The Architecture of the European MIMD Supercomputer GENESIS[*]

by

U. Bruening, W.K. Giloi, and W. Schroeder-Preikschat

GMD Research Center for Innocative Computer Systems and Technology
at the Technical University of Berlin
Hardenbergplatz 2, D-1000 Berlin 12, e-mail: giloi@gmdtub.uucp

Abstract

GENESIS is a European supercomputer development project funded by the European Commission within the ESPRIT-II program. The aim of the GENESIS project is to render a contribution to the world-wide efforts concerning the development of the next generation of scalable parallel computers of the highest possible performance and cost-effectiveness. Scalability means that the whole spectrum of parallel computers, ranging from super workstations with several hundred MFLOPS to supercomputers with several hundred GFLOPS, can be realized with the same hardware and software components. This requirement leads to a distributed memory architecture. The performance requirement calls for a peak performance of the single node of at least 100 MFLOPS. The performance of the interconnect through which the nodes communicate must match the node performance. In order to obtain competitive products, the specific cost should not exceed $200/MFLOPS in the next couple of years and $100/MFLOPS in the mid-nineties. The paper discusses the technological and architectural measures by which these goals can be accomplished. Based on the rationale presented, the design decisions concerning the node architecture, the interconnection network, the protocol structure, and the operating system structure and functionality are discussed. The paper also addresses the issues of application software models and supporting tools. Finally, the Virtual Shared Memory Architecture approach is briefly outlined. This approach reconciles the advantages of the distributed memory architecture with a conventional programming style.

1 INTRODUCTION

Since the mid seventies supercomputers were synonymous with vector machines. The ability of the pipelined vector processor to execute with every clock tick one or even two or three (chained) floating point operations, in combination with an extremely high clock frequency, accounted for a peak performance of several hundred MFLOPS at a time when the largest mainframes delivered less than 10 MIPS per (scalar) processor.

---

[*] This work was partly sponsored by the European Commission, ESPRIT grant no. P2702

In the meantime, the more cost-effective alternative to the vector machine has become the MIMD architecture consisting of a large number (some hundreds to thousands) of nodes. First attempts in this direction have been the iSPC2 of Intel Scientific [1] with up to 128 nodes and a vector peak performance of 10 MFLOPS per node, or the SUPRENUM [2] with up to 256 nodes and a vector peak performance of 20 MFLOPS (double precision) per node. The SUPRENUM with its peak performance of 5 GFLOPS outperforms both the CRAY-Y MP and the CRAY-2.

A large number of nodes rules out memory sharing as the means of communication between the nodes, because the shared memory would create untolerable ´hot spots` of memory access conflicts. To avoid that problem, large MIMD architectures must be *distributed memory architectures*, i.e., systems which have no global memory but only the local node memories. In a distributed memory architecture each node executes a number of cooperating processes that communicate through message-passing with each other, as well as with processes in other nodes . To this end, the nodes are interconnected by an *interconnection network*, and an appropriate protocol hierarchy must exist to ensure an efficient and yet secure message passing.

Unlike shared memory architectures, distributed memory architectures are highly scalable. Scalability means that with the same hardware and system software one can arbitrarily configure small, medium size, or large systems and, thus, cover the entire range from super workstations to supercomputers of the highest performance. Table 1 characterizes three product classes and the peak performance which they can be expected to have. The lower performance figures pertain to about 1992 and the higher one to the situation three years later. Of course, the supercomputer may have a size smaller or larger than the 1024 nodes listed (a practical limit would probably be reached with 4096 nodes).

Table 1 Product classes and their performance

System Type	Performance Range	Packaging
Super Workstation	0.1...0.4 GFLOPS	1...4 processors single board tower enclosure
Super CAD System	0.8...6.4 GFLOPS	single cluster (up to 16 nodes) cabinet enclosure
Super Computer	100...400 GFLOPS	1024 nodes

The nodes of a highly parallel MIMD architecture are realized in low-power VLSI technology, i.e., CMOS or BiCMOS. This allows a complete node, including many Mbytes of memory, to be accommodated on a single circuit board and cooled by forced air (e.g.: the 20 MFLOPS SUPRENUM node with 8 Mbytes of EDC-DRAM and 128 Kbytes of SRAM vector memory is a 19"x10" board that consumes only 60 watts [3]). Consequently, the cost of packaging and cooling and, thus, the specific cost (dollars per MEGAFLOPS) are relatively low. For example, the specific cost of the SUPRENUM is only one third of that of the CRAY-Y MP. For future designs, however, even this is much too high. For a system that is under design now and scheduled to be marketed in the 1991/92 time frame, the specific cost should be brought down to about \$200/MFLOPS, in order to be competitive. For the mid-nineties we anticipate specific cost around \$100/MFLOPS. In a scientific "number cruncher," MFLOPS pertains to IEEE standard double precision.

Until 1989, the only way to obtain an acceptable peak performance from a node computer was to *vectorize* it by supplementing its CPU by a (pipelined) *vector processor*. Consequently, the architecture of such a node is quite similar -- at a much smaller scale -- to that of a vector machine. The current state of the art for off-the-shelf floating-point processor components is a clock frequency of 33 MHz, resulting in 66 MFLOPS of peak performance for two chained operations (double precision) [4]. In a year or two, such processors will run at a clock speed of 50...100 MHz for BiCMOS or ECL devices, respectively.

However, *pipelined scalar processors* have recently evolved that are equally powerful.[1] An example is the Intel i870 (alias N11), whose 50 MHz version will eventually offer a peak performance of 100 MFLOPS for two chained operations. Compared to node vectorization, the scalar solution is considerably cheaper and requires much less development efforts. On the other hand, this new brand of pipelined scalar processor mandate highly "intelligent," optimizing compilers which go way beyond the global optimizations performed by the best compilers presently existing. The first existing compiler for the i860, for instance, leads to a *Linpack benchmark* of about 3 MFLOPS (double precision, 33 MHz version). On the other hand, the same processor can perform the hand-coded inner product algorithms at a rate of 27 MFLOPS. In a hand-coded ray tracing algorithm [5], 50 MFLOPS of single precision have been reached.

