On the Coexistence of Shared-Memory and Message-Passing in the Programming of Parallel Applications

J. Cordsen[1] and W. Schröder-Preikschat[2]

[1] GMD FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany
[2] University of Potsdam, Am Neuen Palais 10, D-14469 Potsdam, Germany

Abstract. Interoperability in non-sequential applications requires communication to exchange information using either the shared-memory or message-passing paradigm. In the past, the communication paradigm in use was determined through the architecture of the underlying computing platform. Shared-memory computing systems were programmed to use shared-memory communication, whereas distributed-memory architectures were running applications communicating via message-passing. Current trends in the architecture of parallel machines are based on shared-memory *and* distributed-memory. For scalable parallel applications, in order to maintain transparency and efficiency, both communication paradigms have to coexist. Users should not be obliged to know when to use which of the two paradigms. On the other hand, the user should be able to exploit either of the paradigms directly in order to achieve the best possible solution.
The paper presents the VOTE communication support system. VOTE provides coexistent implementations of shared-memory and message-passing communication. Applications can change the communication paradigm dynamically at runtime, thus are able to employ the underlying computing system in the most convenient and application-oriented way. The presented case study and detailed performance analysis underpins the applicability of the VOTE concepts for high-performance parallel computing.

1 Introduction

Future parallel machines will exhibit a hybrid architecture, based on shared-memory *and* distributed-memory. This line is already followed with all state-of-the-art special purpose parallel computers. Similar holds for parallel machines based on workstation clusters: actual high-performance workstations are shared-memory multiprocessor systems.

Beside the difficulty developing scalable and efficient parallel programs at all, the problem of an application programmer is exploiting the various computer architectures in the most efficient way. Due to false sharing, relying on the shared-memory paradigm, as introduced by a *virtual shared-memory* (VSM) system, results in a decrease of end user performance while running in a distributed environment. Vice versa, relying on a distributed-memory paradigm, i.e. message-passing, results in a decrease of end user performance while running in a shared-memory environment. Thus, in the attempt of developing scalable and portable parallel application software, programmers are concerned with a tradeoff.

The purpose of the VOTE system is to support application programmers confronted with the tradeoff between shared-memory and message-passing communication [4]. Shared-memory communication is physically limited to operate locally only. Therefore, to implement a global shared-memory communication facility, VOTE supports a highly efficient core system. The core system implements a global address space by means of VSM. At default, a multiple reader/single writer invalidation-based sequential consistency maintenance is provided. A flexible set of functions is available allowing many powerful optimizations to programs running under control of sequential consistency [11]. As a step beyond, VOTE supports message-passing communication (MPI standard [7]) through a functional enrichment of its VSM system. Thus, VOTE provides to the application programmer architectural transparency by means of sequential consistency and a message-passing communication system.

The rest of this paper is organized as follows. Section 2 deals with the principle problem of supporting both communication paradigms. Afterwards, Section 3 covers an application case study by demonstrating the implementation of successive overrelaxation dynamically changing between the two communication paradigms. Section 4 discusses the achieved runtime results. Related works are presented in Section 5, and Section 6 concludes the paper.

2 Communication Paradigms

Todays trend in the design of scalable high-performance computing systems is based on a cooperation between software and hardware to support both an efficient shared-memory and message-passing abstraction [10]. The main problem of this cooperation is due to the incompatibility of the two views of consistency semantics as given by the two communication paradigms.

Message-passing is an explicit communication style, lacking any measures of (implicit) consistency maintenance. Global effects, in the sense of making changes to some logically shared but physically distributed state, are achieved only when the respective (parallel) application program explicitly calls communication functions. In contrast, changing portions of the logically shared but physically local state works implicit by simply reading/writing the memory cells (i.e. program variables).

The shared-memory communication paradigm is a single reader/single writer (SRSW) scheme, occasionally extended by cache-based shared-memory machines to a multiple reader/single writer scheme. Every modifying memory access works implicit. However, since for performance reasons data replication usually takes place, this paradigm requires either invalidating or updating the data copies held in the caches or the memories. These measures of consistency maintenance implicity influence the operation of all processes and generally impact the runtime behavior of a shared-memory parallel program.

