Speculative Parallelism Improves Search?

T.A. Marsland and Yaoqing Gao

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Technical Report TR 95-05

April 10, 1995

Abstract

The extreme efficiency of sequential search, and the natural tendency of tree pruning systems to produce wide variations in workload, partly explains why it is proving difficult to achieve more than 30-50% efficiency for massively parallel implementations of the $\alpha - \beta$ algorithm. Here we introduce typical enhanced sequential algorithms and address the major issues of parallel game-tree searching under conditions of severe pruning. It is this pruning that makes the parallelization difficult. After examining previous work on parallel $\alpha - \beta$ algorithms, we present a new method called Dynamic Multiple Principal Variation Splitting (DM-PVSplit) and implement it on the AP1000. In this algorithm, high performance is achieved by using some novel approaches: Parallel speculative search of candidate principal variations is used to reduce re-search delay and so obtain more quickly a better estimate of the subtree value. This is achieved by configuring a flat processor arrangement as a dynamically changeable tree structure. Also, with the aid of a group-based scheduling strategy, the game tree is split dynamically at different levels. This provides better load balance and takes more advantage of parallelism. Preliminary experiments show that the scalability of the DM-PVSplit algorithm is good for massively parallel machines.

Keywords: $\alpha - \beta$ search, Parallel tree splitting, Dynamic Multiple PVSplitting, Massively parallel processing.

1 Introduction

With advances in VLSI technology and parallel architecture, systems consisting of a thousand or more processors are coming into commercial use. This has led to an upsurge of interest in parallel processing approaches to improve the efficiency of tree search problems that seek one or more optimal or sub-optimal solutions in a defined problem space. Two person games is one domain where a special kind of branch-and-bound algorithm is commonly used to reduce the search space. Although much effort has been put into further improvement of sequential methods, such as use of memory functions, selective extensions and the null move, parallelization offers further performance improvement. The level of speedup depends mainly on the design and choice of the parallel algorithm, and this involves trade-offs in search time, memory space and communication. Because the powerful pruning mechanism leads to highly variable tree sizes that conflict with a uniform workload requirement for good performance, $\alpha - \beta$ search has proved difficult to parallelize. Even though various approaches have been tried, including parallel window search, search tree partitioning, and principal variation splitting, the potential remains for significantly better parallelization methods. For massively parallel systems, the exisitng approaches exhibit only acceptable efficiency both in practice [5] [13], and by simulation [10]. Here, we address the issues of parallel game tree search and present a new parallel algorithm, called Dynamic Multiple Principal Variation Splitting (DM-PVSplit). Our algorithm first defines a critical-node set and spawns those nodes for parallel search. It then splits the other branches dynamically when the branching conditions are met. This speculative search is used to reduce synchronization overhead and re-search delay.

Before discussing major parallel search issues, we first examine important enhancements to the sequential search method, and provide a brief review of the existing work on parallel $\alpha - \beta$ algorithms. In Section 4, emphasis is put on a new parallel algorithm, and this is followed by results from some preliminary experiments. We conclude that although game-tree search on massively parallel systems is a challenge, there is evidence that better systems are possible.

1.1 Enhanced Game-tree Search

A two-person game-playing process is usually modelled by a tree, where a node represents a position and a branch corresponds to a move. A program starts with the current position and generates all legal moves, all legal responses to these moves, and so on until a leaf node, or a horizon node at a specified depth is reached. At each horizon node, a heuristic evaluation function assigns a value to that position. The aim of the search is to find the most desirable continuation under the assumption of best play by both sides. A true minimax search of these approximating game trees is simple, but expensive. The $\alpha - \beta$ pruning procedure is most effective at reducing the game tree search-space. It maintains a *search window* (α, β) that causes the elimination of subtrees whose value cannot affect the minimax value of the root.

Indeed modern chess programs have such good move ordering mechanisms (including iterative deepening and the use of hash-transposition tables) that they search within a factor of two of the minimal game-tree size. For a uniform tree of width w and depth d, Knuth and Moore [12] quote that there are $w^{\lfloor \frac{d}{2} \rfloor} + w^{\lceil \frac{d}{2} \rceil} - 1$ horizon (terminal) nodes in the minimal tree. They also labelled the nodes in that tree as being of type 1, type 2 and type 3, and these are PV, CUT and ALL nodes respectively [15]. For the minimal tree the type 2 (CUT) node has exactly one successor, but in actual trees a few successors. It has been shown that an average tree has six different types of node [18], but the inherent structure is more difficult to represent graphically. Although we show in Figure 1 the structure of the minimal tree, we have in mind during our discussion a typical search tree.

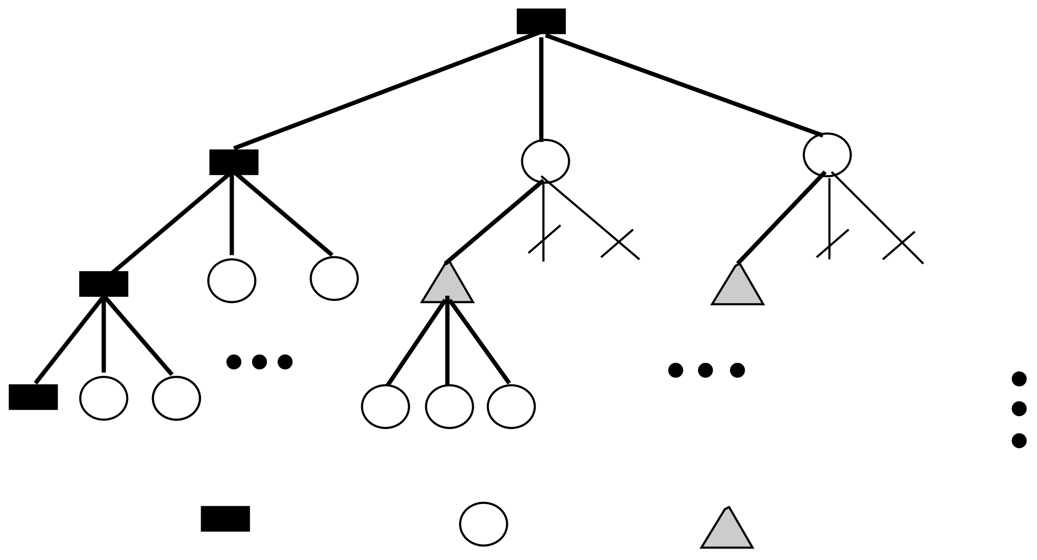
Figure 1: Structure of a minimal $\alpha - \beta$ search tree

