The ABDADA Distributed Minimax Search Algorithm

Jean-Christophe Weill
Institut d'Intelligence Artificielle
Université de Paris 8 – Vincennes à Saint-Denis
24, avenue du Pont Royal
94230 CACHAN
France

ABSTRACT

This paper presents a new method to parallelize the minimax tree search algorithm. This method is then compared to the "Young Brother Wait Concept" algorithm in an Othello program implementation and in a Chess program. Results of tests done on a 32-node CM5 and a 128-node CRAY T3D computers are given.

1 INTRODUCTION

In the search for power to run our game-playing programs as fast as possible, the use of parallel computers is a stimulating choice. During the past few years, parallel algorithms have evolved from fixed master-slave relationships, where the master looked for slaves to complete its task, to more dynamic master-slave relationships, where unemployed processors look for a master in order to find some task to do. But those master-slave relationships, which are exemplified by the work-stealing schedulers of Jamboree[12] and YBWC[7] still suffer from synchronization overheads.

In this paper, we describe a non-synchronized parallel algorithm named ABDADA.

After a formal description and pseudo code for the ABDADA parallel scheme, we present an experimental comparison of ABDADA to YBWC.

2 THE ABDADA ALGORITHM

The ABDADA search algorithm is based upon both YBWC[9, 7] and $\alpha\beta$*[6]. From YBWC, it keeps the basic concept: parallel evaluation of successor positions of a game position is allowed if and only if the eldest brother (i.e. the first visited brother) is fully evaluated. The $\alpha\beta$* algorithm, like the algorithm from Otto and Felten [18], relies on a shared transposition table. All processors start the search simultaneously at the root. With the help of the additional transposition table information (e.g. how many processors are exploring a subtree rooted at a node), it is possible to control speculative parallelism. $\alpha\beta$* was originally implemented using a complex and inefficient control, so we chose to mix YBWC and $\alpha\beta$*.

The ABDADA[1] algorithm can be described as follows:

1. Let $\mathcal{T}$ be a shared transposition table. To the standard definition of the table[13], we add, for each entry, a new field `nproc` which is the number of processors currently evaluating the node related to the transposition table entry.
2. All the processors begin the search simultaneously at the root of the game tree.
3. When a processor enters the evaluation of a position P, it increments the field `tt[P].nproc` in the transposition table.
4. When a processor leaves a position P (because this position is fully evaluated or has been pruned), it decrements `tt[P].nproc`.
5. The analysis of a position is done in three phases:
    (a) The eldest son is analysed, regardless of the position of the other processors;
    (b) the other sons which are not currently analysed by other processors are analysed;
    (c) then the sons which are not completely evaluated are analysed (i.e. the corresponding entry in the transposition table has not been evaluated to full depth).[2]

Figure 1 shows the ABDADA algorithm. It can be obtained by modifying a sequential $\alpha\beta$ using a transposition table in the following way:

- We add to the usual parameters of the procedure a Boolean `exclusiveP` which indicates whether the corresponding node should be evaluated *exclusively*, i.e. the processor is allowed to evaluate the current node if no other processor is currently evaluating it. This parameter will be passed to the transposition table retrieve procedure `RetrieveAsk` as shown on figure 2.

---

[1] ABDADA is the acronym of the French name "Alpha-Bêta Distribué avec Droit d'Anesse" which can be translated into "Distributed Alpha-Beta Search with Eldest Son Right".

[2] This point is analogous to the "Helpful Master Concept"[8], if, in the second phase, the processor has found some node, other processors are not allowed to work on this node unless there is nothing else to do. Like in the "Helpful Master Concept" when a master waits for the completion of its slave children, it becomes their slave.

```
1  funct abdada(Position, $\alpha$, $\beta$, depth, exclusiveP) $\equiv$
2      var exclusive, iteration, alldone;
4          if depth $= 0$ then exit evaluate(Position); fi
5          Best $\leftarrow -\infty$;
6          RetrieveAsk(Position, $\alpha$, $\beta$, depth, exclusiveP);
8          Generate Move when waiting for the answer
10         GenMove(Position);
12         RetrieveAnswer($\&\alpha$, $\&\beta$, $\&$Best);
14         The current move is not evaluated if causing
15         a cutoff or if we are in exclusive mode and
16         another processor is currently evaluating it.
18         if $(\alpha \geq \beta) \vee$ (Best = ON_EVALUATION)
19           then exit Best; fi
21         iteration $\leftarrow 0$;
22         alldone $\leftarrow$ false;
24         while (iteration $< 2) \wedge (\alpha < \beta) \wedge \neg$alldone do
25             iteration $\leftarrow$ iteration $+ 1$;
26             alldone $\leftarrow$ true;
28             $M \leftarrow$ FirstMove(Position);
30             while $(M \neq 0 \wedge \alpha < \beta)$ do
31                 exclusive $\leftarrow$ (iteration $= 1$)
32                 $\wedge$ (NotFirstMove($M$));
34                 On the first iteration, we want
35                 to be the only processor
36                 to evaluate young sons.
38                 value $\leftarrow -$abdada(Position $\bullet M, -\beta,$
39                 $-\max(\alpha,$ Best), depth $- 1$, exclusive);
40                 if value $= -$ON_EVALUATION
41                   then alldone $\leftarrow$ false;
42                 elsif value $>$ Best
43                     then Best $\leftarrow$ value;
44                         if Best $\geq \beta$
45                           then skip endsearch; fi
46                 fi
47                 $M \leftarrow$ NextMove(Position) od;
49         od
50         endsearch:
51         StoreHash(Position, $\alpha$, $\beta$, Best);
52         exit Best.
```

