A Parallel Symbolic-Numerical Approach to Algebraic Curve Plotting*

Christian Mittermaier, Wolfgang Schreiner, and Franz Winkler

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
FirstName.LastName@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

Abstract. We describe a parallel hybrid symbolic-numerical solution to the problem of reliably plotting a plane algebraic curve. The original sequential program is implemented in the software library CASA on the basis of the computer algebra system Maple. The new parallel version is based on Distributed Maple, a distributed programming extension written in Java. We describe the mathematical foundations of the algorithm, give sequential algorithmic improvements and discuss our parallelization approach.

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 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 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 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 quite obvious but can be explained easily. Purely numerical methods use linearization 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

* Supported by projects P11160-TEC (HySaX) and SFB F013/F1304 of the Austrian Fonds zur Förderung der wissenschaftlichen Forschung.

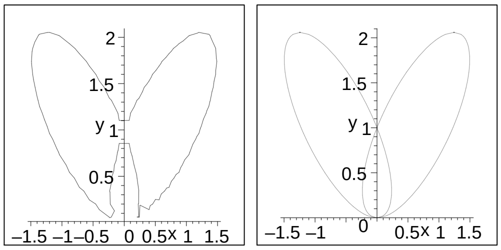


Fig. 1. Comparing Maple's `implicitplot` and CASA's `pacPlot`

more branches intersect each other, the Jacobian becomes singular as the partial derivatives with respect to both variables vanish. Increasing the granularity in the numerical plotting algorithm makes the picture smoother, but nevertheless the singular points are still missing.

A "good" plotting algorithm should fulfill the following informal requirements:

– **Draw topologically correct plots.** The course of the branches of the curve through singular points should be represented correctly, singular 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.
– **Use an efficient method for rendering the curve.** Computing time should be as low as possible without sacrificing correctness. Increasing efficiency and creating smoother pictures is not acceptable if it means a compromise on topological 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. Also the sequential version of `pacPlot` can be improved a lot by avoiding unnecessary computations in algebraic extension fields.

The program system CASA is based on Maple and contains a collection of algorithms for designing, analyzing and rendering algebraic curves and surfaces. CASA has been developed by a research group under the direction of the third author at RISC-Linz [MW96].

In a companion paper [SMW00] we describe the `pacPlot` algorithm from the point of view of parallelization while in this work we concentrate on the mathematical and algorithmic details.

The structure of this paper is as follows: In Section 2 we describe the mathematical and algorithmic ideas of the `pacPlot` algorithm and in Section 3 we list the most time consuming parts of the program. Section 4 deals with sequential algorithmic improvements of the original algorithm, Section 5 describes our parallelization approach, Section 6 presents experimental results which demonstrate the efficiency of the algorithm and the performance of the underlying distributed system. Finally, in Section 7 we summarize our results.

2 The `pacPlot` Algorithm

The `pacPlot` algorithm (plane algebraic curve plot) 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 [Tra96].

The hybrid approach combines the exactness of symbolic methods with the speed and efficiency of numerical approximations. `pacPlot` uses a symbolic preprocessing step where the points determining the topological structure of the curve are computed. For tracing the branches between these points Nam's Extended Newton Method [Tra94] is used. The general structure of the algorithm looks as follows:

1. Determine all critical points of the curve.
2. Find out starting points for visualization.
3. Trace the simple branches of the curve by using numerical approximation.

Definition 1. *Let $\mathcal{C}$ be an affine plane algebraic curve defined by the polynomial $f \in \mathbb{Q}[x,y]$. $(\xi,\eta) \in \overline{\mathbb{Q}}^2$ is a* critical point *of $\mathcal{C}$ iff $f(\xi,\eta) = \frac{\partial f}{\partial x}(\xi,\eta) = 0$. ($\overline{\mathbb{Q}}$ is the algebraic closure or $\mathbb{Q}$, i.e. the field of algebraic numbers.)*

In order to clarify how critical points determine the topological structure of the curve, we present the following picture (Figure 2) of a curve defined by the polynomial $f(x,y) := 8y^2 - 8x^2 - 8y^3 - 8x^3 + 2y^4 + x^5$.

