1989 International Conference on Supercomputing (ICS '89)
Crete, Greece, June 5-9, 1989

# VERY HIGH-SPEED COMMUNICATION IN LARGE MIMD SUPERCOMPUTERS

by W.K. Giloi and W. Schroeder-Preikschat

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

**Abstract.** The next generation of supercomputers will be largely parallel MIMD architectures, ranging in peak performance from 10 to 100 GFLOPS in the mid nineties to 1000 GFLOPS in the late nineties. *Largely parallel* means that such a system will consist of hundreds or thousands of processing nodes (PN), and each PN will have a peak performance of several hundred MFLOPS. Obtaining such an extremely high performance is not only an issue of appropriate node architecture but requires also a very high bandwidth interconnection network and an extremely fast implementation of the inter process communication (IPC) protocol. The paper deals with an IPC protocol implementation that reduces the communication startup time to approximately 20 microseconds, by combining highly efficient software solutions, given in the form of lightweight processes, with dedicated hardware, given in the form of a specific communication processor in each PN, to perform the rendezvous required between sender and receiver processes.

## 1. Introduction

The next generation of supercomputers will be largely parallel MIMD architectures, ranging in peak performance from 10 to 100 FLOPS in the mid nineties to 1000 GFLOPS in the late nineties. *Largely parallel* means that such a system will consist of several hundreds to thousands of processing nodes (PN). Each PN will have a peak performance of several hundred MFLOPS. The programming paradigm of such system is based on the partitioning of the application program into a large number of cooperating processes. To this end there must be an appropriate, extremely efficient *inter process communication* (IPC) protocol hierarchy, based on which the programmer will be provided with the appropriate language constructs to implement the IPC and, consequently, the implicit or explicit synchronization of the cooperating application tasks.

Performance, safeness, and programmability of a MIMD system all depend very strongly on the models and mechanisms of inter process cooperation. Various models can be considered [1]; however, there is no single solution to the different requirements defined at the different levels [2]. Rather, the optimal solution at each individual level depends on certain performance parameters and on the functionality at the hardware level. These issues will be addressed in the paper: Functionality and performance requirements are established, and it is shown how these requirements can be met.

## 2. General Requirements

It is safe to state as a general premise concerning the realization of inter node communication that large MIMD architectures must be *message passing systems*. While the use of shared memory is a most simple and efficient approach in small multiprocessor systems, it is not the appropriate solution for large MIMD architectures for reasons of system manageability, efficiency, and availability.

*Manageability*
The IPC policies appropriate for the paradigm of largely parallel MIMD systems are more straightforwardly and conveniently implemented by high-level message passing constructs (e.g., send-receive) than by the lower level mechanisms for controlling the access to critical regions of shared memory.

*Efficiency*
Of course, message passing policies could be implemented in a shared memory environment; however, it is more efficient to implement the message-passing IPC protocol also by message passing mechanisms. In this case the implementation may readily be by hardware. Furthermore, the danger of *hot spot contention* of shared memory access is avoided.

*Availabilty*
Even with the fairly reliable technology available, systems featuring a very large number of PNs will exhibit a relatively poor overall MTBF, if no provisions are taken to

make the system immune against failures of the hardware. Therefore, a sufficient degree of fault tolerance at the granularity of the PN should be designed into the system. This is feasible only if the architecture is designed as a loosely coupled *distributed system*.

A major design decision concerning the protocol hierarchy is whether to use synchronous or asynchronous communication mechanisms [2]. This is strongly influenced by a number of system parameters such as: the relative performance of the software and hardware components, the communication system architecture, and the application requirements.

In the attempt to optimize the system one is faced with the 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. Hence, 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.

This dichotomy can be resolved by combining software, given in the form of *lightweight processes* (LWPs), with hardware, given in the form of a dedicated *communication processor* (CP). Operating system tasks as well as user tasks will be implemented by *teams* of LWPs which all share the same address space. That is, the domain of protection is the team and not the LWP, thus reducing the cost of process switches within a team drastically [3].

