Journal of Symbolic Computation 35 (2003) 305–347

Journal of Symbolic Computation

www.elsevier.com/locate/jsc

Distributed Maple: parallel computer algebra in networked environments

Wolfgang Schreiner[*], Christian Mittermaier, Karoly Bosa

Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, 4040 Linz, Austria

Received 12 February 2001; accepted 23 September 2002

Abstract

We describe the design and use of Distributed Maple, an environment for executing parallel computer algebra programs on multiprocessors and heterogeneous clusters. The system embeds kernels of the computer algebra system Maple as computational engines into a networked coordination layer implemented in the programming language Java. On the basis of a comparatively high-level programming model, one may write parallel Maple programs that show good speedups in medium-scaled environments. We report on the use of the system for the parallelization of various functions of the algebraic geometry library CASA and demonstrate how design decisions affect the dynamic behaviour and performance of a parallel application. Numerous experimental results allow comparison of Distributed Maple with other systems for parallel computer algebra.
© 2003 Elsevier Science Ltd. All rights reserved.

1. Introduction

This paper gives a comprehensive overview on the design and the use of "Distributed Maple", an environment for parallel computer algebra on multiprocessors and heterogeneous computer clusters. The starting point of our work was in 1998, the goal—to parallelize parts of the software library CASA (computer algebra software for constructive algebraic geometry). CASA has since 1990 been developed by various researchers at RISC-Linz (Mnuk and Winkler, 1996) on the basis of the computer algebra system Maple (Maple, 2001).

Thus we have developed Distributed Maple as a work platform for parallel and networked environments (Schreiner, 1998). In contrast to some other related approaches, the underlying technological basis should be "time-safe" and widely accessible such that

[*] Corresponding author. Tel.: +43-732-2468-9920; fax: +43-732-2468-9930.
E-mail address: wolfgang.schreiner@risc.uni-linz.ac.at (W. Schreiner).
URL: http://www.risc.uni-linz.ac.at.

0014-5793/03/$ - see front matter © 2003 Elsevier Science Ltd. All rights reserved.
doi:10.1016/S0747-7171(02)00137-2

the developed applications remain useable for the foreseeable future. In particular, the system should be portable to any new commercial version of Maple, i.e. not require any special kernel extensions, such that the system and applications can be easily distributed to other researchers. Furthermore, our goal was 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.

We have tackled this goal by developing a configurable coordination program that starts and connects external computation kernels on various machines and schedules concurrent tasks for execution on them. This program is written in the programming language 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.

Starting from the initial design (Schreiner, 1999), we have gradually refined and extended the system, analysed its performance on various platforms (Schreiner, 2000), and started to introduce fault tolerance features (Schreiner et al., 2001). Also an interface of the coordination program to the computer algebra system Mathematica has been developed (Pau and Schreiner, 2000).

For the time being, we have used the Distributed Maple environment for developing parallel versions of the following CASA functions: `pacPlot` for the reliable plotting of algebraic plane curves (Schreiner et al., 2000b; Mittermaier et al., 2000), `ssiPlot` for the reliable plotting of surface to surface intersections (Schreiner et al., 2000c), `neighbGraph` for the neighbourhood analysis of algebraic curves (Schreiner et al., 2000a). The initial versions of these algorithms were developed in the frame of the diploma thesis (Mittermaier, 2000) and later refined and partially replaced (Schreiner, 2001). They also required the parallelization of various subalgorithms, in particular standard problems of computer algebra like multivariate resultant computation (Collins, 1971) or real root isolation (Collins and Akritas, 1976).

While we have based our work on a substantial body of knowledge on systems for parallel computer algebra (see Section 2), Distributed Maple incorporates a number of original ideas:

Time-safety and portability. Distributed Maple is built on top of a state of the art commercial computer algebra system. In contrast to other developments, the system only relies on basic interfaces to the Maple kernel and does not require any kernel extensions. Consequently, Distributed Maple has since 1998 survived four release changes (Maple 4–7) without major changes. The environment is so portable that applications can be executed in many different environments (from wide-area heterogeneous clusters to shared-memory multiprocessors). It is so general that it can be applied to schedule tasks of other computer algebra systems (e.g. Mathematica).

Abstraction level. Distributed Maple provides a programming model which is based on functional/logic/dataflow parallelism. Thus it allows the creation of a large number of implicitly scheduled tasks with automatic resolution of data dependencies and of globally shared data structures with implicit synchronization. The model therefore operates on a much higher level of abstraction than other models that only allow one process per processor and support communication/synchronization by explicit message passing. Consequently, Distributed Maple programs can be much more concise and elegant (i.e. close to the mathematical description of the problem solution) than programs in other systems.

Fault tolerance. Distributed Maple incorporates extensive support for fault-tolerance which is novel in this application area in particular and under the restrictions of a hybrid parallel computing environment in general. This allows us to write programs that take many days without risking computational loss by the failure of a computing node or of a communication link. The mechanisms reported in this paper include the logging of all computed results on stable storage and rescheduling of tasks that are computed on a failed non-root node. Current developments (not reported in this paper) deal with mechanisms that even tolerate the failure of the root node.

Applications. We have used Distributed Maple to develop the first parallel versions for a number of non-trivial applications from algebraic geometry (parallel curve and surface plotting and parallel neighbourhood analysis). The sequential versions were available as legacy libraries implemented in Maple; their parallel versions include multiple parallel subalgorithms and nested parallelization. This is one of the few examples of the parallelization of complex computer algebra solutions composed from various algorithms.

Both the Distributed Maple system itself and the library of parallel versions of CASA respectively Maple algorithms are in stable versions freely available under the GNU Library General Public License at

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

The rest of this paper is organized as follows. In Section 2, we sketch relevant work of other researchers on parallel computer algebra. In Section 3, we describe the architecture and programming interface of the system, analyse its performance on multiple architectures, and outline its support for fault tolerance. In Section 4 we describe the use of the system for the parallelization of various computer algebra algorithms and give numerous experimental results on several architectures. In Section 5, we draw our conclusions. Appendix contains the input data for the benchmarks in Section 4.

2. Related work

Distributed Maple has evolved from our own experience in the development of parallel computer algebra environments and from learning from the work of other researchers. In the following, we only cite research that is more or less directly relevant for the work described in this paper. Many papers on parallel computer algebra can be found in Della Dora and Fitch (1989), Zippel (1990), Hong (1994) and Hitz and Kaltofen (1997); summaries are also available in (Roch and Villard (1997) and Gautier et al. (2001).

The programming interface of Distributed Maple is based on a para-functional model as adopted by the PARSAC-2 system (Küchlin, 1990) for computer algebra on a shared memory multiprocessor. The model was refined in PACLIB (Hong et al., 1995) on which a para-functional language was based (Schreiner, 1996).

An early approach to use Maple as an engine for parallel computer algebra was “Sugarbush” (Char, 1990) that used the coordination language Linda to manage concurrent activities on multiprocessors and computer networks. Unlike Distributed Maple, the Maple kernel was extended by C/Linda primitives for tuple space access and provided a native Maple/Linda programming interface; similar to Distributed Maple, Sugarbush used the internal linear expression for communicating task descriptions and results. The system was used e.g. for parallelizing big number arithmetic (Char and Johnson, 1997); because of its special kernel it was not distributed and fell out of use with the decline of Linda support.

Without any special parallel programming support, Wang (1991) used Maple in a workstation cluster to parallelize the characteristic set algorithm. Maple kernels were directly started on various machines and communicated by reading and writing to files in a global network file system. Because of the large process granularity, this approach nevertheless achieved good speedups.

The ‖MAPLE‖ ("Parallel Maple") environment (Siegl, 1993) used the Guarded Horn Clause language Strand for coordinating the activities of multiple Maple kernels running on multiprocessors and computer networks. The programmer wrote programs in Strand with constructs for evaluating Maple expressions by the Maple kernel linked to the runtime system on the current processor. The programmer thus dealt with two programming languages, Strand for writing the parallel program and Maple level for writing the actual computations. Since the system depended on a special linkable version of the Maple kernel, it could not be distributed; with the decline of Strand, it fell out of use.

Chan et al. (1994) describes an environment where Maple computations can be distributed across a network of workstations by use of the DSC system (Diaz et al., 1991) which on the basis of standard Internet services ships source code and input data to computers for execution and retrieves the produced output. The parallel programming model is based on the master–slave paradigm extended by a co-routine like distribution mechanism; it uses files for communication and is thus close to the system level. The system was used with good success for the parallelization of various polynomial algorithms.

The parallel Maple system described in Bernardin (1997) extended the kernel by message passing primitives for the Intel Paragon distributed memory multiprocessor and provided a corresponding Maple programming interface. On top of the message passing model, it provided a limited version of a parallel functional model similar to that of Distributed Maple; however it only allowed the main program to create other tasks (no nested parallelism). The system was not distributed and is not in use any more.

The FoxBox system (Diaz and Kaltofen, 1998) provides via a Maple interface access to parallel implementations of polynomial factorization algorithms implemented in C++ on top of the message passing library MPI. This is a complementary approach to Distributed Maple and the systems mentioned above, because it allows Maple to use an external parallel program but not to write a parallel program that uses Maple.

The computer algebra system "muPad" is on the surface similar to Maple. Its kernel can be dynamically extended by a package for "macro parallelism" implemented in PVM (Metzner et al., 1999). This package allows us to write master–worker programs for distributed environments based on the concepts of message passing, global variables and work groups. The parallel programming model is on a considerably lower level than Distributed Maple.

The system “PVMaple” was developed for solving systems of differential equations (Petcu, 2000). Similar to the Intel Paragon Maple and to muPad, it provides a Maple interface for writing message passing programs. However, inspired by Distributed Maple, it uses an external process for performing the actual inter-process communication on top of PVM; Maple and this process communicate via shared files. The system runs on clusters of PCs under Microsoft Windows.

An ongoing activity attempts to combine the para-functional language Glasgow Parallel Haskell (Loidl et al., 1999) with Maple such that (in analogy to ‖MAPLE‖) parallel Haskell programs can coordinate the activities of multiple Maple kernels. Similar to Distributed Maple and PVMaple, the runtime system is connected via a pipe to a separate Maple process (Schreiner and Loidl, 2000).

Other activities on parallel computer algebra in distributed environments include the following ones: Hong (1993) gives a parallel implementation of the quantifier elimination algorithm for workstation networks on the basis of a distributed version of SACLIB. Bubeck et al. (1995) describes the DTS distributed thread system which was implemented on top of PVM and extended the functionality of PARSAC-2 to distributed environments; it was for instance used to implement a parallel version of multivariate resultant computation on a workstation cluster. Bertoli et al. (1994) describes a distributed multiprocessor kernel for the computer algebra library STURM on top of PVM. The very efficient PAC++ runtime kernel (Gautier and Roch, 1994) supports load balancing in distributed environments; it has e.g. been used to solve problems with algebraic numbers. Cooperman (1998) combines the message passing library MPI with the GAP system for writing parallel GAP programs.

