}$. These arrays allow us to later compute subresultants of higher indices

Algorithm 6.17 CachingSubresultant$(a, b, \rho, \mathcal{Q}, \mathcal{R}, \mathcal{A})$

Input: $a, b \in \mathbb{K}[x] \setminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$, the list $\mathcal{Q}$ of all the quotients in the Euclidean sequence, the list $\mathcal{R}$ of the remainders that have been computed so far; we assume that $\mathcal{R}$ contains at least $r_\mu, \ldots r_{\ell-1}, r_\ell$ with $0 \leq \mu \leq \ell - 1$, and the list $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence

Output: Subresultants $S_{n_v}(a, b)$, $S_{n_{v+1}}(a, b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$; the list $\mathcal{R}$ of the remainders is updated in-place

1: *deduce* $\Big(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_\ell = \deg(r_\ell)\Big)$ *from* a, b and $\mathcal{Q}$
2: **if** $n_\ell \leq \rho$ **then** $v := \ell$
3: **else** *find* $0 \leq v < \ell$ *such that* $n_{v+1} \leq \rho < n_v$.
4: **if** $v = 0$ **then**
5: | **return** $\Big(a, \mathrm{lc}(b)^{m-n-1} b\Big)$
6: **for** i **from** $\max(v, \mu + 1)$ **down to** v **do**
7: | $r_i := r_{i+1} q_{i+1} + r_{i+2}$; $\mathcal{R} := \mathcal{R} \cup (r_i)$
8: *compute* $S_{n_v}, S_{n_{v+1}}$ *using Proposition* 6.5 *from* r_v, r_{v+1}
9: **return** $\Big(S_{n_v}, S_{n_{v+1}}\Big)$

instead of calling Half-GCD again. This situation occurs, for example, when the regular GCD of two polynomials in RegularGCD has main degree greater than 1. In a bottom-up approach to computing regular GCDs, one starts by computing the subresultants S_0 and S_1. If neither has a regular leading coefficient, then one can compute, speculatively, S_2 and S_3; see Algorithm 6.3.

For polynomials $a, b \in \mathbb{Z}[y]$, it is simple to create a modular algorithm to compute subresultants speculatively (or the entire subresultant chain). One can work over the finite field $\mathbb{Z}_p$ for some prime number p to apply Algorithm 6.16. Then, with sufficiently many primes and applying the *Chinese remainder theorem* (see Section 2.1.3), we can reconstruct the subresultants over $\mathbb{Z}[y]$. In this modular method, it is useful to use an iterative and probabilistic approach to the reconstruction; see [137]. We calculate subresultants modulo different primes $p_0, p_1, \ldots$, one at a time and iteratively, continuing to add modular images to the reconstruction until it *stabilizes*. That is to say, the reconstruction does not change when we add another image to the CRT direct product (Equation 2.1).

For polynomials $a, b \in \mathbb{Z}[x, y]$ a similar approach can be used. First, one reduces to $\mathbb{Z}_p[x, y]$ by working modulo some prime number p. Then, we proceed by evaluation-interpolation. We choose a set of evaluation points of size N and evaluate each coefficient of the polynomials in $(\mathbb{Z}_p[x])[y]$. Then, we compute (speculative) subresultant images over $\mathbb{Z}_p[y]$. By interpolating back x in each coefficient of each subresultant, we reconstruct

the subresultants in $\mathbb{Z}_p[x, y]$. Finally, we can repeat the entire process with a new prime number p' and proceed with iterative CRT reconstruction to reconstruct the subresultants over $\mathbb{Z}[x, y]$. The number of evaluation points is determined from an upper-bound on the degree of subresultants and resultants with respect to x. From [86, Theorem 6.22], the following inequality holds: $N \geq \deg(b, y)\deg(a, x) + \deg(a, y)\deg(b, x) + 1$. An Algorithm implementing these ideas is presented later in Section 6.3.3 as Algorithm 6.23.

In order to use the idea of speculative subresultants transparently within regular GCD computations, we have developed an additional `SubresultantChain` class. This class in constructed using the two polynomials whose subresultant chain is to be computed. This class provides an interface which allows the subresultant chain to be computed both speculatively and *lazily*. Indeed, an object of the class is constructed instantly without computing anything. The lazy nature arises where subresultants are computed only as explicitly requested. When a subresultant of a particular index is requested, it is returned (if already computed) and otherwise computed speculatively. The class's interface hides all of this lazy computation; see Listing 6.1. Notice that `SubresultantChain` is actually a class template templated by the polynomial type. The only assumption is that the polynomial type may be viewed "recursively" as a univariate polynomial with (possibly) polynomial coefficients. Indeed, we have a `RecursivelyViewedPolynomial` abstract class which defines a suitable interface and fits within our algebraic class hierarchy (see Section 4.2).

With the `SubresultantChain` class, it is easy to see how RegularGCD may be modified to take advantage of speculative computation of subresultants. Take the special case of RegularGCD shown in Algorithm 6.3. On Line 1 we create a subresultant chain S. This does not actually compute anything. Consider the first iteration of the **while** loop. We have $i = 0$. The access to the subresultant coefficient (i.e. the resultant) on Line 5 accesses $\mathrm{lc}(S_0, v)$. This triggers the speculative computation of S_0 and S_1, which are then both stored in the `SubresultantChain` class. On the next iteration on the **while** loop, $i = 1$ and S_1 has already been computed from the previous loop. If a regular GCD has still not been found, the next iteration of the loop has $i = 2$ and the subresultants S_2 and S_3 are computed speculatively. This process continues as required, and avoids unnecessarily computing the majority of the subresultants.

```
template <class RecursivePoly>
class SubresultantChain {
    //Construct the subreuslsltant chain between P and Q w.r.t. variable v.
    SubresultantChain(RecursivePoly P, RecursivePoly Q, Symbol v);

    //Get the subresultant S_k of index k.
    RecursivePoly subresultantOfIndex(int k);

    //Get the principle leading coefficient s_k of index k.
    //This will return zero if the degree of S_k is less than k.
    RecursivePoly principalSubresultantCoefficientOfIndex(int k);

    //Get the initial of the subresulant S_k of index k.
    RecurisvePoly subresultantInitialOfIndex(int k);

    //Get the resultant of P and Q viewed with v as main variable.
    RecursivePoly resultant();
}
```

Listing 6.1: The `SubresultantChain` class interface.

6.3 Concurrency Opportunities

In this section, we highlight the opportunities for concurrent execution offered by the algorithms for computing triangular decompositions presented in 6.1. To do so, we review the key ideas underlying those algorithms and show how concurrency can be exposed. Each of these concurrency opportunities follows one or more of the patterns described in Section 5.1.

We start with the top-level Triangularize procedure in Section 6.3.1, describing how its organization can invoke either the map pattern or, after a small transformation, the workpile pattern. The core subroutines of Triangularize are examined in Section 6.3.2 where asynchronous generators and the pipeline pattern are employed. For the critical operation of computing subresultants, Section 6.3.3 illustrates how the map pattern may be used to gain parallelism through modular computations of subresultants. Finally, in Section 6.3.4, we examine the application of both the fork-join pattern and the map pattern to the removal of redundant components.

6.3.1 Map, Workpile, and the TRIANGULARIZE Procedure

A serial version of the TRIANGULARIZE procedure was shown in Algorithm 6.1. That method proceeded incrementally, intersecting each polynomial in the input set with the current collection of partial solutions (i.e. regular chains). Now, in Algorithm 6.18, we formulate TRIANGULARIZE to make use of the parallel map to compute the embarrassingly parallel inner **for** loop. Indeed, *component-level parallelism* is possible where one can intersect a polynomial with each intermediate regular chain in parallel.

Algorithm 6.18 TRIANGULARIZE(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
1: 𝒯 := {∅}
2: for p ∈ F do
3: |   𝒯' := ∅
4: |   parallel_for T ∈ 𝒯 do
5: |   |   𝒯' := 𝒯' ∪ INTERSECT(p, T)
6: |   𝒯 := REMOVEREDUNDANTCOMPONENTS(𝒯')
7: return 𝒯
```

It follows from Algorithm 6.18 that whenever INTERSECT(p, T) returns more than one regular chain, there is an opportunity for concurrent execution. Consider again the idea that the collection of regular chains created by TRIANGULARIZE and INTERSECT form a tree (as discussed in Section 6.1). Algorithm 6.18 is essentially a breadth-first search over this tree and, hence, can be considered to be "Triangularize by Level", where a level consists of all nodes a particular distance from the root. Since each branch is independent, the next step of the breadth-first search can be performed over each branch concurrently, that is, the intersects at one level can be performed simultaneously. One can see Lines 4–5 as a map step where INTERSECT maps each current regular chain.

The parallel application of INTERSECT over the current list of regular chains $\mathcal{T}$ can be seen as coarse-grained parallelism as each call to INTERSECT represents substantial work. However, it is also a form of irregular parallelism for two reasons. First, the source of parallelism here is based on the geometry of the algebraic set $V(F)$ and its decomposition into multiple components; hence, it is component-based parallelism. This is not inherently bad, but it does imply that the amount of parallelism cannot be known *a priori*. Second, each such call to INTERSECT may represent an unbalanced amount of work where, for example, one component is much more simple than the others. This further reduces the amount of parallelism which can be exploited by the map step. Recall

from Section 5.1.1 that a sequence of map steps requires threads to operate in lockstep, synchronizing at the end of each map step. Thus, if the intersections during a particular map step are unbalanced, then the algorithm must wait for the slowest intersection to finish before beginning the next map step, greatly reducing parallelism.

Further synchronization is also needed where one removes redundant components after every step. Compare this to Algorithm **??** where redundant components are removed only once at the end. Nonetheless, removing redundant components intermittently can be very useful for performance as entire branches of the search tree can be pruned. We discuss some experimentation showing this momentarily.

Algorithm 6.19 TriangularizeByTasks(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
1: Tasks := { (F, ∅) };  T := {}
2: while |Tasks| > 0 do
3: |   (P, T) := pop a task from Tasks
4: |   Choose a polynomial p ∈ P; P' := P \ {p}
5: |   for T' in Intersect(p, T) do
6: |   |   if |P'| = 0 then T := T ∪ {T'}
7: |   |   else Tasks := Tasks ∪ {(P', T')}
8: return RemoveRedundantComponents(T)
```

To avoid the unbalanced map steps and multiple synchronization points, we wish to reorganize the Triangularize algorithm in order to employ the workpile pattern (Section 5.1.2). This should improve load-balancing. We consider this organization of Triangularize to be "by tasks"; see Algorithm 6.19. In this reorganization, we invert the nested loops of Triangularize to first iterate over the current collection of regular chains and then iterate over polynomials in the input system. Since the former is actually of a variable and unknown size, this is achieved by creating tasks. Note that this is equivalent to the recursive Triangularize procedure shown as Algorithm 6.6, where the recursion has been unwound.

A triangualrize task encompasses a single regular chain along with a list of polynomials which remain to be intersected with that chain. Therefore, any splitting found by an intersection creates new tasks with different regular chains but with the same remaining list of polynomials. Whether or not a splitting occurs, tasks are continually added back to the workpile until the list of polynomials in the task becomes empty. Once this list is empty, that component is considered fully solved and is added to the list of results.

In this new organization, the potential parallelism is greater due to the favourable work-balancing made possible via the workpile pattern. However, this organization loses the ability to remove redundant components after each intersection, thus possibly performing unnecessary work. On the other hand, this organization allows each independent list of polynomials to be simplified and reordered with respect to each regular chain (this subtlety is hidden behind choosing a polynomial in Line 4 of Algorithm 6.19; see the discussion of CLEANSET in Section 6.1). The order in which polynomials are intersected can potentially lead to large savings in the overall amount of computational work, as described earlier in Section 6.1.

Both this increased parallelism and decreased computational work was discussed in [12]. Here, let us present some additional quantitative measures to compare the two organizations. Let us count the number of times TRIANGULARIZE and TRIANGULARIZEBYTASKS each call INTERSECT to compute a triangular decomposition of the input system. This represents the number of individual intersections computed between a polynomial and regular chain. On average, TRIANGULARIZEBYTASKS is much more efficient. Again, this can be attributed to the ordering of the intersections and the simplifications of the list of polynomials with respect to each independent regular chain. In our test suite of nearly 3000 polynomial systems (see Section 6.4), when computing a Kalkbrener decomposition (resp. a Lazard-Wu decomposition) TRIANGULARIZE calls INTERSECT an average of 19.85 (resp. 20.05) times, while TRIANGULARIZEBYTASKS calls INTERSECT an average of 10.24 (resp. 10.53) times. Overall, TRIANGULARIZEBYTASKS achieves better performance in execution time and parallel speed-up. Nonetheless, removing redundancies can lead to up to a 70× improvement in performance, as seen by Sys2874.

However, *TriangularizeByTasks*, on average, achieves much higher parallel speed-up. It is left to future work to determine a hybrid approach which allows for intermediately removing redundant components while retaining the amount of parallelism available in TRIANGULARIZEBYTASKS. We discuss this idea further in Chapter 8.

For the remainder of this section, we consider only TRIANGULARIZEBYTASKS for its generally higher performance and much higher parallelism. This performance is discussed in detail in Section 6.4. For simplicity, any following reference to TRIANGULARIZE implies the modified version TRIANGULARIZEBYTASKS presented in Algorithm 6.19.

			Kalkbrener				Lazard			
System	Task Ints.	Level Ints.	Task Time (s)	Task Speed-up	Level Time (s)	Level Speed-up	Task Time (s)	Task Speed-up	Level Time (s)	Level Speed-up
Sys2875	184	71	2.44	6.23	0.12	0.95	2.44	6.23	0.12	0.90
8-3-config-Li	231	146	2.49	4.70	1.92	3.23	9.63	4.52	7.05	3.11
Sys2128	329	206	3.37	7.91	0.41	1.69	3.29	7.75	0.41	1.70
Sys2881	169	106	3.60	5.57	0.47	1.24	3.60	5.57	0.48	1.18
Sys2885	259	73	3.70	7.82	0.53	2.05	3.69	8.48	0.54	2.09
Sys2161	666	287	8.80	7.91	1.70	3.06	8.67	7.85	1.71	3.16
W44	448	72	10.14	8.61	0.89	1.81	10.67	8.67	0.90	1.87
Sys2449	835	299	10.54	8.47	1.11	2.58	10.85	8.84	1.12	2.53
Sys2882	477	286	12.50	5.29	12.15	3.68	16.69	6.06	12.61	3.11
dgp6	979	214	29.04	8.49	7.63	2.03	37.38	10.27	6.89	1.72
Sys2880	4126	871	56.57	10.10	4.14	3.29	57.37	10.47	3.87	3.07
Sys2874	2936	154	70.43	10.22	0.94	1.49	70.93	10.17	0.93	1.47

Table 6.1: Counting the number of calls to Intersect by TriangularizeByTasks (Algorithm 6.19; "Task Ints.") and by Triangularize (Algorithm 6.18; "Level Ints") for examples where the latter is more efficient. Number of intersects using either algorithm is the same whether solving in Kalkbrener mode or Lazard mode, except for 8-3-config-Li (374 Lazard Task, 191 Lazard Level) and Sys2882 (538 Lazard Task, 291 Lazard Level). The resulting serial execution times for solving in both Kalkbrener mode and Lazard mode for each algorithm is shown ("Task Time", "Level Time"). The parallel speed-up factor on a machine with 12 cores is also shown for an execution with all parallel schemes active except asynchronous generators (see Section 6.4 details on the machine and parallel schemes).

6.3.2 Asynchronous Generators with Intersect, RegularGCD and Regularize

We now turn our attention to parallel opportunities in the core subroutines of Triangularize. As seen in Section 6.1 and Figure 6.1, these subroutines are highly interdependent and mutually recursive. Moreover, they all return a list of regular chains (or pairs of polynomials and regular chains). Whenever more than one regular chain is returned, this suggests an opportunity for concurrency. In this section we examine how modelling the subroutines of Triangularize as generators leads to concurrency opportunities.

Intersect

Let $p \in \mathbb{K}[\underline{X}]$ and $T \subseteq \mathbb{K}[\underline{X}]$ be a regular chain. The operation Intersect(p, T) is quite complicated in general, as we saw in Algorithms 6.7, 6.9, 6.10. Yet, for the purpose of discussing concurrency opportunities, it is sufficient to consider the most common scenario; call this IntersectTypical. The case where $p \in \mathbb{K}$ is easy to treat. Hence, we assume $p \notin \mathbb{K}$. By calling the algorithm Regularize, one can reduce to the case where init(p) is regular w.r.t. sat(T), hence we assume that this property holds. Let $v = \text{mvar}(p)$. Proceeding by induction on the number n of variables, one can also

reduce to the case where T_v^+ is empty. Algorithm 6.20 implements this case, which follows essentially from the application of Proposition 2.37 in Section 2.5 together with a reasoning by induction on $\dim(T_v^-)$.

Note that Algorithm 6.20 is a *generator function*. Recall this notion from Section 5.1. The keyword **yield** outputs a value to the generator's caller and then resumes execution. In contrast, **return** is used to return a value and terminate the function.

Algorithm 6.20 IntersectTypical(p, T)

Input: $p \in \mathbb{K}[\underline{X}]$, $p \notin \mathbb{K}$, $v := \mathrm{mvar}(p)$, a regular chain $T \subseteq \mathbb{K}[\underline{X}]$ such that $\mathrm{init}(p)$ is regular w.r.t. $\mathrm{sat}(T)$ and $T_v^+ = \varnothing$

Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}$

```
1:  if v ∉ mvar(T) then
2:      yield T ∪ {p}
3:      for S in Intersect(init(p), T) do
4:          for U in Intersect(tail(p), S) do
5:              yield U
6:  else
7:      for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
8:          if dim(T_i) ≠ dim(T_v^-) then
9:              for T_{i,j} ∈ Intersect(p, T_i) do
10:                 yield T_{i,j}
11:         else
12:             if g_i ∉ K and mvar(g_i) = v then
13:                 yield T_i ∪ {g_i}
14:             for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
15:                 for T_{i,j,k} ∈ Intersect(p, T_{i,j}) do
16:                     yield T_{i,j,k}
```

Algorithm 6.20 provides at least two opportunities for concurrency, as seen by several **yield** statements. First, the case $v \notin \mathrm{mvar}(T)$ splits the computation in two sub-cases:

(*i*) $\mathrm{init}(p) \neq 0$: where $T \cup \{p\}$ is returned, and

(*ii*) $Z(\mathrm{init}(p), T) \neq \varnothing$: where recursive calls with $\mathrm{tail}(p)$ are made.

Similarly, the case $v \in \text{mvar}(T)$, for each pair (g_i, T_i) such that $\dim(T_i) = \dim(T_v^-)$ holds, splits the computation in two sub-cases:

(*i*) $g_i \notin \mathbb{K}$ and $\text{mvar}(g_i) = v$: where $T_i \cup \{g_i\}$ is returned, and

(*ii*) $V(\text{lc}(g_i, v)) \cap W(T) \neq \varnothing$: where recursive calls with p are made.

Moreover, observe that the function call RegularGcd(p, T_v, v, T_v^-), when it returns more than one pair, provides additional opportunities for concurrency. Finally, notice that since Intersect is a recursive algorithm, and since we have structured it as a generator function, then Intersect is also a consumer.

RegularGCD

Consider now the RegularGCD operation shown in Algorithm 6.21. First, the subresultant chain S of p and q, regarded as univariate polynomials in v, is computed. This itself can be done via an evaluation-interpolation scheme performed in a parallel fashion, as we will discuss in Section 6.3.3. Then, the RegularGCD algorithm searches for a regular GCD by regularizing the initial of subresultants, beginning at index 0 (the resultant), until a subresultant is found whose initial is regular w.r.t. $\text{sat}(T)$ and returning that subresultant as the regular GCD. This bottom-up search has been described previously in Section 6.1 and 6.2.

By computing Regularize$(\text{lc}(S_i, v), C)$ on Line 5 of Algorithm 6.21, one can always find a regular GCD, up to splitting the regular chain into multiple components. It is this splitting that allows RegularGCD to act as a generator, yielding one regular GCD pair (g_k, T_k) at a time. Moreover, as we will see shortly, having Regularize act as a generator also allows RegularGCD to act as a consumer and thus yield (g_k, T_k) pairs almost immediately as the regular chain T_k is produced from Regularize.

Algorithm 6.21 RegularGcd(p, q, v, T)

Input: $p, q \in \mathbb{K}[\underline{X}]$, $v = \text{mvar}(p) = \text{mvar}(q)$, a regular chain $T \subseteq \mathbb{K}[\underline{X}]$ such that $v' < v \ \forall\, v' \in \text{mvar}(T)$, and $\text{init}(p)$, $\text{init}(q)$ are regular w.r.t. $\text{sat}(T)$

Output: a set of pairs $\{(g_1, T_1), \ldots, (g_e, T_e)\}$ with $g_k \in \mathbb{K}[\underline{X}]$ and $T_k \subset \mathbb{K}[\underline{X}]$ for $1 \leq k \leq e$ such that Relation (R_2) holds and, if $\dim(T_k) = \dim(T)$, then g_k is a regular gcd of p, q w.r.t. T_i

```
1:  if mdeg(p) > mdeg(q) then S := subres(p, q) else  S := subres(q, p)
2:  T := {(T, 0)}
3:  while T ≠ ∅ do
4:  |  Choose a pair (C, i) ∈ T; T := T \ {(C, i)}
5:  |  for D in Regularize(lc(S_i, v), C) do
6:  |  |  if dim(D) < dim(C) then
7:  |  |  |  yield (0, D)
8:  |  |  else if lc(S_i, v) ∈ sat(D) then
9:  |  |  |  T := T ∪ {(D, i + 1)}
10: |  |  else
11: |  |  |  yield (S_i, D)
12: end while
```

Regularize

We now consider Regularize, focusing on the most common scenario as with Intersect. Algorithm 6.22 presents this case, stating the assumptions which follow from Proposition 2.37 in Section 2.5 together with a reasoning by induction on $\dim(T_v^-)$. Just as in the previous two algorithms, Regularize may both be implemented as an asynchronous generator and use generators as it calls Intersect and RegularGCD.

While the previous algorithms have used recursive calls to yield potentially many components, Regularize presents an explicit case where the splitting of a regular chain into multiple can be seen. In Lines 10–12 we see that the regular chain is split by a regular GCD and a pseudo-quotient computation into two cases, one where T_v is replaced by g_i and another where it is replaced by q_i. One should recognize here that the pseudo-quotient computation on Line 11 may be considerable work. Hence, yielding $T_i \cup \{g_i\}$ to the caller before this operation enables concurrent execution with non-trivial amounts of work.

Algorithm 6.22 Regularize(p, T)

Input: $p \in \mathbb{K}[\underline{X}]$, $p \notin \mathbb{K}$, $v := \text{mvar}(p)$, a regular chain $T \subseteq \mathbb{K}[\underline{X}]$ such that init(p) regular w.r.t. sat(T_v^-) and $T_v^+ = \varnothing$

Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that (R_1), (R_2) hold

```
1:  if v ∉ mvar(T) then return T
2:  for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
3:      if dim(T_i) < dim(T_v^-) then
4:          for T_{i,j} ∈ Regularize(p, T_i) do
5:              yield T_{i,j}
6:      else
7:          if g_i ∈ 𝕂 or mvar(g_i) < v or mdeg(g_i) = mdeg(T_v) then
8:              yield T_i
9:          else
10:             yield T_i ∪ {g_i}
11:             q_i := pquo(T_v, g_i, v)
12:             for T_{i,j} ∈ Regularize(p, T_i ∪ {q_i}) do
13:                 yield T_{i,j}
14:             for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
15:                 for T_{i,j,k} ∈ Regularize(p, T_{i,j}) do
16:                     yield T_{i,j,k}
```

Opportunities for Pipeline

The above discussion of Intersect, RegularGCD, and Regularize shows that each of those routines can be implemented as a generator function and, moreover, each is a consumer of several generators. Each top-level call to Intersect thus creates a tree of generator function calls, evolving as the call stack grows with further subroutine calls and shrinks as subroutines complete. This is therefore the pipeline pattern, as discussed in Section 5.1.

More generally, recall the flow graph of Triangularize and its subroutines. We repeat this illustration here in Figure 6.2 for clarity. Among the subroutines, via mutually recursive calls, every subroutine has a path to Regularize. Every routine receives from its subroutine calls—and returns to its caller—a list of regular chains. When this list has size larger than 1, concurrency is possible between any two pairs of subroutines. If we model every subroutine as an asynchronous generator, this creates a dynamic parallel

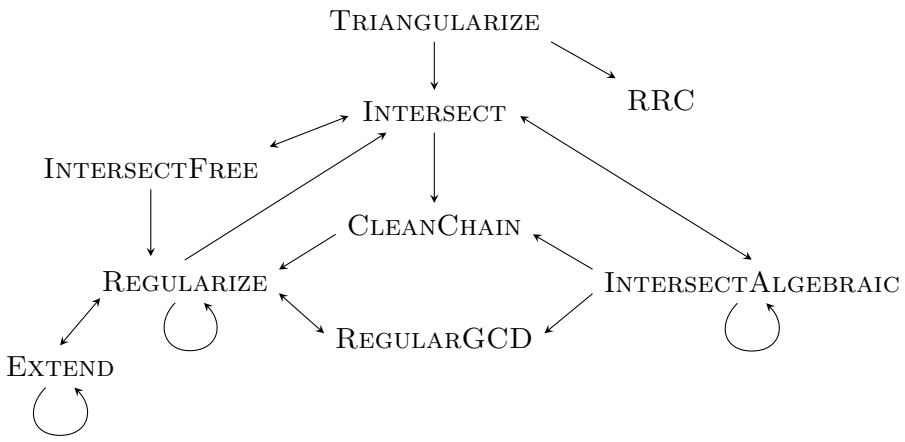

Figure 6.2: A flow graph of function calls within the TRIANGULARIZE algorithm.

pipeline following the call stack. This allows components to *flow* between subroutines as soon as they are discovered.

Indeed, consider the alternative, where none of the routines are implemented as generators and they all return lists of regular chains (as is there specification in Section 6.1.1). Since every routine returns a list, this creates a barrier or synchronization point where each routine must collect all of its components from its subroutines, and then return the entire list at once. At any moment, each routine is processing a single regular chain meanwhile the other regular chains are waiting, idle, to be returned in the output list.

These concurrency opportunities represent more fine-grained parallelism than we saw in TRIANGULARIZE since the amount of work diminishes with each recursive and subroutine call. Indeed, recursion involves a decrease in the number of variables or a decrease in dimension. Further, it is worth noting that the work is likely unbalanced between splittings. For instance, the polynomials g_i and q_i, returned with the regular chain T_i at Lines 10 and 12 of Algorithm 6.22, may have very different degrees since g_i is typically of degree 1 (see Section 6.2). These irregular parallelism challenges are addressed through cooperation between the generators and the coarse-grained parallelism offered by TRIANGULARIZE calling INTERSECT (recall the discussion in Section 6.3.1). This cooperative nature has already been discussed generally in Section 5.4 as part of our parallel support library. We discuss the application of this parallel support specifically to triangular decomposition later in Section 6.3.5

6.3.3 Parallelism in Computing Subresultants

As we have discussed in Section 6.2, the computation of subresultant chains is essential to computing regular GCDs. Consequently, it is a core operation within Triangularize. In practice, the computation of subresultant chains can become a bottleneck when the coefficient sizes and the degrees of the input polynomials become larger and larger. Parallelizing this computation is a way to use more computing resources (in particular cache memories and hardware threads), which can have a significant positive impact on the performance of the client procedures. This parallelization is more fine-grained than the Intersect tasks of Triangularize. Nonetheless, it can sometimes be the most computationally expensive subroutine and thus should not be ignored as a candidate for parallelization. Moreover, balancing workload among threads is very easy in this case.

Recall from Section 6.2 that computing subresultants for univariate and bivariate polynomials can be performed using CRT and evaluation-interpolation schemes. In both cases, one must collect multiple images of the solution (images modulo many prime numbers, or images at many different evaluation points). Computing each image in parallel is an obvious approach. To illustrate how this can be done, consider Algorithm 6.23 which describes a parallel scheme for computing the subresultant chain between two bivariate polynomials in $\mathbb{Z}[x, y]$. This algorithm uses interpolation and then CRT to reconstruct a bivariate solution over the integers from univariate images over a prime field.

The main part of the algorithm begins at Line 3, where NextGoodPrime generates a stream of distinct primes that are good for a, b, that is, not cancelling their leading coefficients. For a given prime p, the parallel for-loop at Lines 5 to 7 collects univariate images of subres($a \bmod p, b \bmod p$) by evaluating x at appropriate values. Parallelism is extracted here by computing the univariate images simultaneously. With the parallel for-loop at Lines 8 to 10, those images are interpolated yielding subres($a \bmod p, b \bmod p$). Here, parallelism is extracted by interpolating each subresultant in the chain independently from the same univariate images. Note that, for every $0 \leq k < \deg(b)$, we interpolate the k-th subresultant of $\bar{a}, \bar{b} \in \mathbb{Z}_p[x, y]$ from the first $r+1$ images of $S_k(\bar{a}\big|_{x=e_j}, \bar{b}\big|_{x=e_j})$ for $0 \leq j \leq r$ where $r = \min(N, N')$, $N = n \deg(a, x) + m \deg(b, x)$, $N' = (m + n - 2k)d$, and $d = \max(\deg(a, x), \deg(b, x))$. Indeed, it is shown in [86] that $\deg(S_k(a, b), x) \leq N'$, and more precisely, $\deg(S_0(a, b), x) \leq N$.

Much like the ideas presented in Section 6.2, the reconstruction of coefficients over the integers can be performed probabilistically. One checks on Line 15 whether or not the solution has changed after adding a new image to the CRT reconstruction. If not, one can return early. This idea, and the upper bound h for the coefficient size is discussed in [137]. Thus, Algorithm 6.23 terminates after finitely many iterations or after stabilization. The

Algorithm 6.23 BIVARIATESRC(a, b)

Input: polynomials $a, b \in \mathbb{Z}[x < y]$ with $m = \text{mdeg}(a)$, $n = \text{mdeg}(b)$, $m \geq n$, $h \in \mathbb{N}$ be a coefficient bound for all subresultants in subres(a, b), and $d = \max(\deg(a, x), \deg(b, x))$

Output: returns subres$(a, b) = \{S_{n-1}(a, b), \ldots, S_0(a, b)\}$ over $\mathbb{Z}[x, y]$

1: $N := n \deg(a, x) + m \deg(b, x)$; $M := 1$; $C^* := [0, \ldots, 0]$
2: **while** $M \leq 2h$ **do**
3: $\quad p :=$ NEXTGOODPRIME(a, b);
4: $\quad \bar{a} := a \bmod p$; $\bar{b} := b \bmod p$
5: $\quad$ **parallel_for** j **from** 0 **to** N **do**
6: $\quad\quad$ Compute a new evaluation point e_j at random from Z_p such that $\text{init}(a)|_{x=e_j} \not\equiv 0 \bmod p$ and $\text{init}(b)|_{x=e_j} \not\equiv 0 \bmod p$.
7: $\quad\quad A_j := \text{subres}(\bar{a}|_{x=e_j}, \bar{b}|_{x=e_j})$
8: $\quad$ **parallel_for** k **from** 0 **to** $n - 1$ **do**
9: $\quad\quad r := \min(N, (m + n - 2k)d)$
10: $\quad\quad B[k] :=$ interpolate the coefficients in x of $S_k \bmod p$ from $[A_0[k], \ldots, A_r[k]], [e_0, \ldots, e_r]$
11: $\quad$ **if** $M = 1$ **then** $C := B$;
12: $\quad$ **else**
13: $\quad\quad$ **parallel_for** k **from** 0 **to** $n - 1$ **do**
14: $\quad\quad\quad C[k] :=$ combine via CRT each coefficient of $C^*[k]$ in $\mathbb{Z}_M$ with $B[k]$ in $\mathbb{Z}_p$
15: $\quad$ **if** $C = C^*$ **then break**
16: $\quad M := Mp$; $C^* := C$
17: **return** C^*

equality $C^* = C$ means that $C[k]$ equals $C^*[k]$ for every $0 \leq k < \text{mdeg}(b)$, that is, the two subresultant chains (computed modulo M and modulo Mp, respectively) are equal.

In some sense, the probabilistic approach offers less parallelism. One could simply use the upper-limit on the coefficient size ($2h$) to compute as many images as required, all in parallel. Following the probabilistic method, we only compute one image over the prime field at a time. However, if we are computing the entire subresultant chain, there many polynomials to which CRT must be applied. We can update each polynomial in the chain independently to gain back some parallelism. This is the **parallel_for** loop on Lines 13–14.

For these three parallel-loops, the work to be performed is very regular, and can be determined by the size of the prime p and modulus Mp, and the degrees of a and b. There **parallel_for** loops are well-suited to the *map* pattern.

Algorithm 6.24 is a variant of Algorithm 6.23 where the evaluation and interpolation steps are performed via FFT. When the coefficient bound h and the degrees n, m, d

are large enough, this FFT-based approach substantially reduces the amount of work (algebraic complexity) without reducing the opportunities for concurrency. However, it increases memory consumption (as zero-padding is needed, see Lines 4 and 5, in order to apply FFT) and requires careful memory manipulation (e.g. data transposition, see Lines 11 and 14) in order to reduce the number of cache misses. Since a three-dimensional

Algorithm 6.24 FastBivariateSRC(a, b)

Input: polynomials $a, b \in \mathbb{Z}[x < y]$ with $m = \text{mdeg}(a)$, $n = \text{mdeg}(b)$, $m \geq n$, $h \in \mathbb{N}$ be a coefficient bound for all subresultants in subres(a, b), and $d = \max(\deg(a, x), \deg(b, x))$

Output: returns subres$(a, b) = \{S_{n-1}(a, b), \ldots, S_0(a, b)\}$ over $\mathbb{Z}[x, y]$, or *FAIL* if no suitable root of unity

1: $N :=$ smallest power of $2 > n \deg(a, x) + m \deg(b, x)$
2: $badOmega := 0$; $M := 1$; $C^* := 0$
3: **while** $M \leq 2h$ **do**
4: $\quad p :=$ NextGoodPrime(a, b);
5: $\quad \bar{a} :=$ ZeroPadding$(a \bmod p, m + 1, N)$
6: $\quad \bar{b} :=$ ZeroPadding$(b \bmod p, n + 1, N)$
7: $\quad \omega :=$ compute a N-th root of unity mod$p$
8: $\quad$ **if** any $\text{init}(a)|_{x=\omega^i}$ or $\text{init}(b)|_{x=\omega^i}$ are zero modulo p for $0 \leq i < N$ **then**
9: $\quad\quad badOmega := badOmega + 1$.
10: $\quad\quad$ **if** $badOmega > 5$ **then return** *FAIL*
11: $\quad\quad$ **else continue**
12: $\quad$ **parallel_for** j **from** 0 **to** m **do**
13: $\quad\quad \alpha_j := \text{FFT}(\text{coeff}(a, j, y), \omega, N, p)$
14: $\quad$ **parallel_for** j **from** 0 **to** n **do**
15: $\quad\quad \beta_j := \text{FFT}(\text{coeff}(b, j, y), \omega, N, p)$
16: $\quad \alpha :=$ Transpose(α); $\beta :=$ Transpose(β)
17: $\quad$ **parallel_for** j **from** 0 **to** $N - 1$ **do**
18: $\quad\quad A_j := \text{subres}(\alpha_j, \beta_j)$
19: $\quad A^t :=$ Transpose3D(A)
20: $\quad$ **parallel_for** k **from** 0 **to** $n - 1$ **do**
21: $\quad\quad B[k] := \frac{1}{N}\text{FFT}([A_0^t[k], \ldots, A_{N-1}^t[k]], \omega^{-1}, N, p)$
22: $\quad$ **if** $M = 1$ **then** $C := B$
23: $\quad$ **else**
24: $\quad\quad$ **parallel_for** k **from** 0 **to** $n - 1$ **do**
25: $\quad\quad\quad C[k] := \text{CRT}(C^*[k], M, B[k], p)$
26: $\quad$ **if** $C = C^*$ **then** `break`
27: $\quad M := Mp$; $C^* := C$
28: **end while**
29: **return** C^*

transposition could have different definitions depending on the context, we specify that which is used at Line 14. Given a three-dimensional array A of format $N \times n \times N$, the TRANSPOSE3D(A) returns a three-dimensional array A^t of format $n \times N \times N$ such that every element $A[j][k][i]$ is mapped to $A^t[k][i][j]$, for $0 \leq j < N$, $0 \leq k < n$, $0 \leq i < N$. The two-dimensional transposition (at Line 11) and the three-dimensional transposition (at Line 13) can be done in efficiently in parallel. This is not necessary in Algorithm 6.24, however, since the contributions of those transpositions on the critical path are negligible.

Returning to the idea of speculative computation of subresultants discussed in Section 6.2, one can easily modify Algorithms 6.23 and 6.24 so that they compute a given pair of consecutive non-zero subresultants from subres(a, b) rather than computing the entire chain. Indeed, rather than computing images of the entire subresultant chain over $\mathbb{Z}_p[y]$, one can call Algorithm 6.17 to compute only a pair of subresultants. Then, the interpolation and CRT steps need only to update two polynomials rather than the entire subresultant chain.

6.3.4 Parallelism in Removing Redundant Components

To remove redundant components efficiently we must address two issues: how to efficiently test single inclusions, e.g. $W(T_i) \subseteq W(T_j)$ and how to efficiently remove redundant components from a large set. The first issue is addressed by taking advantage of the heuristic algorithm ISNOTINCLUDED, see [187, `heuristic-no-split`, pp. 168], which is very effective in practice. Handling large sets of regular chains is possible by structuring the computation as a divide-and-conquer algorithm. We have already seen this structure in the serial version, Algorithm 6.14.

Given a set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains, REMOVEREDUNDANTCOMPONENTS($\mathcal{T}$), abbreviated RRC($\mathcal{T}$) and shown in Algorithm 6.25, removes redundant chains by dividing $\mathcal{T}$ into two subsets, producing two irredundant sets by means of recursion. Then, the two sets are merged by checking for pair-wise inclusions between the two sets. The divide-and-conquer nature of REMOVEREDUNDANTCOMPONENTS is undoubtedly admissible to ubiquitous fork-join parallelism. Particularly, one *forks* the computation to compute one of the recursive calls in parallel, and then *joins* upon return. These are indicated by the keywords **spawn** and **join**, respectively.[1]

Notice as well that the loops for inclusion testing in the merge step are also embarrassingly parallel. For each regular chain T_1 in the set $\mathcal{T}_1$, we must check if its quasi-component is included in any of the quasi-components of the regular chains in $\mathcal{T}_2$. Each

[1] We avoid the use of **fork** in pseudo-code since this may indicate forking an independent process.

