Analyzing the Performance of Distributed Maple on Various Parallel and Distributed Systems*

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

November 22, 2000

Abstract

This report systematically analyzes the performance of the parallel computer algebra environment Distributed Maple on various multiprocessors and networked computers using artificial benchmarks as well as real-life applications. This study exhibits the performance bottlenecks of the environment and sheds light on the suitability of different architectures for Distributed Maple applications.

*Supported by the grants SFB F013/F1304 of the Austrian Science Foundation (FWF) and HPRI-1999-CT-00026 of the EU Human Potential Programme.

Contents

1 Introduction

In the last two years, researchers at RISC-Linz have developed on the basis of Distributed Maple various parallel computer algebra algorithms and applications in the area of algebraic geometry, e.g.

- parallel resultant computation [3],
- parallel plotting of algebraic curves in the plane [5, 1],
- parallel plotting of algebraic space curves [4],
- parallel computation of the neighborhood graph of an algebraic curve [6].

The parallel programs have been executed in various parallel and networked environments, such as

- our local network of Unix workstations and Linux PCs that were formerly linked by shared 10Mbit Ethernet segments and are now connected by switched 100Mbit Ethernet lines,
- a Silicon Graphics Origin 2000 NUMA multiprocessor with 64 processors located in the computing center of our university,
- a combination of machines from our local network and of the Origin, both interconnected by an ATM line over which multiple logical Ethernet connections are multiplexed.

In addition, we recently got in the frame of the European programme TRACS (Training and Research on Advanced Computing Systems) access to the following parallel architectures in the Edinburgh parallel computing center:

- an 18 processor Sun HPC 6500 bus-based shared memory multiprocessor,
- a Beowulf Cluster with one frontend node and 16 compute nodes running under Linux and connected by two 100Mbit Ethernet networks.

Having access to this variety of applications and architectures, we took the opportunity to pursue a (long overdue) investigation on the performance of Distributed Maple. We have organized this study in two parts:

1. a low-level analysis of the system on different architectures based on a simple model of its internal operation, and
2. a high-level comparison of the performance of multiple applications on the available architectures.

The purpose of this investigation is twofold: first, to get an improved understanding of the performance characteristics of Distributed Maple; second, to analyze the suitability of various architectures (multiprocessors and clusters) for the purpose of parallel computer algebra on the basis of this environment.

The report is consequently organized as follows:

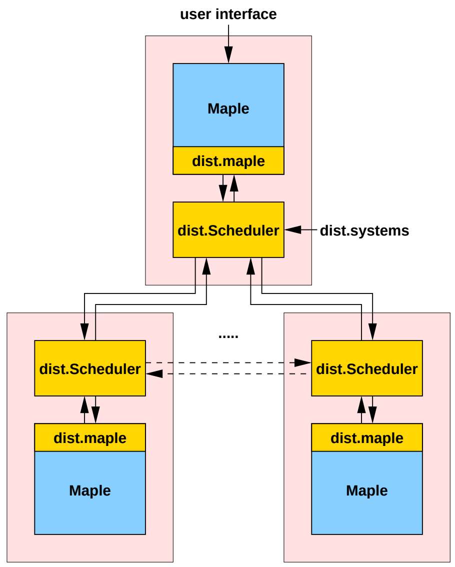

Figure 1: Distributed Maple Architecture

- Section 2 gives a short overview of Distributed Maple,
- Section 3 presents a simple model of the system which is then used to analyze its performance on various architectures with the help certain key benchmarks,
- Section 4 compares the performance of four parallel computer algebra algorithms/applications on the available architectures and attempts to explain them on the basis of the previous results,
- Section 5 summarizes our findings and conclusions.

2 Distributed Maple

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple [2]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node connected to a session comprises two components (see Figure 1):

Scheduler The Java program `dist.Scheduler` coordinates the node interaction. The scheduler process attached to the frontend kernel starts instances of the scheduler on other machines and communicates with them via Java sockets.

Maple Interface The program `dist.maple` running on every Maple kernel implements the interface between kernel and scheduler. Both components use (named) pipes to exchange messages (which in general embed Maple objects in a linearized format).

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

For more details of the system, see [2].

3 System Benchmarks

In this section, we construct a simple performance model of Distributed Maple and benchmark various architectures on the basis of this model. We thus improve our understanding of the performance characteristics of the system with respect to different architectures.

