EXPERIENCE WITH DISTRIBUTED PROGRAMMING IN ORCA

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

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

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems−*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs−*concurrent programming structures*;

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Orca, all-pairs shortest paths problem, successive overrelaxation, traveling salesman problem

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

ABSTRACT

Orca is a language for programming parallel applications on distributed computing systems. Although processors in such systems communicate only through message passing and not through shared memory, Orca provides a communication model based on logically shared data. Programmers can define abstract data types and create instances (objects) of these types, which may be shared among processes. All operations on shared objects are executed atomically.

Orca's shared objects are implemented by replicating them in the local memories of the processors. Read operations use the local copies of the object, without doing any interprocess communication. Write operations update all copies using an efficient reliable broadcast protocol.

In this paper, we briefly describe the language and its implementation and then report on our experiences in using Orca for three parallel applications: the Traveling Salesman Problem, the All-pairs Shortest Paths problem, and Successive Overrelaxation. These applications have different needs for shared data: TSP greatly benefits from the support for shared data; ASP benefits from the use of broadcast communication, even though it is hidden in the implementation; SOR merely requires point-to-point communication, but still can be implemented in the language by simulating message passing.

We discuss how these applications are programmed in Orca and we give the most interesting portions of the Orca code. Also, we include performance measurements for these programs on a distributed system consisting of 10 MC68020s connected by an Ethernet. These measurements show that significant speedups are obtained for all three programs.

1. INTRODUCTION

Orca is a new programming language intended for implementing parallel applications on loosely-coupled distributed systems. Unlike the majority of other languages in this category [1], Orca supports a communication model based on shared data. We have taken this decision because we feel that, for many applications, it simplifies programming. Since distributed systems lack shared memory, however, this sharing of data is logical rather than physical.

Two important issues arise out of this decision: how to *implement* logically shared data efficiently and how to *use* it. The first issue has been addressed in several other publications [2, 3, 4]. The key ideas in an efficient distributed implementation of Orca are *replication* of shared data and the use of a highly efficient *broadcast protocol* for updating copies of shared data.

In this paper, we will focus on our experiences in using Orca. Thus far, Orca has been used for a number of small to medium size applications. Small applications include: matrix multiplication, prime number generation, sorting, the All-pairs Shortest Paths problem, the Traveling Salesman Problem, alpha-beta search, Fast Fourier Transformation, and Successive Overrelaxation. As a somewhat larger application, we have implemented a distributed chess problem solver. Some of these applications could just as easily have been written in a language providing only message passing and no shared data. Others, however, greatly benefit from the support of logically shared data. The chess problem solver, for example, uses several shared data structures, such as a transposition table and a killer table [5].

Below, we will look at three parallel applications with different needs for shared data. The first application, the Traveling Salesman Problem, is a good example demonstrating the usefulness of logically shared data. Although the algorithm we use can be (and has been) implemented with message passing, such an implementation is much more complicated and has an inferior performance.

The second application, the All-pairs Shortest Paths (ASP) problem, basically requires broadcast messages for efficiency. ASP could be implemented efficiently in a message-passing language that supports broadcasting. The Orca implementation of ASP is rather elegant, since the physical broadcasting is hidden in the implementation of the language; it is not visible to the programmer. In addition to this higher-level of abstraction, the performance is at least as good as with explicit one-to-many message passing.

The third and last application we discuss is Successive Overrelaxation (SOR). This application merely needs point-to-point message passing. We include this application to show how message-passing can be implemented in Orca.

The outline of the rest of the paper is as follows. In Section 2, we will present a short description of Orca, together with its underlying model. We will also briefly describe the current implementations of the language. In Sections 3 to 5 we will discuss the three applications mentioned above. For each application, we will give the most interesting portions of the Orca program. In addition, we will show the speedup of each program, to give an idea of the performance of parallel Orca programs. The implementation used for these measurements runs on an Ethernet-based distributed system. Finally, in Section 6, we will present some conclusions.

2. THE ORCA PROGRAMMING LANGUAGE

In this section we will give a concise introduction to Orca. A much more thorough description can be found in [4]. We will first describe Orca's underlying model, called the *shared data-object model*. Next, we will look at the most important aspects of the language design. We will conclude with a brief description of the language implementation.

2.1. The shared data-object model

The shared data-object model provides the programmer with logically shared data. The entities shared in our model are determined by the programmer. (This is in contrast with Kai Li's Shared Virtual

Memory [6], which only allows fixed-size pages to be shared.) In our model, shared data are encapsulated in *data-objects**, which are variables of user-defined abstract data types. An abstract data type has two parts:

- A specification of the operations that can be applied to objects of this type.
- The implementation, consisting of declarations for the local variables of the object and code implementing the operations.

Instances (objects) of an abstract data type can be created dynamically. Each object contains the variables defined in the implementation part. These objects can be shared among multiple processes, typically running on different machines. Each process can apply operations to the object, which are listed in the specification part of the abstract type. In this way, the object becomes a communication channel between the processes that share it.

The shared data-object model uses two important principles related to operations on objects:

1. All operations on a given object are executed *atomically* (i.e., *indivisibly*). To be precise, the model guarantees *serializability* [7] 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.
2. All operations apply to *single* objects, so an operation invocation can modify at most one object. Making *sequences* of operations on different objects indivisible is the responsibility of the programmer.

These two principles make the model easy to understand and efficient. Moreover, in our experience, they provide sufficient support for many medium to large grained parallel applications. Distributed applications like banking and airline reservation systems can profit from more support (e.g., atomic multi-object operations), but such applications are not our major concern here. Other languages address this application domain, for example Argus [8] and Aeolus [9]. Also, parallel applications on *closely-coupled* (shared-memory) systems can use a finer grain of parallelism (e.g., parallelism within objects), but again these are not the type of applications we are interested in here.

The shared data-object model is related to several other models for parallel or distributed programming, such as the distributed object-based model of Emerald [10] and the monitor model of Concurrent Pascal and Mesa [11, 12]. The primary distinguishing property of our model is the total lack of centralized control of shared data. Operations on monitors, for example, are usually implemented using semaphores and are executed one at a time. Our model, on the other hand, only guarantees serializability, but

* We will sometimes use the term “object” as a shorthand notation for data-objects. Note, however, that unlike in most parallel object-based systems, objects in our model are purely passive.

does not actually serialize all operations. As we will see, our model can be implemented efficiently by replicating objects and performing read operations on shared objects in parallel on local copies.

The model is also related to Linda's Tuple Space [13]. It differs from Tuple Space by allowing programmers to define operations of arbitrary complexity on shared data structures. Linda supports a fixed number of low-level primitives for manipulating single tuples, which we feel is a disadvantage [14].

