Chapter 3

Parallel Alpha-Beta Game Tree Search

There is a substantial body of literature on the parallel search of game trees. This is partly because deeper searches yield potentially better play, at least in chess [Tho82], and partly because the basic Alpha-Beta algorithm is an important technique that is a challenge to parallelize efficiently. This chapter discusses the fundamental concepts and techniques of parallel game tree search based on Alpha-Beta pruning. Selected previous research in the area is surveyed. Also, the differences between chess and checkers as the application domain are highlighted. The ideas introduced in this chapter form the foundation of the subsequent chapters describing *Chinook*'s parallel search.

3.1 Parallel Performance

To search a deeper game tree in the same amount of time, researchers have turned to parallelism. Searching one ply deeper in constant time requires a four to eight-fold increase in search power when chess is the application domain [Gil78]. In checkers, an extra one ply requires speeding up the search by a factor of two [SCT92].

A popular measure of parallel performance in game tree search, and parallel computing in general, is the *speedup*. Intuitively, speedup is the measure of how much faster a problem is solved using n processors instead of just one processor. If using n processors results in a n-fold speedup, then it is a *linear speedup*. However, true linear speedup is rare and highly dependent on the nature of the problem. Speedup is defined as:

$$\text{speedup} = \frac{\text{“best” solution time on one processor}}{\text{solution time on } n \text{ processors}}$$

Formula 3.1 — Definition of Speedup

The notion of "best" solution time on one processor is both important and contentious. Typically, the best sequential solution time used is derived from the best implementation available to the authors instead of from an analytical definition of what is best. Although this is a practical approach to the quantity of the speedup, the parallel version of a poor sequential algorithm may be misleading as to the quality of the speedup. For the exact same algorithm, using an unoptimized sequential implementation and slower processors can actually improve the speedup values [SuG91]. Despite its drawbacks, there is little consensus on an alternative measure of overall parallel performance and therefore, this thesis will also use the speedup metric.

Closely related to speedup is the notion of *speedup efficiency*. It represents the amount of speedup potential that is actually achieved. It is defined as:

$$\text{speedup efficiency} = \frac{\text{speedup on } n \text{ processors}}{n}$$

Formula 3.2 — Definition of Speedup Efficiency

Figure 3.1 shows a typical speedup graph with lines and curves representing linear speedup, linear speedup with a smaller slope and diminishing returns for additional processors. There is an additional curve representing parallel *slowdown*, where adding processors beyond a certain threshold decreases performance, and can even make the parallel implementation slower than the sequential implementation. Speedup and speedup graphs will be recurring themes throughout this thesis.

