A Survey of Parallel Search Algorithms Over Alpha-Beta Search Trees Using Symmetric Multiprocessor Machines

Masters Project

James Swafford

Fall 2008

Advisor: Dr. Ronnie Smith

Table of Contents

1 Abstract....3
2 Background....3
2.1 Prophet....3
2.2 Terminology....3
2.3 Alpha-Beta....4
2.3.1 The Alpha-Beta Algorithm....4
2.3.2 Node Types....7
2.3.3 An Analysis of Alpha-Beta....8
3 Parallel Search Algorithms....9
3.1 Shared Transposition Tables....11
3.1.1 The Shared Transposition Table Algorithm....11
3.1.2 The Shared Transposition Table Implementation in Prophet....11
3.1.3 Shared Transposition Table Performance in Prophet....12
3.2 Root Splitting....14
3.2.1 The Root Splitting Algorithm....14
3.2.2 The Root Splitting Implementation in Prophet....14
3.2.3 Root Splitting Performance in Prophet....15
3.3 Young Brothers Wait (YBW)....16
3.3.1 The YBW Algorithm....16
3.3.2 The YBW Implementation in Prophet....17
3.3.3 YBW Performance in Prophet....20
3.4 Dynamic Tree Splitting (DTS)....21
3.4.1 The DTS Algorithm....21
3.4.2 The DTS Implementation in Prophet....22
3.4.3 DTS Performance in Prophet....28
4 Conclusions....29
5 Future Work....29
5.1 How Far Can We Go?....29
5.1.1 Linear Scaling as an Upper Bound....29
5.2 Beyond SMP....30
5.2.1 Limitations of SMP Architectures....30
5.2.2 NUMA Architectures....30
5.2.3 Distributed Computing....30
5.3 Improving Split Point Selection in DTS....31
6 Bibliography....32
6.1 Publications....32
6.2 Books....32
6.3 Software....32
6.4 Websites....32

1 Abstract

The goal of this project is to explore various algorithms to parallelize alpha-beta search trees. In this paper four parallel search algorithms are presented, each designed to run on symmetric multiprocessor (SMP) architectures. In order of complexity, these algorithms are : (1) Shared Transposition Tables, (2) Root Splitting, (3) Young Brothers Wait (YBW), and (4) Dynamic Tree Splitting (DTS). Each of these algorithms was studied in detail and implemented in the computer chess program Prophet. This paper begins with an overview of the alpha-beta algorithm, followed by a description of the parallel search algorithms. The implementation of each algorithm in Prophet is described and presented with empirical results. Finally, ideas for future work are discussed.

2 Background

2.1 Prophet

Prophet is a chess engine that has been developed solely by me, starting in the Spring of 2000. Prophet is a conventional chess engine, meaning the core algorithms are used by many other programs. Over the years I have exchanged ideas with other chess engine authors via web based discussion forums, tournaments, email, and the sharing of source code.

Prophet utilizes a (depth first) alpha-beta search, along with several move ordering heuristics and techniques to extend "promising" lines of play and reduce or prune lines that are likely no good. Leaves of the search tree are assigned scores using an evaluation function that assesses the position using heuristics that can be found in most chess books, such as "doubled rooks on open files are good" and "isolated pawns are bad." Material imbalance is by far the most dominant term in the evaluation function.

More information about Prophet can be found at http://chessprogramming.org/prophet/ .

2.2 Terminology

- *Search tree*: an acyclic graph, with vertices representing chess positions, and edges representing moves
- *Ply* : distance from the root of the tree
- *Node*: a position (vertex) within the search tree

- *Quiescence search* : limited search done to ensure leaf nodes are evaluated in "quiet" positions. In most programs the quiescence search examines captures, and possibly checks or check escaping moves

- *Depth*: plies remaining until the search transitions into the quiescence stage

- *Transposition table:* a hash table, with the key representing the chess position and the value(s) representing bounding/score information and suggested move(s)

- *Iterative deepening*: the process of doing a depth 1 search, followed by a depth 2 search, up to a specified depth or until time runs out. The benefit of improved move ordering by use of the transposition table and other data structures usually makes an iterative search to depth N faster than a simple depth N search.

