ORCA: A LANGUAGE FOR PARALLEL PROGRAMMING OF DISTRIBUTED SYSTEMS†

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

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

ABSTRACT

Orca is a language for implementing parallel applications on loosely coupled distributed systems. Unlike most languages for distributed programming, it 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 implementation of Orca takes care of the physical distribution of objects among the local memories of the processors. In particular, an implementation may replicate and/or migrate objects in order to decrease access times to objects and increase parallelism.

This paper gives a detailed description of the Orca language design and motivates the design choices. Orca is intended for applications programmers rather than systems programmers. This is reflected in its design goals to provide a simple, easy to use language that is type-secure and provides clean semantics.

The paper discusses three example parallel applications in Orca, one of which is described in detail. It also describes one of the existing implementations, which is based on reliable broadcasting. Performance measurements of this system are given for three parallel applications. The measurements show that significant speedups can be obtained for all three applications. Finally, the paper compares Orca with several related languages and systems.

1. INTRODUCTION

As communication in loosely coupled distributed computing systems gets faster, such systems become more and more attractive for running parallel applications. In the Amoeba system, for example, the cost of sending a short message between Sun workstations over an Ethernet is 1.1 milliseconds [1]. 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

† A preliminary version of this paper appeared in the proceedings of the First Usenix/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems, Ft. Lauderdale, Oct. 1989.

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

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 an existing sequential language extended with message passing for interprocess communication [2]. We felt that, for parallel applications, both the use of message passing and a sequential base language have many disadvantages, making them complicated for applications programmers to use.

Since then, we have developed a new language for distributed programming, called *Orca* [3, 4, 5]. Orca is intended for distributed applications programming rather than systems programming, and is therefore designed to be a simple, expressive, and efficient language with clean semantics. Below, we will briefly discuss the most important novelties in the language design.

Processes in Orca can communicate through shared data, even if the processors on which they run do not have physical shared memory. The main novelty of our approach is the way access to shared data is expressed. Unlike shared physical memory (or distributed shared memory [6]), shared data in Orca are accessed through user-defined high-level operations, which, as we will see, has many important implications.

Supporting shared data on a distributed system imposes some challenging implementation problems. We have worked on several implementations of Orca, one of which we will describe in the paper. This system uses a reliable broadcast protocol. Both the protocol and the integration with the rest of the system are new research results.

Unlike the majority of other languages for distributed programming, Orca is not an extension to an existing sequential language. Instead, its sequential and distributed constructs (especially data structures) have been designed together, in such a way that they integrate well. The language design addresses issues that are dealt with by few other languages. Most distributed languages simply add primitives for parallelism and communication to a sequential base language, but ignore problems due to poor integration with sequential constructs. A typical example is passing a pointer in a message, which is usually not detected and may cause great havoc. Orca provides a solution to this problem, and keeps the semantics of the language clean. At the same time, the Orca constructs are designed to have semantics close to conventional languages, thus making it easy for programmers to learn Orca.

An important goal in the design of Orca was to keep the language as simple as possible. Many interesting parallel applications exist outside the area of computer science, so the language must be suitable for general-applications programmers. Orca lacks low-level features that would only be useful for systems programming. In addition, Orca reduces complexity by avoiding language features aimed solely at increasing efficiency, especially if the same effect can be achieved through an optimizing compiler. Language designers frequently have to choose between adding language features or adding compiler optimizations. In general, we prefer the latter option. We will discuss several examples of this design principle in the paper. Finally, the principle of orthogonality [7] is used with care, but it is not a design goal by itself.

Another issue we have taken into account is that of debugging. As debugging of distributed programs is difficult, one needs all the help one can get, so we have paid considerable attention to debugging. Most important, Orca is a type-secure language. The language design allows the implementation to detect many errors during compile-time. In addition, the language run time system does extensive error checking.

The paper gives an overview of Orca, a distributed implementation of Orca, and its performance. It is structured as follows. In Section 2, we will describe the Orca language and motivate our design choices. In Section 3, we will present an example application written in Orca. In Section 4, we will discuss one implementation of Orca, based on reliable broadcast. We will also describe how to implement this broadcast primitive on top of LANs that only support unreliable broadcast. We will briefly compare this system with another implementation of Orca that uses Remote Procedure Call [8] rather than broadcasting. In Section 5, we will give performance measurements for several applications. In Section 6, we will compare our approach with those of related languages and systems. Finally, in Section 7 we will present our conclusions.

2. ORCA

Orca is a procedural, strongly typed language. Its sequential statements and expressions are fairly conventional and are roughly comparable (although not identical) 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, and general graphs. Pointers have intentionally been omitted to provide security. Also, the language lacks global variables, although such variables can be simulated by passing them around as reference parameters.

The rest of this section is structured as follows. We will first motivate our choice for shared data over message passing. Next, we will look at processes, which are used for expressing parallelism. Subsequently, we will describe Orca's communication model, which is based on shared data-objects. Synchronization of operations on shared objects is discussed next, followed by a discussion of hierarchically used objects. Finally, we will look at Orca's data structures.

2.1. Distributed Shared Memory

Most languages for distributed programming are based on message passing [9]. This choice seems obvious, since the underlying hardware already supports message passing. Still, there are many cases in which message passing is not the appropriate programming model. Message passing is a form of communication between two parties, which interact explicitly by sending and receiving messages. Message passing is less suitable, however, if several processes need to communicate indirectly, by sharing global state information.

There are many examples of such applications. For example, in parallel branch-and-bound algorithms the current best solution (the bound) is stored in a global variable accessed by all processes. This is not to say the algorithms actually need physical shared memory: they merely need *logically* shared data. Such algorithms are much harder to implement efficiently using message passing than using shared data.

The literature contains numerous other examples of distributed applications and algorithms that would greatly benefit from support for shared data, even if no physical shared memory is available. Applications described in the literature include: a distributed speech recognition system [10]; linear equation solving, three-dimensional partial differential equations, and split-merge sort [11]; computer chess [12]; distributed system services (e.g., name service, time service), global scheduling, and replicated files [13].

So, the difficulty in providing (logically) shared data makes message passing a poor match for many applications. Several researchers have therefore worked on communication models based on *logically shared data* rather than message passing. With these models, the programmer can use shared data, although the underlying hardware does not provide physical shared memory. A memory model that looks to the user as a shared memory but is implemented on disjoint machines is referred to as *Distributed Shared Memory (DSM)*.

Many different forms of DSM exist. Li's Shared Virtual Memory (SVM) [6] is perhaps the best-known example. It simulates physical shared-memory on a distributed system. The SVM distributes the pages of the memory space over the local memories. Read-only pages may also be replicated. SVM provides a clean, simple model, but unfortunately there are many problems in implementing it efficiently.

A few existing programming languages also fall into the DSM class. Linda [14] supports a globally shared Tuple Space, which processes can access using a form of associative addressing. On distributed systems, Tuple Space can be replicated or partitioned, much as pages in SVM are. The operations allowed on Tuple Space are low-level and built-in, which, as we will argue later, complicates programming and makes an efficient distributed implementation hard.

The Emerald language [15] is related to the DSM class, in that it provides a shared name space for objects, together with a location-transparent invocation mechanism. Emerald does not use any of the replication techniques that are typical of DSM systems, however.

The most important issue addressed by Orca is how data 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 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 is 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 key idea in Orca is to access shared data structures through higher level operations. Instead of using low-level instructions for reading, writing, and locking shared data, we 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

[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.

(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.

2.2. Processes

Parallelism in Orca is explicit, because compilers currently are not effective at generating parallelism automatically. Implicit parallelism may be suitable for vector machines, but, with the current state-of-the-art in compiler technology, it is not effective for distributed systems.

