Experiences with the Orca Programming Language

Gregory V. Wilson & Henri E. Bal

Version 2.0

Abstract

We investigate the capabilities and shortcomings of Orca, a Modula-like parallel programming language supporting shared data objects on distributed memory platforms, by examining implementations of five non-trivial parallel applications: game tree searching, active chart parsing, image skeletonization, simulation of a chaotic predator/prey system, and polygon overlay.

1 Introduction

Twenty years ago, a small number of visionaries believed that massive parallelism was the future of high-performance computing. Five years ago, that view had become commonplace; today, users are increasingly sceptical of such claims. The main reason is that parallel computers have remained very difficult to program.

Many new programming systems have been developed to support parallel programming [7]. However, while comparisons of the performance of highly-parallel hardware are common [33], there have been only a few comparisons or assessment of the strengths and weaknesses of parallel programming systems. [1] implemented numerical quadrature in eight different PPSs, while [12] presents implementations of four problems—solution of a skyline matrix equation, the Hamming problems, generation of the formulae of paraffins, and a simple discrete event simulation—in eight PPSs. So far as we are aware, only [29] presents a quantitative experimental comparison of two different parallel programming systems.

Inspired by this work, the first author designed a suite of six problems with which to compare PPSs [35]. This paper describes the lessons learned from implementing some of these problems using Orca, a Modula-like language supporting apparently-shared memory on a variety of distributed-memory platforms [3]. The paper begins by describing Orca, and how it has evolved from its initial conception to its present form. Each of the applications implemented is then presented, and shortcomings of Orca, the model it embodies, and its current implementation are then given.

Our experience of this work has been extremely valuable, and has led both to re-implementation of some features of the language, and to new research into alternative mechanisms to replace some of those presently in Orca. This work would not have been possible without the hard work and dedication of the students responsible for implementing the individual problems. Peter Boncz (now at the University of Amsterdam) wrote the $\alpha\beta$ search program of Section 3.1, and also developed a theoretical model to describe its behavior. Anil Sukul wrote the active charter parser (Section 3.2); Rudolf de Boer, the image skeletonizer (Section 3.3); and Dion Nicolaas, the predator–prey simulation (Section 3.4). The first author was supported during this work by a HCM grant from the CEC.

2 Design and Rationale of Orca

Work on Orca was initiated in 1985 by Andy Tanenbaum. By then, a prototype of the Amoeba distributed operating system [22] was operational at the Vrije Universiteit, running on eight MC68000 processors connected by a token ring. Both the system and application programs were written in C, but Tanenbaum felt it would be desirable to have a higher-level language in which to write parallel applications. A Ph.D. student, Henri Bal, undertook the design and implementation of such a language.

In preparation, Bal and van Renesse implemented some example parallel programs (mostly coarse-grained and non-numerical) on Amoeba in C [4]. This demonstrated both the feasibility of writing applications on top of Amoeba, and that doing so required a great deal of effort. Even such simple things as sharing an integer counter between processes proved difficult, primarily because Amoeba's library routines for message passing, process and thread creation, and synchronization were designed with distributed operating system servers, rather than parallel applications, in mind.

This experiment led Bal to design Orca. Its key feature is that it hides message passing from its users, giving them the illusion of shared memory instead. The key technology used to do this was *replication* of shared data. If local copies of shared data are used, read operations can be executed in local memory. To update the copies, the new values or the modifications are sent to all processors which contain a copy. On many distributed systems, this can be implemented efficiently using hardware multicast or broadcast.

A preliminary design of the language provided shared scalar variables and shared sets, the latter inspired by SETL [26]. This was soon generalized to arbitrary data structures. We considered the Linda model [10, 11], which allows processes to share tuples in a common associative memory called tuple space, but felt that Linda's support for complex data structures was too low-level.

Instead, Orca's shared data structures are based on the notion of an *abstract data type*, or ADT. Processes can share arbitrary data structures, so long as these are encapsulated as instances of ADTs. We call such variables *data-objects*.

Programmers can define their own operations on shared data structures. One key decision in the design of Orca was that every such operation would execute *indivisibly*. Whatever ADTs operations the user defines, the model guarantees that they will not interfere with one another. This is similar to the semantics of monitors, and greatly simplifies programming.

An important design issue raised by indivisibility is how to deal with condition synchronization. We chose to express it inside operations, and hide it from users of the ADT. For example, to express a bounded buffer ADT, the *put* operation blocks if the buffer is full and the *get* blocks if the buffer is empty. A blocking operation automatically suspends the process which invokes it. Again, this is similar to monitors, except that in our model operations run to completion once they have started (to guarantee indivisibility), whereas with monitors they may block half way and continue later. Another difference is that data-objects do not use explicit *wait* and *signal* primitives. To ease programming, we use Boolean guards to specify when an operation may start, and have the runtime system deal with suspending and waking up processes. Figure 1 shows a simple bounded-buffer object.

Once the model was completed, we began implementing Orca. We chose a syntax similar to that of Modula-2 as a basis, both because of familiarity and because its enforced "cleanliness" would simplify compiler analysis and optimization. For this latter reason, we chose to ban global variables and pointers, and to make the language type-secure [13] to ease the detection of common memory access errors. These decisions are reflected in the design of the graph data type discussed below. Finally, the memory model chosen was sequential consistency [17]; while weaker forms of consistency [21] are potentially more efficient, they are also harder to understand.

Orca is one of the few imperative languages to contain a first-class graph data type. A graph in Orca consists of nodes identified by nodenames. The notation `G[n].f` is used to denote field `f` of a node with nodename `n` of graph `G`. Like pointers, nodenames denote cells of dynamic data structures. The syntax is similar to array indices; however, the graph to which the nodename refers must be specified in a denotation. This simplifies static analysis of Orca programs and makes the graph abstraction suitable for distributed-memory systems. Also, the usage of nodenames is validated at runtime, thus making graphs type-secure.

Before describing the implementation of Orca, it is instructive to compare our model with *shared virtual memory* (SVM) [18]. Like Orca, SVM offers programmers shared memory instead of message passing. Unlike Orca, however, SVM simulates a physically-shared memory by partitioning the address space into fixed-size pages, which are moved and copied over the network into processors' local memories.

```
OBJECT IMPLEMENTATION buffer;
   CONST MAXSIZE = 10;            # Maximum size of the buffer

   # Local state of the object:

   buf: ARRAY[integer 0..MAXSIZE-1] OF integer;   # the buffer itself
   in, out: integer;              # index of next element to put/get
   size: integer;                 # current size

   OPERATION put(x: integer);
   BEGIN
      GUARD size < MAXSIZE DO     # blocks until there is room
      buf[in] := x;               # store element
      in := (in + 1) % MAXSIZE;   # bump input index
      size +:= 1;                 # increment size
      OD;
   END;

   OPERATION get(x: OUT integer);
   BEGIN
      GUARD size > 0 DO           # blocks while buffer is empty
      x := buf[out];              # retrieve element
      out := (out + 1) % MAXSIZE; # bump output index
      size -:= 1;                 # decrement size
      OD;
   END;
END;
```

Figure 1: A Bounded Buffer Object in Orca

Orca differs in being based on user-defined objects, rather than fixed-size pages. It is therefore free from false sharing. Also, objects in Orca are accessed through high-level user-defined operations, whereas SVM is accessed through low-level machine instructions. This gives us more flexibility in the implementation. With SVM, for example, it is inefficient to update shared pages after every write, since many consecutive write instructions may be issued. Therefore, replicas are always invalidated rather than updated. In our model, the programmer typically defines a single, high-level operation to change a data structure, so we can broadcast these operations and update copies of shared data.

