A DISTRIBUTED IMPLEMENTATION OF THE SHARED DATA-OBJECT MODEL

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

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
Email: bal@cs.vu.nl

ABSTRACT

The shared data-object model is designed to ease the implementation of parallel applications on loosely coupled distributed systems. Unlike most other models for distributed programming (e.g., RPC), the shared data-object model allows processes on different machines to share data. Such data are encapsulated in data-objects, which are instances of user-defined abstract data types. The shared data-object model forms the basis of a new language for distributed programming, *Orca*, which gives linguistic support for parallelism and data-objects. A distributed implementation of the shared data-object model should take care of the physical distribution of objects among the local memories of the processors. In particular, an implementation may replicate objects in order to decrease access times to objects and increase parallelism.

The intent of this paper is to show that, for several applications, the proposed model is both easy to use and efficient. We first give a brief description of the shared data-object model and Orca. Next, we describe one of several existing implementations of Orca. This implementation replicates all objects on all processors and updates replicas through a reliable broadcast protocol. We describe all three layers of this implementation: the Orca compiler, the Orca run time system, and the reliable broadcast protocol. Finally, we report on our experiences in using this implementation. We describe three parallel applications written in Orca and give performance measurements for them. We also compare these figures with those of a nondistributed (shared-memory) implementation of Orca. The measurements show that significant speedups can be obtained for all three applications.

1. INTRODUCTION

As communication in loosely coupled distributed computing systems is getting faster, such systems become more and more attractive for running parallel applications. In the Amoeba system [Mullender and Tanenbaum 1986], for example, the cost of sending a short message between Sun workstations over an Ethernet is 1.4 milliseconds [Van Renesse et al. 1989]. Although this is still slower than communication in most multicomputers (e.g., Hypercubes and transputer grids), it is fast enough for many coarse-grained parallel applications. In return, distributed systems are easy to build from off-the-shelf components, by interconnecting multiple workstations or microprocessors through a local area network (LAN). In addition, such systems can easily be expanded to far larger numbers of processors than shared-memory multiprocessors.

In our research, we are studying the implementation of parallel applications on distributed systems. We started out by implementing several coarse-grained parallel applications on top of the Amoeba system, using Remote Procedure Calls (RPC) [Birrell and Nelson 1984] for interprocess communication [Bal et al. 1987]. RPC is widely used in the distributed systems community

* This research was supported in part by the Netherlands organization for scientific research (N.W.O.) under grant 125-30-10.

for implementing distributed servers (e.g., file servers) [Tanenbaum and Van Renesse 1985]. For parallel programming, however, RPC has several disadvantages [Tanenbaum and Van Renesse 1988]. RPC is a synchronous (blocking) communication primitive, so a separate mechanism is needed for obtaining parallelism. Of more significance, the programming model of RPC is based on message passing, which is conceptually input/output. This makes efficient sharing of data among processes very hard.

The RPC model does not provide (logically) shared data, since processes on different machines run in separate address spaces. Data that are shared among multiple processes have to be encapsulated by a server process and can only be accessed indirectly through a remote call to this server. Parallel applications, however, often need a finer level of sharing, with a much lower overhead.

As an example of such a parallel application, consider parallel branch-and-bound algorithms. Such algorithms store the current best solution (the bound) in a global variable accessed by all processors. This is not to say the algorithms actually need physical shared memory; as the bound is updated only once in a while, parallel branch-and-bound algorithms can be implemented efficiently on distributed systems. In our experience, however, implementing the algorithms efficiently using RPC is complicated.

In this paper, we will look at an alternative model for distributed programming that supports logically shared data. This model, the *shared data-object model* [Bal and Tanenbaum 1988], allows processes to share data without requiring physical shared memory. Also, we have designed a new programming language, *Orca* [Bal and Tanenbaum 1988; Bal et al. 1989], based on this model. The intent of this paper is to show that, for several applications, the model is both easy to use and efficient. We do so by describing an implementation of Orca on a loosely coupled system and reporting on our experiences in using this implementation for several small-scale but realistic applications.

The issue of providing logically shared data in an environment without shared memory has been addressed by several other languages and operating systems. Linda's Tuple Space [Ahuja et al. 1986], for example, is a global, content-addressable shared memory, which has been implemented on various types of parallel systems. For many applications this model is much easier to use than RPC. The operations defined on Tuple Space provide a low level of abstraction, however, which we feel is a disadvantage for distributed programming [Kaashoek et al.]. Other interesting proposals include parallel object-oriented languages (e.g., Emerald [Jul et al. 1988]), which provide a uniform address space for objects, and Kai Li's shared virtual memory [Li 1988], which simulates physical shared memory. (These and other systems are surveyed in [Bal and Tanenbaum 1988]). Also, several researchers have looked at distributed applications that can be implemented with logically shared data. Example applications are: speech recognition [Bisiani and Forin 1987], linear-equation solving, three-dimensional partial differential equations [Li 1988], and global scheduling and replicated files [Cheriton 1985].

The rest of the paper is structured as follows. In Section 2, we will give a brief description of the shared data-object model and Orca. In Section 3, we will discuss one implementation of the model, based on reliable broadcast. We will also describe how to implement this broadcast primitive on top of LANs that only support unreliable broadcast. In Section 4, we will report on our experiences in using this implementation of Orca. We will give performance measurements for several applications. Also, we will compare these performance figures with those of a nondistributed (shared-memory) implementation of Orca. Finally, in Section 5 we present our conclusions.

2. THE SHARED DATA-OBJECT MODEL

The most important issue addressed by our model is how data structures can be shared among distributed processes in an efficient way. In languages for multiprocessors, shared data structures are stored in the shared memory and accessed in basically the same way as local variables, namely through simple load and store instructions. If a process is going to change part of a shared data structure and it does not want other processes to interfere, it locks that part. All these operations (loads, stores, locks) on shared data structures involve very little overhead, because access to shared memory is hardly more expensive than access to local memory.

In a distributed system, on the other hand, the time needed to access data very much depends on the location of the data. Accessing data on remote processors may be orders of magnitude more expensive than accessing local data. It is therefore infeasible to apply the multiprocessor model of programming to distributed systems. The operations used in this model are far too low-level and will have tremendous overhead on distributed systems.

The starting-point in our model is to access shared data structures through higher level operations. Instead of using low-level instructions for reading, writing, and locking shared data, we propose to let programmers define composite operations for manipulating shared data structures. Shared data structures in our model are encapsulated in so-called *data-objects*[1] that are manipulated through a set of user-defined operations. Data-objects are best thought of as instances (variables) of *abstract data types*. The programmer specifies an abstract data type by defining operations that can be applied to instances (data-objects) of that type. The actual data contained in the object and the executable code for the operations are hidden in the implementation of the abstract data type.

Although data-objects logically are shared among processes, their implementation does not need physical shared memory. In worst case, an operation on a remote object can be implemented with a remote procedure call. The general idea, however, is for the implementation to take care of the physical distribution of data-objects among processors. As we will see in Section 3, one way to achieve this goal is to replicate shared data-objects. By replicating objects, access control to shared objects is decentralized, which decreases access costs and increases parallelism. This is a major difference with, say, monitors [Hoare 1974], which centralize control to shared data.