3.1 Performance Model

The system can be thought of multiple *layers* which communicate by exchanging messages:

- *Between Maple and the scheduler process:* The Maple processes linearizes an object in the Maple heap to a sequence of bytes ("a message") which is written to a Unix FIFO file ("named pipe"). From this pipe, an *input thread* of the scheduler process reads the message into the process memory. Conversely, an *output thread* of the scheduler process writes such a message to an (unnamed) pipe from where the Maple process reads it and reconstructs it as a Maple object in its own memory.
- *Within the scheduler process:* an input thread determines the destination of the received message and places it into a memory buffer shared with either some output thread for further submission or with the server thread for local processing. The server thread takes one message after the other from its input buffer, handles it, and potentially generates new messages which are placed into the buffer of some output thread.
- *Between the scheduler processes:* an output thread takes one message after the other from its input buffer and writes it to a socket; this socket is connected to a scheduler process on a remote machine where an input thread reads the message into the memory of the local process.

Corresponding to this model, the communication performance of the system is crucially affected by the *latency* imposed by forwarding a message between two subsequent system layers. For instance, the time for creating a task in the local Maple process and submitting it to a remote Maple process is essentially composed of the following components:

1. linearization of the Maple task description by the Maple process to build a message,

2. sending the message through a pipe to the input thread of the scheduler process,
3. reading of the message by the input thread into the memory of the scheduler process,
4. transferring the message from the input thread to the server thread via a shared buffer,
5. processing the message by the server thread,
6. forwarding the message to some output thread via a shared buffer,
7. sending the message through a socket to the input thread of a remote server process,
8. reading of the message by the input thread into the memory of the remote scheduler process,
9. transferring the message from the input thread to the server thread via a shared buffer,
10. processing the message by the server thread,
11. forwarding the message to some output thread via a shared buffer,
12. sending the message through a pipe to the Maple process,
13. reading of the message by the Maple process and reconstructing it as a Maple task description.

We do not attempt to measure each of these times individually (which is very hard for some of them) but restrict our observations to the following key times (see Figure 2):

- Thread Roundtrip: the time it takes a message to travel from a thread via a shared buffer to another thread and back (measured by a dummy message).
- Local Roundtrip: the time it takes to query the result of a task stored in the local scheduler process (i.e., the time from the call of `dist[wait]` until the return from this call).
- Remote RoundTrip: the time it takes a message to travel from an output thread via a socket to an input thread and back (measured by a dummy message).
- Pipe Out: the time it takes to write a small message to an output stream (this is a bit different for Maple and Java, we ignore the difference and take the slightly larger Maple value).
- Socket Out: the time it takes for an output thread to write a small message to a socket connected to a remote machine.

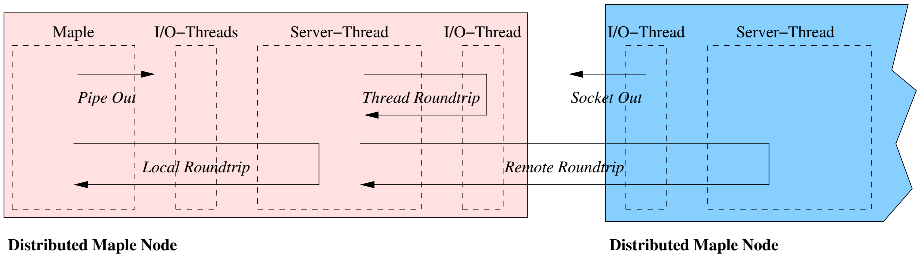

Figure 2: Communication Model

- Start Overhead: the time the linearization of a call `dist[start](f,0)` takes (for some function `f`).
- Wait Overhead: the time the linearization of a call `dist[wait](0)` takes.
- Latency: the minimun time difference between two messages sent by a remote node and received by the local Maple kernel.

  We measure this time by starting in a first phase a very large number of tasks and then waiting in a second phase for the results of these tasks measuring the time of this second phase. By setting in the `dist.systems` entry [2] for the remote machine the parameters `maxvalue` and `minvalue` both to -1, all generated tasks are immediately forwarded to the remote machine as a stream of `task` messages; from this machine a sequence of alternate `result` and `close` messages are returned. Since this return stream determines the time of the second phase, we can measure effectively the rate of remote message delivery.

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); this communication operation also includes the conversion between the Maple-internal and the external (linear) representation of a Maple object;
- Socket bandwidth: the amount of data that can be communicated per time unit between two instances of the Java scheduler (across nodes); this communication operation only involves the transfer of raw bytes along a socket (without conversion).