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

Initially, an Orca program consists of a single process, but new processes can be created explicitly through the **fork** statement:

```
fork name(actual-parameters) [ on (cpu-number) ];
```

This statement creates a new, anonymous, child process. Optionally, the new process can be assigned to a given processor. Processors are numbered sequentially; the **fork** statement may contain an **on**-part with an expression that specifies the processor on which to run the child process. If the **on**-part is absent, the child process is created on the same processor as its parent. The system does not move processes around on its own initiative, since this is undesirable for many parallel applications.

A process can take parameters, as specified in its definition. Two kinds are allowed: input and shared. A process may take any kind of data structure as value (input) parameter. In this case, the process gets a copy of the actual parameter. The parent can also pass any of its *data-objects* as a shared parameter to the child. In this case, the data-object will be shared between the parent and the child. The parent and child can communicate through this shared object, by executing the operations defined by the object's type, as will be explained later. For example, if a process *child* is declared as

```
process child(Id: integer; X: shared AnObjectType); begin ... end;
```

a new child process can be created as follows

```
MyObj: AnObjectType;    # declare an object
  ....
  # create a new child process, passing the constant 12 as
  # value parameter and the object MyObj as shared parameter.
fork child(12, MyObj);
```

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.

2.3. Shared data-objects and abstract data types

A shared data-object is a variable of an abstract data type (object type). An abstract data type definition in Orca consists of two parts: a *specification* part and an *implementation* part. The specification part defines the operations applicable to objects of the given type. As a simple example, the specification part of an object type encapsulating an integer is shown 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.

The implementation part contains the data used to represent objects of this type, the code to initialize the data of new instances of the type, and the code implementing the operations. Part of the implementation of type *IntObject* is shown in Figure 2.

```
object specification IntObject;
        x: integer;     # internal data

        operation Value(): integer;
        begin
                return x;       # return the current value
        end;

        operation Assign(v: integer);
        begin
                x := v;         # assign a new value
        end;
        ...
begin
        x := 0; # initialize objects to zero
end;
```

Fig. 2. Implementation part of an object type IntObject.

An operation implementation is similar to a procedure. An operation can only access its own local variables and parameters and the local (internal) data of the object it is applied to.

Once an object type has been defined, instances (objects) of the type can be created by declaring variables of the type. When an object is created, memory for the local variables of the object is allocated and the initialization code is executed. From then on, operations can be applied to the object. The Orca syntax to declare an object and apply an operation to it is illustrated below:

```
X: IntObject;
tmp: integer;

X$Assign(3);               # assign 3 to X
X$Add(1);                  # increment X
tmp := X$Value();          # read current value of X
```

Orca supports a single abstract data type mechanism, which can be used for encapsulating shared and non-shared data. In other words, the mechanism can also be used for regular (sequential) abstract data types. Even stronger, the same abstract type can be used for creating shared as well as local objects. Neither object declarations nor object-type declarations specify whether objects will be shared. This information is derived from the usage of objects: only objects that are ever passed as shared parameter in a **fork** statement are shared. All other objects are local and are treated as normal variables of an abstract data type.

Most other languages use different mechanisms for these two purposes. Argus [16], for example, uses clusters for local data and guardians for shared data; clusters and guardians are completely different. SR [17] provides a single mechanism (resources), but the overhead of operations on resources is far too high to be useful for sequential abstract data types [18].

The fact that shared data are accessed through user-defined operations is an important distinction between our model and other models. Shared virtual memory, for example, simulates physical shared memory, so shared data are accessed through low-level read and write operations. Linda’s Tuple Space model also uses a fixed number of built-in operations to add, read, and delete shared tuples. Having users define their own operations has many advantages, both for the ease of programming and for the implementation, as we will discuss shortly.

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 using message passing. 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 4, 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 [19], which centralize control to shared data.

2.4. Synchronization

An abstract data type in Orca can be used for creating shared as well as local objects. For objects that are shared among multiple processes, the issue of synchronization arises. Two types of synchronization exist: mutual exclusion synchronization and condition synchronization [20]. We will look at them in turn.

Mutual exclusion synchronization

Mutual exclusion in our model is done implicitly, by executing all operations on objects *indivisibly*. Conceptually, each operation locks the entire object it is applied to, does the work, and releases the lock only when it is finished. To be more precise, the model guarantees *serializability* [21] 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 increase 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.

Since users can define their own operations on objects, it is up to the user to decide which pieces of code should be executed indivisibly. For example, an abstract data type encapsulating an integer variable may have an operation to increment the integer. This operation will be done indivisibly. If, on the other hand, the integer is incremented through separate read and write operations (i.e., first read the current value, then write the incremented value back), the increment will be done as two separate actions, and will thus not be indivisible. 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. The model does not provide mutual exclusion at a granularity lower than the object level. Other languages (e.g., Sloop [22]) give programmers more accurate control over mutual exclusion synchronization.

Our model does not support indivisible operations on a collection of objects. Operations on multiple objects require a distributed locking protocol, which is complicated to implement efficiently. Moreover, this generality is seldom needed by parallel applications. We prefer to keep our basic model simple 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.

Condition synchronization

The second form of synchronization is condition synchronization, which allows processes to wait (block) until a certain condition becomes true. In our 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;
begin
    guard $condition_1$ do $statements_1$ od;
    ...
    guard $condition_n$ do $statements_n$ od;
end;
```

The conditions are Boolean expressions, called *guards*. To simplify the presentation, we will initially assume that guards are side-effect free. The problem of side effects will be considered later, when discussing hierarchically used objects.

The operation initially blocks until at least one of the guards evaluates to “true.” Next, one true guard is selected nondeterministically, and its sequence of statements is executed.

The Boolean expressions may depend on the parameters and local data of the operation and on the data of the object. If a guard fails, it can later become true, after the state of the object has been changed. It may thus be necessary to evaluate the guards several times.

We have chosen this form of condition synchronization because it is highly simple and fits well into the model. An alternative approach that we considered and rejected is to use a separate synchronization primitive, independent of the mechanism for shared objects. To illustrate the difference between these two alternatives, we will first look at a specific example.

Consider a shared *Queue* object with operations to add elements to the tail and retrieve elements from the head:

```
operation Add(x: item);  # add to tail
operation Get(): item;  # get from head
```

A process trying to fetch an element from an empty queue should not be allowed to continue. In other words, the number of *Get* operations applied to a queue should not exceed the number of *Add* operations. This is an example of a *synchronization constraint* on the order in which operations are executed. There are at least two conceivable ways for expressing such constraints in our model:

1. Processes trying to execute *Get* should first check the status of the queue and block while the queue is empty. Doing a *Get* on an empty queue results in an error.
2. The *Get* operation itself blocks while the queue is empty. Processes executing a *Get* on an empty queue therefore block automatically.

In both cases, a new primitive is needed for blocking processes. In the first case this primitive is to be used directly by user processes; in the second case only operations on objects use it. Also, the first approach calls for an extra operation on queues that checks if a given queue is empty. (For both approaches, unblocking the process and removing the head element from the queue should be done in one indivisible action, to avoid race conditions.)

The first approach has one major drawback: the *users* of an object are responsible for satisfying synchronization constraints. This is in contrast with the general idea of abstract data types to hide implementation details of objects from users. The second approach is much cleaner, as the *implementer* of the object takes care of synchronization and hides it from the users. We therefore use the second approach and do condition synchronization inside the operations. The model allows operations to block; processes can only block by executing operations that block.

An important issue in the design of the synchronization mechanism is how to provide blocking operations while still guaranteeing the indivisibility of operation invocations. If an operation may block at any point during its execution, operations can no longer be serialized. Our solution is to allow operations only to block *initially*, before modifying the object. An operation may wait until a certain condition becomes true, but once it has started executing, it cannot block again.

2.5. Hierarchical objects

Abstract data types are useful for extending a language with new types. This method for building new types is hierarchical: existing abstract data types can be used to build new ones. The internal data of an object can therefore themselves be objects. Note that hierarchical objects are not derived from the constituent objects by extending them (as can be done in object-oriented languages). The old and new objects have a “use” relation, not an “inheritance” relation.

This nesting of objects causes a difficult design problem, as we will explain below. Suppose we have an existing object type *OldType*, specified as follows:

```
object specification OldType;
  operation OldOperation1(): boolean;
  operation OldOperation2();
end;
```

We may use this object type in the implementation of another type (we omit the specification of this type):

```
object implementation NewType;
  NestedObject: OldType;  # a nested object
  operation NewOperation();
  begin
    guard NestedObject$OldOperation1() do
      ...
      NestedObject$OldOperation2();
    od;
  end;
end;
```

Objects of the new type contain an object, *NestedObject*, of type *OldType*. The latter object is called a *nested* object, because it is part of another object. Note that instances of *NewType* are still *single* objects whose operations are executed indivisibly. The nested object is invisible outside its enclosing object, just like any other internal data.

The implementer of *NewType* can be seen as a *user* of *OldType*. So, the implementer of *NewType* does not know how *OldType* is implemented. This lack of information about the *implementation* of the operations on *OldType* causes two problems.

The first problem is illustrated by the use of *OldOperation1* in the guard of *NewOperation*. We need to know whether the guard expressions have side effects, as they may have to be evaluated several times. Unfortunately, we do not know whether the invocation of *OldOperation1* has any side effects. If the operation modifies *NestedObject*, it does have side effects. We can only tell so, however, by looking at the *implementation* of this operation, which goes against the idea of abstract data types.

The second problem is more subtle. Suppose a process declares an object *NewObject* of type *NewType* and shares it with some of its child processes. If one of the processes invokes *NewOperation* on *NewObject*, the implementation of this object will invoke *OldOperation2* on the nested object. The problem is that the latter operation may very well *block*. If so, we violate the rule that operations are only allowed to block *initially*. In this situation, there are two equally unattractive options:

1. Suspend the process invoking *NewOperation*, but allow other processes to

access the object. This means, however, that the operation will no longer be indivisible.

2. Block the calling process, but do not allow any other processes to access the object. This implies that the process will be suspended forever, because no other process will be able to modify *NestedObject*.

One could solve this problem by disallowing blocking operations on nested objects, but again this requires looking at the *implementation* of an operation to see how it may be used.

Cooper and Hamilton have observed similar conflicts between parallel programming and data abstraction in the context of monitors [23]. They propose extending operation specifications with information about their implementation, such as whether or not the operation suspends or has any side effects. We feel it is not very elegant to make such concessions, however. The specification of an abstract data type should not reveal information about the implementation.

We solve these two problems by refining the execution model of operations. Conceptually, an operation is executed as follows. The operation repeatedly tries to evaluate its guards and then tries to execute the statements of a successful guard. Before evaluating a guard, however, the operation (conceptually) creates a copy of the entire object, including any nested (or deeply nested) objects. This copy is used during the evaluation of the guard and execution of the statements. The operation *commits* to a certain alternative, as soon as both

1. The guard succeeds (evaluates to true), and
2. The corresponding statements can be executed without invoking any blocking operations on nested objects.

As soon as a guard fails or the statements invoke a blocking operation, the copy of the entire object is thrown away and another alternative is tried. So, an operation does not commit until it has finished executing a successful guard and its corresponding statements, without invoking any blocking operations on nested objects. If all alternatives of an operation fail, the operation (and the process invoking it) blocks until the object is modified by another process. If an operation commits to a certain alternative, the object is assigned the current value of the copy (i.e., the value after evaluating the selected guard and statements).

This scheme solves both of the above problems. An operation on a nested object used inside a guard (e.g., *OldOperation1* in the code above) may have side effects; these side effects will not be made permanent until the guard is actually committed to. An operation on a nested object may also block. As long as all guards of that operation fail, however, the alternative containing the invocation will never be committed to. The operation has no effects until it commits to a certain alternative. Before commitment, it may try some alternatives, but their effects are thrown away. If the operation commits to an alternative, both the guards and statements of the alternative are executed without blocking. Therefore, operation invocations are still executed indivisibly.

The key issue is how to implement this execution model efficiently. It is quite expensive to copy objects before trying each alternative. In nearly all cases, however, the compiler will be able to optimize away the need for copying objects. Many object types will not have any nested objects, so they do not suffer from the problems described above. Also, an optimizing compiler can check if an operation used in a guard or body is side-effect free and nonblocking. To do so, it needs to access the implementation code of nested objects. This is not any different from other global optimizations (e.g., inline substitution), which basically need to access the entire source program. Also, the same mechanism can be used to test for circularities in nested object definitions.

Our solution therefore preserves abstraction from the programmer's point of view, but sometimes requires global optimizations to be efficient. The current Orca compiler performs these optimizations. This approach keeps the language simple and relies on optimization techniques for achieving efficiency.

2.6. Data structures

In most procedural languages, data structures like graphs, trees, and lists are built out of dynamically allocated and deallocated blocks of memory, linked together through *pointers*. For distributed programming, this approach has many disadvantages. The main difficulty is how to transmit a complex data structure containing pointers to a remote machine. Pointers, if implemented as addresses, are only meaningful within a single machine, so they need special treatment before being transmitted. Even more important, most languages do not consider such graphs to be first-class objects, so it is hard to determine *what* has to be transmitted.

In addition to these problems, giving the programmer explicit control over allocation and deallocation of memory usually violates type security. A programmer can deallocate memory and then use it again, leading to obscure bugs.

In Orca, these problems are solved through the introduction of a *graph* data type. A graph in Orca consists of zero or more *nodes*, each having a number of *fields*, similar to the fields of a record. Also, the graph itself may contain *global fields*, which are used to store information about the entire graph (e.g., the root of a tree or the head and tail of a list). Individual nodes within a graph are identified by values of a *nodename* type. A variable or field of a *nodename* type is initialized to NIL, which indicates it does not name any node yet. As an example, a binary-tree type may be defined as follows:

```
type node = nodename of BinTree;
type BinTree =
  graph  # global field:
    root: node; # name of the root of the tree
  nodes  # fields of each node:
    data: integer;
    LeftSon,
    RightSon: node;  # names of left and right sons
  end;
```

This program fragment declares a graph type *BinTree*. Each node of such a graph contains a data field and fields identifying the left and right sons of the node. Furthermore, the graph has one global field, identifying the root node of the tree.

A tree data structure is created by declaring a variable of this type. Initially, the tree is empty, but nodes can be added and deleted dynamically as follows:

```
t: BinTree;
n: node;

n := addnode(t);            # add a node to t, store its name in n
deletenode(t, n);           # delete the node with given name from t
```

The construct **addnode** adds a new node to a graph and returns a unique name for it, chosen by the run time system. The run time system also automatically allocates memory for the new node. In this sense, **addnode** is similar to the standard procedure *new* in Pascal [24]. As a crucial difference between the two primitives, however, the **addnode** construct specifies the data structure for which the new block of memory is intended. Unlike in Pascal, the run time system of Orca can keep track of the nodes that belong to a certain graph. This information is used whenever a copy of the graph has to be created, for example when it is passed as a value parameter to a procedure or remote process. Also, the information is used to delete the entire graph at the end of the procedure in which it is declared.

The global fields of a graph and the fields of its nodes are accessed through designators that are similar to those for records and arrays:

```
t.root := n;                        # access the global field of t
t[n].data := 12;                    # access data field of node n
t[n].LeftSon := addnode(t);         # create left son of n
n := t[n].LeftSon                   # store name of left son in n
```

Note that the designator for the field of a node specifies the name of the node as well as the graph itself. This notation differs from the one in Pascal, where nodes are identified by pointers only. The notation of Orca may be somewhat more cumbersome, but it has the advantage that it is always clear which data structure is accessed. Also, it makes it possible to represent a nodename as an index into a graph, rather than as a machine address. Nodenames can therefore be transmitted to remote machines without losing their meaning.

Graphs in Orca are type-secure. If a certain node is deleted from a graph and one of its fields is subsequently accessed, a run-time error occurs, as illustrated by the following piece of code:

```
n := addnode(t);
deletenode(t, n);
t[n].data := 12;  # causes a run-time error
```

The run time system checks whether the graph *t* contains a node with the given name. Furthermore, each invocation of **addnode**(t) returns a different name, so the same nodename will not be re-used for denoting a different node. Whenever a node has been deleted from a graph, any future references to the node will cause a run-time error.

The data structuring mechanism of Orca has some properties of arrays and some properties of pointer-based data structures. The mechanism supports dynamic allocation of memory through the **addnode** primitive. Graphs, like arrays, are first-class entities in Orca. This design has several advantages: they can easily be passed to remote processes; assignment is defined for graph variables; functions may return a value of a graph type; and graphs are automatically deallocated at the end of their enclosing procedure. The latter feature reduces the need for automatic garbage collection of nodes. Nodenames in Orca have the safety advantages of both pointers and array indices. Like pointers, they cannot be manipulated through arithmetic operations; like array indices, any illegal usage of a nodename will be detected at run time.

The graph type of Orca also has some disadvantages, compared to pointers. With pointers, for example, any two data structures can be hooked together through a single assignment statement. With graphs, this is more difficult. If the programmer anticipates the join, the data structures can be built using a single graph. If separate graphs are used, one will have to be copied into the other.

Another disadvantage is the run-time overhead of graphs. A graph is represented as a table with pointers to the actual nodes, so the nodes are accessed indirectly through this table [3]. Also, there is a cost in making graphs type-secure, since each node access has to be validated. We are currently working on decreasing these costs through global optimizations.

3. AN EXAMPLE OBJECT TYPE AND APPLICATION

In this section, we will give an example of an object type definition in Orca and of a parallel application that uses this object type. The object defines a generic job queue type, with operations to add and delete jobs. It is used in several parallel programs based on the replicated workers paradigm. With this paradigm, a master process repeatedly generates jobs to be executed by workers. Communication between the master and workers takes place through the job queue. One such application, parallel branch-and-bound, will be discussed.

3.1. An example object type

The specification of the object type GenericJobQueue is shown in Figure 3. The formal parameter *T* represents the type of the elements (jobs) of the queue.

```
generic (type T)
object specification GenericJobQueue;
    operation AddJob(job: T);          # add a job to the tail of the queue
    operation NoMoreJobs();            # invoked when no more jobs will be added
    operation GetJob(job: out T): boolean;
        # Fetch a job from the head of the queue. This operation
        # fails if the queue is empty and NoMoreJobs has been invoked.
end generic;
```

Fig. 3. Specification part of the object type definition GenericJobQueue.

Three different operations are defined on job queues. *AddJob* adds a new job to the tail of the queue. The operation *NoMoreJobs* is to be called when no more jobs will be added to the queue (i.e., when the master has generated all the jobs). Finally, the operation *GetJob* tries to fetch a job from the head of the queue. If the queue is not empty, *GetJob* removes the first job from the queue and returns it through the **out** parameter *job*; the operation itself returns “true” in this case. If the queue is empty and the operation *NoMoreJobs* has been applied to the queue, the operation fails and returns “false”. If none of these two conditions—queue not empty or *NoMoreJobs* invoked—holds, the operation blocks until one of them becomes true.

The implementation part is shown in Figure 4. Objects of this type contain two variables: a Boolean variable *done* and a variable *Q* of type *queue*. The latter type is defined as a

graph with two global fields, identifying the first and last element of the queue. Each element contains the **nodename** of the next element in the queue and data of formal type *T*.

The implementation of *AddJob* uses straightforward list manipulation. The *GetJob* operation is more interesting. It contains two **guards**, reflecting the two conditions described above.

3.2. An example parallel application in Orca

We will now look at one example application in Orca: 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 problem is solved using a parallel branch-and-bound algorithm.

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 job queue, as shown in Figure 5.

The Orca code for the master and worker processes is shown in Figure 6. The master process creates and initializes the shared object *minimum*, and forks one worker process on each processor except its own one. Subsequently, it generates the jobs by calling a function *GenerateJobs* (not shown here) and then forks a worker process on its own processor. In this way, job generation executes in parallel with most of the worker processes. The final worker process is not created until all jobs have been generated, so job generation will not be slowed down by a competing process on the same processor.

Each worker process repeatedly fetches a job from the job queue and executes it by calling the function *tsp*. The *tsp* function generates all routes that start with a given initial route. If the initial route passed as parameter is longer than the current best route, *tsp* returns immediately, because such a partial route cannot lead to an optimal solution. If the route passed as parameter is a full route (visiting all cities), a new best route has been found, so the value of *minimum* should be updated. It is possible, however, that two or more worker processes simultaneously detect a route that is better than the current best route. Therefore, the value of *minimum* is updated through the indivisible operation *Min*, which checks if the new value presented is actually less than the current value of the object.

If the job queue is empty and no more jobs will be generated, the operation *GetJob* will return “false” and the workers will terminate.

4. 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 several implementations of the language [3]. The implementation described in this paper is

```
generic
object implementation GenericJobQueue;
        type ItemName = nodename of queue;
        type queue =
                graph  # a queue is represented as a linear list
                        first, last: ItemName;              # first/last element of queue
                nodes
                        next: ItemName;                     # next element in queue
                        data: T;                            # data contained by this element
                end;

        done: boolean;  # set to true if NoMoreJobs has been invoked.
        Q: queue;       # the queue itself

        operation AddJob(job: T);
                p: ItemName;
        begin  # add a job to the tail of the queue
                p := addnode(Q);            # add a new node to Q, return its name in p
                Q[p].data := job;           # fill in data field of the new node; next field is NIL
                if Q.first = NIL then       # Is it the first node?
                        Q.first := p;       # yes; assign it to global data field
                else
                        Q[Q.last].next := p;  # no; set predecessor's next field
                fi;
                Q.last := p;                # Assign to "last" global data field
        end;

        operation NoMoreJobs();
        begin       # Invoked to indicate that no more jobs will be added
                done := true;
        end;

        operation GetJob(job: out T): boolean;
                p: ItemName;
        begin       # Try to fetch a job from the queue
                guard Q.first /= NIL do             # A job is available
                        p := Q.first;               # Remove it from the queue
                        Q.first := Q[p].next;
                        if Q.first = NIL then Q.last := NIL; fi;
                        job := Q[p].data;           # assign to output parameter
                        deletenode(Q,p);            # delete the node from the queue
                        return true;                # succeeded in fetching a job
                od;

                guard done and (Q.first = NIL) do
                        return false;       # All jobs have been done
                od;
        end;

begin       # Initialization code for JobQueues ; executed on object creation.
        done := false;  # initialize done to false
end generic;
```

Fig. 4. Implementation part of the object type definition GenericJobQueue.

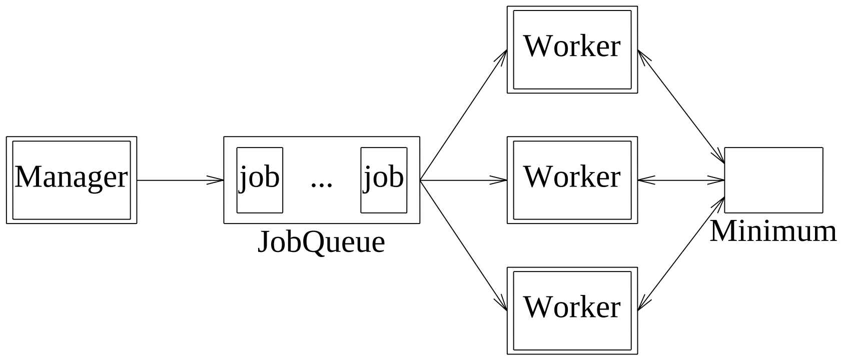

Fig. 5. Structure of the Orca implementation of TSP. The Manager and Workers are processes. The JobQueue is a data-object shared among all these processes. Minimum is a data-object of type IntObject; it is read and written by all workers.

based on *replication* and *reliable broadcasting*. We will briefly discuss a second implementation in Section 4.4.

Replication of data is used in several fault-tolerant systems (e.g., ISIS [25]) 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.

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 7). 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.

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, to create and manipulate 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 broadcasting, so that the

