

Parallel Retrograde Analysis on a Distributed System

Henri Bal* Victor Allis

Dept. of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam, The Netherlands
email: bal@cs.vu.nl
phone: +31 20 444 7733

Abstract

Retrograde Analysis (RA) is an AI search technique used to compute endgame databases, which contain optimal solutions for part of the search space of a game. RA has been applied successfully to several games, but its usefulness is restricted by the huge amount of CPU time and internal memory it requires.

We present a parallel distributed algorithm for RA that addresses these problems. RA is hard to parallelize efficiently, because the communication overhead potentially is enormous. We show that the overhead can be reduced drastically using message combining.

We implemented the algorithm on an Ethernet-based distributed system. For one example game (awari), we have computed a large database in 50 minutes on 64 processors, whereas one machine took 40 hours (a speedup of 48). An even larger database (computed in half a day) would have required 400 MByte of internal memory on a uniprocessor and would compute for weeks.

Keywords: game-tree search, retrograde analysis, distributed systems.

1 Introduction

Retrograde Analysis (RA) is an important search technique developed within the field of Artificial Intelligence. It is applicable to search spaces which can be completely enumerated within the memory of a computer system. Given such a complete enumeration, RA first marks all end positions (e.g., checkmate), and then, by making *unmoves* from the end positions works its way back to the positions farthest from the end positions, on the way determining the game-theoretical value of *all* positions in the search space. Thus, RA searches bottom-up, whereas most other search algorithms, such as *Breadth-First Search*, *Depth-First Search*, *A**, and *Alpha-Beta Search* proceed top-down.

The main advantage of RA is that for each position in the state space the optimal solution is determined, whereas application of a top-down search technique only provides the optimal solution for a single starting position, and the positions on the solution path.

Retrograde analysis has been used to create *endgame databases* for several two-player games. Such databases contain the game-theoretical value for all positions in certain kinds of

*This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

endgames. (An endgame is the final phase of a game, in which the board contains few pieces.) The database needs to be computed only once, and can then be used during subsequent plays. Below, we give three examples of such databases.

First, Ken Thompson created chess endgame databases[13]. All endgame databases for positions with at most five pieces are available on CD-ROM. Also, a large number of six-piece endgame databases have been investigated [10]. The creation of these omniscient databases has even led to changes in the rules of chess. As another example, endgame databases are the main component of the checkers program *Chinook* [9]. *Chinook* is the reigning Man-Machine World-Champion checkers player. It has an endgame database for all interesting positions of at most eight men. It will therefore always play the best move for every position with eight or fewer pieces on the board. In addition, if the program analyses positions with more than eight pieces, it can use the database whenever its forward search leads to positions that are within the database. Another board game, nine men's morris, has been proven a draw by creating a database for all legal positions of the game [6].

The main disadvantage of RA is that it requires a huge amount of CPU time and internal memory. For each position in the search space an entry must be created in the database. The information to be stored in such an entry normally requires a few bits or bytes. For games like chess, checkers, and nine men's morris, the size of the search space grows exponentially with the number of pieces on the board. The search spaces of the largest five-piece chess endgames consist of some 1 billion entries. Even though a reduction of the search space through the exploitation of board symmetries can be obtained, the availability of internal memory is a limiting factor in the creation of these databases. The use of virtual memory does not solve the problem, as there is almost no locality in the algorithm.

The high computational cost of RA is mainly caused by several expensive operations that are applied to each entry. From the board position of the entry, the set of parent positions must be generated using unmoves. The costs of this operation depend on the complexity of the rules of the game. Also, conversions are needed between a board position and a database address. Conceptually, the database is indexed with board positions, but an RA program internally uses addresses (integers) for indexing, since these require much less memory. The program uses conversion operations (similar to perfect hash functions) to calculate the address from a board position and vice versa. As a result of all these operations, the creation of a typical endgame database of, say, 50 million entries, takes in the order of several hours of CPU time on today's workstations.

In this paper we describe a parallel distributed retrograde analysis algorithm. The algorithm is intended to be implemented on a distributed-memory system, such as a collection of workstations connected by a network. The parallelism ensures that more CPU cycles are available to execute the costly conversion and unmove operations, thus allowing us to perform the RA faster. The distribution combines local memories of a large number of machines, making more internal memory available, thus allowing us to apply RA to larger search spaces. It is of course also possible to use a single machine with a very large main memory, but this may not be cost-effective, as shown by Wood and Hill [14].