There are several enhanced variants of the $\alpha - \beta$ algorithm. They aim to traverse a tree that is as close as possible to the minimal size (function ABS in Appendix A shows the underlying structure of the algorithm). One of them, *aspiration search* [2], artificially narrows the search window and gambles that, even with this reduced search space, the minimax value of the root will fall into this window. An improvement to aspiration search is the fail-soft $\alpha - \beta$ algorithm [9], where a tighter bound based on the value returned from the previous failed aspiration search is used to research the tree. If $minmax(v)$ is the minimax value of the node v, and R be the value returned by a fail-soft $\alpha - \beta$ search, the following inequalities hold [9]:

$$\begin{aligned} minmax(v) \le R \le \alpha \quad & if \quad minmax(v) \le \alpha & : fail - low \\ minmax(v) = R \quad & if \quad \alpha < minmax(v) < \beta & : success \\ minmax(v) \ge R \ge \beta \quad & if \quad minmax(v) \ge \beta & : fail - high \end{aligned}$$

for any search window (α, β).

To simplify difficulties with failed searches, the concept of a null window was introduced, to form *Principal Variation Search* (PVS) [14]. Its fundamental idea is that alternatives to the best move found so far are assumed inferior until proven otherwise. In practice some variation of PVS (shown in Appendix B) is used in conjunction with powerful knowledge-based enhancements like transposition tables, the history heuristic, iterative deepening and the null move, as described in typical encyclopedia articles.

2 Parallel $\alpha - \beta$ Search: A Brief Overview

2.1 Major Issues

The objective of a parallel algorithm is to achieve a speedup over the best available sequential algorithm, but in a way that is proportional to the number of processors used.

It is obvious that the main source of improvement comes from searching different parts of the game tree at the same time. The simplest parallelization that comes to mind is direct tree splitting. A game tree is divided into several subtrees which are assigned to independent processors. Clearly it is inferior on theoretical grounds, because many resources are devoted to fruitless parallel searches without the benefit of the guidance provided by an improving bound. Thus, direct tree-splitting has a high probability for anomalously poor parallel speedup, where the parallel program visits an impressive number of nodes, but still does not search more deeply than a single processor running the $\alpha - \beta$ algorithm.

On the other hand, all the benefits of pruning can only be achieved by fully searching one subtree in order to establish a bound for the search of the next subtree. If one adheres to the standard algorithm in an overly strict manner, there may be little opportunity for parallelism. Thus parallelizing the $\alpha - \beta$ search involves controlling the following key factors:

- Search overhead: the extra effort that a given parallel algorithm does in comparison to a sequential one. This comes down to the issue of how to exploit parallelism without excessive duplication of search.

- Communication and synchronization overhead. When a game tree is searched in parallel by several processors, a *communication overhead* arises when the better pruning information is exchanged, or when distributing work among processors. When a processor is waiting for other processors to finish their work before being able to continue, *synchronization overhead* is incurred. The latter overhead reflects poor work balance and measures excessive processor idleness.
- Bookkeeping overhead. The management and control of tasks and where they are located must be simple.

2.2 Previous Work

In Baudet's [2] parallel aspiration search method, the search window is subdivided into a multitude of narrower disjoint ranges. Each processor searches the tree with a different, non-overlapping range. Because the best continuation is guaranteed to fall in one of these ranges, this algorithm is faster than a sequential one with a full window. It also has the advantage of low communication overhead and has no need for synchronization. But the companion theoretical work shows that the maximum speedup is limited to a factor of 5-6, even with unlimited processors searching a randomly ordered tree. Baudet's method gets no benefit from the structure present in well-ordered trees.

Akl, Barnard and Doran [1] introduced the "*Mandatory Work First*" algorithm. By searching critical nodes first, the algorithm attempts to achieve many of the cutoffs that occur in the serial case. The algorithm has low search overhead, but suffers from the heavy memory requirements needed to store information about partially evaluated subtrees.

Finkel and Fishburn introduced a parallel $\alpha - \beta$ algorithm on a tree of processors. The root processor evaluates the root position. Each interior processor evaluates its assigned position by generating the successors and queuing them for parallel assignment to its slaves. The leaf processors search the subtrees rooted at its assigned position with the sequential $\alpha - \beta$ algorithm. When a processor finishes, it reports the subtree value to its master. Such a naive tree splitting algorithm results in high search overhead due to lost cutoffs. The theoretical model predicts that at least order of $\sqrt{N}$-fold speedup are achieved for N processors. A speedup of 5.31 was achieved by simulation on a 27 processor system configured as a processor tree of depth 3 and width 3 [8].

The basis for the most popular and successful parallel algorithms now in use appears to be the *Principal Variation Splitting* (PVSplitting) approach [14]. It was developed for the first parallel chess machines: *Ostrich* [17] and *Parabelle* [15]. PVSplitting is based on the assumption that game trees can be well ordered by state-of-the-art heuristic knowledge about the application domain. It concentrates on searching the first path along the principal variation before searching the other branches. But PVSplitting has the major disadvantage that only a single processor is available to search replacement principal variations, and during much of that time all other processors are idle, since a static processor assignment is made. It is this static processor allocation mechanism that leads to load imbalance.

Later more dynamic processor allocation schemes were developed. Notable are Schaeffer's *Dynamic PVSplit* algorithm [20] and the *Enhanced Principal Variation Splitting* (EPVS) method of Hyatt et al. [11]. Schaeffer's distributed search introduced dynamic work assignments. In his algorithm there is one *Controller Process* to allocate work to *Searcher* processes that use *PVSplit* to do the searching. When the *Controller* has no other work to allocate it reassigns an idle *Searcher* to help others. The problem with this algorithm is that the central *Controller* may become a bottleneck. Schaeffer's experiments [20] show that 9 processors can achieve 5.67-fold speedups, but beyond that he believes not much more is possible, suggesting that the approach is not suitable for massive parallelism.

