Overcoming the Startup Time Problem in Distributed Memory Architectures

W. Schroeder-Preikschat

GMD Research Center
for Innovative Computer Systems and Technology Berlin
Hardenbergplatz 2, D-1000 Berlin 12, Germany

Abstract

Massively parallel systems are distributed memory architectures consisting of a very large number of autonomously operating, interconnected nodes. The nodes cooperate through message passing via a high-speed interconnection network. The critical issue in such systems is communication latency. A major component of communication latency is the message startup time, i.e., the time it takes to excecute the appropriate operating system kernel function for message passing. One of the necessary system optimizations therefore concerns the startup time minimization. Our approach to this problem is to provide different versions of the kernel for the different modes of operation of the system. All versions have the same interface to the other modules of the distributed operating system and, thus, are interchangeable. This is accomplished by representing the message-passing kernel as an abstract datatype.

1. Introduction

Massively parallel systems are distributed memory architectures consisting of a very large number of autonomously operating, interconnected nodes. The nodes cooperate through message passing via a high-speed interconnection network. Consequently, the operating system of such a machine is a distributed operating system [21],[20]. One of the features of a distributed operating system is distribution transparency, needed to reference objects (system entities as well as user tasks) "in a consistent manner regardless of such factors as access, location, migration, and replication" [7].

The most important requirement for such a system is efficiency. The system may offer all the desired benefits such as resource sharing, parallelism, scalability, and fault-tolerance, and yet it will not be acceptable if it fails on the side of performance.

One of the most crucial performance related parameters is the message startup time. Future massively parallel systems must be capable of executing a message startup sequence -- preparing, issuing, accepting and queuing of data transfer requests, along with interrupt handling, context switching and process scheduling -- in the order of magnitude of 10 microseconds [12].

This paper deals with design aspects of message-passing kernels for high-performance, massively parallel distributed memory architectures. It presents first experiences made with the PEACE distributed operating system [19]. PEACE extended well-established operating systems principles [6] into the area of highly parallel distributed memory systems. The notion of program families [16] provided the framework for a system that caters to a variety user requirements. Members of the PEACE family consist of dedicated server processes that provide application-oriented services.

Basically, the PEACE approach is to use a small and efficient message passing kernel as foundation for all higher level system activities. In that respect PEACE is comparable to V [2] and Amoeba [14]. However, PEACE differs from the systems mentioned by a design that meets the specific needs of massively parallel systems. Highest consideration is given to the efficient implementation of the family of PEACE nuclei. All members have the same communication interface but different functionality and performance. The proper nucleus is then selected according to the application requirements.

The need for a message-passing nucleus family is evidenced by a case study that shows the potential disproportion between multi-tasking and communication.

2. Case Study Environment

2.1 SUPRENUM

SUPRENUM is a large scale MIMD supercomputer for numerical applications[4]. One of the unique features of this system is its two-level structure consisting of *clusters* and *nodes*. A cluster consists of 16 processing nodes (PN) and up to 4 service nodes (SN), interconnected by a parallel bus system with a transmission rate of 320 Mbytes per second. The clusters form the lower level of the hierarchy. At the higher level, up to 16 clusters can be interconnected by a torus (4-cube) structure. This SUPRENUM "core system" is illustrated in Figure 1.
Designed primarily for double-precision, floating-point array processing, the PN features in addition to the Motorola MC68020-based CPU a vector processor with a peak performance of 20 MFLOPS for two chained operations. A well-matched memory structure, consisting of 8 Mbytes of DRAM and two 64-Kbyte vector registers, results in a high sustained vector performance for sufficiently long vectors. Reflecting the