Figure 1: The ABDADA parallel scheme on the $\alpha\beta$ Search

- We have to define a new value `ON_EVALUATION` which is different from any value returned by the evaluation function.
- The `RetrieveAsk` procedure has to return this value as a `score` if there are other processors evaluating the node and `exclusiveP` is `true`. Otherwise, if there is no forward pruning due to the transposition table, the entry `nproc` is incremented.
- Using the same method, the storing procedure decrements the entry `nproc` when a processor leaves a node.
- The three phases are implemented using the variable `iteration` which can take the value `1` and `2` in the inner loop (lines 29 to 42). The first time we enter the inner loop, we reset the `exclusive` flag only on the first move while we reset it on every move in the second iteration. When the `exclusive` flag is set, a son which has been pruned because another processor is currently evaluating it will have a return value of `-ON_EVALUATION`. If we find such a son, then we reset the `alldone` flag meaning that another iteration will be necessary to know the value of this son.

With this definition, we can easily see that compared to a YBWC search, the ABDADA search will attempt to read the transposition table at most twice as many times, and write roughly the same number of times.

3 RESULTS FROM EXPERIMENTS

To analyse the behaviour of the parallel scheme, we used the parallelization of two sequential game programs: a competitive Othello program[3] and an early version of the *Frenchess* chess program.

The Othello program has an unstable evaluation function as the search is extended. Most Othello programmers know of the odd–even problem which results in an evaluation at odd depth being far better than one at an even depth because whoever just moved has just given mobility to their opponent. (Othello is a game of zugzwang.) Furthermore, the Othello program shows some very unstable principal variations. This has the uncommon effect that the $\alpha\beta$ search is better than NegaScout[19, 20] search for this program.

In contrast, the chess program used a fast incremental evaluation function based on an Oracle[3] combined with Piece/Color/Square tables, and produced a fairly stable evaluation function and also stable principal variations associated with optimised key move ordering techniques.

Whatever the number of processors is, the global number of entries in the transposition table is kept constant.

3.1 DEFINITIONS

Before giving some results from the experiments we made on a CM5 Thinking Machine Corporation Computer with 32 SPARC nodes, we need some basic definitions of the measures we used.

---

[3] The sequential Othello program named *Bugs* finished fourth in the Waterloo Othello tournament in 1994, and finished second of the Paderborn Othello tournament in 1995.

```
proc RetrieveAsk(Position, $\alpha$, $\beta$, depth, exclusiveP) $\equiv$
    var entry, answer; end
    entry $\leftarrow \mathcal{T}$(Position);
    answer.$\alpha \leftarrow \alpha$; answer.$\beta \leftarrow \beta$;
    answer.score $\leftarrow -\infty$;
    if entry $= 0$ then
                  If not exists then exit
                  skip endprobe;
    fi
    if entry.height $=$ depth $\wedge$ exclusiveP $\wedge$ entry.nproc $> 0$
      then
          Only one processor allowed if exclusivity
          is required
          answer.score $\leftarrow$ ON_EVALUATION;
          skip endprobe
    fi
    if entry.height $\geq$ depth
      then
          if entry.flag $=$ VALID
            then answer.score $\leftarrow$ entry.score;
                 answer.alpha $\leftarrow$ entry.score;
                 answer.$\beta \leftarrow$ entry.score
          elsif entry.flag $=$ UBOUND $\wedge$ entry.score $< \beta$
              then answer.score $\leftarrow$ entry.score;
                   answer.$\beta \leftarrow$ entry.score
          elsif entry.flag $=$ LBOUND $\wedge$ entry.score $> \alpha$
              then answer.score $\leftarrow$ entry.score;
                   answer.$\alpha \leftarrow$ entry.score
          fi
          if entry.depth $=$ depth $\wedge$ answer.$\alpha <$ answer.$\beta$
            then
                 Increment the number of processors
                 evaluating this node
                 entry.nproc $\leftarrow$ entry.nproc $+ 1$
          fi
      else
          This is the first processor to evaluate this node
          entry.depth $\leftarrow$ depth;
          entry.flag $\leftarrow$ UNSET;
          entry.nproc $\leftarrow 1$;
    fi
    endprobe:
    Now send the answer
    Send.ttableanswer(answer.$\alpha$, answer.$\beta$, answer.score);
.
proc StoreHash(Position, $\alpha$, $\beta$, score, depth) $\equiv$
    var entry; end
    entry $\leftarrow \mathcal{T}$(Position);
    if entry $= 0 \vee$ entry.height $>$ depth then exit fi
    if entry.height $=$ depth
      then entry.nproc $\leftarrow$ entry.nproc $- 1$
       else entry.nproc $\leftarrow 0$
    fi
    if score $\geq \beta$ then entry.flag $\leftarrow$ LBOUND
    elsif score $\leq \alpha$ then entry.flag $\leftarrow$ UBOUND
                      else entry.flag $\leftarrow$ VALID
    fi
    entry.score $\leftarrow$ score;
    entry.depth $\leftarrow$ depth;
.
```