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

All measurements shown in the following sections were performed in a large number of rounds to determine precise results.

3.2 Linux PC

The following execution times refer to two PCs with PIII@500MHz processors running under Linux 2.2.13-tcpfix (TCP/IP patch applied) and connected by a 100Mbit switched Ethernet line. Distributed Maple uses the Sun port of JDK 1.3 for Linux and Maple V Release 5.1. The JVM settings (green threads versus native threads) did not significantly influence the results.

Measurement	Time (ms)
Thread Roundtrip	<0.1
Local Roundtrip	0.8
Remote Roundtrip	31.0
Pipe Out	0.5
Socket Out	<0.1
Start Overhead	0.2
Wait Overhead	0.1
Latency	0.5
Bandwidth pipe:	1030 KB/s
Bandwidth socket:	3938 KB/s

We can infer that

- intra-process communication (as determined by the thread roundtrip time) can be neglected in all subsequent analysis;
- a local task result can be queried and retrieved in about 0.8 ms, i.e., about 1250 local task results per second can be queried;
- a remote task result can be queried and retrieved in about 31.8 $(= 0.8 + 31.0)$ ms, i.e., about 31 remote task results per second can be queried;
- a new task can be created in about 0.6 $(= 0.5 + 0.1)$ ms, i.e., about 1660 tasks per second can be created;
- it takes a message about 15.5 $(= 31/2)$ ms to travel to a remote process;
- the latency indicates that about 2000 messages per second can be transmitted;
- the pipe bandwidth is about four times less than the socket bandwidth (which is about 40% of the raw network bandwidth); thus it is the rate at which Maple can receive data rather than the network capacity that limits the transfer of large objects between Maple kernels. The pipe bandwidth is actually only 70% of the bandwidth that a process can write to a pipe on the denoted PC; this shows the overhead that is necessary for converting an object in the Maple heap to the external linear representation.

As for the rate in which tasks can be distributed to remote machines, we can infer the following facts:

- if the `maxload` parameter in `dist.systems` is set to 0, a remote task can be started on the corresponding machine every 32.1 ms ($= 15.5 + 0.5 + 0.1 + 15.5 + 0.5$) which is the sum of the times for sending a `task` message, forwarding this message to Maple, linearizing the task result to a `result` message, delivering this message to the scheduler, and from there to the local scheduler followed by a `close` message that indicates that a new task may be delivered.

  Essentially, the rate of task execution and result delivery is thus determined by the remote roundtrip time (plus some minor overhead).

- if the `maxload` parameter in `dist.systems` is set to a large ($\gg 10$) value, a remote task can be started at most every 0.25 ($= 0.5/2$) ms; however the corresponding results can be only delivered every 0.5 ms.

  Essentially, for a large number of tasks, the rate of task execution and result delivery is thus determined by the latency.

The fact that the remote roundtrip time is 31 ms (about 300 times the roundtrip time indicated by the Unix command `ping`) indicates the basic overhead associated to individual software layers involved in sending a message through the Java socket interface. However, the fact that the latency is 0.5 ms (31 times less than 15.5 ms) shows that the communication path is either heavily pipelined and/or that the overhead only occurs when a thread is blocked on an empty socket and has to be awaked (which we suspect).

For efficient execution on Linux PCs, the granularity of tasks should therefore be in the order of at least one second or so.

3.3 SGI Octane

The following execution times refer to two SGI Octanes with R10000@250MHz processors running under Irix 6.5 and connected by a 100Mbit switched Ethernet line. Distributed Maple uses the SGI port of JDK 1.2.2 for Irix and Maple V Release 5.0. The JVM settings (green threads versus native threads, JIT enabled versus JIT disabled) do not significantly influence the results.

Measurement	Time (ms)
Thread Roundtrip	<0.1
Local Roundtrip	1.2–2
Remote Roundtrip	150–300
Pipe Out	0.7
Socket Out	0.4
Start Overhead	0.4
Wait Overhead	0.3
Latency	10–12
Bandwidth pipe:	853 KB/s
Bandwidth socket:	4100 KB/s

Most of the results (and the subsequent analysis) are similar to those of the previous section with one major exception: the remote roundtrip time is extremely large (up to 300 ms and more) with a very large variance (about 10% are still outside the range of 150–300 ms). However, as the previous case, the message latency is much lower which indicates that the communication path is heavily pipelined or (more likely) that the overhead is in notifying a thread blocked on an empty socket about the arrival of a message.

