View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at VU

PARALLEL PROGRAMMING USING SHARED OBJECTS AND BROADCASTING

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. The basic idea is to allow programmers to define objects upon-which user-defined operations are performed, in effect, abstract data types. Each object is replicated on each machine that needs it. Reads are done locally, with no network traffic. Writes are done by a reliable broadcast algorithm. A language for parallel programming, Orca, based on distributed shared objects has been designed, implemented, and used for some applications. Its implementation uses the reliable broadcast mechanism. For applications with a high read/write ratio to the shared objects, we show that our approach can frequently achieve close to linear speedup with up to 16 processors.

BLURB FOR CENTER COLUMN

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.

Keywords: broadcasting, shared objects, distributed systems

PARALLEL PROGRAMMING USING SHARED OBJECTS AND BROADCASTING

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

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

As computers continue to become less expensive, there is increasing interest in harnessing multiple CPUs together to build large, powerful distributed and 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 both 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.

Multiprocessors

Systems with multiple processors can be divided into two categories: those that contain physical shared memory, called *multiprocessors*, and those that do not, called *multicomputers*. Multiprocessors have 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. Communication is via the shared memory. Multicomputers do not have shared memory and must communicate by 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 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. To reduce the amount of bus traffic, the processors are usually equipped with a cache for storing copies of the most recently accessed memory words. The caches are kept consistent by special hardware that monitors bus traffic and updates or invalidates copies of words that are about to be modified by another processor. A bus-based multiprocessor is shown in Fig. 1(a).

The second general approach to building a multiprocessor is using some kind of switching network, such as the *crossbar switch*, shown in Fig. 1(b). 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. The problem with the crossbar switch is that to

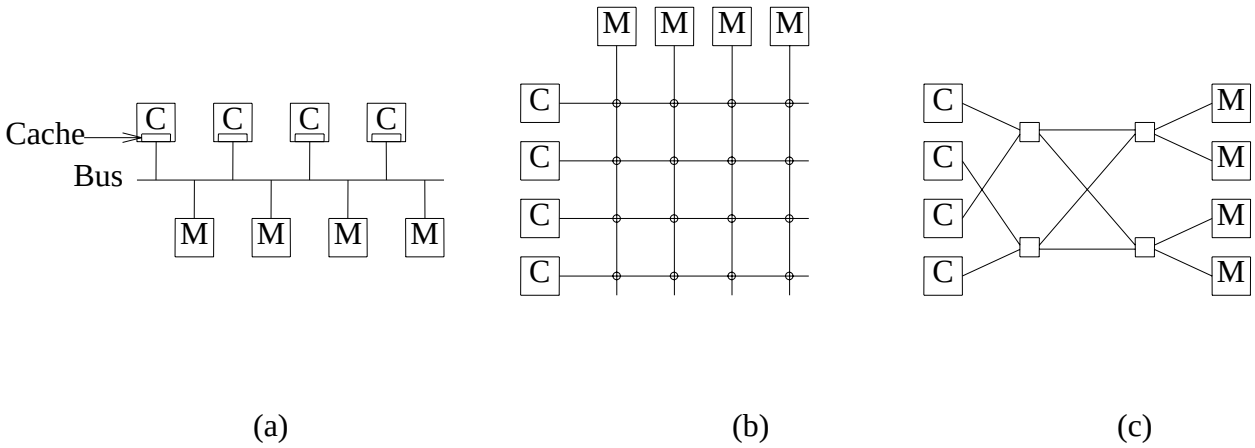


Fig. 1. Three multiprocessor designs. (a) A single-bus. (b) Cross bar switch. (c) Omega network. C indicates CPU (possibly with cache); M indicates memory module.

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 a switching network, for example, the omega network, shown in Fig. 1(c). In this figure, the CPUs are on the left and the memories are on the right. The omega network is a sophisticated (hardware) 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. The problem with this approach is that a substantial delay is incurred and that a large number ($n \log_2 n$) of switches is needed.

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 various interconnection schemes for multiprocessors, there are various interconnection schemes for multicomputers. Figure 2(a) shows a

simple bus-based multicomputer. Each CPU has its own local memory, which is not accessible by the others. In Fig. 2(b) and Fig. 2(c) we see other interconnection schemes, a grid and a hypercube, respectively. 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.). A hypercube is an *k*-dimensional cube. One can imagine a 4-dimensional hypercube as a pair of ordinary cubes with the corresponding vertices connected, as shown in Fig. 2(c), where the lower left nodes at the rear have been omitted for clarity. In a hypercube, the number of connections to each processor grows logarithmically with the number of CPUs, but the worst case delay is also logarithmic. With a grid, the delay grows as the square root of the number of processors, which is much worse for large systems.