The Munin system [8] is intermediate between "classic" SVM and Orca. In Munin, shared objects are classified as belonging to one of several pre-defined classes, each of which is handled by a specific protocol. These protocols are customized for such common cases as producer-consumer structures, read-mostly variables, synchronization objects such as locks, and so on. Each protocol determines whether the object is replicated or centralized, and how read and updates are managed. However, no explicit association between objects as ADTs and the (possibly concurrent) operations on them is made.

The actual implementation of Orca started in 1988. The compiler used the Amsterdam Compiler Kit [30]. The first runtime system (RTS) was implemented on a shared-memory machine; it stored shared data-objects in shared memory and protected accesses to them using locks. The first distributed RTS ran on top of Amoeba, using Remote Procedure Call (RPC) as the single communication primitive. Write operations were implemented by doing an RPC to each processor containing a copy of the object, which was expensive.

A much more efficient implementation on Amoeba was then developed by Frans Kaashoek, a Ph.D. student working on reliable group communication. He implemented an efficient, totally-ordered, reliable broadcast primitive in the Amoeba kernel. The Orca RTS based on this replicates each object on all processors that can access it, and implements write operations by broadcasting the operation and its parameters. Since broadcasts are totally ordered, consistency of the replicas is maintained [14].

A disadvantage of Kaashoek's initial RTS was that it always replicated shared objects. For objects with a low read/write ratio, it is more efficient to store the object on one processor. We attacked this problem with compile-time and run-time optimizations [6]. In the optimized system, the compiler estimates how often each type of process reads and writes the shared objects. It passes this information to the RTS, which uses it to decide whether or not to replicate an object. If the estimated read/write ratio is too low, the object is stored on a single processor and accessed through RPCs by other processors.

The Orca implementations described above ran only on Amoeba. Starting in 1993, we developed a new implementation that is much more portable. The new compiler generates ANSI C and can easily be retargeted to other processors. A virtual parallel machine called *Panda*, which provides threads, RPC, and totally-ordered group communication in a system-independent way is now the interface to the run-time system on all machines [9]. The new Orca RTS only calls the Panda primitives, so that porting Orca to a new platform only requires re-implementation of a small number of Panda primitives. At present, this has been done on several operating systems (including Amoeba, SunOS, and Solaris) and multicomputers (Thinking Machines Corporation CM-5 and the Parsytec GCel, Parsytec PowerXplorer, and CS-2).

Many people have contributed to the Orca project. Frans Kaashoek and Andy Tanenbaum have already been mentioned. Jack Jansen developed the original RPC-based Amoeba RTS. Ceriel Jacobs wrote the new Orca compiler. Raoul Bhoedjang, Tim Rühl, Koen Langendoen, Rutger Hofman, Heinz-Peter Heinzle, and Marco Oey each wrote parts of Panda or the Orca RTS. Several visitors (Irina Athanasiu, Saniya Ben Hassen, Noel Carmody) wrote example applications in Orca.

3 Applications Implemented

Five of the applications described in [35] were implemented in this work. The sections below briefly describe each in turn, along with its Orca implementation. Section 4 then summarizes our experiences with these applications, and the lessons learned from them.

3.1 Game Tree Search

Kece is a zero-sum game similar to Scrabble. Given an ordered list of W words, two players try to construct a crossword pattern on an $N \times N$ board. The game starts with at least one initial word placed on the board. At any time, only the first T words in the list may be used, where T is typically two or three. The score for each move is the number of letter overlaps created; this must be at least one. Words may only be played left-to-right or top-to-bottom. Figure 2 shows a game in progress; the word "trouble" could be played in any of the positions shown to earn a score of either 1 or 2, while "systolic" or "compiler" could be played elsewhere for other scores.

Computers can play zero-sum games by searching game trees. The simplest algorithm for doing this is called minimax. It evaluates a game tree bottom-up, assigning the minimum child value to each node at odd levels, and the maximum child value at even levels [15, 23]. The $\alpha\beta$ algorithm improves this by cutting off branches which minimax would have searched. Figure 3 illustrates this: $\alpha\beta$ stops searching the second and third subtrees once it finds that they are inferior to the leftmost (first-searched) subtree.

The $\alpha\beta$ algorithm maintains two cutoff values, α and β, for each node. The root node starts with $(-\infty, \infty)$; each nodes passes $(-\beta, -\alpha)$ to its descendents. The α-value of each leaf node is computed by a game-specific evaluation func-

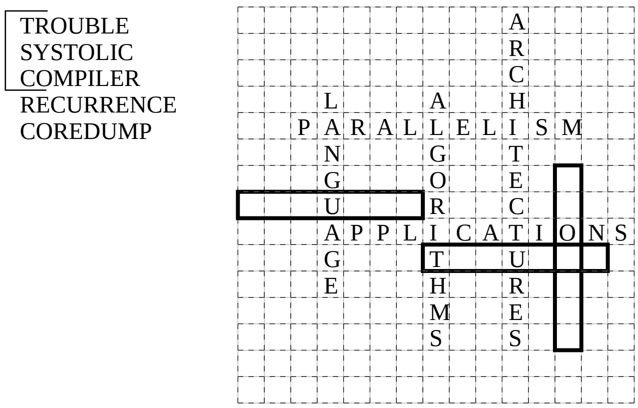


Figure 2: A Kece Game in Progress

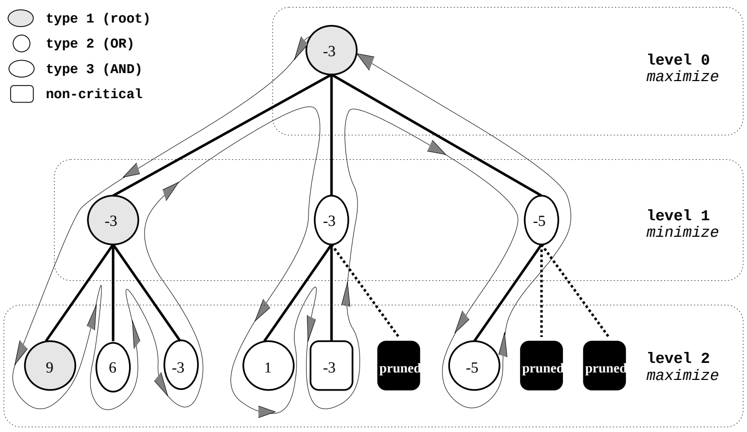


Figure 3: Searching a Kece Game Tree

tion. When the α value of a child is available, the parent node uses its negation to update its own α. If this is now larger than the parent's β value, the algorithm *prunes* this branch of the search.

