A DISTRIBUTED IMPLEMENTATION OF SHARED VIRTUAL MEMORY WITH STRONG AND WEAK COHERENCE*

by

W. K. Giloi, C. Hastedt, F. Schoen, W. Schroeder-Preikschat

GMD Research Center for Innovative Computer Systems and Technology
at the Technical University of Berlin, Germany

Abstract

A *virtual shared memory architecture* (VSMA) is a distributed memory architecture that looks to the application software as if it were a shared memory system. The major problem with such a system is to maintain the coherence of the distributed data entities. *Shared virtual memory* means that the shared data entities are pages of local virtual memories with demand paging. Memory coherence may be *strong* or *weak*. *Strong coherence* is a scheme where all the shared data entities look from the outside as if they were stored in one coherent memory. This simplifies programming of a distributed memory system at the cost of a high message traffic in the system, needed to maintain the strong coherence. The efficiency of the system can be increased by adding a *weak coherence* scheme that allows for multiple writes by different threads of control into the same page. The price of the weak coherence scheme is the need for explicit program synchronizations, needed to reestablish at the end the strong coherence of the result. For the computer architect, the challenging question is how to implement a VSMA most efficiently and, specifically, by what architectural means to support the implementation. In the paper a new solution to this question is presented based upon an innovative distributed memory architecture in which communication is conducted by a dedicated communication processor in each node rather than by the node CPU. This will make the exchange of short, fixed-size messages, e.g., invalidation notices, very efficient. Therefore, it becomes more appropriate to minimize the overall administrative overhead, even at the cost of more message traffic. On that rationale, a novel, capability-based mechanism for both strong and weak coherence of shared virtual memory is presented. The weak coherence scheme is built on top of the strong coherence, utilizing its mechanisms. The proposed implementation is totally distributed and based on a strict need to know philosophy. Consequently, the elaborate pointer lists and their handling at runtime typical for other solutions is not needed.

Keywords: Distributed memory architecture, virtual shared memory architecture, strong and weak data coherence, communication hardware, parallelizing compilers

1. INTRODUCTION

Shared memory architectures offer the advantage that an application program can be automatically parallelized by the compiler, while dynamic load balancing may be done jointly by the operating system and the runtime system and supported by appropriate hardware [1]. This allows applications to be programmed as if the system were a conventional single processor architecture.

---

* This work was partly sponsored by the Ministry of Research and Technology of the German Federal Government, grant No. ITR 90022

However, large MIMD systems cannot be realized in any reasonable way other than as *distributed memory architecture*. For any sizable number of processors the access contentions to a shared memory would severely degrade the performance of the system. Moreover, only distributed memory systems are scalable. Scalability means that with the same hardware and system components one can realize the entire spectrum ranging from superworkstations to supercomputers with hundreds or thousands of nodes [2]. In a distributed memory architecture the nodes communicate through message-passing via the *interconnect*. This leads to a programming model consisting of a large number of cooperating processes that communicate through appropriate inter process communication protocols (IPC). There exist no global data objects; rather, data are encapsulated into processes. We recognize here one of the basic constituents of *object-oriented programming*.

However, conventional programming languages such as Fortran, Lisp, Ada, and C, have the view of *shared data objects* existing in a global address space and being accessible by various procedures. Computation is seen as a succession of state changes of scalar memory objects. Parallel processing is viewed in the same manner, with the proviso that now several memory states are transformed simultaneously in different *threads of control* (TOC). Consequently, parallelism is the fine grain parallelism of the primitive statements of the programming language. If one wants distributed memory architectures to become accepted by the majority of programmers, one must find ways to reconcile them with the conventional programming style.

The ideal solution would be to have a superbly intelligent compiler which performs a global data flow analysis over the entire program, parallelizes the program so that an optimal workload distribution is achieved, and puts the *send* and *receive* constructs needed for the communication among the cooperating processes and the synchronization of their activities into the right place. However, such a compiler is at least years way. For the time being it is a more practical solution to make the distributed system look to the application software as if there existed a global sharable memory. Hence, by building a *Virtual Shared Memory Architecture* (VSMA) on top of the distributed (physical) system [3], the advantages of the distributed memory architecture can be reconciled with the conventional programming style.

