PLOTTING ALGEBRAIC SPACE CURVES BY CLUSTER COMPUTING

WOLFGANG SCHREINER, CHRISTIAN MITTERMAIER, FRANZ WINKLER
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Email: FirstName.LastName*@risc.uni-linz.ac.at*
http://www.risc.uni-linz.ac.at *

We describe a parallel solution to the problem of reliably plotting a space curve defined by the intersection of algebraic surfaces. 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 evaluate the performance of the program on a cluster of workstations and PCs, on a massively parallel multiprocessor, and on a cluster that couples workstations and multiprocessor.

1 Introduction

We describe a parallel solution to the problem of reliably plotting an algebraic space curve. The starting point of our work is the software library CASA (computer algebra software for constructive algebraic geometry) which has been developed since 1990 by various researchers under the direction of the third author[1]. CASA is built on top of the computer algebra system Maple.

For parallelizing a number of CASA functions, we have developed "Distributed Maple", a system that allows to implement parallel algorithms in Maple and to execute them in any networked environment[2]. The core of this system is a scheduler program written in Java which is in charge of coordinating the activities of multiple Maple kernels running on various machines. The system has evolved from our own experience in the development of parallel computer algebra environments[3] and from the experience of other researchers in adapting Maple to the use on multiprocessors and computer networks[4,5,6,7].

The presented solution is derived from a sequential algorithm that applies a hybrid combination of symbolic and numerical methods. A related solution for the plotting of plane curves was presented elsewhere[8], the algorithm for space curves is based on a similar concept but the details and the parallelization are crucially different. In the following sections, we present the problem, describe the sequential algorithm, illustrate the parallelization idea, describe the parallel programming environment, and present experimental results.

*Supported by the FWF projects P11160-TEC (HySaX) and SFB F013/F1304.

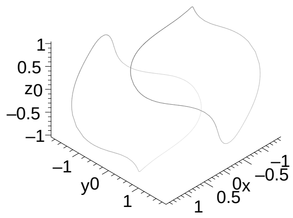


Figure 1. CASA `ssiPlot`

2 Plotting Surface-to-Surface Intersections

The basic objects of the software library CASA are algebraic sets represented e.g. as systems of polynomial equations[1]. Algebraic sets represented by trivariate polynomials model surfaces in the Euclidean space; intersections of two such surfaces define space curves. An important problem is the *reliable* plotting of such curves. Conventional numerical methods often yield qualitatively wrong solutions, i.e., plots where some "critical points" (e.g. singularities) are missing. On the contrary, the CASA function `ssiPlot` produces reliable plots such as the one of the curve defined by the intersection of the surfaces $(x^2+y^2+z^2)^2-4x^2-4y^2$ and x^2+xy^2z-1 shown in Figure 1.

`ssiPlot` applies a hybrid combination of exact symbolic algorithms for the computation of all critical points and of fast numerical methods for the interpolation between these points[9]. Still the computation time of the symbolic methods dominates the total plotting time and even on fast workstations only curves with total degree up to 5 or so can be plotted in acceptable time. Our goal is therefore to speed up this function by parallelization.

`ssiPlot` proceeds in three steps to plot a space curve a (see Figure 2 that illustrates the idea with a plane curve):

1. Compute the critical points of a in the z-direction and sort them according to their z-coordinates.
2. Intersect a with the horizontal planes that lie in the middle of the 3D stripes determined by the z-coordinates of the critical points.
3. Trace a from each intersection point in both directions towards the border points of the stripe.

Step 1 of the algorithm is a symbolic pre-processing phase that applies exact (arbitrary precision) arithmetic to compute those points that determine the topological structure of a. These critical points are the singularities

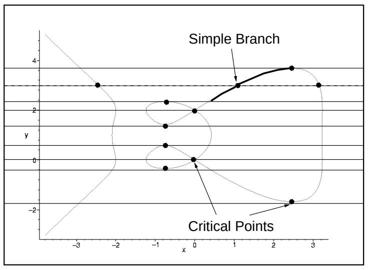


Figure 2. The Plotting Algorithm

(roughly speaking, points where two or more branches intersect each other) and the extrema in one coordinate direction. Step 2 of the algorithm uses the 3D stripes determined by these points to decompose a into branches that are "simple" in the sense that they do not self-intersect and have only points where the derivative of a does not vanish. Step 3 can safely plot each such branch by linear approximation.

