IFIP Working Conference 5
Stanford, CA, USA, August 22-26, 1988

Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems

by

P. M. Behr, W. K. Giloi, W. Schröder

GMD Research Center
for Innovative Computer Systems and Technology

1. INTRODUCTION

Communication overhead is one of the crucial performance determining factor in highly parallel MIMD multicomputer systems. Besides the speed of the physical communication medium the throughput in such systems is mainly determined by the overhead induced by the communication protocols.

Therefore, it is mandatory to optimize the protocol hierarchy, by employing optimal software solutions that, in turn, are supported by fast protocol hardware, as well as a high-speed interconnection structure. A major design decision concerning the protocol hierarchy is whether to use synchronous or asynchronous communication mechanisms. This is strongly influenced by a number of system parameters such as: the relative performance of the hardware and software components, the communication system architecture, and the protocol hierarchy.

Large MIMD systems are by definition **message-based systems,** because for a larger number of nodes the 'hot spot contention' caused by shared memory access can be counter-productive. Consequently, all considerations below refer to message-based, i.e., **distributed systems.**

Performance, security, and usability of a MIMD system are highly determined by the models and mechanisms of inter process cooperation. Various models have been proposed as the basis for programming inter process cooperation in a MIMD environment. However, there is no single solution to the different requirements defined at the different system levels, as the optimal solution at each individual level depends on certain performance parameters and on the functionality at the hardware level.

In the attempt to optimize the system one is faced with a dichotomy of programmability versus efficiency.

- Programmability demands that the programmer sees an abstract machine that hides as much as possible the difficulties arising in writing, testing, and executing parallel programs. To this end, the lower level protocols should be hidden from the user.
- Efficiency demands that the user should be enabled to deal with the lower level protocols in order to minimize the communication overhead.

The question whether to employ synchronous or asynchronous communication mechanisms must be answered at each of the following three levels of the computer system:

- the application programming level
- the operating system level
- the hardware level.

In this paper we shall first take a look at the contradicting demands and trade-offs between synchronous and asynchronous communication, occurring at the three levels. Subsequently, we present two cases in point for design decisions of real systems and their underlying rationale.

2. SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION TRADE-OFFS

2.1. Application Programming Level

Parallel processing in distributed MIMD systems is efficient only if the functions that are executed in parallel are sufficiently complex, so that the parallel processing gain outweighs the unavoidable communication overhead. Experiences have taught us that any granularity of parallel execution finer than that of cooperating processes tends to violate that condition. Therefore, parallel execution of a multitude of cooperating processes constitutes the appropriate structure of parallel processing in MIMD architectectures.

A widely used model of inter process cooperation in distributed systems is that of a client-server relationship: Client processes request services from server processes. Under normal conditions, the server will always honor a request; however, the client has no control as to when a requested service will be delivered (principle of "cooperative autonomy"). Usually, client and server both will operate on the same shared data objects; hence, there is a potential for non-determinacy.

The simplest and therefore most popular solution to this problem is to suspend the client process as long as the server process is active. Since in distributed systems the client has no knowledge of when the server will start executing, it is suspended immediately after having issued the request. Such a simple scheme has the same effect as a procedure call and, therefore, is called **remote procedure call** (RPC) [1]. The advantage of the RPC is that it preserves the procedural view of the program, as well as the end-to-end significance of the application-controlled message exchange. Because of the synchronization of client and server, the message transfer can be performed directly between the address spaces of both processes with no intermediate buffering. However, though the RPC mechanism supports the distribution of processes in a MIMD system, it does not satisfy the parallel execution requirements.

More sophisticated schemes are the **rendevous** mechanism [2] or the **remote invocation send** mechanism [3]. Yet in these mechanisms there is still an unnecessary synchronization point at which the client process is suspended, until the receiver accepts the message, i.e., the rendezvous is established.

In a large MIMD system where high system performance is to be obtained through a high degree of parallelism, RPC or the rendezvous, which both reduce the potential for parallel execution of cooperating processes, are not appropriate as the only communication mechanism;

rather, the system should support an asynchronous **no-wait** inter process communication (IPC) [3] so that a client is not suspended after having sent a request to a server.

At the application programming level, asynchronous communication is the appropriate solution, since it does not prescribe any synchronization mechanism. Rather, synchronization is performed by constructs independent of the communication protocol and, thus, can either be user-defined or built-in, i.e., hidden from the user. In asynchronous communications a client process needs not to wait for a rendezvous or a reply by the receiver; rather, it can go on with its work until the one and only synchronization point is reached where the result of the receiver's work is needed.