In the following sections, we will elaborate the basic idea by looking at the issue of synchronization. Two types of synchronization can be distinguished: [Andrews and Schneider 1983] mutual exclusion synchronization prevents multiple simultaneous writes (or reads and writes) to the same data from interfering with each other; condition synchronization allows processes to wait for a certain condition to become true. We discuss both types of synchronization in turn, in Sections 2.1 and 2.2. Finally, in Section 2.3 we describe a language based on this model.

2.1. Mutual exclusion synchronization

Shared-variable languages usually provide some kind of *locking* construct for mutual exclusion synchronization. In a distributed environment, however, such locking primitives are too low-level and have a high overhead. In our model, mutual exclusion is done implicitly, by executing all operations on objects *indivisibly*. Conceptually, each operation locks the entire object it is applied to and releases the lock only when it is finished. To be more precise, the model guarantees *serializability* [Eswaran et al. 1976] of operation invocations: if two operations are applied simultaneously to the same data-object, then the result is as if one of them is executed before the other; the order of invocation, however, is nondeterministic.

An implementation of the model need not actually execute all operations one by one. To in-

[1] We will sometimes use the term object as a shorthand notation. Note, however, that this term is used in many other languages and systems, with various different meanings.

crease the degree of parallelism, it may execute multiple operations on the same object simultaneously, as long as the effect is the same as for serialized execution. For example, operations that only read (but do not change) the data stored in an object can easily be executed in parallel.

As operations are indivisible, mutual exclusion synchronization to shared data-objects is taken care of automatically. As a simple example, consider an object encapsulating an integer variable, as specified in Figure 1.

```
object specification IntObject;
  operation Value(): integer;          # return current value
  operation Assign(val: integer);      # assign new value
  operation Add(val: integer);         # add val to current value
  operation Min(val: integer);         # set value to minimum of current value and val
end;
```

Fig. 1. Specification part of an object type IntObject.

Suppose two processes P_1 and P_2 share an object *X* of this type. If they simultaneously try to apply the *Assign* operation to *X*, the resulting value will either be that of P_1's or P_2's invocation, but the value will never be some strange mixture of the bits. Similarly, if P_1 and P_2 simultaneously increment the value of *X* by invoking the operation

```
X$Add(1);
```

the value will always be incremented twice, because the operations are serialized.

On the other hand, *sequences* of operations are not executed indivisibly. For example, the sequence

```
tmp := X$Value();         # get value of object X
X$Assign(tmp+1);          # increment value and store result back in X
```

is not an indivisible action. If two processes execute this sequence simultaneously, the value of *X* may be incremented once or twice. This rule for defining which actions are indivisible and which are not is both easy to understand and flexible: single operations are indivisible; sequences of operations are not. Orca does not provide mutual exclusion at a granularity lower than the object level.

Our model does not support indivisible operations on multiple objects. Operations on multiple objects would require a distributed locking protocol, which is complicated to implement efficiently. Instead, we prefer to keep our basic model as simple as possible and implement more complicated actions on top of it. Operations in our model therefore apply to single objects and are always executed indivisibly. However, the model is sufficiently powerful to allow users to construct locks for multi-operation sequences on different objects, so arbitrary actions can be performed indivisibly.

2.2. Condition synchronization

Condition synchronization allows processes to wait (*block*) until a certain condition becomes true. The simplest form of condition synchronization is repeated testing (*busy waiting*) of a shared variable, until it has a certain value. Since busy waiting wastes computing cycles, most parallel languages use a separate condition synchronization mechanism, such as a semaphore, eventcount, or condition variable [Andrews and Schneider 1983].

In the shared data-object model, condition synchronization is integrated with operation invocations by allowing operations to block. Processes synchronize implicitly through operations on shared objects. A blocking operation consists of one or more guarded commands:

```
operation op(formal-parameters): ResultType;
  local declarations
begin
   guard condition_1 do statements_1 od;
   guard condition_2 do statements_2 od;
   ...
   guard condition_n do statements_n od;
end;
```

The conditions must be side-effect free boolean expressions. The operation initially blocks (suspends) until at least one of the conditions (guards) evaluates to true. Next, one true guard is selected nondeterministically, and its sequence of statements is executed.

2.3. Orca

We have used the shared data-object model for designing a new language called *Orca* for distributed application programming. Unlike the majority of other languages for parallel or distributed programming, Orca is not an extension to an existing sequential language. Instead, its sequential and distributed constructs have been designed together, in such a way that they integrate well.

Orca is a procedural, strongly typed language. Its statements and expressions are fairly conventional and comparable to those of Modula-2. The data structuring facilities of Orca, however, are substantially different from those used in Modula-2. Orca supports records, unions, dynamic arrays, sets, bags, general graphs, and generic types. Pointers have intentionally been omitted to provide type-security.

Parallelism in Orca is based on explicit creation of sequential processes. Processes are conceptually similar to procedures, except that procedure invocations are serial and process invocations are parallel.

Processes communicate through shared data-objects, which are instances of *abstract data types*. An abstract data type definition consists of two parts: a *specification* part and an *implementation* part. The specification part defines the operations applicable to objects of the given type. (An example of a specification part was given in Figure 1.) The implementation part contains the data of objects of this type, the code to initialize the data of new instances of the type, and the code implementing the operations.

Objects are created by declaring variables of an abstract data type. The declaration does not specify whether the object will be shared. When an object is created, the run time system allocates memory for the local variables of the object and executes the initialization code.

Objects declared local to a process may be shared with other (child) processes by passing them as shared parameters when the children are created. For example, if a process *child* is declared as

```
process child(Id: integer; X: shared IntObject);
```

a new child process can be created as follows

```
fork child(12, X);
  # create a new child process, passing the constant 12 as
  # value parameter and the object X as shared parameter.
```

The children can pass shared objects to *their* children, and so on. In this way, the objects get distributed among some of the descendants of the process that created them. If any of these processes performs an operation on the object, they all observe the same effect, as if the object were in shared memory, protected by a lock variable.

In summary, Orca allows processes to share data encapsulated in objects, which are instances of abstract data types. Sharing of objects is only possible between a parent and its descendants, which is sufficient for the applications Orca intends to support. Each process sharing an object may apply operations to the object, as defined by the object's abstract data type. The effects of operation invocations are observed by all processes sharing the object. Simultaneous invocations of operations on the same object are conceptually serialized. Condition synchronization is expressed through operations that block.

3. A DISTRIBUTED IMPLEMENTATION OF ORCA

Although Orca is a language for programming distributed systems, its communication model is based on shared data. The implementation of the language therefore should hide the physical distribution of the hardware and simulate shared data in an efficient way. We have designed several different models for implementing the language [Bal and Tanenbaum 1988]. The implementation described in this paper is based on *replication* and *reliable broadcasting*.

Replication of data is used in several fault-tolerant systems (e.g., ISIS [Joseph and Birman 1987]). to increase the availability of data in the presence of processor failures. Orca, in contrast, is not intended for fault-tolerant applications. In our implementation, replication is used to decrease the access costs to shared data.

Very briefly stated, each processor keeps a local copy of each shared data-object. This copy can be accessed by all processes running on that processor (see Figure 2). Operations that do not change the object (called *read* operations) use this copy directly, without any messages being sent. Operations that do change the object (called *write* operations) broadcast the new values (or the operations) to all the other processors, so they are updated simultaneously.