## 3. Performance Requirements

Practically, a system with hundreds or thousands of PNs is realizable only if very highly integrated circuits (VLSIC) are employed, leading to the use of CMOS or BICMOS circuits (rather than ECL or GaAs technology). Currently, the maximum clock frequency of CMOS processors is about 30 MHz; through the use of BICMOS it can be increased to 50.....60 MHz. Consequently, it can be (conservatively) estimated that soon the "natural" processing power of a single-processor PN will be approximately 50 MIPS and 100 MFLOPS for two chained operations.[1] Such a processing rate matches with the access bandwidth of static CMOS memory (SRAM) or even dynamic memory (DRAM).[2]

However, there exists another parameter limiting the peak performance of a PN, which is the performance of the IPC. Thus, as much efforts must be devoted to the realization of an approriate communication system as to the goal of obtaining the highest possible performance from the PN. Simulations have shown that for typical supercomputer applications, e.g., the solving of partial differential equations, a node performance of about 100 MFLOPS requires that the startup time be limited to no more than 40 microseconds [5].

It would be a mistake to view these problems solely as a question of the design of the physical *interconnection network* (IN), as have so many authors over the years. What must be considered carefully is the entire functionality of the *abstract machine level* communication model and not just the physical behavior of the underlying IN hardware. Therefore it is mandatory to optimize the entire protocol hierarchy, by employing optimal software solutions which, in turn, are sufficiently supported by fast protocol hardware as well as by an adequate bandwidth of the IN.

Note that in a message-passing system the notion of memory addresses does not exist outside the node memory. Rather, message passing among PNs is a memory-to-memory copying process performed by DMA hardware. For the sake of uniformity, the same mechanism is used even for message passing between different user tasks residing in the same PN.

## 4. Communication Performance Parameters

In the endeavor to optimize the performance of the communication system in large message based MIMD architectures, two major levels must be distinguished:

- the physical transmission level
- the IPC level.

The performance at the physical transmission level can be expressed by the transmission time $\mathbf{T}_{\text{trans}}$ for a message. $\mathbf{T}_{\text{trans}}$ can be expressed by the formula:

$$\mathbf{T}_{\text{trans}} = (\mathbf{T}_{\text{latency}} + \mathbf{L}_{\text{message}}/\mathbf{R}_{\text{bit}}) + \mathbf{F}(\mathbf{N}_{\text{hops}})$$

Here, $\mathbf{T}_{\text{latency}}$ is the elapsed time between the issuing of the send command and the actual start of the bit stream that is the message. In case the IN is not blocking-free,

1 Currently, floating-point processor devices are available that allows one to realize pipelined vector processors with 80 MFLOPS peak performance [4].

2 DRAM requires the use of the static column mode or videoRAM, in combination with memory interleaving, to obtain the necessary access bandwidth.

$\mathbf{T}_{latency}$ includes any wait time caused by blockings. $\mathbf{R}_{bit}$ is the effective rate at which the message is transmitted, measured in bits per second, and $\mathbf{L}_{message}$ is the length of the message, measured in the number of bits.

$\mathbf{F}(\mathbf{N}_{hops})$ is a factor which depends on the number of stations (the hops) over which the message must travel. In store-and-forward package switching, $\mathbf{F}(\mathbf{N}_{hops}) = \mathbf{N}_{hops}$. In other schemes such as *wormhole routing*, $\mathbf{F}(\mathbf{N}_{hops}) \geq 1$ is a function which increases less than linearily.

The overall performance of the message passing system -- hardware and software -- can be measured by the total time needed to perform a complete IPC exchange:

$$\mathbf{T}_{ipc} = \mathbf{T}_{startup} + \mathbf{T}_{trans}$$

$\mathbf{T}_{startup}$ is the total time needed to exercise the IPC protocol. It is assumed that during that time the PN cannot do anything else. Consequently, the cost of a message exchange can be expressed in MFLOPS lost. $\mathbf{T}_{startup}$ depends on two parameters:

- the protocol functionality and
- the implementation.

From the functionality point of view, what one would like to have is the capabilities of:

- programming in a multi-tasking environment;
- dynamic process creation and relocation;
- domains of protection (fire walls) at the task level;
- asynchronous communication of objects of arbitrary size at the programming level [2]

A message passing architecture by which all these demands are satisfied is the SUPRENUM supercomputer [6]. In SUPRENUM the communication protocol handling is performed by the kernel of the distributed PEACE operating system [7]. PEACE has been optimized to provide all these functions by a most efficient implementation. To this end, PEACE is based on the team concept outlined above (PEACE is one of the fastest message-based operating system presently existing).