3. The software system

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. The session trace below demonstrates the use of the environment:

```
aquila!33> maple
    |\^/|     Maple 6 (IBM INTEL LINUX22)
._|\|   |/|_. Copyright (c) 2000 by Waterloo Maple Inc.
 \  MAPLE  /  All rights reserved. Maple is a registered trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.1.7 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[virgo,linux], [andromeda,octane]]);
```

```
connecting virgo...
connecting andromeda...
                                     okay

> t1 := dist[start](int, x^n, x):
> t2 := dist[start](int, x^n, n);
> 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 `dist[initialize]`, we ask the system to start the distributed backend and create two additional Maple kernels on machine `aquila` 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 working directory.

After the distributed session has been successfully established, two 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 `dist[terminate]`.

3.1. *Software architecture*

The core of Distributed Maple is a scheduler program which 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 Fig. 1):

Scheduler. The Java program `dist.Scheduler` coordinates node interaction. The initial scheduler process (created by the Maple kernel attached to the user frontend) reads all system 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 in the compact linear format that Maple uses for library files).

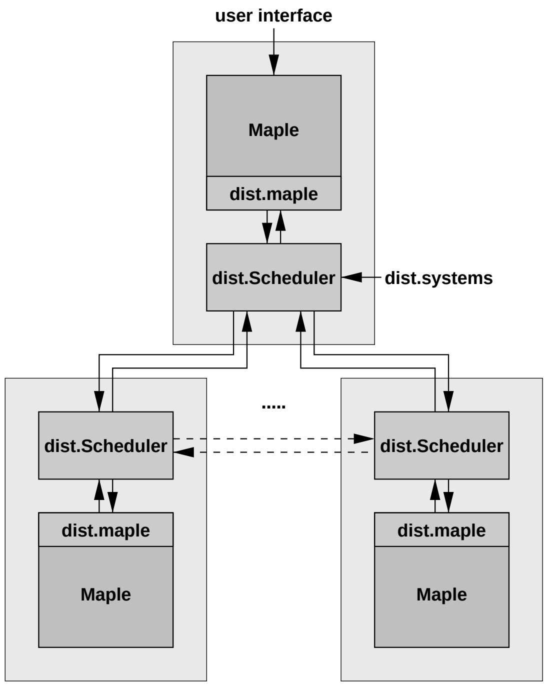

Fig. 1. Software architecture.

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 execution, additional socket connections between remote scheduler instances are created on demand.

3.2. Programming interface

Session initialization

In addition to the commands `dist[initialize]` and `dist[terminate]` that establish respectively end a distributed session, we need a possibility to initialize the Maple kernels on all machines by loading the code of user-defined functions and the contents of application-specific Maple libraries.

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

If `dist[all]` refers to a particular file, (a copy of) this file must be visible on every machine participating in the session. Thus e.g. a session-wide library may be loaded on every kernel.

However, based on this command, the application library also defines

`` `dist/load`( ``*`fun`*`)` loads the code of the function *fun* (defined in the current kernel) to every Maple kernel connected to the session.

With this command, we can directly disseminate function code from the current kernel to all remote kernels ("mobile code").

Functional parallelism

The parallel programming model is based on functional principles which is sufficient for many computer algebra algorithms:

`dist[start](`*f*`,`*a*`,...)` creates a task evaluating the expression $f(a,\ldots)$ and returns a reference t to this task.

`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.

The execution of a task may take place on any machine connected to the distributed session and must therefore not rely on any global Maple variables. When and on which machine a task is scheduled for execution is entirely in the responsibility of the underlying runtime system. Tasks may freely create other tasks; arbitrary Maple objects may be passed as task arguments and returned as task results (including references to other tasks).

However, if a task takes a task identifier as its arguments, it should be created by the following call:

`dist[process](`*f*`,`*a*`,...)` creates a task evaluating $f(a,\ldots)$ and returns a reference t to this task. *The task is executed on a kernel of its own.*

The rationale for this command is as follows: tasks created with `dist[start]` are scheduled on a fixed number of Maple kernels. If a task is suspended, another task may be executed on the respective kernel. Since the kernel is single-threaded, the last suspended task must be the first one to resume execution, i.e. a kernel hosts a whole stack of tasks of which only the task t on top is active. If t waits for the results of a task t' suspended on the kernel (which may only happen if t was passed the reference t' as its argument), program execution deadlocks. The use of `dist[process]` ensures that such deadlock cycles are avoided.

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 some task results may become obsolete.

`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 is deleted from the system (however, if it has already started, it continues execution until termination).

The usefulness of `dist[delete]` is currently rather limited because it does not abort an already executing task. We will in the future investigate mechanisms to abort tasks created through calls of `dist[process]` and of tasks created by aborted tasks.

Shared 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 object and returns its handle *d*. Any task may use *d* to read from or write to the shared object no matter on which machine the task is executed.

`dist[get](`*d*`)` blocks the execution of the current task until the 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 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 object referenced by *d*.

Shared 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).

Management functions

The programming interface provides a number of functions for managing a distributed session, e.g. by setting program parameters, generating visualization information (see Section 3.3.3), or setting the fault tolerance mode (see Section 3.5). For details, see Schreiner (1998).

Program skeleton

We demonstrate some of the parallel program constructs introduced in the previous section by a widely used program skeleton, a data-parallel program where the main program spawns a number of worker tasks to process part of the input; it then waits for the results and constructs from them the overall output:

```
main := proc(input)
  local fun, data, tasks, task, result, index, output;

  fun := proc(arg)
    ...
    RETURN(result);
  end;

  initD(input, data);
  tasks := [];
  while not termD(data) do
    task := dist[start](fun, argD(data));
    tasks := [op(tasks), task];
    updateD(data);
  od;

  initO(output);
  while tasks <> [] do
    result := dist[wait](tasks[1]);
    tasks := tasks[2..nops(tasks)];
    updateO(output, result);
  od;
  RETURN(output);
end:
```

If the output can be constructed from the task results in any order, we may replace the second loop by

