Parallel Algorithms in Constructive Algebraic Geometry

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur"
in der Studienrichtung
Technische Mathematik
(Stzw. Mathematische Computerwissenschaften)

eingereicht von

Christian Mittermaier

Angefertigt am
Institut für symbolisches Rechnen (RISC)
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz
4040 Linz, Austria

Eingereicht bei
o.Univ.-Prof. Dr. Franz Winkler

February 24, 2000

Eidesstattliche Erklärung:

Ich, Christian Mittermaier, erkläre hier an Eides statt, daß ich die vorliegende Arbeit zur Gänze selbst verfaßt und keine anderen Quellen als die angegebenen verwendet habe.

Pasching,

Acknowledgements

I am very grateful to my advisors Prof. Dr. Franz Winkler and Dr. Wolfgang Schreiner for guiding me in my work. They offered me the possibility of writing a diploma thesis on an interesting topic in the framework of a project. Their patience, helpfulness and continuous support in difficult situations were invaluable.

I would also like to thank Dr. Josef Schicho cordially for providing valuable examples for my work and for our fruitful discussions.

Last but not least special thanks to Mag. Alfred Neubauer, my mathematics teacher at school, who strengthened me in my decision to study the beautiful science of mathematics.

Thank you all,

Christian

Abstract

The computer algebra software library CASA provides numerous algorithms for constructive algebraic geometry on the top of the Maple system. Usually, CASA functions operate on polynomials or rational functions and the algorithms work fine for inputs of moderate degree. However, when the degree increases, computing time explodes dramatically, making interactive usage of the system almost impossible.

In this work we investigate parallelization ideas for three important CASA algorithms, namely for plotting plane algebraic curves, plotting surface-to-surface intersections and for resolution of singularities. Besides the parallel algorithms we also give some improvements of the sequential programs which already reduce computing time significantly.

We have systematically tested and benchmarked our algorithms using the *Distributed Maple* environment on various machine configurations employing different numbers of processors. We have achieved significant speedups which reduce computing time to a moderate size.

Zusammenfassung

Die Computeralgebra-Softwarebibliothek CASA stellt zahlreiche Algorithmen für konstruktive algebraische Geometrie auf der Basis von Maple zur Verfügung. CASA-Funktionen operieren auf Polynomen oder rationalen Funktionen. Für moderate Inputgrößen liefern die Algorithmen in kurzer Rechenzeit das gewünschte Ergebnis, steigt jedoch der Grad der Eingabepolynome, so explodiert die benötigte Rechenzeit auf ein Vielfaches, wodurch ein interaktives Benutzen des Systems erschwert wird.

In dieser Arbeit untersuchen wir Parallelisierungsideen für drei wichtige CASA Algorithmen zum Plotten ebener algebraischer Kurven, zum Plotten von Raumkurven, die durch den Schnitt zweier Flächen definiert sind und zum Auflösen nicht-gewöhnlicher Singularitäten algebraischer Kurven. Zusätzlich beschreiben wir algorithmische Verbesserungen der sequentiellen Programme, die die Rechenzeiten schon signifikant reduzieren.

Die parallelen Algorithmen wurden sorgfältig und systematisch auf verschiedenen parallelen Umgebungen mit verschiedener Anzahl an Prozessoren getestet und bewertet. Wir konnten gute Speedup-Werte erzielen, wodurch sich die Rechenzeit auf ein tragbares Niveau reduzierte.

This work was partially supported by the FWF, project number P11160-TEC (HySaX) and SFB F013, project F1304.

Contents

Chapter 1

Introduction

Algebraic geometry is an active field of research with many applications in computer aided geometric design, robot kinematics, coding theory, etc. Apart from highly theoretical reasoning about the objects of classical algebraic geometry involving concepts from modern algebra (e.g. module theory, ring theory), sheaf theory and many more, there exists a whole variety of well known algorithms with which one can do constructive mathematics in this field. These algorithms are essentially based on computer algebra methods for manipulating polynomials and therefore can easily be implemented in any modern computer algebra system.

The program package CASA (Computer Algebra Software for constructive Algebraic geometry) [MW96] is designed for performing computations and reasoning about geometric objects in classical algebraic geometry, in particular affine and projective varieties over an algebraically closed field $\mathbb{K}$ of characteristic 0. Moreover, the field has to be computable in th sense of the underlying computer algebra system Maple. In the absence of any special indication of the field, CASA always assumes that the field of computation is the rational numbers $\mathbb{Q}$ or a finite algebraic extension thereof. CASA has been developed at RISC Linz over the last decade by a research group under the direction of Prof. Franz Winkler.

The objects CASA deals with are algebraic sets in four different representations:

1. *Implicit representation*, as zero set of a system of finitely many polynomial equations.
2. *Projected representation*, birationally projected to a hypersurface in an appropriate space.
3. *Parametric representation*, by a tuple of rational functions, if a parametrization of the algebraic set exists.
4. *Representation by places*, as a local parametrization by power series.

The operations available in CASA include

- ideal theoretic operations such as $+, \cdot, /$,
- creating algebraic sets in different representations,

- rational parametrization of curves and implicitization of parametrically given algebraic sets,
- plotting both, implicitly and explicitly given curves and surfaces,
- computation of syzygy bases,
- etc.

Our work started on top of the huge code base already available as a series of Maple program files, which form the CASA library. The system is designed for interactive usage, the user can create, manipulate and represent algebraic sets in the frame of the Maple front-end user interface, however, when examples become more involving, that is, the polynomials defining the algebraic variety are of high degree or the field $\mathbb{Q}$ is extended by algebraic numbers, computing time increases drastically at the expense of interactivity. One suggestion to overcome this disadvantage was to make use of parallel and distributed computing facilities. For this, the *Distributed Maple* package has been developed at our institute, allowing several independent Maple kernels on various machines connected by a network to participate in computation, hopefully leading to faster result.

In this paper, we analyze three important CASA functions for plotting plane algebraic curves in implicit representation, for plotting surface-to-surface intersections and for resolution of singularities of plane algebraic curves. We show how this functions significantly benefit from the distributed computing environment provided by *Distributed Maple*. In contrast to numerical mathematics, where parallelization often is a straightforward exercise, several pitfalls arise in parallel symbolic computing:

- High data dependencies require new algorithmic ideas in order to generate independent tasks which can be computed in parallel.
- Due to exact computations, the task size becomes virtually unpredictable making load balancing a difficult problem.
- High memory demands impinge upon limitations of the underlying system.

By distributed computing combined with sequential algorithmic improvements we could achieve good speedup values for the functions we analyzed. Apart from the development of parallel algorithms we spent a considerable amount of work in carefully testing and analyzing the dynamic behavior of our algorithms on various system configurations, such as a cluster of PCs and workstations and a distributed shared memory multiprocessor machine. Execution traces showing the work load of the individual processors participating in computation as well as tasks distribution are presented at the end of each chapter.

The structure of this thesis is as follows: In Chapter 2 we describe several software systems for parallel computer algebra. In Chapter 3 we discuss an algorithm for reliably plotting plane algebraic curves and invent parallelization strategies. In Chapter 4 we describe a generalization of the plotting algorithm from Chapter 3 for surface-to-surface intersections. Chapter 5 deals with resolution of singularities using the concept of neighboring points. In Chapter 6 we conclude summarizing the experiences gained from our work.

Chapter 2

Software Systems for Parallel Computer Algebra

In this Chapter we describe various software systems for parallel symbolic computation. Section 2.1 describes extensions of the Maple computer algebra system which allow employing parallel computing. In Section 2.2 we describe two parallel computer algebra libraries based on SAC-2. In Section 2.3 we give a brief overview about well known parallel computer algebra systems such as MuPAD, DSC, etc.

2.1 Maple Based Systems

Several attempts have been made to combine the computer algebra system Maple with parallel computation features. Basically, this is done pursuing two strategies:

1. Add parallel programming facilities directly to the Maple kernel.
2. Wrap Maple by an external system which takes care of the parallel execution of tasks.

The disadvantage of the first approach is that extensions to the Maple kernel require approval from Waterloo Maple (`http://www.maplesoft.com`) , the company developing and selling Maple. The modified kernels usually are not made accessible for the public and hence represent isolated solutions for special computer platforms. On the other hand, the second class of systems always suffers from inefficiency when dealing with fine grained parallelism. We briefly describe a selection of Maple based parallel computer algebra systems.

2.1.1 ||Maple||

||Maple|| (speak: parallel Maple) [Sie93] is a portable system for parallel symbolic computation built as an interface between the parallel programming language Strand and the sequential computer algebra system Maple. The||Maple|| system has two layers (see Figure 2.1). The top layer is the parallel programming language Strand which controls the parallel execution of an algorithm.

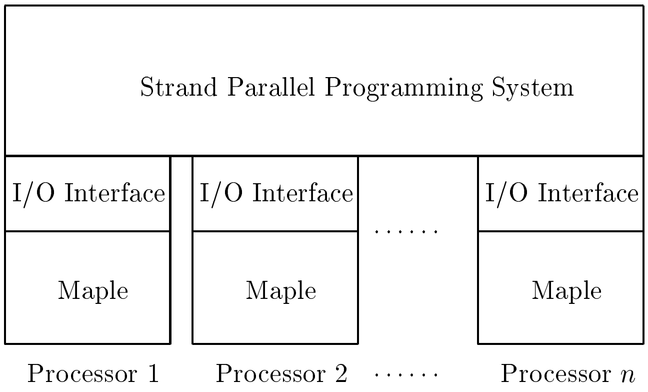

Figure 2.1: Structure of the ||Maple|| System

For performing sequential operations arbitrary Maple functions or sequences of Maple statements can be executed in the underlying Maple systems. Strand is a parallel programming language of the guarded Horn clause type. A basic construct is of the form

$$HL\ - G_1, \ldots, G_n | B_1, \ldots, B_m.$$

where the bodies B_j are executed as soon as the guards G_i are fulfilled. ||Maple|| supports various parallel programming techniques such as parallel task generation, pipelining and manager/worker processing. The system was developed at the beginning of the 1990s and is no longer in use (because the Strand language is not supported any more).

2.1.2 Parallel Maple through Linda

The *Sugarbush* system [B.W94, Cha90] combines the parallelism of C/Linda with the Maple V Release 3 kernel, allowing distributed and parallel Maple computations on platforms where both are available.

The Linda system introduces language constructs for parallelism into a conventional programming language without disrupting existing features of the language. Linda operations are sufficient to cover a wide variety of different parallel computing paradigms such as message passing and shared memory. The Linda approach has been implemented in several languages, among them C and Fortran.

Sugarbush gets its parallelism by building Linda operations into Maple. The system is batch-oriented due to the fact that computations which can benefit significantly from parallelism are lengthy, lasting several minutes and more. The system also provides generation of execution trace files and utilities for processing and viewing them.

2.1.3 Maple on Massively Parallel, Distributed Memory Machines

In [Ber97] porting the Maple kernel to the Intel Paragon family of massively parallel, distributed memory machines is described. This type of machine is scalable and contains up to several thousands of nodes. The processors within a node can cooperate using a message passing library. Figure 2.2 shows how the nodes are interconnected.

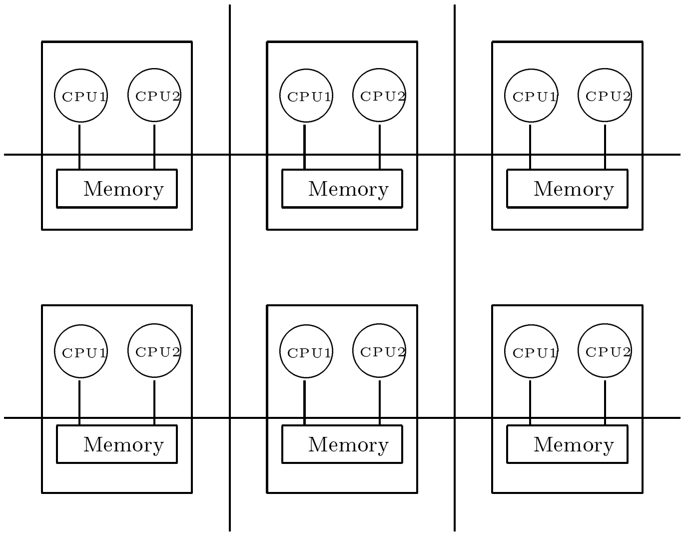

Figure 2.2: Paragon Overview

The goal of this approach was to integrate the distributed computing facilities tightly into Maple in order to achieve maximal inter-node communication efficiency. To this end, the basic primitives for using the machine's message passing library have been implemented within the Maple user level, such that code written in Maple's interpreted programming language can take full advantage of all nodes of the Paragon machine. The primitives are

- `mynode()` which returns the number of the executing node.
- `numnodes()` which returns the total number of nodes accessible to the current process.
- `csend(type, data, node)` which sends `data` to the node with number `node`, tagged with an arbitrary 32-bit integer `type`.
- `crecv(type, node)` which receives `data` from node `node` with type `type`.
- `lastnode()` returns the last node from which a message was received.

In this system, the SIMD (single instruction, multiple data) model of parallel computing is used.

Using the message passing primitives described above, a manager/worker based model of distributed computing using only Maple's user-level language was implemented. It basically provides the following commands:

2.1.4 Distributed Maple

Distributed Maple [Sch98] is an environment for writing parallel programs on the basis of the computer algebra system Maple. The system can be used in any network environment where Maple and Java are available. It supports creation os tasks, which are executed by Maple kernels running on the machines on the network and provides message passing facilities via a global heap.

Each machine connected to a *Distributed Maple* session consists of the following two components (see Figure 2.3:

1. *Scheduler.* The Java based scheduler manages the interaction of the Maple kernel running on the various nodes. The scheduler running on the machine to which the user frontendis attached, reads the system specific data from the `dist.systems` file and starts schedulers on the machines connected to the session.

2. *Maple Interface.* The scheudler and the Maple kernel exchange messages via pipes. A message may contain any Maple expression in Maple's library format.

After a session has been established the schedulers accept tasks from their attached Maple kernels and schedule them among the machines connected to the session.

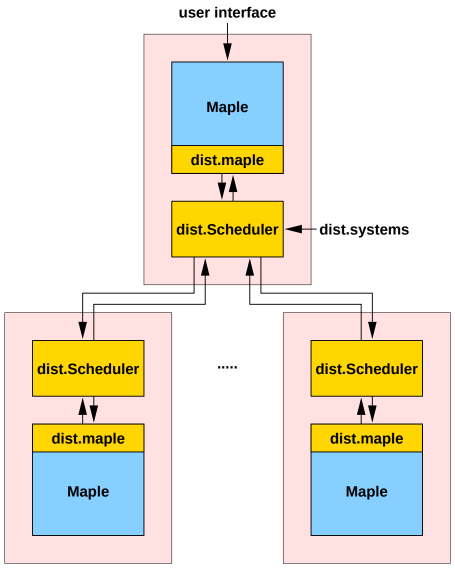

Figure 2.3: Distributed Maple Architecture

The user interacts with the *Distributed Maple* system via the text oriented Maple frontend or the graphical user interface. Initializing a session, creation of tasks and inter-task communication is done by Maple commands provided by the `dist.maple` package. The most important commands are:

`dist[all](`*`command`*`)` . The Maple statement *command* is executed on every Maple kernel connected to the session.

`dist[start](`f`, `a`, ... )` . This command creates a task evaluating the expression $f(a, \ldots)$ and returns a reference t to this task. The expression is evaluated in any machine connected to the session. It may again create new tasks and receive or return any Maple objects.

`dist[wait](`t`)` . This command blocks execution of the current task until the task referenced by t has been finished and returned its result.

This parallel programming model is essentially based on functional principles, which is sufficient for a variety of computer algebra algorithms. Additionally, non-deterministic task synchronization and shared data object are rpovided. The environment also includes mechanisms for handling very large task results and messages, which cannot be stored in the address space of the Java Virtual Machine but have to be maintained on disk. It also provides facilities for the online visualization of load distribution and for post-execution analysis of the dynamic behavior of a session.

2.2 SAC-2 Based Systems

The SAC-2 computer algebra library is a collection of carefully designed algorithms for various operations over integers, rationals, finite fields, algebraic numbers, and polynomials with such coefficients []. The algorithms in the library are coded in the programming language ALDES. In order to produce an executable program using the library, one usually translates the ALDES code into Fortran and uses the Fortran compiler.

We describe two libraries of parallel algorithms for computer algebra, which are based on the SAC-2 system.

2.2.1 PARSAC-2

The PARSAC-2 system [Küc90] is a parallel version of the SAC-2 computer algebra library. The parallel library is essentially based on threads of control which are suitable for symbolic computation on shared memory machines, so called S-threads.

The S-threads environment is designed to meet the challenges arising from special problems in parallel symbolic computation such as data dependency with unpredictable task lengths and memory demands. PARSAC-2 extends the SAC-2 library by a new system foundation (the S-threads) at and below SAC's list processing level.

2.2.2 PACLIB

The PACLIB system [HNS95] is based on the SACLIB library, a C version of the computer algebra library SAC-2. The PACLIB kernel meets several demands which raised in the development of the parallel system:

- Upward compatibility with SCALIB
- High level parallel programming model
- Light weight concurrency

- Non-deterministic and speculative parallelism
- Communication by heap references
- Parallel garbage collection

The programming interface provides generation of tasks, which receive SACLIB objects as input arguments and return a SACLIB object as result. The result may be retrieved later by using a handle to this task. SACLIB also supports inter-task communication by the means of *streams*, i.e. automatically synchronized lists.

For visualizing the dynamic aspects of a PACLIB application, the kernel generates on demand profiling information during a program run. For interactive analysis an X11 based visualization tool has been developed [HSF].

2.3 Other Systems

Various other parallel computer algebra systems for different environments have been developed. We briefly review some of them describing their programming model and the associated parallel or distributed environment.

2.3.1 PAC++

The PAC++ (Parallel Algebraic Computing) library [GR94] provides facilities for asynchronous remote procedure calls which are dynamically load balanced with runtime support. It is designed in C++ in order to use some of its useful features such as inheritance and overloading. Although the library may be used with any parallel environment, it also provides communication facilities which can be called from different systems and languages (sequential as well as parallel), such as UNIX-style libraries (see Figure 2.4. Additionally PAC++ objects are strongly typed. PAC++ uses a dynamic load-sharing algorithm

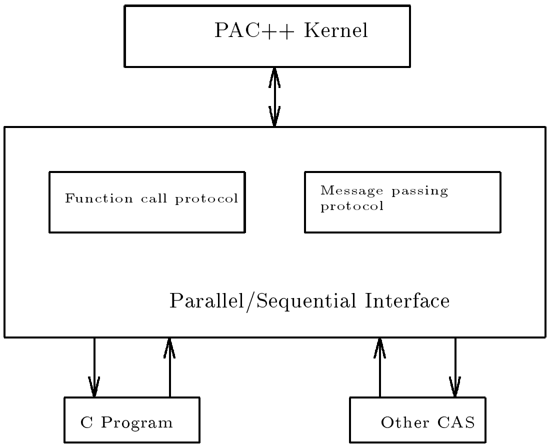

Figure 2.4: Parallel-Sequential Interface of PAC++

in order to overcome difficulties arising from the highly dynamic behavior of algebraic computations and to distribute work load evenly.

A parallel PAC++ program is a set of processes called *servers* with the ability to do algebraic computation. They are mapped on physical processors. Each of this servers has several entry points which are called *services*. A remote procedure call consists of choosing the processor on which the specific service is to be executed and the sending of the arguments. The result will be returned to the caller in packed form.

2.3.2 MuPAD

MuPAD [MRSW98] is a general purpose computer algebra system offering two programming concepts for parallel processing:

1. *Micro-parallelism* for shared memory architectures and
2. *macro parallelism* for distributed architecture.

On shared memory machines, basically two parallel constructs are available: parallel `for` loops and parallel blocks. Besides this constructs, *semaphores* and *locks* are available. Macro-parallelism in MuPAD is tightly coupled with micro-parallelism. The basic idea is to have clusters of shared memory machines which communicate over a network. However, there is no distributed version of MuPAD (in the sense of message passing), from the macro parallelism point of view MuPAD consists of a set of independent clusters, where each cluster in turn is formed of a number of running MuPAD kernels and micro-parallelism (see Figure 2.5).

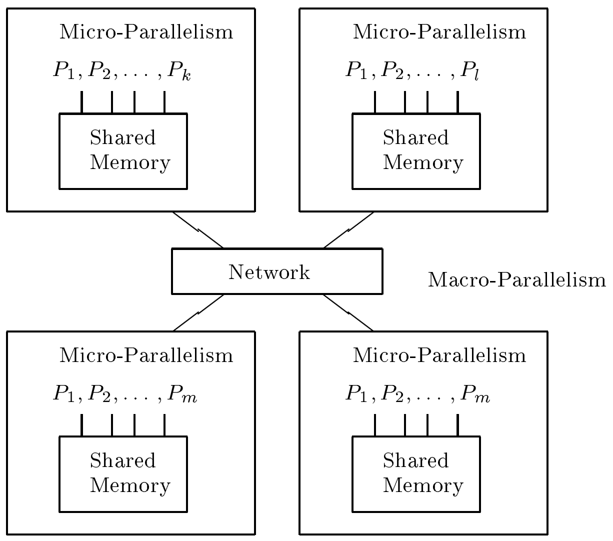

Figure 2.5: Micro-Parallelism and Macro-Parallelism.

MuPAD provides to the user a special concept to program this distributed environment. The interface consists of *net* variables, a kind of shared memory between the clusters, as well as queues and pipes between different cluster.

2.3.3 DSC

DSC [DKSV91] is a general purpose software system for distributed processing of large computer algebra computations over a network of UNIX workstations using the TCP/IP protocol. The system hides processor scheduling and the actual remote execution of a parallel subtask from the user.

The DSC system can send C or Lisp source code and input data files that constitute the parallel subtask to the remote processor, whose DSC server daemon process in turn will compile and run the received program on the input file. The parallel subtask is supposed to produce a corresponding output file, which will be returned to the sender for collection of results.

The C user library of the system provides functions for starting parallel tasks, waiting for results in a deterministic and non-deterministic way and the killing of tasks.

2.3.4 Calypso

CALYPSO [Ces97] is a computer algebra library for parallel symbolic computation. It is designed to be used as the computational engine by existing computer algebra systems or application programs which need high performance computer algebra algorithms. Its architecture can be divided into the following layers:

- *Machine Interface.* CALYPSO uses the MPI interface in order to provide primitives for parallel computing.

- *Basic Algorithms.* Several basic parallel multiple precision algorithms have been implemented in C++ (e.g. integer multiplication, FFT, etc.)

- *CALYPSO Interface.* This interface defines a frontend- backend model of computation. The frontend manages sequential tasks (such as initialization) while the remaining processors in the backend do the actual parallel computation.

- *CALYPSO C++ Classes.* At this level, functions are provided, which do not need explicit parallelism.

Chapter 3

Plotting Plane Algebraic Curves

In this chapter we describe the parallelization of a hybrid symbolic numerical algorithm for reliably plotting plane algebraic curves. We start from an existing sequential algorithm and give two algorithmic improvements which already reduce computing time by an order of magnitude. The improved algorithm then is the starting point for our parallelization approach. Profiling this algorithm reveals that the most time consuming operations are real root isolations, resultant computations and the computation of solutions of a triangular polynomial system in two variables. We describe a speculative parallel approach to polynomial real root isolation, review parallel resultant computation by homomorphic images and show how the computation of the solutions of the triangular system can significantly benefit from parallel computation.

The structure of this chapter is as follows: in Section 3.1 we describe the problem of reliably plotting plane algebraic curves. Section 3.2.1 describes the sequential plotting algorithm. In Section 3.2.2 we give an analysis of the most time consuming parts of the original algorithm. In Section 3.2.3 we show how computing time of the original method can be reduced significantly by two sequential algorithmic improvements. Section 1 deals with polynomial real root isolation describing the sequential Uspensky method, two parallelization approaches which have already been analyzed before and our speculative parallel algorithm. In Section 3.4 we give a brief summary about parallel resultant computation by homomorphic images. In Section 3.5 we describe the parallel version of the plotting algorithm. In Section 3.6 we present speedup diagrams and performance evaluations of the parallel algorithm and analyze its dynamic behavior.

3.1 Introduction

All modern computer algebra systems provide functions for plotting and visualizing the real affine part of the graph of curves which are given in implicit representation. These algorithms work well for curves defined by bivariate polynomials of reasonably low degree, however, as soon as the defining polynomial becomes somewhat complicated or the curve has self-intersecting branches, that is, singular points, these methods usually fail and return a picture from which one can only vaguely guess the shape of the curve.

Consider, for example, the *tacnode* curve, defined by the polynomial

$$f(x, y) := 2x^4 - 3x^2y + y^4 - 2y^3 + y^2,$$

a curve of degree four. In Figure 3.1 we see the graph of this curve produced by two different plotting algorithms. The picture on the left hand side was produced by Maple's `implicitplot` algorithm, which, as almost all plotting algorithms, uses numerical methods such as Newton's method in order to generate the visualization of the curve. The picture on the right hand side was

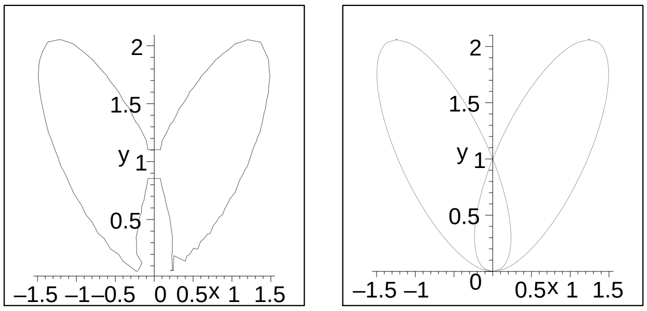

Figure 3.1: Comparing Maple's `implicitplot` and CASA's `pacPlot`

produced by CASA's `pacPlot` algorithm, which is a hybrid symbolic numerical algorithm, combining both, the exactness of symbolic methods and the speed and efficiency of numerical approximation.

The difference in the quality of the output of the two algorithms is impressive but can be explained easily. Purely numerical methods use linearisation in order to get an approximation. For this, the Jacobian of the defining polynomial has to be computed; but at those points on the curve, where two or more branches intersect each other, the Jacobian becomes singular as the partial derivatives with respect to both variables vanish. By increasing the number of points computed by the plotting algorithm (the default value for `implicitplot` is 50), the quality of the output can be increased to some reasonable level, however the origin is always missing and furthermore, computing time increases drastically (see Figure 3.2).

In algebraic geometry, one is particularly interested in those points of a curve, where two or more branches intersect each other. Those points are called singular points, their multiplicity on the curve plays an important role, for instance in the parametrization problem. Of course, all interesting properties of an algebraic curve and distinguished points on that curve can be determined by solely doing mathematical computations without any necessity to have a visual impression of the curve. However, if one wants to see a plot of the curve, the picture should at least match the computational results up to the level of resolution of the output device. Thus, a "good" plotting algorithm should fulfill the following informal requirements:

- **Draw topologically correct plots.** The course of the individual branches of the curve through singular points should be represented correctly, sin-

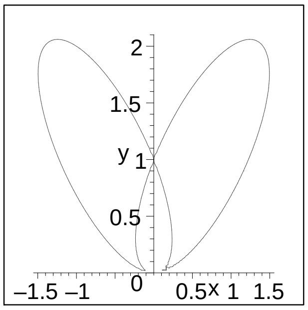

Figure 3.2: Maples's `implicitplot` Using a Finer Grid (10000 points).

gular points must not be missed, in particular even isolated singularities, originating from two complex branches intersecting in one real point, should be represented by exactly one point in the plot.

- **Use an efficient method for rendering the curve.** Computing time should be as low as possible without sacrificing correctness.

In the following sections we will analyze CASA's `pacPlot` algorithm which fulfills the requirements stated above and we show how computing time can be cut down by finding ways of employing parallel computation.

3.2 The `pacPlot` Algorithm

The `pacPlot` algorithm (plane algebraic curve plot) is part of the CASA system and uses a hybrid symbolic numerical approach in order to plot plane algebraic curves defined by bivariate polynomials over $\mathbb{Q}$. It was developed in 1995 by Quoc-Nam Tran as a by-product of `ssiPlot`, an algorithm for visualizing surface to surface intersection, from which the general idea was adopted to the special case of plane curves [Tra96].

3.2.1 The Sequential Algorithm

The hybrid symbolic numerical approach pursued in `pacPlot` combines both, the exactness of symbolic methods together with the speed and efficiency of numerical approximations. Roughly speaking, `pacPlot` uses a symbolic preprocessing step where the points determining the topological structure of the curve are determined by exact algorithms. For tracing the branches in between the previously computed points, a numerical method, namely Nam's Extended Newton's Method, is used. The general structure of the algorithm looks like the following:

1. Determine all critical points of the curve, that is, those points which determine the topological structure of the curve.

2. Find out starting points for the visualization.

3. Trace the simple branches of the curve by using numerical approximation.

By *critical points* we mean singular points (of arbitrary multiplicity) as well as extremal points in one coordinate direction (say in y-direction).