In PEACE the startup time for a message transfer is caused by the following activities [8]:

| | |
|---|---|
| 7.5 % | by supervisor space switches |
| 15.0 % | by process dispatching |
| 17.5 % | by team isolation |
| 20.0 % | by interrupt handling. |
| 40.0 % | by the communication routines proper. |

That is, nearly two third of the startup time would disappear if there were only one process running on each node. Implementations less efficient than PEACE will exhibit even a much higher overhead caused by process switches (in most existing message-based systems the typical startup time is in the order of magnitude of a millisecond).

## 5. What Complexity Can Be Sacrificed?

Any reduction of protocol functionality must be paid for by certain sacrifices. In principle, sacrifices may be made with respect to:

- system safeness
- convenience of programming
- parallelism
- error detection / fault tolerance

*System safeness*
To provide the desired robustness against programming errors, *fire walls* must exist to protect system tasks and user tasks. System security may be sacrificed only if it can be guaranteed that the applications programs are largely free of bugs. Otherwise, tracing and debugging a program running on a machine with hundreds or thousands of concurrently PNs could become a nightmare. Another important issue of system security is the demand for *end-to-end significance* of communication [9], meaning that a communication has been successfully completed only after the sender task has received an acknowledgement from the receiver task, signalling that the latter has received the message correctly. An acknowledgement that signals only that a message has been buffered at the receiving end has no end-to-end significance; thus, end-to-end significance calls for synchronous communication.

*Convenience of programming*
The overhead caused by interrupt handling and environment switching could be eliminated by having only one single process on each PN. However, multiprocessing is mandatory for reasons of system scalability, load balancing, and fault tolerance, which all require process redistribution and, thus, a multiprocessing environment. System scalability means that a program can be written regardless of the size (the number of PNs) of the system. Load balancing is another case that requires a redistribution of processes, and so is fault tolerance, where the tasks performed by a failing PN must be redistributed over the other PNs. An additional bonus of a multitasking environment is that the user can be provided by the operating system with server tasks, rather than having to bind additional software modules for the required functionalities into the application programs.

*Parallelism*
Asynchronous communication, as implemented by a *non-blocking send*, makes it easier to exploit the potential of concurrent execution of the application program tasks. On the other hand, because of the buffer management involved,

it has a considerably higher overhead than synchronous communication [2], and it makes it difficult to provide the required end-to-end significance of communication. There are certain classes of applications where a simple *lock-step synchronization* is quite adequate, i.e., a scheme where communication steps and computation steps alternate; in this case, communication is synchronous. There exist important applications, however, where programming convenience may call for asynchronous communication.

In our opinion, the optimal approach is to employ synchronous communication as the basic primitive of the *abstract machine*. If the user wants a no-wait send, this can be performed by special server tasks provided as library functions. The advantage of this approach is that only those users have to pay for the higher overhead of asynchronous communication who want it.

For all these reasons, a large, expensive, general-purpose number cruncher should not be run like a dedicated machine. Besides, even then could a startup time as low as 20 microseconds never be achieved.

## 6. The Synchronous IPC Model

In order to implement network-wide IPC a multi-layered approach is typical. For message-passing MIMD machines, which offer multi-processing on the individual PNs, at least two problem-oriented protocol layers are necessary. The bottom layer deals with inter-node communication, on top of which the IPC layer is settled. For high-performance IPC, the fusion of these two layers becomes a key position. Figure 1 illustrates the two levels at which the IPC protocol is executed, level 0 and level 1.

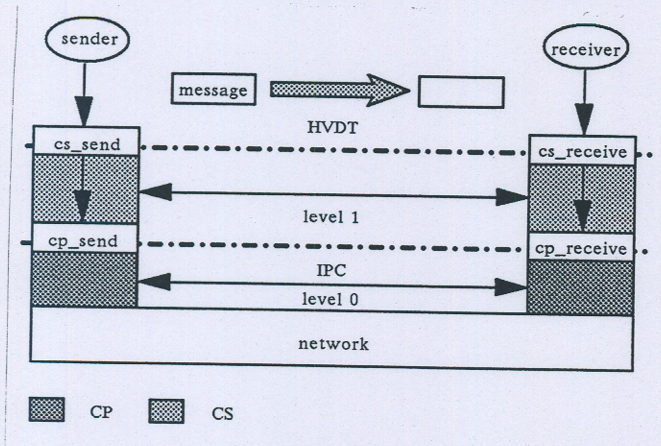

Figure 1.

