Latency Hiding in Message-Passing Architectures

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

GMD Institute for Computer Architecture and Software Technology, Berlin, Germany
e-mail: w.giloi(a)computer.org

Abstract

The paper demonstrates the advantages of having two processors in the node of a distributed memory architecture, one for computation and one for communication. The architecture of such a dual-processor node is discussed. To exploit fully the potential for parallel execution of computation threads and communication threads, a novel, compiler-optimized IPC mechanism allows for an unbuffered no-wait send and a prefetched receive without the danger of semantics violation. It is shown how an optimized parallel operating system can be constructed such that the application processor's involvement in communication is kept to a minimum while the utilization of both processors is maximized. The MANNA implementation results in an effective message start-up latency of only 1...4 microseconds. It is also shown how the dual-processor node is utilized to efficiently realize virtual shared memory.

Keywords: Distributed memory message-passing architecture, dual-processor node, latency hiding, unbuffered no-wait send, parallel operating system, virtual shared memory

1. Introduction

The two dominant programming paradigms for parallel computers are the *shared memory model* and the *message passing model*. The shared memory model offers the advantage of a global address space that supports *pointer types* and allows parallel executing program segments to communicate through shared variables. The message-passing model, on the other hand, reflects the manner in which the hardware of a distributed memory architecture works. Thus, it can be more efficiently implemented, at the prize of demanding from the programmer optimized data distribution over the distributed memories and programming in terms of message passing constructs.

The shared memory paradigm may be provided on a distributed memory computer either by the *distributed shared memory architecture* (DSMA) or the *virtual shared memory architecture* (VSMA) built on top of a message-passing hardware. Since the DSMA has only a limited upward scalability [Lea 92], it is not a suitable architecture for massively parallel computers. This makes the message-passsing architecture still the prevailing parallel computer architecture. The problem with message-passing programming can be mitigated by appropriate tool sets such as PVM [Bea 93] or MPI or altogether avoided by parallelizing compilers [Hae 93]. If one wants to support pointer languages, one may switch to the VSMA mode. Consequently, the task for the computer architect is to make message passing as well as the support of VSMA as efficient as possible. This is what this paper is about.

First we shall revisit a concept that we introduced some years ago [GaS 89], viz. the notion of having two processors in each node of a distributed memory architecture: the *application processor* (AP) and the *communication processor* (CP). Presently, this concept is being employed in a number of commercial parallel computers (e.g., Intel Paragon, Meiko CS-2, Thinking Machine CM-2). However, it is our impression that this concept still is not generally understood. Therefore, we first present the rationale for the dual-processor node (Section 2).

The remainder of the paper is structured as follows. In Section 3, we discuss a realization of the dual-processor concept, given in the form of the MANNA node. MANNA (massivley-parallel architecture for numerical and non-numerical applications) is a virtual shared memory machine developed at the GMD Research Institute for Computer Architecture and Software Technology (FIRST) [Gil 94]. Performance figures obtained with that node are given. In Section 5, we introduce the semantics of latency-minimizing communication constructs, the *synchronized no-wait send* and the *prefetched blocking receive*. Section 6 deals with the operating system issues. A parallel operating system design is presented that takes maximum advantage of the dual-processor node architecture. In Section 7 it is shown how the MANNA node supports efficiently a virtual shared memory mechanism.

2. The Concept of the Dual-Processor Node

The message start-up time depends strongly on the functionality of the operating system *kernel* which, in

turn, is dictated by the mode of operation of the system. The simplest case is the single-user, single-tasking operation with one thread of control or several concurrent, unscheduled threads. In this *single address space model*, the entire node software—application as well as *kernel* functions—is running without protection in the same address space. This simple mode provides a *dedicated machine* on which communication is handled by runtime library routines. With the high-speed PEACE operating system [Sch 91] running on the 50-MIPS i860XP superscalar processor, we have a latency for a single message-passing transaction of 24 microseconds. As Table 1 demonstrates, this currently amounts to the world record in speed.