M: box wid 4*boxwid ht 4*boxht B1: box wid 1.25*boxwid "process-1" with .nw at M.nw+(0.3*boxwid, -0.3*boxwid) B2: box wid 1.25*boxwid "process-n" with .sw at M.sw+(0.3*boxwid, 0.3*boxwid) X: box ht 1.5*boxht "copy" "of" "X" with .e at M.e - (0.3*boxwid, 0) arrow from B1.e to X.w+(0, 0.2*boxwid) arrow from B2.e to X.w-(0, 0.2*boxwid) "CPU 1" at M.n + (0, 0.3*boxwid) N: box wid 4*boxwid ht 4*boxht with .w at M.e+(boxwid, 0) C1: box wid 1.25*boxwid "process-1" with .nw at N.nw+(0.3*boxwid, -0.3*boxwid) C2: box wid 1.25*boxwid "process-n" with .sw at N.sw+(0.3*boxwid, 0.3*boxwid) Y: box ht 1.5*boxht "copy" "of" "X" with .e at N.e - (0.3*boxwid, 0) arrow from C1.e to Y.w+(0, 0.2*boxwid) arrow from C2.e to Y.w-(0, 0.2*boxwid) "CPU 2" at N.n + (0, 0.3*boxwid) line "n e t w o r k" below from M.sw-(0,boxht) to N.se-(0,boxht) line from M.s to M.s-(0,boxht) line from N.s to N.s-(0,boxht)

Fig. 2. Replication of data-objects in a distributed system

The implementation is best thought of as a three layer software system, as shown below:

compiled application programs
run time system
reliable broadcasting

The top layer is concerned with applications, which are written in Orca and compiled to machine code by the Orca compiler. The executable code contains calls to the Orca run time system, for example for creating and manipulating processes and objects.

The middle layer is the run time system (RTS). It implements the primitives called by the upper layer. For example, if an application performs an operation on a shared data-object, it is up to the RTS to ensure that the system behaves as if the object was placed in shared memory. To achieve this, the RTS of each processor maintains copies of shared objects, which are updated using reliable broadcasting.

The bottom layer is concerned with implementing the reliable broadcast, so that the RTS does not have to worry about what happens if a broadcast message is lost. As far as the RTS is concerned, broadcast is error free. It is the job of the bottom layer to make it work.

Below, we will describe the protocols and algorithms in each layer. This section is structured top down: we first discuss the applications layer, then the RTS layer, and finally the reliable broadcast layer.

3.1. Top layer: Orca application programs

Application programs are translated by the Orca compiler into executable code for the target system.[2] Most of the compiler is based on conventional compiler technology. In fact, our compiler has been built using the *Amsterdam Compiler Kit*, which is a toolkit for implementing portable compilers [Tanenbaum et al. 1983]. Up until now ACK has mainly been used for sequential languages like C and Pascal and for uniprocessor implementations of parallel (or pseudo-parallel) languages like Modula-2, occam, and Ada®. As it turns out, ACK is useful for distributed languages like Orca as well.

The code produced by the compiler contains calls to RTS routines that manage processes, shared data-objects, and complex data structures (e.g., dynamic arrays, sets, and graphs). In this paper, we will only discuss how operation invocations are compiled.

As described above, it is very important to distinguish between *read* and *write* operations on objects. The compiler therefore analyses the implementation code of each operation and checks whether the operation modifies the object to which it is applied.[3] It stores this information in an *operation descriptor*. This descriptor also specifies the sizes and modes (input or output) of the parameters of the operation.

If an Orca program applies an operation on a given object, the compiler generates a call to the RTS primitive *INVOKE*. This routine is called as follows:

INVOKE(object, operation-descriptor, parameters ...);

The first argument identifies the object to which the operation is applied. The second argument is the operation descriptor. The remaining arguments of *INVOKE* are the parameters of the operation. The implementation of this primitive is discussed below.

3.2. Middle layer: The Orca run time system

The middle layer implements the Orca run time system. As mentioned above, its primary job is to manage shared data-objects. In particular, it implements the *INVOKE* primitive described above. For efficiency, the RTS replicates objects so it can apply operations to local copies of objects whenever possible.

There are many different design choices to be made related to replication. The most important ones are:

---

[2] We assume the target system does not contain multiple types of CPUs. Although a heterogeneous implementation of Orca is conceivable, we do not address this issue here.
[3] The actual implementation is somewhat more complicated, since an operation may have multiple guards (alternatives), some of which may be read-only.

Replication strategy:

The RTS may either replicate all objects on all processors (*full replication*) or it may try to replicate objects only on those processors that frequently read the object (*partial replication*). In the latter case, the RTS may use compile-time information as well as run-time statistics for deciding where to store replicas of objects.

Updating of replicas:

After a write operation, the replicas of an object should either be invalidated or updated. Updating can either be implemented by sending the new value of the object to the other processors or by applying the operation itself to each copy.

Mutual exclusion synchronization:

Write operations on a given object can be synchronized in at least two different ways. One way is to appoint some copy of the object as *primary copy* and direct all write operations to this primary copy. An alternative way is to treat all copies as equals and use a *distributed update protocol* that takes care of mutual exclusion.

Each of these alternatives has its own advantages and disadvantages, as discussed in [Bal and Tanenbaum 1988]. The RTS described in this paper uses full replication of objects, updates replicas by applying write operations to all replicas, and implements mutual exclusion through a distributed update protocol. (We have also implemented a second RTS, which uses partial replication based on run-time statistics and which updates copies through a primary-copy update protocol. In addition, we have implemented a third RTS on a true shared-memory multiprocessor, for comparison purposes.)

We have chosen to use an update scheme rather than an invalidation scheme for two reasons. First, in many applications objects contain large amounts of data (e.g., a 100K bitvector). Invalidating a copy of such an object is wasteful, since the next time the object is replicated its entire value must be transmitted. Second, in many cases updating a copy will take just as much CPU time and network bandwidth as sending invalidation messages.

The presence of multiple copies of the same logical data introduces the so-called *inconsistency problem*. If the data are modified, all copies are modified too. If this updating is not done as one indivisible action, different processors temporarily have different values for the same logical data. (The inconsistency problem appears in many other areas where data are replicated, for example replicated file servers and CPU caches.)

The semantics of the shared data-object model define that simultaneous operations on the same object must conceptually be serialized. The exact order in which they are to be executed is not defined, however. If, for example, a read operation and a write operation are applied to the same object simultaneously, the read operation may either observe the value before or after the write, but not an intermediate value. However, all processes having access to the object must see the events happen in the same order.

The RTS described here solves the inconsistency problem by using a distributed update protocol that guarantees that all processes observe changes to shared objects *in the same order*. One way to achieve this would be to lock all copies of an object prior to changing the object. Unfortunately, distributed locking is quite expensive and complicated.

Our update protocol does not use locking. The key to avoid locking is the use of an *indivisible, reliable broadcast* primitive, which has the following properties:

- Each message is sent reliably from one source to all destinations.
- If two processors simultaneously broadcast two messages (say m_1 and m_2), then either all

destinations first receive m_1, or they all receive m_2 first. Mixed forms (some get m_1 first, some get m_2 first) are excluded by the software protocols.

This primitive is implemented by the bottom layer of our system, as will be described in Section 3.3, Here, we simply assume the indivisible, reliable broadcast exists.