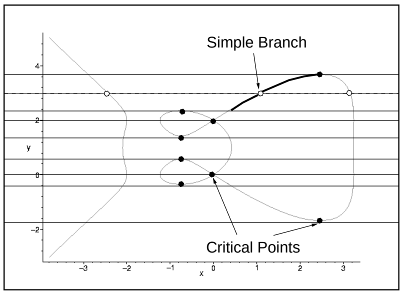


Fig. 2. Critical Points and the Topological Structure of an Algebraic Curve

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 2, the starting points are marked by grey dots. Next, all simple branches are traced in both directions using the Extended Newton method starting at the initial points of visualization.

Before we describe critical point computation in detail, we mention two important observations:

1. We are interested in the real solutions of the system $f(x,y) = \frac{\partial f}{\partial x}(x,y) = 0$ only, as we want to plot the real part of the algebraic curve.
2. We do not need the exact solutions of the system. 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, introduction of and computation with real algebraic numbers and their numerical evaluation is avoided.

For computing critical points we first transform the system $f = \frac{\partial f}{\partial x} = 0$ into a set of triangular systems of the form $p(y) = q(x,y) = 0$ ($p(y)$ irreducible over $\mathbb{Q}$) such that the solutions of all triangular systems are exactly the solutions of the original system. Isolating intervals for the real zeros of a triangular system are obtained 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 $p(y)$.
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 which combinations $X_k \times Y_j$, ($k \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$) contain a common zero of $p{=}q{=}0$.

Theorem 1 guarantees that we can always construct a polynomial $r(x)$ with the desired property.

Theorem 1. *Let $\mathbb{K}$ be an algebraically closed field and let $f(x,y) = \sum_{i=0}^{m} f_i(x) y^i$, $g(x,y) = \sum_{i=0}^{n} g_i(x) y^i$ be elements of $\mathbb{K}[x,y]$ of positive degrees m and n in y, respectively. Let $r(x) := res_y(f,g)$. 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. $\exists \eta \in \mathbb{K} : \; f(\xi,\eta) = g(\xi,\eta) = 0$.

Steps 2 and 3 are repeated until all intervals Y_j, $j \in \{1, \ldots, n\}$ are processed. In order to check which combinations $X_k \times Y_j$ contain a common zero of the system $p = q = 0$ we proceed as follows:

1. Factor p over $\mathbb{Q}$ and apply the following steps to each factor. W.l.o.g. we assume that p is the only irreducible factor.
2. Compute the greatest square free divisor $q^*(x, \eta)$ of $q(x, \eta)$ over $\mathbb{Q}(\eta)[x]$, where η is an arbitrary real root of $p(y)$. $q^*(x, \eta) = q(x, \eta)/\gcd(q(x, \eta), \frac{\partial}{\partial x} q(x, \eta))$.
3. Take an interval Y_j, $j \in \{1, \dots, n\}$ and consider p 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 to determine the appropriate intervals X_k . Such an algorithm requires as input the real algebraic number η_j represented by the isolating interval Y_j, the minimal polynomial $p(y)$, the squarefree polynomial $q(x, y) \in (\mathbb{Q}[y]_{/\langle p_i \rangle})[x]$ representing $q(x, \eta_j)$, for which we want to isolate the real roots and a list of candidates $X_1, \dots, X_m$ for the isolating intervals (isolating intervals for the real roots of the resultant $r(x)$).

The algorithm used in step 4 computes the sign of the real algebraic numbers $q(\xi_1, \eta)$ and $q(\xi_2, \eta)$, where ξ_1 and ξ_2 are the end points of an interval X_k. If the signs are different or at least one of of the signs is zero then we have found an isolating interval. For determining the signs, Algorithm 1 is used. For details we refer to [Loo83].