The EPVS algorithm tries to identify the node types in the tree and use that knowledge to reduce search overhead and minimize the problem of load imbalance. When a processor finds no additional work to do at the current divide node, it assumes that the remaining busy processors are searching complex branches and sends a signal to them. The sequential search then advances two plies deeper before the parallel division is restarted. A hash table (to recognize move transpositions) prevents the set of processors from re-examining branches within this subtree that were examined by a single processor before it was stopped. The EPVS algorithm is programmed for use in a high performance parallel computer chess program on a Cray X-MP. Interestingly, Hyatt's 1988 PhD thesis contains an even more efficient method, Dynamic Tree Search (DTS), that seems to be comparable to Feldmann et al.'s Young Brothers Wait (YBW) concept [4] in efficiency, but is for a shared memory multiprocessor.

The hash table plays an important role in all these algorithms, but especially in Felten and Otto's [6] dynamic parallel game search program on a Hypercube. As with the sequential methods, if a move at a node is found in the hash-transposition table, the corresponding successor is searched first. After the evaluation of the first successor is complete, all other nodes are evaluated in parallel. If there is no transposition table entry available for some node, all the successors may be searched in parallel. The speedup by the *Waycool* chess program is 101 on a 256-processor hypercube. Part of the success of this method is attributed to the bigger distributed hash-transposition table that the hypercube made possible.

On the other hand, the two important features of Feldmann et al.'s algorithm [4] are the distributed search control and the *Young Brothers Wait* (YBW) concept. In this algorithm, each searcher executes the same search and control code. When it is idle, a searcher requests work from other searchers. The aim of the YBW method is to reduce superfluous search overhead. Parallel search of inner game nodes is started only after the first successor of the inner node is evaluated completely. This algorithm has been used in *Zugzwang* on a 256-transputer array, and then on a 1024-processor array [5]. A good speed-up of 350 was obtained, but this figure may be a function of the relatively slow speed of the processors, since the working overheads are then a smaller fraction of the elapsed time.

3 Parallel Search: DM-PVSplit Algorithm

3.1 A Closer Look at Principal Variation Splitting

In practice, the distribution of values at terminal nodes in a tree is usually not random. Especially with the guidance of domain-specific knowledge, the best move can be determined with a high probability of success. For a chess game tree, it was estimated in 1980 that the first branch is best with 70% probability of success and the best move is in the first quarter of the branches with 90% probability of success [14]. With newer move ordering heuristics, these probabilities may now be higher, so that today's game trees are even better ordered. The Principal Variation Search (PVS) algorithm takes advantage of the observed game-tree structure. With PVS, the leftmost path is examined first, and the bound obtained is used to prune the search of the alternative choices. The PVSplitting algorithm is a straightforward extension of PVS to support parallel search. Tree decomposition is delayed until the first path along the principal variation has been searched. After a revised bound is obtained, the remaining branches are distributed among the available processors. It is known that the major problem with the PVSplitting algorithm is the necessary synchronization at the PV nodes. Also each null-window search is carried out by a single processor alone. Ultimately, all other processors may be idle for a considerable time waiting in case an improved bound is provided by the remaining processor (as will be the case if it is now searching a new principal variation). In this static processor allocation method, such load imbalance causes heavy synchronization overhead.

Because PVSplitting is a depth-first algorithm that creates parallel work by splitting nodes along the principal variation, it has negligible memory needs and low search overhead. But the algorithm has two disadvantages: First the performance depends on how well a game tree is ordered, and second the low search overhead comes at the cost of increased synchronization.

3.1.1 Null-window parallel search:

Aspiration search (using a heuristically narrowed window) is also popular, because in general the narrower the window the faster the search. However, Principal Variation Search employs a null window (smaller even than the minimal aspiration window) to simply refute the merits of a subtree. Figure 2 compares the ratio of nodes visited by a null window and a full window search on strongly ordered game trees with different width and depth parameters. Its characteristic shape has been referred to as the "minimax wall". But Figure 2 also shows that a null window search which fails high (the left half of the graph) still visits $15 - 25\%$ of the total nodes seen by the corresponding full window search. When a null window search fails high, a re-search is necessary. The size of a null window search plus later re-search with a wider window may be higher than that of a single search on the subtree with a correct window. However, the use of a transposition table significantly reduces that total cost.

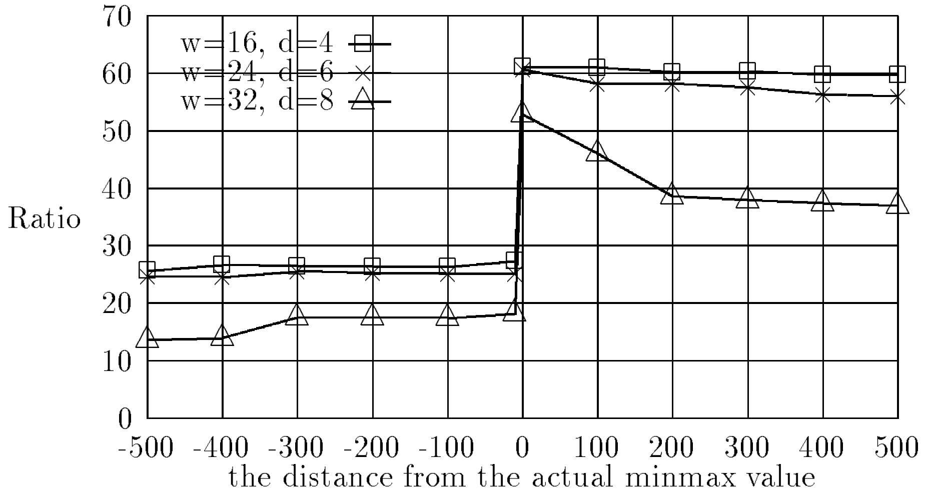

Figure 2: Relative cost of a null window search on strongly ordered trees

On the other side, fail-low null window searches can produce anomalously bad behavior when searched in parallel. For example, consider Figure 3 which illustrates how several *parallel* null window searches can "incorrectly" fail high, causing re-searches, even though the best move is actually in the first quarter of the subtrees at the root. This is our explanation why unexpectedly bad speedup is seen in some parallel game-tree search examples.