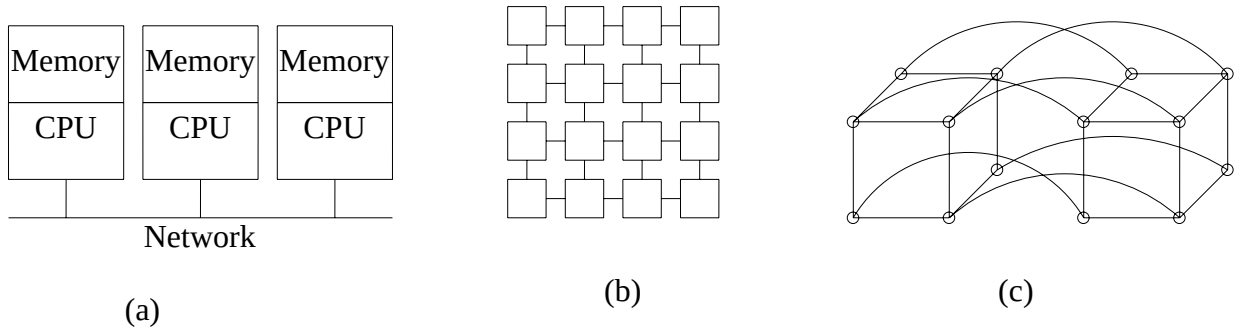


Fig. 2. Multicomputers. (a) Bus. (b) Grid. (c) Hypercube.

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. Many variations on this theme exist. Choices include: blocking vs. nonblocking, and buffered or nonbuffered among others.

A serious problem with message passing is that, conceptually, it is really input/output, which makes programming complicated. While all programs do I/O, it does not have the abstraction power of say, procedures or abstract data types on which to build a system, and in most programming languages it is almost an afterthought. To hide the bare input/output, Birrell and Nelson [1] proposed a scheme called *remote procedure call* (RPC) which hides the message passing to a limited extent. With RPC, the caller calls a *stub* (library) routine, which sends the message. Nevertheless, RPC introduces its own problems, such as restricting the use of general pointers and global variables, and making the use of array parameters very expensive. RPC also requires programmers to deal with complex failure semantics.

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 this paper.

Distributed Shared Memory

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 said to have a *distributed shared memory*.

In its simplest form, the shared 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.

A significant improvement to the basic algorithm has been proposed, implemented, and analyzed by Li and Hudak [2]. In their design, thrashing (page traffic caused by concurrent readers) 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. Li and Hudak also present several algorithms, both centralized and distributed, for locating the pages.

Another approach is to weaken the semantics of the shared memory. This has been proposed for multiprocessors [3], but can also be applied to distributed shared memory [4]. The tradeoff here is greater efficiency but the price is more complexity for the programmer. It is simplest for the programmer to think in terms of a model in which a read always returns the most recent value written. Making this true only under certain conditions introduces the possibility of subtle errors creeping in.

A completely different approach is to base the sharing not on pages, but on more software-oriented concepts. In Linda [5], for example, an abstract tuple space is shared. Operations are available to insert and delete tuples from this space. Another approach is to share objects, that is, abstract data types on which a set of well-defined operations are possible. Emerald, for example, illustrates this method [6]. Emerald programs can perform operations on objects without regard to where the programs and objects are located. However, although both the tuple-based and object-based schemes are conceptually simple, both suffer from performance problems due to the considerable amount of network traffic. We will come back to this point after we have described our approach, and we will explain why we believe it is an advance over previous methods.

Our Design

In our research, we have devised and implemented an alternative model that preserves the coherency of object-based shared memory (i.e., based on abstract data types) has simple semantics, and can be implemented efficiently. The model uses our new parallel programming language, Orca, and is implemented through *reliable broadcasting*. It consists of four layers, as shown in Fig. 3.