Figure 2: The transposition table management for the ABDADA scheme

These definitions follow those given in a previous paper[15]:

The Communication Overhead (CO) is the overhead that is caused when the parallel program sends messages back and forth between processors. It also includes the time spent to encode and decode a message. It is dependent on both hardware and software.

The Search Overhead (SO) is the cost attributable to the extra nodes searched in the parallel version compared[4] to the sequential version. It is defined by the formula

$$SO = \frac{\text{(Nodes searched for N CPUs)}}{\text{(Nodes searched for 1 CPU)}} - 1.$$

The Synchronization Overhead (SY) is the cost attributed when a processor becomes idle. This can happen when it has no job to do or when it is waiting for a result (communication) from another processor.

The Time Overhead (TO) is the observable measure of overhead. It is defined as

$$TO = \text{(Time using N Cpus)} \times \frac{N}{\text{(Time using 1 Cpu)}} - 1.$$

and is approximately related to the other overheads by

$$TO = SO + CO + SY.$$

The Relative Speedup (RS) is defined by

$$RS = \frac{\text{(Time using 1 CPU)}}{\text{(Time using N CPUs)}}.$$

for the same algorithm, i.e. using only one CPU, we use the same algorithm as for the parallel version. This enables us to measure the performance of the parallelization scheme.

The Absolute Speedup (AS) is, in contrast, defined by

$$AS = \frac{\text{(Time of the best sequential algorithm)}}{\text{(Time using N CPUs)}}.$$

This measure is suited for comparing parallel algorithms. Some schemes can parallelize poor sequential algorithms well, achieving a good relative speedup. However, they may be slower on N CPUs than a better sequential algorithm and a parallel algorithm with a smaller relative speedup. In our problems, we have used the recursive $\alpha\beta$ search for the Othello program as a "best sequential algorithm" and the recursive Negascout search for the chess program.

The Efficiency is defined by

$$EF = \frac{\text{(Absolute Speedup)}}{N}.$$

.

---

[4] In the case of the ABDADA search, we count as a node a node that is not immediately pruned by the speculative search control. In this case, the search overhead is created by the fact that many processors are searching the same nodes.