In the course of our research we have found that parallel distributed RA is a very hard problem, mainly because of the large communication overhead. As an example, let us assume that we perform an RA of a search space of 50 million entries on 10 processors, each storing 10% of the database entries. Assuming a total lack of locality, 90% of all communication between two database entries results in a message between two processors. For a typical application domain, where in each position on average 5 unmoves are possible, the number of messages

sent during the calculation of the database is roughly $50,000,000 \times 5 \times 0.9 = 225,000,000$, or 22,500,000 messages per processor. Assuming that each message requires 1 msec, each processor must spend approximately 22,500 seconds (about 6 hours) just on communication. A sequential implementation is likely to be faster. Other problems to be solved include load balancing, synchronization, and distributed termination.

This paper presents a parallel RA algorithm that optimizes communication and load balancing. The paper also presents measurements for one example game (awari), which show that good performance can be achieved with our approach. For a 17-stone awari database, for example, we achieve a speedup of 48 on 64 processors connected by an Ethernet. The largest awari database we have computed so far is the 21-stone database, which comprises over 354 million entries. It took less than 12 hours to compute the database on 60 processors. We have not been able to compute this database on a single machine. On one machine, the program would require 400 MByte of internal memory and would compute for weeks.

The paper is organized as follows. Section 2 describes the sequential RA algorithm. In Section 3, we present our parallel RA algorithm, based on this sequential algorithm. The implementation and performance of the parallel RA algorithm on a distributed system are discussed in Sections 4 and 5. In Section 6 we discuss our results in the context of related research. Finally, the Appendix contains the pseudo-code for our parallel algorithm.

2 Sequential Retrograde Analysis

In this section we briefly describe how to do retrograde analysis sequentially, using an algorithm similar to the one in [1]. As RA can be applied to many different types of problems, each leading to variations on the main theme, we restrict ourselves to a broad, but well-defined class of problems. The assumptions which define this class of problems are presented in Section 2.1. Next, in Section 2.2, we present the RA algorithm.

2.1 Assumptions

We assume that we create an endgame database for a two-player zero-sum game with perfect information, and that the value of each game is an integer between $-N$ and N . The value of a game is based on the value of the end position of the game, and points the players may have accumulated during the game (by making moves which are worth a specified number of points). In awari and go, a player can score points during the game, by capturing stones. In chess and checkers, the value of the game is based solely on the end position; the value can be a win (1) draw (0), or loss(-1), so $N = 1$.

For each legal game position we would like to calculate the best score obtainable for the player to move. For non-end positions, the score is based on the values of its children and on the values of the moves leading to those children. Let us suppose that in the current position, P , the only legal move leads to a position, Q , with value k . As in Q the opponent is to move, this means that the opponent can win in Q by k points. From the perspective of the player to move in P , this is equivalent to a value of $-k$. If the move from P to Q scores x points, then the total value obtainable through this move is $-k + x$. Thus, when knowing the values of all children of a position, and the values of all moves leading to those children, we can calculate the value of the position. Furthermore, we assume that if the game gets into a cycle (repetition of positions), the game is a draw (i.e., both players are awarded 0 points).

2.2 The sequential algorithm

The state space searched by RA is a directed, possibly cyclic graph. An edge from node P to node Q in the graph represents a move from position P to Q , which is also named an unmove from position Q to position P . We call P a *parent* of Q , and Q a *child* of P . Note that a child can have many parents, since there may be many different positions P from which a move to position Q can be made.

From some nodes no moves are possible. In each game, the value of such nodes, which are called *end nodes*, can be determined immediately from the rules of the game. Examples are checkmate (-1 for the player to move) and stalemate (0) in chess. For each non-end node, the value can be calculated once the values of all children are known, or earlier, if the node can obtain the highest value possible through one of its children. In the latter case, there is no need to determine the values of the other children.

To implement RA, we store two numbers for each database entry (see Figure 4 in the Appendix):

- the number of children of which the value is currently unknown, called *UnknownChildren*, and
- the best value so far, obtainable through one of the children whose value is known, called *BestValue*.

Thus, if *UnknownChildren* has dropped to 0, *BestValue* is the final value of the position. Alternatively, once *BestValue* reaches a value which cannot be surpassed (e.g., value $+1$ in chess), then, regardless of the value of *UnknownChildren*, *BestValue* is the final value of the position.

