

An Object-Oriented Platform for Distributed High-Performance Symbolic Computation

Wolfgang Blochinger Wolfgang Kuchlin Christoph Ludwig
Andreas Weber¹

*Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
<http://www-sr.informatik.uni-tuebingen.de>*

Abstract

We describe the *Distributed Object-Oriented Threads System* (DOTS), a programming environment designed to support object-oriented *fork/join* parallel programming in a heterogeneous distributed environment. A mixed network of Windows NT PC's and UNIX workstations is transformed by DOTS into a homogeneous pool of anonymous compute servers forming together a multicomputer. DOTS is a complete redesign of the *Distributed Threads System* (DTS) using the object-oriented paradigm both in its internal implementation and in the programming paradigm it supports. It has been used for the parallelization of applications in the field of computer algebra and in the field of computer graphics. We also give a brief account of applications in the domain of symbolic computation that were developed using DTS.

Key words: distributed threads system, heterogeneous networks, Windows NT cluster, symbolic computation, elliptic curves method

1 Introduction

The Distributed Object-Oriented Threads System (DOTS) is a programming environment for the parallelization of irregular and highly data-dependent algorithms. It extends support for *fork/join* parallel programming from shared memory threads to a distributed memory environment. DOTS works on the

¹ Supported by *Deutsche Forschungsgemeinschaft* under grants Ku 966/4-1 and Ku 966/6-1.

principle that a forked thread can be executed on a remote machine, provided its input and output parameters can be copied over the network and it does not engage in shared memory communication. This programming paradigm can be characterized as SPMD (*single program, multiple data*) with asynchronous remote procedure calls.

In DOTS, each network node is a multi-threaded shared-memory multiprocessor (possibly consisting of a single processor). Under DOTS the network again forms a symmetric multiprocessor. Together they form a hierarchical multiprocessor. The idea is to support a uniform and well structured parallelization paradigm on this machine. Large grained threads are to be forked over the network, small grained ones over the processors in a shared memory multiprocessor. Since each network node is to be oversaturated with threads, communication latency is hidden implicitly by running another computation thread in parallel.

The primary design goals of DOTS were on the one hand to facilitate the transformation of existing sequential C++ Code into a distributed application, and on the other hand to provide a flexible and handy tool for the rapid prototyping of algorithm layouts and for the support of highly irregular symbolic computations, especially in data-dependent divide-and-conquer algorithms. DOTS is designed to be used where elaborate low-level code optimizations for parallelization are impractical, e.g. because the control flow is data-dependent, the algorithm is too complex, or the application programmer wants to focus on high-level algorithm or application development.

DOTS uses an object-oriented approach both in its internal implementation and in the programming paradigm it supports. It provides a convenient way for the object serialization of the parameter and result objects that are involved in the distributed part of the computation.

DOTS has adopted many of the basic concepts introduced by its predecessor system DTS [2]. The primary motivation for the development of DTS was to extend the target platforms of the PARSAC-2 library [17,18] from shared memory multiprocessors to a network of shared memory multiprocessors. Besides being suitable for the realization of this initial goal, it turned out that the distributed fork/join paradigm provided by DTS could successfully be used for parallelizing many other applications.

This article will mainly focus on the more recent DOTS system. Section 2 gives a general overview of the software development process using DOTS. In Section 3, we will deal with the design and implementation of the system. Applications and performance measurements will be presented in Section 4. In Section 5, DTS and its applications are described in a recapitulating fashion. We conclude with a comparison with related work in Section 6.

2 Designing Distributed Applications with DOTS

2.1 Overview of DOTS

DOTS consists of two main components: The DOTS run-time class library and the so called DOTS Commander.

The DOTS Run-Time Class Library is a C++ class library that contains classes and functions which are necessary for coding a DOTS application. This library is to be linked with the user program.

The DOTS Commander is a program which controls the distributed computation. It starts and terminates all processes on the different computers that take part in the distributed computation. It also provides the user interface.

In order to use DOTS for a distributed computation the following main steps have to be carried out.

- Code parts suitable for distributed computations have to be identified and have to be rewritten as static functions of the appropriate classes. The distributed computation of these functions has to be performed by forking and joining DOTS threads. Forking instance methods is currently not allowed. This would require a distributed shared object state model. Because of its potential great system overhead, an implementation on the same level of efficiency as the other parts of DOTS is one of the challenges for the further development of DOTS. In DOTS, object state can easily be passed as an additional argument object of the thread. This strategy also leads to a reduction of the communication overhead because unused parts of the object state can be determined and do not have to be transferred.
- Code for the serialization of parameter objects has to be provided. DOTS provides a good infrastructure to make this a straight-forward programming task. Basically, all members of the parameter classes have to be listed in a statement involving a `DOTS_Archive` class and operators for packing (`<<`) and unpacking (`>>`). Using the template and operator overloading mechanisms of C++, recursion through the member classes and picking the appropriate objects for functions to be forked and joined is done automatically by the DOTS library.

2.2 Object-Serialization

The remote execution of a DOTS thread implies the transmission of argument and result objects over a network connection. This process requires the trans-

formation of the involved objects into serial data structures before transferring them, and their reconstruction after the transmission from this serial representation. The system class `DOTS_Archive` implements a suitable serial data structure and contains predefined operators for packing (`<<`) and unpacking (`>>`) all simple C++ data types into and from this structure. The class `DOTS_Archive` supports computations in heterogeneous networks by automatically converting the data into a standard representation within the serialization process (e. g., homogenizing big-endian and little-endian architectures).

DOTS offers two different ways for supplying code for (de-)serialization using the `DOTS_Archive` class:

Implicit Object-Serialization. This technique is based on defining a (de-)serialization operator for each argument and result class. Consider the example given in Fig. 1. The function `dots_func` will be forked over the network. The main advantage of this technique lies in reusing code in nested classes which leads to compact and clearly structured programs.

Explicit Object-Serialization. This method is based on redefining the system's standard methods for (de-)serialization with user supplied code. The main reason for us to provide this option is to facilitate the integration of legacy marshalling code written in C in DOTS. Since for new applications the implicit serialization is the method of choice, we will not give further details here.