Algorithm 6.25 RemoveRedundantComponents($\mathcal{T}$)

Input: a finite set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains
Output: regular chains forming an irredudant decomposition of the same algebraic set as $\mathcal{T}$

```
1: if |T| ≤ 1 then return T
2: ℓ := ⌈ |T|/2 ⌉; T_≤ℓ := first ℓ elements of T;
3: T_>ℓ := T \ T_≤ℓ
4: T_1 := spawn RemoveRedundantComponents(T_≤ℓ)
5: T_2 := RemoveRedundantComponents(T_>ℓ)
6: join
7: T_1' := ∅; T_2' := ∅
8: parallel_for T_1 in T_1
9: |   if ∀T_2 in T_2 IsNotIncluded (T_1, T_2) then
10: |   |   T_1' := T_1' ∪ {T_1}
11: parallel_for T_2 in T_2
12: |   if ∀T_1 in T_1' IsNotIncluded (T_2, T_1) then
13: |   |   T_2' := T_2' ∪ {T_2}
14: return T_1' ∪ T_2'
```

T_1 is independent and can be checked for inclusion against $\mathcal{T}_2$ simultaneously. We thus apply the map pattern here to gain further parallelism. In fact, this additional parallelization will aide in the overall parallel performance since, as the recursion unwinds and threads being idle, the list of components to merge simultaneously grows larger. Thus, these now idle threads can be used to execute the map steps.

Due to the fact that the removal of redundant components is merely a post-processing step of Triangularize, as seen in Algorithm 6.19, the parallelization of this step will not compete with other parallelized code within Triangularize for hardware resources. Moreover, since the input list of regular chains to RRC is well-defined, its parallelization follows regular parallelism. It is still dependent on the input system producing multiple components in order to exploit parallelism, but when there are multiple components, they can be handled via regular parallelism.

Speaking of this idea of competing with other parallelized code regions within Triangularize, we will now see how the parallel support introduced in Chapter 5 can help mitigate such competition.

6.3.5 Implementing the Parallelism

The previous sections described the many opportunities for concurrency within triangular decomposition. We have coarse-grained parallelism in the tasks of Triangularize

when it calls INTERSECT. We also have more fine-grained parallelism in the asynchronous generators and dynamic pipelines created between subroutines. These sources of parallelism are irregular since they rely on the geometry of the particular polynomial being solved to split into multiple components. Moreover, the splitting of component is itself only discovered throughout the triangular decomposition process.

On the other hand, we have also seen how computing subresultants via modular methods and removing redundancies from lists of regular chains can be performed using the map pattern and fork-join parallelism. These instances are in fact regular parallelism.

Despite these challenges of irregular or fine-grained parallelism, our experimentation (which is later presented in Section 6.4) confirms that each of these areas of parallelism are beneficial, and, moreover, compound together to bring further parallel speed-up. This can be attributed not only to the parallelization of each routine independently, but also to the effective cooperation of these parallel regions. In the remainder of this section we will examine how we have applied the parallel support of Chapter 5 to our implementation of triangular decomposition. Ultimately, this equates to a scheduling problem, where hardware resources must be shared and prioritized between different parallel regions. Our `ExecutorThreadPool` performs this scheduling. Moreover, Section 5.4 described how the thread pool handles cooperative parallelism through priority tasks and optional parallelism through encapsulating parallel constructs.

Let us start with the idea of cooperative parallelism in triangular decomposition. In the case of our many parallel schemes for triangular decomposition, the tasks of TRIANGULARIZE are the only priority task. Indeed, these tasks represent large amounts of work with respect to asynchronous generators or a particular subresultant chain computation. We want these coarse-grained triangularize tasks to be given priority to hardware resources. But, in the case where there are few such tasks, idle threads in the thread pool should be used by the more fine-grained parallelism of generators and subresultant chains. Moreover, a top-level call to INTERSECT from TRIANGULARIZE may produce multiple components to then create multiple triangularize tasks, exposing even further parallelism. Hence, these tasks should be given priority access to hardware resources as they will expose further parallelism which we may exploit in future. This intuition is confirmed by our experimentation, see Section 6.4, where the parallelization of triangularize tasks alone leads to the greatest parallel speed-up when compared to the speed-up gained from the other parallel schemes in isolation.

Turning to optional parallelism, we saw that the `ExecutorThreadPool` provides the ability to *reserve* threads toward implementing the map pattern and fork-join parallelism. The functional interface of the `ExecutorThreadPool` keeps the specifics of the parallelism

encapsulated. Client codes, in particular computing subresultant chains and removing redundant components, require no special code structures to handle the cases where parallelism does or does not occur, depending on how many threads there are available to be reserved in the thread pool.

However, if there are few independent components for a particular system to be solved, then component-level parallelism will be lacking and many threads will be idle in the thread pool. Those threads will be picked up automatically and used by the parallel subresultant chain computation. This cooperation can be seen as a sort of load-balancing, where threads are prioritized for coarse-grained triangularize tasks but additional parallelism can be exploited via subresultants if idle threads exist. Note that the removal of redundant components is merely a post-processing step of TriangularizeByTasks (Algorithm 6.19) and thus does not require cooperation, but can still effectively employ the `ExecutorThreadPool` to avoid over-subscription in its simultaneous use of the fork-join and map patterns.

In Section 5.3 we saw how the `ExecutorThreadPool` and `AsyncGenerator` classes could be used to implement parallel patterns. The translation of the pseudo-codes provided in this section (Algorithms 6.19–6.25) to a parallel implementation using that support is straightforward. We summarize the necessary translations.

- TriangularizeByTasks corresponds directly to the workpile pattern and may be implemented similarly to the workpile example in Listing 5.7.

- The subroutines of Triangularize being modelled as generators, can all be translated to use `AsyncGenerator` to produce outputs one at a time. Indeed, where **yield** is shown in Algorithms 6.20–6.22, one can call `generateObject()`. See the example in Listing 5.12.

- The **parallel_for** used for computing subresultants and removing redundant components (Algorithms 6.23–6.25) is directly translated to the map pattern using `obtainThreads()` and `waitForThreads()` as shown in Listing 5.9.

- The fork-join parallelism for removing redundant components translates directly to the thread pool's `obtainThread()` and `returnThread()` methods. See the example in Listing 5.9.

6.4 Experimentation and Discussion

The preceding sections have examined many opportunities and implementation details for parallelism within triangular decomposition algorithms. We have seen the use of:

(*i*) parallelism in triangularize tasks, which uses the workpile pattern to adapt to irregular component-level parallelism;

(*ii*) asynchronous generators to create producer-consumer pairs and dynamic pipelines;

(*iii*) the map pattern to exploit parallelism in subresultant chain computations; and

(*iv*) fork-join parallelism and the map pattern in a divide-and-conquer approach to the removal of redundant components (RRC).

While the major opportunities for parallelism rely on irregular parallelism coming from component-level parallelism, recall that the parallel RRC is regular parallelism given that multiple components have already been found. Moreover, parallel subresultant chain computations do not require multiple components at all to obtain parallelism. Nonetheless, all schemes are important since no singular scheme is sufficient in all cases.

We take this section to thoroughly examine our implementation through comprehensive experimentation. We have implemented triangular decomposition over the field of rational numbers and have applied to it the various parallel schemes. This implementation has been extensively evaluated through attempting to solve over 3000 systems of polynomial equations. The test suite is derived from systems described in the scientific literature, many of which have been collected into suites by [24] and [173], as well as from user-data and bug reports provided by *MapleSoft* and the *RegularChains* library [125]. The entire collection of systems may be downloaded from the BPAS website [7]. The properties of these systems are described in Section 1.2.

It is important to note that the test suite contains only 1076 systems which result in more than one component in their solution. Therefore, in all of the data which follows, one should recognize that no parallel speed-up is possible via component-level parallelism in the remaining roughly 2000 systems. Our experimentation does not separate these two groups of systems since, particularly in the case of the latter, we must ensure that our parallel schemes do not introduce undue overheads when there is no parallelism to exploit. Where examples have multiple components, parallel speed-ups reach up to 10.8× on a 12-core machine. Where examples do not have multiple components, our parallel schemes do not add considerable overhead, with only some non-trivial examples experiencing minor slowdown; see Figures 6.7–6.9. Timing data for some particular systems in this test suite is presented below in Table 6.2.

In all our experimental trials described in the following subsections, our implementation is configured by a variety of parameters. Firstly, any combination of the 4 different above-mentioned parallel schemes can be "turned on" to run in parallel, or otherwise run in serial. Secondly, the system to be solved can be solved in the sense of Kalkbrener or in the sense of Lazard and Wu, the latter often being much more difficult; see the definition of triangular decomposition in Section 2.5. In the trials which follow, we have solved 2815 systems in Kalkbrener mode and 2793 systems in Lazard mode. The difference is a result of limiting computations to 3 hours in wall time. Later references to the test suite refer to these subsets which are solvable in less than 3 hours.

Our discussion is organized into four subsections. First, in Section 6.4.1, we compare our current implementation of triangular decomposition in the BPAS library against that of the *RegularChains* library as distributed in *Maple 2020*. Second, Section 6.4.2 compares our initial version of parallel triangular decompositions presented in [12] against the present implementation, highlighting the performance improvements we have made since then. Third, we examine the effectiveness of each of the four aforementioned parallel schemes in isolation in Section 6.4.3. That is to say, comparing configurations where only one parallel scheme is active at a time against a purely serial configuration. Finally, Section 6.4.4 examines combinations of those parallel schemes being simultaneously active and thus competing and cooperating to obtain hardware resources. This final section highlights key issues and potential for future work.

Our experimentation was collected on a compute node running Ubuntu 18.04.4 with two Intel Xeon X5650 processors each with 6 cores (12 physical threads with hyperthreading; 24 threads total) at 2.67 GHz, and a 12x4GB DDR3 memory configuration at 1.33 GHz. BPAS was compiled with *GMP* 6.1.2 [91] and *NTL* 11.4.3 [163].

6.4.1 Comparing Against the *RegularChains* Library

We begin our discussion on experimental results by first comparing our implementation against that of the *RegularChains* library [125] as distributed in *Maple 2020* [128]. When evaluating parallel software it is important to first ensure that apparent parallel speed-up is not being derived from a poor serial implementation. Hence, we compare our serial implementation of triangular decomposition against the *RegularChains* library which has seen nearly 25 years of development and improvement.

Firstly, we have verified the correctness of our implementation against *RegularChains*. Where decompositions are identical (up to component ordering), verification is immediate. Otherwise, we have verified the solutions using a method based on *constructible sets*,

described in [187, Ch. 9] and implemented as part of the *RegularChains* library.

Secondly, performance of these two implementation is compared. The results of this comparison is summarized in Figure 6.3. In this figure, we plot the so-called *runtime ratio*, the running time for solving a system using *Maple* divided by the running time for solving that system using BPAS. While the *RegularChains* library is not explicitly parallelized, some internal routines, such as polynomial arithmetic, may be. Hence, we force *Maple* to run in serial by setting `kernelops[numcpus]` equal to 1.

In almost all cases we can see that BPAS is outperforming *Maple*, with BPAS performing up to 25 times faster. This could be attributed to the fact that BPAS is implemented in C and C++ compared to *RegularChains* which is implemented in the Maple scripting language. It may also be attributed to the high-performance polynomial arithmetic of BPAS for multivariate polynomials over the rational numbers [11]. In the few cases where *Maple* outperforms BPAS, these could be attributed to differences in factorization and GCD computations, both of which are crucial (and often costly) subroutines of triangular decomposition. *Maple* implements the GCD algorithm presented by [111], meanwhile BPAS uses a modified version of the EEZ-GCD algorithm [179] discussed in the next subsection below. The former can be more efficient when coefficient sizes are very large. Further optimization is needed to ensure BPAS performs better in all cases.

For some of the systems in our test suite, Table 6.2 shows their performance in BPAS in serial, in *Maple*, and in BPAS in parallel. The table also gives a description of the regular chains computed in the triangular decomposition, including their dimension and degrees. In the very challenging examples shown in Table 6.2, parallel speedup is limited by the fact that a single component in the decomposition dominates the computation.

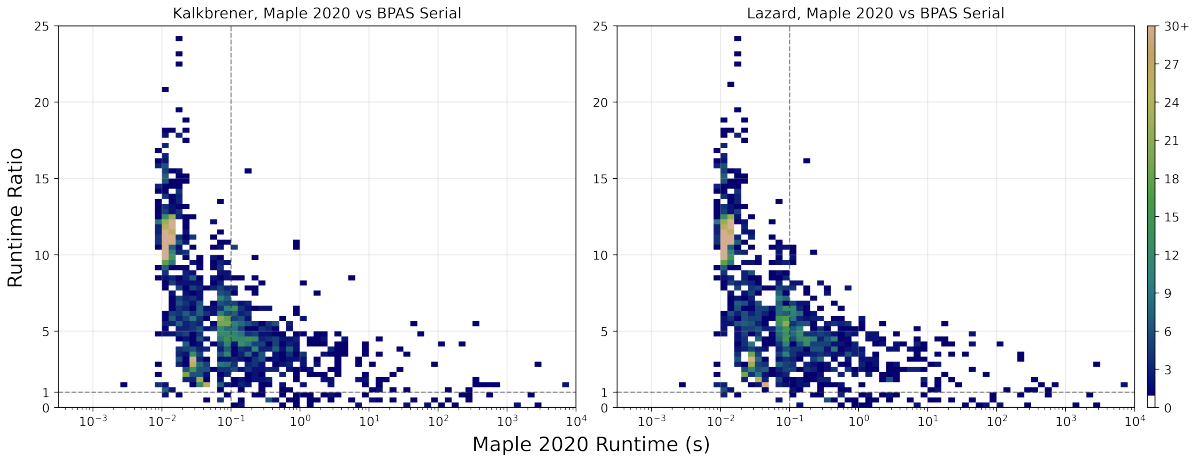

Figure 6.3: Comparing the runtime performance of triangular decomposition in the *RegularChains* library of *Maple 2020* against the serialized implementation in BPAS. The colouring of the two-dimensional histogram shows the number of systems which fall into each bin.

	Kalkbrener					Lazard				
System	Serial Time (s)	Maple Ratio	Parallel Speed-up	Dim., Deg.	Num. Comp.	Serial Time (s)	Maple Ratio	Parallel Speed-up	Dim., Deg.	Num. Comp.
Leykin-1	1.01	4.64	1.82	4, 1	19	1.71	4.50	2.00	4, 1	19
Sys2873	1.01	4.13	4.97	0, 1	1	1.01	4.13	4.97	0, 1	1
Gonnet	1.15	2.47	4.75	3, 1	3	1.14	2.51	4.48	3, 1	3
Sys1792	1.17	3.99	2.65	2, 4	9	1.18	2.70	2.59	2, 4	9
Sys2946	1.24	0.70	4.41	5, 1	4	1.57	0.91	3.09	5, 1	13
Sys2647	1.27	3.51	2.65	6, 2	4	2.62	3.06	3.89	6, 8	5
Pappus	1.27	3.08	3.01	6, 1	10	5.65	4.15	3.88	6, 1	119
Sys2945	1.30	2.77	3.57	2, 4	8	1.29	2.82	3.48	2, 4	8
W33	1.38	1.93	2.59	2, 1	1	1.63	1.80	2.46	2, 1	4
Sys3011	1.51	1.68	2.19	2, 1	1	1.55	1.88	2.23	2, 1	4
Sys2916	1.52	1.65	2.22	2, 1	1	1.55	1.88	2.22	2, 1	4
MontesS16	1.56	2.21	4.20	3, 1	7	1.58	2.23	3.98	3, 1	7
Wu-Wang	1.61	2.41	1.91	1, 1	5	2.04	1.90	2.24	1, 1	5
Hairer-2-BGK	1.80	1.47	3.33	2, 1	1	1.60	1.83	2.52	2, 1	4
Sys2353	2.16	3.84	4.35	7, 1	8	2.23	3.76	4.62	7, 1	8
W2	2.19	2.96	1.87	2, 1	6	2.50	2.50	2.16	2, 1	6
nld-3-5	2.22	4.09	2.68	0, 8	83	2.22	4.09	2.68	0, 8	83
Sys2875	2.44	3.17	6.23	0, 2	2	2.44	3.17	6.23	0, 2	2
8-3-config-Li	2.49	3.47	4.70	7, 2	15	9.63	4.15	4.52	7, 2	61
Sys2128	3.37	4.53	7.91	6, 1	9	3.29	4.54	7.75	6, 1	9
Sys2881	3.60	2.87	5.57	0, 2	2	3.60	2.87	5.57	0, 2	2
Sys2885	3.70	2.33	7.82	3, 1	3	3.69	2.39	8.48	3, 1	3
Sys2297	4.40	3.52	4.73	9, 2	12	4.34	3.35	4.80	9, 2	12
W5	6.96	3.89	5.83	4, 8	48	6.99	3.36	5.88	4, 8	48
Reif	7.81	1.96	5.74	-1	0	7.81	1.96	5.74	-1	0
Sys2161	8.80	5.40	7.91	8, 2	23	8.67	4.99	7.85	8, 2	23
W44	10.14	2.08	8.61	3, 2	6	10.67	1.88	8.67	3, 2	6
Mehta3	10.19	1.75	7.65	3, 6	35	9.84	4.75	1.89	3, 6	37
Sys2449	10.54	4.86	8.47	8, 1	7	10.85	4.17	8.84	8, 1	7
Sys2882	12.50	2.51	5.29	7, 3	12	16.69	2.50	6.06	7, 3	33
Sys2943	17.25	1.17	2.60	3, 4	4	21.90	1.35	2.65	3, 4	16
dgp6	29.04	2.76	8.49	3, 2	15	37.38	2.03	10.27	3, 2	15
Sys2880	56.57	4.32	10.10	5, 2	37	57.37	3.60	10.47	5, 2	37
Sys2874	70.43	5.39	10.22	3, 2	5	70.93	3.06	10.17	3, 2	5
Sys3270	149.11	1.04	3.72	0, 900	4	149.11	1.04	3.72	0, 900	4
Sys3283	167.82	1.90	3.46	0, 878	2	167.82	1.90	3.46	0, 878	2
Sys3281	214.47	1.22	3.07	0, 70	4	214.47	1.22	3.07	0, 70	4
KdV	456.08	1.38	3.68	12, 1	7	462.34	1.37	3.63	12, 1	7
Themos-net	1098.57	2.77	1.01	0, 24	1	1098.57	2.77	1.01	0, 24	1
tryme	3100.90	0.75	1.18	0, 8	7	3100.90	0.75	1.18	0, 8	7
childDraw-2	4499.91	0.32	1.25	0, 21	3	4499.91	0.32	1.25	0, 21	3
Sys1651	4792.44	1.39	1.16	0, 8	1	4792.44	1.39	1.16	0, 8	1
Sys2984	4793.55	1.39	1.16	0, 8	1	4793.55	1.39	1.16	0, 8	1
Pinchon-1	9608.51	0.30	1.36	0, 78	1	9608.51	0.30	1.36	0, 78	1

Table 6.2: Run times of some selected systems from the test suite solved in both Kalkbrener and Lazard modes. The runtime ratio between the *RegularChains* library of *Maple* and our serial implementation is presented as Maple Ratio. Parallel speed-up factor is determined against the configuration with all parallel schemes being active except asynchronous generators. The maximum dimension and product of main degrees of any triangular set in the decomposition is also presented, as is the number of triangular sets (number of components) in the final decomposition.

6.4.2 Comparing Against our Previous Implementation

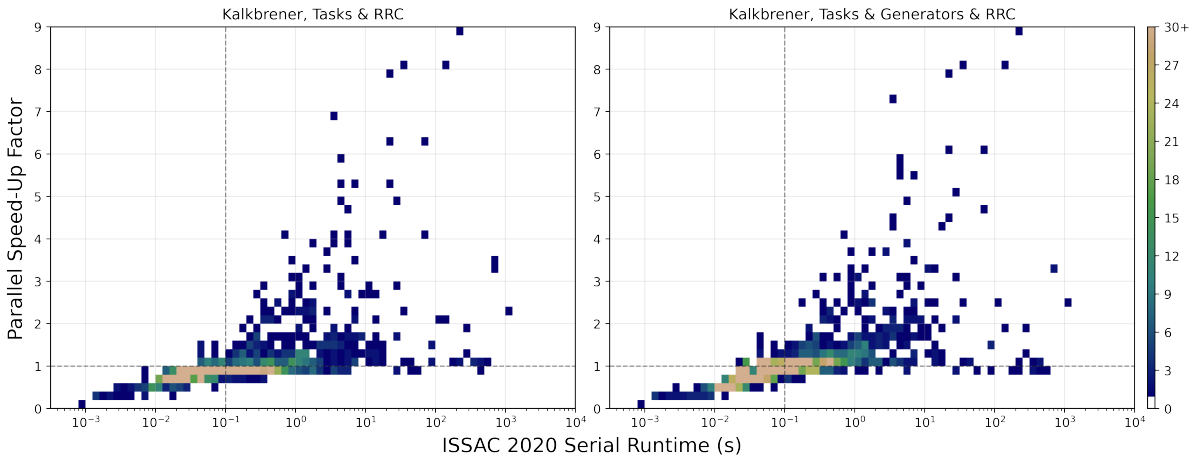

Figure 6.4: From ISSAC 2020 [12], comparing the parallel speed-up obtained for solving in the sense of Kalkbrener, without the use of asynchronous generators (left) and with the use of asynchronous generators (right). In both cases, parallel triangularize tasks and parallel removal of redundant components (RRC) are on. Parallel RRC is performed using *Cilk*.

We begin this section by reviewing our early work presented in [12]. This past work and its experimental results motivated our research direction and the work presented throughout this thesis. At the time of our previous work, we had implemented prototypes of three of our parallel schemes: parallel triangularize tasks, asynchronous generators, and parallel RRC. Figure 6.4 shows the parallel speed-ups obtained at that time without and with the use of asynchronous generators together with parallelized triangularize tasks and RRC. Ignoring trivial cases solved in less than 100ms—where parallel overheads expectedly dominated—we saw substantial and promising speed-ups. In particular, as can be seen by comparing the left plot with the right plot in Figure 6.4, the addition of asynchronous generators improved parallel speed-up in general.

Since the implementation of [12], the BPAS library has seen many performance improvements. Figure 6.5 highlights these improvements by comparing the runtime ratios between the previous implementation of [12] versus the present. Notably, performance has improved by a factor of up to 30. These performance gains can be attributed to improvements in algebraic algorithms and improvements in parallelization schemes.

With respect to algebraic algorithms, BPAS was once reliant on *Maple*'s C interface to make external calls to *Maple* in order to compute GCDs and factorizations of multivariate polynomials. Note that the apparent vertical lines at 0.05s in Figure 6.5 can be attributed to this fact as there is a minimum amount of time required to start and initialize the

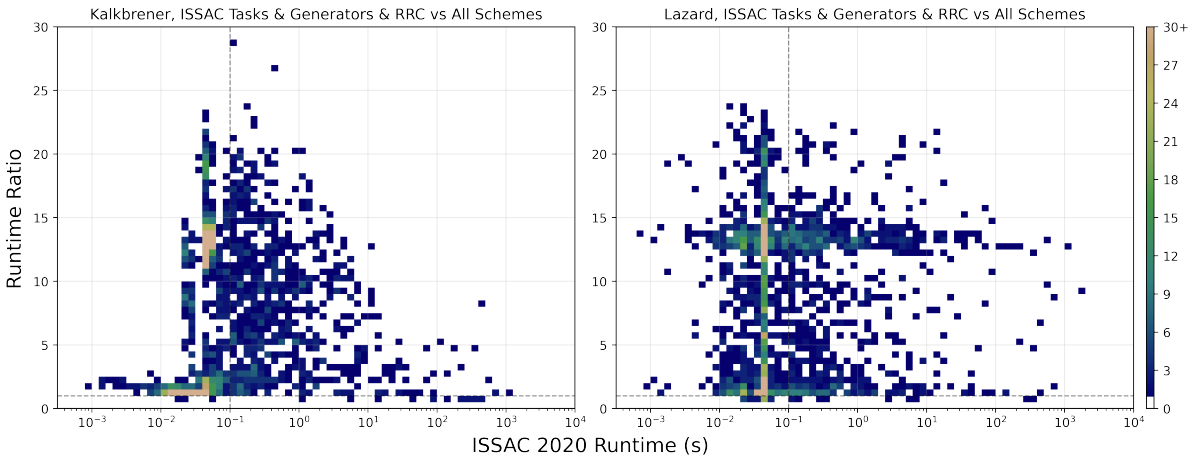

Figure 6.5: Comparing the runtime performance of parallel triangular decomposition at the time of [12] against the present parallel triangular decomposition.

Maple kernel. We have since implemented our own multivariate GCD and factorization algorithms based on Hensel lifting, which are modified versions of [178, 179]. In particular, our multivariate Hensel lifting follows new techniques with vastly improved performance first introduced by [136]. This was discussed in Section 6.1.

With respect to parallelization schemes, our previous implementation had three parallel regions: parallel triangularize tasks, asynchronous generators, and parallel RRC. We have since introduced a fourth parallel region which is the parallelization of the computation of subresultant chains. Moreover, the previous implementation had used *Cilk* [124] within parallel RRC to implement the fork-join and map patterns (see Section 6.3.4). We have now replaced this with the fork-join support of our `ExecutorThreadPool` (see Section 5.3.2). We have examined directly the effect of moving from *Cilk* (which, in any case, has had its support deprecated in GCC 7 and dropped completely in GCC 8) to our own thread pool. Figure 6.6 summarizes this comparison by showing the runtime ratios between using *Cilk* for parallel RRC, and using `ExecutorThreadPool` for parallel RRC (that is, *Cilk* / `ExecutorThreadPool`). In these test cases, we use the current state of implementation and all code was serial except for RRC. We see that for simple cases the overheads of making calls to *Cilk* is very apparent. For long-running cases, performance is similar, but slightly better, without the use of *Cilk*.

6.4.3 The Effectiveness of Each Parallel Scheme

Recall that we have four different areas of parallelism: (*i*) a workpile of triangularize tasks, (*ii*) asynchronous generators, (*iii*) parallel computation of subresultant chains, and

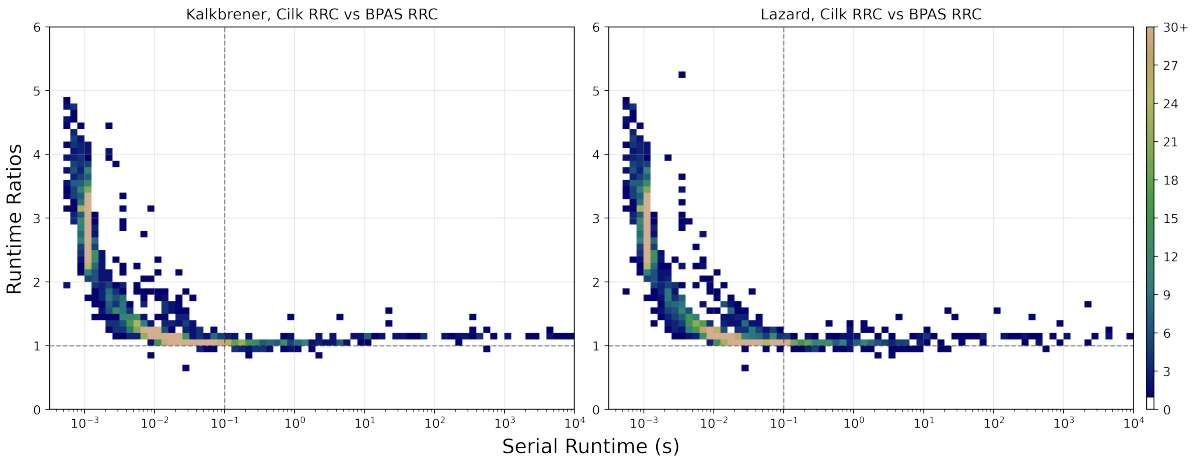

Figure 6.6: Comparing the runtime performance of parallelizing the removal of redundant components (RRC) using *Cilk* against parallelizing them via the `ExecutorThreadPool` of BPAS in both Kalkbrener and Lazard modes. All other code in these cases is run serially.

(*iv*) the fork-join and map patterns in the removal of redundant components. Moreover, recall that the first two exhibit irregular component-level parallelism. In contrast, the parallelization of subresultants and the removal of redundant components exhibit regular parallelism. Note that while the latter is also dependent on multiple components to obtain speed-up, the organization of the algorithm as divide-and-conquer implies that parallelism is regular and has little overheads in the cases where the number of components is small.

We examine the parallel speed-up obtained by having only one of our four parallel schemes active at a time. Such experimentation allows us to isolate the effects of each parallel scheme independently on the overall performance of triangular decomposition. This data is summarized in Figure 6.7, for solving systems in the sense of Kalkbrener, and in Figure 6.8, for solving systems in the sense of Lazard. Since the trends are identical for both Kalkbrener and Lazard, we will simply speak to the four different schemes directly.

It is obvious that the coarse-grained task-based parallelism is most effective at achieving parallel speed-up. Moreover, there are a reasonable number of cases which also benefit from the regular parallelism of subresultant chain computations and the removal of redundant components. It would be unrealistic to expect that every parallel scheme would be beneficial for every possible input system, particularly in the schemes where speed-up is derived from component-level parallelism. However, we do hope that each parallel scheme may exploit the parallelism which it can without unnecessarily slowing down cases where there is no parallelism to exploit. This is the case for parallel triangularize tasks, parallel subresultant chain computations, and parallel RRC (when ignoring cases with trivial runtime where parallel overheads necessarily dominate). In stark contrast,

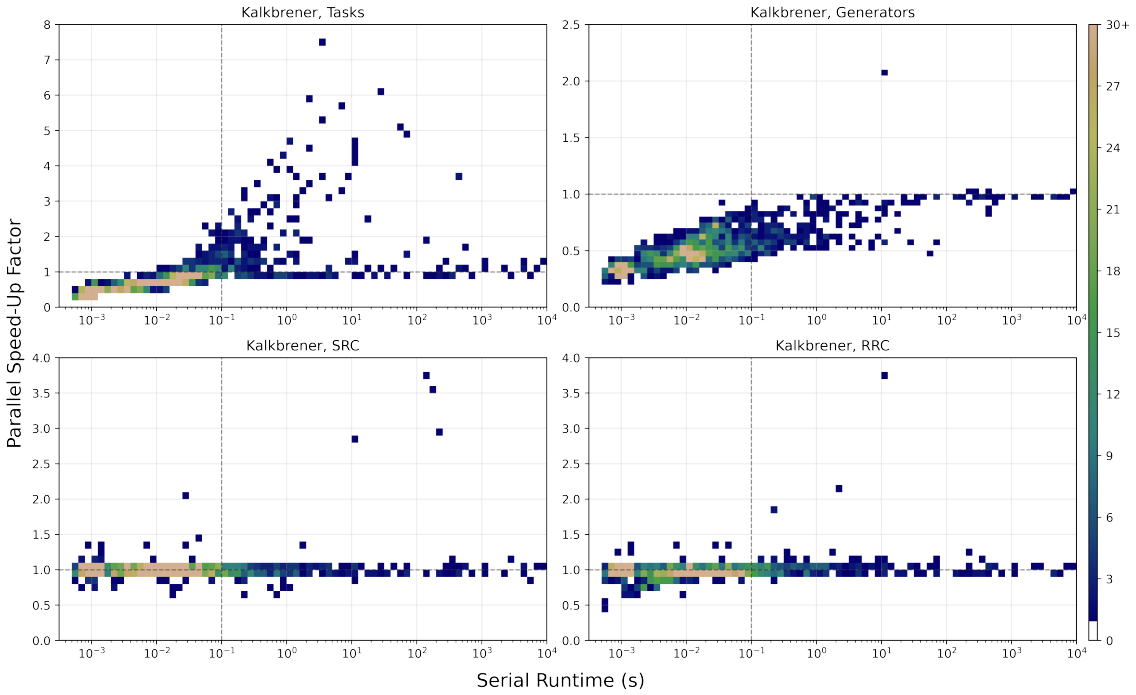

Figure 6.7: Comparing the parallel speed-up obtained from using only one parallel scheme at a time when solving in Kalkbrener mode. Top row: parallel triangularize tasks, asynchronous generators. Bottom row: parallel subresultant chains (SRC), parallel removal of redundant components (RRC).

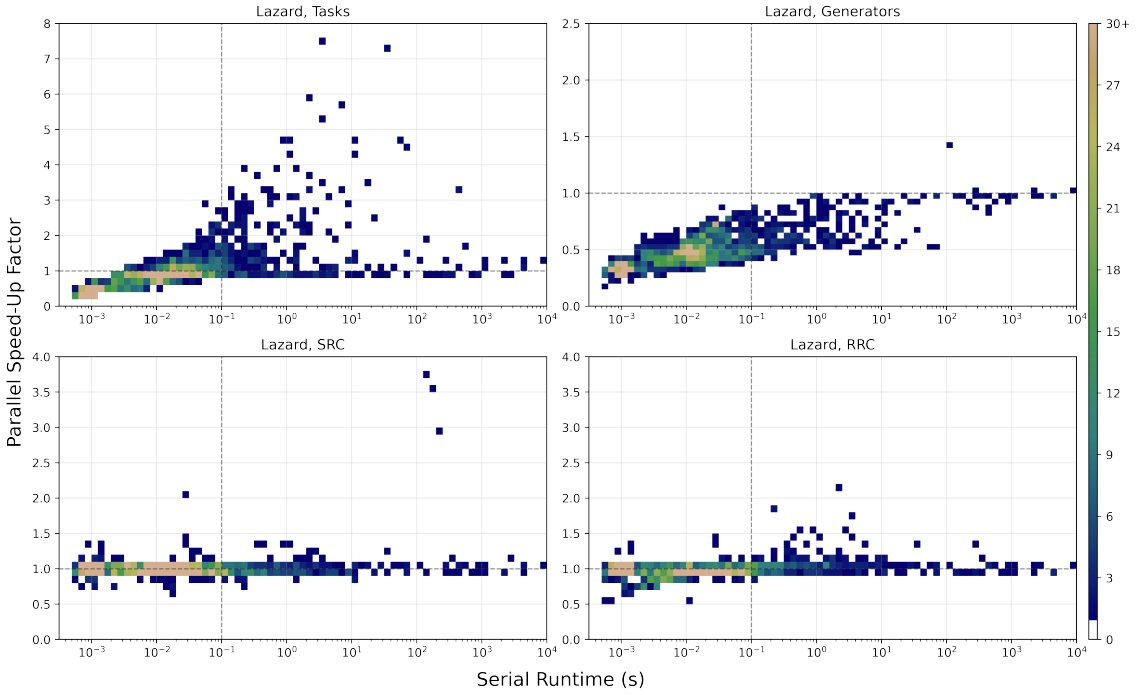

Figure 6.8: Comparing the parallel speed-up obtained from using only one parallel scheme at a time when solving in Lazard mode. Top row: parallel triangularize tasks, asynchronous generators. Bottom row: parallel subresultant chains (SRC), parallel removal of redundant components (RRC).

the lack of parallelism achieved by our asynchronous generators is surprising. This issue will be discussed in the next section where we further investigate the cooperation of generators with our other parallel schemes.

6.4.4 Cooperation of Parallel Schemes

In this section we look to evaluate the ability of our many different parallel schemes to cooperate together to achieve parallel speed-up. It would be improbable that every parallel scheme would be useful for every possible input system. However, we do hope, and indeed will see in the following experimentation, that there always exists some input systems which benefit from a combination of parallel schemes.

Since triangularize tasks and RRC both represent coarse-grained parallelism, and because they do not interfere with each other for access to hardware resources (recall Algorithm 6.19 where RRC is a post-processing step), we take our base configuration in this section to have those parallel schemes turned on. The parallel speed-up achieved by parallelizing triangularize tasks along with RRC is illustrated in Figure 6.9. These plots confirm that simultaneously parallelizing tasks and RRC is beneficial, since speed-ups of up to 10.8× are now obtained on a 12-core (24-thread with hyperthreading) machine.

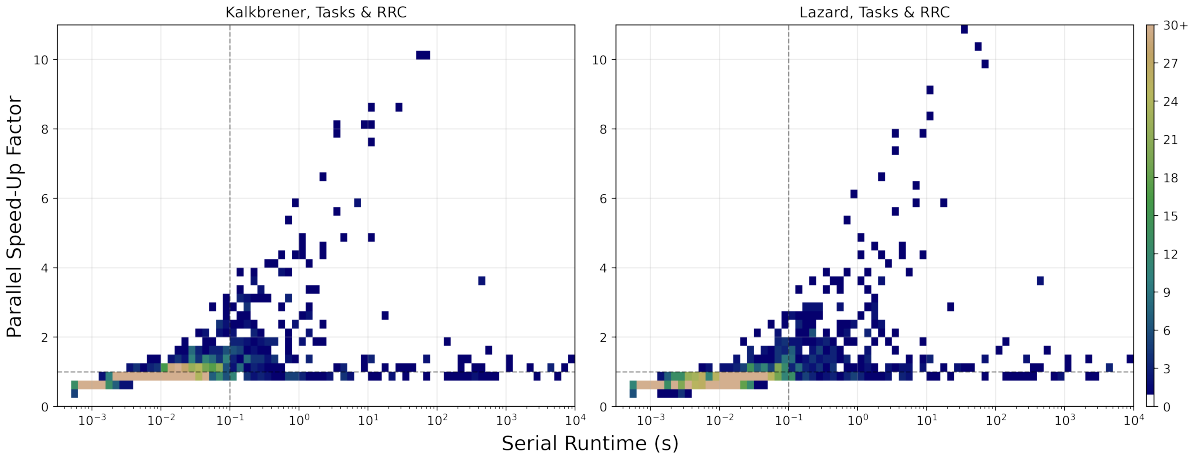

Figure 6.9: The parallel-speedup obtained from using parallel triangularize tasks and parallel removal of redundant components (RRC) together for solving in Kalkbrener and Lazard modes.

Comparing against this base implementation, we now look to examine the effects of adding the two fine-grained parallel schemes (asynchronous generators and parallel subresultants chains) independently, and then together. This produces three different comparison points: (*i*) Figure 6.10 compares the effect of adding asynchronous generators, (*ii*) Figure 6.11 compares the effect of adding parallel subresultant chains, and

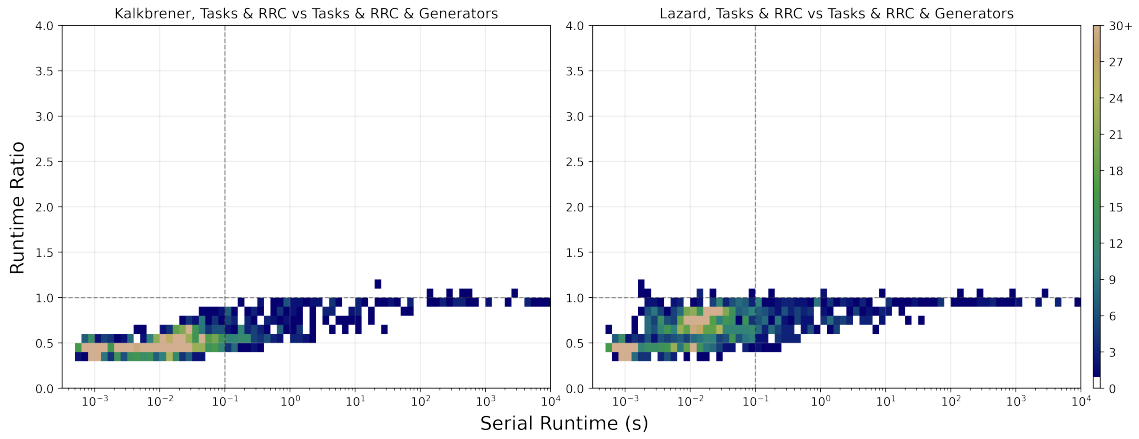

Figure 6.10: Using parallel triangularize tasks and parallel removal of redundant components (RRC) as the base case, compare the addition of asynchronous generators to overall performance for solving in Kalkbrener and Lazard modes.