Once an entry has obtained its final value, it must notify all its parents, so that it contributes in determining the value of those parents. To ensure that all children notify their parents exactly once, each entry contains a third field called *Status*, which can have one of three values: *Unknown*, *Known*, or *Propagated*. *Unknown* indicates that the value of the position is not yet final. *Known* indicates that the value is final, but its parents have not yet been notified. *Propagated* means that the value is final, and the parents have been notified.

The basic RA algorithm is now easy to describe. The algorithm consists of three phases. First, the database is traversed once to initialize it. For each node we determine the number of children. If the node has no children (an end node), we determine its value using domain knowledge and set *BestValue* to this value. The *Status* of the node is set to *Known*. Otherwise, the number of children is stored in *UnknownChildren*, *BestValue* is set to $-N$, and *Status* is set to *Unknown*.

After this initialization, the second phase begins, which does multiple traversals through the database. The database is traversed and each node whose value has been determined, but has not yet informed its parents, notifies all its parents of its value. Each parent subtracts 1 from *UnknownChildren* and updates *BestValue* if it is improved by the value of the child. In case the parent itself now reaches its final value, it recursively informs each of its own parents.

After this second pass through the database, all positions which should obtain the values $-N$ or N have already obtained that value. For a proof of this statement, see [1]. Thus, only positions which have values in the range $-N+1$ to $N-1$ may at this point have an unknown value. By making another pass through the database, using the knowledge that none of the positions with unknown value can reach a value higher than $N-1$, another set of positions

can obtain their final value. After the iteration is finished, the upper bound on possible values is again decremented, and the process is repeated, until the upper bound is decremented to 0. It can be proven that all positions which at that point have not yet established their final value are draws, and should obtain the value 0 [1]. The third phase thus sets all remaining positions to 0.

Thus, an RA for a game where each position obtains a value between $-N$ and N , consists of an initialization pass, N iterations in which parents are recursively notified, and a final pass in which the remaining entries are set to 0.

3 A Parallel Retrograde Analysis Algorithm

We now present our parallel RA algorithm, based on the sequential algorithm outlined above. We first give an overview of the algorithm, and then describe the algorithm itself and some implementation issues. The Appendix contains the pseudo-code for the algorithm.

3.1 Overview

We parallelize retrograde analysis by partitioning the elements of the database among the available processors. Each processor is responsible for computing the value of the entries assigned to it. The value of an entry is updated whenever new information about the entry is obtained from its children. As soon as the final value of an entry has been determined, this value is sent to all processors containing a parent of the entry. These processors use this value to update the values of the parents. We call such messages *update messages*.

A major problem with this algorithm is the communication overhead due to the update messages. Each entry obtains its final value once and then has to inform all its parents. As explained in Section 1, this may lead to a very large number of messages and thus to poor performance.

The key idea in solving this problem is the observation that a processor that sends the final value of an entry to its parent does not depend on the result of this operation. Also, the order in which the values arrive at the receiving processors does not matter. As a result, it is possible to defer sending the update messages.

All outgoing update messages are therefore buffered at the sending processor. By the time the messages are actually sent, they can usually be combined with other messages for the same destination processor. This *message combining* optimization results in fewer (but larger) messages. On most networks this is far more efficient, since messages have a high start-up cost. As we will see, this optimization drastically reduces the communication overhead. In addition, the actual message is sent asynchronously, so the process that performs the computations can continue while the message is transmitted over the network and handled by the receiving processor.

3.2 Description of the Parallel Algorithm

Each processor in our algorithm executes the same code on different parts of the database. The data structure used for the database is the same as for the sequential algorithm, except that each processor contains only a part of the entire array. The processors also contain buffers for sending and receiving update messages, shown in Figure 5 (in the Appendix).

