USING BROADCASTING TO IMPLEMENT DISTRIBUTED SHARED MEMORY EFFICIENTLY

Andrew S. Tanenbaum
M. Frans Kaashoek
Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

Parallel computers come in two varieties: those with shared memory and those without. The former are hard to build; the latter are hard to program. In this paper we propose a hybrid form that combines the best properties of each.

July 2, 1991

USING BROADCASTING TO IMPLEMENT DISTRIBUTED SHARED MEMORY EFFICIENTLY

Andrew S. Tanenbaum
M. Frans Kaashoek
Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

As computers continue to get cheaper, there is increasing interest in harnessing multiple CPUs together to build large, powerful parallel systems. The goal of this work is to achieve high performance at low cost. In this paper we will first survey the two major design approaches taken so far, multiprocessors and multicomputers, and point out their strengths and weaknesses. Then we will introduce and discuss a hybrid form that uses an unusual software organization on conventional hardware to achieve a system that is easy to build and easy to program. Finally, we will discuss a prototype system that we have built, describe some applications we have written for it, and give measurements of its performance.

There is much confusion in the literature about the terms ''parallel'' and ''distributed'' system. For our purposes, a *distributed* system is one in which independent jobs are executing on independent CPUs on behalf of independent users, but sharing some resources, such as a common file server. A typical example is a collection of (possibly heterogenous) engineering workstations connected by a fast LAN (Local Area Network) and sharing a file server.

In contrast, a *parallel* system is one in which a user is attempting to utilize multiple (usually identical) CPUs in order to speed up the execution of a single program. The CPUs may be in a single rack and connected by a high-speed backplane bus, they may be distributed around a building and connected by a fast LAN, or some other topology may be used. What is important is that a large number of CPUs are cooperating to solve a single problem. Thus the difference between the two is really a question of the software organization, not the hardware. In this paper we will concentrate on parallel systems, using as metric how much speedup can be achieved on a single problem by using n identical processors as compared to using only 1 processor.

A brief outline of the rest of this paper follows. We will first discuss the two major kinds of parallel computers in use, so we can contrast our design with them later. Then we will examine so-called NUMA architectures, which are an attempt at combining the two types of parallel machines. Our system is similar to NUMA in some important respects. After that, we will present our method of efficiently doing reliable broadcasting on unreliable hardware, which is the central concept in our proposal. We will then look at a paradigm for programming using this method. Finally, we will discuss a parallel programming language that we have designed and implemented to allow programmers to use our system conveniently.

Multiprocessors

Parallel computers can be divided into two categories: those that contain physical shared memory, called *multiprocessors*, and those that do not, called *multicomputers*. A simple taxonomy is given in Fig. 1. In the former type, there is a single, global, shared address space visible to all processors. Any processor can read or write any word in the address space by simply moving data from or to a memory address. In the latter type, the processors need some other way to communicate, for example, message passing.

The key property required of any multiprocessor is *memory coherence*. When any processor writes a value v to memory address m, any other processor that subsequently reads the word at memory address m, no matter how quickly after the write, will get the value v just written.

Multiprocessor Hardware. There are two basic ways to build a multiprocessor, both of

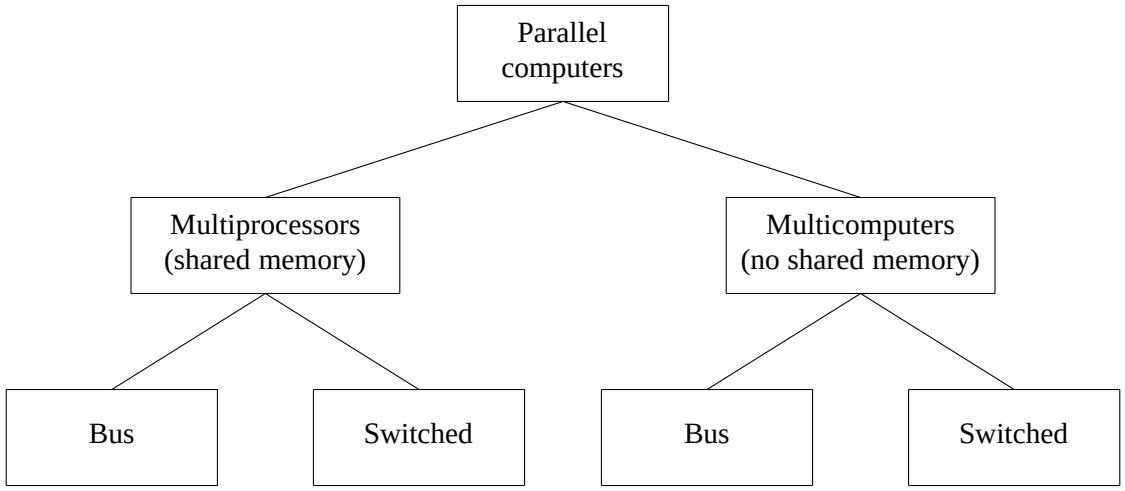


Fig. 1. A taxonomy of parallel computers.

them expensive. The first way is to put all the processors on a single bus, along with a memory module. To read or write a word of data, a processor makes a normal memory request over the bus. Since there is only one memory module and there are no copies of memory words anywhere else, the memory is always coherent.

The problem with this scheme is that with as few as 4 - 8 processors, the bus will be completely saturated. To get around this problem, each processor is normally given a cache memory, as shown in Fig. 2. The caches introduce a new problem, however. If processors 1 and 2 both read the contents of memory address *m* into their caches, and one of them modifies it, when the other one next reads that address it will get a *stale* value. Memory is then not coherent. Incoherent memory is hard for programmers to deal with, and is thus unacceptable.

The stale data problem can be solved in many ways, such as having all writes go through the cache to update the external memory, or having each cache constantly *snoop* on the bus (i.e., monitor it), taking some appropriate action whenever another processor tries to read or write a word that it has a local copy of, such as invalidating or updating its cache entry. Nevertheless, caching only delays the problem of bus saturation. Instead of saturating at 4-8 processors, a well-designed single-bus system might saturate at 32-64 processors. Building a single bus system with appreciably more than 64 processors is not feasible with current bus technology.

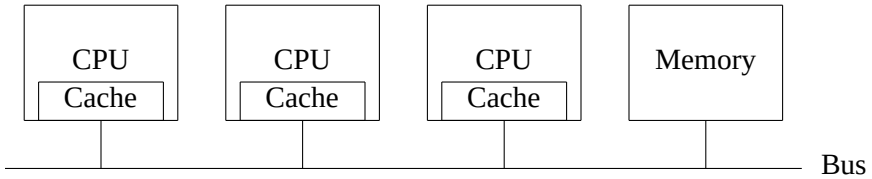


Fig. 2. A single-bus multiprocessor.

The second general approach to building a multiprocessor is using some kind of switching network, such as the *crossbar switch*, shown in Fig. 3(a). In this organization, each of the *n* processors can be potentially connected to any one of the *n* memory banks via a matrix of little electronic switches. When switch *ij* is closed (by hardware), processor *i* is connected to memory bank *j* and can read or write data there. Since several switches may be closed simultaneously, multiple reads and writes can occur at the same time between disjoint processor-memory

combinations.

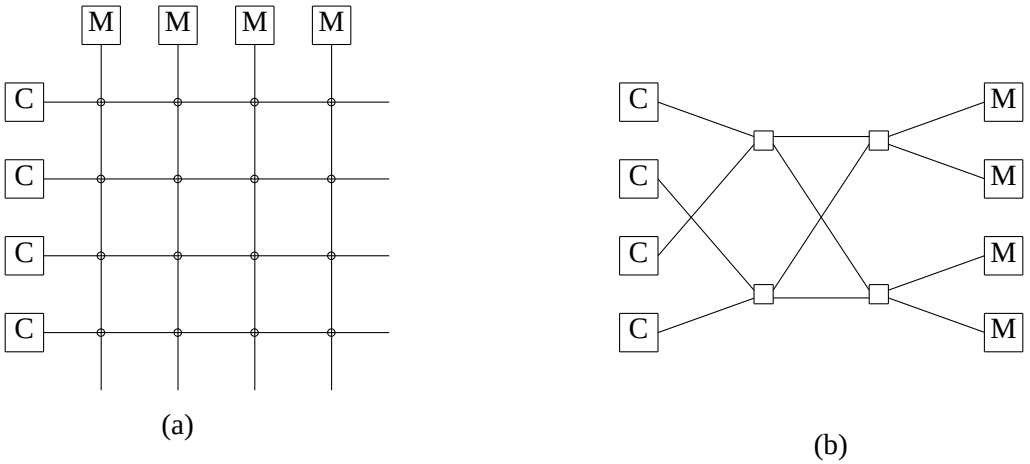


Fig. 3. (a) Crossbar switch. (b) Omega network. C is for CPU; M is for Memory.