Other differences between the two models are the way shared data are addressed and modified. Data in Tuple Space are addressed associatively and are modified by first taking them out of Tuple Space, then modifying them, and then putting them back. In our model, shared objects are addressed and modified directly, much like normal variables, so the implementation of our model is somewhat simpler and using it more closely resembles ordinary programming.

2.2. Linguistic support for the shared data-object model

We have designed a new programming language called *Orca*, which gives linguistic support for the shared data-object model. Orca is a simple, procedural, type-secure language. It supports abstract data types, processes, a variety of data structures, modules, and generics. Below, we will look at the expression of parallelism and synchronization in Orca and we will discuss Orca's data structuring mechanism, which was especially designed for distributed programming.

Parallelism in Orca

Parallelism in Orca is based on explicit creation of sequential processes. Processes are similar to procedures, except that procedure invocations are serial but newly created processes run in parallel with their creator. 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(processor-number) ] ;
```

This statement creates a single new process, which can optionally be assigned to a specific processor by specifying the processor's identifying number. Processors are numbered sequentially; the total number of processors available to the program can be obtained through the standard function `NCPUS`. If the **on** part is omitted, the new process will be run on the same processor as its parent.

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, which is passed by its parent (creator) in the **fork** statement. The data structures in the parent and child are thereafter independent of each other—changing one copy does not affect the other—so they cannot be used for communication between parent and child thereafter.

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. This mechanism can be used for sharing objects among any number of processes. The parent can spawn several child processes and pass objects to each of them. The children can pass the 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. Note that there are no “global” objects. The only way to share objects is by passing them as parameters.

Synchronization

Processes in a parallel program sometimes have to synchronize their actions. This is expressed in Orca by allowing operations to *block* until a specified predicate evaluates to *true*. A process that invokes a blocking operation is suspended for as long as the operation blocks.

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.

The implementation of an operation has the following general form:

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

The *conditions* (guards) are side-effect free Boolean expressions that depend only on the internal data of the object and the parameters of the operation. The *statements* may read or modify the object’s data.

If the operation is applied to a certain object, it blocks until at least one of the guards is true. If a guard initially fails, it can succeed at a later stage after another process has modified the internal data of the object. As soon as one or more guards succeed, one true guard is selected nondeterministically and its corresponding statements are executed.

The testing of the guards and the execution of the statements of the selected guard together are an indivisible action. As long as all guards fail, the operation has no effect at all, as the object’s data can only be modified by the statements. This means that serializability is still easy to achieve, so all operation invocations are executed as atomic actions.

Data structures

We will now turn our attention to the data structuring facilities of Orca. Data structures are important to the design of any language, whether sequential or distributed. Below, we will look at *distribution* of data structures and at *type-security*.

First, we want data structures to be treated similar to scalar variables. In particular, any data structure can be passed as a parameter to processes and operations. This is especially important if data structures are encapsulated within abstract data types, because we want to be able to pass any object as an input or shared parameter to a remote process, no matter what its internal data look like. In contrast, most other distributed languages only allow scalar data or arrays to be sent to a remote process or require the programmer to write code that converts data structures into a transmittable form [15].

Second, we want the data structuring mechanism to be type-secure. Erroneous usage of data structures should be detected either during compile-time or run-time, but should never wreak havoc and generate a core dump. This issue is critical, as it makes debugging of distributed programs much easier.

The basic idea behind the data structuring mechanism of Orca is to have a few built-in primitives that are secure and suitable for distribution. More complicated data structures can be defined using the standard types of the language (integer, real, boolean, char) and the built-in data structuring capabilities. Frequently, new data structures will be designed as abstract data types. To increase the usefulness of such types, Orca supports *generic* abstract data types.

Orca has the following type constructors built-in: arrays, records, unions, sets, bags, and graphs. Arrays in Orca are similar to flexible arrays in Algol 68 [16]. They have dynamic bounds that can be changed during run-time by assigning a new value to the array. The graph type-constructor can be used for building arbitrary complex data structures, such as lists, trees, and graphs. Most of the other type constructors are similar to their Modula-2 counterparts, except that each data type in Orca can have components of any type.

At run time, data structures are represented by *descriptors*. These descriptors are used for two different purposes. First, descriptors allow any data structure to be passed as a (value) parameter to a remote process. The run time system (RTS) is capable of transmitting variables of any type, even graphs.

In Orca, nodes can be added to, and deleted from, a graph through the following constructs:

```
n := addnode(G);              # add a new node to G
deletenode(G, n);             # delete node n from G
```

The first statement allocates a new node for graph G and stores its name in the variable n. The second statement deallocates this node. The RTS keeps track of which nodes constitute a given graph; this information is stored in the graph's descriptor. Whenever the graph is sent to a remote machine, the RTS uses this information for marshalling the nodes of the graph into transmittable form.

The second purpose of descriptors is to allow extensive run-time type checking to be done. The RTS, for example, checks all array and graph references. If a node is deleted from a graph and subsequently referenced, a run-time error will be given. This is in sharp contrast to languages such as C and Modula-2, in which references to deallocated memory may cause severe damage.

2.3. Implementation

An implementation of Orca consists of two parts: a compiler and a run time system. A prototype Orca compiler has been built using the *Amsterdam Compiler Kit* [17], which is a toolkit for building portable compilers. The compiler generates the EM intermediate code, extended with calls to library routines implemented by the run time systems. The library routines are used for the management of processes, data-objects, and complex data structures.

The compiler performs several important optimizations. Foremost, it analyses the implementation code of operations and determines which ones are read-only. As explained before, this allows the RTSs to apply read operations to local copies of objects.

At present, we have three prototype implementations of the run time system, on two different machines. One RTS runs on a shared-memory multiprocessor, consisting of 10 MC68020 CPUs and 8 MB shared memory, connected through a VME bus. Each CPU board contains 2 MB of local memory, used for storing programs and local data. This RTS was written mainly for comparison purposes, since Orca is intended for distributed systems.

The second RTS runs on a distributed system containing 10 nodes that are connected by a 10Mbit/sec Ethernet. Each node consists of a CPU board, identical to the ones used in the multiprocessor, and an Ethernet controller board using the Lance chip. This RTS runs on top of the Amoeba distributed operating system [18]. It uses Amoeba's Remote Procedure Call for interprocess communication.

The third RTS, which is used for this paper, also runs on the distributed system. Unlike the Amoeba RTS, it runs on top of the bare hardware. It uses the reliable broadcast protocol described in [19] for communication. It replicates all shared data-objects on all processors. Operations that only read (but do not modify) their object's data are applied to the local copy of the object. Write operations are applied to all copies, by broadcasting the operation and its parameters. Since our broadcast protocol is very fast, this scheme is efficient for many applications, as we will see. The implementation of this RTS is described in more detail in [3]. Below, we will study three parallel applications that have been implemented in Orca. Also, we will give measurements of the performances of these programs on the broadcast RTS.

3. THE TRAVELING SALESMAN PROBLEM

In the Traveling Salesman Problem (TSP), the goal is to find the shortest route for a salesman to visit (exactly once) each of the *n* cities in his territory. The problem can be solved using a parallel *branch-and-bound* algorithm. Abstractly, the branch-and-bound method uses a *tree* to structure the space of possible solutions. A *branching rule* tells how the tree is built. For the TSP, a node of the tree represents a partial tour. Each node has a branch for every city that is not on this partial tour. Figure 1 shows a tree for a 4-city problem. Note that a leaf represents a full tour (a solution). For example, the leftmost branch represents the tour New York - Chicago - St. Louis - Miami.

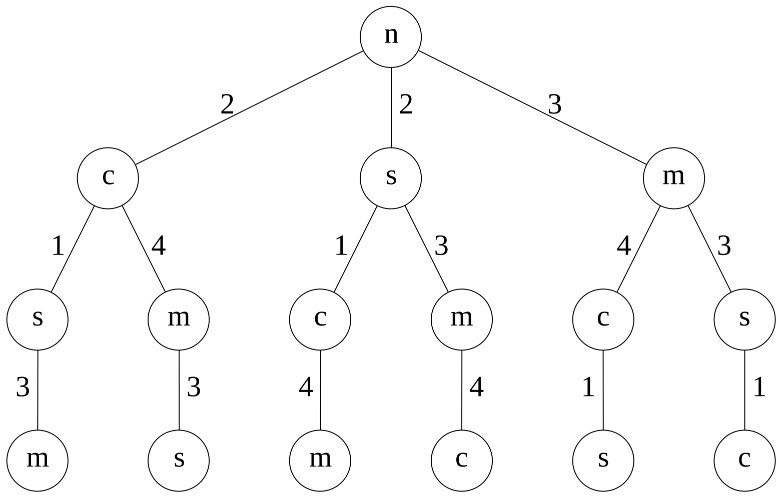

Fig. 1. Search tree for 4-city TSP (New York, Chicago, St. Louis, Miami).

A *bounding rule* avoids searching the whole tree. For TSP, the bounding rule is simple. If the length of a partial tour exceeds the length of any already known solution, the partial tour will never lead to a solution better than what is already known. In Figure 1 for example, the leftmost full route has length 6, so the partial route starting with New York - Miami - Chicago (of length 7) cannot lead to an optimal solution.

3.1. Parallel TSP in Orca

Parallelism in a branch-and-bound algorithm is obtained by searching parts of the tree in parallel. A *manager process* traverses the top part of the tree, up to a certain depth D. For each node at depth D, the manager generates a job to be executed by *worker* processes. A job consists of the evaluation of the sub-tree below a given node. Effectively, the search tree is distributed among several processes. The manager process searches the top D levels; one or more worker processes traverse the nodes at the lower N – D levels.

To implement the bounding rule, the workers need to keep track of the shortest full route found so far. On a distributed system, one might be tempted to let each worker keep a local minimum (i.e., the length of the shortest path found by the worker itself). After the entire tree has been searched, the lowest value of these local minima can be determined. This scheme is easy to implement and has little communication overhead. Unfortunately, it also makes the bounding rule far less effective than in a sequential program. In the example of Figure 1, only the worker that has evaluated the leftmost route will know there exists a path of length 6. Other workers will be ignorant of this fact, and may start working on partial routes longer than 6, which cannot lead to an optimal path.

In our implementation, the length of the best route so far is kept in a data-object `minimum`, which is shared among all the workers (see Figure 2). This object is of type `IntObject`, which is a predefined object type encapsulating a single integer variable. Each process can directly read the value through an operation invocation. The Orca RTS will automatically replicate the object on all processors, so reading the object does not involve any interprocess communication. Whenever the object changes (i.e., whenever a new better full route is found), all copies are updated immediately.

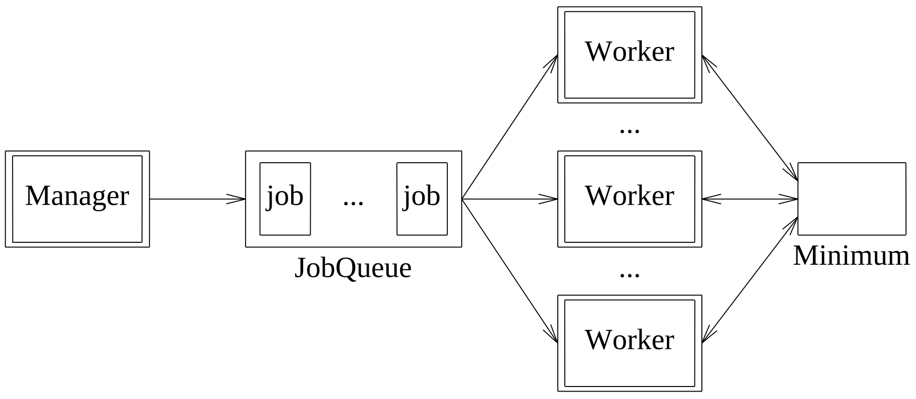

Fig. 2. Structure of the Orca implementation of TSP. The Manager and Workers are processes. The JobQueue is a data-object shared among all these processes. Minimum is a data-object of type IntObject; it is read and written by all workers.

The manager and worker processes communicate through a shared `JobQueue` data-object. The manager adds jobs to this queue, which are subsequently retrieved and executed by the workers. The JobQueue used by the TSP program is an instantiation of a *generic* object type (see Figure 3).

```
generic (type T)
object specification GenericJobQueue;
    operation AddJob(job: T);    # add a job to the tail of the queue
    operation NoMoreJobs();      # invoked when no new 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 of generic object type GenericJobQueue.

Three operations are defined for JobQueue objects. `AddJob` adds a new job to the tail of the queue. In the TSP program, this operation will only be invoked by the manager process. The operation

NoMoreJobs is to be called when all jobs have been added to the queue. 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. If the queue is empty and NoMoreJobs has been invoked, the operation returns “false”. In this way, it is easy to determine when the workers can be terminated. If none of these two conditions—queue not empty or NoMoreJobs invoked—holds, the operation blocks until one of them becomes true.

The type definitions and constants used by the TSP program are shown in Figure 4. The distances between the cities are stored in a two-dimensional array of type DistTab. The entries in a given row C of this array are *presorted* by the distances to city C. Entry [C, i] of the array contains the *i*th closest city to C and the distance from C to that city. With this representation, it is easy to implement the nearest-city-first heuristic.

```
# distance table, sorted by nearest-city-first heuristic
type pair =
    record
        ToCity: integer;  # to which city
        dist: integer;    # distance to that city
    end;
type DistArray = array[integer] of pair;
type DistTab = array[integer] of DistArray;

# job type:
type PathType = array[integer] of integer;
type JobType =
    record
        len: integer;   # length of partial route
        path: PathType; # the partial route itself
    end;

const NrTowns = 12;  # number of towns
const MaxHops = 3;   # search depth of manager
```

Fig. 4. Declarations for types and constants used by the TSP program.

