IMPLEMENTING DISTRIBUTED ALGORITHMS USING REMOTE PROCEDURE CALLS

H.E. Bal *
R. van Renesse
A.S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

Remote Procedure Call (RPC) is a simple yet powerful primitive for communication and synchronization between distributed processes. A problem with RPC is the fact that it tends to decrease the amount of parallelism in the application, due to its synchronous nature. This paper shows how light-weight processes can be used to circumvent this problem. The combination of blocking RPC calls and light-weight processes provides both simple semantics and efficient exploitation of parallelism.

The communication primitive of the Amoeba Distributed Operating System is based on this combination. We will describe how two important classes of algorithms, branch and bound and alpha-beta search, can be run in a parallel way using this primitive. The results of some experiments comparing these algorithms on a single processor and on Amoeba are also discussed.

1. INTRODUCTION

As computing technology advances, it becomes increasingly difficult and expensive to make computers faster by just increasing the speed of the chips. Electrical signals in copper wire travel at 2/3 the speed of light, or about 20 cm/nanosecond, so very fast computers must be very small, which leads to severe heat dissipation problems among other things. The obvious solution is to harness together a large number of moderately fast computers to achieve the same computing power as one very fast computer, but at a fraction of the cost.

Many ways of organizing multiple processors into distributed systems have been proposed. At one end of the spectrum are the *loosely-coupled systems* consisting of a number of independent computers, each with its own operating system and users, exchanging files and mail over a public data network. At the other end of the spectrum are *tightly-coupled systems* with multiple processors on the same bus and sharing a common memory. In between are systems consisting of mini- or microcomputers communicating over a fast local network and all running a single, system-wide operating system. We have used a system in the latter category as a testbed for the implementation of some distributed algorithms.

In this paper we will briefly describe this system, called Amoeba, and its communication primitive, which is essentially a Remote Procedure Call (RPC). The main intent of the paper is to

* This research was sponsored in part by the Netherlands Organization for Pure Scientific Research (Z.W.O.) under project number 125-30-10

describe how some fairly complex distributed algorithms can be implemented on such a system using RPC. Measurements on the performances of these algorithms are presented in the last section.

2. THE AMOEBA SYSTEM

The Amoeba Distributed Operating System [Mullender and Tanenbaum 1985; Tanenbaum and Mullender 1981; Mullender and Tanenbaum 1984, 1986; Tanenbaum et al. 1986] consists of a collection of (possibly different) processors, each with its own local memory, which communicate over a local network. Currently, we use mainly Motorola 68010 processors connected by a 10 Mbps token ring (Pronet), although Amoeba also runs on the VAX, NS16032, PDP-11 and IBM-PC. Amoeba is based on the client-server model [Tanenbaum and Van Renesse 1985]. The system is composed of four basic components. First, each user has a personal workstation, to be used for editing on a bit-map graphics terminal and other activities that require dedicated computing power for interactive work. Second, there is a pool of processors that can be dynamically allocated to users as needed. For example, a user who wants to run a 5-pass compiler might be allocated 5 pool processors for the duration of the compilation, to allow the passes to run largely in parallel. Third, there are specialized servers: file servers, directory servers, process servers, bank servers (for accounting) etc. Fourth, there are gateways that connect the system to similar systems elsewhere.

The Amoeba communication primitive is based on Remote Procedure Call (RPC) [Birrell and Nelson 1984; Nelson 1981]. RPC is a mechanism for communication across a network. It resembles a normal procedure call. Amoeba uses a simple form of RPC: the client sends a request to any server that is willing to offer a certain service and some server sends a response back. RPC has the advantage of simple semantics, similar to the procedure calls with which every programmer is familiar. It is a higher level construct than asynchronous message passing, so it is potentially easier to use.

One problem with RPC is the fact that the caller (client) is blocked during the call, so a separate mechanism is needed to obtain parallelism. In Amoeba, a process (or *cluster*) consists of one or more light-weight processes called *tasks*. Tasks share a common address space and run in parallel. While a task is blocked in an RPC other tasks in its cluster may run if they have work to do. The combination of blocking RPC calls and light-weight processes provides both simple semantics and efficient exploitation of parallelism. In the following sections we will describe how they can be used together to implement parallel algorithms for branch-and-bound and alpha-beta search.

