Preliminary Version: April 2, 1999
Final version to appear in: EURO-CM-PAR'99 Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics, Weimar, Germany, March 20–25, 1999, Barry H.V. Topping (ed.), Civil-Comp Press, Edinburgh.

DEVELOPING A DISTRIBUTED SYSTEM FOR ALGEBRAIC GEOMETRY*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

*Supported by the Austrian Science Foundation (FWF) grant P11160-TEC (HySaX).

Abstract

We report on a project on parallelizing parts of the software library CASA for computing with and reasoning about objects in algebraic geometry with applications in computer aided geometric design and solid modeling. CASA is implemented in the language of the computer algebra system Maple; we therefore have developed "Distributed Maple" as an environment for executing parallel Maple programs on multiple computers of a network. Using this environment, we have started the parallelization of some important CASA algorithms; for a particular subproblem we achieve a speedup of 15 on a heterogeneous network with 24 processors distributed among PCs, workstations, and shared memory multiprocessors.

1 Introduction

This paper describes an ongoing effort on developing a distributed system for solving problems in algebraic geometry with applications in computer aided geometric design and solid modeling. The starting point of our work is the software library CASA (computer algebra software for constructive algebraic geometry) which has been developed by various researchers at RISC-Linz since 1990 [1]. The library is written in the language of the computer algebra system Maple [2] and currently consists of about 800 functions of which about 80 are visible to the user.

In order to parallelize a number of CASA functions, we have developed a system called "Distributed Maple" that allows to implement parallel algorithms in Maple and that can be easily installed in any networked environment [3]. The core of this system is an external scheduler program that is in charge of coordinating the activities of multiple Maple kernels running on various machines connected via a local or wide area network. The system has evolved from our own experience in the development of parallel computer algebra environments [4, 5] and from the experience of other researchers in adapting Maple to the use on multiprocessors and computer networks [6, 7, 8, 9, 10].

In contrast to some other approaches, our goal is to provide an environment where parallel programming is possible *within* Maple such that the mathematical programmer does not have to leave the familiar environment of the computer algebra system. Furthermore, the underlying technological basis should be "time-safe" and accessible to a wide audience such that the resulting code can be easily exchanged and used for the foreseeable future.

We have tackled this goal by developing a simple but easily configurable communication and scheduling program that starts and connects external computation kernels on various machines. This program is written in Java and can be executed on any machine running some implementation of the Java Virtual Machine. We have written a small Maple package that implements an interface to the scheduler and provides a high-level parallel programming model for Maple. Both the scheduler (compiled Java classes) and the Maple interface (Maple source code) are available at

`http://www.risc.uni-linz.ac.at/software/distmaple.`

In the following sections, we illustrate the application context of our work by showing the CASA solution to a particular problem, describe the interface and implementation of Distributed Maple, and demonstrate the use of the system by parallelizing a part of the CASA solution. We conclude the paper by outlining our further work on the development of a parallel variant of CASA.

2 CASA

The software library CASA is designed for computing with and reasoning about objects in classical algebraic geometry (affine and projective algebraic geometry over an algebraically closed field F of characteristic 0). The objects that CASA deals with are algebraic sets in various representations, e.g., as the set of common zeros of a system of polynomial equations. Algebraic sets in two variables represent curves in the plane; algebraic sets in three variables represent surfaces in space (or intersections of such, i.e. space curves). Figure 1 illustrates plots of the planar curve represented by the polynomial $y^4 - 96a^2y^2 + 100a^2x^2 - x^4$ and of the space curve represented by $y^4 - 384y^2 + 400x^2 - x^4 + z^3, x^2 + y^2 + z^2 - 30$, respectively. For a detailed description of CASA, see [1].

One important problem that CASA solves is the *reliable* plotting of algebraic curves. Purely numerical methods may and often do yield qualitatively wrong solutions, i.e., plots where some "critical points" (e.g., singularities) disappear. The CASA functions `pacPlot` for plotting plane curves and `ssiPlot` for plotting intersections of space

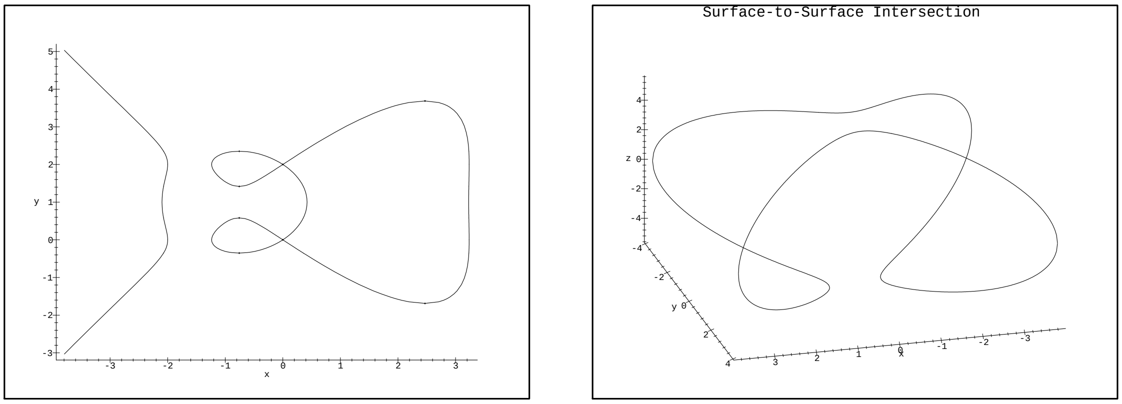

Figure 1: Plotting Algebraic Curves

curves apply 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 [11]. 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 12 or so can be plotted by CASA in a time acceptable for interactive usage.