$\alpha\beta$ search was first described in the 1950s [23]. Subsequent analysis of its theoretical properties can be found in [15] and [24], while discussion of its parallelization can be found in [19, 25].

A program which uses $\alpha\beta$ search to play Kece contains two principal data structures: an $N \times N$ array of characters, showing the current configuration of the board, and an $N \times N$ array of integers recording the overlay count for each character. Given these two structures, the score for a move can be computed simply and incrementally. The major work of a sequential Kece program therefore lies in generating new moves in a given game situation. This is done by a routine `match`, which makes use to two pre-computed functions:

- `nextpos(w,i)` returns the position of the next occurrence of character i in word w, or 0 if i does not occur in w.
- `lastpos(w, c, i)` returns the largest index smaller than i at which c occurs in w, or 0.

Two broad classes of move generation strategies are possible in $\alpha\beta$ search: incremental (as-needed) generation, and ordered (all-at-once, then sort) generation. Both strategies were implemented as part of a prototype Kece program written in ANSI C. The heuristic used for ordered move generation was the immediate score for the move. Experiments showed that ordered move generation was clearly superior.

The prototype C program was then translated into sequential Orca. Once again, the board was represented as an array of characters. This structure was put in a record together with the WordList, which contained the playable words:

```
TYPE Board = RECORD
  player : integer;
  score  : integer;
  wl     : WordList;
  board  : ARRAY [integer 0..(N*N)-1] OF char;
END;
```

While these four items are logically separate, they were combined into a record to reduce the number of parameters passed to each procedure call. Since Orca does not permit global variables, this kind of packaging is often necessary.

The WordList object stores words in order, and provides the `nextpos` and `lastpos` operations described above. During search, a second list called MoveList is also used to keep track of possible moves. Both of these lists were implemented using graphs.

Profiling of the Orca program showed it to be 20 times slower than the C. The two operations `wl$next` and `ml$put` caused a word record to be copied recursively upon each invocation. Since the set of words available for play does not change during the game, the object-based implementation was abandoned, and a module used instead. While this led to convoluted expressions of the form:

```
board.wordlist[w_ptr].word.str[i]
```

it sped the program up by a factor of five.

With this optimization in place, the sequential Orca program was parallelized using *tree-splitting*. This was modelled on a parallel $\alpha\beta$ Orca program developed previously at the Vrije Universteit [5].

The Kece program represents the game tree as a shared data-object, which is initialized and managed by a controller process. Along with this controller, there are one or more worker processes which search subtrees. Assume that we wish to search the game tree to depth D. We begin by having the controller search the top of the tree to depth S, $1 \leq S < D$. The leaves of this shared tree are then used as initial states for the workers, who search the J-deep subtrees below it ($J = D - S$). The controller task farms these positions to the workers by putting references to the leaves in a shared job queue. The workers take jobs from this queue, retrieve the appropriate information from the game tree, and start searching. When a worker finishes a subtree it stores its α value in the game tree and takes a new job. A barrier is used to control the start and termination of all work.

This parallel algorithm suffers from three overheads:

Search overhead: In serial $\alpha\beta$ search, subtrees are evaluated one by one, so that the result of searching a left subtree can prune the search of a right subtree. If subtrees are searched in parallel, such pruning does not take place. In the extreme case, in which all subtrees are searched simultaneously, $\alpha\beta$ search reduces to minimax search.

Communication overhead: To reduce search overhead, workers must share partial results. As the frequency of communication or the number of workers increases, this can become a burden.

Load imbalance: If job sizes are uneven, some workers may terminate and be idle while others are still working. This can be ameliorated by creating many more jobs than there are workers.

To reduce these overheads, the original algorithm was modified in two ways. First, the original implementation generated the search tree to depth S before farming any work. In the revised implementation, workers take jobs as soon as they become available. This turns the first stage of the program into a producer-consumer problem. (Because of memory limitations, it was necessary to block the controller when a certain number of jobs were in the pending leaf pool.) Second, the worker procedure was modified to poll the search tree structure regularly to ensure that the subtree it was working on was still interesting, i.e. that the initial position of the subtree had not been pruned by some other worker's search.

3.2 Active Chart Parsing

Consider the sentence "Jane saw Fred in the park with the telescope". Depending on how it is parsed, it may mean:

- Jane and Fred were both in the park, and Jane used a telescope to see Fred;
- Jane saw Fred when Fred was in the park carrying the telescope;
- Jane and Fred met in a park that contained a telescope

and so on. This ambiguity exists because the grammar used to generate this sentence is ambiguous.

Active chart parsing (ACP) is an efficient technique for generating all possible derivations of a sentence from an ambiguous grammar [36]. ACP starts by inserting one node between each word in the sentence to be parsed, plus one node at the start of the sentence and one at its end (Figure 4). ACP then connects adjacent nodes with arcs showing the terminal categories to which the intervening word belongs. The algorithm then repeatedly adds new arcs according to the following rule: if the grammar contains $A \rightarrow BC$, and there are nodes n_{left}, n_{middle}, and n_{right} such that $n_{left} \xrightarrow{B} n_{middle}$ and $n_{middle} \xrightarrow{C} n_{right}$, then the arc $n_{left} \xrightarrow{A} n_{right}$ is added. This process continues until no more arcs can be added. All *Sentence* arcs spanning the entire sentence are then parsings of the sentence.

The key to ACP's efficiency is that arcs can be stored as they are generated, so that they only need to be generated once. A sequential implementation of ACP typically relies on the following data structures:

- A lookup table which stores the reverse of the grammar. Instead of storing the rules by which terms may be expanded, this records the ways in which pairs of terms might be generated. For a T-term grammar this can be a $T \times T$ matrix of lists of rules, a hash table, a list of lists, or some other structure.
- A dictionary showing the terms to which each terminal belongs.
- A chart representing a list of nodes, each of which contains information on incoming and outgoing arcs. (This is the "active chart" that gives the algorithm its name.)
- A set of arc descriptors, each containing: the source and destination node of the arc; the name of the term the arc represents; and the arc's constituent edges (which is needed in order to reconstruct the parse tree once a complete sentence is found).