Definition 3.2.1. *Let $\mathcal{C}$ be an affine plane algebraic curve defined by the polynomial $f \in \mathbb{Q}[x,y]$.*

1. $(\xi, \eta) \in \overline{\mathbb{Q}}^2$ *is a* critical point *of $\mathcal{C}$ iff* $f(\xi,\eta) = \frac{\partial f}{\partial x}(\xi,\eta) = 0$.
2. $\mathbf{CP}(\mathcal{C}) := \{(\xi,\eta) \in \mathbb{Q}^2 \mid f(\xi,\eta) = \frac{\partial f}{\partial x}(\xi,\eta) = 0\}$ *denotes the* set of critical points *of the curve $\mathcal{C}$.*

In order to clarify how critical points determine the topological structure of the curve, we present the following picture (Figure 3.3) of a curve defined by the polynomial

$$f(x,y) := 8y^2 - 8x^2 - 8y^3 - 8x^3 + 2y^4 + x^5$$

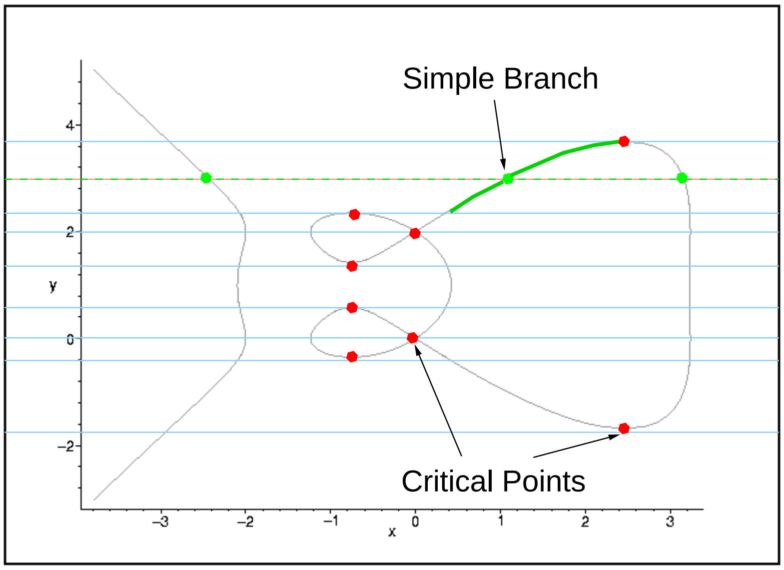

Figure 3.3: Critical Points and the Topological Ttructure of an Algebraic Curve

The critical points are marked by black dots (we do not distinguish between singular points and extremal points). The curve is partitioned into horizontal stripes, each of them bounded by two critical points. Within such a stripe, the curve has no self intersecting branches and the Jacobian is regular in every point. The picture of the curve is generated by tracing all simple branches for each stripe. For every stripe, starting points for the visualization of all simple branches have to be computed. This is done by intersecting the curve in the middle of every stripe with a horizontal line. The intersection points form the set of starting points for the rendering process. In Figure 3.3, the starting points are marked by grey dots. Theoretically, for determining the starting points within a stripe it is sufficient to intersect the curve with a horizontal line anywhere within the boundaries of the stripe, but in order to avoid numerical problems in the neighborhood of critical points, which are due to the fact that at least one partial derivative vanishes in a critical point, it is more convenient to do the intersection in the middle of the stripe. Next, all simple branches are traced in both directions by the Extended Newton's Method starting at the initial points of visualization until the border of the stripe is reached. The picture of the entire curve is obtained by combining the individual traces for all simple branches.

More precisely Algorithm 3.2.1(Critical Points) illustrates the pseudo code of Step 1 of `pacPlot`. We proceed by discussing the individual steps of this algorithm. If the curve $\mathcal{C}$ is not irreducible, we first factor $\mathcal{C}$ into its irreducible components and apply the algorithm to each component individually.

In Step 2 of Algorithm 3.2.1(Critical Points) CASA's `bsolve` function is used to obtain the set $\mathcal{T}$ of triangular systems. This algorithm is based on a research paper published by Michael Kalkbrener in 1990. We do not go into the mathematical details of this method here nevertheless mentioning that it is based on primitive polynomial remainder sequences, but refer to the original paper. However, we describe the specification of the algorithm:

Let $\mathbb{K}$ be a computational field and $F \subset \mathbb{K}[x,y]$ a finite subset. The `bsolve` algorithm computes a finite number of sets $E_1, \ldots, E_m \subset \mathbb{K}[x,y]$ such that the following conditions hold:

1. For every $i \in \{1, \ldots, m\}$:
   (a) E_i has one or two elements.
   (b) if E_i has two elements, then E_i consists of a nonzero polynomial $f \in \mathbb{K}[x]$ and a polynomial $g \in \mathbb{K}[x,y] - \mathbb{K}[x]$. Furthermore, every zero $a_1 \in \overline{\mathbb{K}}$ of f can be extended to a zero $(a_1, a_2) \in \overline{\mathbb{K}}^2$ of g.
2. $(a_1, a_2) \in \overline{\mathbb{K}}^2$ is a common zero of the polynomials in F if and only if there exists $i \in \{1, \ldots, m\}$ such that (a_1, a_2) is a common zero of the polynomials in E_i.

Step 3 is more tricky due to the following reasons:

1. We need the real solutions of the triangular systems only, as we want to plot the real part of the algebraic curve.
2. We do not need the exact solutions of the systems but it is sufficient to have isolating intervals for the x and y values of reasonably small length as the resolution of the output device is limited to pixel level.
   By this approach, we also avoid introducing and computing with real algebraic numbers and do not have to worry about their numerical evaluation.

In order to make this informal idea clear, we need some mathematical background:

Algorithm 3.2.1 (Critical Points) Compute critical points of a plane irreducible algebraic curve $\mathcal{C}$ defined by the bivariate polynomial $f(x, y) \in \mathbb{Q}[x, y]$.

Input: $f(x, y) \in \mathbb{Q}[x, y]$, irreducible.
Output: L, list of critical points with respect to y-direction, i.e. solutions of the system $f(x, y) = \frac{\partial f}{\partial x}(x, y) = 0$.

1: $f_x := \frac{\partial f}{\partial x}(x, y);\;\; L := \emptyset$.
2: Triangulize $f = f_x = 0$.

- Gives a set $\mathcal{T} := \{(p_i(y), q_i(x, y)) | i = 1 \ldots k\}$ of triangular systems.
- The union of all solutions of $p_i = q_i = 0,\ (p_i, q_i) \in \mathcal{T}$ is the solution set of the original system.

3: Solve each $p_i = q_i = 0,\ (p_i, q_i) \in \mathcal{T}$.

- Factor p_i giving a list of $p_{ij}(y)$.
- Let L_{ij} be the solution set of $p_{ij} = q_i = 0$.

4: $L := \bigcup_i \left(\bigcup_j L_{ij}\right)$.

Definition 3.2.2. *Let* $f, g \in \mathbb{Q}[x, y]$ *and* $\mathcal{S} := \{(x_i, y_i) \in \mathbb{R}^2 \mid i \in \{1, \ldots, m\}\}$ *the set of distinct real common zeros of the system* $f = g = 0$.
A set of isolating rectangles *for* $\mathcal{S}$ *is a set*

$$\mathcal{I} := \{]\xi_{i1}, \xi_{i2}[\times]\eta_{i1}, \eta_{i2}[\subset \mathbb{Q} \times \mathbb{Q} \mid i \in \{1, \ldots, m\}\}$$

such that the following conditions hold:

1. $\forall i \in \{1, \ldots, m\} : \ (x_i, y_i) \in]\xi_{i1}, \xi_{i2}[\times]\eta_{i1}, \eta_{i2}[$.
2. $\forall i, j \in \{1, \ldots, m\} : i \neq j \Rightarrow]\xi_{i1}, \xi_{i2}[\times]\eta_{i1}, \eta_{i2}[\cap]\xi_{j1}, \xi_{j2}[\times]\eta_{j1}, \eta_{j2}[= \emptyset$.

Step 3 of Algorithm 3.2.1(Critical Points) is expected to compute isolating rectangles for each triangular system $p_i(y) = q_i(x, y) = 0$. W.l.o.g. we assume that q_i is irreducible over $\mathbb{Q}$. Then the isolating rectangles for $p_i = q_i = 0$ can be computed by the following steps:

1. Find isolating intervals $Y_1, \ldots, Y_n \subset \mathbb{Q}$ for the real roots $\eta_1, \ldots, \eta_n$ of $p_i(y)$.

2. From p_i and q_i compute a polynomial $r(x)$ among whose real roots $\alpha_1, \ldots, \alpha_m$ we can also find for each real root η_j $(j \in \{1, \ldots, n\}$ of p_i all α_k's such that (α_k, η_j) is a zero of $p_i = q_i = 0$.

3. Find isolating intervals $X_1, \ldots, X_m \subset \mathbb{Q}$ for all real roots $\alpha_1, \ldots, \alpha_m$ of $r(x)$.

4. Check for which combinations $X_k \times Y_j$, $(k \in \{1, \ldots, m\},\ j \in \{1, \ldots, n\})$ we have an isolating rectangle for a common zero of $p_i = q_i = 0$.

Theorem 3.2.3 guarantees that we can always construct a polynomial $r(x)$ with the desired property. Although this theorem holds for multivariate polynomials, we state it for the special case of bivariate polynomials. For a proof of this theorem we refer the reader to [Win96].

Theorem 3.2.3. *Let $\mathbb{K}$ be an algebraically closed field and let*

$$f(x,y) = \sum_{i=0}^{m} f_i(x)y^i, \quad g(x,y) = \sum_{i=0}^{n} g_i(x)y^i$$

be elements of $\mathbb{K}[x,y]$ of positive degrees m and n in y. Let $r(x) := \mathrm{res}_y(f,g)$. If $(\xi,\eta) \in \mathbb{K}^2$ is a common zero of f and g, then $r(\xi) = 0$. Conversely, if $r(\xi) = 0$, then one of the following holds:

1. $f_m(\xi) = g_n(\xi) = 0$,
2. $\exists \eta \in \mathbb{K} : \ f(\xi,\eta) = g(\xi,\eta) = 0$.

Remark 3.2.4. In our special case of a triangular system $(p(y), q(x,y))$, case 1 is not possible, because the leading coefficient of p is constant.

In order to check, which combinations $X_k \times Y_j$ $(k \in \{1,\dots,m\},\ j \in \{1,\dots,n\})$ give an isolating rectangle, we proceed as follows:

1. Factor p_i over $\mathbb{Q}$ and apply the following steps to each irreducible factor. For simplicity, we assume that p_i is the only irreducible factor.
2. Compute the greatest square free divisor of $q_i(x,\eta)$, where η is an arbitrary real root of $p_i(y)$.
   We obtain the greatest square free divisor by dividing $q_i(x,\eta)$ by $\gcd(q_i(x,\eta), \frac{\partial}{\partial x} q_i(x,\eta))$ over $\mathbb{Q}(\eta)[x]$.
3. Take an arbitrary but fixed interval Y_j, $j \in \{1,\dots,n\}$ and consider p_i as the minimal polynomial for the real root η_j represented by Y_j.
4. Use a real root isolation algorithm for polynomials with real algebraic number coefficients for computing the isolating rectangles.
   Such an algorithm requires as input the real algebraic number η_j represented by the isolating interval Y_j, the minimal polynomial $p_j(y)$, the squarefree polynomial $q_i(x,y) \in (\mathbb{Q}[y]_{/\langle p_i \rangle})[x]$ representing $q_i(x,\eta_j)$, for which we want to isolate the real roots and a list of candidates $X_1,\dots,X_m$ for the isolating intervals of all real roots of $q_i(x,\eta_i)$.

Steps 2 and 3 are repeated until all intervals Y_j, $j \in \{1,\dots,n\}$ are processed.

In the sequel we describe an algorithm for real root isolation for polynomials with real algebraic number coefficients. Algorithm 3.2.2 also computes isolating intervals for the real roots of the resultant of p and q. In the critical point computation we do this before applying algebraic real root isolation for determining isolating rectangle for the critical points and pass the list of isolating intervals for the resultant to the algorithm as a parameter.

For determining the sign of q considered as an element of $\mathbb{Q}(\alpha)[x]$, where p is the minimal polynomial for α, Algorithm 3.2.3 is used. For correctness of this algorithm, we refer to [Loo83]. Again, this algorithm uses an isolating interval representation for the algebraic number α.

Algorithm 3.2.2 (Algebraic Real Root) Real root isolation for polynomials with real algebraic number coefficients

Input: $p(y) \in \mathbb{Q}[y]$, minimal polynomial for the real algebraic number α,
$I = [\eta_1, \eta_2[$, isolating interval for α,
$q(x, y) \in (\mathbb{Q}[y]_{/p})[x]$, square free for every root of p.
Output: L, list of isolating intervals for the real roots of q.

$r(x) := \mathrm{res}_y(p, q)$.
$L' := \{]\xi_{i1}, \xi_{i2}[\subset \mathbb{Q} \mid i \in \{1, \ldots, m\},]\xi_{i1}, \xi_{i2}[$is an isolating interval for a real root of $r\}$
$\{$The intervals in L' are candidates for isolating intervals.$\}$
for $i \in \{1, \ldots, m\}$ **do**
  $l := \mathrm{sgn}(q(\xi_{i1}, y), p(y),]\eta_1, \eta_2[)$.
  $r := \mathrm{sgn}(q(\xi_{i2}, y), p(y),]\eta_1, \eta_2[)$.
  if $l = 0 \ \vee \ r = 0 \ \vee \ l \cdot r < 0$ **then**
    $L := L \cup \{]\xi_{i1}, \xi_{i2}[\}$.
  end if
end for
return(L).

Algorithm 3.2.3 Computing the sign of a real algebraic number β

Input: $p(y)$, minimal polynomial for the real algebraic number α,
$I =]\eta_1, \eta_2[$, isolating interval for α,
$q(y) \in \mathbb{Q}[y]_{/\langle p \rangle}$ representing the real algebraic number $\beta \in \mathbb{Q}(\alpha)$.
Output: s, the sign of β.

Compute q^* and b such that $(1/b)q^* = q$, $q^* \in \mathbb{Z}[y]$ and $b \in \mathbb{Q} - \{0\}$. $I^*(=]\eta_1^*, \eta_2^*[) := I$.
Set $\overline{q}$ to the greatest squarefree divisor of q^*.
loop
  Set n to the number of roots of $\overline{q}$ in I^*.
  if $n = 0$ **then**
    $s := \mathrm{sgn}(b) \cdot \mathrm{sgn}(q^*(\eta_2^*))$.
    return(s).
  end if
  $w := (\eta_1^* + \eta_2^*)/2$.
  if $p(\eta_1^*)p(w) < 0$ **then**
    $\eta_2^* := w$.
  else
    $\eta_1^* := w$.
  end if
end loop

We summarize the various ideas described above and give a step-by-step list of instructions for computing the set $\mathbf{CP}(\mathcal{C})$ for a plane algebraic curve $\mathcal{C}$, as it is done by the original implementation of `pacPlot` and briefly sketched in Algorithm 3.2.1(Critical Points). $\mathcal{C}$ is defined by the bivariate polynomial $f \in \mathbb{Q}[x,y]$, L will eventually contain a numerical approximation the critical points of $\mathcal{C}$. This is sufficient for our purposes as two distinct points which are very close to each other coincide on pixel level.

1. $f_x := \frac{\partial f}{\partial x}$; $L := \emptyset$.
2. Use the `bsolve` algorithm to obtain a set T of triangular systems from f and f_x.
   T has the form $T := \{(p_i(y), q_i(x,y)) \mid i \in \{1,\ldots,k\}\}$ for some $k \in \mathbb{N}$. The union of the solutions of all triangular systems is the solution set of $f = f_x = 0$.
3. For each triangular system $(p(y), q(x,y)) \in T$ do the following:
   (a) Factor p over $\mathbb{Q}$ giving a set $P := \{p_j \mid \prod_j p_j = p\}$ of factors.
   (b) For each non-constant factor $p_j \in P$ do the following operations:
   - If p_j is linear, then compute the exact solution η of $p_j = 0$. Substitute η for y in q and compute isolating intervals $\{]\xi_{i1}, \xi_{i2}[\mid i \in \{1,\ldots,m\}\}$ for the real roots of $q(x,\eta)$ whose length is smaller than the resolution of the output device.
     $L := L \cup \{(\xi_i, \eta) \mid \xi_i = (\xi_{i1} + \xi_{i2})/2,\ i = 1\ldots m\}$.
   - If $\deg(p_j) > 1$ then
     i. Let $I_1 = \{]\eta_{j1}, \eta_{j2}[\mid j \in \{1,\ldots,m\})\}$ be a set of isolating intervals of the real roots of p_j whose length is smaller then the resolution of the output device.
     ii. Let α be a root of p_j and $q' := q(x,\alpha)$. Compute the quotient q'' of q' and $\gcd(q', \frac{\partial}{\partial x} q')$ over $\mathbb{Q}(\alpha)$.
     q'' is the greatest squarefree divisor of $q(x,\alpha)$ for every root α of p_j. Set $q^* := q''[\alpha \leftarrow y]$.
     iii. $r(x) := \mathrm{res}_y(p_j(y), q^*(x,y))$.
     iv. Let $I_2 := \{]\xi_{i1}, \xi_{i2}[\mid i \in \{1,\ldots n\}\}$ be a set of isolating intervals for the real roots of r whose length is smaller than the resolution of the output device.
     v. For every interval $]\eta_1, \eta_2[\in I_1$, use Algorithm 3.2.2(Algebraic Real Root) in order to determine isolating rectangles for the solutions of $p_j = q = 0$.
     Pass p_j, $]\eta_1, \eta_2[$, q^* and I_2 as parameters to Algorithm 3.2.2(Algebraic Real Root).
     As result, we get a set of isolating rectangles R of the form $R = \{X_j \times Y_j \subset \mathbb{Q} \times \mathbb{Q} \mid X_j \in I_2,\ Y_j \in I_1\}$, where $X_j =]\xi_{j1}, \xi_{j2}[$ and $Y_j =]\eta_{j1}, \eta_{j2}[$.
     vi. $L := L \cup \{(\xi_j, \eta_j) \mid \xi_j = (\xi_{j1} + \xi_{j2})/2,\ \eta_j = (\eta_{j1} + \eta_{j2})/2\}$
4. Return L.

3.2.2 Profiling `pacPlot`

In this section we give an analysis of `pacPlot`'s most time consuming parts based on sample executions of the algorithm for high-degree algebraic curves. The goal of this analysis is to isolate those parts of the algorithm which may benefit from parallel computation.

The sample runs were executed on an Intel PIII@450MHz Linux machine, the fastest machine class available at our institute. For the defining polynomials of the investigated curves and the plots see Appendix A.

(Sub-) Algorithm	Example 1	Example 2	Example 3	Example 4
real root isolation w.r.t. y	100.3	160.2	4.0	113.4
algebraic squarefree	[1]	7213.1	1203.8	[1]
resultant computation	542.0	455.8	81.8	4175.2
real root isolation w.r.t. x	37.0	27.3	2.8	57.5
real root isolation over $\mathbb{Q}(\alpha)$	6525.4	1807.7	252	20060.8
$\sum$ critical point computation	$>$ 7213.0	9666.6	1546.2	$>$24483.7
$\sum$ `pacPlot`	$>$7267.4	9689.0	1552.2	$>$ 24518.3

Table 3.1: Profiling the `pacPlot` algorithm (all timings in seconds)

Clearly, critical point computation is dominated by the gcd computation in the algebraic extension field of $\mathbb{Q}$. This step involves Maple's `RootOf` expressions for high-degree polynomials with large coefficients coming from the resultant computation. Whenever one computes in algebraic extension fields, computing times increase tremendously. Also resultant computation and real root isolation for polynomials with real algebraic number coefficients require large parts of the computing time. Ordinary real root isolation is time consuming only in some of the cases, but still it is worth to put some effort in developing a parallel algorithm for this sub-problem. The overall impression obtained from this execution profile perfectly meets ones expectations, namely that the symbolic part of `pacPlot` is the most time consuming one, while execution time for the numerical part is negligible.

3.2.3 Sequential Algorithmic Improvements

Replacing gcd-Computation over $\mathbb{Q}(\alpha)$

As we have seen in Section 3.2.2, critical point computation is mainly dominated by a gcd computation in an algebraic extension field of $\mathbb{Q}$. However, this gcd operation does not provide any information in the critical point computing step because it only ensures that the polynomial $q \in (\mathbb{Q}[y]_{/\langle p\rangle})[x]$, which is passed to the real root isolation algorithm for polynomials with algebraic number coefficients (Algorithm 3.2.2(Algebraic Real Root)) as a parameter is square free.

[1] Maple fails with an `object too large` error in this position, however, computation runs through when omitting the gcd

Algebraic real root isolation is used for checking which combinations of isolintervals form isolating rectangles for the critical points. By making this check more explicit instead of hiding it behind a real root isolation algorithm, we get completely rid of this time consuming operation and additionally generate lots of independent tasks for which employing parallel computation becomes easy. Algorithm 3.2.4 describes the idea in pseudo code.

Algorithm 3.2.4 A more explicit and efficient algorithm for computing isolating rectangles for the real solutions of the triangular system $p(y) = q(x, y) = 0$.

Input: $p(y) \in \mathbb{Q}[y]$, irreducible over $\mathbb{Q}$,
  $q(x, y) \in \mathbb{Q}[x, y]$, a bivariate polynomial,
  $\{Y_1, \ldots, Y_m\}$, a set of isolating intervals for the real roots of p,
  $\{X_1, \ldots, X_n\}$, list of isolating intervals for the real roots of $r(x) = \mathrm{res}_y(p, q)$ of the form $]\xi_{1i}, \xi_{2i}[$.
Output: L, set of isolating rectangles for the real solutions of $p = q = 0$.

$L := \emptyset$.
for i from 1 to n **do**
  $q_l(y) := q(\xi_{1i}, y) / \gcd(q(\xi_{1i}, y), \frac{\partial}{\partial y} q(\xi_{1i}, y))$.
  $q_r(y) := q(\xi_{2i}, y) / \gcd(q(\xi_{2i}, y), \frac{\partial}{\partial y} q(\xi_{2i}, y))$.
  {q_l, q_r are the greatest squarefree divisors of $q(\xi_{1i}, y)$ and $q(\xi_{2i}, y)$). }
  $q_l^* := q_l \mod p$.
  $q_r^* := q_r \mod p$.
  for j from 1 to m **do**
    $l := \mathrm{sgn}(q_l^*, p, Y_j)$.
    $r := \mathrm{sgn}(q_r^*, p, Y_j)$.
    if $l = 0 \lor r = 0 \lor l \cdot r < 0$ **then**
      $L := L \cup \{X_i \times Y_j\}$.
    end if
  end for
end for
return(L).

For this approach, it is no longer necessary to compute the squarefree divisor of the polynomial $q(x, \alpha)$ for every real root α of $p(y)$ in advance. Hence, the gcd computation over $\mathbb{Q}(\alpha)$ becomes obsolete. However, Algorithm 3.2.3 requires a squarefree representation of the real algebraic number. By first substituting the left and right endpoint of the interval for the x values, we reduce the gcd computation over $\mathbb{Q}(\alpha)$ to a gcd computation over $\mathbb{Q}$ which is much more faster (at most two seconds for each gcd in the examples from Section 3.2.2). From a mathematical point of view, the original approach is more elegant as we have to do several gcd computations here. But when efficiency is investigated carefully, our approach is superior to the former.

Speeding Up the Sign Computations

In step four of Algorithm 3.2.3, Moebius-transformations are used in order to transform the real roots of $\overline{q}$ in the interval I^* to the interval $]0, \infty[$ in order to apply Descartes' rule of signs. These transformations involve the left and right end-point η_1^* and η_2^*, which are huge rational numbers because all intervals obtained from real root isolation in the critical point computation algorithm are refined to at least the resolution of the output device. The huge rational numbers make the Moebius-transformations time consuming and furthermore, for some isolating intervals refinement is superfluous as they do not occur in any isolating rectangle. In order to make sign computation more efficient and simultaneously avoiding extra work, we pursue the following strategy, which has successfully been implemented in our version of the `pacPlot` algorithm:

1. Do not refine the isolating intervals in the real root isolation process.
2. Apply sign computation to the unrefined intervals only, in order to check which intervals actually form isolating rectangles.
3. Having determined the isolating rectangles, do refinement for all occuring intervals.

Execution Time Comparisons

Comparing execution times of the improved sequential algorithm (Table 3.2) with execution times of the original method (Table 3.1) is not possible in the level of the individual subalgorithms, however, comparing the total computing time for Step 1, we see that our improved version is significantly better. Moreover, by substitution the gcd computation over the algebraic extension field of $\mathbb{Q}$ with an equivalent but less complex approach makes the big examples (Example 1 and 4) executable in the sense that Maple does not fail with an `'object too large'` error.

	Example 1	Example 2	Example 3	Example 4
real root	83	140	3	64
resultant	386	254	56	3081
checking	8383	289	89	8279
refinement	127	52	8	49
$\sum$ Step 1	8989	737	158	11556

Table 3.2: Execution times of the improved sequential `pacPlot` algorithm

3.3 Real Root Isolation

As we have seen in Section 3.2.2 real root isolation may take some computational effort which cannot be neglected. Furthermore, real root isolation is an important subproblem in many computer algebra algorithms as it is more efficient than computing the exact values of the real zeros of a polynomial.

There exists a whole variety of well known and well analyzed algorithms (based on theorems by Fourier, Rolle, and Sturm). The method considered to be most efficient is essentially based on Descartes' rule of signs and a theorem which is due to Vincent. The algorithm is widely known as Uspensky's method, although the original method has been improved by Akritas. It is used in `pacPlot` as well as `ssiPlot`, which we will deal with in Section 4.2.

In Section 3.3.1 we describe the mathematical preliminaries for polynomial real root isolation. In Section 3.3.2 we present the sequential Uspensky method. Section 3.3.3 reviews a natural parallelization attempt for the Uspensky algorithm. Section 3.3.4 describes an approach which is based on parallel arithmetics. In Section 3.3.5 we present our speculative parallel algorithm for real root isolation.

3.3.1 Mathematical Foundations

We briefly review theorems and definitions related with Descartes' rule of signs and the effect of Moebius transformations on the roots of univariate polynomials (see [Akr89, CL83]).

Definition 3.3.1. *Let $f(x) \in \mathbb{R}[x]$ be a univariate polynomial of degree n, $f(x) = \sum_{i=0}^{n} a_i x^i$.*

1. $\mathrm{coeff}(f) := (a_{i_1}, \ldots, a_{i_m})$*, such that* $\{i_1, \ldots, i_m\} \subseteq \{0, \ldots, n\}$, $a_{i_j} < a_{i_k}$ *for* $j < k$, $a_k \neq 0$ *for* $k \in \{i_1, \ldots, i_m\}$ *and* $a_k = 0$ *for* $k \in \{0, \ldots, n\} - \{i_1, \ldots, i_m\}$.
   $\mathrm{coeff}(f)$ *denotes the sequence of nonzero coefficients of the polynomial f.*
2. *Let $(a_k)_{k\in\{0,\ldots,m\}}$ be a finite sequence of nonzero real numbers.*
   Then $\mathrm{var}(a_0, \ldots, a_m)$ *denotes the number of sign variations of* $(a_k)_{k\in\{0,\ldots,m\}}$.
3. *For the polynomial f, the sign variation* $\mathrm{var}(f)$ *is defined as* $\mathrm{var}(\mathrm{coeff}(f))$.

Theorem 3.3.2 (Descartes). *Let $f(x) \in \mathbb{R}[x]$ be a real univariate polynomial and λ denote the number of positive real roots of f counted with their multiplicity. Then* $\mathrm{var}(f) = \lambda + 2k$ *for some* $k \in \mathbb{N}_0$.

Descartes' rule states that the number of sign variations of a real polynomial exceeds the number of positive real roots (counted with their multiplicity) by an even number. This proposition is rather weak as the exact number of positive real roots is given in two special cases only.

Corollary 3.3.3. *Let $f(x) \in \mathbb{R}[x]$ be a nonzero real polynomial.*

1. *If* $\mathrm{var}(f) = 0$ *then f has no positive real root at all.*
2. *If* $\mathrm{var}(f) = 1$ *then f has exactly one positive real root.*

The converse is in general not true.

Example 3.3.4. Let $f(x) = (x-1)(x^2+1) = x^3 - x^2 + x - 1$. Consider the polynomial equation $f(x) = 0$ having 1 as the only positive real root. $\mathrm{var}(f) = 3$.

However, the following lemma holds:

Lemma 3.3.5. *Let $f(x) \in \mathbb{R}[x]$ be a nonzero real polynomial having no positive real root. Then* $\mathrm{var}(f) = 0$.

Theorem 3.3.6 (Cauchy). *Let $f(x) \in \mathbb{C}[x]$, $f(x) = a_0 + a_1 x + \cdots + a_n x^n$ be a polynomial of degree n. Then every root α of f satisfies*

$$|\alpha| < 1 + \frac{\max(|a_0|, \ldots, |a_{n-1}|)}{|a_n|},$$

where $|\alpha| = \sqrt{\Re(\alpha)^2 + \Im(\alpha)^2}$.

Next we study the effect of Moebius transformations on the roots of a polynomial. In the sequel we concentrate on three special types of Moebius transformations which play an essential role in the real root isolation process.

Definition 3.3.7. *Let $M(x) := \frac{ax+b}{cx+d}$, where $a, b, c, d \in \mathbb{C}$ and $ad-bc \neq 0$. Then the linear, birational substitution $x := M(x)$ is called a* Moebius transformation.

Definition 3.3.8.

1. *For $a \in \mathbb{R}$, let $T(x,a)$ be the Moebius transformation $x := x + a$. $T(x,a)$ is called* translation.
2. *For $a \in \mathbb{R}$, $a \neq 0$, let $S(x,a)$ be the Moebius transformation $x := ax$. $S(x,a)$ is called* stretching.
3. *Let $I(x)$ be the Moebius transformation $x := 1/x$. $I(x)$ is called* inversion.

We are now ready to study the effect of Moebius transformations on the roots of a univariate polynomial. Let

$$f(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = (x - \alpha_1)\cdots(x - \alpha_n)$$

.

1. Let $b \in \mathbb{R}$. When we apply the translation $T(x,b)$ to f, we obtain

$$\begin{aligned} f(I(x,b)) &= (x + b - \alpha_1)(x + b - \alpha_2)\cdots(x + b - \alpha_n) \\ &= (x - (\alpha_1 - b))(x - (\alpha_2 - b))\cdots(x - (\alpha_n - b)) \end{aligned}$$

   The real parts of the zeros $\alpha_1, \dots, \alpha_n$ decrease or increase by b, depending on b being positive or negative.