The problem with the crossbar switch is that to connect n processors to n memory banks requires n^2 switches. As n becomes large, say 1024 processors and 1024 memories, the switch becomes prohibitively expensive and unmanageable.

An alternative switching scheme is to build a multiprocessor using the omega network shown in Fig. 3(b). In this figure, the CPUs are on the left and the memories are on the right. The omega network is a sophisticated packet switching network that connects them. To read a word of memory, a CPU sends a request packet to the appropriate memory via the switching network, which sends the reply back the other way.

Many variations of this basic design have been proposed, but they all have the problem that for a system with n processors and n memories, the number of switching stages needed is on the order of $\log_2 n$, and the number of switching elements is $n \log_2 n$. As an example of what this means, consider a system of 1024 RISC processors running at 50 MHz. With $n = 1024$, 10 switches must be traversed from the CPU to the memory and 10 more on the way back. If this is to occur in 1 cycle (20 nsec), each switching step must take no longer than 1 nsec, and 10,240 such switches are required. A machine with such a large number of very high speed packet switches is clearly going to be expensive and difficult to build and maintain, even assuming the designers are able to make many optimizations.

Multiprocessor Software. In contrast to the multiprocessor hardware, which, for large systems, is complicated, difficult to build, and expensive, software for multiprocessors is straightforward. Since all processes run within a single shared address space, they can easily share data structures and variables. When one process updates a variable and another one reads it immediately afterwards, the reader always gets the value just stored (memory coherence property).

To avoid chaos, co-operating processes must synchronize their activities. For example, while one process is updating a linked list, it is essential that no other process even attempt to read the list, let alone modify it. Many techniques for providing the necessary synchronization are well known. These include spin locks, semaphores, and monitors, and are discussed in any standard textbook on operating systems. The advantage of this organization is that sharing is easy and cheap and uses a methodology that has been around for years and is well understood.

Multicomputers

In contrast to the multiprocessors, which, by definition, share primary memory, *multicomputers* do not. Each CPU in a multicomputer has its own, private memory, which it alone can read and write. This difference leads to a significantly different architecture, both in hardware and in software.

Multicomputer Hardware. Just as there are bus and switched multiprocessors, there are bus and switched multicomputers. Figure 4 shows a simple bus-based multicomputer. Note that each CPU has its own local memory, which is not accessible by remote CPUs. Since there is no shared memory in this system, communication occurs via message passing between CPUs. The ‘‘bus’’ in this example can either be a LAN or a high-speed backplane; conceptually, these two are the same, differing only in their performance. Since each CPU-memory pair is essentially independent of all the others, it is straightforward to build very large multicomputer systems, certainly much larger than multiprocessors.

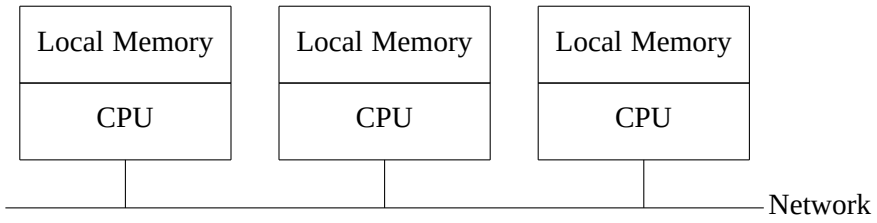


Fig. 4. A single-bus multicomputer.

Switched multicomputers do not have a single bus over which all traffic goes. Instead, they have a collection of point-to-point connections. In Fig. 5, we see two of the many designs that have been proposed and built: a grid and a hypercube. A grid is easy to understand and easy to lay out on a printed circuit board or chip. This architecture is best suited to problems that are two dimensional in nature (graph theory, vision, etc.).

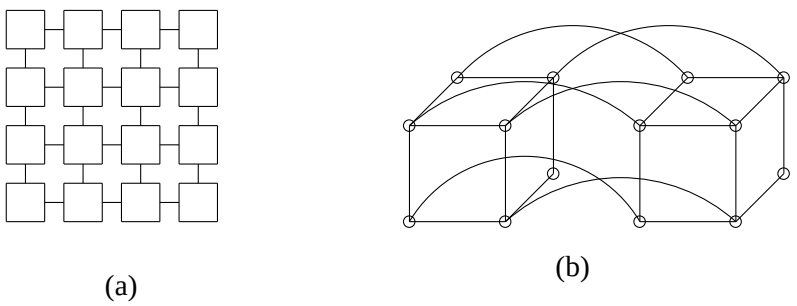


Fig. 5. Multicomputers. (a) Grid. (b) Hypercube.

Another design is a *hypercube*, which is an *n*-dimensional cube. One can imagine a 4-dimensional hypercube as a pair of ordinary cubes with the corresponding vertices connected, as shown in Fig. 5(b), where the lower left nodes at the rear have been omitted for clarity. Similarly, a 5-dimensional hypercube can be represented as two copies of Fig. 5(b) In general, an *n*-dimensional hypercube has 2^n vertices, each holding one CPU. Each CPU has a fan-out of *n*, so the interconnection complexity grows logarithmically with the number of CPUs.

Multicomputer Software. Since multicomputers do not contain shared memory (by definition), they must communicate by message passing. Various software paradigms have been devised to express message passing. The simplest one is to have two operating system primitives, SEND and RECEIVE. The SEND primitive typically has three parameters: the

destination address, a pointer to the data buffer, and the number of bytes to be sent. In the simplest form, the RECEIVE primitive might just provide a buffer, accepting messages from any sender.

Many variations on this theme exist. To start with, the primitives can be *blocking* (synchronous), or *nonblocking* (asynchronous). With a blocking SEND, the sending process is suspended after the SEND until the message is actually accepted and acknowledged. With a nonblocking SEND, the sender may continue immediately. The problem with allowing the sender to continue is that it may be in a loop, sending messages much faster than the underlying communication hardware can deliver them. Permitting this may result in lost messages.

A way to alleviate this problem somewhat is to have senders address messages not to processes, but to *mailboxes*. A mailbox is a special kind of buffer that can hold multiple messages. However, here too, overflow is still a possibility.

A more fundamental problem with message passing is that, conceptually, it is really input/output. Many people believe that input/output should not be the central abstraction of a modern programming language. To hide the bare input/output, Birrell and Nelson [1] proposed a scheme called *remote procedure call* (RPC), shown in Fig. 6. The idea is to hide the message passing, and make the communication look like an ordinary procedure call. The sender, called the *client*, calls a *stub routine* on its own machine that builds a message containing the name of the procedure to be called and all the parameters. It then passes this message to the driver for transmission over the network. When it arrives, the remote driver gives it to a stub, which unpacks the message and makes an ordinary procedure call to the *server*. The reply from server to client follows the reverse path.

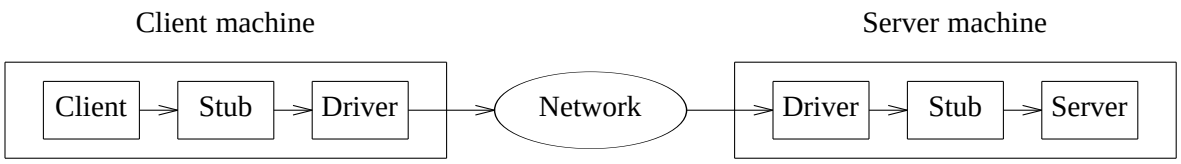


Fig. 6. Remote procedure call from a client to a server. The reply follows the same path in the reverse direction.

Although this scheme hides the message passing to some extent, since the client and server each think of the other as an ordinary local procedure, it is difficult to make it entirely transparent. For example, passing pointers as parameters is difficult, and passing arrays is costly. Thus the programmer must usually be aware that the semantics of local procedure calls and remote ones are different.

The conclusion of all the above is that multiprocessors and multicomputers have significantly different characteristics. Multiprocessors are hard to build but easy to program. In contrast, multicomputers are easy to build, but hard to program. What people would like is a system that is easy to build (i.e., no shared memory) but also easy to program (i.e., shared memory). How these two contradictory demands can be reconciled is the subject of the rest of this paper.

NUMA Machines

Various researchers have proposed intermediate designs that try to capture the desirable properties of both architectures. Most of these designs attempt to simulate shared memory on multicomputers. All of them have the property that a process executing on any machine can access data from its own memory without any delay, whereas access to data located on another machine entails considerable delay and overhead, as a request message must be sent there and a reply received. A system in which a single address space is shared among otherwise disjoint machines is called a *distributed shared memory* and forms the primary subject of this article.

