On Solving a Problem in Algebraic Geometry by Cluster Computing (Long Version)*

Wolfgang Schreiner, Christian Mittermaier, 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 solution to the problem of reliably plotting a plane algebraic curve. The sequential program is implemented in the software library CASA on top of the computer algebra system Maple. The parallel version is based on Distributed Maple, a parallel programming extension written in Java. We evaluate its performance 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 a plane algebraic 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 [7]. CASA is built on top of the computer algebra system Maple.

In order to parallelize 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 [9]. 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 [5] and from the experience of other researchers in adapting Maple to the use on multiprocessors and computer networks [2, 11, 3, 1].

The presented solution is derived from a sequential algorithm that applies a hybrid combination of symbolic and numerical methods. The parallelization of symbolic methods differs crucially from that of numerical ones: it is very hard to predict the size of intermediate objects and load balancing is extremely difficult. In Section 2, we illustrate the problem and describe the sequential algorithm. In Section 3, we sketch the basic parallelization ideas. Section 4 reviews the main features of Distributed Maple that we have used for the parallel implementation. In Section 5, we present experimental results which demonstrate the performance of the solution in various environments. In Section 6, we summarize our results.

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

2 Plotting Algebraic Curves

The basic objects of the software library CASA are algebraic sets represented e.g. as systems of polynomial equations [7]. Algebraic sets represented by bivariate polynomials model plane 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. For instance, the left diagram in Figure 1 shows a plot of the curve $2x^4 - 3x^2y + y^4 - 2y^3 + y^2$ generated by the Maple function `implicitplot`. The numerical approximation fails to capture two singularities; even if we improve the quality of the diagram by refining the underlying grid, only one of the missing singularities emerges.

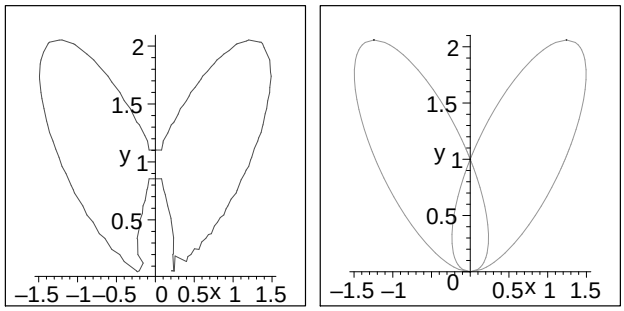


Fig. 1. Maple `implicitplot` versus CASA `pacPlot`

On the other hand, the CASA function `pacPlot` produces the correct diagram shown to the right. This is achieved by 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 [8]. However, the computation time of the symbolic methods dominates the total plotting time and even on fast workstations only curves with total degree up to 8 or so can be plotted in a time that is acceptable for interactive usage. Our goal is therefore to speed up this function by parallelization.

`pacPlot` proceeds in three steps to plot an algebraic curve a (see Figure 2):

1. Compute the critical points of a in the y-direction and sort them according to their y-coordinates.
2. Intersect a with the horizontal lines that lie in the middle of the stripes determined by the y-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 topo-

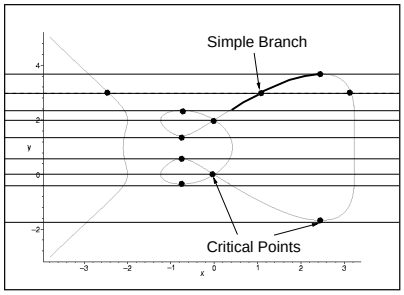


Fig. 2. The Plotting Algorithm

logical structure of a. These critical points are the singularities (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 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.

`pacPlot` spends virtually all computation time in Step 1, the computation of the critical points of a. The problem of this step is, given some $a(x, y) \in \mathbb{Q}[x, y]$, to find every real solution ("root") $(x, y) \in \mathbb{R} \times \mathbb{R}$ of the system $S := \{a = 0, \frac{\partial a}{\partial x} = 0\}$. Since exact arithmetic in $\mathbb{R}$ is not possible, each root (x, y) is actually "isolated" by a pair of intervals $([x', x''], [y', y''])$ with $x', x'', y', y'' \in \mathbb{Q}$ such that (x, y) is the only root of S with $x' < x < x''$ and $y' < y < y''$.

