Parallel Game Algorithms

John Leung

CS535 – Parallel Algorithms

Dr. G. S. Young

Cal Poly Pomona

Spring, 2003

Abstract

The study of algorithms in computer games will be discussed in this paper. In particular, game tree structures, search methods, and pruning will first be introduced. Later, the two main branch of parallel game tree algorithms - the synchronous and the asynchronous methods, respectively represented by the Young Brothers Wait Concept (YBWC) and the Asynchronous Parallel Hierarchical Iterative Deepening (APHID) will be discussed in greater detail.

1. Introduction

Ever since the computer became programmable in the 1950s, humans have been intrigued to program "intelligent" (in reality, just brute force exhaustive search) computer games. In fact, even Claude Shannon (of Information Theory) and Alan Turing (of Turing Test and Turing Machine) have worked on chess programs in the 1950s. However, it was not until the 1982, did the first chess computer Belle achieve a master-level player status. And it was not until 1997 did the first chess computer beat a grandmaster player, Gary Kasparov. That chess computer as many of us may know, was Deep Blue [7]. However, the fact that Deep Blue was built with 256 VLSI special purpose chess processors may only be known to some of us. So how does parallel processing help in computer games? We shall find out.

2. Background Information

Before diving into the parallel methods of computer games right away, we need to establish the basic ideas behind solving computer games. First, the general approach to computer games is to build a tree structure of possible moves based on current board positions. The next layer (also known as ply) of nodes consists of possible counter moves to that, then counter-counter moves, and so on. The deeper the computer searches this tree, the farther it will be able to look and plan ahead. In fact, it has been shown that there is a strong correlation between the depth of the search and the level of play of the computer. An evaluation function is used at each node to give a numerical value for how "good" certain moves are versus others. The average number of branches (offsprings) per node, which is the average number of possible moves per turn, is called the branching factor. The larger the branching factor, the harder it is to solve the game. Othello is one of the easier games; the branching factor is only around five to seven. Thus, Othello programs have outplayed humans since the 1980s. Go, on the other hand, has a branching factor of around 300, and has been shown to be PSPACE-complete (Papadimitriou [6]). The branching factor is so large that given a machine that can search a million nodes per second, it will take 230 years to search just 6 levels deep – a true beginner's level.

Now let us look at some fundamental tree search algorithms. The three basic tree searching algorithms are Breadth-first, Depth-first, and Iterative Deepening. In Breadth-first search, the search starts at the root and expands one level deeper at a time. All nodes at one level are evaluated before any evaluation at a deeper level begins. Since it starts at

the root and work its way down, the breadth-first search is both optimal and complete. However, because of this, the space complexity is $O(branching\text{-}factor^{depth})$ since it has to keep track of all the nodes visited. The time complexity is also $O(branching\text{-}factor^{depth})$, and is the same for all search methods discussed in this paper. The only way to improve on the time complexity is through linear speedups by parallelization, to be discussed later.

For the Depth-first search (also known as backtracking), it always searches down one path to the deepest level it can go before it traces back up to search other paths, in a similar fashion. The advantage in doing this is that the space requirement is now only *O(branching-factor * depth)* since it only need to keep track of branches in one single path. The disadvantage is that it is no longer optimal and complete because if the solution on the very last path but is only a few layers deep, it may not reach it before time expires. Also, if a path is infinite, it will never get out of it unless we set a maximum depth for the search. This leads us to the next search method, the Iterative Deepening search.

The Iterative Deepening search is a modified version of the Depth-first search. It does a depth-first search by setting the depth limit to 1, then 2, and so on iteratively (hence the name). The nice thing in doing so is that it is now both optimal and complete, and requires a space bound of only *O(branching-factor * depth).* The drawback to this approach is that since memory is not kept for the previous unsuccessful searches, it needs to re-expand the nodes the second time it visits them. There are researches being done on how to avoid re-expanding all visited nodes while keeping the space bound relatively

close to *O(branching-factor * depth),* however, this is not the focus of this paper and so it will not be discussed further.