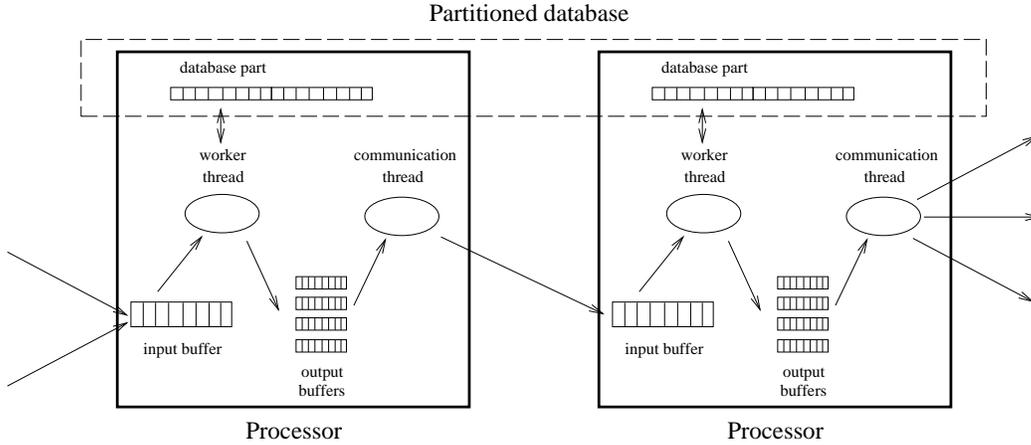


Figure 1: Overall structure of the parallel algorithm.

An element of a buffer represents one *update* message and specifies the address of a parent entry and the final value of the child that is sending the message. Each processor contains one input buffer containing messages sent to it and one output buffer for every other processor.

Every processor uses two threads. A *worker thread* performs the actual computations. Whenever this thread has determined the final value for an entry, it generates messages for the parents of this entry and stores them in the appropriate output buffers. A second thread, the *communication thread* repeatedly copies messages from these buffers over the network to the input buffers of other processors, where they will be picked up by the worker threads. The overall structure of the parallel algorithm is illustrated in Figure 1. Below, we explain the algorithms for the worker and communication threads in more detail.

A worker thread (see Figure 7) first initializes all entries in its part of the database, as discussed in Section 2.2. Next, it performs two tasks:

- It traverses the local partition of the database, and checks for each entry whether its final value has been determined, as explained in Section 2.2.
- It processes the messages in the input buffer.

These two tasks are interleaved, but to avoid congestion in the input buffer, any messages currently in the buffer are always serviced before processing the next entry in the local partition.

Each message in the input buffer contains the final value of a child of an entry stored on the current processor (see Figure 8). To process such a message, the worker thread checks if the new value is better than the entry's current value, and if so, updates the field *BestValue* (see Figure 9). Furthermore, the worker thread subtracts one from *UnknownChildren*. If all children of an entry have now reported their final value or if no higher score can be attained the final value for the entry has been determined.

Whenever the worker thread has determined the final value of an entry (either while processing the local part of the database or while processing incoming messages), it needs to generate outgoing messages. The worker thread determines the parents of the entry by calling the game's *unmove* function (see Figure 10). For each parent, the worker adds one message to the appropriate output buffer. The value field in the message is determined by the value

of the entry and the points scored by the move from the parent to the entry (as explained in Section 2.1). If the parent entry is stored on the same processor, the buffering mechanism is avoided and the worker thread immediately updates the parent entry itself.

The final step of the worker thread is to make one more pass through the local partition of the database and set all entries whose values have not been determined yet to zero. This final pass was explained in Section 2.2.

The algorithm for the communication thread (Figure 11) is simple. The thread repeatedly selects a non-empty output buffer and then sends all data in the buffer to the input buffer on the destination processor, using a single message.

3.3 Implementation Issues

Important implementation issues are when to send messages (i.e., when to activate the communication thread) and which buffer to select. The optimal strategy is game-dependent. In general, if messages are sent too soon, the algorithm will send many small messages, resulting in a high communication overhead. On the other hand, if messages are sent too late, the receiving processors may run out of work, resulting in wasted CPU time. Another issue is how to partition the database among the available processors, which has an impact on load balancing. These issues will be discussed in Section 5.2.

The algorithm uses processes that repeatedly generate work for each other. Each update message may result in new work, since the value in the message may be used to establish the final value of the parent entry. Therefore, it is difficult to determine when the algorithm can terminate a given iteration. Termination detection is required at the end of each iteration of the algorithm, since all processes must always be working on the same iteration. We have implemented a reasonably efficient termination detection algorithm, but the description of the algorithm is beyond the scope of this paper.

Another issue we have ignored so far is deadlock. In theory, deadlock could occur if the input and output buffers get full. In this case, some worker and communication threads might block. If a cycle exists among such blocked threads, a deadlock situation might arise. To prevent this situation, we dynamically adjust the sizes of the buffers, so that threads do not have to block while putting data into a buffer.

4 Implementation on a distributed system

We have implemented the parallel Retrograde Analysis algorithm described above on a distributed system. In this section, we will first describe the hardware and software environment we used and then we discuss some important details of the implementation. Performance measurements on this system will be given in the next section.