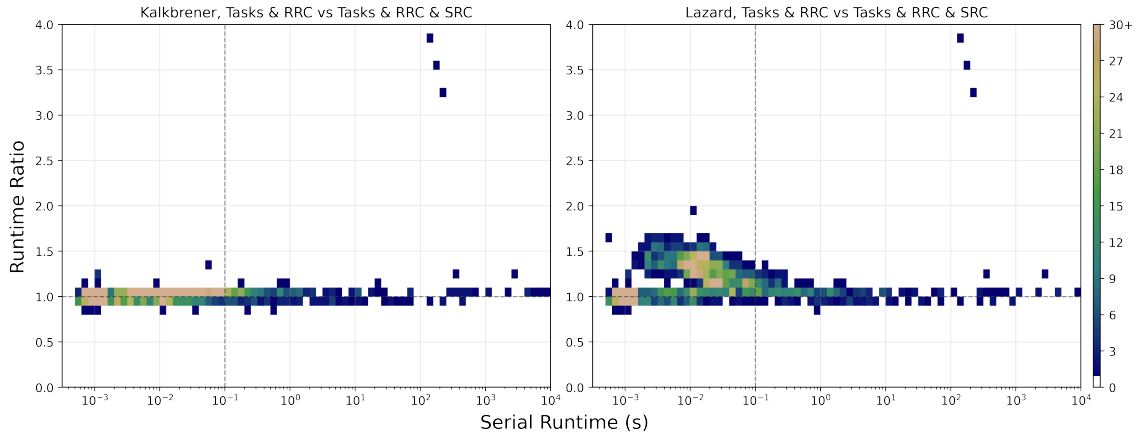

Figure 6.11: Using parallel triangularize tasks and parallel removal of redundant components (RRC) as the base case, compare the addition of parallel subresultant chains to overall performance for solving in Kalkbrener and Lazard modes.

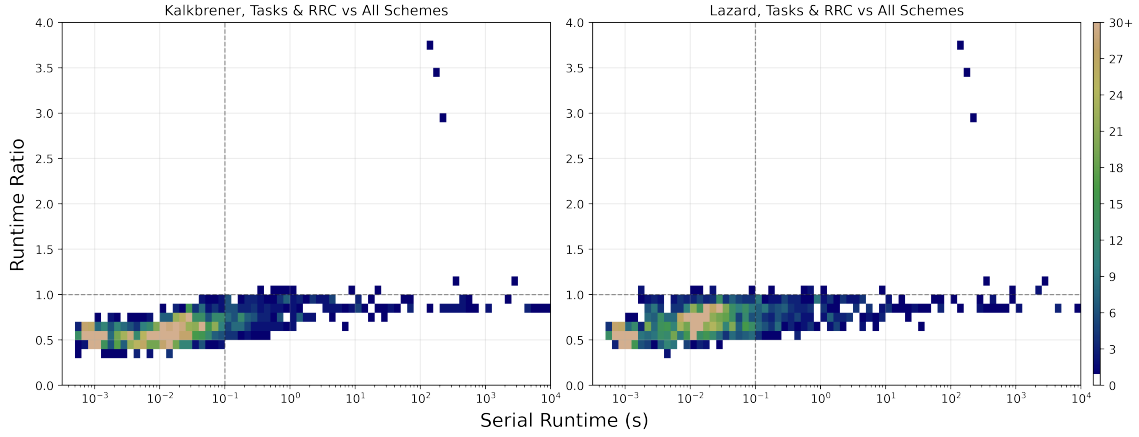

Figure 6.12: Using parallel triangularize tasks and parallel removal of redundant components (RRC) as the base case, compare the addition of both asynchronous generators and parallel subresultants to overall performance for solving in Kalkbrener and Lazard modes.

(*iii*) Figure 6.12 compares the effect of adding both fine-grained schemes. In all of these figures the runtime ratio with respect to the base configuration is shown. Thus, any case where runtime ratio is greater than 1 indicates that the addition of the fine-grained parallel scheme(s) increased not only overall performance but also increased the parallel speed-up achieved.

The highest performing configuration in general uses parallel triangularize tasks, parallel subresultant chains, and parallel removal of redundant components. As expected from the individualized parallel speed-ups seen in Figures 6.7 and 6.8, the addition of parallel subresultants further increased performance and parallel speed-up, meanwhile, the addition of asynchronous generators reduced performance. Using both fine-grained schemes together lead to the slow-downs of generators being dominant.

In our previous work [12] we observed that the use of generators actually helped obtain higher amounts of parallel speed-up, as can be seen between the two plots in Figure 6.4. Our new data suggests otherwise. The use of generators alone, and the addition of generators to existing parallel schemes, most often slows down the overall performance of triangular decomposition.

The newly discovered slow-downs can be attributed to several factors. Firstly, the state of the implementation in our previous work was less refined than the present one. In particular, we have noted the addition of high-performance algorithms for GCDs and factorization. This has drastically improved the overall performance of the triangular decomposition algorithm, as we have seen already in Figure 6.5. Therefore, within a single triangularize task, which is where GCDs and factorizations are computed, the use of generators may have been more beneficial when each task became "stuck" in the computation of a GCD or factorization, thus allowing different subroutines to operate concurrently and achieve speed-up. Where GCDs and factorizations can be computed much more quickly now, we see that the overheads of the generators become more apparent.

Recall also that the parallelism of asynchronous generators in triangular decomposition is derived from multiple components begin found within a single INTERSECT task. Throughout the many mutually-recursive subroutines of INTERSECT, a long pipeline of producer-consumer pairs may be created and many threads occupied for that pipeline. In the case when there is only one or two components found within an INTERSECT task, the use of these many threads rightfully slows down the computation where the components must be passed through the pipeline and thus between threads and between different CPU caches. This not only causes additional parallel overheads with inter-thread communication but also destroys data locality and reduces performance via cache misses. This is particularly the case since roughly two thirds of the systems in our test-suite

produce only a single component in their solution. Thus, passing that single component between threads and through queues between each producer-consumer adds considerable overhead for absolutely no parallelism.

Moreover, in these cases with few components, the asynchronous generators are "holding" onto threads and hardware resources, but they are not actively being used since consumers are merely left waiting for producers to return a single component. This reduces the availability of threads in the thread pool that could otherwise be used by other parallel schemes. A more robust solution would need more dynamic behaviour; ideally, asynchronous generators would run serially until the generator finds that it will produce more than one item. It is at that point that the generator should yield its first generated item and continue asynchronously from then on. The pipeline pattern, in general, is a useful parallel pattern that can give great speed-ups. Because of the irregular parallelism of triangular decompositions, however, additional work is needed in the implementation of our asynchronous generators to avoid overheads in the cases where there is no possible parallelism to exploit.

6.5 Conclusions and Future Work

Throughout this work we have examined opportunities to exploit parallelism in triangular decomposition algorithms. A key challenge to this effort is the fact that triangular decomposition exhibits irregular parallelism. Much of the parallelism which can be obtained depends on the geometry of the particular problem being solved. The geometry must split into multiple components in order for the algorithms to gain parallel speed-up by working on each component concurrently, and thus performing so-called component-level parallelism.

We have investigated four different parallel schemes within triangular decomposition routines: (*i*) parallel triangularize tasks, (*ii*) asynchronous generators, (*iii*) parallel computation of subresultant chains, and (*iv*) a parallel divide-and-conquer approach to the removal of redundant components (RRC). The first two follow irregular parallelism where the collection of components on which they operate dynamically grows as the algorithm progresses, thus intermittently revealing parallelism that cannot be known to exist in advance. In contrast, although RRC is dependent on component-level parallelism, its parallelism is more regular since the list of components on which it operates is known before the algorithm begins, allowing for regular and algorithmic decomposition of the work into parallel tasks. Parallel subresultant chains is the most regular since work can very easily be divided into parallel tasks through well-known degree bounds.

The implementation of these parallel schemes is detailed throughout Chapter 5 and this implementation extensively tested in Section 6.4 using thousands of polynomial systems. Our experimentation shows that all of these parallel schemes enable concurrent execution and parallel speed-up in some amount. We have achieved up to 10.8× parallel speed-up on a 12-core (24-thread with hyperthreading) machine. Parallelizing triangularize tasks, subresultant chain computation, and RRC simultaneously showed the greatest overall benefit. Generators, while theoretically promising, showed surprising deficiencies in implementation, which were discussed in Section 6.4.4. This highlights that more research on the topic is needed.

Indeed, there is much potential for future work on the topic of parallelizing triangular decomposition. In terms of algebraic algorithms, we look to extend the notion of speculative subresultants to handle computing subresultants with an arbitrary number of variables. We also look to modify the organization of the TRIANGULARIZE algorithm. We have seen that the task-based approach of TRIANGULARIZEBYTASKS in Algorithm 6.19 is generally more efficient, but loses the benefit of removing redundant components after each intersection. We will investigate the possibility of efficiently removing redundancies during computations instead of as merely a post-processing step. Of course, this intermittent removal of redundant components could potentially be parallelized to occur concurrently to the INTERSECT operations. Additionally, parallelizing those multivariate speculative subresultant chain computations would be a natural direction.

The issue of efficient asynchronous generators also requires further investigation. Presently, we have implemented the pipeline pattern in a traditional way, one thread is responsible for one stage of the pipeline and data flows through the pipeline. A different solution, and one used by Intel's *Thread Building Blocks* [131, Chapter 9], is to transpose functions and data so that one thread is bound to one item and it is the functional stages of the pipeline that flow past a single data item. This approach is more complex but has the benefit of improving the locality of data items. This is particularly important since our data items consist of regular chains and polynomials which can become quite large, making data locality important for performance. Moreover, as discussed in Section 6.4.4, an asynchronous generator could be implemented to run serially until more than one piece of data is produced, yielding it to its caller, and only then proceeding asynchronously.

These many directions for future work are considered further in Chapter 8 where we examine algorithms and designs which may lead to even further performance improvements for triangular decomposition.

Chapter 7

Parallel, yet Lazy, Hensel Factorization

Factorization via Hensel's lemma, or simply Hensel factorization, provides a mechanism for factorizing univariate polynomials with multivariate power series coefficients. In particular, for a multivariate polynomial in $(X_1, \ldots, X_n, Y)$, monic and square-free as a polynomial in Y, one can compute its roots with respect to Y as power series in $(X_1, \ldots, X_n)$. For a bivariate polynomial in (X_1, Y), the classical Newton–Puiseux method is known to compute the polynomial's roots with respect to Y as univariate Puiseux series in X_1. The transition from power series to Puiseux series arises from handling the non-monic case.

The *Hensel–Sasaki Construction* or *Extended Hensel Construction* (EHC) was proposed in [155] as an efficient alternative to the Newton–Puiseux method for the case of univariate coefficients. In the same paper, an extension of the Hensel–Sasaki construction for multivariate coefficients was proposed, and then later extended, see e.g., [103, 156]. In [4], EHC was improved in terms of algebraic complexity and practical implementation.

In this chapter, we present an implementation of multivariate power series which underlies a parallel algorithm for Hensel factorization based on repeated applications of Weierstrass preparation theorem. Of course, the fact that power series may have an infinite number of terms presents interesting challenges to computer scientists. How to represent them on a computer? How to perform arithmetic operations effectively and efficiently with them?

One standard approach is to implement power series as *truncated power series*, that is, by setting up in advance a sufficiently large accuracy, or precision, and discarding any power series term with a degree equal or higher to that accuracy. Unfortunately, for some important applications, not only is such accuracy problem-specific, but sometimes

cannot be determined before calculations start, or later may be found to not go far enough. This scenario occurs, for instance, with modular methods [120] for polynomial system solving [64] based on Hensel lifting and its variants [87], or when computing limits of real rational functions [4]. It is necessary then to implement power series with data structures and techniques that allow for dynamic updates.

Since a power series has potentially infinitely many terms, it is natural to represent it as a function, that we shall call a *generator*, which computes the terms of that power series for a given accuracy. This point of view leads to natural algorithms for performing arithmetic operations (addition, multiplication, division) on power series based on *lazy evaluation*.

Lazy evaluation in computer algebra has some history, see the work of Karczmarczuk [109] (discussing different mathematical objects with an "infinite" length) and the work of Monagan and Vrbik [141] (discussing sparse polynomial arithmetic). Lazy univariate power series, in particular, have been implemented by Burge and Watt [41] and by van der Hoeven [98]. However, up to our knowledge, our implementation is the first for *multivariate* power series in a compiled code.

Our implementation of lazy and parallel power series supports an arbitrary number of variables. However, the complexity estimates of our proposed methods are measured in the bivariate case; see Sections 7.3 and 7.4. This allows us to obtain sharp complexity estimates, giving the number of operations required to update each factor of a Hensel factorization individually. This information helps guide and load-balance our parallel implementation. Further, limiting to the bivariate case allows for comparison with existing works.

Denote by $M(n)$ a polynomial multiplication time [86, Ch. 8] (the cost sufficient to multiply two polynomials of degree n), Let $\mathbb{K}$ be algebraically closed and $f \in \mathbb{K}[[X_1]][Y]$ be a polynomial in Y with power series coefficients in X_1. Let f have degree d_Y in Y and total degree d. Our Hensel factorization computes the first k terms of all factors of f within $\mathcal{O}(d_Y^3 k + d_Y^2 k^2)$ operations in $\mathbb{K}$. We conjecture in Section 7.4 that we can achieve $\mathcal{O}(d_Y^3 k + d_Y^2 M(k) \log k)$ using relaxed algorithms [98]. The EHC of [4] computes the first k terms of all factors in $\mathcal{O}(d^3 M(d) + k^2 d M(d))$. Kung and Traub show that, over the complex numbers $\mathbb{C}$, the Newton–Puiseux method can do the same in $\mathcal{O}(d^2 k M(k))$ (resp. $\mathcal{O}(d^2 M(k))$) operations in $\mathbb{C}$ using a linear lifting (resp. quadratic lifting) scheme [118]. This complexity is lowered to $\mathcal{O}(d^2 k)$ by Chudnovsky and Chudnovsky in [56]. Berthomieu, Lecerf, and Quintin in [23] also present an algorithm and implementation based on Hensel lifting which performs in $\mathcal{O}(M(d_Y) \log(d_Y)\, k M(k))$; this is better than previous methods with respect to d (or d_Y), but worse with respect to k.

However, these estimates ignore an initial root finding step. Denote by $R(n)$ the cost of finding the roots in $\mathbb{K}$ of a degree n polynomial (e.g. [86, Th. 14.18]). Our method then performs in $\mathcal{O}(d_Y^3 k + d_Y^2 k^2 + R(d_Y))$. Note that the $R(d_Y)$ term does not depend on k, and is thus ignored henceforth. For comparison, however, Neiger, Rosenkilde, and Schost in [148] present an algorithm based on Hensel lifting which, *ignoring polylogarithmic factors*, performs in $\mathcal{O}(d_Y k + kR(d_Y))$.

Nonetheless, despite a higher asymptotic complexity, the formulation of EHC in [4] is shown to be practically much more efficient than that of Kung and Traub. Our serial implementation of lazy Hensel factorization (using plain, quadratic arithmetic) is orders of magnitude faster than that implementation of EHC; see Section 7.6. Though, we admit that EHC is in general a more powerful algorithm capable of handling more cases (i.e non-monic inputs). Similarly, we show that our serial lazy power series is orders of magnitude faster than the truncated implementations of *Maple*'s [128] `mtaylor` and *SageMath*'s [154] `PowerSeriesRing`. This highlights that a lazy scheme using suboptimal routines—but a careful implementation—can still be practically efficient despite higher asymptotic complexity.

Further still, it is often the case that asymptotically fast algorithms are much more difficult to parallelize, and have high parallel overheads, e.g. polynomial multiplication based on FFT. In Section 7.5, we look to improve the practical performance (i.e. when $k \gg d$) of our previous lazy implementation through the use of parallel processing rather than by reducing asymptotic bounds of the serial algorithms.

In Hensel factorization, computing power series terms of each factor relies on the computed terms of the previous factor. In particular, the output of one Weierstrass preparation becomes the input to another. These successive dependencies naturally lead to a parallel *pipeline*, or chain of *producer-consumer* pairs. Within numerical linear algebra, pipelines have already been employed in parallel implementations of singular value decomposition [93], LU decomposition, and Gaussian elimination [134]. Meanwhile, to the best of our knowledge, the only other use of parallel pipeline in symbolic computation has been in our implementation of triangular decomposition; see Chapter 6 and [12].

In the case of Hensel factorization, work reduces with each pipeline stage, limiting throughput. To overcome this challenge, we first make use of our complexity estimates to dynamically estimate the work required to update each factor. Second, we compose parallel schemes by applying the celebrated map-reduce pattern within Weierstrass preparation, and thus within a stage of the pipeline. Assigning multiple threads to a single pipeline stage improves load-balance and increases throughput. Experimental results show this composition is effective, with a parallel speed-up of up to 9× on a 12-core

machine.

The remainder of this chapter is organized as follows. Section 7.1 reviews power series notations and presents theorems and constructive proofs for Weierstrass Preparation Theorem and Hensel's lemma. Lazy power series and univariate polynomials with power series coefficients, their implementation, and experimental results, is presented in Section 7.2. Algorithms, complexity analyses, and lazy implementation details of Weierstrass preparation and Hensel factorization are given, respectively, in Sections 7.3 and 7.4. Section 7.5 then presents our parallel variations of those algorithms, where our complexity estimates are used for dynamic scheduling. Finally, Section 7.6 discusses experimental data of our implementation in the C language for the case $\mathbb{K} = \mathbb{Q}$.

We note that the work on lazy power series, presented in Section 7.2, is joint work with Mahsa Kazemi [32]. Theory and prototype implementation in Python, and experimentation by Kazemi; implementation in C by the present author.

7.1 Operations on Power Series and Univariate Polynomials over PowerSeries

We take this section to recall basic concepts and notation of multivariate power series and univariate polynomials over power series (UPoPS). Further, we present constructive proofs for the theorems of Weierstrass preparation and Hensel's lemma for UPoPS, from which algorithms are adapted; see Sections 7.3 and 7.4. Further introductory details may be found in Section 2.6.

Let $\mathbb{K}$ be an algebraically closed field. We denote by $\mathbb{K}[[X_1, \ldots, X_n]]$ the ring of formal power series with coefficients in $\mathbb{K}$ and with variables $X_1, \ldots, X_n$.

Let $f = \sum_{e \in \mathbb{N}^n} a_e X^e$ be a formal power series, where $a_e \in \mathbb{K}$, $X^e = X_1^{e_1} \cdots X_n^{e_n}$, $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$, and $|e| = e_1 + \cdots + e_n$. Let k be a non-negative integer. The *homogeneous part* of f in degree k, denoted $f_{(k)}$, is defined by $f_{(k)} = \sum_{|e|=k} a_e X^e$. The *order* of f, denoted $\mathrm{ord}(f)$, is defined as $\min\{i \mid f_{(i)} \neq 0\}$, if $f \neq 0$, and as ∞ otherwise.

Recall several properties regarding power series. First, $\mathbb{K}[[X_1, \ldots, X_n]]$ is an integral domain. Second, the set $\mathcal{M} = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \mathrm{ord}(f) \geq 1\}$ is the only maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$. Third, for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \mathrm{ord}(f) \geq k\}$. Note that for $n = 0$ we have $\mathcal{M} = \langle 0 \rangle$. Further, note that $f_{(k)} \in \mathcal{M}^k \setminus \mathcal{M}^{k+1}$ and $f_{(0)} \in \mathbb{K}$. Fourth, a unit $u \in \mathbb{K}[[X_1, \ldots, X_n]]$ has $\mathrm{ord}(u) = 0$ or, equivalently, $u \notin \mathcal{M}$.

Let $g, h \in \mathbb{K}[[X_1, \ldots, X_n]]$. The *sum* and *difference* $f = g \pm h$ is given by $\sum_{k \in \mathbb{N}} (g_{(k)} \pm h_{(k)})$. The product $p = g\,h$ is given by $\sum_{k \in \mathbb{N}} \left(\Sigma_{i+j=k}\, g_{(i)} h_{(j)} \right)$. Notice that these formulas

naturally suggest a *lazy evaluation* scheme, where the result of an arithmetic operation can be incrementally computed for increasing *precision*. A power series f is said to be known to precision $k \in \mathbb{N}$, when $f_{(i)}$ is known for all $0 \leq i \leq k$. Such an update function to compute new terms, parameterized by k, for addition or subtraction is simply $f_{(k)} = g_{(k)} \pm h_{(k)}$; an update function for multiplication is $p_{(k)} = \sum_{i=0}^{k} g_{(i)} h_{(k-i)}$. Lazy evaluation is discussed further in Section 7.2. From these update formulas, the following observation follows.

Observation 7.1 (power series arithmetic)**.** Let $g, h \in \mathbb{K}[[X_1]]$ with $f = g \pm h$ and $p = g\,h$. $f_{(k)} = g_{(k)} \pm h_{(k)}$ can be computed in 1 arithmetic operation in $\mathbb{K}$. $p_{(k)} = \sum_{i=0}^{k} g_{(i)} h_{(k-i)}$ can be computed in $2k-1$ arithmetic operations in $\mathbb{K}$.

Proof. For any power series in $\mathbb{K}[[X_1]]$, any of its homogeneous parts belong to $\mathbb{K}$. Computing $f_{(k)}$ is thus only one operation in $\mathbb{K}$. Computing $p_{(k)}$ requires k multiplications and $k-1$ additions in $\mathbb{K}$. □

Now, let $f, g \in \mathbb{A}[Y]$ be univariate polynomials over power series (UPoPS) where $\mathbb{A} = \mathbb{K}[[X_1, \ldots, X_n]]$. Writing $f = \sum_{i=0}^{d} a_i Y^i$, for $a_i \in \mathbb{A}$ and $a_d \neq 0$, we have that the degree of f (denoted $\deg(f, Y)$ or simply $\deg(f)$) is d. Note that arithmetic operations for UPoPS are easily derived from the arithmetic of its power series coefficients. Let f and g be of equal degree, appending zero terms to the one of lower degree otherwise. Further, let $g = \sum_{i=0}^{d} b_i Y^i$. Then, $f + g = \sum_{i=0}^{d} (a_i + b_i) Y^i$ and $f\,g = \sum_{i=0}^{2d} \left(\sum_{j+\ell=i} a_j\, b_\ell \right) Y^i$. A UPoPS is said to be known up to precision k if each of its power series coefficients are known up to precision k.

A UPoPS f is said to be *general (in Y) of order j* if $f \bmod \mathcal{M}[Y]$ has order j when viewed as a power series in Y. That is, for $f \notin \mathcal{M}[Y]$, writing $f = \sum_{i=0}^{d} a_i Y^i$, we have $a_i \in \mathcal{M}$ for $0 \leq i < j$ and $a_j \notin \mathcal{M}$. For example, let $f = (X_2^2 + X_1^2) + Y^2 + (1 + X_2^2) Y^3$. We have $f \equiv Y^2 + Y^3 \bmod \mathcal{M}[Y]$ and, viewing $f \bmod \mathcal{M}[Y]$ as a power series in Y, its order is 2. Thus, f is general in Y of order 2.

7.1.1 Weierstrass Preparation Theorem and Hensel Factorization

The Weierstrass Preparation Theorem (WPT) is fundamentally a theorem regarding factorization. In the context of analytic functions, WPT implies that any analytic function resembles a polynomial in the neighbourhood of the origin. Generally, WPT can be stated for power series over power series, i.e. for the power series $\mathbb{K}[[X_1, \ldots, X_n]][[Y]] = \mathbb{A}[[Y]]$.

This can be used to prove that $\mathbb{A}$ is both a unique factorization domain and a Noetherian ring. See [32] for such a proof regarding power series over power series. Here, it is sufficient to state the theorem for UPoPS.

First, we begin with a simple lemma which serves as the basis of our eventual proof of WPT and our implementation.

Lemma 7.2. Let $f, g, h \in \mathbb{K}[[X_1, \ldots, X_n]]$ such that $f = gh$. Let $f_i = f_{(i)}, g_i = g_{(i)}, h_i = h_{(i)}$. If $f_0 = 0$ and $h_0 \neq 0$, then g_k is uniquely determined by $f_1, \ldots, f_k$ and $h_0, \ldots, h_{k-1}$

PROOF. We proceed by induction on k. Since $f_0 = g_0 h_0 = 0$ and $h_0 \neq 0$ both hold, the statement holds for $k = 0$. Now let $k > 0$, assuming the hypothesis holds for $k - 1$. To determine g_k, it is sufficient to expand $f = gh$ modulo $\mathcal{M}^{k+1}$:

$$f_1 + f_2 + \cdots + f_k = g_1 h_0 + (g_1 h_1 + g_2 h_0) + \cdots + (g_1 h_{k-1} + \cdots + g_{k-1} h_1 + g_k h_0);$$

and, recalling $h_0 \in \mathbb{K} \setminus \{0\}$, we have

$$\begin{aligned} f_k &= g_1 h_{k-1} + \cdots + g_{k-1} h_1 + g_k h_0 \\ g_k &= {}^1\!/_{h_0} \left(f_k - g_1 h_{k-1} - \cdots - g_{k-1} h_1 \right). \end{aligned}$$

□

Theorem 7.3 (Weierstrass Preparation Theorem)**.**
Let f be a polynomial of $\mathbb{K}[[X_1, \ldots, X_n]][Y]$ so that $f \not\equiv 0 \bmod M[Y]$ holds. Write $f = \sum_{i=0}^{d+m} a_i Y^i$, with $a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$, where $d \geq 0$ is the smallest integer such that $a_d \notin \mathcal{M}$ and m is a non-negative integer. That is, f is general in Y of order d. Then, there exists a unique pair p, α satisfying the following:

(*i*) $f = p\,\alpha$,

(*ii*) α is an invertible element of $\mathbb{K}[[X_1, \ldots, X_n]][[Y]]$,

(*iii*) p is a monic polynomial of degree d,

(*iv*) writing $p = Y^d + b_{d-1} Y^{d-1} + \cdots b_1 Y + b_0$, we have $b_{d-1}, \ldots, b_0 \in \mathcal{M}$.

PROOF. If $n = 0$, writing $f = \alpha Y^d$ with $\alpha = \sum_{i=0}^{m} a_{i+d} Y^i$ proves the existence of the decomposition. Now, assume $n \geq 1$. Write $\alpha = \sum_{i=0}^{m} c_i Y^i$, with $c_i \in \mathbb{K}[[X_1, \ldots, X_n]]$. We will determine $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ modulo successive powers of $\mathcal{M}$. Since we require

α to be a unit, $c_0 \notin \mathcal{M}$ by definition. $a_0, \ldots, a_{d-1}$ are all 0 mod $\mathcal{M}$. Then, equating coefficients in $f = p\,\alpha$ we have:

$$\begin{aligned}
a_0 &= b_0c_0 \\
a_1 &= b_0c_1 + b_1c_0 \\
a_2 &= b_0c_2 + b_1c_1 + b_2c_0 \\
&\vdots \\
a_{d-1} &= b_0c_{d-1} + b_1c_{d-2} + \cdots + b_{d-2}c_1 + b_{d-1}c_0 \\
a_d &= b_0c_d + b_1c_{d-1} + \cdots + b_{d-1}c_1 + c_0 \\
a_{d+1} &= b_0c_{d+1} + b_1c_d + \cdots + b_{d-1}c_2 + c_1 \\
&\vdots \\
a_{d+m-3} &= b_{d-3}c_m + b_{d-2}c_{m-1} + b_{d-3}c_{m-2} + c_{m-3} \\
a_{d+m-2} &= b_{d-2}c_m + b_{d-1}c_{m-1} + c_{m-2} \\
a_{d+m-1} &= b_{d-1}c_m + c_{m-1} \\
a_{d+m} &= c_m
\end{aligned} \tag{7.1}$$

and thus $b_0, \ldots, b_{d-1}$ are all 0 mod $\mathcal{M}$. Then, $c_i \equiv a_{d+i} \bmod \mathcal{M}$ for all $0 \leq i \leq m$. All coefficients have thus been determined mod $\mathcal{M}$. Let $k \in \mathbb{Z}^+$. Assume inductively that all $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ have been determined mod $\mathcal{M}^k$ we will now determine them mod $\mathcal{M}^{k+1}$.

It follows from Lemma 7.2 that b_0 is uniquely determined mod $\mathcal{M}^{k+1}$ from the equation $a_0 = b_0c_0$. Consider now the second equation. Since b_0 is known mod $\mathcal{M}^{k+1}$, and $b_0 \in \mathcal{M}$, the product b_0c_1 is also known mod $\mathcal{M}^{k+1}$, despite c_1 only being known mod $\mathcal{M}^k$. Then, we can determine b_1 using Lemma 7.2 and the formula $a_1 - b_0c_1 = b_1c_0$. This procedure follows for $b_2, \ldots, b_{d-1}$. With $b_0, \ldots, b_{d-1}$ known mod $\mathcal{M}^{k+1}$ and, again, $b_0, \ldots, b_{d-1} \in \mathcal{M}$, each $c_0, \ldots, c_m$ can be determined mod $\mathcal{M}^{k+1}$ from the last $m+1$ equations. $\square$

The following example shows Weierstrass preparation applied to a UPoPS.

Example 7.4. Let $f = \frac{1}{1+X_1+X_2}Y^3+Y^2+X_2Y+X_1$ be a UPoPS in $\mathbb{Q}[[X_1, X_2]][Y]$. Using the notation of Theorem 7.3, we have $d = 2$ and $a_d = 1$ and thus applying Weierstrass preparation yields $f = p\,\alpha$ with $\deg(p, Y) = 2$ and $\deg(\alpha, Y) = 1$. In particular, we have $p = Y^2 + b_1Y + b_0$ and $\alpha = c_1Y + c_0$. Equating coefficients, we have:

$$\begin{aligned} a_0 &= b_0c_0 \\ a_1 &= b_0c_1 + b_1c_0 \\ a_2 &= b_1c_1 + c_0 \\ a_3 &= c_1 \end{aligned}$$

Modulo $\mathcal{M}$ we have:

$$\begin{array}{lll} a_3^{(1)} = 1 - X_1 - X_2 & b_1^{(1)} = -X_1 + X_2 & c_1^{(1)} = a_3^{(2)} \\ a_2^{(1)} = 1, \quad a_1^{(1)} = X_2, \quad a_0^{(1)} = X_1 & b_0^{(1)} = X_1 & c_0^{(1)} = X_1 - X_2 + 1 \end{array}$$

Modulo $\mathcal{M}^2$ we have:

$$\begin{aligned} a_3^{(2)} &= 1 - X_1 - X_2 + X_1^2 + 2X_1X_2 + X_2^2 \\ a_2^{(2)} &= 1, \quad a_1^{(2)} = X_2, \quad a_0^{(2)} = X_1 \end{aligned}$$

$$\begin{array}{ll} b_1^{(2)} = -X_1 + X_2 + 3X_1^2 - 2X_1X_2 + X_2^2 & c_1^{(2)} = a_3^{(3)} \\ b_0^{(2)} = X_1 - X_1^2 + X_1X_2 & c_0^{(2)} = 1 + X_1 - X_2 + 2X_1X_2 - 4X_1^2 \end{array}$$

Modulo $\mathcal{M}^3$ we have:

$$\begin{aligned} a_3^{(3)} &= 1 - X_1 - X_2 + X_1^2 + 2X_1X_2 + X_2^2 - X_1^3 - 3X_1^2X_2 - 3X_1X_2^2 - X_2^3 \\ a_2^{(3)} &= 1, \quad a_1^{(3)} = X_2, \quad a_0^{(3)} = X_1 \end{aligned}$$

$$\begin{aligned} b_1^{(3)} &= -X_1 + X_2 + 3X_1^2 - 2X_1X_2 + X_2^2 - 14X_1^3 + 13X_1^2X_2 - 6X_1X_2^2 + X_2^3 \\ b_0^{(3)} &= X_1 - X_1^2 + X_1X_2 + 5X_1^3 - 4X_1^2X_2 + X_1X_2^3 \end{aligned}$$

$$\begin{aligned} c_1^{(3)} &= a_3^{(1)} \\ c_0^{(3)} &= 1 + X_1 - X_2 + 2X_1X_2 - 4X_1^2 + 18X_1^3 - 11X_1^2X_2 + 4X_1X_2^2 - X_2^3 \end{aligned}$$

One requirement of WPT is that $f \not\equiv 0 \bmod \mathcal{M}[Y]$. That is to say, f cannot vanish at $(X_1, \ldots, X_n) = (0, \ldots, 0)$ and, specifically, f is general of order $d = \deg(p)$. A suitable linear change in coordinates can always be applied to meet this requirement; see Algorithm 7.1 in Section 7.2.3. Since Weierstrass preparation provides a mechanism to factor a UPoPS into two factors, suitable changes in coordinates and several applications of WPT can fully factorize a UPoPS. The existence of such a factorization is given by Hensel's lemma for UPoPS.

Theorem 7.5 (Hensel's Lemma)**.**
Let $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i$ be a monic polynomial with $a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$. Let $\bar{f} = f(0, \ldots, 0, Y) = (Y - c_1)^{d_1}(Y - c_2)^{d_2} \cdots (Y - c_r)^{d_r}$ for $c_1, \ldots, c_r \in \mathbb{K}$ and positive integers $d_1, \ldots, d_r$. Then, there exists unique $f_1, \ldots, f_r \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$, all monic in Y, such that:

(*i*) $f = f_1 \cdots f_r$,

(*ii*) $\deg(f_i, Y) = d_i$ for $1 \leq i \leq r$, and

(*iii*) $\bar{f}_i = (Y - c_i)^{d_i}$ for $1 \leq i \leq r$.

Proof. We proceed by induction on r. For $r = 1$, $d_1 = d$ and we have $f_1 = f$, where f_1 has all the required properties. Now assume $r > 1$. A change of coordinates in Y, sends c_r to 0. Define $g(X_1, \ldots, X_n, Y) = f(X_1, \ldots, X_n, Y + c_r) = (Y + c_r)^d + a_{d-1}(Y + c_r)^{d-1} + \cdots + a_0$. By construction, g is general of order d_r and WPT can be applied to obtain $g = p\,\alpha$ with p being of degree d_r and $\bar{p} = Y^{d_r}$. Reversing the change of coordinates we set $f_r = p(Y - c_r)$ and $f^* = \alpha(Y - c_r)$, and we have $f = f^* f_r$. f_r is a monic polynomial of degree d_r in Y with $\bar{f}_r = (Y - c_r)^{d_r}$. Moreover, we have $\bar{f}^* = (Y - c_1)^{d_1}(Y - c_2)^{d_2} \cdots (Y - c_{r-1})^{d_{r-1}}$. The inductive hypothesis applied to f^* implies the existence of $f_1, \ldots, f_{r-1}$. □

7.2 Lazy Power Series

Our power series implementation is both lazy and high-performing. To achieve this, our design and implementation has two goals:

(*i*) compute only terms of the series which are truly needed; and

(*ii*) have the ability to "resume" a computation, in order to obtain a higher precision power series without restarting from the beginning.

Of course, the lazy nature of our implementation refers directly to (*i*), while the high-performance nature is due in part to (*ii*) and in part to other particular implementation details to be discussed.

Facilitating both of these aspects requires the use of some sort of generator function—a function which returns new terms for a power series to increase its precision. Such a *generator*, is the key to high-performance in our implementation, yet also the most difficult part of the design.

Our goal is to define a structure encoding power series so that they may be dynamically updated on request. Each power series could then be represented as a polynomial alongside some generator function. A key element of this design is to "hide" the updating of the underlying polynomial. In our C implementation this is done through a functional interface comprising of two main functions: (*i*) getting the homogeneous part of a power series, and (*ii*) getting the polynomial part of a power series, each for a requested degree. These functions call some underlying generator to produce terms until the requested degree is satisfied.

Functions for homogeneous part and polynomial part are shown using *Maple*-like pseudo-code in Listing 7.1 as `homog_part_ps` and `polynomial_part_ps`, respectively. The key element to these functions are their automatic calls to the generator function `GEN` if the requested degree is greater than the current degree of the power series.

```
1  homog_part_ps := proc(ps, d::integer)
2      if (d > ps[DEG]) then
3          for i from ps[DEG] + 1 to d do
4              ps[POLY] := ps[POLY] + ps[GEN](i)
5          end do;
6          end if;
7      return homogeneous_part(ps[POLY], d);
8  end proc;
9
10 polynomial_part_ps := proc(ps, d::integer)
11     if (d > ps[DEG]) then
12         for i from ps[DEG] + 1 to d do
13             ps[POLY] := ps[POLY] + ps[GEN](i)
14         end do;
15     end if;
16     return truncate_poly(ps[POLY], d);
17 end proc;
```

Listing 7.1: A lazy power series design where a generator function is called on demand through some top-level functional interface.

```
geometric_series_ps := proc(vars::list)
    local homog_parts := proc(vars::list)
        return d -> sum(vars[i], i=1..nops(vars))^d;
    end proc;
    ps := table();
    ps[DEG] := 0;
    ps[GEN] := homog_parts(vars); #capture vars in closure, return a
       function
    ps[POLY] := ps[GEN](0);
    return ps;
end proc;
```

Listing 7.2: The geometric series as a lazy power series.

As a first example, consider, the construction of the geometric series as a lazy power series, in *Maple*-style pseudo-code, in Listing 7.2. A power series is a data structure holding a polynomial, a generator function, and an integer to indicate up to which degree the power series is currently known. In this simple example, we see the need to treat functions as first-class objects. The manipulation of such functions is easy in functional or scripting languages, where dynamic typing and first-class function objects support such manipulation.

This manipulation becomes further interesting where the generator of a power series must invoke other generators, as in the case of arithmetic (see Section 7.2.2).

In support of high-performance we choose to implement our power series in the strongly-typed and compiled C programming language rather than a scripting language. On one hand, this allows direct access to our underlying high-performance polynomial implementation [11], but on the other hand creates an impressive design challenge to effectively handle the need for dynamic function manipulation.

In the remainder of this section we detail our resulting solution, which makes use of a so-called *ancestry* in order for the generator function of a newly created power series to "remember" from where it came. We begin in Section 7.2.1 with an overview of the basic power series representation, its data structure, and our solution to generator functions in C. In Section 7.2.2 we discuss power series multiplication and division, thus discussing the combination of this data structure with our run-time support for creating a new generator dynamically. Section 7.2.2 also presents experimentation of our implementation against *SageMath* and *Maple* showing orders of magnitude improvement in computation time.

7.2.1 Data Structure, Generators, and Ancestors

The organization of our power series data structure is focused on supporting incremental generation of new terms through continual updates. To support this, the first fundamental design element is the storage of terms of the power series. The current polynomial part, i.e. the terms computed so far, of a power series are stored in a *graded representation*. An array of (pointers to) polynomials is maintained whereby the index of a polynomial in this array is equal to its (total) degree. This representation is said to be *dense* as the array holds all terms, even those which are zero. This is an array of homogeneous polynomials representing the homogeneous parts of the power series, called the *homogeneous part array*.