At the top level (user level), the transfer of a data stream of arbitrary size to a receiver process is requested by the sender process. At the bottom level (level 0) the data transfer is initiated. The intermediate level (level 1) serves as the "glue" between the top and the bottom level. Its main purpose is to provide in cases where the semantics at the user level differs from that at the hardware level the necessary mapping of user-level communication requests, *cs_send* and *cs_receive*, onto the corresponding communication processor primitives, *cp_send* and *cp_receive* (*cs* stands for communication system, *cp* stands for communication processor). Since the mapping overhead increases with the *semantic gap* between CS and CP, keeping this gap as small as possible is a most important prerequisite for efficiency.

Synchronous communication eliminates the semantic gap and; therefore, it is the most efficient communication policy. In the following section, some implementational details of the synchronous and the asynchronous communication model are presented.

## 7. Implementing the IPC Protocol

The IPC protocol must handle the network wide communication among user processes. In addition to carrying out data transfers, this task requires process management capabilities. The horizontal view of IPC in Figure 1 relates to the data transfer aspect, whereas the vertical view relates to the PN-bounded IPC protocol activities. As pointed out above, it is the latter that causes the bulk of the IPC startup time. Thus, the key to success is IPC protocol simplicity. Figure 2 illustrates a most fundamental IPC protocol implementation.

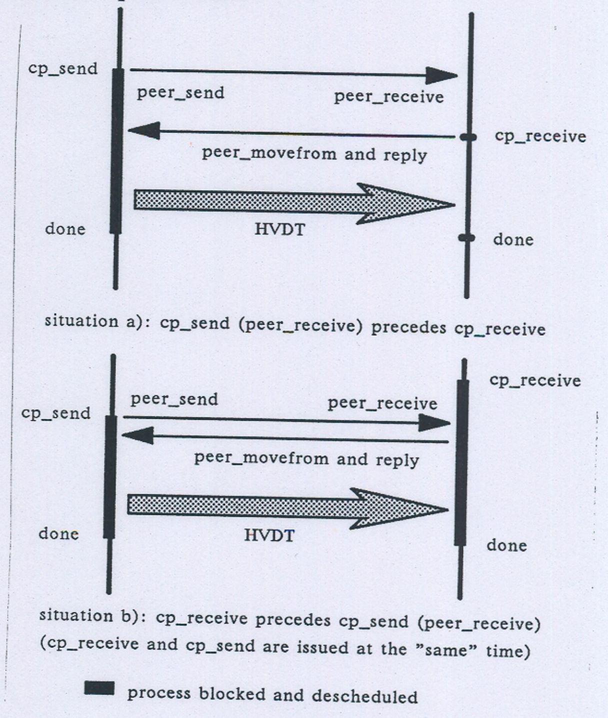
cp_send
peer_send
peer_receive
cp_receive
peer_movefrom and reply
HVDT
done
done
situation a): cp_send (peer_receive) precedes cp_receive
cp_receive
cp_send
peer_send
peer_receive
peer_movefrom and reply
HVDT
done
done
situation b): cp_receive precedes cp_send (peer_receive)
(cp_receive and cp_send are issued at the "same" time)
process blocked and descheduled

Figure 2.

The IPC protocol is split into two phases, *synchronization* and *data transfer*. For synchronization short and fixed size IPC messages are transmitted. The header field of the messages is mandatory and contains all the information needed to address both sender and receiver process and to encode the message type. The parameter field is optional and carries the descriptor of an arbitrarily sized memory segment that shall be transferred during the data transfer phase.

Synchronization is achieved by the processing of *peer_send* and *peer_receive* by the sender CP and receiver CP. A *peer_send* is the implementation of the CP primitive *cp_send*, whereas *peer_receive* implements *cp_receive*. The invocation of *cp_send* by a sender process leads to the execution of *peer_send* by the sender CP, which immediately blocks the sender process. The effect of *peer_send* is the transmission of a high-volume data transfer (HVDT) request to the receiver CP. This IPC message contains a user level message descriptor as parameter field and, arrived at the peer site, causes the receiver CP to perform the *peer_receive*, i.e., handle the interrupt and queue the HVDT request. The receiver process indicates its readiness to accept a message (HVDT request) by invoking *cp_receive*. The process is blocked only if no HVDT request has been received from the sender CP at the time *cp_receive* is executed by the receiver CP, i.e., if *cp_receive* precedes *peer_receive*. As soon as *peer_receive* and *cp_receive* have been executed by the receiver CP, the rendezvous between the sender and receiver process is established, and the data transfer phase can be started.