Currently, we are working on the parallelization of the function `pacPlot` which operates in three phases:

1. (Symbolic) Compute all extremal points (including singularities) of the curve in direction x. Each interval on the x-axis between two extremal points defines one or more "simple" paths of the curve.

2. (Symbolic) For each simple path, find an initial point between its extrema for visualizing the curve.

3. (Numerical) Use Newton iteration to trace each simple path from its initial point in both directions towards the extremal points.

The first phase is by far the most time-consuming one accounting in most cases for more than 95% of the total computation time. It is beyond the scope of this paper to explain the detailed operation of this phase but its time-relevant steps are essentially [11]:

1. Compute the resultant $C[x]$ of some polynomials $A[x, y]$ and $B[x, y]$ over the integers.

2. Isolate the real roots of $C[x]$ (i.e., determine a set of disjoint intervals such that each real root of $C[x]$ is contained in one and only one interval);

3. For each real root r of $C[x]$, isolate the real roots of $B[y \leftarrow r]$.

The structure of Step 3 lends itself immediately to a parallel implementation which we have already adopted. The parallelization of the real root isolation (Step 2 and within every task of Step 3) is currently under way. The resultant computation in Step 1 remains the single most time time-consuming operation in `pacPlot`; we will describe in Section 4 in detail a parallel solution to this problem. The seamless and efficient integration of the various parallel algorithms and parallelization layers in `pacPlot` is our future concern.

3 Distributed Maple

Distributed Maple is an environment for writing parallel programs in the computer algebra system Maple. It allows to create concurrent tasks and to execute them by Maple kernels running on various machines of a network. In the following, we are going to sketch the use of the system, its programming interface, and its software architecture. For more details, see [3].

3.1 A Sample Session

The user interacts with Distributed Maple via a conventional Maple frontend (text or graphical), i.e., she operates within the familiar Maple environment for writing and executing parallel programs. Maple commands establish a distributed session in which tasks are created for execution on any connected machine (see Figure 2). The session trace below demonstrates the use of the environment by a simple example:

```
milkyway!> maple
    |\^/|     Maple V Release 5 (Universitae...
._|\|   |/|_. Copyright (c) 1981-1997 by Wat...
 \  MAPLE  /  reserved. Maple and Maple V ar...
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.0.7 (c) 1998 Wolfgang S...
See http://www.risc.uni-linz.ac.at/software/...
> dist[initialize]
>   ([[gintonic,linux], [andromeda,octane]]);
connecting gintonic...
connecting andromeda...
                       okay
```

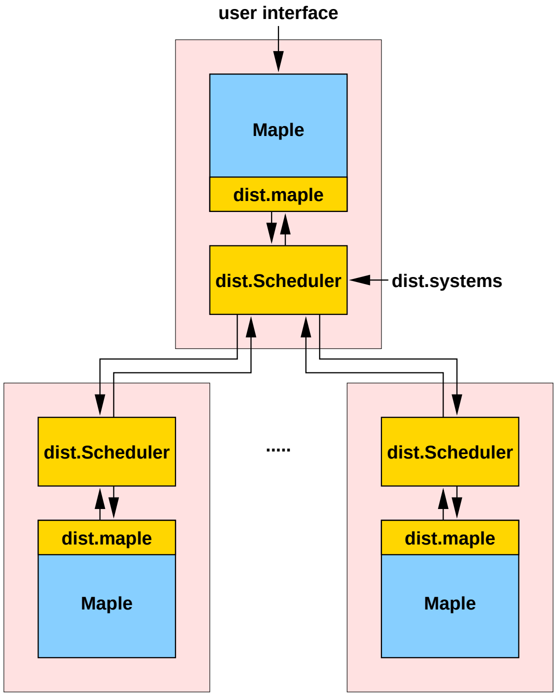

Figure 2: A Distributed Maple Session

```
> t1 := dist[start](int, x^n, x);
                          t1 := 0

> t2 := dist[start](int, x^n, n);
                          t2 := 1

> dist[wait](t1) + dist[wait](t2);
                        (n + 1)     n
                       x           x
                       -------- + -----
                        n + 1     ln(x)

> dist[terminate]();
                           okay

> quit;
```

We first load the file `dist.maple` which implements the interface to the distributed backend by a Maple package `dist`. By issuing the command `dist[initialize]`, we ask the system to start the distributed backend and create two additional Maple kernels on machine `gintonic` of type `linux` and on machine `andromeda` of type `octane`, respectively. The machine types are used to lookup the system-specific startup information which is located in a file `dist.systems` in the current working directory.