4.1 The Distributed System

We implemented the algorithm on top of a distributed system consisting of 64 SPARC processors connected by a 10 Mbit/sec Ethernet. The system runs the Amoeba distributed operating system [12] and is used as a *processor pool* [11]. The processor pool is mainly used to run computationally intensive jobs. Each node in the pool is a single-board computer (a SPARC Classic clone), containing a 50 Mhz SPARC processor and 32 MByte of local memory,

but no peripheral devices (e.g., a keyboard or display). The system is accessible from our department's Unix workstations.

The processors in the pool are connected by several Ethernet segments. Each segment contains eight processors. The segments are connected by a low-cost (Kalpana) Ethernet switch, which forwards messages between segments with a low overhead (40 microseconds). The use of multiple segments increases the total bandwidth available, but for our application latency is more important than bandwidth. We have not been able to demonstrate any performance gain (or loss) compared to using a single segment.

The architecture of our system is similar to that of a collection of workstations on a network, which is a suitable platform for parallel processing [2]. The network we use is significantly slower than more modern technologies (e.g., ATM), so our performance results could be improved by using faster networks.

The RA program was written in Orca [3, 4]. Orca is an imperative, type-secure, parallel language. Its communication model is based on shared objects. Processes on different machines can share objects and apply operations to these objects.

The implementation of Orca on Amoeba is described in [5, 8]. Each Orca process is implemented as a thread, which is managed by the Amoeba microkernel. An operation on a remote object is implemented as a Remote Procedure Call. The Orca runtime system we use runs the RPC protocol in user space [8], on top of unreliable (IP-like) communication primitives provided by Amoeba.

4.2 Implementation Issues

The buffers used by a processor are implemented as a single Orca object, stored on that processor. The communication thread repeatedly gets data from an output buffer and puts these data in the input buffer of the destination processor. The latter operation results in communication over the network (a Remote Procedure Call), since it accesses a remote processor's object. The worker thread puts update messages into the output buffers and also periodically checks its input buffer. These operations are local. The Orca runtime system takes care of mutual exclusion synchronization of the shared objects, using Amoeba's locking primitives.

An important implementation issue is the selection of the output buffer for transmission to a remote machine. Our implementation tries to send as much data as possible in one message (to reduce communication overhead), so it always selects the buffer containing the largest number of update messages. To avoid sending small messages, it also uses a *minimum message size*, currently set to 20. If the communication thread tries to get data while all buffers contain fewer than 20 messages, the thread blocks until a buffer with enough messages is available. An exception is if the worker thread on the same processor is idle. In this case, the processor does not have anything else to do, so it sends out buffers of any size. This typically occurs at the end of an iteration.

5 Performance Measurements

We will present performance measurements of the parallel Retrograde Analysis program for an example game, awari. The rules for awari and some game-specific aspects of our program are described first. Next, we describe and analyze the performance of the awari RA program on top of the Amoeba distributed system.

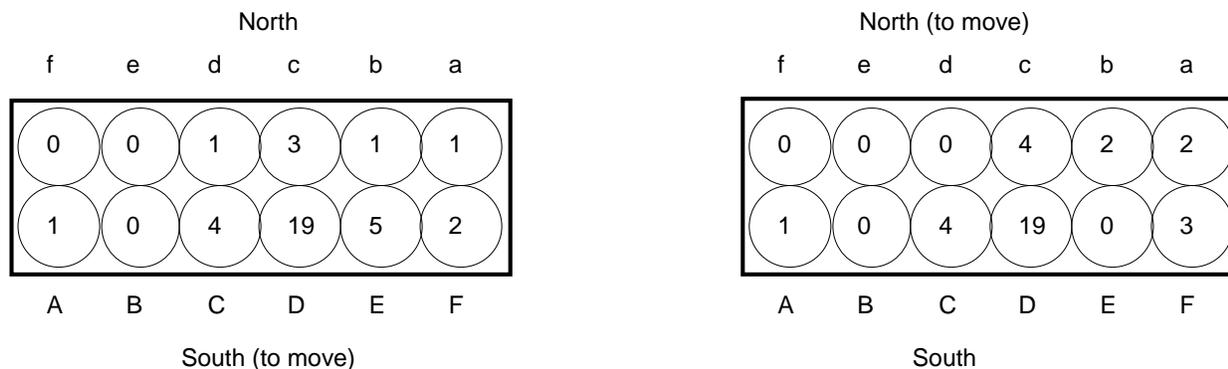


Figure 2: Example board positions from awari. Player South sows the 5 stones in Pit E and captures 2 stones from Pit d.

