Chapter 5

Improving Parallel Chinook

The experimental results show that Principal Variation Splitting's (PVSplit) poor speedups are mainly due to processor starvation. However, the impact of uneven load balancing, due to stragglers, is also a factor in causing the poor speedups. In this chapter, we consider techniques to deal with both the starvation and the load balancing problem. Experimental data is gathered using the Tinsley-*Chinook* 1992 test suite to show the benefits of the new strategies.

5.1 Reducing Starvation: Principal Variation Frontier Splitting

Since the low branching factor of checkers game trees leads to searcher starvation, the issue of how to create sufficient parallel work needs to be addressed. One obvious possibility is to allow multiple split nodes, which increases the degree of parallelism. The *Zugzwang* approach and the DBTE algorithms depend fundamentally on multiple split nodes, but adopting one of those algorithms would require a substantial reworking of *ParaChinook*. If the variation splitting framework of PVSplit is to be maintained, how the extra split nodes are chosen raises some interesting design decisions.

First, since most of the search effort is along the principal variation, the candidate split nodes should probably be on the principal variation as well. More generally, if the algorithm recursively calls itself for non-principal variation lines of play, then the current variation from root to leaf node should contain the candidate split nodes. Dynamic PVSplit (DPVSplit) also recognizes this observation.

Second, the subtrees rooted at the upper nodes of the principal variation represent the coarsest-grained units of work because the subtrees are deeper. In other words, split nodes located closer to the root of the tree contain higher granularity work than split nodes located closer to the leaves. Therefore, if there is no work to be given out at the current split node, then the new split node should be above the ply of the current split node instead of below it. DPVSplit also allows multiple split nodes, but those split nodes are closer to the leaves of the tree and potentially suffer from low granularity.

Third, and lastly, there should be a preference for splitting ALL nodes before CUT nodes. In a strongly ordered game tree, it is likely that the first subtree will generate the cutoff at a CUT node. Therefore, concurrently searching multiple subtrees of a CUT node in an attempt to create more parallel work may simply be creating unnecessary work. However, if given a choice of either splitting a CUT node or allowing searchers to starve, the CUT node could be split.

Based on these observations, a new variation of PVSplit is introduced. *Principal Variation Frontier Splitting* (PVFSplit) supports multiple split nodes along the current variation. The *frontier splitting* technique is analogous to the general notion of the frontier nodes of a tree, where the search is being expanded. The algorithm is a depth-first strategy that addresses the practical issues of work granularity and the ALL/CUT node structure of the Alpha-Beta game tree. Consequently, a newly created split node is located on the current variation, it is closer to the root of the game tree than the current split node and splitting ALL nodes is preferred to splitting CUT nodes.

The impact of frontier splitting is two-fold. First, searchers do not need to starve if the parallel work at a split node is exhausted. A new split node with more work may be created. Second, as a positive side effect, frontier splitting reduces the effective synchronization fan-in at split nodes. By the time the last search result of a split node is returned, the other searchers are, ideally, already working at a different split node instead of waiting to synchronize. However, strictly speaking, frontier splitting does not completely eliminate synchronization fan-in. Because the split nodes can only exist on the current variation, the processors must still synchronize at the root node. Rather than synchronizing at each split node, as is the case of PVSplit, the root node is the only synchronization point under PVFSplit. Of course, even this synchronization requirement can be eliminated if one wishes to allow multiple variations from the root position to be searched in parallel. The current version of *ParaChinook* does not allow multiple variation splitting.

The biggest drawback of PVFSplit is that the searches rooted at the new split node may have to be initiated without the benefit of the bounds information returned by the first subtree. PVFSplit trades off the potential of more search overhead for the potential benefits of reduced starvation through multiple split nodes. Without bounds information, we are faced with either allowing a full window search of the subtrees or we can use heuristics to create an artificial minimal window. In the current implementation of PVFSplit, a minimal window is heuristically created for all split nodes greater than 7 plies from the leaf nodes. The best score at the deepest split node is the basis of the minimal window. If no such score is available, then the value of the current subtree from the last iteration of iterative deepening is used.

Figure 5.1 contains the pseudo-code for the Principal Variation Frontier Splitting algorithm. The shading highlights the main differences between PVSplit and PVFSplit.

The `Value[]` array is now two-dimensional since each node of the current variation could potentially be a split node under PVFSplit. Some new local variables (`newPly`, `heurAlpha`, `heurBeta` and `pNew`) are also defined.

The call to `isAnALLNode()` before assigning work from the current split node makes explicit what was implicit in the pseudo-code and discussion of PVSplit: if the current variation is not the principal variation, it may contain CUT nodes. Therefore, if the current split node is an ALL node, then the subtrees are farmed out to parallel searchers. If the current split node is a CUT node, then each subtree is searched in sequence with a recursive call to PVFSplit.

A new body of pseudo-code deals with choosing and assigning work from a new split node before PVFSplit can recurse out of the current split node. This is the key to allowing multiple split nodes on the current variation. The function `GetOtherSplitNode()` selects the split node from among the candidate nodes along the current variation. If the current split node is at ply n, then the nodes at plies n - 1, n - 2, n - 3, etc. back to the root are in decreasing order of preference. Another consideration is whether the node is an ALL node or a CUT node. Every ALL node on the current variation must already be split and exhausted of work before a CUT node is selected. But if the alternative is starvation, a CUT node is selected to be a split node. Once the new split node is selected, `SynthMinWindow()` is used to decide on the search window to use for the search. As discussed earlier, full window searches are only allowed within 7 plies of the leaf nodes. Otherwise, a heuristic minimal window is synthesized.