A sketch of the algorithm computing the set of all intervals that isolate the critical points is depicted in Figure 3 (for mathematical details, see [6]):

1. We triangulize the system into a set of systems $\{p'(y), q(x, y)\}$ with $p'(y) \in \mathbb{Q}[y], q(x, y) \in \mathbb{Q}[x, y]$ that preserves the roots. Each polynomial $p'(y)$ is factorized over $\mathbb{Q}$ into a set of $p(y)$; it suffices to solve each system $\{p(y), q(x, y)\}$.
2. We determine the resultant $r(x) \in \mathbb{Q}[x]$ of $\{p(y), q(x, y)\}$. All solutions (x, y) of $\{p(y), q(x, y)\}$ must be in $X \times Y$ where X is the set of solutions of $r(x)$ and Y is the set of solutions of $p(y)$. We thus compute isolating intervals for the real roots of $r(x)$ and $p(y)$ separately.
3. For every combination of intervals $(x, y) \in X \times Y$, we test whether (x, y) indeed isolates a solution of $\{p(y), q(x, y)\}$. This requires to compute, for every candidate $x \in X$, two polynomials $q'(y) \in \mathbb{Q}[y]$ and $q''(y) \in \mathbb{Q}[y]$ and to test every candidate $y \in Y$ with respect to these polynomials.

After the isolating intervals for the critical points have been determined, they need to be appropriately refined: given the isolating intervals $([x', x''], [y', y''])$, we decrease their widths such that $x'' - x' < e$ and $y'' - y' < e$ for a given $e \in \mathbb{Q}, e > 0$.

```
P := criticalPoints(a(x, y)):
  P := ∅
  S := {(p(y), q(x, y)) |
    ∃p'(y) : (p'(y), q(x, y)) ∈ triangulize(a(x, y), ∂a(x,y)/∂x), p(y) ∈ factor(p'(y))}
  for (p(y), q(x, y)) ∈ S do
    r(x) := resultant_x(p(y), q(x, y))
    X := realroot(r(x))
    Y := realroot(p(y))
    for x ∈ X do
      q'(y) := squarefree(q(x, y), x.0, p(y))
      q''(y) := squarefree(q(x, y), x.1, p(y))
      for y ∈ Y do
        if test(q'(y), q''(y), y, p(y)) then P := P ∪ {(x, y)}
      end
    end
  end
end
```

Fig. 3. Computation of Critical Points

The above presentation for the isolation and subsequent refinement of intervals for the critical points of an algebraic curve contains various improvements of the algorithm that was originally used in `pacPlot`. These improvements have been found by the second author in the course of his diploma thesis [6]; they already reduce the computation time of `pacPlot` by an order of magnitude. The experimental results presented in Section 5 thus measure the parallelization speedups against a new version of `pacPlot` that uses this improved algorithm.

3 The Parallelization Approaches

Investigating the dynamic behavior of `pacPlot`, it turns out that it does not pay off to parallelize the outermost loop which iterates over all systems in S: typically there are not more than two systems and only one system is not immediately recognized as trivial. Therefore we apply parallelism on several other levels:

1. parallel resultant computation,
2. parallel real root isolation,
3. parallel solution test,
4. parallel interval refinement.

Parallel Resultant Computation A single resultant computation may take a good deal of the computation time of the whole algorithm. Our parallelization approach applies a modular method to compute resultants in multiple homomorphic images $\mathbb{Z}_{n_i}[x, y]$ where the n_i are a set of prime numbers. We thus get a divide and conquer structure where both the divide phase (the modular resultant computation) and the conquer phase (the "Chinese Remaindering" computation) can be parallelized as illustrated in Figure 4. For details, see [4, 10].

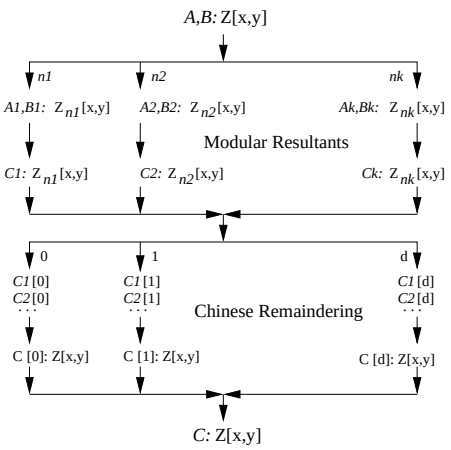


Fig. 4. Parallel Resultant Computation

Parallel Real Root Isolation The isolation of the real roots of the resultant by Uspensky's method, a recursive divide and conquer search algorithm, may also take a significant computation time. A naive parallelization of this algorithm typically 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. For details, see [6].