```
while tasks <> [] do
  task := dist[select](tasks);
  result := task[1]; index := task[2];
  tasks := [op(tasks[1..index-1]), op(tasks[index+1, nops(tasks)])];
  updateO(output, result, index);
od;
```

Especially if there are large variations in the execution times of the individual tasks (or in the performance of the individual machines), this non-deterministic form of synchronization is advantageous.

While the above parallel program pattern is easy to use, it does not scale well because the main program becomes a communication bottleneck. We may solve this problem by distributing the synchronization by the use of shared data structures; this sort of *dataflow style* program is demonstrated in Section 4.4.2.

3.3. Scheduler operation

The Maple kernel is a single-threaded process which communicates by a simple communication protocol with the scheduler on the same node. All capabilities for parallel and distributed program execution are embedded in this scheduler.

3.3.1. Session control

On execution of `dist[initialize]`, the Maple kernel starts the scheduler on the current node (the *root*) and passes to it the list of machines to be connected to the session. The root scheduler reads the information in the configuration file `dist.systems` to startup schedulers on the remote nodes; the remote schedulers establish socket connections to the root scheduler and start the corresponding remote Maple kernels.

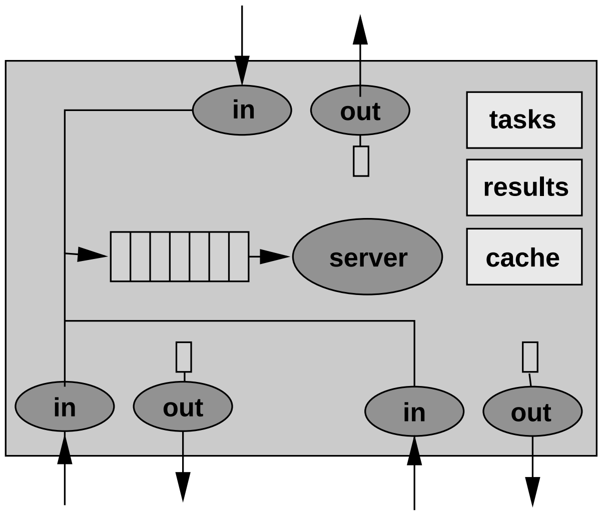

Fig. 2. The task scheduler.

Initially, thus there exist only connections between the root node and each remote node. However, all nodes know of each other, i.e. a node knows the address of a machine and the number of a port on which (a thread of) the remote scheduler is listening for connection requests. When a node needs to send a message to one of its peers, it can thus establish a direct connection for message transfers. The connection remains persistent through the rest of the session such that no more startup overhead is involved.

If a task is created by a call `dist[process]` (see Section 3.2), the scheduler starts an additional kernel process to which it forwards this task for execution; if the task is blocked the kernel remains idle until the task can resume execution. After termination of this task, the additional kernel is retained in a "kernel pool" such that for a new task created by `dist[process]`, this kernel can be recycled without additional startup overhead.

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.

3.3.2. Internal operation

The operation of the scheduler is implemented by a number of concurrent threads as shown in Fig. 2. Threads listening on all input channels put the received messages into a central buffer from where a server thread takes them, processes them, and creates new messages that are placed in some of the output buffers. Threads listening on the output buffers take these messages and send them to the corresponding output channels. This heavily multithreaded implementation of the communication interface simplifies the program design and maximizes the overall system throughput; the only synchronization point between two tasks are the shared message buffers.

After a distributed session has been established, the scheduler accepts tasks from the Maple process and schedules these tasks among any node connected to the session. A task is a pair $\langle t, d \rangle$ where t is an integer number identifying the task and d is a byte array to be submitted to the Maple process describing the task to be executed. Initially, the scheduler informs each Maple process about the range of task identifiers it may use for assignment to new tasks such that each node in the system can independently create new tasks.

When a 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 the machine where the task has been created (not necessarily on the machine 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 for its result. Additionally each scheduler holds a cache of all results that it has ever seen such that after the first request to non-local task results further requests can be immediately satisfied.

If the scheduler cannot immediately deliver a task result requested by the attached computation kernel, it sends a new task for execution instead. 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. Since, the computation kernel is not multithreaded, a task can only continue execution when all the tasks started later have terminated (last-in first-out principle).

All remote schedulers send new tasks to the root node scheduler which distributes them among all machines. Currently a simple load balancing scheme is used where the root node 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. The load balancing bounds for each machine can be configured by the user in the `dist.systems` configuration file.

3.3.3. Visualization

An important help for the performance tuning of parallel programs is a visual representation of its dynamic behaviour. The Distributed Maple scheduler supports on-line visualization as well as post-execution visualization.

If the user issues during the session a call `dist[visualize]`, a window pops up in which the scheduler on the root node displays his knowledge on the status of each machine and on the total system utilization (see Fig. 3). This allows us to get a quick overview on the dynamic behaviour of the parallel program and on the overall state of the distributed environment.

More information can be produced by issuing a command `dist[trace]`. Then a trace file with detailed event information is produced from which after program execution a tool

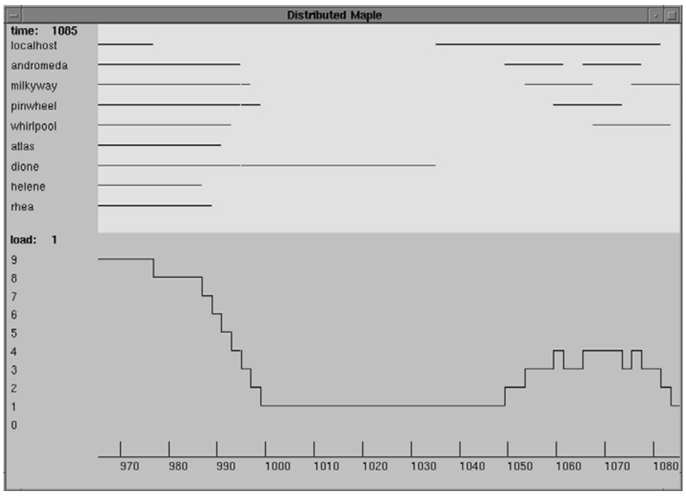

Fig. 3. Session visualization.

`dist.visual` generates various diagrams in printable format (some examples are shown in Section 4).

3.4. Performance analysis

The sequential performance of Distributed Maple is determined by Maple. For the performance of parallel applications, the communication performance is relevant. We model the communication system by multiple message exchange layers:

- *Between Maple and the scheduler process:* The Maple kernel linearizes an object in the heap to a sequence of bytes which is written to a Unix named pipe from where it is read by a thread of the scheduler process.
- *Within the scheduler process:* an input thread places the received message into a memory buffer shared with the server thread.
- *Between the scheduler processes:* a thread takes a message from the input buffer and writes it to a socket connected to a scheduler process on a remote machine where a thread reads the message.

The communication performance of the system is crucially determined by the *latency* of forwarding a message between two subsequent system layers. We restrict our observations to the following key times (see Fig. 4):

- Thread roundtrip: the minimum time it takes a message to travel from a thread via a shared buffer to another thread and back.
- Local roundtrip: the time it takes to query the result of a task stored in the local scheduler process.
- Remote roundtrip: the time it takes a message to travel from an output thread via a socket to an input thread and back.
- Pipe out: the time it takes to write a small message to an output stream

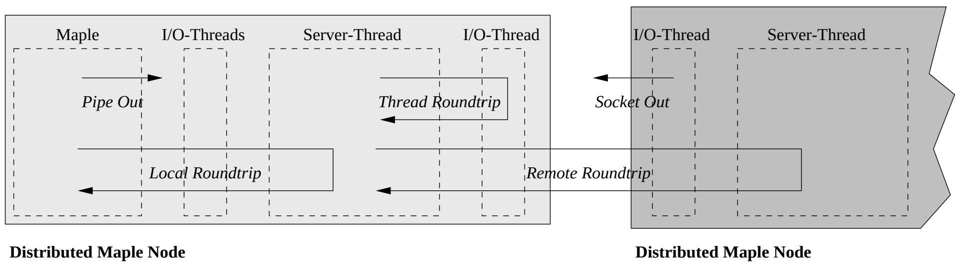

Fig. 4. Communication model.

Table 1
Performance data

Measurement	Time (ms) Linux PC	SGI Octane	Sun E3000
Thread roundtrip	<0.1	<0.1	<0.1
Local roundtrip	0.8	1.2–2	1
Remote roundtrip	31.0	150–300	100
Pipe out	0.5	0.7	0.1
Socket out	<0.1	0.4	<0.1
Start overhead	0.2	0.4	0.3
Wait overhead	0.1	0.3	<0.1
Latency	0.5	10–12	0.8
Bandwidth pipe:	1030 kB s^{-1}	853 kB s^{-1}	512 kB s^{-1}
Bandwidth socket:	3938 kB s^{-1}	4100 kB s^{-1}	5120 kB s^{-1}

- Socket out: the time it takes for an output thread to write a small message to a socket connected to a remote machine.
- Start overhead: the time for the linearization of a call `dist[start](f,0)`.
- Wait overhead: the time the linearization of a call `dist[wait](0)` takes.
- Latency: the minimum time difference between two messages sent by a remote node and received by the local Maple kernel.

In addition we capture the time that it takes to transfer large objects between two Maple kernels by two parameters:

- Pipe bandwidth: the amount of data that can be communicated per time unit between a Maple kernel and the Java scheduler (within a node).
- Socket bandwidth: the amount of data that can be communicated per time unit between two instances of the Java scheduler (across nodes).

These times become critical (only) in applications where the linearized Maple objects become large, i.e. several hundreds of kBs or more.

The times in Table 1 refer to two Linux 2.2 PCs with PIII@500 MHz processors connected by a 100 Mbit Ethernet, two SGI Octanes with R10000@250 MHz processors and connected by a 100 Mbit Ethernet, and a Sun E3000 bus-based shared memory multiprocessor with 250 MHz UltraSPARC processors.

In total, the performance is apparently determined by two parameters:

1. The *remote roundtrip time*, i.e. the time required for a message to cycle via sockets between two Java processes. This roundtrip time is an order of magnitude larger than the actual *latency*, i.e. the minimum time distance between subsequent elements of a stream of messages.
2. The *pipe bandwidth*, i.e. the amount of data communicated per second between Maple and the scheduler. This rate is much smaller than the *socket bandwidth* and thus becomes a bottleneck for inter-Maple communication.

The first issue is the price we pay for the abstractions provided by the software layers of the JVM. The second issue shows that for the transfer of large objects between Maple kernels the speed of the processor is the limiting factor rather than the capacity of the network; this could only be overcome by letting the nodes within a process communicate via shared memory rather than sockets.

3.5. Fault tolerance

The only mechanism originally available in Distributed Maple for dealing with faults was the watchdog mechanism described in Section 3.3 for shutting down the system in case of failures. However, as we begin to attack larger and larger problems, the meantime between session failures (less than a day) becomes a limiting factor in the applicability of the system.

There are numerous possibilities for faults that may cause a session failure: a machine becomes unreachable (usually a transient fault, i.e. the machine is rebooted or is temporarily disconnected), a process in the scheduling layer or in the computation layer crashes (a bug in the implementation of the Java Virtual Machine, of the Java Development Kit, of Maple, or of Distributed Maple itself) or the computation itself aborts unexpectedly (a bug in the application). While the last issue can be overcome and the Distributed Maple software itself is very stable, there certainly exist bugs in the lower software levels that are out of our control; machine/network/operating system faults may happen in any case.

We have therefore started to investigate how to introduce fault tolerance mechanisms that let the system deal with faults in such a way that the time spent in long running session is not wasted by an eventual failure. We start with a simple version of the system model on which the following explanations are based.

3.5.1. System and execution model

A session comprises of a set of *nodes* each of which holds a pair of processes: a *kernel* and a *scheduler*. Initially, a single task runs on the *root* kernel; this task may subsequently create new tasks which are distributed via the schedulers to other kernels and may in turn create new tasks.

The programming model described in Section 3.2 gives rise to the execution model depicted in Fig. 5: a kernel may emit a message $\mathsf{task} : \langle t, d\rangle$ where t is the identifier of a task and d represents its description. This message needs to be eventually forwarded to some idle kernel which then returns a message $\mathsf{result} : \langle t, r\rangle$ where r represents the

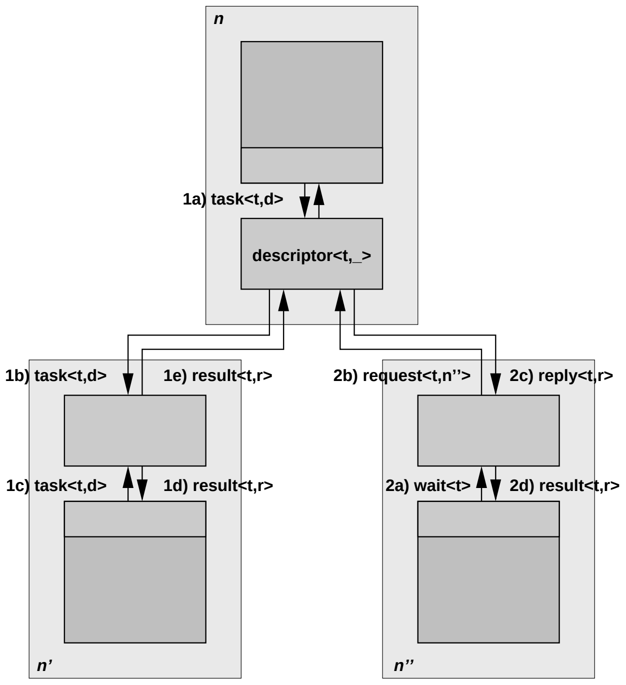

Fig. 5. Execution model.

computed result. When a kernel emits a wait : $\langle t\rangle$, the currently executing task is assumed to be blocked until the scheduler responds with the corresponding result. If this result is not available, the kernel is blocked; therefore the scheduler may submit to this kernel another task for execution. When this new task terminates, the scheduler may send the awaited result to the kernel or again submit another task for execution.

A task identifier t is a pair $\langle n, i\rangle$ where n refers to the node where the task was created and i is a counter. The address n encoded in t serves as the rendezvous point between the node n' computing the result r of t and any node n'' requesting r. When a scheduler on n receives a task : $\langle t, d\rangle$ from its kernel, it allocates a result descriptor that will eventually hold r; the task itself is scheduled for execution on n'. When a kernel on n'' issues a wait : $\langle t\rangle$, the scheduler on n'' forwards a request : $\langle t, n''\rangle$ to n. If r is not yet available, this request is queued in the result descriptor. When the kernel on n' eventually returns the result : $\langle t, r\rangle$, the scheduler on n' forwards this message to n, which sends a reply : $\langle t, r\rangle$ to n''.

3.5.2. Logging results: first-order tasks

A first step towards fault tolerance is to log task results on stable storage. We assume that the root has access to a writable file system. If a session fails, we can re-run it without re-executing the tasks whose results have been logged. Our first focus is on programs whose

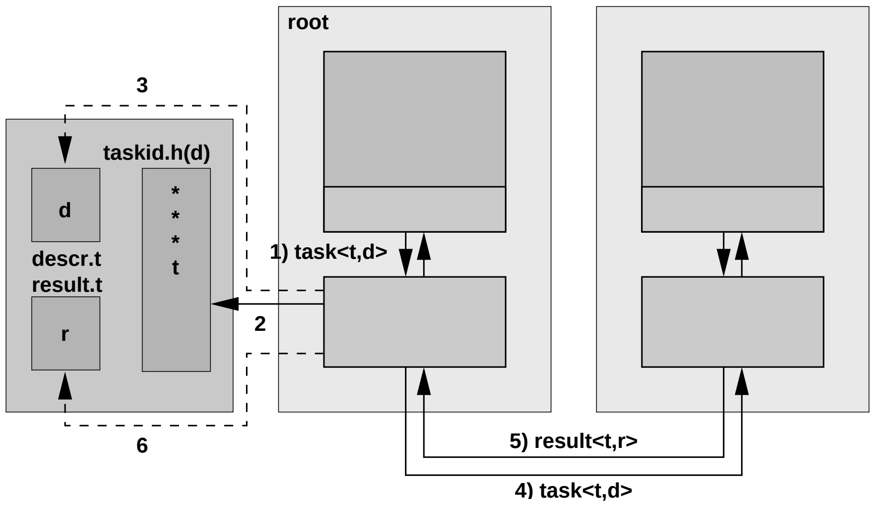

Fig. 6. Logging results: fist-order tasks.

tasks are *first order*: no task description d and no task result r contains any task identifier t, i.e. task identifiers are not passed as parts of task arguments/results (see Fig. 6):

Logging. When the root receives a task : $\langle t, d\rangle$, it computes a hash code $h(d)$ and appends to file `taskid.` $h(d)$ the task identifier t. Then the root starts an asynchronous thread to write the task description d into a new file `descr.` t. When a node sends a result : $\langle t, r\rangle$ to some node n different from the root, it forwards a copy to the root. When the root receives this result, it creates an asynchronous thread to write the task result r into a new file `result.` t.

All data are written in a format that enables a reader to recognize incomplete writes. At any time, `taskid.` $h(d)$ holds a sequence of task identifiers t (the last of which may be incomplete) for which there may exist description files `descr.` t and/or result files `result.` t (both with possibly incomplete contents). When the session terminates normally, the files are discarded.

Recovery. When a session is re-started after a failure, the root may receive from a node n a task : $\langle t, d\rangle$ such that a file `taskid.` $h(d)$ exists. If for some complete task identifier t' in this file there exist file `descr.` t' with a complete description identical to d and file `result.` t' with a complete result r, the root need not schedule t but may immediately return result : $\langle t, r\rangle$ to n.

The mechanism is simple and efficient: the only overhead occurs on the root for writing the log files and the potential sending of duplicate results to the root.

3.5.3. Logging results: higher-order tasks

Assume that a task t creates another task and embeds its identifier t' in result r. If r is logged and the session fails, in the recovery session this result may be read from the log such that task identifier t' is re-created. A task may subsequently issue a wait : $\langle t'\rangle$

referring to a no longer existing task or, even worse, to a task that computes a different result than in the failed session.

In order to allow *higher-order tasks*, i.e. tasks which receive task identifiers as arguments or return them as results, we introduce a *session identifier* which distinguishes task identifiers from different sessions. In an original session, the session identifier is initialized to 0 and a file `session` is written with content 0. In a recovery session, the previous identifier s is read from `session`, the new identifier is taken as $s + 1$ and overwrites the content of `session`. If the recovery session also fails, a new recovery session may be initiated and thus the identifier may grow to an arbitrary size (subject to an implementation limit).

A *task identifier* now is a triple $\langle s, n, i\rangle$ that refers to the session s in which the task was created. We generalize the logging and recovery mechanism as follows:

Logging. When a non-root kernel creates a task, the scheduler forwards it to the root and, until it receives an acknowledgment message from the root, does *not* process any further messages from the kernel. On receipt of this message, the root creates an entry in the corresponding hash file, writes *synchronously* the description and then returns the acknowledgment which enables the scheduler to process further kernel messages.

A task t that has created another task t' is thus not able to disseminate the identifier t' before the description of t' is appropriately logged on the root. Consequently, if a logged task result contains a reference to t', it is guaranteed that the root holds the description of t' in the log.

Recovery. We now have to deal with task identifiers that may (via logged descriptions or results) refer to previous sessions: the root handles such tasks.

If a kernel on node n issues a $\mathsf{wait} : \langle t\rangle$ where the session identifier of t is not that of the current session, the scheduler on n sends a $\mathsf{request} : \langle r, n\rangle$ to the root. If the root receives this request, it looks up whether it holds a result descriptor for t; if yes, it proceeds as usual, i.e. it responds with the result or, if this is not yet available, queues the request in the descriptor.

If the root does not hold a result descriptor for t, it creates one and queues the request there. It then looks up file `result.` t for the result of t logged in a previous session. If this file exists and holds a complete result r, the scheduler writes r into the descriptor and responds with $\mathsf{reply} : \langle t, r\rangle$.

Otherwise, the scheduler looks up `descr.`t (which must exist by the above logging mechanism) for the description d of t. The scheduler creates a new task : $\langle t, d\rangle$ which is handled as usual. When a kernel issues a $\mathsf{result} : \langle t, r\rangle$ for a task t of a previous session, the scheduler forwards this result to the root.

This mechanism guarantees that a recovery session s can complete the execution of a previous session using all results logged in any session $s' < s$.

3.5.4. Tolerating node failures

The previous sections dealt with mechanisms to ensure that after a failure a recovery session may reuse previously computed results. In this section, we sketch a mechanism that enables a session to cope with faults *without* aborting. We restrict our attention to the following scenario: a machine executing a non-root node becomes unreachable (stop failure) and the root continues operation with the remaining nodes (if the root fails, the session also fails).

A necessary condition to detect this failure is that the root cannot contact a node for a certain period of time. We thus let the root periodically check whether a message has been recently received from every node and, if not, send a ping message that has to be acknowledged. If no acknowledgment arrives within a certain time bound, this node is considered as *dead*. This does not necessarily mean that the node is actually dead; it may be slow in responding, the connection to the root may have been transiently interrupted or connections to other nodes may still exist. We must therefore assume that even a dead node may send messages to the root or to any other node. Thus, when the root designates a node as dead, it informs all other nodes correspondingly: every node closes the connection to the dead node and ignores any remaining messages from this node (such messages may arrive before the actual closing of the connection).

There are two main problem that the root now has to deal with:

1. the management of all result descriptors that have been stored on the dead node, and
2. the rescheduling of all tasks that were executing on the node at the time of its alleged death.

Since the root is in charge of task scheduling, the root sees every task created in the session. Furthermore, by the logging mechanism discussed in the previous sections, the root sees every result computed in the session. For every node n, the root can therefore maintain two sets T_n and S_n:

1. T_n denotes all tasks scheduled on n; for a subset T_n^{r} the results are available (in the logging files). All tasks in $T_n - T_n^{\mathrm{r}}$ have to be executed again; the root puts them back into the pool of tasks to be scheduled for execution.
2. S_n denotes all tasks or shared objects whose descriptors are stored on n; for a subset S_n^{r} the results are available (in the logging files). The root becomes the owner of elements in S_n; it allocates the corresponding result descriptors and, for all elements of S_n^{r}, fills them with results.

   Subsequently, every node will send requests for a result in S_n to the root. However, there may be still outstanding requests sent to n but not yet answered at the time of its death. Every node n' therefore holds a table R_n of all request : $\langle t, n' \rangle$ messages sent to node n but not yet answered by a reply : $\langle t, r \rangle$. When n is marked dead, the node re-sends all messages in R_n to the root which will eventually answer them.

Thus all tasks scheduled on an eventually dead node n are executed (possibly on a different node n') and every descriptor originally housed by n finds a new home on the root to which all open and all future requests are redirected.

3.5.5. Summary

In contrast to other approaches, the Distributed Maple environment that operates with an essentially functional parallel programming model can tolerate faults with relatively simple mechanisms, i.e. without global snapshots as required in message passing programs. The runtime overhead imposed by the logging mechanism is very moderate; adding tolerance of node failures on top does then not require much extra overhead. One reason for this simplicity is the delegation of all logging activities to a single root node that also performs the task scheduling decisions (and remains a single point of failure); the model is therefore not scalable beyond a certain number of nodes. However, it is suitable for the system environments that we have used up to now ($\leq$30 nodes). Recently, we have extended the model to avoid the still existing single point of failure (the root).

4. Applications

A major motivation for the development of Distributed Maple was the parallelization of parts of CASA, a Maple library developed by various researchers at RISC-Linz for solving problems in algebraic geometry (Mnuk and Winkler, 1996). The basic objects of CASA are algebraic sets represented e.g. as systems of polynomial equations. Algebraic sets represented by bivariate polynomials model plane curves. Algebraic sets represented by trivariate polynomials model surfaces in space; intersections of such surfaces define space curves.

We have developed novel parallel variants of various CASA algorithms, many of them originating in the diploma thesis (Mittermaier, 2000). For this purpose, we have also implemented known parallel variants of some basic Maple functions. We are going to sketch these before we turn to the parallel CASA algorithms.

The exact inputs of all benchmarks are listed in the Appendix.

4.1. Parallel Maple functions

4.1.1. Bivariate resultant computation

In the context of CASA, we are interested in computing the resultant of two bivariate polynomials over the integers. The following description of a parallel algorithm for solving this problem is composed from material of Schreiner (1999).

Based on Collins (1971), various parallel versions of the modular algorithm have been implemented on workstation networks (Seitz, 1990; Bubeck et al., 1995) and on shared memory machines (Hong and Loidl, 1994; Schreiner, 1996). Their common idea is to compute the resultants in the individual modular domains in parallel; they differ in their approaches to combine the modular resultants to yield the integer resultant. We apply the idea used by Hong and Loidl (1994) where the sequential structure of the combination phase is maintained but the individual resultant coefficients are computed in parallel.

The parallel algorithm is sketched in Fig. 7. In the actual implementation each task computes multiple modular resultants respectively multiple coefficients of the integer resultant. By adjusting the number of elements computed per task we can effectively control its grain size. For estimating the execution time of a “mresultant” task we use the complexity bound given in Collins (1971) multiplied by an experimentally determined constant for the processor speed.

We have benchmarked the parallel resultant algorithm with three sample inputs (taken from the plotting of algebraic plane curves, see Section 4.2) for which the sequential program takes on a PIII@450 MHz PC 448, 63, and 977 s respectively. The parallel variant was executed on a 24 processor heterogeneous computer cluster, an 18-processor