Maschine	Operating System	Latency [µs]
Paragon XP/S	OSF/1	240
nCUBE/2	PUMA	110
iPSC/860	NX	100
CM-5	CMOST	65
Paragon	PUMA	50
nCUBE/2	Active Messages	32
MANNA *	PEACE	24
MANNA **	PEACE	4

* 1-processor node ** 2-processor node

Table 1: Message-passing latency of parallel machines

Preemptive scheduling and the separation of task environments must be added if a *multi-tasking environment* is to be supported. In a single-processor node, each communication activity leads now to an operating system *kernel* trap performing an environment switch. This brings the start-up time for the PEACE operating system on a single i860XP up to 80...100 microseconds. Over one third of that time must be attributed to the *kernel* trap that becomes part of each communication.

The effect of communication latency can be drastically reduced by furnishing the node with a dedicated *communication processor* (CP) in addition to the CPU (AP) [GaS 89]. In this symbiosis the task of the AP is to uninterruptedly *produce megaflops*, while the CP executes the communication tasks of the operating system *kernel*. Both processor work in parallel; consequently, the communication start-up time occurs in the CP but is hidden from the AP. The AP sees only the latency of sending a communication request to the CP.

3. Example: The Dual-Processor MANNA Node

The latency hiding scheme described above has been implemented on the MANNA computer. Figure 1 depicts the block diagram of the dual-processor MANNA node.

Both AP and CP are superscalar processors i860XP. Thus, both processors have the same memory management and can snoop on each other's caches. The processors communicate through the shared 32-Mbyte node memory. The elaborate memory design features burst transfer support from 4 interleaved memory banks, a three-staged, pipelined memory control unit, and the page mode of operation. Consequently, the memory access latency of 7 clock ticks is overlapped with the previous access cycle.

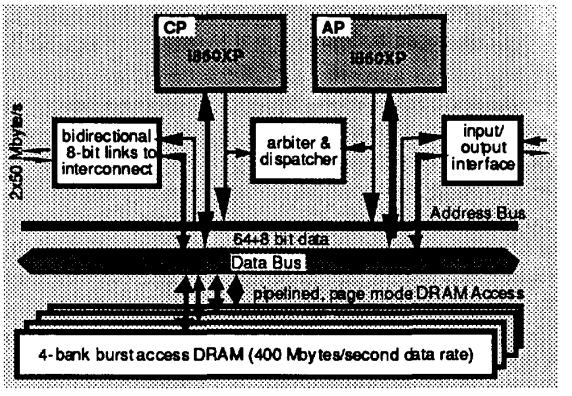

Figure 1 Block diagram of the 2-processor MANNA node

This latency hiding scheme results in an access time of 1 clock tick, provided the pipeline is filled and the accesses are to the same page. This gives the DRAM node memory almost the behavior of a secondary cache. A bidirectional communication link with a data rate of 2x50 Mbytes/second connects the node with the interconnection network, a hierarchy of crossbars [GaM 91].

One important design decision concerns the choice of the caching strategy, *write-through* or *copy-back*. *Write-through* leads to a higher traffic on the bus while *copy-back* leads to more snooping (in this case the caches must snoop not only on *writes* but also on *reads*). Figure 2 shows the relative performance of the MANNA node for the matrix multiplication [Bru 92].

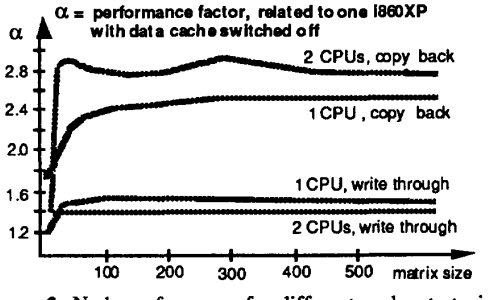

Figure 2 Node performance for different cache strategies

The sustained performance of the matrix multiplication is 68 MFLOPS (dp) for a matrix size ≥100.

4. The Usual Message-Passing Constructs

Three kinds of *send* constructs may be used in message-passing programming models:

- *remote procedure call*: remote activation of a thread;
- *rendezvous*: blocking send — non-blocking reply — blocking receive;
- *no-wait send*: non blocking send — blocking receive.

The remote procedure call (RPC) usually blocks the calling thread until the called procedure has signalled completion. There is no concurrency between the calling thread and the called thread. In object-oriented programming, the *remote object invocation* (ROI) is used instead.