The algorithm spends most of the computation time in Step 1, the computation of the critical points of a. The problem of this step is, given two surfaces $f(x, y, z), g(x, y, z) \in \mathbb{Q}[x, y, z]$, to find every real solution ("root") $(x, y, z) \in \mathbb{R}^3$ of the system $S := \{f = 0, g = 0, J_x(f, g) = 0\}$, where

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

denotes the Jacobian of f and g with respect to y and z. Since exact arithmetic in $\mathbb{R}$ is not possible, each root (x, y, z) is actually "isolated" by a triple of intervals $([x', x''], [y', y''], [z', z''])$ with $x', x'', y', y'', z', z'' \in \mathbb{Q}$ such that (x, y, z) is the only root of S with $x' < x < x''$, $y' < y < y''$, and $z' < z < z''$.

These isolating intervals are determined by first computing the determinant $j(x, y, z) \in \mathbb{Q}[x, y, z]$ of J_x, then computing the *Dixon resultant* (see below) $d(z) \in \mathbb{Q}(z)$ of f, g, and j, and finally isolating the real roots of d:

$\boldsymbol{P}$ **:= criticalPoints(**$\boldsymbol{f(x, y, z), g(x, y, z)}$**):**
  $j(x, y, z) := \det(J_x(f, g))$
  $d(z) := \text{dresultant}_{y,z}(f, g, j)$
  $P := \text{realroot}(d)$
end

```
d(z) := dixon(D):
  r := rows(D)
  c := columns(D)
  d(z) := 1
  for k from 1 to min(r, c) do
    d(z) := d(z) * D_{k,k}(z)
    for i from k + 1 to r do
      t(z) := simplify(D_{i,k}(z) / D_{k,k}(z))
      for j from k + 1 to c do
        D_{i,j}(z) := simplify(D_{i,j}(z) - t * D_{k,j}(z))
      end
      D_{i,k}(z) := 0
    end
  end
end
```

Figure 3. Computation of Dixon Resultant

The time-critical part of this algorithm is the computation of the Dixon resultant which eliminates several variables of a polynomial system while preserving the common roots. The Dixon resultant $d(z)$ is the determinant of the Dixon matrix D which is formed from the coefficients of the so called Dixon polynomial which in turn is constructed from the input polynomials f, g, and j (for mathematical details, see the diploma thesis of the second author[10]).

The Dixon resultant is computed from the Dixon matrix D by Gaussian elimination as depicted in Figure 3. In this subalgorithm of the critical point computation, `ssiPlot` spends 60–80% of the total execution time. It is this function that we most urgently need to speed up by parallelization.

3 The Parallelization Approaches

In order to speed up `ssiPlot`, we may apply parallelism in various subalgorithms (with focus on the computation of the Dixon resultant).

Dixon Resultant The only significant potential for parallelism in the computation of the Dixon resultant $d(z)$ from the Dixon matrix D is in computing, in iteration k of the outer loop, all the $(k-r)*(k-c)$ new elements of D in parallel. Having selected the number of tasks t (which is usually greater than the number of processors to improve load balancing), we thus start a

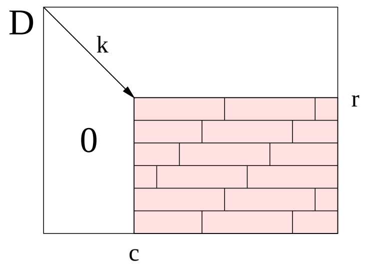


Figure 4. Dixon Resultant Tasks

task for every $\frac{(r-k)*(c-k)}{t}$ elements (see Figure 4). While the number of matrix elements per task thus decreases with increasing k, the coefficient sizes of the corresponding polynomials increase and polynomial arithmetic ($-$, $*$) and polynomial simplification ("simplify") become more complex. Thus the overall task grain size actually *grows* which allows to exploit the available parallelism efficiently.