Computers in which references to some memory addresses are cheap (i.e., local) and others are expensive (i.e., remote) have become known as *NUMA* (*Non Uniform Memory Access*)

machines. In this section we will describe three of the more interesting types of NUMA machines. They differ primarily in the granularity of access and the mechanism by which remote references are handled.

Word-oriented NUMA Machines. One of the earliest machines was Cm*, built at Carnegie-Mellon University and consisting of a collection of LSI-11 minicomputers [2]. Each LSI-11 had a microprogrammed MMU and a local memory. The MMU microcode could be downloaded when execution began, allowing part of the operating system to be run there. The LSI-11s were grouped together into clusters, the machines in each cluster being connected by an intracluster bus. The clusters were connected by intercluster buses.

When an LSI-11 referenced its own local memory, the MMU just read or wrote the required word directly. However, when an LSI-11 referenced a word in a remote memory, the microcode in the MMU built a request packet specifying which memory was being addressed, which word was needed, the opcode (READ or WRITE), and the value to be written for WRITEs. The packet was then sent out over the buses to the destination MMU via a store-and-forward network. At the destination, it was accepted, processed, and a reply generated, containing the requested word (for READ) or an acknowledgement (for WRITE).

The more remote the memory, the longer the operation, with the worst case being about 10 times the best case. It was possible for a program to run entirely out of remote memory, with a performance penalty of about a factor of 10. There was no caching and no automatic data movement. It was up to the programmer to place code and data appropriately for optimal performance. Nevertheless, to the programmer, this system had a single shared address space accessible to all processors, even though it was actually implemented by an underlying packet-switching network.

Page-oriented NUMA Machines. Cm* represents one end of the spectrum—sending requests for individual words from MMU to MMU in ‘‘hardware’’ (actually MMU microcode) over a set of tightly coupled backplane buses. At the other extreme are systems that implement virtual shared memory on a collection of workstations on a LAN. As in Cm*, the users are presented with a single shared virtual address space, but the implementation is quite different.

In its simplest form, the virtual memory is divided up into fixed size pages, with each page residing on exactly one processor. When a processor references a local page, the reference is done by the hardware in the usual way. However, when a remote page is referenced, a page fault occurs, and a trap to the operating system occurs. The operating system fetches the page, just as in a traditional virtual memory system, only now the page is fetched from another processor (which loses the page), instead of from the disk.

As in Cm*, pages are fetched by request and reply messages, only here they are generated and processed by the operating system, instead of by the MMU microcode. Since the overhead is so much higher, an entire page is transferred each time, in the hope that subsequent references will be to the same page. If two processors are actively using the same page at the same time, the page will thrash back and forth wildly, degrading performance.

A significant improvement to the basic algorithm has been proposed, implemented, and analyzed by Li and Hudak [3]. In their design, thrashing is reduced by permitting read-only pages to be replicated on all the machines that need them. When a read-only page is referenced, instead of sending the page, a copy is made, so the original owner may continue using it. Other pages may also be shared, but when a page is written, other copies of it are invalidated.

Li and Hudak present several algorithms for locating pages. The simplest is to have a centralized manager that keeps track of the location of all pages. All page requests are sent to it, which then forwards them to the processor holding the page. A variation of this scheme is to have multiple managers, with the leftmost *n* bits of the page number telling which manager is responsible for the page. This approach spreads the load over multiple managers. Even more decentralized page location schemes are possible, including the use of hashing or broadcasting.

It is worth pointing out that the basic page-oriented strategy could (almost) have been used with Cm*. For example, if the MMUs noticed that a particular page was being heavily referenced by a single remote processor and not at all being referenced by its own processor, the MMUs could have decided to ship the whole page to the place it was being used instead of

constantly making expensive remote references. This approach was not taken because the message routing algorithm used the page number to locate the page.

NUMA machines have been designed where a page may be transported a maximum of only *k* times, to reduce thrashing. After that, it is wired down and all remote references to it are done as in Cm*. Alternatively, one can have a rule saying that once moved, a page may not be moved again for a certain time interval.

Object-oriented NUMA Machines. An inherent problem with page-oriented NUMA systems is the amount of data transferred on each fault is fixed at exactly one page. Usually this is too much. Sometimes it is too little. It is hardly ever exactly right. The next step in the evolution of NUMA machines is an object-based system, in which the shared memory contains well-defined objects, each one with certain operations defined on it. When a process invokes an operation on a local object, it is executed directly, but when it invokes an operation on a remote object, the object is shipped to the invoker. Alternatively, the operation name and parameters can be sent to the object, with the operation being carried out on the remote processor, and the result being sent back. Both of these schemes have the property that no unnecessary data are moved (unlike the page-oriented NUMA machines that move 1K or 4K or 8K, to fetch even a single byte).

It should be noted that our definition of NUMA machines is somewhat broader than what some other writers have used. We regard any machine which presents the programmer with the illusion of shared memory, but implements it by sending messages across a network to fetch chunks of data as NUMA. Whether the code that implements this is in the MMU, as in Cm*, or in the operating system, as in the work of Li and Hudak, is simply an implementation choice. Similarly, whether the unit fetched is a word, a page, or an object, is also an implementation decision. The essential commonality of NUMA machines is that when a remote read is made, the system sees an exception, sends a message to the remote memory, gets a reply back containing the data needed, and restarts the instruction. In this view, Cm* and the model of Li and Hudak simply differ in where the exception handler runs and the block size returned on a fault.

Weakened Semantics. A completely different approach to implement distributed shared memory is to weaken the semantics, giving up the demand of absolute coherence. For example, suppose three processes, *A*, *B*, and *C* all simultaneously update a word, followed by process *D* reading it. If total coherence is required, the word will probably have to be sent first to *A*, then to *B*, then to *C*, and finally to *D* to carry out the three writes and the read.

However, from *D*'s point of view, it has no way of knowing which of the three writers went last (and thus whose value did not get overwritten). Thus in the Munin system, Bennet et al. [4] have proposed that returning any one of the values written should be legal. The next step is to delay remote writes until a read is done that might observe the values written. It may even be possible to perform an optimization and avoid some of the writes altogether. This strategy has been implemented in the Munin system, which uses a form of strong typing on shared objects to make it easier to perform these and other optimizations.

Other systems go even farther in this direction. Mether [5], for example, maintains a primary copy of each page and possibly one or more secondary copies. The primary copy is writable; the secondary copies are read-only. If a write is done to the primary copy, the secondary copies become obsolete and reads to them return stale data. Applications simply have to accept this.

Three ways are provided for getting resynchronized: the owner of a secondary copy can ask for an up-to-date copy, the owner of the primary copy can issue an up-to-date copy to the readers, or the owner of a secondary copy can just discard it, getting a fresh copy on the next read.

A New Model

Although weakening the semantics of the shared memory may improve the performance, it has the disadvantage of ruining the ease of programming that the shared memory model was designed to provide. In effect, it offers only the syntax of shared variables, while forcing the programmer to deal with semantics even less convenient than message passing.

In our research, we have devised an alternative model that preserves the coherency of (object-based) shared memory, yet has been implemented efficiently. The model, which we call *reliable broadcasting*, consists of four layers, as shown in Fig. 7.

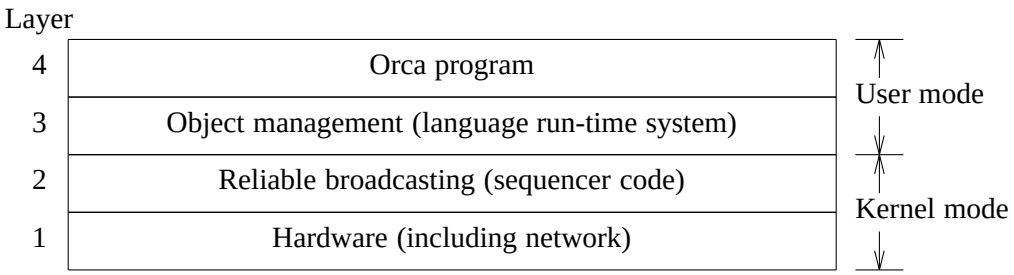


Fig. 7. Reliable broadcasting layer structure.

Layer 1 is the bare CPU and networking hardware. Our scheme is most efficient on networks that support broadcasting (sending a message to all machines) or multicasting (sending a message to a selected group of machines) in hardware, but it is not required. Ethernet, earth satellites, and cellular radio are examples of networks having broadcasting or multicasting. (For simplicity, we will henceforth use the term ‘‘broadcasting’’ to mean either one.) Broadcasting is assumed to be *unreliable*, that is, it is possible for messages to be lost.