Data transfer is carried out by the execution of *peer_movefrom* by the receiver CP. This leads to the initialization of the DMA on both sides, in order to "blast" a complete data segment from the sender to the receiver address space. For this purpose the *peer_movefrom* is encoded in an IPC message and transmitted to the sender CP. Prior to this, the receiver CP already has set up its DMA. Arrived at the sender´s site, *peer_movefrom* causes the sender CP to set up its DMA accordingly and, thus, start the HVDT. Because the sender process has been descheduled and, therefore, another process may be active at the sender´s site, interrupt handling is necessary to execute the *peer_movefrom*, i.e., to accept the IPC message.

The completion of the HVDT implies process scheduling, for at least the sender process has been blocked by *cp_send*. It may also imply process scheduling at the receiver CP, namely in cases where the receiver process was blocked for having called *cp_receive* before a *peer_receive* took place. Once having been rescheduled, sender and receiver will return from *cp_send* and *cp_receive*, respectively. The rescheduling activities are interrupt-triggert, when the DMA transfers have been finished. Thus, the entire IPC protocol causes two interrupts at each CP. As we will outline later, the interrupt handling overhead can be reduced to one interrupt at each site if a dedicated hardware CP is used.

Note that the rendezvous and, thus, the HVDT is controlled completely at the receiver site. This enables the receiver CP to schedule several HVDTs with respect to memory and processor utilization at the receiver site. Thus, HVDT requests need not be processed on a first-come-first-serve basis.

## 8. Rationale for the Proposed IPC Protocol

The simplicity of this protocol allows for a very straightforward implementation, however, at the risk that both processes involved are descheduled, whenever *cp_receive* precedes *cp_send*. This situation could be avoided by modifying the IPC protocol such that a HVDT request is passed not only to the receiver but also to the sender. In the implementation discussed above, the semantics of *peer_receive* is such that, unlike the sender process, the receiver process needs not be descheduled. Executing *peer_receive* at the sender site, i.e., queueing the HVDT request, has an analog effect: the sender process needs not be descheduled when *cp_receive* precedes *cp_send*, while the receiver process is always descheduled.

However, such a modification would still not resolve the problem how to handle the situation where *cp_send* and *cp_receive* are executed at the same time. Such a potential collision must be resolved by the IPC protocol. Moreover, this is another case where both processes will be descheduled, since no side has sufficient knowledge about the state of its peer.

The desirable mechanism would be to deschedule a process only if it is not ready for the rendezvous for lack of resources, e.g., a buffer. However, this can be achieved only at the cost of a considerable increase in protocol overhead caused by the extra protocol phase needed for checking the execution state at the peer site. In our opinion, it will not pay to spend additional protocol overhead all the time to avoid a worst case situation that may occur very seldom. Consequently, the simpler scheme described above is preferred.

## 9. Combination of Hardware and Software

The functionality of the IPC protocol is too complex, however, to be totally "cast into hardware." Rather, the appropriate solution is to combine hardware and software functionalities such that:

- the exchange of data objects between two cooperating tasks is performed by a lightweight server process,

- the underlying communication needed to establish the rendezvous between sender and receiver is carried out by dedicated communication hardware.

Of course, it is equally important that the interconnection network has a very low latency and is capable of transferring data at a very high rate. These requirements can be met by a novel, hierarchical interconnection structure called TICNET (Totally Interconnected Cluster NETwork) [10]. TICNET is a two-level interconnection network. At the lower level, groups of PNs are interconnected by high-speed parallel buses, thus forming a *cluster*. The cluster buses function also as the switching points of a distributed crossbar which, at the upper level, interconnects the clusters. Performance figures obtainable with state-of-the-art technology are up to 640 MBytes per second for the parallel bus system and up to 100 Mbytes per second for the (byte-serial) crossbar links.

Employing a server LWP for the transfer of data objects from one task to another provides the flexibility of having different servers for different IPC models such as:

- synchronous communication
- single-buffer asynchronous communication (SBAC)
- asynchronous ("no-wait send") communication.