```
$\mathbf{C} := \mathbf{resultant}(\mathbf{A}, \mathbf{B})$ :
  $m := \mathrm{degree}(A)$; $n := \mathrm{degree}(B)$
  $cb := \mathrm{coeffBound}(A, B)$; $db := \mathrm{degreeBound}(A, B)$
  $C := 0$; $P := 1$; $T := [\,]$; $L := [\,]$
  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
      $t := \mathrm{start}(\mathrm{mresultant}, p, db, A, B)$
      $P := P * p$; $T := T \,||\, [t]$; $L := L \,||\, [p]$
    end
  end
  $R := \mathrm{waitAll}(T)$
  $T := [\,]$
  for $i \in [0 \ldots db]$ do
    $t := \mathrm{start}(\mathrm{cra}, L, [R[j]_i \mid j \in [1 \ldots \mathrm{length}(R)])$
    $T := T \,||\, [t]$
  end
  $C := \mathrm{waitAll}(T)$
end
```

Fig. 7. Parallel resultant computation.

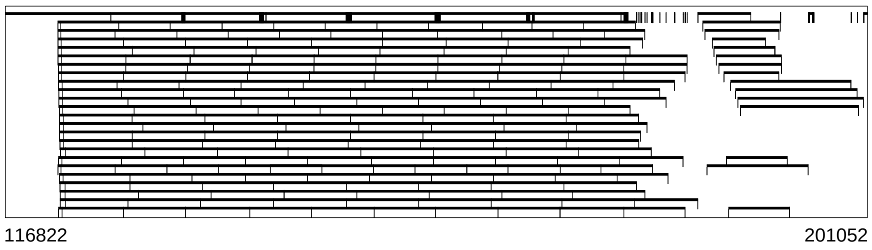

Fig. 8. Bivariate resultant computation.

Sun HPC 6500 system, and a Linux-based Beowulf cluster with 16 compute nodes linked by two 100 Mbit switched Ethernets. Relative to a PIII@450 MHz processor, the raw computing powers of cluster, Sun and Beowulf are (for 16 processors) 8.95, 11.68, and 22.83, respectively.

The trace in Fig. 8 illustrates a sample execution in the heterogeneous cluster with 24 processors listed on the vertical axis; each line denotes a task executed on a particular machine. We can clearly distinguish the modular resultant computation phase with many tasks saturating all processors from the combination phase where only few tasks are created.

Example	System	4	8	12	16	20	24
1 (448s)	Cluster	192	112	88	73	73	66
	Sun	160	83	60	48	-	-
	Beowulf	89	45	31	23	-	-
2 (63s)	Cluster	36	25	25	26	24	24
	Sun	26	16	13	11	-	-
	Beowulf	13	7	6	5	-	-
1 (977s)	Cluster	370	204	143	111	93	84
	Sun	330	167	117	94	-	-
	Beowulf	174	87	59	44	-	-

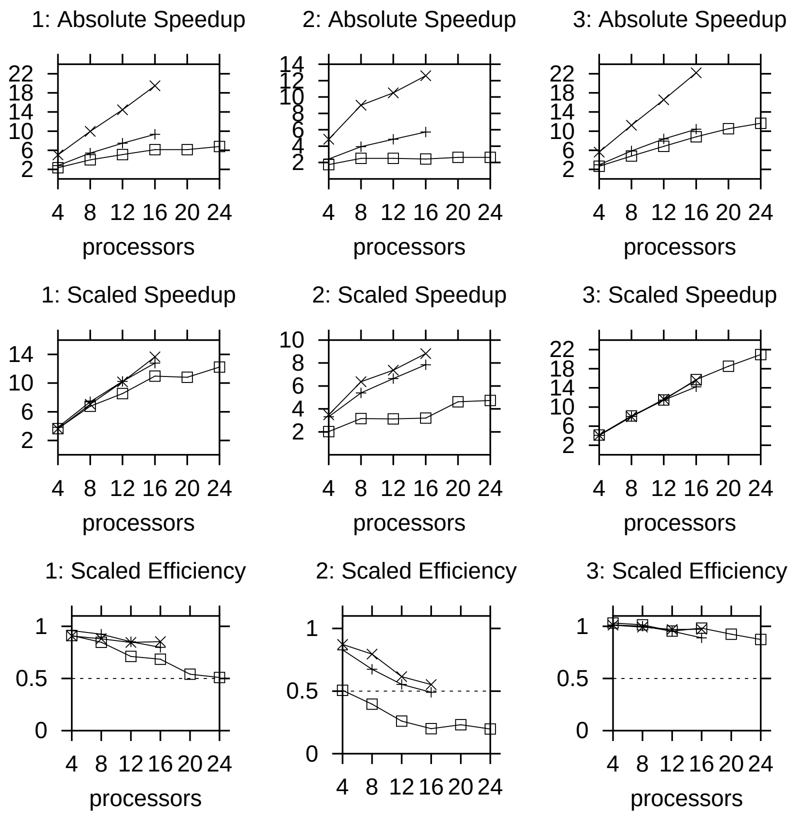

Fig. 9. Bivariate resultant computation: experimental results.

Fig. 9 lists the execution times for each input in each system environment with a varying numbers of processors. The subsequent row of diagrams visualizes the absolute speedups 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 $n / \sum_{i=1}^{n} p_i$ (where p_i denotes the relative performance of processor i), the third row visualizes the scaled efficiency, i.e. $T_s / T_n \sum_{i=1}^{n} p_i$, which compares the speedup we actually got to an upper bound of the speedup we could have got. The markers $\Box$, $+$, and $\times$, denote execution on cluster, Sun, and Beowulf, respectively. We see that for Example 3 all three environments give similar scaled speedups while in the small Example 2 the heterogeneous cluster environment significantly lags behind. On the other hand, the Beowulf cluster with its fast communication fabric gives in all examples the best speedups.

We may contrast the above figures to other results from literature: Seitz (1990) reports for a problem whose sequential execution took 1153 s a speedup of 12 with a cluster of 16 Sun workstations. Bubeck et al. (1995) reports for a problem whose sequential execution took 2500 s a speedup of 5.5 on a cluster of 13 Sun workstations. The PACLIB implementation on a shared memory machine achieved with 16 processors a speedup of 11.5 for a problem whose sequential execution took 180 s (Hong and Loidl, 1994). The pD implementation on a shared memory processor achieved a speedup of 11.5 for a problem whose execution took 160 s (Schreiner, 1996). Thus our results seem to be better than the ones reported for clusters and comparative to those reported for shared memory machines.

4.1.2. Real root isolation

The problem of isolating the real roots of $A \in \mathbb{Q}(x)$ can be efficiently solved by the modified Uspensky algorithm (Collins and Akritas, 1976). The major idea for the parallelization of this algorithm is based on its divide and conquer structure by executing each recursive branch in parallel. However, as has been noted in Collins et al. (1990), for most concrete polynomials this "search tree" is very unbalanced with a few long branches and a small average number of nodes (about 2) in every level. To utilize the available computing resources, we thus apply *speculative parallelism* resources. In each recursive step, we split the interval not only into two subintervals but into 2^s subintervals for some $s \geq 1$. The case $s = 1$ corresponds to the standard method; if $s > 1$, we widen the search tree by a factor of 2^{s-1} and flatten it by a factor of s. This strategy increases the potential for parallelism but also the amount of work done in the algorithm. The core of the parallel algorithm is depicted in Fig. 10.

We have benchmarked the program on a cluster of 20 Linux-PCs linked by 100 Mbit Ethernet lines and on a 128 processor SGI Origin 3800 distributed shared memory multiprocessor. We benchmarked four sample inputs for which the execution of the sequential CASA function `realroot_sb` (which is 2–5 times faster than the Maple function `realroot`) takes on a Pentium III@640 MHz 20, 26, 85, 104 s and on the Origin 23, 33, 109, 107 s, respectively. Examples 1 and 2 are polynomials of degrees 73 and 81 taken from runs of `pacPlot` described in Section 4.2; Example 3 is the Chebyshev polynomial T_{110} (where $T_1(x) = x$ and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$) which has a wide and deep search tree; Example 4 is the Mignotte polynomial $x^{100} - 2(5x - 1)^2$ whose search tree is narrow and deep. Examples 3 and 4 thus describe the best and the worst-case scenario for real root isolation; Examples 1 and 2 are typical intermediate cases. Fig. 11 shows the trace of Example 2 with $s = 2$ and 8 processors on the cluster.

Fig. 12 gives the execution times for the four examples in both environments (cluster and origin) with a varying number of processors and speculation parameter s