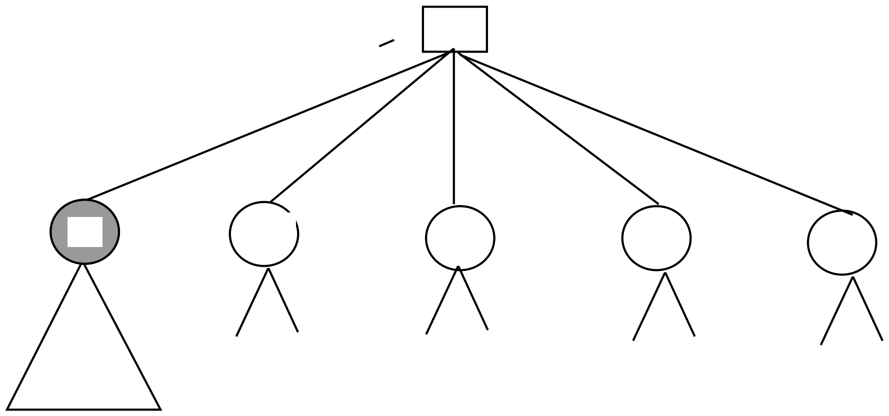

Figure 3: Parallel re-search anomaly

Based on our experiments and analysis of other related algorithms, we now provide a detailed description of dynamic multiple principal variation splitting, a method which attempts to correct this flaw.

3.2 Multiple Principal Variation Splitting

Design of a better parallel algorithm is crucial to the efficient search of a game tree on massively parallel machines. The algorithm we present here is called the *Dynamic, Multiple Principal Variation Splitting* algorithm (DM-PVSplit). It also relies on the assumption that trees are well-ordered. That is, the leftmost branch is the best move with a high probability of success and all branches to the right of the leftmost one are also well ordered.

3.2.1 Searching multiple principal variations in parallel:

The hypothesis is that the first few successors at a type 1 node are candidate principal moves, and hence are especially hard (time-consuming) to refute even with a null-window search. Thus they may as well be searched in parallel. As soon as one of them generates a bound it is shared with the other candidates. We anticipate that the best move almost always lies in this portion. Therefore, at each type 1 node the successors are divided into two sets: one is called its principal variation set (PV-set), which consists of the "first/best" move plus a few more promising ones (specified shortly), and the other is called its inferior set where a null-window search is applied.

In our algorithm, the PV-set is searched in parallel first, instead of only the leftmost branch, as done by both the PVSplitting algorithm and the Young Brothers Wait method, as if every branch in the set is a principal variation. The size of the PV-set is matched to the *search width* and is determined by the probability distribution of all the possible moves at a position. The search width, s, is defined as follows: let v be the root of a game tree and $v.i$ $[1 \leq i \leq w]$ be all the successors of v, let $P(v.i)$ be a success probability that $v.i$ is the best move, and let ϑ be the expected success probability that the best move falls into the set. The search width (s) of v is the size of the set where the following equation holds:

$$\sum_{i=1}^{s} P(v.i) \geq \vartheta \text{ and } \sum_{i=1}^{s-1} P(v.i) < \vartheta$$

This general model provides all the necessary scope for heuristic variations. The probability function will be application dependent and will reflect confidence in the evaluation function in this region of the tree. Also the probability function can control the speculative search width s so that it may be narrower in the right half of the tree, where the probability of a new variation emerging is much lower, while deeper in the tree s can be wider, because the cost of search is so much lower and uncertainties from the quiescent search at the horizon greater. For simplicity, in some of our preliminary experimental results s was simply held constant, to provide a basis for comparison.

Thus, at each PV node, the subtrees rooted there are partitioned and the PV-set is searched by several processors with a normal window. Most effort is put into the PV-set, since the best move is expected to come from that set. The decomposition of the remaining subtrees is delayed until the paths along the multiple principal variations have been searched. A null window search is applied to these remaining subtrees and no re-search is expected. To speed searching the PV-set, once a better bound is found at a node in the PV-set, it is broadcast to other processors to update their search windows. Since we typically choose over 0.95 as the expected success probability ϑ at a PV node, almost all the remaining null window searches fail low. Therefore the granularity of searching each node in the inferior set is relatively uniform. The scheduling mechanism can take advantage of this to balance the work load among the processors.

There can be several variants of the multiple PVSplitting algorithm. For example, the partitioning and splitting can be restricted to levels within a pre-specified height from the root, where the parallel search will be most effective. However, when the search width, s, of the promising positions is always set to 1, the behaviour of the new algorithm is like that for PVSplitting, which might therefore be viewed as a special case of our new algorithm.

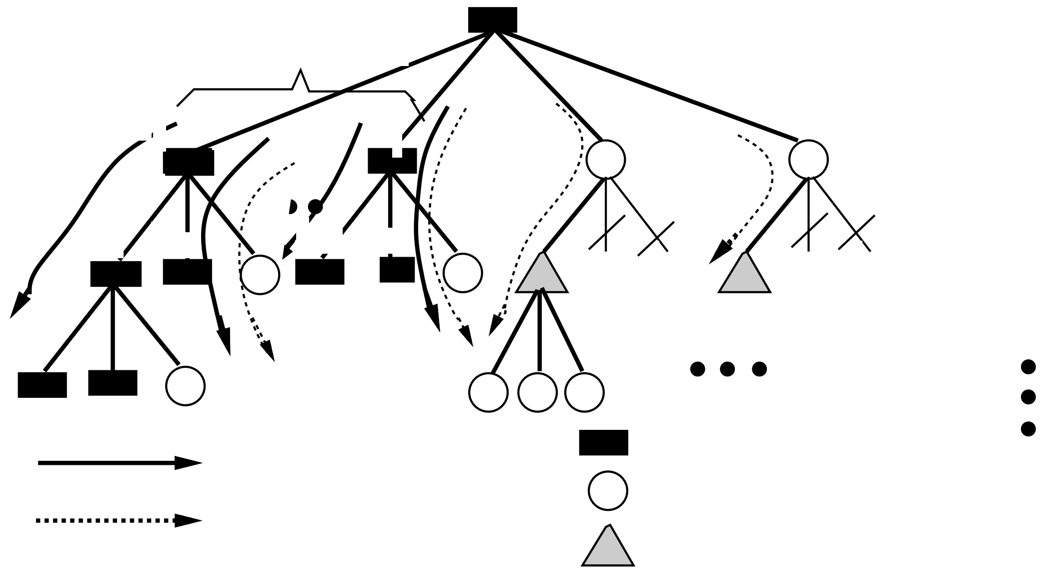

Figure 4: Multiple principal variation split-based $\alpha - \beta$ search

3.2.2 Identify the node's type in a null window search:

Once a choice move has been found, and the evaluation of the PV-set is complete, decomposition of the remaining subtrees starts. The value of the choice move forms a null window to verify that the alternatives are inferior to the most plausible move found from the PV-set. The node types in typical subtrees are clearly identified in the PVS/NWS pseudo-codes shown in Appendix B.