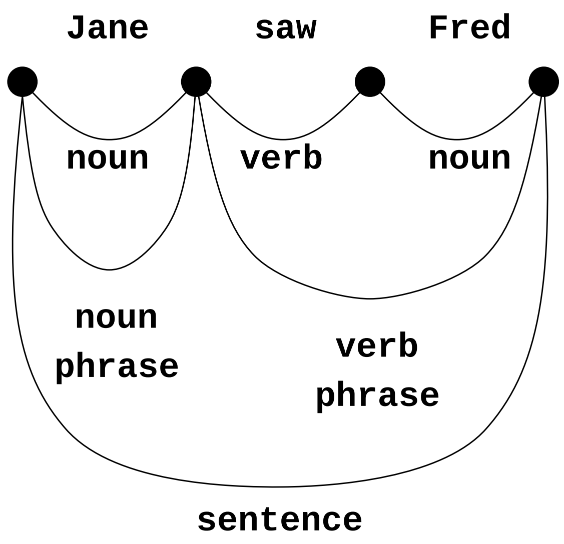


Figure 4: An Example of Active Chart Parsing

- A set of descriptors of arcs which have been generated, but which have not yet been added to the chart. This may be managed as a queue, a stack, or some sort of priority queue if heuristics are being used to guide parsing. In this work, the pending edges table was implemented as a queue, and is referred to below as the PEQ.

Given these structures, the sequential ACP algorithm is as shown in Figure 5.

The sequential Orca implementation of ACP used this same algorithm, and relied on graphs to implement dynamic data structures such as the PEQ. Initially, this program used objects for all ADTs. For reasons discussed in Section 4.1, however, this proved very expensive. A second sequential Orca version of ACP was therefore written which did not use objects. This program was approximately the same length as its C equivalent, but ran approximately 5 times slower, primarily due to the heavy use of graphs to implement dynamic data structures.

ACP can be parallelized by creating one worker process for each inter-word node, and allocating one or more workers to each available processor. After some local initialization, each worker executes the algorithm given in Figure 6, which is a parallel version of the one given in Figure 5. Whenever a worker generates a new arc, it must inform the nodes responsible for the arc's endpoints. This was done by passing a "mailbox" structure to all workers as a shared parameter. The mailbox is implemented as an array of PEQs, one for each worker. Each

```
read sentence
for each word W do
  add all categories to which W belongs to the PEQ
end for
while PEQ not empty do
  elt := dequeue(PEQ)
  if(elt is a sentence) then
    print sentence
  end if
  add elt to the chart
  expand elt
  add new arcs to PEQ
  combine elt with left and right neighbors of source and destination
  add new arcs to PEQ
end while
```

Figure 5: The Active Chart Parsing Algorithm

time a new edge is generated, the generating worker updates the PEQs of the source and destination nodes by invoking operations on their mailboxes. (Note that in the parallel version of ACP, the PEQ must be an object, since only objects may be shared between processes.)

When running sequentially, ACP terminates when the (single) PEQ is empty. Terminating the parallel version is more difficult; the fact that a node's PEQ is empty is no reason for it to stop, since other nodes may later generate edges for it. This problem, which is analogous to termination detection in the Arc Consistency Problem [2], can be expressed as two conditions:

1. all workers are idle
2. all PEQs are empty

Information about the number of active workers and the contents of all PEQs is stored in an object shared by all the workers (the parameter `work` of Figure 6); these are stored together to simplify synchronization. The number of active workers is maintained as an integer and is initialized to the number of words in the sentence. Information on the contents of the PEQs is maintained as an array of Booleans. Each time a node adds an edge to the PEQ of another worker, the corresponding field in the array of the `work` object is assigned `true`. Each time a node enters the main loop of Figure 6 it assigns `false` to its entry in `work`.

Whenever a worker exits the main loop, it decrements the number of active workers by calling an operation on `work`. It then calls another operation on `work` which checks the termination condition. If this condition is false, the number of active workers is incremented and the worker re-enters its main loop. Otherwise, all workers may safely terminate.

```
PROCESS worker(
  W      : word_t;                                 # this node's word
  own_id : integer;                                # this node's ID
  M      : SHARED array [integer] of mailbox_t;    # mailboxes (by ID)
  EC     : SHARED EdgeCollection;                  # edges
  work   : SHARED Work;                            # termination control
  ...
);
  do local initializations
  add all categories to which W belongs to M[own_id] and M[own_id + 1]
  while M[own_id] not empty do
    elt = dequeue(M[own_id])
    if (elt is a sentence) then
      print_sentence
    fi
    add elt to local incoming/outgoing edges in edge collection EC
    add edges generated by adding elt to own incoming/outgoing lists
    handle termination detection using 'work' object
  od
END
```

Figure 6: Parallel Active Chart Parsing Algorithm

Inspection shows that the EdgeCollection and Work objects had a major impact on the volume of communication in the program. Two optimizations were therefore implemented to reduce this.

The EdgeCollection object was introduced in the serial Orca version to implement the arc data structure of the serial C version. Its main purpose is to implement an equivalent to pointers in Orca, so that parse trees can be reconstructed when parsing is finished. Although this is the most natural way to store the parse tree, it forces processes to notify each other whenever any change is made anywhere in the tree. We could have each node store a subset of the graph, and identify edges with (node, index) tuples. There is however a problem with this strategy: since an edge is added to the PEQs of both its source and destination, we need to know in advance its index in the outgoing edge list of its source node. However, since an edge is first added to the PEQ of the source node, it is impossible to know without any communication what this index will (eventually) be. In order to implement local pointers, the following scheme is used. Each node N maintains an array `ocnt` of integers, one for each node. `ocnt[i]` indicates the number of edges that N has added (i.e. sent) to node N_i. The triplet (src-node, N, $ocnt_N[src - node]$) then uniquely identifies an edge added by node N to node $src - node$. When ACP terminates, all nodes store their outgoing edges in a special object; by scanning through the outgoing edge list of the first node, one can find all sentence arcs and then trace the corresponding parse tree.

A second optimization is to modify termination detection control. As explained earlier, the `work` object manages bookkeeping for termination. Each time a worker enters its main loop, and each time a worker generates a new edge, `work` is updated.

In the new strategy, each node N maintains an array `workarr` of integers, one for each other node. `workarr[i]` denotes the number of pending edges that have been added to node i by N during its main loop. After its main loop, each worker sends its `workarr` to `work`, which then adds the elements of `workarr` to its own array. The second part of the termination condition then becomes: "all entries in the array maintained by `work` are 0". In effect, this optimization combines many individual messages to `work` into a single packet.

3.3 Image Skeletonization

Skeletonization is a common operation in image processing, arising in handwriting recognition, the analysis of electron density maps of proteins, and many other areas. During skeletonization, an image consisting of pixels with values in the range $1 \ldots G$ is reduced to line segments one pixel wide. (The value 0 denotes an empty, or clear, pixel.) This is done in two phases. In the first, the image is "thinned" by repeatedly clearing pixels which lie on the edges of segments (Figure 7). Pixels with low gray values (i.e. faint pixels) are cleared first, and darker pixels later, in order to retain the most significant part of the skeleton. In the second phase, adjoining pixels are labelled to identify connected components. Numerous thinning algorithms are described in the image processing literature [32, 16].