```
$\mathbf{r} := \mathbf{uspensky}(\mathbf{p}, \mathbf{i}, \mathbf{c}, \mathbf{d})$
    // $r$ is set of isolating intervals for the roots of $p(x+i)$ in $(0, 1]$
    // where interval bound $b$ is transformed to $(b+c)/2^{sd}$
    $r := \{\}$
    $A(x) := \mathbf{get}(p)$                         // get polynomial and
    $A(x) := A(x+i)$                                   // complete transformation
    if $A(1) = 0$ then                                 // is right boundary a root?
        $r := r \cup \{[(1+c)/2^{sd}, (1+c)/2^{sd}]\}$
        $A(x) := A/(x-1)$
    end
    $A^*(x) := x^{\deg(A)} A(1/x)$                     // termination case?
    $v := \text{signvars}(A^*(x+1))$
    if $v = 0$ then return end
    if $v = 1$ then $r := r \cup \{(c/2^{sd}, (1+c)/2^{sd})\}$; return end
    $p' := \mathbf{data}()$                            // recursion case:
    $\mathbf{put}(p', 2^{s \deg(A)} A(x/2^s))$         // prepare transformation
    $t := \{\mathbf{start}(\text{uspensky}, p', j, c * 2^s + j, d+1) : 0 \le j < 2^s - 1\}$
    $r := r \cup \{\mathbf{wait}(u) : u \in t\}$
end
```

Fig. 10. Parallel real root isolation.

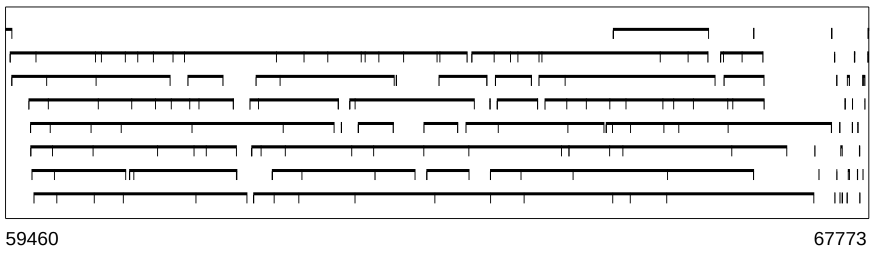

Fig. 11. Real root isolation.

(1, 2, 3, 4 denoted by markers $+$, $\times$, $\square$, $\blacksquare$ in the subsequent diagrams). All in all, the examples suggest that speculating too much in real root isolation usually hurts; limited speculation with $s = 2$ may improve the results a bit (but also deteriorate them a bit). However, for isolating very close roots, speculation is essential to gain any speedups at all. A prosaic reason for the limited speedups is that the chosen examples are not very big; this reflects the fact that realroot isolation is a subalgorithm typically applied in the context of larger applications (as shown in the following sections). If the inputs were larger, these applications would usually run into other time or memory limits.

Example (C/O)	s	4	8	12	16	20
1 (20s/23s)	1	12/18	8/12	7/11	7/10	7/10
	2	9 /17	7/12	6/10	6/11	7/10
	3	14/23	7/12	6/9	6/10	6/10
	4	29/35	15/17	11/13	9/11	8/13
2 (26s/33s)	1	13/18	8/13	7/12	6/13	7/12
	2	13/19	9/12	7/15	7/11	7/12
	3	18/25	10/13	8/11	7/10	6/11
	4	29/43	15/23	11/17	9/16	8/15
3 (85s/109s)	1	29/41	19/30	11/21	12/14	11/12
	2	37/50	23/42	17/29	16/22	17/32
	3	43/59	24/31	22/23	17/14	20/16
	4	91/130	59/71	40/43	33/37	33/30
4 (104s/107s)	1	79/117	77/103	75/102	77/100	80/100
	2	37/103	23/74	17/69	16/69	17/69
	3	91/116	42/58	43/53	44/52	49/52
	4	117/176	78/127	45/55	39/44	44/43

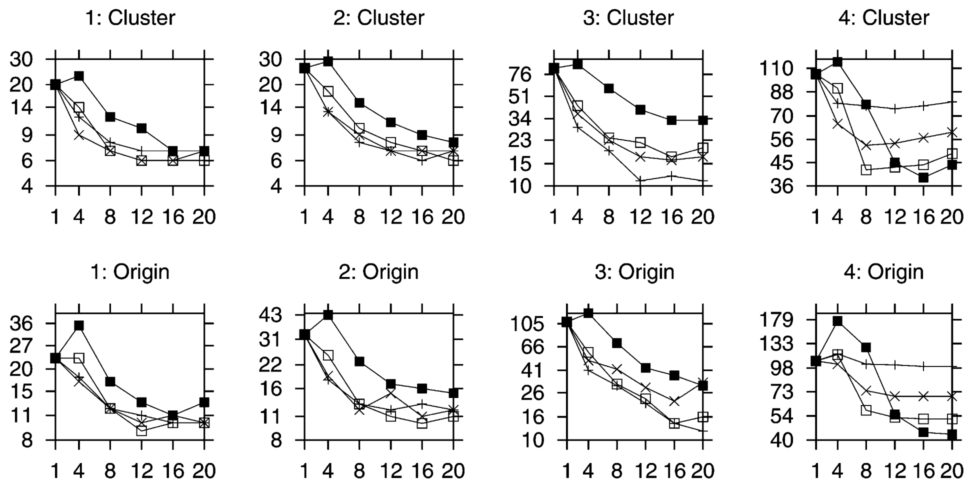

Fig. 12. Real root isolation: experimental results.

Collins et al. (1990) describes a shared memory parallel implementation of the bisection method. On a 20 processor Encore Multimax, a speedup of 2.5 was achieved for a randomly generated polynomial of degree 40 (sequential execution time about 4 s) and a speedup of about 4.6 for an artificial polynomial of degree 20 with 20 roots (sequential execution time about 5 s). Our parallel implementation compares favourably with this one.

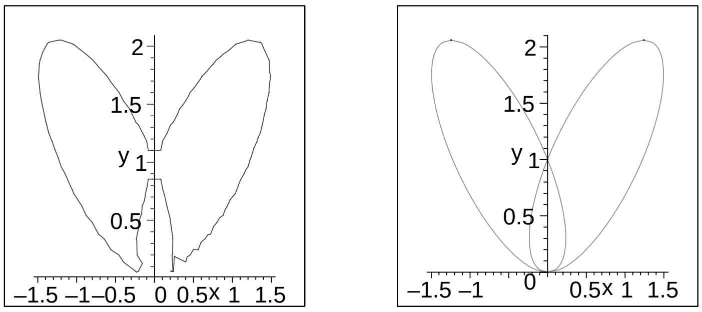

Fig. 13. Maple `implicitplot` versus CASA `pacPlot`.

The distributed memory implementation of Decker and Krandick (1999) parallelizes polynomial arithmetic in each node of the search tree. With 20 Cray T3E processors, speedups of 15 and 13 are reported for random polynomials with 2000 bit integer coefficients (sequential execution time 80 s respectively 40 s); a speedup of 15 is reported for a Chebyshev polynomial for which sequential execution takes 25 s; a speedup of 3 with four processors is reported for a Mignotte polynomial for which sequential execution takes 70 s. This C-based implementation uses arbitrary precision floating point interval arithmetic which is much more efficient than our Maple-based version. It thus achieves its speedups with polynomials that are orders of magnitudes larger than those in our benchmark.

The original presentation of speculative real root isolation in Mittermaier (2000) is based on a more complex task management scheme. However, the above scheme is not only simpler but also gives considerably better speedups.

4.2. Plotting of algebraic plane curves

One problem in algebraic geometry is the reliable plotting of algebraic curves. Conventional methods often yield qualitatively wrong solutions, i.e. plots where some “critical points” (e.g. singularities) are missing. For instance, the left diagram in Fig. 13 shows a plot of the plane curve $2x^4 - 3x^2y + y^4 - 2y^3 + y^2$ generated by Maple’s `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. On the other hand, CASA’s `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 (Nam, 1994). Our presentation of the parallelization of this algorithm is based on Schreiner et al. (2000b) and Mittermaier et al. (2000).

`pacPlot` proceeds in three steps to plot an algebraic curve a:

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.

```
$P := \textbf{criticalPoints}(a(x,y))$:
  $P := \emptyset$
  $S := \{\langle p(y), q(x,y)\rangle \mid \exists p'(y) :$
    $\langle p'(y), q(x,y)\rangle \in \text{triangulize}(a(x,y), \frac{\partial a(x,y)}{\partial x}), p(y) \in \text{factor}(p'(y))\}$
  for $\langle p(y), q(x,y)\rangle \in S$ do
    $r(x) := \underline{\text{resultant}_x(p(y),\ q(x,y))}$
    $X := \underline{\text{realroot}(r(x))}$
    $Y := \underline{\text{realroot}(p(y))}$
    $\underline{\textbf{for } x \in X \textbf{ do}}$
      $q'(y) := \text{squarefree}(q(x,y),\ x.0,\ p(y))$
      $q''(y) := \text{squarefree}(q(x,y),\ x.1,\ p(y))$
      $\underline{\textbf{for } y \in Y \textbf{ do}}$
        if $\text{test}(q'(y),\ q''(y),\ y,\ p(y))$ then $P := P \cup \{\langle x, y\rangle\}$
      end
    end
  end
  $\underline{\textbf{for } p \in P \textbf{ do } \text{refine}(p) \textbf{ end}}$
end
```

Fig. 14. Computation of critical points.

3. Trace a from each intersection point in both directions towards the border points of the stripe.

The algorithm 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”) $\langle x, y\rangle \in \mathbb{R} \times \mathbb{R}$ of the system $S := \{a = 0, \partial a/\partial x = 0\}$. Since exact arithmetic in $\mathbb{R}$ is not possible, each root $\langle x, y\rangle$ is actually “isolated” by a pair of intervals $\langle [x', x''], [y', y'']\rangle$ with $x', x'', y', y'' \in \mathbb{Q}$ such that $\langle x, y\rangle$ is the only root of S with $x' < x < x''$ and $y' < y < y''$.

The algorithm computing the set of all intervals that isolate the critical points is sketched in Fig. 14. This algorithm contains various improvements introduced by Mittermaier (2000); they already reduce the computation time by an order of magnitude. Our experimental results thus measure the parallelization speedups against this improved version.

We have parallelized this algorithm on various levels (underlined in Fig. 14):

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

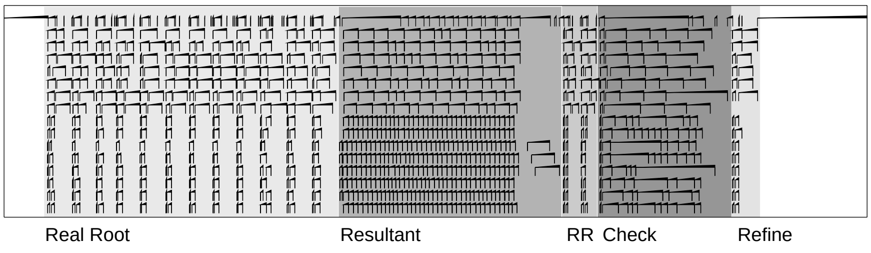

Fig. 15. Plotting of algebraic plane curves.

Parallel resultant computation and parallel real root isolation were discussed in Section 4.1. The tests which of the candidates $\langle x, y\rangle$ are indeed solutions of the given system can be performed in parallel in a straightforward fashion. Likewise, we can apparently refine all isolating intervals in parallel to the desired accuracy.

We have benchmarked the parallel variant of `pacPlot` with four randomly generated algebraic curves for which the sequential program on a PIII@450 MHz PC takes 6870, 470, 155, and 11,748 s respectively. The parallel variant has been executed in three system environments consisting of 24 processors each: a cluster of four SGI Octane dual-processor machines and 16 Linux PCs, a SGI 2000 multiprocessor, and a cluster of four dual-processor Octanes and 16 processors of the Origin multiprocessor. The raw computing powers of cluster, Origin, and mixed configuration relative to a PIII@450 MHz processor are 18.3, 17.1, and 18.7, respectively.

The profile in Fig. 15 illustrates the execution of Example 2 in a cluster with 16 processors listed on the vertical axis (eight Octane processors above eight 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 Fig. 16 lists the execution times measured in each environment for each input with varying numbers of processors; the following diagrams display the same information as described in Section 4.1.1. The markers $+$, $\times$, and $\square$ denote execution on cluster, Origin, and in the mixed configuration, respectively.

Analysing the experimental data gives some interesting results. Most obviously, the speedup 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 16–24 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

Example	System	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

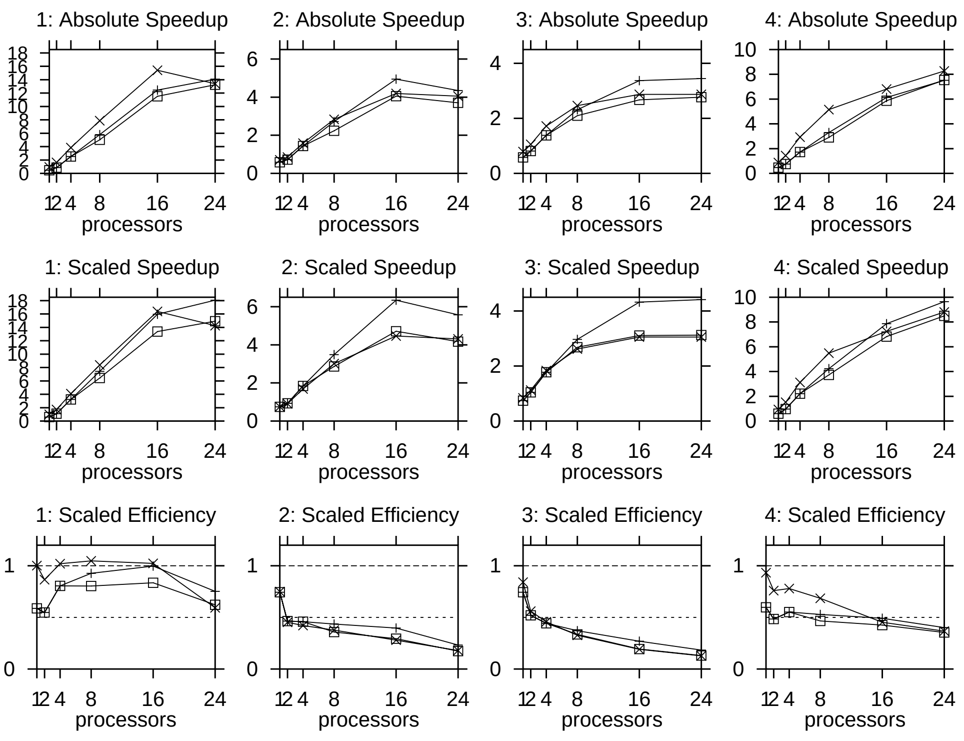

Fig. 16. Plotting of algebraic plane curves: experimental results.

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.

In Schreiner (2000) we give additional timings on a Sun HPC 6500 shared memory multiprocessor and on a dedicated Beowulf cluster; they are in some cases significantly better than the above figures.

4.3. Plotting of surface to surface intersections

The CASA function `ssiPlot` generalizes `pacPlot` to the three-dimensional case: it solves the problem of reliably plotting algebraic space curves defined by surface to surface intersections. The following presentation of the parallelization of this algorithm is based on Schreiner et al. (2000c).

The solution idea is analogous to the one sketched in Section 4.2 for the two-dimensional case: we first determine the critical points of a in one coordinate direction, say z. These points define planes along the other two coordinate directions. The collection of these planes partition the space into a number of subsequent layers. Within each layer, we determine starting points on the curve from which we may safely trace the simple branches of a by numerical methods.

Again, the algorithm spends most of the execution time on the computation of the critical points which is crucially different from the two-dimensional case. Now the problem is, given two surfaces $f(x, y, z), g(x, y, z) \in \mathbb{Q}[x, y, z]$, to find every real root $\langle x, y, z\rangle \in \mathbb{R}^3$ of the system $\{f = 0, g = 0, J_x(f, g) = 0\}$, where $J_x(f, g)$ is the Jacobian of f and g with respect to y and z. Since exact arithmetic is not possible in $\mathbb{R}$, each root $\langle x, y, z\rangle$ is actually isolated by a triple of intervals $\langle[x', x''], [y', y''], [z', z'']\rangle$ where $x', x'', y', y'', z', z'' \in \mathbb{Q}$ such that $\langle x, y, z\rangle$ is the only root of the system for which $x' < x < x'', y' < y < y''$, and $z' < z < z''$.

Isolating intervals for the z-coordinates of the critical points 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* $d(z) \in \mathbb{Q}(z)$ of f, g, and j, and finally isolating the real roots of d. The most time-critical part is the computation of $d(z)$ which at once eliminates several variables of a polynomial system while preserving the common roots. The $d(z)$ is computed from a matrix D by fraction-free Gaussian elimination as depicted in Fig. 17. After each reduction step, the entries of D represent rational functions; they have to be normalized to simple polynomials. In this subalgorithm, `ssiPlot` spends 60–80% of the total execution time.

To speed up `ssiPlot`, we apply parallelism in various subalgorithms, the most significant one is the computation of $d(z)$ from D. In iteration k of the outer loop, we compute all the $(k - r) * (k - c)$ new elements of D in parallel (the underlined part in Fig. 17). Having selected the number of tasks t (which is usually greater than the number of processors to improve load balancing), we thus start a task for every $(r - k) * (c - k)/t$ elements. While the number of matrix elements per task thus decreases with increasing k, the coefficient sizes of the corresponding polynomials increase and polynomial simplification becomes more complex. Thus the overall task grain size actually *grows* which allows us to exploit the available parallelism efficiently.