One important aspect of *ParaChinook*'s implementation of PVFSplit not included in the pseudo-code, to improve its clarity, deals with cutoffs. It is possible for a cutoff to occur at a split node that is closer to the root node than the current split node. Such a condition is detected and PVFSplit immediately backs up to the split node where the cutoff occurred. This phenomenon is an important advantage of PVFSplit over PVSplit because that cutoff would not have been detected until the algorithm, after much more searching, systematically backed up to that higher node.

```
VAR { Changed Global Variable }
    Value     :    array[ 1..MAX_DEPTH ] of  { Results of each subtree  at each ply }
                   array[ 1..MAX_WIDTH ] of integer;
FUNCTION
    PVFSplit( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                    { Index and value of current move }
    move : moveType;                           { Current move being searched }
    newPly : integer;                          { Ply of new split node with work }
    heurAlpha, heurBeta : integer;             { Window for new split node }
    pNew : position;                           { Position of new split node }

BEGIN { PVFSplit }

{ Check if at threshold for minimum granularity.  Do not recurse to leaves. }
    if ( ply = ( TreeDepth - MinGranularity ) ) then
        return( PVS( p, alpha, beta, ply ) );  { Yes.  Do sequentially. }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then               { Any legal moves in this position? }
        return( Evaluate( p ) );               { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );                { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 2;                         { Continue with other subtrees, later }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;
    WorkOut[ ply ] := 0;                       { Search results still outstanding }

{ Search first ordered move with full window. Unroll the loop by one }
    Score[ ply ] :=
        -PVFSplit( p.Move[ ply ][ 1 ],
                        -Beta[ ply ], -Alpha[ ply ], ply + 1 );
    Value[ ply ][ 1 ] := Score[ ply ];
    Best[ ply ] := 1;

    if ( Score[ ply ] ≥ Beta[ ply ] ) then    { Cutoff after pv? }
        Index[ ply ] := ∞;                     { Yes.  Will never enter loop }
    Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );       { Improve window? }

{ Search each sibling subtree with minimal window }
{ Split node incomplete if there are local subtrees unsearched or if there are searchers still to return }
    while ( ( Index[ ply ] ≤ Width[ ply ] ) OR ( WorkOut[ ply ] > 0 ) ) do
      begin { while }
        if ( "∃idle searcher" AND Index[ ply ] ≤ Width[ ply ] ) then
          begin { Local work out }             { Search each local subtree }
            index := Index[ ply ];             { An unique move leads to the subtree }
            move := Moves[ ply ][ index ];
            Index[ ply ] := Index[ ply ] + 1;  { Advance index }

            if ( isAnALLNode( p ) ) then       { Respect ALL/CUT node structure }
              begin { Local ALL node }
                Value[ ply ][ index ] :=       { Search in parallel }
                    -FarmWork( p.move,         { Negamax value and minimal window }
                        -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );
                WorkOut[ ply ] := WorkOut[ ply ] + 1;    { Keep track of work given }
              end { Local ALL node }
            else
                Value[ ply ][ index ] :=       { Search in sequence }
                    -PVFSplit( p.move,         { Negamax value and minimal window }
                        -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );
          end;  { Local work out }
```

{ While-loop continuation }

```
        else if ( "∃ idle searcher" ) then     { If no more local work }
          begin { Non-local work out }         { ...give work from other split node }
            newPly := GetOtherSplitNode( pNew );      { Magical function }
            index := Index[ newPly ];          { An unique move leads to the subtree }
            move := Moves[ newPly ][ index ];
            Index[ newPly ] := Index[ newPly ] + 1;  { Advance index }

            SynthMinWindow( pNew, heurAlpha, heurBeta );  { Might use min window }
            Value[ newPly ][ index ] :=        { Negamax value and minimal window }
                -FarmWork( pNew.move,          { pNew is the  split node }
                    -heurBeta, -heurAlpha, newPly + 1 );
            WorkOut[ newPly ] := WorkOut[ newPly ] + 1; { Keep track of work given }
          end;  { Non-local work out }
```

```
        else if ( index := ReturnedResult() ) then
          begin  { Local results in }
            value := Value[ ply ][ index ];
            WorkOut[ ply ] := WorkOut[ ply ] - 1;   { Work received }

            if ( value > Score[ ply ] ) then  { Score improved, but did it fail high? }
              begin { Update Best }
                if ( value > Alpha[ ply ] and value < Beta[ ply ] ) then
                  begin
                    Value[ ply ][ index ] :=  { Yes.  Re-search  }
                        -FarmWork( p.move, -Beta[ ply ], -value, ply + 1 );
                    WorkOut[ ply ] := WorkOut[ ply ] + 1;    { Still not finished }
                  end;
                else
                    Score[ ply ] := value;     { No }
                Best[ ply ] := index;
              end;  { Update Best }

            if ( Score[ ply ] ≥ Beta[ ply ] ) then   { Cutoff? }
              begin
                Index[ ply ] := ∞;             { Yes.  Force exit from loop }
                interruptSearchers();          { Tell searchers to stop and return }
              end;
            Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );
                                               { Improve window? }
          end;  { Local results in }
      end;  { while }
    return( Score[ ply ] );                    { Done. Return best score }
END;  { PVFSplit }
```

Figure 5.1 - *ParaChinook*'s Principal Variation Frontier Splitting (PVFSplit)
(Shading indicates key changes from PVSplit(). Boxes group key functionality.)