Figure 4 also declares a type JobType, which defines the jobs to be executed by the workers. A job consists of an initial (partial) route for the salesman and the length of this partial route. The latter number is included for efficiency reasons only; it could also be derived from the path itself.

An outline of the Orca code for the manager process is shown in Figure 5. The manager creates and initializes the shared object minimum, initializes the distance table, and forks one worker process on each processor except its own. Subsequently, the manager generates the jobs and stores them in a shared object of type TspQueue, which is an instantiation of the generic type GenericJobQueue of Figure 3. When all jobs have been generated, the manager forks a worker process on its own processor and waits until all work has been done. In this way, job generation overlaps 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. The manager uses a shared counter (`WorkersActive`) to keep track of the number of active worker processes. When the counter drops to zero, all workers have terminated. The manager then prints the final value of `minimum` (i.e., the length of the optimal solution).

```
process manager(distance: DistTab);
    minimum: IntObject;     # length of current best path (shared object)
    q: TspQueue;            # the job queue (shared object)
    i: integer;
    WorkersActive: IntObject;   # number of active workers (shared object)
begin
    minimum$assign(MAX(integer));      # initialize minimum to infinity
    WorkersActive$assign(NCPUS());     # initialize number of workers
    for i in 1.. NCPUS() - 1  do
        # fork one worker per processor, except current processor
        fork worker(minimum, q, distance, WorkersActive) on(i);
    od;
    Generate the jobs for the workers and  store them in q.
    Each job is an initial path of MaxHops hops.
    q$NoMoreJobs(); # all jobs have been generated now
    fork worker(minimum, q, distance, WorkersActive) on(0);
       # jobs have been generated; fork a worker on this cpu too
    WorkersActive$AwaitValue(0);   # wait until workers have finished
    WriteLine("minimum = ", minimum$value());  # length of shortest path
end;
```

Fig. 5. Outline of Orca code for the manager process of the TSP program

In the implementation of Figure 5, a job contains an initial path with a fixed number of cities (`MaxHops`). Basically, the manager process traverses the top `MaxHops − 1` levels of the search tree, while the workers traverse the remaining levels. The manager process generates the jobs in "nearest-city-first" order. It is important that the worker processes execute the jobs in the same order they were generated. This is the reason why we use an ordered queue rather than an unordered bag for storing the jobs.

The code for the worker processes is shown in Figure 6. 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. A worker process obtains the length of the current best route by applying the `value` operation to the shared object `minimum`. This operation returns the current value of the integer stored in the object. Since the operation does not modify the object's data, it uses a local copy of the object, without doing any interprocess communication.

If the route passed as parameter is a full route (visiting all cities), a new best route has been found,

```
process worker(minimum: shared IntObject; q: shared TspQueue;
              distance: DistTable; WorkersActive: shared IntObject);
    job: JobType;
begin
    while q$GetJob(job) do   # while there are jobs to do:
        tsp(MaxHops, job.len, job.path, minimum, distance);  # search subtree
    od;
    WorkersActive$dec();    # this worker becomes inactive
end;

function tsp(hops: integer; len: integer; path: shared PathType;
            minimum: shared IntObject; distance: DistTable);
    city, dist, me, i: integer;
begin
    # Search a TSP subtree that starts with initial route "path".
    if len >= minimum$value() then return; fi;    # cut-off
    if hops = NrTowns then
        # We found a full route better than current best route.
        minimum$min(len); # Update minimum, using indivisible "min" operation.
    else
        # "path" really is a partial route.  Call tsp recursively
        # for each subtree, in "nearest-city-first" order.
        me := path[hops];  # me := last city on path
        for i in 1.. NrTowns do
            city := distance[me][i].ToCity;
            if not present(city, hops, path) then    # is city on path?
                path[hops+1] := city;                # no, append city to path
                dist := distance[me][i].dist;        # distance me->city
                tsp(hops+1, len+dist, path, minimum, distance);
            fi;
        od;
    fi;
end;
```

Fig. 6. Orca code for the worker processes of the TSP program

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 value is really changed, the run time system automatically updates all copies of the object.

3.2. Performance of parallel TSP

We have determined the performance of the TSP program on the broadcast RTS by measuring its execution time for solving three randomly generated input graphs with 12 cities each. The manager process searches 2 levels of the tree, so it generates $11 \times 10 = 110$ jobs, each of which solves a 10 city TSP problem. Figure 7 shows the average speedups obtained for these three problems. (The speedup on P processors is defined as the ratio of the execution times on 1 and P processors. Thus, by definition, the speedup with 1 processor is always 1.0.) With 10 processors, the Orca program is 9.96 times faster as with one processor.

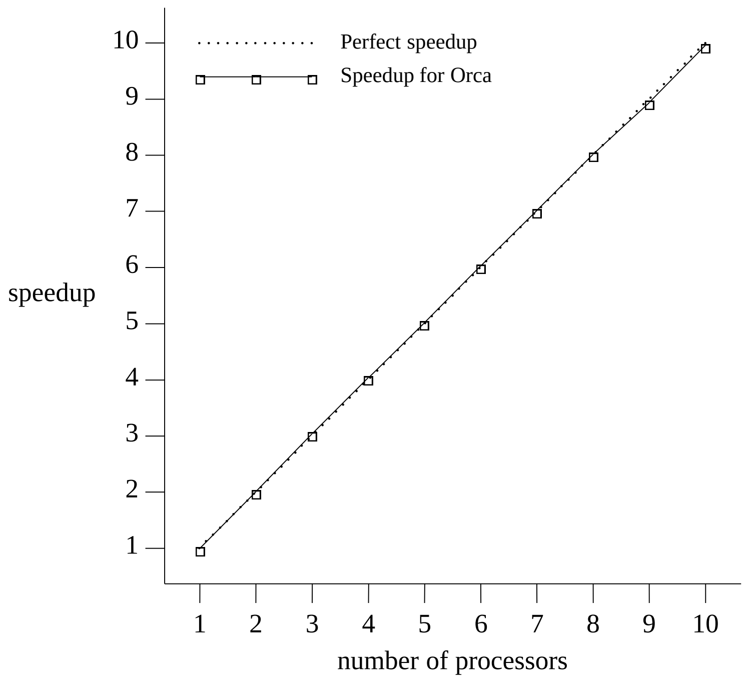

Fig. 7. Measured speedup for the Traveling Salesman Problem, averaged over three randomly generated graphs with 12 cities each.