After the distributed session has been successfully established, calls of `dist[start]` create two tasks evaluating the Maple expressions `int(x^n, x)` and `int(x^n, n)`, respectively. The two `dist[wait]` calls block the current execution until the corresponding tasks have terminated and then return their results. Finally, the distributed session is closed by a call of `dist[terminate]`. If the user issues during the session the command `dist[visualize]`, a window pops up that displays online the state of each machine and the total utilization as shown in Figure 3. By issuing a command `dist[trace]` a trace file is produced that may be used after program execution to generate corresponding diagrams for printing (as those in Section 4).

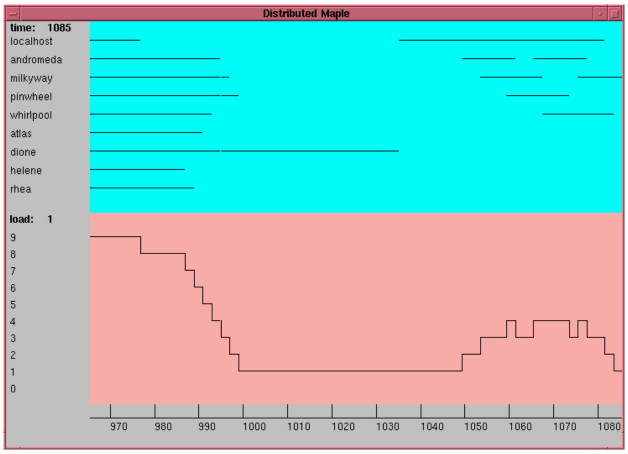

Figure 3: Session Visualization

3.2 Programming Interface

Session Initialization In addition to the commands `dist[initialize]` and `dist[terminate]` that establish a distributed session, we need a possibility to initialize the Maple kernels on all connected machines by loading user code and program libraries.

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

Functional Parallelism The parallel programming model is essentially based on functional principles which is sufficient for many kinds of computer algebra algorithms:

`dist[start](f, a, ...)` creates a task evaluating the expression $f(a, \ldots)$ and returns a reference t to this task. The execution of f may take place on any machine connected to the distributed session and should therefore not cause any side effects. Tasks may create other tasks and arbitrary Maple objects (including task references) may be passed as task arguments and returned as task 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.

When and on which machine a task is scheduled for execution is entirely in the responsibility of the underlying runtime system.

Non-Determinism and Speculation The performance of a parallel program may be improved by processing the results of a set of tasks not in a predetermined order but in the order in which they happen to arrive. Therefore we need a non-deterministic form of task synchronization; this is especially useful in speculative algorithms where the results of some tasks may turn out to be not of interest any more:

`dist[select](tlist)` blocks the execution of the current task until *any* task t in the list of task handles *tlist* has terminated and returns a list r such that $r[1]$ is the result of t and $r[2]$ is the index of t in *tlist*.

`dist[delete](t)` announces that the result of task t is not required any more. If this task has not yet started execution, it will be deleted from the system (however, if it has already started, it will continue execution until termination).

Shared Data Objects If a parallel program processes large data in multiple phases with task interaction between phases, it may be more efficient to let tasks preserve their states across phases rather than creating for every phase new tasks to which the corresponding data have to be passed. Therefore we introduce a concept that allows tasks to interact in a safe way by side effects.

`dist[data]()` creates an *empty* shared data object and returns its handle d. Any task may use d to read from or write to the shared data object no matter on which machine the task is executed.

`dist[get](d)` blocks the execution of the current task until the data object referenced by d is non-empty and then returns its content. Multiple tasks may independently wait for and retrieve the result of the same data object d.

`dist[put](d, v)` writes the value v (which may be any Maple object including tasks and data handles) into the shared data object referenced by d (overwriting any previously written value). All tasks blocked on d get released.

`dist[clear](d)` empties the shared data object referenced by d.

Shared data objects may be used to implement various forms of inter-task communication, such as shared memory, single assignment objects, communication channels and non-strict lists (streams).

3.3 Software Architecture

The core of Distributed Maple is a scheduler program which is implemented in Java (package `dist` with main class `Scheduler`) and is completely independent and even *unaware* of Maple; it can in fact embed and schedule tasks from any kind of computation kernels that implement a specific communication protocol. Correspondingly each node connected to a Distributed Maple session comprises two components (see Figure 2):

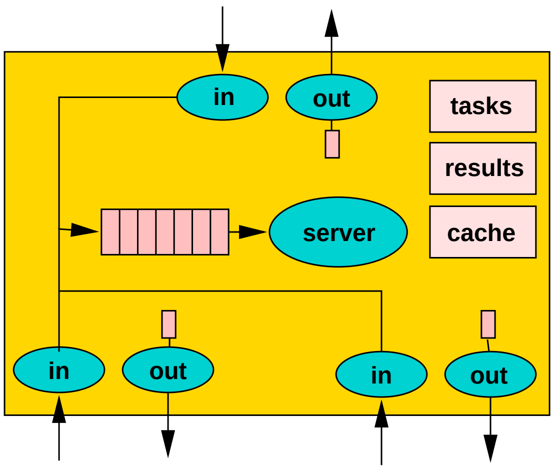

Figure 4: The Task Scheduler

Scheduler This program coordinates the interaction between nodes and schedules tasks among nodes. The initial scheduler process (invoked from 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 Internet sockets.