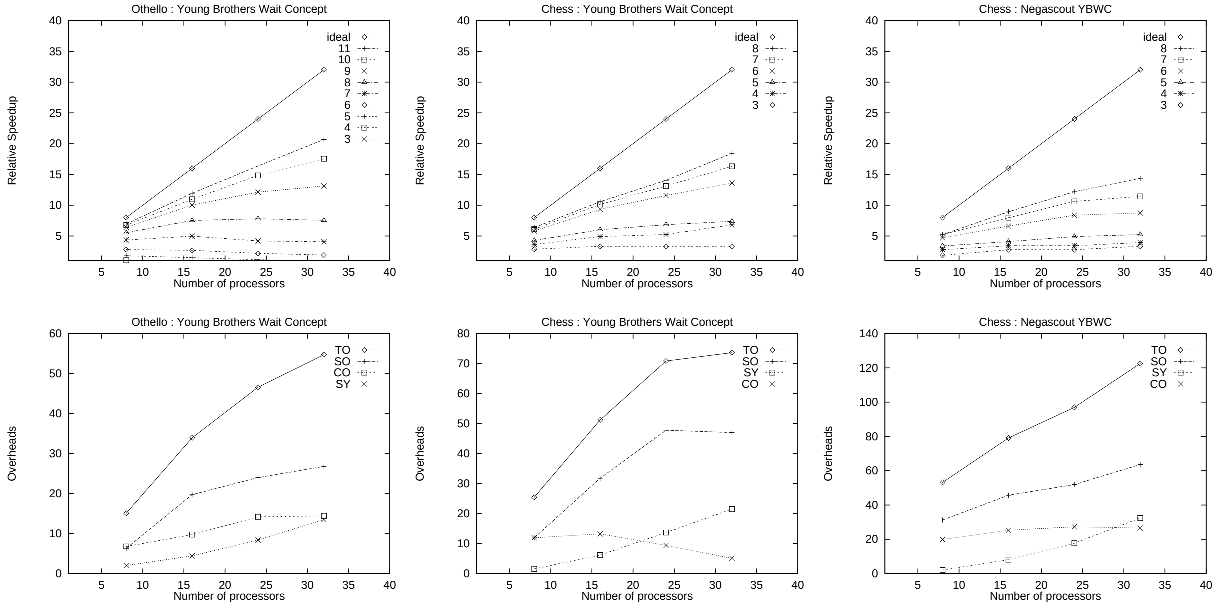

Figure 3: Results with the "Young Brothers Wait Concept" scheme applied to the $\alpha\beta$ algorithm.

The top graphs show the relative speedup versus the number of processors and the bottom graphs show the different overheads. Chess results represent the average on the 24 Bratko-Kopec[4] positions. One processor using Negascout needs 52045 seconds to search all these positions at depth 8, the minimum time is achieved by the position 8 in 25 seconds. Position 5 requires 13827 seconds. Othello results represent the average on the first 50 moves of a high level computer Othello tournament game. At depth 11, on one processor, the Othello program searches all the game in 38205 seconds. Top graphs are curves for depth 3 to 11 for Othello and depth 3 to 8 for chess. Bottom graphs are for depth 11 for Othello and depth 8 for chess.

3.2 YOUNG BROTHERS WAIT CONCEPT RESULTS

We implemented the YBWC search on the Othello program and on the chess program. As this scheme was primarily chosen for the definition of *Frenchess*,[5] we tried many possible optimisations to ensure that the results were the best possible. The results are shown on figure 3.

As it can be seen, for the Othello program, we obtained a relative speedup of 20.7 for depth 11 trees. For on the chess program at depth 8 trees, we obtained a speedup of 18.4 using $\alpha\beta$ and 14.3 using Negascout[6] (In our chess program, the sequential Negascout version is 1.5 times faster than the $\alpha\beta$ version). These speedups are relative to the same program used on a one processor computer (with the same number of entries in the transposition table), so as YBWC requires a non-recursive minimax search,[7] these speedups are relative to a non-recursive $\alpha\beta$ search (or a non-recursive Negascout search when applicable)[7, 23].

For both programs the main overhead is the Search Overhead combined with a relatively reduced Communication Overhead (by the fact of our optimisations see [23]) and a greater Synchronization Overhead in the chess program than in the Othello program.

3.3 ABDADA RESULTS

The ABDADA scheme was performed on the same positions for both games as YBWC. Figure 4 summarizes the results.

Here the Synchronization Overheads are non-existent since all processors are busy. The Communication-Overhead represents the time spent waiting for the transposition table answers (i.e. the simulation of the shared memory). The main overhead is the Search-Overhead. We must also note that this time the reference sequential programs use recursive implementations of both $\alpha\beta$ and Negascout algorithms. Thus the speedup for Othello using $\alpha\beta$ search is an absolute speedup like the speedup for Chess using Negascout. It should also be noted that the ABDADA speedup for small depths is intrinsically much better than for YBWC.

We also measured the number of messages used to read and write the transposition table. The ratio between the number of reads and writes tells us that ABDADA is far from doubling the number of read messages. For the chess program, on the Bratko-Kopec positions, [8] YBWC's read-

---

[5] Frenchess running ABDADA on a 128 processors Cray T3D finished fourth at the 8th World Computer-Chess Championship. The version used for this article includes less search and evaluation heuristics.

[6] Of course, Negascout's relative speedup is less than $\alpha\beta$'s since the relative tree searching inefficiency of $\alpha\beta$ provides a greater potential for improvement by a parallel scheme.

[7] It is difficult to implement YBWC on a recursive search since when a processor receives an evaluation-answer from one of its slaves, it must be able to produce a jump in the search depth if this evaluation produces a pruning of the current master node. So the values α and β must at least be kept in arrays indexed by nodes depths (in the "normal" version of $\alpha\beta$ search, they are kept on stack) and special arrangements must be made in order to prevent the disruptive use of depth that can be caused by this pruning. So YBWC needs a hugely modified search algorithm compared to ABDADA.