Parallel Solution Test and Interval Refinement The tests which of the candidates (x, y) are indeed solutions of the given system can be performed in parallel in a straight-forward fashion. Likewise, we can apparently refine all isolating intervals in parallel to the desired accuracy.

We have implemented all four ideas on the basis of Distributed Maple.

4 Distributed Maple

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple developed by the first author [9]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node of a session comprises two components (see Figure 5):

Scheduler The Java program `dist.Scheduler` coordinates the interaction between nodes. The initial scheduler process (created by the Maple kernel attached to the user frontend) reads all application-specific information from the configuration file `dist.systems`; it then starts instances of the scheduler on other machines and communicates with them via sockets.

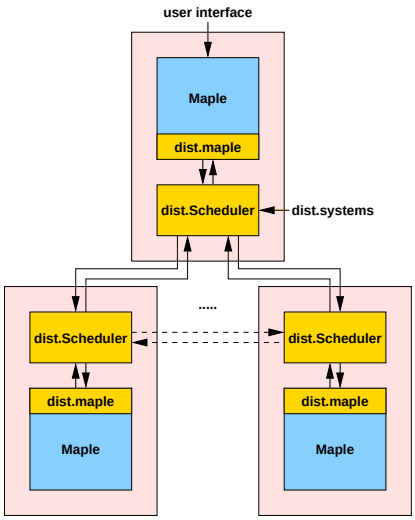


Fig. 5. Distributed Maple Architecture

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 in the compact linear format that Maple uses for library files).

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.

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. The session trace below demonstrates the use of the environment by a simple example:

```
> 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);
                     (n + 1)     n
                    x           x
                    -------- + -----
                     n + 1     ln(x)
```

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 machine `gintonic` and on machine `andromeda`, respectively (the keys `linux` and `octane` are used to lookup the machine-specific information stored in `dist.systems`).

The following operations represent the core programming interface:

`dist[all](`*command*`)` lets the Maple 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 participating in the session.

`dist[start](`*f*`, `*a*`, ...)` creates a task evaluating *f(a, ...)* and returns a task reference *t*. Tasks may create other tasks and arbitrary Maple 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 essentially based on functional principles which is sufficient for many kinds of computer algebra algorithms [5]. In addition, the environment supports a non-deterministic form of task synchronization for speculative parallelism and shared data objects which allow tasks to communicate by side effects on a global store.

5 Experimental Results

We have systematically benchmarked the parallel variant of `pacPlot` implemented in Distributed Maple with four randomly generated algebraic curves for which the sequential program takes 6870s, 470s, 155s, and 11748s respectively. These times refer to the execution on a PIII@450MHz PC which is for this computation the fastest machine available to us.

More precisely, the given numbers denote the sequential execution of the parallel algorithm which turns out to take for the four examples 76%, 63%, 98%, 101% of the execution time of the sequential algorithm! A more detailed benchmark and subsequent analysis of the algorithm explains this as a consequence of the fact that the speculative parallel real root isolation computes narrower intervals than its sequential counterpart. While this is more costly, the subsequent detection of matching combinations of x-intervals and y-intervals becomes much faster, which yields in two of the examples a signficant improvement over the sequential algorithm. For all further comparisons, we therefore use the sequential execution of the parallel algorithm as the comparison basis.

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 (point-to-point) Ethernet;

- a Silicon Graphics Origin multiprocessor (64 R12000@300 Mhz, 24 processors used) located at the university campus to which the network of our institute is connected by a high-bandwidth ATM line;
- a mixed configuration consisting of our 4 dual-processor Octanes and 16 processors of the Origin multiprocessor.

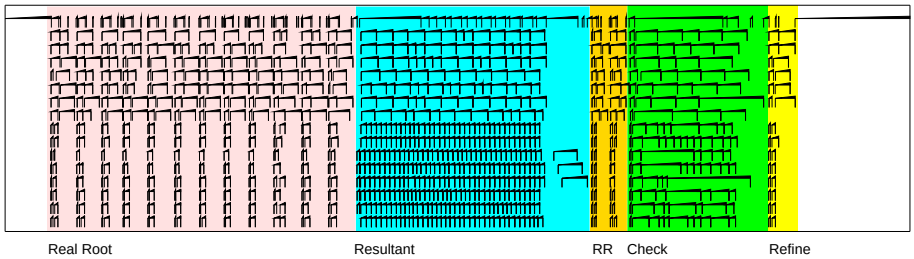
The raw computing powers of cluster, Origin, and mixed configuration are 18.3, 17.1, and 18.7, respectively; these numbers (measured by a representative Maple benchmark) denote the sum of the raw performances of all processors relative to a PIII@450MHz processor. In the cluster and in the mixed configuration,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