More detailed benchmarks combining a PC with an Octane (once as a local machine, once as a remote machine) indicate that it is essentially the Octane as the receiving partner of a communication that it is responsible for the large roundtrip time: sending a message from an Octane to a PC takes 20–50 ms but sending a message in the other direction takes 150–200 ms (in both situations again with very large variance).

The results essentially independent of the fact whether both communication partners reside on a single machine (a dual-processor Octane with a single bus shared memory) or on an SGI Origin 2000 distributed shared memory multiprocessor with a fast interconnection fabric. In all these situations, the JDK port apparently uses the same communication mechanism. This problem is also independent of the version of JDK (both Java 1.1.5 and 1.2.2 were used); it thus indicates a general deficiency of the Java Virtual Machine port for MIPS/Irix architectures with respect to socket communication.

The consequences for the rate at which tasks can be sent to a remote machine are as follows:

- if the `maxload` parameter is set to 0, remote tasks can be started every 150–300 ms (with a corresponding result delivery rate),
- if the `maxload` parameter is set to a very large value, a remote task can be started every 5–6 ms (but results can be only delivered at a rate of 10–12 ms).

Depending on the load distribution scheme, the granularity of tasks should be therefore not less than say five seconds (first case) respectively one second (second case).

3.4 Sun E3000

The following execution times refer a Sun E3000 bus-based shared memory multiprocessor with 250 MHz UltraSPARC processors. We use Sun JDK 1.2.2 for Solaris (native threads, JIT enabled) and Maple V Release 5.1.

Measurement	Time (ms)
Thread Roundtrip	<0.1
Local Roundtrip	1
Remote Roundtrip	100
Pipe Out	0.1
Socket Out	<0.1
Start Overhead	0.3
Wait Overhead	<0.1
Latency	0.8
Bandwidth pipe:	512 KB/s
Bandwidth socket:	5120 KB/s

Again, most of the analysis is similar to the preceding sections. It is striking to see that the shared memory system provides a bit higher socket bandwidth than the networked systems discussed in the previous sections but that the remote roundtrip time is about three times higher than in the case of the networked Linux PCs (with little variance, unlike the SGI Origin). The actual latency between subsequent messages is again very small, similar to the Linux PCs; many of the other key timings are below the measurement threshold.

4 Application Benchmarks

The following sections are based on several application benchmarks published elsewhere [3, 5, 4, 6]. The published benchmarks were performed on a cluster of workstations and Linux PCs, on an Origin multiprocessor, and combinations of both environments. We are now going to extend these results with results gained from the following systems at the Edinburgh Parallel Computing Center:

1. an 18-processor Sun HPC 6500 system with 400MHz UltraSPARC II processors and 18 Gbytes of shared memory,
2. a Linux-based Beowulf cluster with one frontend node with a PIII@500MHz processor and 16 compute nodes with AMD-K7@650Mhz processors all equipped with 128 MB memory and linked by two 100Mbit switched Ethernets (one for system traffic, one for application traffic).

4.1 Resultant Computation

This benchmark demonstrates the parallelization of a fundamental computer algebra algorithm, the computation of the resultant of bivariate polynomials, by a modular algorithm which determines many homomorphic images of the results [3]. The benchmarks were performed on the following environments:

- C: our heterogeneous network of 4 SGI Octanes (2 R10000@250MHz each) and 8 Linux PCs (PII@266–350MHz) linked by three 10Mbit shared Ethernet segments,

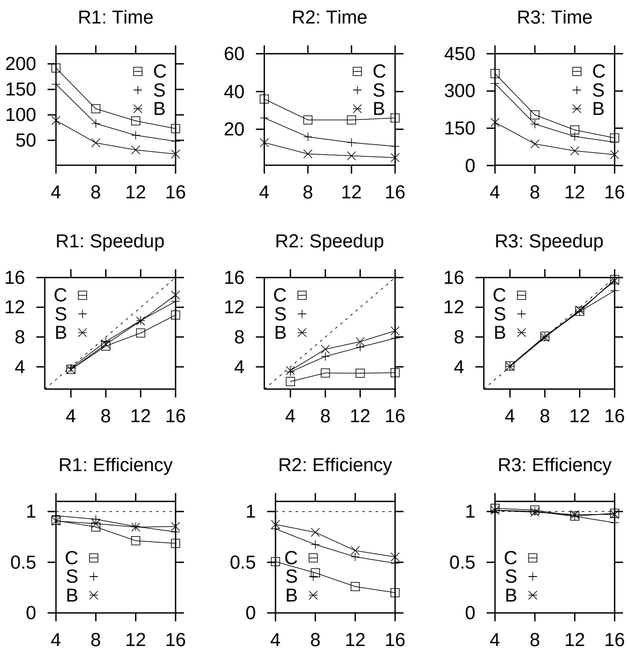

Figure 3: Resultant Computation

- S: the Sun HPC 6500,
- B: the Beowulf.