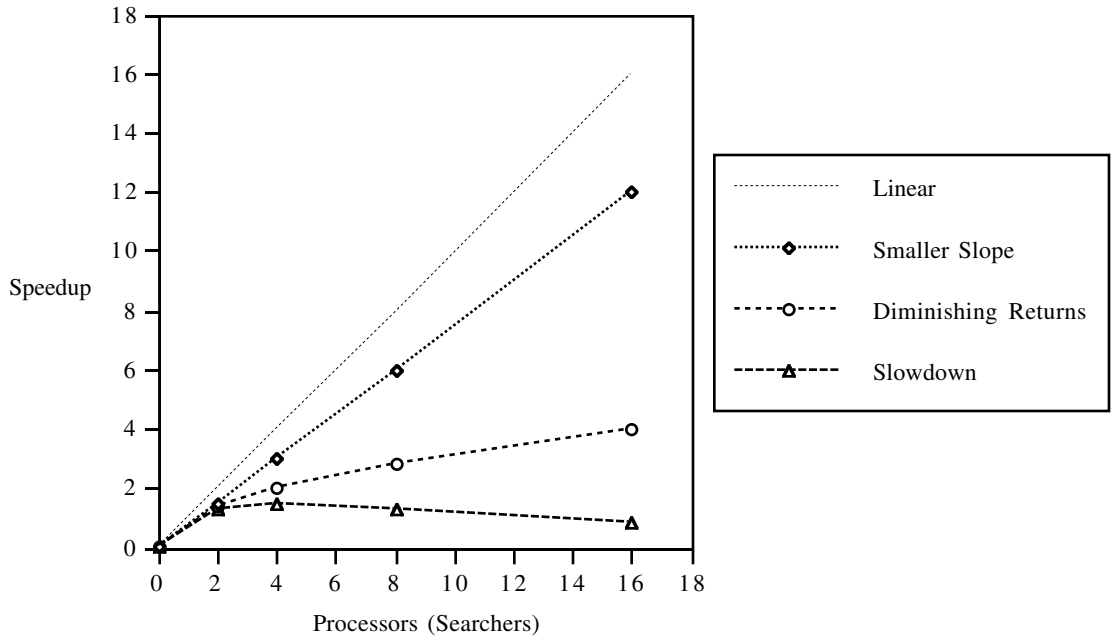


Figure 3.1 — Example Speedup Graph

An important concept related to parallel performance is the notion of work *granularity* . A sequential algorithm does not have to incur the costs associated with coordinating, communicating with or managing the work given to parallel processors. Each point of interaction between parallel processes is a drain on performance. Granularity captures the notion of how much actual work is completed by the algorithm between points of interaction. *Fine-grained* parallelism and *low granularity* work both imply short intervals of useful computation between points of interaction. Therefore, *coarse-grained* parallelism or *high granularity* work, where there is a large amount of computation between interaction points, is preferable.

Another important concept is *load balancing*. At a given point in the computation, the amount of parallel work available, called the *degree of parallelism*, may be less than the number of available processors. Therefore, some processors receive no work and the result is *starvation*. Work can also be unevenly distributed, particularly if it is impossible to predict the amount of computation required for a piece of work. A processor that is given a disproportionately large piece of work may become a *straggler*, forcing the other processors to wait at a

synchronization point. If the number of synchronizing processors, the *synchronization fan-in*, is large then the performance losses due to stragglers is also large.

There are two broad strategies for dealing with load balancing problems. First, instead of allowing processors to be idle, a piece of work already assigned to a processor can be preempted, further subdivided and redistributed. This technique can also be used to avoid starvation and to deal with stragglers. Second, if it is possible, the parallel algorithm can try to create many pieces of work *a priori*. If the parallel processors are overloaded with many units of work, then idle processors are simply given additional work. The danger is that subdividing existing pieces of work, or creating more units of work by making them smaller, also reduces the granularity of the work.

3.2 Sources of Parallelism

To date, there have been four broad strategies used to extract parallelism from game playing programs. A combination of these strategies can be used in practice.

With *component parallelism*, the computationally expensive parts of the program are parallelized. Among the hardware assisted chess machines *Belle* [CoT82], *Hitech* [BeE89] and *Deep Thought* [HAC90], parallelizing the move generation and evaluation function has been a popular and successful strategy at the circuit level. Among the software-only programs, the low granularity of component parallelism has made it a less attractive approach, with *Cray Blitz* [HGN90] being a notable exception.

Another approach is *system parallelism*. In the distributed chess program *ParaPhoenix* [Sch89a], one team of parallel searchers applies the relatively expensive chess knowledge of *Phoenix* in its Alpha-Beta search. A second team of parallel searchers, *Minix*, uses a limited set of knowledge and heuristics to do a speculative, special purpose Alpha-Beta search for tactical advantages. In effect, *ParaPhoenix* consists of two systems, each contributing to the program's play, but each building different game trees. *Minix* is reported to improve the play of *ParaPhoenix* and, given the diminishing returns for additional *Phoenix* searchers, it is an alternate way to utilize additional processors. However, aside from *ParaPhoenix*, system parallelism in computer game playing has not gained the attention of researchers.

For speeding up the underlying game tree search, *parallel aspiration search* was one of the first strategies to be tried [Bau78]. In parallel aspiration search, the Alpha-Beta search window is partitioned into disjoint ranges and given to different searchers. Each searcher is given a smaller window, therefore the searches are finished more quickly. However, each searcher

must still build a minimal tree, regardless of the window size, thus limiting the potential speedup to an upper bound of 5 to 6 [Bau78].