A paradigm shift from shared-memory to message-passing communication is almost straightforward. It makes public (i.e. global) shared data private (i.e. local) and, thus, non-shared. The process shifting to the message-passing paradigm will be deleted from the directory information maintained by the VSM system to keep track of potential data copy holders. Due to the paradigm shift, programming complexity increases significantly. Running under control of the shared-memory paradigm, consistency was ensured by the VSM system. In the message-passing case, the programmer becomes responsible maintaining a globally consistent view of the common data sets.

A paradigm shift from message-passing to shared-memory is not that straightforward. The main problem is the unification of the local data copies into a single data image. In other words, private data copies are made public or invalidated and the directory information is updated. In most cases, this task cannot be performed automatically because of the lack of information about the order of modifications done by the individual processes. In the current implementation of VOTE, it is therefore up to the application program to announce the locality of consistent data.

3 Case Study – Successive Overrelaxation

This section presents two implementations of a successive overrelaxation algorithm based on the red & black technique [1]. This kind of application is a typical representative for parallel numerical algorithms operating on dense data structures. The amount of data being shared is little and increases only with the number of computing nodes. Data sharing is performed only by a pair of processes, that is each process communicates with two other processes, the predecessor and successor. Merely the first and last process communicate only with a single process.

The following subsections present two implementations of a SPMD-based (single program/multiple data) successive overrelaxation program. The first code communicates via shared-memory and uses a global barrier synchronization. The second code takes full advantage of the communication support system VOTE, that is it changes the communication paradigm from shared-memory to message-passing and vice versa.

3.1 Shared-Memory Communication

The code example in Fig. 1 uses shared-memory communication and is a very compact implementation of a successive overrelaxation. The red & black technique is used to avoid overwriting the old value of a matrix element before it is used for the computation of the new values of the respective neighbor. Therefore, computation is done only on every other row per iteration (line $8 + 16$). This approach requires two instead of one global barrier synchronization (lines 12

```
01: void sor (float r[M][N], float b[M][N], int f, int t) {
02:   int i, j, k;
03:
04:   while (!converged (r,b,f,t)) {
05:     for (j = f; j <= t; j++) {
06:       for (k = 0; k < N-1; k++)
07:         b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k]+r[j][k+1])*.25;
08:       if ((j += 1) > t) break;
09:       for (k = 1; k < N; k++)
10:         b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k-1]+r[j][k])*.25;
11:     }
12:     vote_barrier();
13:     for (j = f; j <= t; j++) {
14:       for (k = 1; k < N; k++)
15:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k-1]+b[j][k])*.25;
16:       if ((j += 1) > t) break;
17:       for (k = 0; k < N-1; k++)
18:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k]+b[j][k+1])*.25;
19:     }
20:     vote_barrier();
21:   }
22: }
```

Fig. 1. Shared-memory version

and 20). The algorithm continues executing the outer loop (line 4–21) until the check of convergence evaluates to true (line 4).

Variables `f` and `t` (line 1) implement a block distribution of matrix `r` and `b`. These variables are calculated on a per-process basis and are a function of the size of a matrix, the total number of processes and the logical identification of a process. The computation steps in lines 5–11 and lines 13–19 produce data used as input for the processes with adjacent values of the variables `f` and `t`. When the barrier synchronization falls, that data is consumed by the communication partner.

3.2 Message-Passing Communication

The second implementation in Fig. 2 comprises three parts. The first part (lines 4–10) implements a switch from shared-memory to message-passing communication. This requires a barrier synchronization making sure that all processes are ready for leaving consistency maintenance of VOTE. Line 5–7 compute the working set of a process using the logical identification (`MPI_Comm_rank()`) and the total number of computing processes (`MPI_Comm_size()`). Calling `vote_access()` is to ensure that the processes get access to memory regions with read/write permission. Note that overlapping memory regions are duplicated into different address spaces, with each of the duplicated regions being given a read/write permission. As a side effect, VOTE becomes unable to guarantee VSM consistency maintenance. Finally, a barrier synchronization guarantees that all processes will have received private copies of the former VSM data before the computation (i.e. second) phase is entered.