Programming of asynchronous communication is more difficult and therefore more error prone. However, most programming errors can be avoided by the use of secure built-in mechanisms, supported by the operating system and the underlying hardware. The mechanisms should be reflected at the programming level by appropriate language constructs with a reliable and straightforward semantics.

2.2. Operating System Level

In addition to the management of computer system resources, the task of an operating system is to provide for some abstract mechanisms that hide hardware-specific details at the application programming level. In this sense the operating system implements a virtual machine and, thus, relieves application programs from having to cope with low-level device interface definitions. In the case of large scale message-based MIMD systems, the most crucial device interface to be managed is the network interface, i.e. the communication hardware.

If the functionality of the low-level communication hardware interface corresponds exactly with the functionality of the communication mechanism used at the application programming level, then no operating system activities are required during the communication phase. Only in this situation is the performance of communication hardware directly exploitable at the application programming level.

In order to improve parallelism within MIMD application programs, a **no-wait send** communication semantics should be provided by the lower-level communication system -- in the optimal case by the communication hardware. In addition, communication should be reliable, i.e. the application programs should not be aware of typical problems like node crash, message loss, message falsification, message duplication, and so on. Hiding these problems from the application programming level significantly increases the implementation overhead within the communication system. Generally, a multi-layer hierarchy of dedicated communication protocols is necessary. A complete hardware solution of these problems is not justifiable unless reliability can be guaranteed by the network.

If reliable asynchronous communication mechanisms are not provided by the hardware, then appropriate software levels are required. However, an implementation at the operating system level is not necessarily the most efficient solution, the reason being buffer and memory management problems [4]. Furthermore, at least one more address space boundary is to be bridged if messages are exchanged by the application programs (processes). Instead of directly passing the message to the peer application process, the message is first stored in operating

system buffers on a send request, to be retrieved on a receive request. For security reasons the operating system must be protected against the application programs by giving it a separate address space. This means passing address space boundaries twice for a single message-passing operation.

The problem can be mitigated by the use of copy-on-write mechanisms implemented by the operating system [5], however, at the cost of additional memory management hardware and sophisticated trap handling mechanisms. Rather than being physically copied, the pages constituting the entire message may be properly marked, so that an access fault will occur if the application program tries to write into the data area which holds the original message. In this case, the time for access control and trap handling must be sufficiently shorter than the time for copying the involved pages of a message. Furthermore, the success of this method depends highly on the application program behaviour.

Using a copy-on-write mechanism makes little sense if the application program tends to work on the entire original data set once it has been virtually transmitted. Because of the potentially large number of (time consuming) access faults, it will be a better choice to really copy the entire message. To hide this problem at the application programming level, an efficient implementation must be supported by dedicated hardware, and at the operating system level one must deal with all the problems of paging systems, such as finding appropriate application-oriented placement and replacement strategies for working sets.

Looking at some of the most recent distributed operating systems such as V [6], AMOEBA [7] and MACH [5], we discover that asynchronous communication primitives are not provided. Rather a synchronous request-response model of communication is favored, the reason being that synchronous communication proves to be the most efficient approach in the case of software implemented communication interfaces. Separate mechanisms such as **lightweight processes** or **multiple threads** are provided, which allow for the implementation of asynchronous communication, e.g., on a library package basis.

It is one of the most important characteristics of a powerful operating system to implement only policies which are most efficient with respect to low-level communication hardware capabilities. From the operating system level point of view this means a strict separation of objectives, leading to two basic, efficient mechanism: synchronous communication primitives and lightweight processes, instead of a single, overloaded mechanism. Consequently, synchronous communication is the appropriate solution at the operating system level.

2.3. Hardware Level

The issue of synchronous versus asynchronous communication at the hardware level concern the interface between the operating system and the message passing hardware of the communication system. If an architecture does not provide any communication support, message passing must be implemented by system software.

Message passing is performed synchronously if the communication hardware of the sending node awaits an acknowledgement from the receiving node before accepting further messages (end-to-end significance). Once the positive acknowledgement has been received, further messages may be sent. In the case of a negative acknowledgement, the message may be

retransmitted, or an error must be signaled to the upper protocol level. Note that an acknowledgement may be issued either by a hardware signal or a low level **acknowledge command**.