The results are depicted in Figure 3:

- the top row shows the absolute execution times for a varying number of processors in each environment,
- the medium row shows the scaled speedups, i.e., the absolute speedups (relative to sequential execution on a PIII@450MHz Linux PC) multiplied by a scaling factor that adjusts the speed of the individual environments (such that the speedup curves can be compared),
- the bottom row shows the scaled efficiencies, i.e., the scaled speedups divided by the number of processors.

The scaling factors were determined by a micro-benchmark that measured the time for the execution of a sample resultant computation with the Maple function `resultant` on each machine in use:

- PIII@450 MHz: 1.0
- Octane: 0.60

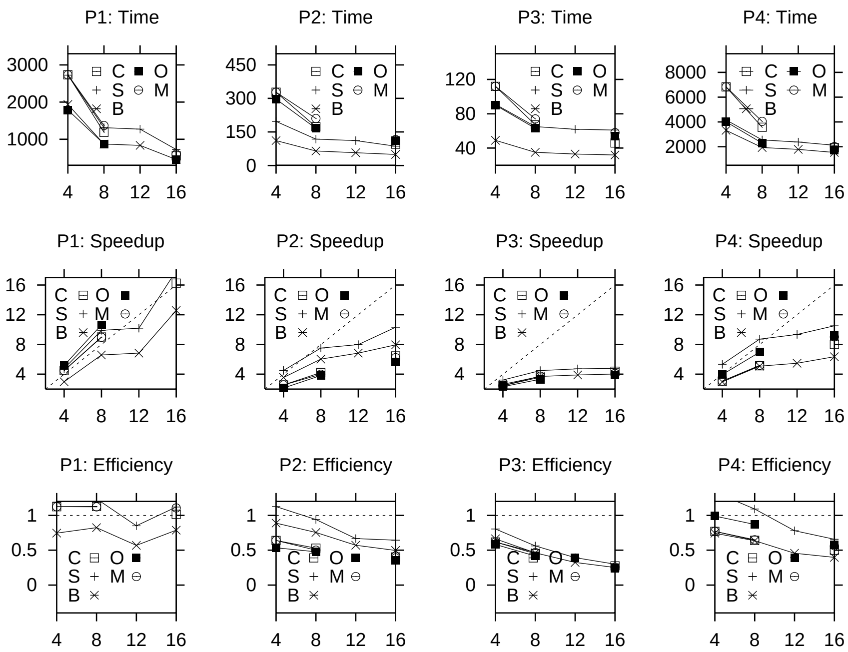

Figure 4: Plotting Algebraic Plane Curves

- Sun HPC: 0.73
- Beowulf: 1.23 (frontend node), 1.44 (compute node).

In other words, the SGI Octane with a R10000@250MHz processor has 60% of the performance of a Linux PC with a PIII@450 processor, i.e., resultant computation on the PC takes only 60% of the time required on an Octane.

The scaled speedups and efficiencies show that for all three examples Sun and Beowulf behave virtually identical with almost optimal performance in the third example. The cluster behaves in the first two examples (especially in the very short second example) significantly worse, reflecting the poor communication medium with relative high latency (shared Ethernet) that was available at our institute when the benchmark was performed.

In absolute times, the Beowulf cluster computes the results always fastest.

4.2 Plotting Algebraic Plane Curves

This benchmark demonstrates the parallelization of an application of the library CASA (Computer Algebra System for Algebraic Geometry) developed by researchers of our institute on the basis of Maple. We have designed and implemented a parallel version of the Maple function `pacPlot` for plotting algebraic curves in the plane [5, 1]; this version uses several parallel subalgorithms (resultant computation, real root isolation, coordinate matching, interval refinement) that are applied in various phases of the computation. The benchmarks were performed in the following environments:

- C: our heterogeneous network of 4 SGI Octanes (2 R10000@250MHz each) and 8 Linux PCs (various Pentium processors) linked by switched 100Mbit Ethernet lines;
- O: an SGI Origin 2000 multiprocessor (64 R10000@300Mhz processors),
- M: a mixed configuration of our 4 SGI Octanes and 8 of the Origin processors (linked to our network via an ATM line),
- S: the Sun HPC 6500,
- B: the Beowulf.