Tree decomposition is another broad strategy that has attracted attention and, arguably, garnered the most success. It is the focus of this thesis. A *split node* exists where the different subtrees rooted at the node can be searched concurrently by parallel searchers in a process known as *tree-splitting* [Fis81]. Each searcher can be a different parallel processor. Alpha-Beta pruning and other search enhancements can be used to order and combine the parallel search results. A spectrum of algorithms have been proposed, differing prominently in how many split nodes are allowed to exist concurrently and in how the split nodes are chosen. Four representative algorithms are considered in detail. *Principal Variation Splitting* (PVSplit) [MaC82] allows exactly one split node at any given time. *Dynamic PVSplit* (DPVSplit) [Sch89a], the *Zugzwang* approach [FMM89] and the *Delayed Branching Tree Expansion* (DBTE) algorithms [Hsu90] allow multiple split nodes. The main difference among the multiple split node approaches is in how the split nodes are selected. PVSplit and DPVSplit are depth-first search algorithms. The *Zugzwang* approach and the DBTE algorithms are not as limited in the way they traverse the game tree.

3.3 Parallel Overheads

Although tree decomposition offers the potential for speedups, it also introduces certain overheads that do not occur in sequential Alpha-Beta search and which detract from linear speedup.

Search overhead (SO) is the ratio between the number of nodes in the tree built by the parallel and sequential algorithms to return the minimax value of the tree. Formally, search overhead is [MOS85]:

$$\mathrm{SO} = \frac{\text{nodes searched on } n \text{ processors}}{\text{nodes searched on one processor}} - 1$$

Formula 3.3 — Definition of Search Overhead

There are two main sources of search overhead, both related to information deficiency. First, there could be a lack of information regarding cutoffs. A parallel algorithm that performs subtree searches concurrently at a CUT node does not know *a priori* which subtree will cause a cutoff and make the other searches irrelevant. Second, there could be a lack of information

regarding optimal search windows. Particularly with minimal window techniques, the values of previously searched subtrees can be used to improve the windows of subsequent searches. Without the benefit of previous search results, the windows of parallel searches may be wider than needed or even incorrect, resulting in larger subtrees or the overhead of re-searching subtrees. The lack of timely information about cutoffs and windows results in search overhead. It is ironic, given all the effort invested in parallel algorithms, that Alpha-Beta is an inherently sequential algorithm due to this information dependency.

Interestingly, it is possible to observe negative search overhead. Even the best sequential search implementations do not always achieve perfect move ordering at cutoff nodes. If a parallel algorithm can exploit this flaw and achieve a cutoff more quickly, it might result in a negative search overhead.

Synchronization overhead (SY) is incurred when an algorithm forces a searcher to wait for another searcher to finish. To combat information deficiency, some parallel algorithms use synchronization to gather all of the information regarding a node so that it can be used in subsequent searches. As well, search extensions, pruning shortcuts and other search enhancements may cause the work given to one searcher to be substantially more than another searcher. This leads to a load balancing problem when a straggler searcher forces other searchers to wait at a synchronization point.

Of course, there are strategies to reduce both the search and synchronization overheads. It is a difficult problem because there is a close relationship between the two overheads and trade-offs must be made carefully.

Communication overhead (CO) is caused by parallel searchers exchanging data, using locks and semaphores to coordinate access to shared resources and to communicate events. Of course, the sequential algorithm does not need to communicate data with parallel processors. For convenience, whatever is not attributable to search or synchronization overhead is defined to be communication overhead.

3.4 Respecting the Structure of the Alpha-Beta Tree

The Alpha-Beta cutoff has the potential to create parallel search overhead. Under perfect move ordering at a CUT node, the subtree that causes the cutoff is searched first and all of the other subtrees can be pruned. However, if a CUT node is used as a split node, and several subtrees are searched in parallel, there is likely to be waste of search effort because complete pruning is

no longer possible. Consequently, it has been observed that one way to reduce search overhead is to search the subtrees of a CUT node in sequential order rather than in parallel.