Independent of the question of programming style, VSMAs have an important advantage over distributed memory system: In the latter data are encapsulated into processes which, in turn, are statically distributed over the nodes of the system, whereas in the former data migrate dynamically to the site where they are needed. This advantage must be paid for by a loss in efficiency caused by the virtual shared memory overhead. Thus, it becomes a major challenge for the computer architect to make that overhead as small as possible.

Since in a VSMA copies of shared data items may be distributed over a number of nodes, the main problem is how to maintain the consistency of the shared data. This problem is analog to the well-known cache coherence problem in multi-cache architectures: The virtual shared memory takes the role of the main memory, and the local memories of the nodes take the role of the caches [3]. Several solutions to this problem have been proposed in the literature [4],[5],[6] and software solutions have been implemented in commercial multicomputer systems [7]. In this paper, we propose the novel solution of a capability-based implementation of shared virtual memory with strong data coherence. It will be shown how one can build on top of a strongly coherent shared virtual memory a *weak coherence mechanism* to improve the efficiency of the VSMA. By these mechanisms and in combination with innovative architectural support we expect to obtain a very efficient VSMA realization.

Our solution takes the *Shared Virtual Memory approach* introduced first by Kai Li [3], i.e., using as shareable entities the *pages* of the local virtual, demand-paging memories of the nodes. This approach has been much refined by adding a novel *weak coherence mechanism* in coexistence with the strong coherence scheme. This new approach has found an innovative, highly efficient implementation that is supported by appropriate architectural measures. Consequently, our scheme is expected to be orders of magnitude faster than a pure software solution [8].

2. DATA COHERENCE IN VSMAs

2.1 The Need for Synchronization

A program written for a shared memory system is not guaranteed to run correctly unless it contains synchronization constructs that enforce the appropriate order of data access as determined by the data dependencies in the program. In physically shared memory systems data access synchronization may be carried out implicitly by putting the memory accesses into the right order. However, in VSMAs with their many concurrent threads of control, which all may read and write the same data entities, it is safer not to assume a particular sequence of events but to synchronize the data accesses explicitly by *critical regions*.

That way the danger of incorrect program execution is avoided -- specifically when taking the weak coherence approach discussed below.

Locking and unlocking critical regions requires indivisible semaphor operations. Implementing a lock by agreement between N sharing nodes of a distributed system requires 2*(N-1) messages [9]. Thus, when executed by software the data access synchronization through critical regions in the VSMA program induces a considerable overhead that may readily exceed the IPC protocol overhead in distributed memory architectures. Consequently, specific architectural support should be provided to increase the efficiency of VSMAs. The most important single measure is to minimize the message passing overhead. Messages must be exchanged for two purposes: (i) to synchronize the concurrent threads of control of the VSMA program and (ii) to synchronize the data accesses. The latter is needed to obey the *data dependencies* in the program and to maintain the *data coherence* in the distributed system. The virtual shared memory is called *coherent* if the value fetched by a read operation is always the value written by the most recent write operation to the same location.

For efficiency reasons, the nodes of a VSMA operate on their own copies of a shared *primary data entity*. There exist synchronization points in the program (usually the beginning or end of critical regions) at which the consistency of the primary entity and its copies must be ensured. The thread of control (TOC) that has access to the primary entity is called its *owner*, and the TOCs that have copies of it are called *copy holders*. Data coherence may be *strong coherence* or *weak coherence* [10].

2.2 Data Coherence

First we introduce the general notion of a data entity and discuss afterwards details such as the appropriate granularity of entities and their representation.

DEF: Data entity

A data entity or, simply, *entity* is a set of data such that (i) the entire set resides in one and the same node and (ii) it forms an entity with respect to access capabilities. Entities may be private or shared. Under certain conditions several replicas of a shared entity may reside in the system, distributed over different nodes.