The results are depicted in Figure 4; we used the following performance factors for determining the scaled speedups and efficiencies:

- PIII@450 MHz: 1.0
- Octane: 0.56
- Origin: 0.74
- Sun HPC: 0.53
- Beowulf: 1.14 (frontend node), 1.21 (compute node).

The various parallel subalgorithms actually behave very differently (with different multiplicative factors) such that this estimation is only very coarse; indeed they may overrate the performance of the Beowulf backend nodes for this computation which results in the overall worse scaled speedup for the Beowulf system (which always gives the lowest absolute times); the overall comparison is therefore not necessarily conclusive.

In absolute times, the Beowulf cluster computes the result generally fastest, while the cluster performs comparatively very poor. Since the major difference between both environments is the inclusion of SGI Octanes in the cluster (both environments use switched 100Mbit Ethernet), the large roundtrip times of the SGI architectures measured in Section 3.3 seem to play an important role.

4.3 Plotting Algebraic Space Curves

This benchmark demonstrates the parallelization of the CASA function `ssiPlot` for plotting intersections of algebraic surfaces, i.e., curves in Euclidean space [6]. While the algorithm uses various subalgorithms, the most important step is a process similar to Gaussian elimination in a matrix of polynomials; after every elimination step, the results (rational functions) are simplified in parallel. The benchmarks were performed in the following environments:

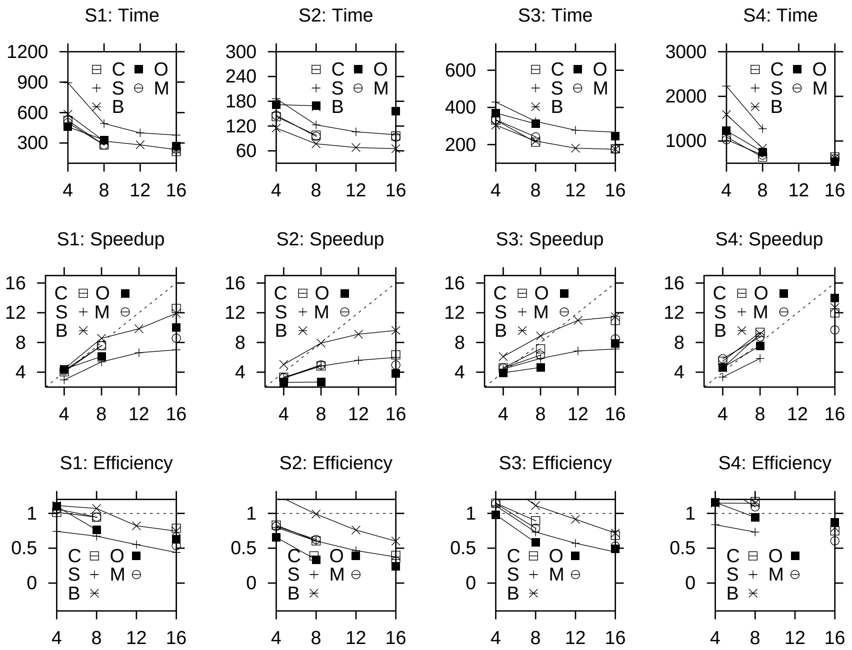

Figure 5: Plotting Algebraic Space Curves

- C: our heterogeneous network of 4 SGI Octanes (2 R10000@250MHz each) and 8 Linux PCs (various Pentium processors) linked by switched 100Mbit Ethernet lines;
- O: an SGI Origin 2000 multiprocessor (64 R10000@300Mhz processors),
- M: a mixed configuration of our 4 SGI Octanes and 8 of the Origin processors (linked to our network via an ATM line),
- S: the Sun HPC 6500,
- B: the Beowulf.

The results are depicted in Figure 5; we used the following performance factors for determining the scaled speedups and efficiencies (as determined from the sequential computation of `ssiPlot`):

- PIII@450 MHz: 1.0
- Octane: 0.56
- Origin: 1.05
- Sun HPC: 0.80
- Beowulf: 1.05 (frontend node), 0.74 (compute node).