Asynchronous message passing implies that the sending node is able to send more than one message without waiting for an acknowledgement. There may be no acknowledgement mechanism at all (datagram service), or the acknowledgements arrive asynchronously. Asynchronous acknowledgements must identify the corresponding message, thus, asynchronous acknowledgements cannot be simple hardware signals or low level commands. Normally they need to be transmitted as special protocol messages. If the software levels use the asynchronous message passing mechanism provided by the hardware, the acknowledgement messages must be processed by the communication hardware itself, to avoid additional protocol overhead in the software. Positive acknowledgements are used to implement time-out mechanisms, while a negative acknowledgement message leads to the retransmission of the corresponding message.

Transmission errors that cannot be processed by the communication hardware must still be signaled to the software level, and, because of their asynchronous nature, cannot be processed individually. Therefore, error processing mechanisms such as backward error recovery must take place.

Our discussion shows that asynchronous message passing at the hardware level results in a rather complex mechanism. Therefore, the use of a no wait send protocol pays only when being executed by a dedicated **communication processor** (CP). The CP shares address spaces with the node CPU; that is, it operates as a coprocessor. This may raise new problems caused by the communication overhead between the node CPU and CP. Consequently, while the CP efficiently conducts asynchronous communication for the application program, synchronuous communication between CP and CPU is required to use the CP efficiently.

The additional copying overhead for mutual exclusion of write- access can be avoided by introducing a capability addressing scheme for the objects managed by the memory management unit (MMU), i.e., at the memory segment level [8]. There exists only one write capability to each object. This simple scheme causes no additional time overhead at the cost of a larger segment descriptor table and improves the data security in the system.

3. TWO CASES IN POINT

In the following, the trade-offs one has to deal with and the design decisions one may end up with are demonstrated by two cases in point. Both examples concern the communication mechanisms a multicomputer systems. Both examples differ considerably in the requirements they have to satisfy.

One case is a distributed, fault-tolerant multicomputer system, in which a larger number of nodes are interconnected through a token ring. The system is heterogeneous, i.e., the nodes may be of different functionality -- they may even be supplied by different vendors. The commonality of the system is that the application programmer has the same abstract view for all the nodes.

The seconds case concernes the design decisions made in the SUPRENUM architecture, a large MIMD/SIMD supercomputer for numerical applications. SUPRENUM consists of 256 processing nodes (PN), each PN being a complete, single-board vector machine with 20 MFLOPS peak performance and 8 Mbytes of main memory. SUPRENUM is a distributed system, i.e., each PN has its own operating system, and the global system functions are jointly performed by the collective of node operating systems. In a number cruncher like this, of course, the aspect of performance maximization has highest priority. Equally important, however, is the aspect of programmability, and it is a challenging task to reconcile these two aspects.

3.1. The Remote Service Request (RSR) in DELTA-4

DELTA-4 is an ESPRIT-funded project concerning the development of an open, dependable, distributed multicomputer architecture. Based on a high performance and fail safe communication system, DELTA-4 supports distributed applications especially in the field of computer integrated manufacturing (CIM) and office automation.

To simplify the programming of DELTA-4 applications, the programmer is confined to a high level policy of interprocess cooperation, called Remote Service Request (RSR). RSR is a refinement of the "Remote Process Invocation" (RPI) protocol which we developed almost ten years earlier [9].

The RSR-model supports RPC-like invocations of remote services, while synchronization is strictly "no-wait". A **client process** requests the execution of a remote service rendered by a **server process** in the the same fashion as a remote procedure call (RPC). However, in contrast to RPC the client and server processes can execute concurrently, for the RSR-semantics guarantees distinct execution contexts for both processes.

If the server terminates, the results specified in the RSR-call are passed back to the environment of the client. If the client requests the results, he either is given access to the result parameters, or an error message is produced. In the latter case the system guarantees that the parameters resume their original state. Figure 1 gives a simple example of the RSR language constructs.

The fault tolerant features of the DELTA-4 system are based on a dedicated communication processor in each node with a built-in voting mechanism on the messages of the replicated processes in the system. The execution of the asynchronous RSR protocol is performed by this communication processor and causes only a negligible overhead during normal, error-free inter process communication.

The required access control for the RSR protocol is achieved by a capability addressing mechanism at the level of memory objects (segments) [8]. To this end, the usual memory management unit is simply expanded so that it exercises access right control on the segments. Thus, objects used for interprocess communication are visible to the hardware, so that it can control the exchange and the access of the (shared) communication objects.