Strong coherence means that the coherence of the virtual shared memory is maintained at all times. In a VSMA, the node memories can be viewed as "caches" containing copies of certain data entities in the virtual shared memory. As in the case of caches, a coherence problem arises as soon as one of the copies is modified by a write access. In a VSMA, this happens whenever a TOC is granted write access to a data entity. The analogy to the cache coherence problem suggests the use of the *write-invalidate* technique usually employed to achieve cache coherence [3]. In a cache architecture, this is done by a *snoop logic* which monitors the instructions on the memory bus, detects write instructions, and consequently invalidates the cache lines involved. In a distributed memory system this must be performed by message passing.

Instead of the write-invalidate approach described above, a write-update scheme could be employed. In that case, rather than destroying all copies except for the primary entity, one only renders them temporarily inaccessible, updates them before the owner has handed the write capability for the primary entity back, and then uses them again. This approach reduces the copying overhead at the cost of increasing considerably the message passing overhead. In connection with the Shared Virtual Memory approach discussed below, the write-invalidate method is more efficient [3].

In many applications, shared data entities are the carriers of values exchanged between different nodes. This causes the need for alternating writes by different TOCs into the same entity. Strong coherence, however, does not allow for multiple writes by different sharers. This may lead to a situation where the data item must be copied back and forth among the sharers[3]. On the other hand, strong coherence among multiple reads and writes performed by different TOCs is not required if the programmer enforces the correct order of access between concurrent computations by appropriate synchronization activities (e.g., locks and unlocks). Assume that the several tasks of a parallel program are coherent within itself but not in relation to the others, unless this is enforced by explicit synchronization in the program. This situation is called *weak coherence* [10].

2.3 Using Pages as Data Entities: Shared Virtual Memory

While most data entities in a VSMA are private (owned by only one of the TOC), there exist also a number of shared entities. For each shared entity access rights must be issued for every entity. Performing this at the granularily of the single memory word would make a system overly inefficient. Therefore, one must look for a more suitable choice of data entities. Choices proposed in the literature are:

* the page of a demand-paging virtual memory [3]
* the cache line [11].

One question to be addressed is why one does not choose the objects of the high-level programming language directly as entities. This has the following reasons. Object types in von Neumann languages typically are scalars (single memory words), arrays, and structures. In our opinion, none of these types is very suitable for being introduced as an entity in the sense of the definition above. As was pointed out above, scalars are too small. Arrays, on the other hand, may easily be too large. Therefore, array operations are typically carried out on message-passing architectures by decomposing the array into subarrays and distributing the subarrays over the nodes of the system, to have them processed in parallel [12].

The solution to this problem is to base the virtual shared memory on the common virtual memory organization with demand paging [3]. In this case, the representation of data entities in the system is the *page*, i.e., a logical entity, the access right control of which can be readily integrated into the func-tionality of the memory management. The superposition of a virtual shared memory on a distributed memory system with virtual memory organization in the nodes is called *Shared Virtual Memory* [3].

The advantage of the shared virtual memory solution is that its management can be combined with the demand paging mechanism of the private node memory [4]. Copies of an entity that must be destroyed as soon as a process is granted the write capability for it can simply be tagged as invalid in their page descriptor. Any attempt to access an invalid page leads to a page fault in the same manner as if the page were not existent in the local memory. In both cases, the effect is that a new copy of that page must first be loaded into the private memory before computation can continue. This can be handled uniformly by the operating system in connection with the demand-paging virtual memory management unit (PMMU) in the node.

In contrast, the cache line is an arbitrary physical entity that does not have any meaning in the abstract machine model. We believe that one should prefer a meaningful logical entity (the page) over an arbitrary physical entity (the cache line) as the unit of access.