The *rendezvous construct* offers the advantage of a synchronous protocol, viz. to work without buffers and buffer management. Once the rendezvous has been established, data objects are transferred directly from user space to user space. There is no danger of data inconsistencies. The end-to-end significance of the mechanism provides for a communication that also synchronizes the two threads. On the other hand, the construct sequentializes processing and communication, thus sacrificing some parallel processing potential.

The *no-wait send* construct corresponds with asynchronous communication. This not only enhances the parallel processing capabilities of the machine but also facilitates programming in the message-passing paradigm, as the programmers need to deal with only one synchronization point. This seems to violate the common wisdom that the *no-wait send* requires buffering of the sent objects in order to preserve the semantics of the program. The rule of procedural programming that the use of a variable frees it for the next definition applies also to the *send* construct. In the *no-wait send* mode sending may not be completed when a following statement redefines the sent object. In this case, the semantics of the program is violated. This is avoided if the *send* construct buffers the value of the object. Buffering of large objects, however, is undesirable, as it consumes additional memory space and—worse—generates copying overhead.

5. Synchronized No-Wait Send, Prefetched Blocking Receive

The *synchronized no-wait send* (SNWS) construct of MANNA implements asynchronous communication while ensuring the consistency of the sent objects without the need for buffering. Furthermore, it maximizes the overlap of computation and communication in the dual-processor node architecture described above. This is accomplished by the following mechanism.

(1) The send is non-blocking and non-copying. Execution of the send procedure proper is initiated in the communication processor at the point where the construct occurs. There exists a synchronization point as in a blocking send which, however, is separated from the communication and deferred until it is really needed.

(2) Synchronization is required when the sent object is redefined. Hence, the compiler inserts a *wait* prior to the first statement after the send in which the argument of the send construct occurs again. At that point, execution is blocked until the communication processor has signalled completion. Figure 3 illustrates the SNWS mechanism.

(3) The compiler reschedules the instructions following the send, to delay the statement with the synchronization point as much as allowed by the data dependencies. Thus, as much work as possible is created between the non-blocking send and the synchronization point.

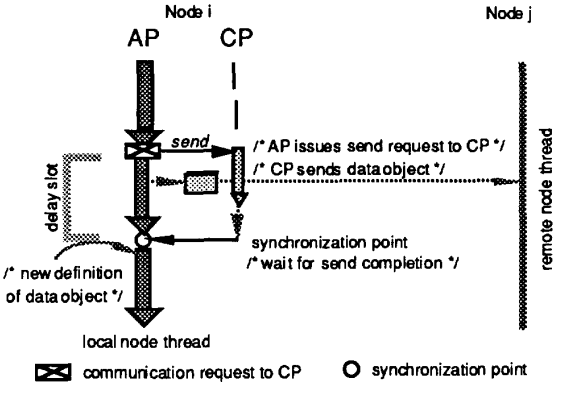

Figure 3 Synchronized No-Wait Send mechanism

The counterpart to the SNWS construct is the *prefetched blocking receive* (PBR). As in SNWS the point of communication and the point of synchronization—one and the same in the usual blocking receive—are now separated. Here, the synchronization point stays where the receive construct occurs, while the compiler moves the actual receive activity ahead in the instruction stream as much as permitted by the data dependencies. Hence, the receive activity is already started right after the last use of the argument of the receive construct. This allows the system to start the receive activity as much ahead in time as possible before the *wait*, thus enhancing the chance that there will be no blocking after all. Figure 4 illustrates the PBR mechanism.

The *receive* message triggers the communication thread. When that thread terminates, it queues a completition signal into the inter processor communication queue. This signal may be used by the application

thread to poll for completion of the receive. The separation of communication and synchronization in the PBR creates a delay, to be filled with useful computation by the user or—better—the compiler. The reader will notice the similarity with a load instruction of a load-store architecture where it is the task of the compiler to schedule the load as early as possible in the instruction stream to avoid wait times because of memory latency.