3. PARALLEL BRANCH AND BOUND USING RPC

The branch-and-bound method is a technique for solving a large class of combinatorial optimization problems. It has been applied to Integer Programming, Machine Scheduling problems, the Traveling Salesman Problem, and many others [Lawler and Wood 1966]. We have chosen to implement the Traveling Salesman Problem (TSP), in which it is desired to find the shortest route for a salesman to visit each of the n cities in his territory exactly once.

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. Fig. 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 London - Amsterdam - Paris - Washington.

A *bounding rule* avoids searching the whole tree. For TSP, the bounding rule is simple. If boxwid = 0.25 boxht = 0.25 B1: box "L" B2: [movewid = 2 B21: box "A"; move ; B22: box "P"; move ; B23: box "W"] with .n at B1.s-(0,.5) B3: [movewid = .75 B31: box "P"; move ; B32: box "W"; move ; B33: box "R"; move; B34: box "W"; move; B35: box "A"; move; B36: box"P"] with .n at B2.s-(0,.5) B4: [movewid=.75 B41: box "W"; move ; B42: box "P"; move ; B43: box "W"; move; B44: box "R"; move; B45: box "P"; move; B46: box "A"] with .n at B3.s-(0,.5) line from B2.B21.n to B1.w line from B2.B22.n to B1.s line from B2.B23.n to B1.e line from B3.B31.n to B2.B21.w line from B3.B32.n to B2.B21.e line from B3.B33.n to B2.B22.w line from B3.B34.n to B2.B22.e line from B3.B35.n to B2.B23.w line from B3.B36.n to B2.B23.e line from B4.B41.n to B3.B31.s line from B4.B42.n to B3.B32.s line from B4.B43.n to B3.B33.s line from B4.B44.n to B3.B34.s line from B4.B45.n to B3.B35.s line from B4.B46.n to B3.B36.s

Fig. 1. Tree of 4-city Traveling Salesman Proplem for London, Amsterdam, Paris, and Washington.

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.

Parallelism in a branch-and-bound algorithm is obtained by searching parts of the tree in parallel. If enough processors were available, a new processor could be allocated to every node of the tree. Every processor would select the best partial path from its children and report the result back to its parent. If there are N cities, this approach would require O(N!) processors. More realistically, the work has to be divided among the available processors. In our model, each processor starts at the node given to it and generates the complete partial tree reachable from that node down to *depth* levels. Each time the processor generates a node at level *depth* it hands out this node to a subcontractor for further evaluation. These evaluations and the generation of the partial tree occur in parallel. Figure 2 shows how the tree of Figure 1 can be searched, using a 2-level processor hierarchy (i.e., a subcontractor has no subcontractors itself).

boxwid = 0.25 boxht = 0.25 B1: box "L" B2: [movewid = 2 B21: box "A"; move ; B22: box "P"; move ; B23: box "W"] with .n at B1.s-(0,.5) B5: [movewid = 2 B51: box "A"; move ; B52: box "P"; move ; B53: box "W"] with .n at B2.s-(0,.5) B3: [movewid = .75 B31: box "P"; move ; B32: box "W"; move ; B33: box "R"; move; B34: box "W"; move; B35: box "A"; move; B36: box"P"] with .n at B5.s-(0,.5) B4: [movewid=.75 B41: box "W"; move ; B42: box "P"; move ; B43: box "W"; move; B44: box "R"; move; B45: box "P"; move; B46: box "A"] with .n at B3.s-(0,.5) line from B2.B21.n to B1.w line from B2.B22.n to B1.s line from B2.B23.n to B1.e

arrow from B2.B21.s - (0,.1) to B5.B51.n + (0,.1) arrow from B2.B22.s - (0,.1) to B5.B52.n + (0,.1) arrow from B2.B23.s - (0,.1) to B5.B53.n + (0,.1)

line from B3.B31.n to B5.B51.w line from B3.B32.n to B5.B51.e line from B3.B33.n to B5.B52.w line from B3.B34.n to B5.B52.e line from B3.B35.n to B5.B53.w line from B3.B36.n to B5.B53.e

line from B4.B41.n to B3.B31.s line from B4.B42.n to B3.B32.s line from B4.B43.n to B3.B33.s line from B4.B44.n to B3.B34.s line from B4.B45.n to B3.B35.s line from B4.B46.n to B3.B36.s