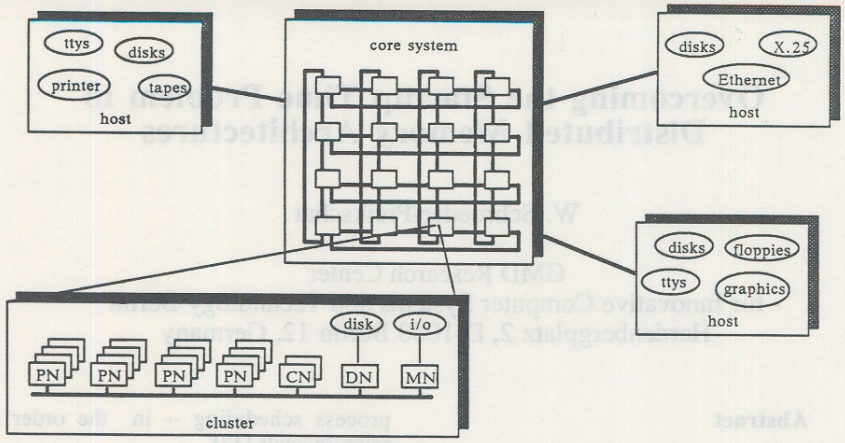

Figure 1: The SUPRENUM architecture

state of technology in 1986, the scalar performance, however, is rather weak. The MC68020 has little over 2 MIPS. Random access to the main memory has 3 wait states on read and 4 wait states on write. While this does not affect much the array processing performance of the node (in this case, the main memory works in the static column mode), it leads to relatively long execution times of sequential code (e.g., operating system routines).

Each cluster comprises 2 communication processors (CN) for interconnection with other clusters. Duplicating the CN in a cluster doubles the transmission bandwidth and makes the interconnection fault tolerant. Each CN has two physical links, a link being a 125 Mbits per second bit-serial token ring connection. The physical peak transfer rate between a pair of communicating nodes is 40 Mbytes per second. Inter-cluster communication via the parallel bus system and intra-cluster communication via the links use the same fault-tolerant worm hole routing protocol. The fact that the inter-cluster interconnection has a lesser bandwidth than the intra-cluster inter-connection favors to some extent cluster locality of the application algorithms.

2.2 PEACE

PEACE has been designed to manage the operations of the SUPRENUM Core System. The PEACE process model distinguishes between lightweight processes (*threads*) and heavyweight processes (*tasks*). Multiple threads of control that share the same address space and scheduling domain are called *teams*. Teams are used to implement tasks and system services.

As indicated in Figure 2, PEACE consists of three major building blocks: *nucleus*, *kernel*, and *distributed operating system*. A more detailed description can be found in [1].

Most of PEACE is executed in non-privileged user mode and defines an application-oriented distributed operating system. This component is constituted by a multitude of server teams that are site-independent and, thus, may be arbitrarily distributed over the parallel machine. The collection of teams provides typical services like naming, file handling, process management, memory management, exception handling, internetworking, loading, etc.

The distributed operating system is based on a kernel that implements an abstract PEACE machine. The kernel encapsulates all hardware-dependent functions. Thus, it contains device drivers, the MMU driver, and the trap and interrupt handler. In addition, the kernel provides services such as creating processes dynamically and associate them with system-wide unique identifiers and address spaces. Because of its proximity to the hardware, the kernel executes in supervisor mode. Internally, the kernel is represented as a multi-threaded team.

Services provided by the kernel and the distributed operating system are invoked by remote procedure calls [15]. This supports a system configuration that manages the parallel machine as a processor bank, thus associating individual nodes exclusively to user tasks.

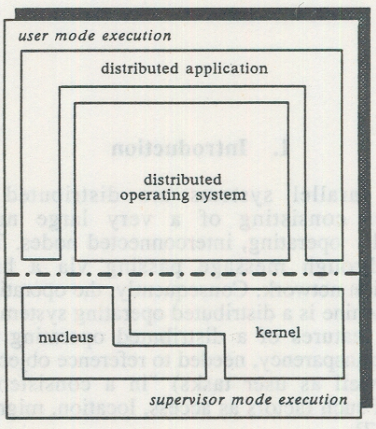

Figure 2: PEACE building blocks

The most basic component is the nucleus, whose sole function is to enable system-wide communication between tasks (teams or threads). Services of the nucleus are also invoked by remote procedure calls, which are handled in the usual way of system calls, namely as local traps. The nucleus is also directly used by communicating tasks of the distributed application.

3. Rationale for the Family Concept

The primary requirement of massively parallel systems is very fast, system-wide inter-process communication. As pointed out in [9], communication performance is less a problem of limited network bandwidth. Rather, it is a problem of system software overhead caused by node-bounded activities such as interrupt handling, buffering, context switching, scheduling and so on. A detailed analysis of typical message-passing kernels such as PEACE reveals that these activities are not necessarily required for communication in a distributed environment. Rather, they are only required if one extends traditional multi-tasking into the distributed environment.