5.2 Experimental Results: Principal Variation Frontier Splitting

Table 5.1 shows the average speedups achieved by PVFSplit on the test suite for 2, 4, 8 and 16 searchers. Appendix C contains the detailed experimental data. The best average speedup is 2.91, achieved with 16 searchers for a speedup efficiency of 18%. Although this is a 52% improvement in speedup over PVSplit, it is still low in absolute terms as reflected by the low speedup efficiency. However, a speedup of 2.91 does translate into more than one additional ply of search in the same amount of time [SCT92].

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.48	0.325	8.76	17.4	12.8	4.90
4	2.03	0.546	24.3	26.3	30.8	8.17
8	2.71	0.803	46.4	46.3	48.8	7.74
16	2.91	1.07	83.5	63.5	67.1	7.00

Table 5.1 — Principal Variation Frontier Splitting (PVFSplit) Parallel Performance (speedup, search overhead and idle time)

Figure 5.2 is the speedup graph for PVFSplit, with a vertical bar representing one standard deviation. As a basis for comparison, the speedup curve for PVSplit is reproduced. Although PVFSplit increases the absolute speedup numbers, it does not change the basic shape of the curve. As searchers are added, the speedup curve flattens to reflect the diminishing returns.

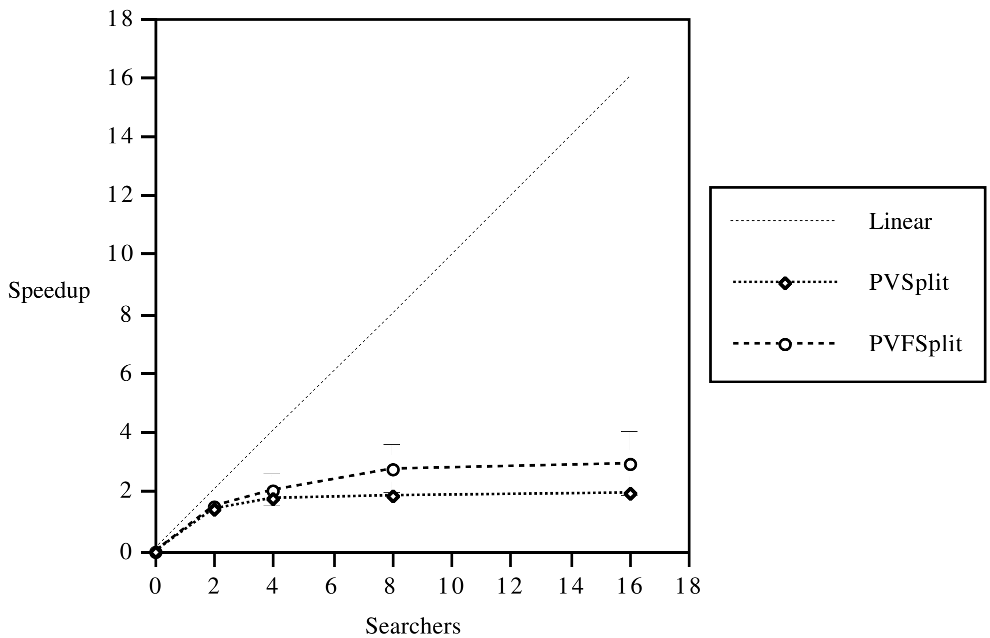

Figure 5.2 — Principal Variation Frontier Splitting (PVFSplit) Speedups
(vertical bar indicates one standard deviation)

With PVSplit, there is a trade-off between having low search overhead and high idle time. An algorithm such as PVFSplit makes the reverse trade-off and increases search overhead in order to reduce idle time.

Inherent in a multiple split node algorithm such as PVFSplit is the risk of higher search overheads due to information deficiency regarding cutoffs and window bounds. In Figure 5.3, we can see that PVFSplit, by being more aggressive than PVSplit in splitting nodes, also suffers from significantly higher search overhead. Of course, the bottom line is that despite the higher search overhead, PVFSplit's speedup is higher.