It is interesting to compare the performance of the Orca TSP program with an earlier implementation of TSP in C, on top of Amoeba [20]. The C/Amoeba version uses Remote Procedure Call for interprocess communication. It is similar to the Orca version, except that replication of the shared variable `minimum` is dealt with by the programmer rather than the RTS. If a worker in the C/Amoeba version finds a better full route, it does not update the shared variable immediately, because the RPC model makes it difficult for servers to return intermediate results. Instead, the worker completes the analysis of the subtree assigned to it and then returns the best path. Moreover, the other worker processes do not update their local copies of the shared variable until they have finished their current jobs.

As a result, worker processes in the C/Amoeba version often will have an out-of-date value of the shared variable. Therefore, they will frequently search more nodes than is necessary. This search overhead may become quite significant. For one of the input graphs we obtained the following statistics:

Total number of nodes in the tree:	108,505,112
Nodes searched by sequential program:	1,272,076
Nodes searched by 10-CPU C/Amoeba program:	1,763,552
Nodes searched by 10-CPU Orca program:	1,149,270

In this example, the C/Amoeba version of TSP searches 39% more nodes than a sequential algorithm. The Orca version running on the broadcast RTS, on the other hand, searches 10% *fewer* nodes. It achieves superlinear speedup in this case (a speedup of 10.75 with 10 processors). Such anomalous behavior of parallel branch-and-bound programs has also been reported by other authors [21]. It is due to the different search order of the parallel algorithm. One processor quickly finds a near-optimal solution, which other processors use for pruning parts of their trees. However, if we were to make a large number of runs with different input data, we would not see a superlinear effect on the average, of course.

4. THE ALL-PAIRS SHORTEST PATHS PROBLEM

In the All-pairs Shortest Paths (ASP) 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. Sequential solutions to the ASP problem are given in several text books on algorithms and data structures [22, 23]. We will first review the standard sequential algorithm, due to Floyd, for solving the ASP problem and then discuss how it can be parallelized. The algorithm assumes that the nodes of the graph are numbered sequentially from 1 to `N` (the total number of nodes) and that each edge in the graph is assigned a positive length (or weight).

The standard sequential solution to the ASP problem uses an iterative algorithm. During iteration `k` it finds the shortest path from every node `i` in the graph to every node `j` that only visits intermediate nodes in the set {1..`k`}. During iteration `k`, the algorithm checks if the current best path from `i` to `k` concatenated with the current best path from `k` to `j` is shorter than the best path from `i` to `j` found so far (i.e., during the first k-1 iterations).

Before the first iteration, such a path only exists if the graph contains a direct edge from node `i` to node `j`. After the last iteration, the resulting path may visit any other node, as the set {1..`k`} includes all nodes if k = N. Therefore, after the last iteration, the resulting path is the shortest path from node `i` to node `j`.

The standard algorithm uses a sequence of matrices for storing the lengths of all these paths. After iteration `k`, element $C^k[i,j]$ contains the length of the shortest path from `i` to `j` found so far (i.e., the best path visiting only nodes between 1 and `k`). During iteration `k`, the matrix C^{k-1} is transformed into a matrix C^k as follows:

$$C^k[i,j] = \text{MINIMUM}(\ C^{k-1}[i,j],\ \ C^{k-1}[i,k] + C^{k-1}[k,j]) \quad (1 \le i,j \le N)$$

Note that the value of row `k` of matrix C^k is equal to row `k` of matrix C^{k-1}, because

$$\begin{aligned}
&C^k[k,j] = \\
&\text{MINIMUM}(\ C^{k-1}[k,j],\ \ C^{k-1}[k,k] + C^{k-1}[k,j]) = \\
&\text{MINIMUM}(\ C^{k-1}[k,j],\ \ C^{k-1}[k,j]) = \\
&C^{k-1}[k,j]
\end{aligned}$$

4.1. Parallel ASP in Orca

This sequential algorithm can be transformed into a parallel algorithm by computing the rows of the matrices $C^k[i,j]$ in parallel. There are two ways for structuring such a parallel ASP algorithm. First, we can have a single master process that executes `N` iterations. During iteration `k`, the master forks one or more slave processes and passes part of the matrix C^{k-1} to each slave. A slave computes one or more rows of C^k, sends these values back to the master, and then terminates.

An alternative way for structuring the parallel algorithm is to let each slave execute an iterative algorithm. In this case, the master process forks a number of slave processes and then terminates. Each slave process performs `N` iterations and then outputs its results. The latter approach is more efficient, because it requires fewer **fork** statements. Also, it has more parallelism, because different slave processes may be working on different iterations.

We therefore use the second approach and structure the algorithm as a number of iterative slave processes. Each slave process computes a fixed number of rows of the C matrices. The different rows of C^k cannot be computed completely independently from each other, however. Suppose a slave process wants to compute row `i` of matrix C^k. That is, it has to compute the values

$C^k[i,j]$, for all j between 1 and N.

To compute the value $C^k[i,j]$, the values of $C^{k-1}[i,j]$, $C^{k-1}[i,k]$, and $C^{k-1}[k,j]$ are needed. The first two values have been computed by this process during the previous iteration. The value of $C^{k-1}[k,j]$, however, has been computed (during iteration `k-1`) by the process that takes care of row `k`. The above argument applies to any value of `i` and `j` between 1 and `N`. Therefore, during iteration `k`, each process needs to know the value of the entire row `k` of matrix C^{k-1}. So, after each iteration `k`, the process that computed the value of row `k+1` of matrix C^k has to send this value to all other processes. Furthermore, a process should not continue computing its part of C^k until it has received row `k` of the previous iteration. Clearly, the processes must be *synchronized* to achieve this.

In conclusion, the ASP problem can be solved in parallel by letting each slave process taking care of some of the rows of the C matrices. Each process performs an iterative algorithm. During iteration `k`, a processor that is assigned rows `lb` up to `ub` computes the values

$C^k[i,j]$, for all `i` between `lb` and `ub`, and all `j` between `1` and `N`.

A processor should not start working on iteration `k` until the value of row `k` of matrix C^{k-1} is available. Apart from this restriction, the processors do not depend on each other. In particular, they need not all be working on the same iteration.

The synchronization constraint described above is implemented through an object of type `RowCollection`, whose specification is shown in Figure 8. This object type defines two operations, `AddRow` and `AwaitRow`. Whenever a process has computed row `k` of matrix C^{k-1}, it executes the operation `AddRow`, passing the iteration number (`k`) and the value of the row (an array of integers) as parameters. Before a process starts working on iteration `k`, it first waits until the value of row `k` for this iteration is available. It does so by invoking `AwaitRow(k)`.

```
module specification AspTypes;
    const N = 200;  # number of nodes in the graph
    type RowType = array[integer] of integer;
end;

object specification RowCollection;
    from AspTypes import RowType;

    # Object used to exchange row k before each iteration
    operation AddRow(iter: integer; R: RowType);
       # Add the row for the given iteration number
    operation AwaitRow(iter: integer): RowType;
       # Wait until the row for the given iteration is available,
       # then return it.
end;
```

Fig. 8. Specification of module AspTypes and object type RowCollection

The implementation of object type `RowCollection` is shown in Figure 9. The internal data of an object of this type consist of an array of rows (i.e., the variable `tab`). There is one row for each iteration of the algorithm. As different processes may be working on different iterations, all these rows are retained during the entire execution of the program.