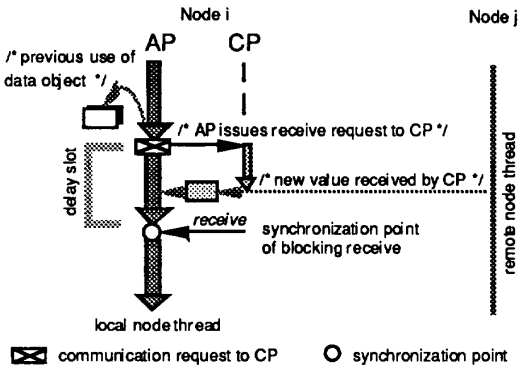

Figure 4 Prefetched Blocking Receive (PBR) mechanism

6. Operating System Optimization

The use of the dual-processor node can be optimized at the operating system level so that application threads and communication threads will maximally overlap. This is demonstrated by the example of the parallel operating system *PEACE* specifically developed for the dual-processor node.

6.1 Anatomy of the Parallel Operating System

PEACE has three intrinsic components: nucleus, kernel, and parallel operating system extensions (POSE). The application may be viewed as a fourth component.

The nucleus implements system-wide inter-process communication, as well as the runtime executive for the processing of threads. It provides a minimal basis and is part of the kernel domain. That is, the kernel is a multi-threaded system component that encapsulates minimal nucleus extensions for device abstractions, dynamic process creation and destruction, association of process objects with naming domains and address spaces, and propagation of exceptions (traps, interrupts). POSE performs the application-oriented services such as naming, process and memory management, file handling, I/O, load balancing, and host access.

Kernel services and POSE services are active jobs. They are implemented by lightweight processes that may be executed concurrently. In contrast, the nucleus is an ensemble of passive objects that schedule active objects.

6.2 Microscopic View

The nucleus performs the three major functions needed for network-wide inter-process communication:
(1) find out where the active/passive objects are located,
(2) enable data transport between nodes, and, therefore,
(3) attach the nodes to the network interface.

The nucleus consist of the 3 problem-oriented protocol layers shown in Figure 5. The layers interact through down-calls and up-calls. Queues are used where possible to decouple the different flows of control. Calls in either direction are asynchronous. Message transfer requests are queued only when needed.

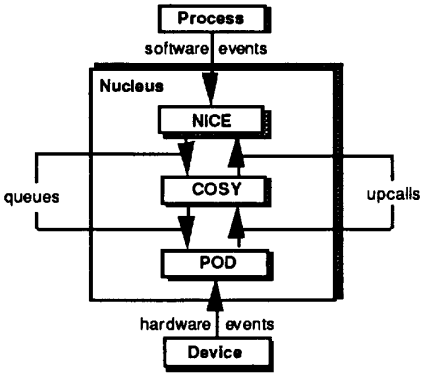

Figure 5 Nucleus architecture

Network-wide communication is carried out by NICE (network independent communication executive). NICE implements the inter-nucleus protocol. State transitions of processes and address spaces are initiated via end-to-end message transfers without the need for intermediate buffering. This makes NICE a management layer.

The underlying communication via the interconnect is conducted by COSY (COSY: communication system). COSY is a protocol suite that can handle all possible system configurations by logically providing a secured data transport of arbitrarily sized messages. POD is the actual network interface (POD: port driver). POD encapsulates the network device and attaches the nucleus to the network. POD makes COSY independent of the actually used network device, regardless of whether the device is physical or logical. This makes the COSY protocols portable.

6.3 Latency Hiding

In the dual-processor node architecture the distinction between application processor (AP) and communication processor (CP) is a software issue. As was mentioned

before, the main purpose of the AP-CP symbiosis is to reduce the effective message start-up time to a few micro-seconds.

The main task of the CP is to perform the communication between threads that reside in different nodes, while local inter-thread communication is handled by the AP. Consequently, the AP executes only NICE code sequences, whereas the CP executes the complete NICE-COSY-POD protocol suite (NCP suite). Figure 6 illustrates this configuration.