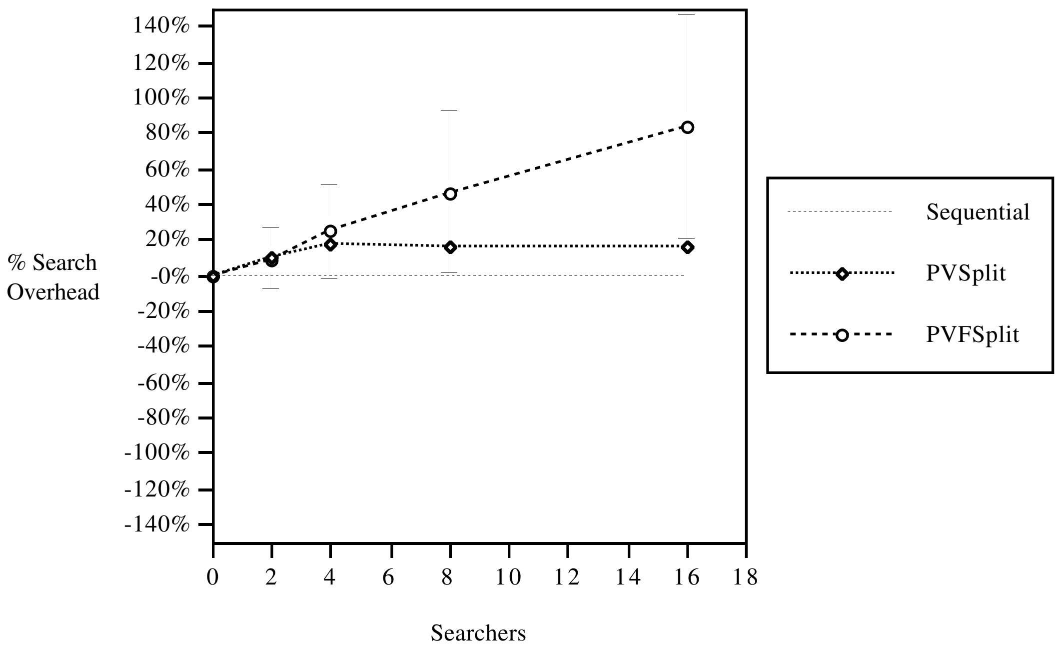

Figure 5.3 — Principal Variation Frontier Splitting (PVFSplit) Search Overhead
(vertical bar indicates one standard deviation)

Interestingly, the search overhead of PVFSplit is dramatically higher than PVSplit, the idle time of PVFSplit is modestly lower, as seen in Figure 5.4, but the speedup is still greater for PVFSplit. This suggests that the trade-off of higher search overhead for reduced idle time is better overall since speedup increases.

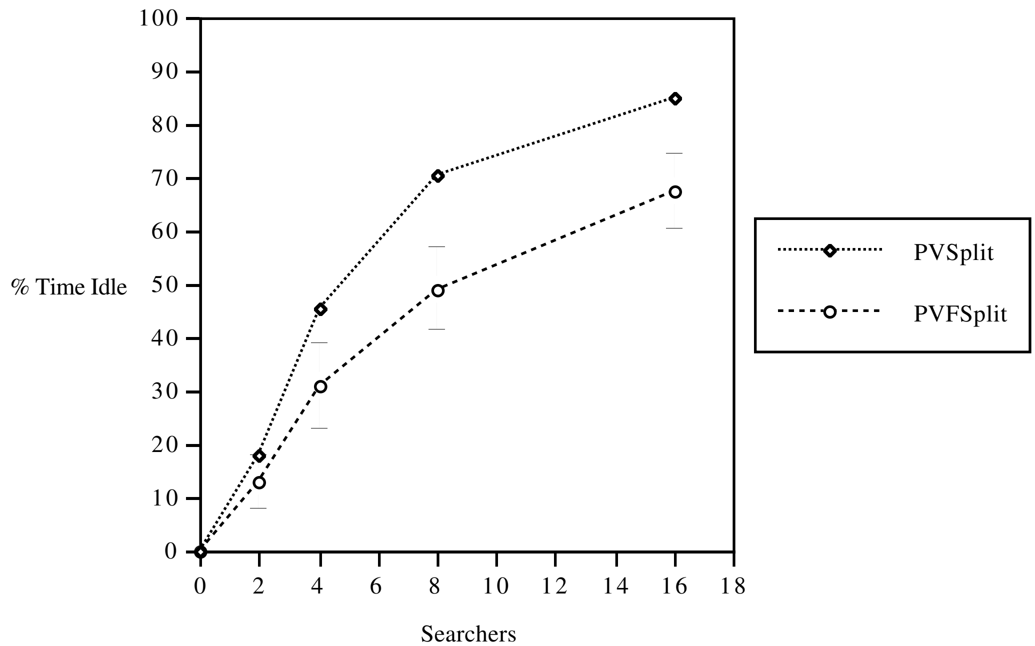

Figure 5.4 — Principal Variation Frontier Splitting (PVFSplit) Percent of Time Idle
(vertical bar indicates one standard deviation)

Despite the success in trading off search overhead for reduced idle time in PVFSplit, the amount of idle time still remains high. Referring to Figures 4.5 and 5.5, the idle time region of the utilization profile has decreased in size while both the speedup efficiency and other overheads regions have grown.

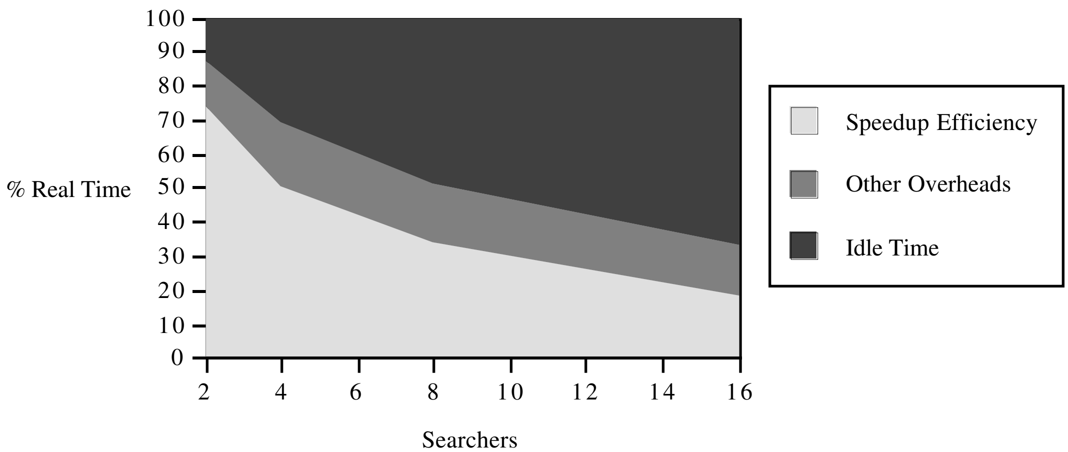

Figure 5.5 — Principal Variation Frontier Splitting (PVFSplit) Normalized Utilization Profile

Reduced starvation through multiple split nodes and reduced synchronization fan-in still leaves a lot of idle time. Although the split node and synchronization policies of PVFSplit can be further enhanced, the fact that search overhead grows as the synchronicity decreases implies that the overall speedup and searcher utilization properties cannot be changed. Even if the need to synchronize at the root position is eliminated, the corresponding growth in search overhead will ensure diminishing returns for additional searchers. Furthermore, the programming and software engineering effort required to debug and maintain a parallel search with so much asynchronous behaviour would be prohibitive.