Real Root Isolation Another building block (which accounts for about 10% of the computation time) is the isolation of the real roots of the Dixon resultant $d(z)$ by Uspensky's method, a recursive divide and conquer search. A naive parallelization of this algorithm yields poor speedups due to the narrowness of the highly unbalanced search tree. A parallelization of the arithmetic in each recursive invocation step is only feasible for tightly coupled multiprocessors. Therefore our approach is to broaden and to flatten the search tree (to a certain extent) by applying speculative parallelism[10].

Initial Points The second step of `ssiPlot` after the computation of the critical points is the computation of the intersection points of the curve and of the horizontal planes determined by the critical points as starting points for the visualization. The basic problem in this step (which accounts for another 10% of the total computation time) is, after the isolating intervals for the x-coordinates and the isolating intervals of the y-coordinates of these points have been determined, to find the matching interval pairs. This requires the testing of all y intervals with respect to every x-interval; these tests are independent of each other and can be performed in parallel.

Simple Branches In large examples, also the generation of the simple branches originating in the initial points contributes significantly to the total execution time; these branches can be generated independently in parallel.

Based on these four ideas, we have implemented in Distributed Maple a parallel variant of `ssiPlot`.

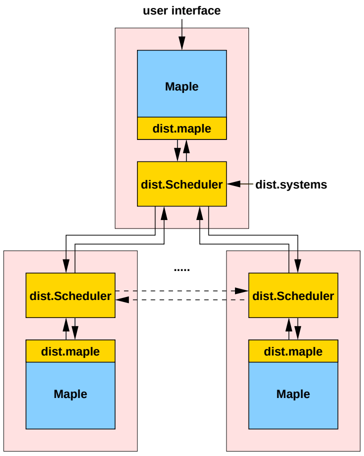


Figure 5. Distributed Maple Architecture

4 Distributed Maple

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple[2]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node connected to a session comprises two components (see Figure 5):

Scheduler The Java program `dist.Scheduler` coordinates node interaction. The initial process (created by the Maple kernel attached to the user frontend) reads all system-specific information from file `dist.systems`; it then starts instances of the scheduler on other machines.

Maple Interface The file `dist.maple` read by every Maple kernel implements the interface between kernel and scheduler. Both components use pipes to exchange messages which may embed any Maple objects.

After a session has been established, every scheduler instance accepts tasks from the attached computation kernel and schedules these tasks among all machines connected to the session. During the session, additional socket connections between remote scheduler instances are established on demand.

The user interacts with Distributed Maple via a (text or graphical) Maple frontend. Maple commands establish a session in which tasks are created for execution on any connected machine, e.g.:

```
> read `dist.maple`;
Distributed Maple V1.0.7 (c) 1998 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[gintonic,linux], [andromeda,octane]]):
connecting gintonic...
connecting andromeda...
> dist[all]("Integrate := proc(e, x) int(e, x) end:"):
> t1 := dist[start](Integrate, x^n, x):
> t2 := dist[start](Integrate, x^n, n):
> dist[wait](t1) + dist[wait](t2);
```

The file `dist.maple` implements the interface to the distributed backend as a Maple package `dist`. The command `dist[initialize]` asks the system to start the distributed backend and create two additional Maple kernels on machines `gintonic` `andromeda`, respectively.

The following operations represent the core programming interface:

`dist[all](`*`command`*`)` lets the statement *command* be executed on every Maple kernel connected to the session. If this command refers to a particular file, this file must be visible on every machine of the session.

`dist[start](`*`f`*`, `*`a`*`, ...)` creates a task evaluating *f(a, ...)* and returns a task reference t. Tasks may create other tasks; arbitrary objects (including task references) may be passed as arguments and returned as results.