The RTS uses an *object-manager* for each processor. The object-manager is a light-weight process (thread) that takes care of updating the local copies of all objects stored on its processor. We assume the object-manager and user processes on the same processor can share part of their address space. Objects (and replicas) are stored in this shared address space. User processes can *read* local copies directly, without intervention by object-managers. Write operations on shared objects, on the other hand, are marshalled and then broadcast to all object-managers in the system. A user process that broadcasts a write operation suspends until the message has been handled by its local object-manager. This is illustrated in Figure 3.

```
INVOKE(obj, op, parameters)
    if op.ReadOnly then                    # check if it's a read operation
        set read-lock on local copy of obj;
        call op.code(obj, parameters);     # do operation locally
        unlock local copy of obj
    else
        broadcast GlobalOperation(obj, op, parameters) to all managers;
        block current process;
    fi;
```

Fig. 3. Implementation of the *INVOKE* run time system primitive. This routine is called by user processes.

Each object-manager maintains a queue of messages that have arrived but that have not yet been handled. As all processors receive all messages in the same order, the queues of all managers are basically the same, except that some managers may be ahead of others in handling the messages at the head of the queue.

The object-manager of each processor handles the messages of its queue in strict FIFO order. A message may be handled as soon as it appears at the head of the queue. To handle a message *GlobalOperation(obj, op, parameters)* the message is removed from the queue, unmarshalled, the local copy of the object is locked, the operation is applied to the local copy, and finally the copy is unlocked. If the message was sent by a process on the same processor, the manager unblocks that process (see Figure 4).

```
receive GlobalOperation(obj, op, parameters) from W →
    set write-lock on local copy of obj;
    call op.code(obj, parameters);   # apply operation to local copy
    unlock local copy of obj
    if W is a local process then
        unblock(W);
    fi;
```

Fig. 4. The code to be executed by the object-managers for handling *GlobalOperation* messages.

Write operations are executed by all object-managers in the same order. If a read operation is executed concurrently with a write operation, the read may either be executed before or after the write, but not during it. Note that this is in agreement with the serialization principle described above.

3.3. Bottom layer: Reliable broadcast

In this section we describe the protocol that allows a group of nodes on an unreliable broadcast network to broadcast messages reliably. The protocol guarantees that all of the receivers in the group receive all broadcast messages and that all receivers accept the messages in the same order. The main purpose of this section is to show that a protocol with the required semantics is feasible; for a detailed description we refer the reader to [Kaashoek et al.].

With current microprocessors and LANs, lost or damaged packets and processor crashes occur very infrequently. Nevertheless, the probability of an error is not zero, so they must be dealt with. For this reason our approach to achieving reliable broadcast is to make the normal case highly efficient, even at the expense of making error-recovery more complex, since error recovery will not be done very often.

The basic reliable broadcast protocol works as follows. When the RTS wants to broadcast a message, M, it hands the message to its kernel. The kernel then encapsulates M in an ordinary point-to-point message and sends it to a special kernel called the *sequencer*. The sequencer's node contains the same hardware and kernel as all the others. The only difference is that a flag in the kernel tells it to process messages differently. If the sequencer should crash, the protocol provides for the election of a new sequencer on a different node.

The sequencer determines the ordering of all broadcast messages by assigning a *sequence number* to each message. When the sequencer receives the point-to-point message containing M, it allocates the next sequence number, s and broadcasts a packet containing M and s. Thus all broadcasts are issued from the same node, by the sequencer. Assuming that no packets are lost, it is easy to see that if two RTSs simultaneously want to broadcast, one of them will reach the sequencer first and its message will be broadcast to all the other nodes first. Only when that broadcast has been completed will the other broadcast be started. The sequencer provides a global ordering in time. In this way, we can easily guarantee the atomicity of broadcasting.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast packet eventually arrives, the kernel will immediately notice a gap in the sequence numbers. It was expecting s next, and it got $s + 1$, so it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for copies of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores old broadcast messages in its *history buffer*. The missing messages are sent point-to-point to the process requesting them.

As a practical matter, the sequencer has a finite amount of space in its history buffer, so it cannot store broadcast messages forever. However, if it could somehow discover that all machines have received broadcasts up to and including k, it could then purge the first k broadcast messages from the history buffer.

The protocol has several ways of letting the sequencer discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message. In this way, the sequencer can maintain a table, indexed by node number, showing that node i has received all broadcast messages 0 up to T_i, and perhaps more. At any moment, the sequencer can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, the sequencer knows that everyone has received broadcasts 0 through 6, so they can be deleted from the history buffer.

If a node does not need to do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval, Δt, can just send the sequencer a special packet acknowledging all received broadcasts.

If, despite all precautions, the sequencer gets out of history space, it enters a synchronization phase to empty its history buffer. The synchronization phase consists of a two-phase commit protocol, during which all nodes are brought up-to-date. In practice, the synchronization phase is hardly ever entered.

In short, to do a broadcast, an application process sends the data to the sequencer, which gives it a sequence number and broadcasts it. There are no separate acknowledgement packets, but all messages to the sequencer carry piggybacked acknowledgements. When a node receives an out of sequence broadcast, it buffers the broadcast temporarily, and asks the sequencer for the missing broadcasts. Since broadcasts are expected to be common—many per second—the only effect that a missed broadcast has is causing some application process to get behind by a few tens of milliseconds once in a while, hardly a serious problem.

In philosophy, the protocol resembles the one described by Chang and Maxemchuk21 , but differs in some major aspects. Messages can be delivered to the user as soon as one (special) node has acknowledged the message. In addition, fewer control messages are needed in the normal case (no lost messages). Our protocol therefore is highly efficient, since, during normal operation, only two packets are needed (assuming that a message fits in a single packet), one point-to-point packet from the sender to the sequencer and one broadcast packet from the sequencer to everyone. A comparison between our protocol and other well known protocols (e.g., those of Birman and Joseph [Birman and Joseph 1987], Garcia-Molina and Spauster [Garcia-Molina and Spauster 1989], and several others). is given in [Kaashoek et al.]

4. EXPERIENCE WITH THE ORCA IMPLEMENTATION

We have built a prototype implementation of the shared data-object model, using the layered approach described in the previous section. The prototype runs on the bare hardware, rather than on top of an operating system. In effect, it is a new kind of operating system designed specifically for parallel applications. It uses the Amoeba protocols [Mullender and Tanenbaum 1986] to communicate with our local UNIX® and Amoeba systems.

The prototype runs on two different systems. One implementation runs on a multiprocessor with 10 16 Mhz MC68020 CPUs. The system contains 8Mb of shared memory, which is accessible through a VME bus. This implementation uses the shared memory to simulate unreliable broadcast messages. The reliability of the network (i.e., the percentage of broadcast messages delivered at a destination) is an adjustable parameter of the system. In this way, we are able to test our protocol with different degrees of reliability. The second implementation runs on a distributed system, containing 10 16 Mhz MC68020 CPUs connected to each other through an 10 Mbit/s Ethernet [Metcalfe and Boggs 1976]. This implementation uses Ethernet multicast communication to broadcast a message to a group of processors. All processors are on one Ethernet and are connected to the network by Lance chip interfaces.

The performance of the broadcast protocol on the Ethernet system is described in [Kaashoek et al.]. The time needed for multicasting a short message reliably to two processors is 1.3 msec. With 10 receivers, a multicast takes 1.5 msec. The time also depends on the number of senders that are active simultaneously. If, for example, 7 processors are simultaneously sending a message to 10 processors, the average time per multicast is 4.6 msec. This high performance is due to the fact that our protocol is optimized for the common case (i.e., no lost messages). During the experiments described below, the number of lost messages was found to be zero.