The variable `tab` is of type `collection`, which is an array of `N` elements of type `RowType`. The declaration for `RowType` (see Figure 8) does not specify actual bounds for the rows. This means that the rows of `tab` will initially be empty. Since Orca allows entire arrays to be reassigned, however, the rows can be initialized with a single assignment statement. The `AddRow` operation assigns a non-empty array to a certain entry of the table. The `AwaitRow` operation blocks until the specified entry of the table is non-empty. (An empty array has a lower bound that is higher than its upper bound.)

An outline of the code for the ASP program itself is shown in Figure 10. It is structured using one master process and a number of slave processes. The master process forks one slave process per processor. Each slave is assigned part of the initial C matrix. A slave initializes its part of the C matrix. Each slave takes a shared object of type `RowCollection` as a parameter. This object, called `RowkColl`, is used for synchronizing the slaves, as described above.

Conceptually, the slave processes compute a sequence of matrices C^0, C^1, ... , C^N. In the implementation, however, each slave simply uses a single array variable, which gets modified in place. There is basically only a single C matrix, which is partitioned among all the slaves.

Each slave process executes `N` iterations. At the beginning of iteration `k`, it first obtains the value of row `k`. This value may either have been computed by itself (if $\mathtt{lb} \le \mathtt{k} \le \mathtt{ub}$) or by some other slave. In the first case, it sends the value to all other slaves, using the `AddRow` operation on `RowkColl`. In the second case, it obtains the value by invoking the `AwaitRow` operation, which blocks until the value is

```
object implementation RowCollection;
    from AspTypes import N, RowType;

    # The local data of objects of type RowCollection consist
    # of an array of rows, one row per iteration. Initially,
    # each row is an empty array.

    type collection = array[integer 1..N] of RowType;
    tab: collection;    # the local data of RowCollection objects

    operation AddRow(iter: integer; R: RowType);
    begin
        tab[iter] := R; # fill in the row for given iteration
    end;


    operation AwaitRow(iter: integer): RowType;
    begin
        # wait until row "iter" is defined, i.e. tab[iter] is non-empty.
        guard lb(tab[iter]) < ub(tab[iter]) do
            return tab[iter];    # return the requested row
        od;
    end;
end;
```

Fig. 9. Orca code for the implementation of object type RowCollection.

available. In either case, the slave process proceeds by updating the values of its rows. As soon as a slave process has finished all N iterations, it is ready to print its results.

4.2. Performance of parallel ASP

The performance of the ASP program for a graph with 200 nodes is shown in Figure 11. The speedup is only slightly less than perfect. The main reason why the speedup for ASP is not perfect is the communication overhead. Before each iteration, the current value of row k must be transmitted across the Ethernet from one processor to all the others.

Despite this overhead, the Orca program obtains a good speedup. With 10 processors, the speedup is 9.17. One of the most important reasons for this good performance is the use of broadcast messages for transferring the row to all processors. Each time the operation AddRow is applied to RowkColl, this operation and its parameters are broadcast to all processors. In our broadcast protocol [19], this causes one point-to-point packet and one multicast packet to be sent over the Ethernet.

The performance of the program also compares favorably with the parallel ASP program described in [24]. Jenq and Sahni have implemented ASP on an NCUBE/7 hypercube, using a work distribution similar to ours. For a graph with 112 nodes they report speedups of approximately 6.5 with 8 processors and 10.5 with 16 processors. As the NCUBE/7 does not support broadcast in hardware, they have used a binary tree transmission scheme for simulating broadcast. The elapsed time for a simulated

```
module implementation asp;
    import RowCollection; from AspTypes import N, RowType;
    type DistTable = array[integer] of RowType;  # table with distances

    process manager();
        RowkColl: RowCollection; # shared object for sending row k
        i, lb, ub: integer;
    begin
        for i in 0 .. NCPUS()-1 do
            determine interval [lb, ub] for next slave
            fork slave(RowkColl, lb, ub) on(i);
        od;
    end;

    process slave(RowkColl : shared RowCollection; lb, ub: integer);
        C: DistTable;   # table with distances between nodes
        i,j,k: integer;
        RowK: RowType;
    begin
        initialize rows lb to ub of C matrix
        for k in 1 .. N do  # do N iterations
            if (k >= lb) and (k <= ub) then
                RowK := C[k]; # I have row k
                RowkColl$AddRow(k, RowK); # add it to shared object RowkColl.
            else # Someone else is computing row k; wait for it.
                RowK := RowkColl$AwaitRow(k);
            fi;
            for all elements C[i,j] of rows C[lb] .. C[ub] do:
                C[i,j] := MINIMUM(C[i,j], C[i,k] + RowK[j])
        od;
        print final results
    end;
end;
```

Fig. 10. Outline of Orca implementation of ASP.

broadcast message is proportional to the logarithm of the number of processors. In contrast, the protocol used in our broadcast RTS typically only uses two messages per reliable broadcast. The time needed for broadcasting a message is almost independent of the number of receivers [19].

The bends in the graph of Figure 11 are caused by the slightly unbalanced work distribution. With 8 CPUs, for example, each processor manages exactly 25 rows of the distance matrix; with 9 CPUs, however, some processors will have 22 rows, but others will have 23 rows. As the elapsed computation time is determined by the processor that terminates last, the speedup with 9 processors cannot exceed 200/23 ≈ 8.7.

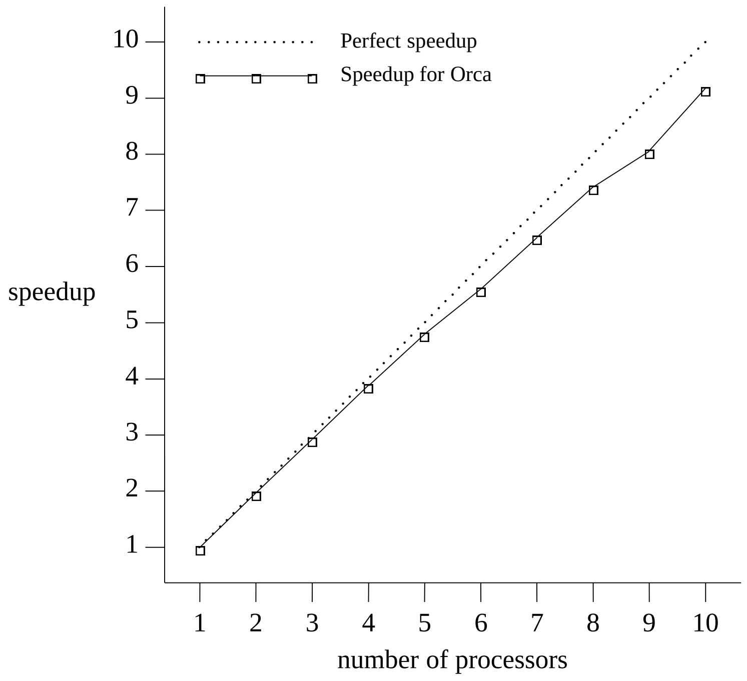

Fig. 11. Measured speedup for the All-pairs Shortest Paths problem, using an input graph with 200 nodes.