The power series data structure is a simple C struct holding this array, as well as integer numbers indicating the degree up to which homogeneous parts are currently known, and the allocation size of the homogeneous part array. The underlying polynomials are sparse multivariate polynomials with rational number coefficients; see [11, 31].

Using our graded representation, the generator function is simply a function returning the homogeneous part of a power series for a requested degree. Unfortunately, in the C language, functions are not readily handled as objects. Hence, we look to essentially create a *closure* for the generator function (see, e.g., [160, Ch. 3]), by storing a function pointer along with the values necessary for the function. For simplicity of implementation, these captured values are passed to the function as arguments. We first describe this function pointer.

In an attempt to keep the generators as simple as possible, we enforce some symmetry between all generators and thus the stored function pointers. Namely: (*i*) the first parameter of each generator must be an integer, indicating the degree of the homogeneous polynomial to be generated; and (*ii*) they must return that homogeneous polynomial. For some generator functions, e.g. the geometric series, this single integer argument is enough to obtain a particular homogeneous part. However, this is insufficient for most cases, particularly for generating a homogeneous part of a power series which resulted from an arithmetic operation.

Therefore, to introduce some flexibility in the generators, we extend the previous definition of a generator function to include a finite number of `void` pointer parameters following the first integer parameter.

The use of `void` pointer parameters is a result of the fact that function pointers must be declared to point to a function with a particular number and type of parameters. Since we want to store this function pointer in the power series struct, we would otherwise need to capture all possible function declarations, which is a very rigid solution. Instead,

`void` pointer parameters simultaneously allow for flexibility in the types of the generator parameters, as well as limit the number of function pointer types which must be captured by the power series struct. Flexibility arises where these `void` pointers can be cast to any other pointer type, or even cast to any machine-word-sized plain data type (e.g. `long` or `double`). In our implementation, these so-called *void generators* are simple wrappers, casting each `void` pointer to the correct data type for the particular generator, and then calling the *true generator*. Section 7.2.2 provides an example in Listing 7.5.

Our implementation, which supports power series arithmetic, Weierstrass preparation, and factorization via Hensel's lemma, requires only 4 unique types of function pointers for these generators. All of these function pointers return a polynomial and take an integer as the first parameter. They differ in taking 0–3 `void` pointer parameters as the remaining parameters. We call the number of these `void` pointer parameter the generator's *order*. We have thus *nullary generators*, *unary generators*, *binary generators*, and *ternary generators*. We then create a union type for these 4 possible function pointers and store only the union in the power series struct. The generator's order is also stored as an integer to be able to choose the correct generator from the union type at runtime.

Finally, these `void` pointers are also stored in the struct to eventually be passed to the generator. When the generator's order is less than maximum, these extra `void` pointers are simply set to `NULL`. The structure of these generators, the generator union type, and the power series struct itself is shown in Listing 7.3. In our implementation, these generators are used generically, via the aforementioned functional interface. In the code listings which follow, these functions are named `homogPart_PS` and `polynomialPart_PS`, to compute the homogeneous part and polynomial part of a power series, respectively. Whereas `homog_part_ps` and `polynomial_part_ps` in the pseudo-code of Listing 7.1 used generator function objects generically, our functions simply use function pointers rather than function objects.

In general, these void pointer generator parameters are actually pointers to existing power series structs. For example, the operands of an arithmetic operation would become arguments to the generator of the result. This relation then yields a so-called *ancestry* of power series. In this indirect way, a power series "remembers" from where it came, in order to update itself upon request via its generator. This may trigger a cascade of updates where updating a power series requires updating its "parent" power series, and so on up the *ancestry tree*. Section 7.2.2 explores this detail in the context of power series arithmetic, meanwhile it is also discussed as a crucial part of a lazy implementation of Weierstrass preparation (Section 7.3) and factorization via Hensel's lemma (Section 7.4).

```
typedef Poly_ptr (*homog_part_gen)(int);
typedef Poly_ptr (*homog_part_gen_unary)(int, void*);
typedef Poly_ptr (*homog_part_gen_binary)(int, void*, void*);
typedef Poly_ptr (*homog_part_gen_ternary)(int, void*, void*, void*);

typedef union HomogPartGenerator {
    homog_part_gen nullaryGen;
    homog_part_gen_unary unaryGen;
    homog_part_gen_binary binaryGen;
    homog_part_gen_terary ternaryGen;
} HomogPartGenerator_u;

typedef struct PowerSeries {
    int deg;
    int alloc;
    Poly_ptr* homog_polys;
    HomogPartGenerator_u gen;
    int genOrder;
    void *genParam1, *genParam2, *genParam3;
} PowerSeries_t;
```

Listing 7.3: A first implementation of the power series struct in C and function pointer declarations for the possible generator functions. `Poly_ptr` is a pointer to a polynomial.

The implementation of this ancestry requires yet one more additional feature. Since our implementation is in the C language, we must manually manage memory. In particular, references to parent power series (via the void pointers) must remain valid despite actions from the user. Indeed, the underlying updating mechanism should be transparent (i.e. hidden) to the end-user. Thus, it should be perfectly valid for an end-user to obtain, for example, a power series product, and then free the memory associated with the operands of the multiplication. Yet, the resulting product must hold on to the operands still as its "parents".

In support of this, we have established a reference counting scheme. Whenever a power series is made, the parent of another power series (by being set as the value of the child's generator parameter) its reference count is incremented. Therefore, the user may "free" or "destroy" a power series when it is no longer needed, but the memory persists as long as some other power series has reference to it. Destruction is then only a decrement of a reference counter.

However, once the counter falls to 0, the data is actually freed, and moreover, a child power series will decrement the reference count of its parents, since that reference has finally been removed.

In a final complication, we must consider the case when a void pointer parameter is not

```
typedef enum GenParamType {
    PLAIN_DATA = 0,
    POWER_SERIES = 1,
    UPOPS = 2,
    MPQ_LIST = 3
} GenParamType_e;

// An updated PowerSeries struct with reference counts and parameter types.
typedef struct PowerSeries {
    int deg;
    int alloc;
    Poly_ptr* homog_polys;
    HomogPartGenerator_u gen;
    int genOrder;
    int refcont;
    void *genParam1, *genParam2, *genParam3;
    GenParamType_e paramType1, paramType2, paramType3;
} PowerSeries_t;

void destroyPowerSeries_PS(PowerSeries_t* ps) {
    --(ps->refCount);
    if (ps->refCount <= 0) {
        for (int i = 0; i <= ps->deg; ++i) {
            freePolynomial(homog_polys[i]);
        }
        if (ps->genParam1 != NULL && ps->paramType1 == POWER_SERIES) {
            destroyPowerSeries_PS((PowerSeries_t*) ps->genParam1);
        }
        // repeat for other parameters.
    }
}
```

Listing 7.4: Extending the power series struct to include reference counting (as the `refCount` field) and management of reference counts via `destroyPowerSeries_PS`.

pointing to a power series. We resolve this by storing, in the power series struct, a value to identify the actual type of a void parameter. A simple `if` condition can then check this type and conditionally free the generator parameter, if it is not plain data. For example, a power series or a UPoPS, see Listing 7.4. We implement this as an enumeration instead of a Boolean so that the implementation is extensible to further parameter types. One may think that storing both a void pointer, along with an enumeration value which encodes the actual type of that pointer, to be wasteful. However, the additional memory usage is minimal compared to that used by the polynomial data itself. Moreover, alternative solutions using, for example, union types, would still need a way of determining the current valid field in the union for a particular context.

7.2.2 Implementing Power Series Arithmetic

With the power series structure fully defined, we are now able to see examples putting its generators to use. Given the design established in the previous section, implementing a power series operation is as simple as defining the unique generator associated with that operation. In this section we present power series multiplication and division using this design. Let us begin with the former.

As we have seen in Section 7.1, the power series product of $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$ is defined simply as $h = fg = \sum_{k \in \mathbb{N}} \left(\Sigma_{i+j=k} \left(f_{(i)} g_{(j)} \right) \right)$. In our graded representation, continually computing new terms of h requires simply computing homogeneous parts of increasing degree. Indeed, for a particular degree k we have $(fg)_{(k)} = \sum_{i+j=k} f_{(i)} g_{(j)}$. Through our use of an ancestry and generators, the power series h can be constructed lazily, by simply defining its generator and generator parameters, and instantly returning the resulting struct. The generator in this case is exactly a function to compute $(fg)_{(k)}$ from f and g.

In reality, the generator stored in the struct encoding h is the *void generator* `homogPartVoid_prod_PS` which, after casting parameters, simply calls the *true generator*, `homogPart_prod_PS`. This is shown in Listing 7.5. The actual power series operator is `multiplyPowerSeries_PS`, returning a lazily constructed power series product. There, the parents `f` and `g` are reserved (reference count incremented) and assigned to be generator parameters, and the generator function pointer set. Finally, a single term of the product is computed. Notice that in `homogPart_prod_PS`, `homogPart_PS` is called on `f` and `g`. This allows the runtime to dynamically compute more terms and update `f` and `g` as needed in order to compute more terms of their product. That is, updating the child power series may cause an update of the parent power series.

Now consider finding the quotient $h = \sum_e c_e X^e$ which satisfies $f = gh$ for a given power series $f = \sum_e a_e X^e$ and an invertible power series $g = \sum_e b_e X^e$. One could proceed by equating coefficients in $f = gh$, with b_0 being the constant term of g, to obtain $c_e = {}^1\!/_{b_0}(a_e - \sum_{i+j=e} b_i c_j)$. This formula can easily be rearranged in order to find the homogeneous part of h for a given degree k:

$$h_{(k)} = \frac{1}{g_{(0)}} \left(f_{(k)} - \sum_{i=1}^{k} g_{(i)} h_{(k-i)} \right).$$

This formula is possible since to compute $h_{(k)}$ we need only $h_{(i)}$ for $i = 1, \ldots, k-1$. Moreover, the base case is simply $h_{(0)} = f_{(0)}/g_{(0)}$, a valid division in $\mathbb{K}$ since $g_{(0)} \neq 0$. The rest follows by induction.

```
Poly_ptr homogPart_prod_PS(int d, PowerSeries_t* f, PowerSeries_t* g) {
    Poly_ptr sum = zeroPolynomial();
    for (int i = 0; i <= d; i++) {
        Poly_ptr prod = multiplyPolynomials(
            homogPart_PS(d-i, f),  homogPart_PS(i, g));
        sum = addPolynomials(sum, prod);
    }
    return sum;
}

Poly_ptr homogPartVoid_prod_PS(int d, void* param1, void* param2) {
    return homogPart_prod_PS(d, (PowerSeries_t*) param1, (PowerSeries_t*)
       param2);
}

PowerSeries_t* multiplyPowerSeries_PS(PowerSeries_t* f, PowerSeries_t* g) {
    if (isZeroPowerSeries_PS(f) || isZeroPowerSeries_PS(g)) {
        return zeroPowerSeries_PS();
    }

    reserve_PS(f); reserve_PS(g);
    PowerSeries_t* prod = allocPowerSeries(1);
    prod->gen.binaryGen= &(homogPartVoid_prod_PS)
    prod->genParam1 = (void*) f; prod->genParam2 = (void*) g;
    prod->paramType1 = POWER_SERIES; prod->paramType2 = POWER_SERIES;
    prod->deg = 0;
    prod->homogPolys[0] = homogPart_prod_PS(0, f, g);
    return prod;
}
```

Listing 7.5: Computing the multiplication of two power series, where `homogPart_prod_PS` is the generator of the product.

```
Poly_ptr homogPart_quo_PS(int d, PowerSeries_t* f, PowerSeries_t* g,
   PowerSeries_t* h) {
    if (d == 0) {
        return dividePolynomials(homogPart_PS(0, f), homogPart_PS(0, g));
    }
    Poly_ptr s = homogPart_PS(d, f);
    for (int i = 1; i <= deg; ++i) {
        Poly_ptr p = multiplyPolynomials(homogPart_PS(i, g),
            homogPart_PS(d-i, h));
        s = subPolynomials(s, p);
    }
    return divideByRational(s, homogPart(0, g))
}
```

Listing 7.6: Computing the division of two power series, where `homogPart_quo_PS` is the genrator of the quotient.

In our graded representation, where power series are updated with successive homogeneous parts, this formula yields a generator for a power series quotient. The realization of this generator in code is simple, as shown in Listing 7.6. Not shown is the void generator wrapper and a top-level function to return the lazy quotient, which is simply symmetric to the previous multiplication example in Listing 7.5. The only trick to this generator for the quotient is that it requires a reference to the quotient itself. This creates an issue of a circular reference in the power series ancestry. To avoid this, we abuse our parameter typing and label the quotient's reference to itself as plain data.

We now look to compare our implementation against *SageMath* [154], and *Maple* 2020. In *Maple*, the `PowerSeries` library [6, 110], a sub-library of the *RegularChains* library, provides lazy multivariate power series, meanwhile the built-in `mtaylor` command provides *truncated* multivariate Taylor series. Similarly, *SageMath* includes only truncated power series. In these latter two, an explicit precision must be used and truncations cannot be extended once computed. Consequently, our experimentation only measures computing a particular precision, thus not using our implementation's ability to resume computation. We compare against all three; see Figures 7.1–7.3.

In *SageMath*, the multivariate power series ring $R[[X_1, \ldots, X_n]]$ is implemented using the univariate power series ring $S[[T]]$ with $S = R[X_1, \ldots, X_n]$. In $S[[T]]$, the subring formed by all power series f such that the coefficient of T^i in f is a homogeneous polynomial of degree i (for all $i \geq 0$) is isomorphic to $R[[X_1, \ldots, X_n]]$. By default, Singular [66] underlies the multivariate polynomial ring S while Flint [94] underlies the univariate polynomials used in univariate power series. Python 3.7 interfaces and joins these underlying implementations. To see exactly how *SageMath* works consider $f \in \mathbb{Q}[[X_1, X_2]]$ with the goal is to compute $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ to precision d. One begins by constructing the power series ring in X_1, X_2 over $\mathbb{Q}$ with the default precision set to k as `R.<x,y> = PowerSeriesRing(QQ, default_prec=k)`. Then `g = f^-1` returns the inverse, and `h = f * g` the desired product, to precision k.

Throughout this chapter our benchmarks were collected for an implementation which specializes $\mathbb{K}$ to $\mathbb{Q}$. Individual trials were performed with a time limit of 1800 seconds on a machine running Ubuntu 18.04.4 with an Intel Xeon X5650 processor running at 2.67 GHz, with 12x4GB DDR3 memory at 1.33 GHz.

The first set of benchmarks are presented in Figure 7.1 where the power series $f = 1 + X_1 + X_2$ is both inverted and multiplied by its inverse. Figures 7.2 and 7.3 present the same but for $f = 1 + X_1 + X_2 + X_3$ and $f = 2 + \frac{1}{3}(X_1 + X_2)$, respectively. In all cases, $f \cdot \frac{1}{f}$ includes the time to compute the inverse. It is clear that our implementation is orders of magnitude faster than existing implementations. This is due in part to

the efficiency of our underlying polynomial arithmetic implementation [11], but also to our execution environment. Our implementation is written in the C language and fully compiled, meanwhile, both *SageMath* and *Maple* have a level of interpreted code impacting performance. We note that, through truncated power series as polynomials, the dense multiplication of a power series by its inverse is trivial for *SageMath* and `mtaylor`.

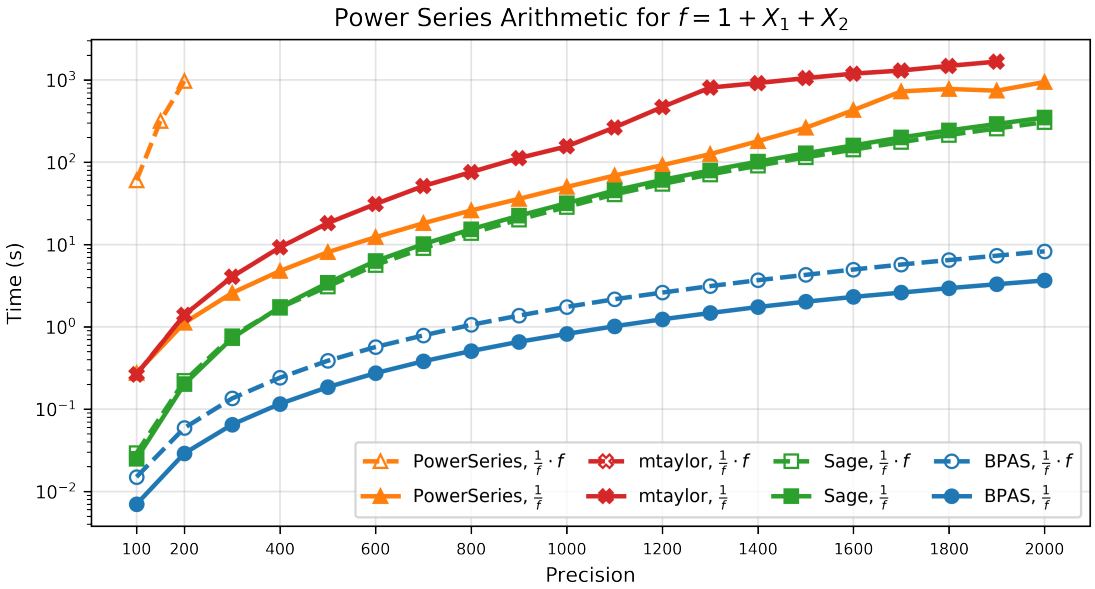

Figure 7.1: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2$

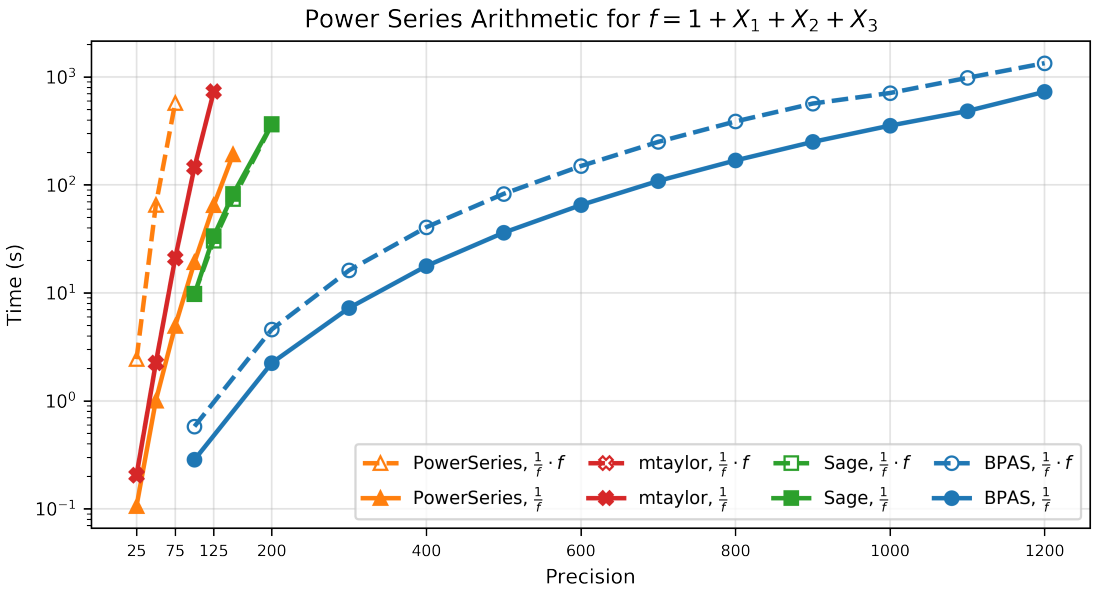

Figure 7.2: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2 + X_3$.

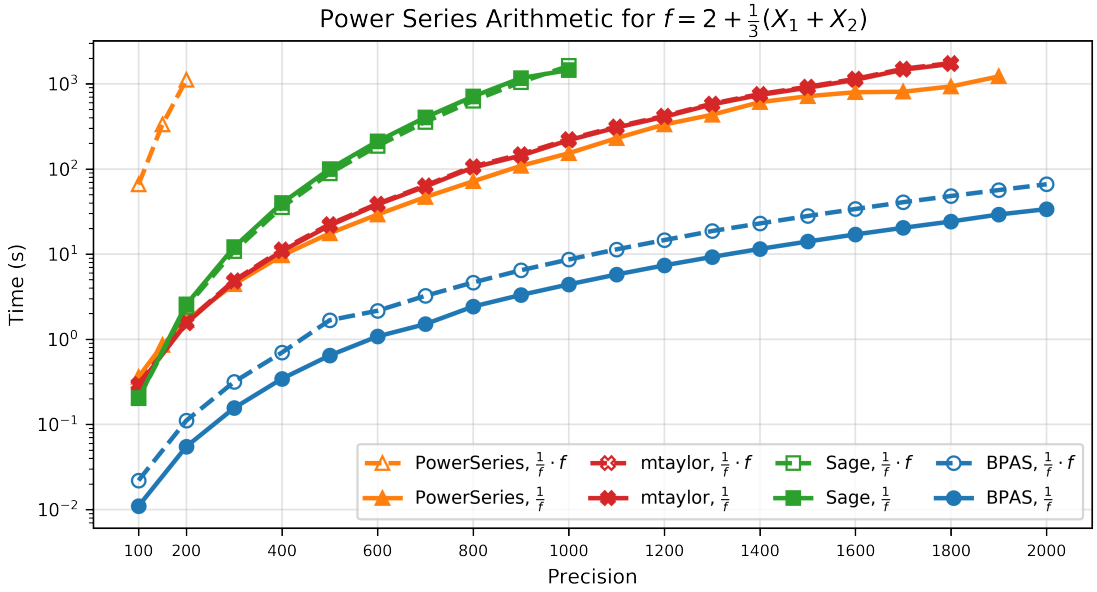

Figure 7.3: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 2 + \frac{1}{3}(X_1 + X_2)$

In *Maple* 2021, we introduced the `MultivariatePowerSeries` library [10] which adapted our lazy power series structure and algorithms to the *Maple* ecosystem. As a result, performance significantly increased within *Maple*, and is nearly on the order of magnitude as our C implementation. We repeat the tests cases shown in Figures 7.2 and 7.3 for *Maple* 2021, with results shown in Figures 7.4 and 7.5. In these latter two figures, MPS denotes the new `MultivariatePowerSeries` implementation, RCPS denotes the `PowerSeries` library of *RegularChains*, and BPAS denotes our C implementation.

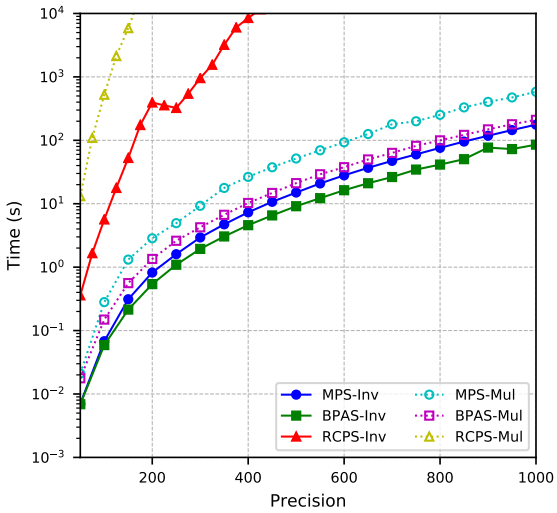

Figure 7.4: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f = 1 + X_1 + X_2 + X_3$.

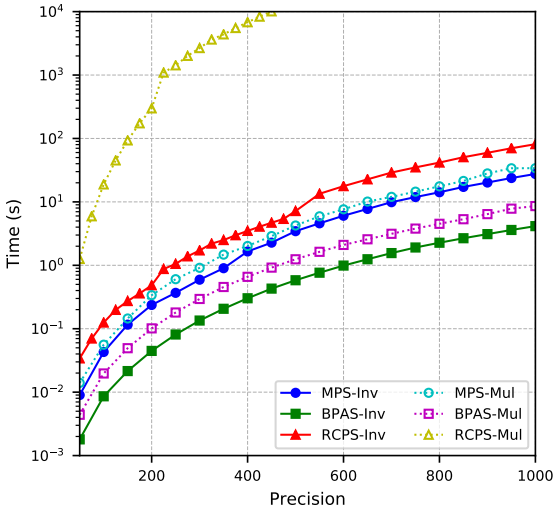

Figure 7.5: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f = 2 + \frac{1}{3}(X_1 + X_2)$.

7.2.3 Extension to UPoPS

A univariate polynomial with multivariate power series coefficients, i.e. a univariate polynomial over power series (UPoPS), is implemented as a simple extension of our existing power series. Following a simple dense univariate polynomial design, our UPoPS are represented as an array of coefficients, each being a pointer to a power series, where the index of the coefficient in the array implies the degree of the coefficient's associated monomial. Integers are also stored for the degree of the polynomial and the allocation size of the coefficient array. In support of the underlying lazy power series, we also add reference counting to UPoPS. The UPoPS struct can be seen in Listing 7.7.

```
typedef struct UPOPS {
    int deg;
    int alloc;
    PowerSeries_t** data;
    PowerSeries_t** weierstrassFData; //see Section 7.3
    int fDataSize;
    int refcount;
} UPOPS_t;
```

Listing 7.7: The univariate polynomial over power series struct.

The arithmetic of UPoPS is inherited directly from its coefficient ring (our lazy power series) and follows a naive implementation of univariate polynomials (see, e.g. [86, Ch. 2]). Through the use of our lazy power series, our implementation of UPoPS is automatically lazy through each individual coefficient's ancestry. Lazy UPoPS addition, subtraction, and multiplication follow easily.

One important operation on UPoPS which is not inherited directly from our power series implementation is Taylor shift. This operation takes a UPoPS $f \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ and returns $f(Y + c)$ for some constant $c \in \mathbb{K}$. Normally, the shift operator would be defined for any element of the ground ring $\mathbb{K}[[X_1, \ldots, X_n]]$, however our use of Taylor shift in applying Hensel's lemma requires only shifting by elements of $\mathbb{K}$, and we thus specialize to that case.

Given $f = \sum_{i=0}^{d} a_i Y^i$ we want to obtain $f(Y + c) = \sum_{i=0}^{d} a_i (Y + c)^i$. Since the coefficients of f are lazy power series, our goal is to compute $f(Y + c)$ lazily as well. That is, to compute $f(Y + c)$ in a way which relies on the underlying lazy power series arithmetic to yield a lazily computed UPoPS. Since our UPoPS are represented in a dense

fashion, we compute the coefficients of $f(Y+c)$ as a polynomial in Y:

$$\begin{aligned} f(Y+c) = &\ a_0 + a_1(Y+c) + a_2(Y+c)^2 + a_3(Y+c)^3 + \ldots \\ = &\ (a_0 + ca_1 + c^2a_2 + c^3a_3 + \ldots) \\ &+(a_1 + 2ca_2 + 3c^2a_3 + \ldots)Y \\ &+(a_2 + 3ca_3 + \ldots)Y^2 \\ &+(a_3 + \ldots)Y^3 + \ldots \end{aligned}$$

The coefficients of the expansion of $f(Y+c)$ create a triangular shape of linear combinations of the original coefficients of f. These linear combinations arise from the binomial expansion of $(Y+c)^i$ and are closely related to the Pascal triangle.

Let $\mathbf{S} = (s_{i,j})$ be the lower triangular matrix such that $s_{i,j}$ is the coefficient of Y^j in the binomial expansion $(Y+c)^i$, for $i = 0, \ldots, d$, and $j = 0, \ldots, i$, where $d = \deg(f)$. Let $\mathbf{A} = (a_i)$ be the vector of the coefficients of f and $\mathbf{B} = (b_i)$ be the vector of the coefficients of $f(Y+c)$, so that we have $f(Y) = \sum_{0 \le i \le d} a_i Y^i$ and $f(Y+c) = \sum_{0 \le i \le d} b_i Y^i$.

Then we can verify that b_i is the inner product of the i-th sub-diagonal of $\mathbf{S}$ with the lower $d+1-i$ elements of $\mathbf{A}$, where d is the degree of f, for $i = 0, \ldots, d$. In particular, for $i = 0$, the coefficient b_0 is the inner product of the diagonal of $\mathbf{S}$ and the vector $\mathbf{A}$.

For example, with $d = 3$, we have:

$$\mathbf{S} = \begin{bmatrix} 1 & & & \\ 1 & c & & \\ 1 & 2c & c^2 & \\ 1 & 3c & 3c^2 & c^3 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Recalling that $c \in \mathbb{K}$, the construction of b_i can be performed in a graded fashion from the linear combinations of homogeneous parts of a_j for $j \le i$. The homogeneous part $b_{i_{(k)}}$ of degree k, can be computed from only $a_{j_{(k)}}$, for $j \le i$. Therefore, a generator for b_i is easily constructed from the homogeneous parts of a_j, for $j \le i$, using multiplication by elements of $\mathbb{K}$ and polynomial addition. Therefore, we can construct the entire UPoPS $f(Y+c)$ in a lazy manner through initializing each coefficient b_i with a so-called linear combination generator, see Algorithm 7.1.

Algorithm 7.1 TaylorShiftUpdate(k, f, $\mathbf{S}$, i)

Input: For $f = \sum_{j=0}^{d} a_j Y^j$, $g = f(Y+c) = \sum_{j=0}^{d} b_j Y^j$, obtain the homogeneous part of degree k for b_i. $\mathbf{S} \in \mathbb{K}^{(d+1)\times(d+1)}$ is a lower triangular matrix of coefficients of $(Y+c)^j$ for $j = 0, \ldots, d$,

Output: $b_{i(k)}$, the homogeneous part of degree k of b_i.

1: $b_{i_{(k)}} := 0$
2: **for** $\ell := i$ to d **do**
3: $\quad j := \ell + 1 - i$
4: $\quad b_{i(k)} := b_{i(k)} + S_{\ell+1,j} \times a_{\ell(k)}$
5: **return** $b_{i(k)}$

Observation 7.6 (Taylor shift complexity)**.** For a UPoPS $f = \sum_{i=0}^{d} a_i Y^i \in \mathbb{K}[[X_1]][Y]$, computing the homogeneous part of degree k for all coefficients of the shifted UPoPS $f(Y+c)$ requires $d^2 + 2d + 1$ operations in $\mathbb{K}$.

Proof. Computing $f(Y+c)$ requires updating $d+1$ power series coefficients via TaylorShiftUpdate. Computing the homogeneous part of degree k of the ith coefficient of $f(Y+c)$ requires $2d-2i+1$ operations in $\mathbb{K}$: $d-i+1$ multiplications and $d-i$ additions. Summing over i from 0 to d yields d^2+2d+1. □

Since the main application of Taylor shift is factorization via Hensel's lemma, we leave its evaluation to Section 7.4 where benchmarks for factorization are presented.

7.3 Lazy Weierstrass Preparation

Let f, p, $\alpha \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ where $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, and $\alpha = \sum_{i=0}^{m} c_i Y^i$. From the proof of WPT (Theorem 7.3), we have that $f = \alpha p$ implies the following equalities:

$$\begin{aligned}
a_0 &= b_0 c_0 \\
a_1 &= b_0 c_1 + b_1 c_0 \\
&\vdots \\
a_{d-1} &= b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\
a_d &= b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\
&\vdots \\
a_{d+m-1} &= b_{d-1} c_m + c_{m-1} \\
a_{d+m} &= c_m
\end{aligned} \tag{7.2}$$

Following the proof, we wish to solve these equations modulo successive powers of $\mathcal{M}$, the maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$. This implies that we will be iteratively updating each power series $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ by adding homogeneous polynomials of increasing degree, precisely as we have done for all lazy power series operations thus far. To solve these equations modulo $\mathcal{M}^{r+1}$, both the proof of WPT and the algorithm operates in two phases. First, the coefficients $b_0, \ldots, b_{d-1}$ of p are updated using the equations from a_0 to a_{d-1}, one after the other. Second, the coefficients $c_0, \ldots, c_m$ of α are updated.

Let us begin with the first phase. Rearranging the equations that express a_0 to a_{d-1} shows their successive dependency where b_{i-1} is needed for b_i:

$$\begin{aligned} a_0 &= b_0c_0 \\ a_1 - b_0c_1 &= b_1c_0 \\ a_2 - b_0c_2 - b_1c_1 &= b_2c_0 \\ &\vdots \\ a_{d-1} - b_0c_{d-1} - b_1c_{d-2} + \cdots - b_{d-2}c_1 &= b_{d-1}c_0 \end{aligned} \tag{7.3}$$

Consider that $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ are known modulo $\mathcal{M}^r$ and $a_0, \ldots, a_{d-1}$ are known modulo $\mathcal{M}^{r+1}$. Using Lemma 7.2 the first equation $a_0 = b_0c_0$ can then be solved for b_0 modulo $\mathcal{M}^{r+1}$. From there, the expression $a_1 - b_0c_1$ then becomes known modulo $\mathcal{M}^{r+1}$. Notice that the constant term of b_0 is 0 by definition, thus the product b_0c_1 is known modulo $\mathcal{M}^{r+1}$ as long as b_0 is known modulo $\mathcal{M}^{r+1}$. Therefore, the entire expression $a_1 - b_0c_1$ is known modulo $\mathcal{M}^{r+1}$ and Lemma 7.2 can be applied to solve for b_1 in the equation $a_1 - b_0c_1 = b_1c_0$. This argument follows for all equations, therefore solving for all $b_0, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$.

In the second phase, we look to determine $c_0, \ldots, c_m$ modulo $\mathcal{M}^{r+1}$. Here, we have already computed $b_0, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$. A rearrangement of the remaining equations of (7.2) shows that each c_i may be computed modulo $\mathcal{M}^{r+1}$:

$$\begin{aligned} c_m &= a_{d+m} \\ c_{m-1} &= a_{d+m-1} - b_{d-1}c_m \\ c_{m-2} &= a_{d+m-2} - b_{d-2}c_m - b_{d-1}c_{m-1} \\ &\vdots \\ c_0 &= a_d - b_0c_d - b_1c_{d-1} - \cdots - b_{d-1}c_1 \end{aligned} \tag{7.4}$$

Consider the second equation. Observe that a_{d+m-1} and b_{d-1} are known modulo $\mathcal{M}^{r+1}$ and that $b_{d-1} \in \mathcal{M}$ holds. Then, the product $b_{d-1}c_m$ is known modulo $\mathcal{M}^{r+1}$ and we deduce c_{m-1} modulo $\mathcal{M}^{r+1}$. The same follows for $c_{m-2}, \ldots, c_0$.

With these two sets of re-arranged equations, we have seen how the coefficients of p and α can be updated modulo successive powers of $\mathcal{M}$. That is to say, how they can be updated by adding homogeneous parts of successive degrees. This design lends itself to be implemented as generator functions.

The first challenge to this design is that each power series coefficient of p is not independent, and must be updated in a particular order. Moreover, to generate homogeneous parts of degree k for the coefficients of p, the coefficients of α must also be updated to degree $k-1$. Therefore, it is a required side effect of each generator of $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ that all other power series are updated. To implement this, the generators of the power series of p are a mere wrapper of the same underlying updating function which updates all coefficients simultaneously. This so-called *Weierstrass update* follows two phases as just explained.

In the first phase, one must use Lemma 7.2 to solve for the homogeneous part of degree r for each $b_0, \ldots, b_{d-1}$. To achieve this effectively, our implementation follows two key points. The first is an efficient implementation of Lemma 7.2 itself. Consider again the equations of Lemma 7.2 for $f = gh$ modulo $\mathcal{M}^{r+1}$:

$$\begin{aligned} f_{(1)} + f_{(2)} + \cdots + f_{(r)} &= (g_{(1)} + g_{(2)} + \cdots + g_{(r)})(h_{(0)} + h_{(1)} + \cdots + h_{(r)}) \\ &= \left(g_{(1)}h_{(0)}\right) + \left(g_{(2)}h_{(0)} + g_{(1)}h_{(1)}\right) + \cdots + \\ &\quad \left(g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}\right). \end{aligned} \tag{7.5}$$

The goal is to obtain $g_{(r)}$. What one should realize is that computing $g_{(r)}$ requires only a fraction of this formula. In particular, we have

$$f_{(r)} = g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}, \tag{7.6}$$

and $g_{(r)}$ can be computed with simply polynomial addition and multiplication, followed by the division of a single element of $\mathbb{K}$, since $h_{(0)}$ has degree 0.

The second key point is that, in order to compute $g_{(r)}$, i.e. the homogeneous parts of degree r of $b_0, \ldots, b_{d-1}$, we must first find $f_{(r)}$, i.e. the homogeneous parts of degree r of $a_0, a_1 - b_0c_1$, $a_2 - b_0c_2 - b_1c_1$, etc. from (7.3). A nice result of our existing power series design is that we can define some lazy power series, say F_i, such that $F_i = a_i - \sum_{j=0}^{i} b_j c_{i-j}$. These F_i can then be automatically updated via its generators when the b_k are updated. The implementation of phase one of Weierstrass update is then simply a loop over solving equation (7.6), where $f_{(r)}$ is automatically obtained through the use of generators on the power series F_i.

Phase two of Weierstrass update follows the same design as in the definition of those

F_i power series. In particular, from (7.4) we can see that each $c_m, \ldots, c_0$ is merely the result of some power series arithmetic. Hence, we simply rely on the underlying power series arithmetic generators to be the generators of $c_m, \ldots, c_0$.

With the above discussion, we have fully defined a lazy implementation of Weierstrass preparation. It begins with an initialization, which simply uses lazy power series arithmetic to create $F_0, \ldots, F_{d-1}, c_m, \ldots, c_0$, and initializes each $b_0, \ldots, b_{d-1}$ to 0. Then, the generators for $b_0, \ldots, b_{d-1}$ all call the same underlying Weierstrass update function. This function is shown in Algorithm 7.2, which is split into two phases as our discussion has suggested. Note that a pointer to the F_i's are stored in the UPoPS struct, see Listing 7.7.

Algorithm 7.2 WEIERSTRASSUPDATE(k, f, p, α)

Input: $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, $\alpha = \sum_{i=0}^{m} c_i Y^i$, $a_i, b_i, c_i \in \mathbb{K}[[X_1, \ldots, X_n]]$ satisfying Theorem 7.3, with $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^k$, $\mathcal{M}$ the maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$.

Output: $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^{k+1}$, updated in-place.