5.1 Awari

Awari is played on a board with 12 pits. Each of the two players owns half of the pits (see Figure 2). Initially, each pit contains 4 stones, so in total there are 48 (identical) stones. The goal of the game is to capture as many stones as possible. A player makes a move by selecting a non-empty pit owned by him or her and *sowing* all stones from this pit counter clockwise, dropping one stone in each consecutive pit. If the last pit to which a stone is added is owned by the opponent and now contains 2 or 3 stones, these stones are *captured*. If the preceding pit is also owned by the opponent and contains 2 or 3 stones, these are also captured, and the capture rule is applied recursively to this pit. An example is shown in the right part of Figure 2, where South captures 2 stones by sowing from Pit E. Additional rules exist, but these will not be discussed here.

An endgame database for awari contains the game-theoretical value for all positions containing up to a maximum number of stones. Even though awari may look simple, its search space is enormous. For the entire game, the database would contain 1400 billion entries.

If the maximum number of stones is S , the game-theoretical value lies between $-S$ and S , so there are $2 \times S + 1$ different values. Our program uses a minimal number of bits per entry to represent the game-theoretical value. For example, the 17-stones database uses 6 bits per entry. As there are 51,895,935 entries in this database, the disk space required to store it is about 38 Mbytes.

Our algorithm uses two additional fields per entry (see Figure 4). For awari, these two fields can be encoded with 3 additional bits. For the 17-stone database, we thus use 9 bits of internal memory for each position, or 57 Mbytes of total internal memory.

The sequential code of the program spends most of its time computing the unmove function and converting between positions and database addresses. The unmove function must compute, for a given position P , all possible positions that can lead to P through either a normal move or a capture move. In the latter case, the unmove function has to add stones to the board. The unmove function for awari is quite complicated, since the capture rules are complicated.

The address conversion routines must be able to determine the index in the database of a given position and vice versa. For this purpose, we use the simple Gödel coding scheme described in [1]. This scheme lexicographically orders all positions, based on the number

of stones in each pit. The index (Gödel number) for a position is the number of positions preceding it in this ordering. This number can be computed using precalculated binomial coefficients. The inverse operation (determining a position from its index) can be done in a similar way.

5.2 Performance for Awari

Our initial implementation performed poorly. We discovered that many processors often were idle, waiting for update messages from other processors. After monitoring the lengths of the input and output queues during the computation, we found that the program suffered from load imbalance. Each processor was busy during some iterations, but mostly idle during other iterations.

This load imbalance was caused by the distribution of entries among processors. This distribution used the Gödel coding scheme described above. Each processor was given the same number of entries, using a contiguous range of Gödel numbers. Due to the way the Gödel numbers were computed, the positions were not distributed randomly. For example, the first processor would get all positions with all stones in the 12th pit, but no positions with many stones in the low-numbered pits.

After we became aware of this problem, we made the distribution cyclic, by assigning position g to processor $g \bmod p$, where p is the number of processors. This assignment effectively randomizes the distribution. After this change, the load imbalance overhead decreased to a few percent, so each processor had the same amount of work to do during each iteration. The performance increased substantially. The time to compute the 17-stones database on 64 processors, for example, decreased from 4113 seconds to 2976 seconds.

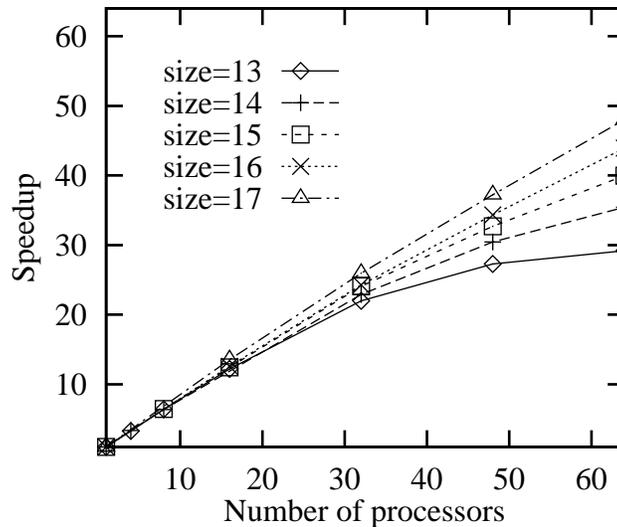


Figure 3: Speedups for the Awari Retrograde Analysis program on a collection of 64 SPARCs connected by an Ethernet.