Layer 1 is the bare CPU and networking hardware. Our scheme is primarily intended for networks that support broadcasting (sending a message to all machines) or multicasting (sending a message to a selected group of machines) in hardware. Ethernet, earth satellites, and cellular radio are examples of networks having broadcasting or multicasting. (For simplicity, we will

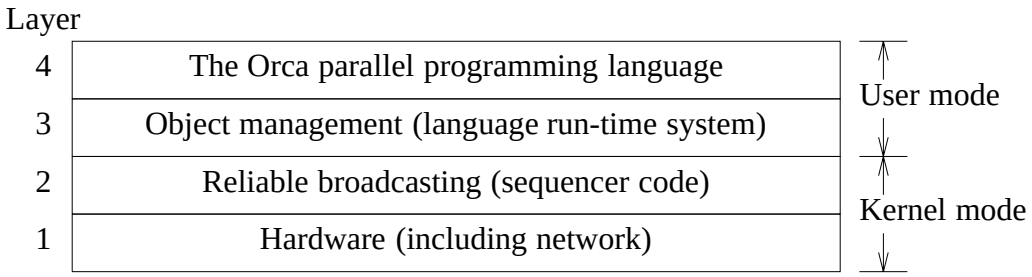


Fig. 3. Structure of the proposed model.

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 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 [7], for parallel programming using distributed shared objects, and have 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 atomic (i.e., indivisible as seen from the outside) and serializable (i.e., happen in some unspecified, but feasible order). 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 can be built on this foundation. In this section we will summarize the mechanism used to achieve reliable broadcasting (in software) over an unreliable network. Space limitations prevent us from describing all the details of the protocol. Readers interested in these should consult the paper by Kaashoek et al. [8].

The hardware/software configuration required for reliable broadcasting is shown in Fig. 4. The ‘‘user programs’’ 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. However, when the application starts up, one of the machines is elected as sequencer (like a committee electing a chairman). If the sequencer machine subsequently crashes, the remaining members elect a new one using one of the many known algorithms (e.g., highest network address wins). The protocol has been designed to withstand sequencer crashes. See [9] for more details on fault tolerance. We will not discuss this point further here.

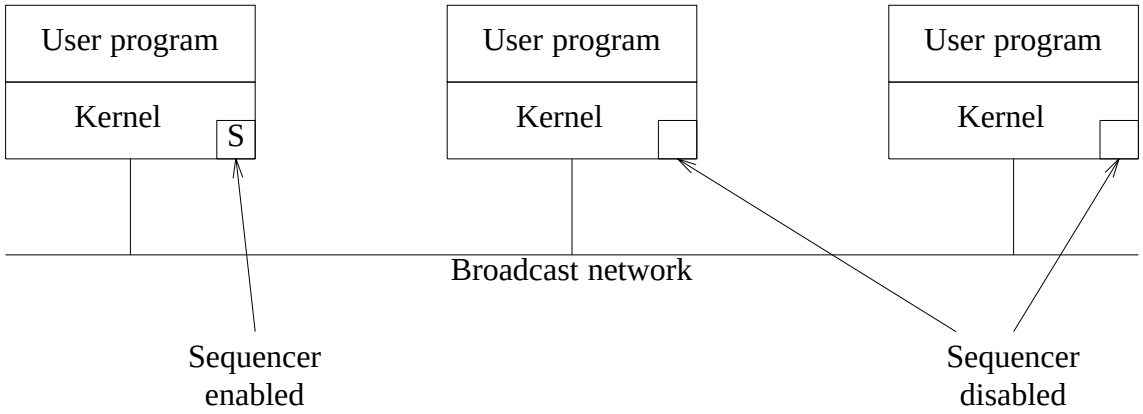


Fig. 4. System structure. Each kernel is capable of becoming the sequencer, but at any instant only one of them functions as sequencer.

The actual sequence of events involved to achieve reliable broadcasting can be summarized as follows.

1. The user traps to the kernel, passing it the message.
2. The kernel accepts the message and blocks the user.
3. The kernel sends it to the sequencer as a point-to-point message.
4. The sequencer adds a sequencer number and broadcasts the message.
5. When the sending kernel sees the broadcast, it unblocks the user.

The sending kernel also starts a retransmission timer in case either the message or the resulting broadcast is lost. A unique id insures that the sequencer will never broadcast the same message twice.

When a *Request for Broadcast* arrives at the sequencer, the unique id is checked to see if the message is a retransmission. If so, the sender is informed that the broadcast has already been done. If not (normal case), the next sequence number is assigned to it. The message is

then broadcast. It is also stored in a *history buffer* in case it must be retransmitted due to a lost message.