The problem with the pipelined scalar processors is that it becomes the compiler´s responsibility to keep the pipeline filled through elaborate instruction rescheduling. Compilers for the i860 or i870 which can do that, i.e., which will generate code nearly as good as by assembler code, are still a couple of years away. Consequently, the incentive for node vectorization is still given, since in the vector case the compiler issue is uncritical. Writing highly optimizing *vectorizing compilers* has become a well established art.

The highly parallel MIMD systems which we envision as the future supercomputer architecture will exhibit a high degree of hardware and software standardization, enforced by the wide-spread use of a new generation of very high performance 64-bit processors, as well as the system software (operating system, compilers) that comes with it. Therefore, products of that type of different vendors will differ not so much in the node architecture but in such areas as specific communication hardware in the node, interconnection network architecture, software tools for program development, and application software packages.

The adequate granularity of programming distributed memory architectures is given by *communicating processes*. To this end, an application program must be partitioned into at least as many processes as the system has nodes. Currently this must be done by the programmer, who may be assisted by some interactive

[1] These processors are usually called a "RISC," which in most cases is a misnomer.

software tools. It is expected that fully automated "parallelizers" [6] will still take a long time to evolve.

The cooperating processes in the system communicate on the basis of an appropriate *inter process communication* (IPC) protocol. Communication is either explicitly programmed by the application programmer, in which case the programming language must offer the appropriate constructs, or it is provided by preprogrammed communication routines available from a library. Data objects are encapsulated into the processes that own them. Hence, we recognize the model of *object-oriented programming*. This modern programming style is a major research issue in computer science; however, it hardly plays a role in the practice of software production for numerical applications. Rather, the predominant programming style is still the imperative, *von Neumann type* of programming, carried out in such langues as Fortran, Lisp, Ada, or C, i.e., languages that require the view of a global address space. Unlike distributed memory architecture, shared memory architecture support the conventional programing style.

This paper presents an overview of the GENESIS design which is based upon the above considerations. GENESIS can be viewed as the European equivalent of the *Touchstone* Project in the USA. The GENESIS design aims at providing an optimal solution in terms of absolute performance, cost-effectiveness, and ease of use. To obtain a high cost-effectiveness, GENESIS is realized in low-power VLSI technology, using the least expensive, high density packaging available and as many off-the-shelf components as possible.

In Section 2 we shall outline the GENESIS node design and its rationale. Prototypes of that node are up and running. Section 3 deals with the GENESIS interconnection structure whose underlying concepts and performance is discussed. Section 4 describes the construction principles and the functionality of the node operating system, PEACE. Special emphasize is put on the issues of scalability and communication efficiency. Section 5 deals briefly with the issues of the GENESIS programming models and parallelization, as well as programming tools and environments. It also shows to what extent a Virtual Shared Memory approach will be supported by the GENESIS architecture.

2 THE GENESIS NODE ARCHITECTURE

2.1 The GENESIS Node Version.1

In the first step of the GENESIS development, we designed and realized a scalar node, the block diagram of which is shown in Figure 1. We recognize that the GENESIS node has two processors: In addition to the node CPU proper, called *Application Processor* (AP), there is a dedicated *Communication Processor* (CP). In addition there is a local communication interface (LCI), connecting AP and CP, and a network link interface (NLI) connecting the node with the interconnection network. The node is used as the basic building block for a highly scalable distributed memory system. Thefeore, thed node design comprises innovative concepts for supporting message passing.