[8] The choice of the Bratko-Kopec positions may be criticised, but

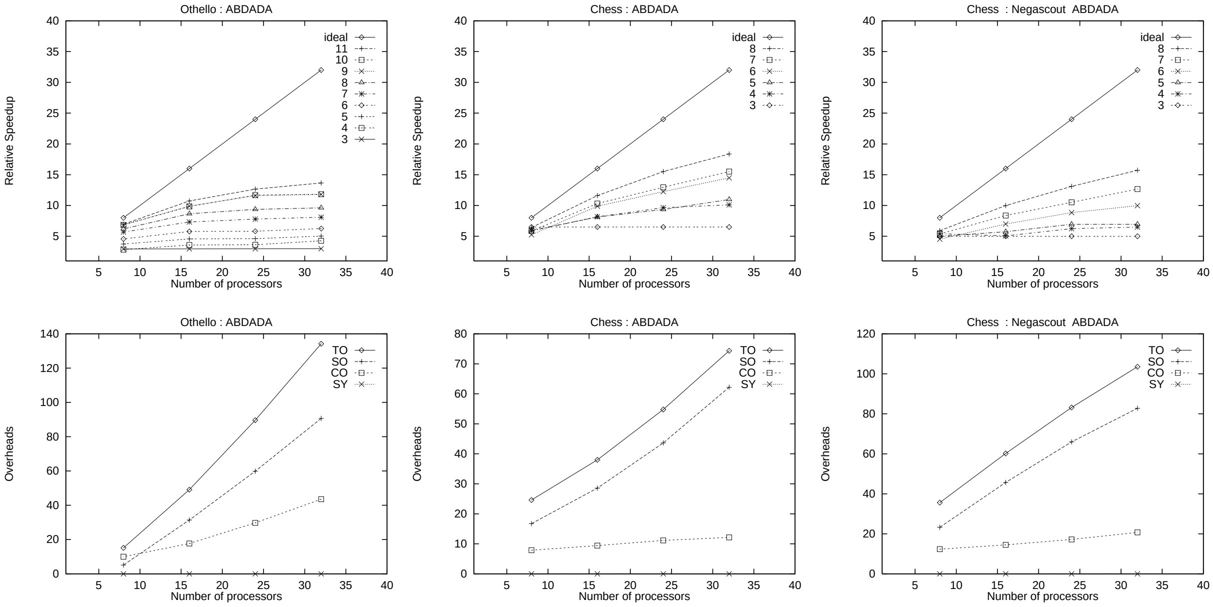

Figure 4: The ABDADA scheme results

write ratio is 1.35 and ABDADA's read-write ratio is 1.7. Thus the overhead of the non-optimised,[9] version given in part 2 in term of number of read messages, is about 25 % as compared to YBWC.

4 DISCUSSION

It is difficult to compare the shown results with the results from different papers. So we have to compare the two schemes on absolute speedup.

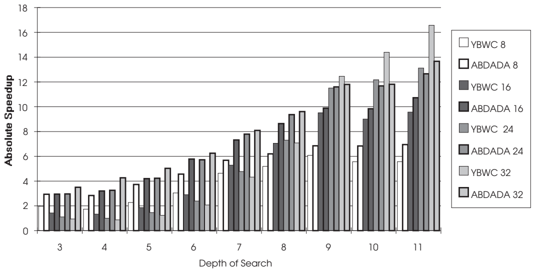

Figure 5: Comparison of absolute speedups for both schemes applied to Othello for 8, 16, 24, 32 processors

Figure 5 shows the comparison of speed on the Othello program compared to the fastest sequential Othello program we have, the recursive $\alpha\beta$. We can see that up to depth 8, ABDADA is more efficient than YBWC search but this changes in depths greater than 9.

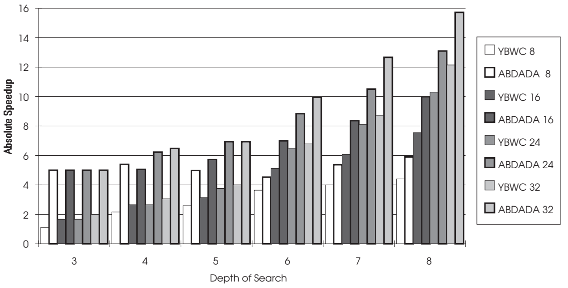

Figure 6: Comparison of absolute speedups for both schemes applied to chess for 8, 16, 24, 32 processors

Figure 6 gives the comparison for the game of chess using the recursive Negascout (which is 30% faster than the non-recursive one[10]). Here we can see that ABDADA search significantly outperforms the YBWC whatever the depth.