Maple Interface The Maple file `dist.maple` read by every Maple kernel implements the interface between Maple kernel and scheduler. Communication between both components is based on Unix pipes to which and from which messages are written; these messages may embed Maple expressions (in the compact linear format that Maple uses for library files).

After a distributed 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. A task is a pair *(taskid, exp)* where *taskid* is an integer identifying the task and *exp* is an uninterpreted string to be submitted to some kernel for evaluation. Initially, every scheduler informs its kernel about the range of task identifiers it may use for assignment to new tasks such that all kernels in the system can independently create new tasks. Each instance of the scheduler holds a local task queue, a result table, and a result cache; their function is explained below.

Multi-Threading The scheduler is implemented by a number of concurrent threads as shown in Figure 4. Threads listening on all input channels put the received messages into a central buffer from where a server thread takes and processes them and creates new messages that are placed in some of the output buffers. Threads listening on these buffers take these messages and send them to the corresponding output channels. The multithreaded implementation of the communication interface is intended to maximize the overall throughput of the system.

Task Scheduling All remote schedulers send new tasks to the central scheduler which distributes them among all machines. Currently a simple load balancing scheme is used where the central scheduler assigns new tasks to remote schedulers until the number of not completed tasks reaches an upper bound; a remote scheduler asks for new tasks whenever the number of received but not yet started tasks falls below a lower bound. By this "watermark" scheme, the communication latency for the transfer of a new task (after termination of a task) can be masked by the execution of an already received task.

Task Results When a new task is created, the scheduler allocates a corresponding "empty" slot in the result table (which will be filled with the result value when the corresponding task will have completed execution). Thus the result of a task is always stored on that machine where the task has been created (not necessarily where it is eventually executed); from the identifier of a task, the scheduler can determine the node holding a task result and correspondingly route requests. Additionally each scheduler holds a cache of all results that it has ever seen such that some requests to non-local task results can be immediately satisfied.

Recursive Server Loops If the scheduler cannot immediately deliver a task result requested by the attached computation kernel, it instead sends a new task for execution. The Maple kernel continues with the execution of the new task (by recursive invocation of the server loop) until this task has been completed. If the originally requested result is then available, the kernel continues with the execution of the previous task; otherwise it receives another task for execution. In this way every kernel (also the kernel connected to the user interface) permanently computes as long as sufficiently many tasks are available.

Watchdogs A watchdog thread in every remote scheduler controls in regular intervals if messages have been received from the central scheduler. If during the last control period no message has been received, the watchdog sends a "ping" message to the central scheduler. If during the subsequent control period no reply is received, the watchdog assumes that the connection is broken and aborts the external application process and the scheduler process. Thus we ensure that broken sessions do not lead to stalled remote processes (which becomes a problem in distributed environments).

4 Parallel Resultant Computation

As discussed in Section 2, one of the most time-consuming components of the CASA function `pacPlot` is the computation of multi-variate polynomial resultants. In this section, we are going to describe the problem, a sequential solution algorithm, a parallel version of this algorithm, and our implementation of this algorithm in Distributed Maple. We conclude by listing benchmark results that demonstrate the actual efficiency of the parallel solution.

Problem Statement Let $A = \sum_{i=0}^{m} A_i x^i$ and $B = \sum_{i=0}^{n} B_i x^i$ be non-zero polynomials over an integral domain I, i.e.,

$$A, B \in I[x]$$

The *Sylvester matrix* of A and B is the $m+n$ by $m+n$ matrix

$$\begin{bmatrix} A_m & A_{m-1} & \dots & A_0 & & \\ & \dots & \dots & \dots & \dots & \\ & & A_m & A_{m-1} & \dots & A_0 \\ B_n & B_{n-1} & \dots & B_0 & & \\ & \dots & \dots & \dots & \dots & \\ & & B_n & B_{n-1} & \dots & B_0 \end{bmatrix}$$

whose upper part consists of n rows of the coefficients of A and whose lower part consists of m rows of the coefficients of B (all entries not shown are zero). The *resultant* of A and B is the determinant of this matrix, i.e.,

$$\text{Resultant}(A, B) \in I.$$

In the context of the plotting of algebraic curves, we are interested in the special case where A and B are polynomials in r variables over the integers, i.e., given

$$A, B \in \mathbb{Z}[x_1, \dots, x_{r-1}][x_r]$$

we want to find

$$\text{Resultant}(A, B) \in \mathbb{Z}[x_1, \dots, x_{r-1}].$$

For plotting algebraic curves in two dimensions, we have to solve this problem for $r = 2$.