Algorithm 1 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

Critical point computation for a curve $\mathcal{C}$ defined by the bivariate polynomial $f(x, y)$ is done by the following steps:

1. $f_x := \frac{\partial f}{\partial x}$; $L := 0$

2. From f and f_x compute a set T of triangular systems $(p(y), q(x, y))$ such that the union of the solutions of all triangular systems is the solution set of $f = f_x = 0$. This can be done using Kalkbrener's `bsolve` algorithm [Kal90].
3. For each triangular system $(p(y), q(x, y)) \in T$ do the following:
   (a) Factor p over $\mathbb{Q}$ giving a set $P := \{p_j | \prod_j p_j = p\}$.
   (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 $]\xi_{1i}, \xi_{2i}[$ for the real roots of $q(x, \eta)$ whose length is smaller then the resolution of the output device. $L := L \cup \{]\xi_{1i}, \xi_{2i}[\times [\eta, \eta]\}$.
   - If $\deg(p_j) > 1$ then
     i. Let $I_1 := \{]\eta_{1i}, \eta_{2i}[\mid \eta_{1i}, \eta 2i \in \mathbb{Q}\}$ be a set of isolating intervals for the real roots of p_j of the required length.
     ii. Let α be a root of p_j and $q' := q(x, \alpha)$. Compute q'' as the quotient $q' / \gcd(q', \frac{\partial}{\partial x} q')$ over $\mathbb{Q}(\alpha)$. $q^* := q''(x, y)$.
     iii. $r(x) := \mathrm{res}_y(pj(y), q^*(x, y))$.
     iv. Let $I_2 := \{]\xi_{1i}, \xi_{2i}[\mid \xi_{1i}, \xi_{2i} \in \mathbb{Q}\}$ 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 $]\eta_{1i}, \eta_{2i}[\in I_1$ consider p_j as minimal polynomial for the real root η represented by the isolating interval and the algebraically square free polynomial $q^*(x, y) \in (\mathbb{Q}[y]_{/\langle p_j \rangle})[x]$ as representation of $q_i(x, \eta)$. Use a real root isolation algorithm for polynomials with real algebraic number coefficients in order to check which intervals from I_2 represent a real algebraic number ξ such that $p_j(\eta) = q(\xi, \eta) = 0$. (Such an algorithm requires q to be square free for every root of p). As a result we get a list of Cartesian product of intervals of the form $L' := \{X_i \times Y_j \mid X_i \in I_2,\ Y_j \in I_1\}$.
     vi. $L := L \cup L'$.

3 Profiling `pacPlot`

We have analyzed the most time consuming parts of `pacPlot` by doing sample executions for randomly generated high degree curves. The sample runs were executed on a PIII@450MHz Linux machine. Critical point computation is dominated by the gcd computation in the algebraic extension field of $\mathbb{Q}$ used for computing the greatest squarefree divisor. Resultant computation and real root isolation for polynomials with real algebraic number coefficients also require a large part of the overall computing time. Ordinary real root isolation is time consuming in only some of the cases, but still it is worth to put some effort in developing a parallel algorithm for this sub-problem.

	Example 1 deg = 14	Example 2 deg = 13	Example 3 deg = 18	Example 4 deg = 21
real root	137	187	7	171
resultant	542	455	82	4175
algebraic squarefree	[1]	7213	1204	[1]
real root over $\mathbb{Q}(\alpha)$	6525	1807	252	20060
$\sum$ critical points	>7213	9666	1546	>24484
$\sum$ `pacPlot`	>7267	9689	1552	>24518

Table 1. Profiling the `pacPlot` algorithm (all timings in seconds)

The overall impression obtained from this execution profile perfectly meets ones expectations, namely that the symbolic part of `pacPlot` is the toughest and most time consuming one.

4 Sequential Algorithmic Improvements

Also the sequential version of `pacPlot` can be improved a lot by avoiding unnecessary computations in algebraic extension fields. This section is based on the first author's diploma thesis [Mit00].

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

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 as a parameter is square free.

Algebraic real root isolation is used for checking which combinations of isolating intervals contain a critical point. 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 2 describes the idea in pseudo code.

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, the algorithm for computing the sign of a real algebraic number requires a squarefree polynomial representation of this 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. From a mathematical point of view, the original approach is more elegant as we have to do several gcd

[1] Maple fails with an `object too large` error in this position but computation runs through when the gcd computation is omitted.