Layer 2 is the software necessary to turn the *unreliable* broadcasting offered by layer 1 into *reliable* broadcasting. It is normally part of the operating system kernel. As a simple example of a possible (but highly inefficient) protocol, reliably broadcasting a message to n machines can be done by having the kernel send each machine, in turn, a point-to-point message, and then wait for an acknowledgement. This protocol takes $2n$ messages per reliable broadcast. Below we will describe a different protocol that does it in a fraction more than 2 messages on the average, instead of $2n$. The main issue to understand is that when layer 3 hands a message to layer 2 and asks for it to be reliably broadcast, layer 3 does not have to worry about how this is implemented or what happens if the hardware loses a message. All of this is taken care of by layer 2.

In addition to be being inefficient, the protocol that sends a point-to-point message to every machine has a more serious problem. When two machines, *A* and *B*, simultaneously do a broadcast, the results can be interleaved: some machines may get the broadcast from *A* before that from *B*, and other machines may receive them in the reverse order, depending on network topology, lost messages, and so on. This property makes programming difficult. In the protocol that we will describe below, this cannot happen. Either all the machines get *A*’s broadcast and then *B*’s broadcast, or all the machines get *B*’s broadcast and then *A*’s broadcast. Broadcasts are globally ordered, and it is guaranteed that all user processes get them in the same order.

Layer 3 is the language run-time system, which is usually a set of library procedures compiled into the application program. Conceptually, programmers can have variables and objects be PRIVATE or SHARED. The PRIVATE ones are not visible on other machines, so they can be accessed by direct memory reads and writes. SHARED objects are replicated on all machines. Reads to them are local, the same as reads to PRIVATE objects. Writes are done by reliable broadcasting. Objects are entirely passive; they contain only data, not processes.

Layer 4 provides language support. Although it is possible for programmers to use the distributed shared memory by making calls directly on layer 3, it is much more convenient to have language support. We have designed a language, called Orca [6] for parallel programming using distributed shared objects, and implemented a compiler for it. In it, programmers can declare shared objects, each one containing a data structure for the object and a set of procedures that operate on it. Operations on shared objects are indivisible and serializable. In other words, when multiple processes update the same object simultaneously, the final result is as if the updates were performed sequentially, in some unspecified order, with each update completed before the next one began.

Reliable Broadcasting

The heart of our proposal is the efficient implementation of indivisible, reliable broadcasting (layer 2). Once that has been achieved, the rest is relatively straightforward. In this section we will describe the mechanism used to achieve reliable broadcasting (in software) over an unreliable network. A more comprehensive description is given by Kaashoek et al. [7].

The hardware/software configuration required for reliable broadcasting is shown in Fig. 8. The circles represent the application programs and their run-time systems (layers 3 and 4). The kernel represents layer 2. The network is layer 1. The hardware of all the machines is identical, and they all run exactly the same kernel and application software.

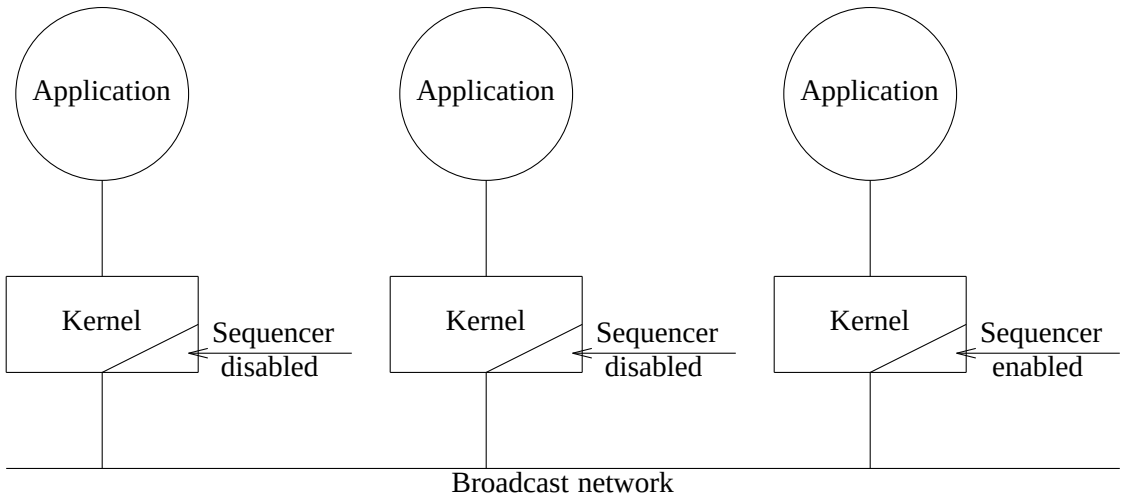


Fig. 8. System structure. Each kernel is capable of becoming the sequencer, but at any instant, only one of them functions as sequencer.

However, when the application starts up, one of the machines is elected as sequencer, like a committee electing a chairman [8]. If the sequencer machine subsequently crashes, the remaining members elect a new one. Many election algorithms are known. The one used in the IEEE 802.4 Token Bus standard, for example, is to pick the machine with the highest network address. We will not discuss this point further here.

The actual sequence of events involved to achieve reliable broadcasting can be summarized as follows. First, the user traps to the kernel, passing it the message. Second, the kernel accepts the message and blocks the user. Third, the kernel sends the message to the sequencer using an ordinary point-to-point message. Fourth, when the sequencer gets the message, it allocates the next available sequence number, puts the sequencer number in a header field reserved for it, and broadcasts the message (and sequence number). Fifth and last, when the sending kernel sees the broadcast message, it unblocks the calling process to let it continue execution.

Let us now consider these events in more detail. When an application process executes a BROADCAST primitive, a trap to its kernel occurs. The kernel then blocks the caller and builds a message containing a kernel-supplied header and the application-supplied data. The header contains the message type (*Request for Broadcast* in this case), a unique message identifier (used to detect duplicates), the number of the last broadcast received by the kernel (a kind of piggybacked acknowledgement), and some other information.

The kernel sends the message to the sequencer using a normal point-to-point message, and simultaneously starts a timer. If the broadcast comes back before the timer runs out (normal case), the sending kernel stops the timer and returns control to the caller. In practice, this case happens well over 99% of the time, because LANs are highly reliable.

On the other hand, if the broadcast has not come back before the timer expires, the kernel assumes either the message or the broadcast has been lost. Either way, it retransmits the message. If the original message was lost, no harm has been done, and the second (or subsequent)

attempt will trigger the broadcast in the usual way. If the message got to the sequencer and was broadcast, but the sender missed the broadcast, the sequencer will detect the retransmission as a duplicate (from the message identifier) and just tell the sender that everything is all right. The message is not broadcast a second time.

A third possibility is that a broadcast comes back before the timer runs out, but it is the wrong broadcast. This situation arises when two processes attempt to broadcast simultaneously. One of them, *A*, gets to the sequencer first, and its message is broadcast. *A* sees the broadcast and unblocks its application program. However its competitor, *B*, sees *A*'s broadcast and realizes that it has failed. Nevertheless, *B* knows that its message probably got to the sequencer (since lost messages are rare) where it will be queued, and broadcast next. Thus *B* accepts *A*'s broadcast and continues to wait for its own broadcast to come back or its timer to expire.

Now consider what happens at the sequencer when a *Request for Broadcast* arrives there. First a check is made to see if the message is a retransmission, and if so, the sender is informed that the broadcast has already been done, as mentioned above. If the message is new (normal case), the next sequence number is assigned to it, and the sequencer counter incremented by 1 for next time. The message and its identifier are then stored in a *history buffer*, and the message is then broadcast. (The management of the history buffer will be described shortly.) The message is also passed to the application running on the sequencer's machine (because the broadcast does not cause an interrupt on the machine that issued the broadcast).

Finally, let us consider what happens when a kernel receives a broadcast. First, the sequence number is compared to the sequence number of the most recently received broadcast. If the new one is 1 higher (normal case), no broadcasts have been missed so the message is passed up to the application program. If the application program is multithreaded, usually one thread will be waiting for the message; if it is single threaded, the kernel waits for the program to try to receive a message.

Now suppose that the newly received broadcast has sequence number 25, while the previous one had number 23. The kernel is immediately alerted to the fact that it has missed number 24, so it sends a point-to-point message to the sequencer asking for a private retransmission of the missing message. The sequencer fetches the missing message from its history buffer and sends it. When it arrives, the receiving kernel processes 24 and 25, passing them to the application program in numerical order. Thus the only effect of a lost message is a minor time delay. All application programs see all broadcasts in the same order, even if some messages are lost.

The reliable broadcast protocol is illustrated in Fig. 9. Here the application program running on machine *A* passes a message, *M*, to its kernel for broadcasting. The kernel sends the message to the sequencer, where it is assigned sequence number 25. The message (containing the sequence number 25) is now broadcast to all machines and also passed to the application program running on the sequencer itself. This broadcast message is denoted by *M25* in the figure.