Therefore, a heavy-handed rule is that ALL nodes should always be split and CUT nodes should never be split. Later on, we will consider exceptions to this rule, but the basic wisdom remains a key component of any parallel search algorithm based on tree decomposition. Therefore, if an algorithm decides to split or not to split a node with consideration given to the tree's node structure, it is informally said to *respect the ALL/CUT node structure of the game tree*. In fact, all of the algorithms discussed below incorporate this concept in some form.

3.5 Principal Variation Splitting

Principal Variation Splitting (PVSplit) [MaC82] is a depth-first, tree-splitting algorithm that grew, in part, out of the sequential Principal Variation Search (PVS) algorithm discussed in Chapter 2. It creates parallel work by splitting nodes along the principal variation. As seen in Figure 2.7, each of the nodes on the principal variation is an ALL node, therefore PVSplit respects the ALL/CUT node structure of the game tree. Recursing down the principal variation concentrates the parallel effort where it is potentially most useful, and backing up the principal variation carries important window information for the search of the sibling subtrees.

PVSplit forms the foundation of *Chinook*'s parallel search and will be discussed in detail in Chapter 4. However, it is important to emphasize two aspects of PVSplit for the current discussion. First, PVSplit only splits Type 1 (ALL) nodes and it splits the nodes in a depth-first, in-order fashion as values are backed up along the principal variation. PVSplit uses the backed up value to improve the search windows of the sibling subtrees. Second, PVSplit only allows a single split node at a time. All of the searchers must synchronize at the current split node before the algorithm can back up one ply and create the next split node. Consequently, PVSplit is particularly susceptible to synchronization overheads.

3.6 Dynamic Principal Variation Splitting

The *Dynamic Principal Variation Splitting* (DPVSplit) algorithm [Sch89a] extends basic PVSplit in a way to deal with the synchronization overhead caused by stragglers. In essence, searchers that are otherwise idle at the synchronization point can be dynamically reassigned to help with busy searchers. As searchers are reassigned to help other searchers, a processor tree hierarchy is formed with the straggler processor at the root of the processor tree. The searchers

are able to share their work because they apply the PVSplit algorithm to their assigned subtrees instead of regular PVS.

Two disadvantages with DPVSplit include the potential for unbalanced processor trees and the reduction in the granularity of parallel work. As idle searchers are assigned and reassigned to busy searchers, it is possible to form an inefficient pipeline of processors instead of a balanced tree, in the worst case. A deeper hierarchy of processors implies more synchronization points and more synchronization losses. Furthermore, splitting up the work of one searcher for a reassigned searcher implies that the work created by the "dynamic" aspect of DPVSplit is smaller than the work created by the basic PVSplit. In effect, DPVSplit allows multiple split nodes to be introduced when necessary to deal with stragglers. However, the additional split nodes are all lower in the game tree than the original split node on the principal variation. Consequently, the subtrees at the new split nodes are smaller and potentially suffer the drawbacks of fine-grained work. In practice, DPVSplit does improve the performance of PVSplit. However, it appears that it does not change the overall asymptotic shape of the speedup curve.

Both PVSplit and DPVSplit are depth-first strategies that largely depend on a single controller. DPVSplit introduces a limited amount of distributed search control when idle searchers are reassigned to help busy searchers. In that case, the straggler process becomes both a manager of searchers and a searcher. However, with both algorithms, the search control remains largely centered on the principal variation.

The advantage of centralized control is that all of the information on cutoffs and search windows can be concentrated and applied at the split node, reducing search overhead. The disadvantage is that the single control point can be come a bottleneck, increasing synchronization overhead. The addition of more searchers increases the synchronization fan-in and if the branching factor of the split node is low, there may be insufficient parallel work for the searchers. Intuitively and empirically, PVSplit-based algorithms perform best with small numbers of processors and trees with high branching factors. The ratio between the number of searchers and the branching factor, called the *parallelism overload factor*, is an important measure of how likely there is to be starvation and how high the synchronization fan-in will be.

Another advantage of PVSplit-based algorithms is that, since they are depth-first searches, the amount of storage required for the search control is linearly proportional with respect to the depth of the search and the number of searchers. The low storage overhead for depth-first search algorithms is a key strength in actual implementations.