Fig. 2. Example of a distributed tree search

In Figure 2, the processor that traverses the top part of the tree (the root processor) searches one level. It splits off three subtrees, each of depth two, which are traversed in parallel by the subcontractors. This algorithm is shown in Figure 3. The algorithm sets the global variable

```
procedure traverse(node,depth,length);
begin
    { 'node' is a node of the search tree. It contains
      a list of the cities on the current partial tour.
      'length' is the length of the partial path so far.
      'depth' is the number of levels to be searched
      before the rest of the tree should be handed
      out to a subcontractor }
    if length < minimum then
    begin  { if length >= minimum skip this node }
        if 'node' is a leaf then
            minimum := length;
        else if depth = 0 then
            hand out subtree rooted at 'node'
            to a subcontractor;
        else
            for each child c of 'node' do
                traverse(c,depth−1,length+dist(node,c));
    end
end
```

Fig. 3. Tree traversal algorithm

A processor only blocks if it tries to hand out a subtree while there are no free subcontractors. Each subcontractor executes the same traversal process, with a different initial node and probably with a different initial depth. In general, a subcontractor may split up the work over even more processors, so a subcontractor may also play the role of a root processor.

The Traveling Salesman Problem has been implemented under Amoeba using the algorithm described above. A processor playing the role of a subcontractor can be viewed as an Amoeba *server*. The service it offers is the evaluation of a TSP subtree. Each server repeatedly waits for some work, performs the work, and returns the result. A processor playing the role of a root processor is a *client*.

The 'handing out of work' is implemented using Remote Procedure Calls. As stated before, a problem with RPC is the fact that the caller (client) is blocked during the call. Therefore, the client cluster is split up into several tasks (see Figure 4). A cluster C_p running on processor p contains one *manager* task M_p that performs the tree traversal. If the cluster has N subcontractors, it also contains N *agent* tasks $A_{p,1}$ .. $A_{p,N}$. An agent $A_{p,j}$ controls the communication with subcontractor j.

After the manager task M_p receives a subtree T to evaluate, it starts the tree traversal of Figure 4. When it finds a subtree that has to be subcontracted out, it tries to find a free agent, say $A_{p,j}$. The agent $A_{p,j}$ sends the work to be done to the manager M_j of subcontractor j, using an RPC with a partial path and the current best solution as parameters. This manager M_j starts exe-

B1: [boxht = 1 boxwid = .75 movewid = .1 B11: box "manager" move B12: box "agent 1" move B13: box "agent 2" move B14: box invis "..." wid 2*boxwid move B15: box "agent n"] box ht B1.ht+.2 wid B1.wid+0.2 at B1 boxht = 1 boxwid = .75 movewid = .1 B2: box "server 1" with .n at B1.B12.s - (0,.4) move B3: box "server 2" with .n at B1.B13.s - (0,.4) move B4: box invis "..." wid 2*boxwid with .n at B1.B14.s - (0,.4) move B5: box "server n" with .n at B1.B15.s - (0,.4) arrow <-> from B1.B12.s - (0,.01) to B2.n + (0,.01) arrow <-> from B1.B13.s - (0,.01) to B3.n + (0,.01) arrow <-> from B1.B15.s - (0,.01) to B5.n + (0,.01)

Fig. 4. Process structure of the TSP program

cuting the process we describe here on processor j. When M_j finishes the evaluation of the subtree, it returns the result to $A_{p,j}$. This agent checks if the current best solution has to be updated, and then becomes available again for the next request from M_p. In the mean time, the manager M_p continues its tree traversal and eagerly tries to find new work to distribute. The entire client cluster only blocks if the manager tries to deal out work while all agents (and thus all subcontractors) are engaged.