Despite the reduction in idle time with PVFSplit, it is still rather high at 67.1% with 16 searchers. What is causing the high idle time? First, although frontier splitting allows PVFSplit to avoid synchronizing at each split node, there is still a need to synchronize at the root of the game tree before the next variation an be searched. This synchronization occurs for each subtree at the root position, and for each iteration in the iterative deepening process. One can easily envision a different PVFSplit in which the need to synchronize at the root node is removed, but that enhancement is not implemented yet in *ParaChinook*. Second, the number of candidate split nodes only grows linearly with respect to the depth of the tree and, therefore, there may insufficient nodes on the current variation to keep all of the searchers busy. Recall that with a parallelism overload factor of 5, which is the case with 16 searchers, there needs to be 5 concurrent split nodes at all times in order to prevent starvation. Third, the impact of stragglers has not been addressed yet. When searchers have to wait for a straggler, the result is also idle time.

5.3 Improving Load Balancing: Straggler Preemption

As discussed earlier, the amount of search effort associated with each unit of work varies. Consequently, there exists a load balancing problem when some searchers are given more work than others. Searchers that force other searchers to wait at synchronization points, often because they have been given a larger piece of work, are referred to as *stragglers*. Stragglers cause synchronization overhead that appears as higher idle time.

Clearly, if there is a load balancing problem, the search effort should be re-balanced. The approach taken with Dynamic PVSplit (DPVSplit) [Sch89a] is to assign idle searchers to help the busy searchers. A simpler but more inefficient solution would be to interrupt the straggler and then apply PVFSplit to the subtree that was previously assigned to the slow searcher. PVFSplit would use the idle searchers, which now includes the straggler, to search the subtree in parallel. In essence, the subtree that caused the straggler is preempted and subdivided so that it can be searched in parallel. This is similar to the "shoot-down" procedure of the *Waycool* chess program [FeO88].

The main disadvantage of the interrupt-based approach is that the effort already invested into searching the subtree is largely lost. An interrupted search is incomplete and the returned result is useless. With DPVSplit, the straggler is not interrupted, so the parts of the subtree that have already been searched are not lost. Still, with the help of a transposition table, the entire computation of the straggler is not lost when it is interrupted. The parts of the subtree that have been completed can be found in the transposition table. Still, the interrupt-based scheme is clearly less efficient and less elegant than DPVSplit. Its main strength is that of simplicity of implementation.

Simplicity was also the goal in designing the definition of what constitutes a straggler. Clearly, it would be inefficient to wait until all of the searchers except for one, the obvious straggler, are idle. Therefore, when a quorum of searchers is idle for longer than a threshold length of time, the remaining searchers who are still computing become candidate stragglers. One of them is selected, interrupted and its subtree is searched in parallel. Currently, when more than a quarter of the searchers are idle for longer than 5 seconds, then the searcher who has been searching its assigned subtree for the longest time is selected for preemption. The quorum definition, time threshold and straggler choice have been derived from informal experiments. It has been observed that a small quorum and short time thresholds result in aggressive load balancing, which can be counter-productive if the overhead of interrupting the straggler is not outweighed by the potential benefits.

Although the straggler preemption technique described above is simple, it does appear to be effective in improving the parallel performance of both PVSplit and PVFSplit. More sophisticated load balancing techniques may be able to further improve the speedups, but as the technique grows in sophistication, so does the overhead of executing it. Still, there is room for further exploration of load balancing strategies in the future.

5.4 Experimental Results: Straggler Preemption

Table 5.2 presents the speedup performance of PVSplit with the simple load balancing strategy for stragglers. To distinguish it, PVSplit with the load balancing procedure is referred to as PVSplit-LB.

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.44	0.346	11.3	23.0	13.0	2.86
4	1.89	0.567	22.3	35.7	37.4	3.69
8	2.06	0.539	21.3	34.6	63.6	3.23
16	2.00	0.530	24.2	34.1	81.4	1.66

Table 5.2 — Principal Variation Splitting with Load Balancing (PVSplit-LB) Parallel Performance (speedup, search overhead and idle time)

For each configuration of searchers, PVSplit-LB achieves marginally higher speedups than PVSplit by reducing the idle time due to synchronization overhead. Given the number of idle searchers with PVSplit, the load balancing scheme is likely to be active in preempting the searchers who have work to reduce starvation. Load balancing and starvation prevention are closely related concepts. However, frontier splitting tries to reduce starvation by acting *a priori,* before idle time accumulates, and straggler preemption tries to balance the work *a posteriori*, after already incurring idle time. Consequently, the performance improvement due to frontier splitting is greater than the improvement due to straggler preemption.

Interestingly, the speedup declines (marginally) between 8 and 16 searchers. It is possible that starvation is forcing the load balancing mechanism to frequently interrupt searchers, incurring the overheads already discussed and decreasing overall performance.

Having both starvation prevention and load balancing is better than either strategy by itself. Table 5.3 shows the improved speedup values for Principal Variation Frontier Splitting with load balancing (PVFSplit-LB). PVFSplit-LB achieves the highest performance observed so far, with an average speedup of 3.32 with 16 searchers.