1: **for** $i := 0$ to $d-1$ **do** ▷ phase 1
2: $\quad F_{i(k)} := a_{i(k)}$
3: $\quad$ **if** $i \leq m$ **then**
4: $\quad\quad$ **for** $j := 0$ to $i-1$ **do**
5: $\quad\quad\quad F_{i(k)} := F_{i(k)} - (b_j\, c_{i-j})_{(k)}$
6: $\quad$ **else**
7: $\quad\quad$ **for** $j := 0$ to $m-1$ **do**
8: $\quad\quad\quad F_{i(k)} := F_{i(k)} - (b_{i+j-m}\, c_{m-j})_{(k)}$
9: $\quad s := 0$
10: $\quad$ **for** $j := 1$ to $k-1$ **do**
11: $\quad\quad s := s \; + \; b_{i(k-j)} \times c_{0(j)}$
12: $\quad b_{i(k)} := \left(F_{i(k)} - s\right) / c_{0(0)}$
13: $c_{m(k)} := a_{d+m(k)}$ ▷ phase 2
14: **for** $i := 1$ to m **do**
15: $\quad$ **if** $i \leq d$ **then**
16: $\quad\quad c_{m-i(k)} := a_{d+m-i(k)} - \sum_{j=1}^{i} (b_{d-j} c_{m-i+j})_{(k)}$
17: $\quad$ **else**
18: $\quad\quad c_{m-i(k)} := a_{d+m-i(k)} - \sum_{j=1}^{d} (b_{d-j} c_{m-i+j})_{(k)}$

7.3.1 The Complexity of Weierstrass Preparation

From the proof of Weierstrass preparation (Theorem 7.3), we derive WeierstrassUpdate (Algorithm 7.2). That proof proceeds modulo increasing powers of the maximal ideal $\mathcal{M}$, For an application of Weierstrass preparation producing p and α, this WeierstrassUpdate acts as the update function for p and α, updating both simultaneously.

By rearranging the first d equations of (7.2) and applying Lemma 7.2 we obtain "phase 1" of WeierstrassUpdate, where each coefficient of p is updated. By rearranging the next $m+1$ equations of (7.2) we obtain "phase 2" of WeierstrassUpdate, where each coefficient of α is updated. From Algorithm 7.2, it is then routine to show the following two observations, which lead to Theorem 7.9.

Observation 7.7 (Weierstrass phase 1 complexity)**.** For WeierstrassUpdate over $\mathbb{K}[[X_1]][Y]$, computing $b_{i(k)}$, for $0 \leq i < d$, requires $2ki + 2k - 1$ operations in $\mathbb{K}$ if $i \leq m$, or $2km + 2k - 1$ operations in $\mathbb{K}$ if $i > m$.

Proof. Over $\mathbb{K}[[X_1]][Y]$, the homogeneous part of a coefficient is simply an element of $\mathbb{K}$. Computing s (Lines 9–11) requires $k-1$ multiplications and $k-2$ additions in $\mathbb{K}$. Computing $(b_j c_{i-j})_{(k)}$ requires $2k-1$ operations in $\mathbb{K}$ (Observation 7.1). For $i \leq m$ there are i such product homogeneous parts computed and i subtractions in $\mathbb{K}$ to compute $F_{i(k)}$. For $i > m$ there are m such product homogeneous parts computed and m subtractions in $\mathbb{K}$ to compute $F_{i(k)}$. Finally, 2 operations in $\mathbb{K}$ are required to compute $b_{i(k)}$ from $F_{i(k)}$, s, and $c_{0(0)}$. For $i < m$ the total is $2ki + 2k - 1$, for $i \geq m$ the total is $2km + 2k - 1$. □

Observation 7.8 (Weierstrass phase 2 complexity)**.** For WeierstrassUpdate over $\mathbb{K}[[X_1]][Y]$, computing $c_{m-i(k)}$, for $0 \leq i < m$, requires $2ki$ operations in $\mathbb{K}$ if $i \leq d$, or $2kd$ operations in $\mathbb{K}$ if $i > d$.

Proof. Computing $(b_{d-j} c_{m-i+j})_{(k)}$ requires $2k-1$ operations in $\mathbb{K}$ by Observation 7.1. For $i \leq d$ there are i such product homogeneous parts and $i-1$ additions in $\mathbb{K}$. For $i > d$ there are d such product homogeneous parts and $d-1$ additions in $\mathbb{K}$. Computing $a_{d+m-1(k)}$ has no cost, since it is the input. Finally, one subtraction in $\mathbb{K}$ finishes the computation of $c_{m-i(k)}$. Hence, for $i \leq d$ the total is $2ki$, for $i > d$ the total is $2kd$. □

Theorem 7.9 (Weierstrass preparation complexity)**.**
Weierstrass preparation producing $f = p\,\alpha$, with $f, p, \alpha \in \mathbb{K}[[X_1]][Y]$, $\deg(p) = d$, $\deg(\alpha) = m$, requires $dmk^2 + dk^2 + dmk$ operations in $\mathbb{K}$ to compute p and α to precision k.

PROOF. Let i be the index of a coefficient of p or α. Consider the cost of computing the homogeneous part of degree k of each coefficient of p and α. First consider $i < t = \min(d, m)$. From Observations 7.7 and 7.8, computing the kth homogeneous part of each b_i and c_i respectively requires $2ki + 2k - 1$ and $2ki$ operations in $\mathbb{K}$. For $0 \leq i < t$, this yields a total of $2kt^2 + 2kt - t$. Next, we have three cases: (a) $t = d = m$, (b) $m = t < i < d$, or (c) $d = t < i < m$. In case (a) there is no additional work. In case (b), phase 1 contributes an additional $(d - m)(2km + 2k - 1)$ operations. In case (c), phase 2 contributes an additional $(m-d)(2kd)$ operations. In all cases, the total number of operations to update p and α from precision $k - 1$ to precision k is $2dmk + 2dk - d$. Finally, to compute p and α up to precision k requires $dmk^2 + dk^2 + dmk$ operations in $\mathbb{K}$. □

A useful consideration is when the input to Weierstrass preparation is monic. This necessarily makes α monic, and the overall complexity of Weierstrass preparation is reduced. This case arises for each application of Weierstrass preparation in Hensel factorization. The following corollary proves this, following Theorem 7.9.

Corollary 7.10 (Weierstrass preparation complexity for monic input)**.** Weierstrass preparation producing $f = p\,\alpha$, with $f, p, \alpha \in \mathbb{K}[[X_1]][Y]$, f monic in Y, $\deg(p) = d$ and $\deg(\alpha) = m$, computing p and α up to precision k requires $dmk^2 + dmk$ operations in $\mathbb{K}$.

PROOF. If f is monic then α is necessarily monic and $c_m = 1$. For $i \geq m$ we save computing $(b_{i-m}c_m)_{(k)}$ for the update of $b_{i(k)}$. For $1 \leq i \leq d$ we save computing $(b_{d-j}c_{m-i+j})_{(k)}$ for $j = i$ for the update of each $c_{m-i(k)}$. First, consider updating p and α from precision $k - 1$ to precision k. Let $t = \min(d, m)$. We have three cases: (a) $t = d = m$, (b) $m = t < i < d$, or (c) $d = t < i < m$. In case (a) we save $d(2k - 1)$ operations in phase 2, as compared to case (a) from the proof of Theorem 7.9. In case (b) we save $(d - m)(2k - 1)$ operations in phase 1 and $m(2k - 1)$ operations in phase 2. In case (c) we save $d(2k - 1)$ operations in phase 2. In all cases we save a total of $d(2k-1)$ operations, resulting in $2dmk$ operations in $\mathbb{K}$ to update p and α from precision $k-1$ to precision k. Finally, to compute p and α up to precision k requires $dmk^2 + dmk$ operations in $\mathbb{K}$. □

7.4 Lazy Hensel Factorization

Recall that the proof of Theorem 7.5 provides a mechanism to factor a UPoPS $f \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ into factors $f_1, \ldots, f_r$ based on Taylor shift and repeated applications

of Weierstrass preparation. The construction begins by first factorizing the polynomial $\bar{f} = f(0, \ldots, 0, Y) \in \mathbb{K}[Y]$, obtained by evaluating all variables in power series coefficients to 0. This can be performed with a suitable (algebraic) factorization algorithm for $\mathbb{K}$. For simplicity of presentation, let us assume that $\bar{f}$ factorizes into linear factors over $\mathbb{K}$, thus returning a list of roots $c_1, \ldots, c_r \in \mathbb{K}$ with respective multiplicities $k_1, \ldots, k_r$[1] The construction then proceeds recursively, obtaining one factor at a time.

Let us describe one step of the recursion, where f^* describes the current polynomial to factorize, initially being set to f. For a root c_i of $\bar{f}$, we perform a Taylor shift to obtain $g = f^*(Y + c_i)$ such that g has order k_i (as a polynomial in Y). The Weierstrass preparation theorem can then be applied to obtain p and $\alpha \in K[[X_1, \ldots, X_n]][Y]$ where p is monic and of degree k_i. A Taylor shift is then applied in reverse to obtain $f_i = p(Y - c_r)$, a factor of f, and $f^* = \alpha(Y - c_r)$, the UPoPS to factorize in the next step. The full procedure for obtaining all factors of f is shown as an iterative process, instead of recursive, in Algorithm 7.3.

Algorithm 7.3 HenselFactorization(f)

Input: $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i, a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$.
Output: $f_1, \ldots, f_r$ satisfying Theorem 7.5.

$\bar{f} = f(0, \ldots, 0, Y)$
$(c_1, \ldots, c_r), (d_1, \ldots, d_r) :=$ roots and their multiplicities of $\bar{f}$
$c_1, \ldots, c_r :=$ sort$([c_1, \ldots, c_r])$ by increasing multiplicity ▷ see Theorem 7.12
$\hat{f}_1 := f$
for $i := 1$ to $r - 1$ **do**
  $g_i := \hat{f}_i(Y + c_i)$
  $p_i, \alpha_i :=$ WeierstrassPreparation(g)
  $f_i := p_i(Y - c_i)$
  $\hat{f}_{i+1} := \alpha_i(Y - c_i)$
$f_r := \hat{f}_r$
return $f_1, \ldots, f_r$

The beauty of this algorithm is that it is immediately a lazy algorithm with no additional effort. Using the underlying lazy operations of Taylor shift (Algorithm 7.1) and Weierstrass preparation (Algorithm 7.2) the entire factorization is performed lazily, returning a factorization nearly instantly. The power series coefficients of these factors can automatically be updated later using their generators, which are simply Taylor shift operations on top of a Weierstrass update.

[1] Our implementation in C requires that $\bar{f}$ factors into linear factors over $\mathbb{Q}$. The case where $\bar{f}$ has roots in $\overline{\mathbb{Q}}$ is handled in our *Maple* implementation [10]

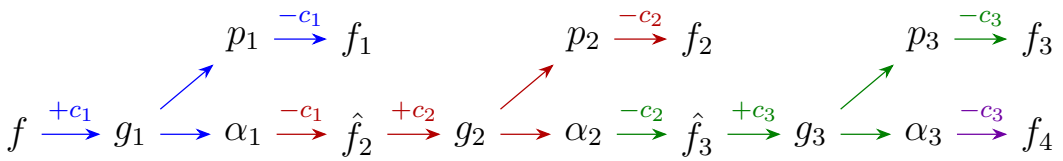

Figure 7.6: The ancestor chain for the Hensel factorization $f = f_1 f_2 f_3 f_4$. Updating f_1 requires updating g_1, p_1, α_1; then updating f_2 requires updating $\hat{f}_2, g_2, p_2, \alpha_2$; then updating f_3 requires updating $\hat{f}_3, g_3, p_3, \alpha_3$; then updating f_4 requires only its own Taylor shift. These groupings form the eventual stages of the Hensel pipeline (Algorithm 7.8).

Note that factors are sorted by increasing degree to enable better load-balance in the eventual parallel algorithm. Fig. 7.6 shows the chain of ancestors created by the Hensel factorization $f = f_1 f_2 f_3 f_4$ and the grouping of ancestors required to update each factor.

7.4.1 The Complexity of Hensel Factorization

Having specified the update functions for WPT and Taylor shift, we saw that a lazy scheme for Hensel factorization was immediate, requiring only the appropriate chain of ancestors. The section summarizes the complexity estimates related to Hensel factorization. They culminate in Corollary 7.13 and and Corollary 7.14 which yield the cost to increase the precision by 1, and the total complexity of Hensel factorization. Here, we ignore the initial cost of factorizing $\bar{f}$ (see the beginning of Chapter 7 for that discussion).

First, we analyze the complexity of HENSELFACTORIZATION for the common case where each factor has degree 1. That is, when the multiplicity of each root of $\bar{f}$ is 1.

Theorem 7.11 (Hensel factorization complexity for simple roots)**.** Applying HENSELFACTORIZATION on $f \in \mathbb{K}[[X_1]][Y]$, where $\deg(f) = d$, with all resulting factors having degree 1, and updating each factor to precision k, requires $^2/_3\, d^3k + {}^1/_2\, d^2k^2 + {}^5/_2\, d^2k - {}^1/_2\, dk^2 + {}^{35}/_6\, dk - 9k$ arithmetic operations in $\mathbb{K}$.

PROOF. For each factor except the last, HENSELFACTORIZATION requires one Taylor shift, one Weierstrass preparation, and two more Taylor shifts. For the first factor we have that the first Taylor shift is of degree d, the Weierstrass preparation produces p_1 and α_1 of degree 1 and $d-1$, respectively, and then the two Taylor shifts are of degree 1 and $d-1$. This pattern continues for each factor but the last. f_d is obtained from the shifted α_{d-1}. The result is: a shift of degree $d-i+1$ for $i = 1, \ldots, d-1$ (for each $\hat{f}_i$), $d-1$ shifts of degree 1 (for each p_i), and a shift of degree $d-i$ for $i = 1, \ldots, d-1$ (for each α_i). From Observation 7.6, obtaining a Taylor shift of degree d' to precision k requires ${d'}^2k + 2d'k + k$ operations in $\mathbb{K}$. Summing over each group of Taylor shifts gives,

respectively, $k\,(1/3\,d^3 + 3/2\,d^2 + 13/6\,d - 4)$, $4k(d-1)$, and $k\,(1/3\,d^3 + 1/2\,d^2 + 1/6\,d - 1)$, for a total of $k\,(2/3\,d^3 + 2d^2 + 19/3\,d - 9)$ operations in $\mathbb{K}$.

The remaining operations arise from the repeated Weierstrass preparations. For i from 1 to $d-1$ we apply Weierstrass preparations to produce p_i, α_i pairs of respective degree $1, d-i$. From Corollary 7.10 we have that each such Weierstrass preparation requires $(d-i)k^2 + (d-i)k$ operations in $\mathbb{K}$. Summing over $i = 1, \ldots, d-1$ yields $1/2\,(d^2k^2 + d^2k - dk^2 - dk)$. Finally, combining this with the previous Taylor shift costs leads to the desired result. □

Theorem 7.12 (Hensel factorization complexity per factor)**.** Let $\hat{d}_i$ be the degree of $\hat{f}_i$ during HenselFactorization applied to $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$. To update f_1, $\deg(f_1) = d_1$ to precision k requires $d_1\hat{d}_2k^2 + d^2k + d_1dk + 2d_1k + 2dk + 2k$ arithmetic operations in $\mathbb{K}$. To update f_i, $\deg(f_i) = d_i$, for $1 < i < r$, to precision k requires $d_i\hat{d}_{i+1}k^2 + 2\hat{d}_i^2k + d_i\hat{d}_ik + 2d_ik + 4\hat{d}_ik + 3k$ arithmetic operations in $\mathbb{K}$. To update f_r, $\deg(f_r) = d_r$, to precision k requires $d_r^2k + 2d_rk + k$ arithmetic operations in $\mathbb{K}$.

Proof. Updating the first factor produced by HenselFactorization requires one Taylor shift of degree d, one Weierstrass preparation producing p_1 and α_1 of degree d_1 and $\hat{d}_2 = d - d_1$, and one Taylor shift of degree d_1 to obtain f_1 from p. From Observation 7.6 and Corollary 7.10 we have that the Taylor shifts require $k(d^2+2d+1) + k(d_1^2+2d_1+1)$ operations in $\mathbb{K}$ and the Weierstrass preparation requires $d_1(d-d_1)k^2 + d_1(d-d_1)k$ operations in $\mathbb{K}$. The total cost counted as operations in $\mathbb{K}$ is thus $d_1\hat{d}_2k^2 + d^2k + d_1dk + 2d_1k + 2dk + 2k$.

Updating each following factor, besides the last, requires one Taylor shift of degree $\hat{d}_i$ to update $\hat{f}_i$ from α_{i-1}, one Taylor shift of degree $\hat{d}_i$ to update g_i from $\hat{f}_i$, one Weierstrass preparation to obtain p_i and α_i of degree d_i and $\hat{d}_{i+1} = \hat{d}_i - d_i$, and one Taylor shift of degree d_i to obtain f_i from p_i. The Taylor shifts require $2k(\hat{d}_i^2 + 2\hat{d}_i + 1) + k(d_i^2 + 2d_i + 1)$ operations in $\mathbb{K}$. The Weierstrass preparation requires $d_i(\hat{d}_i - d_i)k^2 + d_i(\hat{d}_i - d_i)k$ operations in $\mathbb{K}$. The total cost counted as operations in $\mathbb{K}$ is thus $d_i\hat{d}_{i+1}k^2 + 2\hat{d}_i^2k + d_i\hat{d}_ik + 2d_ik + 4\hat{d}_ik + 3k$.

Finally, updating the last factor to precision k requires a single Taylor shift of degree d_r costing $d_r^2k + 2d_rk + k$ operations in $\mathbb{K}$. □

Corollary 7.13 (Hensel factorization complexity per iteration)**.** Let $\hat{d}_i$ be the degree of $\hat{f}_i$ during the HenselFactorization algorithm applied to $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$. Computing the kth homogeneous part of f_1, $\deg(f_1) = d_1$, requires $2d_1\hat{d}_2k + d_1^2 + d^2 + 2d_1 + 2d + 2$ operations in $\mathbb{K}$. Computing the kth homogeneous part of f_i, $\deg(f_i) = d_i$,

$1 < i < r$, requires $2d_i\hat{d}_{i+1}k + d_i^2 + 2\hat{d}_i^2 + 4\hat{d}_i + 2d_i + 3$ operations in $\mathbb{K}$. Computing the kth homogeneous part of f_r, $\deg(f_r) = d_r$, requires $d_r^2 + 2d_r + 1$ operations in $\mathbb{K}$.

PROOF. Follows from Observation 7.6, Corollary 7.10, and Theorem 7.12. □

Corollary 7.14 (Hensel factorization complexity)**.** HENSELFACTORIZATION producing $f = f_1 \cdots f_r$, with $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$, requires $\mathcal{O}(d^3k + d^2k^2)$ operations in $\mathbb{K}$ to update all factors to precision k.

PROOF. Let $f_1, \ldots, f_r$ have respective degrees $d_1, \ldots, d_r$. Let $\hat{d}_i = \sum_{j=i}^{r} d_j$ (thus $\hat{d}_1 = d$ and $\hat{d}_r = d_r$). From Theorem 7.12, each f_i, $1 \leq i < r$ requires $\mathcal{O}(d_i\hat{d}_{i+1}k^2 + \hat{d}_i^2k)$ operations in $\mathbb{K}$ to be updated to precision k (or $\mathcal{O}(d_r^2k)$ for f_r). We have $\sum_{i=1}^{r-1} d_i\hat{d}_{i+1} \leq \sum_{i=1}^{r-1} d_i d < d^2$ and $\sum_{i=1}^{r} \hat{d}_i^2 \leq \sum_{i=1}^{r} d^2 = rd^2 \leq d^3$. Hence, all factors can be updated to precision k within $\mathcal{O}(d^3k + d^2k^2)$ operations in $\mathbb{K}$. □

Corollary 7.14 shows that the two dominant terms in the cost of computing a Hensel factorization of a UPoPS of degree d, up to precision k, are d^3k and d^2k^2. From the proof of Theorem 7.12, the former term arises from the cost of the Taylor shifts in Y, meanwhile, the latter term arises from the (polynomial) multiplication of homogeneous parts in Weierstrass preparation. This observation then leads to the following conjecture. Recall that $M(n)$ denotes a polynomial multiplication time [86, Ch. 8]. From [98], relaxed algorithms, which improve the performance of lazy evaluation schemes, can be used to compute a power series product in $\mathbb{K}[[X_1]]$ up to precision k in at most $\mathcal{O}(M(k)\log k)$ operations in $\mathbb{K}$ (or less, in view of the improved relaxed multiplication of [97]).

Conjecture 7.15. Let $f \in \mathbb{K}[[X_1]][Y]$ factorize as $f_1 \cdots f_r$ using HENSELFACTORIZATION. Let $\deg(f) = d$. Updating the factors $f_1, \ldots, f_r$ to precision k using relaxed algorithms requires at most $\mathcal{O}(d^3k + d^2M(k)\log k)$ operations in $\mathbb{K}$.

Comparatively, the Hensel–Sasaki Construction requires at most $\mathcal{O}(d^3M(d)+dM(d)k^2)$ operations in $\mathbb{K}$ to compute the first k terms of all factors of $f \in \mathbb{K}[X_1, Y]$, where f has total degree d [4]. The method of Kung and Traub [118], requires $\mathcal{O}(d^2M(k))$. Already, Corollary 7.14—where $d = \deg(f, Y)$—shows that our Hensel factorization is an improvement on Hensel–Sasaki (d^2k^2 versus $dM(d)k^2$). If Conjecture 7.15 is true, then Hensel factorization can be within a factor of $\log k$ of Kung and Traub's method. Nonetheless, this conjecture is highly encouraging toward future practical performance improvements, where $k \gg d$. This is particularly true where we have already seen that our current (and suboptimal) method performs better in practice than Hensel–Sasaki and the method of Kung and Traub [32]. Proving this conjecture is left to future work.

7.5 Parallel Algorithms

The previous two sections presented lazy algorithms for Weierstrass preparation, and Hensel factorization. It also presented complexity estimates for those algorithms. Those estimates will soon be used to help dynamically distribute hardware resources (threads) in a parallel variation of Hensel factorization; in particular, a Hensel factorization pipeline where each pipeline stage updates one or more factors, see Algorithms 7.7–7.9. But first, we will examine parallel processing techniques for Weierstrass preparation.

7.5.1 Parallel Algorithms for Weierstrass Preparation

Algorithm 7.2 shows that p and α from a Weierstrass preparation can be updated in two phases: p in phase 1, and α in phase 2. Ultimately, these updates rely on the computation of the homogeneous part of some power series product. Algorithm 7.4 presents a simple map-reduce pattern (see Section 5.1) for computing such a homogeneous part. Moreover, this algorithm is designed such that, recursively, all ancestors of a power series product are also updated using parallelism. Note that UpdateToDegParallel called on a UPoPS simply recurses on each of its coefficients.

Algorithm 7.4 UpdateToDegParallel(k, f, t)

Input: A positive integer k, $f \in \mathbb{K}[[X_1, \ldots, X_n]]$ known to at least precision $k-1$. If f has ancestors, it is the result of a binary operation. A positive integer t for the number of threads to use.

Output: f is updated to precision k, in place.

```
1: if f_(k) already computed then
2: |  return
3: g, h := FirstAncestor(f), SecondAncestor(f)
4: UpdateToDegParallel(k, g, t);
5: UpdateToDegParallel(k, h, t);
6: if f is a product then
7: |  V := [0, ..., 0]                                  ▷ 0-indexed list of size t
8: |  parallel_for j := 0 to t − 1
9: |  |  for i := jk/t to (j+1)k/t − 1 while i ≤ k do
10: |  |  |  V[j] := V[j] + g_(i) h_(k−i)
11: |  f_(k) := Σ_{j=0}^{t−1} V[j]                      ▷ reduce
12: else if f is a p from a Weierstrass preparation then
13: |  WeierstrassPhase1Parallel(k,g,f,h,WeierstrassData(f),t)
14: else if f is an α from a Weierstrass preparation then
15: |  WeierstrassPhase2Parallel(k, g, h, f, t)
16: else
17: |  UpdateToDeg(k, f)
```

Using the notation of Algorithm 7.2, recall that, e.g., $F_i := a_i - \sum_{j=0}^{i-1}(b_j c_{i-j})$, for $i \leq m$. Using lazy power series arithmetic, this entire formula can be encoded by a chain of ancestors, and one simply needs to update F_i to trigger a cascade of updates through its ancestors. In particular, using Algorithm 7.4, the homogeneous part of each product $b_j c_{i-j}$ is recursively computed using map-reduce. Similarly, Lemma 7.2 can be implemented using map-reduce (see Algorithm 7.5) to replace Lines 9–12 of Algorithm 7.2. Phase 1 of Weierstrass, say WEIERSTRASSPHASE1PARALLEL, thus reduces to a loop over i from 0 to $d-1$, calling Algorithm 7.4 to update F_i to precision k, and calling Algorithm 7.5 to compute $b_{i(k)}$.

Algorithm 7.4 uses several simple subroutines: FIRSTANCESTOR and SECONDANCESTOR gets the first and second ancestor of a power series, WEIERSTRASSDATA gets a reference to the list of F_i's, and UPDATETODEG calls the serial update function of a lazy power series to ensure its precision is at least k; see Section 7.2.

Now consider phase 2 of WEIERSTRASSUPDATE. Notice that computing the homogeneous part of degree k for c_{m-i}, $0 \leq i \leq m$ only requires each c_{m-i} to be known up to precision $k-1$, since each $b_j \in \mathcal{M}$ for $0 \leq j < d$. This implies that the phase 2 **for** loop of WEIERSTRASSUPDATE has independent iterations. We thus apply the map pattern directly to this loop itself, rather than relying on the map-reduce pattern of UPDATETODEGPARALLEL. However, consider the following two facts: the cost of computing each c_{m-i} is different (Observation 7.8 and Corollary 7.10), and, for a certain number of available threads t, it may be impossible to partition the iterations of the loop into t partitions of equal work. Yet, partitioning the loop itself is preferred for coarser and greater parallelism.

Algorithm 7.5 LemmaForWeierstrass(k, f, g, h, t)

Input: $f, g, h \in \mathbb{K}[[X_1, \ldots, X_n]]$ such that $f = gh$, $f_{(0)} = 0$, $h_{(0)} \neq 0$, f known to precision k, and g, h known to precision $k-1$. $t \geq 1$ the number of threads to use.

Output: $g_{(k)}$.

1: $\mathcal{V} := [0, \ldots, 0]$ ▷ 0-indexed list of size t
2: **parallel_for** $j := 0$ to $t-1$
3: | **for** $i := {}^{jk}/_{t} + 1$ to ${}^{(j+1)k}/_{t}$ **while** $i < k$ **do**
4: | | $\mathcal{V}[j] := \mathcal{V}[j] + g_{(k-i)} h_{(i)}$
5: **end for**
6: **return** $\left(f_{(k)} - \sum_{j=0}^{t-1} \mathcal{V}[j]\right) / h_{(0)}$

Algorithm 7.6 WeierstrassPhase2Parallel(k, f, p, α, t)

Input: $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, $\alpha = \sum_{i=0}^{m} c_i Y^i$, $a_i, b_i, c_i \in \mathbb{K}[[X_1, \ldots, X_n]]$ satisfying Theorem 7.3. $b_0, \ldots, b_{d-1}$ known modulo $\mathcal{M}^{k+1}$, $c_0, \ldots, c_m$ known modulo $\mathcal{M}^k$, for $\mathcal{M}$ the maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$. $t \geq 1$ for the number of threads to use.

Output: $c_0, \ldots, c_m$ known modulo $\mathcal{M}^{k+1}$, updated in-place.

1: $work := 0$
2: **for** $i := 1$ to m **do** ▷ estimate work using Observation 7.8, Corollary 7.10
3: | **if** $i \leq d$ **then** $work := work + i - (a_{d+m} = 0)$ ▷ eval. Boolean as an integer
4: | **else** $work := work + d$
5: $t' := 1$; $targ := work \,/\, t$
6: $work := 0$; $j := 1$
7: $\mathcal{I} := [-1, 0, \ldots, 0]$ ▷ 0-indexed list of size $t+1$
8: **for** $i := 1$ to m **do**
9: | **if** $i \leq d$ **then** $work := work + i - (a_{d+m} = 0)$
10: | **else** $work := work + d$
11: | **if** $work \geq targ$ **then**
12: | | $\mathcal{I}[j] := i$; $work := 0$; $j := j + 1$
13: **if** $j \leq t$ **and** $t' < 2$ **then** ▷ work did not distribute evenly; try again with $t = {}^{t}/_{2}$
14: | $t := t \,/\, 2$; $t' := 2$
15: | **goto** Line 6
16: **else if** $j \leq t$ **then** ▷ still not even, use all threads in UpdateToDegParallel
17: | $\mathcal{I}[1] := m$; $t' := 2t$; $t := 1$
18: **parallel_for** $\ell := 1$ to t
19: | **for** $i := \mathcal{I}[\ell - 1] + 1$ to $\mathcal{I}[\ell]$ **do**
20: | | UpdateToDegParallel(k, c_{m-i}, t')

Hence, for phase 2, a dynamic decision is made to either apply the map pattern to the loop over c_{m-i}, or to apply the map pattern within UPDATETODEGPARALLEL for each c_{m-i}, or both. This decision process is detailed in Algorithm 7.6, where t partitions of equal work try to be found to apply the map pattern to only the loop itself. If unsuccessful, $t/2$ partitions of equal work try to be found, with 2 threads to be used within UPDATETODEGPARALLEL of each partition. If that, too, is unsuccessful, then each c_{m-i} is updated one at a time using the total number of threads t within UPDATETODEGPARALLEL.

7.5.2 Parallel Algorithms for Hensel Factorization

Let $f = f_1 \cdots f_r$ be a Hensel factorization where the factors have respective degrees $d_1, \ldots, d_r$. From Algorithm 7.3 and Figure 7.6, we have already seen that the repeated applications of Taylor shift and Weierstrass preparation naturally form a chain of ancestors, and thus a pipeline. Using the notation of Algorithm 7.3, updating f_1 requires updating g_1, p_1, α_1. Then, updating f_2 requires updating $\hat{f}_2, g_2, p_2, \alpha_2$, and so on. These groups easily form stages of a pipeline, where updating f_1 to degree $k-1$ is a prerequisite for updating f_2 to degree $k-1$. Then, meanwhile f_2 is being updated to degree $k-1$, f_1 can simultaneously be updated to degree k. This pattern holds for all successive factors.

Algorithms 7.7 and 7.8 show how the factors of a Hensel factorization can all be simultaneously updated to degree k using asynchronous generators, denoted by the constructor ASYNCGENERATOR, forming the so-called *Hensel pipeline*. Algorithm 7.7 shows a single pipeline stage as an asynchronous generator, which itself consumes data from another asynchronous generator—just as expected from the pipeline pattern. Algorithm 7.8 shows the creation, and joining in sequence, of those generators. The key feature of these algorithms is that a generator (say, stage i) produces a sequence of integers (j) which signals to the consumer (stage $i+1$) that the previous factor has been computed up to precision j and the required data is available to update its own factor to precision j.

Notice that Algorithm 7.8 still follows our lazy evaluation scheme. Indeed, the factors are updated all at once up to precision k, starting from their current precision. However, for optimal performance, the updates should be applied for large increases in precision, rather than repeatedly increasing precision by one.

Further considering performance, Theorem 7.12 showed that the cost for updating each factor of a Hensel factorization is different. In particular, for $\hat{d}_i := \sum_{j=i}^{r} d_j$, updating factor f_i scales as $d_i \hat{d}_{i+1} k^2$. The work for each stage of the proposed pipeline is unequal and the pipeline is unlikely to achieve good parallel speed-up. However, Corollary 7.13

Algorithm 7.7 HenselPipelineStage(k, f_i, t, gen)

Input: A positive integer k, $f_i = Y^{d_i} + \sum_{i=0}^{d_i-1} a_i Y^i, a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$. A positive integer t the number of threads to use within this stage. gen a generator for the previous stage.

Output: a sequence of integers j signalling f_i is known to precision j, ending with k.

```
1: p := Precision(f_i)                          ▷ get the current precision of f_i
2: do
3: |   k' := gen()                              ▷ A blocking function call until gen yields
4: |   for j := p to k' do
5: |   |   UpdateToDegParallel(j, f_i, t)
6: |   |   yield j
7: |   p := k'
8: while k' < k
```

Algorithm 7.8 HenselFactorizationPipeline(k, $\mathcal{F}$, $\mathcal{T}$)

Input: A positive integer k, $\mathcal{F} = \{f_1, \ldots, f_r\}$, the output of HenselFactorization. $\mathcal{T} \in \mathbb{Z}^r$ a 0-indexed list of the number of threads to use in each stage, $\mathcal{T}[r-1] > 0$.

Output: $f_1, \ldots, f_r$ updated in-place to precision k.

```
1: gen := ( ) → {yield k}                       ▷ An anonymous function asynchronous generator
2: for i := 0 to r − 1 do
3: |   if T[i] > 0 then
   |   |   ▷ Capture HenselPipelineStage(k, f_{i+1}, T[i], gen) as a
   |   |     function object, passing the previous gen as input
4: |   |   gen := AsyncGenerator(HenselPipelineStage, k, f_{i+1}, T[i], gen)
5: do
6: |   k' := gen()                               ▷ ensure last stage completes before returning
7: while k' < k
```

shows that the work ratios between stages do not change for increasing k, and thus a static scheduling scheme is sufficient.

Notice that Algorithm 7.7 takes a parameter t for the number of threads to use within the pipeline stage. As we have seen in Section 7.5.1, the Weierstrass update can be performed in parallel. Consequently, each stage of the Hensel pipeline uses t threads to exploit such parallelism. We have thus composed the two parallel schemes, applying map-reduce within each stage of the parallel pipeline. This composition serves to load-balance the pipeline. For example, the first stage may be given t_1 threads and the second stage given t_2 threads, with $t_1 > t_2$, so that the two stages may execute in nearly equal time.

To further encourage load-balancing, each stage of the pipeline need not update a single factor, but rather a group of successive factors. Algorithm 7.9 applies Theorem 7.12 to attempt to load-balance each stage s of the pipeline by assigning a certain number of threads t_s and a certain group of factors $f_{s_1}, \ldots, f_{s_2}$ to it. The goal is for $\sum_{i=s_1}^{s_2} d_i \hat{d}_{i+1} \,/\, t_s$ to be roughly equal for each stage.

Algorithm 7.9 DISTRIBUTERESOURCESHENSEL($\mathcal{F}$, t_{tot})

Input: $\mathcal{F} = \{f_1, \ldots, f_r\}$ the output of HENSELFACTORIZATION. $t_{tot} > 1$ the total number of threads.

Output: $\mathcal{T}$, a list of size r, where $\mathcal{T}[i]$ is the number of threads to use for updating f_{i+1}. The number of positive entries in $\mathcal{T}$ determines the number of pipeline stages. $\mathcal{T}[i] = 0$ encodes that f_{i+1} should be computed within the same stage as f_{i+2}.

1: $\mathcal{T} := [0, \ldots, 0, 1]$; $t := t_{tot} - 1$ ▷ $\mathcal{T}[r-1] = 1$ ensures last factor gets updated
2: $d := \sum_{i=1}^{r} \deg(f_i)$
3: $\mathcal{W} := [0, \ldots, 0]$ ▷ A 0-indexed list of size r
4: **for** $i := 1$ to $r - 1$ **do**
5: | $\mathcal{W}[i-1] := \deg(f_i)(d - \deg(f_i))$ ▷ Estimate work by Theorem 7.12, $d_i \hat{d}_{i+1}$
6: | $d := d - \deg(f_i)$
7: $totalWork := \sum_{i=0}^{r-1} \mathcal{W}[i]$

8: $ratio := 0$; $targ := 1 \,/\, t$
9: **for** $i := 0$ to r **do**
10: | $ratio := ratio + (\mathcal{W}[i] \,/\, totalWork)$
11: | **if** $ratio \geq targ$ **then**
12: | | $\mathcal{T}[i] :=$ ROUND$(ratio \cdot t)$; $ratio := 0$

13: $t := t_{tot} - \sum_{i=0}^{r-1} \mathcal{T}[i]$ ▷ Give any excess threads to the earlier stages
14: **for** $i := 0$ to $r - 1$ **while** $t > 0$ **do**
15: | $\mathcal{T}[i] := \mathcal{T}[i] + 1$; $t := t - 1$
16: **return** $\mathcal{T}$

7.6 Experimentation and Discussion

The previous section introduced parallel schemes for Weierstrass preparation and Hensel factorization based on the composition of the map-reduce and pipeline parallel patterns. Our lazy power series and parallel schemes have been implemented in C/C++ as part of the Basic Polynomial Algebra Subprograms (BPAS) library [7]. These parallel algorithms are implemented using the generic support for task parallelism, thread pools, and asynchronous generators, as described in Chapter 5.

In these experiments, all data shown is an average of 3 trials. BPAS was compiled using GMP 6.1.2 [90]. We work over $\mathbb{Q}$ as these examples do not require algebraic numbers to factor into linear factors. We thus borrow univariate integer polynomial factorization from NTL 11.4.3 [163]. As mentioned in Section 7.2.2, we have also implemented our lazy power series and UPoPS as the `MultivariatePowerSeries` library starting from *Maple* 2021. While that implementation is not parallelized, it is able to handle examples requiring algebraic numbers in the factorization, which our implementation in BPAS cannot. See [10] for details.

We begin with examining the serial implementation of Weierstrass preparation. The `PowerSeries` sub-library of the *RegularChains* library in *Maple* provides an implementation of Weierstrass preparation against which to compare. However, we note that the latter is not a lazy implementation, returning only a truncated UPoPS. We have studied two families of examples:

(W_i) $\frac{1}{1+X_1+X_2}Y^d + Y^{d-1} + \cdots + Y^2 + X_2Y + X_1$ and

(W_{ii}) $\frac{1}{1+X_1+X_2}Y^d + Y^{d-1} + \cdots + Y^{\lceil d/2 \rceil} + X_2Y^{\lceil d/2 \rceil - 1} + \cdots + X_2Y + X_1$

The first results in p of degree 2, while the second results in p of degree $\lceil k/2 \rceil$, thus emphasizing the performance of phase two and phase one of the algorithm, respectively. The leading coefficient being the inverse of a power series makes the overall computation more challenging. The results of this experiment are summarized in Figures 7.7 and 7.8. Not only is our implementation orders of magnitude faster than *Maple*, but the difference in computation time further increases with increasing precision. This can be attributed to our efficient underlying power series arithmetic, as well as our smart implementation of Lemma 7.2.

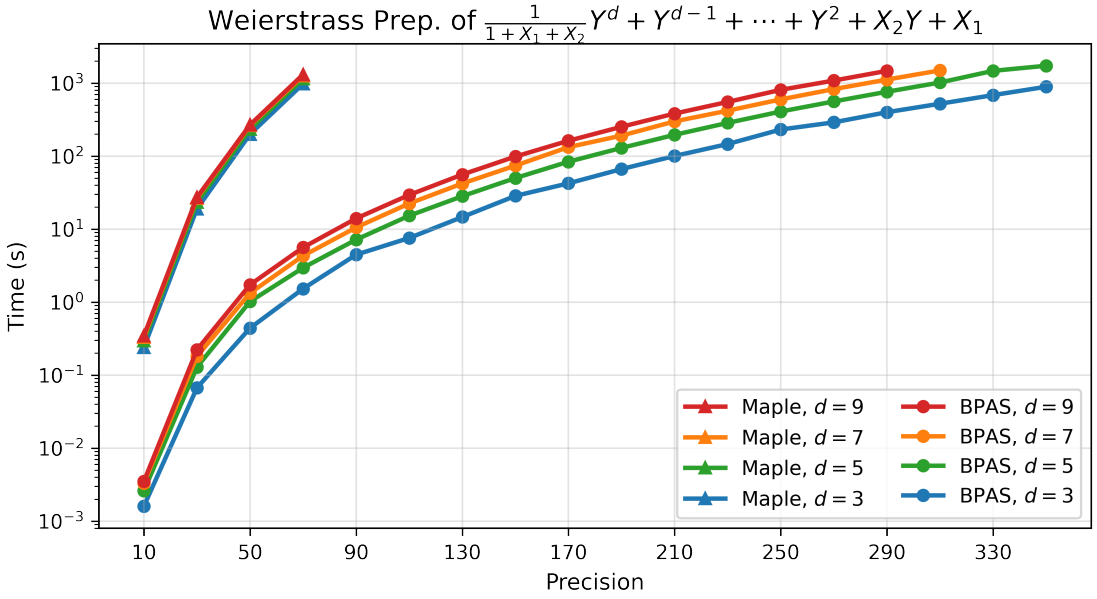

Figure 7.7: Applying Weierstrass preparation on family W_i for increasing precisions.