2.3 Forking DOTS Threads

Forking a function over the network can be done via the DOTS library function `dots_fork`. This function takes three parameters: the function to be forked, its argument object, and the *thread group* to which the forked thread is assigned. We also provide a variant of the fork function, called `dots_lfork`. This variant takes the same parameters but allows a more efficient execution of the generated DOTS thread: the argument object will not be serialized immediately. It might happen that the DOTS thread can be executed locally; in this case no serialization and deserialization is necessary, resulting in improved performance compared to the `dots_fork` function, which will always serialize and deserialize its argument object. However, the argument object of a `dots_lfork` must not be changed until a corresponding `dots_join` is called. Thus using `dots_lfork` requires more care on the part of the programmer than using `dots_fork`. The differences in the internal execution between these variants are explained in more detail in Sec. 3.3.

```

class X {
public:
    int a;
    int b;

    /* The operators << and >> have to be defined for X.
     * The corresponding ones for the simple datatypes
     * can be used within the definition.
     */
    friend DOTS_Archive& operator<<(DOTS_Archive& arch, X& x)
        {return arch << x.a << x.b;}
    friend DOTS_Archive& operator>>(DOTS_Archive& arch, X& x)
        {return arch >> x.a >> x.b;}
};

class Y {
public:
    int c;
    X x;

    /* The operators << and >> have to be defined for Y.
     * Notice that now the ones defined for X can be used
     * in the definition as can the corresponding ones
     * for the simple datatypes.
     */
    friend DOTS_Archive& operator<<(DOTS_Archive& arch, Y& y)
        {return arch << y.c << y.x;}
    friend DOTS_Archive& operator>>(DOTS_Archive& arch, Y& y)
        {return arch >> y.c >> y.x;}
};

Y* dots_func(X* arg);

```

Fig. 1. Implicit Object-Serialization

2.4 *Joining DOTS Threads*

The functional result of a DOTS thread can be obtained via a call to `dots_join`. This function takes a thread group as input parameter and returns the result of an arbitrary thread in the group that has already terminated or of the thread that will be the first one to terminate if none has terminated so far. Moreover, a status flag indicating an error will also be returned. If no thread in the group has already terminated, `dots_join` will block until one thread in the group will be finished with its computations. Thus the semantics of `dots_join` is the one of a *join any* construct.

2.5 Cancellation of DOTS Threads

With subsequent calls to `dots_join` the results of all forked threads in a group can be retrieved. However, in many applications—e. g. in search problems—it is not necessary to obtain the results of all forked threads in the group, but one or some of the returned results are sufficient to allow the computation to continue.

In such a case all remaining threads in a thread group, which are still queued (cf. Section 3), can be terminated by a call to the `dots_cancel` function, which takes a thread group as a parameter. Thus subsequent computations do not have to compete for computational resources with others that are now known to be unnecessary.

By the combination of the *join any* semantics of `dots_join` with the functionality of `dots_cancel`, DOTS is a powerful tool for the distributed computation of irregularly structured search problems.

3 Design and Implementation of DOTS

3.1 System Overview

Fig. 2 shows the internal structure of the DOTS Commander. Its main component is the Global Manager, which—in conjunction with the Scheduler—conducts the execution process. The Logging Manager controls the distributed logging system of DOTS. DOTS follows the SPMD model, which means that all involved hosts run the same program. The task of the Session Manager is to manage the node processes running the same program on all hosts. Fig. 3 shows the components of a node process. Together with the Global Manager the Local Manager controls the execution of DOTS threads. The Logger processes the logging messages of the node process. All the described functional components of the DOTS Commander and of the node processes execute in parallel.

3.2 General Life Cycle of DOTS Threads

We first look at the general life cycle of a DOTS thread, beginning with its creation by `dots_fork` and finishing up by delivering its result through `dots_join`. Fig. 4 shows the example of three hosts, one running the DOTS Commander

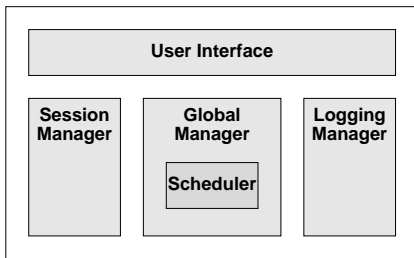


Fig. 2. The DOTS Commander

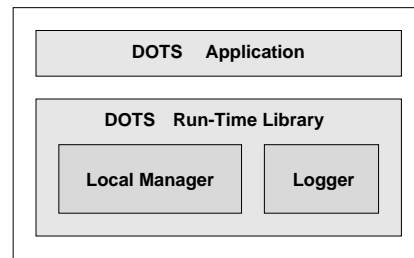


Fig. 3. A DOTS node process

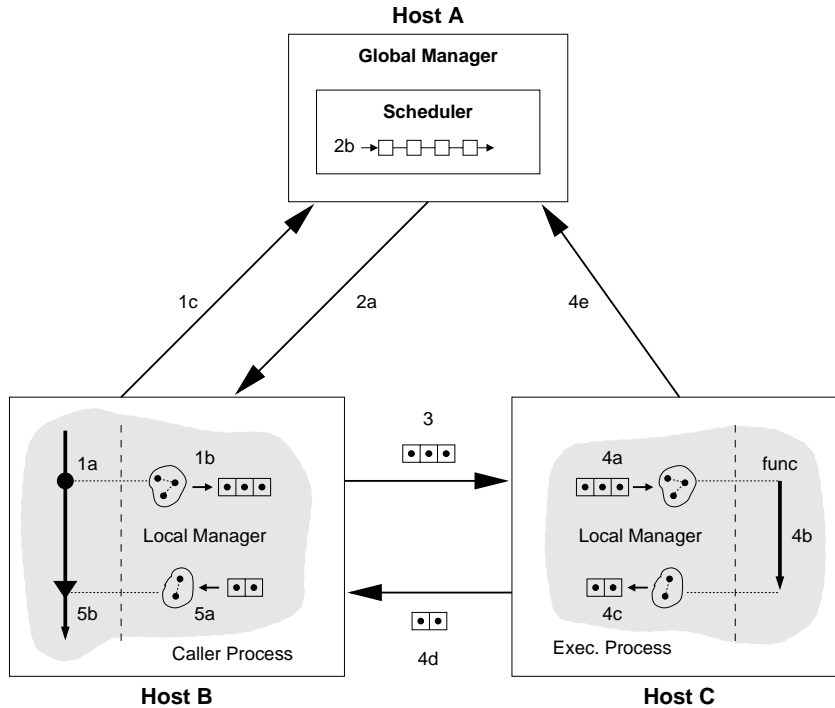


Fig. 4. Life Cycle of a DOTS Thread

and two running a node process. The execution process consists of the following five steps (cf. Fig. 4):