2. Let $b \in \mathbb{R}$, $b \neq 0$. Applying the stretching $S(x,b)$ to f, we obtain

$$\begin{aligned} f(S(x,b)) &= b^nx^n + a_{n-1}b^{n-1}x^{n-1} + \cdots + a_1bx + a_0 \\ &= (bx - \alpha_1)\cdots(bx - \alpha_n) \\ &= b^n(x - \frac{\alpha_1}{b})\cdots(x - \frac{\alpha_n}{b}) \end{aligned}$$

   Dividing by b^n we get

$$\begin{aligned} \left(\frac{1}{b^n}\right) f(S(x,b)) &= x^n + \left(\frac{1}{b}\right) a_{n-1}x^{n-1} + \cdots + \left(\frac{1}{b^{n-1}}\right) a_1x + \left(\frac{1}{b^n}\right) a_0 \\ &= \left(x - \frac{\alpha_1}{b}\right)\cdots\left(x - \frac{\alpha_n}{b}\right) \end{aligned}$$

   By a stretching substitution the roots $\alpha_1, \dots, \alpha_n$ are divided by the factor b.

3. When we apply the inversion transformation $I(x)$ to f, we get

$$\begin{aligned} f(I(x)) &= \frac{1}{x^n} + \frac{a_{n-1}}{x^{n-1}} + \cdots + \frac{a_1}{x} + a_0 \\ &= \left(\frac{1}{x^n}\right)(1 - x\alpha_1)\cdots(1 - x\alpha_n) \\ &= \left(\frac{1}{x^n}\right)\left(x - \frac{1}{\alpha_1}\right)\cdots\left(x - \frac{1}{\alpha_n}\right) \end{aligned}$$

Multiplying by x^n we obtain

$$\begin{aligned} x^n f(I(x)) &= a_0 x^n + \cdots + a_{n-1}x + 1 \\ &= \left(x - \frac{1}{\alpha_1}\right) \cdots \left(x - \frac{1}{\alpha_n}\right) \end{aligned}$$

By the inversion substitution, the roots $\alpha_1, \ldots, \alpha_n$ are inverted.

3.3.2 The Sequential Algorithm

The Uspensky method computes isolating intervals for all positive real roots of a squarefree polynomial $f(x)$. In order to find isolating intervals for the negative roots, the algorithm is applied to $f(-x)$. The idea of the method is as follows:

1. For the input polynomial f we first apply a stretching transformation to f in order to obtain a polynomial g whose real roots in $]0,1[$ are in one-to-one correspondence with the real roots of f in $]0,\infty[$.
   Then we proceed recursively as follows:

2. Whenever we come to this point we are interested in the real roots of g in $]0,1[$.
   By a combination of the inversion and translation transformation, we compute a polynomial $g^*(x)$ from $g(x)$ such that the real roots of g^* in $]0,\infty[$ correspond to the real roots of g in $]0,1[$.

3. We determine the sign variation $s := \mathrm{var}(g^*)$. If $s = 0$ or $s = 1$ then by Corollary 3.3.3 we know that g^* has no real root or exactly one real root in $]0,\infty[$ and hence g has no real root or exactly one real root in $]0,1[$.
   If $s = 0$ we return the empty set, if $s = 1$ we return the set $\{]0,1[\}$ and go back one step in the recursion.

4. If $s > 1$ then Descartes' rule of signs does not give any information about the exact number of positive real roots. In this case, we proceed as follows:

   (a) First, we check if $1/2$ is a real root of g. If so, we add the interval $[\frac{1}{2},\frac{1}{2}]$ to the list of so far computed isolating intervals and divide g by $(2x-1)$. The quotient no longer contains $1/2$ as a root.

   (b) By an application of a stretching transformation we compute a polynomial $g_l(x)$ from $g(x)$ such that the real roots of g_l in $]0,1[$ correspond to the real roots of g in $]0,\frac{1}{2}[$. We recursively go to step 2 with g_l substituted for g.

   (c) By an application of a transformation operation we compute a polynomial g_r from g_l such that the real roots of g_r in $]0,1[$ correspond to the real roots of g in $]\frac{1}{2},1[$. We recursively go to step 2 with g_r in the place of g.

   (d) We undo the translations and stretchings in the results of (b) and (c) and add them to the set of isolating intervals computed do far. Then we return this set.

Algorithms 3.3.1(Uspensky Main) and 3.3.2(Uspensky Sub) give a more detailed pseudo code description of the method. Vincent's Theorem (3.3.9) guarantees that the termination case (namely $s = 0$ or $s = 1$) is eventually reached.

Algorithm 3.3.1 Uspensky Main Uspensky's method for real root isolation. The positive real roots of the input polynomial are transformed to $]0,1[$ and Algorithm 3.3.2 is invoked.

Input: $f(x) \in \mathbb{Q}[x]$square free polynomial of degree n.
Output: L, a list of isolating intervals for the positive real roots of $f(x)$.

Let 2^k be a root bound for $A(x)$
if $k \leq 0$ **then**
  Set L to the result of Algorithm 3.3.2 **(Uspensky Sub)** applied to $f(S(x, 2^k))$.
else
  Set L to the result of Algorithm 3.3.2 **(Uspensky Sub)** applied to $2^{-nk} f(S(x, 2^k))$.
end if
$L := \{]2^k a, 2^k b[\mid]a, b[\in L\}$

Theorem 3.3.9 (Vincent, 1836). *Let $f(x) \in \mathbb{R}[x]$ be a squarefree polynomial and $M_i(x)$ be a Moebius transformation of the form $x := a_i + \frac{1}{x}$ $(i \in \mathbb{N})$, such that $a_1 \geq 0$ and $\forall i > 1 : a_i > 0$. Then for some $k \in \mathbb{N}$: $\operatorname{var}(f(M_k(\ldots(M_2(M_1(x)))))) \leq 1$.*

Proof. See [Vin36] □

Remark 3.3.10.

1. For efficiency reasons, we do not compute the exact number of sign variations in step 2 of Algorithm 3.3.2. It suffices to determine if g^* has 0, 1 or more than 1 sign variations.

2. Each $M_i(x)$ can be replaced by the equivalent pair of transformations $I(T(x, a_i))$.

3. In our algorithm, the coefficients a_i of the Moebius transformations $M_i(x)$ are always set to 1.

Remark 3.3.11. One carefully has to think about the way of implementing Moebius transformations, in particular the translation, in computer algebra systems like Maple, because most of the computing time is needed for doing the transformations.
In Maple, the best way of implementing the transformation $T(x, 1)$ seems to be the following (see Figure 3.5):

- Convert the polynomial g into Horner's scheme.

- Substitute $x + 1$ for x in the Horner scheme representation of g.

- Expand the Horner scheme representation of g again.

In the original CASA implementation of Uspensky's method $T(x, 1)$ was computed by directly substituting $x + 1$ for x in g and expanding. This turned out to be the most inefficient way of computing such a transformation.
Scaling and inversion transformations are efficiently computed by direct substitution and expanding.

Algorithm 3.3.2 Uspensky Sub Uspensky's method for computing isolating intervals for the real roots of the input polynomial in $]0,1[$. This Algorithm is used by Algorithm 3.3.1(Uspensky Main)

Input: $g(x) \in Q[x]$, a squarefree polynomial
Output: L, a set of isolating intervals for the zeros of $g(x)$ in $]0,1[$

$g^*(x) := (x+1)^n g(T(I(x),1))$.
  $\{$The zeros of $g(x)$ in $]0,1[$ correspond to the zeros of $g^*(x)$ in $]0,\infty[$.$\}$
$s := \mathrm{var}(g^*)$.
if $s = 0$ **then**
  return$(\emptyset)$.
else if $s = 1$ **then**
  return$(\{]0,1[\}$.
end if
if $g(1/2) = 0$ **then**
  $L := L \cup \{[\frac{1}{2},\frac{1}{2}]\}$. $\{$We found an exact root, i.e. a single point interval.$\}$
  $g(x) := g(x)/(2x-1)$.
end if
$\{$Left half of the interval$\}$
$g_l(x) := 2^n g(S(x,\frac{1}{2}))$.
Apply Algorithm 3.3.2(Uspensky Sub) recursively to $g_l(x)$ with result L_l.
$L := L \cup \{]a_i/2, b_i/2[\mid]a_i,b_i[\in L_l\}$.
$\{$Right half of the interval$\}$
$g_r(x) := g_l(T(x,1))$.
Apply Algorithm 3.3.2(Uspensky Sub) recursively to $g_r(x)$ with result L_r.
$L := L \cup \{](a_i+1)/2, (b_i+1)/2[\mid]a_i,b_i[\in L_r\}$.

The way isolating intervals for the real roots are computed by the divide and conquer approach pursued in Uspensky's method (Algorithm 3.3.2(Uspensky Sub)) can be associated with a binary search tree. Each node in the tree is associated with a subinterval of $]0,1[$, and the tree has the following properties:

- The root node is associated with $]0,1[$.
- A node at depth n [1] is associated to an interval of the form $]\frac{i}{n+1}, \frac{i+1}{n+1}[$, $i \in \{0,\ldots,n\}$ and carries the sign variation of the test polynomial g^* as value.
- Every interval associated to a leaf of the tree either contains no zero or exactly one zero of the input polynomial and hence has value 0 or 1.
- The Moebius transformations which are performed by Algorithm 3.3.2 are associated with the edges of the tree. A vertex leading to a left subtree corresponds to the scaling $S(x, \frac{1}{2})$. A vertex leading to a right subtree corresponds to the compound transformation $T(S(x, \frac{1}{2}), 1)$.

Example 3.3.12. Consider the polynomial

$$f(x) = (x-1)(x-3)(x-5) = x^3 - 6x^2 + 11x - 6.$$

An upper bound for the absolute values of the roots of f is $2^3 = 8$. So Algorithm 3.3.2 is applied to

$$g(x) := x^3 - \frac{9}{8}x^2 + \frac{23}{63}x - \frac{15}{64}.$$

In Figure 3.4 we can see the binary tree generated by the isolation process.

3.3.3 A Naive Approach for a Parallel Algorithm

An obvious approach for a parallel version of the real root isolation algorithm is to execute Algorithm 3.3.2(Uspensky Sub) for g_l and g_r in parallel. This approach has already been studied by Collins et al. [CJK90] on a shared memory multiprocessor machine using PARSAC-2. As Collins and his colleagues observed, the efficiency of this approach depends exclusively on the search tree associated with the real root isolation process. In the ideal case, this binary tree is balanced having 2^n nodes on level n. Collins observed that for randomly generated polynomials the tree is not balanced at all and at most a speedup of 2 is achievable.

Nevertheless we have implemented this approach using the *Distributed Maple* system, which requires only little changes in the original algorithm. In each step of the recursion, the local processor computes on one half of the interval $]0,1[$ while the other half is investigated remotely.

The motivation for testing this approach was the hope, that the polynomials occurring in the `pacPlot` context lead to almost balanced trees for which the algorithm would perform better. However, experiments did not confirm this assumption, hardly any speedup was achieved.

[1] We start counting the depth in the tree from 0. Hence, the root node has depth 0.

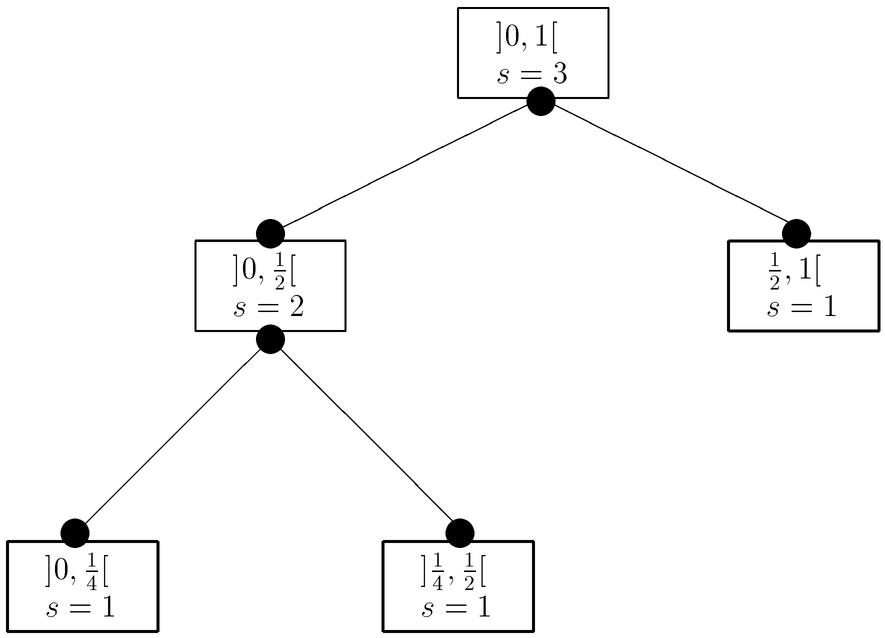

Figure 3.4: The Binary Tree Associated with the Polynomial in Example 3.3.12

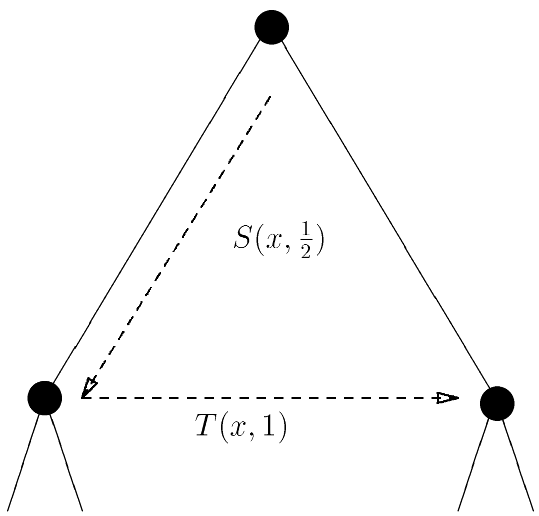

Figure 3.5: Computation of the Transformations.

3.3.4 Parallel Arithmetic

Investigating the sequential algorithm for its most time consuming operations, we found out that the computation of g^* and g_r dominates computing time. In [DK] Krandick and Decker realized that the translation operation $T(x,1)$ used for g^* and g_r are of a simple structure which allows each coefficient of the two polynomials to be computed independently from the others. Using this information one can split the translation operation into parallel blocks where each block deals with a consecutive sequence of coefficients. In this approach, the sequential execution pattern of the algorithm is maintained, however, the computation of transformations is done in parallel.

Before we give a detailed description of this algorithm, we take a closer look at the action of $T(x,1)$ on a polynomial f. Let $f(x) = \sum_{i=0}^{n} a_i x^i$ be a real polynomial of degree n. Then

$$\begin{aligned} f(T(x,1)) &= \sum_{i=0}^{n} a_i (x+1)^i \\ &= \sum_{i=0}^{n} a_i \sum_{j=0}^{i} \binom{i}{j} x^j \\ &= \sum_{i=0}^{n} x^i \left(\sum_{j=0}^{n-i} a_i \binom{i}{j} \right). \end{aligned}$$

Example 3.3.13. Consider the polynomial f of degree 4:

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

Applying $T(x,1)$ to f we obtain

$$\begin{aligned} f(T(x,1)) = & \quad (a_0 + a_1 + a_2 + a_3 + a_4) + (a_1 + 2a_2 + 3a_3 + 4a_4)x \\ & \quad +((a_2 + 3a_3 + 6a_4)x^2 + (a_3 + 4a_4)x^3 + a_4 x^4 \\ = & \quad \begin{pmatrix} x^0 & x^1 & x^2 & x^3 & x^4 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 & 6 \\ 0 & 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \end{aligned}$$

In Example 3.3.13, the matrix contains the Pascal triangle of height 5. In general, for a polynomial f of degree n, this matrix contains the Pascal triangle of height $n+1$. As the transformation $T(x,1)$ does not change the degree of a polynomial, this matrix can be precomputed in advance.

Definition 3.3.14. *Let $n \in \mathbb{N}_0$. $P_n := (p_{ij})_{1 \le i,j \le n+1}$ denotes the* Pascal matrix *of order n and is defined as follows:*

$$p_{ij} = \begin{cases} 1 & \text{if } i = j \text{ or } i = 1 \\ 0 & \text{if } i < j \\ p_{i,j-1} + p_{i-1,j-1} & \text{else} \end{cases}$$

Lemma 3.3.15. *Let $f(x) \in \mathbb{R}[x]$, $f(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n, P_n the Pascal matrix of order n.*
Then the coefficient b_i of x^i in $f(T(x,1))$ is given by

$$b_i = \sum_{j=i}^{n} p_{i+1,j+1} a_j$$

A stretching operation of the form $S(x,b)$, $b \in \mathbb{R} - \{0\}$ affects the coefficients of the polynomial f as follows:

$$f(S(x,b)) = \sum_{i=0}^{n} a_i b^i x^i$$

Multiplying the equation by $\left(\frac{1}{b^n}\right)$ yields

$$\frac{1}{b^n} f(S(x,b)) = \sum_{i=0}^{n} (a_i b^{n-i}) x^i$$

We can now apply these techniques to the transformations occurring in the Uspensky algorithm.

- g^* is obtained from g by the following two steps:

  $$g_1^* := x^n g(I(x)); \; g^* := g_1^*(T(x,1))$$

  By computing g_1^*,the order of the coefficients is switched, that is, a_i becomes a_{n-i}. Hence, the coefficient a_i^* of x^i in g^* is given by $\sum_{j=i}^{n} p_{i+1,j+1} a_{n-j}$.
- The coefficient a_i^l of x^i in g_l is given by $2^{n-i} a_i$.
- The coefficient a_i^r of x^i in g_r is given by $\sum_{j=i}^{n} pi + 1, j + 1a + l^l$.

Additionally, we observe the following: Computing g^* and its sign variation can be done in a single step by computing the coefficients of g^* in ascending order and comparing the signs of two coefficients following each other. Computation can be stopped as soon as two sign variations have been counted. In order to do this computation in parallel, we need two lemmata.

Lemma 3.3.16. *For $n \in \mathbb{N}$ let $(a_0, \ldots, a_n)$ be a sequence of nonzero real numbers, $k \in \mathbb{N}_0$ such that $k < n-1$ and $\{i_1, \ldots, {}_1{}_k\} \subset \{2, \ldots, n-1\}$, satisfying $i_1 < i_2 < \ldots < i_k$. Then $\operatorname{var}(a_1, \ldots, a_n) = \operatorname{var}(a_1, \ldots, a_{i_1}) + \operatorname{var}(a_{i_1}, \ldots, a_{i_2}) + \cdots + \operatorname{var}(a_{i_k}, \ldots, a_n)$.*

Lemma 3.3.17. *Let $f(x) \in \mathbb{R}[x]$ be a nonzero real polynomial of degree n.*
Then for all i, $0 \leq i \leq n$, the coefficient a_i^* of x^i in $f(T(x,1))$ is nonzero.

Furthermore, we can guarantee that the constant coefficient of the input polynomial g is nonzero. If it is zero, we divide g by x obtaining a single point isolating interval $[0,0]$. The quotient no longer has a zero coefficient as g is considered to be squarefree. It is necessary to guarantee this as by the inversion $I(x)$ in the computation of g^*, the constant coefficient of g becomes leading coefficient of g^*.

The idea of a real root isolation algorithm based on Uspensky's method and parallel polynomial arithmetics is as follows. Let $g(x) \in \mathbb{Q}[x]$, $g(x) = \sum_{i=0}^{n} a_i x^i$ be the squarefree input polynomial such that $a_0 \neq 0$ and p denote the number of available processors.

1. g^* is of the same degree as g, so we split the index set $I = \{0, \ldots, n\}$ into p overlapping blocks (see Figure 3.6.
2. On each processor p_j ($j \in \{1, \ldots, p\}$) the coefficients of g^* whose indices are in I_{p_j} are computed in descending order.
3. Having computed a new coefficient, its sign is compared to the sign of th previously computed coefficient. If the signs are different, the sign variation counter for this processor is increased by one.
4. Step 3 is repeated until either all indices in I_{p_j} have been processed or the value of the sign variation counter equals two.
5. In order to determine if g^* has zero, one or more than one sign variations, the sign variation counters of all processors are summed up.

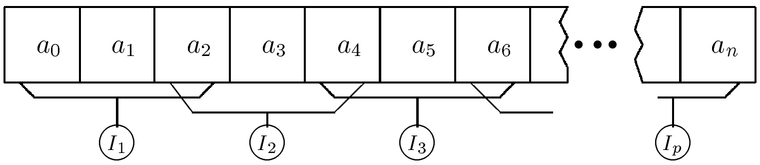

Figure 3.6: Splitting the Coefficients of g into Overlapping Blocks

For computing g_l it is not necessary to split the computation of the transformation into parallel tasks. In order to compute g_r we proceed analogously to the computation of g^*.

1. g_r is of the same degree as g and g_l. We split the index set $I = \{0, \ldots, n\}$ of the coefficients of g_l into p disjoint blocks (see Figure 3.7).
2. On each processor p_j ($j \in \{1, \ldots, p\}$ we compute the coefficients of g_r whose indices are in I_{p_j}.

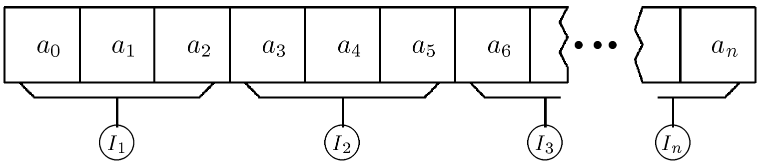

Figure 3.7: Split the Coefficients into Non-Overlapping Blocks for Computing g_r.

Implementing this approach requires only Algorithm 3.3.2(Uspensky Sub) to be changed. Algorithms 3.3.3, 3.3.4, 3.3.5 show the pseudo code for this approach. We have implemented this algorithm in the frame of the *Distributed Maple* system evaluating it on several test polynomials which were generated at random as well as taken from the `pacPlot` algorithm. In any case this approach resulted in an remarkable slowdown of computing time. We analyzed this behavior coming to the following conclusions:

- Precomputing the Pascal matrix for high degree polynomials is extremely time consuming. However, precomputing is necessary, as every parallel task computing coefficients of g^* and g_r makes uses of this matrix. Replacing the Pascal matrix by binomial coefficients evaluated at runtime is even more inefficient.
- Communication overhead for starting parallel tasks and waiting for the results is extremely high in this fine grained parallel approach.
- The conclusion we draw is that the most efficient way of doing the transformations is by using Horners's scheme and direct substitution.

Algorithm 3.3.3 Uspensky's method for computing isolating intervals for the real roots of the input polynomial in $]0,1[$ using parallel polynomial arithmetics in order to compute Moebius transformations.

Input: $A := [a_0, \ldots, a_n]$, list of coefficients of the squarefree polynomial $g(x) \in \mathbb{Q}[x]$ such that $a_0 \neq 0$,
p, the number of processors.
Output: L, set of isolating intervals for the real roots of $g(x)$ in $]0,1[$.

begin parallel
  Compute the sign variation s_i for the coefficients with indices $[n-(i-1)(k-1), \ldots, \max(0, \ldots, n-i(k-1))]$ of $g^* = (x+1)^n g(I(T(x,1)))$ using Algorithm 3.3.4, where $i = 1, \ldots, p$ and $k = \lfloor (n+1)/p \rfloor + 1$.
end parallel
$s := \sum_{i=1}^{p} s_i$.
if $s = 0$ **then**
  return$(\emptyset)$.
else if $s = 1$ **then**
  return$(\{]0,1[\})$.
end if
if $\sum_{i=0}^{n} a_i 2^{-i} = 0$ **then**
  $L := L \cup \{[\frac{1}{2}, \frac{1}{2}]\}$.
  Let A be the list of coefficients of $\left(\sum_{i=0}^{n} a_i x^i\right)/(2x-1)$.
end if
$A_l := [2^{n-i} a_i \mid a_i \in A]$.
Apply Algorithm 3.3.3 recursively to A_l with result L_l.
$L := L \cup \{]a_i/2, b_i/2[\mid]a_i, b_i[\in L_l\}$.
begin parallel
  Set $A_{r,i}$ to $[(i-1)k, \ldots, \min(n, ik-1)]$ and A_r to $A^*_{r,1} \bullet \cdots \bullet A^*_{r,p}$, where $A^*_{r,i}$ is the result of the application of Algorithm 3.3.5 to A_l and $A_{r,i}$.
end parallel
Apply Algorithm 3.3.3 recursively to A_r with result L_r.
$L := L \cup \{](a_i+1)/2, (b_i+1)/2[\mid]a_i, b_i[\in L_r\}$.
return(L).

3.3.5 A Speculative Algorithm

Parallelization of Uspensky's algorithm for real root isolation by investigating the two halfs of the interval $]0,1[$ simultaneously does not pay off. In most

Algorithm 3.3.4 Compute the sign variation of the coefficients of $(x+1)^n g(I(T(x,1)))$, where $g(x) \in \mathbb{Q}[x]$, whose indices are in the set I

Input: $A := [a_0, \ldots, a_n]$, list of coefficients of a squarefree polynomial $g(x) \in \mathbb{Q}[x]$ of degree n,
$I := \{l, l+1, \ldots, r-1, r\}$, set of indices such that $0 \leq l < r \leq n$.
Output: s, the sign variation of the sequence of coefficients of $x^l, \ldots, x^r$ in $(x+1)^n g(I(T(x,1)))$.

Let $P = (p_{ij})$ be the Pascal matrix of order n.
$v := 0;\ k := l+1$.
$s := \mathrm{sgn}(\sum_{j=l}^{n} p_{l+1,j+1} a_{n-j})$.
while $k \leq r$ and $v < 2$ **do**
  $s' := \mathrm{sgn}(\sum_{j=k}^{n} p_{k+1,j+1} a_{n-j})$.
  if $s \neq s'$ **then**
    $v := v+1;\ s := s'$.
  end if
  $k := k+1$.
end while
return(v).

Algorithm 3.3.5 Compute the coefficients of $g(T(x,1))$ for a polynomial $g \in \mathbb{Q}[x]$ whose indices are in the set I

Input: $A := [a_0, \ldots, a_n]$, list of coefficients of a polynomial $g(x) \in \mathbb{Q}[x]$,
$I := \{l, \ldots, r\}$, a set of indices such that $0 \leq l < r \leq n$.
Output: Coefficients a_i^t of $g(T(x,1))$, $(l \leq i \leq r)$.

Let $P_n := (p_{ij})$ be the Pascal matrix of order n.
for all $i \in I$ **do**
  $a_i^t := \sum_{k=i}^{n} p_{k+1,j+1} a_k$.
end for
return$([a_l^t, \ldots, a_r^t])$.

examples, the binary trees generated in the isolation process are narrow and consequently only a small subset of the available processors is active at a time.

By generalizing this strategy we can generate trees in such a way that on each level there are at least as many nodes as processors. For this, we start splitting the interval $]0,1[$ into 2^d subintervals, such that $2^d > p$, where p is the number of processors. For each subinterval we apply a Moebius transformation $M(x)$ to the input polynomial $f(x)$ such that the zeros of $f(M(x))$ in $]0,\infty[$ correspond to the zeros if $f(x)$ in the subintervals. By checking the sign variation of a transformed polynomial we can decide if a subinterval has to be investigated further or if it contains exactly one real root or no real root of f. After we have collected all subintervals which correspond to the same elve of the associated tree and have to be investigated further, we again split each of them into several subintervals such that the number of intervals to be checked is at least p.

If we can split each interval into 2^d subintervals ($d > 0$), this means that we can investigate d levels of the tree generated by the original method in one step. Of course, many subintervals will not contain a real root of the input polynomial,

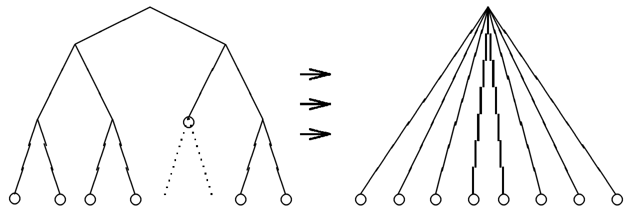

Figure 3.8: Uspensky's Method versus Speculative Approach

however the isolating intervals will be found faster. By pursuing this strategy, the depth first search approach of the original Uspensky method is turned into a breadth first search approach. This is more suitable for parallel computing, as computations on one particular level do not depend on each other.

We give a more detailed description and a pseudo code algorithm for this method (Algorithm 3.3.7(Speculative Real Root)). Let $g(x) \in \mathbb{Q}[x]$ be a square-free polynomial of degree n, p denote the number of available processors.

1. We check the sign variation of g. If $\mathrm{var}(g) = 0$ or $\mathrm{var}(g) = 1$ then we have reached a termination case and do not have to proceed further. If $\mathrm{var}(g) > 1$ then we add the interval $]0,1[$ to the set I of intervals to be investigated further and do the following.

2. We split each interval in I into 2^d subintervals such that the total number of subintervals is at least p. The subintervals are put into the set W of intervals we are working on.

3. If $W = \emptyset$ we have found all isolating intervals an got to step 6.

4. For each interval $]a,b[$ in W we start a parallel task which computes a transformed polynomial g^* such that the zeros of g^* in $]0,\infty[$ correspond to the zeros of g in $]a,b[$ and determines the sign variation of g^*.

5. We wait for at most p sign variations computed by the parallel tasks.
   - If the sign variation equal 1, we add the corresponding interval to the solution set L
   - If the sign variation is greater than 1, we add the corresponding interval to the set I

   and return to step 2.
6. Return the result list L.

In Algorithm 3.3.6 and 3.3.7(Speculative Real Root) this idea is summarized in pseudocode form.

Algorithm 3.3.6 From the input polynomial g, compute a transformed polynomial g^* such that the zeros of g in $]a,b[$ correspond to the zeros of g^* in $]0,\infty[$ and Compute the sign variation of g^*. Furthermore check if b is a zero of g.

Input: $g(x)$ a squarefree polynomial,
$]a,b[$, a subinterval of $]0,1[$.
Output: s_k, the sign variation of g^* and
if b is a zero of g, the interval $[b,b]$.

$M := \emptyset$
Transform the roots of $g(x)$ in $]a,b[$ to the roots of $g'(x)$ in $]0,1[$
if $g'(1) = 0$ **then**
  $M := \{[b,b]\}$
  $g'(x) := g'(x)/(x-1)$
end if
Transform the zeros of $g'(x)$ in $]0,1[$ to the zeros of $g^*(x)$ in $]0,\infty[$
$s := \mathrm{var}(g^*(x))$
return $(s, M,]a,b[)$

Remark 3.3.18. Choosing a good strategy for determining the number of subintervals each interval in I is split is essential for the efficiency of this algorithm. If the number is too big, far too much superfluous work is done by isolating intervals which do not contain a real root. On the other hand, if the number is too small, there are not enough tasks for an efficient parallelization.
We tried out the following three strategies for a given number p of processors.