```
type PathType = array[integer] of integer;
type JobType =
        record
                len: integer;      # length of partial route
                path: PathType;# the partial route itself
        end;
type DistTab =  ...;   # distances table
object TspQueue = new GenericJobQueue(JobType);
   # Instantiation of the GenericJobQueue type


process master();
        minimum: IntObject;        # length of current best path (shared object)
        q: TspQueue;               # the job queue (shared object)
        i: integer;
        distance: DistTab;                 # table with distances between cities
begin
        minimum$assign(MAX(integer));             # initialize minimum to infinity
        for i in 1.. NCPUS() - 1  do
                # fork one worker per processor, except current processor
                fork worker(minimum, q, distance) on(i);
        od;
        GenerateJobs(q, distance);# main thread generates the jobs
        q$NoMoreJobs();      # all jobs have been generated now
        fork worker(minimum, q, distance) on(0);
           # jobs have been generated; fork a worker on this cpu too
end;


process worker(
                minimum: shared IntObject;                  # length of current best path
                q: shared TspQueue;                         # job queue
                distance: DistTab)                          # distances between cities

        job: JobType;
begin
        while q$GetJob(job) do    # while there are jobs to do:
                tsp(job.len, job.path, minimum, distance);
                # do sequential tsp
        od;
end;
```

Fig. 6. Orca code for the master and worker processes of TSP.

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.

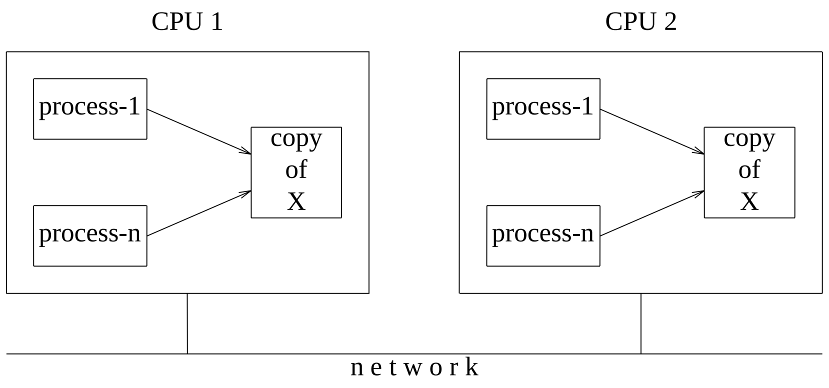

Fig. 7. Replication of data-objects in a distributed system

4.1. Top layer: Orca application programs

Application programs are translated by the Orca compiler into executable code for the target system.[2] 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] In most languages, this optimization would be difficult to implement. Consider, for example, a Pascal statement containing an indirect assignment through a pointer variable:

```
p^.f := 0;
```

It is hard to determine which data structure is affected by this statement. Orca does not have this problem, since the name of the data structure is given by the programmer. The Orca equivalent of the Pascal code given above would look like:

```
G[n].f := 0;
```

which explicitly specifies the name of the data structure that will be modified. So, in Orca the compiler can determine which operations modify the object's data structures and which do not.

The compiler stores its 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. (It is a network-

[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.

wide name for the object.) 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.

4.2. 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, such as where to replicate objects, how to synchronize write operations to replicated objects, and whether to update or invalidate copies after a write operation. We have looked at many alternative strategies [26]. The RTS described in this paper uses full replication of objects, updates replicas by applying write operations to all replicas, and implements mutual exclusion synchronization through a distributed update protocol.

The full replication scheme was chosen for its simplicity and good performance for many applications. An alternative is to let the RTS decide dynamically where to store replicas. This strategy is employed in another implementation of Orca [26].

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 bit vector). 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 no more CPU time and network bandwidth than 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 must be modified. If this updating is not done as one indivisible action, different processors will temporarily have different values for the same logical data, which is unacceptable.

The semantics of shared data-objects in our 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 observe either 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 4.3, 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. Objects (and replicas) are stored in an address space shared by the object-manager and user processes. User processes can *read* local copies directly, without intervention by the object-managers. Write operations on shared objects, on the other hand, are marshalled and then broadcast to all the 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 8.