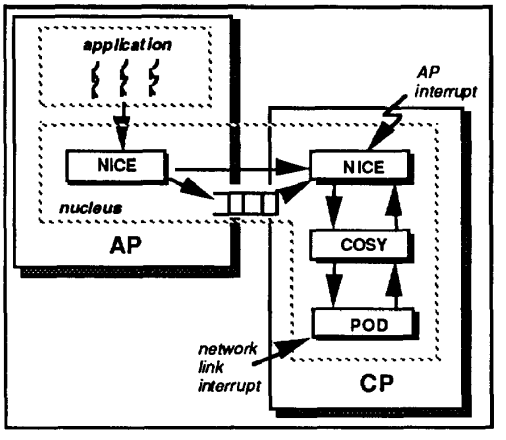

Figure 6 Nucleus-level coupling

The AP-NICE component establishes a message-passing interface to the application threads. This interface comprises primitives for inter-process communication and inter-thread synchronization. All local communication activities are handled by AP-NICE, while remote message-passing requests are forwarded to CP-NICE. The coupling of both NICE modules is implemented by an atomic queue that stores the network-wide communication requests. AP-NICE enqueues and CP-NICE dequeues the requests.

CP-NICE removes the communication requests from the queue and starts the NCP suite. This happens in parallel to AP-NICE and the application task(s). Thus, communication and computation overlap. The message start-up time for AP threads is determined by the AP-NICE execution path that queues the communication request. *On the MANNA node this takes under 4 micro-seconds.*

In addition to handling the send requests issued by the AP, the CP receives incoming messages and delivers them to the threads without interrupting the AP. A thread that is waiting on a message gets ready to run as soon as the CP has received the message from the link. Note that in this configuration the AP is never interrupted by communication requests, only the CP.

The extremely low latency allows for the design of fine-grain parallel applications. Specific compiler optimization may be employed: Similar to the pipeline operation of a RISC processor, the compiler may try to keep the "AP-CP pipeline" running by application code restructuring. The communication threads are automatically created by the run-time system whenever instructed by the compiler-generated code. This approach integratres compiler, run-time system, operating system kernel, and node architecture, thus ensuring extremely low latencies even in a multi-tasking environment. What the user sees is a "monolithic" abstract machine.

7. Support of Virtual Shared Memory

Virtual shared memory (VSM) [Li 86] is a software layer on top of a message-passing system that provides the global address space of a virtual memory. Thus, the scalability of a distributed memory architecture is combined with the easier-to-use shared memory programming paradigm. Shared data objects—usually pages—migrate to the nodes where they are referenced. This is usually accomplished by a page fault mechanism similar to the one employed in virtual memory [Li 86].

The identity of the owner of an object is found in the page tables of the node's virtual memory management. Moreover, the owner of an object (the writer) must know which other nodes have a copy of it. In the MANNA architecture, the owner of an object is therefore furnished with a compiler-generated *access list* containing the identifiers of all the nodes that share the object. In addition, a tag indicates the current access capability of the node. The access list is maintained by a PEACE service called *consistency manager*. There may be consistency managers for single shared objects or for groups of objects [CHS 92]. In lieu of owner identifiers, the page tables in the nodes contain the consistency manager addresses of the pages. Normally, the consistency manager never changes its location. Consequently, the page tables need not be updated after a change of ownership. Furthermore, since the consistency manager has a copy of the object, it can directly satisfy requests for copies, without having first to go to the owner. This saves a significant amount of message traffic, and it allows the system to destroy obsolete processes without destroying the shared objects they own. The executor of this mechanism is the CP.

The existence of multiple copies of memory objects in the system creates a potential consistency problem whenever a copy is modified by a write. This problem can be handled by an invalidation mechanism similar to the one applied for cache coherence. If one wants the VSM to behave exactly like a shared memory, the system must

ensure sequential consistency of the VSM accesses. Sequential consistency can be obtained by the MRSW mode of operation (MRSW: multiple reader — single writer), i.e., on a *write* all other copies of the written object are invalidated. This simple approach, however, may lead to a significant performance loss.

A more efficient solution is the use of the MRMW semantics in phases where this does not affect the correctness of program execution (MRMW: multiple readers — multiple writers). Consider, for example, the *lock-step* mode of operation where computation phases and communication phases alternate (this is typical for data parallel applications). During computation, the nodes may unrestrictedly write into their local copies of a shared object. Before the communication phase is entered, the diverging copies are then unified into one consistent object. This may be automatically performed by the system [Gea 91].