1. Split each interval into $\lfloor \log_2 p \rfloor$ pieces. This leads to a high degree of speculation, most of the investigated subintervals do not contain a real root and the amount of superfluous computations is very high. We could not observe any speedup using this approach.
2. Split each of the m intervals into 2^d pieces, such that $|W| + m2^d \leq p < |W| + m2^{d+1}$ ($d = 1$ if $p \leq |W|$) and investigate exactly p subintervals in parallel. W denotes the set of subintervals which have not been investigated yet.
3. Split the m subintervals into 2^d pieces such that d satisfies the inequality $m2^d \leq p < m2^{d+1}$ ($d = 1$ if $p \leq |W|$). Schedule all subintervals for execution not taking into account that tasks from a previous step may still be active.

Algorithm 3.3.7 (Speculative Real Root) A speculative algorithm for isolating the real roots of the squarefree polynomial $g \in \mathbb{Q}[x]$ in $]0,1[$.

Input: $g(x) \in \mathbb{Q}[x]$, a squarefree polynomial,
p, the number of processors

Output: L, a list of isolating intervals for the zeros of $g(x)$ in $]0,1[$.

```
I := ∅; L := ∅; M := ∅; W := ∅; i := 0;
s := var(g(x))
if s = 1 then
  return(]0, 1[).
else if s = 0 then
  return(∅).
else
  I := {]0, 1[}
end if
loop
  if |I| > 0 then
    d := max(1, ⌊log_2(p/|I|)⌋)
    Split each interval in I into 2^d subintervals and keep the subintervals
    in W.
  end if
  if |W| = 0 and there are no pending tasks then exit loop
  j := max(1, i)
  begin parallel
    Apply Algorithm 3.3.6 to every interval ]a, b[ in W, computing the tuple
    (s, M, ]a, b[).
  end parallel
  i := i + |W|.
  for k = j to j + min(i − j, p) do
    Wait for a result (s_k, M_k, (a_k, b_k)) of Algorithm 3.3.6.
    if s_k = 1 then
      L := L ∪ {]a_k, b_k[}
    else if s_k > 1 then
      I := I ∪ {]a_k, b_k[}
    end if
    M := M ∪ M_k
  end for
end loop
return(L ∪ M)
```

Strategies 2 and 3 lead to best speedup values, 3 is even superior to 2, as by this approach each processor is continuously fed with tasks without waiting for a synchronization step.

Remark 3.3.19. In our Maple implementation of the algorithm we carry out isolation of positive and negative intervals in a single step. For each interval, we additionally mark if it belongs to the isolation process of $g(x)$ (for the positive zeros) or $g(-x)$ (for the negative zeros). In many cases the number of intervals generated by the speculative approach but do not contain a zero of the input polynomial is kept small.

3.4 Resultant Computation

Resultant computation in `pacPlot` is used as an elimination procedure in order to compute solutions of a two-dimensional system of bivariate polynomial equations. In [Sch99], a discussion of a parallel implementation of the resultant computation for bivariate polynomials using *Distributed Maple* can be found. We give a brief survey on the mathematical theory (see [Mis93, Win96, Col71]) and implementation strategies for this topic.

Definition 3.4.1. *An* integral domain I *is a commutative ring with identity satisfying*

$$\forall a, b \in I: \; ab = 0 \Rightarrow (a = 0) \vee (b = 0)$$

First we need some notation. Let I be an integral domain and

$$f(x) = \sum_{i=0}^{m} a_i x^i, \; g(x) = \sum_{i=0}^{n} b_i x^i$$

be non-constant polynomials in $I[x]$ such that $m \geq n$.
The Sylvester matrix $M(f,g)$ is the $(m+n) \times (m+n)$ matrix

$$M(f,g) := \begin{pmatrix} a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 & 0 & \dots & \dots & \dots & 0 \\ 0 & a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 & 0 & \dots & \dots & 0 \\ & & & & \vdots & & & & & & & \\ 0 & \dots & \dots & \dots & 0 & a_m & a_{m-1} & \dots & \dots & \dots & a_1 & a_0 \\ b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0 & 0 & \dots & \dots & \dots & 0 \\ 0 & b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0 & 0 & \dots & \dots & 0 \\ & & & & \vdots & & & & & & & \\ 0 & \dots & \dots & \dots & 0 & b_n & b_{n-1} & \dots & \dots & \dots & b_1 & b_0 \end{pmatrix}$$

The lines of $M(f,g)$ consist of the coefficients of the polynomials $x^{n-1}f(x), \dots, xf(x), f(x)$ and $x^{m-1}g(x), \dots, xg(x), g(x)$, i.e. there are n lines of coefficients of f and m lines of coefficients of g.
The determinant of $M(f,g)$ is the resultant of f and g, $\mathrm{res}(f,g) = \det(M(f,g))$.

In our context we are interested in the case where f and g are multivariate polynomials in r variable, that is

$$f(x_1, \dots, x_r) = \sum_{i=0}^{m} a_i(x_1, \dots, x_{r-1}) x_r^i, \; g(x_1, \dots, x_r) = \sum_{i=0}^{n} b_i(x_1, \dots, x_{r-1}) x_r^i$$

considered as polynomials in $I[x_1, \ldots, x_{r-1}][x_r]$. For plotting plane algebraic curves, $r = 2$.

Collins [Col71] describes an efficient sequential algorithm for multivariate polynomial resultant computation over $\mathbb{Z}$ which is based on polynomial remainder sequences and computation in homomorphic images of $\mathbb{Z}$. The general strategy is as follows:

- Consider $f, g \in \mathbb{Z}[x_1, \ldots, x_{r-1}][x_r]$.
- We reduce f and g modulo various primes p to $f_{(p)}$ and $g_{(p)}$ and compute $\mathrm{res}_{x_r}(f_{(p)}, g_{(p)})$.
- By the Chinese remainder algorithm we reconstruct $\mathrm{res}_{x_r}(f, g)$.
- For the subproblem of resultant computation over $\mathbb{Z}_p$ we use evaluation homomorphisms for the variables $x_1, \ldots, x_{r-1}$ and subsequent interpolation.
- The problem is ultimately reduced to resultant computation in $\mathbb{Z}_p[x]$, which can be done by the subresultant polynomial remainder sequence algorithm (see [Win96]).

We may only use primes p such that $\deg(f) = \deg(f_p)$ and $\deg(g) = \deg(g_p)$. Using modular arithmetic in $\mathbb{Z}_p$ we compute the modular resultant $c := \mathrm{res}_{x_r}(f_{(p)}, g_{(p)}),\ c \in \mathbb{Z}_p[x_1, \ldots, x_r]$. We then apply the Chinese remainder algorithm to lift the result from $\mathbb{Z}_p$ to $\mathbb{Z}_{P \cdot p}$,

$$C' := \mathrm{cra}(C, c, P, p) \in \mathbb{Z}_{P \cdot p}[x_1, \ldots, x_r]$$

where P is the product of all primes used so far and C the combination of the corresponding modular resultants. The algorithm iterates until P exceeds the precomputed upper bound for the size of the coefficients of the resultant and then returns C. Lemma 3.4.3 gives a bound for the size of the coefficients of the resultant.

Definition 3.4.2. *The* norm *of a multivariate polynomial f over $\mathbb{Z}$ is inductively defined as follows:*

$$\begin{array}{l} \mathrm{norm}(f(x_1)) = \sum_{i=0}^{m} |a_i| \quad \textit{for } f = \sum_{i=0}^{m} a_i x^i \in \mathbb{Z}[x_1] \\ \mathrm{norm}(f(x_1, \ldots, x_r)) = \sum_{i=0}^{m} \mathrm{norm}(a_i) \quad \textit{for } f = \sum_{i=0}^{m} a_i x^i \in \mathbb{Z}[x_1, \ldots, x_r] \end{array}$$

Lemma 3.4.3. *Let $f(x_1, \ldots, x_r) = \sum_{i=0}^{m} a_i(x_1, \ldots, x_{r-1}) x_r^i$ and $g(x_1, \ldots, x_r) = \sum_{i=0}^{n} b_i(x_1, \ldots, x_{r-1}) x_r^i$ be polynomials in $\mathbb{Z}[x_1, \ldots, x_r]$. Let $d = \max_{0 \le i \le m}(\mathrm{norm}(a_i))$ and $e = \max_{0 \le i \le n}(\mathrm{norm}(b_i))$ and c an integer coefficient of $\mathrm{res}_{x_r}(f, g)$.*
Then $|c| \le (m+n)! d^n e^m$.

Furthermore, by inspection of the Sylvester matrix, we can obtain a degree bound for the resultant in its main variable. For this, let $m_r := \deg_{x_r}(f)$, $n_r := \deg_{x_r}(g)$, $m_{r-1} := \deg_{x_{r-1}}(f)$, $n_{r-1} := \deg_{x_{r-1}}(g)$. Then $\deg_{x_{r-1}}(\mathrm{res}_{x_r}(f, g) \le m_r n_{r-1} + n_r m_{r-1}$.

A sketch of the sequential algorithm is given in Algorithm 3.4.1

Various parallel versions of the modular algorithm have been investigated and implemented on shared memory machines and on workstation networks.

Algorithm 3.4.1 A modular algorithm for computing the resultant of two multivariate polynomials over $\mathbb{Z}$

Input: $f, g \in \mathbb{Z}[x_1, \ldots, x_r]$, $\deg_{x_r}(f) = m$, $\deg_{x_r}(g) = n$.
Output: $C := \mathrm{res}_{x_r}(f, g)$.

$C := 0$; $P := 1$
Set cb to an upper bound for the coefficients in $\mathrm{res}_{x_r}(f, g)$.
while $P \leq cb$ **do**
  Take a new prime number p.
  $f_{(p)} := f \mod p$.
  $g_{(p)} := g \mod p$.
  if $\deg_{x_r}(f_{(p)}) = m \wedge \deg_{x_r}(g_{(p)}) = n$ **then**
    $c := \mathrm{res}_{x_r}(f_{(p)}, g_{(p)})$.
    $C := \mathrm{cra}(C, c, P, p)$.
    $P := P \cdot p$.
  end if
end while
return(C).

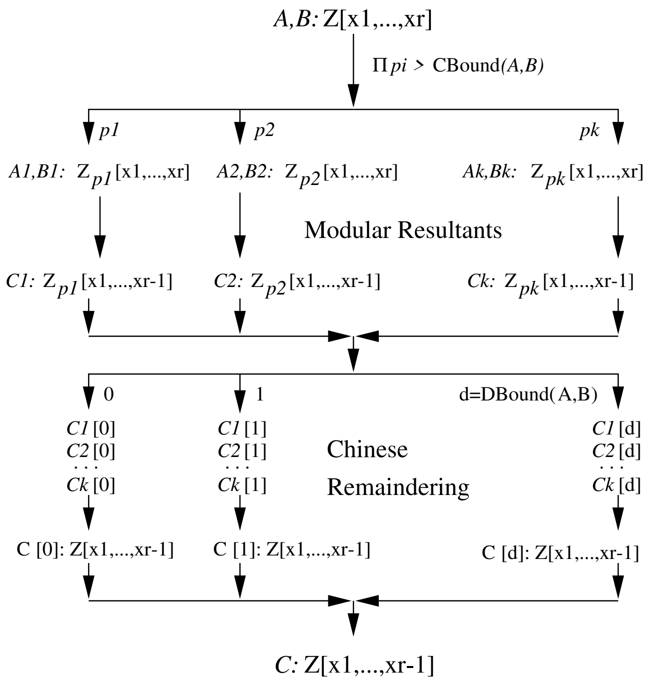

Figure 3.9: Structure of the Parallel Modular Resultant Algorithm

Their common idea is to compute the resultants in the individual modular domains in parallel; they differ in their approaches to efficiently combine the modular resultants in order to obtain the resultant over the integers. In this approach we apply the idea used by [HL94] where the sequential structure of the combination phase is maintained but the individual coefficients of the resultant are computed in parallel (see Figure 3.9).

Algorithm 3.4.2 A parallel modular algorithm for computing the resultant of two multivariate polynomials over $\mathbb{Z}$

Input: $f, g \in \mathbb{Z}[x_1, \dots, x_r]$, $\deg_{x_r}(f) = m,\ \deg_{x_r}(g) = n$.
Output: $C := \mathrm{res}_{x_r}(f, g)$.
$C := 0;\ P := 1$
$T := [];\ L := []$.
Set cb to an upper bound for the coefficients in $\mathrm{res}_{x_r}(f, g)$.
Set db to a degree bound for $\mathrm{res}_{x_r}(f, g)$.
while $P \leq cb$ **do**
    Take a new prime number p.
    $f_{(p)} := f \mod p$.
    $g_{(p)} := g \mod p$.
    if $\deg_{x_r}(f_{(p)}) = m \wedge \deg_{x_r}(g_{(p)}) = n$ **then**
        $t := \mathrm{start}(\mathrm{res}_{x_r}(f_{(p)}, g_{(p)}))$.
        $P := P \cdot p;\ T := T||[t];\ L := L||[p]$.
    end if
end while
$R := \mathrm{WaitForAll}(T)$.
$T := []$.
for $i \in \{1, \dots, db\}$ **do**
    $t := \mathrm{start}(\mathrm{cra}(L, [R[j]_i \mid j \in \{1, \dots \mathrm{length}(R)]))$.
    $T := T||[t]$.
end for
$\mathrm{WaitForAll}(T)$.
return(C).

3.5 The `dpacPlot` Algorithm

In this section we describe how the various ideas developed above are incorporated into a parallel version of the `pacPlot` algorithm, the `dpacPlot` algorithm (distributed `pacPlot`). Parallelization affects the critical point computation step of `pacPlot` only. As a basis for parallelization we take the algorithm for which we have done the sequential improvements concerning the refinement of the isolating intervals and the check which combinations of isolating intervals actually give isolating rectangles for the critical points. Obviously, the various subalgorithm occurring in `pacPlot` simply have to be replaced by their parallel counterpart. Parallelization of the sequential improvements very easy. These parts consist of loops where each iteration is independent from the others. So we simply start as many parallel tasks as there are loop iterations and wait for the results.

In Algorithm 3.5.1(Parallel Critical Points) we give a high-level description of how critical point computation is performed in `dpacPlot`.

Algorithm 3.5.1 (Parallel Critical Points) A parallel algorithm for critical point computation

---

Input: $f(x,y) \in \mathbb{Q}[x,y]$, irreducible
Output: L, list of critical points if f with respect to y-direction.

$f_x := \frac{\partial f}{\partial x}(x,y);\ L := \emptyset$.
Apply `bsolve` to get a set T of triangular systems from f and f_x.
for all $(p,q) \in T$ **do**
    $F := \{p_i \mid \prod_i p_i = p\}$
    for all $p \in F$ **do**
        $Y := \{]\eta_{i1}, \eta_{i2}[\}$ set of isolating intervals for the real roots of p obtained by the **parallel real root isolation** algorithm.
        $r := \mathrm{res}_y(p,q)$ resultant of p and q computed by the **parallel modular approach**.
        $X := \{]\xi_{i1}, \xi_{i2}[\}$ set of isolating intervals for the real roots of r computed by the **parallel real root isolation** algorithm.
        do in parallel
            For each $]\eta_1, \eta_2[\in Y$ and each $]\xi_1, \xi_2[\in X$ compute the sign of $q(\xi_1, \alpha)$ and $q(\xi_2, \alpha)$, where α is the real root of p represented by the isolating interval $]\eta_1, \eta_2[$. Add $]\xi_1, \xi_2[\times]\eta_1, \eta_2[$ to L if the signs are different or at least one sign is 0.
        end do
    end for
end for
do in parallel
    Refine each interval occurring in L to the desired granularity.
end do

From our experiences gained in benchmarking the `dpacPlot` algorithm we found the following factors which limit speedup:

- The number of sign computations for checking which combinations of isolating intervals give isolating rectangle for the critical points.
- The complexity of the sign computations. Depending on the size of the rational numbers which delimit the isolating interval for the algebraic number and the size of the coefficients of the polynomial which represents the real algebraic number for which the sign has to be determined, computing time may vary remarkably.

3.6 Execution Times and Conclusions

We have systematically benchmarked the `dpacPlot` algorithm with the randomly generated curves used for the sequential profiling (Section 3.2.2) in three environments consisting of 24 processors each:

- A cluster of 16 Linux PCs operating various Pentium processors and 4 Silicon Graphics Octanes (2 R10000@250MHz each) linked by three 10Mbit Ethernet subnetworks connected to a router;

- a Silicon Graphics Cray Origin 2000 multiprocessor with 64 R12000 processors at 300MHz each. 24 processors were used;
- a mixed configuration consisting of the 4 dual-processor Octanes and 16 processors of the Origin multiprocessor. The Origin 2000 and the Octanes are linked by a high bandwidth ATM line, while the Octanes are linked via a 10Mbit Ethernet network.

The following table illustrates the performance of the various benchmarking configurations in the three environments relative to a PIII@450MHz processor. These numbers were measured by a representative Maple benchmark.

Performance (relative to PIII@450MHz)						
Environment	1	2	4	8	16	24
Cluster	0.58	1.16	2.32	5.40	12.56	18.30
Origin	0.78	1.56	3.12	6.24	12.48	18.72
Mixed	0.58	1.16	2.32	4.64	10.88	17.12

- The first table in Figure 3.10 lists execution times in seconds for Step 1 of the `dpacPlot` algorithm measured on the various configurations of benchmarking environments and processor numbers. For each example, we additionally give the execution time of the improved sequential algorithm in parenthesis.
- The second table in Figure 3.10 presents the absolute speedups $\frac{T_s}{T_n}$, where T_s denotes the sequential execution time and T_n denotes the parallel execution time using n processors. The first row of diagrams in Figure 3.11 [1] visualizes this data, the second row visualizes these speedups multiplied by $\frac{n}{\sum_{i=1}^{n} p_i}$, where p_i denotes the relative performance of processor i.
- The third table in Figure 3.10 presents the scaled efficiency $\frac{T_s}{T_n \sum_{i=1}^{n} p_i}$, which compares the speedup we actually got to the upper bound of the speedup we could have got. The third row of diagrams in Figure 3.11 [1] visualizes this data.

Analyzing the experimental data gives various intersting results:

- Most obviously, the speedups for larger examples are better than with smaller ones. For Example 1, the cluster/mixed configuration gives an absolute speedup of 16 but only a speedup of 8 for Example 2. Example 3 is too small to yield a speedup better than 2. Example 4 operates with extremely large intermediate data that have to be communicated among the processors; its speedup is thus bound by 8. This example also shows the superiority of the networking performance on the Origin multiprocessor.
- Example 1 actually gives *too* good results, namely a scaled efficiency between 1 and 2. While the less dramatic super-one efficiencies in Examples 2 and 4 may be partially explained by errors in our performance estimations, the results of Example 1 are a consequence of the fact that the speculative real root isolation algorithm computes narrower isolating intervals

[1] The markers $+$, $\Box$, and $\times$ denote execution on cluster, Origin and in the mixed configuration, respectively.

Execution times for Step 1 (s)							
Example	Architecture	1	2	4	8	16	24
1 (8989s)	Cluster	14537	8049	2652	1288	634	564
	Mixed	14537	8049	2652	1346	554	550
	Origin	7136	4013	1642	798	418	540
2 (737s)	Cluster	852	512	293	299	184	165
	Mixed	852	512	293	174	118	122
	Origin	613	361	210	136	101	109
3 (158s)	Cluster	269	198	121	85	82	78
	Mixed	269	198	121	82	68	75
	Origin	196	146	92	63	63	65
4 (11556s)	Cluster	25058	15200	6488	3675	1979	2682
	Mixed	25058	15200	6488	4171	2601	1532
	Origin	12470	6683	3517	2156	1595	1407

Absolute Speedups							
Example	Architecture	1	2	4	8	16	24
1 (8989s)	Cluster	0.6	1.1	3.4	6.7	14.1	16.0
	Mixed	0.6	1.1	3.4	6.7	16.2	16.3
	Origin	1.3	2.2	5.5	11.3	21.5	16.7
2 (737s)	Cluster	0.9	1.4	2.5	2.5	4.0	4.5
	Mixed	0.9	1.4	2.5	4.2	6.2	6.1
	Origin	1.2	2.0	3.5	5.4	7.3	6.8
3 (158s)	Cluster	0.6	0.8	1.3	1.9	1.9	2.0
	Mixed	0.6	0.8	1.3	1.0	2.3	2.1
	Origin	0.8	1.1	1.7	2.5	2.5	2.5
4 (11556s)	Cluster	0.5	0.8	1.8	3.1	5.8	4.3
	Mixed	0.5	0.8	1.8	2.8	4.4	7.5
	Origin	0.9	1.7	3.3	5.4	7.3	8.2

Scaled Efficiency							
Example	Architecture	1	2	4	8	16	24
1 (8989s)	Cluster	1.1	1.0	1.5	1.3	1.1	0.9
	Mixed	1.1	1.0	1.5	1.4	1.5	1.0
	Origin	1.6	1.4	1.8	1.8	1.7	0.9
2 (737s)	Cluster	1.5	1.2	1.1	0.5	0.3	0.2
	Mixed	1.5	1.2	1.1	0.9	0.6	0.4
	Origin	1.5	1.3	1.1	0.9	0.6	0.4
3 (158s)	Cluster	1.0	0.7	0.6	0.3	0.2	0.1
	Mixed	1.0	0.7	0.6	0.3	0.2	0.1
	Origin	1.0	0.7	0.6	0.4	0.2	0.1
4 (11556s)	Cluster	0.8	0.7	0.8	0.6	0.5	0.2
	Mixed	0.8	0.7	0.8	0.6	0.5	0.2
	Origin	1.2	1.1	1.1	0.9	0.6	0.4

Figure 3.10: Execution Times, Speedups and Scaled Efficiency

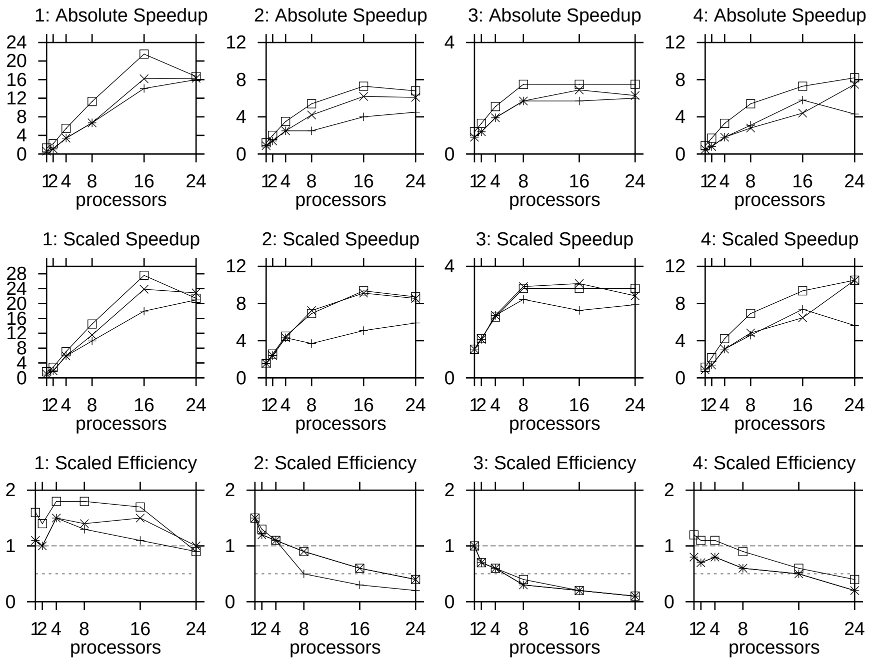

Figure 3.11: Experimental Results

than its sequential counterpart. While this is more costly, the subsequent detection of matching combinations of x-intervals and y-intervals in this example benefits from qualitatively better data becoming twice as fast.

- The scaled speedups of the mixed configuration is in most cases comparable to that of the Origin. The cluster configuration with 8 to 24 processors (here also the Linux PCs are employed) behaves significantly worse. This seems to indicate that the network performance of the Linux PCs is lower than that of the SGI machines (in both configurations, all communication goes through Ethernet).

- When we consider the execution times of the parallel subalgorithms individually (see Figure 3.6), we realize that speedups are partially much higher than the speedup of the overall algorithm. In Example 2, with 24 processors, the parallelization of resultant computation gives absolute speedups of 10.2 (Origin), 7.9 (mixed) and 5.6 (cluster). The parallelization of the checking phase gives absolute speedups of 14.1, 16.1(!) and 12.0 in the respective configurations. Although both phases together account for almost 80% of the total work, the less efficient parallelization of the remaining (much shorter) phases limits the overall speedup to 6.8, 6.0 and 4.5, respectively.

In Figure 3.13 we see execution profiles generated from *Distributed Maple* for Example 2 running on 16 processors. The top diagram corresponds to execution on the cluster environment, the second diagram illustrates execution on the

Origin multiprocessor and the third diagram reflects execution in the mixed environment. We see the real root isolation phase followed by the phase for resultant computation (divide and conquer and combine), the second real root isolation, the solution tests and the refinement of the intervals.

- Real root isolation typically consists of a huge amount of small tasks, each of which corresponds to an investigation of a subinterval of the interval $]0, 1[$. Our speculative approach generates enough tasks to avoid idle processors.

- Resultant computation splits in two parts, namely computing resultants in homomorphic images (which can be seen in the leftmost part of the resultant-phase) and combining the modular resultants using the Chinese remainder algorithm (which can be seen in the rightmost part of the resultant phase). One could optimize the Chinese remainder phase by adjusting the task granularity in order to employ more processors.

- The solution test consists of many tasks whose length is virtually unpredictable as it depends on several parameters which can hardly be determined. Typically, most of the tasks require moderate execution time and only few tasks dominate computation and hence limit the speedup.

- In the refinement phase, the number of tasks is limited by the number of solutions determined in the previous step. Therefore it might happen that for large processor numbers, not all processors are involved in computation.

Detailed Execution Time Analysis: Curve 1							
Environment	Operation	1	2	4	8	16	24
Cluster	Realroot	56	44	34	45	35	30
	Resultant	824	396	196	117	61	52
	Checking	13558	7561	2355	1072	487	432
	Refinement	88	35	55	42	37	37
Origin	Realroot	40	32	27	23	23	26
	Resultant	555	281	145	81	45	39
	Checking	6373	3614	1426	667	323	447
	Refinement	158	76	35	17	16	18
Mix	Realroot	55	44	34	24	24	26
	Resultant	824	396	196	105	53	40
	Checking	13558	7561	2355	1175	442	424
	Refinement	88	35	55	28	22	47

Detailed Execution Time Analysis: Curve 2							
Environment	Operation	1	2	4	8	16	24
Cluster	Realroot	108	80	72	45	35	30
	Resultant	485	232	118	86	49	44
	Checking	231	154	75	35	17	24
	Refinement	22	52	22	25	22	26
Origin	Realroot	78	57	45	44	39	53
	Resultant	332	163	86	48	32	25
	Checking	150	108	59	30	17	18
	Refinement	50	29	26	9	8	9
Mix	Realroot	108	80	72	50	41	56
	Resultant	485	232	118	49	40	32
	Checking	231	154	75	38	23	18
	Refinement	22	52	22	13	10	10

Detailed Execution Time Analysis: Curve 3							
Environment	Operation	1	2	4	8	16	24
Cluster	Realroot	8	14	12	8	10	12
	Resultant	114	61	31	19	17	17
	Checking	138	109	68	42	40	37
	Refinement	6	8	5	9	7	7
Origin	Realroot	6	14	11	8	11	16
	Resultant	80	42	23	13	14	13
	Checking	97	80	49	32	30	27
	Refinement	10	6	4	4	4	3
Mix	Realroot	8	14	12	8	10	13
	Resultant	114	61	31	18	17	17
	Checking	138	109	68	46	33	35
	Refinement	6	8	5	4	4	4

Detailed Execution Time Analysis: Curve 4							
Environment	Operation	1	2	4	8	16	24
Cluster	Realroot	86	42	30	32	29	20
	Resultant	7112	3356	1678	968	601	512
	Checking	17600	11605	4615	2528	1215	2013
	Refinement	144	78	48	27	20	20
Origin	Realroot	60	31	26	18	17	17
	Resultant	4778	2480	2437	783	513	439
	Checking	7442	4021	2039	1245	967	851
	Refinement	107	67	37	20	17	16
Mix	Realroot	86	42	30	22	40	20
	Resultant	7112	3356	1678	984	586	486
	Checking	17600	11605	4615	3000	1832	894
	Refinement	144	78	48	38	19	16

Figure 3.12: Detailed Execution Times

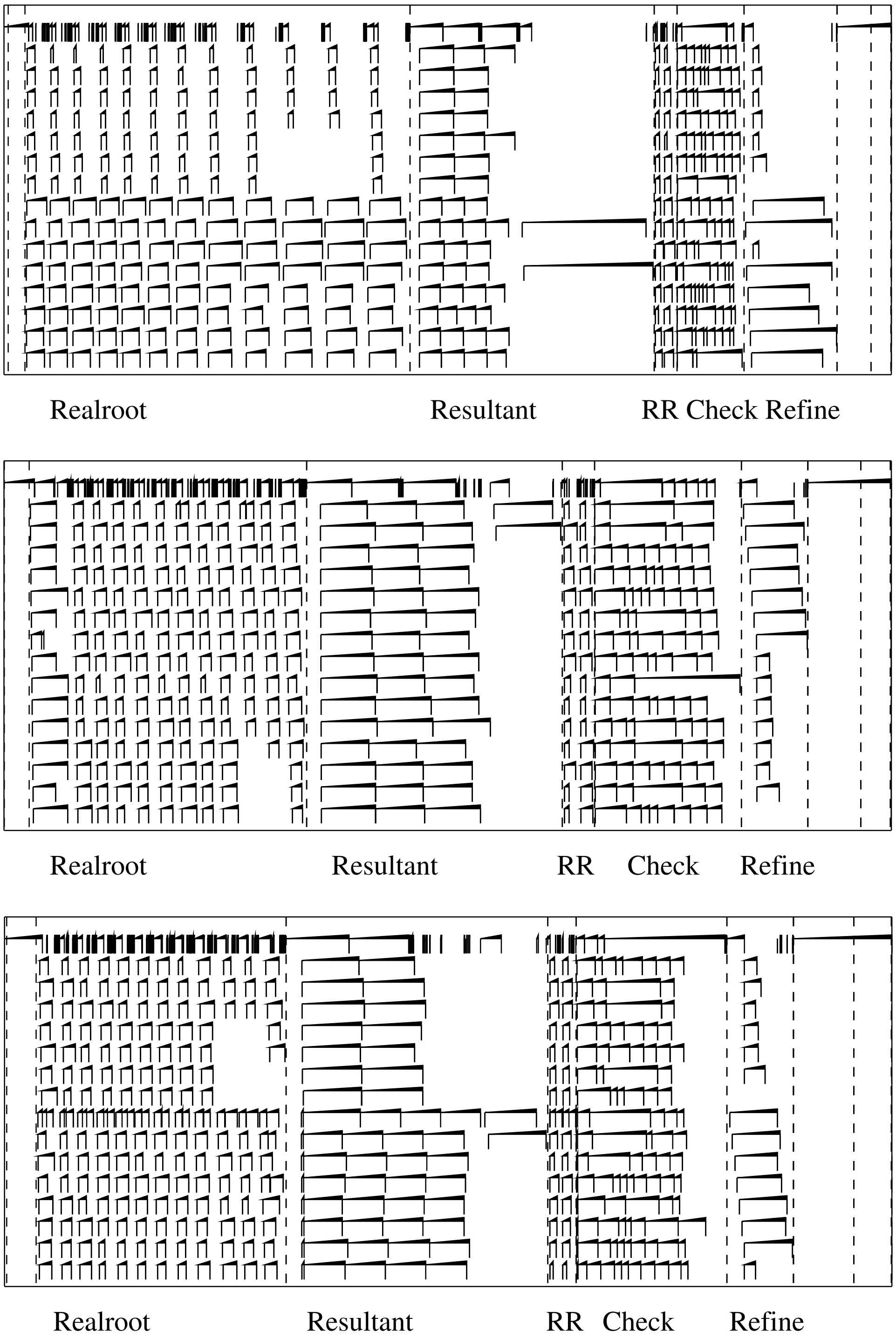

Figure 3.13: Dynamic Behavior

Chapter 4

Plotting Surface to Surface Intersections

In this chapter we describe the parallelization of a hybrid symbolic numerical algorithm for reliably tracing space curves defined by surface-to-surface intersections. Our basis is the already existing sequential algorithm for which we do some careful profiling in order to isolate the most time consuming parts. These investigation revealed that computation of the Dixon resultant, a multivariate resultant formulation, real rot isolation and computation of initial points for visualization account for most of the computing time. We describe parallel algorithms for these three subproblems and show the structure of the parallel plotting algorithm.