The *M25* message arrives at machines *B* and *C*. At machine B the kernel sees that it has already processed all broadcasts up to and including 24, so it immediately passes *M25* up to the application program. At *C*, however, the last message to arrive was 23 (24 must have been lost), so *M25* is buffered in the kernel, and a point-to-point message requesting 24 is sent to the sequencer. Only after the reply has come back and been given to the application program will *M25* be passed upwards as well.

Now let us look at the management of the history buffer. Unless something is done to prevent it, the history buffer will quickly fill up. However, if the sequencer knows that all machines have correctly received broadcasts, say, 0 through 23, it can delete these from its history buffer.

Several mechanisms are provided to allow the sequencer to discover this information. The basic one is that each *Request for Broadcast* message sent to the sequencer carries a piggybacked acknowledgement, *k*, meaning that all broadcasts up to and including *k* have been correctly received. This way, the sequencer can maintain a piggyback table, indexed by machine number, telling for each machine which broadcast is the last one received. Whenever the history buffer begins to fill up, the sequencer can make a pass through this table to find the smallest value. It can then safely discard all messages up to and including this value.

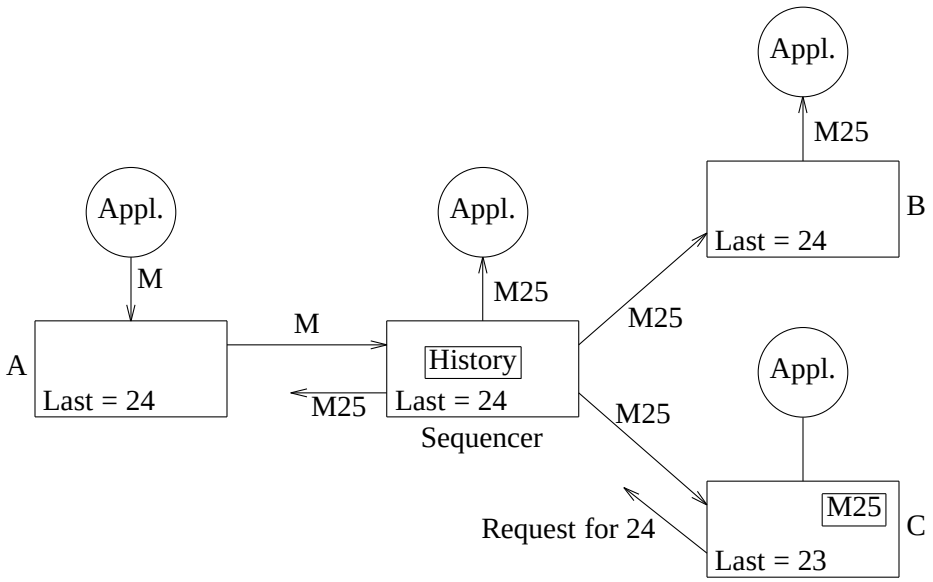


Fig. 9. The application of machine *A* sends a message to the sequencer, which then adds a sequence number (25) and broadcasts it. At *B* it is accepted, but at *C* it is buffered until 24, which was missed, can be retrieved from the sequencer.

If one machine happens to be silent for an unusually long period of time, the sequencer will not know what its status is. To inform the sequencer, it is required to send a short acknowledgement message when it has sent no broadcast messages for a certain period of time. Furthermore, the sequencer can broadcast a *Request for Status* message, which directs all other machines to send it a message giving the number of the highest broadcast received in sequence. In this way, the sequencer can update its piggyback table and then truncate its history buffer.

Although in practice *Request for Status* messages are rare, they do occur, and thus raise the mean number of messages required for a reliable broadcast slightly above 2, even when there are no lost messages. The effect increases slightly as the number of machines grows.

There is a subtle design point concerning this protocol that we would like to clarify. There are two ways to do the broadcast. In method 1 (described above), the user sends a point-to-point message to the sequencer, which then broadcasts it. In method 2, the user broadcasts the message, including a unique identifier. When the sequencer sees this, it broadcasts a special *Accept* message containing the unique identifier and its newly assigned sequence number. A broadcast is only ‘‘official’’ when the *Accept* message has been sent.

These protocols are logically equivalent, but they have different performance characteristics. In method 1, each message appears in full on the network twice: once to the sequencer and once from the sequencer. Thus a message of length *m* bytes consumes 2*m* bytes worth of network bandwidth. However, only the second of these is broadcast, so each user machine is only interrupted once (for the second message).

In method 2, the full message only appears once on the network, plus a very short *Accept* message from the sequencer, so only half the bandwidth is consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *Accept*. Thus method 1 wastes bandwidth to reduce interrupts compared to method 2.

We have implemented both methods and are now running experiments comparing them. Depending on the results of these experiments, we may go to a hybrid scheme, using method 1 for short messages and method 2 for long ones.

In summary, this protocol allows reliable broadcasting to be done on an unreliable network in just over two messages per reliable broadcast. Each broadcast is indivisible, and all applications all messages in the same order, no matter how many are lost. The worst that can happen is

that a short delay is introduced when a message is lost, which rarely happens. If two processes attempt to broadcast at the same time, one of them will get to the sequencer first and win. The other will see a broadcast from its competitor coming back from the sequencer, and will realize that its request has been queued and will appear shortly, so it simply waits.

Comparison. Kaashoek et al., [7], give a detailed comparison of our scheme with other published protocols for doing reliable broadcasting. Here we will compare our scheme with some of the majors ones. There are three other kinds of work to which we can compare ours:

- Distributed shared memory
- Broadcasting to achieve fault tolerance
- Broadcasting as an operating system service

Most research on distributed shared memory is based on the work of Li and Hudak [3]. In their method, fixed-size pages are moved around the network in point-to-point messages. If broadcasting is used at all, it is only for invalidating pages. In contrast, in our method broadcasting is the basis of the entire system. Furthermore, messages are variable-sized, containing only the data that are needed. Since most of the time, the amount of data needed is much less than a page, being able to send short messages is a significant performance improvement.

Some recent work in distributed shared memories has emphasized weakening the semantics of what shared memory means [5]. We consider this unacceptable, since the goal is to give the programmer an easy-to-understand model to work with, and weak-semantics systems put all the burden of coherency back on the programmer.

Other work has concentrated on using broadcasting to achieve fault tolerance. Chang and Maxemchuk [9] describe a family of protocolss for reliable broadcasting, of which non-fault tolerance is a special case. Their protocol for the non-fault tolerant cases uses a sequencer, like ours, but they have the nodes ordered in a logical ring, with the sequencer advancing along the ring with every message sent. This motion of the sequencer is almost free when the traffic is heavy, but costs an extra message per broadcast when it is not. On the average, their protocol requires 2 to 3 messages per reliable broadcast, whereas ours does it in just a fraction over 2.

In addition, their protocol uses more storage because they have to store the history buffer on all machines, due to the moving sequencer. With a 1 megabyte history buffer and 100 machines, we need 1M of memory for the history buffer and they need 100M.

Finally, in our method 1, each reliable broadcast uses one point-to-point message and one broadcast message. With $n >> 1$ machines, we generate about n interrupts per reliable broadcast. In their protocol (similar to what we have called method 2), all messages are broadcast, so they need between $2n$ and $3n$ interrupts per reliable broadcast. When there are hundreds of broadcasts per second, their scheme uses much more CPU time than ours.

Another family of fault-tolerant protocols is the ISIS system of Birman and Joseph [10]. They use a distributed two-phase commit protocol to achieve global ordering, something we achieve in 2 messages per broadcast. Since they realize that their protocol is inefficient, they propose alternative protocols with weaker semantics. Our method shows that it is not necessary to weaken and complicate the semantics to achieve efficiency. In ISIS' favor, however, is that it provides a high degree of fault tolerance, which our basic protocol does not. The sidebar on page 21 shows that our method can also provide fault tolerance, if desired. Furthermore, ISIS supports simultaneous use of overlapping process groups, whereas in our scheme groups are disjoint. Each application forms its own group and no attempt is made to provide global ordering between different applications.

A few researchers have proposed providing broadcasting as an operating system service. In this category is the system of Cheriton and Zwaenepoel [11], which supports broadcasting, but does not guarantee reliability. It has the problem of making the programmer's job more difficult compared to a system in which broadcasts are guaranteed to be reliable.

Performance of Reliable Broadcasting

We have modified the Amoeba kernel to support our reliable broadcast protocol. It runs on a collection of 16 MHz Motorola 68030 processors connected by a 10 Mbps Ethernet. We have run various experiments on this system to measure its performance. In the first experiment, one process continuously broadcasts null messages as fast as it can in order to measure the maximum broadcast rate by a single process. In this experiment we achieved 370 reliable broadcasts per second with up to 16 processors.