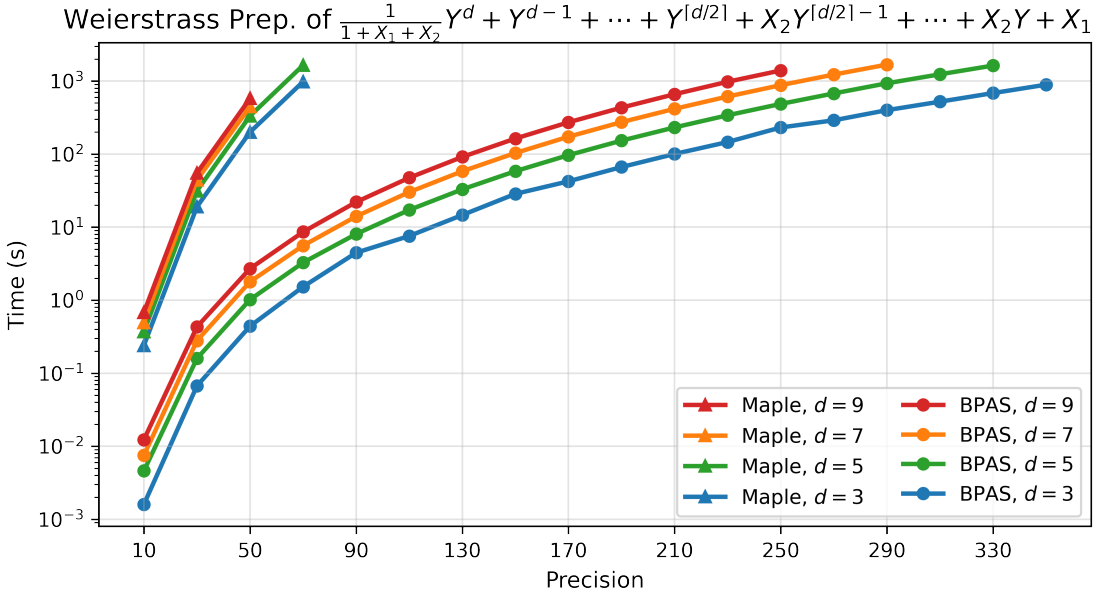

Figure 7.8: Applying Weierstrass preparation on family W_{ii} for increasing precisions.

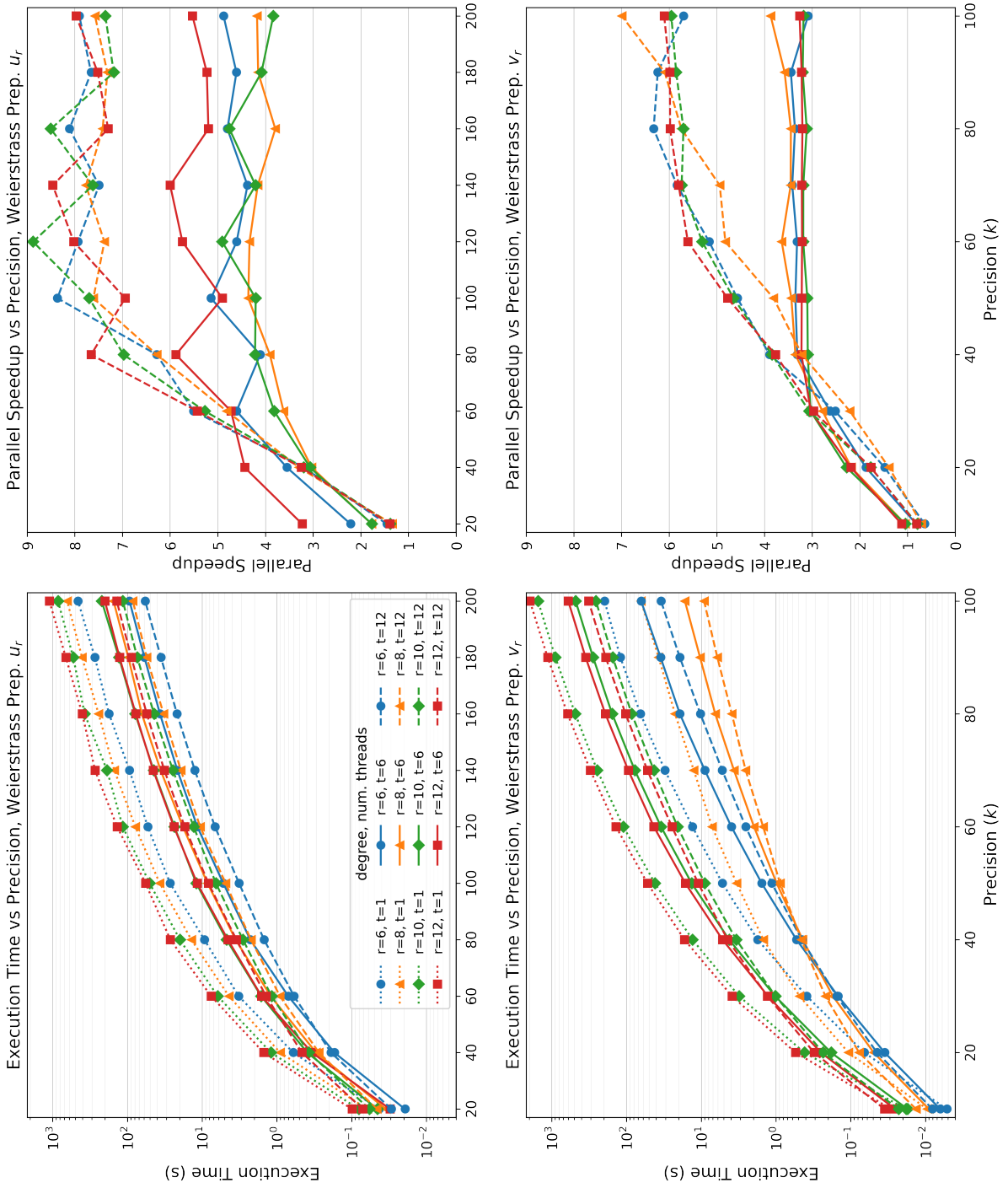

Figure 7.9: Comparing Weierstrass preparation of u_r and v_r for $r \in \{6, 8, 10, 12\}$ and number of threads $t \in \{1, 6, 12\}$. First column: execution time of u_r and v_r; second column: parallel speed-up of u_r and v_r. Profiling of v_6 shows that its exceptional relative performance is attributed to remarkably good branch prediction.

Next, we examine the parallel speed-up achieved for our parallel variation of Weierstrass preparation. We again study to families of UPoPS:

$$(i)\ \ u_r = \sum_{i=2}^{r}(X_1^2 + X_2 + i)Y^i + (X_1^2 + X_2)Y + X_1^2 + X_1X_2 + X_2^2$$
$$(ii)\ \ v_r = \sum_{i=\lceil r/2 \rceil}^{r}(X_1^2 + X_2 + i)Y^i + \sum_{i=1}^{\lceil r/2 \rceil - 1}(X_1^2 + X_2)Y^i + X_1^2 + X_1X_2 + X_2^2$$

Applying Weierstrass preparation to u_r results in p with degree 2. Applying Weierstrass preparation to v_r results in p with degree $\lceil r/2 \rceil$. Figure 7.9 summarizes the resulting execution times and parallel speed-ups. Generally, speed-up increases with increasing degree in Y and increasing precision computed.

Recall that parallelism arises in two ways: computing summations of products of homogeneous parts (the **parallel_for** loops in Algorithms 7.4 and 7.5), and the **parallel_for** loop over updating c_{m-i} in Algorithm 7.6. The former has an inherent limitation: computing a multivariate product with one operand of low degree and one of high degree is much easier than computing one where both operands are of moderate degree. Evenly partitioning the iterations of the loop does not result in even work per thread. This is evident in comparing the parallel speed-up between u_r and v_r; the former, with higher degree in α, relies less on parallelism coming from those products. Better partitioning is needed and is left to future work.

For our serial version of Hensel factorization, we compare against two possible operations in the `PowerSeries` of *Maple*. The `ExtendedHenselConstruction` (EHC) implements the Extended Hensel Construction and `FactorizationViaHenselLemma` (FVHL) implements Hensel factorization via repeated applications of Weierstrass preparation theorem. While EHC, in general, factors UPoPS over the field of Puiseux series, the test cases used here result in only power series in the outputs.

We examine to test cases, one produces three factors and the other produces four factors. This examples cause $\bar{f}$ to split into linear factors over $\mathbb{Q}$, therefore not requiring algebraic numbers or Puiseux series. Those polynomials are:

$$f_1 = (Y-1)(Y-2)(Y-3) + X_1(Y^2 + Y)$$
$$f_2 = (Y-1))(Y-2)(Y-3)(Y-4) + X_1(Y^3 + Y)$$

The results of this experimentation is summarized in Figure 7.10 for the two UPoPS f_1 and f_2 This result is that our implementation is orders of magnitude faster than EHC. Meanwhile, FVHL is unable to compute factors, in a reasonable time, beyond a precision of 100. We observe that the gap between our implementation and EHC increases both as UPoPS degree increases and as power series precision increases.

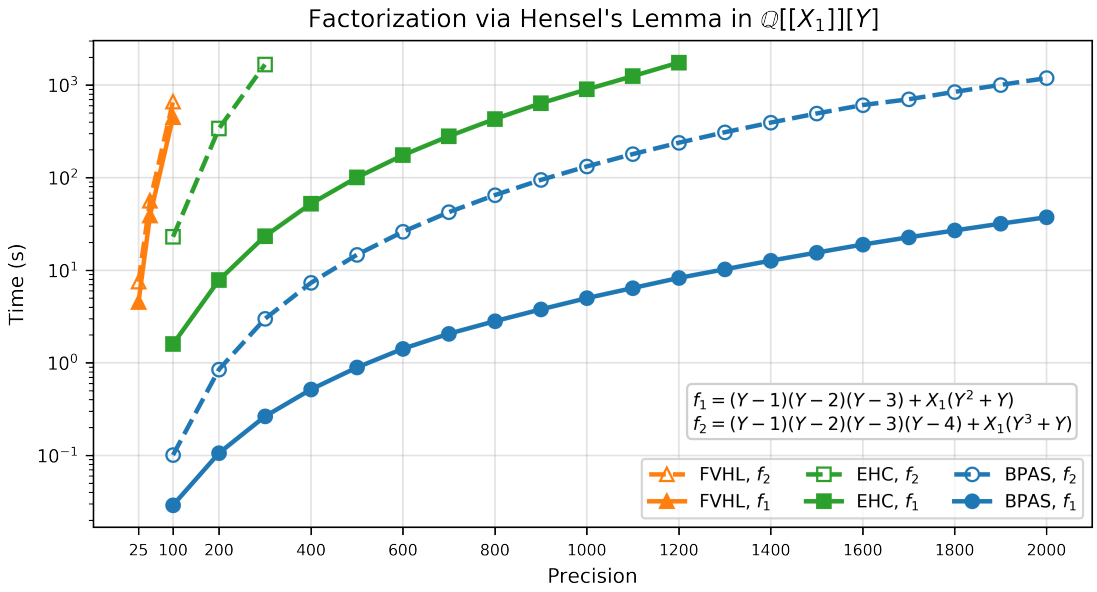

Figure 7.10: Applying factorization via Hensel's lemma to the UPoPS $f_1 = (Y-1)(Y-2)(Y-3) + X_1(Y^2+Y)$ and $f_2 = (Y-1)(Y-2)(Y-3)(Y-4) + X_1(Y^3+Y)$.

Next, we evaluate our parallel Hensel factorization using three similar families of polynomials:

$$(i)\ \ x_r = \prod_{i=1}^{r}(Y-i) + X_1(Y^3+Y)$$

$$(ii)\ \ y_r = \prod_{i=1}^{r}(Y-i)^i + X_1(Y^3+Y)$$

$$(iii)\ \ z_r = \prod_{i=1}^{r}(Y+X_1+X_2-i) + X_1X_2(Y^3+Y)$$

These families represent three distinct computational configurations: (i) factors of equal degree, (ii) factors of distinct degrees, and (iii) multivariate factors. The comparison between x_r and y_r is of interest in view of Theorem 7.12.

Despite the inherent challenges of irregular parallelism arising from stages with unequal work, the composition of parallel patterns allows for load-balancing between stages and the overall pipeline to achieve relatively good parallel speed-up. Figure 7.11 summarizes these results while Table 7.1 presents the execution time per factor (or stage, in parallel). Generally speaking, potential parallelism increases with increasing degree and increasing precision.

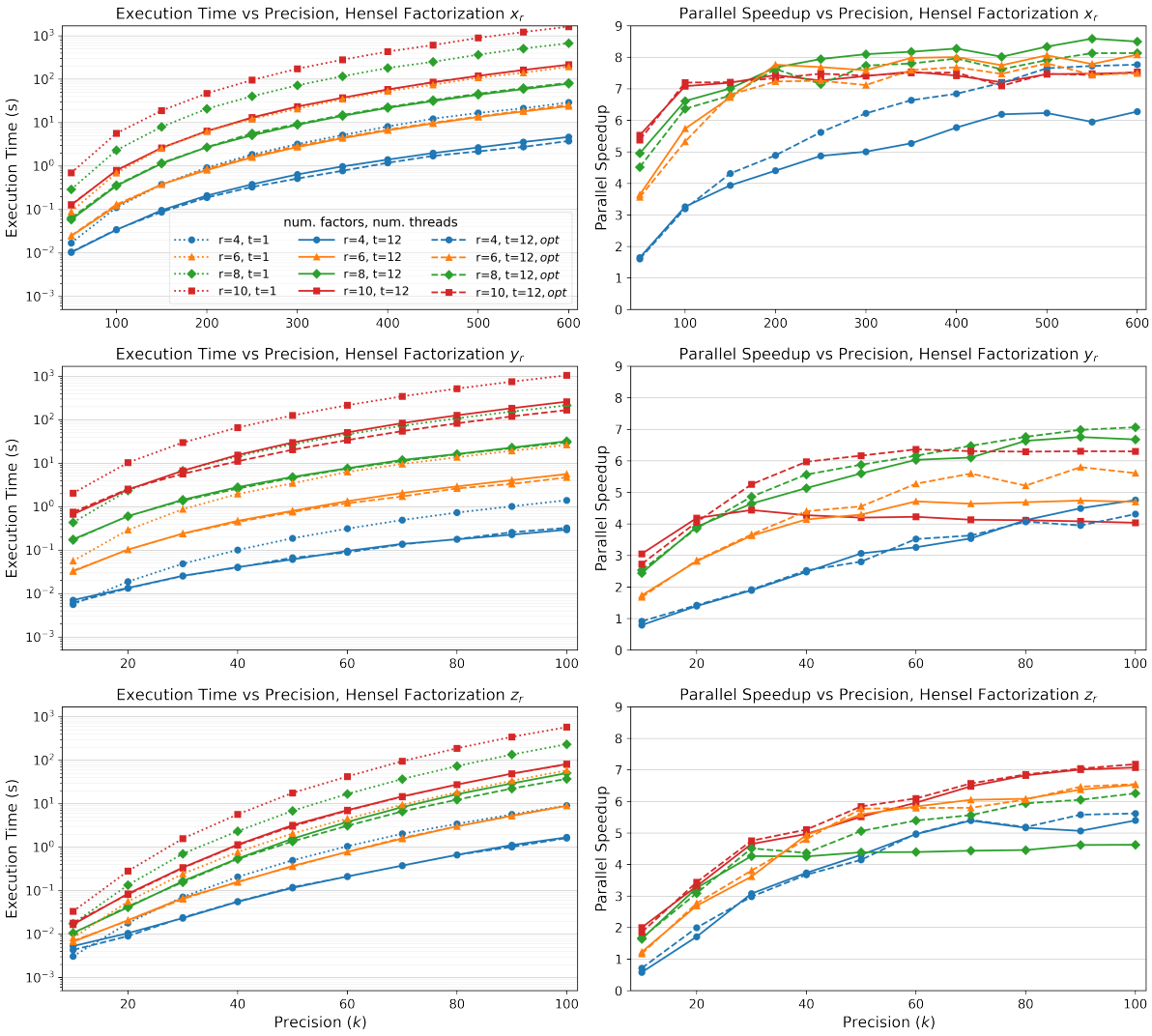

Figure 7.11: Comparing parallel Hensel factorization for x_r, y_r, and z_r for $r \in \{4, 6, 8, 10\}$. First column: execution time; second column: parallel speed-up. For number of threads $t = 12$ resource distribution is determined by Algorithm 7.9; for $t = 12, opt$ serial execution time replaces complexity measures as work estimates in Algorithm 7.9, Lines 4–6.

		factor	serial time (s)	shift time (s)	Complexity-Est. threads	parallel time (s)	wait time (s)	Time-est. threads	parallel time (s)	wait time (s)
x_4	$k = 600$	f_1	18.1989	0.0012	6	4.5380	0.0000	7	3.5941	0.0000
		f_2	6.6681	0.0666	4	4.5566	0.8530	3	3.6105	0.6163
		f_3	3.4335	0.0274	1	4.5748	1.0855	0	-	-
		f_4	0.0009	0.0009	1	4.5750	4.5707	2	3.6257	1.4170
	totals		28.3014	0.0961	12	4.5750	6.5092	12	3.6257	2.0333
y_4	$k = 100$	f_1	0.4216	0.0003	3	0.1846	0.0000	4	0.1819	0.0000
		f_2	0.5122	0.0427	5	0.2759	0.0003	4	0.3080	0.0001
		f_3	0.4586	0.0315	3	0.2842	0.0183	0	-	-
		f_4	0.0049	0.0048	1	0.2844	0.2780	4	0.3144	0.0154
	totals		1.3973	0.0793	12	0.2844	0.2963	12	0.3144	0.0155
z_4	$k = 100$	f_1	5.2455	0.0018	6	1.5263	0.0000	7	1.3376	0.0000
		f_2	2.5414	0.0300	4	1.5865	0.2061	3	1.4854	0.0005
		f_3	1.2525	0.0151	1	1.6504	0.1893	0	-	-
		f_4	0.0018	0.0018	1	1.6506	1.6473	2	1.5208	0.7155
	totals		9.0412	0.0487	12	1.6506	2.0427	12	1.5208	0.7160

Table 7.1: Times for updating each factor within the Hensel pipeline, where f_i is the factor with i as the root of $\bar{f}_i$, for various numbers of threads per stage. Complexity-estimated threads use complexity estimates to estimate work within Algorithm 7.9; time-estimated threads use the serial execution time to estimate work and distribute threads.

The distribution of a discrete number of threads to a discrete number of pipeline stages is a challenge; a perfect distribution requires a fractional number of threads per stage. Nonetheless, in addition to the distribution technique presented in Algorithm 7.9, we can examine hand-chosen assignments of threads to stages. One can first determine the time required to update each factor in serial, say for some small k, and then use that time as the work estimates in Algorithm 7.9, rather than using the complexity estimates. This latter technique is depicted in Figure 7.11 as *opt* and in Table 7.1 as Time-est. threads. This is still not perfect, again because of the discrete nature of threads, and the imperfect parallelization of computing summations of products of homogeneous parts.

In future, we must consider several important factors to improve performance. Relaxed algorithms should give better complexity and performance. For parallelism, better partitioning schemes for the map-reduce pattern within Weierstrass preparation should be considered. Finally, for the Hensel pipeline, more analysis is needed to optimize the scheduling and resource distribution, particularly considering coefficient sizes and the multivariate case.

Chapter 8

Towards the Next Generation of Triangular Decomposition

Throughout Chapter 6 we described our efforts to support a high-performance triangular decomposition implementation. This included speculative subresultants, component-level parallelism, and low-level parallelism in subresultants. Yet, we suggested throughout that chapter that further concurrency could be exploited with a little more effort. Particularly when we consider the sources of component-level parallelism, and the cooperation of the various parallel regions. Work in Chapter 7 regarding parallel Hensel factorization has further demonstrated that irregular parallelism can be effectively exploited through layering of parallel regions and thoughtful resource distribution.

With the experiences we gained from this previous work, we now look to examine and design what will become our next generation implementation of triangular decomposition. This includes software design, parallelism and practical implementation techniques for improved performance, and a new data structure and paradigm for organizing regular chains throughout a decomposition.

This chapter is organized as follows. Section 8.1 begins by discussing improved software design for regular chains and triangular decomposition as a whole. A reflection on legacy code, code refactoring, and code rewriting is presented. Our next generation implementation has already begun and has implemented the designs described in this section. The following sections, however, are purely algorithmic, with implementations and experimentation forthcoming.

Experiences optimizing irregular parallel applications are integrated into new algorithmic organizations of Triangularize in Section 8.2. This section examines opportunities for more concurrency, better approaches to cooperative parallelism and resource distribution, and reducing redundant computations. The idea of reducing redundant

computations is further extended in Section 8.3, where we discuss a new paradigm and data structure for encoding regular chains. This paradigm is based on dynamic evaluation (see Section 2.2.1), splitting trees, and the experiences of dynamic evaluation applied to cylindrical algebraic decomposition [53]. We call this paradigm the regular chain "universe", taking inspiration from the idea that every regular chain instance should exist in a common space, be unique, and know about the existence of one another.

8.1 Generic and Polymorphic Regular Chains

Our implementation of regular chains and triangular decomposition has seen great success in parallelization and performance. Our first implementation of triangular decomposition from [12] has been further developed and expanded with additional algorithmic support for improved serial performance and additional concurrency schemes for improved parallel performance. This was detailed in Sections 6.2 and 6.3. In this section, we discuss new design aspects related to the implementation of triangular decomposition that consider not just performance but also maintainability and adaptability. Let us begin with a retrospective view on our design and implementation of triangular decomposition as described in Chapter 6. The reality of working in a collaborative software development environment was missing from that discussion.

The most complete implementation of triangular decomposition is the implementation of *RegularChains* in *Maple*. On one hand, adapting code from a scripting environment to the compiled and object-oriented environment of C/C++ was itself a respectable exercise. On the other hand, *RegularChains*, by being part of *Maple*, has access to the wide range of functionalities provided by a mature and stand-alone computer algebra system. As we have discussed previously, triangular decomposition requires the support of extensive foundational algorithms including GCDs, factorization, polynomial arithmetic, and linear algebra.

In the case of BPAS, these functionalities were developed in parallel with the triangular decomposition code. The first serial implementation of triangular decomposition was mainly by Robert H.C. Moir, meanwhile the foundational algorithms were implemented by the author. Due to the implementation occurring in tandem, many stopgap measures were introduced as "temporary" solutions to immediate problems. Unfortunately, many of those temporary solutions were not so temporary. Moreover, as is expected from any larger-scale or complex coding project, debugging for correctness and performance was a considerable challenge. Finally, our various parallel schemes used within triangular decomposition were added (without the greatest attention to design) after the design

and implementation of the serial code.

As was introduced in Chapter 1, scientific and mathematical software is often challenged by maintainability and usability. Our software was no exception. For example, one issue is that our implementation of triangular decomposition assumes that the field to solve over was the rational numbers. This was not the original design or intention, but it is the reality of the implementation; it relies directly on our implementation of multivariate polynomials with rational number coefficients.

Nonetheless, we now look to improve our design and implementation to remedy these issues. Toward our final goal of implementing the regular chain universe, the underlying data structures of regular chains would need to be drastically modified. Unfortunately, the code implementing regular chains and triangular decompositions was quite rigid and fragile. Indeed, much of the triangular decomposition routines broke encapsulation and relied on particular internal details of the regular chain objects. Moreover, the code itself grew increasingly complex with the integration of debugging, profiling, and parallelism. This leads to several possibilities: (*i*) continue to adapt and modify the existing code toward the goal of the regular chain universe; (*ii*) to refactor the existing code to simplify design and restore the required modularity; or (*iii*) accept the original implementation as a prototype, learn from previous mistakes, and start again.

Although starting anew requires a lot of upfront work and commitment, the long-term benefits of improving modularity and maintainability are extremely advantageous. We chose to start over, allowing us to develop a better design with improved flexibility and maintainability. A driving principle in this design is *separation of concerns*. With the knowledge of past experiences guiding our new design, we have several goals.

(*i*) Develop triangular decomposition algorithms which are capable of solving over any field of numbers.

(*ii*) Separate triangular decomposition algorithms from the implementation of regular chains and triangular sets.

(*iii*) Separate the regular chain and triangular set classes from the underlying data structure which stores the polynomials.

The first two goals have an obvious solution given our previously defined algebraic class hierarchy of Chapter 4. We should implement regular chains and triangular sets as template classes, parameterized by a multivariate polynomial ring. These template classes should not rely on the interface of a specific polynomial class, but rather only on the interface provided by an abstract multivariate polynomial ring class. Precisely, we

have developed a `RecursivelyViewedPoly` abstract class which represents polynomials that can be viewed recursively based on some variable ordering. This interface requires concrete classes to implement methods like main variable, initial, tail, pseudo-division, etc. This is a suitably generic interface from which triangular sets and regular chains can be implemented. Using `Derived_from` (see Section 4.1.3), we can also enforce that polynomials used as a regular chain template parameter have a field as a ground ring.

Implementing a new regular chain template class provides the opportunity to restore encapsulation. Our previous implementation of triangular decomposition algorithms made Triangularize and its subroutines methods of the regular chain class. Rather, we have now implemented them as functions which use only the public interface of regular chains.

Our last goal requires some additional design considerations. While it is possible to add another layer of encapsulation between regular chains and the data structure storing its polynomials, this adds yet another layer of indirection and further reduces data locality. Is the design benefits of encapsulation worth the (potential) decreases in runtime performance? This leads to one important design question. How could this underlying data structure be implemented so that it remains generic enough to work with any `RecursivelyViewedPolynomial`, but maintains efficient access to the underlying polynomials? Specifically, could the data structure sufficiently encapsulate the storage of polynomials behind an interface but without requiring unnecessary copying of internal data, for example, to be returned by *getter methods*.

For contrast, let us consider the original design of regular chains and triangular sets used in our previous implementation of Chapter 6. Triangular sets are implemented as a class with instance variables:

- `polys`, a `std::vector` of `RationalNumberPoly`s, and
- `vars`, an ordered `std::vector` of variables defining the polynomial ring

Variables are implemented as `Symbol` objects—a simple encapsulation of `std::string`s. Then, regular chains were declared a subclass of triangular sets, yet re-defined most triangular set methods without relying on inheritance.

Notice some specific flaws of this design with respect to our design goals. While a `std::vector` does provide an interface to its underlying array of data, it presupposes an organization of the data as a dense list, which is not overly flexible to our eventual goal of implementing splitting trees. Moreover, the `polys` and `vars` vectors act as *parallel arrays* (or *Structure of Arrays*) where the polynomial with main variable `vars[i]` can be found

at `polys[i]`. Maintaining and accessing this structure is cumbersome and unnecessarily burdens the triangular set and regular chain code.

Our improved design has two parts. For the first part of our design, we have developed a `PolynomialRing` data structure which encapsulates variables, their ordering, and operations on ordered sets of variables. Recall that a common operation on a regular chain T is to, given a variable v, find T_v, T_v^-, and T_v^+. Another common operation is to ensure the *compatibility* of a polynomial p with a particular polynomial ring. Does p have any variables not contained in a particular ring? Is the ordering of variables of p consistent with the polynomial ring? These questions of variable ordering, containment, and less than or greater than subsets (for T_v^- and T_v^+, respectively), are all encapsulated by our new `PolynomialRing` class. It stores an ordered list of variables (`Symbol` objects) and its interface answers questions related to ordering, indexing, and compatibility. This frees our new triangular set and regular chain classes from much of the bookkeeping of variable management.

The second part of our design is an ordered polynomial set abstract data structure, `OrderedPolySet`. Through composition, this class stores the `RecursivelyViewedPoly` polynomials of a triangular set. As the name suggests, the storage takes into account the ordering of their main variable. This class is abstract to allow for different concrete implementations, meanwhile the new triangular set and regular chain classes require only the interface provided by the abstract class. `OrderedPolySet` uses the `PolynomialRing` class to define the ordering of the polynomials in the set. The actual storage of polynomials is not important, and is fully encapsulated by the interface. However, the interface also allows for sufficiently efficient access to the stored data.

The `OrderedPolySet` interface takes inspiration from our `AsyncGenerator` design (Section 5.3.3), and from the design of `std::vector`. Specifically, we avoid data movement and copies as much as possible. From the former, our polynomial set provides methods with C++ *move semantics* to access and add polynomials without any copying of data. From the latter, we overload the *subscript operator* (or *array index operator*) as `operator[](const Symbol& v)`. A nice property of this operator is that it allows for "overloading" of the return type (which is not possible for methods in general) to return either a reference or a `const` reference, depending on the needs of the caller. The compiler is able to deduce whether a `const` or non-`const` reference is needed and then call the correct function. The advantage of returning a (`const`) reference is, again, avoiding unnecessary copies and data movement.

Notice that this operator takes a variable `v` as parameter rather than the typical integer index. This serves several purposes. First, it allows clients (e.g. triangular sets)

to implement the natural use case of accessing a polynomial corresponding to a particular variable (T_v). Second, clients no longer need to calculate a variable's index in the overall ordering of a `PolynomialRing`. Third, it avoids the semantics or expectation that the `OrderedPolySet` class is implemented in a (dense) array-like way.

Indeed, the interface of the polynomial set allows for adding, removing, and accessing a polynomial of a particular main variable, but gives no indication to how the data is stored internally. See the interface shown in Listing 8.1.

```
template<class RecursivePoly>
class OrderedPolySet : Derived_from<RecursivePoly, RecursivelyViewedPoly> {

    //Construct a set where R defines the variable ordering.
    OrderedPolySet(const PolynomialRing& R);

    //Return true iff a polynomial with main variable v is in this set.
    bool contains(const Symbol& v) const;

    //Get a reference to the polynomial in the set with main variable v.
    //If no polynomial exists, this results in undefined behaviour
    const RecursivePoly& operator[](const Symbol& v) const;
    RecursivePoly& operator[](const Symbol& v);

    //Add a polynomial, return true iff successful
    bool addPolynomial(RecursivePoly&& p);

    //Remove a polynomial whose main variable is v, return true iff removed
    bool removePolynomial(const Symbol& v);

    //Get polynomials in the set with main variable less (greater) than v
    OrderedPolySet<RecursivePoly> lessThan(const Symbol& v) const;
    OrderedPolySet<RecursivePoly> greaterThan(const Symbol& v) const;
}
```

Listing 8.1: The `OrderedPolySet` interface. Note that `RecurisvelyViewedPoly` does have template parameters to facilitate CRTP, but they are ommitted for clarity of presentation; see Section 4.1.3

In this interface we can see the methods `lessThan(Symbol)` and `greaterThan(Symbol)` which return `OrderedPolySet` objects that are subsets of the existing polynomial set. This mimics the triangular set operations T_v^- and T_v^+. Returning the subsets as another instance of `OrderedPolySet` is a crucial piece of the design, and continues the encapsulation of the polynomial storage. In particular, it allows for the two objects to potentially share, for example, a pointer to the actual underlying polynomials. Thus, again, avoiding data movement. In this current implementation, the idea of shared polynomials is not

yet implemented. Yet, it will be one of the main goals of the regular chain universe, see Section 8.3.

Using `OrderedPolySet`, we have completely re-implemented triangular sets, regular chains, and triangular decomposition to work generically with any `RecursivelyViewedPoly` over a field. Moreover, triangular sets and regular chains are implemented as a class hierarchy, with regular chains inheriting from triangular sets and avoiding code duplication.

This design and initial implementation already enjoys several successes. Our previous implementation of triangular decomposition totalled 21,000 lines of code, inflated by a haphazard design and "temporary" workarounds. The new design based on, and including, `OrderedPolySet` is less than half of that, with only 8,100 lines of code. The resulting code is, subjectively, much cleaner, maintainable, and self-documenting. This new implementation has been confirmed to be correct where the output decompositions are identical between both versions of the code.

Initial experimentation also shows that, serially, the new implementation is 5% to 100% faster at solving polynomial systems of our test suite (see Section 6.4). Further experimentation is needed, and will be performed in the future, after implementing the designs proposed in the next two sections. Nonetheless, we infer that this performance improvement is the result of more efficient storage of polynomials in `OrderedPolySet` and fewer data copies. Another possibility is that an unknown performance bug has been fixed during the translation from the old implementation.

With an improved design and implementation of regular chains and triangular decomposition, modifying it for even further performance improvements should be much easier and much more successful. In the next two sections, we detail our designs for prospective implementations and performance improvements.

8.2 A more Dynamic and Adaptive TRIANGULARIZE

Component-level parallelism in a key feature of the performance of our triangular decomposition presented in Chapter 6. Therein, we discussed how parallel patterns may be employed to exploit component-level parallelism via the workpile of TRIANGULARIZEBYTASKS, asynchronous generators, and the removal of redundant components.

The results of that work were favourable, yet highlighted some key challenges.

(*i*) Roughly two-thirds of the polynomials systems tested exhibited no component-level parallelism to exploit.

(*ii*) Removing redundant components throughout the computation is required for certain systems. Without the intermediate removal of redundant components, triangular decomposition can take up to 70× longer.

(*iii*) Asynchronous generators, while initially beneficial in the experimentation of [12], have now been shown to be less capable of providing parallel speed-up.

Further, experience gained from designing and implementing the parallel Hensel factorization (Chapter 7) suggests that exploiting fine-grained regular parallelism can be synergistic with coarse-grained irregular parallelism.

In this section we examine four possibilities to combat these challenges and improve parallel performance.

Solving systems modulo a prime

To address the issue of finding independent components for component-level parallelism, there are two possible directions. First, we will expand the suite of polynomial systems tested. Are there many polynomials systems in practice which have multiple components? The Lasker-Noether Theorem (Theorem 2.22) suggests yes. Second, we can always solve systems over a finite field, which will cause the regular chains to factor and split more frequently, even if computations would not split over the rational numbers. This strategy was examined in [144]. From a decomposition computed modulo a prime, it may also be possible to then apply Hensel lifting to recover the solution over the rational numbers. Such lifting was applied successfully to triangular decompositions of special kinds (equiprojectable and zero-dimensional) in [64].

Given our new design of triangular decomposition presented in the previous section, our implementation no longer relies directly on polynomials over rational numbers. Rather, only on polynomials adhering to the `RecursivelyViewedPoly` interface. This opens the door to solving systems modulo a prime, provided that a suitable polynomial class exists. Work is ongoing to develop highly efficient multivariate polynomials over a prime field in BPAS; see [145]. Then, solving over prime fields can be investigated. The improved component-level parallelism that will be exposed as a result of solving modulo a prime will likely cause the next two challenges discussed below to become even more prevalent.

Intermediate removal of redundant components

More components and more splitting means a higher probability of redundant components existing in a triangular decomposition. This implies a higher probability of redundant computations occurring. Avoiding those redundant computations will be beneficial to both overall performance and to reducing the consumption of hardware resources. That, in turn, will leave resources available to be used by other components or other parallel schemes within triangular decomposition, enabling improved parallel speed-up.

However, as we have seen in Section 6.3, different organizations of TRIANGULARIZE yield different ways to achieve practical performance. Consider again the idea that incremental triangular decomposition produces a tree of components. The root of the tree is the empty regular chain and the leaves are the final output of the triangular decomposition. An edges connects nodes T and T' when T' is the result of a call to INTERSECT with T from TRIANGULARIZE.

The first organization, shown in Algorithm 6.18, was called "by level", as the algorithm proceeded breadth-first through this tree and removed redundant components after each incremental step. This was good at avoiding redundant computations, but created a synchronization point at each incremental step which was detrimental to parallel performance. The second organization, shown in Algorithm 6.19, was called "by tasks" and modified the algorithm to process regular chains one at a time. Without any sorting of the pending tasks, this results in a depth-first traversal through the tree of components. While Algorithm 6.19 was better in general for overall performance and parallelism, Table 6.1 showed that intermediate removal of redundant components was very important in some cases. Solving over prime fields is likely to exacerbate that need.

A hybrid approach to TRIANGULARIZE will be needed to best exploit parallelism while also avoiding redundant branches and computations. The algorithm will need to proceed by tasks, yet still somehow remove redundant components, even as many traversals of this tree are occurring simultaneously via component-level parallelism. Two key features of this hybrid approach will be *asynchronous set inclusion testing* and tree pruning via *task cancellation*.

Recall the ISNOTINCLUDED(T_i, T_j) function determines if $W(T_i) \not\subseteq W(T_j)$. Therefore, testing quasi-component set inclusion is one (part of a) test for redundant components. However, in our hybrid approach, it will not be enough to only check $W(T_i) \subseteq W(T_j)$ in order to prune T_i from the computation. Indeed, this was an acceptable check for Algorithm 6.18, since the set of polynomials remaining to be intersected with each component was the same. With TRIANGULARIZEBYTASKS, each task has its own collection of polynomials which remain to be intersected with its regular chain. Therefore, one

will also need to ensure that the set of polynomials left to intersect with T_i is a *superset* of the polynomials to intersect with T_j. Consider the following two examples.

Example 8.1. Let T_1 and T_2 be regular chains of $\mathbb{K}[x > y > z > t]$. Using the notation of Algorithm 6.19, let a pair of tasks be (P_1, T_1), (P_2, T_2) where

$$(P_1, T_1) = \left(\{x, y\}, \left\{ \begin{array}{c} z \\ t \end{array} \right) \quad \text{and} \quad (P_2, T_2) = \left(\{x-1, y\}, \left\{ \begin{array}{c} \\ t \end{array} \right) . \right.$$

Clearly we have $W(T_1) \subseteq W(T_2)$. However, letting those tasks execute to completion will yield $T_1' = \{x, y, z, t\}$ and $T_2' = \{x-1, y, t\}$ with $W(T_1') \not\subseteq W(T_2')$.
Consider another pair of tasks (P_3, T_3) and (P_4, T_4) where

$$(P_3, T_3) = \left(\{x, y\}, \left\{ \begin{array}{c} z \\ t \end{array} \right) \quad \text{and} \quad (P_4, T_4) = \left(\{y\}, \left\{ \begin{array}{c} \\ t \end{array} \right) . \right.$$

Again, we have $W(T_3) \subseteq W(T_4)$. We also have $P_3 \supseteq P_4$. Letting these tasks execute to completion, we will have, respectively, $T_3' = \{x, y, z, t\}$, $T_4' = \{y, t\}$ with $W(T_3') \subseteq W(T_4')$.