*Synchronous communication* is a primitive operation of the abstract machine which is strongly supported by hardware. Consequently, no additional server process is needed. Figure 3 provides a more detailed view of the synchronous communication mechanism by which objects of arbitrary size can be transferred from a sender to a receiver task without intermediate buffering.

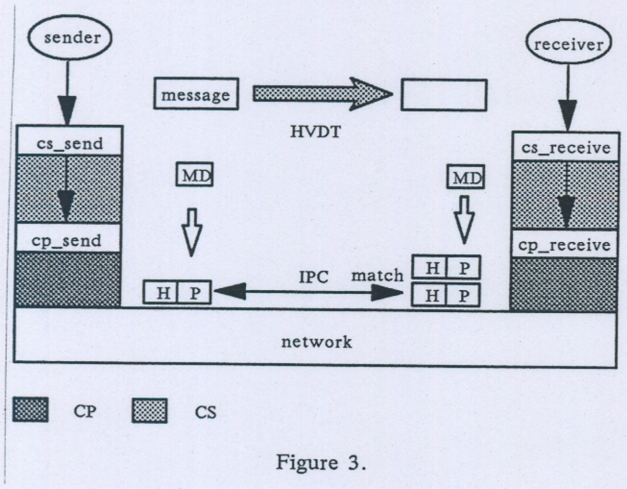

Figure 3.

The sender process invokes *cs_send* which, in turn, is mapped onto a *cp_send*, passing a message descriptor (MD) as argument. Subsequently, the sender process is temporarily blocked to prohibit it from overwriting the message buffer until the message has been consumed, i.e., until HVDT is completed. On the peer`s site, *cs_receive* is invoked by the receiver process, leading to the execution of *cp_receive,* also with a MD as argument. Note that the mappings, from *cs_send* to *cp_send* and *cs_receive* to *cp_receive*, respectively, can be performed by the compiler. The execution of *cp_send* and *cp_receive* leads to the initialization of an IPC packet, i.e., the header (H) is defined and the MD is copied into the parameter field (P). As result of *cp_send* the IPC protocol is started, i.e., the IPC packet is passed to the receiver CP, where it is queued. A match between the queued IPC packet and the IPC packet defined by *cp_receive* indicates that the rendezvous between sender and receiver process is established. As outlined above, at this point in time HVDT will be enabled by the receiver CP.

*Single Buffer Asynchronous Communication (SBAC)* has the following semantics. A task can send a data object to a buffer at the receiving end by a non-blocking send; however, it cannot send again until the previously sent message has been consumed. Therefore, only one buffer per channel is needed. Buffers can be of arbitrary size; they are declared by the programmer, created and allocated by the compiler, and implemented by a LWP as illustrated in Figure 4.

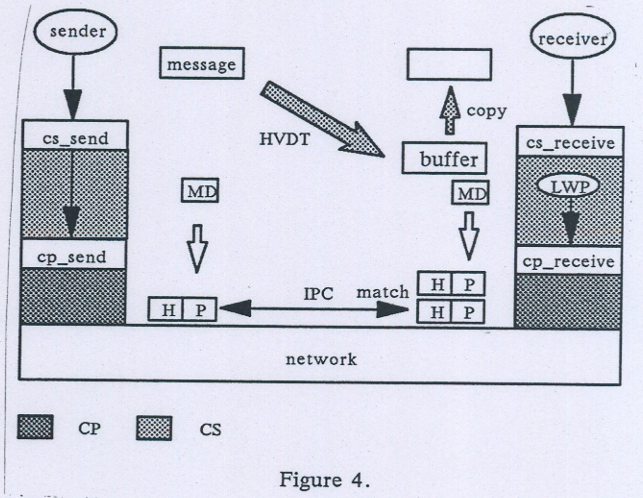

Figure 4.

The major difference to synchronous communication is the need for an additional message buffer and a LWP. The purpose of the LWP is to provide a separate thread of control within the context of the receiver process. The key idea behind this model is to double and migrate the receiver process down to level-1. The LWP double then executes the synchronous communication protocol described above. The execution state of the LWP determines the state of the additional message buffer, i.e., whether the buffer is empty or full. ´LWP ready for HVDT` (after having called *cp_receive*) indicates an empty buffer, into which the sender´s message can be copied. 'LWP not ready` indicates a full buffer, i.e., the previously sent message has not yet been consumed by the receiver process, and the sender process must be blocked until the LWP changes its state. Note, the synchronous IPC protocol implements the required synchronization between sender process and LWP and not between sender and receiver process.