Let us now 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.

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. This is the essence of our reliable broadcast protocol.

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. Piggybacked acknowledgements contained in the *Request for Broadcast* messages provide it with this information. In addition, periodically each machine sends the sequencer this information, even if it has nothing to broadcast. The sequencer can also request it.

A variant on the broadcast protocol described above is also worth discussing. 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 (to sequencer + broadcast) wastes bandwidth to reduce interrupts compared to method 2 (user broadcast + *Accept* broadcast).

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 see 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 because modern LANs have very low (but not zero) error rates. 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., [9], give a detailed comparison of our scheme with other published protocols for doing reliable broadcasting. Here we will just compare our scheme with some of the more significant ones. Chang and Maxemchuk [10] describe a family of protocols 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 [11]. 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. ISIS, however, provides a high degree of fault tolerance, which our basic protocol does not. Although not discussed in this paper, our protocol also can provide fault tolerance if desired [9]. 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, as we do. In this category, for example, is the V system of Cheriton and Zwaenepoel [12], 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.

Object Management

On top of the broadcast layer is a layer that manages shared objects, implemented by a package of library procedures (in user space). Our design is based on the explicit assumption that shared objects are read much more often than they are written. Based on our initial measurements of some parallel applications (e.g., the traveling salesman problem), 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 reliablly broadcasting either its new value or an operation code and parameters to let 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 copies 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 atomic update and serializability (managed by the run-time system), 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 (although the window is larger in our model than with a multiprocessor—milliseconds instead of microseconds). Thus although our memory model does not exhibit true coherence, in reality, serializability plus total global message ordering are sufficient properties, and we do have them.

Much research on distributed shared memory is based on the work of Li and Hudak [2]. In their method, fixed-size pages are moved around the network in point-to-point messages. This method is frequently inefficient because many data structures are smaller than a page. The rest of the page is not needed, but must be copied anyway.

In addition, if two unrelated shared data structures happen, by accident, to reside on the same page (false sharing), competition for this page may cause it to thrash back and forth. The larger the page size, the worse the problem. Furthermore, accesses to a shared data structure may require multiple machine instructions. Pages are not automatically locked while an object is being accessed (because the paging system does not know when access to a software object begins and ends). Consequently, a page may have to be accessed several times to complete a single logical operation on a data structure. With an object-based system this cannot happen.

To get around the inefficiency of distributed shared memory using fixed-size pages, some researchers have proposed using objects, as we have. However, lacking our reliable broadcasting, they have needed other methods to gain performance. A common approach has emphasized weakening the semantics of what shared memory means. For example, suppose *A*, *B*, and *C* all update the same word in quick succession, followed by a read by *D*. Inspired by the work of the DASH system [3] on multiprocessor caches, the Munin system [4], allows *D* to get any of the three values written. This optimization allows writes to be postponed until a read is done. Munin also allows programmers to declare different semantics for different shared objects, in order to perform other optimizations. We believe that our model offers a simpler and cleaner semantic model, is easier for programmers to use, and provides good performance (see sidebars).

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. Orca is a procedural language whose sequential constructs are roughly similar to languages like C or Modula 2 but which also supports parallel processes and shared objects.

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 (i.e., atomic), 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 are guaranteed to see exactly the same sequence of serial events. Thus the programmer's model is that the system supports operations. These may be invoked at any moment, but if any invocation would conflict with an operation currently taking place, the second operation will not begin until the first one has completed. In other words, the system has the responsibility for making sure that parallel activities do not interfere with one another.

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. 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 use the standard default case of running it on the current processor. 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. Orca is what is technically called *object based* (in contrast with object oriented) in that it supports encapsulated abstract data types, but without inheritance.

A common way of programming in Orca is the Replicated Worker Paradigm [5]. 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.

It is essential that the synchronization operation be programed as a single Orca operation (although this operation may contain arbitrarily many Orca statements). Having an operation to merely return the value, and then making a decision on it in open code loses the crucial atomicity property.

As a slightly more elaborate example, consider an implementation of the Traveling Salesman Problem (TSP) in Orca. In this problem, the computer is 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. 5.