Now that we have established the basic search methods, we will go over the tree structures and search methods used in game-tree searches. First is the minimax game tree. Below is an example of such trees:

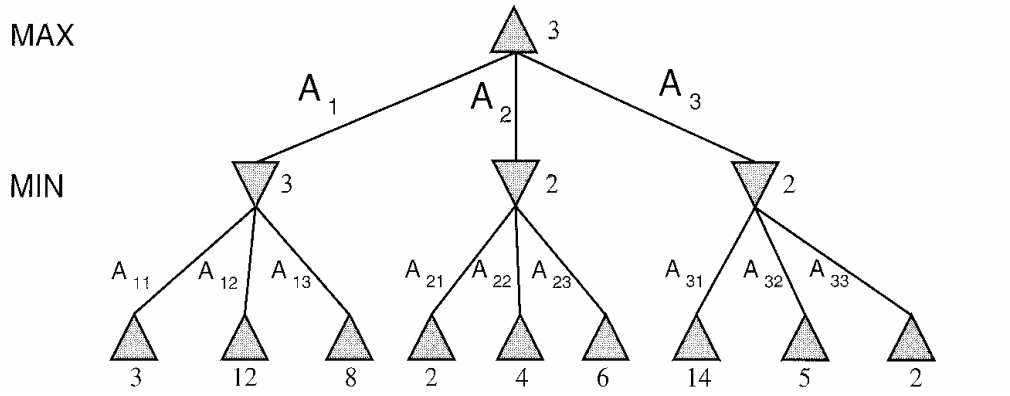

Fig.1. A minimax tree

The minimax game tree is generated based on moves applicable for positions on the game board. An evaluation function determines the value of possible moves for both players. In the figure above, A_1, A_2, and A_3, represent paths to moves that the computer can make, and 3, 2, and 2 are the corresponding values returned by the evaluation function for such moves. These three values are calculated based on the fact that you know your opponent will minimize any subsequent moves you will make. For example, if the next move you make is A_1, your opponent has three possible counter moves, A_{11}, A_{12}, and A_{13}. These moves are based on the assumption of the counter-counter moves you will make against each of the counter moves. The reason this is called the minimax

game tree is that the tree is constructed base on the fact that you will always try to obtain the maximum possible value for moves while knowing that your opponent will counter with the minimum. Looking at the diagram again, since you know the opponent will select the minimum value from your possible counter-counter moves, it is much wiser to select path A_1 since that can lead to a maximum value of 3 rather than 2 from the other two paths.

So, what can we do with the minimax game trees? One very important tool in game tree searches is pruning. Pruning is the process of eliminating the need to search an entire sub tree based on values given by the evaluation function. The pruning of minimax trees has a special name, called αβ Pruning, or more commonly, αβ Search. The following diagram best shows this pruning process in αβ search.

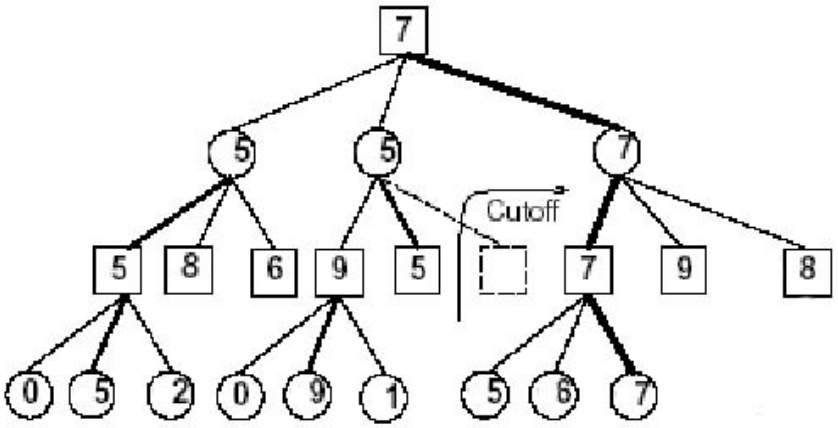

Fig. 2. αβ Pruning

Most αβ search algorithms use either the iterative deepening search method or just the depth first search with a predefined maximum depth. Let's trace the above