Sequential Algorithm Collins [12] describes an efficient sequential algorithm for multivariate polynomial computation based on a modular approach: we use various prime numbers p to map the coefficients of A and B into modular domains $\mathbb{Z}_p = \{0, \dots, p-1\}$, i.e., we compute polynomials $a = \text{Map}_p(A)$, $b = \text{Map}_p(B)$ where

$$a, b \in \mathbb{Z}_p[x_1, \dots, x_{r-1}][x_r].$$

We may only use primes p such that degree(a)=degree(A) and degree(b) = degree(B) where degree(P) denotes the degree of polynomial P in the main variable. Using modular arithmetic in $\mathbb{Z}_p$, we compute the modular resultant $c := \text{Resultant}_p(\text{a,b})$, i.e.,

$$c \in \mathbb{Z}_p[x_1, \dots, x_{r-1}].$$

We then apply the Chinese Remainder Algorithm to lift the result from $\mathbb{Z}_p$ to $\mathbb{Z}_{P*p}$, i.e.,

$$\text{ChineseRemainder}(C, c, P, p) \in \mathbb{Z}_{P*p}[x_1, \dots, x_{r-1}]$$

where P is the product of the primes used so far and C is the combination of the corresponding modular resultants:

$$C \in \mathbb{Z}_P[x_1, \dots, x_{r-1}]$$

The algorithm iterates until P exceeds a predetermined bound cb for the size of the coefficients of the resultant and then returns C. The algorithm is sketched below:

C **:= Resultant(**A**,** B**)**
  m := degree(A); n := degree(B)
  cb := CoeffBound(A, B)
  C := 0; P := 1
  while $P \leq cb$ **do**
    p := a new prime number;
    if $A_m \bmod p \neq 0 \ \wedge\ B_n \bmod p \neq 0$ **then**
      a := Map$_p$(A); b := Map$_p$(B)
      c := Resultant$_p$(a, b)
      C := ChineseRemainder(C, c, P, p)
      $P := P * p$
    end
  end
end Resultant

Parallel Algorithm Various parallel versions of the modular algorithm have been investigated and implemented on shared memory machines [13] and on workstation networks [14]. 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 to yield the integer resultant. We apply the idea used by [13] where the sequential structure of the combination phase is maintained but the individual coefficients of the resultant are computed in parallel.

The parallel algorithm starts for each modular domain a task that maps A and B into this domain and computes the coefficients of the modular resultant (up to a predetermined degree bound db). The task results are collected and for each resultant coefficient a task is started that iteratively combines the corresponding modular coefficients. The task results are collected to build the overall result.

C **:= ResultantParallel(**A**,** B**)**
  m := degree(A); n := degree(B)
  cb := CoeffBound(A, B)
  db := DegreeBound(A, B)
  C := 0; P := 1; T := [ ]; L := [ ]
  while $P \leq cb$ **do**
    p := a new prime number;
    if $A_m \bmod p \neq 0 \ \wedge\ B_n \bmod p \neq 0$ **then**
      t := start(ResultantModular, p, db, A, B)
      $P := P * p$; $T := T \parallel [t]$; $L := L \parallel [p]$
    end
  end
  R := WaitForAll(T)
  T := [ ]
  for $i \in [0 \ldots db]$ **do**
    t := start(ChineseRemainderCoeff, L,
      $[R[j]_i \mid j \in [1 \ldots \text{length}(R)]]$)
    $T := T \parallel [t]$
  end
  C := WaitForAll(T)
end ResultantParallel

The structure of the algorithm is illustrated in Figure 5.

Implementation We have implemented above algorithm for $r = 2$ using the Distributed Maple constructs `dist[start]` and `dist[wait]`. Furthermore, we have replaced the Maple function `resultant/bivariate` (which is called by Maple for computing the resultants of bivariate polynomials) by a version that uses our parallel implementation if the degrees of both input polynomials in the main variable are are greater than five and the degree of one input polynomial is greater than ten. All other cases are still solved by sequential methods.

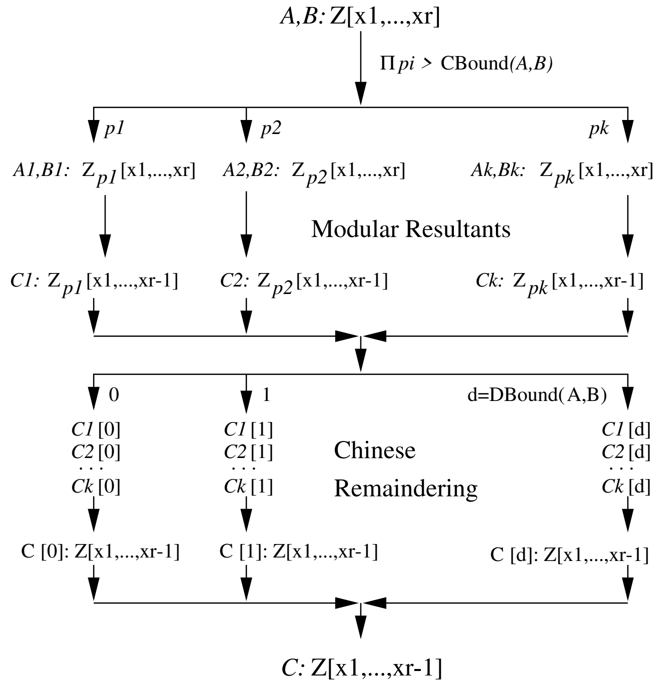

Figure 5: Parallel Resultant Computation

The implementation differs from above algorithm in that each task computes multiple modular resultants respectively multiple coefficients of the integer resultant. By adjusting the number of elements computed by a task we can effectively control its grain size.