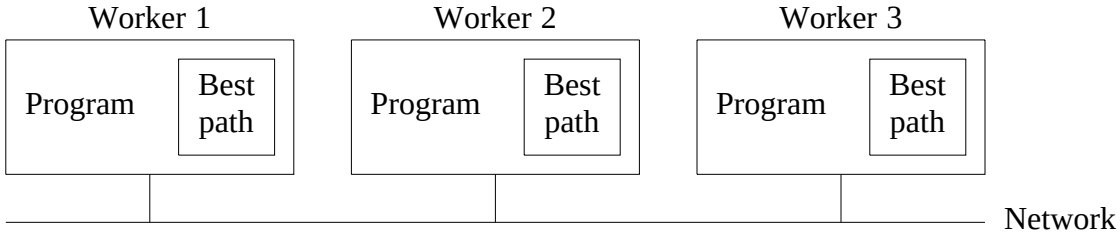


Fig. 5. Model of TSP program using replicated worker paradigm.

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 16 processes.

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 small shared memory machines 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 [6]. 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 [5]. 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 contrast, Orca's operations on shared objects can be simple or complex, as the programmer wishes.

Summary

To summarize, two kinds of multiple processor systems exist: 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 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 800 reliable broadcasts per second using 68030s on an Ethernet.

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). As a consequence, most operations require no network traffic. Orca operations are atomic, providing the programmer with a simple semantic model.

The 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.

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. We believe that this paradigm offers a new and effective way to exploit parallelism in future computing systems.

Acknowledgements

We would like to thank Dick Grune and Erik Baalbergen for carefully reading the paper and making many 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. 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.

3. K. Gharachorloo, D. Lenoski, J. Laudon, and P. Gibbons ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *17th Annual Int’l Symp. on Comp. Arch.*, pp. 15-26, 1990.

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

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

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

8. 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.

9. M.F. Kaashoek and A.S. Tanenbaum, ''Group Communication in the Amoeba Distributed Operating Systems,'' *Proc. 11th Int. Conf. on Distr. Comp. Syst.* pp. 222-230, May 1991.

10. J. Chang and N.F. Maxemchuk, ''Reliable Broadcast Protocols,'' *ACM Trans. Computer Systems.*, Vol. 2, No. 3, Aug. 1984, pp. 251-273.

11. 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.

12. 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.

Sidebar on Performance of Reliable Broadcasting

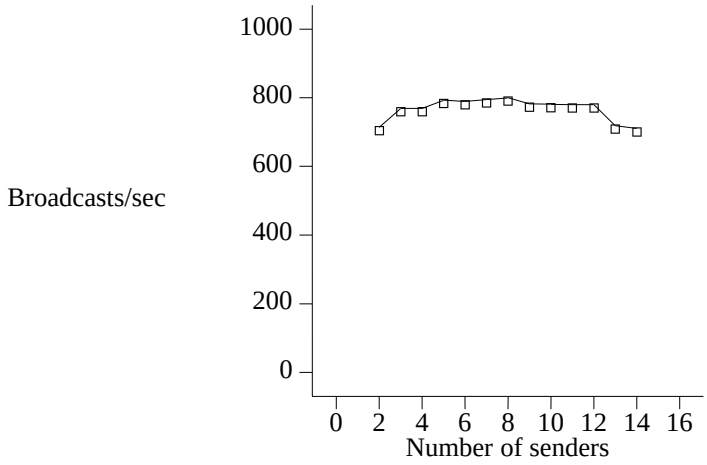
We have modified the Amoeba kernel [1-2] to support our reliable broadcast protocol. It runs on a collection of 16 MHz Motorola 68030 processors connected by a 10 Mbps Ethernet. We have run various experiments on this system to measure its performance. In the first experiment, one process continuously 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:



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 our system (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, we extrapolate this limit to about 80,000 operations per second on shared objects.

References

1. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM* vol. 33, pp. 46-63, Dec. 1990.
2. S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, H. van Staveren, "Amoeba—A Distributed Operating System for the 1990s," *IEEE Computer Magazine*, vol. 23, pp. 44-53, May 1990.