minimax tree with depth first search with maximum depth of 4. The squares indicate values return from a maximum evaluation and circles indicate values from a minimum evaluation. The search starts down the leftmost path down to the deepest level. At that node, it sees a 0, 5, and 2; naturally it selects the maximum value of 5 (notice that the selected paths are medium **bold**, and later, the final path selected is **heavy bold**). The evaluation of opponent's moves in that level returns the 5 we just found, along with an 8 and a 6. Thus our opponent will select the minimum value of 5. Now it searches down the second path and does the same evaluations. The minimum evaluation sees the 9 and the 5 and does not need to search any further since the entire second path is deemed useless because it gave the same (or less) evaluation value as the first path. Thus, all further searches which may be needed are pruned, or cutoff-ed. Now it searches down the last path. The first return value it sees is the 7, which is good; but there may be smaller values than a 5 still ahead, so it needs to search further. It then sees the 9, so further search is necessary. It finally sees the 8, and the search is done. We have a winner! Our minimum evaluation will select the 7 and is thus a better move than the 5 from the first path. Note that if the 9 from the last search was a 5 (or less) instead, we would cutoff any further search.

Because of the way most of the computer games of interest are played against an opponent, the $\alpha\beta$ search is the most commonly used basic search for further customization. And of course, parallel game algorithms are of no exception.

3. Parallelizing the $\alpha\beta$ search

After analyzing the game tree (namely the αβ) search algorithm, one can probably guess that trying to parallelize it poses some problems. First, because of the unpredictable nature of the where pruning occurs; the node to divide up the task for good load balancing is difficult to determine. Search overheads (unnecessary searches to paths which does not help the overall evaluation or double searching of nodes which will be searched by another parallel process) will occur if the parallelism is not managed properly. Some sequentially tailored methods such as iterative deepening and αβ enhancements may need to be reworked. A good method to share of the transposition table – a hash table to store the searched results, is also needed. These are just some of the problems encountered when parallelizing the game tree search. The rest of the paper will try to address some of these issues.

3.1 YBWC

It has been found that cutoffs occur frequently after the evaluation of the leftmost node of sub-trees. Knowing this fact, we can avoid unnecessary overhead of communications between the processors if the distribution of searching the sub trees among processors wait until the leftmost node has been evaluated. The YBWC – Young Brothers Wait Concept described in detail in [1] and [2] is one of the first parallel game tree search algorithms that utilize this key information. It is very well known and many parallel game tree algorithms are based on it.

The key idea behind the YBWC is to wait for the leftmost node (the eldest child branch) to be completely evaluated before the rest of the children (the younger brothers to the eldest node, hence the name) are allowed to be considered for parallelization. The

decision of whether or not to parallel search sub-trees of a node is determined by the processor evaluating that node, which is based on the information available at that node. Notice that because of the way YBWC works, parallelism does not start until the master processor traverses down the leftmost path to the deepest level. Only after it evaluated the parent of that leftmost leaf, can it start to divide up the work of the sub problems (searching the sub trees) to the other processors.

It is interesting to note that since there is a high probability of sub trees being pruned after evaluating the leftmost node, load balancing should be one of the key benefits of YBWC since we now have a better estimate of the load distribution of the search tree. But as it turns out, even though we gained an upper hand in load balancing from better estimate of load distributions, we lose it back elsewhere. For example, let's consider now the sub problems which are near the lower depths of a game tree. The computation of these sub problems are not too intensive since they are already near the lower depths and will therefore terminate very soon. These sub problems are so small and few that many of the processors are left idle. Thus we have inefficient use of parallel processing power at the lower depths of the search tree, which can offset (or in many cases, more than offset) the gains we had in the better estimation of load distribution.

The nodes mentioned above where the decision is made to distribute work amongst processors are called synchronization nodes (it needs to wait for all sub problems to finish, hence the name synchronization, before returning the information to the parent node). Below is a diagram showing possible synchronization nodes. Because of the way these synchronization nodes cause idle processors at the lower levels, as

discussed above, researchers have been thinking of ways to overcome this. This brings us to the next parallel game tree search topic, the APHID.