3.7 Other PVSplit-Based Algorithms

Some of the other efforts in PVSplit-based parallel algorithms include the *Cray Blitz* [HSN89] and *Waycool* [FeO88] chess programs. The *Waycool* program is interesting for its reported 101-fold speedup on a 256 node hypercube multicomputer. However, as pointed out by several researchers, including Hsu [Hsu90], that speedup value does not take into consideration the beneficial growth in transposition table size as processors are added.

Although both projects introduced interesting load balancing techniques that are effective for their individual programs, it is difficult to compare their relative merits. In fact, the inability to meaningfully and quantitatively compare different search ideas, except when implemented within a single chess program, is a fundamental limitation of research in this area. Nonetheless, qualitative comparisons of parallel search ideas, particularly between PVSplit-based algorithms and other parallel search paradigms, can be instructive.

3.8 The *Zugzwang* Approach

One of the key design decisions made for the *Zugzwang* distributed chess program [FMM89] is to completely distribute the parallel search control. Each searcher executes the same search and control code and are virtually identical in functionality. Each searcher is responsible for requesting work from other searchers. Each searcher is able to receive work and search it locally or redistribute subproblems to other searchers. The lack of a central controller implies that the potential bottleneck noted for PVSplit and DPVSplit may be avoided.

Upon receiving a problem, namely a node to be searched, a searcher breaks up the node into subproblems which it can divide further and search locally, or if asked, can be passed on to another searcher. The creator of the split node, as in PVSplit, is its master and is ultimately responsible for its completion. Given a choice of subproblems to redistribute, the subtree closest to the root of the original problem is preferred. Alpha-Beta pruning and other search enhancements are also used. There can be multiple split nodes within a single searcher and multiple split nodes among the parallel searchers. It should be noted that each searcher under DPVSplit performs a similar procedure. The differences arise in how work is shared between searchers.

To supplement the cyclic process of receiving work, dividing the work and, possibly, reassigning some of the work are two principles described as Young Brothers Wait and Helpful Master.

Viewing the different subtrees rooted at a node as siblings, *Young Brothers Wait* states that, although the work has been divided, the moves ordered after the first, or eldest, subtree are not available for redistribution until the first subtree has been completed. Consequently, if the split node is a CUT node and the cutoff occurs after the first subtree, no search overhead is incurred through the premature reassignment of work. Furthermore, information from the first subtree that may improve the search windows of the siblings can be applied before the work is given out. The Young Brothers Wait concept is a restatement of PVSplit's criteria for creating a new split node: the first ordered subtree must be searched first.

Interestingly, the behaviour of Young Brothers Wait is the same at both CUT and ALL nodes. Both types of nodes are completely splittable after the first search result is returned. There is no reported mechanism for preventing the splitting of CUT nodes, although it would be simple to implement. Even if the elder subtree does not cause a cutoff, the likelihood of each subsequent subtree causing the cutoff is high in a strongly ordered tree, assuming that the node is behaving as a CUT node. Young Brothers Wait avoids search overhead only in the case of best move ordering, which is not always the case in practice.

With the *Helpful Master* concept, the master of a split node that is waiting for search results to be returned can decide to *assign itself* to help a searcher finish a subtree for which the master is waiting. Thus, the Helpful Master is a load balancing strategy that also has a completely distributed control structure. Unlike DPVSplit, each searcher is empowered to decide for itself if it will help another searcher. The distributed search initiation and the Helpful Master results in a dynamic and effective form of load balancing.

In general, *Zugzwang*'s search begins with a single searcher being given the root position of the game tree. As it divides the node, it follows the Young Brothers Wait concept and ultimately creates work than can be redistributed to other searchers. They, in turn, create more work which can be used to load balance the search. The Helpful Master concept also aids in load balancing the search, particularly in the case of a straggler process.

The strongest and most unique feature of the *Zugzwang* approach is the complete distribution of search control among peers. It is a natural approach for the distributed hardware used to implement the chess playing program. On the one hand, the strategy reduces the synchronization fan-in at split nodes and avoids the bottleneck of single split node algorithms. On the other hand, the strategy creates more synchronization points and the frequent interactions between searchers to receive work and communicate information may result in

low-granularity parallelism. From the reported speedups and the success of the *Zugzwang* program in tournament play, the ideas seem promising.