In addition to improving processor utilization in general, multi-tasking is a functionality that introduces scaling transparency into parallel, distributed applications. However, in cases where the application scales well with the actual number of nodes, it introduces nothing else but overhead and, thus, slows down overall application performance. Therefore, the main problem in the design and development of operating systems for massively parallel systems is to avoid that "artificial" bottleneck, without sacrificing multi-tasking all together. Whether multi-tasking is really needed depends on the application and on the actual number of nodes available for execution. As a general guideline, applications should be structured so that there is exactly one task per node. This constitutes the majority of parallel (numerical) programs.

3.1 PEACE Nucleus Benchmarking

A multi-tasking PEACE nucleus has been benchmarked in a scenario where each node executes only a single task. Two versions of the nucleus were tested: one with non-protected and one with protected address spaces. All benchmarks were executed 100000 times; from this the time for the single benchmark operation was computed.

At the time of writing this paper, a pure single-tasking member of the family of nuclei is not yet available. Therefore, internal functions of the multi-tasking nucleus have been isolated and individually benchmarked to determine pure multi-tasking overhead. These functions will not be present in a single-tasking nucleus. They typically deal with process scheduling and dispatching and interrupt handling.

A SUPRENUM hardware platform consisting of two PNs has been employed for the benchmarking. Both PNs resided in the same cluster. Note that multi-tasking implies node-bounded activities. This makes it irrelevant whether to use PNs from the same or from different clusters. In either case the same physical network interface, the same protocol and, thus, the same low-level communication software is used in SUPRENUM.

3.1.1 Rendezvous Time Parameters. Usually, a PEACE rendezvous means the exchange of a send-receive-reply sequence between a client and a server. In order to support remote procedure calls most efficiently, the PEACE nucleus provides an alternative interface to servers. This interface combines the corresponding receive and reply into a single operation called *replace*, meaning to replace a rendezvous (remote procedure call) with another one. The mechanization first issues a reply and immediately afterwards performs a receive. Table 1 lists the timing for the PEACE rendezvous based on send-replace sequences.

configuration	time (μ sec.)		
	local		*remote*
	without MMU	*with MMU*	*with MMU*
1 - to - 1	360	420	784
2 - to - 1	312	361	537
4 - to - 1	284	331	525
8 - to - 1	270	315	525
16 - to - 1	262	305	525

Table 1: Rendezvous time parameters

Since the inter-process communication of PEACE is based on transferring fixed-size message packets of 64 bytes, 128 bytes are exchanged with each rendezvous. These packets are buffered by the nucleus, resulting in two copy operations both at the client and the server site (one copy in/out with each send and replace).

The transfer of arbitrary sized messages is supported by a separate mechanism called *high-volume data transfer*. This mechanism allows for the system-wide exchange of memory segments without intermediate buffering. It is supported by "intelligent" DMA hardware in each node. The critical factor for the user-level message startup time therefore consists in the rendezvous. Once established, high-volume data transfer takes place.

For the case of the local rendezvous, the performance figures of a nucleus with and without address space protection are shown. Address space protection is provided by the memory management unit (MMU). The different timing parameters show the MMU overhead in the process of copying data (arguments and message packets) between nucleus and user space. The same overhead exists in the case of the remote rendezvous. Note that no task context switches took place, for only a single task was executed on a node.

A many-to-one relation between clients and server has been investigated. Up to 16 client threads issued a rendezvous request to a single server. Hence, the amount of overhead caused by server scheduling/dispatching and interrupt handling at the server site was revealed. Situations with more than one client were represented by a multi-threaded client team, whereas the server team was always single-threaded.
In the case of having n>1 client threads requesting simultaneously a rendezvous, the resulting benchmark performance was divided by n. Hence, the timing measurements for each configuration have been normalized to a single rendezvous.

3.1.2 Basic Time Consumed. For benchmarking, dedicated nucleus instances were generated. The nucleus call interface was excluded. Table 2 shows the results.

action		time (μ sec.)	
		with MMU	*without MMU*
nucleus call		21	19
message buffering		49	43
segment validation		255	0
SUPRENUM housekeeping		32	32
interrupt latency		88	86
team scheduling		68	68
dispatching	*thread*	38	38
	team	56	56
switching	*thread*	21	21
	team	40	25

Table 2: Basic time consumed

Nucleus call overhead simply is the time it takes to request and deliver the process identification of the calling thread. It encompasses trap handling, monitor locking/unlocking, and argument passing between user space and supervisor (nucleus) space.

Segment validation is performed only as part of high-volume data transfers. This means to verify the validity of a given logical address with respect to the team address space that is involved in the data transfer. *Housekeeping* pertains to the locking and unlocking of the vector floating point unit in the case of task switching.