Some explanations can be advanced:

- The game trees are not of the same size, the branching factor for Othello is 10 in midgame positions while it is 40 for chess. Thus Othello game trees are deeper;
- Since our Othello evaluation function is unstable, Othello game trees are less ordered than chess trees. This makes Negascout less efficient than $\alpha\beta$ and YBWC search becomes more efficient since we show in [23] that YBWC is more efficient in random trees than in strongly ordered trees.

Without more data, it is difficult to find what the real explanation is. The only other remark we have on our results

---

there is no good choice and we did it in order to have results comparable with those from previous publications.

[9] In the pseudo code given in part 2 the eldest node is read twice and so are all nodes evaluated in the second pass. This is the version we used in our experiments for simplicity of code but it is really easy to improve the search by searching only the non-evaluated nodes in the third pass.

[10] This apparently strange behaviour observed independently by Brockington[5] can be explained by the fact that we use high level languages in which procedure stacks use are much more optimised than indexed arrays use. Similar results can be observed for the Quicksort algorithms (see [24], page 100).

is that the search overheads are greater with ABDADA than with YBWC, which is easily explained by the fact that in the YBWC search there is sharing of $\alpha\beta$ window improvement during the search. When a processor finishes the evaluation of a child of a position P, the owner of the position may distribute, if needed, the new bounds on the $\alpha\beta$ window to his slaves. [11] ABDADA does not share windows, so in the situation described in Figure 7, YBWC will examine fewer nodes than ABDADA. New bounds are distributed only if the eldest move of the position P is not the best one, (i.e. the tree is not very well ordered).

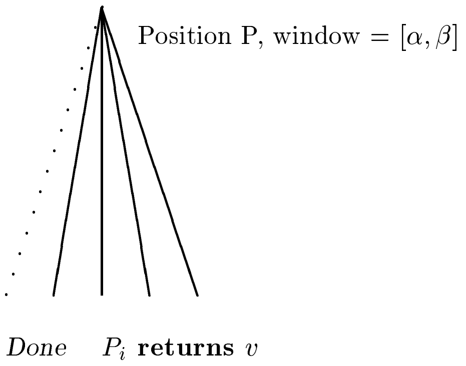

Figure 7: Situation where YBWC examines fewer nodes than ABDADA.

Let's suppose that the children of the position $\mathcal{P}$ are being evaluated by some processors (the eldest son of $\mathcal{P}$ is already explored by both YBWC and ABDADA) and the processor examining P_i returns a value v that will cause the pruning of the position $\mathcal{P}$ (i.e. when $v \geq \beta$). In this situation YBWC will cause the pruning (making them idle) of the other processors examining other children of $\mathcal{P}$ whereas ABDADA will continue the exploration of these children.

From these results and our experience, we can define when to choose each of the algorithms depending on the goal and the means:

YBWC:

Advantages:

- Works without transposition table.
- Works with global transposition tables.
- Works with local transposition tables.
- Efficient for deep problems.
- Can be optimised as function of machine topology[7].

Drawbacks:

- Complex to implement: Even with the definition given in [7], it is difficult to ensure that the basic master-slave relationship is working properly. The debugging part of the scheme is long, mainly because of the non-deterministic nature of Work Stealing based Process.[5]
- Since the search algorithm must be non-recursive, it is difficult to implement the same heuristics as in the sequential programs.
- Non-efficient for small size problems.

ABDADA:

Advantages:

- Easy and quick to implement: As we can see on Figure 1, the main algorithm is very close to sequential algorithms, so few modifications are necessary.
- Efficient for small and deep problems.
- Failure resistant: it is very easy to change the algorithm to be failure resistant (i.e. to give an answer even if one or more processors are dead), you only have to put a time-out when waiting for transposition table answers and then you have an algorithm which will return an answer even when only one processor is present !

Drawbacks:

- Needs a very fast global transposition table. This is where all the implementation effort is made to ensure that transposition table management is fast enough for this algorithm.
- Needs to be redefined in order to take into account situations such as those of figure 7.[12]

So YBWC is our choice if the computer is not able to give us very fast message passing (for small messages) such as a network of workstations, and ABDADA is our choice on shared memory or real fast small message passing computers. On those computers and having enough time for implementation, we propose to first implement ABDADA to verify its efficiency and, if it is not as efficient as required, to implement YBWC, not being certain that this will be more efficient but to ensure you have not ignored reasonable alternatives to ABDADA.

5 APPLICATION ON A 128-PE CRAY T3D

After doing this comparison, we had to implement a new chess program named Frenchess on a 128-PE CRAY T3D. This program finished equal third of a field of 24 at the 8th World Computer Chess Championship in Hong Kong (May 25 – May 30 1995). Since we began to port Frenchess to the CRAY T3D in February, we had very little time before the World Championship, so we chose to use ABDADA as a parallel scheme.