This implementation fully utilizes the parallelism present in the algorithm. Furthermore, the implementation is highly flexible. It uses depth-first search, but it can easily be adapted to other strategies, such as breadth-first or best-first.

4. PARALLEL ALPHA-BETA SEARCH USING RPC

Alpha-beta search is an efficient method for searching game trees for two-person, zero-sum games. A node in such a game tree corresponds to a position in the game. Each node has one branch for every possible move in that position. A value associated with the node indicates how good that position is for the player who is about to move (let's assume this player is 'white'). At even levels of the tree, this value is the *maximum* of the values of its children; at odd levels it is the *minimum*, as the search algorithm assumes black will choose the move that is least profitable for white. Most implementations negate the values of the odd level nodes, so the values are maximized at all levels.

The alpha-beta algorithm finds the best move in the current position, searching only part of tree. It uses a *search window* (alpha,beta) and prunes positions whose values fall outside this window. The algorithm is shown in Figure 5.

Alpha-beta search differs significantly from branch-and-bound in the way the best solution is constructed. A branch-and-bound program (potentially) updates its solution every time a processor visits a leaf node (see Figure 3). That processor only needs to know the current best solution and the value associated with the leaf. An alpha-beta program, on the other hand, has to *combine* the values of the leaves and the interior nodes, using the structure of the tree. Some parallel alpha-beta programs realize this by having a dedicated processor for every node (up to a certain level) that collects the results of the child processors [Finkel and Fishburn 1982]. As a disadvantage of this approach, processors associated with high level interior nodes spend most of their time waiting for their children to finish.

Our solution avoids this problem by working the other way round. The child processors compute the values for their parent nodes, so there is no need for their parent processors to wait. To do this, an *explicit* tree structure is built, containing the alpha and beta bounds at each node. The search tree is no longer just a concept, but it is actually built as a data structure. This tree is distributed over all processors, each processor containing that part of the tree it is working on.

The process structure of alpha-beta is somewhat simpler than that of TSP, because the

```
function AlphaBeta(node,depth,alpha,beta): integer;
begin
  if depth = 0 then
      alpha := evaluation(node)
  else
    for each child c of 'node' do
    begin
        r := −AlphaBeta(c,depth−1,−beta,−alpha)
        if r > alpha then
        begin
            alpha := r;
            if alpha >= beta then
                exit loop;   { pruning }
        end
    end
  AlphaBeta := alpha
end
```

Fig. 5. Sequential alpha-beta algorithm

shared tree can be used for synchronization within the client cluster. Hence there is no need for a manager task. The client cluster contains as many tasks as there are subcontractors (see Figure 6).

Each task essentially executes the sequential alpha-beta algorithm of Figure 5. To keep other tasks from evaluating the same positions, each task leaves a trace of what is has done already by building the tree. Each task does a depth-first search in the tree until it either finds an unvisited node or it decides that the subtree rooted at the current node should be evaluated by another processor. In the first case it generates all children of the unvisited node and continues with the first child node. In the second case it sends the node to a subcontractor using RPC and waits for the result.

After a subtree has been evaluated (whether local or remote) its result should be used to update the alpha and beta values of other nodes in the tree. This is illustrated in Figure 7.
In Figure 7(a), the subtrees rooted at nodes 3, 4, 6, and 7 have been evaluated. After the subtree rooted at node 8 has been evaluated the value of the parent of node 8 (node 5) is updated (as 20 > 15). This is shown in Figure 7(b). Furthermore, the evaluation of the subtree rooted at 5 has now been completed. As its final value (-20) is the highest value of level 1, the value of node 1 is updated too.