- Step 1:** The process begins with a call to `dots_fork` in the user program (1a). The argument object is serialized, and the corresponding `DOTS_Archive` object is stored for later use (1b). In order to register the new DOTS thread in the system, a message containing a thread information object is sent to the Global Manager (1c). In DOTS, thread information objects are used to store data associated with the execution of DOTS threads, like a thread ID or the ID of the node process chosen to execute the thread. Thread information objects are passed from one execution stage to the next and are updated during a step when necessary. When all described actions are completed, `dots_fork` returns to the user program.
- Step 2:** The Global Manager extracts the thread information object from the message and passes it to the Scheduler. The task of the Scheduler is to

determine a node process for the execution of the DOTS thread according to its scheduling strategy. The default scheduling strategy is to assign a DOTS thread to the node which currently executes the smallest number of DOTS threads. In order to prevent the hosts from being overloaded, for each host the maximal number of DOTS threads to be executed at a time can be individually determined by the user. The best bounds depend on the application as well as on the number of available CPUs and speeds of the CPUs. The scheduling strategy can easily be changed by the programmer by subclassing the scheduler interface class. If a node process is determined, its ID is stored in the thread information object, which is sent back to the caller process (2a). If no node process is currently available (because all hosts execute the maximal number of threads) the thread information object is queued (2b). In this case the execution process of the DOTS thread will be continued when a node process completes the execution of another DOTS thread (see Step 4).

Step 3: The Local Manager of the caller process sends a message containing the thread information object and the archive with the serialized argument object to the executor node process.

Step 4: The Local Manager of the executor process produces a copy of the argument object from the archive contained in the message (4a). The thread function is now executed with the argument object in a new system thread (4b). If the execution has completed, the result object is serialized (4c) and sent in a message to the caller process (4d). The Global Manager is now informed by a message that a new DOTS thread can be executed on this host (4e).

Step 5: After receiving the message from the executor process, the Local Manager of the caller process extracts the `DOTS_Archive` from the message and produces a copy of the result object (5a). Now the result is ready to be obtained via `dots_join`. If a call to `dots_join` has already occurred, the caller is now unblocked.

3.3 Optimizations of the Execution Process

In two special cases the described execution process is changed in order to minimize the communication overhead of DOTS threads. The optimizations presented below will have their maximal effect if a DOTS thread is created with `dots_fork`. In this case the serialization of the argument object is deferred to Step 3 in the normal execution process and thus can be completely eliminated in the situations described below.

3.3.1 Local execution

This optimization takes effect when the Scheduler determines that the caller process is to be identical with the executor process. In this case Steps 3 to 5 are combined into one single step, which is entirely executed in the caller process. Almost all of the communication overhead can be avoided, leading to an efficient local execution of the DOTS thread fork.

3.3.2 Execution as a virtual thread

When a call of `dots_join` appears and the affected DOTS thread is still waiting in the queue of the Scheduler for its execution, the DOTS thread is removed from the queue and executed as a local procedure call. Instead of blocking and waiting for the remote execution of the DOTS thread, the caller thread of `dots_join` is continuing its execution with a function call for the function with the argument object (Cf. Fig. 5). In [19] this has been called the *virtual threads* concept. The effect is the same as that of a real thread fork, but the efficiency is much higher in case of a loaded system. Thus the potentially unbounded logical concurrency of an application is dynamically reduced to the bounded amount of real parallelism the hardware can provide.

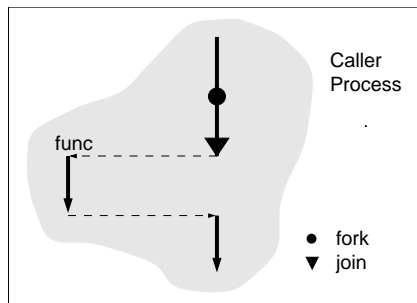


Fig. 5. Execution as Local Procedure Call

Using `dots_fork` instead of `dots_lfork` requires in both cases that the serialized argument object must first be retrieved from the stored archive. For this reason `dots_lfork` should be used when ever possible. However, the use of `dots_fork` has the advantage that the argument object is frozen at the time of the fork.

3.4 Implementation of DOTS

The implementation of DOTS is based on the *Adaptive Communication Environment* (ACE) toolkit [28]. ACE is a system-independent, object-oriented platform for developing communication software. There are implementations

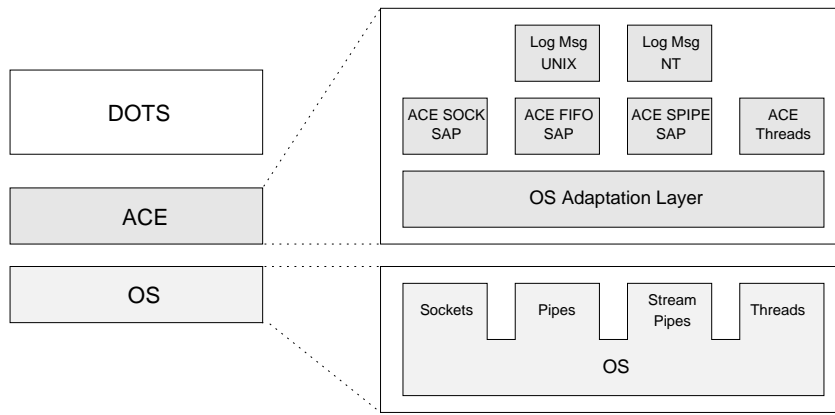


Fig. 6. Layered Structure of DOTS and ACE

of ACE for all major UNIX systems (e.g. Solaris 2.x, AIX, SGI IRIX), for Windows NT (Win32) and for MVS OpenEdition. Fig. 6 shows the overall structure of DOTS and those parts of ACE which were used for its implementation.