We have used the Ethernet implementation for developing several parallel applications written in Orca. Some of these are small, but others are larger. The largest application we currently have is a parallel chess program, consisting of about 2500 lines of code. Smaller applications include matrix multiplication, prime number generation, sorting, and successive overrelaxation. In this section we give preliminary performance measurements of three sample programs running on the Ethernet implementation.

An implementation of Orca designed for a shared-memory multiprocessor would be simpler and, in general, faster than a distributed implementation, since it could put shared objects in the shared memory. Systems with physical shared memory, however, are much harder to build than memory-disjoint systems, especially if a large number of processors (e.g., thousands) is required. To build highly parallel shared-memory systems, a switching network is required, which may be very costly [Almasi and Gottlieb 1989]. It is interesting to compare the performance of our model on distributed and shared-memory systems, and see how much performance is lost by using simpler and less expensive hardware. For this purpose, we also wrote a shared-memory implementation of Orca. This implementation runs on the VME-based multiprocessor described above.

Below, we will compare the performances of the distributed and nondistributed implementations. Both implementations use exactly the same processor boards. The distributed implementation uses the Ethernet for point-to-point and broadcast communication. The nondistributed implementation uses the shared memory for storing shared objects.

4.1. Parallel branch-and-bound

The first application we will discuss is parallel branch-and-bound. As a representative example, consider the traveling salesman problem (TSP). A salesman is given an initial city in which to start, and a list of cities to visit. Each city must be visited once and only once. The objective is to find the shortest path that visits all the cities.

The algorithm we have implemented in Orca uses one *manager* process to generate initial paths for the salesman, starting at the initial city but visiting only part of the other cities. A number of *worker* processes further expand these initial paths, using the nearest-city-first heuristic. A worker systematically generates all paths starting with a given initial path and checks if they are better than the current shortest full path. The length of the current best path is stored in a data-object of type *IntObject* (see Figure 1). This object is shared among all worker processes. The manager and worker processes communicate through a shared queue data structure, as shown in Figure 5.

Every time a worker finds a shorter full path, it updates this variable, using the (indivisible) operation *Min.* On the other hand, if a worker ever finds a partial path that is longer than the current best path, it is pointless to continue, so the path being investigated is abandoned.

It should be clear that reading of the current best path length will be done very often, but since this is a local operation, there is no communication overhead. Updating the best path happens much less often, but still only requires two broadcast messages (one update message and one acknowledgement).

Although updates of the best path happen infrequently, it is very important to broadcast any improvements immediately. If a worker uses an old (i.e., inferior) value of the best path, it will investigate paths that could have been pruned if the new value had been known. In other words, the worker will search more nodes than necessary. This *search overhead* may easily become a dominating factor and cause a severe performance degradation.

In the RPC model, it is very difficult to let processes share data that are always kept up-to-date. A halfway solution is to let each worker maintain its own local minimum and update this local variable whenever the worker gets a new job. This approach still suffers from a significant search overhead, however [Bal et al. 1987]. With the shared data-object model, on the other hand, right X: box "Manager" arrow J: box wid 1.5*boxwid arrow Wm: box "Worker" arrow <-> M: box wid 0.75*boxwid ht 0.75*boxht Wh:box "Worker" with .s at Wm.n+(0, 0.5*boxht) Wl:box "Worker" with .n at Wm.s-(0, 0.5*boxht) arrow from J.e to Wh.w arrow from J.e to Wl.w arrow <-> from Wh.e to 1/2 <M.w, M.nw> arrow <-> from Wl.e to 1/2 <M.w, M.sw> box "job" wid 0.35*boxwid ht 0.75*boxht with .w at J.w+(0.1*boxwid,0) box "job" same with .e at J.e-(0.1*boxwid,0) box "..." invis same at J box invis "JobQueue" ht 0.5*boxht with .n at J.s box invis "Minimum" wid 0.75*boxwid ht 0.5*boxht with .n at M.s box wid 0.9*boxwid ht 0.9*boxht at X box same at Wm box same at Wh box same at Wl

Fig. 5. Structure of the Orca implementation of TSP. The Manager and Workers are processes. The JobQueue is a data-object shared among all these processes. Minimum is a data-object of type IntObject; it is read and written by all workers.

sharing data is easy.

The performance of the traveling salesman program (for a randomly generated graph with 12 cities) on the shared-memory and distributed implementations of Orca are given in Figure 6.

With fewer than 5 processors, the shared-memory implementation is slightly faster. This performance difference is caused by the relatively high computational overhead of operations in our prototype distributed implementation. With 6 or more processors, however, the distributed system is faster. (Note that Figure 6 shows the performance for one specific TSP graph; for other randomly generated graphs we have observed similar behavior.)

Although surprising at first sight, this behavior is easy to explain. In the distributed RTS, each processor will have its own local copy of the shared object *Minimum*. Thus, all processors can simultaneously read their copies. In the shared-memory RTS, on the other hand, the object is put in the shared memory and protected by locks, so it becomes a sequential bottleneck.

In our prototype implementation of the RTS, the situation is particularly bad, because:

1. Operations are implemented inefficiently and thus are expensive. The *Value* operation, which is used to read the current value of *Minimum*, takes about 40 µsec.
2. Exclusive locks—rather than readers/writer locking—are used.
3. The hardware we use allows only one processor at a time to access the shared memory.

As the *Value* operation is executed very frequently, it will often have to wait for the lock to be free. Undoubtedly, the contention problem would be less severe in a well-tuned shared-memory implementation on more advanced hardware. Still, it is not clear whether the problem can be eliminated entirely in this way, without using local copies of objects.