`dist[wait](`*`t`*`)` blocks the execution of the current task until the task represented by t has terminated and returns its result. Multiple tasks may independently wait for and retrieve the result of the same task t.

This parallel programming model is based on functional principles which is sufficient for many kinds of computer algebra algorithms[3]. The environment also supports non-deterministic task synchronization for speculative parallelism and shared data objects which allow tasks to communicate via a global store (distributed across the network).

5 Experimental Results

We have systematically benchmarked the parallel variant of `ssiPlot` implemented in Distributed Maple with four random curves for which the sequential program takes 2119s, 475s, 1523s, and 5973s, respectively. These times refer to the execution on a PIII@450MHz PC which is for this computation the fastest machine available to us. The parallel variant has been executed in three system environments consisting of 24 processors each:

- The cluster of our institute which comprises 4 Silicon Graphics Octanes (2 R10000@250Mhz each) and 16 Linux PCs (various Pentium processors) linked by a 100 Mbit switched (full duplex point-to-point) Ethernet;
- a Silicon Graphics Origin multiprocessor (64 R12000@300 Mhz, 24 processors used) with 20 GB main memory located at the university campus to which our network is connected by a high-bandwidth ATM line;
- a mixed configuration consisting of 8 machines from the cluster and 16 processors of the Origin multiprocessor.

The total computational performance (as measured by a representative Maple benchmark) of cluster, Origin, and mixed configuration is 19, 22.6, and 22.6 times the one of a PIII@450MHz processor. In the cluster and in the mixed configuration, the frontend Maple kernel was executed on an Octane.

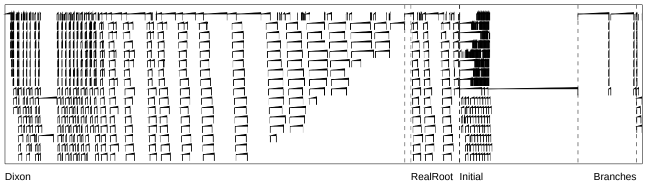
The top diagram in Figure 6 generated from a trace file illustrates a sample execution in the cluster configuration with 24 processors; we see the Dixon resultant computation phase followed by the phases for real root isolation, initial point computation, and branch computation. The number of tasks in the later stages of the first phase is limited by the number of matrix elements to be updated. The increased complexity of each matrix element computation however yields tasks of much larger grain size than in the beginning.

The table lists the execution times for each input in each environment with varying processor numbers. The first row of diagrams illustrates the absolute speedups $\frac{T_s}{T_n}$, the second row illustrates 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 illustrates the scaled efficiency $\frac{T_s}{T_n \sum_{i=1}^{n} p_i}$. The markers $+$, $\Box$, and $\times$ denote execution on cluster, Origin, and in the mixed configuration, respectively.

Apparently, the speedups for larger examples are better than for smaller ones; for Example 1, the Cluster/mixed configuration gives an absolute speedup of 12 but only a speedup of less than 5 for the small Example 2. Example 3 gives a speedup of 10 and the large Example 4 a speedup of 12.

Actually, the examples yield *too* good results: for processor numbers up to 4 or even 8, we get scaled efficiencies between 1 and 2. These super-linear speedups are caused by the increased amount of heap memory in all Maple kernels which reduces the garbage collection times correspondingly. Only for larger processor numbers, the parallelization overhead outweighs these gains.

In most cases the cluster/mixed configuration gives results that are as good as or even better than on the Origin. This indicates that the capacity of the 100 Mbit switched Ethernet is for this computation sufficiently high and/or that Distributed Maple on the Origin is comparatively less efficient.