It should be pointed out that this test is the worst possible case. Since all machines but one are silent, they are not sending piggybacked acknowledgements back to the sequencer. Without these acknowledgements, the sequencer must send out a *Request for Status* every 64 messages.

It is instructive to note that even though we are measuring broadcasting, the performance is better than most (point-to-point) RPC systems, partly because our protocol requires only 2 messages per broadcast, whereas RPCs often require 3, the last being an acknowledgement.

As an aside, these results differ from our earlier results because we have now designed and implemented a new kernel that can handle broadcasting over an arbitrary internetwork consisting of LANs and buses. This scheme also supports transparent process migration and automatic network reconfiguration and management. The differences between the results here and the previously published ones are entirely accounted for by these changes.

The second experiment consists of having not one, but multiple processes broadcasting at the same time, to see what the effect of contention is. In this experiment, we varied the number of senders from 2 to 14, with the receiving group equal to the number of senders. The results are as follows:

# Senders	**Broadcasts/sec**
2	714
3	769
4	769
5	793
6	789
7	795
8	800
9	782
10	781
11	780
12	780
13	718
14	710

Multiple senders get a higher throughput than just one because if two machines send messages to the sequencer almost simultaneously, the first one to arrive will be broadcast first, but the second one will be buffered and broadcast immediately afterwards. This simple form of pipelining increases the parallelism of the system, and thus increases the broadcast rate. As the number of senders increases, performance drops slightly due to contention for the Ethernet. We were unable to make consistent measurements for 15 and 16 processors due to technical limitations of our equipment.

A word about scaling is appropriate here. It has been pointed out to us that the sequencer will become a bottleneck in very large systems. While this is true, few current systems or applications suffer from this limit. With current technology (2 MIPS CPUs and 10 Mbps Ethernet), we can support on the order of 800 reliable broadcasts per second, as shown above. Since broadcast messages are usually short, there is bandwidth to spare. For many problems, the read to write ratio is quite high. Suppose (conservatively) that 90 percent of the operations are reads and 10 percent are writes. Then we can support 8000 operations per second on shared objects (7200 reads, done locally, and 800 writes, done by broadcasting). We know of very few

applications that would be hindered by being able to perform only 8000 operations per second on shared objects. Furthermore, with 20 MIPS RISC processors and 100 Mbps fiber optic networks, this limit will be closer to 80,000 operations on shared objects per second within a few years.

Object Management

On top of the broadcast layer is the object management layer, implemented by a package of library procedures (in user space). This layer manages shared objects. Our design is based on the explicit assumption that shared objects are read much more often than they are written. Ratios of 10:1 or even 100:1 are not at all unusual. Therefore, we have chosen to replicate each object on all machines that use the object. (Note that multiple, independent applications may be running at the same time, so not every machine needs every object.) All replicas have equal status: there is no concept of a primary object and secondary copies of it.

Two operations are defined on objects: READ and WRITE. READs are done on the local copy, without any network traffic. A READ on a shared object is only slightly more expensive than a READ on a PRIVATE object (due to some locking). A WRITE to a shared object can be done by sending a reliable broadcast with the new value of the object, in whole or in part, or it can be done by sending an operation code and some parameters, letting each machine recompute the new value. The former strategy is attractive for small objects; the latter one for large objects; it is up to the run-time system to pick one. Since all machines process all broadcasts in the same order, when equilibrium is reached, all objects will settle down to the same value.

This scheme does not provide complete memory coherence because if machine *A* initiates a reliable broadcast to update a shared object, and machine *B* reads the (local copy of the) object a nanosecond later, *B* will get the old value. On the other hand, it does provide for indivisible update and serializability, which is almost as good, as can easily be seen in the following example. Consider a multiprocessor with a true shared memory. At a certain moment, process *A* wants to write a word, and process *B* wants to read it. Since the two operations may take place a microsecond apart, the value read by *B* depends on who went first. Despite the memory being coherent, the value read by *B* is determined by the detailed timing. Our shared object model has a similar property. In both cases, programs whose correct functioning depends on who wins the race to memory are living dangerously, at best. Thus although our memory model does not exhibit true coherence, in reality, serializability plus global message ordering are sufficient properties, and we do have them.

Orca

While it is possible to program directly with shared objects, it is much more convenient to have language support for them. For this reason, we have designed the Orca parallel programming language and written a compiler for it.

There are four guiding principles behind the Orca design:

- Transparency
- Semantic simplicity
- Serializability
- Efficiency

By *transparency* we mean that programs (and programmers) should not be aware of where objects reside. Location management should be fully automatic. Furthermore, the programmer should not be aware of whether the program is running on a machine with physical shared memory or one with disjoint memories. The same program should run on both, unlike nearly all other languages for parallel programming, which are aimed at either one or the other, but not both. (Of course one can always simulate message passing on a multiprocessor, but this is far from optimal.)

Semantic simplicity means that programmers should be able to form a simple mental model of how the shared memory works. Incoherent memory, in which reads to shared data sometimes return good values and sometimes stale (incorrect) ones is ruled out by this principle.

In a parallel system, many events happen simultaneously. By making operations *serializable*, we guarantee that operations on objects are indivisible, and that the observed behavior is the same as some sequential execution would have been. Operations on objects are guaranteed not to be interleaved, which contributes to semantic simplicity, as does the fact that all machines see the same sequence of serial events. Of course, the compiler and run-time system do not actually serialize events any more than they have to to guarantee the semantics.

Finally, *efficiency* is also important, since we are proposing a system that can actually be used for solving real problems.

Now let us look at the principal aspects of Orca that relate to parallelism and shared objects. Orca is a procedural language whose sequential constructs are roughly similar to languages like C or Modula 2. Parallelism is based on two orthogonal concepts: *processes* and *objects*. Processes are active entities that execute programs. They can be created and destroyed dynamically. It is possible to read in an integer, *n*, then execute a loop *n* times, creating a new process on each iteration. Thus the number of processes is not fixed at compile time, but is determined during execution.

The Orca construct for creating a new process is the

```
fork func(param, ...)
```

statement, which creates a new process running the procedure *func* with the specified parameters. The user may specify which processor to use, or let the run-time system use its own load-balancing heuristics. Objects may be passed as parameters (call by reference). A process may fork many times, passing the same objects to each of the children. This is how objects come to be shared among a collection of processes. There are no global objects in Orca.

Objects are passive. They do not contain processes or other active elements. Each object contains some data structures, along with definitions of one or more operations that use the data structures. The operations are defined by Orca procedures written by the programmer. An object has a specification part and an implementation part, similar in this respect to Ada® packages or Modula 2 modules.

A common way of programming in Orca is the Replicated Worker Paradigm [12]. In this model, the main program starts out by creating a large number of identical worker processes, each getting the same objects as parameters, so they are shared among all the workers. Once the initialization phase is completed, the system consists of the main process, along with some number of identical worker processes, all of which share some objects. Processes can perform operations on any of their objects whenever they want to, without having to worry about all the mechanics of how many copies are stored and where, how updates take place, which synchronization technique is used, and so on. As far as the programmer is concerned, all the objects are effectively located in one big shared memory somewhere, but are protected by a kind of monitor that prevents multiple updates to an object at the same time.

As a minimal example of an object specification, consider a simple object consisting of an integer variable with two operations on it: *read* and *write*. If this object is subsequently shared among multiple processes, any of them can read or write the value of the integer. The Orca specification part looks like this:

```
object specification SharedInt;
    operation read():integer;
    operation write(val: integer):integer;
end;
```

The implementation part looks like this:

```
object implementation SharedInt;
    n: integer;  # the value

    operation read(): integer;
    begin
        return n;
```

```
    end;

    operation write(val: integer);
    begin
        n := val;
    end;
end;
```

To declare and use an object of type *SharedInt*, the programmer might write:

```
s: SharedInt;
i: integer;  # ordinary integer

s$write(100);  # set object to 100
i := s$read();  # set i to 100
```

Although the programming style suggested by this trivial example is sufficient for some programs, for many others some kind of synchronization method is required. A common example is *barrier synchronization*, in which *n* workers are busy computing something, and only when all of them are finished may the next step begin. Synchronization in Orca is handled with *guarded commands*, in which an operation consists of a number of (guard, statement) pairs. Each guard is a side-effect-free Boolean expression, and each statement is an arbitrary piece of sequential Orca code. When the operation is invoked, the language run-time system evaluates the guards one at a time (in an unspecified order), and as soon as it finds one that is true, the corresponding statement is executed. For example, to implement barrier synchronization, the main process could create a shared object with operations to initialize it, increment it, and synchronize on it (i.e., wait until it reached *n*). The main process would initialize it to 0, then fork off all the workers. When each worker finished its work, it would invoke the *increment* operation, and then the *synchronize* operation. The latter would block the calling process until the value reached *n*, at which point all the processes would be released to start the next phase.