Using ACE as the implementation foundation for DOTS provides the following advantages:

- The platform independence of ACE makes DOTS available on a wide range of operating systems and hardware platforms. Currently we are using implementations of DOTS on Solaris 2.x, IRIX 6.x and on Windows NT. It is to be expected that DOTS can be easily ported to all other platforms that are supported by ACE.
- ACE's class encapsulation of the C based APIs of the different operating systems substantially improves the type-safety of the implementation. This can prevent the occurrence of run time errors and therefore improves the correctness and software quality.
- Performance measurements presented in [29] show that the additional abstraction layers of ACE do not cause any significant performance loss compared with using operating system calls directly.

4 Applications and Performance Measurements

This sections presents distributed applications designed using DOTS as the underlying parallelization platform. We concentrate on the description of a distributed implementation of an integer factorization application employing the elliptic curves method. Additionally, results of an application in the domain of computer graphics and results of a system performance comparison are given. A survey of DTS applications is presented in Section 5.

The problem of the factorization of a positive integer N into its prime divisors has important applications in coding theory and particularly in cryptography. Therefore a lot of algorithms approaching this problem have been developed, many of them during the last twenty years.

The Elliptic Curve Method. H. W. Lenstra's Elliptic Curve Method (ECM) [20], one of these algorithms, advances on the idea of the $p-1$ -method [24,7], but it replaces the residues mod N by points on elliptic curves mod N . The points on an elliptic curve with coordinates over a field of characteristic $p \notin \{2, 3\}$ form an Abelian group. However, the group law can also be applied to the points on a curve mod N except for the case that an inverse involved in the computation doesn't exist. But if $x + N\mathbb{Z} \in \mathbb{Z}/N\mathbb{Z}$ has no inverse, then $\gcd(x, N) \neq 1$.

So, if N is not prime and one adds arbitrary points on a curve mod N , then there is a chance that the group law, i. e. the computation of the inverse, fails and one has found an integer k such that $\gcd(k, N) \neq 1$. By the number of additions of points (on several curves) one is willing to perform, one can control the probability that the algorithm finds a nontrivial divisor of N .

LiDIA. The computational algebra library LiDIA [21] contains a sequential implementation of the ECM that uses the suggestions in [24]. This algorithm performs the group law on a number of elliptic curves, and the calculation on different curves are completely independent. Apart from the very first curves, the time needed for the computations on a single curve ranges from several seconds to a few minutes. So if one assigns to each curve its own thread, the granularity is too fine for the threads to be distributed by a simple e-mail method, but still coarse enough to hide network communication overhead. Furthermore, each thread performs the same algorithm on its set of initial values; thus the ECM fits the SPMD paradigm.

Unfortunately, there is no way of making LiDIA multi-thread safe without major modifications in the library's source code. Therefore the distributed version of the ECM had to be re-implemented, but the code follows LiDIA closely.

The Implementation. Since there were already positive experiences with the GNU MP library [15] in multi-threaded programs—one only has to apply a minor patch—the new implementation relies on this library for the involved

arithmetic. The element methods of the residue classes essentially expand to calls of the GMP functions.

The algorithm requires a table of primes that, for performance reasons, can't be copied each time a thread is forked. Thus every process has exactly one copy of the prime table that is generated during the initialization procedure. Since all threads are permitted read-only access to this static table this will cause no synchronization problems.

Finally a MT-safe random number generator had to be provided which was accomplished by a class that encapsulates the seed value and calls the MT-safe function `nrand48` of the standard C library.

After providing these MT-safe classes, it was straight forward to reproduce LiDIAs implementation of the ECM. First a sequential version ("*decs*") was created that does not link DOTS. This version was used as a reference when we measured the speedup of the distributed version ("*decp*").

The implementation proceeds as follows: After the elimination of small prime factors by trial division, a queue of jobgroups is generated, where each group represents a number of curves which are used to look for factors consisting of up to $1 + \log_{10} \sqrt{N}$ but not more than 34 decimals. Notice that not only the jobgroups are getting larger, but that also the effort per curve increases within the jobgroups. The curves are processed in a function `dec_calc`. Any divisors thus found will be added to the result vector.

Starting with *decs* the parallelization with DOTS was straightforward. Arguments and return values of `dec_calc` were encapsulated, respectively, and the call of `dec_calc` was moved to a wrapper function `dec_thread`. This function takes a pointer to an instance of the argument class and returns the address of an instance of the return value class. Thus `dec_thread` meets the signature that DOTS requires for functions to be registered as distributed functions. Furthermore, the (de-)serialize operators for both of those classes had to be provided. But that only meant to apply the respective operators to the instance elements. Those are already defined in DOTS for the elementary types, and it was implemented for the Integer class by a string representation, which can be easily serialized in turn with the tools provided by DOTS.

In *decp* the calculations on each curve are done in a separate thread: For all curves in the queue's first two jobgroups a call of `dec_thread` is forked, where `dots_lfork` was chosen for performance reasons. The order in which the results are returned does not really matter, so that by `dots_join` the return value of any curve out of the first group is checked as soon as it is available. If no nontrivial factor was found, no action is taken. Otherwise all previously forked threads are cancelled, N is reduced and all unjoined threads are re-forked. If all curves of a jobgroup have been processed and there is another jobgroup in the queue,

calls of `dec_thread` are forked again for all curves in the next group.

Performance Measurements. For the measurements of the speedup, a pool of four Sun UltraSPARC 1 machines (143 MHz, 32 MByte RAM) was used, which were connected by a 10 MBit Ethernet. On each machine we ran one process that was allowed to contain up to two DOTS-threads. The sequential version *decs* was also executed on one of those machines. In order to estimate the overhead that is introduced by DOTS itself we also measured the runtime allowing the DOTS-commander to start only one process consisting of one thread.