Example	System	1	2	4	8	16	24
1 (2119s)	Cluster	4166	935	522	280	216	184
	Origin	2053	784	459	330	269	242
	Mixed	4166	935	499	280	241	200
2 (475s)	Cluster	677	237	143	98	96	104
	Origin	420	227	172	169	156	155
	Mixed	677	237	146	95	93	98
3 (1523s)	Cluster	2725	649	332	213	178	163
	Origin	1430	696	370	312	245	223
	Mixed	2725	649	334	242	175	157
4 (5973s)	Cluster	12026	2152	1089	638	639	560
	Origin	5764	2297	1228	754	543	493
	Mixed	12026	2152	1028	683	601	478

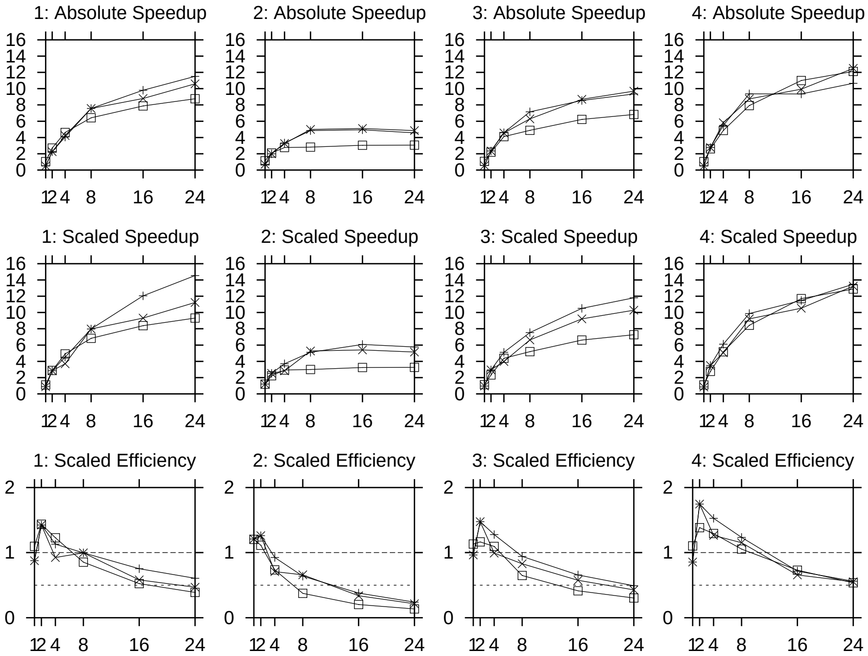


Figure 6. Experimental Results

6 Conclusions

Also in distributed environments the parallel implementation of a computer algebra algorithm can achieve significant speedups which compete with those on a massively parallel multiprocessor. Our work presents the third major function of the CASA library (besides the plotting of a plane algebraic curve and the computation of its neighborhood graph) that we have parallelized on the basis of Distributed Maple (together with a number of parallelized Maple functions). We will continue in this direction to yield a library of parallel implementations of CASA functions.

References

1. M. Mnuk and F. Winkler. CASA - A System for Computer Aided Constructive Algebraic Geometry. *DISCO'96*, volume 1128 of *LNCS*, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.
2. Wolfgang Schreiner. Developing a Distributed System for Algebraic Geometry EURO-CM-PAR'99, Weimar, Germany, March 20-25, 1999, Barry H.V. Topping (ed.), Civil-Comp Press, Edinburgh, pp. 137-146.
3. Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of the SACLIB/PACLIB Kernels. *JSC*, 19:111–132, 1995.
4. Bruce Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. *ISSAC'90*, pages 96–103, Tokyo, Japan, August 20–24, 1990. ACM Press.
5. Kurt Siegl. Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. *Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press.
6. Angel Diaz and Erich Kaltofen. FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation. O. Gloor, editor, *ISSAC'98*. ACM Press, New York, 1998.
7. Laurent Bernadin. Maple on a Massively Parallel, Distributed Memory Machine. *PASCO'97*, pages 217–222, Maui, Hawaii, July 20–22, 1997.
8. Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing, Euro-Par, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.
9. Tran Quoc Nam. *A Hybrid Symbolic-Numerical Approach in Computer Aided Geometric Design (CAGD) and Visualization.* PhD thesis, Johannes Kepler University, Linz, Austria, 1996.
10. Christian Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Diploma thesis, Johannes Kepler University, Linz, 2000.