The structure of this chapter is as follows: In Section 4.1 we describe the problem of reliably plotting surface-to-surface intersections. In Section 4.2 we describe the sequential plotting algorithm and give a detailed profiling of its most time consuming operations. In Section 4.3 we describe the Dixon resultant for n polynomial equations in $n+1$ variables. In Section 4.4 we show how the sequential algorithm benefits from parallel computing. In Section 4.5 we present speedup diagrams and performance evaluations of the parallel algorithm and analyze its dynamic behavior.

4.1 Introduction

Approximating and representing surface-to-surface intersections is a fundamental and difficult problem in Computer Aided Geometric Design (CAGD) and solid modeling. Given two trivariate polynomials with rational number coefficients, the algorithm produces a space graph which is an approximation of the real algebraic set of the intersection in Euclidean space (a space curve). In Figure 4.1 we see a plot of the intersection of the two surfaces

$$\begin{aligned} f &= (x^2+y^2+z^2)^2-4x^2-4y^2 \\ g &= x^2+xy^2z-1 \end{aligned}$$

Several algorithms have been developed for solving this problem; roughly speaking, surface intersection methods can be classified into two main categories:

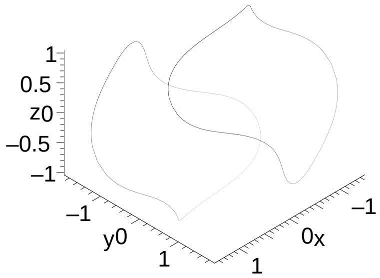

Figure 4.1: A Sample `ssiPlot` Plot

1. **Symbolic Methods**, based upon the primitive element theorem of Algebraic Geometry.
2. **Numerical Approaches**: These methods generate point sequences of each separated branch of the intersection curve in a direction determined by the curve's local differential geometry.

Symbolic methods are reliable in the sense that points of the curve which are near or at a singularity and small loops are handled correctly. However, these methods are quite time consuming as they involve decompositions of algebraic sets and computation of birational mappings.

On the other hand, numerical methods are incomplete by themselves as they require starting points for each simple branch of the intersection. Finding these starting points is a difficult task. Moreover, the methods are not reliable and have great difficulties when tracing through singularities. As in the case of plane algebraic curve plotting, the reason is that near or at a singular point the condition number of the linear system whose solution approximate a location on the curve, is so large that roundoff errors destroy all accuracy.

In [], Quoc-Nam Tran developed an algorithm which is a compromise between symbolic and numerical methods. The general idea of the `ssiPlot` algorithm is similar to that of `pacPlot`. It overcomes the difficulties of numerical tracing, namely finding a starting point for every real branch and tracing through singularities by using techniques for real rot isolation for polynomials with real algebraic number coefficients. Furthermore the method does not require the time expensive computations of birational mappings, reconstructions, etc. as in purely symbolic algorithms.

4.2 The `ssiPlot` Algorithm

The `ssiPlot` algorithm (surface-to-surface intersection plot) is part of the CASA system and uses a hybrid symbolic numerical approach for plotting surface - to - surface intersections defined by two trivariate oplynomials over $\mathbb{Q}$.

4.2.1 The Sequential Algorithm

The hybrid symbolic numerical approach pursued in `ssiPlot` combines the exactness of symbolic methods together with the speed and efficiency of numerical approximations. In `ssiPlot` a symbolic preprocessing step is used where the points determining the topological structure of the space curve are computed by exact algorithms. For tracing the simple branches inbetween the previously computed points, Tran's Extended Newton Method [Tra94] is used. The general structure of the algorithm looks as follow:

1. Compute all critical points of the intersection curve.
2. Generate initial points of visualization for each simple branch of the curve.
3. Trace the simple branches of the curve using numerical approximation.

Critical points are defined analogously to the plane algebraic curve case.

Notation 1. For $f, g \in \mathbb{Q}[x, y, z]$, J_x denotes the Jacobian of f and g with respect to y and z. That is,

$$J_x := \begin{pmatrix} \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \\ \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} \end{pmatrix}.$$

Definition 4.2.1. *Let $f, g \in \mathbb{Q}[x, y, z]$, $\mathcal{C}$ the algebraic set defined by $\mathcal{C} := \{(\xi, \eta, \zeta) \in \mathbb{R}^3 \mid f(\xi, \eta, \zeta) = g(\xi, \eta, \zeta) = 0\}$, and J_x denote the Jacobian of f and g with respect to x, such that* $\det(J_x) \neq 0$.

1. *$(\xi, \eta, \zeta) \in \mathcal{C}$ is a* critical point w.r.t. *x of $\mathcal{C}$ iff $f(\xi, \eta, \zeta) = g(\xi, \eta, \zeta) = J_x(\xi, \eta, \zeta) = 0$.*
2. $\mathbf{CP}(\mathcal{C}) := \{(\xi, \eta, \zeta) \in \mathcal{C} \mid (\xi, \eta, \zeta)$*is a critical point*$\}$.

Critical points determine the topological structure of the intersection curve in the same way as they do it for plane algebraic curves. They partition the three-dimensional Euclidean space into 3D-stripes such that within every stripe the space curve has no intersecting branches and the Jacobian is regular in every curve point. The picture of the space curve is generated by tracing all simple branches for all stripes. For every stripe, the initial points of visualization have to be computed. For this, all critical points are ordered with respect to their x-coordinate. For each interval on the x-axis delimited by x_1 and x_2 the midpoint ξ of the interval is computed and the curve is intersected with the plane $x = \xi$. The intersection points are starting points for rendering the curve.

Next all simple branches are traced in both directions by using the Extended Newton Method [Tra94] starting at the initial points of visualization until the border of a stripe is reached. The picture of the entire space curve is obtained by combining the individual traces for all simple branches.

In step 1 and 2 of `ssiPlot` symbolic methods for variable elimination, namely the Dixon resultant, and polynomial real root isolation are used. The Dixon resultant is described in detail in Section 4.3, we only mention here that it is used to eliminate several variables from a system of polynomial equations at once. We first present a pseudocode algorithm for critical point computation and proceed with discussing the computation of starting points for the rendering process.

Remark 4.2.2. 1. In Algorithm 4.2.1(Critical Points) we only compute isolating intervals for the x-coordinates of the critical points. This is sufficient for determining the starting points of visualization as in this step only information about the x - coordinates is required.

Algorithm 4.2.1 (Critical Points) Computation of critical points for a space curve $\mathcal{C}$ defined by the intersection of two surfaces.

Input: $f, g \in \mathbb{Q}[x, y, z]$ representing two surfaces whose intersection is a one-dimensional algebraic set.

Output: L, a list of isolating intervals for the x-coordinates of the critical points of $\mathcal{C}$.

1: $d := \det(J_x(f, g))$.
2: $h := \mathrm{dres}_{y,z}(f, g, d)$ $\{h$ is the Dixon resultant of f, g w.r.t y and $z.\}$
3: $L := \{]\xi_1\xi_2[\mid \exists\xi : \xi_1 < \xi < \xi_2 \wedge h(\xi) = 0\}$ $\{L$ is a list of isolating intervals for the real roots of h with the desired granularity$\}$.
4: **return**(L).

2. As in `pacPlot` we never compute exact solutions of polynomial systems but only isolating intervals of desired precision. The width of an interval has to be smaller then the resolution of the output device.

3. If $\det(J_x) = 0$ we exchange the order of the variables in order to get a non-singular Jacobian.

4. The Dixon resultant used in step 2 of Algorithm 4.2.1(Critical Points) allows elimination of several variables in a polynomial system at the same time. For details we refer to Section 4.3.

Computation of initial points for visualization is more complicated than in the plane algebraic curve case. Elimination techniques based upon resultant computation and real root isolation for polynomials with real algebraic number coefficients have to be applied. In Algorithm 4.2.2(Initial Points) the pseudo code for initial point computation is presented. For a fixed x-coordinate ξ all solutions of $f(\xi, y, z) = 0$ and $g(\xi, y, z) = 0$ are computed. Basically the same techniques as in `pacPlot`'s checking step, where the intervals which form isolating rectangles are determined, are used in this place. In the sequel we describe the computation of an initial point for a given ξ in detail. For shorthand we write f and g instead of $f(\xi, y, z)$ and $g(\xi, y, z)$ respectively.

1. We compute $A(y) := \mathrm{res}_z(f, g)$. From Theorem 3.2.3 we know that any zero of A either makes the leading coefficient of f and g vanish or can be extended to a common zero of the polynomial system.

2. We compute the greatest squarefree divisor of A and determine isolating interval $]\eta_1, \eta_2[$ for the real roots of A.

3. Any real root of A for which $\mathrm{lcoeff}(f) = 0$ and $\mathrm{lcoeff}(g) = 0$ is also a real root of $lc := \gcd(\gcd(\mathrm{lcoeff}_z(f), \mathrm{lcoeff}_z(g)), A)$. We also compute isolating intervals for the real roots of lc and set Y to the set of those isolating intervals for A which are not isolating intervals for lc.

4. For every interval $]\eta_1, \eta_2[\in Y$ we do the following:

   (a) Set η to the real algebraic number defined by $]\eta_1, \eta_2[$.

   (b) First we compute q, the gcd of $f(\xi, \eta, z)$ and $g(\xi, \eta, z)$ over $\mathbb{Q}(\eta)[z]$ (any common zero of the two polynomials is also a common zero of the gcd).

Algorithm 4.2.2 (Inital Points) Computation of initial points for visualization of the surface-to-surface intersection given by two trivariate polynomials.

Input: $f, g \in \mathbb{Q}[x, y, z]$ representing two surfaces whose intersection is a one-dimensional algebraic set,
$x_1 < x_2 < \ldots < x_r$ sequence of x-coordinates of the critical points of the intersection curve.

Output: I, set of initial points for visualization.

$I := \emptyset$.
for $i = 1$ to $r - 1$ **do**
  $\xi := (x_1 + x_{i+1})/2$.
  $A_i(y) := \mathrm{res}_z(f(\xi, y, z), g(\xi, y, z))$.
  $\overline{A}_i(y) := A_i(y)/\gcd(A_i(y), \frac{\partial}{\partial y} A_i(y))$.
  $lc(y) := \gcd(\gcd(\mathrm{lcoeff}_z(f(\xi, y, z)), \mathrm{lcoeff}_z(g(\xi, y, z))), \overline{A}_i(y))$.
  $L_1 := \{]\eta_1, \eta_2[\subset \mathbb{Q} \mid]\eta_1, \eta_2[$ is an isolating interval for $\overline{A}_i(y)\}$
  $L_2 := \{]\eta_1, \eta_2[\subset \mathbb{Q} \mid]\eta_1, \eta_2[$ is an isolating interval for $lc(y)\}$.
  $Y := L_1 - L_2$.
  for all $]\eta_1, \eta_2[\in Y$ **do**
    Let η be the real root defined by $]\eta_1, \eta_2[$.
    $q := \gcd(f(\xi, \eta, z), g(\xi, \eta, z))$.
    $Z := \{]\zeta_1, \zeta_2[\in \mathbb{Q} \mid]\zeta_1, \zeta_2[$ is an isolating interval for $q(\xi, \eta, z)\}$.
    $Z' := \{(\zeta_1 + \zeta_2)/2 \mid]\zeta_1, \zeta_2[\in Z\}$.
    $I := I \cup \{\xi\} \times \{\frac{\eta_1 + \eta_2}{2}\} \times Z'$.
  end for
end for
return(I).

(c) By real root isolation for polynomials with real algebraic number coefficients we determine isolating intervals $]\zeta_1, \zeta_2[$ for the zeros of q.

(d) We add the point $(\xi, \frac{\eta_1+\eta_2}{2}, \frac{\zeta_1+\zeta_2}{2})$ to the set of initial points for visualization.

4.2.2 Profiling `ssiPlot`

In this section we give an analysis of `ssiPlot`'s most time consuming parts based on sample executions of the algorithm for examples of medium and high degree. The goal of this analysis is to determine the those operations in the algorithm which may benefit from parallel computation.

The sample runs were executed on an PIII@450MHz processor in a Linux environment. For the defining polynomials for the surface-to-surface intersections and the plots see Appendix B.

(Sub-) Algorithm	Example 1	Example 2	Example 3	Example 4
Dixon	1709	285	1263	4685
Realroot	123	32	157	342
Initial points	206	102	40	809
Traces	24	22	20	59
$\sum$ `ssiPlot`	2119	475	1523	5973

Table 4.1: Profiling the `ssiPlot` algorithm (all timings in seconds)

In our examples execution time is dominated by the computation of the Dixon resultant. Real root isolations and computation of the initial points for visualization also take a significantly high amount of time. Furthermore we notice that, different from `pacPlot`, tracing the simple branches of the space curve also takes a considerable amount of time. This is due to the fact that a three dimensional Newton method is used here.

4.3 The Dixon Resultant

There are various multivariate resultant formulations [KS95], among them the Dixon resultant, which allows elimination of several variables from a polynomial system of a special form at one time. We describe the basic theory of the Dixon resultant, recall a result about its generalization to arbitrary polynomial systems and describe our parallel implementation in Maple.

4.3.1 The Dixon Resultant for Generic n-degree Polynomials

There exist many techniques for solving a system of algebraic polynomial equations. Resultant computations are perhaps the most popular way to obtain information about solutions of polynomial equations. Successive Sylvester resultant computations can be used to solve a system of polynomial equations in many variables by eliminating variable one by one. However, successive resultant techniques are rather inefficient, computational complexity depends highly on the order of the variables and there is no automatic way for determining the most efficient ordering.

Dixon's method is an efficient way for simultaneously eliminating variables from a system of nonhomogeneous equations [KSY94]. The main result reported in Dixon's original paper [Dix08] was a method to obtain the resultant of a set of three *generic bidegree* polynomials. It is a generalization of Cayley's formulation of Bezout's efficient method for computing the resultant of two bivariate polynomials.

Definition 4.3.1. *The $n+1$ polynomials $f_1, \dots, f_{n+1}$ in $x_1, \dots, x_n$ are said to be generic n-degree if there exist nonnegative integers $m_1, \dots, m_n$, such that*

$$\forall 1 \leq j \leq n: \ f_j = \sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} a_{j,i_1,\dots,i_n} x_1^{i_1} \cdots x_n^{i_n}$$

Let $F = \{f_1(x_1, \dots, x_n), \dots, f_{n+1}(x_1, \dots, x_n)\}$ be a set of $n+1$ generic n-degree polynomials in n variables and $d_{\max_i} = \max(\deg_{x_i}(f_1), \dots, \deg_{x_i}(f_n))$ for all $1 \leq i \leq n$. We form the determinant

$$\Delta(x_1, \dots, x_n, y_1, \dots, y_n) = \begin{vmatrix} f_1(x_1, x_2, \dots, x_n) & \cdots & \cdots & f_{n+1}(x_1, x_2, \dots, x_n) \\ f_1(y_1, x_2, \dots, x_n) & \cdots & \cdots & f_{n+1}(y_1, x_2, \dots, x_n) \\ f_1(y_1, y_2, \dots, x_n) & \cdots & \cdots & f_{n+1}(y_1, y_2, \dots, x_n) \\ \cdots & \cdots & \cdots & \cdots \\ f_1(y_1, y_2, \dots, y_n) & \cdots & \cdots & f_{n+1}(y_1, y_2, \dots, y_n) \end{vmatrix}$$

where $y_1, \dots, y_n$ are new variables and $f_i(y_1, \dots, u_k, x_{k+1}, \dots, x_n)$ stands for uniformly replacing x_j by y_j for $1 \leq j \leq k$ in f_i.

If we plug in y_i in Δ, then Δ vanishes, so Δ is divisible by $\prod_{i=1}^{n}(x_i - y_i)$.

Definition 4.3.2. *Let F be a set of $n+1$ generic n-degree polynomials in n variables and $\Delta(x_1, \dots, x_n, y_1, \dots, y_n)$ be the $(n+1) \times (n+1)$ determinant as described above. Then*

$$\delta(x_1, \dots, x_n, y_1, \dots, y_n) = \frac{\Delta(x_1, \dots, x_n, y_1, \dots, y_n)}{(x_1 - y_1) \cdots (x_n - y_n)}$$

is called the Dixon polynomial.

Lemma 4.3.3. *Let F be a set of $n+1$ generic n-degree polynomials in n variables and δ be the Dixon polynomial. Then the following holds:*

1. $\deg_{y_i}(\delta) = ((n+1-i)d_{\max_i}) - 1$.
2. $\deg_{y_i}(\delta) = (i\ d_{\max_i}) - 1$.

Any common zero of F makes the Dixon polynomial vanish no matter what the values of $i_1, \dots, y_n$ are. Hence, all coefficients of the power products of $y_1, \dots, y_n$ in the Dixon polynomial vanish.

Let E' be the set of all polynomials in $x_1, \dots, x_n$ which are coefficients of the power products of $y_1, \dots, y_n$. This set has exactly

$$s := \prod_{i=1}^{n}((n+1-i)d_{\max_i}) \ = \ n! \prod_{j=1}^{n} d_{\max_i}$$

equations (one for each power product of $y_1, \ldots, y_n$), each of which is of degree $(i\ d_{\max_i}) - 1$ in x_i. Also, there are

$$\prod_{i=1}^{n} i d_{\max_i} \ = \ n! \prod_{i=1}^{n} \ = s$$

power products in $x_1, \ldots, x_n$ in the equations in E'. Let D be the $s \times s$ coefficient matrix of E'. Then

$$E' \equiv D \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ \vdots \\ x_n^2 \\ \vdots \\ \prod_{i=1}^{n} x_i^{i d_{\max_i}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

If each powerproduct of $x_1, \ldots, x_n$ (including $x_1^0 \cdots x_n^0 = 1$) is viewed as a new variable v_i, then we get a set E of s homogeneous linear equations in s variables:

$$E \equiv D \begin{pmatrix} v_1 \\ \vdots \\ v_s \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

The $s \times s$ coefficient matrix D of the linear system is called Dixon matrix, its determinant $\det(D)$ is the Dixon resultant.

If a common zero exists for F (say $(\xi_1, \ldots, \xi_n)$) then this is a solution for E' also. Furthermore, this result yields a nontrivial solution for E, namely $v_1 = 1, v_2 = \xi_1, v_3 = \xi_2, \ldots, v_s = \xi_1^{d_{\max_1}-1} \xi_2^{d_{\max_2}-1} \cdots \xi_n^{d_{\max_n}-1}$, implying that $\det(D) = 0$. Hence we get a criterion for the existence of a common solution for a system of $n+1$ generic n-degree polynomial in n variables.

Intuitively speaking, the Dixon resultant is a polynomial in the coefficients of the generic n-degree system viewed as distinct indeterminants that is not identically zero but vanishes for those particular values for the coefficients for which the system has a common zero.

4.3.2 Extending Dixon's Resultant to the General Case

Unfortunately Dixon's resultant does not work for most algebraic and geometric problems. The condition of being generic n-degree cannot be satisfied by most practical problems, the Dixon matrix usually becomes singular or even non-square. As a consequence, Dixon's resultant vanishes identically, without providing any information about the common solutions of equations.

In [Tra97] a generalization of Dixon's resultant is given for polynomials with coefficients in a field and polynomials with coefficients in a polynomial ring. These generalizations no longer require the polynomials to be generic. We restrict ourselves to the case of polynomials with coefficients in a polynomial ring here.

From now on, let $\mathbb{K}$ be a field of arbitrary characteristic and consider the polynomial ring $\mathbb{K}[y_1, \dots, y_m]$.

Definition 4.3.4. *Let $F \subset \mathbb{K}[y_1, \dots, y_m][x_1, \dots, x_n]$ be a system of $n+1$ equations in n variables. F is called regular if and only if for every equation $f \in F$,* $\mathrm{coeff}_{x_1^0 \cdots x_n^0}(f) \neq 0$.

For every regular system F we have the important property that for a non-singular Dixon matrix D, $\det(D) = 0$ always gives a necessary condition for F to have a common zero.

Lemma 4.3.5. *Every system $F = \{f_1, \dots, f_{n+1}\} \subset \mathbb{K}[y_1, \dots, y_m][x_1, \dots, x_n]$ of $n+1$ equations in n variables can be transformed into a regular system F' by using the following linear transformations:*

$$
\begin{aligned}
w &= w \\
x_1 &= x_1' + c_1 w \\
x_2 &= x_2' + c_2 w \\
&\dots \\
x_n &= x_n' + c_n w
\end{aligned}
$$

where w is a new variable, $c_1 \in \mathbb{K}[y_1, \dots, y_m]$ $(1 \leq i \leq n)$ and $f_j(c_1, \dots, c_n) \neq 0$ for $1 \leq j \leq n+1$.

For the following results we always assume that the system F is regular.

Theorem 4.3.6. *Let $F \subset \mathbb{K}[y_1, \dots, y_m][x_1, \dots, x_n]$ be a system of $n+1$ equations in n variables. If the Dixon matrix D is of full column rank, then F has a common zero of for any submatrix D' of the same rank as D, $\det(D') = 0$ holds.*

In case the Dixon matrix is not of full column rank, one has to use full rank decomposition as described in [Tra96]. In that, there exist matrices C and R such that

1. $D = CR$

2. C is of full column rank.

3. R is of full row rank

Theorem 4.3.6 then holds with C in the place of D.

4.3.3 Using the Dixon Resultant in `ssiPlot`

The `ssiPlot` algorithm makes use of the extended Dixon resultant in the critical point computations step in the following way. We are given a set of three trivariate polynomials $F = \{f, g, d\}$ (f and g are the defining equations of the two surfaces and d is the determinant of the Jacobian with respect to x of f and g). In order to compute x-coordinates of the critical points, F is considered to be a subset of $\mathbb{Q}[x][y, z]$. Then the following steps are performed:

1. We compute the extended Dixon resultant h of F. h is a polynomial in x.

2. By the property of the Dixon resultant we know, that the zeros of h can be extended to a common solution for the system F.

3. As we need only isolating intervals for the critical points (in this case, their x-coordinates only), we apply real root isolation to h and return the list of isolating intervals.

The most time critical step in the Dixon resultant computation is the computation of the submatrix D' of the Dixon matrix D, which has the same rank as D and the computation of the determinant of D'. Both operations are done in one step by using a Gaussian elimination algorithm in order to compute a triangular matrix of maximal rank from D. The determinant then is the product of all elements in the main diagonal of the triangular matrix. In Algorithm 4.3.1(Dixon) this computation is described in detail. Our analysis of Algorithm 4.3.1(Dixon)

Algorithm 4.3.1 (Dixon) Computation of the extended Dixon resultant for polynomials with coefficients in a polynomial ring.

Input: $D \in \mathbb{Q}[x]^{s_1 \times s_2}$, the Dixon matrix of full column rank
Output: d, the extended Dixon resultant of D.
1: $d := 1$; $r := 1$.
2: **for** $c = 1$ to s_2 **while** $r \leq s_1$ **do**
3:     Compute the row with index $\geq r$ which has the smallest element in column c and make this row to the rth row (pivoting).
4:     $d := d \cdot d_{rc}$.
5:     **for** $i = r + 1$ to s_2 **do**
6:         $t := \text{simplify}(\frac{d_{ic}}{d_{rc}})$.
7:         **for** $j = c + 1$ to s_1 **do**
8:             $d_{ij} := \text{simplify}(d_{ij} - t \cdot d_{rj})$.
9:         **end for**
10:         $d_{ic} := 0$.
11:     **end for**
12:     $r := r + 1$.
13: **end for**
14: **return**(d).

revealed that the most time consuming part is the simplification operation in step 8 of the algorithm. By this simplification procedure it is guaranteed that the expression is not a rational function but a polynomial.

4.3.4 A Parallel Algorithm for Computing the Extended Dixon Resultant

The Gaussian elimination structure of Algorithm 4.3.1(Dixon) has little possibilities for parallelization attempts. The iterative steps of the outermost loop are not independent from each other, however, the actual elimination step, in which an appropriate multiple of a row are added to all rows below in order to generate zero columns offers a bunch of independent operations of polynomial arithmetic which can perfectly be done in parallel. In order to overcome the inefficiency of *Distributed Maple* when dealing with fine grained task structures, the individual operations are blocked into packages consisting of several elimination operations (see Figure 4.2). The blocking factor is dynamically adapted

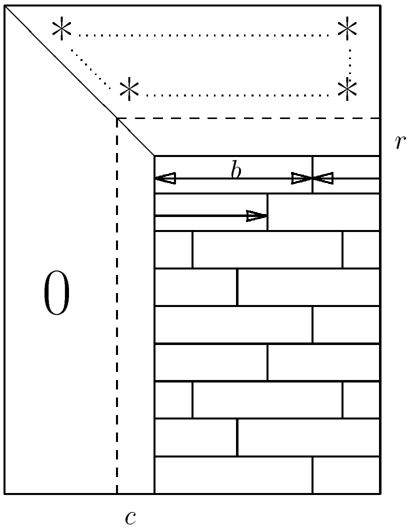

Figure 4.2: Task Blocking in the Gaussian Elimination Step

to an appropriate size taking into consideration the the total number of operations to be performed and the number of available processors. In the sequel we describe the parallelization idea and give a pseudocode algorithm (Algorithm 4.3.2(Parallel Dixon)).

Let $D = (d_{ij}) \in \mathbb{Q}[x]^{s_1 \times s_2}$ be the Dixon matrix, which is of full column rank and p denote the number of processors. The basic structure of the parallel algorithm is equal to the sequential case. So we suppose that in the process of computing a triangular submatrix of maximal rank $r-1$ columns have already been processed and we are about to start generate zeros for the entries d_{ir} $(i > r)$.

1. We compute the block size for the parallel tasks by the following formula: $b := \lfloor \frac{(s_1-r)(s_1-c)}{p} \rfloor$ and set the task counter equal to 0.

2. We iterate over the rows to be processed (starting from $r+1$) and compute the factor t for the current line as $t := \text{simplify}(d_{ic}/d_{rc})$.

3. We iterate over the column entries in the current line starting from $c+1$ (the entries from 1 to c are equal to 0) and do the following:
   - Add the expression $(d_{ij} - td_{rj})$ to the current task block and increase the task counter by 1.
   - If the task counter equal the task block size b, then schedule the current block for execution and reset the task counter to 0.
4. When loop 2 has terminated we have to check if there are pending operations in a block which has not yet reached the task block size. If so, this block is also scheduled for execution.
5. We wait for the results of all blocks and replace the corresponding entries in D by the computed elements.

Algorithm 4.3.2 (Parallel Dixon) Parallel computation of the extended Dixon resultant for polynomials with coefficients in a polynomial ring.

Input: $D \in \mathbb{Q}[x]^{s_1 \times s_2}$, the Dixon matrix of full column rank,
p, the number of processors
Output: d, the extended Dixon resultant.

$d := 1$; $r := 1$; $l := []$.
for $c = 1$ to s_2 **while** $r \leq s_1$ **do**
  Compute the row with index $\geq r$ which has the smallest element in column c and make this row to the rth row (pivoting).
  $tc := 0$; $b := \lfloor \frac{(s_1 - r)(s_1 - c)}{p} \rfloor$.
  $d := d \cdot d_{rc}$.
  for $i = r + 1$ to s_2 **do**
    $t :=$ simplify$(\frac{d_{ic}}{d_{rc}})$.
    for $j = c + 1$ to s_1 **do**
      $l := l \bullet [d_{ij} - t \cdot d_{rj}]$.
      $tc := tc + 1$.
      if $tc = b$ **then**
        Start a simplify-task for the list l.
        $tc := 0$; $l := []$.
      end if
    end for
    $d_{ic} := 0$.
  end for
  if $l \neq []$ **then**
    Start a simplify-task for the list l.
  end if
  Wait for the results of all tasks and update D.
  $r := r + 1$.
end for
return(d).