After the value of a node has been improved this new value can be used as a tighter alpha bound for its children. Each child can use this new alpha value as a tighter beta bound for its own children, and so on. So new values are propagated down the tree, to ensure each node uses the smallest possible alpha-beta window. In principle, new bounds can even be propagated across processor boundaries. However, this would also increase the communication overhead. We have not yet experimented with this kind of propagation.

```
B1: [
        boxht = 1
        boxwid = .75
        movewid = .1
        B12: box "task 1"
        move
        B13: box "task 2"
        move
        B14: box invis "..." wid 2*boxwid
        move
        B15: box "task n"
        line from B13.n + (0,1) to B13.n + (-.2,.75)
        line from B13.n + (0,1) to B13.n + (.2,.75)
        line from B13.n + (-.2,.75) to B13.n + (-.3,.5)
        line from B13.n + (-.2,.75) to B13.n + (-.05,.5)
        line from B13.n + (.2,.75) to B13.n + (.05,.5)
        line from B13.n + (.2,.75) to B13.n + (.3,.5)
        "shared tree" at B14.n + (-0.2,0.5)
]
box ht B1.ht+.2 wid B1.wid+0.2 at B1
boxht = 1
boxwid = .75
movewid = .1
B2: box "server 1" with .n at B1.B12.s - (0,.4)
move
B3: box "server 2" with .n at B1.B13.s - (0,.4)
move
B4: box invis "..." wid 2*boxwid with .n at B1.B14.s - (0,.4)
move
B5: box "server n" with .n at B1.B15.s - (0,.4)
arrow <-> from B1.B12.s - (0,.01) to B2.n + (0,.01)
arrow <-> from B1.B13.s - (0,.01) to B3.n + (0,.01)
arrow <-> from B1.B15.s - (0,.01) to B5.n + (0,.01)
```

Fig. 6. Process structure of the alpha-beta program

5. DISCUSSION

We have done some measurements on the TSP and the alpha-beta programs. The hardware used was a collection of 10 MHz 68010 CPU's connected by a 10 Mpbs token ring. For each program, we ran both a sequential (single processor) version and a parallel (multi-processor) version. For simplicity, the parallel versions use only a 2-level processor hierarchy. They use one processor for the client process and a varying number of processors for the servers.

The depths of the subtrees are important parameters of the TSP algorithm. If the client processor distributes work at a too high level, the effectiveness of pruning will be severely weakened. For example, if it traverses just one level, then the best solution in the leftmost branch of the tree cannot be used as a bound in its neighbor branch, as these branches are searched simultaneously. Increasing the depth of the root subtree will decrease this effect, at the cost of more communication between the root processor and its subcontractors. To achieve high performance, a good compromise has to be found. For an 11-city problem we found the optimal search depth of the client to be three levels. The results for an 11-city problem using this search depth are shown in Fig. 8.
The last entry in the table shows the speedup over the 1-server version. With 7 processors (1 client and 6 servers) a 5-fold speedup over the sequential program is achieved. Note that with only one server, there is still some parallelism, as the client can find the next subtree to hand out,

boxwid = 0.25 boxht = 0.25 B1: box "1" "38" at B1.ne + (.03,.03) move 3 B2: box "2" "28" at B2.ne + (.03,.03) B3: [B31: box "2" "-23" at B31.ne + (.03,.03) move 1 B32: box "5" "-15" at B32.ne + (.03,.03)] with .n at B1.s - (0,.5) B4: [B41: box "3" "25" at B41.ne + (.03,.03) move .4 B42: box "4" "38" at B41.ne + (.03,.03)] with .n at B3.B31.s - (0,.5) B5: [B51: box "6" "15" at B51.ne + (.03,.03) move .2 B52: box "7" "9" at B52.ne + (.03,.03) move .2 B53: box "8"] with .n at B3.B32.s - (0,.5) B6: [B61: box "2" "-38" at B61.ne + (.03,.03) move 1 B62: box "5" "-28" at B62.ne + (.03,.03)] with .n at B2.s - (0,.5) B7: [B71: box "3" "25" at B71.ne + (.03,.03) move .4 B72: box "4" "38" at B72.ne + (.03,.03)] with .n at B6.B61.s - (0,.5) B8: [B81: box "6" "15" at B81.ne + (.03,.03) move .2 B82: box "7" "9" at B82.ne + (.03,.03) move .2 B83: box "8" "28" at B83.ne + (.03,.03)] with .n at B6.B62.s - (0,.5) line from B1.w to B3.B31.n line from B1.e to B3.B32.n