The speedups with the new distribution scheme are shown in Figure 3, using up to 64 processors. We compute the speedup relative to the parallel Orca program on a single processor. Note, however, that our algorithm entirely avoids the buffering mechanism if run on one processor (see the if-statement in Figure 10), so it behaves exactly as a sequential algorithm

database size	number of CPUs	elapsed time	combining factor	msgs/sec	msgs/sec/CPU	volume (MB)
13	1	10205	-	-	-	-
	4	3053	52.0	72.7	18.2	55
	16	838	32.2	537.2	33.6	69
	64	350	12.0	3599.0	56.2	72
15	1	39100	-	-	-	-
	16	3146	34.8	489.9	30.6	256
	64	978	22.3	2590.1	40.5	269
17	1	142919	-	-	-	-
	64	2976	27.7	2164.6	33.8	851

Table 1: Performance characteristics.

in this case. The speedups increase with the size of the database. For a 13 stone database the speedup on 64 processors is 29.2. For 17 stones, the speedup is 48.

To analyze the performance of the program, we give some information about its behavior, using different database sizes and different numbers of processors (see Table 1). The first entry in the table gives the time (in seconds) to compute the database. (The time to write the database to disk after it has been computed is not included in our measurements, because Amoeba does not support parallel I/O.) The *combining factor* is the total number of update messages generated by the program divided by the total number of messages sent by the communication threads. This factor indicates how many logical messages are combined into one physical message. For the 17-stone database and 64 processors, for example, on average 1 message will be sent for every 27.7 updates. Each update requires 5 bytes (4 bytes for the Gödel code and 1 byte for the value).

Clearly, sending one message with about 139 bytes (5×27.7) over an Ethernet is far more efficient than sending 27.7 messages with 5 bytes of user data. In the Orca system we use, a remote invocation with little or no data takes 2.0 msec. Each additional byte of user data takes a few microseconds extra, so a 139 byte message is only slightly more expensive than a 5 byte message. Therefore, message combining greatly reduces the communication overhead.

The next three columns of the table give the total number of messages per second sent by all processors together, the average number of messages per second sent by each processor, and the total volume of the data (in megabytes) transferred over the network. All these numbers are for the messages generated by the communication thread for exchanging buffers. Our program also generates other kinds of messages (e.g., for termination detection), but these are sent far less often (typically a few times a second). The data volume is based only on user data and does not take protocol headers into account, so the amount of data actually transferred over the wire will be even higher.

As can be seen, the number of messages and the data volume increase with an increasing number of processors. With an increasing database size, the number of messages per second decreases, resulting in better speedups. The latter effect is due to the fact that message combining becomes more and more effective as the database size grows (see Table 1). For example, we have computed that the number of updates generated by the program increases almost as fast as the sequential computation time. Thus, the relative communication overhead without message combining would hardly decrease for increasing problem sizes. Therefore, the better speedups for large databases are due to the effectiveness of message combining.

Besides using message combining, we also reduce the communication overhead by overlapping computation and communication. We implemented this using separate worker and communication threads. The disadvantage of this scheme is the overhead in switching to and from the communication thread. On our system, each thread switch takes about 200 microseconds. Still, the latency for a message is much higher, so performance is gained by doing communication asynchronously with computation.

6 Discussion and Related Work

We have designed and implemented a parallel algorithm for performing retrograde analysis on a distributed system. Retrograde analysis is a useful AI search technique, which has been used successfully to compute endgame databases for several games. If the game’s search space is large, RA requires a huge amount of CPU time and internal memory. Therefore, parallelism and distribution improve the applicability of the technique.

We parallelize RA by partitioning the database among the available processors. In the paper, we have studied the problems with this approach and solutions to these problems. Foremost, the communication overhead of a distributed parallel RA algorithm is potentially excessive, but we have shown that the amount of communication can be reduced drastically using message combining.

Another severe problem is load balancing. A good solution to this problem is to give each processor the same number of entries, while making sure that the distribution is random. A regular pattern in the distribution easily leads to load imbalance. As a result of this randomization, there is no locality in the distribution. In other words, the distribution does not try to cluster parent and child nodes on the same processor, which might reduce communication overhead. Our work shows that randomization is very important for this application. Moreover, clustering nodes would be a game-specific optimization, whereas all our current optimizations do not depend on the underlying game.