The second phase is a computation loop performing according to the successive overrelaxation scheme (lines 11–34). Computation is identical to the shared-memory version, excepted the global barrier synchronizations are replaced by sequences of MPI calls for communication. In particular, `MPI_Isend()`

```
01: void sor (float r[M][N], float b[M][N], int f, int t) {
02:   int i, j, k, Id, Procs;  MPI_Request r; MPI_Status s;
03:
04:   vote_barrier ();
05:   MPI_Comm_Rank (MPI_COMM_WORLD, &Id); MPI_Comm_Size (MPI_COMM_WORLD, &Procs);
06:   if (Id == 1) i = f; else i = f-1;
07:   if (Id == Procs) j = t; else j = t+1;
08:   vote_access (&b[i][0], &b[j][N], ReadWrite);
09:   vote_access (&r[i][0], &r[j][N], ReadWrite);
10:   vote_barrier ();
11:   while (!converged (r,b,f,t)) {
12:     for (j = f; j <= t; j++) {
13:       for (k = 0; k < N-1; k++)
14:         b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k]+r[j][k+1])*.25;
15:       if ((j += 1) > t) break;
16:       for (k = 1; k < N; k++)
17:         b[j][k] = (r[j-1][k]+r[j+1][k]+r[j][k-1]+r[j][k])*.25;
18:     }
19:     if (Id > 1) MPI_Isend (&b[f][0],N,MPI_FLOAT,Id+1,Id,MPI_COMM_WORLD,&r);
20:     if (Id < Procs) MPI_Recv (&b[t+1][0],N,MPI_FLOAT,Id+1,Id+1,MPI_COMM_WORLD,&s);
21:     if (Id < Procs) MPI_Isend (&b[t][0],N,MPI_FLOAT,Id-1,Id,MPI_COMM_WORLD,&r);
22:     if (Id > 1) MPI_Recv (&b[f-1][0],N,MPI_FLOAT,Id-1,Id-1,MPI_COMM_WORLD,&s);
23:     for (j = f; j <= t; j++) {
24:       for (k = 1; k < N; k++)
25:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k-1]+b[j][k])*.25;
26:       if ((j += 1) > t) break;
27:       for (k = 0; k < N-1; k++)
28:         r[j][k] = (b[j-1][k]+b[j+1][k]+b[j][k]+b[j][k+1])*.25;
29:     }
30:     if (Id > 1) MPI_Isend (&r[f][0],N,MPI_FLOAT,Id+1,Id,MPI_COMM_WORLD,&r);
31:     if (Id < Procs) MPI_Recv (&r[t+1][0],N,MPI_FLOAT,Id-1,Id-1,MPI_COMM_WORLD,&s);
32:     if (Id < Procs) MPI_Isend (&r[t][0],N,MPI_FLOAT,Id-1,Id,MPI_COMM_WORLD,&r);
33:     if (Id > 1) MPI_Recv (&r[f-1][0],N,MPI_FLOAT,Id+1,Id+1,MPI_COMM_WORLD,&s);
34:   }
35:   if (Id > 1) vote_inform (&b[f-1][0], &b[f-1][N], NoAccess);
36:   if (Id > 1) vote_inform (&r[f-1][0], &r[f-1][N], NoAccess);
37:   if (Id < Procs) vote_inform (&b[t+1][0], &b[t+1][N], NoAccess);
38:   if (Id < Procs) vote_inform (&r[t+1][0], &r[t+1][N], NoAccess);
39:   vote_barrier ();
40: }
```

Fig. 2. Message-passing version including switches

and `MPI_Recv()` are used to exchange the relevant data between the communicating processes. Sending of data using `MPI_Isend()` is non-blocking, while `MPI_Recv` blocks the caller until the data has been received. This procedure, the pair of a non-blocking send and blocking receive, guarantees synchronization proper for ensuring a pairwise ordering of memory operations of the processes producing and consuming the same data regions. Because, compared to the shared-memory solution, explicit global (barrier) synchronization is no longer required, the message-passing solution shows up with increased concurrency.