Algorithm 2 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 $q(\xi_{2i}, y)).\ \}$
  $q_l^* := q_l \bmod p$.
  $q_r^* := q_r \bmod p$.
  for j from 1 to m **do**
    $l := \mathrm{sgn}(q_l^*, p, Y_j)$.
    $r := \mathrm{sgn}(q_r^*, p, Y_j)$.
    if $l = 0 \ \vee\ r = 0 \ \vee\ l \cdot r < 0$ **then**
      $L := L \cup \{X_i \times Y_j\}$.
    end if
  end for
end for
return(L).

computations here. But when efficiency is investigated carefully, our approach is superior to the former.

4.2 Speeding Up the Sign Computations

In the algorithm for computing the sign of a real algebraic number 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 occurring intervals.

4.3 Execution Time Comparisons

Comparing execution times of the improved sequential algorithm (Table 2) with execution times of the original method (Table 1) is not possible in the level of the individual sub-algorithms, however, comparing the total computing time for Step 1, we see that our improved version is significantly better. Moreover, if we replace the gcd computation over the algebraic extension field of $\mathbb{Q}$ by our equivalent but less complex approach, the big examples (Example 1 and 4) executable in the sense that Maple does not fail with an `'object too large'` error.

	Example 1 deg = 14	Example 2 deg = 13	Example 3 deg = 18	Example 4 deg = 21
real root	83	97	3	64
resultant	386	150	55	3121
checking	6180	166	86	8279
refinement	127	52	8	49
$\sum$ critical points	6870	470	155	11748

Table 2. Execution times of the improved sequential `pacPlot` algorithm

5 The Parallel Plotting Algorithm

Parallelization of the plotting algorithm is based on the sequentially improved version. Parallelism is applied on several levels:

1. *Parallel Resultant Computation* Our parallelization approach applies a modular method to compute resultants in multiple homomorphic images $\mathbb{Z}_{p_i}[x, y]$, where the p_i are prime numbers. We get a divide and conquer structure where both, the divide phase (the modular resultant computations) and the conquer phase (the application of the Chinese Remainder Theorem) can be parallelized obviously. For details, see [HL94, Sch99].
2. *Parallel Real Root Isolation* We use Uspensky's method [CL83], a divide and conquer search algorithm, for computing isolating intervals. A naive parallelization of this algorithm typically yields poor speedups due to the narrowness of the highly unbalanced search trees associated with the isolation process [CJK90].
   Our approach is to broaden and flatten the search trees up to a certain extent such that in every step each processor can participate in the computation of isolating intervals. Our algorithm is based on speculative parallelism yielding better results in many of the cases. For details, see [Mit00].

3. *Parallel Solution Test and Interval Refinement* The tests which intervals $X \times Y$ do indeed contain a solution of the given triangular system can be performed in parallel in a straight-forward fashion. Likewise we can refine all isolating intervals in parallel to the desired accuracy.

Algorithm 3 (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 of 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**
  - $F := \{p_i \mid \text{the } p_i \text{ are the irreducible factors of } p \text{ over } \mathbb{Q}\}$
  - **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**

The various ideas developed above are incorporated in the parallel version of the `pacPlot` algorithm, the `dpacPlot` algorithm (distributed `pacPlot`). Parallelization affects the critical point computation step of `pacPlot`, only. Obviously, the various subalgorithms simply have to be replaced by their parallel counterpart. In Algorithm 3 (Parallel Critical Points) we give a high level description of how critical points are computed in `dpacPlot`.

We used the *Distributed Maple* environment [Sch98], a system for writing parallel programs on the basis of Maple, for implementing our algorithms. It allows to create tasks and execute them on Maple kernels running on various machines in a network.

Each node of a *Distributed Maple* session comprises two components, a Java based scheduler coordinating the interaction between nodes and the Maple interface, which is a link between kernel and scheduler. Both components use pipes to exchange messages (which may employ any Maple objects). The parallel programming paradigm is essentially based on functional principles which is sufficient for many kinds of algorithms in computer algebra.