```
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. 8. 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 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 9).

```
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. 9. 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.

4.3. Bottom layer: Reliable broadcast

In this section we describe a simple 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, without going into too much detail about the protocol itself.

With current microprocessors and LANs, lost or damaged packets and processor crashes occur 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 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 directly 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. The sequencer can explicitly ask for this information if it runs out of history space,

Besides the protocol described above (Method 1), we have designed and implemented another protocol (Method 2) that does not send messages to the sequencer first. Instead, the kernel of the sender immediately broadcasts the message. Each receiving kernel stores the message and the sequencer broadcasts a short acknowledgement message for it. These acknowledgements again carry sequence numbers, which define the ordering of the original messages. If a kernel receives an acknowledgement with the right (i.e., next in line) sequence number, it delivers the original message to the application.

Both protocols guarantee the same semantics, but have different performances under different circumstances. With Method 1, each message is sent over the network twice (once to the sequencer and once from the sequencer to the other kernels). Method 2 uses less bandwidth than Method 1 (the message appears only once on the network), but generates more interrupts, because it uses two broadcast messages (one from the sender to the other kernels and one short message from the sequencer to all kernels). For the implementation of the Orca run time system we use Method 1, because the messages generated by the run time system are short and because Method 1 steals less computing cycles from the Orca application to handle interrupts.

In philosophy, the protocol described above somewhat resembles the one described by Chang and Maxemchuk [27], but they differ in some major aspects. With our protocol, 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 [28], Garcia-Molina and Spauster [29], and several others) is given in [30].

4.4. Comparison with an RPC-based Protocol

Above, we have described one implementation of Orca, based on full replication of objects and on a distributed update protocol using indivisible broadcasting. Below, we will compare this implementation with another one based on partial replication and Remote Procedure Call (RPC).

Updating replicas with RPC is more complicated than with indivisible broadcast. The problem is that all replicas must be updated in a consistent way. To assure consistency, the RPC system uses a two-phase update protocol. During the first phase, all copies are updated and locked. After all updates have been acknowledged, the second phase begins, during which all copies are unlocked.

This protocol is much more expensive than the one based on broadcasting. The time for an update to complete depends on the number of copies. It therefore makes sense to use a *partial* replication strategy, and only replicate objects where they are needed. The RPC system maintains statistics about the number of read and write operations issued by each processor for each object. Based on this information, it decides dynamically where to store the object and where to keep copies. The system can dynamically migrate the object or create and delete copies.

The statistics impose some overhead on the operations, but in general the savings in communication time are well worth this overhead. Still, in most cases, the RPC system has more communication costs than the broadcast system. For the TSP program, for example, it is far more efficient to update the global bound variable through a single broadcast message than through multiple RPCs.

The RPC system is more efficient if the read/write ratio of an object is low. In this case, the broadcast system will needlessly replicate the object, but the RPC system will observe this behavior and decide dynamically not to replicate the object.

5. PERFORMANCE OF EXAMPLE APPLICATIONS

In this section we will take a brief look at the performance of some example Orca programs. The main goal of this section is to show that, at least for some realistic applications, good speedups can be obtained with our approach.

The prototype distributed implementation we use is based on the layered approach described in the previous section. The prototype runs on top of the Amoeba system, which has been extended with the broadcast protocol described earlier.

The implementation runs on a distributed system, containing 16 MC68030 CPUs (running at 16 Mhz) connected to each other through an 10 Mbit/s Ethernet [31]. The implementation uses Ethernet multicast communication to broadcast a message to a group of processors. All processors are on one Ethernet and are connected to it by Lance chip interfaces.

The performance of the broadcast protocol on the Ethernet system is described in [30]. The time needed for multicasting a short message reliably to two processors is 2.6 msec. With 16 receivers, a multicast takes 2.7 msec.[4] This high performance is due to the fact that

[4] In an earlier implementation of the protocol [32] the delay was 1.4 msec. The difference is entirely due to a new routing protocol on which the group communication protocol is implemented. (The Amoeba kernel can now deal with different kinds of networks and route messages dynamically over multiple networks).

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 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. In addition to TSP, smaller applications include matrix multiplication, prime number generation, and sorting. Below, we will give performance measurements of three sample programs running on the Ethernet implementation.

5.1. Parallel Traveling Salesman Problem

The first application, the Traveling Salesman Problem (TSP), was described in Section 3.2. The program uses two shared objects: a job queue and an IntObject containing the length of the current best path (see Figure 5). 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 one broadcast message.

Although updates of the best path happen infrequently, it is 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.

The performance of the traveling salesman program (for a randomly generated graph with 12 cities) is given in Figure 10. The implementation achieves a speedup close to linear. With 16 CPUs it is 14.44 times faster than with 1 CPU.

5.2. Parallel all-pairs shortest paths problem

The second application we describe here is the All-pairs Shortest Paths problem (ASP). 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 [33], 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 300 nodes) is given in Figure 11. The parallel algorithm performs 300 iterations; after each iteration, an array of 300 integers is sent from one processor to all other processors. In spite of this high communication overhead, the implementation still has a good performance. With 16 CPUs, it achieves a speedup of 15.88. One of the main reasons for this good performance is the use of broadcast messages for transferring the array to all processors.

5.3. Successive Overrelaxation

Both TSP and ASP benefit from the use of broadcasting. We will now consider an application that only needs point-to-point message passing. The application is successive overrelaxation (SOR), which is an iterative method for solving discretized Laplace equations on a grid.

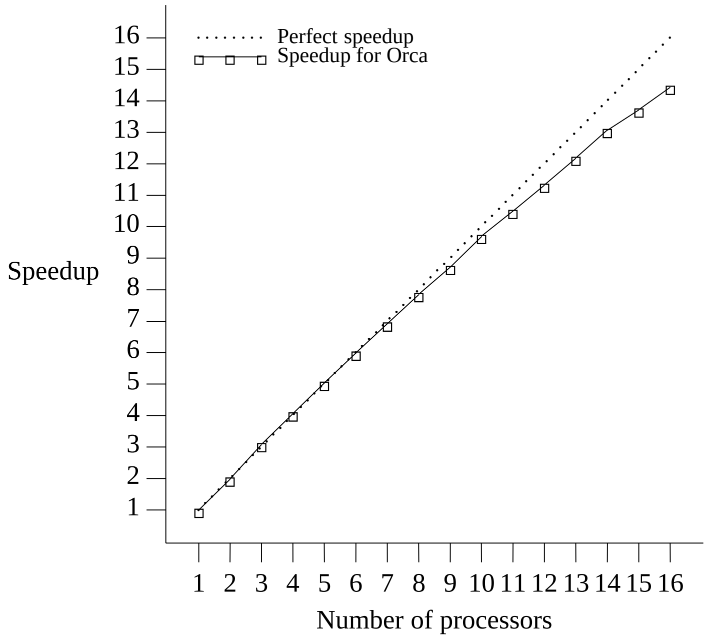

Fig. 10. Measured speedup for the Orca implementation of the Traveling Salesman Problem.

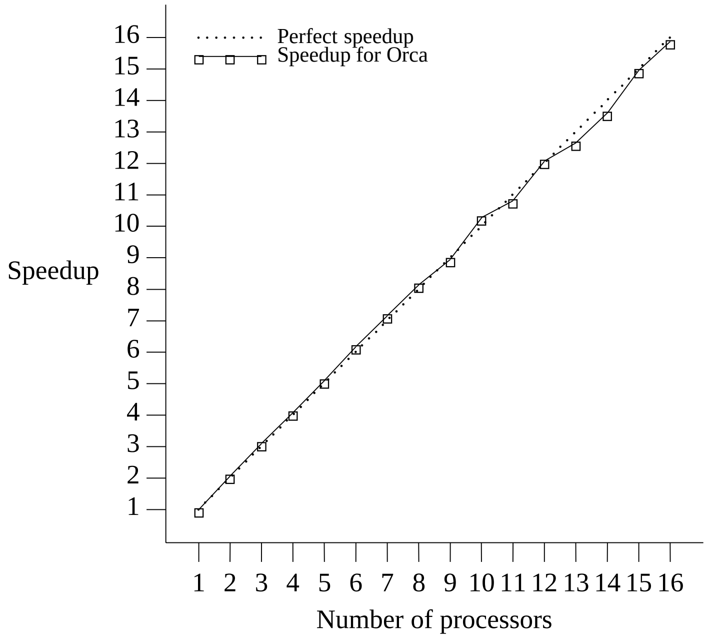

Fig. 11. Measured speedup for the Orca implementation of the All-pairs Shortest Paths problem.

During each iteration, the algorithm considers all non-boundary points of the grid. For each point, SOR first computes the average value of its four neighbors and then updates the point using this value.

We have parallelized SOR by partitioning the grid into regions and assigning these regions to different processors. The partitioning of the grid is such that, at the beginning of an iteration, each processor needs to exchange values with only two other processors. The parallel algorithm therefore only needs point-to-point message passing. With our current prototype implementation of Orca, however, all communication is based on broadcasting. The message passing is simulated in Orca by having the sender and receiver share a buffer object. Since shared objects are updated through broadcasting, all processors will receive the update message. So, SOR is a worst-case example for our system.