Team scheduling only takes place if several teams (tasks) need be executed by the node. Preemptive and timer-driven scheduling is performed. The time listed encompasses team switching, i.e. a complete context (address space) switch. Thread switching is concerned only with swapping CPU register sets. The difference between the times for team scheduling and team switching indicates the raw team scheduler overhead. Thread and team dispatching gives the raw overhead needed for ready list manipulation. Two operations are involved in a single dispatching activity, viz. setting a thread/team ready to run and selecting the next thread/team for execution.

Interrupt latency is the time needed to propagate hardware interrupts to threads. It encompasses the saving and restoring of CPU registers, interrupt handler invocation and return, and synchronization. The complete sequence is executed for remote thread communication.

3.1.3 Analysis. A remote PEACE rendezvous between two tasks based on *send-replace* sequences implies at least two nucleus calls, four message buffering activities, two team dispatchings, and two housekeeping activities. In this model, team switching, i.e., complete task context switches, are omitted because only a single task is executed per node. If a task is required to block, it enters the nucleus idle loop and starts polling on external events such as incoming messages.

In a one-to-one rendezvous the interrupt latency caused by the receive of messages is encountered twice. Thus, a total of 502 microseconds with MMU and 472 microseconds without MMU has been encounted for the individual rendezvous. Considering the entire rendezvous time of 784 microseconds, approximately 64 percent of it are caused by node-bounded activities. It should be noted that this portion means pure overhead for applications which have only a single task executed by the multi-tasking PEACE nucleus. None of the basic parameters considered above will be relevant for a single-tasking nucleus.

In a two-to-one rendezvous, a performance gain of 247 microseconds has been obtained for the DMA mode of operation. With higher load at the server site, i.e., with an increasing number of clients that are sending to the same server, a speed-up of the individual rendezvous was observed. In this scenario, the server is always busy because of pending client messages, i.e., its message queue will never be empty. Therefore, the server does not block on *replace*, which would happen in the case of an empty message queue. This means that neither scheduling/dispatching nor housekeeping activities must be performed, and that the server never enters the idle loop. Similar considerations hold for the case of local rendezvous.

3.2 Lessons Learned

The analysis confirms our conjecture [18] that an increase of network bandwidth and interface hardware performance alone is not the answer to the performance bottleneck problem. It also shows that a faster host [9] is not the ultima ratio either. Rather, devising more efficient software solutions is as important as the speed-up of communication hardware.

The PEACE performance figures discussed above are very impressive, especially when comparing them with the Amoeba nucleus, which also protects address spaces [22]. Nevertheless, the performance is not good enough for massively parallel systems. Multi-tasking functionality creates an unnecessary performance bottleneck in all cases where applications do not require it.

In the scope of parallel computing there often are applications which scale well with the actual number of nodes and, thus, make multi-tasking superfluous. On the other hand, for reasons of fault tolerance and multi-user support, multi-tasking should not be totally sacrificed either. Rather, a solution must be found that lets the application pay only for those functions that are really required.

3.2.1 The Family Approach. Basically, the program family concept [16] means to identify a *minimal subset* as well as *minimal extensions* of system functions during the software design process. The minimal subset constitutes a common abstraction to all higher-level software modules. Ideally, it encapsulates solely the *mechanisms* and not the *policies*. Policies should be introduced as minimal extensions. This leads to the step-wise functional enrichment of the system. Remarkably, this bottom-up construction is controlled in a top-down fashion: lower-level modules are introduced only when required by higher-level modules. Applying this approach recursively leads to a hierarchical system organization. The *uses relation* [16] between the modules is an important instrument for associating modules to levels in the hierarchical system. Consequently, this leads to a pure application-oriented system structure, in which only those system functions (i.e., modules) are defined that are required by a given application.

Originally, this concept has been devised to improve maintainability of very large and complex software systems, e.g., operating systems. A program family design naturally results in a highly modular system structure, with each module being an abstract data type [11]. It helps to concentrate on the pertinent facts during system development while providing a flexible framework for incremental construction, selective adoption, and future extension of any kind of software.

3.2.2 Consequences for PEACE. The family concept is used in PEACE to realize an application-oriented distributed operating system. Teams represent abstract data types which implement system services. They are loaded on demand, when the application needing these services is loaded.