The basic operations representing the core programming interface consist of methods for executing a Maple command on every machine connected to the session, creating a task for evaluating an arbitrary Maple expression (tasks are represented by task handles) and waiting for the result of a task represented by its handle.

6 Experimental Results

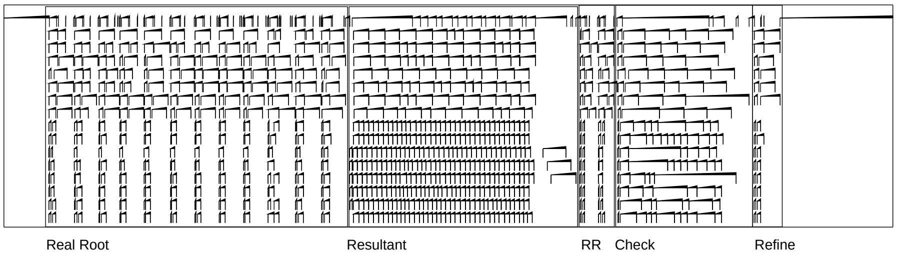
We have systematically benchmarked the `dpacPlot` algorithm implemented in *Distributed Maple* with four randomly generated algebraic curves for which the sequentially improved program takes 6870s, 470s, 155s and 11748s respectively. These times refer to a PIII@450MHz PC. The parallel algorithm has been executed in three system environments consisting of 24 processors each:

- A cluster which comprises 4 Silicon Graphics Octanes (2 R10000@250Mhz each) and 16 Linux PCs (various Pentium processors) linked by a 100 Mbit switched (point-to-point) Ethernet;
- a Silicon Graphics Origin multiprocessor (64 R12000@300 Mhz, 24 processors used);
- a mixed configuration consisting of our 4 dual-processor Octanes and 16 processors of the Origin multiprocessor interconnected via a high-bandwidth ATM line.

The raw total computational performance of cluster, Origin, and mixed configuration is 18.3, 17.1, and 18.7, respectively; these numbers (measured by a representative Maple benchmark) denote the sum of the performances of all processors relative to a PIII@450MHz processor. In the cluster and in the mixed configuration, the initial (frontend) Maple kernel was executed on an Octane.

The top diagram in Figure 3 generated from a Distributed Maple profile illustrates the execution in the cluster configuration with 16 processors listed on the vertical axis (8 Octane processors above 8 Linux PCs) and each line denotes a task executed on a particular machine; we see the real root isolation phase followed by the phases for resultant computation, the second real root isolation, the solution tests and the solution refinements. This visualization is generated from a run with Example 2 and illustrates the dynamic behavior, i.e. the number of tasks generated in each step of the algorithm. The length of these intervals does not reflect the execution time of these parts.

The table in Figure 3 lists the execution times measured in each system environment for each input with varying numbers of processors. The subsequent row of diagrams visualizes the absolute speedups $\frac{T_s}{T_n}$ (where T_s denotes the sequential execution time and T_n denotes the parallel execution time with n processors), the second row visualizes these speedups multiplied with $\frac{n}{\sum_{i=1}^{n} p_i}$ (where p_i denotes the relative performance of processor i), the third row visualizes the scaled



Execution Times (s)							
Example	Environment	1	2	4	8	16	24
1 (6870s)	Cluster	14992	8035	2732	1186	552	488
	Origin	7290	4217	1789	872	446	513
	Mixed	14992	8035	2732	1368	597	519
2 (470s)	Cluster	810	648	328	173	95	108
	Origin	667	541	297	166	112	116
	Mixed	810	648	328	210	116	127
3 (155s)	Cluster	267	191	112	67	46	45
	Origin	196	147	90	63	54	54
	Mixed	267	191	112	74	58	56
4 (11748s)	Cluster	25178	15559	6820	3562	1915	1563
	Origin	13397	8223	4009	2281	1726	1420
	Mixed	25178	15559	6820	4042	2004	1599