*Unrestricted asynchronous Communication* means that instances of a data object can be sent blocking-free an arbitrary number of times. Since the compiler has no knowledge as to how often that will happen, sufficient buffer space must be allocated and managed. There exists the potential of buffer space overflow and a need for garbage collection, which both tend to reduce system performance drastically. Therefore, asynchronous communication -- if at all -- should be provided as a library function, so that only those users who want it must pay the price for it. Nevertheless, if this kind of communication shall be supported, an additional LWP and a buffer space is needed in the sender team. The sender LWP then works for the sender process in a non-blocking fashion and executes the synchronous IPC protocol with the receiver LWP.

## 10. Adding a Communication Processor

From the discussion above it becomes quite clear that the functionality provided by the CP must include:

- communication primitives
- DMA block transfers
- memory management functions
- process scheduling.

*Communication primitives*
The CP must be able to perform queuing and dequeuing primitives as indivisible operations, to implement *peer_send* and *peer_movefrom* efficiently. Performing a message transfer within 10 microseconds requires that the message length is fixed and as short as possible. It suffices to have messages containing the header information: (sender_id, receiver_id, message_type), which can readily be represented by 16 bytes (2 words). This rules out that the message exchange channel provided by the CP might be used also for the transfer of data.

*DMA block transfer*
The CP must be able to read in a DMA mode a block of data from the node memory and send it out on the interconnection network (IN), or receive a block of data from the IN and write it into the node memory. This capability can be refined to include the (structured) access to *data structure objects* [11], at the cost of a more elaborate address generator hardware.

*Memory management functions*
Since the CP assumes responsibilities of the operating system, it must be provided with the appropriate capabilities, including the right to operate with physical rather than logical memory addresses. Therefore, the CP must have access to system tables in the supervisor space, e.g., the memory segment table. This allows the CP, for example, to write into the address space of a suspended user process.

*Process scheduling*
The CP must have the ability to schedule or deschedule user processes by inserting them into or taking them out of the ready queue.

So far, the question is yet unanswered whether the CP is a dedicated hardware processor or a software processor running on the node CPU. However, it is obvious that the limit of 20 microseconds for the startup time can be met by the node CPU only under the following conditions.

(1) The node CPU encompasses the special communication functionalities listed above. This means that the designer must be free to design his own CPU.

(2) The node CPU is a multiple register set machine which allows the switching between operating system kernel, interrupt handler, and user process(es), respectively, to be performed in negligible time.

Having one CPU performing both communication and computation offers the decisive advantage that no interface -- and no additional overhead encountered with it -- between CPU and CP is needed. In the case that communication shall be performed by a dedicated hardware CP, it becomes extremely important that the communication between node CPU and CP is as efficient as possible.

If the designer is not free to build up his own CPU, a compromise is required if a multi-processing mode of operation shall be efficiently supported. This compromise consists of a dedicated hardware CP and software CP. The hardware CP is responsible for performing the low-level IPC protocol (i.e., queueing and routing the IPC messages) and HVDT (i.e., initialization of DMAs). The task of the software CP is to interface the hardware CP to sender and receiver processes being executed by the node CPU. This includes process dispatching, scheduling and context switching. Figure 5 illustrates this configuration, which is strongly supported by PEACE.

In PEACE, processes are interfaced to the low-level communication system (COSY) by a network independent communication executive (NICE). The NICE modul provides network-wide IPC and differs in the implementation for single-processing and multi-processing modes of operation. The interface between software CP and hardware CP is represented by COSY. From the point of view of the node CPU, the hardware CP is a device and, thus, COSY encapsulates the corresponding low-level device driver routines.