For estimating the execution time of a "ResultantModular" task we use the complexity bound [12]

$$\begin{aligned}&(1 + \max(\text{degree}_r(A), \text{degree}_r(B)))\\ &* \textstyle\prod_{i=1}^{r}(1 + \max(\text{degree}_i(A), \text{degree}_i(B)))\\ &* \textstyle\sum_{i=1}^{r}(1 + \max(\text{degree}_i(A), \text{degree}_i(B)))\end{aligned}$$

where degree$_i(P)$ denotes the degree of P in the i-th variable. Based on this bound (multiplied by an experimentally determined constant that represents the processor speed) our implementation adjusts the number of modular images per task such that each task is expected to execute roughly 10 seconds. Smaller grain sizes unduly increase the execution overhead while larger grain sizes may cause significant load imbalances.

We also adjust the number of coefficients computed per "ChineseRemainderCoeff" task such that it processes for any input a constant number of modular coefficients. We have experimentally determined a value for this constant (1800) such that the execution time of a the task roughly matches the execution time of a "ResultantModular" task.

The first four modular resultants are actually computed sequentially in order to derive an improved value for the degree bound such that fewer coefficients have to be considered in the rest of the algorithm.

Benchmark Results We have benchmarked our implementation with numerous inputs and machine configurations; the results of some of these experiments are shown below. The system environment in which the parallel program has been executed consists of (various subsets of) the following machines:

- 8 Linux PCs (1 Pentium at 350 Mhz, 2 Pentium at 333 Mhz, 5 Pentium at 266Mhz),
- 4 Silicon Graphics Octane multiprocessors (2 R10000 at 250 Mhz each),
- 1 Sun Ultrasparc workstation (1 Sparc at 296 Mhz),
- 1 Silicon Graphics Origin multiprocessor (32 R10000 at 195MHz, 7 processors used).

The whole environment comprises 24 processors whose total performance is 18 times that of a Linux PC at 333 Mhz which serves as the reference architecture for sequential execution. The Origin multiprocessor in the university campus (20 km far from the site of our institute) is connected to the local area Ethernet of our institute (in which all other machines reside) by a high-speed ATM line.

We have applied the CASA function `pacPlot` to four algebraic curves represented by the randomly generated bivariate polynomials listed in Figure 6 and have measured the execution times of the corresponding resultant computations (using the sequential resultant function provided by Maple and our parallel version). The results are listed in Figure 7 and visualized in Figure 8.

The largest speedup was 15 (compared to a sequential execution time of 1267 seconds) with 24 processors, which corresponds to an efficiency of more than 80% (considering the total system performance). All four examples achieved an efficiency of at least 50% with 8 processors, three examples achieved an efficiency of at least 50% with 20 processors. The example that ran shortest (sequential execution time of 84 seconds) achieved a maximum speedup of 3.5 with 8 processors (50% efficiency).

The visualization of a program run shown in Figure 9 (curve 12-7-2, 24 processors) gives an impression of the dynamic behavior of the algorithm (the diagrams have been produced from a trace file generated by the scheduler on issuing the Maple command `dist[trace]`). The left figure shows the load of each machine, the middle figure displays all tasks, and the right figure shows the machine utilization during the computation of the resultant.

We see that a very short initial phase where only one machine is active (the sequential computation of four modular resultants) is followed by a long phase where all machines are active computing modular resultants in parallel. This phase is followed by a shorter phase where only part of the machines are active combining the coefficients of the modular resultants by the Chinese Remainder Algorithm. Since tasks have been calibrated to run about 10 seconds, there is not sufficiently much parallelism to keep all machines active in this phase.

The fact that some of the Chinese Remainder tasks seem to execute significantly longer than other tasks can be attributed to the fact that exactly these tasks are on Linux PCs: while the raw (integer) processing power of these machines is larger than those of all other machines, their communication bandwidth (external sockets and/or internal pipes) is apparently much lower. Since the Chinese Remaindering tasks receive very large input arguments, they start execution on these PCs later than on other machines and/or need more time to transfer their results. This is also the reason why we choose to run the initial Maple kernel of a Distributed Maple session on a (otherwise slower) Silicon Graphics workstation; this gives much better overall speedups. We are currently investigating whether the installation of newer versions of the Linux operating system kernel may help to overcome this communication bottleneck.

