In: *A. Ferreira, J. Rolim, H. Simon, S.-H. Teng (Eds.): Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98)*, volume 1457 of LNCS, pages 206–217, Berkeley, CA, USA, August 1998. Springer-Verlag.

The Distributed Object-Oriented Threads System DOTS

Wolfgang Blochinger, Wolfgang Küchlin, and Andreas Weber*

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, 72076 Tübingen, Germany
{blochinger,kuechlin,weber}@informatik.uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de/

Abstract. We describe the design and implementation of the *Distributed Object-Oriented Threads System* (DOTS). This system 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. DOTS extends the support for *fork/join* parallel programming from shared memory threads to a distributed environment. It is currently implemented on top of the *Adaptive Communication Environment (ACE)*. A heterogeneous network of Windows NT PC's and of UNIX workstations is transformed by DOTS into a homogeneous pool of anonymous compute servers. DOTS has been used recently in applications from computer graphics and computational number theory. We also discuss the performance characteristics of DOTS for a workstation cluster running under Solaris and a PC network using Windows NT, as they were obtained from a prototypical example.

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 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 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 thread.

* Supported by *Deutsche Forschungsgemeinschaft* under grant Ku 966/4-1.

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 process.

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.
- 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 execution of a DOTS thread implies the transmission of argument and result objects over a network connection. This process requires the transformation 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 (homogenizing e. g. big-endian and little-endian architectures).