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.53	0.402	11.8	23.0	8.41	2.57
4	2.18	0.491	27.0	24.5	23.8	5.61
8	2.80	0.779	59.8	45.0	40.8	6.58
16	3.32	1.01	70.9	41.4	62.5	5.67

Table 5.3 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB) Parallel Performance
(speedup, search overhead and idle time)

The speedup curves for PVSplit-LB and PVFSplit-LB are in Figure 5.5. Note that the average speedup for PVFSplit-LB is always higher than the average speedup for PVSplit-LB.

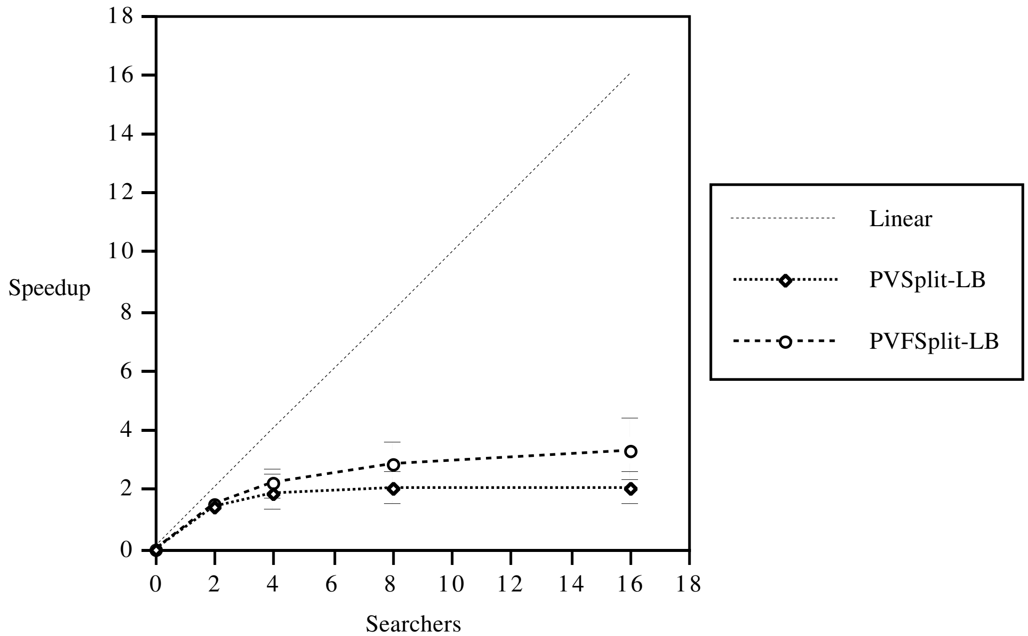

Figure 5.5 — PVSplit and PVFSplit with Load Balancing Speedups
(vertical bar indicates one standard deviation)

As the best performing algorithm of these experiments, PVFSplit-LB's utilization profile, given in Figure 5.6, is worth examining. Although the idle time portion still remains a significant part of the profile, it is substantially smaller than in the case of PVSplit without load balancing (Figure 4.5).

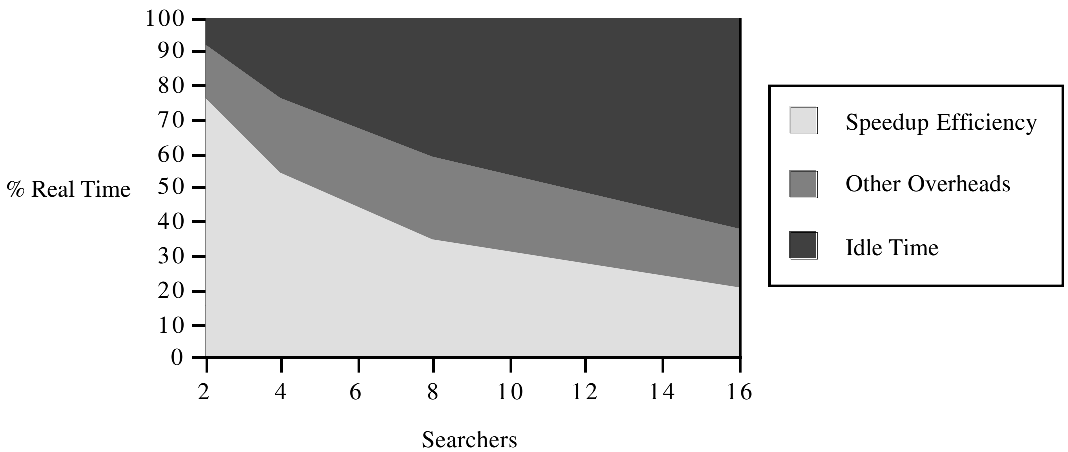

Figure 5.6 — Principal Variation Frontier Splitting (PVFSplit-LB) with Load Balancing
Normalized Utilization Profile

Frontier splitting and straggler preemption have been effective in increasing the speedup efficiency of the parallel search by trading off idle time for more speedup and getting higher search overhead as a negative side effect. Since neither search overhead nor idle time can be eliminated completely, PVSplit-based algorithms will always suffer from diminishing returns.

5.5 Summary of Experimental Results: All Algorithms

A total of four related algorithms have been evaluated. Principal Variation Splitting (PVSplit) is the basis of all of the algorithms. Because PVSplit suffers from starvation due to narrow game trees with low branching factors, Principal Variation Frontier Splitting (PVFSplit) was introduced to increase the average degree of parallelism. Because both PVSplit and PVFSplit suffer from load imbalances due to stragglers, a simple but useful straggler preemption strategy was implemented and evaluated.