5. SUCCESSIVE OVERRELAXATION

Successive overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid [25]. The sequential SOR algorithm works as follows. 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. Next, it determines the new value of the point through the following correction:

$$Gnew[r,c] = G[r,c] + \omega \times (av - G[r,c])$$

where *av* is the average value of the four neighbors and ω is the so-called *relaxation parameter* [25]. The entire process terminates if, during the current iteration, no single point has been changed by more than a certain quantity.

Parallel implementations of SOR have been described in several research papers [26, 27]. The SOR program described below is based on the parallel Red/Black SOR algorithm used for the Amber system [27]. This algorithm treats the grid as a checkerboard and alternately updates all black points and all red points. As each point only has neighbors of the opposite color, each update phase can easily be parallelized. The grid can be partitioned among the available processors, which can all update different points of the same color in parallel.

5.1. Parallel SOR in Orca

As explained in the Amber paper, the distribution of the grid among the processors is of vital importance to the performance of parallel SOR. We have used a similar distribution scheme as in the Amber implementation. The grid is partitioned into regions, each containing several rows of the grid. Each region is assigned to a separate processor. Alternative distribution schemes of the grid would be less efficient. Putting the entire grid in a single shared object would create a severe bottleneck, since the grid is read and written very frequently. The other extreme, putting each point of the grid in a separate shared object, is also inefficient, since it would introduce a very high communication overhead.

With the above distribution scheme, all processors repeatedly compute new values for the points in their region, based on the current value of the point and its four neighbors. For a point on the upper or lower boundary of a region, however, one of the neighbors is stored on a remote processor. The processors, therefore, have to exchange the values of their boundary points before each iteration. This is illustrated in Figure 12.

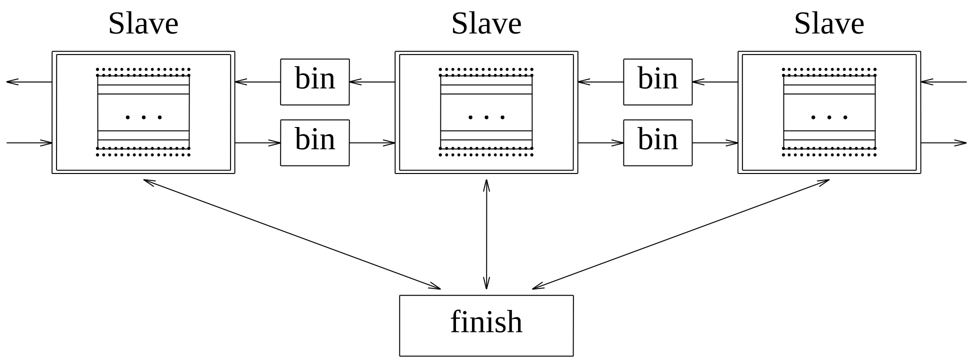

Fig. 12. Structure of the Orca implementation of Successive Overrelaxation. Each slave process maintains some portion of the rows of the grid. The dotted boxes are copies of the last row of the left neighbor and the first row of the right neighbor. These rows are transferred through shared "bin" objects. The processes also communicate through a shared barrier object "finish," to decide when the convergence criterion has been reached.

In Orca, the SOR program uses one *slave* process per processor. The slaves execute a number of iterations. Each iteration consists of two phases, one for the black points and one for the red points of the grid. Before each phase, a slave sends its first row to its left neighbor and its last row to its right neighbor. Next, it waits for the last row of its left neighbor and the first row of its right neighbor. Subsequently, it updates those points in its regions that have the right color.

The exchange of the rows is implemented through shared objects of type `RowBin`, which is an instantiation of the generic type `GenericBin` (see Figure 13). Basically, a bin object is a message buffer capable of holding a single message, in this case a row of the grid. The `put` operation on a bin blocks while the bin is full; the `get` operation blocks while the bin is empty.

```
generic (type T) object specification GenericBin;
    operation put(e: T);     # put item in the bin; block while bin is full
    operation get(e: out T); # fetch item from bin; block while bin is empty
end generic;

generic object implementation GenericBin;
    bin: T;         # the buffer containing the item
    empty: boolean; # indicates whether there's an item in the buffer now

    operation put(e: T);
    begin
        guard empty do  # wait until bin is empty
            bin := e;       # put item in bin
            empty := false; # bin is no longer empty
        od;
    end;

    operation get(e: out T);
    begin
        guard not empty do  # wait until there's an item in the bin
            e := bin;       # fetch the item
            empty := true;  # bin is now empty
        od;
    end;
begin
    empty := true;  # bin is initially empty
end generic;
```

Fig. 13. Specification and implementation of generic object type GenericBin

As mentioned above, the SOR program should continue updating points until each point has reached a stable value, approximately the average value of its four neighbors. Each slave process therefore keeps track of the maximum change of all points in its region. If, at the end of an iteration, the slaves agree that no point has changed by more than a certain value, the program terminates. This is implemented through an object `finish`, which basically implements barrier synchronization [28]. After each iteration, each slave determines if it wants to continue the SOR process or not. If all slaves want to stop, the entire SOR program terminates; otherwise, the slaves continue with the next iteration.

5.2. Performance

The SOR program described above is a difficult one for our broadcast RTS, since SOR mainly uses point-to-point communication. Apart from the termination protocol, each processor only communicates with its left and right neighbor.