The final phase switches back to sequential consistency maintenance provided by the VSM system. The block distribution of the shared data is re-established. This is done by removing the overlapping memory regions (line 35–39). These calls instruct VOTE to update its directory information, establishing a consistent mapping of the ownerships of memory regions. Since shared-memory processing is allowed to start only when the directory information has been updated entirely, global synchronization becomes necessary. Therefore, the final barrier synchronization is placed in line 39.

4 Performance Results

The discussion of the results obtained is somewhat difficult, since the performance of the program based on the shared-memory communication of VOTE could not be compared directly with the results of other VSM systems. Only very few VSM systems are available and their performance results are not competitive to VOTE, because they all run on top of fairly heavyweight microkernel-based operating systems. In contrast, VOTE is implemented as part of the parallel PEACE operating system family [14] and runs on the parallel MANNA computing system [2].

System	Platform	CPU	Network	Time (ms)
Mether	SunOS4.0	25 MHz MC68020	1.2 MB/s	70-100
Munin	V	25 MHz MC68020	1.2 MB/s	13-31
Myoan	OSF/1	50 MHz i860XP	200 MB/s	4.068
VOTE	PEACE	50 MHz i860XP	47.68 MB/s	0.667

Table 1. Comparison of access fault handling times

Table 1 summarizes the performance of handling a read access fault in various systems. Mether and Munin run on rather old hardware but a comparison of Myoan [3] and VOTE is fair because of the same type of CPU used in both systems. Myoan runs on the Intel Paragon machine with a network throughput which is more than four times better than the throughput of the MANNA communication network. Yet the performance of VOTE is more than six times better than handling a read access fault in Myoan, although communication and data transfer are responsible for about 80 % of the total costs.

In the following, VOTE is compared with the performance of typical message-passing systems that run on the MANNA computing system. The overheads being present in the successive overrelaxation algorithm are presented. Afterwards, the results achieved with VOTE are compared to both a synchronous and an asynchronous implementation on top of the Parix message-passing system.

4.1 VOTE, PVM and MPI

On the MANNA computing system, the quality in performance of the PVM and MPI programming systems differ quite a lot. The PVM system [13] is designed and implemented explicitly to run on top of the PEACE operating system. In contrast, the MPI package is a port of the abstract device implementation of MPI [6]. This package was easily ported, but it shows up with pure runtime results.

Fig. 3 shows the communication overheads when the successive overrelaxation runs one hundred cycles of the outer loop. The overheads are due to the

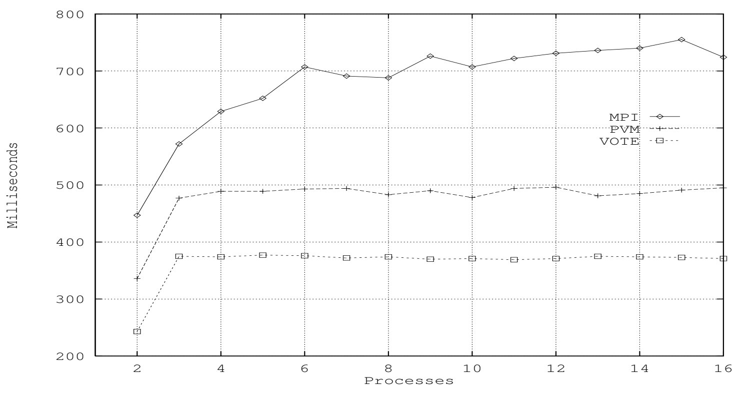

Fig. 3. Overheads in VOTE, PVM and MPI

necessary communication with all three systems using an asynchronous function to send data and a blocking function to receive data. Each loop iteration requires to both send and receive data four times. Only the two processes with the lowest and highest logical identification do half the communication.

At a first glance, it gets clear that the performance achieved through MPI is out of discussion[3]. The curves of PVM and VOTE have the same shape, with VOTE performing about one hundred milliseconds better (about 25 % percent less overhead). The explanation is quite simple. VOTE avoids the creation of copies of the data being transferred to a remote process.