On the CM5 computer, shared memory was simulated, each processor being in charge of a part of the global transposition table; we had to rely on the active message layer to get some very good performance which pure CMMD[22] could not give us otherwise.

This method of shared memory simulation has the advantage of making simultaneous accesses to a same entry

---

[11] It should be noted that this feature of the YBWC is, in my mind, the main difference with PVSplit based searches[14, 16, 17, 15, 21, 10] or Jamboree[11].

[12] There are basically two ways to solve this:

- To stock in the transposition table, not only the number of processors but also which processors are currently evaluating a given position, so that we can distribute the new windows bounds to all concerned processors.
- To check, when finishing the evaluation of a node, that its parent node has not been pruned (i.e. by looking up the corresponding entry in the transposition table).

The latter method is the easiest to implement but this only partially solves the problem while the former will require more fundamental modifications of the search process.

impossible (this is important to ensure that ABDADA works correctly, i.e. that the number of processors is correct for a given transposition table entry).

In an adaptation of Frenchess for a CRAY SUN CS 6400, we used shared memory with mutex (mutual exclusion) locks to guarantee the exclusive access to each transposition table entry. Precise measurement of the speedup was not possible on this machine[13] but it looked very much like what we had seen on the CM5 even though single processor speed was around four times greater.

On the CRAY T3D, we used the SHMEM library[1] after we found how to ensure mutual exclusion with the help of the `shmem_swap` call[2].

Single PE performance on the T3D was a big disappointment for us : on a 50 MHz SPARC 10 (or a single processor of the CRAY CS6400), the sequential algorithm was visiting around 20000-25000 positions per second. We had similar benchmarks on a 150 MHz DEC Alpha station. But, on the CRAY T3D, the speed on one PE was only in the 7000-9000 positions/second range ! Probably, this poor performance was due to two problems :

1. cache problems : cache too small and cache invalidation,
2. poor optimisations from the compiler.

Nevertheless and despite the fact that we did not have enough CPU time[14] to complete all the tests to compute the speedup, parallel performances were great : to have results comparable to those we obtained on the CM5, we used a search with a simple evaluation, no extensions except check extensions, no futility cutt-offs and no null move pruning.

Depth	3	4	5	6	7	8	9
Speedup	1.36	3.07	5.75	12.9	23.3	41.2	65.9

Table 1: Speedup for a 128-PE CRAY T3D compared to 1PE with 128 times smaller transposition table on the Bratko-Kopec positions.

Due to the short time, we did the first test without a constant number of transposition table entries : when using 128 processors we had 128 more entries (1M entries per processor) than when using only one processor. Table 1 shows that in those conditions, at depth 9, we have an average absolute speedup of 65.9 on the 24 Bratko-Kopec positions. During real games, where each player has 2 hours to play 40 moves, Frenchess searches most positions to depth 10, 11 or 12 and even more in simplified endgames. We did not have enough time to compute the positions on one processor at depth 10, so we compared the relative speedup of 128 processors compared to 32 with a constant transposition table in table 2.

Since the speedup for 32 processors with a constant transposition table size is about 28 at depth 9 and that the speedup is increasing with search depth, we can expect a speedup superior to 75 for 128 PEs for depth 10. This shows clearly that ABDADA is very efficient on the program Frenchess for a 128-PE CRAY T3D.

Depth	3	4	5	6	7	8	9	10
Speedup	1.05	1.13	1.23	1.22	1.49	1.98	2.62	2.68

Table 2: Relative Speedup for a 128-PE CRAY T3D compared to a 32-PE CRAY T3D with a constant transposition table size.

6 CONCLUSION

We have described a new parallelism scheme efficient for minimax search. We have shown that under some conditions this new scheme can be more efficient than YBWC.

Doing all those comparisons on the same computers using the same code, we have shown that comparisons of parallel algorithms are only valid for the given problem (i.e. the nature of the game, the algorithms, and the evaluation function) on a given computer. There is no possible way to say, given one comparison, on whatever problem and on whatever computer an algorithm is universally better than all others.

Furthermore, the analysis done of the CRAY T3D, despite the lack of CPU time, showed that ABDADA is still very efficient with 128 processors.

We would like to continue this work in order to give a comparison of ABDADA and YBWC, using the measures of *critical-path length* and *work performed*[12], to have more predictive power on the behaviour of those algorithms. This, done on two different games, may help us understand better the parallel searches.

Probably ABDADA is only a small step toward the definition of future parallel algorithms, but when designing them, we should keep in mind the power of a global transposition table.

6.1 ACKNOWLEDGEMENTS

Thanks to Marc-François Baudot for his great help on the definition and the implementation of the chess program *Frenchess*.