In Figure 5.7, we can see the speedup curves for all four algorithms. Although frontier splitting and load balancing give PVFSplit-LB the best speedup curve, it remains in the same diminishing returns shape as PVSplit.

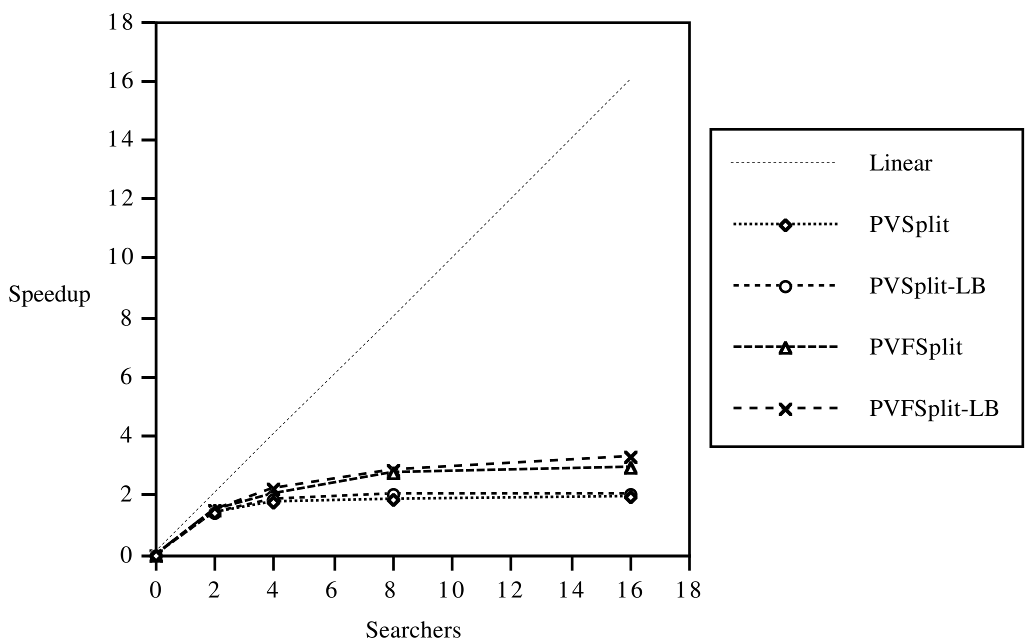

Figure 5.7 — All Algorithms Speedups
(higher curves are better)

Figure 5.8 shows the same speedup information as Figure 5.7, but with unequal axes ranges, in order to show a close-up of the speedup curves.

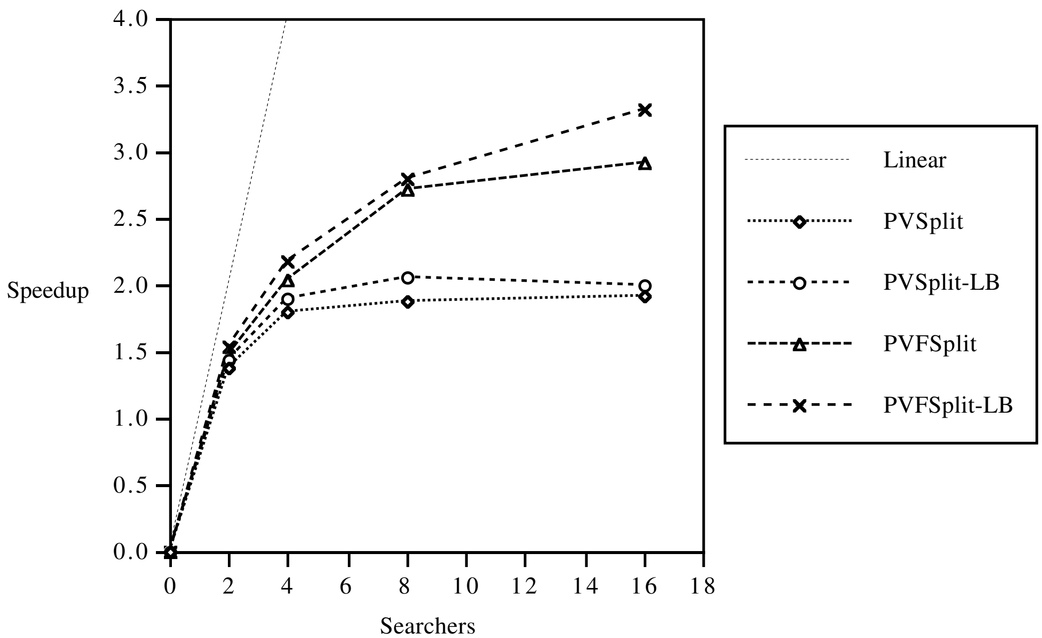

Figure 5.8 — All Algorithms (Close-Up) Speedups
(higher curves are better)

For completeness, Figures 5.9 and 5.10 show the search overhead and idle time graphs of all four algorithms.

The only anomaly occurs when the search overhead curves of PVFSplit and PVFSplit-LB cross over in Figure 5.9. Since interrupting a straggler results in the loss of partial search results, it is not surprising that for certain configurations of searchers, PVFSplit-LB has more search overhead than PVFSplit. However, it is speculated, interrupting a straggler and preempting its subtree typically means that the subtree is searched more quickly and the search result may be used to improve the parameters (i.e. windows) of other searches in progress sooner, thus reducing search overhead through reduced information deficiency.