3. A CAPABILITY MECHANISM TO ENFORCE STRONG DATA COHERENCE

3.1 Rules for Granting Access Capabilities

3.1.1 Access Capabilities

Based upon the rationale given above, pages of the demand-paging virtual memories of the nodes of a VSMA are used as the shared data entities. In our approach, access to shared pages are controlled by granting access capabilities as defined below.

DEF: Access Capability
An access capability to a shared page is a quadruple:
(*page_id, access_right_specification,owner_id, copy_set_list_pointer*).

DEF: Access rights
An access right is either *read access* or *write access*. The read access is either *valid* or *invalid*.

DEF: Owner of a shared page with strong coherence
Each shared page has exactly one owner, which is the TOC that was the last one to have written into it. The page for which the owner holds the write capability is called a *primary page*. Replicas of the primary page for which other TOCs may have a read capability are called *copies*. The owner of a page lists in a *copy set list* all the other TOCs which have copies of that page.

3.1.2 Rules for Strong Coherence

In the following we present the two rules that guarantee that no TOC can read a page while another TOC is writing into it [3]. The rules also guarantee that a write capability cannot be taken away from a TOC while it is in a critical region.

Write Capability Grant Rule:
A write capability to a shared page can be held by only one TOC at a time, i.e., there is only one owner. On occurrence of a write fault in a TOC, the TOC will request the write capability from the current owner. As long as the owner is in a critical region, it can ignore the request; else it must honor it. Before granting the write capability, the current owner must change its capability from *write* to *read.* Subsequently, it sends the write capability together with the copy set list to the requestor. Before the new owner can exercise the write capability received, it must invalidate all other copies of that page by sending out invalidation notices to the TOCs in the copy set list.

Read Capability Grant Rule:
A read capability may be simultaneously requested from the owner by any number of TOCs. As long as the current owner is in a critical region, it can ignore the request; else it must honor it. Before sending the read capability, the current owner must change its capability from *write* to *read.* Subsequently, it sends the read capabilities together with a copy of the page to the requestor.

4. SHARED VIRTUAL MEMORY IMPLEMENTATION

4.1 The Linked List Solution

A solution proposed for the implementation of a VSMA is to have in each node an operating system server called *page manager.* The page manager administers the pages of the node and their access rights and requests access capabilities to additional pages in the case of page faults. Therefore, the page manager operates on a data structure called *page table* which has an entry for each sharable page containing [3]:

* the *owner field,* indicating the node that owns the page (executing the TOC that has most recently written to it);
* the *copy set* field, listing all nodes that have copies of the page;
* the *access right* field, defining the access right the node has to the page;
* the *lock field* used for synchronization of access to the page.

Changes of ownership of a page occurring during program execution can be handled by the following mechanism [3]. If the page addressed as the owner in a page fault request is not the owner any more, it forwards the request to the node designated by its owner field. That way, the current owner will eventually be reached.

Another approach is to have in each node for every shared entity a doubly linked list connecting the owner with the copyholders [5],[7]. That way the owner knows where to send invalidation messages, and the copyholders know where to get an updated page from after an invalidation. In case of a change of ownership, the linked list is extended accordingly at the site of the old owner; therefore, the copy-holders need not be notified about the change [3].

These list structures are very elaborate. If all sharable pages can be shared by all existing TOCs, than a copy of the page table for all sharable pages must exist in every node. Fortunately, this is not the normal case; typically, each TOC shares only a certain subset of the set of all sharable pages. In this case the page table in each node must contain only entries for the subset of pages shared by that node. The typical way to implement such a table is by hash coding.

The page table solution minimizes the amount of messages to be exchanged at the cost of having to store large lists in the node memory and burdening the node CPU with managing them. Nevertheless, it may be the appropriate solution if the startup time penalty for a message exchange is high, i.e., in the order of magnitude of several hundred microseconds typical for a fast software solution [13].

4.2 Innovative Architectural Support for the VSMA

In [14] we have shown that the communication startup time problem of message-passing systems can be greatly reduced by equipping each node with a dedicated *communication processor* (CP). This holds true specifically in the case that the node operating system must support multitasking. Multitasking, however, is a requirement for making a distributed memory system scalable and reconfigurable. In VSMAs there is another strong reason for multitasking: Since one cannot afford to let the system idle whenever a page fault occurs, there must be other tasks ready to be activated.