In each run, the programs were used to factorize

$$\begin{aligned}
 N &= 75632729957404777706513257257027317 \\
 &= 789814893838727^1 * 95760070552491115171^1,
 \end{aligned}$$

i. e. the prime factors of N have 15 and 20 decimals, respectively. The resulting

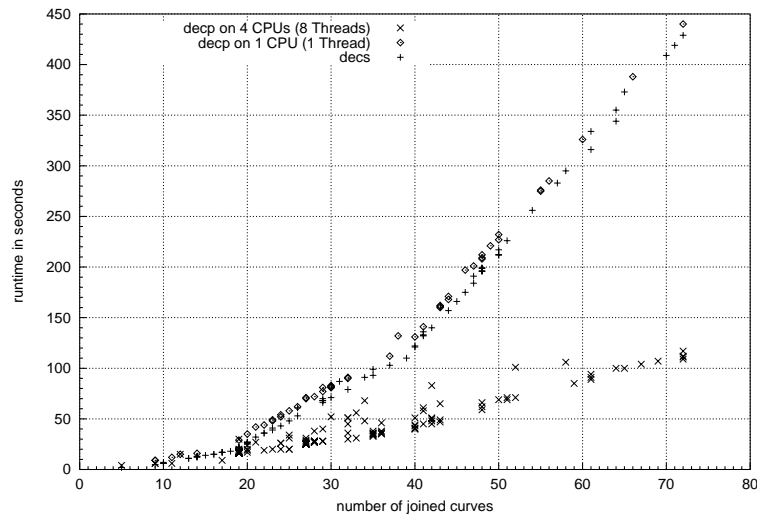


Fig. 7. The measured runtimes

125 runs of *decp* on 4 CPUs, 50 on 1 CPU and 75 runs of *decs* have been measured.

runtimes are given in figure 7, drawn over the number of curves that were joined before a factor of N was found, if any.

One sees that initially the communication overhead outweighs the fourfold computing power. But as soon as a call of `dec_calc` takes several seconds, the overhead does not dominate any longer and one gets realizable speedups. Indeed, in the case that 72 curves were joined, the average speedup was 3.82 if one used *decp* on four machines. This corresponds to an efficiency of 95.5%.

As a consequence of the measurements, it seems already attractive to distribute applications over several machines if they consist of independent calculations that take only some seconds. E. g. *decs* calculated on the average 4.4 seconds per curve if it processed 50 curves, but *decp* already came up to an efficiency of more than 75%.

4.2 Volume Rendering on a PC Network

While requiring enormous computational power, the fast rendering of 3D volumetric data becomes more and more crucial for many applications in the field of medical science and in other scientific domains. As is described in full detail in [23], DOTS was successfully used to considerably speed up a volume rendering application on inexpensive and widely used Windows NT PCs. The pool consisted of up to five Intel PentiumPro and up to four Intel Pentium II PCs running at 200 Mhz and at 300 MHz, respectively. The PCs were connected via a 10 Mbps Ethernet network. The process of casting 256^2 rays into a 128^3 data set with oversampling in ray direction could be accelerated almost linearly by the number of involved hosts. Detailed performance results are given in [23].

4.3 Comparing the Performance of DOTS on Solaris 2.x and Windows NT

For a performance comparison between DOTS running on Solaris and DOTS running on Windows NT we used a program for the distributed computation of the Mandelbrot set. The selected complex area is divided into a number of horizontal stripes, and each stripe is treated by a separate DOTS thread. Since this simple application does not depend on particular properties of one of the two different operating systems, we were able to use exactly the same source code for both platforms, making the comparison more accurate. In spite of the simplicity of the Mandelbrot application, we were able to simulate essential properties of more sophisticated distributed applications by varying the following largely independent program parameters:

- The computation time (*the granularity*) of a DOTS thread depends on the maximal number of allowed iterations per point.
- The amount of network traffic per DOTS thread (*the network weight*) can be varied by choosing different sizes of the area to be calculated.
- The relative running time of the individual DOTS threads used in the computation *the irregularity* depends on the distribution of points of convergence in the chosen area. This distribution can vary considerably leading to non uniformly sized tasks.

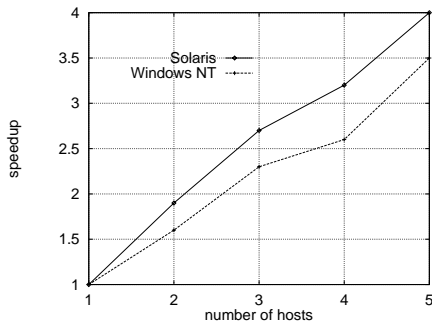


Fig. 8. Speedup

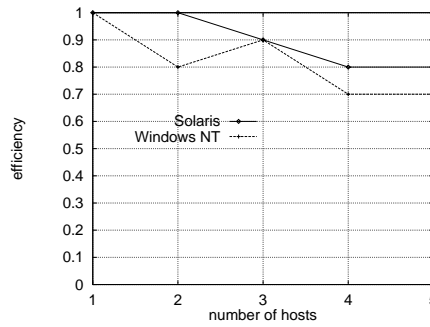


Fig. 9. Efficiency

For all subsequently presented measurements we have formed DOTS threads with a granularity of up to 5 seconds and a network weight of about 8 KByte on both platforms. This setting represents a rather worst case approximation (low granularity, high weight) of the properties to be found in typical application domains of DOTS.

Performance Measurements under Solaris. The pool of hosts consisted of 3 Sun UltraSPARC 1 workstations (each running at 143 MHz with 32 MByte of main memory) and 3 four-processor Sun HyperSPARC 10 workstations (running at 90 Mhz with 160–512 MByte main memory). All hosts were connected by a 10 Mbps Ethernet network. One host was used to execute the DOTS Commander, up to five other hosts were used to execute the application program.

Performance Measurements under Windows NT. For performance measurements under Windows NT we used a pool of 6 PCs with Intel Pentium processors (150 Mhz, 64 MByte main memory) connected by a 10 Mbps Ethernet network. One host was used to execute the DOTS Commander, up to five other hosts were used to execute the application program.