Furthermore, to control re-search overhead, the information gathered from the null-window search of a subtree is used on a research to enable ignore-left cut-offs and prove-best cut-offs. The data from partially informed NegaScout [19] is smaller, but is more useful in our case because it can be distributed across processors.

The processes in the system can be classified into two categories: searching processes that aim at finding a better bound, and verifying processes that search current branches with a null window and try to prove them inferior to the best move searched so far, see Figure 4.

Since searching processes are speculative, multiple principal variation search is a promising approach for systems with many processors. Although DM-PVSplit may do more work than PVSplitting when we choose an unsuitable search width, these speculative searches provide a high degree of parallelism and eliminate some other re-searches. It is effective because the moves in the PV-set are all candidate principal moves, and hence may be especially hard (time-consuming) to refute. They may as well be searched in parallel since their trees could be as large as the principal variation's.

3.2.3 Group-based scheduling strategy:

To balance the loads in the system efficiently, we use a group-based dynamic scheduling strategy. The physical processors are virtually configured as a tree structure. During the search, the structure is reconfigured dynamically according to specified rules. Different from the approaches in Schaeffer's [20], Hyatt's [7] and Feldmann's [5] works, the physical processors are initially configured as one group, i.e. a tree structure of depth 1. The root is the master and all others are slaves. After a tree is assigned to the root processor, it first divides its slaves into 's' subgroups and then specifies a master for each group. After that, the subtrees rooted at the nodes in the PV-set are assigned to the subgroup of the master. Once a processor become a master, it works in a self-scheduled manner. A master not only waits for the answer from its slaves, and updates $\alpha - \beta$ bounds, but also schedules its slaves for best load balance. When none of the master's slaves have work, they all rejoin the group of the next higher master, so the size of groups at the same depth are not necessarily equal. More processors are assigned to the more plausible moves. This strategy combines local scheduling within a subgroup with global scheduling by rejoining. Obviously it avoids the bottleneck caused by central control, and imbalance caused by blind random task-stealing. It works remarkably well when the search is performed on parallel machines where the communication overhead depends on the distance between processors, because the group-based scheduling strategy benefits from data and computation locality.

4 Implementation and performance evaluation

4.1 Parallel search on the AP1000

Our algorithm is implemented on an AP1000, a large-scale message passing machine (see Figure 5). It consists of processor elements that are connected with three networks called the broadcast network (B-Net), torus network (T-NET), and synchronization network (S-Net). Each element is a SPARC IU with a FPU. A SUN 4 workstation is used as a host machine to control the AP1000. To reduce the communication latency, direct message sending from the cache of a processor, bypassing the DMA channel and main memory access, is supported and so is the reception of messages from the network directly into a circular buffer allocated in the user space. A variety of libraries support parallel programming, and especially for (asynchronous) non-blocking message passing. Since a wormhole routing technique is used, the communication latency does not depend on the distance between the processors and is determined only by the setup time of message passing and the size of messages. Experiments [21] have shown that the relationship between message size and point-to-point communication time can be specified as a linear function of message size N, i.e., $t_o + t_b \times N$. Here t_o is the setup time of $6.9us$, and t_b is the effective communication bandwidth and $0.069us/byte$. Since the size of a message in the game-tree search is small, the most time consuming part of message passing is the setup.

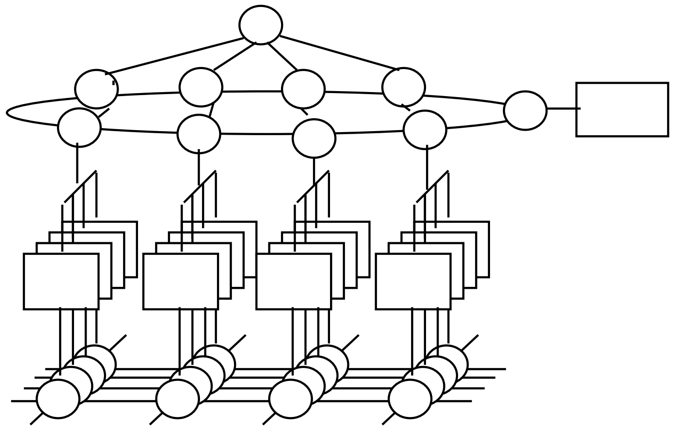

Figure 5: The architecture of the AP1000

To implement the DM-PVSplit algorithm, the processors in the AP1000 are configured as a dynamically changeable tree structure. To enable meaningful performance comparisons between different algorithms, we must implement different algorithms on the same machine and must guarantee that experiments are made in such a way that all the different search algorithms traverse the same game tree.

Although there are several different ways of creating game trees in the literature [16] [3] [19], we are more interested in an on-the-fly tree generation method [19] for our experiments, due to its flexibility and memory efficiency. Initially users can specify different weights which reflect the probability of the best move at all nodes in the tree (or at nodes at any level in the tree). Given the same width w, depth d and seed (for random numbers), the same tree can always be generated with the memory requirement $O(wd)$. By collecting the statistics about the average overhead for generating moves and evaluating positions in an actual game-playing program (e.g., for Chess or Chinese Chess), the method can be extended to generate a simulated game tree, with properties typical of the target game.

4.2 Results of Experiments

The trees used in our experiment are classified into three categories: weakly ordered, moderately ordered and strongly ordered, where the first move is best at any given node with the probability of 0.3, 0.5 and 0.7, respectively.

Since in general the search time is directly proportional to the number of the nodes searched, we first use ABS (a sequential version of the algorithm) as a basis for the performance of the DM-PVS algorithm.