- *Split point* : Nodes that are being searched in parallel (or that are set up to be searched in parallel)

2.3 Alpha-Beta

2.3.1 The Alpha-Beta Algorithm

The alpha-beta algorithm is a technique that can significantly reduce the number of nodes required to obtain an evaluation. The algorithm is theoretically sound, meaning the result will always be equal to the result obtained by performing a min-max search (or the recursive variation "negamax") on the same tree. Allen Newell and Herbert Simon are credited with proposing the algorithm in the late 1950s [5].

An alpha-beta search works by passing bounding values through the search. Alpha serves as a lower bound on a node's true value, and can never decrease, while beta serves as an upper bound, and can never increase [5]. Initially, alpha is set to $-\infty$ and beta to $+\infty$. As the zero-sum property holds for the game of chess (what's good for player A is equally bad for player B), the inverse of the lower bound for the player on move is the upper bound for the opponent, and the inverse of the upper bound for the player on move is the lower bound for the opponent. Hence, on recursive calls to the alpha-beta search function, -beta is passed as the alpha parameter and –alpha as the beta parameter. When the search reaches its first leaf node, alpha is raised and the value propagated back up through the tree. Some pseudocode is below:

```
int search(Position pos,int alpha,int beta,int depth) {
        if (depth <= 0)
                return evaluate();
        List moves = genMoves();
        while (move = moves.next()) {
```

```
                pos.makeMove(move);
                int score = -search(pos,-beta,-alpha,depth-1);
                if (score > alpha) {
                        if (score >= beta)
                                return beta;
                        alpha = score;
                }
                pos.unMakeMove(move);
        }
        return alpha;
}
```

Rules for using the alpha-beta (α-β) bounds are as follows:

1. If a recursive call to search returns a value <= α, the move just searched is inferior to another line this player can force. Don't change α or β. Continue to search for better moves.
2. If a recursive call to search returns a value > α and < β, the move just searched is better than any move the player on move has found so far, and can't be refuted by the opponent. Raise α to the returned score and continue searching for an even better move.
3. If a recursive call to search returns a value >= β, this move is "too good" for the player on move. The opponent can prevent the player from reaching this line with perfect play. Don't search any more child nodes; return β.

The following search tree illustrates the algorithm:

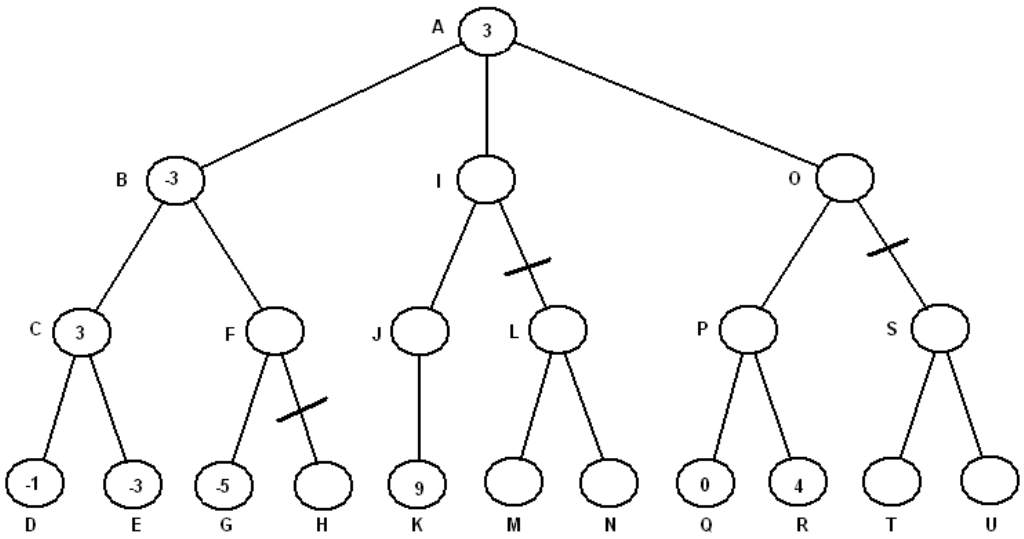


Figure 1 – an Alpha-Beta Search Tree

1. The search is initiated with $\alpha = -\infty$ and $\beta = +\infty$.

2. Node C examines both child nodes D and E and determines the best move is the one that leads to E.

3. C passes its best score of 3 back up to B, which, when negated, raises α to -3. The bounds passed to F are $\alpha=-\infty$, $\beta=3$.

4. G is examined and given a score of 5, which exceeds β. F returns β (+3) without examining H.

5. B returns -3 to A which raises α to +3.

6. I is called with $\alpha=-\infty$, $\beta=-3$.

7. J is called with $\alpha=+3$, $\beta=+\infty$.

8. K is called with $\alpha=-\infty$, $\beta=-3$. Eval() returns +9 which exceeds β. β is returned to J.

9. J returns $\alpha=+3$ to I.

10. I receives +3 from J, which when negated to -3 is equal to β. I returns β (-3) without examining L.

11. A receives -3 from I. α is not raised.

12. O is called with α=-∞, β=-3.

13. P is called with α=+3, β=+∞.

14. Q is called with α=-∞, β=-3. Eval() returns 0, a fail high. -3 is returned to P.

15. R is called with α=-∞, β=-3. Eval() returns +4, a fail high. -3 is returned to P.

16. P returns α=+3 to O.

17. O receives 3 which causes a fail high. It returns β=-3 to A without examining S.

18. A receives -3 and does not raise α.

19. The final result is α=3, with the best line A->B->C->E.

2.3.2 Node Types

There are three types of nodes in an alpha-beta search tree. They are PV, CUT, and ALL nodes [1]. (Or, to use the terminology used by Knuth when he originally described them in his 1975 paper "An Analysis of Alpha-Beta Pruning," [3] Type 1, Type 2, and Type 3 nodes.)

PV Nodes: PV nodes are only encountered along the left edge of the tree. Therefore, there are very few of them. With a tree of height *d*, there will be at most *d* PV nodes. The parent of a PV node is always a PV node. The first child of a PV node is a PV node. Other children are CUT nodes. Every child of a PV node must be searched.

CUT Nodes: A CUT node is a node in which a beta cutoff is possible. Once a child is examined with a score >= beta, the search can return, leaving the remaining children unsearched. Therefore, it is desirable to try to sort the moves in a way that the moves most likely to lead to a beta cutoff are searched first. In a perfectly ordered tree, a CUT node searches only one child. CUT nodes are children of PV or ALL nodes. CUT nodes should be avoided as split points, since a move that returns a score >= beta renders other searches still in progress useless.

ALL Nodes: ALL nodes are nodes in which every child must be searched. They are children of CUT nodes. Children of ALL nodes are CUT nodes. Since every child of an ALL node must be searched, ALL nodes are a good place to create split points.

Figure 2 illustrates the three node types in a tree of depth 4.