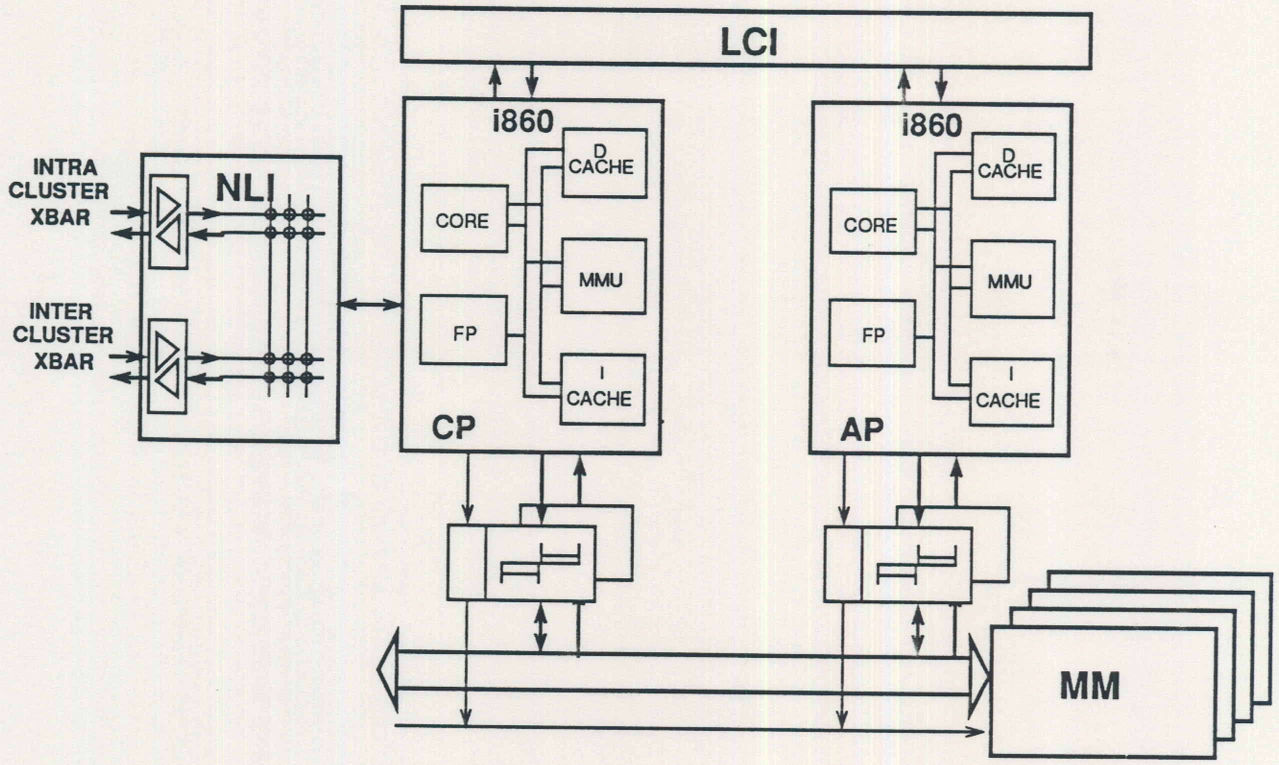


Figure 1 GENESIS Node Version.1 -- Block diagram

The realization of the AP is based upon the use of the most powerful existing microprocessor, the Intel i870. The i870 will feature a peak performance of 50 MIPS and 100 MFLOPS, respectively, and a memory transfer rate of 320 Mbytes per second. The memory interface consists of a 3-stage pipeline, the main memory consists of 4 banks of DRAM with 8 Mbytes each.

The CP works with the i870 processor too. It communicates with the AP via the LCI for the purpose of instruction issuing and synchronization. Furthermore, it shares the node memory with the AP. The CP controls the NLI, transports messages, and implements the message-passing protocol. Specifically, the CP meets the following requirements:

- it has the same logical view of memory as the AP, i.e., the same MMU functions and main memory access modes.;
- it performs the high speed data transport by performing the address computations needed and controlling the appropriate devices;
- it supports the high-level language implementation of the protocols.

The LCI has been designed for maximum efficiency of communication between AP and CP. The LCI transfers message passing instructions from the AP to the CP and acknowledgements back from the CP to the AP. Its hardware will eventually be realized by an application specific integrated circuit (ASIC).

The block diagram of the NLI is depicted in Figure 2. It provides 4 byte-serial, bi-directional links, i.e., 8 uni-directional lines, whose data rate in a first version will be 50 Mbytes per second each. The links include all necessary logic for wormhole routing. Its hardware will eventually be realized by an application specific integrated circuit (ASIC).

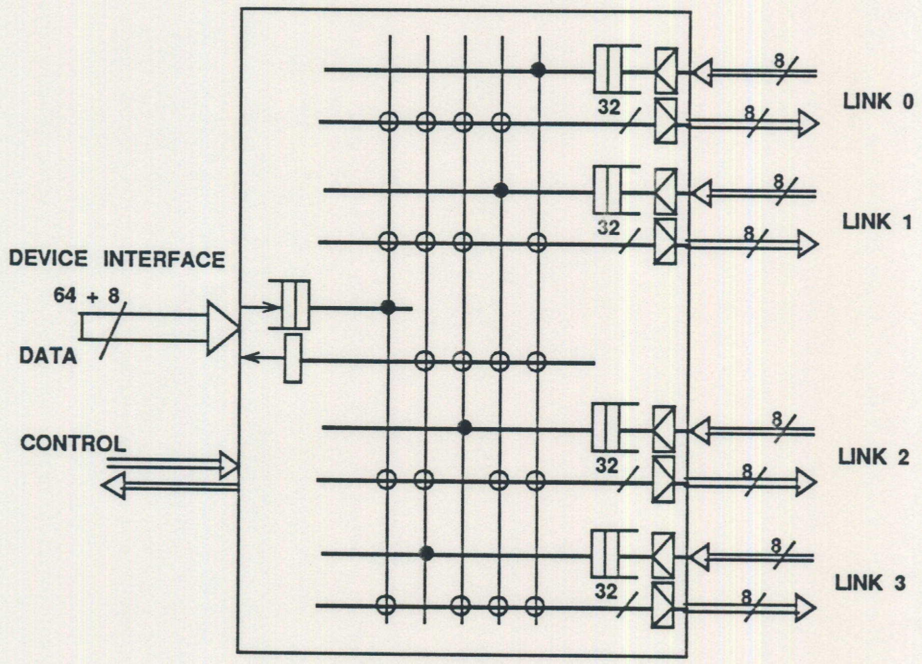


Figure 2 Network Link Interface -- Block diagram

The existence of a dedicated CP greatly enhances the efficiency of the AP [7]. All the AP must do when it encounters a *send* command is to put a *send message* into the send queue. The CP, which is polling that queue, executes the send operation, while the AP can go on with its work. Conversely, when the CP receives a *receive message* from the CP of another node, it puts it into the *receive queue*. The AP, on executing a *receive* command, goes to that queue and looks for the corresponding message. This has the following advantages:

- AP and CP work concurrently;
- in a single task environment, the AP works without interrupts;
- in a multi task environment, the AP works without interrupts or environment switches.

Thus, the communication startup time can be reduced by an order of magnitude [7]. Adding the CP offers additional advantages, viz.:

- A high-MIPS CP can very efficiently emulate the address generator needed to implement the high volume data transfer scheme described in Section 4.

- In a virtual shared memory architecture, the CP can also perform the functions of the *page manager* and *concurrency controller* of the node.

In all these cases, the AP is freed to concentrate its work on doing the computations, i.e., producing MEGAFLOPS.

2.1 The GENESIS Node Version.2

Figure 3 depicts the extended GENESIS node design [8],[9], featuring three processors: the node CPU (i870), the CP (i870), and the vector processor. The GENESIS Node Version.1 described above realizes part of this design, namely the CPU and the CP. Adding the vector processor, including the high-speed vector memory banks, has been planned so far for the second phase of the GENESIS development. A prototype version of the vector processor has already been built, using the BIT B2110/20 floating-point processors (33 MHz), and benchmarks have been run on it.