The communication between the CPU and the CP of the node takes place via *send* and *receive queues* as illustrated in Figure 1. When the CPU encounteres a send instruction, it only puts the send request into the send queue and goes on with its work. The CP, on the other hand, polls the send queue and executes

the send requests. Another task of the CP is to put messages received from another node into the receive queue. On executing a receive instruction, the CPU goes to the receive queue and takes the message out. Thus, the CPU need not be interrupted. Moreover, in a multi-tasking environment, the CPU need not perform environment switches whenever communication takes place.

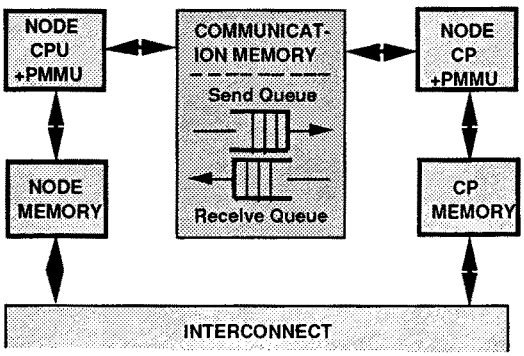

Figure 1 Communication between CPU and CP

Thus, a very efficient communication of short, fixed size messages is provided. The same mechanism is employed to establish the *rendezvous* needed to perform a secure data exchange between the address spaces of the different TOCs of the application program [13]. The data communication is supervised by an appropriate *lightweight* operating system process, supported by hardware DMAs in both the sending and the receiving node [13].

The CP in the node of a VSMA can readily perform also the functions of the page manager as described below. The copy set table the page manager must maintain for every page owned can be realized as a hash-coded associative memory. In this case, these tables can be stored in the private memory of the CP, and the CP may execute the hash function. Moreover, the CP -- rather than dedicated special hardware as proposed in [1] -- may support the node CPU in dynamically scheduling the concurrent execution of the tasks inside the node.

It can be shown that a very favorable interconnection structure for highly parallel MIMD systems is the hierarchy of crossbars, e.g., the *Simple TICNET* or the *Extended TICNET* [15]. Networks of this type combine a high connectivity (low probability of blocking) with a high transmission bandwidth and low cost of realization [15]. The specific demands on the interconnect put forth by VSMAs is the need for a very fast mechanism for conveying very short, fixed size messages from node to node. The very favorable blocking behavior of a hierarchical crossbar interconnect helps to meet that demand.

4.3 The Capability-Based Strong Data Coherence Mechanism

The capability-based strong data coherence mechanism implements the two governing rules given in Section 3.1.2, the *write capability grant rule* and the *read capability grant rule*. Capabilities follow the definition in Section 3.1.1 with the provise that in the case of a read capability the copy set is empty and need not be considered. As postulated by the rules, the owner of the page is responsible for sending a write or read capability to a requestor. In case of a read request, the owner node (more specifically, its CP) will just send a *read capability message* followed by a copy of the page. In case of a write request, the owner will send a *write capability message* followed by the copy set list. The new owner then must first send *invalidation messages* to all the nodes in the copy set. All three kinds of messages can be packed into one fixed message format, consisting of

(page_id, op_code, owner_id).

The receiving CP uses the message to update the appropriate page table entry for the identified page in the demand-paging memory management unit (PMMU). The PMMU page table entries and the copy set lists are the only data structures needed in our approach.

Note that in our scheme strong data coherence is maintained at all times, regardless of whether a TOC that has a read capability to a page really is reading it. Our rationale is that if one has a very fast way of passing

short messages, it is simpler and more efficient not to have to manage complex data structures and not to have to know anything about the state of the TOCs in other nodes. Short messages are exchanged directly between the CPs of communicating nodes, while larger blocks of data, e.g., pages or copy set lists, are transferred by a mechanism called *high volume data transfer* (HVDT) and described in [14]. For larger blocks of data the hardware (DMA) supported HVDT mechanism is the most efficient way of communication.