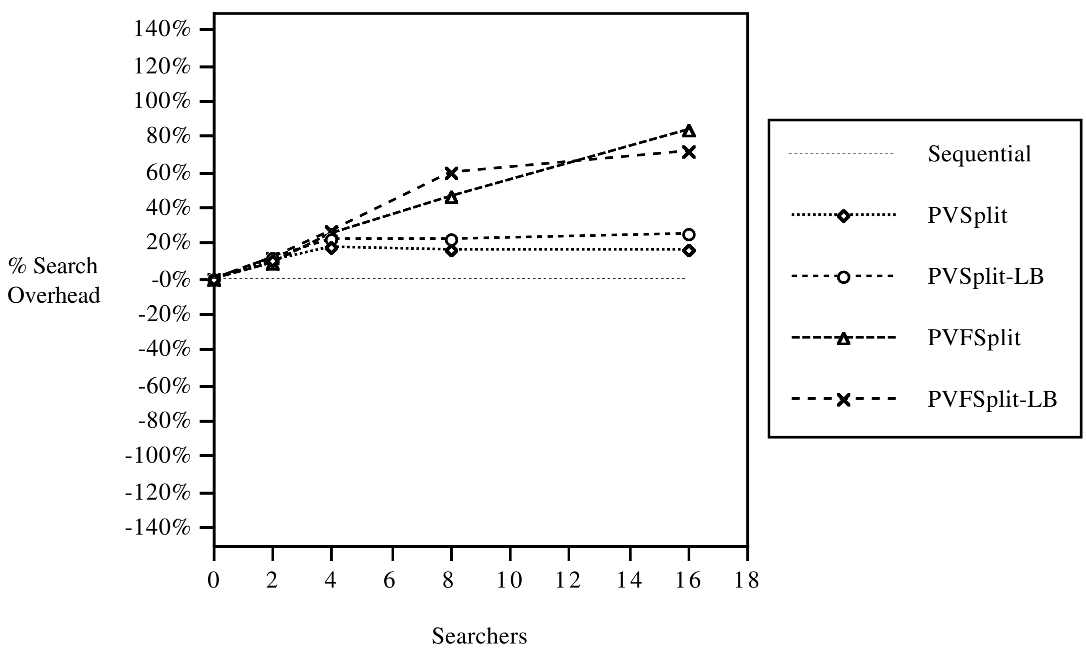

Figure 5.9 — All Algorithms Search Overhead
(lower curves are better)

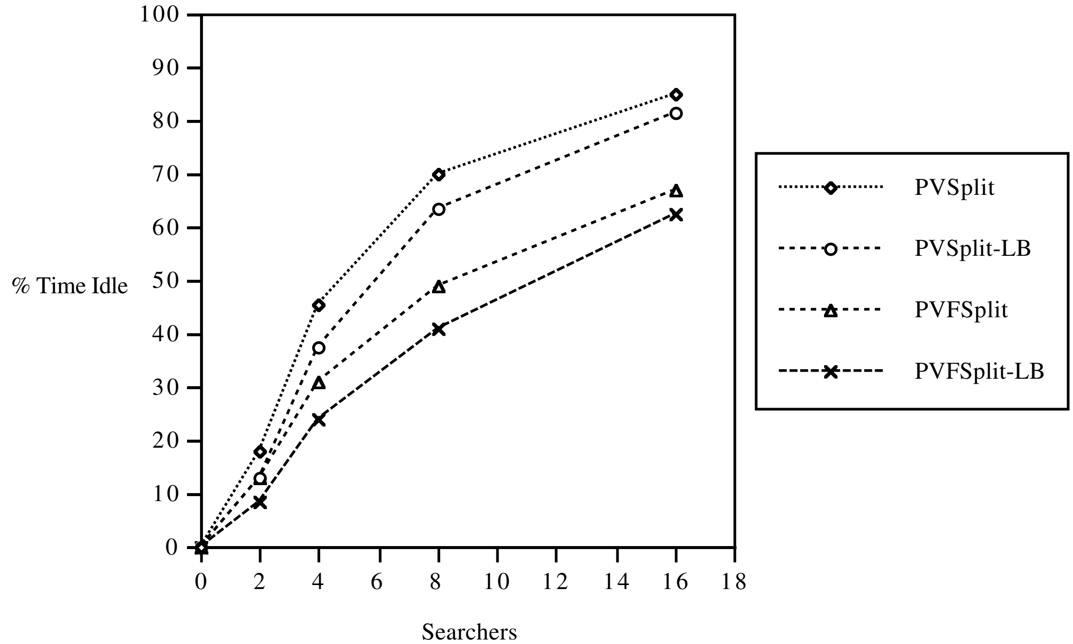

Figure 5.10 — All Algorithms Percent of Time Idle
(lower curves are better)

5.6 Concluding Remarks

From these experimental results, we can conclude that frontier splitting is an useful technique to reduce the starvation of searchers. Furthermore, the need for a load balancing strategy to deal with stragglers is also clear.

In retrospect, arguing that PVFSplit, both with and without load balancing, is better than PVSplit is somewhat unfair. Principal Variation Splitting was originally intended to form the *basis* of parallel search algorithms and does not address application-dependent issues such as game tree branching factor and stragglers. Nonetheless, the poor performance of PVSplit on narrow checkers game trees emphasizes the close relationship between branching factor and the potential speedup. Also, the experimental results point out that some of the successes of computer chess research cannot be directly adopted in other, closely related domains.