The distributed implementation achieves almost perfect speedup. With 10 CPUs it is 9.52 times faster than with 1 CPU. The shared-memory implementation achieves a speedup of only 6.71. For comparison, the RPC-based implementation of TSP described in [Bal et al. 1987] achieves a speedup of only 6.29 for the same input graph, using the same hardware. The lower speedup of the RPC implementation is caused by its high search overhead.

```
Graph: [     # gg 1 .. 10, 35.1 .. 350 define xy_gg   (($1)−(.37))*.292398,(($2)−(13.057))*0.00835687 define
x_gg   (($1)−(.37))*.292398 define y_gg   (($1)−(13.057))*0.00835687    frameht = 3    framewid = 3
Frame:   box ht frameht wid framewid with .sw at 0,0 invis    line from Frame.nw to Frame.ne invis    line
from Frame.sw to Frame.se    line from Frame.sw to Frame.nw    line from Frame.se to Frame.ne invis
   textht = .166667    textwid = .8 Label:   box invis wid 0 ht 2*textht "Time" "(in seconds)" wid textwid
with .e at Frame.w - (0.2,0) + (-.2,0)    textht = .166667 Label:   box invis wid 0 ht 1*textht "number of proces-
sors" with .n at Frame.s - (0,2 * textht)    ticklen = .1 Ticks_gg: line  left ticklen from (0,y_gg(50))    "50 "
rjust at last line.end    line  left ticklen from (0,y_gg(100))    "100 " rjust at last line.end    line  left ticklen
from (0,y_gg(150))    "150 " rjust at last line.end    line  left ticklen from (0,y_gg(200))    "200 " rjust at
last line.end    line  left ticklen from (0,y_gg(250))    "250 " rjust at last line.end    line  left ticklen from
(0,y_gg(300))    "300 " rjust at last line.end    line  left ticklen from (0,y_gg(350))    "350 " rjust at last
line.end    ticklen = .1 Ticks_gg: line  down ticklen from (x_gg(1),0)    box invis "1" ht .25 wid 0 with .n at
last line.end    line  down ticklen from (x_gg(2),0)    box invis "2" ht .25 wid 0 with .n at last line.end
   line  down ticklen from (x_gg(3),0)    box invis "3" ht .25 wid 0 with .n at last line.end    line  down tick-
len from (x_gg(4),0)    box invis "4" ht .25 wid 0 with .n at last line.end    line  down ticklen from (x_gg(5),0)
   box invis "5" ht .25 wid 0 with .n at last line.end    line  down ticklen from (x_gg(6),0)    box invis "6" ht
last line.end    line  down ticklen from (x_gg(8),0)    box invis "8" ht .25 wid 0 with .n at last line.end
   line  down ticklen from (x_gg(9),0)    box invis "9" ht .25 wid 0 with .n at last line.end    line  down tick-
len from (x_gg(10),0)    box invis "10" ht .25 wid 0 with .n at last line.end Lgg: xy_gg(1,319.6) "□" at
xy_gg(1,319.6) line  from Lgg to xy_gg(2,161); Lgg: Here "□" at xy_gg(2,161) line  from Lgg to xy_gg(3,109.3);
Lgg: Here "□" at xy_gg(3,109.3) line  from Lgg to xy_gg(4,83.6); Lgg: Here "□" at xy_gg(4,83.6) line  from Lgg to
xy_gg(5,68.3); Lgg: Here "□" at xy_gg(5,68.3) line  from Lgg to xy_gg(6,59.3); Lgg: Here "□" at xy_gg(6,59.3) line
from Lgg to xy_gg(7,54); Lgg: Here "□" at xy_gg(7,54) line  from Lgg to xy_gg(8,50); Lgg: Here "□" at
xy_gg(8,50) line  from Lgg to xy_gg(9,49); Lgg: Here "□" at xy_gg(9,49) line  from Lgg to xy_gg(10,47.6); Lgg:
Here "□" at xy_gg(10,47.6) Lgg: xy_gg(1,334.2) "∆" at xy_gg(1,334.2) line dashed from Lgg to xy_gg(2,168.2);
Lgg: Here "∆" at xy_gg(2,168.2) line dashed from Lgg to xy_gg(3,112.7); Lgg: Here "∆" at xy_gg(3,112.7) line
dashed from Lgg to xy_gg(4,85.3); Lgg: Here "∆" at xy_gg(4,85.3) line dashed from Lgg to xy_gg(5,68); Lgg: Here
"∆" at xy_gg(5,68) line dashed from Lgg to xy_gg(6,56.7); Lgg: Here "∆" at xy_gg(6,56.7) line dashed from Lgg to
xy_gg(7,48.9); Lgg: Here "∆" at xy_gg(7,48.9) line dashed from Lgg to xy_gg(8,43.1); Lgg: Here "∆" at
xy_gg(8,43.1) line dashed from Lgg to xy_gg(9,38.9); Lgg: Here "∆" at xy_gg(9,38.9) line dashed from Lgg to
xy_gg(10,35.1); Lgg: Here "∆" at xy_gg(10,35.1) Lgg: xy_gg(5,330) "□" at xy_gg(5,330) line  from Lgg to
xy_gg(6,330); Lgg: Here "□" at xy_gg(6,330) line  from Lgg to xy_gg(7,330); Lgg: Here "□" at xy_gg(7,330) box
invis wid 0 ht 1*textht "Shared-memory RTS" ljust at xy_gg(7.5,330) Lgg: xy_gg(5,300) "∆" at xy_gg(5,300) line
dashed from Lgg to xy_gg(6,300); Lgg: Here "∆" at xy_gg(6,300) line dashed from Lgg to xy_gg(7,300); Lgg: Here
"∆" at xy_gg(7,300) box invis wid 0 ht 1*textht "Distributed RTS" ljust at xy_gg(7.5,300)

]
```

Fig. 6. Measured execution times for the distributed and shared-memory implementations of the Traveling Salesman Problem.

4.2. Parallel alpha-beta search

Alpha-beta search is an efficient method for searching game trees for two-person, zero-sum games (e.g., chess). A node in such a game tree corresponds to a position in the game. Each node has one branch for every possible move in that position. A value associated with the node indicates how good that position is for the player who is about to move. At even levels of the tree, this value is the *maximum* of the values of its children; at odd levels it is the *minimum*, as the search algorithm assumes each player will choose the move that is least profitable for his or her opponent. The alpha-beta algorithm finds the best move in the current position, searching only part of the tree. It prunes moves that cannot lead to optimal positions.

We have implemented a parallel version of alpha-beta in Orca, using essentially the same algorithm as in [Bal et al. 1987]. Like the TSP program, the alpha-beta program consists of one manager process and a number of worker processes, one for each processor. The manager builds the top part of the search tree, up to a certain depth. This part of the tree is stored in a data-object shared among the manager and workers. Each worker repeatedly takes a leaf node of the top part of the tree and analyses the corresponding board position, using the normal (sequential) alpha-beta algorithm. After the evaluation has been finished, it uses the resulting value to update the alphas and betas of nodes in the (shared) top part of the tree.

The performance of the parallel alpha-beta program for a randomly generated search tree of depth 6 and fan-out 38 is shown in Figure 7. The speedup obtained (6.4 with 10 CPUs) is less than for branch-and-bound. This is not surprising, since alpha-beta search is hard to parallelize efficiently [Bal and Van Renesse 1986]. However, the performance differences between the distributed and nondistributed implementations of Orca are very small.

4.3. Parallel all-pairs shortest paths problem

The third and last application we describe here is the All-pairs Shortest Paths problem. In this problem it is desired to find the length of the shortest path from any node i to any other node j in a given graph. The parallel algorithm we use is similar to the one given in [Jenq and Sahni 1987], which is a parallel version of Floyd's algorithm. The distances between the nodes are represented in a matrix. Each processor computes part of the result matrix. The algorithm requires a nontrivial amount of communication and synchronization among the processors.

The performance of the program (for a graph with 200 nodes) on our two implementations is given in Figure 8. The shared-memory implementation is slightly more efficient. The performance difference is caused by the high communication overhead of the algorithm. The parallel algorithm performs 200 iterations; after each iteration, an array of 200 integers is sent from one processor to all other processors. In spite of this high communication overhead, the distributed implementation still has a good performance. With 10 CPUs, it achieves a speedup of 9.17 (as opposed to 9.48 for the shared-memory system). One of the main reasons for this good performance is the use of broadcast messages for transferring the array to all processors.

5. CONCLUSION

We have described a new model and programming language for implementing parallel applications on distributed systems. In contrast with most other models for distributed programming (e.g., the RPC model), our model allows processes on different machines to share data. The implementation of the model takes care of the physical distribution of shared data among processors. In particular, the implementation replicates shared data, so each process can directly read the local copy on its own processor.