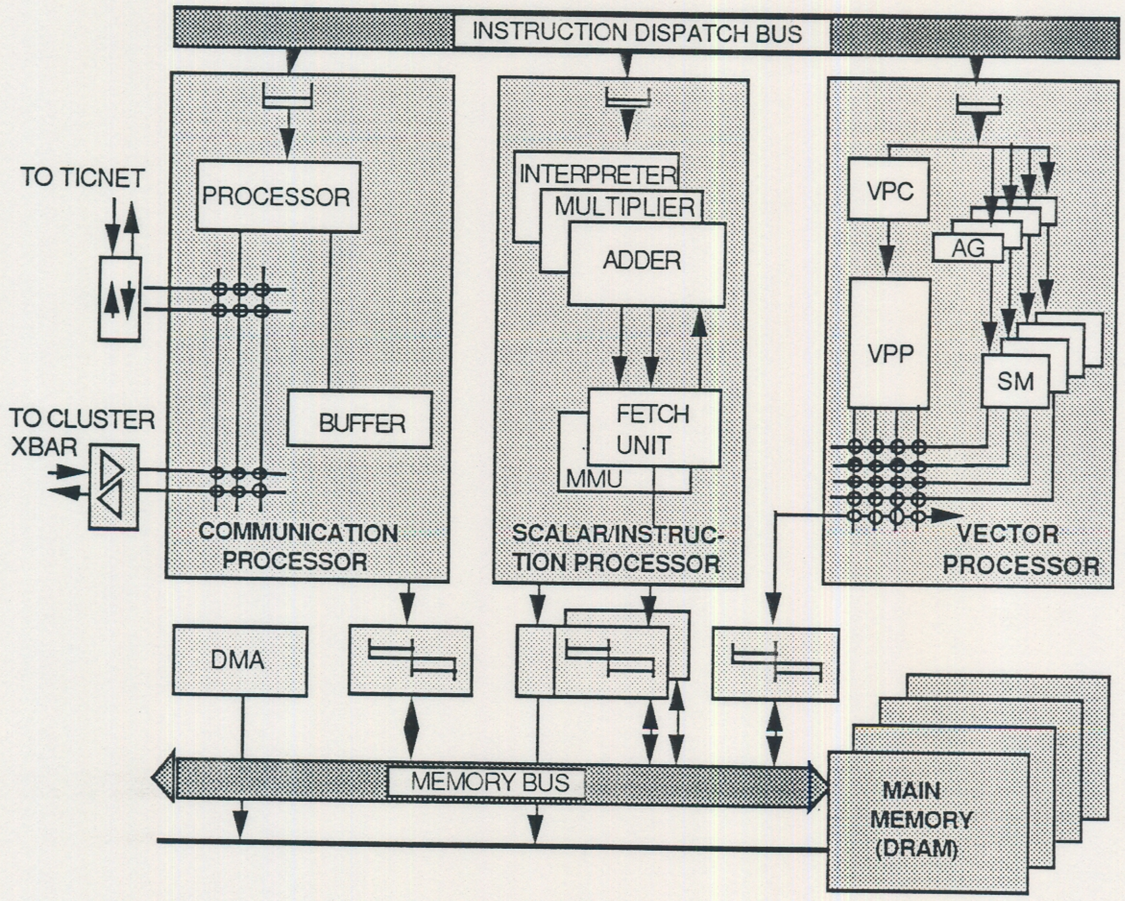


Figure 4 GENESIS Node Version.2 -- block diagram

The question that has arisen in the meantime is whether it will still pay to add the vector processor. With the i870 (50 MHz) as the *application processor*, a peak performance of 100 MFLOPS for two chained operations will be reached. On the other hand, a new ECL chip set recently announced by a leading manufacturer will allow us to build a vector processor that would bring this figure up to 200 MFLOPS (at 100 MHz clock frequency). An outstanding feature of the vector processor design shown in Figure 3 is a very favorable ratio of sustained performance versus peak performance for the *Linpack* and the *Lawrence Livermore Loop* benchmarks [10],[11]. Because of the compiler situation, that ratio is still quite poor in the case of the scalar high-MFLOPS processors.

Adding a vector processor to the node therefore still seems to be a good bargain: It increases the hardware cost of the node by only 50...60 % while it doubles the peak performance (with an even higher increase of sustained performance). Moreover, while a vector processor can readily execute complex vector operations, e.g., sparse vector processing, the pipelined scalar processor must do this by software. Last but not least, a vector processor, in connection with its address generators, is able to fetch two operands and store the result with every clock tick for arbitrarily long vectors, provided, the memory bandwidth allows it, while the vector performance of the scalar processor depends on its cache size.

However, adding a vector processor doubles also the hardware development cost and time. This not only may cause funding problems, specifically for a smaller manufacturer, but it carries the danger of missing the market window. In addition, it must be considered that with BiCMOS and a clock frequency of 100 MHz in the mid-nineties, 200 MHz towards the end of the decade [12], pipelined scalar processors will eventually be available featuring a peak performance of 200 or 400 MFLOPS, respectively. This will be combined with a large cache size. The critical compiler issue will be resolved in a couple of years, to the point that the compiler may support a multiprocessor node [5]. Having several application processors in the node sharing the common node memory will be the cheapest way of increasing the node performance further. Moreover, a very high performance machine with scalar nodes will find a much broader application spectrum.

All these are strong arguments against perpetuating node vectorization. Therefore, the notion of node vectorization as the next step must be reconsidered. Rather, in view of the development outlined above it may be more appropriate for us to concentrate our efforts on the development of a multi-processor node and on the compiler issues implied.

3 THE GENESIS INTERCONNECTION NETWORK

The crucial issue of distributed memory architectures is *communication overhead.* Great care must be taken to make communication extremely efficient in order to avoid a situation where the communication overhead would consume too much of the parallel processing gain. The communication overhead can be expressed in terms of MFLOP wasted because of *communication latency.*

Communication latency stems from two sources: (1) the latency of the interconnection network and (2) the time it takes the operating system to start a communication. The network latency is primarily a function of its transmission bandwidth, whereas the startup time depends on the protocols employed and the degree of hardware support provided to the operating system. It does not suffice to minimize one or the other of the two factors. However, the factors may have different weights depending on the message size. Short, fixed size messages require a very short startup time and a favorable blocking behavior of the interconnect, while transmission bandwidth is hardly of concern. High volume data transfer, on the other hand, calls primarily for a high communication bandwidth. Simulations have yielded the rule of thumb that each node-to-node link must have at least as many Mbytes per second as the node has MFLOPS [7].