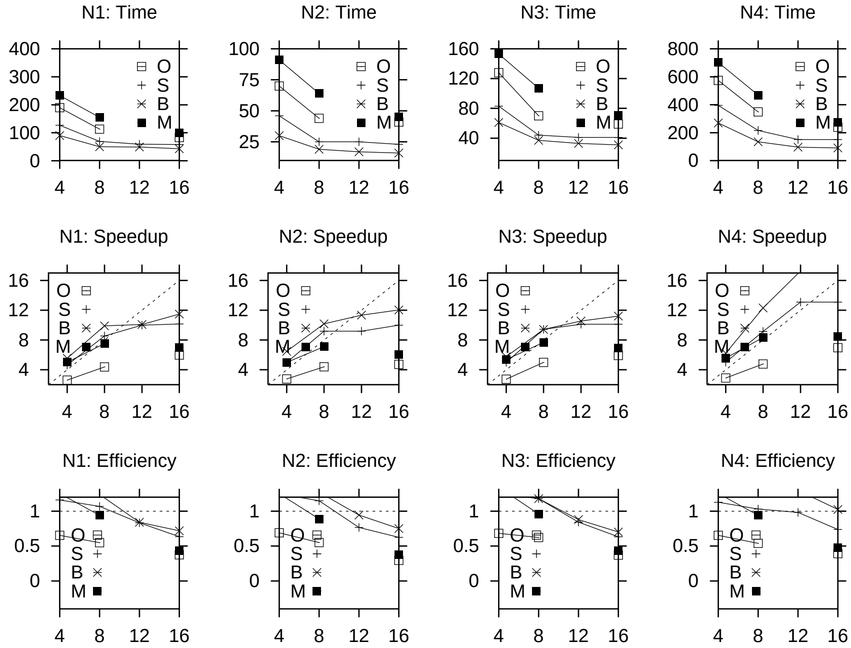

Figure 6: Neighborhood Graph Computation

These figures differ significantly from the previous figures, which shows the sensitivity of the processor speed to the kind of computation performed; in particular, the Origin processors behave much better and the Beowulf compute nodes processors behaves worse than previously measured, which affects the scaled speedups accordingly.

In absolute times, the Sun HPC machine gives generally larger execution times than all the other environments (whose results are very close).

4.4 Neighborhood Graph Computation

This benchmark demonstrates a parallel version of CASA's `neighbGraph` function for computing the neighborhood graph of an algebraic curve [4]. The most time critical part of the computation (and the only parallelized part of it) is the computation of a matrix of partial derivatives in a data-flow like manner starting from a single bivariate polynomial in the left upper corner from which derivatives in both variables are systematically constructed. The benchmarks were performed in the following environments:

- O: an SGI Origin 2000 multiprocessor (64 R10000@300Mhz processors),
- M: a mixed configuration of our 4 SGI Octanes (linked via 10Mbit shared Ethernet segments) and 8 of the Origin processors (linked to our network via an ATM line),

- S: the Sun HPC 6500,
- B: the Beowulf.

The results are depicted in Figure 6; we used the following performance factors for determining the scaled speedups and efficiencies:

- PIII@450 MHz: 1.0
- Octane: 0.51
- Origin: 1.20
- Sun HPC: 0.88
- Beowulf: 1.14 (frontend node), 1.21 (compute node).

The figures were derived from sequential executions of `neighbGraph` on the respective machines *except* for the Beowulf system where we used the figures derived from comparison the computation time of a sample `gcd` computation. With the original figure, the scaling gave extremely superlinear speedups for this machine (while using the `gcd` as a comparison base for the other machines gave superlinear speedups there). This shows that different parts of the computation have vastly different instruction mixes and that correspondingly the comparison of scaled speedups and efficiencies should be taken *cum grano salis*.

In absolute times, the Beowulf system performs best followed by the Sun HPC; the mixed configuration performs poor but also the performance of the Origin is weak. This seems to be a consequence of the large message roundtrip times discussed in Section 3.3 which plays an important role in this algorithm where a central task schedules the execution of a set of other tasks based on the results of previously terminated tasks.

5 Conclusions

We have learned that the performance of Distributed Maple is essentially determined by two parameters:

1. The *remote roundtrip time*, i.e., the time required for a dummy message to cycle via sockets between two Java processes. This time ranges from mediocre 31 ms (networked Linux PC) up to unbelievingly 300 ms (SGI Octane) and thus determines how quick a remote query can be answered.

   The roundtrip time is one to two orders of magnitude larger than the actual *latency*, i.e., the minimum time distance between subsequent elements of a stream of messages (e.g., if the `dist.systems` parameter `maxvalue` is set to a large value and a large number of tasks is created). This shows that the communication path is either heavily pipelined, or more likely, that the overhead occurs only when a thread is blocked on an empty socket.