The simplify-task of Algorithm 4.3.2(Parallel Dixon) consists of a loop which processes over the elements in the list l and applies the simplify function to each element.

4.4 The `dssiPlot` Algorithm

In this section we describe how the time consuming sequential steps of `ssiPlot` can successfully be replaced by parallel algorithms developed in Chapter 3 and this chapter. In contrast to `pacPlot` parallelization affects all three major steps of the sequential algorithm.

Obviously, the various subalgorithms occurring in `ssiPlot` simply have to be replaced by their parallel counterparts. This applies in particular to the computation of the Dixon resultant and the subsequent real root isolation. Investigating initial point computation carefully reveals that this step can easily be done using the ideas of Algorithm 3.2.4 and some minor adaptions to the surface context. In particular, Algorithm 3.2.4 can be used in step 13 of Algorithm 4.2.2(Initial Points), where the isolating intervals for the third coordinates of the initial points are computed. In Algorithm 4.4.1 the parallel approach for initial point computation is presented. Parallelization of step 3 of `ssiPlot`, the generation of the traces for all simple branches of the space curve, is done straightforward by simultaneously generating the plot for each simple branch starting at its initial point.

Algorithm 4.4.1 Parallel computation of the initial points for visualization of the surface-to-surface intersection given by two trivariate polynomials

---

Input: $f, g \in \mathbb{Q}[x, y, z]$ defining the surface-to-surface intersection,
  $x_1 < x_2 < \ldots < x_r$ sequence of x-coordinates of the critical points of the intersection curve.
Output: I, set of initial points for visualization.
  $I := \emptyset$.
  for $i = 1$ to $r - 1$ **do**
    $\xi := (x_i + x_{i+1})/2$.
    Apply `bsolve` to get a set T of triangular systems from $f(\xi, y, z)$ and $g(\xi, y, z)$.
    for all $(p(y), q(y, z)) \in T$ **do**
      $F := \{p_i \mid \prod_i p_i = p\}$
      for all $p \in F$ **do**
        $Y := \{]\eta_1, \eta_2[\}$ set of isolating intervals for the real roots of p with appropriate diameter.
        $r(z) := \mathrm{res}_y(p, q)$
        $Z := \{]\zeta_1, \zeta_2[\}$ set of isolating intervals for the real roots of r with appropriate diameter.
        begin parallel
          For each $]\eta_1, \eta_2[\in Y$ and each $]\zeta_1, \zeta_2[\in Z$ compute the sign of $q(\eta, \zeta_1)$ and $q(\eta, \zeta_2)$, where η is the real root of p represented by the isolating interval $]\eta_1, \eta_2[$. Add $(\xi, \frac{\eta_1+\eta_2}{2}, \frac{\zeta_1+\zeta_2}{3})$ to I if the signs are different or at least one sign is 0.
        end parallel
      end for
    end for
  end for
  return(I).

The structure of the `dssiPlot` algorithm now looks as follows:

1. Compute the critical points of the space curve by the means of the parallel algorithm for the extended Dixon resultant and the parallel real root isolation algorithm presented in section .
2. Compute the initial points for visualization using Algorithm 4.4.1.
3. Generate the traces for all simple branches of the curve in parallel.

Remark 4.4.1. In our examples, the Dixon resultant computed in the first step of `ssiPlot` is a really huge polynomial with large coefficients. Thus, in our parallel implementation of the `dssiPlot` algorithm we have to use a *Distributed Maple* extension which allows file buffering of large messages as this messages do no longer fit into the address space of the Java Virtual Machine.

4.5 Execution Times and Conclusions

We have systematically benchmarked the parallel `neighbGraph` algorithm with the curves used for the sequential profiling (see Section 4.2.2) in two environments consisting of 24 processors each:

- a Silicon Graphics Cray Origin 2000 multiprocessor with 64 R12000@300MHz processors. 24 processors were used;
- a mixed configuration of 4 dual-processor Octanes with 8 R10000@250MHz processors and 16 processors of the Origin multiprocessor. The Origin 2000 and the Octanes are linked by a high bandwith ATM line, while the Octanes are linked via a 10 Mbit Ethernet network.

The cluster environment described in Section 3.6, which involves 16 Linux PCs at various speed turned out to be not suitable for benchmaring this algorithm. The Linux PCs have a significantly low networking performance which slows down execution times tremendously as huge amounts of data have to be communicated among the individual processors.

The following table illustrates the performance of the various benchmarking configurations in the two environments relative to a PIII@450MHz processor. These numbers were measured by a representative Maple benchmark.

Performance (relative to PIII@450MHz)						
Environment	1	2	4	8	16	24
Origin	0.78	1.56	3.12	6.24	12.48	18.72
Mixed	0.58	1.16	2.32	4.64	10.88	17.12

- The tables in Figure 4.3 give an overview of the total parallel execution times of the `dssiPlot` algorithm int the two benchmarking environments using different numbers of processors.
- The tables in Figure 4.6 list the detailed execution times for the parallel subalgorithms in `dssiPlot` for all four examples.

- The first table in Figure 4.5 lists the absolute speedup values $\frac{T_s}{T_n}$, where T_s denotes the the sequential execution time and T_n denotes the parallel execution time using n processors. The first row of diagrams in Figure 4.4[1] visualizes this data.
- The second table in Figure 4.5 presents the scaled speedups $\frac{nT_s}{T_n \sum_{i=1}^{n} p_i}$, where p_i is the relative performance of processor i. These values are visualized in the second row of diagrams in Figure 4.4[1].
- The third table in Figure 4.5 gives the scaled efficiency $\frac{T_s}{T_n \sum_{i=1}^{n} p_i}$, which compares the speedup we actually got to the upper bound of speedup we could have got. The third row of diagrams in Figure 4.4[1] visualizes this data.

Origin						
	1	2	4	8	16	24
Example 1	2058	783	514	332	345	308
Example 2	447	233	180	190	158	154
Example 3	1262	687	396	319	267	228
Example 4	6026	2296	1304	807	615	649

Mix						
	1	2	4	8	16	24
Example 1	3761	1105	647	437	337	270
Example 2	671	309	215	198	173	168
Example 3	2395	851	475	353	284	273
Example 4	12072	3047	1686	1008	671	643

Figure 4.3: Total Execution Times

Analyzing the experimental data leads to the following results:

- Most obviously, the speedups for larger examples are better than for smaller ones. Example 2 is too small to achieve a better speedup than 3.1. For high numbers of processors (16 and 24), the mixed configuration achieves better results than the Origin multiprocessor. This is due to a not yet clarified phenomenon, namely file buffered communication is more efficiently done in the mixed configuration than on the Origin.
- The super-one efficiencies occuring in almost all examples when using up to 8 processors may partly be explained by the finer intervals computed by the speculative real root isolation algorithm, which speeds up sign computations in the initial point computation tremendously. Additionally we assume that using several Maple kernels participating in the computation yields to fewer garbage collection steps than in the sequential version, where all the data has to fit into Maple's memory

[1]The markers + and × denote execution on the Origin and in the mixed environment, respectively

- For the individual subalgorithms the speedups are much higher compared to the total speedup of the parallel algorithm. In Example 4, the Dixon resultant achieves a speedup of 16.9 on the Origin and 16.4 in the mixed configuration using 24 processors. The superlinear speedups for computing the individual branches may again partly be explained by fewer garbage collection steps as there are no algorithmic differences to the sequential version of `ssiPlot`.

The diagrams in Figure 4.7 show the dynamic behavior of the `dssiPlot` algorithm running on 16 processors of the Origin (top diagram) and the mixed configuration (bottom diagram), respectively. Clearly, we see the Dixon resultant computation followed by a real root isolation phase. Both subalgorithms lead to a huge number of small tasks which are scheduled among the processors. In the initial point computation step, tasks are not balanced at all. This is due to the fact that several resultant computations occur, each of them being too small to be computed by the parallel algorithm. At the rightmost end of each diagram we can see a short parallel phase for computing the individual branches.

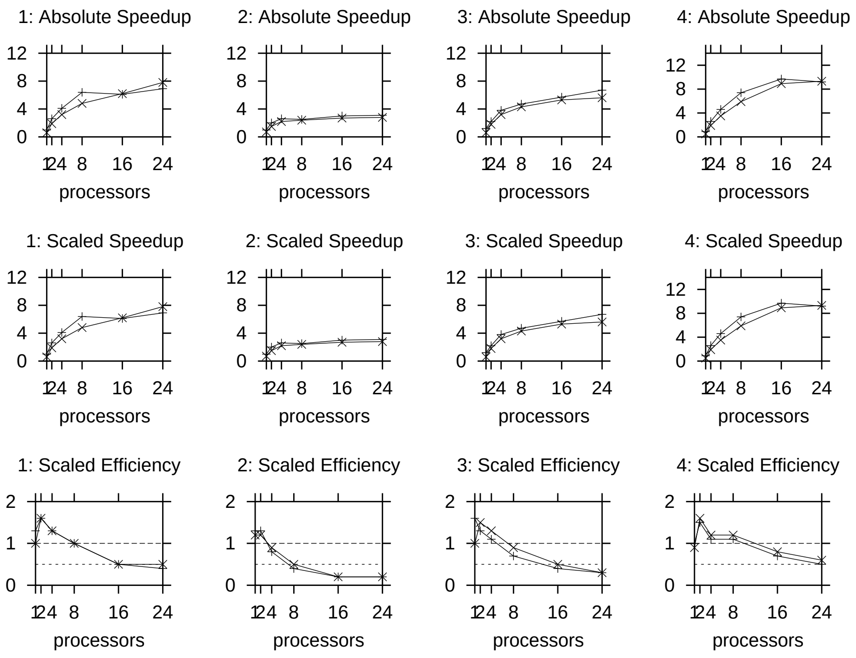

Figure 4.4: Experimental Results

Absolute speedup							
Example	Architecture	1	2	4	8	16	24
1	Origin	1.0	2.6	4.1	6.4	6.1	6.9
	Mix	0.6	1.9	3.2	4.8	6.2	7.8
2	Origin	1.0	2.0	2.6	2.5	3.0	3.1
	Mix	0.7	1.5	2.2	2.4	2.7	2.8
3	Origin	1.2	2.2	3.8	4.7	5.7	6.7
	Mix	0.6	1.8	3.2	4.3	5.3	5.6
4	Origin	0.9	2.6	4.6	7.4	9.7	9.2
	Mix	0.5	1.9	3.5	5.9	8.9	9.3

Scaled speedup							
Example	Architecture	1	2	4	8	16	24
1	Origin	1.3	3.3	5.3	8.3	7.9	9.0
	Mix	1.0	3.2	5.4	8.1	8.6	10.9
2	Origin	1.3	2.6	3.3	3.2	3.9	4.0
	Mix	1.2	2.5	3.7	4.1	3.8	3.9
3	Origin	1.6	2.6	4.5	5.6	6.8	8.0
	Mix	1.0	3.1	5.5	8.8	11.6	11.0
4	Origin	1.0	3.1	5.5	8.8	11.6	11.0
	Mix	0.9	3.2	5.9	10.0	12.5	13.0

Scaled efficiency							
Example	Architecture	1	2	4	8	16	24
1	Origin	1.3	1.6	1.3	1.0	0.5	0.4
	Mix	1.0	1.6	1.3	1.0	0.5	0.5
2	Origin	1.3	1.3	0.8	0.4	0.2	0.2
	Mix	1.2	1.2	0.9	0.5	0.2	0.2
3	Origin	1.6	1.3	1.1	0.7	0.4	0.3
	Mix	1.0	1.5	1.3	0.9	0.5	0.3
4	Origin	1.0	1.5	1.3	1.1	0.7	0.5
	Mix	0.9	1.6	1.4	1.2	0.8	0.6

Figure 4.5: Absolute Speedups, Scaled Speedups, and Scaled Efficiency

Example 1, detailed execution times [1]							
Environment	Operation	1	2	4	8	16	24
Origin	Dixon	1748	584	358	189	208	173
	Realroot	135	49	35	22	26	26
	Initial points	137	123	96	93	88	86
	Branches	10	5	3	2	2	2
Mix	Dixon	3144	803	425	267	180	134
	Realroot	377	72	46	32	39	23
	Initial points	185	193	143	103	89	84
	Branches	17	5	3	3	2	2

Example 2, detailed execution times[1]							
Environment	Operation	1	2	4	8	16	24
Origin	Dixon	340	143	102	80	67	67
	Realroot	33	24	17	33	11	11
	Initial points	45	32	33	60	60	60
	Branches	12	8	5	3	2	3
Mix	Dixon	492	195	119	94	76	73
	Realroot	70	32	22	18	15	12
	Initial points	64	39	37	62	60	59
	Branches	19	12	7	3	3	3

Example 3, detailed execution times[1]							
Environment	Operation	1	2	4	8	16	24
Origin	Dixon	985	387	215	169	105	95
	Realroot	181	79	51	32	52	29
	Initial points	46	182	91	85	83	80
	Branches	10	5	4	3	2	2
Mix	Dixon	1758	515	277	179	126	123
	Realroot	498	106	58	44	40	35
	Initial points	65	172	93	88	83	82
	Branches	16	8	4	3	1	2

Example 4, detailed execution times[1]							
Environment	Operation	1	2	4	8	16	24
Origin	Dixon	4326	1169	624	401	266	277
	Realroot	385	126	90	83	74	103
	Initial points	1166	940	553	290	247	245
	Branches	67	14	5	3	2	2
Mix	Dixon	9170	1690	890	519	303	285
	Realroot	1254	188	147	94	76	73
	Initial points	1409	1102	591	353	259	244
	Branches	116	25	7	4	3	2

[a]Parallel execution on one processor uses the sequential real root isolation algorithm.

Figure 4.6: Detailed Execution Times

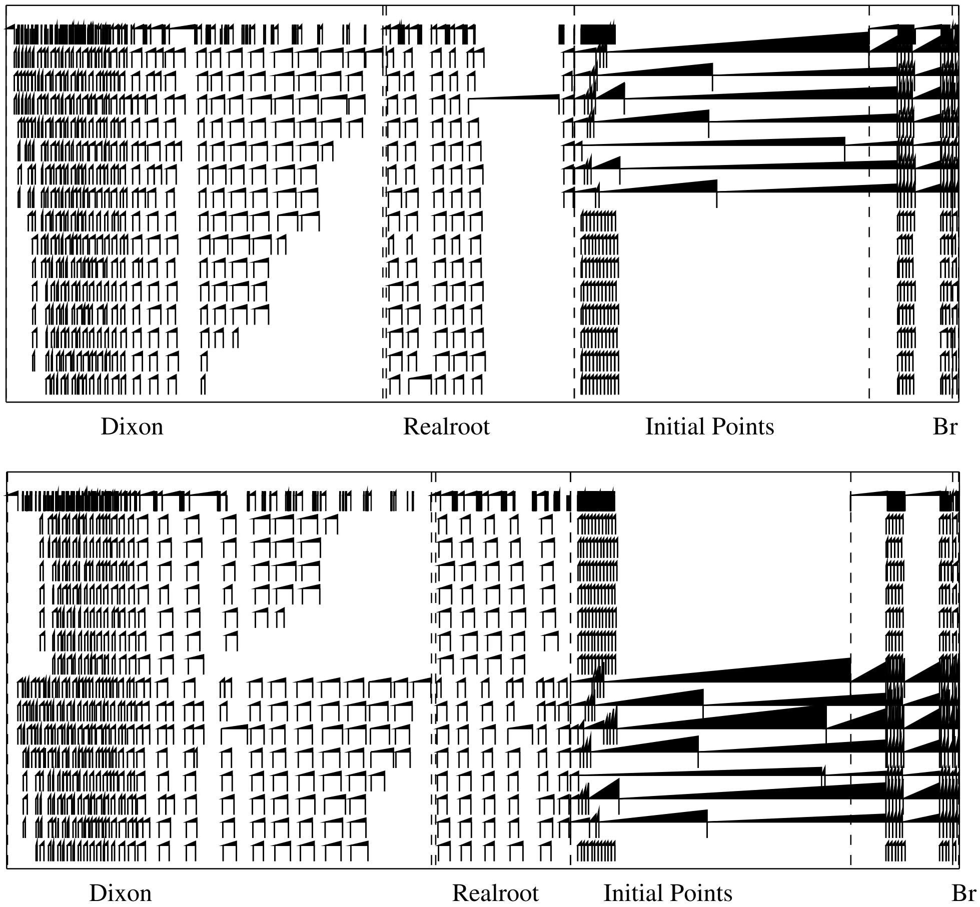

Figure 4.7: Dynamic Behavior

Chapter 5

Resolution of Singularities

In this chapter we describe a method for resolving the singularities of an algebraic curve using quadratic transformations and the concepts of neighboring points. Resolution of singularities plays an important role in the parametrization problem and for computing adjoints. Here, we emphasize on the parametrization problem context.

The structure of this chapter is as follows: In Section 5.1 we present basic mathematical facts about singular points of a curve. In Section 5.2 we describe the CASA-algorithm for computing the singular points of a curve and give a sequential improvement reducing computing time by at least an order of magnitude. In Section 5.3 we describe the effect of quadratic transformations in singular points and review the concept of neighboring singularities. In Section 5.4 we analyze the most time consuming parts of the algorithm `neighbGraph`. In Section 5.5 we give a parallel algorithm for computing singularities of the form $(x : 1 : 0)$, which accounts for most of the computing time in determining the neighborhood graph. In Section 5.6 we give speedup diagrams and performance evaluations of the parallel algorithm and analyze its dynamic behavior.

5.1 Singularities of a Plane Algebraic Curve

In this section we review some definitions and results about singular points of a plane algebraic curve $\mathcal{C}$ and their role in the parametrization problem. For more details and proofs we refer the reader to [Wal78, Win96, SW90]. In the sequel, we write $\mathbb{A}^2(\mathbb{K})$ and $\mathbb{P}^2(\mathbb{K})$ for the affine plane and the projective plane, respectively, over the field $\mathbb{K}$. When $\mathbb{K}$ is clear from the context, we simply write $\mathbb{A}^2$ and $\mathbb{P}^2$, respectively.

An *affine plane algebraic curve* $\mathcal{C}$ in $\mathbb{A}^2$ is the set of zeros of a squarefree polynomial $f(x, y) \in \mathbb{K}[x, y]$, i.e.

$$\mathcal{C} := \{(\xi, \eta) \mid f(\xi, \eta) = 0,\ (\xi, \eta) \in \mathbb{A}^2\}.$$

f is called a defining polynomial of $\mathcal{C}$. The curve $\mathcal{C}$ is of degree d, iff d is the degree of the defining polynomial.
A *projective plane algebraic curve* $\mathcal{C}^*$ in $\mathbb{P}^2$ is the set of zeros of a squarefree homogeneous polynomial $F(x, y, z) \in \mathbb{K}[x, y, z]$, i.e.

$$\mathcal{C}^* := \{(\xi : \eta : \zeta) \mid F(\xi, \eta, \zeta) = 0,\ (\xi : \eta : \zeta) \in \mathbb{P}^2\}.$$

F is called a defining polynomial of the curve. The degree of the projective curve $\mathcal{C}^*$ is defined analogously to the affine case.

For simplicity, we will write the defining polynomial of the curve $\mathcal{C}$ (or $\mathcal{C}^*$, respectively), meaning the polynomial with smallest degree defining $\mathcal{C}$ (or $\mathcal{C}^*$, respectively).

Definition 5.1.1. 1. *Let $\mathcal{C}$ be a curve in $\mathbb{A}^2$ defined by the polynomial $f(x,y)$. Let $P = (\xi, \eta)$ be a point on $\mathcal{C}$. P is a* simple point *on $\mathcal{C}$, iff $\frac{\partial f}{\partial x}(P) \neq 0$ or $\frac{\partial f}{\partial y}(P) \neq 0$. In this case, the* tangent *to $\mathcal{C}$ at P is uniquely defined as*

$$\frac{\partial f}{\partial x}(P)(x-\xi) + \frac{\partial f}{\partial y}(P)(y-\eta) = 0.$$

If P is not simple, i.e. both partial derivatives vanish at P, then P is called a multiple point *or* singularity *on $\mathcal{C}$. Let m be such that for all i,j with $i+j < m$ the partial derivatives $\frac{\partial^{i+j} f}{\partial x^i \partial y^j}$ vanish at P but not all partial derivatives of order m vanish. Then m is called the* multiplicity *of P on $\mathcal{C}$ and P is called an m-fold point. In this case, the polynomial*

$$\sum_{i+j=m} \frac{1}{i!j!} \frac{\partial^m f}{\partial x^i \partial y^j}(P)(x-\xi)^i(y-\eta)^j$$

factors completely into linear factors, the tangents *of $\mathcal{C}$ at P.*

2. *Let $\mathcal{C}^*$ be a curve in $\mathbb{P}^2$ defined by the homogeneous polynomial $F(x,y,z)$. Let $P = (\xi : \eta : \zeta)$ be a point on $\mathcal{C}^*$. W.l.o.g. $\zeta = 1$ (for the other coordinates proceed analogously). P is a simple or multiple point on $\mathcal{C}^*$ depending on whether $Q = (\xi, \eta)$ is a simple or multiple point on the affine curve defined by $F(x,y,1)$.*

An m-fold point P is an *ordinary* singular point iff the m tangents to $\mathcal{C}$ at P are distinct. Otherwise P is called *non-ordinary*. The property of an m-fold point of being ordinary or non-ordinary is called the *character* of P.

Definition 5.1.2. *The irreducible affine curve $\mathcal{C}$ in $\mathbb{A}^2$ defined by the irreducible polynomial $f(x,y)$ is called* rational *(or* parametrizable*) iff there are rational functions $\phi(t), \chi(t) \in \mathbb{K}(t)$ such that*

1. *for almost all $t_0 \in \mathbb{K}$, $(\phi(t_0), \chi(t_0))$ is a point on $\mathcal{C}$ and*
2. *for almost every point (ξ_0, η_0) in $\mathcal{C}$ there exists a $t_0 \in \mathbb{K}$ such that $(\xi_0, \eta_0) = (\phi(t_0), \chi(t_0))$.*

In this case, (ϕ, χ) is called a (rational) parametrization *of $\mathcal{C}$.*

The rationality problem for an affine curve is equivalent to the rationality problem for the associated projective curve. A parametrization of a curve $\mathcal{C}$ can always be computed from a parametrization of the associated projective curve $\mathcal{C}^*$ and vice versa.

Example 5.1.3. Consider the tacnode curve defined by the polynomial

$$f(x,y) := 2x^4 - 3x^2y + y^4 - 2y^3 + y^2 = 0$$

in $\mathbb{A}^2(\mathbb{C})$. It can be parametrized as

$$\phi(t) = \frac{t^3 - 6t^2 + 9t - 2}{2t^4 - 16t^3 + 40t^2 - 32t + 9}, \ \chi(t) = \frac{t^2 - 4t + 4}{2t^4 - 16t^3 + 40t^2 - 32t + 9}$$

That is, the points of the tacnode curve are exactly the values of $(\phi(t), \chi(t))$ for $t \in \mathbb{C}$ except for finitely many exceptions.

A good measure for the complexity of an algebraic curve is the genus of the curve. It turns out that exactly curves of genus 0 are rational. We first deal with ordinary singularities only and continue with describing a problem occurring when a curve also has non-ordinary singularities.

Definition 5.1.4. *Let $\mathcal{C}^*$ be an irreducible curve of degree d in $\mathbb{P}^2$ with singular points $P_1, \ldots, P_n$, having multiplicities $r_1, \ldots, r_n$, respectively. Let all these singular points be ordinary. Then the* genus *of $\mathcal{C}^*$ is defined as*

$$\text{genus}(\mathcal{C}^*) := \frac{1}{2}\left[(d-1)(d-2) - \sum_{i=1}^{n} r_i(r_i - 1)\right]$$

The curves of genus 0 are those curves which achieve the theoretically highest possible count of singularities. Moreover, these are exactly those curves, which can be parametrized. For the following theorem, we refer to [Wal78]

Theorem 5.1.5. *An algebraic curve $\mathcal{C}$ (having only ordinary singularities) is rationally parametrizable if and only if* $\text{genus}(\mathcal{C}) = 0$.

We return to Example 5.1.3 again. We have given a rational parametrization of the tacnode curve. The curve has two double points at $(0:0:1)$ and $(0:1:1)$. Analyzing the character of the double points more carefully reveals that $(0:0:1)$ is a non-ordinary double point. This is no contradiction to Theorem 5.1.5, as this theorem requires ordinary singularities only. For defining the genus f a curve with non-ordinary singularities we have to count these singularities properly. In the subsequent sections we describe one possible approach to achieving such a count.

In the case of the tacnode curve, these considerations suggest that we take the non-ordinary singularity at the origin as two "coincident" or "neighboring" double points, so that we can still apply Theorem 5.1.5 in some extended sense. In Section 5.3 we will see how the application of certain transformations to the curve pulls apart these neighboring points and allows an extended way of counting the multiplicity of non-ordinary singularities such that Theorem 5.1.5 holds without any modifications.

5.2 A Sequential Improvement of the Algorithm `singularities`

In the course of analyzing CASA's functions dealing with computation of singularities, we discovered that the `singularities` algorithm, which computes the singular points of a plane algebraic curve, is extremely time consuming when dealing with polynomials having a high number of terms (see Table 5.1). By a

(Sub-) Algorithm	Example 1	Example 2	Example 3	Example 4
`affine sing.`	100	25	1495	208

Table 5.1: Profiling the `singularities` algorithm (all timings in seconds)

couple of profiling executions, we found out that the `bsolve` algorithm which is used for solving the polynomial system

$$\begin{aligned} f(x,y) &= 0 \\ \frac{\partial f}{\partial x}(x,y) &= 0 \\ \frac{\partial f}{\partial y}(x,y) &= 0 \end{aligned}$$

is the most time consuming part. It seems to be a hard task to parallelize `bsolve` as it uses a series of primitive polynomial remainder sequences each of which depends on the ones computed before, in order to triangulize the system. Instead of thinking about a parallel `bsolve` algorithm we use resultant and gcd computations in order to compute the solutions of the polynomial system defined by f and its partial derivatives.

In the sequel, let $\mathrm{gsfd}(f)$ denote the greatest squarefree divisor of the polynomial f. We are given a bivariate polynomial $f(x,y)$ defining the affine curve $\mathbb{C}$ and we want to compute all affine singular points of the curve neglecting multiplicities. For this, we have to compute the solutions of the polynomial system

$$\begin{aligned} f(x,y) &= 0 \\ \frac{\partial f}{\partial x}(x,y) &= 0 \\ \frac{\partial f}{\partial y}(x,y) &= 0 \end{aligned}$$

We proceed as follows:

1. Let $r_1 := \mathrm{res}_y(f, \frac{\partial f}{\partial x})$, $r_2 := \mathrm{res}_y(f, \frac{\partial f}{\partial y})$ and $g(x) := \mathrm{gsfd}(r_1, r_2)$.
2. Factor g over $\mathbb{Q}$ giving a set of irreducible factors $G := \{g_i | \prod_i g_i = g\}$.
3. For each factor $g_i \in G$ do the following:
   (a) Let ξ be a root defined by g_i.
   (b) $h(y) := \mathrm{gsfd}(\frac{\partial f}{\partial y}(\xi, y))$.
   (c) Factor h over $\mathbb{Q}$ giving a set of factors $H := \{h_i | \prod_i h_i = h\}$.
   (d) For each factor h_i in H, let η denote the root defined by h_i and check, if (ξ, η) is a common solution of the system.

Profiling this approach reveals that it is superior to `bsolve`. In Table 5.2 we summarize execution times substantiating that this idea is more efficient.

(Sub-) Algorithm	Example 1	Example 2	Example 3	Example 4
`affine sing.`	1.7	1.2	3.2	4.2

Table 5.2: Profiling the improved `singularities` algorithm (in seconds)

5.3 The Neighborhood Graph of a Plane Algebraic Curve

5.3.1 Mathematical Foundations

We consider the projective plane $\mathbb{P}^2$ and define a coordinate transformation T as follows:

$$x' := yz \qquad y' := xz \qquad z' := xy$$

Both, $(x : y : z)$ and $(x' : y' : z')$ are coordinates of a point in $\mathbb{P}^2$ in two different coordinate systems. The transformation is called *quadratic transformation* or *Cremona transformation* and has the following properties:

1. Each point of $\mathbb{P}^2$ with exception of $(1 : 0 : 0)$, $(0 : 1 : 0)$ and $(0 : 0 : 1)$ is again transformed into a unique point of $\mathbb{P}^2$. The three exceptional points are called *fundamental points* of the quadratic transformation.

2. Any non-fundamental point on the lines $x = 0$, $y = 0$ and $z = 0$ is transformed to $(1 : 0 : 0)$, $(0 : 1 : 0)$ and $(0 : 0 : 1)$ respectively. These three lines are called *irregular lines* of the quadratic transformation.

3. If a point $(x : y : z)$ is not on an irregular line, then the transformed point $(yz : xz : xy)$ is not on an irregular line in the transformed coordinate system.

Now we study the action of a quadratic transformation on an irreducible projective curve $\mathcal{C}^*$.

Definition 5.3.1. *Let $\mathcal{C}^*$ be an irreducible algebraic curve defined by the homogeneous polynomial $F(x, y, z)$.*

1. *The polynomial $G(x, y, z) = F(yz, xz, xz)$ is called the* algebraic transformed *of F.*

2. *The* quadratic transform *of F is defined as the irreducible factor of G which is not a fundamental line.*
   We denote it by F' and say, the curve $\mathcal{C}'^*$ is the quadratic transform of $\mathcal{C}^*$.