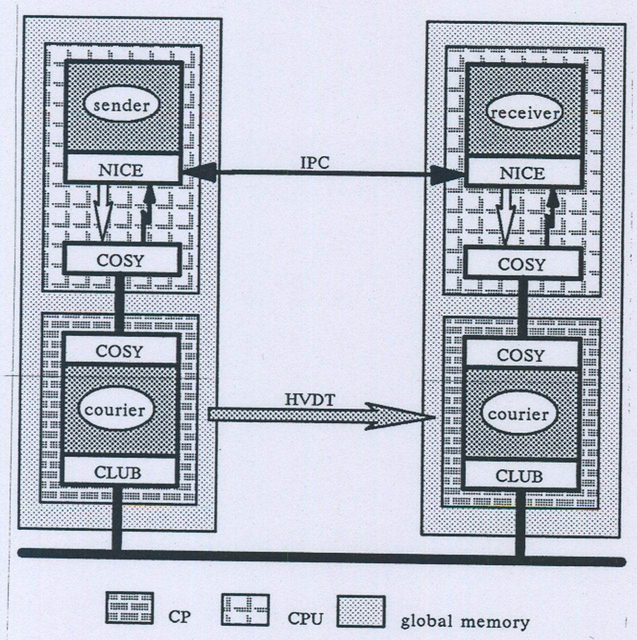

Figure 5.

Each hardware CP executes at least one process, the *courier*, whose task is the processing of IPC packets and the initialization of HVDT. The upper interface to the courier is represented by COSY and the lower interface gives direct access to the communication network, i.e. CLUB (cluster bus).

The interaction between software CP and hardware CP then works as follows. A HVDT request, carried by an IPC packet, is passed by both the sender and receiver process down to the courier. This is accomplished by a cooperation between NICE and COSY. The sender courier then routes the IPC packet to the corresponding receiver courier, where the packet is queued. Under control of the receiver courier the HVDT is initialized following the IPC protocol outlined above. Note, the activities of both hardware CPs (i.e., couriers) are performed in parallel to the activities of both software CPs (especially NICE). This means that process dispatching and scheduling is done in parallel to handling IPC packets and setting up HVDTs, i.e., executing the IPC protocol.

As soon as a HVDT has been completed, the node CPU is interrupted by the CP. There will be only one interrupt per HVDT at each node CPU. The interrupt indicates that rescheduling of sender and receiver process can be performed, which will be done by NICE.

From a software point of view, this configuration is very closely related to the communication models discussed in the paper. On the one hand, node CPU as well as hardware CP share the same global address space. On the other hand, sender (receiver) process and sender (receiver) courier may be considered as belonging to the same team, both sharing the team´s address space. In addition to that, the courier is a LWP that works for the original sender (receiver) process. Using a dedicated hardware CP for communication purposes means that a PEACE team can be executed in parallel, as it would be the case in a traditional shared-memory multi-processor system environment.

## Acknowledgement

The authors whish to express their gratitude to Messrs. Dr. Ulrich Bruening and Sergio Montenegro for critical reviews of the paper and valuable discussions.

This work was supported by the European Commission under the ESPRIT Program, grant no. 2447

## References

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

[2] Behr P.M., Giloi W.K., Schroeder W.:"Synchronous Versus Asynchronous Communication in High Perform-ance Multicomputer Systems", Proc. IFIP WG 2.5 Working Conference 5: Aspects of Computation on Parallel Proces-sors, Stanford University, August 1988, 187-196

[3] Cheriton D.R.:"Multi-Process Structuring And the Thoth Operating System, Univ. of Waterloo, UBC Tech. Report 79-5, 1979

[4] Bruening U., Giloi W.K.:"Architecture of a Functionally Parallelized Processor With Hardware Synchronization and Communication", Proc. 1989 Supercomputer Conference, May 1989

[5] Mierendorff H.:"Bounds on the Startup Time for the GENESIS Node", Internal Paper, GMD-F2.G1 1989

[6] Giloi W.K.:"The SUPRENUM Architecture", Proc. CONPAR 88, Part A, British Computer Society 1988, 1-8

[7] Schroeder W.:"The Distributed PEACE Operating System and Its Suitability for MIMD Message-Passing Architectures", Proc. CONPAR 88, Part A, British Compu-ter Society 1988,17-24

[8] Behr P.M., Schoen F., Schroeder W.:"The PEACE Message Passing Kernel Family", Internal Tech. Report, GMD FIRST 1988

[9] Saltzer J.H., Reed D.P., Clark D.D.:"End-to-End Arguments in System Design", ACM Trans. on Compu-ter Systems 2,4 (Nov. 1984), 277-288

[10] Giloi W.K., Montenegro S.:"Super Interconnection Networks for Supercomputers, Proc. 1989 Supercomputer Conference, May 1989

[11] Giloi W.K., Berg H.K.:"Introducing the Concept of Data Structure Architecture", Proc. Internat. Conf. on Parallel Processing 1977, IEEE Catalog no. 77CH1253-4C, 44-51