5. ADDING A WEAK COHERENCE MECHANISM

The weak coherence scheme allows for multiple writes by several TOCs into *different locations of the same page*. Some first implementations of the weak coherence scheme have been proposed in the literature [7], [16]. We present here a novel implementation which is implemented by aid of the strong coherence mechanisms defined above. Hence, strong coherence and weak coherence may coexist in a program. Our implementation is optimized for the VSMA described in Section 4.2.

We assume that a program block in which a multi-write is allowed is declared as a *weak block*, indicated, e.g., by a **begin_weak - end_weak** construct. In all program sections not declared as weak block the strong coherence protocol is obeyed by default. During execution of a weak block the ownership of the shared pages may arbitrarily change; i.e., when the weak block is left, the owner of a shared page may be the TOC who owned it on entering the block or it may be any other sharing TOC.

On entering the weak block the owner will send on request a *weak write capability* to any number of other TOCs. Inside the weak block all these TOCs can write the page concurrently. Eventually, on leaving the weak block, the different copies must be *merged* into one updated primary page that reflects all the changes made by the participating TOCs. Note that the weak block is entered from and left in a state of strong coherence.

This can be efficiently mechanized in the following way. Let $P_o = \cup_i p_{io}$ be the value of the original primary page. Let $P = \cup_i p_i$ be the value of the page after a multiple write. The new value P can be expressed by the identity

$$P = P - P_o + P_o = [\cup_i (p_i - p_{io})] + P_o$$

That is, the multiple writers form the differences $(p_i - p_{io})$ between the elements p_{io} of the primary page and the modified elements p_i. Each such difference can be viewed as an *update mask*. The updated page P is obtained by adding all the update masks to the original primary page P_o. Since different TOCs must not write into the same location there exists the additional condition:

$$[\cup_i (p_i - p_{io})] \cap [\cup_k (p_k - p_{ko})] = \emptyset,$$

if $\cup_i (p_i - p_{io})$ and $\cup_k (p_k - p_{ko})$ are the update masks of any two different writers and ø is the empty set.

The TOC that requests a weak write capability for a shared page receives from the owner a copy of the primary page P_o. As a first step, the CP of the receiving node duplicates that page. One copy will be used as *working copy*, the other one will be kept by the CP as *local reference copy*. From now on, the TOCs can concurrently write into their working copies.* At the end, they will subtract the local reference copy from the result to obtain the update mask. Subsequently, they will request the *strong write capability* from whoever the current page owner is at that time and update the page received. After the last sharer has done this, the result is the updated primary page in the sense of strong coherence. This mechanism allows the merge of all update masks to be performed in a distributed, pipelined fashion, rather than having one specific "page master" perform all the merges. Thus the potential bottleneck of a sequential merge is avoided.

It should be mentioned that any other strategy can be pursued in the merge operation; e.g., in certain applications it may be more efficient to perform a tree-structured merge. Unlike the solution described above, such a strategy will now require that the owner, who at the beginning distributes the multiple write pages, will also notify each writer where to send the result at the end. This information is readily available in the owners copy set list.

---

* Note that of N sharers only N-1 nodes need an extra local reference page.

6. CONCLUSION

The paper demonstrates that all the hardware support needed to implement the VSMA concept efficiently is to have a dedicated, fast communication processor in each node and to organize the private memory of the node as a virtual memory based on demand paging. Whether a broadcast capability would yield an additional gain (e.g., by broadcasting the invalidation messages) is a question for a more detailed investigation. One must consider that in the case of strong program locality broad-casting may not gain much; moreover, broadcasting does not go very well with the demand for a secure communication, i.e., a protocol with positive acknowledgement.