While this example is quite trivial, it highlights the main issue of directly applying intermediate removal of redundant components to TriangularizeByTasks. One must check the regular chain in a triangularize task and its remaining set of polynomials to accurately determine redundancies.

For a pair of tasks (P_i, T_i) and (P_j, T_j), determining redundancies via both $W(T_i) \subseteq W(T_j)$ and $P_i \supseteq P_j$ being true may be too restrictive. That is, having $W(T_i) \subseteq W(T_j)$ and $P_i \supseteq P_j$ is a sufficient condition for T_i to be redundant, but not a necessary condition. Indeed, it is possible that $W(Z(P_i, T_i)) \subseteq W(Z(P_j, T_j))$ even without $P_i \supseteq P_j$.

Example 8.2. Let T_1 and T_2 be regular chains of $\mathbb{K}[x, y, z]$. Let (P_1, T_1) and (P_2, T_2) be a pair of triangularize tasks where

$$(P_1, T_1) = \left(\{x^2 - 1, y + 1\}, \left\{ \begin{array}{c} y^2 - 1 \\ z \end{array} \right) \right. \text{ and } (P_2, T_2) = \left(\{xz^2 + xz, y + 1\}, \left\{ \begin{array}{c} y^2 - 1 \\ z^2 + z \end{array} \right) . \right.$$

$W(T_1) \subseteq W(T_2)$ and neither P_1 nor P_2 is a superset of the other. However, we have:

$$Z(P_1, T_1) = \left\{ \begin{array}{l} x^2 - 1 \\ y + 1 \\ z \end{array} \right. \quad \text{and} \quad Z(P_2, T_2) = \left\{ \begin{array}{l} y + 1 \\ z^2 + z \end{array} \right.$$

implying $W(Z(P_1, T_1)) \subseteq W(Z(P_2, T_2))$.

Notice that in the previous example, reducing the elements of P_2 with respect to T_2 would yield $P_2' = \{y+1\} \subseteq P_1$. Therefore, leading to our previously stated sufficient condition for redundancy. However, it is often the case that computing such reductions and simplifications is a non-trivial amount of work, let alone the cost of computing whether or not $W(T_1) \subseteq W(T_2)$ holds. Further experimentation will need to explore the balance between cost and reward of additional processing toward potentially finding redundancies. Likely, a heuristic algorithm will need to be employed. The IsNotIncluded function is also heuristic, suggesting heuristics are indeed suitable to test for redundancies.

Given two tasks (P_i, T_i) and (P_j, T_j) let us write $(P_i, T_i) \preceq (P_j, T_j)$ if the task (P_i, T_i) is redundant with respect to (P_j, T_j). That is, $W(T_i) \subseteq W(T_j)$ and $P_i \supseteq P_j$. Algorithm 8.1 shows a simple scheme for combining triangularize tasks with the intermediate removal of redundant components. For each regular chain T' returned from Intersect there are three redundancy checks: (i) if T' is redundant with respect to any of the pending tasks, give up processing the task T'; (ii) if T' is redundant with respect to any of the completed tasks in $\mathcal{T}$, give up processing the task T'; (iii) if any of the pending tasks are redundant with respect to T', remove that task from the list of tasks.

Notice that we do not test whether any of the completed tasks in $\mathcal{T}$ are redundant with respect to any pending task. Indeed, pruning a leaf node will not avoid any redundant computation since that task is already fully completed. Moreover, one must still call RemoveRedundantComponents before returning since the sufficient condition described earlier is not necessary, and some redundancies may be missed by the intermediate checks.

The hybrid organization described in Algorithm 8.1 requires special attention if it is to be parallelized with the workpile pattern like TriangularizeByTasks of Algorithm 6.19. In particular, there may possibly be pending tasks in the workpile itself as well as tasks *in flight*— tasks which are actively being processed by some other thread. This is where the aforementioned asynchronous set inclusion testing and task cancellation must be used. To avoid the excessive synchronization caused by checking for redundancies in Triangularize "by level" (Algorithm 6.18), we should test for set inclusions asynchronously.

Asynchronicity is derived in two ways. First, the algorithm must check against tasks in flight without interrupting their processing. Second, where a triangularize task produces more than one component, it is useful to check for redundancies immediately after the component is produced, meanwhile Intersect continues to process the remaining components. If Intersect is implemented as a generator, then this is immediate. Otherwise, one may *fork* the computation to check for redundancies concurrently.

Algorithm 8.1 TriangularizeHybrid(F)

Input: a finite set $F \subseteq \mathbb{K}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbb{K}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
1:  Tasks := { (F, ∅) };  𝒯 := {}
2:  while |Tasks| > 0 do
3:  |   (P, T) := pop a task from Tasks
4:  |   Choose a polynomial p ∈ P; P' := P \ {p}
5:  |   for T' in Intersect(p, T) do
6:  |   |   if ∃ (P̃, T̃) ∈ Tasks, (P', T') ≼ (P̃, T̃) then
7:  |   |   |   continue                                  ▷ Drop T' as a task
8:  |   |   else if ∃ T̃ ∈ 𝒯, (P', T') ≼ (∅, T̃) then
9:  |   |   |   continue                                  ▷ Drop T' as a task
10: |   |   else if ∃ (P̃, T̃) ∈ Tasks, (P̃, T̃) ≼ (P', T') then
11: |   |   |   Tasks := Tasks \ {(P̃, T̃)}
12: |   |   else if |P'| = 0 then
13: |   |   |   𝒯 := 𝒯 ∪ {T'}
14: |   |   else
15: |   |   |   Tasks := Tasks ∪ {(P', T')}
16: return RemoveRedundantComponents(𝒯)
```

Tasks in flight require yet another important consideration when they are found to be redundant. This requires *task interruption and cancellation* to gracefully interrupt a running thread and have it self-terminate. Intrusively terminating a thread is ill-advised. For example, if a thread is in a critical region, its termination will leave a mutex locked forever. With the C++20 standard, the *Thread Support Library* introduced the notion of *cooperative interruption* via *stop tokens*. In this paradigm, a worker thread should occasionally ask its stop token whether or not a stop has been requested. If so, it should terminate itself gracefully. Cooperation arises where the request to stop is made to the stop token by another independent thread.

This presents a potential solution to task cancellation when redundancies are found. The idea of stop tokens can be integrated into the `ExecutorThreadPool` and `AsyncGenerator` classes to support this. Careful design will be needed so that the stop token, and concurrency in general, continues to be encapsulated and hidden from the client code. This is left to future work.

Improving Asynchronous Generators

As was discussed throughout Section 6.3, asynchronous generators have theoretically promising opportunities for concurrency and parallelism within triangular decomposi-

tion. However, experimentation showed surprising deficiencies in practice. Indeed, one flaw in implementation was discovered after careful examination. Some data was unnecessarily copied between threads as the data moved from producer to consumer of an asynchronous generator. Where data items are large (such as intermediate polynomials affected by expression swell), this data transfer between threads and cores leads to sharp declines in data locality and cache performance. This intuition is further confirmed where asynchronous generators were successfully employed in Hensel factorization (see Section 7.6), and they passed only integers between producer and consumer.

Moreover, we have implemented asynchronous generators and the pipeline pattern in a traditional way. One thread is responsible for one stage of the pipeline and data flows through the pipeline (i.e. is passed between threads). A different solution, and one used by Intel's Thread Building Blocks [131, Ch. 9], is to transpose functions and data so that one thread is bound to one data item and it is the functional stages of the pipeline that flow between threads. This approach is more complex but has the benefit of improving the locality of data items. This is particularly important where data to be passed is large and data locality is thus crucial for performance.

A final consideration for asynchronous generators is their role in the cooperative parallel schemes found within triangular decomposition. Since asynchronous generators consume a thread pool thread (if one is idle) as soon as they are created, they are occupying hardware resources whether or not concurrency is actually available to exploit. That is to say, one may launch an asynchronous generator just for it to return a single data item. We need a more robust solution, particularly where the generation of more than one data item is not guaranteed. A possible solution is for asynchronous generators and pipelines to proceed serially until it is guaranteed that multiple data items will be produced. Then, the generator can yield to its caller (or pipeline stage to the next stage) a generated data item and simultaneously fork the execution to proceed asynchronously with the consumer. Again, a careful design will be needed to ensure that `AsyncGenerator` continues to encapsulate its parallel aspects. Indeed, forking the execution once the client code region (i.e. producer) has already begun poses many potential challenges for data races and coherency.

Additional Low-level Parallelism

Experience gained from the implementation of Hensel factorization (Chapter 7) indicates that adding fine-grained regular parallelism is an effective strategy to help mitigate the irregularity of irregular parallelism. The fine-grained parallelism in question may not alone be sufficient to give the entire application reasonable amounts of parallelism or

parallel speed-up. However, combining it with coarser irregular parallelism is a valid strategy to obtain good parallel speed-up for the entire program.

In terms of triangular decomposition, we have already seen that the regular parallelism found in computing subresultants provided some benefits. However, this was limited to the univariate and bivariate cases. Continuing to recursively apply interpolation to achieve parallel schemes for trivariate subresultants and beyond is unlikely to lead to success. As the number of variables increases, the sparsity (the percentage of possible terms which are zero) of a polynomial typically does as well. Typical evaluation-interpolation schemes are therefore inefficient. Sparse interpolation (such as methods based on Zippel [190]) should be employed. Parallel implementations based on Zippel's method have already seen success; see, e.g., [139].

Other obvious areas where low-level parallelism can be exploited include arithmetic [138], GCDs [102], and factorization [139]. A particularly important operation is triangular decomposition is pseudo-division. This operation has already been the focus of a precise implementation targeting data locality [11, 31], but a parallel implementation is currently missing. Parallel pseudo-division will be examined in future work.

8.3 A Regular Chain "Universe"

Our final and most drastic proposed change for our next generation of triangular decomposition is the application of dynamic evaluation in a regular chain "universe". The main idea is that, for a polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$, the affine space $\overline{\mathbb{K}}^n$ is the "universe", and any regular chain created or manipulated throughout a triangular decomposition should have a same "view" of this universe.

Where triangular decomposition proceeds incrementally, one can view this as an iterative refinement of the universe. Let p be a monic polynomial so that the regular chain $T = \{p\}$ has $W(T) = \overline{W(T)}$. T then separates the affine space $\overline{\mathbb{K}}^n$ into two parts (which are not necessarily connected components), one where p is 0 (i.e. $W(T)$) and one where p is not 0 (i.e. $\overline{\mathbb{K}}^n \setminus W(T)$).

It is then easy to see that regular chains can be seen as a sort of tree, defining constraints on the affine space. Indeed, where a regular chain's quasi-component is not irreducible, there may be more than one possible choice of value on each variable. Consider $T = \{x^2 - y, y^2 - 3y + 2\}$. Through a process similar to back substitution, T corresponds to a splitting tree over $\overline{\mathbb{K}}^n$. This is modelled in Figure 8.1.

If regular chains had a shared view of the universe, then each regular chain (and each splitting tree) is able to make use of refinements and splittings discovered by other regular

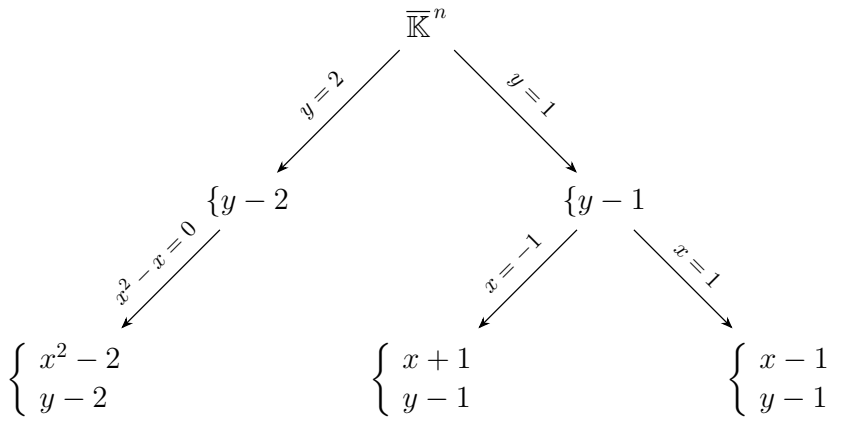

Figure 8.1: Viewing the regular chain $T = \{x^2 - y, y^2 - 3y + 2\}$ as a splitting tree.

chains. That is, a splitting discovered by one component may be immediately used by another component. Algebraically, this can be seen as discovering the components of a direct product of fields. We saw this applied in Example 2.16. Now, consider the context of regular chains and splitting of quasi-components in the following example.

Example 8.3. Let T_1, T_2 be regular chains of $\mathbb{K}[x > y > z]$ where

$$T_1 = \begin{cases} (z+2)y^2 + 2 \\ z^2 + 2z - 3 \end{cases} \quad \text{and} \quad T_2 = \begin{cases} y + 1 \\ z^2 + 2z - 3 \end{cases}$$

Assume that the factorization of $z^2+2z-3 = (z-1)(z+3)$ is not known. Now, consider $p = x^2yz + 3x^2y + x^2z + 3x^2 + x = (y+1)(z+3)x^2 + x$. We want to compute $Z(p, T_1)$ and $Z(p, T_2)$. Proceeding on T_1 causes a split where we regularize $(y+1)(z+3)$ and compute a regular GCD of $(z+3)$. The intersection produces T_3 and T_4; proceeding on T_2 produces T_5:

$$T_3 = \begin{cases} (4y+4)x^2 + x \\ 3y^2 + 2 \\ z - 1 \end{cases}, \quad T_4 = \begin{cases} x \\ y^2 - 2 \\ z + 3 \end{cases}, \quad T_5 = \begin{cases} x \\ y + 1 \\ z^2 + 2x - 3 \end{cases}.$$

If the next polynomial to intersect with T_5 was $q = xz+x+y^2-z-2 = (z-1)(x+1)+y^2-1$, then the intersection $Z(q, T_5)$ would again find the splitting of $z^2 + 2z - 3$ into $z - 1$ and $z + 3$. However, notice that T_5 could have been split using the same regular GCD $z + 3$,

yielding:

$$T_{5,a} = \begin{cases} x \\ y+1 \\ z-1 \end{cases} \quad \text{and} \quad T_{5,b} = \begin{cases} x \\ y+1 \\ z+3 \end{cases} .$$

Then, $Z(q, T_{5,a})$ would easily produce $T_{6,a} = T_{5,a}$. Meanwhile, the intersection $Z(q, T_{5,b})$ would quickly be abandoned because the iterated resultant $\text{res}(q, T_{5,b}) = -4 \neq 0$. That is to say, by sharing the splitting information of their lower parts, redundant computations could have been avoided between T_1 and T_5.

Towards this regular chain universe we have several goals.

(*i*) Every regular chain throughout a particular triangular decomposition should exist only once.

(*ii*) Where a splitting is found in one regular chain, any other regular chain that shares the same constraint should automatically be split as well.

(*iii*) This design must incorporate thread safety to enable component-level parallelism.

The first two goals suggest the need for a unique and shared data structure between all regular chains. That is, a single "universe" object that incorporates the many (different) splittings of affine space. In the case of triangular decomposition, the splitting of the affine space need not strictly be a partition. Indeed, redundant components are common and different branches in the tree may have overlapping geometry, particularly in positive dimension. The latter goal requires avoiding data races through synchronized access to the shared data structure, but also a careful implementation to minimize the impact of synchronization on parallel performance.

Let us begin with a possible design of the universe shared data object. From Figure 8.1 and its surrounding discussion, it seems reasonable to encode regular chains as a tree which incrementally adds constraints. The root node is the entire ambient space, and the leaf nodes are the current regular chains in the triangular decomposition. Following the recursive representation of a tree, each node in the tree (each regular chain) can be seen as a polynomial p, where $\text{mvar}(p) = v$, alongside another regular chain T_v^-. In view of Figure 8.1, let $T_1 = \{y - 2$ and $T_2 = \{y - 1$. The three leaf nodes could then be identified, respectively, as the pairs $(x^2 - 2, T_1)$, $(x + 1, T_2)$, $(x - 1, T_2)$.

Unfortunately, strictly encoding the regular chain as a tree is insufficient. Rather, a directed acyclic graph (DAG) is needed. Consider again Example 8.3. There, T_5 can be

seen as a child of T_2, adding the constraint $x = 0$. However, if we apply the splitting $z^2 + 2z - 3 \rightarrow z - 1, z + 3$, then T_5 no longer has a unique path to the root node ($\overline{\mathbb{K}}^n$), and the graph is no longer a tree. Let us call such splits *latent splits*. These latent splits will require special care in our design as they transform the tree into a *rooted DAG* (a DAG with a single source node) and cause one regular chain to be viewed as many.

When the graph encoding the universe is a tree, each active regular chain in the universe corresponds to a leaf node. When the encoding is rather a rooted DAG, each active regular chain is the universe corresponds to a path between the root and a leaf node (sink node). Even with considering possible latent splits, we can still implement the DAG in a recursive way, where each regular chain is a polynomial paired with a lower regular chain. The base case, where there is no lower regular chain, uses the empty set as its lower regular chain.

We suggest implementing the DAG implicitly through a combination of references to its leaf nodes and a dictionary (hash table). An individual node in the DAG will be modelled as a polynomial paired with a lower regular chain. However, the lower regular chain in the pair will be a dictionary key rather than a regular chain explicitly. This key will allow traversal of the DAG through recursively accessing nodes via dictionary keys. If, rather, (a pointer to) the lower regular chain was stored explicitly, then a latent split would require updating every child of the node being split. Using the dictionary allows the implicit tree to be modified by only modifying one dictionary entry. To do so, the key of the node which was split becomes associated with a list of regular chains that is the result of that split.

Given a polynomial ring $\mathbb{K}[x_1 > \ldots > x_n]$, we know the maximum length of any path in the DAG is equal to n. Each path can then be represented as a list of integers of length at most n. In the worst case, a dictionary key could be an array (or concatenation) h of n positive integers such that $h[i]$ corresponds to a node whose polynomial has main variable x_{i+1}, and the value of $h[i]$ indexes (as $1, 2, \ldots$) a particular node at that level of the DAG. The root would be given a special key where all entries of this list are 0. Keys corresponding to paths of length less than n would simply be padded with 0's. A more sophisticated implementation could use bit-wise operations to *pack* the indices into a single machine word. Assume a machine word is 64 bits. Then, a key could be a 64-bit integer with $\lfloor 64/n \rfloor$ sequential bits corresponding to each x_i from 1 to n.

Consider Figure 8.2. In the first diagram with a see a regular chain tree and its corresponding dictionary for the universe defined by $\mathbb{K}[x_1 > x_2 > x_3]$. With the idea of keys as given in the previous paragraph, the non-zero entries of each key along a path form a "super set" of its parent. For example, $h_4 = [0, 2, 1] \supset h_1 = [0, 0, 1]$, and

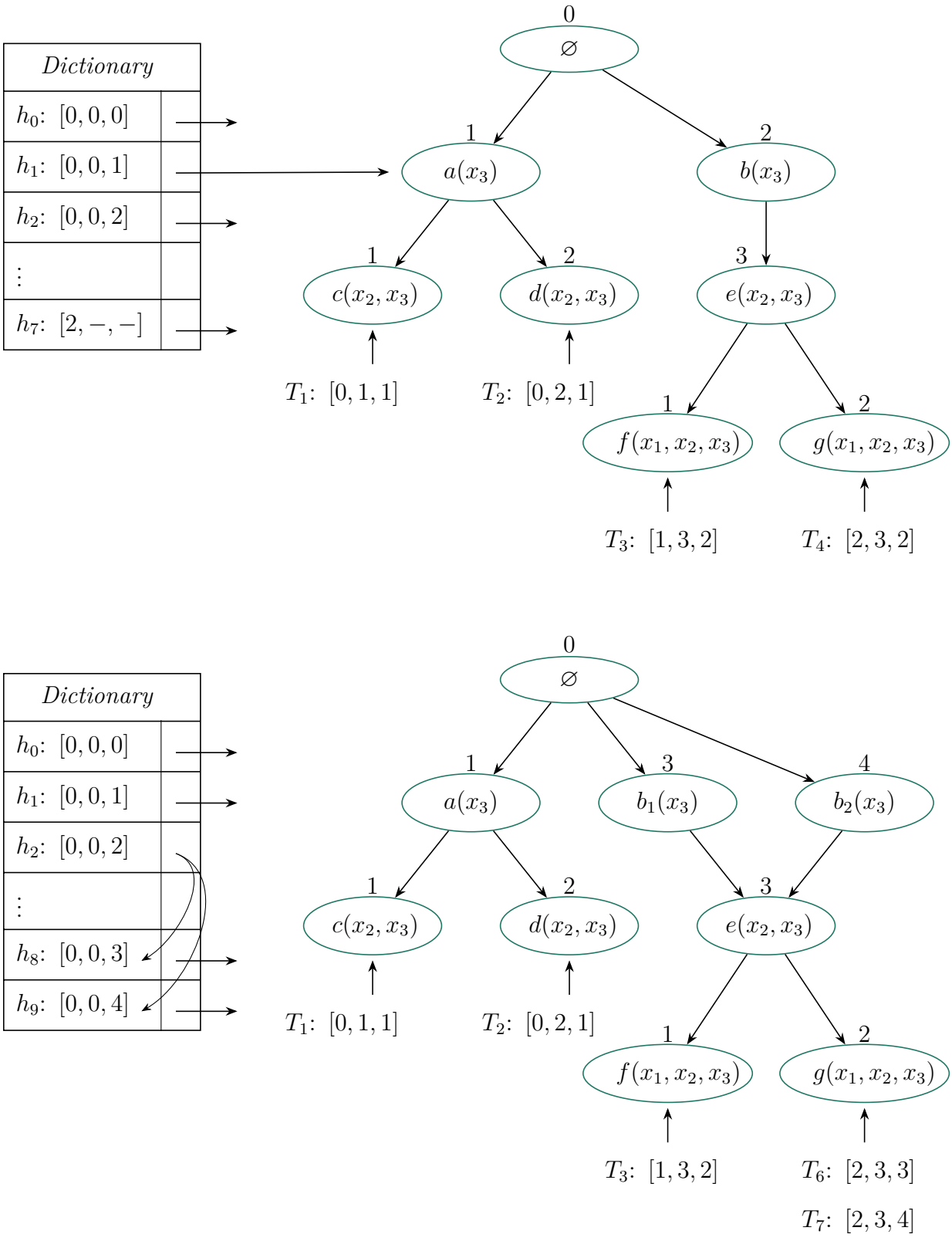

Figure 8.2: A latent split in the regular chain universe, the replacement of node $b(x_3)$ with nodes $b_1(x_3)$ and $b_2(x_3)$. This causes T_3 to become outdated, and a traversal starting from T_3 will reveal a split as the dictionary entry for $[0,0,2]$ now points to two other dictionary entries.

.

$h_6 = [1, 3, 2] \supset h_5 = [0, 3, 2]$. The dictionary then associates keys with nodes in the DAG. This gives a "horizontal" view of the splitting tree, where one can access any particular node in the tree. Given a key, the dictionary would extract the first non-zero index to find the corresponding node in the tree. Meanwhile, the remaining parts of key give the remaining parts of the path. Therefore, the DAG is only represented implicitly through keys and the paths they represent.

Still looking at the first diagram in Figure 8.2, the active regular chains T_1, T_2, T_3, T_4, correspond to a leaf node and a path through the DAG. These regular chain objects act as the client's view of the regular chain universe. That is, the view of TRIANGULARIZE and its subroutines. These objects give a "vertical" view of the splitting tree, each corresponding to a particular path in the DAG, and thus a particular regular chain. For example, T_4 has the path $[2, 3, 2]$. T_4 thus encodes the path accessing the node at index 2 for main variable x_1, the node at index 3 for main variable x_2, and the node at index 2 for main variable x_3. Note that the $-$ entries in the dictionary keys correspond to a wild card value. Any key with 2 in its first position maps to the node $g(x_1, x_2, x_3)$.

The second diagram of Figure 8.2 shows the result of a latent split; $b(x_3)$ has been split into $b_1(x_3)$ and $b_2(x_3)$. Notice that this changes the node indices at the level corresponding to main variable x_3. Node 2 with $b(x_3)$ split into node 3 with $b_1(x_3)$ and node 4 $b_2(x_3)$. Ensuring each node has a unique id, including the entire past history, is important, as we will now see.

This latent split affects T_3 and T_4. Let us assume that computations with T_4 triggered the split, and therefore T_4 is replaced with T_6 and T_7 in the normal way. However, T_3 has not yet discovered the split. It will discover the split automatically, however, as soon as it attempts to make a tree traversal. From the latent split, the dictionary's entry for h_2 has been replaced with a list $[h_8, h_9]$. When a tree traversal with T_3 begins, its key indicates that the path should go from node 3 at level x_2 to node 2 at level x_3. However, attempting to access node 2 at level x_3 (i.e. h_2) in the dictionary will return a list of new keys rather than a DAG node. Generically, any tree traversal has the potential to return multiple paths (i.e. multiple regular chains), where new splittings have been witnessed.

Throughout the TRIANGULARIZE algorithms and its subroutines, tree traversals occur through recursive subroutine calls with T_v^-. Therefore, any access to T_v^- has the potential to return multiple regular chains. Fortunately, little modification is needed in the triangular decomposition algorithms since almost every operation already returns a list. For example, a call to regularize with $p = 1$ and $T = \{x_2^2 - 1, (x_3 - 1)(x_3 + 3)\}$ would normally trivially return $[(1, T)]$, since $p \in \mathbb{K}$. However, if the algorithm discovers that $T_{x_1}^-$ has been split by a latent splitting, it can return $[(1, T_1), (1, T_2)]$, with

$T_1 = \{x_2^2 - 1, x_3 - 1\}$, and $T_2 = \{x_2^2 - 1, x_3 + 3\}$. The client (i.e. function caller) therefore does not need to implement any special case for when a latent splitting is discovered.

The one exception pseudo-division; its specification expects a single pseudo-remainder to be returned. There are two places where this pseudo-division is used explicitly. The first is Intersect (Algorithm 6.7), where a pseudo-remainder computation determines if the input polynomial p is already contained in $\text{sat}(T)$. The second is Regularize (Algorithm 6.13), where a pseudo-quotient is used to explicitly split the regular chain into two components: one replacing T_v with the regular GCD, and the other replacing T_v with the pseudo-quotient. A simple modification to $\text{pquo}(p, T)$ (resp. $\text{prem}(p, T)$) can be made so that it returns a list of pairs of pseudo-quotients (resp. pseudo-remainders) and regular chains $(p_1, T_1), \ldots, (p_e, T_e)$ such that, for $1 \leq i \leq e$, p_i is the pseudo-quotient (resp. pseudo-remainder) of p with respect to T_i and $T \to T_1, \ldots, T_e$.

While pseudo-division is the only place where the caller function will explicitly need to be modifierd, the called functions will require a simple transformation. In the Triangularize algorithms as specified in Section 6.1.1, each takes as input only one regular chain and returns a list. However, latent splits mean that a single regular chain T may have since been replaced by multiple paths and thus multiple regular chains. A simple transformation is possible. Let us take Regularize as an example, with the other subroutines following symmetrically. We define a new Regularize which takes a potentially "outdated" path, traverses the regular chain tree, and calls the original Regularize for every unique path it finds during the traversal. This procedure is shown in Algorithm 8.2.

Notice that, for the sake of pseudo code, we directly access the dictionary and paths of the regular chains. In reality, Lines 1–19 of Algorithm 8.2 would be encapsulated as a single method call to the regular chain class. A potential method signature[1] is:

```
1 std::vector<RegularChain> RegularChain::getPaths();
```

Where no latent split has occurred, this method simply returns a `vector` of length one containing the original regular chain. Moreover, notice that since a regular chain is encoded as a path, this function only returns more paths. Each path is a list of n integers (or a single *packed integer*), and the underlying polynomials are not copied.

One key consideration has been missing so far from this discussion. In Figure 8.2, the regular chains were all constructed "bottom-up". Every regular chain has a univariate polynomial in the least-ordered variable, then a bivariate polynomial in the two least-ordered variables, etc. However, this is typically not the case. Working in $\mathbb{K}[x_1 > x_2 > x_3]$, a single incremental step will usually produce a regular chain T with a single

[1]Note that we omit the `RegularChain` template parameters for clarity

Algorithm 8.2 REGULARIZEPATH(p, T)

Input: $p \in \mathbb{K}[x_1 > \cdots > x_n]$, T a regular chain of $\mathbb{K}[x_1 > \cdots > x_n]$ given as a path through the universe tree.

Output: A set of pairs $(p_1, T_1), \ldots, (p_e, T_e)$ such that for each $1 \leq i \leq e$, T_i is a regular chain, $p \equiv p_i \bmod \sqrt{\text{sat}(T_i)}$, and $T \to T_1, \ldots, T_e$.