As a slightly more elaborate example, consider an implementation of the Traveling Salesman Problem (TSP) in Orca. In this problem, the computer is a given a starting city and a list of cities to visit, and it has to find the shortest path that visits each city exactly once and returns to the starting city.

The usual algorithm for solving TSP is branch and bound. Suppose the starting city is New York, and the cities to be visited are London, Sydney, Tokyo, Nairobi, and Rio de Janeiro. The main program starts out by computing some possible path (e.g., using the closest city next algorithm), and determining its length. Then it initializes a soon-to-be-shared object, *BestPath*, containing this path and its length. As the program executes, this object will always contain the best path found so far, and its length. Next, it forks off some number of workers, each getting the shared object as parameter, as shown in Fig. 10.

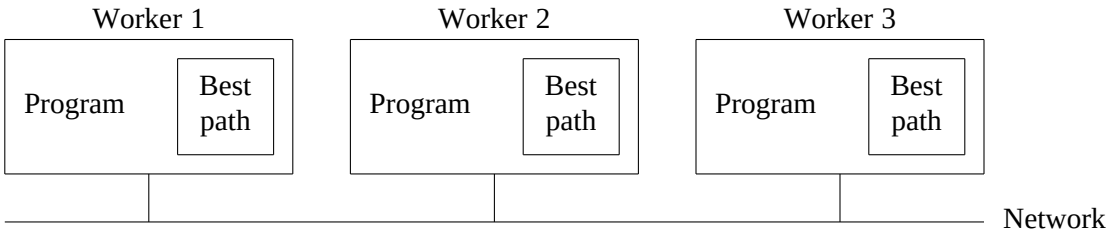


Fig. 10. Model of TSP program using replicated worker paradigm.

Each worker is given a different partial path to investigate. The first one tries paths beginning New York-London, the second one tries paths beginning New York-Sydney, the third one

tries paths beginning New York-Tokyo, and so on. Very roughly, the algorithm used by a worker that is given a partial path plus *k* cities to visit is to first check if the length of the partial path is longer than the best complete path found so far. If so, the process terminates itself. If the partial path is still a potential candidate, the process generates *k* new partial paths to be investigated, one per city in the list, and forks these off to *k* new workers. To avoid forking off too many processes, when the list of remaining cities to be visited is less than *n*, the process tries all the possible combinations itself. Other optimizations are also used. Whenever a new best path is found, an update operation on the shared object *BestPath* is executed, to replace the previous best path by the new one. When the Orca compiler detects an assignment to a variable in a shared object, it generates code to cause the run-time system to issue a reliable broadcast of the new value. In this manner, all the details of managing shared objects are hidden from the programmer.

A moment's thought will show that reads of *BestPath* will occur very often, while writes will hardly occur at all, certainly not after the program has been running for a while and has found a path close to the optimal one. Remember that reads are done entirely locally on each machine, whereas writes require a reliable broadcast. The net result is that the vast majority of operations on the shared object do not require network traffic, and the few that do only take two messages. Consequently, the solution is highly efficient. We have achieved almost linear speedup with 10 processes. In fact, actual measurements have shown the distributed version to outperform one run on a shared memory machine (without caching), because the latter suffers from serious memory contention as large numbers of processors constantly try to read the memory word containing the length of the best path so far.

Comparison. It is instructive to briefly compare Orca with some alternative approaches to parallel programming. There are a large number of languages based on message passing, which is of a conceptually lower level than shared objects. The approach usually used in languages that support shared memory is the use of semaphores or monitors to protect critical regions. Both of these work adequately on shared memory machines with a small number of processors, but poorly on large distributed systems because they use a locking scheme that is inherently centralized.

A system that is somewhat similar to Orca is Emerald [13]. Like Orca, it supports shared objects. However, unlike Orca, it does not replicate objects. This means that when a caller on machine 1 invokes an operation on an object located on machine 2 using a parameter on machine 3, messages must be sent to collect all the necessary information in the same place. There is no automatic migration, so if the programmer does not arrange for things that go together to be colocated, execution will be inefficient.

Alternatively, for efficiency, on each call, the programmer can specify that the parameter objects are to be sent to the machine where the object resides and to remain there after the invocation. The programmer can also specify whether the result is to remain there or not. In a truly transparent system these issues would not arise.

In Orca, in contrast, it is up to the run-time system to decide whether it wants to replicate objects or not, and if so, where. Using the broadcast system described above, all objects are replicated on machines that need them (but other Orca implementations do it differently). In any event, it is not the programmer's responsibility; object management is handled automatically, and the decision whether to perform operations locally or remotely is made by the system. The Orca approach comes much closer to providing the semantics of a shared-memory multiprocessor.

Yet another shared object scheme is Linda [12]. Linda supports the concept of a shared tuple space that is equally accessible to processes on all machines. Linda is fully location transparent, but the primitives are low level: inserting and deleting tuples. In Orca, in contrast, the operations on shared objects can be simple or arbitrarily complex, as the programmer wishes.

In summary, Orca supports the model of having dynamically created parallel processes perform user-defined operations on shared objects. These objects are implemented by maintaining identical copies on all machines. An operation that reads an object, but does not change it, is done entirely locally, with no network traffic. The efficiency of this operation is only slightly

worse than an operation on an unshared object.

An operation that modifies an object is handled differently. The operation name and parameters are put in a message which is passed to the run-time system (object management layer) for transmission to all other machines in one indivisible reliable broadcast. The method by which the kernel provides this abstraction was discussed in detail above. When the message arrives, the operation is carried out locally. This method is preferable to carrying out the operation on the sending machine and broadcasting the new object to reduce the size of the message.

This division of labor between the layers yields a great conceptual simplicity: the programmer defines and invokes operations on shared objects, the run-time system handles reads and writes on these objects, and the reliable broadcast layer implements indivisible updates to objects using the sequencer protocol. Parallel applications in Orca

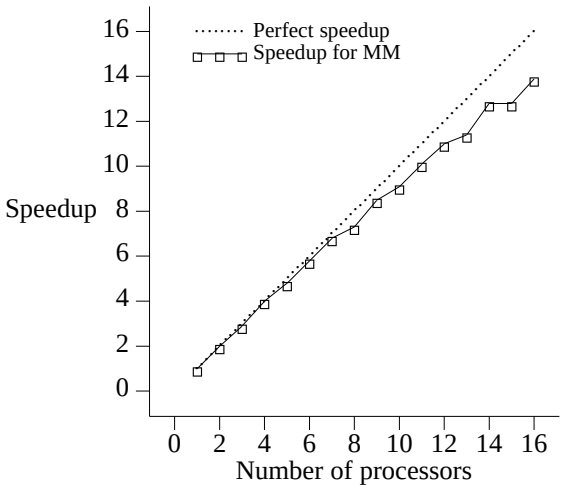
Orca is a procedural, type-secure language intended for implementing parallel applications on distributed systems [1-3]. Below, we will briefly discuss four examples of problems it has been used for and give their measured speedups on the broadcast system.

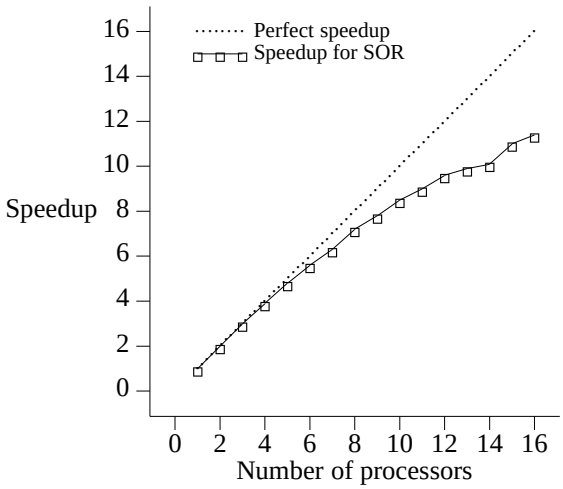
Matrix multiplication (MM) is an example employing ‘‘trivial parallelism.’’ Each processor is assigned a fixed portion of the result matrix. Once the work-to-do has been distributed, all processors can proceed independently from each other. The speedup is not perfect because it takes some time to initialize the source matrices (of size 250×250) and the portions are fixed but not necessarily equal if the matrix size is not divisible by the number of processors.