With current technology up to 100 Mbytes of transmission bandwidth are readily feasible. Within that limit, the choice of installed bandwidth is a matter of economy. In connection with the *wormhole routing* strategy a low transmission latency can be achieved as long as the network has a sufficiently favorable blocking behavior. However, economy requirements also call for cost-optimization. Therefore, the tolarable communication overhead must be identified which, in turn, depends on the nature of the applications for which the system is designed. If the application algorithms exhibit a strong locality of communication, a simpler and cheaper network, e.g., the 2D-mesh [12], may suffice. If there is no strong locality, one must go to networks that have a higher degree of connectivity such as the *hypercube* [12], the *TICNET* [13], or *multi-level crossbar networks* [7],[14]. An important area where different network types differ considerably is the *blocking behavior* [14]. In order to obtain a good blocking behavior, the network should provide sufficiently many alternative, non-interfering data paths, a requirement that neither the 2D-mesh nor the nonredundant multi-stage interconnection network can fulfill.

A good blocking behavior is provided by the hypercube. A hypercube of dimensionality N has (log N) edges to which a node is linked, thus providing (log N) alternative data paths. However, a link, that is, a bi-directional channel with, say, 100 MHz transmission bandwidth, is quite expensive. Ten such channels, as needed for a 10-cube interconnect for 1024 nodes, may readily cost more money and real estate than the node proper with all its processors and megabytes of memory.

Since a node can send or receive through only one channels at a time, it is more economical to have only a minimal number of channels and provide the multiplicity of alternative data paths by virtue of complex multi-way switches in the network. Such multi-way switches can be provided by crossbars. A single crossbar for interconnecting a large number of nodes is not feasible; rather, the crossbar size should be chosen so that it fits on a chip. To interconnect a large number of nodes, one therefore must employ multi-level crossbar switches. It has been shown that multi-level crossbar networks have an even better blocking behavior than the hypercube [14] but can be realized by VLSI components and, thus, are less costly [7].

GENESIS shares with SUPRENUM a two-level interconnection structure [2],[7],[14]. At the lower level *clusters* are formed by interconnecting a given number of nodes, while the clusters are interconnected at the higher level. In the GENESIS architecture this is realized by a two-dimension network of crossbars as indicated in Figure 4. Each cluster has 2 crossbars: one to interconnect the nodes of the cluster and a second one to provide links to all other clusters. Only 2 links are needed in each node, which leads to a most economical realization.

4 THE NODE OPERATING SYSTEM

It is a characteristic feature of the distributed memory architecture that each node must have its own operating system kernel, to perform the following tasks:

- manage the resources (processes, memory, etc.) of the node;
- carry out inter process communication (IPC) with other nodes.

Furthermore, an appropriate *node operating system* (NOS) must satisfy the following requirements:

- message-passing must be carried out with minimal startup time;
- scalability must be supported by providing the view of an abstract machine;
- an arbitrary distribution of global services over the nodes must be allowed.

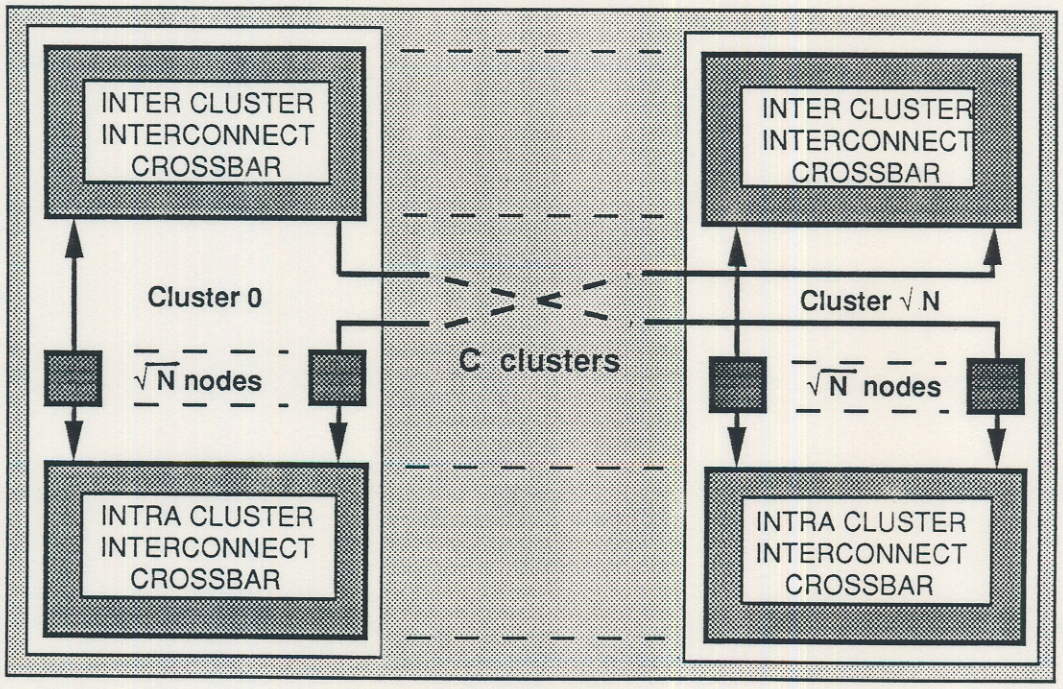

INTER CLUSTER INTERCONNECT CROSSBAR
INTER CLUSTER INTERCONNECT CROSSBAR
Cluster 0
Cluster √N
C clusters
√N nodes
√N nodes
INTRA CLUSTER INTERCONNECT CROSSBAR
INTRA CLUSTER INTERCONNECT CROSSBAR

Figure 4 The GENESIS two-dimensional crossbar interconnect

These requirements call for a highly modular NOS; therefore, a standard operating system such as UNIX would be unsuitable and also much too slow. The NOS used in the GENESIS computer is a refined version of the *Program Execution And Communication Environment* (PEACE) developed for the SUPRENUM computer [16]. Compared to SUPRENUM PEACE, the new GENESIS PEACE version is even more "generic," thus supporting ultimate scalability. Moreover, GENESIS PEACE supports a virtual node memory with demand paging.