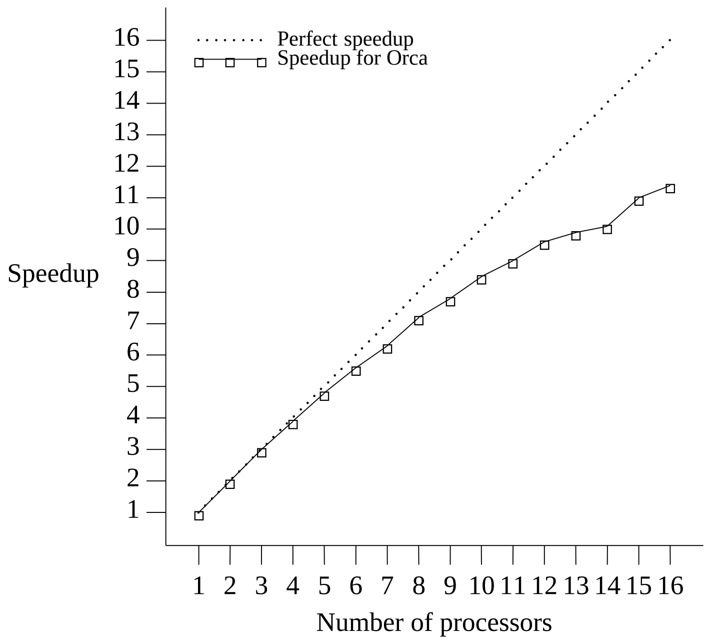

Fig. 12. Measured speedup for the Orca implementation of Successive Overrelaxation.

The measured speedup for SOR is shown in Figure 12. Despite the high communication overhead, the program still achieves a reasonable speedup. The speedup on 16 CPUs is 11.4.

6. RELATED WORK

In this section, we will compare our language with several related languages and systems. In particular, we will look at objects (as used in parallel object-based languages), Linda’s Tuple Space, and Shared Virtual Memory.

Objects

Objects are used in many object-based languages for parallel or distributed programming, such as Emerald [15], Amber [34], and ALPS [35]. Objects in such languages typically have two parts:

1. Encapsulated data.
2. A *manager process* that controls access to the data.

The data are accessed by sending a message to the manager process, asking it to perform a certain operation on the data. As such objects contain a process as well as data, they are said to be *active*.

Although, in some sense, parallel object-based languages allow processes (objects) to share data (also objects), their semantics are closer to message passing than to shared variables. Access to the shared data is under full control of the manager process. In ALPS, for example, all operations on an object go through its manager process, which determines the order in which the operations are to be executed. Therefore, the only way to implement the model is to store an object on one specific processor, together with its manager process, and to translate all operations on the object into remote procedure calls to the manager process.

Our model does not have such centralized control. Objects in Orca are purely passive: they contain data, but no manager process. Access control to shared data-objects is distributed; it is basically determined by only two rules:

1. Operations must be executed indivisibly.
2. Operations are blocked while their guards are false.

Therefore, the model can be implemented by replicating data-objects on multiple processors, as we discussed in Section 4. Read operations can be applied to the local copy, without any message passing being involved. Moreover, processes located on different processors can apply read operations simultaneously, without losing any parallelism.

Linda's Tuple Space

Linda [14] is one of the first languages to recognize the disadvantages of central manager processes for guarding shared data. Linda supports so-called *distributed data structures*, which can be accessed simultaneously by multiple processes. In contrast, object-based languages typically serialize access to shared data structures. Linda uses the Tuple Space model for implementing distributed data structures.

In general, distributed data structures in Linda are built out of multiple tuples. Different tuples can be accessed independently from each other, so processes can manipulate different tuples of the same data structure simultaneously. In principle, multiple **read** operations of the same tuple can also be executed simultaneously. Tuples are (conceptually) modified by taking them out of Tuple Space first, so modifications of a given tuple are executed strictly sequentially.

Although the idea of distributed data structures is appealing, we think the support given by the Tuple Space for implementing such data structures has important disadvantages. For distributed data structures built out of single tuples, mutual exclusion synchronization is done automatically. Operations on complex data structures (built out of multiple tuples), however, have to be synchronized explicitly by the programmer. In essence, Tuple Space supports a fixed number of built-in operations that are executed indivisibly, but its support for building more complex indivisible operations is too low-level [36].

In Orca, on the other hand, programmers can define operations of arbitrary complexity on shared data structures; all these operations are executed indivisibly, so mutual exclusion synchronization is always done automatically by the run time system. This means it is the job of the implementation (the compiler and run time system) to see which operations can be executed in parallel and which have to be executed sequentially. As discussed above, one way of doing this is by distinguishing between read and write operations and executing reads in parallel on local copies; more advanced implementations are also feasible.

Shared Virtual Memory

Shared Virtual Memory (SVM) [6] simulates physical shared memory on a distributed system. It partitions the global address space into fixed-sized pages, just as with virtual memory. Each processor contains some portion of the pages. If a process tries to access a page that it does not have, it gets a page-fault, and the operating system will then fetch the page from wherever it is located. Read-only pages may be shared among multiple processors. Writable pages must reside on a single machine. They cannot be shared. If a processor needs to modify a page, it will first have to invalidate all copies of the page on other processors.

There are many important differences between the implementation of our model and SVM. SVM is (at least partly) implemented inside the operating system, so it can use the MMU registers. In Orca, everything except for the broadcast protocol is implemented in software outside the operating system. This difference gives SVM a potential performance advantage.

Still, our model has important advantages over SVM. First, shared data-objects are accessed through well-defined, high-level operations, whereas SVM is accessed through low-level read and write instructions. Consequently, we have a choice between invalidating objects after a write operation or updating them by applying the operation to all copies (or, alternatively, sending the new value). With SVM, there is no such choice; only invalidating pages is viable [6]. In many cases, however, invalidating copies will be far less efficient than updating them.

Several researchers have tried to solve this performance problem by relaxing the consistency constraints of the memory (e.g., [37, 38]). Although these weakly consistent memory models may have better performance, we fear that they also ruin the ease of programming for which DSM was designed in the first place. Since Orca is intended to simplify applications programming, Orca programmers should not have to worry about consistency. (In the future, we may investigate whether a compiler is able to relax the consistency transparently, much as is done in the Munin system [39].)

A second important difference between Orca and SVM is the granularity of the shared data. In SVM, the granularity is the page-size, which is fixed (e.g. 4K). In Orca, the granularity is the object, which is determined by the user. So, with SVM, if only a single bit of a page is modified, the whole page has to be invalidated. This property leads to the well-known problem of ‘‘false sharing.’’ Suppose a process *P* repeatedly writes a variable *X* and process *Q* repeatedly writes *Y*. If *X* and *Y* happen to be on the same page, this page will continuously be moved between *P* and *Q*, resulting in thrashing. If *X* and *Y* are on different pages, thrashing will not occur. Since SVM is transparent, however, the programmer has no control over the allocation of variables to pages. In Orca, this problem does not occur, since *X* and *Y* would be separate objects and would be treated independently.

A more detailed comparison between our work and Shared Virtual Memory is given in [40].

7. CONCLUSION

We have described a new model and language for parallel programming of distributed systems. In contrast with most other models for distributed programming, our model allows processes on different machines to share data. The key idea in our model is to encapsulate shared data in data-objects and to access these objects through user-defined operations. The advantages of this approach for the programmer and the implementer are summarized below.

Since operations on objects are always executed indivisibly, mutual exclusion synchronization is done automatically, which simplifies programming. Condition synchronization is integrated into the model by allowing operations to suspend. The mechanism for suspending operations is easy to use and is only visible to the implementer of the operations and not to their users.