```
$d(z)$ := dresultant($D$):
  $r := \mathrm{rows}(D)$
  $c := \mathrm{columns}(D)$
  $d(z) := 1$
  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
      $t(z) := \mathrm{simplify}(\frac{D_{i,k}(z)}{D_{k,k}(z)})$
      for $j$ from $k+1$ to $c$ do
        $D_{i,j}(z) := \mathrm{normal}(D_{i,j}(z) - t * D_{k,j}(z))$
      end
      $D_{i,k}(z) := 0$
    end
  end
end
```

Fig. 17. Computation of Dixon resultant.

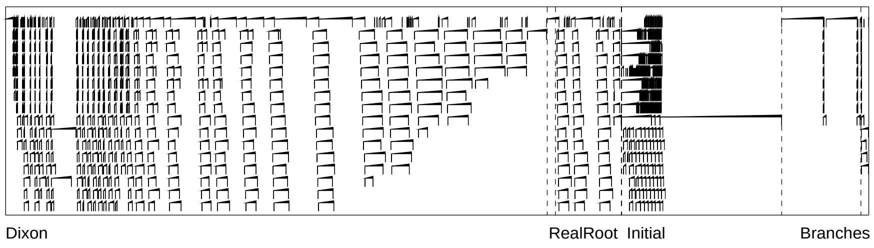

Fig. 18. Plotting of algebraic space curves.

We have benchmarked the parallel variant of `ssiPlot` with four random curves for which the sequential program takes on a PIII@450 MHz PC 2119, 475, 1523, and 5973 s, respectively. The parallel variant was executed in the same environment as described in Section 4.2. The diagram in Fig. 18 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 in Fig. 19 lists the execution times for each input in each environment with varying processor numbers; the following diagrams display the same information as described in Section 4.1.1. Sometimes, the benchmarks apparently yield *too* good results: for processor numbers up to four or even eight, we get scaled efficiencies between 1 and 2.

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

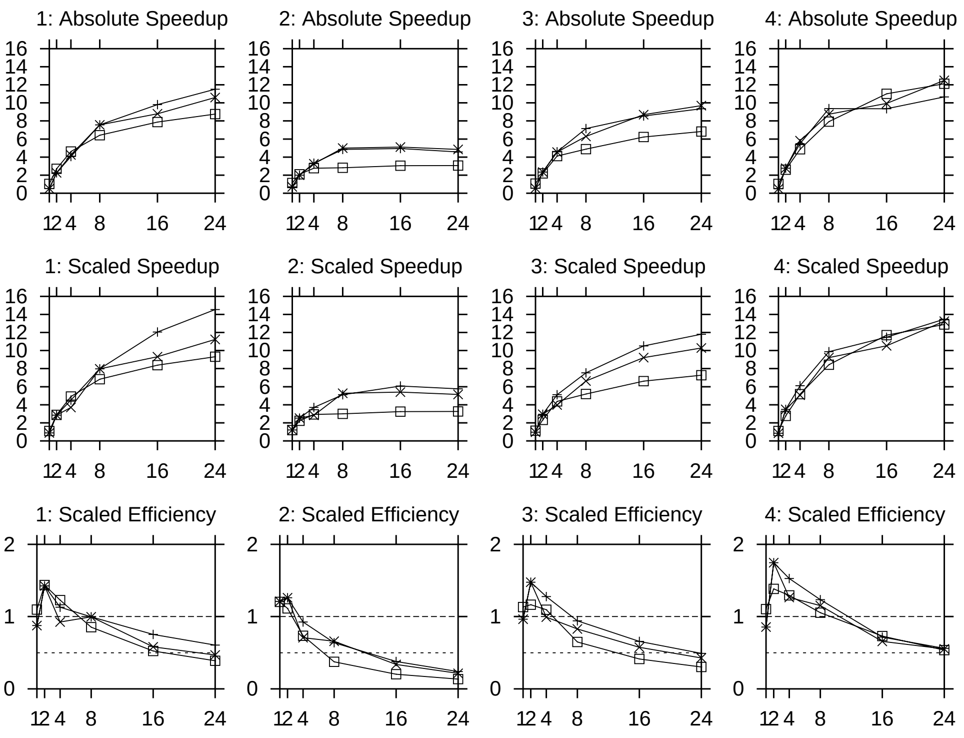

Fig. 19. Plotting of algebraic space curves: experimental results.

These superlinear speedups are caused by the increased amount of heap memory in all Maple kernels which reduces the garbage collection times correspondingly. Only for larger

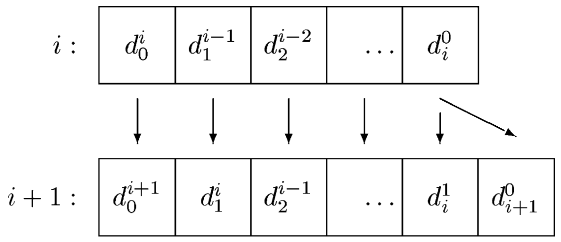

Fig. 20. Computation of derivatives of total order $i + 1$.

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.

4.4. *Neighbourhood analysis of algebraic curves*

A singularity of an algebraic curve is called ordinary, if its multiplicity denotes the number of tangents through this point. An important subproblem in the rational parameterization of algebraic curves (Sendra and Winkler, 1990) is to transform a curve C_i that has non-ordinary singularities into a curve C_{i+1} where a non-ordinary singularity is resolved into ordinary ones. The following presentation of the parallelization of the transformation of C_i to C_{i+1} is based on Schreiner et al. (2000a, 2001).

The most time-consuming part of the transformation is the computation of all singularities of a homogeneous polynomial $p(x, y, z)$ in two steps:

1. **(b, n) := derivatives(p)**
   Let $d_u^v \in \mathbb{Q}(x, z)$ be $\partial p(x, 1, z)^{u+v}/\partial x^u z^v$ of total order $u + v$. We compute a sequence $[b_i]_{i=0}^n$ where b_0 is the greatest square free divisor (gsfd) of $p(x, 1, 0)$, $b_{i+1}(x)$ is the greatest common divisor (gcd) of $b_i(x)$ and of all $d_v^u(x, 0)$ with $u + v = i + 1$, and n is the smallest order such that $\deg b_n = 0$. The roots of $b_i(x)$ are the x-coordinates of all singularities $(x, 1, 0)$ of order at least $i + 1$. The x-coordinates of all singularities $(x, 1, 0)$ of order i are thus the roots of $b_{i-1}(x)/b_i(x)$ for $1 < i \le n$. The partial derivatives of total order $i + 1$ are generated from (and overwrite) the derivatives of order i as shown in Fig. 20.
2. **S := singularities(b, n)**
   Let q_i denote b_{i-1}/b_i for $2 \le i \le n$. Every root a of q_i is the x coordinate of a singularity $(a, 1, 0)$ of multiplicity i; S is the set of all such $(i, (a, 1, 0))$.

The first step accounts for most of the computation time of each curve transformation. Our goal is therefore to speed it up by parallelization.

4.4.1. *Parallel algorithm: manager–worker style*

Fig. 20 suggests to organize the computation of all d_u^v in a triangular matrix shown in Fig. 21: each line i contains all d_u^v with $u + v = i$, each column j contains all d_j^v, i.e.

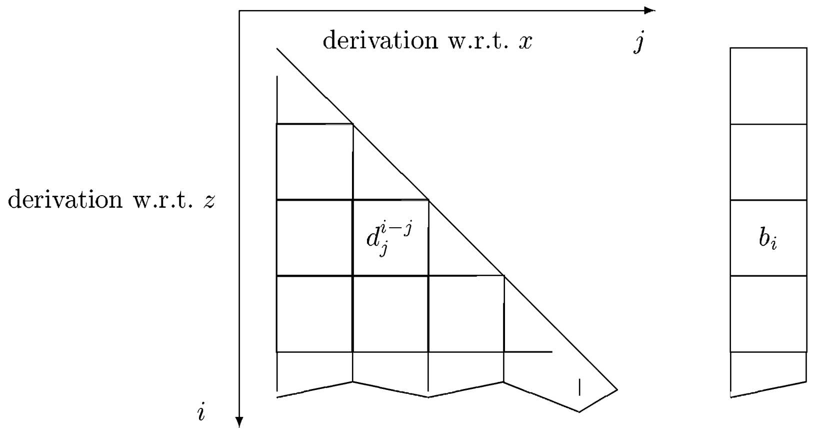

Fig. 21. The matrix of partial derivations.

the matrix holds at position (i, j) the derivative d_j^{i-j}. The basic parallelization idea is to compute all those positions (i, j) with $i \geq j$ in parallel whose data dependencies have been resolved, i.e. for which $(i - 1, j)$ is available (if $i > j$) respectively $(i - 1, j - 1)$ is available (if $i = j$).

However in order to increase its granularity, a task should compute multiple matrix elements, i.e. we have to *block* the computation accordingly. We achieve this by partitioning the triangular matrix into square blocks of size $m * m$ (for some blocking factor m) that comprise all partial derivatives that are computed by the same task. The blocks along the diagonal boundary of the triangular matrix are themselves triangular and only have to compute $(m^2 - m)/2$ elements.

The tasks are created by a main program which computes iteratively the d_u^0, i.e. the derivatives along the diagonal boundary of the matrix. When it has computed the diagonal boundary of a triangular block, it starts a corresponding task that computes the remainder of this block. When a task has terminated, the main program starts a new task for computing that square block that is adjacent to the lower boundary of the result block.

Actually, the result of a task need not be the values of all d_u^v in the corresponding block because we are only interested in

1. the last line of the block which is required by the task computing the adjacent block;
2. the gcd of each line of the block which is required by the main program to compute the greatest common divisor of the whole matrix line.

Since the gcd is commutative and associative, the program may receive in any order the results computed by the tasks of line i and combine them with the current value of b_i. In a final step, b_{i+1} is then combined with b_i.

To let the algorithm efficiently execute on a machine with a limited number of processors, we have to adopt an appropriate *scheduling strategy*: initially, tasks are created for computation of the first p triangular blocks. Whenever a task terminates, we "enable" the adjacent square block whose computation thus becomes possible; if the terminated