The notion underlying the PEACE design is that of a *famlily of operatings systems* [17] with a *familiy of message passing kernels* [18]. Despite its extreme modularity, the efficiency of PEACE is unsurpassed by any other operating system. This has been accomplished by a structure consisting of *teams of light-weight processes* and *leagues of teams*. Figure 5 illustrates the process model of the PEACE operating system.

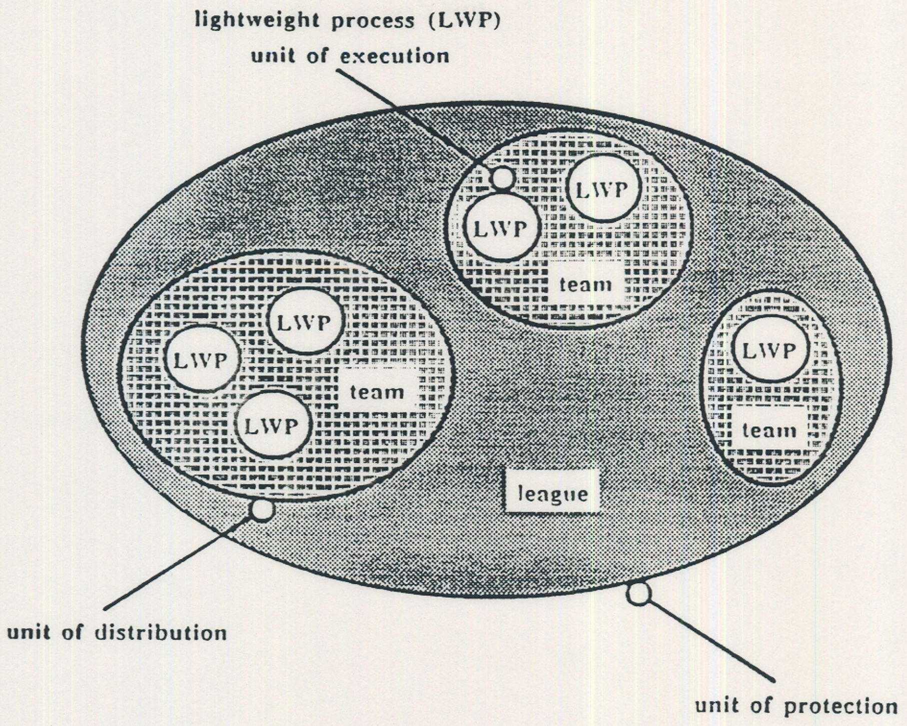


Figure 5 Process model of the PEACE operating system

The message passing kernel of a node communicates with the message passing kernel of another node by a *non-buffered synchronous IPC mechanism* based on *send-receive-reply sequences*. This is the fastest possible form of communication [19]. Short messages between teams are exchanged with very low overhead through *signal-await* sequences. System services can be invoked anywhere in the system by *remote procedure calls* (RPC). While this mechanism takes care of the high speed message exchange needed for the purpose of system organization and synchronization, data objects of application programs are transferred by a mechanism called *high volume data transfer* (HVDT) as described below. In case the programming model calls for *asynchronous communication*, this is readily provided by an appropriate lightweight process that works "on top" of the basic synchronous communication as illustrated by Figure 6. In this scheme, the lightweight process sends the message, while the original sender process, after having started the lightweight process, continues with execution.

Figure 7 illustrates the HVDT mechanism. Sender process and receiver process both communicate with a PEACE server called *Network Independent Communication Executive* (NICE). NICE is the interface to the *Communication System* (COSY) of the node. Each COSY includes a *DMA address generator* that can be set-up to address the elements of data structure objects (e.g., arrays) in the desired order. The first step of a HVDT is to establish a *rendezvous* between the COSY of the sender node and the receiver node, respectively. This ensures the receiver´s readiness, i.e., provides the *end-to-end significance* required for a secure communication. It also initializes the address generators. Subsequently, a data object of arbitrary size and structure can be copied directly from the address space of the sender process into the address space of the receiver process. In GENESIS, the COSY has its own, dedicated processor, viz. the CP.