It is interesting to note that our implementation is based on parallelization constructs that correspond to those of the parallel computer algebra library PACLIB where a similar parallel version of the modular resultant algorithm was implemented in C [13]. Likewise this parallel algorithm was implemented in the para-functional language pD which was compiled to C code that used the PACLIB runtime system [5]. The PACLIB implementation on a shared memory multiprocessor (16 Intel386 at 20 Mhz) achieved a speedup of 11.5 for a problem whose sequential execution time was 180 seconds [13]. The pD implementation achieved a speedup of 11.5 for a problem whose execution time took 160 seconds.

We may compare these results with the resultant computation that arises in the plotting of curve 10-7-2 where the sequential Maple call takes 214 seconds and the parallel implementation in Distributed Maple on a heterogeneous computer network achieves a speedup of 8 with 16 processors and of 10 with 24 processors, respectively. We are not aware of any other reports in literature on speedups gained by parallel resultant computation in distributed environments.

5 Conclusions

The parallelization of multivariant resultant computation in Distributed Maple on a heterogenous network of PCs, workstations, and multiprocessors has shown very promising results. This algorithm is however only one (if major) component of the CASA function `pacPlot` whose speedup is our ultimate goal. We are therefore working on the parallelization of real root isolation and the seamless integration of the different parallel algorithms and parallelization layers such that the resulting code can be efficiently executed in different system environments.

Our future parallelization efforts will extend to other parts of the CASA library such as the function `ssiPlot` for plotting intersections of space surfaces [11] and the powerful CASA facilities for analyzing and manipulating algebraic sets. This will also give helpful insights for further improvements of the Distributed Maple system especially with respect to load balancing in an heterogeneous network. Combining different parallel algorithms and tuning them for different application contexts and system environments will present an interesting challenge.

Name	Tot.Deg.	Value
9-11-3	14	$23x^4y^8 - 17x^6y^6 + 22x^8 + 84x^7y^7 + 20x^4y^6 - 49x^3y^8 - 80x^4y^2 - 5x^3y - 92x^3y^2 - 47y^8 + 5xy^5$
10-7-2	13	$55x^8y^2 + 66x^9y^4 + 27y^3 + 31x^2y^5 - 75x^2y^6 - 70y^8 - 97x^2y^9$
10-8-2	18	$-22x^9y^9 - 26x^2y^5 + 85x^8y - 50x^2y^6 - 74x^6y^4 - 31x^8y^7 - 172x^7y^7$
12-7-2	20	$99xy^9 - 12x^6y^2 - 26x^2y^{11} - 47x^8y^{11} - 61x^5y^8 - 90x^9y^{11} + 94x^2y$

Figure 6: Benchmark Curves

Processors	Performance	Time (s)			
		9-11-3	10-7-2	10-8-2	12-7-2
Sequential	1	634	214	84	1267
1	0.73	958	317	122	1920
2	1.73	386	135	53	746
3	2.73	251	91	35	477
4	3.46	192	72	36	370
8	6.48	112	47	25	204
12	9.66	88	43	25	143
16	12.1	73	35	26	111
20	15.34	73	29	24	93
24	17.99	66	29	24	84