The importance of quadratic transformations stems from the fact that by a finite sequence of quadratic transformations and coordinate changes (i.e. moving a point to the origin) the singularities of an irreducible plane curve can be resolved. In Figure 5.1 we see how the singularities of the tacnode curve are transformed by an application of a quadratic transformation. The ordinary double-point at $(0 : 1 : 1)$ is resolved into two ordinary points and the non-ordinary singularity at $(0 : 0 : 1)$ is resolved too. That means, the curve can

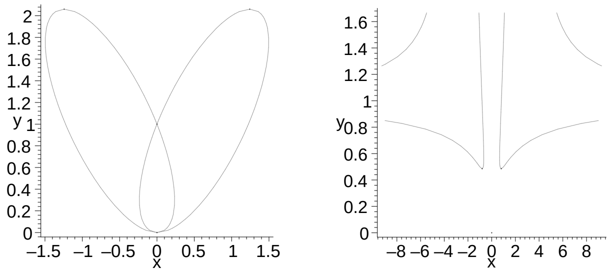

Figure 5.1: The Tacnode Curve and its Transformed Version

be transformed into another one having only ordinary singularities. Moreover, the rationality of the irreducible curve is invariant under these transformations. The following theorem states how the singularities of a curve are affected by quadratic transformations ([Wal78]).

Theorem 5.3.2. *Let $\mathcal{C}^*$ be a curve of degree d defined by the homogeneous polynomial F and having $(1:0:0)$, $(0:1:0)$ and $(0:0:1)$ as points of multiplicity r_1, r_2 and r_3 respectively ($r_i \geq 0$). Let F' be the quadratic transform of F and $\mathcal{C}^{'*}$ the curve defined by F'. Then, if no tangent at any of these points is an irregular line, the following holds:*

1. *The degree of F' is $2d-r_1-r_2-r_3$ and $F'(x,y,z) = F(yz,xz,xy)/x^{r_1}y^{r_2}z^{r_3}$. Furthermore, if $F(x,y,z) = f_d(x,y) + \cdots + f_{r_r}(x,y)z^{d-r_3}$, then $F' = x^{d-r_3-r_1}y^{d-r_3-r_2}f_{r_3} + \cdots + z^{d-r_3-1}x^{1-r_1}y^{1-r_2}f_{d-1} + z^{d-r_3}x^{-r_1}y^{-r_2}f_d$.*

2. *There is a one to one correspondence, preserving multiplicities, between the tangents to $\mathcal{C}^*$ at $(1:0:0)$, $(0:1:0)$ and $(0:0:1)$ and the non-fundamental intersections of $\mathcal{C}^{'*}$ with the irregular lines $x=0$, $y=0$ and $z=0$, respectively.*

3. *An r-fold point of $\mathcal{C}^*$ not on irregular line is transfered into an r-fold point on $\mathcal{C}^{'*}$, and the tangents at these points correspond in multiplicity. In particular, the character of the r-fold point is preserved.*

4. *$\mathcal{C}^{'*}$ has multiplicity $d-r_2-r_3$, $d-r_1-r_3$, and $d-r_1-r_3$ at $(1:0:0)$, $(0:1:0)$ and $(0:0:1)$, respectively, the tangents being distinct from the irregular lines corresponding to the non-fundamental intersections of $\mathcal{C}^*$ with $x=0$, $y=0$ and $z=0$, respectively.*

An outline of a method for obtaining this sequence of quadratic transformations resolving the singularities of a irreducible curve $\mathcal{C}^*$ looks as follows.

1. Choose a non-ordinary singularity of $\mathcal{C}^*$ and make a change of coordinates such that the singularity is moved to $(0:0:1)$, none of its tangents is a fundamental line and no other fundamental point is singular.

2. Apply the quadratic transformation to $\mathcal{C}^*$, getting the transform curve $\mathcal{C}^{'*}$.

3. Check, whether there exists a non-fundamental intersection of $\mathcal{C}'^*$ and $z = 0$ being a non-ordinary singularity. If this is the case, apply (1) and (2) to $\mathcal{C}'^*$ and this non-ordinary singular intersection point. Otherwise, choose any other non-ordinary singularity and repeat the process until there are no non-ordinary singularities left.

This methods selects a particular coordinate system and a particular order of how the singularities are resolved. Independent of these selections, the method always achieves an irreducible curve having only ordinary singularities in a finite number of steps. However, the rationality problem can be solved in a computationally more efficient way. For this purpose we introduce the concept of neighboring points.

Let $\mathcal{C}^*$ be the irreducible curve of degree d defined by $F(x,y,z)$ and $T = (T_1, \dots, T_n)$ a finite sequence of quadratic transformations together with a suitable change of coordinates reducing $\mathcal{C}^*$ to a curve which has only ordinary singularities. T generates the sequence of irreducible curves

$$\mathcal{C}^* = \mathcal{C}_0^* \xrightarrow{T_1} \mathcal{C}_1^* \xrightarrow{T_2} \cdots \xrightarrow{T_n} \mathcal{C}_n^*,$$

where $\mathcal{C}_{i+1}^*$ is the quadratic transformation obtained from $\mathcal{C}_i^*$ by T_{i+1} for $0 \leq i \leq n-1$.

Definition 5.3.3. *Let P be an r-fold point on $\mathcal{C}^*$ and suppose that during the process described by T this point has not yet been moved to a fundamental point till the action of the i-th quadratic transformation.*

1. *The* first neighborhood *of P with respect to T is defined as the set of all non-fundamental intersections of $\mathcal{C}_{i+1}^*$ with the irregular line $z = 0$, assuming that P was moved to $(0 : 0 : 1)$ by the according change of coordinates.*
2. *The points in the first neighborhood of P with respect to T are called* neighboring points of P at its first neighborhood.

If P' is a neighboring point of P at its first neighborhood, then multiplicity and character of P' are defined as multiplicity and character of P' as a point on $\mathcal{C}_{i+1}^*$. Similarly, if $\{P_1', \dots, P_s'\}$ is the first neighborhood of P w.r.t T, we get the second neighborhood of P with respect to T as the union of the first neighborhoods of P_k', $k = 1, \dots, s$. The points in the second neighborhood of P w.r.t T are called the *neighboring points of P at its second neighborhood.* Multiplicity and character of points in the second neighborhood are defined analogously to those of the points in the first neighborhood, but one has to keep in mind that in general not all points are lying on the same curve. In general, we call any point in one of the neighborhoods of P a *neighboring point* of P. The neighboring points of P with multiplicity higher than 1 are called the *singular neighboring points* of P

Definition 5.3.4. *Let P be a singular point on $\mathcal{C}^*$, T the sequence of quadratic transformations from above.*

1. *The* neighborhood tree *of P w.r.t $\mathcal{C}^*$ and T has P as its root and the neighborhood trees of the singular neighboring points in the first neighborhood of P as its subtrees.*

2. *The* neighborhood graph *of $\mathcal{C}^*$ w.r.t T is defined as the collection of all neighborhood trees of singular points of $\mathcal{C}^*$ w.r.t. T.*

The neighboring points of simple points are always simple points, and if P is an ordinary r-fold point its first neighborhood contains exactly r simple points. Therefore, whenever a neighborhood tree contains an ordinary singularity, the associated branch of the tree terminates. Consequently, the neighborhood graph of any curve is finite. The following theorem shows how to decide rationality of a curve with the help of neighborhood graphs ([SW90]).

Theorem 5.3.5. *Let $\mathcal{C}^*$ be an irreducible projective curve, $T = (T_1, \dots, T_n)$ be a sequence of quadratic transformations reducing $\mathcal{C}^*$ and $\mathcal{C}^* = \mathcal{C}_0^*, \dots, \mathcal{C}_n^*$ the sequence of projective curves generated by T. Let d_i denote the degree of $\mathcal{C}_i^*$, S_i the set of singularities of $\mathcal{C}_i^*$ and N_i the neighborhood graph of $\mathcal{C}_i^*$.*
Let r_P denote the multiplicity of a point in S_i or N_i, respectively.

1. *$\mathcal{C}^*$ is rational, if and only if $(d_n-1)(d_n-2) = \sum_{P\in S_n} r_P(r_P-1)$.*

2. *For every i, $0 \le i \le n$,*

$$(d_i-1)(d_i-2) = \sum_{P\in N_i} r_P(r_P-1) = (d_{i+1}-1)(d_{i+1}-2) = \sum_{P\in N_{i+1}} r_P(r_P-1)$$

3. *$\mathcal{C}^*$ is rational, if and only if $(d_0-1)(d_0-2) = \sum_{P\in N_0} r_P(r_P-1)$.*

In general we do not compute the whole sequence of transformed curves of the given curve $\mathcal{C}^*$. Instead we act in the following way. Let $\{P_1, \dots, P_s\}$ be the set of all non-ordinary singularities of the curve $\mathcal{C}^*$ of degree d. For every P_k there always exists a sequence of transformations $T(P_k) = (T_{1,k}, \dots, T_{n_k,k})$ reducing $\mathcal{C}^*$ to a curve having only ordinary singularities and such that P_k is moved to a fundamental point by the action of $T_{1,k}$. Then, for every P_k we only compute the sequence $T(P_k)$ till all the neighboring points of P_k w.r.t $T(P_k)$ have been determined, that is, till another $P_{k'}$ is moved to a fundamental point. We denote this sequence by $T^*(P_k) = (T_{1,k}, \dots, T_{r_k,k})$, $r_k \le n_k$ and it generates the sequence

$$\mathcal{C}^* \xrightarrow{T_{1,k}} \mathcal{C}_1^*(P_k) \xrightarrow{T_{2,k}} \cdots \xrightarrow{T_{r_k,k}} \mathcal{C}_{r_k}^*(P_k),$$

where in general, $\mathcal{C}_{r_k}^*(P_k)$ can have non-ordinary singularities, but these are not singular neighboring points of P_k. At the end of this process, we have

$$\begin{array}{c} \mathcal{C}^* \longrightarrow \mathcal{C}_1^*(P_1) \longrightarrow \cdots \longrightarrow \mathcal{C}_{r_1}^*(P_1), \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ \mathcal{C}^* \longrightarrow \mathcal{C}_s^*(P_s) \longrightarrow \cdots \longrightarrow \mathcal{C}_{r_s}^*(P_s). \end{array}$$

Lemma 5.3.6. *Let $P_1, \dots, P_s$ be the singularities of the projective curve $\mathcal{C}^*$ of degree d. Let $S = \{P_1, \dots, P_s\} \cup N(P_1) \cup \ldots \cup N(P_s)$, where $N(P_k)$ is the set of all neighboring singularities of P_k w.r.t. $T^*(P_k)$ as above. For every $P \in S$ let r_P denote the multiplicity of P.*
Then $\mathcal{C}^*$ is rational if and only if $(d-1)(d-2) = \sum_{P\in S} r_P(r_P-1)$.

5.3.2 The `neighbGraph` Algorithm

In CASA, the neighborhood graph of a plane projective curve can be computed by the means of the `neighbGraph` function. Additionally, this function also returns the changes of coordinates for every step, such that no tangent at $(0 : 0 : 1)$ is an irregular line and no other fundamental point is on the curve. The neighborhood graph is represented by the following list structure:

$$
\begin{array}{lll}
N := [& [m_1, & (x_1 : y_1 : z_1), t_1, [\\
 & & [m_{11}, (x_{11} : y_{11} : z_{11}), t_{11}, [\ldots]], \\
 & & [m_{12}, (x_{12} : y_{12} : z_{12}), t_{12}, [\ldots]], \\
 & & \ldots\ldots\ldots\ldots\ldots\ldots \\
 & & [m_{1r_1}, (x_{1r_1} : y_{1r_1} : z_{1r_1}), t_{1r_1}, [\ldots]]], \\
 & [m_2, & (x_2 : y_2 : z_2), t_2, [\\
 & & [m_{21}, (x_{21} : y_{21} : z_{21}), t_{21}, [\ldots]], \\
 & & [m_{22}, (x_{22} : y_{22} : z_{22}), t_{22}, [\ldots]], \\
 & & \ldots\ldots\ldots\ldots\ldots\ldots \\
 & & [m_{2r_2}, (x_{2r_2} : y_{2r_2} : z_{2r_2}), t_{2r_2}, [\ldots]]], \\
 & \ldots & \ldots\ldots\ldots\ldots\ldots\ldots \\
 & [m_s, & (x_s : y_s : z_s), t_s, [\\
 & & [m_{s1}, (x_{s1} : y_{s1} : z_{s1}), t_{s1}, [\ldots]], \\
 & & [m_{s2}, (x_{s2} : y_{s2} : z_{s2}), t_{s2}, [\ldots]], \\
 & & \ldots\ldots\ldots\ldots\ldots\ldots \\
 & & [m_{sr_s}, (x_{sr_s} : y_{sr_s} : z_{sr_s}), t_{sr_s}, [\ldots]]] \\
 ], & &
\end{array}
$$

where m_i is the multiplicity of the singularity $(x_i : y_i : z_i)$ and t_i is the linear coordinate transformation setting up an appropriate coordinate system. Each singularity of the initial curve is followed by the list of its neighboring points at its first neighborhood, which themselves are followed by the list of neighboring points at their first neighborhood (the second neighborhood of the singularities of the initial curve), etc. A terminal node of the tree is represented by the empty list []. Algorithm 5.3.1(Neighborhood) shows how the neighborhood graph of a plane projective curve is computed.

Example 5.3.7. Consider the cusp, defined by the polynomial $y^2 - x^5$. This curve has a non-ordinary singularity at $(0 : 0 : 1)$ and $(-1 : 1 : 0)$, its neighborhood graph looks as follows:

$$
\begin{array}{lll}
N := [& [2, & (0 : 0 : 1), \{x = x + y, y = y - x\}, [\\
 & & [2, (1 : 1 : 0), \{\}, []]] \\
 &], & \\
 & [3, & (-1 : 1 : 0), \{x = x + y, y = y - x\}, [\\
 & & [2, (2 : 1 : 0), \{x = x + 2y, y = y - 2x\}, []]] \\
 &] & \\
 ] & &
\end{array}
$$

In order to compute the class of singularities of the form $(x : 1 : 0)$ of a projective curve $\mathcal{C}^*$ defined by the homogeneous polynomial $F(x, y, z)$ we pursue the following strategy.

Algorithm 5.3.1 (Neighborhood) Computation of the neighborhood graph of a plane projective curve $\mathcal{C}^*$ represented by the homogeneous polynomial $F(x, y, z)$

Input: $F(x, y, z)$, homogeneous polynomial defining the projective curve $\mathcal{C}^*$,
$S = \{[m, (x : y : z)\}$, set of singularities of F

Output: N, the neighborhood graph of $\mathcal{C}^*$.

$N := []$.
for $[m, (x : y : z)] \in S$ **do**
  Move $(x : y : z)$ to $(0 : 0 : 1)$.
  Choose $n \in \mathbb{N}_0$ such that by the linear change of coordinates $t := \{x = x + ny, y = y - nx\}$ no tangent at $(0 : 0 : 1)$ is an irregular line and no fundamental point is on the curve.
  $F_T := F(yz, xz, xy)$.
  if $\text{ldeg}_z(F_T) > m$ **then**
    $F_T := F_T / z^m$.
    Let S' be the set of singularities of F_T the form $(x : 1 : 0)$ together with their multiplicities.
    $S' := S' - \{[m, (0 : 1 : 0)]\}$.
    Let N'' be the result of the recursive application of Algorithm 5.3.1(Neighborhood) to F_T and S'.
    $N' := [m, (x : y : z), t, N'']$.
  else
    $N' := [m, (x : y : z), t, []]$.
  end if
  $N := N \bullet N'$.
end for
return(N).

1. Dehomogenize F w.r.t. y.

2. Compute the partial derivatives of F w.r.t x and z and substitute 0 for z in all derivatives and F itself, yielding 3 univariate polynomials in x.

3. If $(x:1:0)$ is a singularity of F it must be a root of the three polynomials and therefore also a root of their gcd.

This process is also applied to higher order partial derivatives in order to get singularities of higher multiplicity. In Algorithm 5.3.2(Infsings) we present the pseudocode for computing the singularities of type $(x:1:0)$.

Algorithm 5.3.2 (Infsings) Computing singularities of the form $(x:1:0)$ of a projective curve $\mathcal{C}^*$

Input: $F(x,y,z)$, a homogeneous polynomial defining the curve $\mathcal{C}^*$.
Output: $S = \{[m,(x:1:0)]\}$, set of singularities on the projective line $(x:1:0)$.

$i := 0$.
$d_0(x,z) := F(x,1,z)$.
$b_0(x) := \mathrm{gsfd}(d_0(x,0))$.
repeat
    $d_{i+1} := \frac{\partial d_i(x,z)}{\partial z}(x,z)$.
    for all $0 \le j \le i$ **do** $d_j(x,z) := \frac{\partial d_j(x,z)}{\partial x}(x,z)$.
    $i := i+1$.
    $l := \gcd(d_1(x,0),\dots,d_i(x,0))$.
    if $\deg(l) \neq 0$ **then**
        $b_i(x) := \gcd(b_{i-1}(x), l)$.
    end if
until $\deg(l) = 0$
$b_i(x) := 1$.
$S := \emptyset$.
for $j = 1$ to $i-1$ **do**
    $p := b_j / b_{j+1}$.
    $P := \{p_k \mid \prod_k p_k = p\}$.
    for all $p_k \in P$ **do**
        if $\deg(p_i) > 0$ **then**
            $S := S \cup \{[j+1,(\alpha:1:0)]\}$ where α a the root defined by p_k.
        end if
    end for
end for
return(S).

Remark 5.3.8. 1. The partial derivatives of order $i+1$ are computed using the information about the partial derivatives of order i. By this way, the amount of work for computing the derivatives is minimized.

2. In Algorithm 5.3.2(Infsings), the roots of the gcds b_i are the x coordinates of all the at least $(i+1)$-fold points $(x:1:0)$ of F.

3. The roots of b_i/b_{i+1} are the x-coordinates of all the exactly $(i+1)$-fold points $(x:1:0)$ of F.

5.4 Profiling the `neighbGraph` Algorithm

In this section we give an analysis of `neighbGraph`'s most time consuming parts based on sample executions of the algorithm for algebraic curves of high degree. The goal of this analysis is to isolate those parts of the algorithm which may benefit from parallel computation.

The sample runs were executed on a PIII@450MHz processor in a Linux environment. For the defining polynomials of the investigated examples and their neighborhood trees we refer to Appendix C. Clearly, computation of the

(Sub-) Algorithm	Example 1	Example 2	Example 3	Example 4
Computing singularities	606	201	405	1830
$\sum$ `neighbGraph`	629	236	423	1899

Table 5.3: Profiling the `neighbGraph` algorithm (all timings in seconds)

singularities of type $(x : 1 : 0)$ takes most of the computing time. We can give two reasons for this behavior:

1. Computing partial derivatives of high order increases the size of the coefficients by an order of magnitude, if the polynomial is of degree > 10.

2. By the application of a quadratic transformation, the degree of the homogeneous polynomial almost doubles. In our examples, polynomials of degree far more then 100 arise.

5.5 Parallel Computing of Singularities on the Projective Line $(x : 1 : 0)$

Algorithm 5.3.2(Infsings) computes the class of singularities $(x : 1 : 0)$ together with their multiplicities by a series of gcd computations on the partial derivatives of the polynomial $F(x, 1, z)$ of the same order. The most involving steps in this algorithm are in fact the computation of the partial derivatives $\frac{\partial^m F(x,1,z)}{\partial x^i \partial z^j}(x, z),\ \ i + j =\leq m$ and the subsequent gcd computation of all derivatives of oder m. These operations set up a triangular structure where each line corresponds to the partial derivatives of F having the same order (see Figure 5.2). The polynomials b_i represent the gcd of the elements in line i, where 0 is substituted for z. For a first parallelization attempt, we note the following observations:

- Each line in the triangle can be treated independently from all other lines.
- By an appropriate blocking of the gcd computation over a line we get tasks of appropriate granularity for the *Distributed Maple* system.

However, this approach turns out to be extremely inefficient. Directly computing a partial derivative of order m, which has the form $\frac{\partial^m F(x,1,z)}{\partial x^i \partial z^j}(x, z),\ \ i + j = m$, involves computation of at least $i + j - 1$ partial derivatives of order less than m (namely j differentiations with respect to z and $i - 1$ differentiations

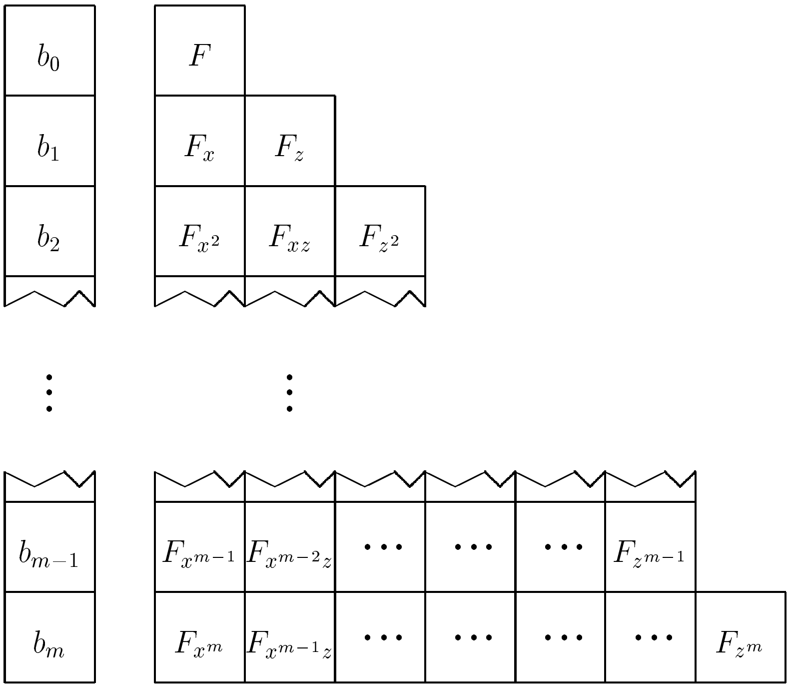

Figure 5.2: Triangular Structure of Algorithm 5.3.2.

with respect to x). In other words, in order to obtain this particular partial derivative, the whole part of the triangle whose lower right corner is defined by this derivative, is computed.

Our goal is to develop an algorithm which avoids this superfluous computations and still gives enough opportunities for parallel computation. The strategy of the original algorithm is not feasible here, as the computation of derivatives of a certain order is tightly coupled to the derivatives of one order less. Nevertheless, by reorganizing the way derivatives are computed, we can achieve our goals, namely

1. Create independent tasks of reasonable size for parallel execution, and
2. only one derivation operation is needed to compute a derivative of a certain order.

For this, let $F(x,z)$ be a bivariate polynomial of total degree d and $d^i_j \in \mathbb{Q}[x,z]$ the derivative $\frac{\partial^{i+j}}{\partial^i x \partial^j z}F(x,z)$ of order $i+j$. Our goal is to compute a list of gcds of the form $(b_0(x), \ldots, b_r(x))$, such that $b_i(x)$ is the gcd of all partial derivatives $d^u_v(x,0)$ where $u+v \leq i$, r is the first order such that $\deg b_r = 0$, and b_0 is the greatest square free divisor of $F(x,0)$. The roots of the $b_i(x)$ are the x-coordinates of all singularities $(x : 1 : 0)$ of order at least $i+1$. The x-coordinates of all singularities $(x : 1 : 0)$ of order i are then given by the roots of the quotient $q(x) := \frac{b_{i-1}(x)}{b_i(x)}$, $2 \leq i \leq r$.

Computing the partial derivatives together with the gcd operations takes most of the time in the sequential algorithm. The basic parallelization idea is

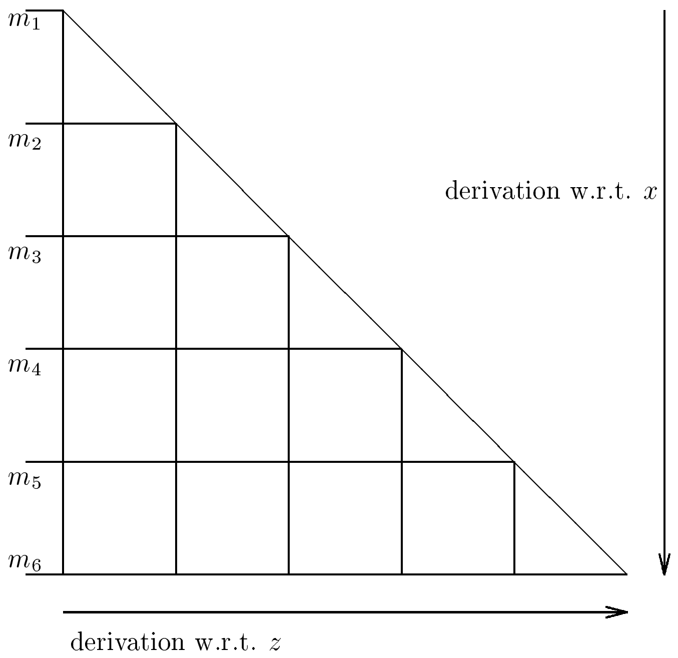

Figure 5.3: Schematic Drawing of the Blocking Structure

to compute all partial derivatives d_j^{i-1}, $i \leq j$ in the triangular matrix shown in Figure 5.2 in parallel whose data dependencies have been resolved. More accurately, for which a partial derivative of the form d_{j-1}^{i-j-1}, $i > j$ corresponding to the matrix entry $(i-1, j)$ or d_{j-1}^{0}, $i = j$ corresponding to the matrix entry $(i-1, j-1)$ has already been computed. In order to increase granularity, each task should compute multiple elements of partial derivatives following each other. other.

We achieve this by partitioning the matrix into square blocks of size $m \times m$ (for some $m > 1$) which contain all partial derivatives computed by the same task. The blocks along the diagonal of the triangular matrix have themselves a triangular shape. The algorithmic idea is as follows:

- The main program, in charge of generating the tasks, itself computes iteratively the partial derivatives of the form d_i^0, that is the partial derivatives along the main diagonal of the matrix.

- When it has finished computing all those derivatives corresponding to the diagonal boundary of a triangular block, the task computing the entries of this block is started.

- A task does not return all partial derivatives covered by a block as we are only interested in the last line of derivatives (which allow execution of the task computing the block below) and the gcds of partial derivatives for all lines in a block.

- A task computes the partial derivatives corresponding to the block in the following way:

1. If the block is of square shape it successively computes all partial derivatives of the same order starting from the first line. After the computation of partial derivatives in a line has been finished, the gcd of these derivatives is built. The last line of derivatives together with the gcds for all lines is returned as result.
2. For a triangular block, the tasks proceed analogously to square blocks, however fewer partial derivatives have to be computed.

- The main program receives the gcd results in arbitrary order updating the corresponding $b_i(x)$ which are initialized with $b_0(x)$.
- When computation of the derivatives is finished, the main program has to compute $\gcd(b_{i-1}, b_i)$ for i starting from 1 until a constant gcd is encountered.

In order to let the algorithm efficiently execute in an environment with a limited number p of processors, we use the following scheduling strategy:

- At first, the main program starts p tasks computing the first p triangular blocks of the main diagonal in the matrix.
- When computation of a block has been finished, the block immediately below this block is enabled for execution because its data dependencies have been resolved. Moreover, if the block was of triangular shape (and it was not among the first p initial blocks) the subsequent triangular block is also enabled for execution.
- Next the main program determines the block with minimal line and column index which is ready for execution and starts a task computing the values of this block.
- As soon as the termination criterion $\deg b_i = 0$, for some i, is reached, only blocks with line index less than and equal to i are enabled and eventually executed.

The termination condition is eventually reached as after a finite number of differentiations (bounded by the total degree d) at least one derivative has a constant value and hence also the gcd of all derivatives of this particular order is constant.

In Algorithm 5.5.1(Parallel Infsings) we present the pseudo code summarizing our ideas. Due to a high number of technical details occurring in the actual implementation we restrict ourselves to an abstract description showing the general structure of the algorithm.

5.6 Execution Times and Conclusions

We have systematically benchmarked the parallel `neighbGraph` algorithm with the curves used for the sequential profiling (see Section 5.4) in two environments consisting of 24 processors each:

- a Silicon Graphics Cray Origin 2000 multiprocessor with 64 R12000@300MHz processors. 24 processors were used;

Algorithm 5.5.1 (Parallel Infsings) A parallel algorithm for computing singularities of the form $(x:1:0)$ together with their multiplicities

Input: $F(x,z)$, a bivariate polynomial of total degree d,
$m > 1$, a blocking factor.
Output: $(b_0(x),\ldots,b_r(x))$, list of gcds s.t. b_i is the gcd of all partial derivatives $d_v^u(x,0)$ with $u+v \le i$.

$b_0(x) := \mathrm{gsfd}(F(x,0))$.
$maxLine := d$.
for $i = 1$ **to** p **do**
  Start computation of the triangular block (i,i) with $d^0_{m(i-1)+1},\ldots,d^0_{mi}$ as input.
end for
repeat
  Let $[(d_{i_1},\ldots,d_{i_m}),(g_{i_1},\ldots,g_{i_m})]$ be the result of any task (l,c) which has finished computation. The d_{i_j}s are the partial derivatives of the last line of block (l,c) and the g_{i_j}s are the gcds of all derivatives of the lines in the block.
  for $k = 1$ **to** m **do**
    $b_{i_k} := \gcd(b_{i_k}, g_{i_k})$.
    if $\deg b_{i_k} = 0 \ \wedge\ i_k < maxLine$ **then** $maxLine := i_k$.
  end for
  Enable the square block $l+1,c)$ for computation using $(d_{i_1},\ldots,d_{i_m})$ as input.
  if $l = c$ **then**
    Enable the triangular block $(l+1, c+1)$ for execution using $(d^0_{lm+1},\ldots,d^0_{(l+1)m})$ as input.
  end if
  Among the enabled blocks with line index less than or equal to $maxLine$ choose one with minimal line index and start a task computing the values of this block.
until there are no more tasks
$i := 0$.
while $\deg b_i \neq 0$ **do**
  $b_{i+1} := \gcd(b_{i+1}, b_i)$.
  $i := i+1$.
end while
return$((b_0,\ldots,b_i))$.

- a mixed configuration of 4 dual-processor Octanes with 8 R10000@250MHz processors and 16 processors of the Origin multiprocessor. The Origin 200 and the Octanes are linked by a high bandwith ATM line, while the Octanes are linked via a 10 Mbit Ethernet network.