```
root := [0, ..., 0]; newPath := [0, ..., 0]                    ▷ lists of length n
Paths := {newPath}
h_T := the path associated with T.
i := the index of the first non-zero element of h_T
while h_T ≠ root do                                             ▷ traverse the tree
    N := Dictionary(h_T)
    if N is a node then                                         ▷ then no split, just continue traversal
        for path ∈ Paths do
            path[i] := h_T[i]                                   ▷ modify in-place
    else
        NewPaths := ∅
        for h_N ∈ N do                                          ▷ N is a list of keys
            for path ∈ Paths do
                newPath := path
                newPath[i] := h_N[i]                            ▷ one path of the latent split
                NewPaths := NewPaths ∪ {newPath}
        Paths := NewPaths
    h_T[i] = 0                                                  ▷ Set key's current index to 0 to traverse up the path
    i := i + 1
T := ∅
for path ∈ Paths do
    T := T ∪ REGULARIZE(p, Dictionary(path))
return T
```

polynomial in all three variables. Then, the second incremental step will compute a resultant between the polynomial in T and a new polynomial from the input system p. Assume that $\text{mvar}(T) = \text{mvar}(p) = x_1$. This resultant will have main variable (at most) x_2, and will recursively be intersected with $T_{x_1}^{-}$ producing, say, T_1. Now, $\dim(T_1) < \dim(T_{x_1}^{-})$. This causes T_{x_1} to be regularized against T_1. Generically, this may lead to a splitting or cause T_{x_1} to be reduced/modified in some way.

In terms of the regular chain universe tree, this would cause a node to be *inserted* into the tree along the path of an existing regular chain, rather than an existing node being replaced by multiple. When a splitting occurs in the same dimension, i.e. a latent split, as shown in Figure 8.2, dimension does not drop and the computation can safely continue along each path. However, when a node is inserted along a path, dimension necessarily

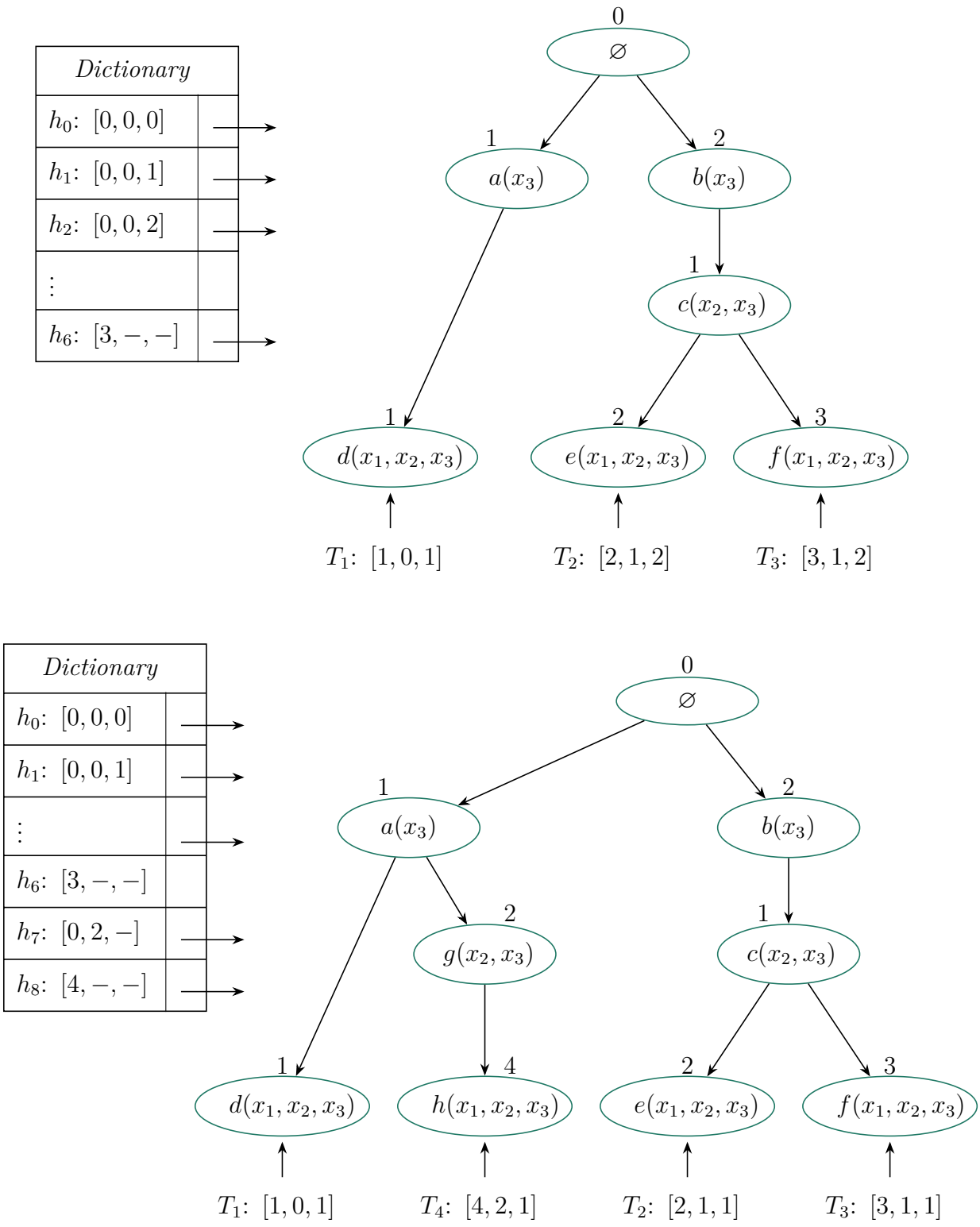

Figure 8.3: Adding a new regular chain to the regular chain universe. T_4 is the result of an intersection between T_1 and a new polynomial p such that T_1 and T_4 have the polynomial $a(x_3)$ in common.

.

drops, and the regularity of the upper part of the regular chain is not maintained in general. In this scenario, we must construct a new path so that the original path is not modified, resulting in the upper part of the chain potentially being duplicated. Using the notation of the previous paragraph, the regular chain T will continue to exist on one path meanwhile we will construct a new path for $T_2 := Z(T_{x_1}, T_1)$. Figure 8.3 shows an example this occurring. In this example, notice that any nodes *below* the inserted node can be shared between paths, but nodes *above* it should be duplicated.

We conclude now by commenting on this design in the context of component-level parallelism. In our design of the regular chain universe, the DAG representing the regular chains is implicit through the paths described by dictionary keys and pointers to the DAG nodes are the dictionary entries. Moreover, individual nodes are not modified. Nodes are effectively read-only and do not require any synchronization. Indeed, TRIANGULARIZE subroutines and latent splits are not destructive. Rather, they only replace existing nodes in the DAG. Because of this, the only synchronization which is required is in the access to the dictionary. In particular, each access to the hash table and each modification or addition to the hash table should be an *atomic* operation. Consider the latent split shown in Figure 8.2. The node with $b(x_3)$ was removed implicitly from the DAG when h_2 had its dictionary entry (pointer to the node) replaced with the list $[h_8, h_9]$. The node itself still exists, although it may be unreachable. This brings us to our last point.

Special attention must be to given to memory usage in this design. Although nodes of the DAG are not modified, the references to them from the dictionary may be removed. This leads to two possible scenarios. The first is unreachable memory, where all pointers to a particular node are lost and that memory segment can never be freed. The second is that, if the node is destroyed at the same time that its entry is removed from the hash table, then memory corruption is possible if another thread is still accessing that node. A potential solution is for the regular chain universe to incorporate a *garbage collection* mechanic. In a naive implementation, references to nodes which are removed from the hash table could be moved into a "trash pile". Then, once a triangular decomposition has fully completed, any node in the trash pile is destroyed. Another solution is to use reference counting on the nodes, destroying it once no references to it remain.

However, it is likely that the naive solution is sufficient. Indeed, since the regular chain universe ensures that regular chains are unique, there is no data copies occurring and the overall memory footprint of the triangular decomposition should be quite small. It is likely that, even without freeing the "trash pile", that a triangular decomposition based on the regular chain universe would use much smaller amounts of memory than our current implementation.

Chapter 9

Conclusions and Future Work

In this thesis we have examined data structures, algorithms, software design, parallelism, and high-performance considerations for solving systems of polynomial equations and polynomial computer algebra more generally. We have realized two implementations (Chapter 6 and Section 8.1) of triangular decomposition in C/C++ which employ these practical techniques for improved performance. Compared to the leading implementation of triangular decomposition of the *RegularChains* library [125], our implementation is up to 30× faster on the thousands of polynomial systems we have tested.

Improved performance has been derived from a variety of sources. In the low-level algorithms performing core operations, our implementation is carefully implemented, targeting data locality. For example, the crucial operation of computing subresultants has been made more efficient through speculative computation, avoiding unnecessary computations, and through exploiting parallelism.

This low-level parallelism is combined with other areas for component-level parallelism in the triangular decomposition algorithms themselves. We have analyzed the algorithms to find concurrency opportunities and then implemented those opportunities in a multithreaded paradigm. The main Triangularize algorithm can be organized in several different ways to take advantage of different algorithmic properties. In one organization, redundant computations are avoided (Algorithm 6.18). In another, the workpile pattern is applied and greater parallelism is possible (Algorithm 6.19). It has also been observed that the subroutines of Triangularize create a dynamic pipeline through their mutual recursion, each producing a collection of regular chains. Modelling those routines as generators, the components may *flow* between subroutines without synchronization.

Further improvements for performance in triangular decomposition revolve around improved parallelism and avoiding even more redundant computations. Through the

lens of dynamic evaluation, we have examined data structures which implement a *regular chain universe* that avoids redundant computations by handling so-called *latent splits*. With unique instances of regular chains automatically sharing information between them, redundant computations should be avoided and overall memory usage drastically reduced.

Pipelines, in Chapter 7, have also been successfully applied in the context of Hensel factorization for polynomials with multivariate power series coefficients. Hensel factorization is a special case of the Hensel-Sasaki Construction, or the Extended Hensel Construction (EHC). EHC, in turn, can be used to compute the non-trivial limit points of a regular chain's quasi-component [5]. That is, one can compute $\overline{W(T)} \setminus W(T)$. A high-performance implementation of Weierstrass preparation theorem and Hensel factorization is one step toward implementing EHC and obtaining an effective method to compute those limit points. This implementation required multivariate power series, which have implemented using lazy evaluation with great success. Not only do lazy multivariate power series allow for the implementation of Hensel factorization as a parallel pipeline, but it is also an effective strategy to avoid redundant computations and greatly improve serial performance.

These various functionalities have been integrated into the Basic Polynomial Algebra Subprograms (BPAS) library [7]. This library provides high-performance implementations of polynomial algebra routines focusing on cache complexity and parallelization for effective computation on modern computer hardware. In addition to functionality, we have discussed several areas of software design which have been applied to the BPAS library.

Considering parallelism, Chapter 5 discussed the implementation of a generic and object-oriented parallelism framework based on the dynamic multithreading paradigm. This module provides generic support for the implementation of parallel patterns and parallel code regions in a way that encapsulates the difficulties of parallel programming within the object-oriented interface. This module enabled basic cooperation between parallel regions through dynamic scheduling and "priority" tasks.

Considering specifically computer algebra software, Chapter 4 described the design of BPAS which provides a type-safe, extensible, and adaptable object-oriented interface. Regarding type safety, it is possible that, through polymorphism, two computationally type safe objects are used together in mathematically incompatible ways. Our solution uses template metaprogramming and the Curiously Recurring Template Pattern (CRTP) to ensure mathematical type safety alongside computational type safety. Template metaprogramming is also used to provide a class hierarchy of algebraic types which is adaptable to, and extensible by, end-users.

Future Work

In recent years, algorithms and implementations in computer algebra have continued to improve upon their performance and accessibility. Continued success of computer algebra systems like *Maple* and *Mathematica* indicate that computer algebra methods are seeing increased use in practice. From the work presented in this thesis, continued directions for future work revolve around improved accessibility and performance of polynomial algebra.

- The open-source BPAS library has been designed specifically for simultaneous ease-of-use and performance. BPAS follows a layered architecture where high-performance C functions are wrapped in layers of parallelism and a C++ object-oriented interface. Another layer could add an interactive interface for improved ease-of-use. In particular, integration of BPAS into the *SageMath* ecosystem is a natural direction. More generally, a Python interface could be developed for BPAS to integrate its functionality into a widely used environment.

- Considering irregular parallelism generally, our work suggests that cooperation, dynamic load-balancing, and layering of parallel regions are required to achieve good parallel speed-up. Our object-oriented multithreading support should be extended to support a more robust and dynamic paradigm. In particular, one should investigate scheduling algorithms which incorporate dynamic information about the work loads and granularity of a requested parallel region. When a concurrency opportunity is discovered, the runtime must answer several questions. If all resources are occupied, should the new concurrency opportunity simply execute in serial? Should the new concurrency opportunity be queued and executed once resources become available? Should other parallel regions be paused and the new opportunity allowed to execute in parallel? With the addition of *task cancellation*, introduced in the C++20 standard, the latter could be implemented effectively.

- Component-level parallelism and low-level parallelism throughout triangular decomposition should continue to be developed. Improved implementation of asynchronous generators are one direction. Another direction is to solve systems over a prime field, where components are more likely to split, and then lift solutions back to the rational numbers. Yet another direction is the implementation of low-level regular parallelism in polynomial arithmetic, like pseudo-division, to help load-balance the irregular component-level parallelism.

- Triangular decomposition in the view of dynamic evaluation should be implemented to avoid redundant computations and excessive memory usage. Our design of the so-called regular chain universe provides a possible direction. That design already considers component-level parallelism in order to combine dynamic evaluation and parallelism towards a next-generation triangular decomposition implementation.

- The ideas of the Hensel factorization pipeline and lazy multivariate power series should be expanded and applied to multivariate Puiseux series and the Extended Hensel Construction. Not only is an effective implementation of EHC useful on its own [4], but it may also be employed to bolster triangular decomposition through the computation of limit points.

Bibliography

[1] J. Abbott and A. M. Bigatti. *CoCoALib: a C++ library for doing Computations in Commutative Algebra*. Available at `http://cocoa.dima.unige.it/cocoalib`.

[2] W. W. Adams and P. Loustaunau. *An introduction to Grobner bases*. Vol. 3. Graduate texts in mathematics. American Mathematical Society, 1994.

[3] P. Alvandi. "Computing Limit Points of Quasi-components of Regular Chains and its Applications". In: (2017).

[4] P. Alvandi, M. Ataei, M. Kazemi, and M. Moreno Maza. "On the Extended Hensel Construction and its application to the computation of real limit points". In: *Journal of Symbolic Computation* 98 (2020), pp. 120–162.

[5] P. Alvandi, C. Chen, and M. M. Maza. "Computing the Limit Points of the Quasi-component of a Regular Chain in Dimension One". In: *Proc. of CASC 2013*. Vol. 8136. Lecture Notes in Computer Science. Springer, 2013, pp. 30–45.

[6] P. Alvandi, M. Kazemi, and M. Moreno Maza. "Computing limits with the regularchains and powerseries libraries: from rational functions to Zariski closure". In: *ACM Communications in Computer Algebra* 50.3 (2016), pp. 93–96.

[7] M. Asadi, A. Brandt, C. Chen, S. Covanov, M. Kazemi, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS) v. 1.791*. `http://www.bpaslib.org`. 2022.

[8] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Sparse Polynomial Arithmetic with the BPAS Library". In: *Proc. of CASC 2018*. Vol. 11077. Lecture Notes in Computer Science. 2018, pp. 32–50.

[9] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "Parallelization of Triangular Decompositions: Techniques and Implementation". In: *Journal of Symbolic Computation* (2022). *(to appear)*.

[10] M. Asadi, A. Brandt, M. Kazemi, M. Moreno Maza, and E. Postma. "Multivariate Power Series in Maple". In: *Proc. of MC 2020*. Vol. 1414. Communications in Computer and Information Science. 2020, pp. 48–66.

[11] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[12] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "On the parallelization of triangular decompositions". In: *Proc. of ISSAC 2020*. ACM, 2020, pp. 22–29.

[13] M. Asadi, A. Brandt, and M. Moreno Maza. "Computational Schemes for Subresultant Chains". In: *Proc. of CASC 2021*. Vol. 12865. Lecture Notes in Computer Science. Springer, 2021, pp. 21–41.

[14] G. Attardi and C. Traverso. "Strategy-Accurate Parallel Buchberger Algorithms". In: *Journal of Symbolic Computation* 22 (1996), pp. 1–15.

[15] P. Aubry. "Ensembles triangulaires de polynomes et resolution de systemes algebriques. Implantation en Axiom." PhD thesis. Paris, France: Universite Paris VI, 1999.

[16] P. Aubry, D. Lazard, and M. Moreno Maza. "On the Theories of Triangular Sets". In: *Journal of Symbolic Computation* 28.1-2 (1999), pp. 105–124.

[17] P. Aubry and M. Moreno Maza. "Triangular Sets for Solving Polynomial Systems: a Comparative Implementation of Four Methods". In: *Journal of Symbolic Computation* 28.1-2 (1999), pp. 125–154.

[18] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Vol. 10. Algorithms and Computation in Mathematics. Springer, 2006.

[19] E. Becker, T. Mora, M. Grazia Marinari, and C. Traverso. "The Shape of the Shape Lemma". In: *Proc. of ISSAC 1994*. ACM, 1994, pp. 129–133.

[20] L. A. Belady. "A Study of replacement algorithms for virtual storage computers". In: *IBM Systems Journal, 5:78–101* (1966).

[21] M. Ben-Ari. *Principles of concurrent and distributed programming*. PHI Series in computer science. Prentice Hall, 1990.

[22] L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K. Heal, G. Labahn, J. P. May, J. McCarron, M. B. Monagan, D. Ohashi, and S. M. Vorkoetter. *Maple Programming Guide*. `www.maplesoft.com/documentation_center/maple2020/ProgrammingGuide.pdf`. 2020.

[23] J. Berthomieu, G. Lecerf, and G. Quintin. "Polynomial root finding over local rings and application to error correcting codes". In: *Appl. Algebra Eng. Commun. Comput.* 24.6 (2013), pp. 413–443.

[24] D. Bini and B. Mourrain. *Polynomial test suite*. `https://www-sop.inria.fr/saga/POL/`. Accessed: 2020-12-15. 1999.

[25] F. Biscani. "Parallel sparse polynomial multiplication on modern hardware architectures". In: *Proc. of ISSAC 2012*. 2012, pp. 83–90.

[26] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *Jorunal of the ACM* 46.5 (1999), pp. 720–748.

[27] J. Böhm, W. Decker, A. Frühbis-Krüger, F.-J. Pfreundt, M. Rahn, and L. Ristau. "Towards Massively Parallel Computations in Algebraic Geometry". In: *Foundations of Computational Mathematics* 21.3 (2021), pp. 767–806.

[28] J. Böhm, W. Decker, S. Laplagne, G. Pfister, A. Steenpaß, and S. Steidel. "Parallel algorithms for normalization". In: *Journal of Symbolic Computation* 51 (2013), pp. 99–114.

[29] W. Bosma, J. Cannon, and C. Playoust. "The Magma algebra system. I. The user language". In: *Journal of Symbolic Computation* 24.3-4 (1997), pp. 235–265.

[30] F. Boulier, F. Lemaire, and M. Moreno Maza. "Well known theorems on triangular systems and the D5 principle". In: *Transgressive Computing 2006, Proceedings*. Granada, Spain, 2006.

[31] A. Brandt. "High Performance Sparse Multivariate Polynomials: Fundamental Data Structures and Algorithms". MSc thesis. London, ON, Canada: University of Western Ontario, 2018.

[32] A. Brandt, M. Kazemi, and M. Moreno Maza. "Power Series Arithmetic with the BPAS Library". In: *Proc. of CASC 2020*. Vol. 12291. Lecture Notes in Computer Science. Springer, 2020, pp. 108–128.

[33] A. Brandt and M. Moreno Maza. "On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation". In: *Proc. of CASC 2021*. Vol. 12865. Lecture Notes in Computer Science. Springer, 2021, pp. 78–99.

[34] P. A. Broadbery, T. Gómez-Díaz, and S. M. Watt. "On the Implementation of Dynamic Evaluation". In: *Proc. of ISSAC 1995*. ACM, 1995, pp. 77–84.

[35] M. Bronstein, M. Moreno Maza, and S.M. Watt. "Generic programming techniques in ALDOR". In: *Asian Workshop on Foundations of Software, AWFS 2007, Proceedings*. 2007, pp. 72–77.

[36] W. S. Brown. "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors". In: *Journal of the ACM* 18.4 (1971), pp. 478–504.

[37] W. D. Brownawell. "Bounds for the degrees in the Nullstellensatz". In: *Annals of Mathematics* 126.3 (1987), pp. 577–591.

[38] B. Buchberger. "The parallelization of critical-pair/completion procedures on the L-Machine". In: *Japanese Symposium on Functional Pogramming, Proceedings*. 1987, pp. 54–61.

[39] B. Buchberger. "Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal". PhD thesis. Innsbruck, Austria: Universitat Insbruck, 1965.

[40] R. Bündgen, M. Göbel, and W. Küchlin. "A fine-grained parallel completion procedure". In: *Proc. of ISSAC 1994*. ACM. 1994, pp. 269–277.

[41] W. H. Burge and S. M. Watt. "Infinite structures in Scratchpad II". In: *Proc. of EUROCAL 1987*. Vol. 379. Lecture Notes in Computer Science. Springer, 1987, pp. 138–148.

[42] J. F. Canny. "Some Algebraic and Geometric Computations in PSPACE". In: *ACM Symposium on Theory of Computing, Proceedings*. ACM, 1988, pp. 460–467.

[43] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post. "Software development environments for scientific and engineering software: A series of case studies". In: *Proc. of ICSE 2007*. IEEE Computer Society. 2007, pp. 550–559.

[44] J. C. Carver, N. C. Hong, and S. Ciraci. "The 4th International Workshop on Software Engineering for HPC in Computational Science and Engineering". In: *Computing in Science and Engineering* 19.2 (2017), pp. 91–95.

[45] B. W. Char, K. O. Geddes, and G. H. Gonnet. "GCDHEU: Heuristic Polynomial GCD Algorithm Based On Integer GCD Computation". In: *Journal of Symbolic Computation* 7.1 (1989), pp. 31–48.

[46] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. "The Basic Polynomial Algebra Subprograms". In: *Proc. of ICMS 2014*. 2014, pp. 669–676.

[47] C. Chen and M. Moreno Maza. "Algorithms for computing triangular decomposition of polynomial systems". In: *Journal of Symbolic Computation* 47.6 (2012), pp. 610–642.

[48] C. Chen, M. Moreno Maza, and Y. Xie. "Cache Complexity and Multicore Implementation for Univariate Real Root Isolation". In: *Journal of Physics: Conference Series* 341 (2011).

[49] C. Chen. "Solving Polynomial Systems via Triangular Decomposition". PhD thesis. London, ON, Canada: University of Western Ontario, 2011.

[50] C. Chen, J. H. Davenport, J. P. May, M. Moreno Maza, B. Xia, and R. Xiao. "Triangular decomposition of semi-algebraic systems". In: *Journal of Symbolic Computation* 49 (2013), pp. 3–26.

[51] C. Chen, O. Golubitsky, F. Lemaire, M. Moreno Maza, and W. Pan. "Comprehensive Triangular Decomposition". In: *Proc. of CASC 2007*. 2007, pp. 73–101.

[52] C. Chen and M. Moreno Maza. "Algorithms for computing triangular decompositions of polynomial systems". In: *Proc. of ISSAC 2011*. ACM, 2011, pp. 83–90.

[53] C. Chen and M. Moreno Maza. "An Incremental Algorithm for Computing Cylindrical Algebraic Decompositions". In: *Proc. of Asian Symposium on Computer Mathematics ASCM 2012*. Springer, 2012, pp. 199–221.

[54] C. Chen and M. Moreno Maza. "Quantifier elimination by cylindrical algebraic decomposition based on regular chains". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 2014, Proceedings*. ACM, 2014, pp. 91–98.

[55] S.-C. Chou and X.-S. Gao. "Ritt-Wu's Decomposition Algorithm and Geometry Theorem Proving". In: *Proc. of Conference on Automated Deduction CADE 1990*. Vol. 449. Lecture Notes in Computer Science. Springer, 1990, pp. 207–220.

[56] D. V. Chudnovsky and G. V. Chudnovsky. "On expansion of algebraic functions in power and Puiseux series I". In: *Journal of Complexity* 2.4 (1986), pp. 271–294.

[57] G. E. Collins. "The Calculation of Multivariate Polynomial Resultants". In: *Journal of the ACM* 18.4 (1971), pp. 515–532.

[58] R. M. Corless and N. Fillion. *A graduate introduction to numerical methods*. Springer, 2013.

[59] S. Covanov, D. Mohajerani, M. M. Maza, and L. Wang. "Big Prime Field FFT on Multi-core Processors". In: *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation, ISSAC 2019, Beijing, China, July 15-18, 2019*. 2019, pp. 106–113.

[60] S. Covanov, D. Mohajerani, M. Moreno Maza, and L. Wang. "Big Prime Field FFT on Multi-core Processors". In: *Proc. of ISSAC 2019*. 2019, pp. 106–113.

[61] D. A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. 4th. Undergraduate texts in mathematics. Springer, 2015.

[62] L. Dagum and R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *Computing in Science and Engineering* 5.1 (1998), pp. 46–55.

[63] X. Dahan, A. Kadri, and É. Schost. "Bit-size estimates for triangular sets in positive dimension". In: *Journal of Complexity* 28.1 (2012), pp. 109–135.

[64] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. "Lifting techniques for triangular decompositions". In: *Proc. of ISSAC 2005*. 2005, pp. 108–115.

[65] X. Dahan and É. Schost. "Sharp estimates for triangular sets". In: *Proc. of ISSAC 2004*. ACM, 2004, pp. 103–110.

[66] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR *4-1-1 — A computer algebra system for polynomial computations*. `http://www.singular.uni-kl.de`. 2018.

[67] J. D. Dora, C. Dicrescenzo, and D. Duval. "About a New Method for Computing in Algebraic Number Fields". In: *European Conference on Computer Algebra, EUROCAL 1985, Proceedings Volume 2: Research Contributions*. Vol. 204. Lecture Notes in Computer Science. Springer, 1985, pp. 289–290.

[68] L. Ducos. "Optimizations of the subresultant algorithm". In: *Journal of Pure and Applied Algebra* 145.2 (2000), pp. 149–163.

[69] L. Ducos. "Algorithme de Bareiss, algorithme des sous-résultants". In: *Informatique Théorique et Applications* 30.4 (1996), pp. 319–347.

[70] D. Duval. "Algebraic Numbers: An Example of Dynamic Evaluation". In: *Journal of Symbolic Computation* 18.5 (1994), pp. 429–445.

[71] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. "How do API documentation and static typing affect API usability?" In: *Proc. of ICSE 2014*. ACM. 2014, pp. 632–642.

[72] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases (F4)”. In: *Journal of pure and applied algebra* 139.1-3 (1999), pp. 61–88.

[73] J.-C. Faugere. “A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)”. In: *Proc. of ISSAC 2002*. ACM, 2002, pp. 75–83.

[74] J.-C. Faugere. “Parallelization of Gröbner Basis”. In: *Proc. of PASCO 1994*. Vol. 5. World Scientific. 1994, p. 124.

[75] J.-C. Faugère. “FGb: A Library for Computing Gröbner Bases”. In: *Mathematical Software - ICMS 2010, Proceedings*. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 84–87.

[76] J.-C. Faugère, P. M. Gianni, D. Lazard, and T. Mora. “Efficient Computation of Zero-Dimensional Gröbner Bases by Change of Ordering”. In: *Journal of Symbolic Computation* 16.4 (1993), pp. 329–344.

[77] G. Fischer. *Plane Algebraic Curves*. AMS, 2001.

[78] P. Fortin, A. Fleury, F. Lemaire, and M. B. Monagan. “High-performance SIMD modular arithmetic for polynomial evaluation”. In: *Concurr. Comput. Pract. Exp.* 33.16 (2021).

[79] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-Oblivious Algorithms”. In: *ACM Transactions on Algorithms* 8.1 (2012), p. 4.

[80] M. Frigo and V. Strumpen. “The cache complexity of multithreaded cache oblivious algorithms”. In: *Proc. of SPAA*. 2006, pp. 271–280.

[81] G. Gallo and B. Mishra. “Efficient algorithms and bounds for Wu-Ritt characteristic sets”. In: *Proc. of MEGA 1990*. Springer. 1991, pp. 119–142.

[82] X.-S. Gao and S.-C. Chou. “Computations with Parametric Equations”. In: *Proc. of ISSAC 1991*. ACM, 1991, pp. 122–127.

[83] X.-S. Gao and S.-C. Chou. “On the dimension of an arbitrary ascending chain”. In: *Chinese Science Bulletin* 38 (1993), pp. 396–399.

[84] X.-S. Gao and S.-C. Chou. “Solving Parametric Algebraic Systems”. In: *Proc. of ISSAC 1992*. ACM, 1992, pp. 335–341.

[85] M. Gastineau and J. Laskar. “Parallel sparse multivariate polynomial division”. In: *Proc. of PASCO 2015*. ACM, 2015, pp. 25–33.

[86] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. 3rd ed. NY, USA: Cambridge University Press, 2013.

[87] J. von zur Gathen. "Hensel and Newton Methods in Valuation Rings". In: *Mathematics of Computation* 42.166 (1984), pp. 637–661.

[88] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for computer algebra*. Kluwer, 1992.

[89] J. Grabmeier, E. Kaltofen, and V. Weispfenning, eds. *Computer algebra handbook*. Springer-Verlag, 2003.

[90] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library (version 6.1.2)*. `http://gmplib.org`. 2020.

[91] T. Granlund et al. *The GNU Multiple Precision Arithmetic Library*. `https://gmplib.org/`. 2016.

[92] D. R. Grayson and M. E. Stillman. *Macaulay2, a software system for research in algebraic geometry*. Available at `http://www.math.uiuc.edu/Macaulay2/`.

[93] A. Haidar, J. Kurzak, and P. Luszczek. "An improved parallel singular value algorithm and its implementation for multicore hardware". In: *Proc. of SC 2013*. ACM, 2013.

[94] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. Version 2.5.2, `http://flintlib.org`. 2015.

[95] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. 5th. Morgan Kaufmann, 2011.

[96] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[97] J. van der Hoeven. "Faster relaxed multiplication". In: *Proc. of ISSAC 2014*. ACM, 2014, pp. 405–412.

[98] J. van der Hoeven. "Relax, but Don't be Too Lazy". In: *Journal of Symbolic Computation* 34.6 (2002), pp. 479–542.

[99] J. van der Hoeven and G. Lecerf. "Interfacing mathemagix with C++". In: *Proc. of ISSAC 2013*. 2013, pp. 363–370.

[100] J. van der Hoeven, G. Lecerf, and G. Quintin. "Modular SIMD arithmetic in Mathemagix". In: *ACM Transactions on Mathematical Software* 43.1 (2016), 5:1–5:37.

[101] J. Hu and M. B. Monagan. "A Fast Parallel Sparse Polynomial GCD Algorithm". In: *Proc. of ISSAC 2016*. 2016, pp. 271–278.

[102] J. Hu and M. B. Monagan. "A fast parallel sparse polynomial GCD algorithm". In: *Journal of Symbolic Computation* 105 (2021), pp. 28–63.

[103] M. Iwami. "Analytic factorization of the multivariate polynomial". In: *Proc. of CASC 2003*. 2003, pp. 213–225.

[104] R. D. Jenks and R. S. Sutor. *Axiom, the scientific computation system*. 1992.

[105] R. Jolly. "Categories as Type Classes in the Scala Algebra System". In: *Proc. of CASC 2013*. 2013, pp. 209–218.

[106] M. E. Kahoui. "An elementary approach to subresultants theory". In: *Journal of Symbolic Computation* 35.3 (2003), pp. 281–292.

[107] M. Kalkbrener. "Three Contributions to Elimination Theory". PhD thesis. Linz, Austria: Johannes Kepler Universität Linz.

[108] E. Kaltofen. "Sparse Hensel Lifting". In: *European Conference on Computer Algebra, EUROCAL 1985, Proceedings Volume 2: Research Contributions*. Ed. by B. F. Caviness. Vol. 204. Lecture Notes in Computer Science. Springer, 1985, pp. 4–17.

[109] J. Karczmarczuk. "Generating Power of Lazy Semantics". In: *Theoretical Computer Science* 187.1-2 (1997), pp. 203–219.

[110] M. Kazemi and M. Moreno Maza. "Detecting Singularities Using the PowerSeries Library". In: *Proc. of MC 2019*. Springer, 2019, pp. 145–155.

[111] J. de Kleine, M. B. Monagan, and A. D. Wittkopf. "Algorithms for the non-monic case of the sparse modular GCD algorithm". In: *Proc. of ISSAC 2005*. ACM, 2005, pp. 124–131.

[112] D. E. Knuth. *The Art of Programming, vol. 2, Semi-Numerical Algorithms*. Addison Wesley, Reading, MA, 1981.

[113] D. E. Knuth. *Literate programming*. Vol. 27. CSLI lecture notes series. Center for the Study of Language and Information, 1992.

[114] D. E. Knuth. "The analysis of algorithms". In: *The Actes du Congrés International des Mathématiciens* 3 (1970), pp. 269–274.

[115] A. J. Ko et al. "The state of the art in end-user software engineering". In: *ACM Computing Surveys (CSUR)* 43.3, article 21 (2011).

[116] J. Kollar. "Sharp Effective Nullstellensatz". In: *Journal of the American Mathematical Society* 1.4 (1988), pp. 963–975.

[117] L. Kronecker. "Die Zerlegung der ganzen Grössen eines natürlichen Rationalitäts-Bereichs in ihre irreductibeln Factoren." In: *Journal für die reine und angewandte Mathematik* 94 (1883).

[118] H. T. Kung and J. F. Traub. "All Algebraic Functions Can Be Computed Fast". In: *Journal of the ACM* 25.2 (1978), pp. 245–260.

[119] M. S. Lam, E. E. Rothberg, and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: *ASPLOS-IV Proceedings*. ACM Press, 1991, pp. 63–74.

[120] M. Lauer. "Computing by homomorphic images". In: *Computer Algebra*. Springer, 1983, pp. 139–168.

[121] D. Lazard. "A new method for solving algebraic systems of positive dimension". In: *Discrete Applied Mathematics* 33.1-3 (1991), pp. 147–160.

[122] G. Lecerf. "Computing the equidimensional decomposition of an algebraic closed set by means of lifting fibers". In: *Journal of Complexity* 19.4 (2003), pp. 564–596.

[123] D. H. Lehmer. "Euclid's algorithm for large numbers". In: *The American Mathematical Monthly* 45.4 (1938), pp. 227–233.

[124] C. E. Leiserson. "Cilk". In: *Encyclopedia of Parallel Computing*. 2011, pp. 273–288.

[125] F. Lemaire, M. Moreno Maza, and Y. Xie. "The RegularChains library in MAPLE". In: *ACM SIGSAM Bulletin* 39.3 (2005), pp. 96–97.

[126] T. C. Lethbridge and R. Lagamiere. *Object-oriented software engineering - practical software development using UML and Java*. MacGraw-Hill, 2001.

[127] X. Li, M. Moreno Maza, and É. Schost. "Fast arithmetic for triangular sets: From theory to practice". In: *Journal of Symbolic Computation* 44.7 (2009), pp. 891–907.

[128] Maplesoft. *Maple 2020*. Maplesoft, a division of Waterloo Maple Inc. Waterloo, Ontario, 2020.

[129] E. W. Mayr and A. R. Meyer. "The complexity of the word problems for commutative semigroups and polynomial ideals". In: *Advances in mathematics* 46.3 (1982), pp. 305–329.

[130] E. W. Mayr. "Membership in polynomial ideals over Q is exponential space complete". In: *Proc. of Symposium on Theoretical Aspects of Computer Science STACS 1989*. Springer. 1989, pp. 400–406.

[131] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[132] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (2017), e103.

[133] S. Meyers. *Effective modern C++*. O'Reilly, 2015.

[134] P. D. Michailidis and K. G. Margaritis. "Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP". In: *Journal of Computational and Applied Mathematics* 236.3 (2011), pp. 326–341.

[135] M. Monagan and R. Pearce. "Parallel sparse polynomial division using heaps". In: *Proc. of PASCO 2010*. ACM. 2010, pp. 105–111.

[136] M. Monagan and B. Tuncer. "Factoring multivariate polynomials with many factors and huge coefficients". In: *Proc. of CASC 2018*. Springer. 2018, pp. 319–334.

[137] M. B. Monagan. "Probabilistic algorithms for computing resultants". In: *Proc. of ISSAC 2005*. ACM, 2005, pp. 245–252.

[138] M. B. Monagan and R. Pearce. "Sparse polynomial division using a heap". In: *Journal of Symbolic Computation* 46.7 (2011), pp. 807–822.

[139] M. B. Monagan and B. Tuncer. "Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation". In: *Proc. of ICMS 2018*. 2018, pp. 359–368.

[140] M. B. Monagan and B. Tuncer. "The complexity of sparse Hensel lifting and sparse polynomial factorization". In: *Journal of Symbolic Computation* 99 (2020), pp. 189–230.

[141] M. B. Monagan and P. Vrbik. "Lazy and Forgetful Polynomial Arithmetic and Applications". In: *Proc. of CASC 2009*. Vol. 5743. Lecture Notes in Computer Science. Springer, 2009, pp. 226–239.

[142] M. Moreno Maza. *On Triangular Decompositions of Algebraic Varieties*. Tech. rep. TR 4/99. Presented at the MEGA-2000 Conference, Bath, England. Oxford, UK: NAG Ltd, 1999.

[143] M. Moreno Maza and R. Rioboo. "Polynomial Gcd Computations over Towers of Algebraic Extensions". In: *Proc. of International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Vol. 948. Lecture Notes in Computer Science. Springer, 1995, pp. 365–382.

[144] M. Moreno Maza and Y. Xie. "Component-level parallelization of triangular decompositions". In: *Proc. of PASCO 2007*. ACM. 2007, pp. 69–77.

[145] M. Moreno Maza and H. Yuan. "Balanced Dense Multivariate Multiplication: The General Case". In: *Proc. of CASC 2022*. (*Submitted*). 2022.

[146] J. Moses and D. Y. Y. Yun. "The EZ GCD algorithm". In: *ACM annual conference, Proceedings*. ACM, 1973, pp. 159–166.

[147] D. Mumford. *The red book of varieties and schemes*. 2nd. Vol. 1358. Lecture Notes in Mathematics. Springer, 1999.

[148] V. Neiger, J. Rosenkilde, and É. Schost. "Fast Computation of the Roots of Polynomials Over the Ring of Power Series". In: *Proc. of ISSAC 2017*. ACM, 2017, pp. 349–356.

[149] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly, 1996.

[150] E. Noether. "Idealtheorie in ringbereichen". In: *Mathematische Annalen* 83.1 (1921), pp. 24–66.

[151] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th. Morgan Kaufmann, 2014.

[152] J. R. Rice. *Scalable Scientific Software Libraries and Problem Solving Environments*. Tech. rep. 1257. Purdue University, 1996.

[153] J. F. Ritt. *Differential equations from the algebraic standpoint*. Vol. 14. American Mathematical Society, 1932.

[154] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1)*. `https://www.sagemath.org`. 2020.

[155] T. Sasaki and F. Kako. "Solving multivariate algebraic equation by Hensel construction". In: *Japan Journal of Industrial and Applied Mathematics* 16.2 (1999), pp. 257–285.

[156] T. Sasaki and D. Inaba. "Enhancing the Extended Hensel Construction by Using Gröbner Bases". In: *Proc. of CASC 2016*. Vol. 9890. Lecture Notes in Computer Science. Springer, 2016, pp. 457–472.

[157] B. D. Saunders, H. R. Lee, and S. K. Abdali. "A parallel implementation of the cylindrical algebraic decomposition algorithm". In: *Proc. of ISSAC 1989*. Vol. 89. 1989, pp. 298–307.

[158] M. Schaefer. "Complexity of Some Geometric and Topological Problems". In: *International Symposium on Graph Drawing, GD 2009, Revised Papers*. Vol. 5849. Lecture Notes in Computer Science. Springer, 2009, pp. 334–344.

[159] A. Schönhage. "Schnelle Berechnung von Kettenbruchentwicklungen". In: *Acta Informatica 1* (1971), pp. 139–144.

[160] M. L. Scott. *Programming Language Pragmatics (3. ed.)* Academic Press, 2009.

[161] J. Segal. "Some problems of professional end user developers". In: *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2007, pp. 111–118.

[162] V. Shoup. "A New Polynomial Factorization Algorithm and its Implementation". In: *Journal of Symbolic Computation* 20.4 (1995), pp. 363–397.

[163] V. Shoup et al. *NTL: A Library for doing Number Theory*. `www.shoup.net/ntl`. 2021.

[164] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[165] Standard C++ Foundation. *C++ FAQ*. Accessed on Dec. 4, 2021. URL: `https://isocpp.org/faq`.

[166] G. Strang. *Introduction to Linear Algebra*. 5th. Wellesley-Cambridge Press, 2016.

[167] B. Stroustrup. *The C++ programming language*. 4th. Addison-Wesley, 2013.

[168] The LinBox group. *LinBox*. v1.6.3. 2019. URL: `http://github.com/linbox-team/linbox`.

[169] K. Thull and C. K. Yap. "A Unifed Approach to HGCD Algorithms for polynomials and integers". In: (1990).

[170] B. Tuncer. "Solving multivariate Diophantine equations and their role in multivariate polynomial factorization". PhD thesis. Burnaby, Canada: Simon Fraser University, 2017.

[171] B. L. van der Waerden. *Modern Algebra: Volume 1*. Frederick Ungar, 1949.

[172] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[173] J. Verschelde. *The database of polynomial systems*. `http://homepages.math.uic.edu/~jan/`. Accessed: 2020-12-25. 2020.

[174] D. Wang. "An Elimination Method for Polynomial Systems". In: *Journal of Symbolic Computation* 16.2 (1993), pp. 83–114.

[175] D. Wang. "Decomposing Polynomial Systems into Simple Systems". In: *Journal of Symbolic Computation* 25.3 (1998), pp. 295–314.

[176] D. Wang. *Elimination Methods*. Texts & Monographs in Symbolic Computation. Springer, 2001.

[177] D. Wang. "On Wu's method for proving constructive geometric theorems". In: *Proc. of International Joint Conference on Artificial Intelligence, IJCAI-89*. Morgan Kaufann, 1989, pp. 419–424.

[178] P. S. Wang. "An improved multivariate polynomial factoring algorithm". In: *Mathematics of Computation* 32.144 (1978), pp. 1215–1231.

[179] P. S. Wang. "The EEZ-GCD algorithm". In: *ACM SIGSAM Bulletin* 14.2 (1980), pp. 50–60.

[180] I. Wolfram Research. *Mathematica, Version 12.3.1*. Champaign, IL, 2021. URL: `https://www.wolfram.com/mathematica`.

[181] W. Wu. "A zero structure theorem for polynomial equations solving". In: *MM Research Preprints* 1 (1987), pp. 2–12.

[182] W. Wu. "On a projection theorem of quasi-varieties in elimination theory". In: *MM Research Preprints* 4 (1989), pp. 40–48.

[183] W. Wu. "On problems involving inequalities". In: *MM Research Preprints* 7 (1992), pp. 1–13.

[184] W. Wu. "On the foundation of algebraic differential geometry". In: *MM Research Preprints* 3 (1989), pp. 1–26.

[185] W. Wu. "On zeros of algebra equations—an application of Ritt principle". In: *Kexeu Tongbao* 31.1 (1986), pp. 1–5.

[186] W. Wu. "Some remarks on characteristic-set formation". In: *MM Research Preprints* 3 (1989), pp. 27–29.

[187] Y. Xie. "Fast Algorithms, Modular Methods, Parallel Approaches, and Software Engineering for Solving Polynomial Systems Symbolically". PhD thesis. London, ON, Canada: University of Western Ontario, 2007.

[188] L. Yang and J. Zhang. *Searching dependency between algebraic equations: an algorithm applied to automated reasoning*. Tech. rep. International Centre for Theoretical Physics, 1990.

[189] R. Zippel. "Newton's iteration and the sparse Hensel algorithm". In: *Proc. of ISSAC 1981*. ACM, 1981, pp. 68–72.

[190] R. Zippel. "Probabilistic algorithms for sparse polynomials". In: *Proc. of Symbolic and algebraic computation, EUROSAM '79*. Vol. 72. Lecture Notes in Computer Science. Springer, 1979, pp. 216–226.

Curriculum Vitae

Name:	**Alexander Brandt**
Post-Secondary Education and Degrees	University of Western Ontario London, ON Ph.D., 2022
	University of Western Ontario London, ON M.Sc., 2018
	Memorial University of Newfoundland St. John's, NL B.Sc. (Hons.), 2017
Selected Honours and Awards	Alexander Graham Bell Canada Graduate Scholarship—Doctoral NSERC; 2019
	Ontario Graduate Scholarship Province of Ontario; 2017, 2019
	Best Presentation University of Western Ontario Research in Computer Science; 2018, 2021
	Canada Graduate Scholarship—Master's NSERC; 2017
	University Medal of Academic Excellence Memorial University of Newfoundland; 2017
	Science Co-op Student of the Year Award Memorial University of Newfoundland; 2016

Related Work Experience

Lecturer
University of Western Ontario
2019 –

Research Assistant
Ontario Research Center for Computer Algebra
University of Western Ontario
2017 – 2022

Teaching Assistant
University of Western Ontario
2017 – 2018

Software Developer
Whitecap Scientific Corporation
2016 – 2018

Research Assistant
Memorial University of Newfoundland
2016 – 2017

Publications

- M. Asadi, **A. Brandt**, David J. Jeffrey, M. Moreno Maza. "Subresultant chains using Bézout matrices". *Proceedings of Computer Algebra in Scientific Computing (CASC) 2022 - Lecture Notes in Computer Science*, (to appear). Springer, 2022.

- M. Asadi, **A. Brandt**, R.H.C. Moir, M. Moreno Maza, Y. Xie. "Parallelization of Triangular Decompositions: Techniques and Implementation". *Journal of Symbolic Computation*, (to appear), 2022.

- **A. Brandt**, M. Moreno Maza. "On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation". *Proceedings of Computer Algebra in Scientific Computing (CASC) 2021 - Lecture Notes in Computer Science*, 12865: 78–99. Springer, 2021.

- M. Asadi, **A. Brandt**, M. Moreno Maza. "Computational Schemes for Subresultant Chains". *Proceedings of Computer Algebra in Scientific Computing (CASC) 2021 - Lecture Notes in Computer Science*, 12865: 21–41. Springer, 2021.

- M. Asadi, **A. Brandt**, M. Kazemi, M. Moreno Maza, E. Postma. "Multivariate Power Series in Maple". *Proceedings of the Maple Conference (MC) 2020 - Communications in Computer Information Science*, 1414: 48-66. Springer, 2020.

- **A. Brandt**, M. Kazemi, M. Moreno Maza. "Power Series Arithmetic with the BPAS Library.". *Proceedings of Computer Algebra in Scientific Computing (CASC) 2020 - Lecture Notes in Computer Science*, 12291: 108–128. Springer, 2020.

- M. Asadi, **A. Brandt**, R.H.C. Moir, M. Moreno Maza, Y. Xie. "On the Parallelization of Triangular Decompositions". *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC) 2020*, 22–29. ACM, 2020.

- **A. Brandt**, R.H.C. Moir, M. Moreno Maza. "Employing C++ Templates in the Design of a Computer Algebra Library". *Proceedings of the International Congress on Mathematical Software (ICMS) 2020 - Lecture Notes in Computer Science*, 12097: 342–352. Springer, 2020.

- M. Asadi, **A. Brandt**, R.H.C. Moir, M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". *Mathematics*, 7(5), 441, 2019.

- **A. Brandt**, D. Mohajerani*, M. Moreno Maza, J. Paudel, L. Wang. "A Technique for Finding Optimal Program Launch Parameters Targeting Manycore Accelerators". *CASCON 2019*, 2019. *Poster*.

- M. Asadi, **A. Brandt**, R.H.C. Moir, M. Moreno Maza. "Sparse Polynomial Arithmetic with the BPAS Library.". *Proceedings of Computer Algebra in Scientific Computing (CASC) 2018 - Lecture Notes in Computer Science*, 11077: 32–50. Springer, 2018.

Software

- Basic Polynomial Algebra Subprograms (BPAS). M. Asadi, **A. Brandt**, C. Chen, S. Covanov, M. Kazaemi, F. Mansouri, D. Mohajerani, R.H.C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, Y. Xie. 2022. `http://bpaslib.org`.

- The MultivariatePowerSeries Package. M. Asadi, **A. Brandt**, M. Kazemi, M. Moreno Maza, E. Postma. 2021. Shipped with *Maple* (Maplesoft), since *Maple 2021*. `http://github.com/orcca-uwo/MultivariatePowerSeries`.

- Parallel and Distributed Barnes-Hut N-Body Simulation. **A. Brandt**. 2021. `http://github.com/alexgbrandt/Parallel-NBody`.
- KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs. **A. Brandt**, D. Mohajerani, M. Moreno Maza, L. Wang. 2019. `http://github.com/orcca-uwo/KLARAPTOR`.