The thinning algorithm used in this work operates by making repeated passes over the pixel array. In the pass for gray level g, $1 \leq g \leq G$, only pixels whose value is g are candidates for removal. Any pixel with this value is removed (i.e. its value is set to 0) unless:

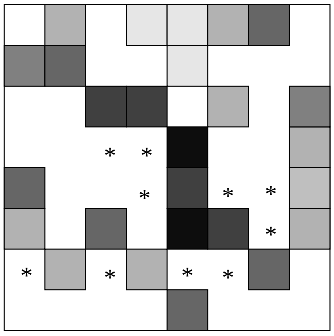
- it is at the tip of a line segment; or
- removing it would disconnect a presently-connected image component.

Figure 8 gives examples of these cases. In the last of these, either of the two pixels shown could be removed, but not both. In a serial implementation, this ambiguity is automatically resolved by the order in which the pixels are examined: the first pixel encountered is removed, and the second is retained. A major concern in a parallel implementation is to ensure the same behavior.

The rules governing pixel removal can be encoded in one of two ways. The first is by constructing an 8-bit code showing the state of each pixels' neighbors. This is done by setting the low-order bit to 1 (0) if the pixel above the one in question is full (empty), the next bit to 1 (0) if the pixel above and to the

0	3	0	1	1	3	5	0
4	5	0	0	1	0	0	0
0	0	6	6	0	3	0	4
0	0	4	5	7	0	0	3
5	0	0	3	6	4	2	2
3	0	5	0	7	6	3	3
3	3	5	3	6	5	5	2
0	0	0	1	5	0	0	0

original image



thinned image

* = pixel deleted

Figure 7: Thinning During Skeletonization

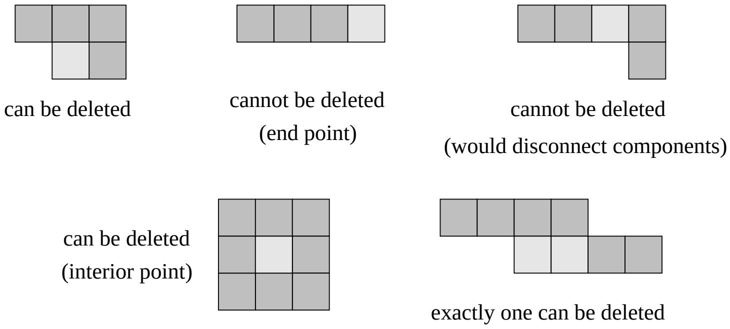


Figure 8: Thinning Cases

right of the pixel in question is full (empty), and so on. Once this code has been constructed, it can be used to index a Boolean lookup table. The second way to encode the rules governing pixel removal is to use explicit conditional switching, in the form of nested `if-then-else` statements. Since the rules used were relatively simple, the latter method was used in this work.

Once the image has been thinned, its connected components can be found using a scan-and-flood algorithm. The outer loop of this algorithm scans across the entire image. Each time an unlabelled pixel is found, a recursive flood-filling algorithm is called to label its pixels. The outer loop also records the seed pixel of each component for later output.

The obvious way to parallelize the first of these two stages is to divide the image into patches, and have a separate process apply the sequential algorithm to its patch. However, in order to ensure a correct result, patches must exchange their outermost two layers of pixels after each iteration. If this is not done, then if a pair of thinnable pixels like those in Figure 8 overlap a patch boundary, the processes responsible for the patches may make different decisions about which to keep and which to delete.

Parallelizing the second half of this problem is more challenging. While connectivity algorithms similar to matrix multiplication are well known, they are inappropriate for the kinds of data manipulated in this problem, since most pixels are connected to only one or two others. A more efficient alternative is to have each process trace the components in its patch, and then construct a graph whose nodes represents local connected components which overlap boundaries (Figure 9). A transitive closure algorithm can then be used to propagate coloring values between processes.

A prototype skeletonization program was implemented in ANSI C. Early tests revealed the problem of "image drift". Because the algorithm deleted pixels as soon as it could, skeletons had a tendency to "drift" downward and to the right (if the scanning order was left-to-right, top-to-bottom). To correct for this, the algorithm was changed so that scanning was done in four different directions on successive thinning iterations. A pre-processing stage was also added to delete isolated pixels. This not only produced a cleaner image, but also reduced processing time.

In the parallel version of the program, each process maintains an internal array representing its patch, plus its own borders and the borders of its neighbors. Each process is passed eight mailbox structures when it is created. At the end of each iteration, it writes its own borders to four of these, which are read by its neighbors, and then reads from the other four, to which its neighbors have written. Section 4 describes how these mailboxes were implemented in order to maximize efficiency. Note that explicit barrier synchronization is done at the end of each iteration to ensure that processes do not race ahead of one another. This barrier also provides an opportunity for graphical output.

This first half of the image skeletonization program was straightforward to implement, but proved difficult to debug. The main reason was the difficulty of

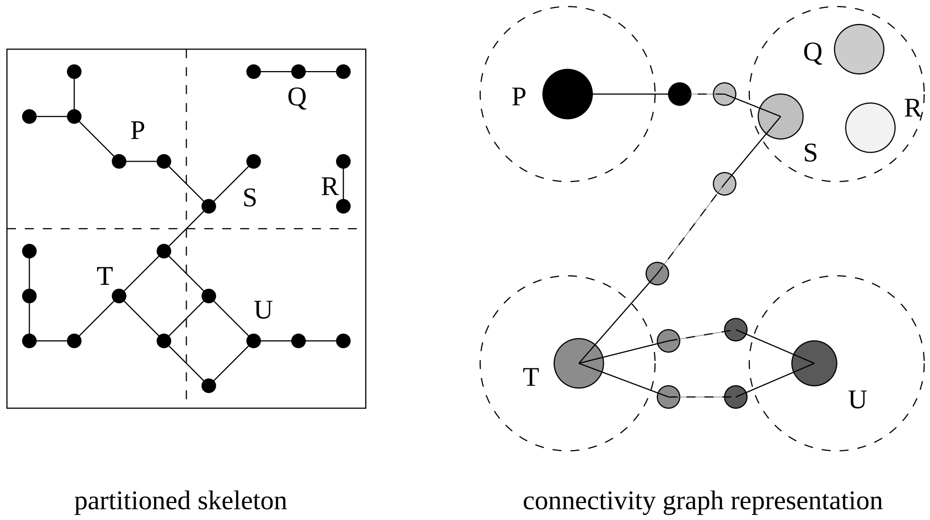


Figure 9: Stitching Skeletonized Segments Together

testing it, which was due in large part to the fact that the parallel version of the program usually generated a slightly different answer than the serial one. This occurred because each patch was being thinned separately—the "cascade effect" of thinning from one corner to the diagonally-opposite corner sequentially did not occur, and so slightly different decisions could be made within each patch.

Once the image was thinned the component labelling algorithm was invoked. Its first pass was identical to the sequential labelling algorithm discussed above: each process scanned its patch, invoking a flood-fill routine to trace out components as they were encountered. Once this was done, each process constructed a `CrossList` structure of the kind shown in Figure 10. Figure 11 shows an example of these structures, with three lists of pixels. Three crosspoints refer to two of those skeletons.