Avoiding data copying becomes possible because VOTE controls both the message-passing functions and the management of memory resources. When data arrives, the status of the addressed process is checked. If the process is ready to receive, data is written directly into the provided address space. Otherwise, a copy is made and the data is intermediately buffered. In contrast, the PVM system always copies the data on both the sending and receiving site.

4.2 VOTE vs. Parix

A comparison with a PowerXplorer running the Parix operating system [12] has been chosen because Parix is a communication and thread library like PEACE. This avoids comparing apples with oranges in the case of a heavyweight microkernel approach on one hand and a high-speed and lightweight runtime executive on the other hand.

The computing systems that were used to run VOTE and Parix have different hardware and, thus, different performance capabilities. VOTE runs on the

[3] Mainly, this is because the transmission of data always involves two communications across the network. In a first message, a header is send. The second message transfers the data.

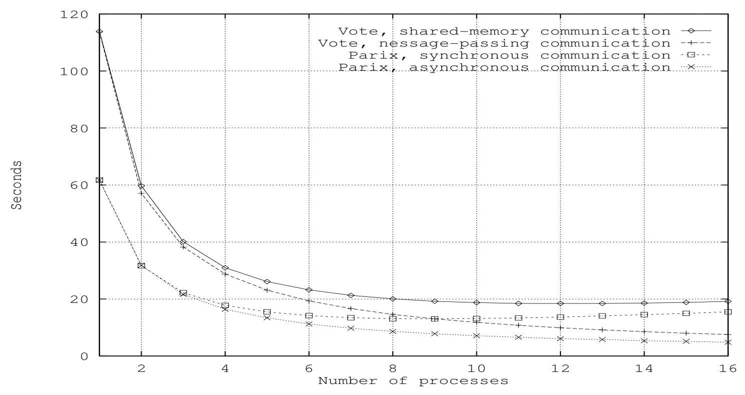

Fig. 4. Comparison of VOTE and Parix results

i860XP-based parallel MANNA machine and Parix uses the PowerXplorer based on a PowerPC-601. The performance capabilities of the computing systems show some significant differences. In the context of Parix, end-to-end communication between partners will be at a much smaller rate than in the VOTE environment. On the other hand, computation is faster compared to the results achievable on top of VOTE. Having both a faster processor[4] and a slower communication network[5] results in bad speedup results. Therefore, Fig. 4 shows runtime results and not speedup result.

Fig. 4 presents four curves, which may be grouped into two pairs. The first pair is the curve of a sequential consistent execution on top of VOTE and the runtime of the Parix implementation using a synchronous implementation of the send function. Observing the course of these two curves, the lack of scalability is already visible for very small numbers of processes. The second pair is given through the implementations that use asynchronous send functions. These curves show up with a better scalability and promise further runtime improvements for higher number of computing processes.

5 Related Work

In the sector of hardware supported distributed shared-memory (DSM) systems, Alewife [9] and Flash [10] multiprocessors are designed to support in hardware both a shared-address space and a general message-passing interface. The two communication paradigms are implemented through a controller using the same

[4] Computation with a PowerPC-601 is nearly twice as fast as with a i860XP processor. This is due to a higher clock speed and a better compiler.

[5] Communication with a message size of 4 KB data is about 1 ms on top of VOTE and 2.5 ms on top of Parix.

FIFO pipelines to pass messages as well as memory operations. This implements a global ordering of events and guarantees the same memory consistency semantic for both communication paradigms. The communication interface of Alewife provides sequential consistency, and that of Flash provides release consistency [5].

The CarlOS system [8] implements in software a message-driven release consistent global address space. As in Alewife and Flash, the two communication paradigms are operated through a single integrated solution. Like Flash, CarlOS implements the release consistency model. Therefore, the user interface requires that shared-memory accesses as well as calls to message-passing functions are attributed with the information about the type and scope of consistency required.

VOTE takes a different approach. Coexistence of communication paradigms means not to integrate both approaches into a single one. Rather, VOTE supports interfaces to the two approaches and allow to use whichever is likely to be most efficient for the communication in question. However, the point is that VOTE provides independent implementations of the two communication paradigms and, thus, is able to support their natural operation models with varying memory consistency models.