Results. Fig. 8 and Fig. 9 show curves depicting the obtained speedups and the corresponding efficiency values for both system platforms.

Discussion. Although substantial speedups of the computation could be achieved with both configurations, DOTS running on a network of workstations under Solaris shows better results for speedup and efficiency compared to the pool of Windows NT PCs. One possible reason for this behavior might be the comparatively heavy design of the Windows NT system threads, which leads to longer response times of the completely multi-threaded DOTS Local Manager. Since Windows NT and PC Hardware nowadays is getting increasing

attention in computer science—not only in the domain of computer graphics—this issue will be one focus in our future research activities.

5 DTS

The Distributed Threads System DTS [2] is the ancestor of DOTS. It was implemented on top of PVM [14] and is well suited for the parallelization of C based applications. Parameter marshalling in DTS is achieved via user supplied copying functions, while in DOTS object serialization is used.

In DTS, special attention was directed to transparently integrate the S-threads system [16] for shared-memory parallel symbolic computation into a distributed environment. DTS has been used successfully for a variety of projects whose aim was to speed up symbolic computations in both Computer Algebra and Computational Logic by parallelization on a network of (possibly parallel) workstations—e. g. theorem provers [4,6], quantifier elimination packages [9], complex root finders [27], or symbolic solvers [2,25].

5.1 Parallel Resultant Computation in PARSAC-2

Given two r -variate polynomials $P, Q \in \mathbb{Z}[x_1, \dots, x_r]$, we are interested in their common roots. The *resultant* $R(P, Q)$ is a polynomial in $r - 1$ variables with the property that $P = P(x_r)$ and $Q = Q(x_r)$ have a non-constant g.c.d. (and hence common roots), if x_1, \dots, x_{r-1} are such that $R(x_1, \dots, x_{r-1})$ is zero. Resultants have important applications in Computer Algebra, such as performing exact arithmetic on algebraic numbers [22] or solving systems of equations [13].

The *modular method* [8] is a divide-and-conquer scheme which recursively maps the multivariate resultant computation to multiple resultant computations of homomorphic images, and, using the Chinese Remainder Algorithm, lifts the image resultants back up to the originally desired resultant. In the base case of univariate polynomials, the resultant may be computed using a polynomial remainder sequence. The complexity of the algorithm is exponential in the number of variables and polynomial in the degree of P and Q . This process therefore contains a large amount of parallelism at several levels of granularity.

The coefficient MHI scheme, here embodied in algorithm IPRES, reduces a resultant computation of P and Q in $\mathbb{Z}[x_1, \dots, x_r]$ to the parallel computation of k resultants of modular polynomials in $\mathbb{Z}_{p_i}[x_1, \dots, x_r]$, $1 \leq i \leq k$,

Case	Polynomial A	Polynomial B	# MPres
1	IPRAN(3,64,1/2,(8,6,4))	IPRAN(3,64,1/2,(7,5,3))	36
2	IPRAN(3,128,1/2,(5,4,3))	IPRAN(3,128,1/2,(5,4,3))	47
3	IPRAN(3,64,1/2,(5,4,3))	IPRAN(3,64,1/2,(5,4,3))	24
4	IPRAN(5,64,1/2,(3,3,2,2,1))	IPRAN(5,64,1/2,(2,2,1,1,1))	12
5	IPRAN(3,32,1/2,(5,4,3))	IPRAN(3,32,1/2,(5,4,3))	13
6	IPRAN(4,64,1/2,(3,3,3,3))	IPRAN(4,64,1/2,(2,2,2,2))	12
7	IPRAN(3,64,1/2,(3,4,5))	IPRAN(3,64,1/2,(2,4,5))	12

Table 1

Polynomials and number of MPRES calls of the IPRES test cases

where the p_i are k distinct prime numbers of close to machine word size. The homomorphisms involved here *reduce* all coefficients modulo the prime.

In the multivariate case, the variable MHI scheme, here embodied in algorithm MPRES, reduces each resultant computation of P and Q in $\mathbb{Z}_{p_i}[x_1, x_2, \dots, x_r]$ to multiple parallel recursive resultant computations of the homomorphic images of P and Q in $\mathbb{Z}_{p_i}[x_2, \dots, x_r]$. The homomorphisms involved here *substitute* variable x_1 by $d + 1$ distinct values, where d is a bound on the degree of the resultant. Each result is a polynomial $R \in \mathbb{Z}_{p_i}[x_2, \dots, x_{r-1}]$, and is itself the homomorphic image of the true resultant of P and Q .

IPRES was tested with 7 cases, varying the number of variables, the degree of the polynomials and their coefficient length. Due to that also the number of MPRES calls changes. Table 1 shows the run-times of the test cases on a cluster of SPARCstations ELC. Function IPRAN() generates an integral random polynomial. The first argument determines the number of variables, the second the coefficient length, the third the percentage of non-null coefficients, and the last argument states the maximal degrees.

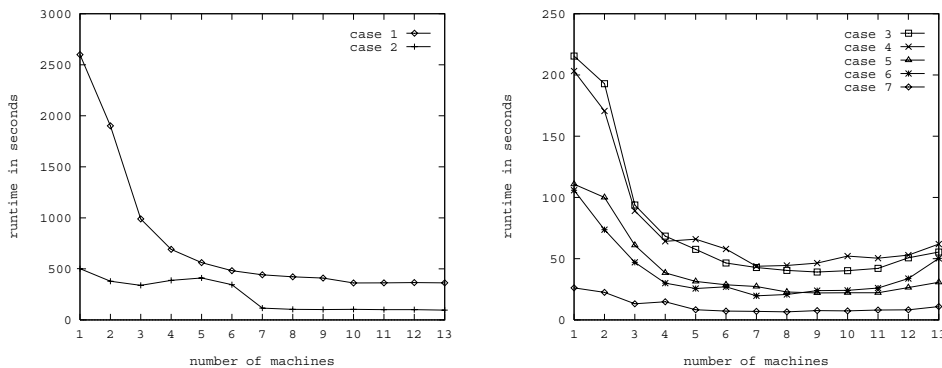


Fig. 10. Run-times of IPRES [2]

Timings of parallel symbolic algorithms are often difficult to interpret, due

to the complexity and irregularity of the computations. The above results are however fairly typical. About half a dozen to a dozen machines can be used profitably, with efficiencies above 50%, which is already a good rate for symbolic computations. Beyond 10 machines, few results are known where a sequentially efficient symbolic code achieves reasonable (say greater 40%) speedup efficiency. Note that, given an exponential algorithm, it would be easy to increase parallel grainsize and speedup efficiency by going to larger inputs, but then sequential timings quickly become untractable. The above tests try to show behavior around the point where parallelization is profitable on a typical workgroup network.