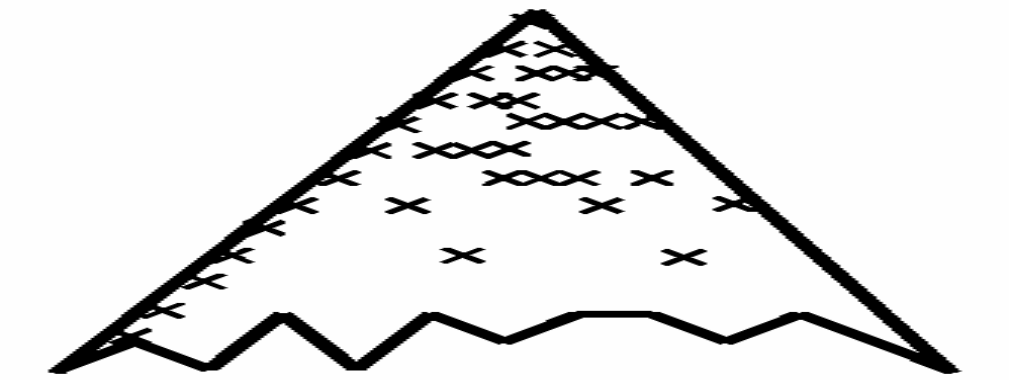

Fig. 3. Synchronization nodes in the YBWC

3.2. APHID

In the APHID – Asynchronous Parallel Hierarchical Iterative Deepening [3], the master starts off and search the d' ply levels of the game tree sequentially (see diagram below). Each of the slaves then gets a portion of the master's leaf nodes to expand. The slaves' work load is d – d'. The value of d', or the amount of work the master handles, is dependent on the structure of the game tree.

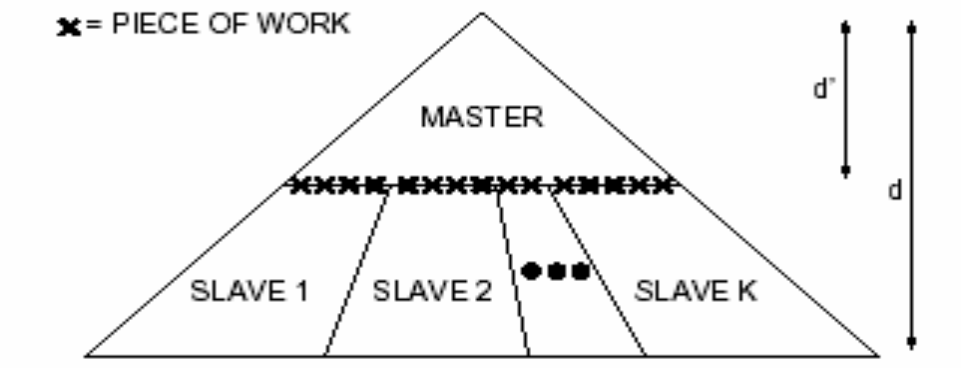

Fig. 4. APHID Parallelization

One may look at diagram above and notice one key element of this algorithm. If you consider the workloads for the slaves(x's in the diagram) as synchronization nodes, you can see that they are much fewer and nicely spread out in one level. This makes it easier to manage and distribute work amongst the processors. In fact, these x's are really synchronization nodes since they are the nodes in which their parents are waiting for the evaluation of their results. This brings up one question I was pondering – by looking at the diagrams, one would think (well, at least I did) that this shouldn't this be the synchronous and the YBWC be asynchronous instead? It sure looks as if it should be the reverse, given the way the synchronization nodes are all over the place in the YBWC and nicely placed and managed in this method. Well one way to look at this is that in the synchronous method of YBWC, the synchronization nodes require the parallel processes for that node to finish their evaluation and "synchronize" their finding before that node can return the evaluation to its parent node. Whereas in the asynchronous method here, each parallel processor can go off and compute its sub problem independently, without needing to "synchronize" their findings with one another. The master then updates its evaluation based on the number of slaves return their findings. If by the time the time limit expires, and not all slaves have return their findings, the master will just return its evaluation based on what it has gathered thus far from the slaves. Of course, there are ways to "remind" the slaves or have them keep track of time to make sure they do return whatever they have found.