The cluster environment described in Section 3.6, which involves 16 Linux PCs at various speed turned out to be not suitable for benchmaring this algorithm. The Linux PCs have a significantly low networking performance which slows down execution times tremendously as huge amounts of data have to be communicated among the individual processors.

The following table illustrates the performance of the various benchmarking configurations in the two environments relative to a PIII@450MHz processor. These numbers were measured by a representative Maple benchmark.

Performance (relative to PIII@450MHz)						
Environment	1	2	4	8	16	24
Origin	0.78	1.56	3.12	6.24	12.48	18.72
Mixed	0.58	1.16	2.32	4.64	10.88	17.12

- Table 5.4 gives an overview of the total sequential execution time, the sequential execution time for computing the singularities of the form $(x : 1 : 0)$ and the maximum time for this computation. Additionally, the total number of singularity computations together with the parallel quota is presented.

- The first table in Figure 5.7 lists the absolute speedup values $\frac{T_s}{T_n}$, where T_s denotes the sequential execution time and T_n denotes the parallel execution time using n processors in the two benchmarking environments. The first row of diagrams in Figure 5.8[1] visualizes this data.

- The second table in Figure 5.7 presents the scaled speedups $\frac{nT_s}{T_n \sum_{i=1} np_i}$, where p_i is the relative performance of processor i. These values are visualized in the second row of diagrams in Figure 5.8[1].

- The third table in Figure 5.7 gives the scaled efficiency $\frac{T_s}{T_n \sum_{i=1} np_i}$, which compares the speedup we actually got to the upper bound of speedup we could have got. The third row of diagrams in Figure 5.8[1] visualizes this data.

Analyzing the experimental data leads to the following results:

- Most obviously, the speedups for larger examples are better than with smaller ones. For Example 4, the Origin configuration gives an absolute speedup of 4.7 using 24 processors. This is not too good, however, communication is a real bottleneck for these experiments. Due to the huge amounts of data to be sent over the network file buffered message passing has to be used. Additionally, only the most time consuming singularity computations are performed in parallel.

[1]The markers + and □ denote execution in the mixed environment and on the Origin, respectively.

- The super-one efficiencies occuring in Examples 2 and 3 may partly be explained by errors in our performance estimations. Moreover, computing the gcds in a tree-like way seems to be slightly more efficient than doing it linearly from left to right.

- In almost all examples, the mixed configuration gives better speedup values than the Origin environment. This is due to the fact that in the mixed configuration the initial scheduler process is running on a SGI Octane, which has a smaller scheduling priority and hence communication operations performed by the initial scheduler involving read/write access to the disk are less likely to be interfered by computational tasks.

- Even though parallel speedups are bounded by 5, individual singularity computations achieve significantly higher speedups. In Example 4, the speedup for the most time consuming singularity computation is 6.7 using 24 Origin processors and 7 in the mixed environment using 16 processors.

- The absolute speedups achieved on the Origin are in most cases comparable to the mixed configuartion. However, scaled speedups often differ significantly, where the mixed configuration is superior to the Origin.

	T_s	T_s sing.	$T_{\max}$ seq. sing.	# sing.	# par. sing.
Example 1	629	606	552	8	3
Example 2	233	201	198	5	1
Example 3	423	405	402	6	1
Example 4	1899	1830	1798	6	3

Figure 5.4: Overview of the Experimental Data

In Figure 5.9 we see execution profiles generated by *Distributed Maple* for Example 4 running on 16 processors in both environments. The top diagram corresponds to execution in the mixed environment, the second diagram illustrates execution on the Origin multiprocessor.

We see the typical structure of a parallel singularity computation as a series of tasks corresponding to the computation of partial derivatives and their gcds. Clearly we see the communication bottleneck on the Origin visualized by two tasks whose length is significantly higher than the length of the others. This behaviour limits speedup tremendously and gives low efficiency values.

P		Example 1	Example 2	Example 3	Example 4
1	T_P	770	402	231	2429
	T_P sing.	738	377	193	2339
	T_P max. sing.	674	377	193	2291
2	T_P	447	256	173	1267
	T_P sing.	418	231	136	1171
	T_P max. sing.	325	231	136	1079
4	T_P	311	213	158	661
	T_P sing.	278	187	120	683
	T_P max. sing.	228	187	120	522
8	T_P	264	114	119	462
	T_P sing.	231	85	82	374
	T_P max. sing.	197	85	82	336
16	T_P	268	190	165	455
	T_P sing.	237	162	127	374
	T_P max. sing.	196	162	127	330
24	T_P	228	124	114	401
	T_P sing.	195	98	77	312
	T_P max. sing.	151	98	77	268

Figure 5.5: Detailed Execution Times on Origin 2000

P		Example 1	Example 2	Example 3	Example 4
1	T_P	1012	276	560	3209
	T_P sing.	963	218	521	3088
	T_P max. sing.	888	218	521	3015
2	T_P	566	197	320	1461
	T_P sing.	516	142	282	1336
	T_P max. sing.	436	12	282	1264
4	T_P	347	148	192	883
	T_P sing.	300	91	152	753
	T_P max. sing.	233	91	152	703
8	T_P	251	119	143	637
	T_P sing.	205	63	106	516
	T_P max. sing.	155	63	106	469
16	T_P	192	157	126	420
	T_P sing.	105	101	88	297
	T_P max. sing.	147	101	88	258
24	T_P	254	104	141	496
	T_P sing.	210	48	101	369
	T_P max. sing.	168	48	101	322

Figure 5.6: Detailed Execution Times in the Mixed Environment

Absolute Speedup							
Example	Architecture	1	2	4	8	16	24
1	Origin	0,8	1,4	2,0	2,4	2,3	2,8
	Mix	0,6	1,1	1,8	2,5	3,3	2,5
2	Origin	1,0	1,3	1,5	2,0	1,4	2,0
	Mix	0,8	1,2	1,6	2,0	1,5	2,2
3	Origin	1,1	1,7	2,0	3,7	2,2	3,4
	Mix	0,8	1,3	2,2	3,0	3,4	3,0
4	Origin	0,8	1,5	2,9	4,1	4,2	4,7
	Mix	0,6	1,3	2,2	3,0	4,5	3,8

Scaled Speedup							
Example	Architecture	1	2	4	8	16	24
1	Origin	1,0	1,8	2,6	3,1	2,9	3,6
	Mix	1,0	1,9	3,2	4,3	4,9	3,5
2	Origin	1,2	1,7	1,9	2,6	1,8	2,6
	Mix	1,3	2,0	2,9	3,4	2,2	3,1
3	Origin	1,4	2,2	2,6	4,7	2,8	4,4
	Mix	1,3	2,2	3,9	5,2	5,0	4,2
4	Origin	1,0	1,9	3,7	5,2	5,4	6,0
	Mix	1,0	2,2	3,9	5,2	6,6	5,3

Scaled Efficiency							
Example	Architecture	1	2	4	8	16	24
1	Origin	1,0	0,9	0,6	0,4	0,2	0,1
	Mix	1,0	0,9	0,8	0,5	0,3	0,1
2	Origin	1,2	0,8	0,5	0,3	0,1	0,1
	Mix	1,4	1,0	0,7	0,4	0,1	0,1
3	Origin	1,4	1,1	0,6	0,6	0,2	0,2
	Mix	1,4	1,1	0,9	0,6	0,3	0,2
4	Origin	1,0	0,9	0,9	0,7	0,3	0,3
	Mix	1,0	1,1	0,9	0,6	0,4	0,2

Figure 5.7: Absolute Speedups, Scaled Speedups and Scaled Efficiency

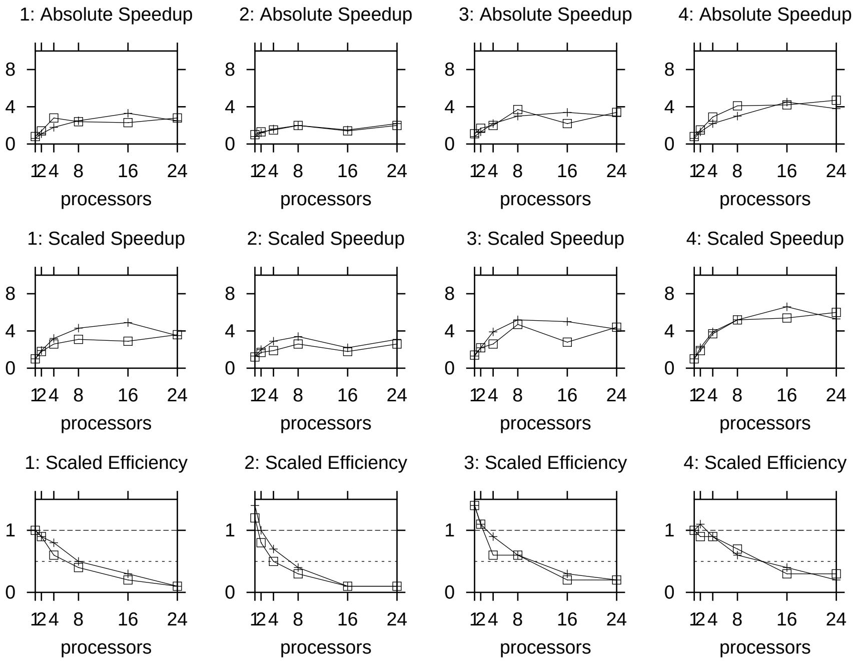

Figure 5.8: Experimental Results

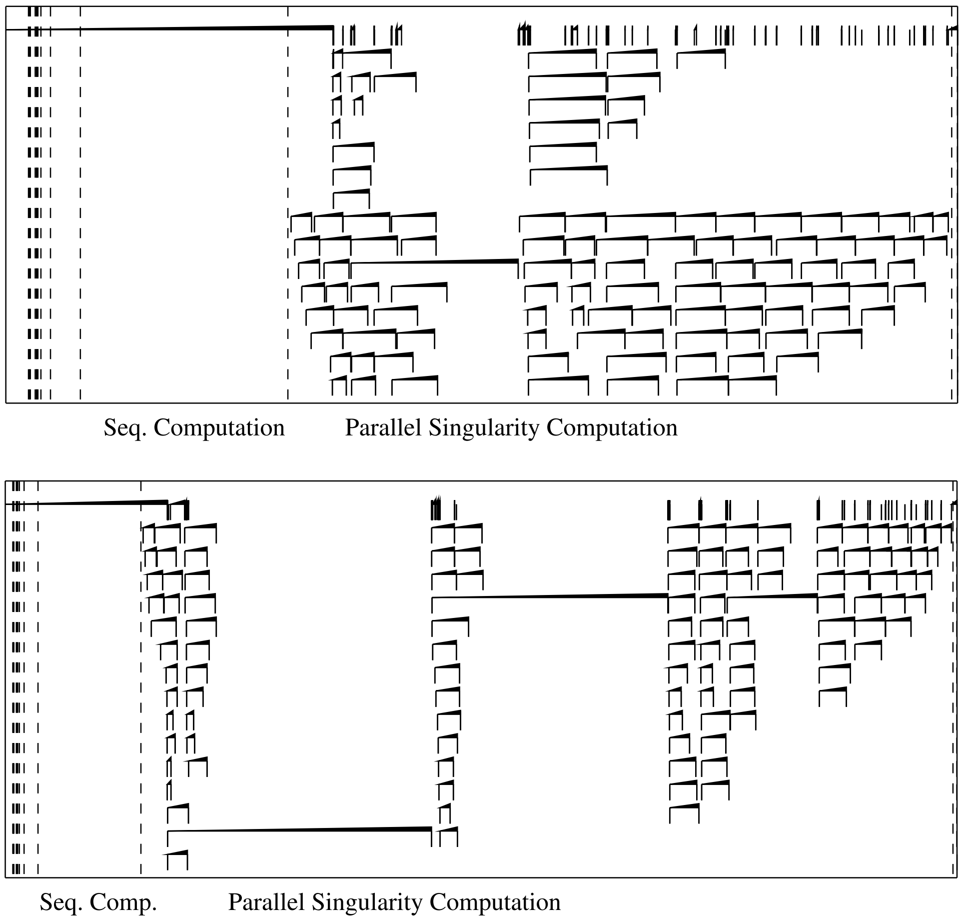

Figure 5.9: Dynamic Behavior

Chapter 6

Conclusions

Summarizing our work, we draw the following conclusions:

- Parallelization of computer algebra algorithms *pays off* as we can see from our own experiences in this area. Even though employing parallel computing is not a straightforward exercise, it is well worth to put a reasonable amount of effort in it.
- Parallelization of symbolic algorithms often requires *new algorithmic ideas* in order to resolve data dependencies and generate a sufficing number of independent tasks. Granularity and load balancing are an important issue in this context, as task size and memory requirements are virtually unpredictable due to the *exact* computation.
- Analyzing and profiling the sequential algorithms often leads to *sequential algorithmic improvements* reducing computing time significantly. We could find such improvements for the `pacPlot` algorithm avoiding computation with algebraic numbers.
- Subtle algorithmic changes of the parallel programs sometimes lead to better performance of the parallel algorithm executed on one processor compared to the sequential method.
- A data management system capable of dealing with *huge intermediate results* is required. The *Distributed Maple* system uses file buffered message exchange for task results in order to overcome memory allocation problems. On the other hand, employing several Maple kernel in computation reduces the amount of memory required for each individual kernel leading to less garbage collections and reducing computing time significantly.
- In some benchmarking environments (Origin 2000, cluster of PCs and workstations) problems during parallel execution emerge. Individual tasks run significantly longer than others cutting down the speedup we could achieve. This problem is not solved yet and has to be investigated further.
- Even though computer algebra systems allow to express mathematical statements in a way very close to usual mathematical notation, one carefully has to think about the effect of this compact notation on the time complexity of the algorithms.

Appendix A

`pacPlot` Examples

Example 1: Defining equation:

$$f(x,y) := 23x^4y^8 - 17x^6y^6 + 22x^8 + 84x^7y^7 + 20x^4y^6 - 49x^3y^8 - \\ 80x^4y^2 - 5x^3y - 92x^3y^2 - 47y^8 + 5xy^5$$

Example 2: Defining equation:

$$f(x,y) := -55x^4y^3 + 40y - 78y^7 + 88x^5y^8 + 81x^8y + 4x^4y^8 + 57x^5y^7 - 11xy^5$$

Example 3: Defining equation:

$$f(x,y) := -22x^9y^9 - 26x^2y^5 + 85x^8y - 50x^2y^6 - 74x^6y^4 - 31x^8y^7 - 172x^7y^7$$

Example 4: Defining equation:

$$f(x,y) := -37x^{11}y^{10} - 42y^8 - 65xy^{10} - 61x^4y^7 + 29x^{11}y^4 + \\ 78x^9y^5 + 68x^5y - 36x^7$$

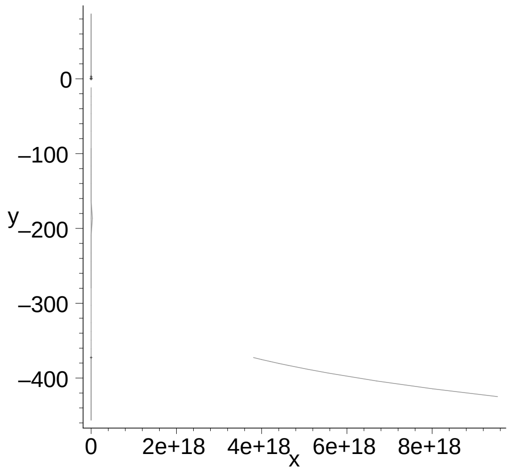

Figure A.1: Example 1

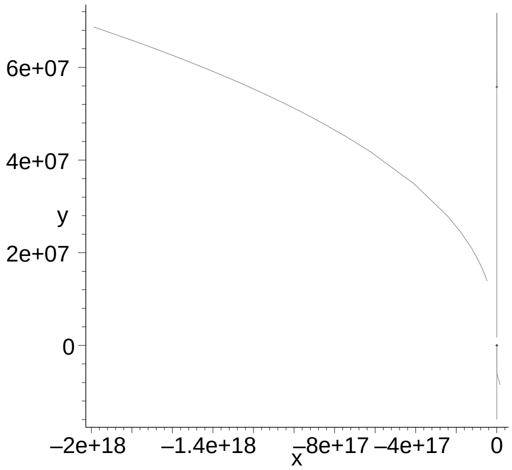

Figure A.2: Example 2

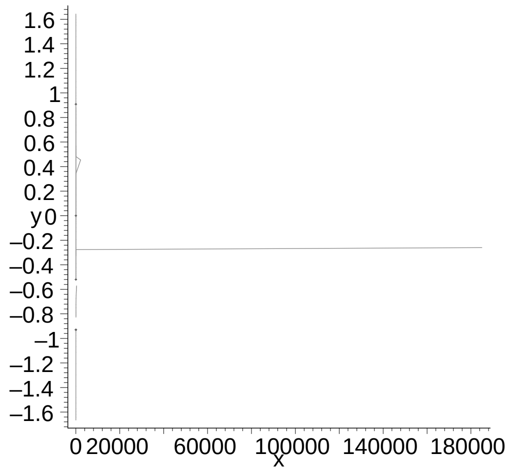

Figure A.3: Example 3

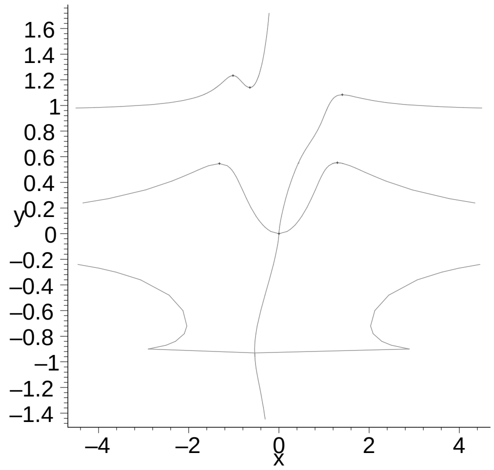

Figure A.4: Example 4

Appendix B

`ssiPlot` Examples

Example 1: Defining equations:

$$f_1(x, y, z) := 38z + 6y^2z - 50z^2 - 47y^2z^3 + 75xy^2 + 67z^4$$
$$f_2(x, y, z) := -21x^2z - 75x^2y^2z + 93y^3z^2 + 23x^5 - 16y^5$$

Example 2: Defining equations:

$$f_1(x, y, z) := -94x^2z - 65x^2y^3 + 90x^4 - 38y^3z^2 - 46yz^3 + 28xz$$
$$f_2(x, y, z) := 23x^2z^2 - 20x^2y^3 - 7z^5 + 4xz^4 - 77x^2yz^2 + 40x^3z^2$$

Example 3: Defining equations:

$$f_1(x, y, z) := -30y^2z - 41x^2z - 5xy^3z - 17x^3y - 49yz - 26y^4z$$
$$f_2(x, y, z) := -87x - 75z^4 + 47xyz^3 - 16xz^2 + 38xy^4 - 53x^5$$

Example 4: Defining equations:

$$f_1(x, y, z) := 36z - 33x^2z^2 - 86y^4z - 8xy^3z - 98x^5 + 78z^5$$
$$f_2(x, y, z) := 92xyz - 67y + 23xy^4 + 89x^2z^3 + 95xy^3 - 10y^2$$

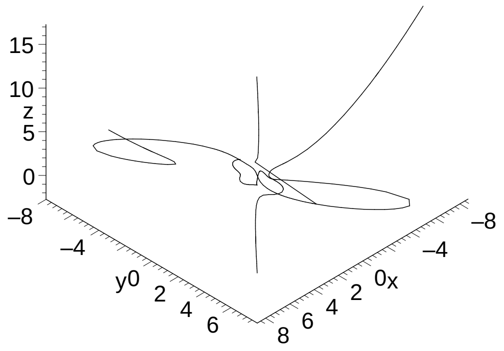

Figure B.1: Example 1

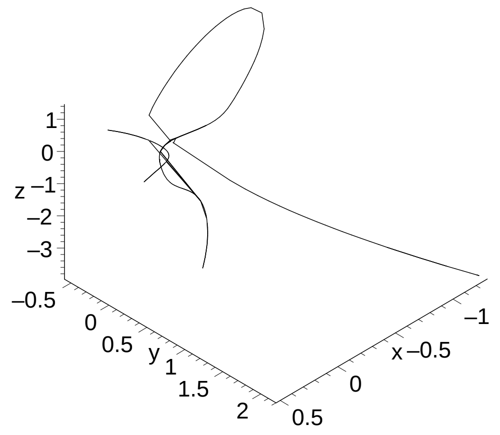

Figure B.2: Example 2

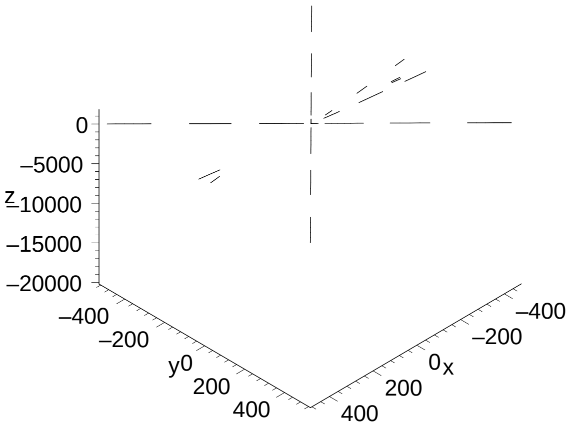

Figure B.3: Example 3

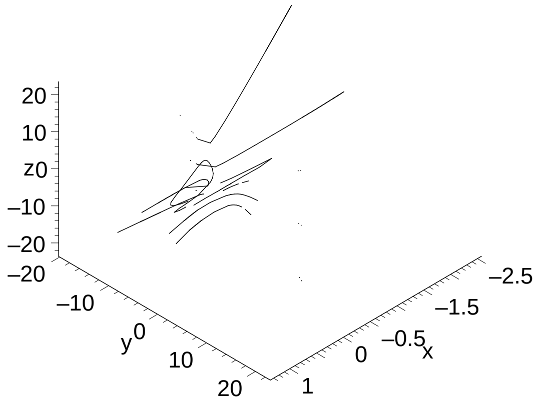

Figure B.4: Example 4

Appendix C

neighbGraph Examples

Example 1: Defining equation:

$$f(x, y) := y^2 - x^9$$

Neighborhood graph:

```
N := [[2, 0, 0, 1, {x = x+y, y = y-x},
        [[2, 1, 1, 0, {}, [[2, -1, 1, 0, {x = x+y, y = y-x},
                [[2, -1, 1, 0, {x = x+y, y = y-x}, []]]
        ]]]]],
      [7, -1, 1, 0, {x = x+y, y = y-x},
        [[2, 2, 1, 0, {y = y-2*x, x = x+2*y},
                [[2, 1, 1, 0, {x = x+y, y = y-x},
                        [[2, -1, 1, 0, {x = x+y, y = y-x}, []]]
      ]]]]]]
```

Example 2: Defining equation:

$$f(x, y) := y^7 - xy^3 + x^6y - x^{15}$$

Neighborhood graph:

```
N := [[4, 0, 0, 1, {x = x+y, y = y-x},
        [[3, 1, 1, 0, {},
                [[2, -1, 1, 0, {x = x+y, y = y-x}, []]]
      ]]],

     [8, -1, 1, 0, {x = x+y, y = y-x},
        [[7, 1, 1, 0, {x = x+y, y = y-x}, []]]
     ]]
```

Example 3: Defining equation:

$$f(x, y) := 2x^{10} + y^6 - 3x^2y - 2y^3 + y^2$$

Neighborhood graph:

```
N := [[2, 0, 0, 1, {x = x+y, y = y-x},
        [[2, 1, 1, 0, {}, []]]],

      [4, -1, 1, 0, {x = x+y, y = y-x},
        [[4, 1, 1, 0, {x = x+y, y = y-x},
                [[2, -1, 1, 0, {x = x+y, y = y-x},
                        [[2, 1, 1, 0, {x = x+y, y = y-x}, []]]
     ]]]]]]
```

Example 4: Defining equation:

$$f(x,y) := y^{12} - y^7x^2 - y^7x + 7 * y^6x^3 + 2y^5x^5 + x^{12} - x^5y^2 - y^3x^2$$

Neighborhood graph:

```
N := [[5, 0, 0, 1, {},
        [[2, -1, 1, 0, {x = x+y, y = y-x},
                [[2, -1, 1, 0, {x = x+y, y = y-x},
                        [[2, -1, 1, 0, {x = x+y, y = y-x}, []]]]]
      ],
      [3, 1, 1, 0, {x = x+y, y = y-x},
        [[3, -1, 1, 0, {x = x+y, y = y-x}, []]]
     ]]]]
```

List of Figures

List of Algorithms

Bibliography

[Akr89] A. G. Akritas. *Elements of Computer Algebra with Applications.* John Wiley & Sons, (1989).

[Ber97] L. Bernardin. Maple on a Massively Parallel, Distributed Memory Machine. In E. Kaltofen M. Hitz, editor, *Parallel Symbolic Computation PASCO'97*, pages 217–222. ACM Press, 1997.

[B.W94] B.W. Char B.W. A Users's Guide to Sugarbush – Parallel Maple through Linda. Technical Report MCS-94-01, Drexel University, Department of Mathematics and Computer Science, 1994.

[Ces97] G. Cesari. CALYPSO: a Computer Algebra Library for Parallel Symbolic Computation. In E. Kaltofen M. Hitz, editor, *Parallel Symbolic Computation PASCO'97*, pages 204–216. ACM Press, 1997.

[Cha90] B. W. Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *ISSAC'90 International Symposium on Symbolic and Algebraic Computation*, pages 96–103, Tokyo, Japan, August 20–24, 1990. ACM Press, New York.

[CJK90] G. E. Collins, J. R. Johnson, and W. Küchlin. Parallel Real Root Isolation Using the Coefficient Sign Variation Method. In R. E. Zippel, editor, *Computer Algebra and Parallelism*, LNCS, pages 71–87. Springer Verlag, (1990).

[CL83] G. E. Collins and R. Loos. Real Zeros of Polynomials. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra, Symbolic and Algebraic Computation*, pages 83–94. Springer Verlag Wien New York, 2nd ed. edition, (1983).

[Col71] G. E. Collins. The Calculation of Multivariate Polynomial Resultants. *Journal of the ACM*, 19:515–532, 1971.

[Dix08] A.-L. Dixon. The Elimination of Three Quantics in Two Independent Variables. *Proceedings of the London Mathematical Society*, 6:468–478, 1908.

[DK] T. Decker and W. Krandick. Towards a Scalable Parallel Algorithm for Polynomial Real Root Isolation. Technical report, Department of Mathematics and Computer Science, University of Paderborn.

[DKSV91] A. Diaz., E. Kaltofen, K. Schmitz, and T. Valente. DSC — A System for Distributed Computation. In M. Watt, editor, *Proceedings of ISSAC'91*, pages 324–333. ACM Press, 1991.

[GR94] T. Gautier and J.-L. Roch. PAC++ System and Parallel Algebraic Numbers Computations. In Hoon Hong, editor, *PASCO'94*, pages 145–153. World Scientific Publishing, 1994.

[HL94] H. Hong and H. W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR94-VAPP VI – Third Joint International Converence on Vector and Parallel Processing*, number 854 in LNCS, pages 325–336. Springer, Berlin, (1994).

[HNS95] H. Hong, A. Neubacher, and W. Schreiner. The Design of the SACLIB/PACLIB Kernels. *Journal of Symbolic Computation*, 19:111–132, 1995.

[HSF] H. Hong, W. Schreiner, and T. Fadgyas. Technical report.

[KS95] D Kapur and T. Saxena. Comparison of Various Multivariate Resultant Formulations. In *Proceedings of the ISSAC'95 Conference*, 1995.

[KSY94] D. Kapur, T. Saxena, and L. Yang. Algebraic and Geometric Reasoning using Dixon Resultants. In *Proceedings of the ISSAC'94 Conference*, 1994.

[Küc90] W. Küchlin. PARSAC-2: A Parallel SAC-2 Based on Threads. In S. Sakata, editor, *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, Japan, August 1990. Springer.

[Loo83] R. Loos. Computing in Algebraic Extensions. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra, Symbolic and Algebraic Computation*, pages 173–187. Springer Verlag Wien New York, 2nd ed. edition, (1983).

[Mis93] B. Mishra. *Algorithmic Algebra.* Texts and Monographs in Computer Science. Springer Verlag, (1993).

[MRSW98] Metzner, Radimersky, Sorgatz, and Wehmeier. *MuPAD User's Guide to Macro Parallelism in MuPAD 1.4.1.* University of Paderborn, August 1998.

[MW96] M. Mnuk and F. Winkler. CASA - A System for Computer Aided Constructive Algebraic Geometry. In *DISCO'96 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 1128 of *LNCS*, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.

[Sch98] W. Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998.

[Sch99] W. Schreiner. Developing a Distributed System for Algebraic Geometry. In B.H.V. Topping, editor, *EURO-CM-PAR'99 Third Euroconference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, (1999). Civil-Comp Press, Edinburgh.

[Sie93] K. Siegl. Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press, New York.

[SW90] R. J. Sendra and F. Winkler. Symbolic Parametrization of Curves. *Journal of Symbolic Computation*, 12(6):607–631, 1990.

[Tra94] Q.-N. Tran. Extended Newton's Method for Finding the Roots of an Arbitrary System of Equations and its Applications. Technical Report 94-49, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, 1994.

[Tra96] Q.-N. Tran. *A Hybrid Symbolic-Numerical Approach in Computer Aided Geometric Design (CAGD) and Visualization.* PhD thesis, RISC-Linz, Johannes Kepler University Linz, Austria, (1996).

[Tra97] Q.-Nam Tran. Extending Dixon's resultant using pseudoinverse matrices and its apllications to geometric reasonning. Technical Report 97-11, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, 1997.

[Vin36] A. J. H. Vincent. Sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées*, pages 341–372, (1836).

[Wal78] R. J. Walker. *Algebraic Curves.* Springer, New York, 1978. Reprint of the 1950 edition of the Princeton University Press.

[Win96] F. Winkler. *Polynomial Algorithms in Computer Algebra.* Texts and Monographs in Symbolic Computation. Springer Wien New York, (1996).