3.9 Delayed Branching Tree Expansion Algorithms

Hsu introduced a family of multiple split node, non-depth-first, parallel Alpha-Beta search algorithms [Hsu90]. His design goal was to allow the use of several hundred, and even thousands, of parallel searchers in a new generation chess machine.[1] The family of Delayed Branching Tree Expansion (DBTE) algorithms avoid the concurrent expansion of sibling subtrees of a CUT node, but the individual algorithms differ in the strategy used to select the next split node. The details of the Leftmost-First (LF), Best-First (BF) and First-Come-First-Served (FCFS) strategies can be found in [Hsu90]. Of the three strategies, only the Leftmost-First DBTE (LF-DBTE) is considered in depth.

The DBTE algorithms are influenced by the hardware model chosen for the new chess machine: a host machine builds the upper plies of the game tree in memory and decides on the split nodes to be given to the next available searcher from a pool of processors. Therefore, a given DBTE algorithm is the scheduling or priority queue strategy for split nodes. The host can be a workstation or a collection of workstations. The parallel searchers are chess search engines using custom chips for the move generator and the static evaluation function. Therefore, the new chess machine exploits both fine-grained component parallelism and coarse-grained search parallelism through tree decomposition.

At the heart of the DBTE algorithms is the notion of the critical tree, which is related to the minimal tree. The critical tree is the actual proof tree and represents nodes that must be searched in order to determine the minimax value of the root position. Hsu distinguishes between the notion of "critical work" that is implied by the critical tree and "mandatory work" that is the basis of work by Akl, Barnard and Doran [ABD82] and Finkel and Fishburn [FiF83]. In fact, Hsu points out that the notion of mandatory work described by Finkel and Fishburn is "not even mandatory for the full window [Alpha-Beta] search which uses a starting window of $(-\infty, +\infty)$" [Hsu90]. However, he also notes that even the critical nodes are not necessarily searched if an aspiration window is used. Still, "the set of the critical nodes does have the merit of being worth evaluating when there is *nothing better* to do" [Hsu90, pg. 73]. In essence, the critical nodes are the preferred choices for the multiple split nodes.

---

[1] Dr. Hsu and his colleagues created the ChipTest, Deep Thought and Deep Blue chess machines [HAC90].

As with the PVSplit-based algorithms, the DBTE algorithms explicitly respect the ALL/CUT node structure of the Alpha-Beta game tree. However, the DBTE algorithms allow for multiple split nodes, which is not the case for the basic PVSplit. The use of multiple split nodes is a fundamental way of growing the search tree in both *Zugzwang* and the DBTE algorithms. Unlike the *Zugzwang* approach, the DBTE algorithms have a centralized scheme for search control. The searchers of the chess machine are not general purpose processors. A piece of work given to a DBTE searcher is indivisible. The host machine controls the search and performs the splitting of nodes.

The DBTE algorithms have been analytically shown to possess two important properties. First, the Leftmost-First DBTE algorithm is proven to provide speedup that is asymptotically close to linear for a fixed number of processors, as the depth of the best-first ordered search tree is increased. Therefore, regardless of the number of searchers, the speedup of the search improves as the size of the search tree grows. In Hsu's view, this is a necessary but not sufficient condition for a good parallel Alpha-Beta algorithm. Second, Hsu proves that the Leftmost-First DBTE algorithm theoretically never searches a node that the weak Alpha-Beta algorithm does not search. The weak Alpha-Beta algorithm does not enjoy cutoffs due to bounds information from ancestor nodes more than two plies away, the so-called *deep cutoffs*. Therefore, the normal Alpha-Beta algorithm, with deep cutoffs, never searches more (and usually fewer) nodes than weak Alpha-Beta. Leftmost-First DBTE searches no more nodes than the weak Alpha-Beta as well. Therefore, given the asymptotic optimality of the speedup and the upper bound on nodes searched provided by the weak Alpha-Beta algorithm, Hsu concludes that the Leftmost-First DBTE algorithm has a non-trivial lower bound on its parallel Alpha-Beta speedup.