One of the benefits of the slaves working on their own problems independently is that if a slave is finished with its work and the master does not have extra work for it to do, it can go ahead and search an extra level deeper. As mentioned earlier, the strength of computer games has a strong correlation to the search depth. However, it is usually more beneficial for the master to hand off work from more load intensive slaves to a lesser one, than having some of the slaves search extra levels while some other slaves are unable to finish its work. However, because of the way APHID works, there is no easy way of doing this. The best method to obtain near optimal load balancing is for the master to better distribute the work loads. One way of ensuring this is to have a finer granularity of sub problems for the slaves. Thus instead of having all sub problems determined in a single level, the master can go down an extra level in paths where it thinks they are more work intensive and sub divide the work load in that level. However, as it turns out, finding a good estimate of where this lies is not easily determined. Thus, even though no processors are ever idle, they are nonetheless not as efficient as if they are nicely load balanced just most of the time. So in actual results, the APHID did not perform as well as YBWC or other synchronous methods for game trees which as less balanced. Below is a diagram that best visualizes the extra level the master evaluates.

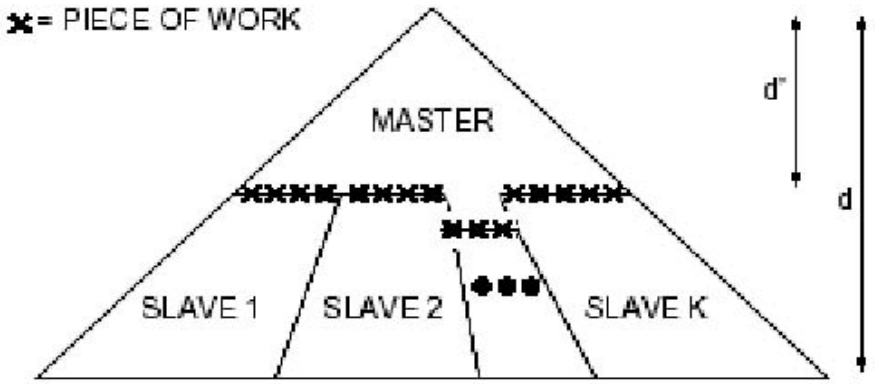

Fig. 5. Improved work load in the APHID

There is one other key benefit to the APHID. Since both the master and the slaves are really performing sequential searches, the APHID can be added to existing sequential programs effortlessly. According to [3], it can actually be incorporated into existing codes in just a day and fine tune it in a few days.

4. Results

Below are some of the results found in [3], of the APHID running on four different game programs.

.