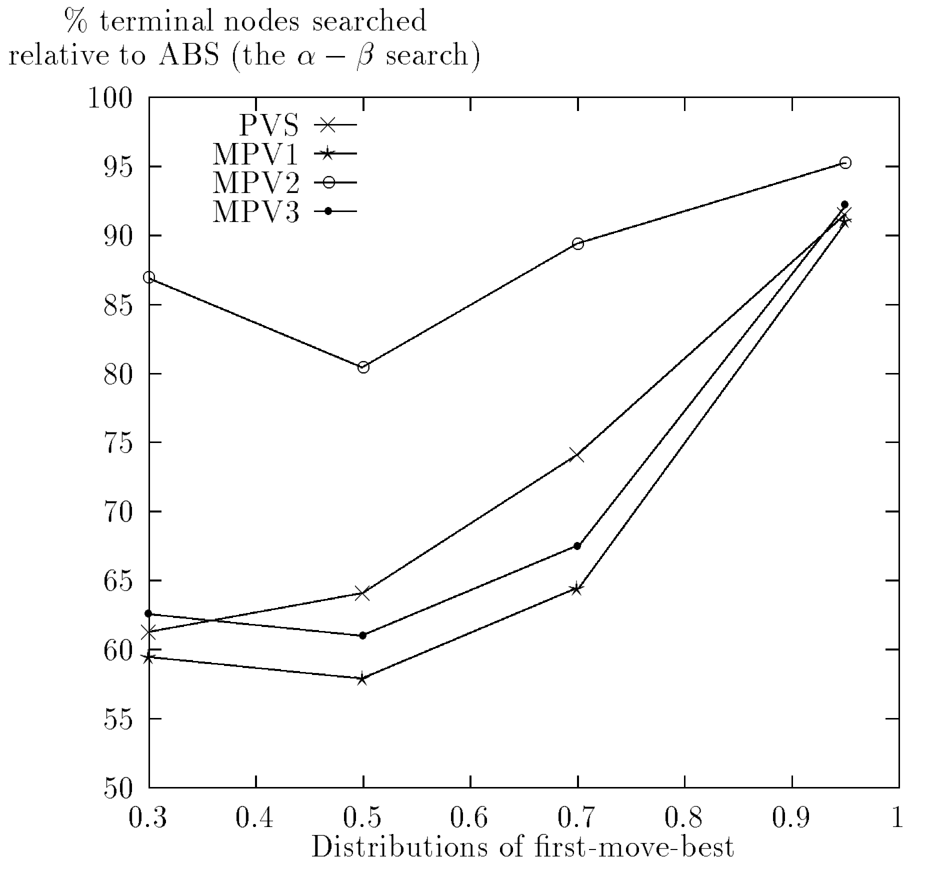

Figure 6: Relative performance on uniform game trees of width 16 and depth 8

Figure 6 and Figure 7 show the average number of terminal nodes visited during searches of 30 artificial trees using sequential versions of the following algorithms: standard $\alpha - \beta$ search (ABS), principal variation search (PVS), multiple principal variation search with information exchange among the PV-sets ($mpv1$ and $mpv3$) and without such exchange ($mpv2$). In the latter three cases all the subtrees rooted at the nodes in the PV-sets are searched by the current window. The difference between $mpv1$ and $mpv3$ is that in the

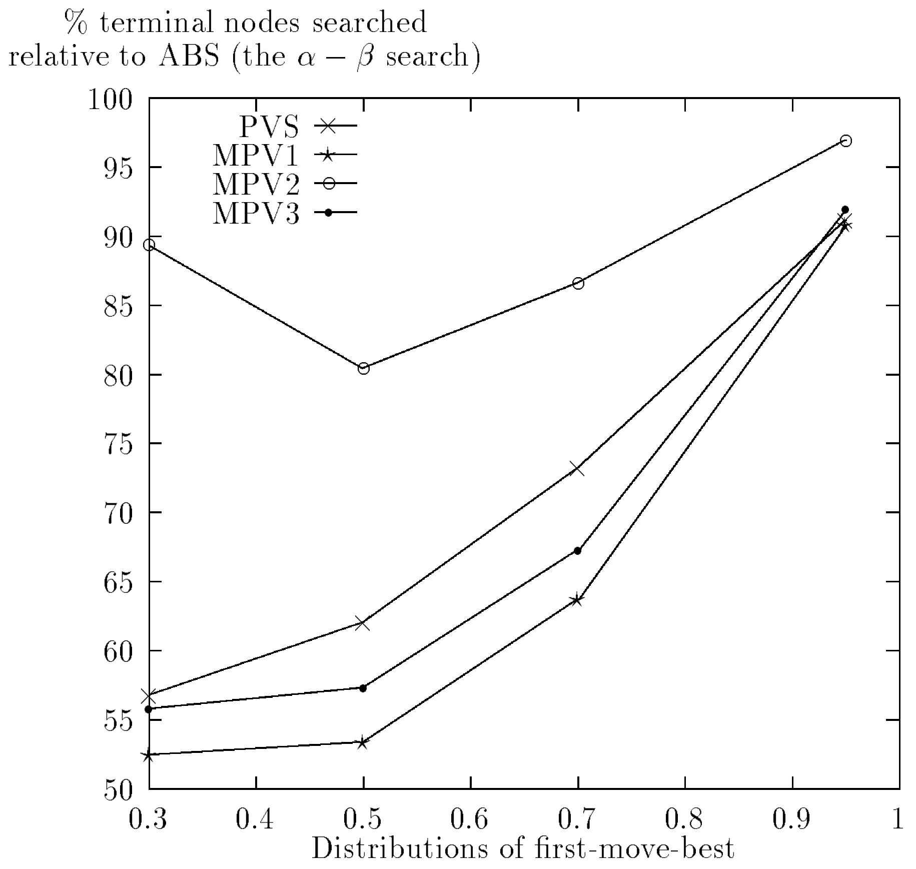

Figure 7: Relative performance on uniform game trees of width 24 and depth 8

former a variable speculative search width (s) is used, while for the latter the speculative width is fixed at a quarter of the branches at a node. Since the expected success probability (ϑ) is set to over 0.95 here, a re-search rarely takes place in $mpv1$, $mpv2$ and $mpv3$.

The results show that both $mpv1$ and $mpv3$ visit fewer nodes than ABS and even than PVS. Although $mpv2$ searches more nodes than $mpv3$, because no exchange of information is made among the nodes in the PV-sets, its performance is still comparable to ABS. The experiments also show (not included in the above figures) that in the worst case, i.e., when both the fixed search width is set too wide, and there is no subtree bound exchange between the nodes in the PV-sets, the ratio of the number of terminal visited by the multiple PVSplitting search to ABS could range from 1 to 8. This extreme case should never arise in practice, because we have heuristics to help choose a suitable search width at different nodes, and bound exchange is always necessary.