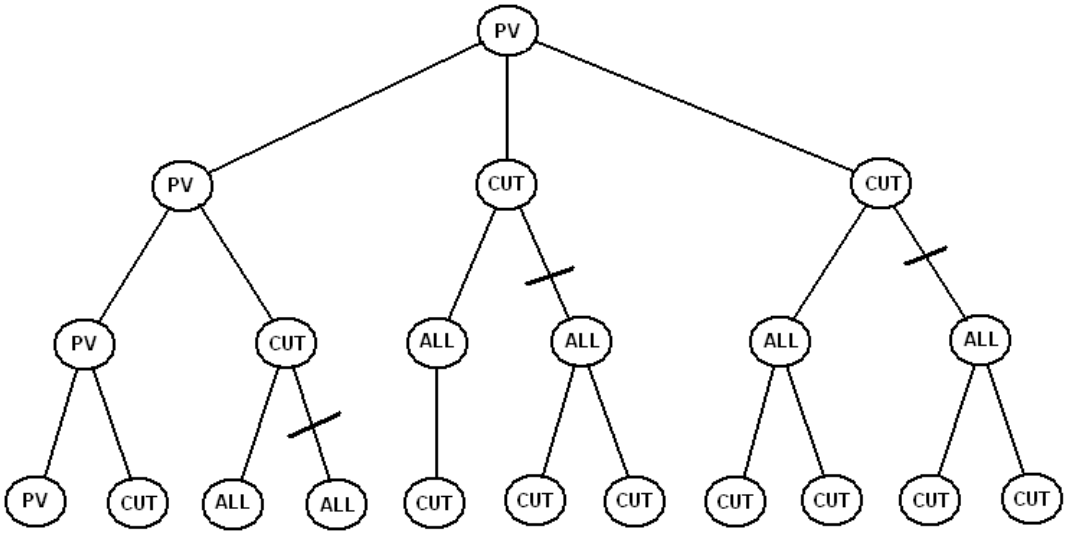


Figure 2 – Node Types in an Alpha-Beta Search Tree

2.3.3 An Analysis of Alpha-Beta

Given a branching factor of b and a search depth of d, a min-max search will visit $O(b^d)$ leaf nodes. With perfect move ordering, alpha-beta will visit $O(b^{d/2})$ nodes. This means the search can go twice as deep in the same amount of time! With worst case move ordering, alpha-beta degenerates into a min-max search. In chess, the branching factor is not constant of course, but research shows the average branching factor to be 35-40 in the middle game.

Consider the alpha-beta tree shown in Figure 3. For simplicity, b is a constant (of 2). The nodes are labeled as type 1, 2, or 3 nodes corresponding to PV, CUT, and ALL nodes respectively. Unvisited nodes are unlabeled.

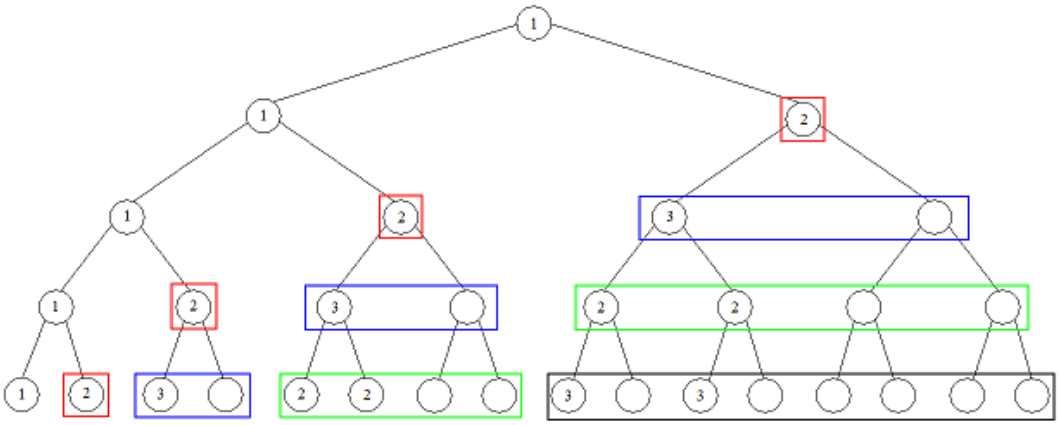


Figure 3 – Node Sequences in an Alpha-Beta Search Tree

A property of alpha-beta is that for each PV node N_i, $1 <= i < d$, there is a subtree T_i of height d-i structured as follows: odd plies, including the root of the subtree, consist of CUT nodes. Even plies consist of ALL nodes.