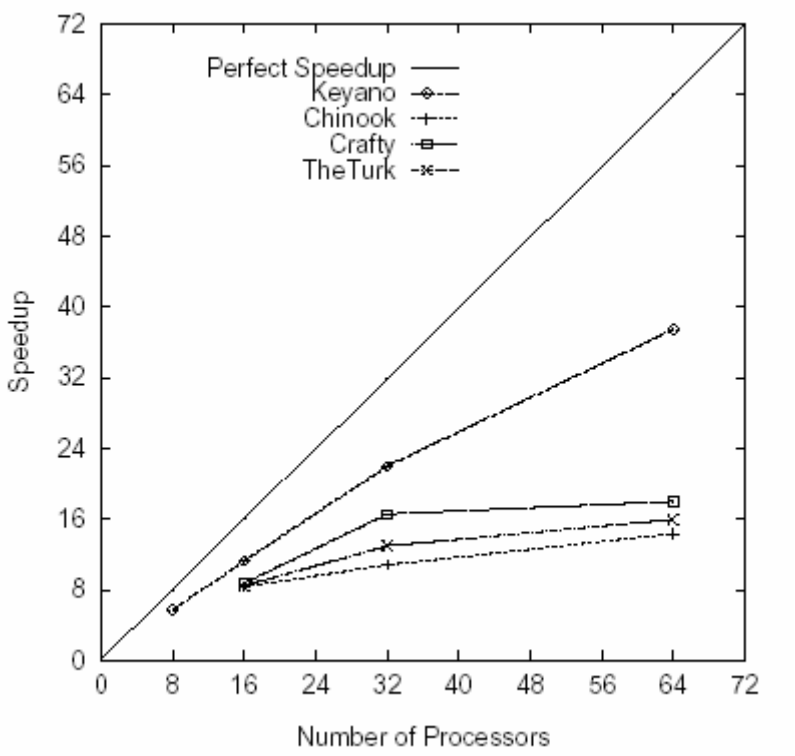

Fig. 6. Speedup vs. Number of Processors for APHID

Keyano, the top line in the graph, is an Othello game program. Because of Othello's low branching factor, load balancing by the APHID can be easily optimized and thus had an almost perfect one to one speedup vs. the number of processors. The middle two are Crafty and The Turk, both Chess programs. The speedup here is only around 12 to 16, for 32 to 64 processors. The reason is mainly due to the high branching factor of chess, giving a not so balanced game tree and thus not let APHID do its magic. The bottom one is Chinook, the world champion of checkers since 1994. Its result is similar to Crafty and The Turk.

Below is a chart comparing the performance of APHID vs. YBWC for Keyano. As mentioned above, the key attribute that the APHID was able to take advantage of is Othello's low branching factor to make it perform better than the YBWC.

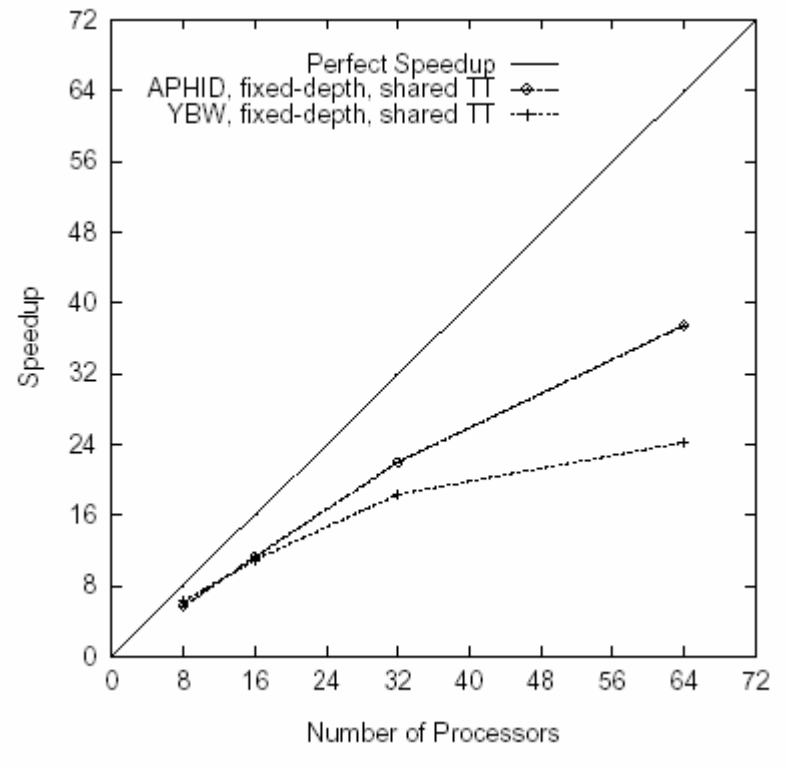

Fig. 7. APHID vs. YBWC Speedups

Below is the performance of the YBWC. As discussed before, the YBWC makes poor use of parallel processors at lower depths. Thus if the depth of the tree is only four to six levels deep, essentially all the levels can be considered lower depths, thus the YBWC does not gain much speedup unless the game tree is large.