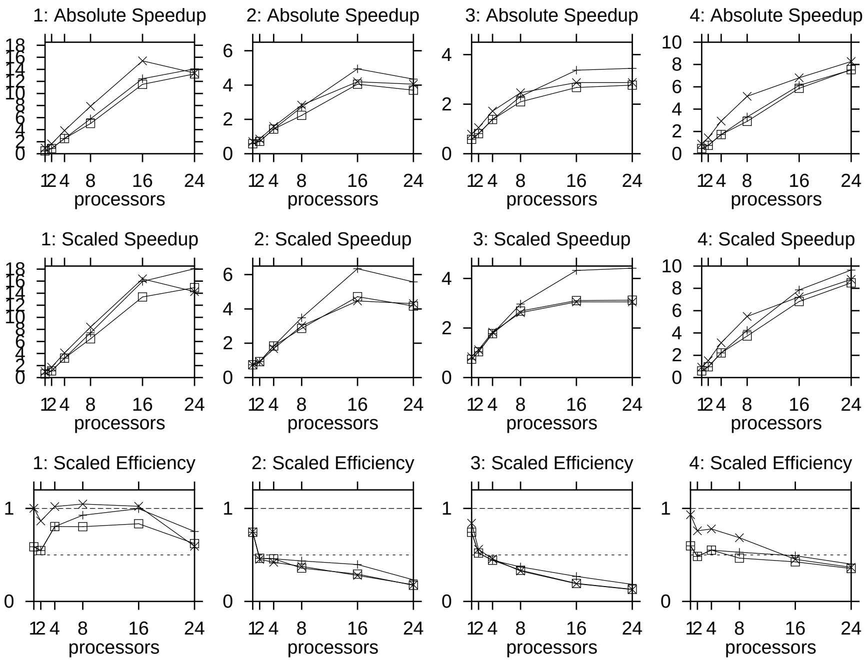


Fig. 3. Experimental Results

efficiency, i.e., $\frac{T_s}{T_n \sum_{i=1}^{n} p_i}$, which compares the speedup we actually got to an upper bound of the speedup we could have got. The markers $+$, $\times$, and $\square$ denote execution on cluster, Origin, and in the mixed configuration, respectively.

Analyzing the experimental data gives some interesting results. Most obviously, the speedups for larger examples is better than with smaller ones; for instance, in Example 1 the Cluster/mixed configuration gives an absolute speedup of 16 but only a speedup of 5 for Example 2. The Origin operates in Example 1 with scaled efficiencies close to 1 and gives in Example 4 (which has very large intermediate data) due to its high-bandwidth interconnection fabric for smaller processor numbers significantly better results than the other environments. Especially with 16 to 24 processors, however, in all examples the scaled speedups/efficiencies of the cluster compete with (are equal or higher than) those of the Origin. Moreover, the cheap Linux PCs in the cluster give overall better performance than the much more expensive Silicon Graphics machines.

When we consider the execution times of the parallel subalgorithms (not listed due to lack of space) individually, we realize that the 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 (cluster), 10.2 (Origin), and 7.9 (mixed). The parallelization of the checking phase gives absolute speedups of 12.3, 14.1, and 16.1 of 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.

7 Conclusions

We have described an algorithm for reliably plotting plane algebraic curves which is based on both, symbolic and numerical methods. Analyzing the algorithm carefully, we have found sequential algorithmic improvements significantly reducing computing time.

Our parallelization of the symbolic part of the `pacPlot` algorithm demonstrates that also in the area of symbolic computation significant absolute speedups can be achieved. This was only possible after careful analysis and redesign of the original sequential algorithms. Also we see that computer networks can give speedups that are comparable to those on a massively parallel multiprocessor. Subtle algorithmic differences between the parallel and sequential version of the program give super-linear speedups in certain situations.

References

[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).

[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).

[Kal90] M. Kalkbrener. Solving Systems of Bivariate Algebraic Equations by Using Primitive Polynomial Remainder Sequences. Technical report, Research Institute for Symbolic Computation (RISC), 1990.

[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).

[Mit00] Christian Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Master's thesis, Johannes Kepler University, Linz, Austria, 2000.

[MW96] M. Mňuk 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 Euro-conference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, (1999). Civil-Comp Press, Edinburgh.

[SMW00] W. Schreiner, C. Mittermaier, and F. Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. Submitted for publication, 2000.

[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).