We may add the corollary that in a preceding development to DELTA-4, called the UPPER system [10], we took already the approach of supporting a no-wait IPC protocol by a dedicated "communication handler" processor (CH) in each node. Since at that time, around 1980, the

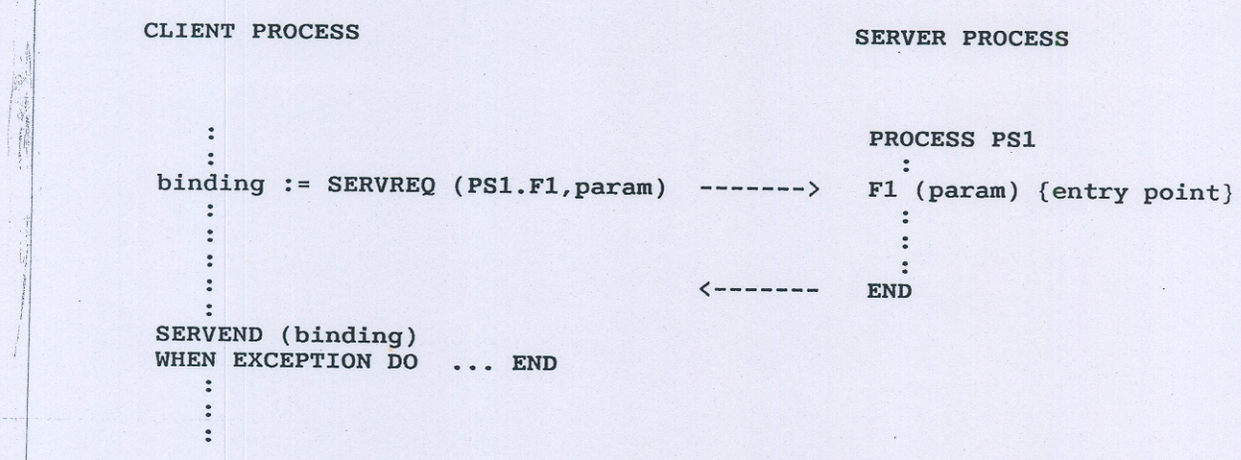

Figure 1: RSR Language Constructs

coprocessor solution did not yet exist in the microprocessor realm, we had to find out that the gain obtained by adding the CH was eaten up by the communication overhead between CPU and CH.

3.2. Inter Process Communication in the SUPRENUM Supercomputer

A MIMD/SIMD machine like SUPRENUM combines two major aspects of programming, the MIMD aspect and the SIMD aspect.

MIMD aspect: An application program must be partitioned into a number of cooperating processes, which then are distributed over the nodes of the system. Consequently, the programming environment must be based upon a process concept, including the appropriate inter process communication protocol (IPC).

SIMD aspect: Each node contains a pipelined vector processor. Consequently, the programming language must be a vector language (e.g., FORTRAN-8X).

The following is a list of requirements set forth for the design of the IPC protocols of the SUPRENUM machine [11].

- Message transfers shall be performed by the message passing kernel of the operating system without buffering and memory management.
- The communication system overhead shall be minimized.
- Application-oriented communication primitives shall be implemented in a problem-oriented fashion.
- Communication primitives and/or functionalities shall be migratable into hardware or firmware for speed-up.

The SUPRENUM node operating system, PEACE (Process Execution And Communication Environment) [12] is extremely modularized, consisting of a multitude of **lightweight processes**. In order to keep nevertheless the overhead of such a system low, PEACE employs the **team concept** [13], a team being a collection of lightweight processes that share a common address space. Within such teams, context switches become very fast. Except for the PEACE kernel, all servers in PEACE are implemented by such teams, the only task of the PEACE kernel being the communication.

The design of the PEACE communication system follows the idea of program families [14]. Figure 2 illustrates the mechanism by which PEACE exercises asynchronous IPC.