The application-oriented family concept is to be applied to the nucleus design, in order to provide a well-defined framework for different instances of the nucleus. Presently, the nucleus family of PEACE consists only of the two nucleus instances discussed so far. This is because scaling transparency was considered the governing user requirement in SUPRENUM. As demonstrated by the case study, however, a more sophisticated family organization is required. There is a need for members ranging from a single-threaded communication library to a complete multi-tasking and multi-user implementation.

In this sense, the PEACE nucleus provides the minimal subset of system functions, whereas kernel and distributed operating system provide minimal extensions. The difference between kernel and distributed operating system is that the kernel adds mostly mechanisms, whereas the distributed operating system introduces policies. The purpose of having a nucleus family is to have the minimal subset of system functions represented in an optimal fashion.

4. The Nucleus Family

For the design of a nucleus family for massively parallel systems, the distinction between single-tasking and multi-tasking is too rigid. The PEACE development demonstrates that a more discriminating approach must be taken. One reason is that a large class of applications must be optimally supported. A second reason is that it is much less risky to develop an operating system incrementally rather than trying to implement the entire system in one shot.

In this sense, a multi-tasking nucleus will begin with supporting static scheduling, to be extended by preemptive scheduling and address space isolation, by memory protection and, finally, by multi-user security. The key aspect is that the nucleus design is required to be complete before implementation starts. All the intermediate steps towards the final implementation are considered as representing an autonomous member of the nucleus family in its own right, exhibiting different functionality and performance characteristics. Figure 3 illustrates this approach by means of a nucleus family tree. As is discussed below, nucleus functionality increases from left to right and from top to bottom. By the same amount by which the functionality is increased, the communication performance drops, i.e. the message startup time increases.

4.1 The Family Tree

At the root of the family tree there are two major user requirements, namely single-tasking and multi-tasking. In case of single-tasking, a distinction is made into single-threading and multi-threading. Note, a task is a PEACE team and hence may be multi-threaded. In the case that only a single thread of con-

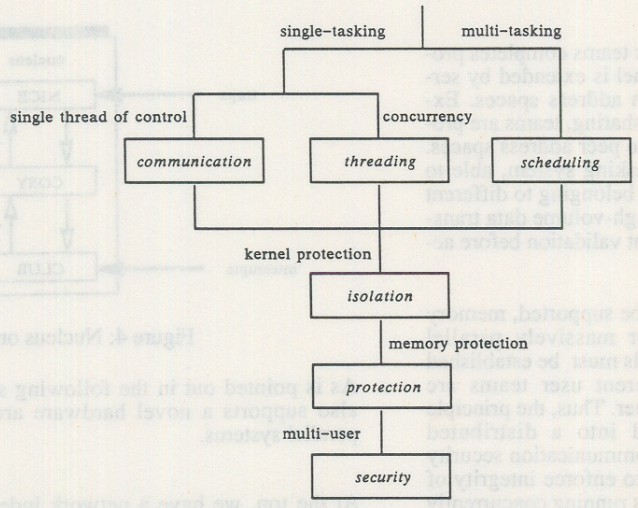

Figure 3: PEACE nucleus family tree

trol must be supported on a node, the nucleus purely implements communication. This is the situation where a one-to-one mapping between a node and a user task is feasible. In PEACE, synchronous inter-process communication is considered as the fundamental communication paradigm, offering the highest possible end-to-end communication performance. Threading introduces concurrency relative to a common address space, i.e. a team. The need for multi-tasking at this stage introduces concurrency relative to several teams belonging to the same user application. Thus, a non-preemptive scheduling strategy is introduced into the nucleus. This constitutes a switch from one-level to two-level scheduling, i.e. it extends thread scheduling by team scheduling.

In all these cases, the nucleus takes the form of a non-protected shared library that is directly linked to the application task. In addition, a node controlled by these nucleus instances is used exclusively for user task execution. Except for the nucleus, no other PEACE system components are located on these nodes. Application tasks (teams) are loaded as part of the node bootstrap procedure. Following the pattern of coroutines, merely dynamic creation/destruction of threads is supported. Since system services are invoked by means of remote procedure calls, PEACE server teams may reside elsewhere and are not required to share a single node with the user teams (i.e. tasks).

The requirement for preemptive scheduling implies a complete nucleus isolation. Now, interrupts must be handled, and the teams are given a limited time quantum for program execution. Dynamic creation/destruction of teams is introduced. However, team address spaces are not yet isolated, i.e., protected. In addition, the nucleus must be invoked through traps, thus introducing more overhead. Consequent, the nucleus as part of the kernel is executed in privileged supervisor mode and protected against user mode tasks. Nucleus (kernel) protection may also be a case for the single-threading and multi-threading instances.