The main purpose of this paper was to show that, for several applications, our model is both easy to use and efficient. We have studied one distributed implementation of our language and measured the performance of three applications. Our model is best suited for moderate-grained parallel applications in which processes share data that are read frequently and modified infrequently. As a good example, the TSP program of Section 4.1 uses a global variable that is read very frequently and is changed only a few times. This program shows an excellent performance. In the two other applications (Alpha-Beta search and the All-pairs Shortest Paths problem), the shared data are changed more frequently. Still, the performances of these applications are high, because we use an efficient mechanism for updating replicas, based on broadcasting rather than point-to-point messages.

```
Graph: [ # gg 1 .. 10, 200 .. 2200 define xy_gg (($1)−(.37))*. 292398,(($2)−(60))*0.00131579 define x_gg (($1)−(.37))*. 292398 define y_gg (($1)−(60))*0.00131579 frameht = 3 framewid = 3 Frame: box ht frameht wid framewid with .sw at 0,0 invis line from Frame.nw to Frame.ne invis line from Frame.sw to Frame.se line from Frame.sw to Frame.nw line from Frame.se to Frame.ne invis textht = Frame.w - (0.2,0) + (-.2,0) textht = .166667 Label: box invis wid 0 ht 1*textht "number of processors" with last line.end line left ticklen from (0,y_gg(400)) "400 " rjust at last line.end line left ticklen from (0,y_gg(600)) "600 " rjust at last line.end line left ticklen from (0,y_gg(800)) "800 " rjust at last line.end line left ticklen from (0,y_gg(1000)) "1000 " rjust at last line.end line left ticklen from (0,y_gg(1200)) "1200 " rjust at last line.end line left ticklen from (0,y_gg(1400)) "1400 " rjust at last line.end line left ticklen from (0,y_gg(1600)) "1600 " rjust at last line.end line left ticklen from (0,y_gg(1800)) "1800 " rjust at last line.end line left ticklen from (0,y_gg(2000)) "2000 " rjust at last line.end line left ticklen from (0,y_gg(2200)) "2200 " rjust at last line.end ticklen = .1 Ticks_gg:line down ticklen from (x_gg(1),0) box invis "1" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(2),0) box invis "2" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(3),0) box invis "3" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(4),0) box invis "4" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(5),0) box invis "5" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(6),0) box invis "6" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(7),0) box invis "7" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(8),0) box invis "8" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(9),0) box invis "9" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(10),0) box invis "10" ht .25 wid 0 with .n at last line.end Lgg: xy_gg(1,2058.6) "□" at xy_gg(1,2058.6) line from Lgg to xy_gg(2,1087.3); Lgg: Here "□" at xy_gg(2,1087.3) line from Lgg to xy_gg(3,804.3); Lgg: Here "□" at xy_gg(3,804.3) line from Lgg to xy_gg(4,612.3); Lgg: Here "□" at xy_gg(4,612.3) line from Lgg to xy_gg(5,522.3); Lgg: Here "□" at xy_gg(5,522.3) line from Lgg to xy_gg(6,469); Lgg: Here "□" at xy_gg(6,469) line from Lgg to xy_gg(7,404.3); Lgg: Here "□" at xy_gg(7,404.3) line from Lgg to xy_gg(8,369.3); Lgg: Here "□" at xy_gg(8,369.3) line from Lgg to xy_gg(9,344.6); Lgg: Here "□" at xy_gg(9,344.6) line from Lgg to xy_gg(10,321.6); Lgg: Here "□" at xy_gg(10,321.6) Lgg: xy_gg(1,2052.9) "∆" at xy_gg(1,2052.9) line dashed from Lgg to xy_gg(2,1084.2); Lgg: Here "∆" at xy_gg(2,1084.2) line dashed from Lgg to xy_gg(3,802.4); Lgg: Here "∆" at xy_gg(3,802.4) line dashed from Lgg to xy_gg(4,610.9); Lgg: Here "∆" at xy_gg(4,610.9) line dashed from Lgg to xy_gg(5,521.5); Lgg: Here "∆" at xy_gg(5,521.5) line dashed from Lgg to xy_gg(6,468.4); Lgg: Here "∆" at xy_gg(6,468.4) line dashed from Lgg to xy_gg(7,403.4); Lgg: Here "∆" at xy_gg(7,403.4) line dashed from Lgg to xy_gg(8,369); Lgg: Here "∆" at xy_gg(8,369) line dashed from Lgg to xy_gg(9,344.8); Lgg: Here "∆" at xy_gg(9,344.8) line dashed from Lgg to xy_gg(10,321.4); Lgg: Here "∆" at xy_gg(10,321.4) Lgg: xy_gg(5,2000) "□" at xy_gg(5,2000) line from Lgg to xy_gg(6,2000); Lgg: Here "□" at xy_gg(6,2000) line from Lgg to xy_gg(7,2000); Lgg: Here "□" at xy_gg(7,2000) box invis wid 0 ht 1*textht "Shared-memory RTS" ljust at xy_gg(7.5,2000) Lgg: xy_gg(5,1800) "∆" at xy_gg(5,1800) line dashed from Lgg to xy_gg(6,1800); Lgg: Here "∆" at xy_gg(6,1800) line dashed from Lgg to xy_gg(7,1800); Lgg: Here "∆" at xy_gg(7,1800) box invis wid 0 ht 1*textht "Distributed RTS" ljust at xy_gg(7.5,1800)

]
```

Fig. 7. Measured execution times for the distributed and shared-memory implementations of Alpha-Beta search.

ACKNOWLEDGEMENTS

We would like to thank Wim van Leersum for implementing the Orca compiler and Erik Baalbergen, Arnold Geels, and the anonymous referees for giving useful comments on the paper.

6. REFERENCES

Ahuja, S., Carriero, N., and Gelernter, D., Linda and Friends, *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, Aug. 1986.