DOTS offers two ways for suppling 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.

```
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

Explicit Object-Serialization. This method is based on redefinition of 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 marshaling 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 to. 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 side 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.

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. Sec. 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 and 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 are executing in parallel.

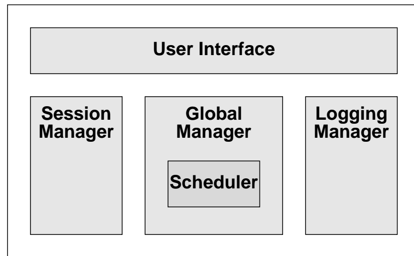

Fig. 2. The DOTS Commander

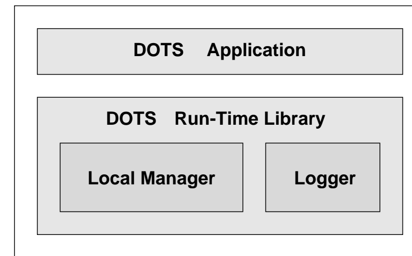

Fig. 3. A DOTS node process

3.2 General Execution Process of DOTS Threads

We first look at the general execution process for 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 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

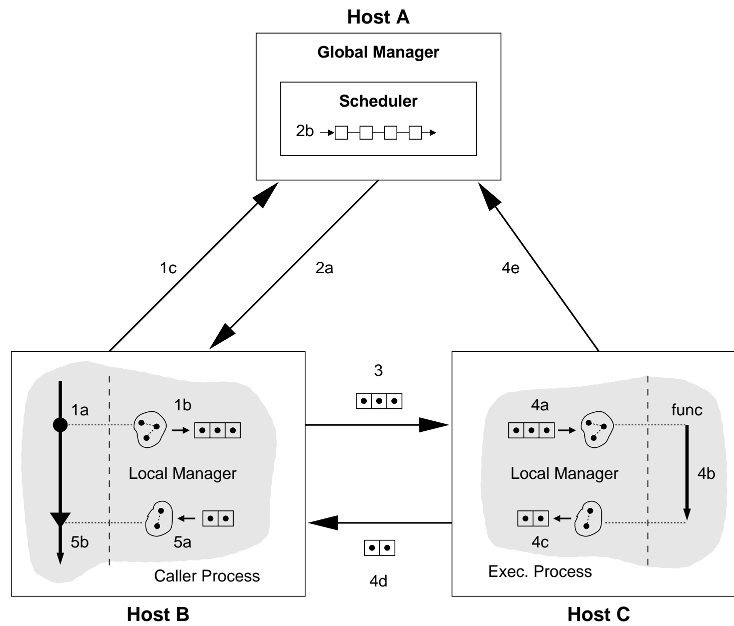

Fig. 4. General Execution Process

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 a maximal effect, if a DOTS thread is created with dots_lfork. 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.

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.

Execution as a local procedure call:
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).

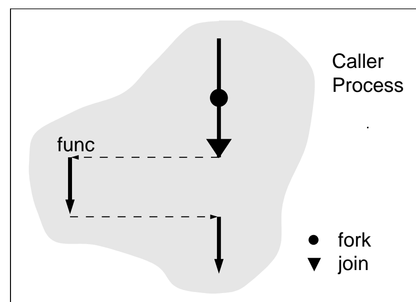

Fig. 5. Execution as Local Procedure Call

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 [12]. ACE is a system-independent, object-oriented platform for developing communication software. There are implementations 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.

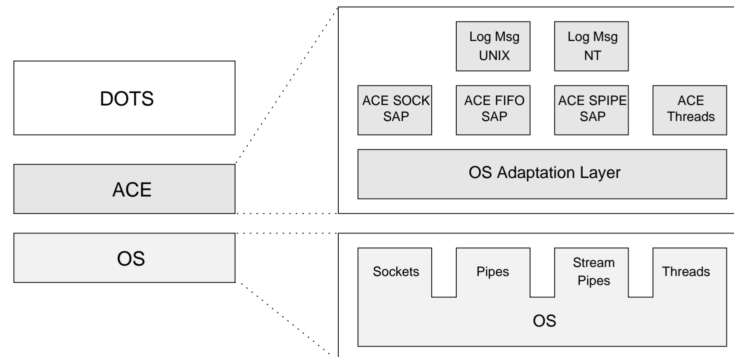

Fig. 6. Layered Structure of DOTS and ACE

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 [13] show that the additional abstraction layers of ACE do not cause any significant performance loss compared with using operating system calls directly.

4 Performance Measurements and Example Applications

For all performance measurements the wall clock time of several program runs was quantified. All performance results which are presented below are based on the arithmetic mean of the measured times. The running time of the sequential versions of the described application is denoted as time_1, while time_n is the time elapsed with $\mathrm{n} > 1$ hosts involved.

4.1 Volume Rendering on a PC Network

While requiring enormous capacities of 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 [10], 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 PentiumII 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 [10].

4.2 Factorization with the Elliptic Curve Method

The LiDIA library [9] for computational algebraic number theory provides an efficient sequential implementation of the elliptic curve factorization method [8], which was the foundation of our distributed application. This factorization method is of a too fine granularity to be distributed by a simple e-mail method as other factorization methods have been distributed, but is coarse grained enough to be suitable for parallelization via our DOTS system. Since the number of generated curves is non-deterministic, we performed a large number of program runs. For the evaluation we have chosen those program runs which used between 60 and 70 elliptic curves. This led to a speedup of 3.5 on a pool of four Sun UltraSPARC 1 workstations running at 143 Mhz with 32 MByte of main memory, connected via a 10 Mbps Ethernet network.

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

For the realization of 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.

For all subsequently presented measurements we have formed DOTS threads with a granularity of about 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. Table 1 shows the results of the performance measurements in a Solaris based network. 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. Table 2 shows the results.

n = #hosts	$\text{speedup}_n = \frac{\text{time}_1}{\text{time}_n}$	$\text{efficiency}_n = \frac{\text{speedup}_n}{n}$
1	1.0	1.0
2	1.9	1.0
3	2.7	0.9
4	3.2	0.8
5	4.0	0.8

Table 1. Performance Measurements on a Solaris Pool

n = #hosts	$\text{speedup}_n = \frac{\text{time}_1}{\text{time}_n}$	$\text{efficiency}_n = \frac{\text{speedup}_n}{n}$
1	1.0	1.0
2	1.6	0.8
3	2.3	0.9
4	2.6	0.7
5	3.5	0.7

Table 2. Performance Measurements on a Windows NT Pool

Discussion. Although with both configurations substantial speedups of the computation could be achieved, DOTS running in 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—mainly in the domain of computer graphics—this issue will be one focus in our future research activities.

5 Related Work

5.1 DTS

DOTS is a complete object-oriented redesign of the PVM [6] based Distributed Threads System (DTS) [2]. DTS has been used successfully for a variety of projects whose aim was to speed up the computation of scientific code by parallelizing it on a network of workstations—e. g. theorem provers [3], complex root finders [11], or symbolic solvers [2].

In spite of its success the DTS system has some drawbacks, which motivated the new design. The central idea of DTS—providing an extension from *fork/join* parallel programming from shared memory threads to network threads—was kept in DOTS. However, the following major advantages could be achieved by the redesign:

- The overhead of using the PVM-system could be avoided. The main problem of the use of PVM for the DTS system has been that all of the software bulk and the instabilities and non-portabilities of PVM resulted in a corresponding problem of DTS.
- The DTS system is programmed in C and supports distributed programming based on C. It gives no support for an object-oriented language. Parameter marshaling was achieved via user supplied copying functions, which were thus not encapsulated. DOTS supports the object-oriented concept of object serialization to communicate parameters. The way in which the DTS copying functions had to be provided by the user was felt to be somewhat user unfriendly by many developers outside of computer science who used DTS to distribute their scientific simulation computations. This concern led to the development of a precompiler [7] which automatically generates the copy functions out of simple pre-processor directives. However, the additional level of source code that has to be precompiled has several disadvantages, e. g., it might conflict with development tools that assume that the source code is syntactically correct C or C++ code (such as an object-oriented CASE tool). In our newly designed DOTS system, archive classes are provided for the serialization and deserialization of objects. The serialization of objects that will be distributed over the network is encapsulated in the class definition, and is thus compatible with the class abstractions provided by C++.
- In the redesigned DOTS system, also PC's running under Windows NT can be integrated into the computation environment.

5.2 Nexus

Nexus [4, 5] provides an integration of multithreading and communication by disjoining 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.

5.3 Cilk-NOW

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

Acknowledgment: We are grateful to C. Ludwig for implementing the distributed elliptic curve factorization method.

Availability of DOTS: If you are interested in a copy of DOTS write an e-mail to one of the authors.

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. A master-slave approach to parallel term rewriting on a hierachical 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.
4. FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus task-parallel runtime system. In *1st Intl. Workshop on Parallel Processing* (Tata, 1994), McGrawHill.
5. 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.
6. 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.
7. GRUNDMANN, T., BUBECK, T., AND ROSENSTIEL, W. Automatisches Generieren von DTS-Kopierfunktionen für C. Tech. Rep. 48, Universität Tübingen, SFB 382, July 1996.
8. LENSTRA, A. K., AND LENSTRA, JR, H. W. Algorithms in number theory. In *Algorithms and Complexity*, J. van Leeuwen, Ed., vol. A of *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam, 1990, chapter 12, pp. 673–715.
9. LiDIA GROUP. *LiDIA Manual Version 1.3—A library for computational number theory*. TH Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Germany, 1997. Available at `http://www.informatik.th-darmstadt.de/TI/LiDIA/Welcome.html`.
10. 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). To appear.

11. SCHAEFER, M. J., AND BUBECK, T. A parallel complex zero finder. *Journal of Reliable Computing 1*, 3 (1995), 317–323.
12. 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`.
13. SCHMIDT, D. C. Object-oriented components for high-speed network programming, 1995. `http://www.cs.wustl.edu/~schmidt/ ACE-papers.html`.