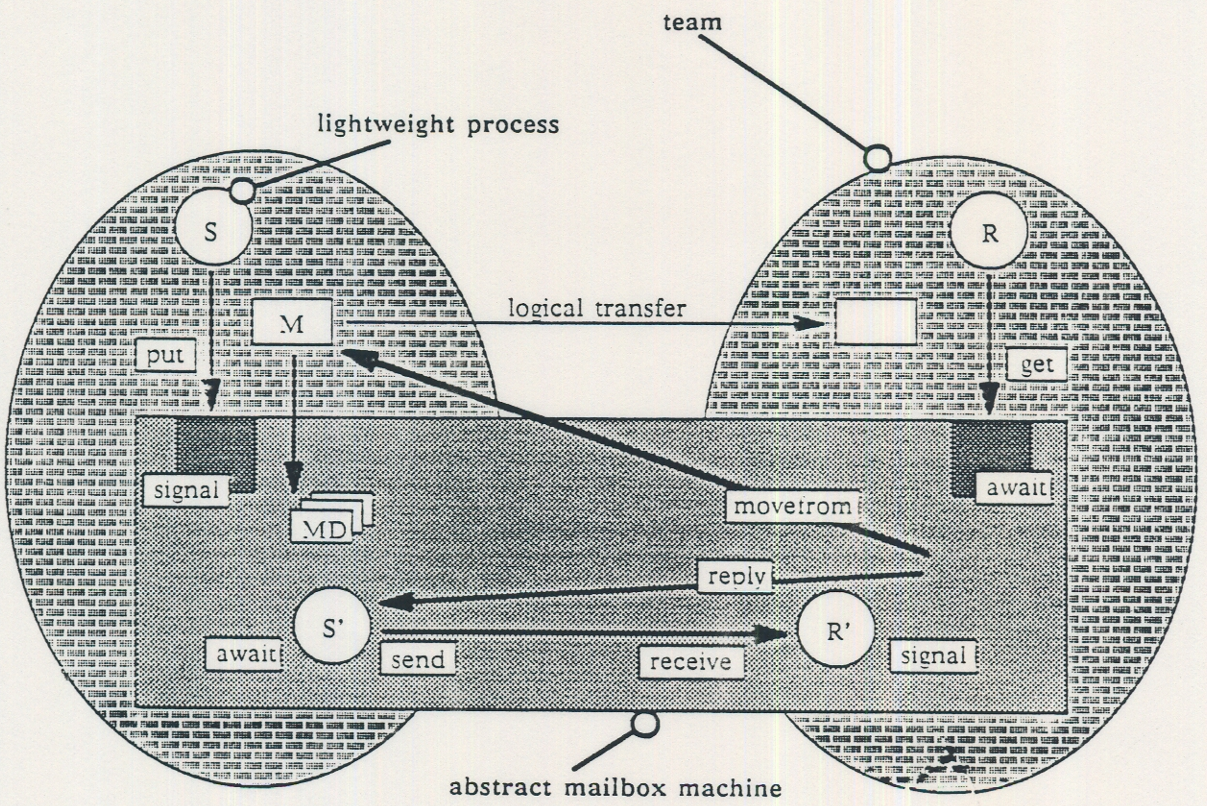


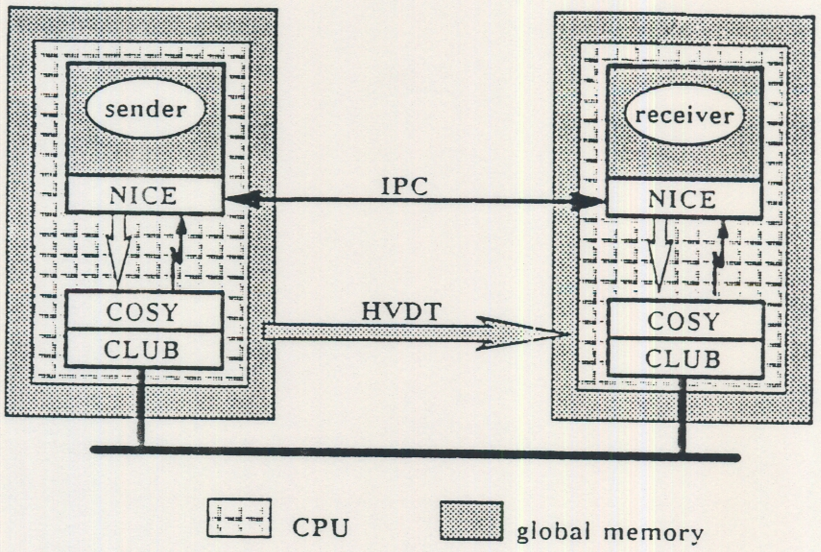
Figure 6 Lightweight process implementation of a no-wait send

5 GENESIS SOFTWARE DEVELOPMENT

5.1 Programming Distributed Memory Architectures

First attempts have been made to develop automatic parallelizers [6] for distributed memory architectures, yet practically usable result may still be some years away. Therefore, in the present state-of-the-art the user must explicitly partition the application program into cooperating tasks, distribute the tasks over the nodes of the system, and put the necessary IPC constructs in the right places. In all grid problems with regular solution spaces, the partitioning of the program can readily be performed by dividing the solution space into as many equal slices as there are nodes in the system and assigning a slice to each node.

In regular grid problems, grid generators, communication routines, and standard algorithms may be preprogrammed for a parameterized solution space and made available to the application programmer in the form of library routines. For example, SUPRENUM offers such a library for linear algebra computations as well as a multigrid PDE solver package. Whenever the user wants to solve a problem of one of the classes for which a library exists, all he has to do is to call bind the library into his application program and call the appropriate routines.



NICE	Network Independent Communication Executive
COSY	Communication System
CLUB	Cluster Bus

Figure 7 Illustration of the high volume data transfer mechanism

If the solution space is not regular or if the application is not a grid problem in the first place, program partitioning may be a more complicated task. In general, a good partitioning may be found only by a trial-and-error procedure. In this case the user does a partitioning, runs the program, and corrects the partitioning, if the first attempt has not led to a satisfactory node utilization. Furthermore, the user will have to program the inter process communication explicitly. In this endeavor the user may be supported by performance evaluation and visualization tools.

5.2 GENESIS Program Development and Execution Environment

In the following we describe several tools considered of vital importance for the program development in a distributed memory architecture such as GENESIS [8]. These tools are being developed by various partners as part of the GENESIS project.

5.2.1 Fortran Compiler

GENESIS-fortran is a fortran-90 type language, i.e., it comprises vector datatypes. In addition, the language must encompass a process concept as well as the necessary IPC constructs. The code generator of the compiler must produce optimized code for the i870 processor. Such a compiler is under development.

5.2.2 Simulator

Program development and verification is supported by simulators running on other conventional UNIX machines. No changes occur in switching between testing a program in a simulator environment and running a program on the real GENESIS machine. Invoking preprocessors rather than the GENESIS-fortran compiler itself will handle the specific MIMD features and embed the user program into a simulation runtime system. The simulator has various options, to provide the programmer with detailed information about the behaviour of the distributed application. An interactive simulation interface will allow the user to control the actual run of a simulation package. The simulator must be fast enough to run simulations even of large problems in reasonable time.

5.2.3 Parallel Debugger

The Parallel Debugger allows the user to debug a distributed fortran program on the GENESIS machine. It is based on the abstract machine view and can be used interactively or on a post mortem dump. Interaction with the debugger takes place via a window-based dialog program running on the host computer. Each task will have its own subwindow attached; thus, debug information can be received and displayed asynchronously. Graphical symbols are used to indicate the state of the processes currently represented on the screen.