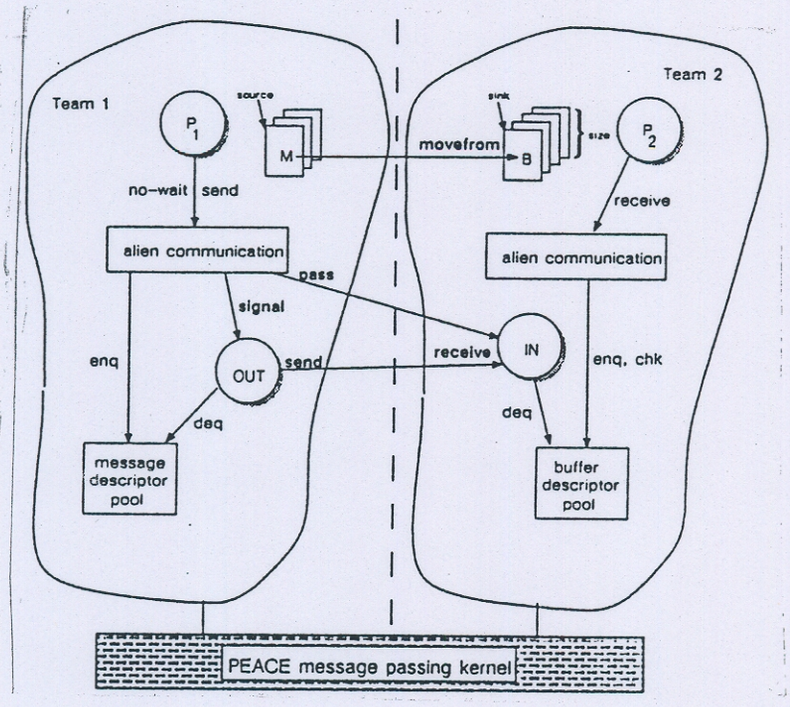

Figure 2: Asynchronous Communication in PEACE

Let us assume, process P1 wants to send a message M to process P2 (into message buffer B) on the basis of a no-wait send semantics. Likewise, Process P2 should not be blocked if no messages were sent by P1. In the following all bold faced items represent fundamental message passing primitives of the PEACE operating system kernel.

Process P1 is assisted by a lightweight process OUT. Both processes reside within the same team, team-1. The general meaning of OUT is to represent an address space connection endpoint, thus giving peer processes the capability for reading from/writing into the address space of team-1. The address space interconnection is realized on the basis of synchronous communication primitives. Instead of sending the entire message M, OUT only delivers a small and fixed size message to the peer site. This message contains the message descriptor for M -- more than one message descriptor may be sent as one single PEACE message. Sending the message means that OUT requests the peer to establish an address space interconnection.

Process P2 is assisted by a lightweight process IN. Both processes reside within the same team, team-2. Basically, the task of IN is to receive message descriptors, using synchronous communication primitives, and to control the data transfer activity for P2. In addition, IN establishes an address space interconnection for team-2 to peer address spaces (represented by peer processes).

Between OUT and IN an address space interconnection is established in order to give team-2 direct data transfer access rights to team-1. The interconnection is requested by OUT by issuing a blocking *send(OUT, request, answer)* to IN. The interconnection is established by IN by issuing a blocking *receive(request)*, having not yet executed *reply(OUT, answer)*, which would deblock OUT. Once a message has been received from OUT, team-2 (i.e., P2 as well as IN) is able to actually transfer the message M into its own address space (viz. buffer B). To this end, the non-blocking "high volume data transfer" primitive *movefrom(OUT, source, sink, size)* is employed, which performs a bulk data transfer without buffering in the communication system. In a similar fashion, *moveto(OUT, source, sink, size)* may be applied by team-2 to perform a high volume data transfer into the address space of team-1. As long as the address space interconnection is not closed, by using the *reply* primitive, team-2 is able to asynchronously transfer arbitrary size messages from/to team-1.

In case that P1 performs further no-wait sends, the corresponding message descriptors may be transmitted directly to IN by issuing a non-blocking *pass(IN, request, OUT)* routine. Basically, this primitive directs the peer network device driver system to buffer the message descriptor within the low-level message queues used for handling message receive interrupts. Again, IN receives a high volume data transfer request, this time passed by P1. As long as IN does not close the address space connection to team-1 (held by OUT) it is guaranteed that *movefrom* and *moveto* can be directly executed by team-2; else the interconnection must be re-established by *send* and *receive* sequences.

Most communication activities of P1 are controlled by user-level libraries. This reduces the number of context switches between user and system mode of operation. Basically, a no wait send is implemented by queueing a message descriptor local to team-1 (using *enq*) and signaling OUT (applying *signal(OUT)*) to request the address space interconnection. In a similar fashion, the communication activities of P2 also are controlled by user-level libraries. More specifically, checking for messages sent by team-1 is reduced to the simple observation of the buffer descriptor pool local to team-2 (performed by *chk*). **Transparent** to P1 and P2, the "alien communication" system moves data addressed by team-1 messages into the buffer space of team-2. This fundamental PEACE IPC renders the transfer of short message descriptors as efficient as possible.