From these figures, the properties of the multiple PVSplitting search are apparent. The number of nodes searched by the DM-PVSplit algorithm can be estimated to range from that of $mpv1$ to that of $mpv2$. Therefore its search overhead is comparable to PVSplitting's. Notice that in the DM-PVSplit algorithm, all the nodes in the PV-sets are searched in parallel (different from the PVSplitting algorithm). Therefore, Multiple PVSplitting provides more opportunities for parallelism at reasonable search overhead cost.

Comparison with other parallel methods:

In our experiments, besides the DM-PVSplit algorithm, we also implemented parallel aspiration search, tree-splitting and PVsplitting search. Figure 8 plots the average speedups achieved by these1 four methods, on typical well ordered trees (0.7 probability first-move-best) of width 32 and depth 8.

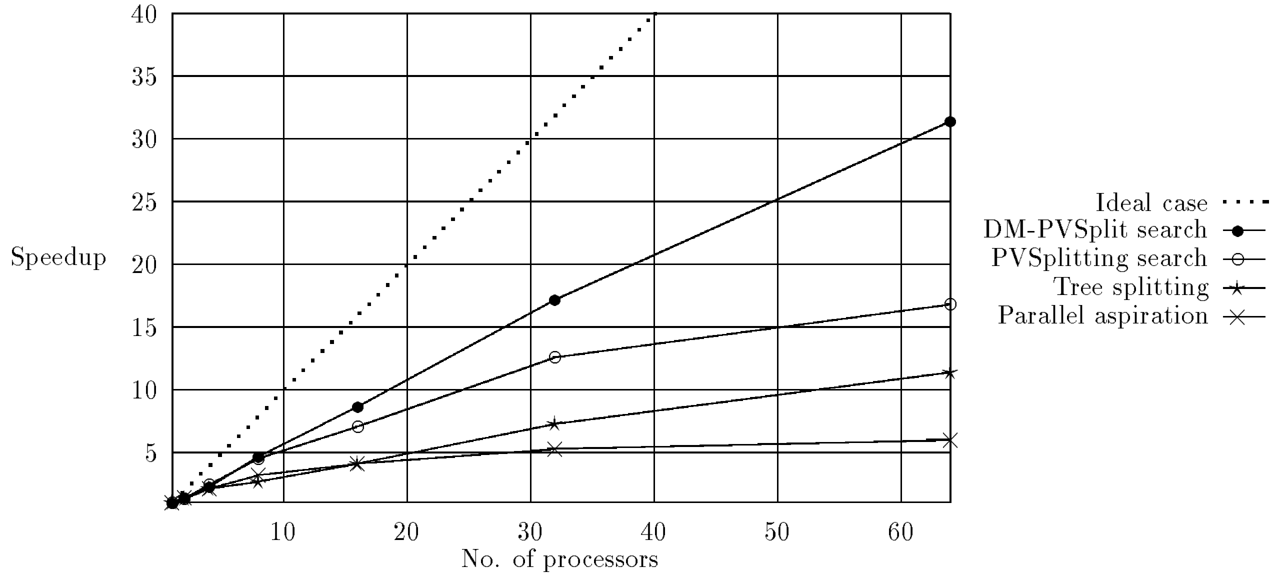

Figure 8: Speedups achieved by different parallel algorithms

Based on the assumption that some heuristics can be used to estimate an appropriate search window in a game, in parallel aspiration search, the tree is searched concurrently with several disjoint narrow windows. But theory shows that only limited speedup can be obtained, regardless of how many processors are available. This is consistent with the null-window search results shown in Figure 2. The DM-PVSplit search shows a better performance than the PVSplitting search on both moderately and strongly ordered trees (as anticipated in Figure 6 and Figure 7). Although it is known that the null window search succeeds often enough to outweigh the extra cost of re-research in PVS, in the parallel case where PVSplitting is used more re-searches can occur leading to anomalously poor performance. When more than one null window search fails high, it is very difficult to make tradeoffs between re-searching them in parallel at the cost of high search overhead, and re-searching them sequentially at the cost of high delay. In DM-PVSplit, the search overhead is mainly caused by searching with a full window those nodes which can actually be proven inferior with a null window. But the advantages are dominant: more parallelism is available since the nodes in the PV-sets are searched in parallel; and relatively uniform work exists for each null window search since a re-search hardly occurs.

Furthermore, if we usually choose an appropriate speculative search width, DM-PVSplit will have lower overhead than PVSplitting. Our experiments also confirm that when the trees become very strongly ordered (more than 0.9 first-move-best), the performance of all these methods comes together, as indeed they must for search of a minimal game tree.

5 Conclusion and future work

In this paper, we propose the DM-PVSplit algorithm. It searches the PV-set to exploit speculative parallelism and guarantees that there is always a variety of work available to distribute. It provides enough parallelism to benefit from massively parallel machines without a significant increase in search overhead. Most effort is first put into the subtrees rooted at the PV-set, while search of the subtrees rooted in the inferior set is delayed until all moves in the PV-set are examined. The method reduces the re-search delay and overhead, and even reduces total search overhead in some cases. For instance, a shallow search on the leftmost branch yields an initial approximate bound, and that bound is used to artificially narrow the search window of the PV-set, to reduce possible search overheads. Through a group-based scheduling strategy, better load balance is achieved without the bottleneck problem caused by central control. The preliminary experiments are promising and have shown that Multiple PVSplitting search achieves better performance than PVSplitting. The future work includes measuring the efficiency of DM-PVSplit in an actual game-playing program, since the trees produced there are more complex due to their nonuniform depth and degree. We are also rewriting the code in ABCL [22], an object-oriented concurrent programming language, to take advantage of parallel object-oriented techniques that model the game tree problem in a more natural way.

6 Acknowledgements

We would like to thank Dr. Alexander Reinefeld (University of Paderborn) for his invaluable help and constructive suggestions. We are also indebted to Prof. Akinori Yonezawa (University of Tokyo) for his many helpful comments and for making the AP1000 available to us.

References

[1] S.G. Akl, D.T. Barnard, and R.J. Doran. The design, analysis, and implementation of a parallel tree search algorithm. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 4(2):192–203, 1982.