5.2.4 Performance Analysis and Visualization Tools

During program development and tuning, there will be a need for detailed performance data as well as an understanding of the behaviour of the individual tasks of the application program. The Performance Analysis and Process Visualization Tools provide insight into the behaviour of a distributed application at the cost of execution speed. The tools can be used on either the real machine or the simulator. Information about the executing tasks will be displayed in graphic form at three levels:

- animated replay of task activities;
- time protocol;
- statistical data on processor utilization and inter process communication

5.2.5 Execution Environment

The user controls the execution of a program by issuing appropriate UNIX system commands, which then are executed by specific servers. The collection of all servers is called the GENESIS Execution Environment . A multitude of requests for the execution of user tasks may occur at the same time. These tasks must be scheduled with the aim to maximize the total system throughput. This is performed by the GENESIS Job Manager. The GENESIS Job Manager maintains a table of active (frozen) jobs as well as a queue of waiting jobs.

File handling is performed by the GENESIS File Manager on each node that has a disk. The user sees one logical file system into which the distributed file systems are integrated. A dedicated, uniform access path syntax will be common to all file server components.

5.3 Virtual Shared Memory Architecture (VSMA)

As is pointed out in the introduction, the Virtual Shared Memory Architecture (VSMA) is the attempt to free the programmer from the task of having to partition his program into cooperating processes, distribute the processes over the nodes, and program the necessary IPC. This will allow the user to use conventional languages that require the view of a global address space. We anticipate at least the smaller systems listed in Table 1 to be eventually of the VSMA type.

A program written for a VSMA must be partitioned by a *parallelizing compiler* into a number of concurrent *threads of control* (TOC), which are then distributed over the nodes. The TOCs may share data entities which they may read or write, depending on the access capability they possess for them. In compliance with the data dependencies in the program, the data accesses must be synchronized by critical regions. It is advantageous to choose the pages of a virtual memory with demand paging as the sharable entities; such an approach is called *Shared Virtual Memory* [20].

Locking and unlocking critical regions requires indivisible semaphor operations. Implementing a lock by agreement between N sharing nodes of a distributed system requires 2*(N-1) messages [21]. Therefore, when executed by software, the data access synchronization through critical regions in the VSMA program induces a considerable overhead that may readily exceed the IPC protocol overhead in distributed memory architectures. Consequently, specific architectural support must be provided to make a VSMA efficient. Here, the most important single measure is minimization of the message passing overhead.

For efficiency reasons, the nodes of a VSMA operate on their own copies of a shared *primary page*. This raises the problem of ensuring the consistency of the primary page and its copies. That is, the main problem of a VSMA is to maintain the *coherence* of the virtual shared memory. Data coherence may be maintained all the time, in which case it is called *strong coherence*, or only at certain synchronization points, in which case it is called *weak coherence*. Adding weak coherence enhances the efficiency of the VSMA. Both kinds of coherence can be achieved in an efficient manner by a capability-based mechanism [22]. For details please refer to [22].

The CP in the GENESIS node will strongly support the VSMA mechanisms described in [22]. The extraordinary message passing efficiency of GENESIS is very helpful too. Hence, GENESIS will provide outstanding architectural support for a future VSMA implementation. The critical issue will again be the compilers.

REFERENCES

[1] Rosenberg R.: *Supercube*, Electronics, Feb. 11, 1985, 15-17

[2] Giloi W.K.: *The SUPRENUM Architecture*, in (Jesshope C.R., Reinartz K.D.(eds.): CONPAR 88, Cambridge University Press 1989, 10-17

[3] Behr P., Montenegro S.: *The SUPRENUM Node Computer*, in (Jesshope C.R., Reinartz K.D.(eds.): CONPAR 88, Cambridge University Press 1989, 18-26

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

[5] Zima H.P., Bast H.J., Gerndt H.M.: SUPERB: *A Tool for Semi-Automatic MIMD/SIMD Parallelization*, Parallel Computing 6 (1988), 1-18

[6] Giloi W.K., Schroeder W.: *Very High-Speed Communication in Large MIMD Supercomputers*, Proc. ICS ´89, ACM Order No. 415891, 313-321

[7] Giloi W.K.: *GENESIS - The Architecture and Its Rationale*, ESPRIT Project P2702, Internal Tech. Report 1989

[8] Bruening U., Giloi W.K.: *Architecture of a Functionally Parallelized Processor With Hardware Synchronization and Communication*, Proc. ICS 89, Internat. Supercomputing Institute 1989, 248-252

[9] Lu X.: *LINPACK Benchmark Test Results on STARLET-II*, FIRST Internal Tech. Report 1989

[10] Lu X.: *Lawrence Livermore Loops Benchmark Test Results on STARLET-II*, FIRST Internal Tech. Report 1989

[11] USA Today, August 3, 1989

[12] Feng T.Y.: *A Survey on Interconnection Networks*, COMPUTER 14,12 (1981)

[13] Giloi W.K., Montenegro S.: *Super Interconnection Structures for Super Computers*, Proc. ICS 89, Internat. Supercomputing Institute 1989, 310-316

[14] Giloi. W.K., Montenegro S.: *Blocking Behavior Analysis of Interconnection Networks for Highly Parallel MIMD Architectures*, paper submitted for publication

[15] Schroeder W.: *The PEACE Operating System and Its Suitability for MIMD Message Passing Systems*, CONPAR 88, Cambridge University Press 1988, 27-34

[16] Parnas D.L.: *On the Design and Development of Program Families*, Research Report BS I 75/2, Technical University of Darmstadt 1975

[17] Schroeder-Preikschat W.: *PEACE -- A Distributed Operating System Family for High-Performance Local Memory Systems*, presented on the 6th ESPRIT Conf. 1989

[18] Behr P.M., Giloi W.K., Schroeder W.: *Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems*, in Wright M.(ed.): Proc. IFIP WG 2.5 Working Conf. on Aspects of Computation and Asynchronous Parallel Processors, North-Holland, Amsterdam 1989, 239-248

[19] Li K.: *Shared Virtual Memory on Loosely Coupled Multiprocessors*, P.D. Thesis, Yale University 1986

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

[21] Giloi W.K. et al.: *A Capability-Based Implementation of Shared Virtual Memory*, paper submitted to EDMCC2