Note that data transfer always is performed directly between peer teams. This means that the network interface controller is directed to transfer a message into the receiver space without any intermediate buffering. The *send* and *receive* primitives are applied by the asynchronous communication system in order to ensure that there are enough buffer resources within the peer team address spaces. A *movefrom* and *moveto* directly results in properly setting up the network interface for a high volume data transfer. There is no need to check for buffer resources and, therefore, the hardware is completely available for the data transfer activity.

The communication coprocessor on each node supports the message exchange between processes up to a very high level, that is, the hardware is able to autonomously send and receive messages consisting of several abitrarily sized objects by single coprocessor instructions. To minimize the protocol overhead, the communication system guarantees a reliable communication between the nodes, based on error correcting codes. This allows the use of simple and efficient blast protocols throughout the system.

4. CONCLUSION

As the discussion shows, asynchronous communication has decisive advantages in the programming of MIMD parallelism. At the hardware level, asynchronous communication can be a performance increasing measure. However, such performance gains are obtained only when the asynchronous mechanisms are adequately supported by dedicated hardware, and a careful trade-off analysis is needed to determine whether the additional hardware resources are worth their cost.

At the operating system software level, for efficiency reasons synchronous communication is the appropriate solution. To support nevertheless the asynchronous communication at the application programming level, either a special communication coprocessor hardware must be provided, or the operating system design must be specifically tuned, by employing lightweight processes as described in the paper. Both approaches may be combined by migrating the functionality of some of the lightweight processes into dedicated coprocessor hardware.

References

[1] Birrel A.D., Nelson B.J.: **Implementing Remote Procedure Calls**, ACM Trans. Computer Systems 2,1 (Jan. 1984), 39-59

[2] US Department of Defense: **ADA Reference Manual**

[3] Liskov B.H.: **Primitives for Distributed Computing**, MIT Laboratory for Computer Science, CSG Memo 175, May 1979

[4] Lantz K. A., Nowicki W. I., Theimer M. M.: **An Experimental Study of Distributed Application Performance**, Technical Report STAN-CS-86-1117 (also available as CSL-85-287), Department of Computer Science, Stanford University, 1985

[5] Young M., Tevanian A., Rashid R., Golub D., Eppinger J., Chew J., Bolosky W., Black D., Baron R.: **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**, ACM Operating Systems Review, 21, 5, *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, 1987

[6] Cheriton D. R.: **The V Kernel: A Software Base for Distributed Systems**, IEEE Software 1, 2, 19-43, 1984

[7] S. J. Mullender, A. S. Tanenbaum: **The Design of a Capability-Based Distributed Operating System**, The Computer Journal,, Vol. 29, No. 4, 1986

[8] Behr P.M., Giloi W.K., Gueth R.: **Object Addressing Architectures**, *Proc. Internat. Workshop on Computer Systems Organization*, IEEE Catalog no. 83CH1879-6 (1983),

202-209

[9] Giloi W.K.,Behr P.M.: **An IPC Protocol and Its Hardware Realization For High-Speed Distributed Multicomputer Systems,** *Proc. 8th Internat. Symposium on Computer Architecture*, IEEE Catalog no. 81CH1593-3, 481-494

[10] Giloi W.K., Behr P.M.: **Making Distributed Multicomputer Systems Safe and Programmable,** *Proc. Internat. Workshop on High-Level Language Computer Architecture*, University of Maryland, College Park, Md. 1982

[11] Behr P. M., Giloi W. K., Mühlenbein H.: **SUPRENUM: The German Supercomputer Architecture,** *Proc. 1986 Internat. Conf. on Parallel Processing*, IEEE Catalog no. 86CH2355-6

[12] Schröder W.: **A Distributed Process Execution and Communication Environment for High-Performance Application Systems,** Lecture Notes in Computer Science, 309, J.Nehmer (Ed.), *International Workshop on "Experiences with Distributed Systems"*, Kaiserslautern (West Germany), Sept. 28 - 30, 1987

[13] Cheriton D. R.: **Multi-Process Structuring and the Thoth Operating System,** Univ. of Waterloo, UBC Technical Report 79-5, 1979

[14] Parnas D. L.: **On the Design and Development of Program Families,** Forschungsbericht BS I 75/2, TH Darmstadt, 1975