Marc-François Baudot, Mark Brockington, Warren Smith and Michael Buro should also be thanked for their help in editing the document.

Thanks also to my thesis reviewers, Tony Marsland, Jacques Pitrat and Michel Gondran for all their constructive criticism on Chapter 4 of my thesis.

The present work has been performed in the framework of the *Frenchess* project, part of a joint research effort with Electricité de France (department EDF/DER/TIEM/IMA) and the Artificial Intelligence Institut of the university of Paris 8.

---

[13] The only way would have been to make sure that nobody else used the same partition as Frenchess. We did not try to negotiate this as we knew we would be moving to the T3D soon and the CS6400 was rather heavily used.

[14] To complete our test, for example computing the speedup at depth 10, we would have had to use the whole machine for several hours just to measure the performance of one processor. Our problem (the world Championship) being a short term practical one, we spent machine time in a more useful way.

REFERENCES

[1] Barriuso, R., and Knies, A. *SHMEM User's Guide Revision 1.08.* CRAY Research Inc, April 1994.

[2] Baudot, M.-F., Weill, J.-C., Seret, J.-L., and Gondran, M. Frenchess: A Cray T3D at the 8th World Computer Chess Championship. In *1st European Cray-T3D Workshop* (Sept. 1995), École Polytechnique Fédérale de Lausanne and CRAY Research.

[3] Berliner, H., and Ebeling, C. Pattern knowledge and search: The SUPREM architecture. *Artificial Intelligence 38*, 2 (Mar. 1989), 161–198.

[4] Bratko, I., and Kopec, D. A test for comparison of human and computer performance. In *Advances in Computer Chess III* (1982), M. Clarke, Ed., Pergamon Press, pp. 31–56.

[5] Brockington, M. An implementation of the young brothers wait concept. Internal report, University of Alberta, 1994.

[6] David, V. *Algorithmique parallèle sur les arbres de décision et raisonnement en temps contraint. Étude et application au Minimax.* PhD thesis, ENSAE, Nov. 1993.

[7] Feldmann, R. *Spielbaumsuche mit massiv parallelen Systemen.* PhD thesis, Fachbereich Mathematik / Informatik – Universität GH Paderborn, 1993.

[8] Feldmann, R., Monien, B., Mysliwietz, P., and Vornberger, O. Distributed game tree search. In *Parallel Algorithms for Machine Intelligence and Vision* (1990), V. Kumar, K. L.N., and P. Fopalkrishman, Eds., Springer Verlag, pp. 66–101.

[9] Feldmann, R., Mysliwietz, P., and Vornberger, O. A local area network used as as parallel architecture. Tech. Rep. 31, University of Paderborn, Sept. 1986.

[10] Hyatt, M., Suter, B., and Nelson, H. A parallel alpha/beta searching algorithm. *Parallel Computing 10*, 3 (1989), 299–308.

[11] Kuszmaul, B. *Synchronized MIMD Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994.

[12] Kuszmaul, B. The StarTech massively-parallel chess program. *ICCA Journal 18*, 1 (Mar. 1995), 3–19.

[13] Marsland, T. A review of game-tree pruning. *ICCA Journal 9*, 1 (1986), 3–19.

[14] Marsland, T., and Campbell, M. Parallel search of strongly ordered game trees. *Computing Surveys 14*, 4 (1982), 553–562.

[15] Marsland, T., Olafsson, M., and Schaeffer, J. Multiprocessor tree-search experiments. In *Advances in Computer Chess IV* (1986), D. Beal, Ed., Pergamon Press, pp. 37–51.

[16] Marsland, T., and Popowich, F. Parallel game tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 4 (1985), 442–452.

[17] Newborn, M. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-10*, 5 (Sept. 1988), 687–694.

[18] Otto, S., and Felten, E. Chess on a hypercube. In *The Third Conference on Hypercube Concurrent Computers and Applications* (1988), vol. 2, pp. 1329–1341.

[19] Pearl, J. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence 14* (1980), 113–138.

[20] Reinefeld, A. An improvement to the scout tree-search algorithm. *ICCA Journal 6*, 4 (1983), 4–14.

[21] Schaeffer, J. Distributed game-tree search. *Journal of Parallel and Distributed Computing 6*, 2 (1989), 90–114.

[22] Thinking Machines Corporation. *CMMD Reference Manual V3.0.* Cambridge, Massachusetts, May 1993.

[23] Weill, J.-C. *Programmes d'échecs de championnat : architechture logicielle, synthèse de fonctions d'évaluation, parallélisme de recherche.* PhD thesis, Université de Paris VIII, Saint-Denis (France), Jan. 1995.

[24] Wirth, N. *Algorithms and Data Structures.* Prentice-Hall International Editions, 1986.