The broadcast RTS replicates all shared objects on all processors. If, for example, one processor wants to send a row to its left neighbor, all processors will receive the `put` operation and apply it to their local copies of the bin. Despite this inefficiency, the broadcast RTS still achieves a remarkably good

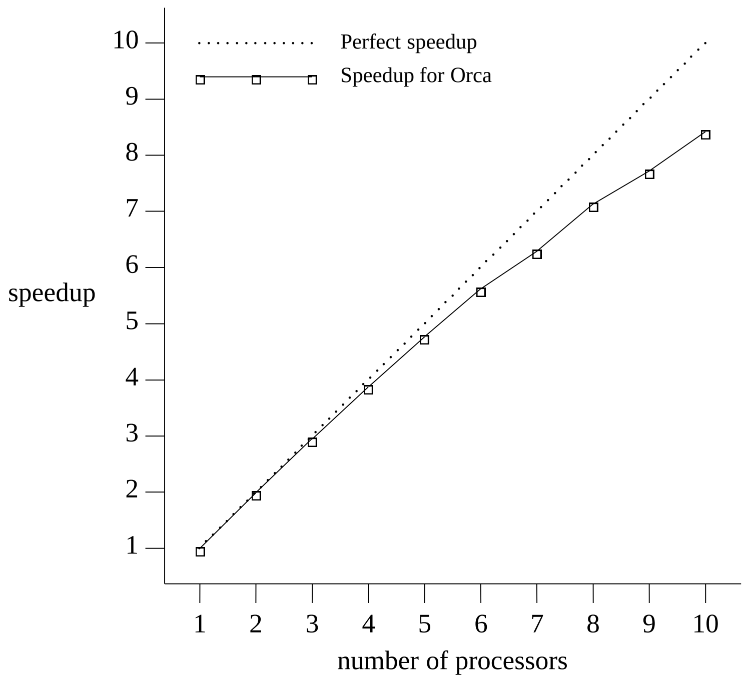

Fig. 14. Measured speedups for Successive Overrelaxation, using a grid with 80 columns and 242 rows.

performance (see Figure 14), due to the highly efficient broadcast protocol being used. (To be fair, the hardware we use does not have hardware floating point processors; with hardware floating point, the relative communication overhead would be higher, so the speedup would be lower.) The speedup on 10 processors is approximately 8.5. The speedup is comparable to that of the Amber implementation, which runs on a collection of Firefly multiprocessor workstations [27].

6. CONCLUSIONS

We have discussed a new programming language, Orca, designed for implementing parallel applications on distributed computing systems. Orca's most distinctive feature is its support for logically shared data. Processes on different machines can share data that are encapsulated in instances (objects) of abstract data types. The implementation (i.e., the compiler and run time system) takes care of the physical distribution of the data among the local memories of the processors. The Orca implementation used for this paper replicates all shared objects. Read operations on shared objects are applied to local copies. After a write operation, all copies are updated, using a reliable broadcast protocol.

We have discussed the usefulness of logically shared data in the context of three applications. The first application, the Traveling Salesman Problem, is a good example of when our approach is most effective. The TSP program uses a shared object with a high read/write ratio. As this object is replicated and the copies are updated immediately after each change, the application is very efficient.

The second program, which solves the All-pairs Shortest Paths problem, uses a shared object for transmitting rows of the distance matrix from one processor to all the others. Since updating objects is implemented through broadcasting, this again is highly efficient. An interesting aspect of this program is the fact that the programmer need not be aware of the broadcasting, as it is handled entirely by the run time system.

The third application, Successive Overrelaxation, does not need shared data or efficient broadcasting. Instead, it uses simple point-to-point message passing. This application is easy to implement in Orca, but the broadcast RTS is somewhat inefficient here. For this application, a selective (heuristic) replication strategy would be better. (This has been demonstrated by another RTS for Orca, described in [4]). Still, even under these unfavorable circumstances the broadcast RTS achieves a speedup of 85% of the theoretical maximum.

The reliable broadcast protocol we use scales well to a large number of processors. Therefore, we expect the RTS to scale well too, especially when heuristic replication is used.

In conclusion, our approach to distributed programming looks promising, at least for the applications discussed in this paper. In the near future, we intend to study more applications and compare our work with that of others. It will be interesting to compare our model with message-passing models as well as with other models supporting logically shared data.

ACKNOWLEDGEMENTS

We would like to thank Wim van Leersum for implementing the Orca compiler and Erik Baalbergen and Susan Flynn Hummel for giving useful comments on the paper.

REFERENCES

H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, “Programming Languages for Distributed Computing Systems,” *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322, Sept. 1989.

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

H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, “A Distributed Implementation of the Shared Data-object Model,” *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL., Oct. 1989.

H.E. Bal, “The Shared Data-Object Model as a Paradigm for Programming Distributed Systems,” Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands, Oct. 1989.

R-J. Elias, “Oracol, A Chess Problem Solver in Orca,” Master thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 1989.

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

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*, Vol. 19, No. 11, pp. 624-633, Nov. 1976.

B. Liskov, “Distributed Programming in Argus,” *Commun. ACM*, Vol. 31, No. 3, pp. 300-312, March 1988.

C.T. Wilkes and R.J. LeBlanc, “Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System,” *Proc. IEEE CS 1986 Int. Conf. on Computer Languages*, pp. 107-122, Miami, FL, Oct. 1986.

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

P. Brinch Hansen, “The Programming Language Concurrent Pascal,” *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 199-207, June 1975.

C.M. Geschke, J. H. Morris Jr., and E.H. Satterthwaite, “Early Experience with Mesa,” *Commun. ACM*, Vol. 20, No. 8, pp. 540-553, Aug. 1977.

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

M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum, “Experience with the Distributed Data Structure Paradigm in Linda,” *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 175-191, Ft. Lauderdale, FL., Oct. 1989.

M. Herlihy and B. Liskov, “A Value Transmission Method for Abstract Data Types,” *ACM Trans. Program. Lang. Syst.*, Vol. 4, No. 4, pp. 527-551, Oct. 1982.

A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker, “Revised Report on the Algorithmic Language Algol 68,” *Acta Informatica*, Vol. 5, pp. 1-236, 1975.

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

S.J. Mullender and A.S. Tanenbaum, “Design of a Capability-Based Distributed Operating System,” *Computer J.*, Vol. 29, No. 4, pp. 289-299, Aug. 1986.

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

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

T.H. Lai and S. Sahni, “Anomalies in Parallel Branch-and-Bound Algorithms,” *Proc. 1983 Int. Conf. Parallel Processing*, pp. 183-190, Aug. 1983.

A.V. Aho, J.E. Hopcroft, and J.D. Ullman, “The Design and Analysis of Computer Algorithms,” Addison-Wesley, Reading, MA, 1974.

E. Horowitz and S. Sahni, “Fundamentals of Data Structures,” Pitman Publishing, London, 1976.

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

J. Stoer and R. Bulirsch, “Introduction to Numerical Analysis,” Springer-Verlag, New York, NY, 1983.

R. Butler, E. Lusk, W. McCune, and R. Overbeek, “Parallel Logic Programming for Numeric Applications,” *Proc. 3rd Int. Conf. on Logic Programming*, pp. 375-388, London, July 1986.

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. 12th Symp. Operating Systems Principles*, ACM SIGOPS, Litchfield Park, AZ, Dec. 1989.

G.S. Almasi and A. Gottlieb, “Highly Parallel Computing,” The Benjamin/Cummings Publishing Company, Redwood City, CA, 1989.