Applying address space isolation to user teams completes protection in its traditional sense. The kernel is extended by services to associate process objects with address spaces. Excepted for explicitly requested segment sharing, teams are prohibited to directly access and manipulate peer address spaces. This makes the nucleus a true multi-tasking system, able to support the execution of multiple teams belonging to different applications. In this nucleus instance, high-volume data transfer, for example, implies explicit segment validation before accessing a team address space.

If a multi-user mode of execution is to be supported, memory protection alone is not sufficient for massively parallel systems. Rather, communication firewalls must be established such that threads belonging to different user teams are prohibited to communicate with each other. Thus, the principle of memory protection is extended into a distributed environment. Precautions concerning communication security are necessary. The nucleus is required to enforce integrity of the different distributed user applications running concurrently on the parallel machine.

4.2 Problem Orientation

The order in which we discussed the members of the nucleus family also defines the order of increasing system functionality. In the same order user-level communication performance drops. Applications which scale well with the given number of nodes are supported by either of the first two nucleus instances (communication and threading), depending on whether synchronous or asynchronous inter-process communication is required. A first step towards multi-tasking is to introduce the scheduling nucleus. This nucleus represents a compromise between scalability and protection. It is used for dedicated and mature applications.

A similar compromise can be made concering nucleus isolation. However, there are two additional major issues to be addressed by the multi-tasking nucleus. First, dynamic nucleus reconfiguration must be feasible. Second, multi-tasking must be supported regradless of the availability of a MMU. This generally enlarges the applicability of the system.

Dynamic nucleus reconfiguration is needed in cases where system availability is the dominating issue. A single-tasking nucleus is used in all cases where tasks are mapped one-to-one to nodes. Nodes crashes, however, may result in switching to a multi-user multi-tasking nucleus. In order to keep the application(s) running, task redistribution may be required. This also may imply that some nodes are now to be shared between a number of tasks belonging to different user applications. In this case, the single-tasking nucleus need be replaced by the multi-tasking nucleus at run time.

5. Internal Nucleus Structure

Common to all the nucleus instances discussed above is the communication aspect. Additional functions are minimal extensions to a minimal subset of communication system functions. The actual structure of a nucleus instance reflectsthis aspect. How the internal organization of the PEACE nucleus looks like is illustrated in Figure 4.

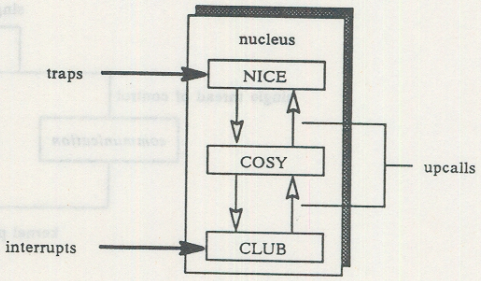

Figure 4: Nucleus organization

As is pointed out in the following section, this organization also supports a novel hardware architecture for massively parallel systems.

At the top, we have a network independent communication executive, called NICE. This module implements system-wide synchronous inter-process communication, while asynchronous communication is supported by means of threads. The module encapsulates the different function of the nucleus and manages rendezvous, threads, teams, and so on. That is, the nucleus family actually is a NICE family, and all members of this family inherit the same NICE interface. The two layers below NICE form the communication system, whose purpose

is to exchange data streams between nodes in the most efficient manner. Rather than addressing processes (i.e. threads and teams), address space segments serve as origin and target for the message transfer.

The major task of the communication system (COSY) is to execute the proper communication protocol that copes with the problems given by the underlying network. E.g., depending on the type of network it performs segmenting/blocking of messages and introduces the view of virtual channels. The interface between NICE and COSY is asynchronous. In the downward direction it uses request queues, whereas the upward direction is implemented by upcalls [3].

In order to keep most of the nucleus portable, a CLUB layer introduces an abstraction of the physical network interfaces. The CLUB layer encapsulates the lowlevel device drivers that control the network interface. It maps logical node addresses associated with threads onto physical node addresses as required by the hardware. Generally, it combines a group of related nodes into a club of homogeneous network hardware interfaces.

Again, the interface between COSY and CLUB is asynchronous, implemented by a request queue in the downward direction and upcalls in the upward direction. The CLUB hardware interface is assumed to be interrupt driven. Upcalls (to COSY and NICE) may be directly triggered by hardware interrupts.