The profile in Figure 6 illustrates the execution of Example 2 in a cluster 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 can clearly distinguish the real root isolation phase followed by the phases for resultant computation, the second real root isolation, the solution checks and the solution refinements (their relative weights are very different for other inputs).

The table in Figure 6 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 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.



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

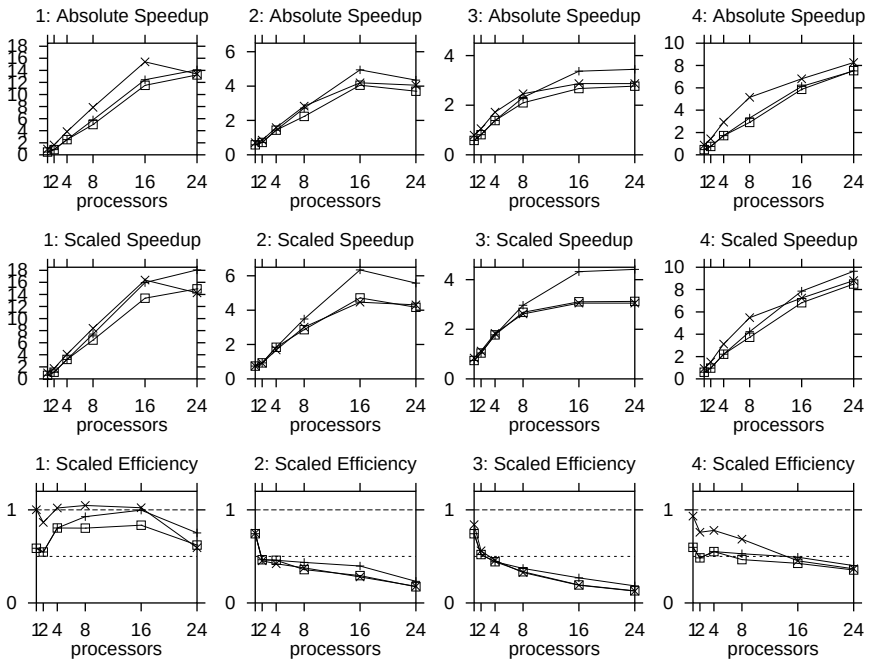


Fig. 6. Experimental Results

6 Conclusions

Our parallelization of a computer algebra algorithm demonstrates that also for symbolic computation significant absolute speedups can be achieved in distributed and heterogeneous environments. Moreover, we see that execution in a computer network can give speedups that are comparable to those on a massively parallel multiprocessor. Only for very large intermediate data the much higher communication performance of the Origin multiprocessor significantly overrules the Ethernet-connected cluster of workstations and PCs. Moreover, the parallel version exhibits an improvement over the original sequential algorithm.

References

[1] L. Bernadin. Maple on a Massively Parallel, Distributed Memory Machine. In *PASCO'97 — Second International Symposium on Parallel Symbolic Computation*, pages 217–222, Maui, Hawaii, July 20–22, 1997. ACM Press, New York.

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

[3] A. Diaz and E. Kaltofen. FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation. In O. Gloor, editor, *ISSAC'98 International Symposium on Symbolic and Algebraic Computation*. ACM Press, New York, 1998.

[4] H. Hong and H. W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In *CONPAR 94 - VAPP VI — Third Joint Internati onal Conferences on Vector and Parallel Processing*, volume 854 of *LNCS*, pages 325–336, Linz, Austria, September 6–8, 1994. Springer.

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

[6] C. Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Master's thesis, Johannes Kepler University, Linz, Austria, 2000. To appear.

[7] 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.

[8] T. Q. Nam. Extended Newton's Method for Finding the Roots of an Arbitrary Systemof Equations and its Applications. In *IASTED'94: 12th International Conference on Applied Informatics*, Annecey, France, 1994.

[9] 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.

[10] W. Schreiner. Developing a Distributed System for Algebraic Geometry. In B. H. 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.

[11] 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.