Sidebar on Parallel applications in Orca

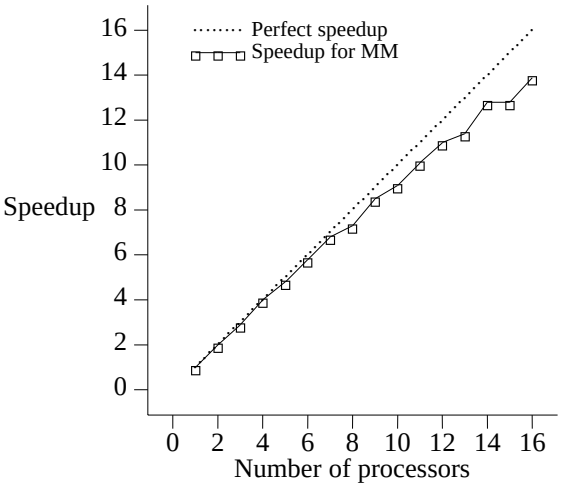
Orca is a procedural, type-secure language intended for implementing parallel applications on distributed systems [1-3]. Below, we will briefly discuss four examples of problems it has been used for and give their measured speedups on the broadcast system.

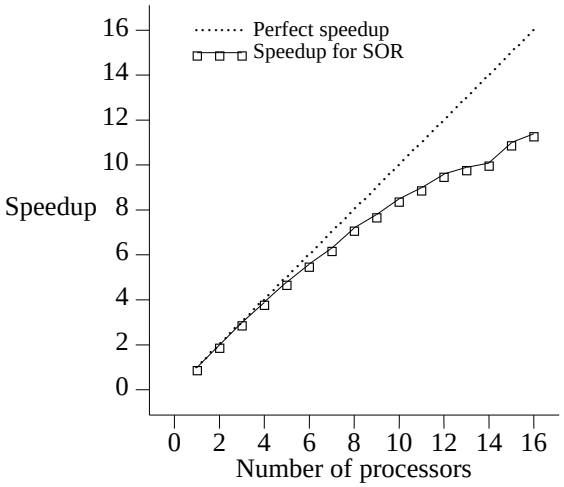
Matrix multiplication (MM) is an example employing ‘‘trivial parallelism.’’ Each processor is assigned a fixed portion of the result matrix. Once the work-to-do has been distributed, all processors can proceed independently from each other. The speedup is not perfect because it takes some time to initialize 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.

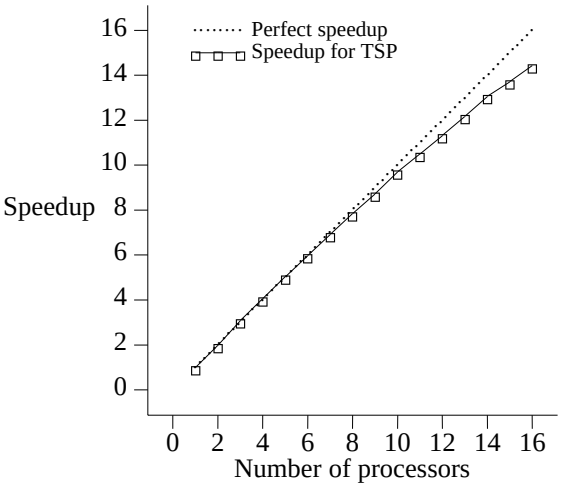
The Traveling Salesman Problem (TSP) is discussed in detail in the paper. Its main characteristic is the shared variable containing the current best solution. This variable is stored in a shared object that is read very frequently. If a new better route is found, all copies of the object are updated 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.

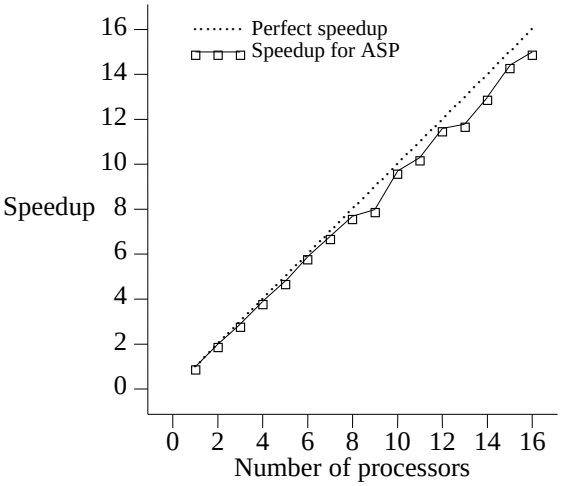
In the all-pairs shortest paths (ASP) problem, communication overhead is much higher. ASP uses an iterative algorithm. Before each iteration, some process selects one row of the distances matrix as pivot row and sends it to all other processes. If implemented with point-to-point messages, the communication 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.

References

1. 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.
2. 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.
3. 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.