line from B2.w to B6.B61.n line from B2.e to B6.B62.n

line from B3.B31.w to B4.B41.n line from B3.B31.e to B4.B42.n

line from B3.B32.w to B5.B51.n line from B3.B32.s to B5.B52.n line from B3.B32.e to B5.B53.n

line from B6.B61.w to B7.B71.n line from B6.B61.e to B7.B72.n

line from B6.B62.w to B8.B81.n line from B6.B62.s to B8.B82.n line from B6.B62.e to B8.B83.n

"7(a)" at B5.B51.s + (0, -.5) "7(b)" at B8.B81.s + (0, -.5)

Fig. 7. Example of alpha-beta search

version	time(secs)	speedup
sequential	637.2	
1 server	548.1	1
2 servers	309.7	1.77
3 servers	218.2	2.51
4 servers	171.7	3.19
5 servers	141.5	3.87
6 servers	124.2	4.41

Fig. 8. Table I: results for 11-city traveling salesman problem.

while the server is working on the previous subtree.

To measure the performance of the alpha-beta algorithm, we implemented the game of *Othello*, using this algorithm. Fig. 9 shows the time to evaluate a position, averaged over five different positions with a fan-out (number of moves) of approximately fifteen. The depth of the search tree was four plies. As for TSP, the division of labour between the client and the servers is important. For the parallel versions the client searched three plies, the servers searched one ply.

The results show that the speedup achieved is significantly worse for alpha-beta search than for TSP. The main reason is that alpha-beta search suffers more from the decrease in pruning efficiency than TSP. The third entry in table 2 shows the number of leaves visited by alpha-beta (i.e.,

version	time(secs)	speedup	#evaluations	search overhead
sequential	266.9		2670	1
1 server	324.6	1	2670	1
2 servers	196.2	1.65	3925	1.47
3 servers	153.3	2.12	4732	1.77
4 servers	125.1	2.59	5676	2.13
5 servers	114.0	2.84	6424	2.40
6 servers	111.5	2.91	6719	2.51

Fig. 9. Table II: results for Othello implementation of alpha-beta search

the number of static evaluations). This number is a yardstick for the total amount of work done. The last entry shows the search overhead over the sequential version.

Our implementations of TSP and alpha-beta search have been deliberately kept simple initially, as we implemented them just to gain some experience with programming using RPC and light-weight processes. However, our results indicate that the primitives offered by Amoeba are sufficiently general for more advanced implementations.

6. REFERENCES

Birrell, A. D. and Nelson, B. J., Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Finkel, R. A. and Fishburn, J. P., Parallelism in Alpha-Beta Search, *Artificial Intelligence*, Vol. 19, pp. 89-106, 1982.

Lawler, E. L. and Wood, D. E., Branch-and-bound Methods: a survey, *Operations Research*, Vol. 14, No. 4, pp. 699-719, July 1966.

Mullender, S. J. and Tanenbaum, A. S., Protection and Resource Control in Distributed Operating Systems, *Computer Networks*, Vol. 8, No. 5, 1984.

Mullender, S. J. and Tanenbaum, A. S., A Distributed File Service Based on Optimistic Concurrency Control, *Proc. 10th ACM Symposium on Operating Systems Principles*, pp. 51-62, Rosario Resort, Orcas Island, Washington, Dec. 1985.

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

Nelson, B. J., Remote Procedure Call, CMU-CS-81-119, Carnegie-Mellon University, May 1981.

Tanenbaum, A. S. and Mullender, S. J., An Overview of the Amoeba Distributed Operating System, *Operating Syst. Rev.*, Vol. 15, No. 3, pp. 51-64, July 1981.

Tanenbaum, A. S., Mullender, S. J., and Van Renesse, R., Using Sparse Capabilities in a Distributed Operating System, *Proc. 6th Int. Conf. on Distributed Computing Systems*, pp. 558-563, Cambridge, Massachusetts, May 1986.

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