```
**(b, n) := derivatives(p)**
  $T := \{\text{task } (im, im) : i = 0 \ldots p-1\}$
  $n := \deg(p)$
  **while** $T \neq \emptyset$ **do**
    wait for some task $(i, j) \in T$ and remove it from $T$
    update $b_i \ldots b_{i+m-1}$ and $n$
    enable $(i+m, j)$
    **if** $i = j$ **then** enable $(i+m, j+m)$ **end**
    disable all $(i, j)$ with $i \geq n$
    **if** there is some enabled $(i, j)$ with minimum $i$ **then**
      disable it and add task $(i, j)$ to $T$
    **end**
  **end**
  **for** $i$ **from** 0 **to** $n-1$ **do**
    $b_{i+1} := \gcd(b_i, b_{i+1})$
  **end**
**end**
```

Fig. 22. The parallel algorithm: manager–worker style.

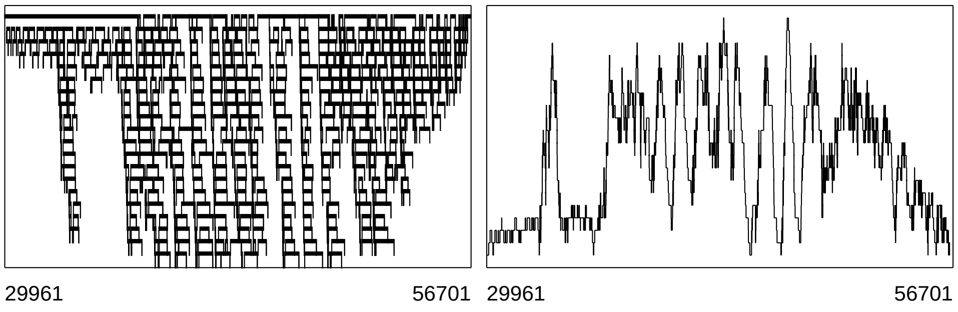

Fig. 23. Manager–worker parallelism.

task has computed a triangular block (and it was not one of the *p* initial tasks), we also enable the subsequent triangular block. Among all enabled blocks, we choose a block with minimum line index, because its computation may make the computation of blocks with larger indices superfluous. When the termination criterion is detected in line *i*, only those tasks will be started that operate on lines with indices less than *i*; when no more task is active, the algorithm terminates.

We have benchmarked the program with four examples for which the sequential computation on a PIII@450 MHz Linux PC takes 552, 198, 402, and 1798 s, respectively. The examples were extracted from the resolution of the singularities of four artificially

Example	System	1	2	4	8	16	24
1 (552s = 88%)	Origin	560	294	190	113	83	80
	Cluster	817	436	233	155	100	101
2 (198s = 85%)	Origin	192	123	70	44	41	45
	Cluster	214	142	91	64	45	49
3 (402s = 95%)	Origin	371	218	128	70	59	62
	Cluster	518	265	153	107	71	73
4 (1798s = 95%)	Origin	2199	928	574	348	239	228
	Cluster	3015	1222	703	469	275	241

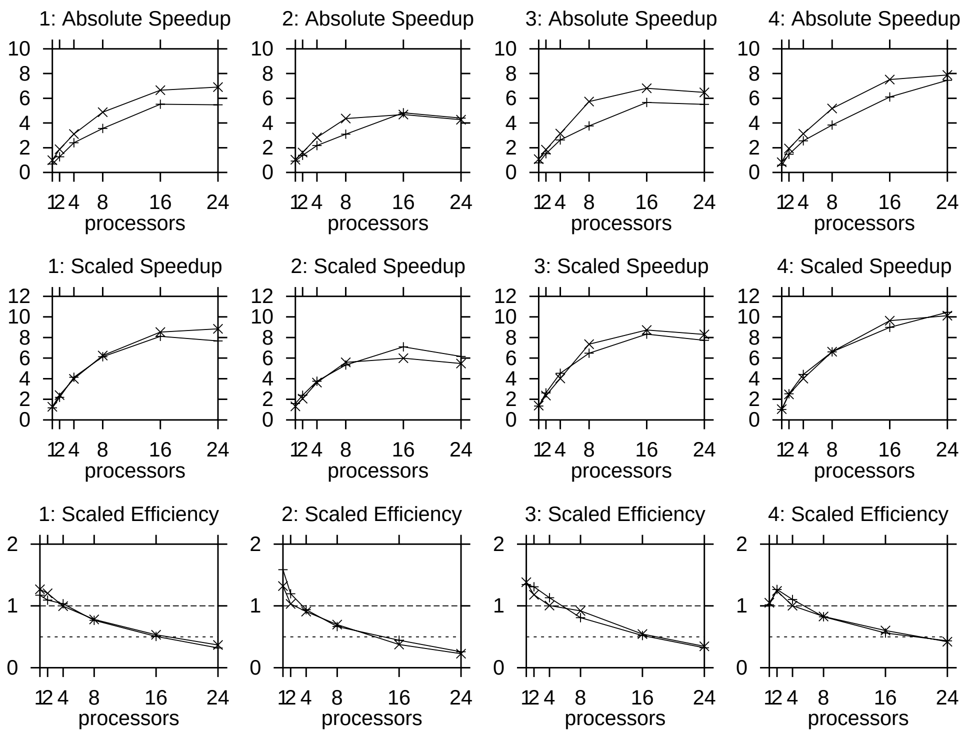

Fig. 24. Experimental results.

generated curves in which they represented the most time-consuming part (88, 85, 95, and 95% of the total runtime). The program has been executed in two environments: a SGI Origin 2000 multiprocessor (64 R12000@300 MHz, 24 processors used) and a cluster of four SGI Octanes (2 R10000@250 MHz each) and of 16 of the Origin processors. The table in Fig. 24 lists the execution times; the subsequent diagrams display the same information as described in Section 4.1.1.

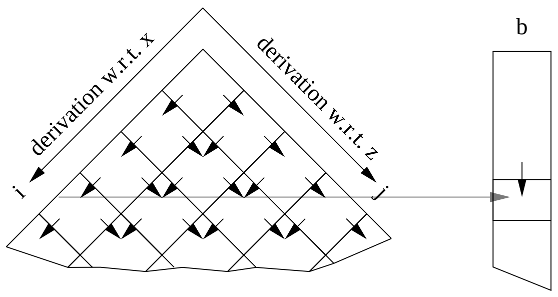

Fig. 25. Data dependencies.

While the parallelization achieved some speedup, the utilization diagram of a sample run in the right of Fig. 23 shows that there is much room for improvement: rarely all machines are busy, in average only about 50% of the computing resources are utilized. Apparently, the manager is not able to issue parallel tasks at a rate that is sufficiently high to provide idle machines with work, i.e. the explicit task scheduling becomes a performance bottleneck.

4.4.2. Parallel algorithm: dataflow style

Because of the low utilization of the manager–worker solution, Schreiner (2001) explores a solution where tasks are created whenever their data dependencies have been resolved. This "dataflow" solution exhibits all parallelism inherent in the problem and leaves the scheduling details to the runtime system.

We decompose the triangle matrix of partial derivatives as shown in Fig. 25: the decomposition yields uniform square blocks of size m^2 (for some m) such that each block can be identified by the coordinate (i, j) of its upper point denoting d_i^j and contains all d_u^v with $0 \leq u < i + m$ and $0 \leq v < j + m$. Once d_i^j is known, all other elements of the block can be determined by derivation with respect to x or z, respectively. As in the original algorithm, we compute within a block the gcd of all d_u^v with the same sum $u + v$ which then contributes to the result vector b_{u+v}. However, now the lines/columns of each block run diagonal to the computation of the derivatives as sketched by the grey arrow in Fig. 25.

Since all elements in a block can be computed from the first element d_i^j, we can start the computation of a block when one of the following condition holds:

- $i = 0$ and d_i^{j-1} is known.
- $j = 0$ and d_{i-1}^j is known.
- $i > 0$ and $j > 0$ and
  - d_i^{j-1} is known *or*
  - d_{i-1}^j is known.

The main program starts the tasks in some order compatible with the task dependencies and passes to each task the references to those tasks that the new task potentially depends