Hsu's important analytical results do not preclude the fact that other parallel algorithms may also possess the same theoretical speedup properties. However, the reporting of such analytical properties of a parallel Alpha-Beta algorithm is sufficiently rare that the Leftmost-First DBTE theoretical results are noteworthy. Certainly, no such analytical results of the PVSplit-based algorithms and the *Zugzwang* approach have been reported. Although results based on an actual implementation of DBTE have not been reported, Hsu did perform simulations of the algorithm to support his theoretical results.

The DBTE algorithms remain a promising alternative to other algorithms. Its main disadvantages are that it gives up the deep cutoffs and that it potentially requires a lot of storage for the nodes of the expanding tree, since it is not a depth-first strategy. However, the need for multiple split nodes is clear as the number of parallel searchers increases. Furthermore, the

desirable theoretical properties of the algorithm distinguish it from other algorithms proposed to date.

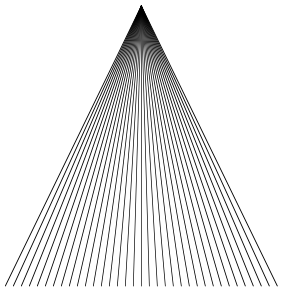
3.10 Application Domains: Chess and Checkers Game Trees

PVSplit, DPVSplit, the *Zugzwang* approach and the DBTE algorithms have primarily been applied to or proposed for chess game trees. However, checkers game trees have two distinguishing features from chess game trees. First, the average branching factor in checkers is much less than chess. In chess, a typical position has a branching factor ranging from 35 to 40 [Gil78]. In the commonly used Bratko-Kopec chess test suite [KoB82], the branching factor is about 35. The checkers game trees of the test suite used for this thesis have an average branching of 2.84.[2] That is to say, the branching factor in chess game trees is over an order of magnitude higher than in checkers game trees. This has clear implications for tree decomposition strategies such as PVSplit. Second, the distribution of branching factor values in checkers trees is bimodal. There is a significant difference in branching factor depending on whether the node represents a capture or non-capture position. In checkers, if a capture is possible, it must be taken. Consequently, the average branching factor for capture nodes in the test suite is 1.28 and for non-captures nodes it is 7.93. Because there are more capture nodes than non-capture nodes, the weighted average is 2.84. The *bimodal distribution* of branching factor values has implications if a chosen split node is also a capture position.

---

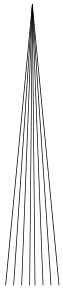
[2] The Tinsley-*Chinook* 1992 test suite will be discussed later and in Appendix B. All reported branching factors for checkers game trees are measured quantities.

Figure 3.2 graphically shows the difference in branching factor between chess and checkers game trees.



Chess (bf = 35)

Checkers (bf = 1.28, capture)



Checkers (bf = 7.93, non-capture)

Checkers (bf = 2.84, weighted average)

Figure 3.2 — Chess and Checkers Game Trees

Because PVSplit and other tree decomposition strategies create parallel work by splitting nodes, a high branching factor means that there is a large quantity of parallel work at each split

node. Starvation is not a serious problem with wide chess game trees and relatively low numbers of searchers. Similarly, if the degree of parallelism offered by the branching factor is higher than the number of searchers, the load balancing problem can be ameliorated by giving a new piece of work to a searcher who finishes early.

In checkers, if the split node is a capture position, it is likely that all but one of the searchers will be starved since the average branching factor is less than 2 at a capture node. Furthermore, since PVSplit recurses down one of the branches before splitting the node, the effective degree of parallelism at a split node which is a capture position is less than one. If the number of searchers is greater than 8, it is possible that some searchers will never receive work. When searchers can starve due to an insufficient branching factor at a split node, the game tree is said to be *branch narrow*, or simply *narrow*. It is purposely a relative term that depends on the parallelism overload factor concept discussed earlier.

Clearly, the low branching factor of checkers game trees means PVSplit may not be able to effectively utilize even a moderate number of parallel searchers. Nonetheless, it was decided that a PVSplit-based version of *Chinook* needed to be implemented as a first step to understanding how the narrow checkers game trees would affect parallel game tree search performance.