Before processes can decide what color to paint their common skeletons, they must know which skeletons they are going to argue about. This is done by searching all skeletons for pixels adjacent to the patch boundary. When such a pixel is found, and is connected to a pixel in the border, its position and the color of its skeleton are recorded in the crossing structure. These structures are stored in a two-dimensional array so that the cross points can be sent to neighboring workers. When all workers have finished searching for pixels from skeletons that cross a patch boundary, they synchronize at a barrier, and start the agreement phase. Each worker searches for crosspoints that are connected, yet have a different color, until no worker can find two crosspoints whose colors

```
TYPE Pixel      = NODENAME OF PixelList;
TYPE PixelList  = GRAPH
    root      : Pixel;
  NODES
    hor, ver  : integer;
    nextPixel : Pixel;
END;
TYPE Skeleton   = NODENAME OF SkelList;
TYPE SkelList   = GRAPH
    root      : Skeleton;
  NODES
    color     : integer;
    skel      : PixelList;
    nextSkel  : Skeleton;
END;
TYPE Crosspoint = NODENAME OF CrossList;
TYPE CrossList  = GRAPH
    root      : Crosspoint;
  NODES
    hor, ver  : integer;
    color     : integer;
    skeleton  : Skeleton;
    nextCross : Crosspoint;
  END;
```

Figure 10: Cross List Structure

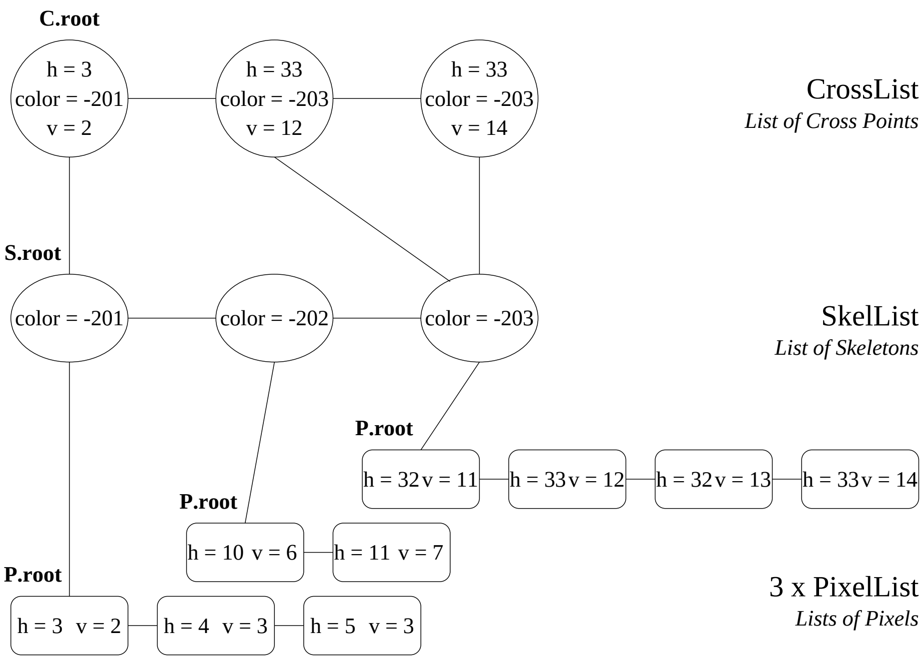


Figure 11: Example of Cross List Structure

do not match. When this happens, the color of the lower-numbered process is adopted by the higher-numbered process (to avoid ambiguity). For the sake of efficiency, pixels are not immediately relabelled when a component is re-colored. Instead, the color associated with the component record is altered, and pixel colors are changed only once all components have been connected.

3.4 Predator–Prey Simulation

In 1952, Alan Turing presented a model of the interaction of two chemicals in a ring of cells [31]. Turing's model is a spatial system in which reactants interact within locations and migrate between locations. It can thus represent populations of predators and prey in a spatially-segmented habitat. His work was later generalized and synthesized to cover many different kinds of reaction-migration systems, and is now of importance in such fields as ecology [27], epidemiology [20], and petroleum engineering [34].

Turing used the following different equations to model the interaction of reactants and migration between adjacent cells. Let X_i and Y_i (for $i = 1 \ldots N$) be the concentrations of reactants (or populations of predators and prey) in each cell i of a ring-shaped world of size n, and let μ and ν be the migration rates between neighboring cells for X and Y respectively. Then:

$$\frac{dX_i}{dt} = f(X_i, Y_i) + \mu \cdot (X_{i-1} - 2X_i + X_{i+1}) \quad (1)$$

$$\frac{dY_i}{dt} = g(X_i, Y_i) + \nu \cdot (Y_{i-1} - 2Y_i + Y_{i+1}) \quad (2)$$

where $f(X, Y)$ and $g(X, Y)$ can be expressed as:

$$f(X, Y) = X(r_X + a_X X + b_X Y) \quad (3)$$

$$g(X, Y) = Y(r_Y + a_Y X + b_Y Y) \quad (4)$$

Note that cyclic boundary conditions are used: location 1 borders location n. By using different values for the coefficients $r_{\{X,Y\}}$, $a_{\{X,Y\}}$ and $b_{\{X,Y\}}$ different systems can be obtained.

To simulate the evolution of this system, these differential equations must be discretized. Iterations over the populations are performed in small time steps (typically 10^{-4}. The populations of the locations are updated probabilistically, i.e. the system decides randomly whether each predator or prey gives birth, migrates, or dies.

Implementing the Turing Ring in a sequential language is straightforward. The ring is represented by an array of records, each of which contains the predator and prey populations for a single cell. The program is then a triply-nested loop, in which the outer loop executes iterations, the middle loop traverses cells, and the inner loop examines individual predators and prey. Profiling an ANSI C implementation based on this scheme showed that 90% of the program's execution time was spent generating random numbers. When this program was translated into sequential Orca, its speed was reduced by a factor of approximately 2.5, mostly due to the dynamic array-bound checking done by the Orca RTS.

The Turing Ring could be parallelized using simple spatial decomposition[1]. For certain ranges of parameters, however, this system is chaotic. As it evolves, the populations of neighboring cells can come to differ by up to three orders of magnitude [27]. If the system is spatially decomposed for parallelization, some form of load balancing will clearly be required. What is more, since the populations of cells rise and fall over time, the load balancing must be dynamic, i.e. work must be re-apportioned as the simulation is running.

On a system supporting true shared memory, task-farming could be used to dynamically allocate work within each time step. On distributed-memory hardware, it makes more sense to allocate each processor a contiguous portion of the ring, and then have busy processors give cells to neighboring processors which are less heavily loaded. One advantage of this scheme is that the communication required by the load-balancing protocol can then be "piggy-backed" on top of the communication of boundary values at the end of each outermost loop.

During each iteration, each worker must:

1. accept the animals which migrated from the neighboring processors during the previous iteration;

2. display the current state of its part of the ring (if the graphical display option is enabled);

3. handle birth and death in its part of the ring;

4. handle migration within its part of the ring, determining which animals move to the neighboring CPUs;

5. send migrating animals to its neighbors; and

6. wait for the other workers to finish this iteration.