6 Conclusion

The future parallel computer architecture interconnects shared-memory multiprocessor systems on a networking (i.e. message-passing) basis. This architecture calls for two programming paradigms: shared-memory and message-passing. For scalable parallel applications, in order to maintain transparency *and* efficiency, both paradigms have to coexist. Users should not be obliged to know when to use which of the two paradigms. On the other hand, the user should not be prevented from exploiting either of the paradigms directly in order to achieve the best possible solution to his problems.

The VOTE system allows employment of both a shared and distributed memory environment in a convenient way. VOTE offers a sequential consistent VSM system with a highly efficient access fault handling scheme. A set of additional performance enhancement techniques implements many powerful optimizations to parallel applications being based on a sequential consistent shared-memory communication. Beside of the global address space and its (implicit) consistency maintenance scheme, message-passing communication functions are supported as well. A programmer is able to choose the communication paradigm best suited to match the specific application demands and to use the underlying computing system in the most convenient way.

The performance numbers presented in this paper prove a high quality standard of both the shared-memory and message-passing communication performance of VOTE. VSM systems running on comparable hardware platforms are clearly outperformed (by a factor of six) and the efficiency of message-passing packages running on the same platform run at least 25 % slower. These results indicate that the VOTE symbiosis of a transparent and efficient communication support system is a viable way to meet the programming challenges of the hybrid memory architecture of future parallel hardware platforms, i.e., networks of shared-memory multiprocessor systems.

References

1. L.M. Adams, H.F. Jordan, "Is SOR Color Blind?" In *SIAM Sci. Stat. Computation*, Vol. 7, No. 2, pp. 490–506, 1986.
2. U. Brüning, W.K. Giloi, W. Schröder-Preikschat, "Latency Hiding in Message Passing Architectures", In *Proceedings of the International Parallel Processing Symposium*, IPPS-8, pp. 704–709, Cancun, Mexico, Apr., 1994.
3. G. Cabillic, T. Priol, I. Puaut, "Myoan: An Implementation of the Koan Shared Virtual Memory on the Intel Paragon", *Technical Report*, 812, Irisa, Rennes, France, Apr., 1994.
4. J. Cordsen (ed.), "The SODA Project", *Studien der GMD*, Nr. 301, ISBN 3-88457-301-2, October, 1996.
5. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 15–26, 28–31th May, 1990.
6. W. Gropp, E. Lusk, "An Abstract Device Definition to Support the Implementation of a High–Level Point–to–Point Message–Passing Interface", *Technical Report MCS-P342-1193*, Argonne National Laboratory, Argonne, IL 60439, 1993.
7. W. Gropp, E. Lusk, A. Skjellum, "Using MPI: Portable Parallel Programming with the Message–Passing Interface", MIT Press, ISBN 0-262-57104-8, Oct., 1994.
8. P.T. Koch, R.J. Fowler, E. Jul, "Message-Driven Relaxed Consistency in a Software Distributed Shared Memory", In *Proceedings of the First USENIX Symposium on Operating Systems Designs and Implementation*, pp. 75–85, Nov., 1994.
9. D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, B.-H. Lim, "Integrating Message-Passing and Shared-Memory: Early Experience", In Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming, pp. 54–63, May, 1993.
10. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy, "The Stanford FLASH Multiprocessor", In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302–313, Apr., 1994.
11. L. Lamport, "How to make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs", In *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 241–248, Sep., 1979.
12. Parsytec, "Parix Programmer's Manual", *Parsytec Computer GmbH*, Aachen, Germany, 1991.
13. A. Geist, A. Beguelin, J.J. Dongorra, W. Jiang, R. Manchek, V.S. Sunderam, "PVM 3 User's Guide and Reference Manual", *Technical Report* ORNL/TM–12187, Oak Ridge National Laboratory, Oak Ridge, USA, May, 1993.
14. W. Schröder-Preikschat, "The Logical Design of Parallel Operating Systems", Prentice-Hall, ISBN 0-13-183369-3.

This article was processed using the LaTeX macro package with LLNCS style