With respect to different user requirements, different NICE modules are used for optimal support. Different network characteristics are covered by dedicated COSY modules, improving the portability of NICE by means of isolation from network details. Network hardware transparency is achieved by CLUB, i.e., there are different CLUB implementations for different network hardware interfaces. This generally improves nucleus portability by means of isolating them from the details of network devices.

6. Thoughts on Hardware Support

So far, a pure software solution to the communication performance bottleneck problem of massively parallel systems has been discussed. Further improvements may come from dedicated support hardware. A stand-alone hardware solution is not feasible: rather a symbiosis between software and hardware solutions is required. This symbiosis will exist in MANNA [5], a forthcoming massively parallel system based on experiences gained by the SUPRENUM development. In MANNA the PEACE nucleus family approach and a novel node hardware architecture are combined into a powerful multicomputer system. This symbiosis is illustrated in Figure 5.

The basic idea is to have COSY and CLUB completely executed by the dedicated communication processer (CP), while the application processor (AP) executes the application tasks. Both processors jointly constitute a single node and are communicating through shared-memory. Several hundreds to thousands of such nodes will then constitute the MANNA system.

The processing of NICE is shared by both processors. This is based on the fact that NICE knows about tasks (in the form of teams) and, therefore, is responsible for exchanging the AP register set on performing a task context switch. The manipulation of ready lists for scheduling purposes, however, is done jointly by both the AP and the CP. In the case of incoming messages, for example, the CP directly dispatches the target thread by placing it onto the AP ready list. This way, interrupt overhead at the AP site caused by message-passing is avoided.

In order to interface AP and CP, a GLUE layer is required for interconnecting NICE and COSY properly. By means of GLUE, user tasks are directly interfaced to the CP without the need to trap via NICE. Because of the two distinguished processors, communication and computation of the same task can

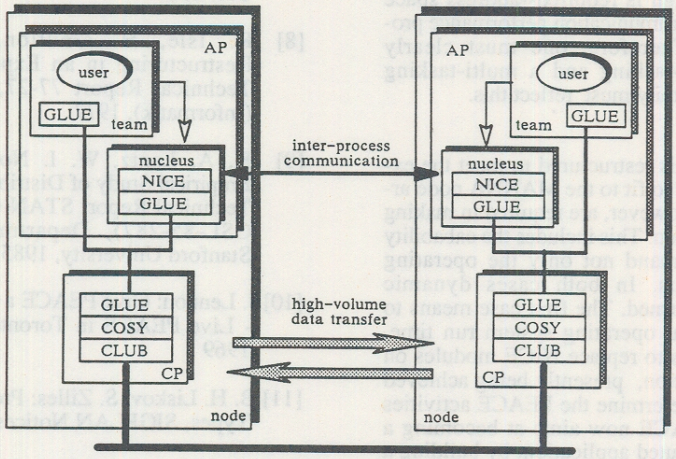

Figure 5: Functional dedicated nucleus representation

be performed in parallel, whereas overlapping by means of threading would always require AP multiplexing.

Typically, upon issuing a message-passing operation the PEACE nucleus (on behalf of NICE) queues new data transfer requests into the COSY interface. Using GLUE, this queuing is made available to user tasks. Hence, asynchronous communication is directly supported by hardware. Upon processing the requests, the CP (on behalf of COSY) performs high-volume data transfer between peer AP (team) address spaces.

The additional overhead implied by the GLUE layers is determined by the complexity of queue operations in a shared-memory multi-processor system. For example, in case of a MC68020 a CAS2 instruction will be used for indivisible queuing [13]. The benign case the critical section is not active then results in a queuing operation of less than 20 microseconds (assuming 20MHz CPU and 0 wait states memory). That is to say, the AP/CP software interface overhead compares to the nucleus call overhead measured for PEACE.

7. Conclusions

As often required by the user community, support for massively parallel systems in general means to make the naked machine available to the parallel application program in order to obtain maximum performance. In contrast to that, distribution transparency often is required too. Both aspects significantly influenced the PEACE development and led to the idea of a nucleus family. The family approach was chosen because there cannot be a single solution for a variety of user requirements if utmost communication performance is to be guaranteed.

For SUPRENUM, scaling transparency was a major user requirement. This led to the implementation of a nucleus family having only two members, providing protected and non-protected address spaces. Both members provide multi-tasking and were subject to benchmarking. The results show that a more sophisticated family design is required--address space protection is not the cause of communication performance problems but multi-tasking. Therefore, one must clearly distinguish between a single-tasking and a multi-tasking environment, and the nucleus family must reflect this.