```
$(\boldsymbol{b}, \boldsymbol{n})$ := derivatives($\boldsymbol{p}$)
  $n := \min(n_x, n_z)$
  for $k$ from $0$ to $n$ by $m$ do
    for $(i, j)$ in tasks($k$) do
      if $i = 0 \wedge j = 0$ then $t_{i,j}$ := start task$(i,\, j,\, m,\, p)$
      else if $i = 0$ then $t_{i,j}$ := start task$_{\mathrm{R}}(i,\, j,\, m,\, t_{i,j-1})$
      else if $j = 0$ then $t_{i,j}$ := start task$_{\mathrm{L}}(i,\, j,\, m,\, t_{i-1,j})$
      else $t_i$ := start task$_{\mathrm{I}}(i,\, j,\, m,\, t_{i,j-1},\, t_{i-1,j})$
    end
  end
  for $k$ from $0$ to $n$ by $m$ do
    $tset := \{t_{i,j} : (i, j) \in \mathrm{tasks}(k)\}$
    while $tset \neq \emptyset \wedge n = \min(n_x, n_y)$ do
      $t$ := select $tset$
      $(\_, \_, g)$ := wait $t$
      update $b_{i+j} \ldots b_{(i+m)+(j+m)}$ and $n$
      tset := tset-$\{t\}$
    end
  end
  for $k$ from $n + m$ to $\min(n_x, n_y)$ by $m$ do
    for $(i, j) \in$ tasks($k$) do stop $t_{i,j}$; end
  end
  for $t \in tset$ do stop $t$ end
end
```

$$\mathrm{tasks}(k) := \{(i, j) : i + j = k \wedge i \bmod m = 0 \wedge j \bmod m = 0\}$$

Fig. 26. The parallel algorithm: dataflow style.

on. Then the program waits for all tasks (i, j) in the order of increasing $i + j$, combines the computed gcds with the vector b and signals by updating n whether the computation can be prematurely terminated. All tasks whose results are not required any more are then stopped. The main program is depicted in Fig. 26.

We have benchmarked the program in a cluster of Linux PCs with problems 1, 2, and 4 described in Section 4.4.1. Fig. 27 illustrates an execution of Example 1 with block size $m = 4$. In the new algorithm, all machines get saturated for 65% of the computation time. The overall utilization is about 75%.

The actual execution times of the new algorithm in comparison with the execution times of the old algorithm are listed in Fig. 28: the top row of diagrams shows the execution time of the programs, the bottom row shows the speedup that the new algorithm gains *over the original one* with the following variants:

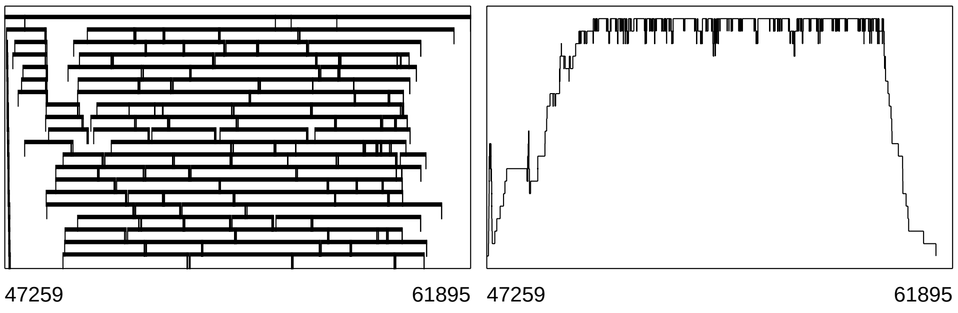

Fig. 27. Dataflow parallelism.

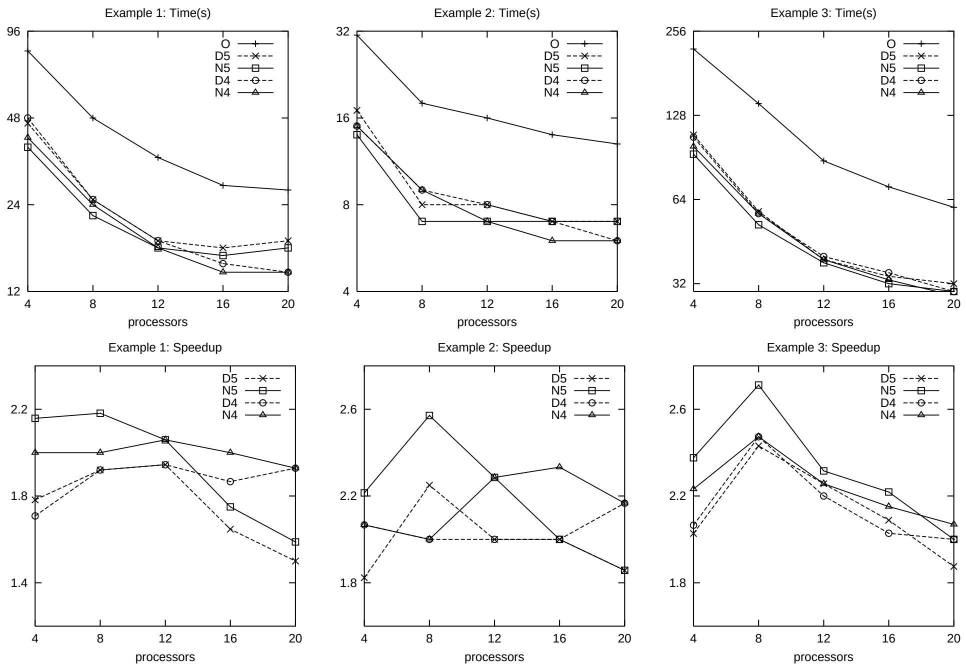

Fig. 28. Execution times and speedups.

- O: the original algorithm.
- N4, N5: the new algorithm with non-deterministic task selection in the main program with block sizes $m = 4$ and 5.
- D4, D5: the new algorithm with deterministic selection in the main program ($t :=$ first(*tset*)) with block sizes $m = 4$ and 5.

All variants of the new algorithm are considerably faster than the original version with an average improvement of 1.8 (Example 1) respectively 2.2 (Examples 2 and 3).

We could thus reduce the sequential execution time from 552, 198 s, respectively 1798 s on a PIII@450 MHz PC down to a parallel execution time of 14, 6 s, respectively 29 s. The fact that the speedups are drastically superlinear is caused by the changed order in which the greatest common divisors are computed (also in the original parallel algorithm); this is a hint for a general algorithmic improvement of the sequential algorithm.

5. Conclusions

The main advantage of Distributed Maple is that it represents on the basis of a widespread commercial computer algebra system a reliable, portable and easy to use platform for the development of parallel computer algebra algorithms. Provided that the algorithms exhibit sufficient parallelism with medium to large grain size (at least 1 s), we can expect reasonable speedups in environments with up to 30 machines/processors or so. The comparatively high-level parallel programming model and the supporting visualization tools allow us to experiment with little effort on different parallelization strategies for a given problem and to investigate the effects on dynamic behaviour and performance.

On the other hand, the hybrid nature of the environment and its high level of abstraction require a number of compromises such that we cannot expect to get the fastest possible parallel solution for a given problem. Given enough time and manpower, a native C implementation on the basis of a message passing library such as MPI or PVM will always yield a more efficient implementation. Nevertheless for the many computer algebra programmers that want to get rather good parallelization success with limited efforts on the basis of existing Maple code, Distributed Maple provides a powerful work platform.

Acknowledgements

Karoly Bosa has implemented under the supervision of Wolfgang Schreiner the fault tolerance mechanisms described in Section 3.5. Many of the parallel algorithms described in Section 4 have in their initial versions been developed by Christian Mittermaier under the supervision of Wolfgang Schreiner. Franz Winkler has initiated the parallelization of parts of CASA and thus triggered the development of Distributed Maple. Furthermore, he has provided insight on the mathematical details of the CASA algorithms that was fundamental for their parallelization. This work is supported by grant SFB F013/F1304 of the Austrian Science Foundation (FWF).

Appendix

The inputs for the benchmarks presented in Section 4 are available at http://www.risc.uni-linz.ac.at/software/distmaple/benchmarks.

References

Bernardin, L., 1997. Maple on a massively parallel, distributed memory machine. In: Hitz, M., Kaltofen, E. (Eds.), PASCO'97—Second International Symposium on Parallel Symbolic Computation, Maui, Hawaii, July 20–22, 1997, ACM Press, New York, pp. 217–222.

Bertoli, P., Hong, H., Neubacher, A., Schreiner, W., Stahl, V., April 1994. The C++ Interface to the STURM Distributed Multiprocessor Kernel. Technical Report 94-32, RISC-Linz, Johannes Kepler University, Linz, Austria.

Bubeck, T., Hiller, M., Küchlin, W., Rosenstiel, W., 1995. Distributed symbolic computation with DTS. In: Ferreira, A., Rolim, J. (Eds.), IRREGULAR'95—Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, Lyon, France, September 1995, Lecture Notes in Computer Science, vol. 980, Springer, Berlin, pp. 231–248.

Chan, K.C., Diaz, A., Kaltofen, E., 1994. A distributed approach to problem solving in Maple. In: Lopez, R.J. (Ed.), Maple V: Mathematics and its Application, Proceedings of the Maple Summer Workshop and Symposium (MSWS'94). Birkhäuser, Boston, pp. 13–21.

Char, B., 1990. 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, 1990, ACM Press, pp. 96–103.

Char, B., Johnson, J., 1997. Some experiments with parallel bignum arithmetic. In: Hitz, M., Kaltofen, E. (Eds.), PASCO'97—Second International Symposium on Parallel Symbolic Computation, Maui, Hawaii, July 20–22, 1997, ACM Press, New York, pp. 94–103.

Collins, G.E., 1971. The calculation of multivariate polynomial resultants. J. ACM 18, 515–532.

Collins, G.E., Akritas, A.G., 1976. Polynomial real root isolation using Descartes' rule of signs. In: Jenks, R.D. (Ed.), ACM Symposium on Symbolic and Algebraic Computation, ACM, New York, pp. 272–275.

Collins, G.E., Johnson, J.R., Küchlin, W., May 1990. Parallel real root isolation using the coefficient sign variation method. In: Zippel, R.E. (Ed.), Computer Algebra and Parallelism, Lecture Notes in Computer Science, Ithaca, USA, Springer-Verlag, Berlin, pp. 71–81.

Cooperman, G., 1998. GAP/MPI: Writing Parallel Programs in GAP Easily, Technical Report, College of Computer Science, Northeastern University, Boston, MA.

Decker, T., Krandick, W., 1999. Parallel real root isolation using the Descartes method. In: Banerjee, P., Prasanna, V.K., Sinha, B.P. (Eds.), High Performance Computing—HiPC'99, Sixth International Conference, Calcutta, India, December 17–20, 1999, Lecture Notes in Computer Science, vol. 1745, Springer, Berlin, pp. 261–268.

Della Dora, J., Fitch, J. (Eds.), 1989. Computer Algebra and Parallelism, Academic Press, London, UK.

Diaz, A., Kaltofen, E., 1998. FoxBox: a system for manipulating symbolic objects in black box representation. In: Gloor, O. (Ed.), ISSAC'98, International Symposium on Symbolic and Algebraic Computation, ACM Press, New York.

Diaz, A., Kaltofen, E., Schmitz, K., Valente, T., 1991. DSC—a system for distributed computation. In: Watt, M. (Ed.), Proceedings of ISSAC'91, ACM Press, New York, pp. 324–333.

Gautier, T., Hong, H., Roch, J.-L., Schreiner, W., 2001. Parallel implementation. In: Weispfennig, V., Grabmeier, J., Kaltofen, E. (Eds.), Handbook of Computer Algebra—Foundations, Applications, Systems, Springer, Heidelberg, Chapter 2.16 (to appear).

Gautier, T., Roch, J.-L., 1994. PAC++ system and parallel algebraic numbers computations. In: Hong, H. (Ed.), PASCO'94—First International Symposium on Parallel Symbolic Computation, Hagenberg/Linz, Austria, September 26–28, 1994, World Scientific Publishing, Singapore, pp. 145–153.

Hitz, M., Kaltofen, E. (Eds.), 1997. PASCO'97—Second International Symposium on Parallel Symbolic Computation, Maui, Hawaii, July 20–22, 1997, ACM Press, New York.

Hong, H., 1993. Parallelization of quantifier elimination on workstation network. In: Cohen, G., Mora, T., Moreno, O. (Eds.), AAECC-10—Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Tenth International Symposium, San Juan, Puerto Rico, May 1993, Lecture Notes in Computer Science, vol. 673, Springer, Berlin, pp. 170–179.

Hong, H. (Ed.), 1994. PASCO'94—First International Symposium on Parallel Symbolic Computation, Hagenberg/Linz, Austria, September 26–28, 1994, World Scientific Publishing, Singapore.

Hong, H., Loidl, H.W., 1994. Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In: Buchberger, B., Volkert, J. (Eds.), CONPAR 94-VAPP VI—Third Joint International Conference on Vector and Parallel Processing, Linz, Austria, September 6–8, 1994, Lecture Notes in Computer Science, vol. 854, Springer, pp. 325–336.

Hong, H., Neubacher, A., Schreiner, W., 1995. The design of the SACLIB/PACLIB kernels. J. Symb. Comput. 19, 111–132.

Küchlin, W., 1990. PARSAC-2: A parallel SAC-2 based on threads. In: Sakata, S. (Ed.), AAECC-8, Eighth International Conference on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes, Tokyo, Japan, August 1990, Lecture Notes in Computer Science, vol. 50, Springer, Berlin, pp. 341–353.

Loidl, H.W., Trinder, P.W., Hammond, K., Junaidu, S.B., Morgan, R.G., Jones, S.L.P., 1999. Engineering parallel symbolic programs in GPH. Concurrency—Practice and Experience 11 (12), 701–751.

Maple, W., Maple 6, 2001. Available from http://www.maplesoft.com.

Metzner, T., Radimersky, M., Sorgatz, A., Wehmeier, S., 1999. User's Guide to Macro Parallelism in MuPAD 1.4.1, Teubner, Stuttgart, Germany.

Mittermaier, C., 2000. Parallel Algorithms in Constructive Algebraic Geometry. Master's Thesis, Johannes Kepler University, Linz, Austria.

Mittermaier, C., Schreiner, W., Winkler, F., 2000. A parallel symbolic-numerical approach to algebraic curve plotting. In: Gerdt, V., Mayr, E.W. (Eds.), CASC-2000, Third International Workshop on Computer Algebra in Scientific Computing, Samarkand, Uzhekistan, October 5–9, 2000, Springer, Berlin, pp. 301–314.

Mnuk, M., Winkler, F., 1996. CASA—A system for computer aided constructive algebraic geometry. In: Calmet, J., Limongelli, C. (Eds.), DISCO'96—International Symposium on the Design and Implementation of Symbolic Computation Systems, Karsruche, Germany, Lecture Notes in Computer Science, vol. 1128, Springer, Berlin, pp. 297–307.

Nam, T.Q., 1994. Extended Newton's method for finding the roots of an arbitrary system of equations and its applications. In: IASTED'94, Twelvth International Conference on Applied Informatics, Annecey, France, 1994.

Pau, C., Schreiner, W., July 2000. Distributed Mathematica—User and Reference Manual. Technical Report 00-25, RISC-Linz, Johannes Kepler University, Linz, Austria.

Petcu, D., 2000. PVMaple, a distributed approach to cooperative work of maple processes. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Seventh European PVM/MPI Users' Group Meeting, Balatonfüred, Lake Balaton, Hungary, September 10–13, 2000, Lecture Notes in Computer Science, vol. 1908, Springer, Berlin, pp. 216–224.

Roch, J.-L., Villard, G., 1997. Parallel computer algebra. Lecture Notes for a Tutorial, ISSAC'97, Hawaii, July 1997, Available from http://www-apache.imag.fr/~jlroch/ps/97-issac.ps.gz.

Schreiner, W., 1996. A para-functional programming interface for a parallel computer algebra package. J. Symb. Comput. 21 (4–6), 593–614.

Schreiner, W., May 1998. Distributed Maple—User and Reference Manual. Technical Report 98-05, RISC-Linz, Johannes Kepler University, Linz, Austria, Available from http://www.risc.uni-linz.ac.at/software/distmaple.

Schreiner, W., 1999. Developing a distributed system for algebraic geometry. In: Topping, B.H.V. (Ed.), EURO-CM-PAR'99, Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics, Weimar, Germany, March 20–25, 1999, Civil-Comp Press, Edinburgh, pp. 137–146.

Schreiner, W., November 2000. Analyzing the Performance of Distributed Maple. Technical Report 00-32, RISC-Linz, Johannes Kepler University, Linz, Austria.

Schreiner, W., 2001. Manager-worker parallelism versus dataflow in a distributed computer algebra system. In: Malyshkin, V. (Ed.), PaCT'2001, Parellel Computing Technologies, 6th International Conference, September 3–7, 2001, Novosibirsk, Russia. Lecture Notes in Computer Science, vol. 2127, Springer, Berlin, pp. 329–343.

Schreiner, W., Kusper, G., Bosa, K., 2001. Fault tolerance for cluster computing based on functional tasks. In: Sakellarion, R., Keane, J., Gurd, J., Freeman, L. (Eds.), 7th International Euro-par Conference, Manchester, UK, August 28–31, 2001. Lecture Notes in Computer Science, vol. 2150, Springer, Berlin, pp. 712–716.

Schreiner, W., Loidl, H.-W., November 2000. GHC-Maple Interface, Available from http://www.risc.uni-linz.ac.at/software/ghc-maple.

Schreiner, W., Mittermaier, C., Winkler, F., 2000a. Analyzing algebraic curves by cluster computing. In: Distributed and Parallel Systems—From Instruction Parallelism to Cluster Computing, DAPSYS'2000, Third Austrian–Hungarian Workshop on Distributed and Parallel Systems, Balatonfüred, Hungary, September 10–13, 2000, Kluwer, Boston, pp. 175–184.

Schreiner, W., Mittermaier, C., Winkler, F., 2000b. On solving a problem in algebraic geometry by cluster computing. In: Euro-Par 2000, Sixth International Euro-Par Conference, Munich, Germany, August 29–September 1, 2000, Lecture Notes in Computer Science, vol. 1900, Springer, Berlin, pp. 1196–1200.

Schreiner, W., Mittermaier, C., Winkler, F., 2000c. Plotting algebraic space curves by cluster computing. In: Gao, X.-S., Wang, D. (Eds.), Fourth Asian Symposium on Computer Mathematics, Chiang Mai, Thailand, December 17–21, 2000, World Scientific Publishers, Singapore, pp. 49–58.

Seitz, S., May 1990. Algebraic computation on a local net. In: Zippel, R.E. (Ed.), Computer Algebra and Parallelism, Lecture Notes in Computer Science, Ithaca, USA, Springer-Verlag, Berlin, pp. 19–31.

Sendra, R.J., Winkler, F., 1990. Symbolic parametrization of curves. J. Symb. Comput. 12 (6), 607–631.

Siegl, K., 1993. Parallelizing algorithms for symbolic computation using ||MAPLE||. In: Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 19–22, 1993, ACM Press, New York, pp. 179–186.

Wang, D., 1991. On the parallelization of characteristic-set-based algorithms. In: Zima, H.P. (Ed.), Parallel Computation—First Internationl ACPC Conference, Lecture Notes in Computer Science, vol. 591, Springer, Berlin, pp. 338–349.

Zippel, R.E. (Ed.), May 1990. Computer Algebra and Parallelism, Lecture Notes in Computer Science, Ithaca, USA, Springer-Verlag, Berlin.