Such a mode of operation can be implemented, e.g., by the following *adaptive consistency model* [Gea 91]. By executing a *define_local* system function a thread receives the unconditioned write capability for its local copy. The write access to the copy is confined to the node, i.e., does not affect the copies of the same object in other nodes. Thus, threads in different nodes may write into the same page, each one into its local copy. The only restriction is that the threads are not allowed to write into the same location in the page, i.e., the write addresses must be mutually exclusive. For example, in array processing each writer may contribute a row or a column to a result matrix. At the end of the write phase, one of the writers executes a *define_global* function. This has two effects: (1) the system will automatically unify the local copies into a single global object whose content is the union of the local changes, while all local copies are invalidated and (2) the thread that executes the *define_global* function becomes the new owner of the unified copy. Unification can be performed by the hardware. The order in which the copies are merged is arbitrary. The merging procedure can be pipelined or be executed in parallel, e.g., by a *logtree merge algorithm*. This mechanism is provided by operating system functions executed on the CP. Consequently, its overhead is hidden from the AP.

8. Conclusion and Future Work

The latency hiding gain obtained by the dual-processor node architecture is enormous: the single-message start-up time may be reduced by up to two orders of magnitude! However, as Section 6 demonstrates, this requires a specific, highly optimized design of a parallel operating system. Existing operating systems—from UNIX to MACH—are not built according to the rules of optimal parallel operating system design [SCH 94]. This explains why the commercial systems mentioned above that adopted the dual-processor concept do not achieve the high latency hiding gain obtained in MANNA.

Section 5 shows how the degree of parallelism between computation and communication can be enhanced. In addition to the dual-processor node, this requires specific optimizations in the operating system and the compiler. Virtual shared memory with adaptive consistency becomes a special operating system service provided on demand.

References

[Bea 93] Beguelin A., Dongerra J., Geist A., Manchek R., Sunderam V.: A User's Guide to PVM — Parallel Virtual Machine, available from <netlib(a)ornl.gov> by the message <send pvm_shar from pvm>

[Bru 92] Bruening U.: MANNA Arbiter and CPU Kern, GMD FIRST Tech. Report 1992

[CHS 92] Cordsen J., Heuer J., Schröder-Preikschat W.: Problem-Oriented Virtual Shared Memory in an Object-Oriented Parallel Operating System, GMD FIRST Tech. Report (Sept. 1992)

[GaM 91] Giloi. W.K., Montenegro S.: Choosing the Interconnect of Distributed Memory Systems by Cost and Blocking Behavior, *Proc. 5th Internat. Parallel Processing Symposium*, IEEE Catalog no. 91TH0363-02 (1991), 438-444

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

[Gea 91] Giloi W.K., Hastedt C., Schoen F., Schroeder-Preikschat W.: A Distributed Implementation of Shared Virtual Memory with Strong and Weak Coherence, in Bode A.(ed.): *Distributed Memory Computing*, Proc. EDMCC2, LNCS 487, Springer-Verlag 1991, 23-31

[Gil 94] Giloi W.K.: MANNA — Prototype of a Next Generation Parallel Computer, to appear in *Proc. 1994 Mannheim Supercomputer Seminar*, Springer 1994

[Hae 93] Haenisch R.: SNAP! Prototyping a Sequential and Numerical Application Palallelizer, *Proc. Internat. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution, and Automatic Parallel Performance Prediction* (March 1993), Springer LNCS

[Lea 92] Lenoski D., Laudon K., Gharachorloo K., Weber W.-D., Gupta A., Hennessy J., Horowitz M., Lam M.: The Stanford Dash Multiprocessor, COMPUTER 3,92, 63-79

[Li 86] Li K.: Shared Virtual Memory on Loosely Coupled Multiprocessors, PhD thesis, Yale University 1986

[Sch 91] Schroeder-Preikschat W.: Overcoming the Startup Time Problem in Distributed Memory Architectures, in Milutiniovic V., Shriver B.(eds.): *Proc. 24th Hawaii Internat. Conf. on System Sciences, vol.1*, IEEE Society Press 1991, IEEE order no. 91TH0350-9, 551-559

[Sch 94] Schroeder-Preikschat W.: *Optimal Parallel Operating System Design*, Prentice-Hall, Englewood Cliffs NJ 1994