CS5314 RESEARCH PAPER ON PROGRAMMING LANGUAGES
FACULTY: Dr. James D. Arthur **SUBMITTED BY: Aditya Varanasi**
Siddharth Anbalahan

ORCA LANGUAGE

ABSTRACT

Microprocessor based shared-memory multiprocessors are becoming widely available and promise to provide cost-effective high performance computing. Small-scale shared-memory multiprocessors are becoming widely available in implementations ranging from single-user workstations to mini-computers. Two factors working together are responsible for this trend. First, microprocessor performance has increased at a remarkable rate. Secondly, the cost of the microprocessor is a small part of total system. Within limits, adding microprocessor to a system can substantially increase performance at little additional cost.[chase89amber]

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. [bal92orca]

In this paper, we briefly describe the language and it's implementations with sample application.

INTRODUCTION

As communication in loosely coupled distributed computing systems gets faster, such systems become more and more attractive for running parallel applications. In the research conducted by Henri.E.Bal it was identified that usage of message passing and a sequential base language have many disadvantages, making them complicated for application programmers to use. [bal92orca]

Orca is a new programming language intended for implementing parallel applications on loosely-coupled distributed systems. Orca supports a communication model based on shared data and it simplifies programming. Since distributed systems lack shared memory, however, this sharing of data is logical rather than physical. [bal92orca]

Processes in Orca can communicate through shared data, even if the processors on which they run do not have physical shared memory. Unlike shared physical memory (or distributed shared memory), shared data in Orca are accessed through user-defined high-level operations, which, as we will see, has many important implications. [bal92orca]

An important goal in the design of Orca was to keep the language as simple as possible. 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, the latter option is preferred .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. [bal92orca]

DISTRIBUTED SHARED MEMORY

Most languages for distributed programming are based on message passing . 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. 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).*

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 (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.

THE SHARED DATA OBJECT MODEL

The shared data-object model provides the programmer with logically shared data. The entities shared in ORCA's model are determined by the programmer. Shared data are encapsulated in *data-objects*, which are variables of user-defined abstract data types. An abstract data type has two parts:[bal90experience]

- 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.

The shared data-object model uses two important principles related to operations on objects: [bal90experience]

1. All operations on a given object are executed *atomically* (i.e., *indivisibly*). To be precise, the model guarantees *serializability* of operation invocations: if two operations are applied simultaneously to the same dataobject, 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.

ORCA is a new programming language , 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. [bal90experience]

PARALLELISM IN ORCA

Parallelism in Orca is based on explicit creation of sequential processes. Initially, an Orca program consists of a single process, but new processes can be created explicitly through the **fork** statement: [bal90experience]

fork name(ac tual -parameter s) **[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. The parent and child processes 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. [bal90experience]

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. [bal90experience]

The implementation of an operation has the following general form:

operation op(forma l -parameter s) : Resu l tType ;
local declarat i ons

begin

guard cond i t i on1 do s tatement s1 od;

guard cond i t i on2 do s tatement s2 od;

. . .

guard cond i t i onn do s tatement sn od;

end;

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.

DATA STRUCTURES

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 . 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. [bal90experience]

IMPLEMENTATION

An example object type [bal92orca]

The specification of the object type GenericJobQueue is shown in below. 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;

ex:1

An example parallel application in Orca [bal92orca]

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-cityfirst" 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 ex 2). This object is shared among all worker processes. The manager and worker processes communicate through a shared job queue, as shown in Figure 1. The Orca code for the master and worker processes is shown in ex 3. 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.

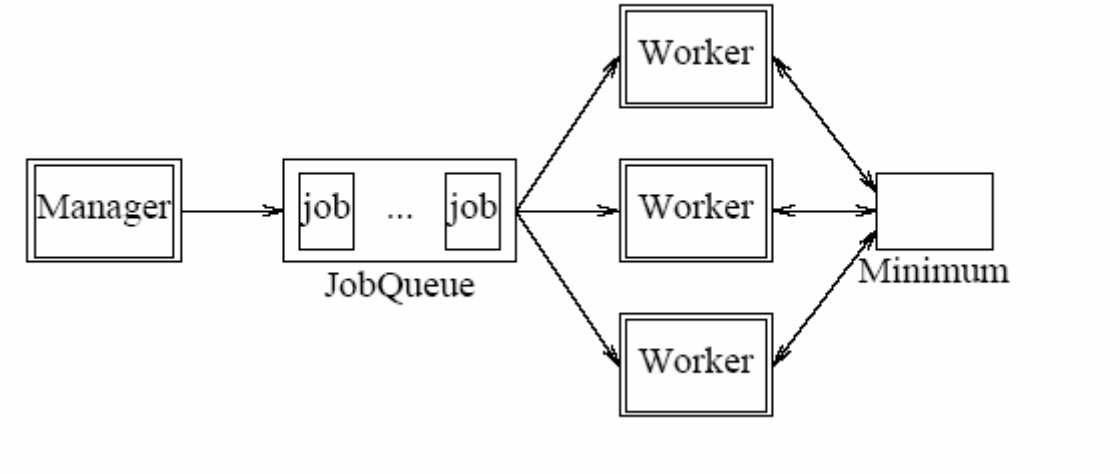

Fig. 1. Structure of the Orca implementation of TSP. The Manager and Workers are processes. The JobQueue is a data-object shared among all these processes. Minimum is a data-object of type IntObject; it is read and written by all workers.

Ex 2: Orca Code for the master and worker processes of TSP.

```
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;
```

CONCLUSION

ORCA is a language 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.

Thus ORCA is implemented to support a new model for parallel programming of distributed systems which allows processes on different machines to share data. Orca is a useful language for writing parallel programs for distributed systems.[bal92orca]

REFERENCES

[bal92orca] *Andrew S. Tanenbaum, Henri E. Bal *,M. Frans Kaashoek* ORCA: A LANGUAGE FOR PARALLEL PROGRAMMING OF DISTRIBUTED SYSTEMS.

[bal90experience] *Andrew S. Tanenbaum, Henri E. Bal *,M. Frans Kaashoek* EXPERIENCE WITH DISTRIBUTED PROGRAMMING IN ORCA

CS5314 RESEARCH PAPER ON PROGRAMMING LANGUAGES

FACULTY: Dr. James D. Arthur **SUBMITTED BY: Aditya Varanasi**
Siddharth Anbalahan