The implementation of our 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. After a write operation, all replicas are updated by broadcasting the operation. This update strategy is only possible because shared data are accessed through user-defined operations. SVM, for example, cannot efficiently update replicas after a write operation, since a logical write operation may require many machine instructions, each modifying memory. Updating the memory by broadcasting the machine instructions would be highly inefficient, as the communication overhead per instruction would be enormous.

We have also defined a language, Orca, based on shared data-objects. The design of Orca avoids problems found in many other distributed languages, such as pointers and global variables. A major goal in the design was to keep the language simple. In particular, we have given several examples of simplifying the language design by having the compiler do certain optimizations.

We have studied one distributed implementation of Orca. This implementation runs on a collection of processors connected through a broadcast network. We have not looked at implementations of Orca on other systems, such as hypercubes. Such an implementation would be feasible, however, since the Orca language itself does not depend on the network topology. To port Orca to other architectures, a new run time system (probably with a new replication strategy) would be needed, but the language and its application programs would not have to be changed.

Our approach is best suited for moderate-grained parallel applications in which processes share data that are read frequently and modified infrequently. A good example is the TSP program, which uses a shared object that is read very frequently and is changed only a few times. This program shows an excellent performance. The applications also benefit from the efficient broadcast protocol used in our implementation. The usefulness of broadcasting was demonstrated by the ASP program.

In conclusion, we think that Orca is a useful language for writing parallel programs for distributed systems. Also, we have shown that the language is efficient for a range of applications.

ACKNOWLEDGEMENTS

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

REFERENCES

1. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum, ‘‘Experiences with the Amoeba Distributed Operating System,’’ *Comm. ACM* **33**(2), pp. 46-63 (Dec. 1990).
2. H.E. Bal, R. van Renesse, and A.S. Tanenbaum, ‘‘Implementing Distributed Algorithms Using Remote Procedure Calls,’’ *Proc. AFIPS Nat. Computer Conf.*, Chicago, Ill. **56**, pp. 499-506, AFIPS Press (June 1987).
3. H.E. Bal, *Programming Distributed Systems,* Silicon Press, Summit, NJ (1990).
4. H.E. Bal and A.S. Tanenbaum, ‘‘Distributed Programming with Shared Data,’’ *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, Fl., pp. 82-91 (Oct. 1988).
5. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ‘‘Experience with Distributed Programming in Orca,’’ *Proceedings IEEE CS 1990 International Conference on Computer Languages*, New Orleans, LA, pp. 79-89 (March 1990).
6. K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *Proc. 5th Ann. ACM Symp. on Princ. of Distr. Computing*, Calgary, Canada, pp. 229-239 (Aug. 1986).
7. C. Ghezzi and M. Jazayeri, *Programming Language Concepts,* John Wiley, New York, NY (1982).
8. A.D. Birrell and B.J. Nelson, ‘‘Implementing Remote Procedure Calls,’’ *ACM Trans. Comp. Syst.* **2**(1), pp. 39-59 (Feb. 1984).
9. H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, ‘‘Programming Languages for Distributed Computing Systems,’’ *ACM Computing Surveys* **21**(3) (Sept. 1989).
10. R. Bisiani and A. Forin, ‘‘Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems,’’ *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., pp. 21-30 (Oct. 1987).
11. K. Li, ‘‘IVY: A Shared Virtual Memory System for Parallel Computing,’’ *Proc. 1988 Int. Conf. Parallel Processing (Vol. II),* St. Charles, Ill., pp. 94-101 (Aug. 1988).
12. E.W. Felten and S.W. Otto, ‘‘A Highly Parallel Chess Program,’’ *Proc. of the Int. Conf.*

on Fifth Generation Computer Systems 1988, Tokyo, pp. 1001-1009 (Nov. 1988).

13. D.R. Cheriton, ‘‘Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems,’’ *ACM Operating Systems Review* **19**(4), pp. 26-33 (Oct. 1985).

14. S. Ahuja, N. Carriero, and D. Gelernter, ‘‘Linda and Friends,’’ *IEEE Computer* **19**(8), pp. 26-34 (Aug. 1986).

15. E. Jul, H. Levy, N. Hutchinson, and A. Black, ‘‘Fine-Grained Mobility in the Emerald System,’’ *ACM Trans. Comp. Syst.* **6**(1), pp. 109-133 (Feb. 1988).

16. B. Liskov, ‘‘Distributed Programming in Argus,’’ *Commun. ACM* **31**(3), pp. 300-312 (March 1988).

17. G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, ‘‘An Overview of the SR Language and Implementation,’’ *ACM Trans. Program. Lang. Syst.* **10**(1), pp. 51-86 (Jan. 1988).

18. H.E. Bal, ‘‘An Evaluation of the SR Language Design,’’ report IR-219, Vrije Universiteit, Amsterdam (August 1990).

19. C.A.R. Hoare, ‘‘Monitors: An Operating System Structuring Concept,’’ *Commun. ACM* **17**(10), pp. 549-557 (Oct. 1974).

20. G.R. Andrews and F.B. Schneider, ‘‘Concepts and Notations for Concurrent Programming,’’ *ACM Computing Surveys* **15**(1), pp. 3-43 (March 1983).

21. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, ‘‘The Notions of Consistency and Predicate Locks in a Database System,’’ *Commun. ACM* **19**(11), pp. 624-633 (Nov. 1976).

22. S.E. Lucco, ‘‘Parallel Programming in a Virtual Object Space,’’ *SIGPLAN Notices (Proc. Object-Oriented Programming Systems, Languages and Applications 1987)*, Orlando, FL **22**(12), pp. 26-34 (Dec. 1987).

23. R.C.B. Cooper and K.G. Hamilton, ‘‘Preserving Abstraction in Concurrent Programming,’’ *IEEE Trans. Softw. Eng.* **SE-14**(2), pp. 258-263 (Feb. 1988).

24. N. Wirth, ‘‘The Programming Language Pascal,’’ *Acta Informatica* **1**(1), pp. 35-63 (1971).

25. T.A. Joseph and K.P. Birman, ‘‘Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems,’’ *ACM Trans. Comp. Syst.* **4**(1) (Feb. 1987).

26. H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum, and J. Jansen, ‘‘Replication Techniques for Speeding up Parallel Applications on Distributed Systems,’’ Report IR-202, Vrije Universiteit, Amsterdam, The Netherlands (Oct. 1989).

27. J. Chang and N.F. Maxemchuk, ‘‘Reliable Broadcast Protocols,’’ *ACM Trans. Comp. Syst.* **2**(3), pp. 251-273 (Aug. 1984).

28. K.P. Birman and T.A. Joseph, ‘‘Reliable Communication in the Presence of Failures,’’ *ACM Trans. Comp. Syst.* **5**(1), pp. 47-76 (Feb. 1987).

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

30. M.F. Kaashoek and A.S. Tanenbaum, ‘‘Group Communication in the Amoeba Distributed Operating System,’’ *11th Int’l Conf. on Distributed Computing Systems*, Arlington, Texas, pp. 222-230 (20-24 May 1991).

31. R.M. Metcalfe and D.R. Boggs, ‘‘Ethernet: Distributed Packet Switching for Local Computer Networks,’’ *Commun. ACM* **19**(7), pp. 395-404 (July 1976).

32. M.F. Kaashoek, A.S. Tanenbaum, S. Flynn Hummel, and H.E. Bal, ‘‘An Efficient Reliable Broadcast Protocol,’’ *ACM Operating Systems Review* **23**(4), pp. 5-20 (Oct. 1989).

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

34. J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield, ‘‘The Amber System: Parallel Programming on a Network of Multiprocessors,’’ *Proc. of the 12th ACM Symp. on Operating System Principles*, Litchfield Park, AZ, pp. 147-158 (Dec. 1989).

35. P. Vishnubhotia, ‘‘Synchronization and Scheduling in ALPS Objects,’’ *Proc. 8th Int. Conf. on Distributed Computing Systems*, San Jose, CA, pp. 256-264 (June 1988).

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

37. R.G. Minnich and D.J. Farber, ‘‘Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory,’’ *Proc. 10th Int. Conf. on Distributed Computing Systems*, Paris, pp. 468-475 (May 1990).

38. P.W. Hutto and M. Ahamad, ‘‘Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories,’’ *Proceedings 10th International Conference on Distributed Computing Systems*, Paris, pp. 302-309 (May 1990).

39. J.K. Bennet, J.B. Carter, and W. Zwaenepoel, ‘‘Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,’’ *Proceedings 2nd Symposium on Principles and Practice of Parallel Programming*, Seattle, WA (March 1990).

40. W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum, ‘‘A Comparison of Two Paradigms for Distributed Shared Memory,’’ IR-221, Vrije Universiteit, Amsterdam, The Netherlands (August 1990).