Presently, the PEACE nucleus is restructured to meet the extended family requirements and to fit to the MANNA node architecture. The largest efforts, however, are required in making the family work for the application. This includes the cabability to extend and contract on demand not only the operating system, but also the nucleus. In both cases dynamic restructuring [8] must be performed. The first case means to introduce new system teams at operating system run time, whereas the second case means to replace NICE modules on the fly. Progress into this direction, presently being achieved in PEACE [17], will largely determine the PEACE activities for the near future. Thus, PEACE now aims at becoming a platform for any kind of distributed application, by building a dynamic alterable process execution and communication environment for distributed and/or massively parallel systems [10].

Acknowledgments

I wish to express my gratitude to all members of the PEACE project. Especially to Joerg Nolte and Michael Sander for discussions on the internal nucleus structure and to Henning Schmidt for careful reading early manuscripts of the paper. I am also very grateful to Prof. Giloi. His effort and advice significantly improved the quality of the paper. Last but not least, many thanks to Ingrid Arnsburg, Gaby Tysper and Werner Raabe for typing and formatting the final version of the paper.

References

[1] R. Berg, J. Cordsen, C. Hastedt, J. Heuer, J. Nolte, M. Sander, H. Schmidt, F. Schoen, W. Schroeder-Preikschat: Making Massively Parallel Systems Work, ESPRIT Project No. 2702, Technical Report, GMD FIRST, Berlin, FRG, 1990

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

[3] D. D. Clark: The Structuring of Systems Using Upcalls, ACM Operating Systems Review, 19, 5, Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985

[4] W. K. Giloi: The SUPRENUM Architecture, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[5] W. K. Giloi: Entwurf und Erprobung einer universellen, massiv - parallelen Rechnerarchitektur für nicht - numerische Anwendung, BMFT Forschungsvorhaben ITR9002 2, FRG, 1990

[6] A. N. Habermann, L. Flon, L. Cooprider: Modularization and Hierarchy in a Family of Operating Systems, Comm. ACM, 19, 5, 266-272, 1976

[7] A. J. Herbert, J. Monk: ANSA Reference Manual, Advanced Network Systems Architecture, Cambridge, UK, 1987

[8] R. Isle, H. Goullon, K.-P. Loehr: Dynamic Restructuring in an Experimental Operating System, Technical Report 77-27, TU Berlin, Fachbereich 20 (Informatik), 1977

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

[10] J. Lennon: Give PEACE a Chance, The Plastic Ono Band - Live PEACE in Toronto, Apple Records, December, 1969

[11] B. H. Liskov, S. Zilles: Programming with Abstract Data Types, SIGPLAN Notices, 9, 4, 1974

[12] H. Mierendorff: Bounds on the Startup Time for the GENESIS Node, ESPRIT Project No. 2447, Internal Paper, GMD-F2.G1, 1989

[13] Motorola: MC68020 32-Bit Microprocessor User's Manual, Second Edition, Prentice Hall, Inc., 1985

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

[15] B. J. Nelson: Remote Procedure Call, Carnegie-Mellon University, Report CMU-CS-81-119, 1982

[16] D. L. Parnas: Designing Software for Ease of Extension and Contraction, IEEE Transaction on Software Engineering, Vol. SE-5, No 2, 1979

[17] H. Schmidt: Making PEACE a Dynamic Alterable System, GMD FIRST, 1990

[18] W. Schroeder: A Distributed Process Execution and Communication Environment for High-Performance Application Systems, Lecture Notes in Computer Science, Vol. 309 (1988), 162-188, Springer-Verlag, Proceedings of the International Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), Sept. 28-30, 1987

[19] W. Schroeder: The Distributed PEACE Operating System and its Suitability for MIMD Message-Passing Architectures, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[20] W. Schroeder-Preikschat, M. Gien: Architecture and Rationale of the GENESIS Family of Distributed Operating System, ESPRIT Project No. 2447, Technical Report, GMD FIRST, Berlin, FRG, 1989

[21] A. S. Tanenbaum, R. van Renesse: Distributed Operating Systems, ACM Computing Surveys, Vol. 17, No. 4 (December), 1985

[22] R. van Renesse, H. van Staveren, A. S. Tanenbaum: Performance of the World's Fastest Distributed Operating System, ACM Operating Systems Review, 22, 4, 1988