[2] G. Baudet. *The design and analysis of algorithms for asynchronous multiprocessors.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1978.

[3] M. Campbell and T.A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[4] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game tree search. In V. Kumar, P.S. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 66–101. Springer-Verlag, New York, 1990.

[5] Rainer Feldmann. *Game tree search on massively parallel systems.* PhD thesis, Dept. of Math. and Computer Sci., University of Paderborn, Paderborn, Germany, 1993.

[6] E.W. Felten and S.W. Otto. Chess on a hypercube. In *The 3rd Conf. on Hypercube Concurrent Computers and Applications, Vol. 2*, pages 1329–1341, 1988.

[7] Ch. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta pruning. In *Proc. AAAI-88, Seventh National Conf. on Artificial Intelligence, Vol. 2*, pages 128–132, 1988.

[8] R.A. Finkel and J.P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[9] J.P. Fishburn. Analysis of speedup in distributed algorithms. Tech. rep. tr431, University of Wisconsin, Madison, 1981. Also, UMI Research Press book, 1984.

[10] Feng hsiung Hsu. *Large scale parallelization of alpha-beta search: an algorithmic and architectural study with computer chess.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1990.

[11] R.M. Hyatt, B.W. Suter, and H.L. Nelson. A parallel alpha-beta tree searching algorithm. *Parallel Computing*, 10(3):299–308, 1989.

[12] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[13] B. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Dept. of Computer Science, MIT, 1994.

[14] T.A. Marsland and M.S. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

[15] T.A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Trans. on Pattern Analysis and Mach. Intell.*, 7(4):442–452, 1985.

[16] A. Musczycka and R. Shinghal. An empirical comparison of pruning strategies in game trees. *IEEE Trans. on Systems, Man and Cybernetics*, 15(3):389–399, 1985.

[17] Monroe Newborn. Unsynchronized interatively deepening parallel alpha-beta search. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 10:687–694, 1988. See also earlier 'A parallel search chess program,' Proc. ACM Annual Conf., pages 272–277, 1985.

[18] A. Reinefeld and T.A. Marsland. A quantitative analysis of minimal window search. In *Procs. 10th Int. Joint Conf. on Art. Intell.*, pages 951–954, Milan, Italy, 1987. (Los Altos: Kaufmann).

[19] A. Reinefeld, J. Schaeffer, and T.A. Marsland. Information acquisition in minimal window search. In *Procs. 9th Int. Joint Conf. on Art. Intell*, pages 1040–1043, 1985. For more details see 'Low overhead alternatives to SSS*', Artificial Intelligence, 31(2):185–199, 1987.

[20] J. Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[21] T. Shimiz, H. Iwashita, and H. Ishihata. Low-latency message communication support for the ap1000. In *the 19th Inter. Symp. on Computer Architecture*, pages 288–297, 1992.

[22] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.

Appendix A: ABS, a standard Fail-soft α - β Search

```
function ABS (n: node; α, β, height: integer): integer;
begin                                             {returns minimax value of n}
    if height = 0 or n is a leaf then
        return Evaluate(n);                       {leaf or frontier node}
    next ← SelectSuccessor(n);                    {generate first successor}
    estimate ← −∞;
    while next ≠ NULL do
        α ← max(α, estimate);                     {raise the alpha bound}
        merit ← − ABS (next, −β, −α, height − 1);
        if merit > estimate then begin
            estimate ← merit;                     {improved value}
            if estimate ≥ β then
                return estimate;                  {cut-off}
        end;
        next ← SelectBrother(next);               {generate brother}
    end while;
    return estimate;                              {return the subtree value}
end;
```

Figure 9: Appendix A: Standard Fail-soft $\alpha - \beta$ algorithm

Appendix B: Principal Variation Search with null windows

The program in Figure 10 represents a version of PVS that explicitly calls a null-window search *(NWS)* procedure, Figure 11. This version is presented here to make it easier to see how the PV, CUT and ALL nodes develop and are related to each other in the general case. By replacing the line

$$merit \leftarrow \; - NWS\ (next,\ -\alpha,\ height - 1);$$

in Figure 10 with

$$merit \leftarrow \; - PVS\ (next,\ -\alpha - 1,\ -\alpha,\ height - 1);$$

and so eliminate the explicit call to NWS, we have another conventional form for the Principal Variation Search.

```
function $PVS$ ($n$ : node; $\alpha$, $\beta$, $height$: integer): integer;   {for PV nodes}
    if $height = 0$ or $n$ is a leaf then
        return $Evaluate(n)$;                              {leaf or frontier node}
    $next \leftarrow SelectSuccessor(n)$;
    $best \leftarrow -PVS\ (next,\ -\beta,\ -\alpha,\ height - 1)$;   {PV node}
    $next \leftarrow SelectBrother(next)$;
    while $next \neq NULL$ do
        if $best \geq \beta$ then
            return $best$;                                 {aspiration CUT node}
        $\alpha \leftarrow max(\alpha,\ best)$;                         {alpha raising}
        $merit \leftarrow -NWS\ (next,\ -\alpha,\ height - 1)$;
        if $merit > best$ then
            if $merit \leq \alpha$ or $merit \geq \beta$
                then $best \leftarrow merit$                      {improved value}
                else $best \leftarrow -PVS\ (next,\ -\beta,\ -merit,\ height - 1)$;   {re-search}
        $next \leftarrow SelectBrother(next)$;
    end while;
    return $best$;                                         {return the subtree value}
end;
```

Figure 10: Appendix B: Principal Variation Search (PVS)

```
function $NWS$ ($n$ : node; $\beta$, $height$: integer): integer; {for ALL and CUT nodes}
    if $height = 0$ or $n$ is a leaf then
        return $Evaluate(n)$;                              {leaf or frontier node}
    $next \leftarrow SelectSuccessor(n)$;
    $estimate \leftarrow -\infty$;
    while $next \neq NULL$ do
        $merit \leftarrow -NWS\ (next,\ -\beta + 1,\ height - 1)$;
        if $merit > estimate$ then
            $estimate \leftarrow merit$;                        {improved value}
        if $merit \geq \beta$ then
            return $estimate$;                             {CUT node}
        $next \leftarrow SelectBrother(next)$;
    end while;
    return $estimate$;                                     {ALL node}
end;
```

Figure 11: Appendix B: Null window search for use in PVS