Graph: [# gg 1 .. 10, 42 .. 450 define xy_gg (($1)−(.37))*.292398,(($2)−(13.44))*0.00644995 define x_gg (($1)−(.37))*.292398 define y_gg (($1)−(13.44))*0.00644995 frameht = 3 framewid = 3 Frame: box ht frameht wid framewid with .sw at 0,0 invis line from Frame.nw to Frame.ne invis line from Frame.sw to Frame.se line from Frame.sw to Frame.nw line from Frame.se to Frame.ne invis textht = .166667 textwid = .8 Label: box invis wid 0 ht 2*textht "Time" "(in seconds)" wid textwid with .e at Frame.w - (0.2,0) + (-.2,0) textht = .166667 Label: box invis wid 0 ht 1*textht "number of processors" with .n at Frame.s - (0,2 * textht) ticklen = .1 Ticks_gg:line left ticklen from (0,y_gg(50)) "50 " rjust at last line.end line left ticklen from (0,y_gg(100)) "100 " rjust at last line.end line left ticklen from (0,y_gg(150)) "150 " rjust at last line.end line left ticklen from (0,y_gg(200)) "200 " rjust at last line.end line left ticklen from (0,y_gg(250)) "250 " rjust at last line.end line left ticklen from (0,y_gg(300)) "300 " rjust at last line.end line left ticklen from (0,y_gg(350)) "350 " rjust at last line.end line left ticklen from (0,y_gg(400)) "400 " rjust at last line.end line left ticklen from (0,y_gg(450)) "450 " rjust at last line.end ticklen = .1 Ticks_gg:line down ticklen from (x_gg(1),0) box invis "1" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(2),0) box invis "2" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(3),0) box invis "3" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(4),0) box invis "4" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(5),0) box invis "5" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(6),0) box invis "6" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(7),0) box invis "7" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(8),0) box invis "8" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(9),0) box invis "9" ht .25 wid 0 with .n at last line.end line down ticklen from (x_gg(10),0) box invis "10" ht .25 wid 0 with .n at last line.end Lgg: xy_gg(1,398) "□" at xy_gg(1,398) line from Lgg to xy_gg(2,200.6); Lgg: Here "□" at xy_gg(2,200.6) line from Lgg to xy_gg(3,134.3); Lgg: Here "□" at xy_gg(3,134.3) line from Lgg to xy_gg(4,101); Lgg: Here "□" at xy_gg(4,101) line from Lgg to xy_gg(5,80.3); Lgg: Here "□" at xy_gg(5,80.3) line from Lgg to xy_gg(6,69.3); Lgg: Here "□" at xy_gg(6,69.3) line from Lgg to xy_gg(7,60); Lgg: Here "□" at xy_gg(7,60) line from Lgg to xy_gg(8,51); Lgg: Here "□" at xy_gg(8,51) line from Lgg to xy_gg(9,47.3); Lgg: Here "□" at xy_gg(9,47.3) line from Lgg to xy_gg(10,42); Lgg: Here "□" at xy_gg(10,42) Lgg: xy_gg(1,432.1) "∆" at xy_gg(1,432.1) line dashed from Lgg to xy_gg(2,218.9); Lgg: Here "∆" at xy_gg(2,218.9) line dashed from Lgg to xy_gg(3,148); Lgg: Here "∆" at xy_gg(3,148) line dashed from Lgg to xy_gg(4,111.4); Lgg: Here "∆" at xy_gg(4,111.4) line dashed from Lgg to xy_gg(5,90); Lgg: Here "∆" at xy_gg(5,90) line dashed from Lgg to xy_gg(6,77.1); Lgg: Here "∆" at xy_gg(6,77.1) line dashed from Lgg to xy_gg(7,66.3); Lgg: Here "∆" at xy_gg(7,66.3) line dashed from Lgg to xy_gg(8,58.2); Lgg: Here "∆" at xy_gg(8,58.2) line dashed from Lgg to xy_gg(9,53.6); Lgg: Here "∆" at xy_gg(9,53.6) line dashed from Lgg to xy_gg(10,47.1); Lgg: Here "∆" at xy_gg(10,47.1) Lgg: xy_gg(5,400) "□" at xy_gg(5,400) line from Lgg to xy_gg(6,400); Lgg: Here "□" at xy_gg(6,400) line from Lgg to xy_gg(7,400); Lgg: Here "□" at xy_gg(7,400) box invis wid 0 ht 1*textht "Shared-memory RTS" ljust at xy_gg(7.5,400) Lgg: xy_gg(5,360) "∆" at xy_gg(5,360) line dashed from Lgg to xy_gg(6,360); Lgg: Here "∆" at xy_gg(6,360) line dashed from Lgg to xy_gg(7,360); Lgg: Here "∆" at xy_gg(7,360) box invis wid 0 ht 1*textht "Multicast RTS" ljust at xy_gg(7.5,360)

]

Fig. 8. Measured execution times for the distributed and shared-memory implementations of the All-pairs Shortest Paths problem.

Almasi, G. S. and Gottlieb, A., Highly Parallel Computing, The Benjamin/Cummings Publishing Company, 1989.

Andrews, G. R. and Schneider, F. B., Concepts and Notations for Concurrent Programming, *ACM Computing Surveys*, Vol. 15, No. 1, pp. 3-43, Mar. 1983.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, Vol. 21, No. 3, Sep. 1989.

Bal, H. E. and Tanenbaum, A. S., Distributed Programming with Shared Data, *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, pp. 82-91, Miami, Fl., Oct. 1988.

Bal, H. E. and Van Renesse, R., A Summary of Parallel Alpha-Beta Search Results, *ICCA Journal*, Vol. 9, No. 3, pp. 146-149, Sep. 1986.

Bal, H. E., Van Renesse, R., and Tanenbaum, A. S., Implementing Distributed Algorithms Using Remote Procedure Calls, *Proc. AFIPS Nat. Computer Conf.*, Vol. 56, pp. 499-506, AFIPS Press, Chicago, Ill., June 1987.

Birman, K. P. and Joseph, T. A., Reliable Communication in the Presence of Failures, *ACM Trans. Comp. Syst.*, Vol. 5, No. 1, pp. 47-76, Feb. 1987.

Birrell, A. D. and Nelson, B. J., Implementing Remote Procedure Calls, *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Bisiani, R. and Forin, A., Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems, *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30, Palo Alto, Calif., Oct. 1987.

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

Cheriton, D. R., Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems, *ACM Operating Systems Review*, Vol. 19, No. 4, pp. 26-33, Oct. 1985.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., The Notions of Consistency and Predicate Locks in a Database System, *Commun. ACM*, Vol. 19, No. 11, pp. 624-633, Nov. 1976.

Garcia-Molina, H. and Spauster, A., Message Ordering in a Multicast Environment, *Proc. 9th Int. Conf. on Distr. Comp. Syst.*, pp. 354-361, Newport Beach, CA, June 1989.

Hoare, C. A. R., Monitors: An Operating System Structuring Concept, *Commun. ACM*, Vol. 17, No. 10, pp. 549-557, Oct. 1974.

Jenq, J.-F. and Sahni, S., All Pairs Shortest Paths on a Hypercube Multiprocessor, *Proc. of the 1987 Int. Conf. on Parallel Processing*, pp. 713-716, St. Charles, Ill., Aug. 1987.

Joseph, T. A. and Birman, K. P., Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems, *ACM Trans. Comp. Syst.*, Vol. 4, No. 1, Feb. 1987.

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

Kaashoek, M. F., Bal, H. E., and Tanenbaum, A. S., Experience with the Distributed Data Structure Paradigm in Linda, *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL., Oct. 1989aa.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., An Efficient Reliable Broadcast Protocol, Report IR-195, Vrije Universiteit, Amsterdam, The Netherlands, July 1989bb.

Li, K., IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. 1988 Int. Conf. Parallel Processing (Vol. II)*, pp. 94-101, St. Charles, Ill., Aug. 1988.

Metcalfe, R. M. and Boggs, D. R., Ethernet: Distributed Packet Switching for Local Computer Networks, *Commun. ACM*, Vol. 19, No. 7, pp. 395-404, July 1976.

Mullender, S. J. and Tanenbaum, A. S., The Design of a Capability-Based Distributed Operating System, *The Computer Journal*, Vol. 29, No. 4, pp. 289-300, Mar. 1986.

Tanenbaum, A. S. and Van Renesse, R., Distributed Operating Systems, *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470, Dec. 1985.

Tanenbaum, A. S. and Van Renesse, R., A Critique of the Remote Procedure Call Paradigm, *Proc. of the EUTECO 88 Conf.*, pp. 775-783, ed. R. Speth, North-Holland, Vienna, Austria, Apr. 1988.

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

Van Renesse, R., Van Staveren, J. M., and Tanenbaum, A. S., The Performance of the Amoeba Distributed Operating System, *Software—Practice and Experience*, Vol. 19, No. 3, pp. 223-234, Mar. 1989.