This strategy was implemented by creating two arrays of link objects, which simulate point-to-point connections between neighboring processes. Each link has operations for writing and reading data denoting the number of predators and prey which are migrating between the border cells of the two processes that share the object. The synchronization in the last step is required to ensure that

[1] An alternative scheme which decomposes the problem at the level of individual predators or prey, has been implemented on massively-parallel SIMD hardware [28], but is not appropriate for the sort of machine for which Orca was intended.

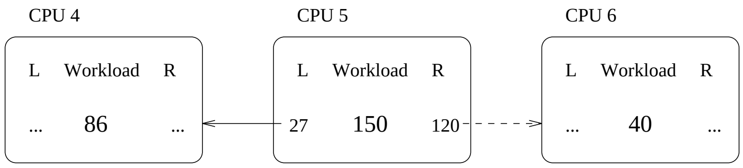


Figure 12: A Worker Which Should Send To One Side

workers receive and process their neighbors' messages before their neighbors proceed into the next iteration, which might overwrite those data.

Initially, all cells in the ring contain the same number of predators and prey, so an even distribution of work can be achieved by giving each processor responsibility for the same number of cells. During each iteration, a processor receives not only its neighbors' workload, but also an estimate of what its neighbors' workload would be if the intervening boundary cell were shifted. This information is packed into the same link objects used to communicate boundary populations; the amount of data communicated between the two processors therefore increases, but the number of communication operations (and hence the frequency with which the startup cost of the message-passing system must be paid) does not. Each process can use this information to determine whether off-loading one or both of its boundary cells would lead to a more even distribution of work.

The rules used to decide when to off-load cells must be chosen with care in order to avoid *thrashing*. This occurs when a processor p_i off-loads a heavy boundary cell c_a to a neighbor p_j. On the next iteration, p_j finds that its workload is much greater than p_i's, and so passes the cell c_a back to p_j (Figure 12).

The link object is the most frequently used shared object in the parallel implementation. During each iteration, migrating animals are written to the link and then read by the neighboring CPU. Furthermore, the load balancing negotiations write some data to the link. Initially, the link object provided separate read and write operations for each of its data elements. However, some write operations (e.g. writing the migrating predators and writing the migrating prey) always take place one after the other. Combining these into one operation reduced network traffic by a factor of two, and total execution time by 30 per cent. A further optimization would be to combine all co-occurring operations in this way. Such combining is done frequently in Orca programs, but can make program implementations more difficult to understand, since the operations combined may have little to do with one another.

To assess the load balancing algorithm, a mix of balanced and unbalanced scenarios were run on varying numbers of processors. With unbalanced input, the execution time of the non-load-balancing implementation was more than double that of the load-balanced implementation. The overhead of the load-balancing algorithm was measured by running the program with and without the load-balancing code for a perfectly-balanced input. The running times for the two cases differed by only one or two per cent, which can be attributed to variant evolution of the system's state due to random number generator variations.

4 Lessons Learned

A great deal was learned about Orca during the implementation of the applications described above. The sections below divide these lessons into those which apply to the sequential portions of Orca, those which apply to the shared data-object model and the Orca language, those which are problems with Orca's current implementation (but which can be modified in future implementations without revising the language's appearance), and those which are actually problems with the applications, independent of the language being used to implement them.

4.1 Sequential Orca

As described in Section 2, graphs were introduced into Orca in order to avoid problems that arise in distributed-memory systems which permit direct pointers. In order to preserve the integrity of graphs, pointers into them are not allowed; instead, assignment of a graph node's value to a free-standing record causes the node's value fields to be copied.

Experience showed that this resulted in a very large overhead for some types of application. In ACP, for example, the dictionary was originally implemented as a list of lists, i.e. as a graph. In the initialization phase of the sequential Orca version, we would like to know all categories to which a word belongs. Since the dictionary stores exactly this information per word, the dictionary contains a function *give_ith_elt_dict(i, symbollist)*, which returns the list of the non-terminals to which the i^{th} word in the dictionary belongs. This is most efficiently done by returning a *pointer* to the symbol list. However, in Orca this is done by returning a *copy* of the list, which is very slow.

In most cases, programmers found it possible to sidestep this problem by re-organizing their data structures in a way that avoided the graph construct. For example, the `next` operation on the word list in Kece, and the `put` operation on the move list, initially involved recursive copying of a data structure. They also involved calls to `malloc` and `free`, since the `addnode` and `deletenode` operations on graphs did not cache graph nodes. Modifying these operations to avoid copying and memory allocation sped up the Kece program by almost a factor of five.

4.2 The Model and Language

Another problem that arose during the parallelization of the ACP program was that shared graphs were indivisible—either the whole graph was replicated on each processor, or the graph was stored on a single processor. This rule prevents problems with pointers being moved between address spaces, but forced programmers to choose between simplicity and efficiency.

For example, the `EdgeCollection` making up the graph was originally a single structure; every time a new edge was discovered, it was added to this graph. However, in the parallel version this meant that every time a process did any "useful" work, it had to broadcast a message to all processors to inform them of the update to the `EdgeCollection`. Section 3.2 describes the circuitous manner in which the programmer circumvented this problem; the resulting code was significantly more complex and harder to understand than the direct, but inefficient, version.

A second difficulty with the model is that Orca does not permit array slicing, i.e. it is not possible to use a notation such as `v[1:N:2]` to select odd-numbered elements of a vector `v`. Such a facility would have been very useful in a variety of contexts, such as allowing the borders of patches in the skeletonization code to be transmitted without first having to be marshalled (copied) into an intermediate structure, or allowing several inter-process link objects to be selected and passed as a single parameter to workers in that code and the Turing Ring, instead of requiring the programmer to pass a separate parameter for each worker-worker boundary. The problem, of course, is one of aliasing. If array sub-sections could be selected and passed as parameters, it would be possible for processes to concurrently update overlapping portions of a single array in an unsafe manner.

Finally, the original design of the Turing Ring called for the load-balancing algorithm to be implemented as an object, so that different algorithms could be encapsulated and parameterized. However, Orca does not allow shared variables as parameters to an operation, so it was not possible to encapsulate the load-balancing strategy in a "generic" object of some kind, as might have been possible in an object-oriented language.

4.3 Current Implementation

Several of the problems encountered during this work are not fundamental, but rather reflect the ongoing development of the Orca compiler and run-time system. Chief among these is the run-time cost of using objects as abstract data types (ADTs) for software engineering purposes. The attractions of doing this are two-fold: a single mechanism is used for abstraction, and programmers can delay decisions about which ADTs are to be used as part of the parallelization of their programs. Unfortunately, the present compiler inserts the protocols required to ensure object consistency in the face of concurrency even if that object is private to a single process. This makes objects unacceptably expensive for in-line sequential use. The solution we are investigating is to "clone" procedures, i.e. to construct versions for both object and non-object parameters, and have the compiler sort out which is needed where. The cost of this approach is a potentially large increase in program image size.