Figure 7: Benchmark Times

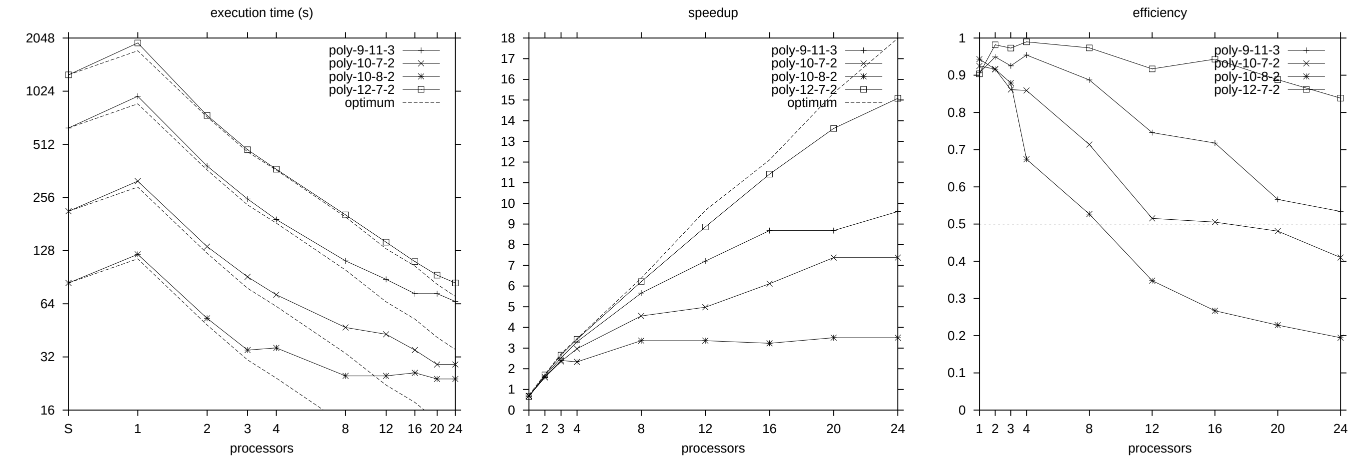

Figure 8: Benchmark Results

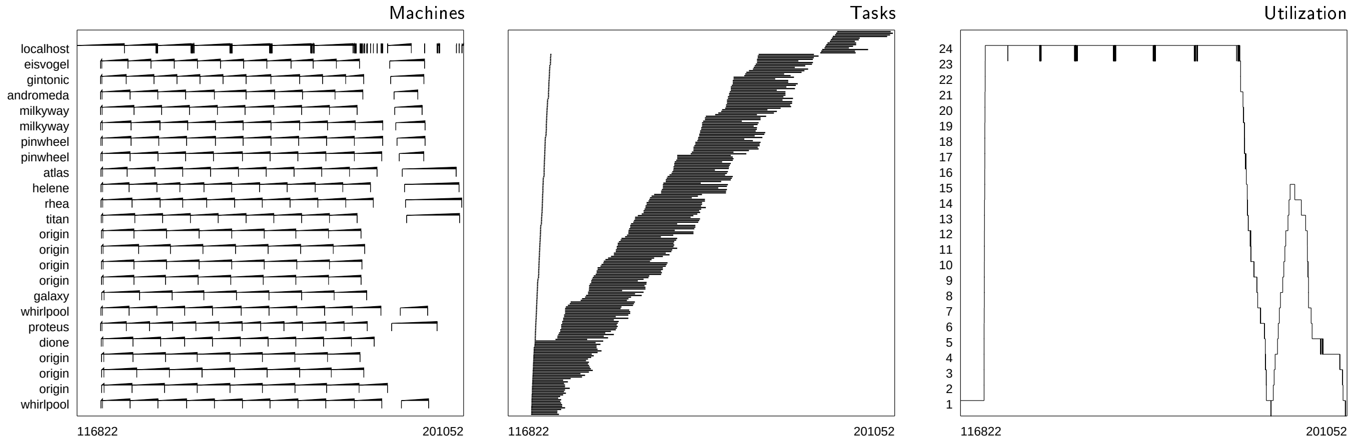

Figure 9: Program Visualization

References

[1] T. Q. Nam and F. Winkler, "CASA Reference Manual (Version 2.3)," Tech. Rep. 97-33, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, October 1997. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1997/97-33.ps.gz.

[2] MapleSoft, "Maple V Release 5," April 1998. http://www.maplesoft.com/maplevr5.html.

[3] 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. http://www.risc.uni-linz.ac.at/software/distmaple.

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

[5] W. Schreiner, "A Para-Functional Programming Interface for a Parallel Computer Algebra Package," *Journal of Symbolic Computation*, vol. 21, no. 4–6, pp. 593–614, 1996. http://www.risc.uni-linz.ac.at/people/schreine/papers/jsc96.ps.gz.

[6] D. Wang, "On the Parallelization of Characteristic-Set-Based Algorithms," in *Parallel Computation — First International ACPC Conference* (H. Zima, ed.), vol. 591 of *Lecture Notes in Computer Science*, (Salzburg, Austria, September 30–October 2), pp. 338–349, Springer, Berlin, 1991.

[7] 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*, (Tokyo, Japan, August 20–24), pp. 96–103, ACM Press, New York, 1990.

[8] K. Siegl, "Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||," in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (San Diego, California, May 19-22), pp. 179–186, ACM Press, New York, 1993. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-08.ps.gz.

[9] A. Diaz and E. Kaltofen, "FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation," in *ISSAC'98 International Symposium on Symbolic and Algebraic Computation* (O. Gloor, ed.), ACM Press, New York, 1998. http://www.math.ncsu.edu/~kaltofen/bibliography/98/DiKa98.ps.gz.

[10] L. Bernadin, "Maple on a Massively Parallel, Distributed Memory Machine," in *PASCO'97 — Second International Symposium on Parallel Symbolic Computation* (M. Hitz and E. Kaltofen, eds.), (Maui, Hawaii, July 20–22), pp. 217–222, ACM Press, New York, 1997.

[11] T. Q. Nam, "A Hybrid Symbolic-Numerical Method for Tracing Surface-to-Surface Intersections," in *ISSAC'95 International Symposium on Symbolic and Algebraic Computation* (A. Levelt, ed.), (Montreal, Canada, July 10–12), pp. 51–58, ACM Press, New York, 1995.

[12] G. E. Collins, "The Calculation of Multivariate Polynomial Resultants," *Journal of the ACM*, vol. 18, pp. 515–532, 1971.

[13] H. Hong and H. W. Loidl, "Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine," in *Parallel Processing: CONPAR 94 - VAPP VI — Third Joint International Conferences on Vector and Parallel Processing* (B. Buchberger and J. Volkert, eds.), vol. 854 of *Lecture Notes in Comuter science*, (Linz, Austria, September 6–8), pp. 325–336, Springer, Berlin, 1994.

[14] S. Seitz, "Algebraic Computation on a Local Net," in *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation* (R. E. Zippel, ed.), vol. 584 of *Lecture Notes in Computer Science*, (Ithaca, USA), pp. 19–31, Springer, Berlin, May 1990.