Results from parallelizing MPRES are similar [2], but the MPRES calls generated by IPRES are on a lower level of granularity. Therefore it may not be profitable to carry them over the network, if there are already enough tasks generated by IPRES. The fork-join parallelism paradigm supported by DTS and DOTS makes it easy to cope with this situation. An application programmer can convert a DOTS network fork into a thread fork on the local machine based on static load information. DOTS itself can do a similar conversion dynamically at run-time, using load information and the *virtual threads* concept. Any network thread still queued when it is joined will be converted by DOTS into a call to the corresponding function.

5.2 Parallel Term-Rewriting in PaReDuX

The integration of parallel computation systems based on the S-threads system into a distributed environment can be particularly well seen on the example of the *Distributed PaReDuX* system [4]. This system implements an equational theorem prover, based on an unfailing completion procedure, on a network of multiprocessor SPARC-stations.

The functions, which are distributed over the network via DTS, are implemented in the PaReDuX system. PaReDuX [5,3] is a multi-threaded parallel equational theorem prover based on the S-threads package. It provides several so called *strategy compliant* completion algorithms. These work on each multi-processor node. Via DTS the distributed system combines a top-level master-slave scheme distributing search parallelism over the net with the strategy compliant parallel completion scheme of PaReDuX that works on each multi-processor node.

By this combination a coarse grained search parallelism is used via DTS whereas small grained work parallelism is issued by PaReDuX on the multi-processors. In the case of unfailing completion a nice summation of the speedups of these two forms of parallelism could be obtained by the Distributed

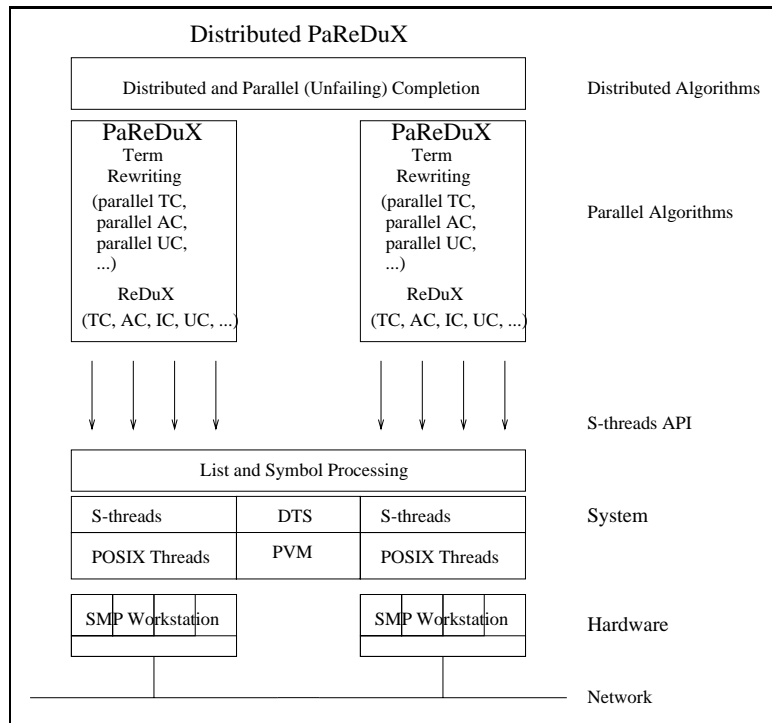


Fig. 11. The *Distributed PaReDuX* system [6].

PaReDuX system, cf. [6]. The architecture of the Distributed PaReDuX system is sketched in Fig. 11.

6 Related Work

6.1 Nexus

Nexus [10,11] provides an integration of multi-threading and communication by separating the specification of the communication's destination and the specification of the thread of control that should respond to that communication. Dynamically created global pointers are used to represent communication endpoints. Remote service requests are used for data transfer and to asynchronously issue a remote execution of specified handler functions. Thus in Nexus threads are created as a result of communication.

In contrast to DOTS, Nexus provides no explicit join construct to retrieve return values from a remote thread. Nexus is primarily designed as foundation for high level communication libraries and as target for parallel programming language compilers, whereas DOTS is intended to be used directly by the application programmer.

6.2 Cilk-NOW

Cilk is a parallel multi-threaded extension of the C language. Cilk-NOW [1] provides a runtime-system for a functional subset of Cilk that enables the user to run Cilk programs on networks of UNIX workstations.

Cilk-NOW supports multi-threading in a continuation passing style, whereas DOTS uses the fork/join paradigm for coding distributed multi-threaded programs. Cilk-NOW implements a work-stealing scheduling technique. In DOTS the Global Manager can be configured with application dependent schedulers by subclassing the system's scheduler interface.

6.3 Millipede

Millipede [12] is a system that also focuses on providing a parallelization environment on widely available hardware platforms. The essential goal of Millipede is to provide a distributed shared memory (DSM) environment for parallel programming on a cluster of Windows NT workstations.

7 Future Work

One of our future goals is to support a wider range of platforms. We are planning to provide a version of DOTS for the IBM S/390 Parallel Sysplex system [26], a clustered multiprocessor architecture. DOTS will then be available on a wide range of system platforms including low cost Windows NT PC networks and high end mainframe clusters, providing one single, easy-to-use programming paradigm for all architectures.

An other area of future research will be the parallelization of other parts of the LiDIA library. One focus will be the quadratic sieve method for integer factorization. The thread group feature of DOTS can be used to integrate both the elliptic curves and the quadratic sieve method into one single distributed factorization application.