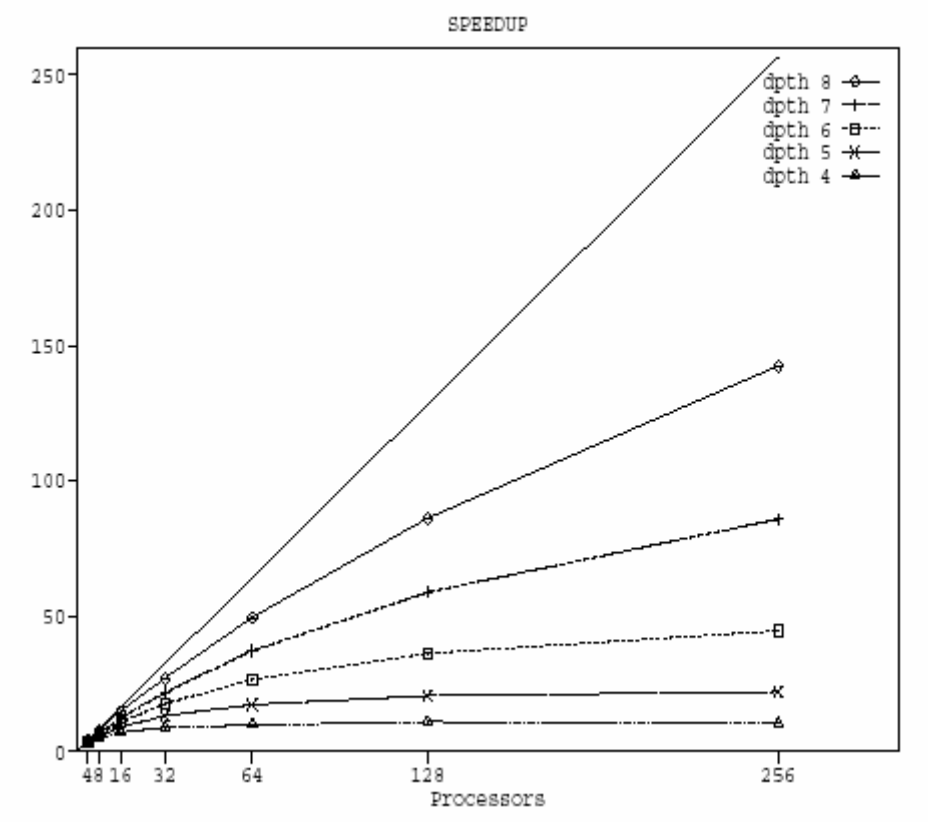

Fig. 8. Speedups for YBWC

Much more detailed data and analysis can be found in [3] for the APHID, and in [1] and [2] for the YBWC. There are also some very informative survey results for other parallel game algorithms in [4].

5. Conclusion

The main focus of this paper was to study the parallel algorithms used in computer games. The two most dominate methods, the Young Brothers Wait Concept

and the Asynchronous Parallel Hierarchical Iterative Deepening, representing the two different approaches to the parallelization was presented. The YBWC had inefficient use of parallel processing power since processors are sometimes left idle. The APHID tried to correct this by having no processors idling at any time, but failed to outperform the YBWC method in tougher computer games because some processors are doing extra searches at deeper levels while others are unable to complete its task. However, if a good load-intensive paths estimator exists, the APHID should theoretically outshine the YBWC. This makes the APHID a better choice for further studies for improvements.

Bibliography

[1] *Studying overheads in massively parallel MIN/MAX-tree evaluation*
Feldmann, R; Mysliwiete, P; Monien, B; Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures, 1994

[2] *Game Tree Search on a Massively Parallel System*
Feldmann, R.; Doctorial Thesis, University of Paderborn, Germany, 1993

[3] *The APHID parallel αβ search algorithm*
Brockington, M.G.; Schaeffer, J.;
Journal of Parallel and Distributed Computing, 1999

[4] *A Taxonomy Of Parallel Game-Tree Search Algorithms*
Mark G. Brockington; ICCA Journal, 1996

[5] *Artificial Intelligence – A Modern Approach*
Russell, S.; Norvig, *P.;* Prentice Hall, 1995

[6] *Computational Complexity*
Christos H. Papadimitriou*;* Addison Wesley, 1994

[7] http://www.research.ibm.com/deepblue