We have given performance measurements for an example game, awari. For large databases, our RA program for awari achieves a speedup of 48 on 64 processors, even though we use a slow network (Ethernet). For example, a 17-stone database (with over 51 million entries) could be computed in 50 minutes on 64 processors, whereas the same problem took almost 40 hours on a single processor.

Earlier research on parallel RA includes the work by Stiller [10], who has computed endgame databases for chess. His program is designed for a SIMD machine (a CM-2), making it hard to compare with our work. The work closest to ours is that by Lake *et al.* [7], who have used a collection of workstations to compute endgame databases for checkers. Their work uses the irreversibility of some moves to partition the databases into smaller pieces that can be computed independently (and thus in parallel). So, they use more coarse-grained parallelism. Also, their work uses caching techniques and optimizes locality of references.

Acknowledgments

We would like to thank Hans Staalman and Elwin de Waal for their work on an earlier version of the parallel program. Also, we are grateful to Saniya Ben Hassen, Raoul Bhoedjang, Dick Grune, Cerial Jacobs, Koen Langendoen, John Romein, Tim Rühl, Jonathan Schaeffer, Andy Tanenbaum, and Cees Verstoep for giving useful comments on a draft of the paper.

References

- [1] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Databases in awari. In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad*, pages 73–86, Ellis Horwood, Chichester, England, February 1991.
- [2] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, February 1995.
- [3] H.E. Bal. *Programming Distributed Systems*. Prentice Hall Int'l, Hemel Hempstead, UK, 1991.
- [4] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
- [5] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, San Diego, September 1993.
- [6] R.U. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 1995.
- [7] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. Technical Report TR-93-13, Alberta, Edmonton, Canada, 1993.
- [8] M. Oey, K.G. Langendoen, and H.E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *15th International Conference on Distributed Computing Systems*, Vancouver, B.C., Canada, May 1995.
- [9] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53:273–289, 1992.
- [10] L. Stiller. Parallel analysis of certain endgames. *ICCA Journal*, 12(2):55–64, 1989.
- [11] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, N.J., 1995.
- [12] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(2):46–63, December 1990.
- [13] K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.
- [14] D.A. Wood and M.D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, February 1995.

Appendix

```

type DatabaseType = array [1 .. Size] of
  record
    UnknownChildren: integer
    BestValue: integer
    Status: (Unknown, Known, Propagated)

```

Figure 4: Declaration for the database type.

```

type buffer = array [1 .. DynamicSize] of
  record
    Address: integer
    Value: integer

```

Figure 5: Declaration for the buffer type.

```

db : DatabaseType[1 .. MyMax];
inbuf : buffer;
out: array [1 .. NCPUS] of buffer;

```

Figure 6: Declarations for the parallel algorithm.

```

process Worker
  for e := 1 to MyMax do # Initialization phase
    if e represents an end position then
      db[e].BestValue := GameTheoreticalValue(e)
      db[e].Status := Known
    else
      db[e].UnknownChildren := number of children of e
      db[e].BestValue := -N
      db[e].Status := Unknown
  for U := N downto 1 do # Do N iteration phases
    UpperBound := U
    for e := 1 to MyMax do
      process_input
      if db[e].Status = Known or
        (db[e].Status = Unknown and
         db[e].BestValue = UpperBound) then
        propagate(e)
      while not terminate this iteration do
        process_input
  for e := 1 to MyMax do # Final phase
    if db[e].status = Unknown then
      db[e].BestValue := 0;

```

Figure 7: The worker thread main loop.

```

procedure process_input
  for [pos, newvalue] in inbuf do
    handle(pos, newvalue)

```

Figure 8: The process input procedure.

```

procedure handle(pos, newvalue)
  db[pos].BestValue := MAX(db[pos].BestValue, newvalue)
  db[pos].UnknownChildren -= 1
  if db[pos].UnknownChildren = 0 or
    db[pos].BestValue = UpperBound then
    propagate(pos)

```

Figure 9: The handle procedure.

```

procedure propagate(e)
  db[e].Status := Propagated
  for pos in UNMOVE(e) do
    newvalue := -db[e].BestValue + moveValue(pos,e)
    if owner(pos) = thiscpu then
      handle(pos, newvalue)
    else
      add [pos, newvalue] to out[owner(pos)]

```

Figure 10: The propagate procedure.

```

process CommunicationThread
  while not terminate program do
    cpu := select an output buffer
    send out[cpu] to cpu's inbuf # send data over network
    out[cpu] := empty

```

Figure 11: The communication thread algorithm.