References

- [1] BLUMOFE, R. D., AND LISIECKI, P. A. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Symposium* (January 1997).

- [2] BUBECK, T., HILLER, M., KÜCHLIN, W., AND ROSENSTIEL, W. Distributed symbolic computation with DTS. In *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR'95* (Lyon, France, Sept. 1995), A. Ferreira and J. Rolim, Eds., vol. 980 of *Lecture Notes in Computer Science*, pp. 231–248.
- [3] BÜNDGEN, R., GÖBEL, M., AND KÜCHLIN, W. Parallel ReDuX \rightarrow PaReDuX. In *Rewriting Techniques and Applications (LNCS 914)* (1995), J. Hsiang, Ed., pp. 408–413. (Proc. RTA'95, Kaiserslautern, Germany, April 1995).
- [4] BÜNDGEN, R., GÖBEL, M., AND KÜCHLIN, W. A master-slave approach to parallel term rewriting on a hierarchical multiprocessor. In *Design and Implementation of Symbolic Computation Systems — International Symposium DISCO '96* (Karlsruhe, Germany, Sept. 1996), J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 184–194.
- [5] BÜNDGEN, R., GÖBEL, M., AND KÜCHLIN, W. Strategy compliant multi-threaded term completion. *Journal of Symbolic Computation* 21, 4–6 (1996), 475–505.
- [6] BÜNDGEN, R., GÖBEL, M., KÜCHLIN, W., AND WEBER, A. Parallel term rewriting with PaReDuX. In *Systems and Implementation Techniques*, W. Bibel and P. Schmitt, Eds., vol. 2 of *Automated Deduction — A Basis for Applications*. Kluwer Academic Publishers, 1998, ch. 9, pp. 231–259.
- [7] COHEN, H. *A Course in Computational Algebraic Number Theory*, 3rd ed. No. 138 in Graduate Texts in Mathematics. Springer, Berlin; Heidelberg; New York, 1996.
- [8] COLLINS, G. E. The calculation of multivariate polynomial resultants. *J. ACM* 19 (1971), 515–532.
- [9] DOLZMANN, A., GLOOR, O., AND STURM, T. Approaches to parallel quantifier elimination. In *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation (ISSAC '98)* (Rostock, Germany, 1998), O. Gloor, Ed., The Association for Computing Machinery, ACM, pp. 88–95.
- [10] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus task-parallel runtime system. In *1st Intl. Workshop on Parallel Processing* (Tata, 1994), McGrawHill.
- [11] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing* 37 (1996), 70–82.
- [12] FRIEDMAN, R., GOLDINAN, M., ITZKOVITZ, A., AND SCHUSTER, A. Millipede: Easy parallel programming in available distributed environments. *Software: Practice and Experience* 27, 8 (August 1997), 929–965.
- [13] GEDDES, K. O., CZAPOR, S. R., AND LABAHN, G. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, MA, 1992.

- [14] GEIST, G. A., AND SUNDERAM, V. S. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference* (Portland, Oregon, May 1991), IEEE, pp. 258–261.
- [15] GRANLUND, T. The GNU multiple precision arithmetic library, edition 2.0.2. Free Software Foundation, Inc., June 1996.
- [16] KÜCHLIN, W. The S-Threads Environment for Parallel Symbolic Computation. In *Second International Workshop on Computer Algebra and Parallelism* (Ithaca, USA, May 1990), R. E. Zippel, Ed., vol. 584 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–18.
- [17] KÜCHLIN, W. W. PARSAC-2: A parallel SAC-2 based on threads. In *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 8th International Conference, AAEECC-8* (Tokyo, Japan, August 1990), S. Sakata, Ed., vol. 508 of *LNCS*, Springer-Verlag, pp. 341–353.
- [18] KÜCHLIN, W. W. PARSAC-2: Parallel computer algebra on the desk-top. In *Computer Algebra in Science and Engineering* (Bielefeld, Germany, August 1995), J. Fleischer, J. Grabmeier, F. W. Hehl, and W. Küchlin, Eds., World Scientific, pp. 24–43.
- [19] KÜCHLIN, W. W., AND WARD, J. A. Experiments with virtual C threads. In *4th. IEEE Symp. Parallel and Distributed Processing* (Arlington, TX, December 1992), IEEE Press.
- [20] LENSTRA, JR., H. W. Factoring integers with elliptic curves. *Annals of Mathematics* 126 (1987), 649–673.
- [21] LiDIA-GROUP. LiDIA – a library for computational number theory. <http://www.informatik.th-darmstadt.de/TI/LiDIA/Welcome.html>, 1997.
- [22] LOOS, R. Computing in algebraic extensions. In *Computer Algebra: Symbolic and Algebraic Computation*, 2nd ed., vol. 4 of *Computing Supplementum*. Springer Verlag, Vienna, 1982, pp. 173–187.
- [23] MEISSNER, M., HÜTTNER, T., BLOCHINGER, W., AND WEBER, A. Parallel direct volume rendering on PC networks. In *The proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)* (Las Vegas, NV, U.S.A., July 1998).
- [24] MONTGOMERY, P. L. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computations* 48, 177 (January 1987), 243–264.
- [25] NAGEL, C. Verteiltes Berechnen von Gröbner Basen mit S-Threads und DTS. Diplomarbeit, Universität Tübingen, Fakultät für Informatik, 1997.
- [26] NICK, J. M., MOORE, B. B., CHUNG, J.-Y., AND BOWEN, N. S. S/390 cluster technology: Parallel sysplex. *IBM Systems Journal* 36, 2 (1997).
- [27] SCHAEFER, M. J., AND BUBECK, T. A parallel complex zero finder. *Journal of Reliable Computing* 1, 3 (1995), 317–323.

- [28] SCHMIDT, D. C. The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software., 1993. [http://www.cs.wustl.edu/~schmidt/ ACE-papers.html](http://www.cs.wustl.edu/~schmidt/ACE-papers.html).
- [29] SCHMIDT, D. C. Object-oriented components for high-speed network programming, 1995. [http://www.cs.wustl.edu/~schmidt/ ACE-papers.html](http://www.cs.wustl.edu/~schmidt/ACE-papers.html).