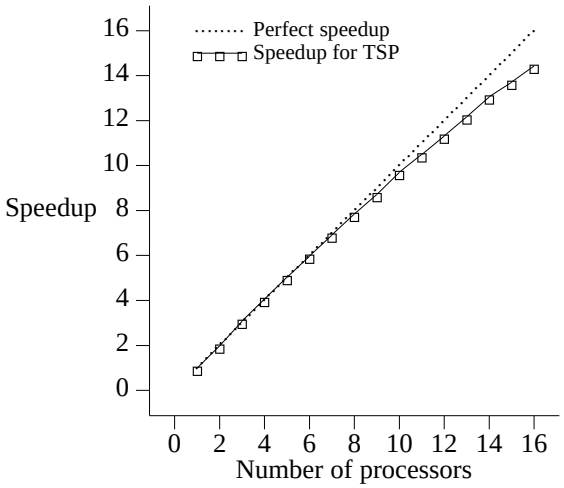
The Traveling Salesman Problem (TSP) is discussed in detail in the paper. Its main characteristic is the shared variable containing the current best solution. This variable is stored in a shared object that is read very frequently. If a new better route is found, all copies of the object are updated immediately, using the efficient broadcast protocol. It is important that the updating takes place immediately, lest some processors continue to use an inferior bound, reducing the effectiveness of the pruning. TSP achieves a speedup close to linear.

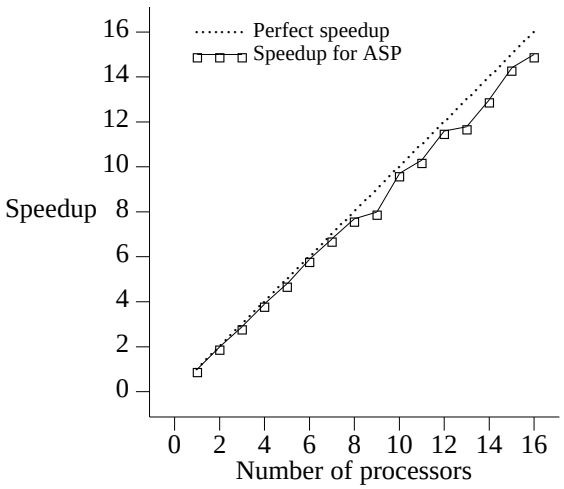
In the all-pairs shortest paths (ASP) problem, communication overhead is much higher. ASP uses an iterative algorithm. Before each iteration, some process selects one row of the distances matrix as pivot row and sends it to all other processes. If implemented with point-to-point messages, the communication overhead would be linear with the number of processors. With our multicast protocol, however, the overhead is reduced to a few messages, resulting in high speedups.

Of course, not all parallel applications benefit from broadcasting. In successive overrelaxation (SOR), each processor mainly communicates with its neighbors. SOR is a worst-case example for our system, because point-to-point messages between neighboring nodes are implemented as broadcast messages received by all nodes. Still, the program achieves a reasonable speedup.


Perfect speedup
Speedup for MM
Speedup
Number of processors
0 2 4 6 8 10 12 14 16


Perfect speedup
Speedup for SOR
Speedup
Number of processors
0 2 4 6 8 10 12 14 16


Perfect speedup
Speedup for TSP
Speedup
Number of processors
0 2 4 6 8 10 12 14 16


Perfect speedup
Speedup for ASP
Speedup
Number of processors
0 2 4 6 8 10 12 14 16

Measured performance for four Orca programs. Each graph shows the speedup of the parallel Orca program on N cpus over the same program running on 1 CPU. (a) Matrix multiplication, using input matrices of size 250*250. (b) The Traveling Salesman Problem, averaged over three randomly generated graphs with 12 cities each. (c) The All-pairs Shortest Paths problem, using an input graph with 300 nodes. (d) Successive Overrelaxation, using a grid with 80 columns and 242 rows.

Summary

To summarize, in this paper we have discussed multiprocessors (with shared memory) and multicomputers (no shared memory). The former are easy to program but hard to build, while the latter are easy to build but hard to program. We then introduced a new model that is easy to program, easy to build, and has an acceptable performance on problems with a moderate grain size in which reads are much more common than writes.

The essence of our approach is to implement reliable broadcasting as a distinct semantic layer and then use this layer to implement shared objects. Reliable broadcasting is achieved by having users send their messages to a sequencer, which then numbers them sequentially and broadcasts them. Machines that miss a message can get it by asking the sequencer, which maintains a history buffer. This scheme requires slightly over 2 messages per reliable broadcast. We have built a kernel based on these principles and measured its performance at up to 1500 reliable broadcasts per second.

We have also designed and implemented a shared object layer and a language, Orca that allows programmers to declare objects shared by multiple processes. Objects are replicated on all the machines that need to access them. When the language run-time system needs to read a shared object, it just uses the local copy. When it needs to update a shared object, it reliably broadcasts the new object (or operation code and parameters). This scheme is simple and has a semantic model that is easy for programmers to understand. It is also efficient because the amount of data broadcast is exactly what is needed. It is not constrained to be 1K or 8K because the page size happens to be that, as in the work of Li and Hudak. Most of our broadcasts are much smaller (typically a few hundred bytes, at most).

Our conclusion is that the use of reliable broadcasting to support replicated, shared objects is a good way to approach parallel programming. It is simple to understand and efficient to implement. The use of a language like Orca makes it easier to use shared objects, but even without Orca, we believe that this paradigm offers a new and effective way to exploit parallelism in future computing systems.

Acknowledgements

We would like to thank Erik Baalbergen, Dick Grune, and Wiebren de Jonge for carefully reading the paper and many making helpful suggestions.

References

1. A.D. Birrell and B.J. Nelson, ‘‘Implementing Remote Procedure Calls,’’ *ACM Trans. Computer Systems*, Vol. 2, No. 1., Feb, 1984, p. 39-59.

2. R.J. Swan, S.H. Fuller, and D.P. Siewiorek, ‘‘Cm* —A Modular, MultiMicroprocessor System,’’ *Proc. Nat. Computer Conf.*, 1977, pp. 645-655.

3. K. Li and P. Hudak ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Trans. Computer Systems* Vol. 7, No. 4, Nov. 1989, pp. 321-359.

4. J.K. Bennet, J.B. Carter, and W. Zwaenepoel, ‘‘Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,’’ *2nd ACM Symp. on Prin. and Prac. of Parallel Prog.*, March 1990, pp. 168-177.

5. R.G. Minnich and D.J. Farber, ‘‘Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory,’’ *Proc. 10th Int’l Conf. on Distrib. Computer Syst.*, IEEE, 1990, pp. 468-475.

6. H.E. Bal, *Programming Distributed Systems* Summit NJ: Silicon Press, 1990.

7. M.F. Kaashoek, A.S. Tanenbaum, S. Flynn Hummel, and H.E. Bal, ‘‘An Efficient Reliable Broadcast Protocol,’’ *Operating Systems Review*, Vol. 23, Oct. 1989, pp. 5-19.

8. M.F. Kaashoek and A.S. Tanenbaum, ‘‘Group Communication in the Amoeba Distributed Operating System,’’ *Proc. 11th Int’l Conf. on Distrib. Compute Syst.*, IEEE, 1991, pp. 222-230.

9. J. Chang and N.F. Maxemchuk, ‘‘Reliable Broadcast Protocols,’’ *ACM Trans. Computer Sysems.*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.

10. K.P. Birman, and T.A. Joseph, ‘‘Reliable Communication in the Presence of Failures,’’ *ACM Trans. on Computer Syst.*, Vol. 5, No. 1, Feb. 1987, pp. 47-76.

11. D.R. Cheriton and W. Zwaenepoel, ‘‘Distributed process Groups in the V Kernel,’’ *ACM Trans. Computer Systems*, Vol. 3, No. 2, May 1985, pp. 77-107.

12. N. Carriero and D. Gelernter, ‘‘Linda in Context,’’ *Commun. ACM*, Vol. 32, No. 4, Apr. 1989, pp. 444-458.

13. E. Jul, H. Hutchinson, and A. Black, ‘‘Fine-Grained Mobility in the Emerald System,’’ *ACM Trans. Computer Syst.*, Vol. 6, No. 1, Feb. 1988, pp. 109-133.

14. H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, ‘‘Programming Languages for Distributed Computing Systems,’’ *ACM Computing Surveys*, Vol. 21, No. 2, Sept. 1989, pp. 261-322.

15. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ‘‘Experience with Distributed Programming in Orca,’’ *Proc. IEEE CS Int. Conf. on Computer Languages*,’’ March 1990, pp. 79-89.

16. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ‘‘A Distributed Implementation of the Shared Data-object Model,’’ *Proc. USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Oct. 1989, pp. 1-9.

17. A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson: ‘‘A Practical Toolkit for Making Portable Compilers,’’ *Commun. ACM*, Vol 26, No. 9, Sept. 1983, pp. 654-660.