2. The *pipe bandwidth*, i.e., the amount of data that can be communicated per time unit between Maple and the Java process. This rate is on all architectures much smaller than the *socket bandwidth* for communication between remote Java processes and is thus the bottleneck for the exchange of objects between Maple processes.

The ultimate reason for the first issue with respect to SGI systems is unclear; it can be handled only by a more efficient implementation of the socket communication mechanism in the Java Virtual Machine. It might be helpful to experiment with alternative (open source) implementations of the JVM to achieve better roundtrip times.

The second issue shows that for the transfer of large objects between Maple kernels the speed of the processor (which determines the pipe bandwidth) is the limiting factor rather than the capacity of the network. We could improve the situation by letting the Maple process communicate with the Java process via shared memory rather than sockets: Maple could generate in a shared memory region the external representation of an object that could be directly accessed by the Java process (thus avoiding one copy operation). On the Maple side, this has become technically feasible with the recent Maple 6 version which provides dynamic linking facilities to extend the Maple kernel by customized code; from the Java side, the Java Native Interface provides similar facilities to extend the Java Virtual Machine. Linking both processes with shared memory communication facilities (e.g. based on System V IPC) would thus give the potential for improving the bandwidth at the price of giving up the platform independence of the Distributed Maple environment.

As for the analysis of the parallel architectures, it has become clear that it is extremely difficult to accurately measure the *relative performance* of different architectures for a computer algebra application, since this performance is very sensitive to the various kinds of computations performed within the algorithm such that a single "performance factor" can be quite misleading. Even among processors with the same instruction set, differences can be very large (e.g. the AMD K7 processor is on some computations faster than the PIII but on some also much worse).

As for *absolute performance*, the benchmarks have shown that so called "Beowulf Systems" (clusters of Linux PCs typically linked by switched 100Mbit Ethernet networks) are as good (and in many cases significantly better) than dedicated multiprocessor systems. Since many modern institute networks belong to the same category (as e.g. the network of our institute), parallel processing with commodity hardware resources is a very attractive option for parallel computer algebra with Distributed Maple.

On the other hand, the performance of the Silicon Graphics systems (dual-processor Octanes and the Origin 2000) is comparatively disappointing. Not only that *for the purpose of computer algebra* their processors are generally slow (however with exceptions, as discussed above) but also their communication performance *in Distributed Maple* is weak. Mixing SGI systems with Linux PC systems in a cluster thus overshadows the good performance of the Linux PCs and thus seems not be a good idea (most of our previous publications used such combinations). Only the standalone Origin system may give good results in some situations (but also very bad results in others).

Distributed Maple does (without modifications, as discussed above) benefit only little from the shared memory of a the Sun HPC multiprocessor. While bandwidth on such systems is high and key timings are very good, some timings (roundtrip times) are worse than for the Beowulf cluster. Moreover, compared to the usually a bit outdated processors of such a system, the widely available processors of up-to-date PCs are better suitable to quickly transfer data into or out of a Maple kernel.

In a nutshell, Linux PCs linked by switched fast Ethernet provide currently the best platform for Distributed Maple.

References

[1] Christian Mittermaier, Wolfgang Schreiner, and Franz Winkler. A Parallel Symbolic-Numerical Approach to Algebraic Curve Plotting. In Vladimir Gerdt and Ernst W. Mayr, editors, *CASC-2000, Third International Workshop on Computer Algebra in Scientific Computing*, Samarkand, Uzbekistan, October 5–9, 2000. Springer, Berlin.

[2] Wolfgang Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998.

[3] Wolfgang Schreiner. Developing a Distributed System for Algebraic Geometry. In Barry H.V. Topping, editor, *EURO-CM-PAR'99 Third Euroconference on Parallel and Distributed Co mputing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press, Edinburgh.

[4] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Analyzing Algebraic Curves by Cluster Computing. In Peter Kacsuk and Gabriele Kotsis, editors, *Distributed and Parallel Systems – From Instruction Parallelism to Cluster Computing, Proceedings of DAPSYS'2000, 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 175–184, Balatonfred, Lake Balaton, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.

[5] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismller, editors, *Euro-Par 2000, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.

[6] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Plotting Algebraic Space Curves by Cluster Computing. In Xiao-Shan Gao and Dongming Wang, editors, *ASCM'2000, 4th Asian Symposium on Computer Mathematics*, Chiang Mai, Thailand, December 17-21, 2000. World Scientific Publishers, Singapore/River Edge.