A second shortcoming of the present implementation is that small communication operations are not automatically marshalled into a single, coarser operation. In the skeletonization program, for example, the most natural implementation would have a worker i write its border elements one at a time to its neighboring worker $i-1$; the compiler and run-time system could combine these one-word writes into a single, larger operation. Doing this automatically would be no more efficient than doing it by hand, but would free programmers from the conceptual burden of the intermediate (and logically unnecessary) stage.

Third, record-to-record assignments are currently done using a generic function which interprets a (compiled) description of the record type and iterates over its fields. The next major release of Orca will include customized record-to-record assignment functions constructed by the compiler. We expect that this will yield a small, but significant, improvement in the speed of some programs.

Finally, the current Orca compiler does not support such things as in-lining code or conditional compilation. In the Kece program, this forced[2] the programmer to expand the word-matching routine in line by hand. In the Turing Ring, several independent versions of the program with different load-balancing strategies had to be maintained, where a single source with some conditional compilation flags would have been tidier. The next major release of the Orca compiler will address these issues.

One problem which several programmers encountered was not Orca's fault, but was due to limitations in the current version of the Amoeba operating system. Because Amoeba does not support virtual memory, the generation of sub-trees by the root process in the Kece program had to be "throttled" to avoid overflowing physical memory. This made that code significantly more complex than it otherwise would have been. Similarly, the skeletonization program's initialization routine had to read a single slice of the input image at a time, and broadcast it to the worker processes before reading the next slice in order to avoid limitations on the maximum size of messages allowed in Amoeba.

4.4 Problems with the Applications

It would not be fair to finish this section without pointing out that some of the problems encountered during this work were intrinsic to the applications implemented. Termination detection in an asynchronous application such as ACP is a hard problem; the complexity and expense of the solution described in Section 3.2 would have to be incurred no matter what system was being used.

[2] "Forced" may be too strong a word—the programmer in question seemed to enjoy doing it.

Similarly, parallelizing $\alpha\beta$ search is known to be extremely difficult; the subtree polling method used to reduce parallel search overhead is just one of many less-than-satisfactory solutions available. Finally, testing the image skeletonization program was difficult because the parallel version often generated different results than its sequential counterpart. This is because the result of a sweep-and-delete algorithm is critically dependent on the order of operations; using a different sweep order in a sequential program would also have led to different answers.

5 Conclusion

Despite the criticisms above, our opinion of Orca as a result of this work is very high. We believe this work demonstrates that the shared data-object model combines the scalability and efficiency of procedural message passing with the ease of use (and of learning) which characterizes the "classic" shared-memory model. While programmers do have to concern themselves with communication and synchronization issues, the language and model encourage them to think about these things in a highly-structured manner. We expect the insights from this work to have a major impact on the future development of Orca and its descendents.

References

[1] Robert G. Babb, editor. *Programming Parallel Processors.* Addison-Wesley, 1988.

[2] H. E. Bal and I. Athanasiu. The Arc Consistency Problem: a Case Study in Parallel Programming with Shared Objects. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[3] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. on Software Engineering*, 18(3), March 1992.

[4] H. E. Bal, R. van Renesse, and A. S. Tanenbaum. Implementing Distributed Algorithms Using Remote Procedure Calls. In *Proc. AFIPS Nat. Computer Conf.*, volume 56, pages 499–506. AFIPS Press, June 1987.

[5] H. E. Bal and Robbert van Renesse. Parallel Alpha-Beta Search. In *Proc. NGI - SION Informatica Symposium: stimulerende informatica*, pages 379–385, April 1986.

[6] Henri E. Bal and M. Frans Kaashoek. Object Distribution in Orca Using Compile-Time and Run-Time Techniques. In *Proceedings of OOPSLA'93*, pages 162–177, October 1993.

[7] Henri E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3), 1989.

[8] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*. ACM Press, 1990.

[9] Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Proc. USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.

[10] N. Carriero, D. Gelernter, and J. Leichter. Distributed Data Structures in Linda. In *Proc. 13th ACM Symposium on the Principles of Programming Languages*, pages 236–42, January 1986.

[11] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4), April 1989.

[12] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. North-Holland, 1992.

[13] C. A. R. Hoare. The Emperor's Old Clothes. *Communications of the ACM*, 24(2):75–83, February 1981.

[14] M. F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1992.

[15] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[16] L. Lam, S-W. Lee, and C. Y. Suen. Thinning Methodologies—A Comprehensive Survey. *IEEE Transactiosn on Pattern Analysis and Machine Intelligence*, 14(9):869–84, September 1992.

[17] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[18] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[19] R. A. Marsland and F. Popowich. Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452, July 1985.

[20] D. Mollison. Spatial Contact Models for Ecological and Epidemic Spread. *Proceedings of the Royal Statistical Society*, B(39(3)):283–326, 1977.

[21] D. Mosberger. Memory Consistency Models. *ACM Operating Systems Research Letters*, pages 18–26, January 1993.

[22] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba—A Distributed Operating System for the 1990s. *IEEE Computer*, 1990.

[23] A. Newell, J. C. Shaw, and H. A. Simon. Chess-Playing Programs and the Problem of Complexity. *IBM Journal of Research and Development*, 2:320–355, 1958.

[24] Judea Pearl. The Solution for the Branching Factor of the Alpha-Betha Pruning Algorithm and its Optimality. *Communications of the ACM*, 25(8):559–564, August 1982.

[25] Jonathan Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[26] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[27] M. A. Smith. Using Massively-Parallel Supercomputers to Model Stochastic Spatial Predator-Prey Systems. *Ecological Modelling*, 58:347–367, April 1991.

[28] M. A. Smith. Dynamic Load-balancing Strategies for Data Parallel Implementations of Reaction-Evolution-Migration Systems. Technical Report 93-2, Edinburgh Parallel Computing Centre, University of Edinburgh, 1993.

[29] Duane Szafron and Jonathan Schaeffer. Experimentally Assessing the Usability of Parallel Programming Systems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, April 1994.

[30] A. A. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A Practical Toolkit for Making Portable Compilers. *Communications of the ACM*, 26(9):654–60, September 1983.

[31] A. M. Turing. The Chemical Basis of Morphogenesis. *Philosophical Transactions of the Royal Society of London*, B(237):37–42, 1952.

[32] P. S. P. Wang and Y. Y. Zhang. A Fast and Flexible Thinning Algorithm. *IEEE Transactions on Computers*, 38(5):741–745, May 1989.

[33] Reinhold P. Weicker. An Overview of Common Benchmarks. *IEEE Computer*, December 1990.

[34] M. F. Wheeler, editor. *Numerical Simulation in Oil Recovery*. Springer-Verlag, 1991.

[35] Gregory V. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, April 1994.

[36] Terry Winograd. *Language as a Cognitive Process (Volume 1: Syntax)*. Addison-Wesley, 1983.