All the functionality of a VSMA with shared virtual memory, including the execution control of concurrent tasks, can readily and efficiently be provided by appropriate routines of a dedicated communication processor in the node. This leaves on the software side as the main effort the compiler issue. Specifically, the compiler for a VSMA should provide a favorable initial distribution of TOCs and data as well as a satisfactory page utilization of the shared virtual memory. Thus, the implementation of VSMAs raises a challenge as far as the compiler is concerned, presenting the opportunity for new, interesting research and innovative solutions.

Strong coherence can be maintained in a transparent manner without compiler support. This advantage must be paid for by a high page traffic because of the frequent page invalidations. *Weak coherence* lowers the page traffic considerably, however, requires explicit synchronization through the definition of *weak blocks*. This leaves the yet unanswered question to what extent this will put the programmer into the same situation as with distributed memory architectures, namely to have to explicitly program data access synchronization. There is one advantage with VSMAs, however: In the worst case, the user can fall back to working with strong coherence only, at the cost of a decrease in efficiency. The unique implementation proposed in the paper supports this by building a weak coherence scheme on top of a strong coherence mechanism, thus allowing both forms of coherence to coexist in the program.

References

[1] Anonymous: *PAX Standard Concurrency Control Architecture*, Revision 2.4 (Sept. 1989), Intel Corp. and Alliant Computer Systems Corp.

[2] Giloi W.K.: *Development of Future Supercomputer Architecture -- The Challenge of the Nineties*, Proc. 6th German-Japanese Technology Forum (1990)

[3] Li K.: *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D.thesis, Yale University 1986

[4] Li K. and Schaefer R.: *A Hypercube Shared Virtual Mermory System*, Proc. 1989 Internat. Conf. on Parallel Processing, IEEE Catalog No. 89CH2701-1, 125-132

[5] Lenoski D. et al.: *Stanford DASH Multiprocessor*, paper submitted for publication 1990

[6] Haridi S., Hagersten E.: *The Cache Coherence Protocol of the Data Diffusion Machine*, in Odijk E. et al.(eds.): PARLE '89 Parallel Architectures and Languages Europe, LNCS 365, Springer 1989, 1-18

[7] McBryan O.: *Aktuelle Entwicklungen im Parallelen Rechnen in den USA*, GMD Spiegel 1/89

[8] Stumm M., Zhou S.: *Algorithms Implementing Distributed Shared Memory*, COMPUTER (May 1990), 54-64

[9] Ricart G., Agrawala A.K.: *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, CACM 24 (Jan. 1981), 9-17

[10] Dubois M., Scheurich C., Briggs F.: Memory Access Buffering in Multiprocessors, Proc. 13th Internat. Symp. on Comp. Architect., IEEE Publication No. 86CH2291-3, 434-442

[11] Cheriton D.R. et al.: *Multi-Level Shared Caching Techniques for Scalability in VMP-MC*, Proc. 16th Annual Internat. Symposium on Computer Architecture (June 1989), 16-24

[12] Kennedy K. and Zima H.P.: *Virtual Shared Memory for Distributed-Memory Machines*, Proc. 4th Hypercube Conference 1989

[13] Schröder W.: *Overcoming the Startup Time Problem in Distributed Memory Architectures*, Proc. Hawaii Internat. Conf. on System Sciences (Jan. 1991), IEEE publication

[14] Giloi W.K., Schroeder W.: *Very High-Speed Communication in Large MIMD Supercomputers*, Proc. 3rd. Internat. Conf. on Supercomputing (June 1989), ACM Order No. 415891, 313-321

[15] Giloi W.K., Montenegro S.: *High-Bandwidth Interconnects for Highly Parallel MIMD Architectures,* Proc. 24th Hawaii Internat. Conf. on System Sciences (Jan-.1991), IEEE publication

[16] Bisiani R., Nowatzyk A., Ravishankar M.: *Coherent Shared Memory on a Distributed Memory Machine,* Proc. 1989 Internat. Conf. on Parallel Processing, IEEE Catalog No. 89CH2701-1, 133-141