As the root of T_i is a CUT node, it will visit one ALL node at ply 2. As the only visited node at ply 2 is an ALL node, b CUT nodes will be visited at ply 3, each of which will visit one ALL node at ply 4. In general, every node visited at an even ply will expand b children, while every node visited at an odd ply will expand one child. So, the number of visited nodes for each subtree is $O(1 * b * 1 * b * \ldots * 1 * b)$ for subtrees of an even height, or $O(1 * b * 1 * b * \ldots * 1)$ for subtrees of an odd height. In each case, the size of the subtree is $O(b^{d/2})$.

Since there are d such subtrees, each with $O(b^{d/2})$ nodes, the entire tree contains $O(b^{d/2})$ nodes.

3 Parallel Search Algorithms

The goal of parallelizing a search is simply to find solutions faster than a sequential search. To accomplish this parallelism should be maximized and overheads minimized. Overheads can broadly be grouped into two categories: search overhead and synchronization overhead.

Amdahl's law states that a program's maximum achievable speedup is bounded by the percent of code that runs sequentially. If only a portion P of a program is parallelized, and the gain of that effort is S, then speedup will be 1/((1-P)+P/S).

To see the effects of this, consider a program consisting of two components. The first of these components consumes 60% of the CPU, and the second 40% of the CPU. If only the second component is parallelized (P=0.4), then even if the second component is parallelized down to zero time (S=+∞), the overall speedup would still only be 1 / (1-0.4) = 1.67. That is, even with an infinite number of processors, this program will never execute more than 1.67 times faster than a sequential version! Clearly as much of the program execution as possible should be parallelized.

Search overhead is any work done in a parallel search that would not be done in a sequential search. If a sequential search visits 100,000 nodes and a parallel version of that search visits 110,000 nodes, there is a 10% search overhead, since 10% more nodes were visited than would have been if just searching in a serial fashion.

Synchronization overhead is the time spent partitioning the work, assigning it to a processor, and collecting and processing the results. In the context of alpha-beta searching this includes the split point selection algorithms, copying of data structures, time spent blocked by a lock around a critical section, and any time a processor must sit idly waiting on another processor to finish a computation.

It follows that good parallel search algorithms have little or no sequential code in the critical path, and minimize search and synchronization overheads.

The main difficulty with parallelizing an alpha-beta search is that alpha-beta is a sequential algorithm. The efficiency of alpha-beta lies in the fact that it uses a priori knowledge [1]. Results from one branch of computation are used to create bounds for subsequent branches. Tighter bounds make cutoffs occur more frequently, which translates into fewer nodes searched. If a branch in a parallelized search is searched before previous branches have been completely searched, then the bounds may not be as tight as they would have been in a sequential search, resulting in fewer cutoffs and higher search overhead. Conversely, if the constraints to split a node are overly rigid, parallelism is potentially being lost, resulting in more synchronization overhead. The challenge therefore is minimizing both.

A note on terminology before describing the algorithms: Prophet uses a thread based implementation for each of the algorithms. Therefore, the term “thread” is used when describing the implementations. In the sections defining the algorithm the more universally accepted term “processor” is used. For the purposes of this report, the two terms are synonymous.

3.1 Shared Transposition Tables

The Shared Transposition Table algorithm is trivial to implement, provided the search routines are already thread safe (assuming a thread based implementation, as opposed to a true multi-processor implementation using shared memory) – meaning multiple searches can be run concurrently without the risk of unwanted interaction.

3.1.1 The Shared Transposition Table Algorithm

At the root of the search tree, multiple search processors are started, all processing their own copy of the root move list. Each processor processes the same list of moves, independently of other processors. When processor 0 finishes searching the entire move list, any remaining busy processors are stopped. Only processor 0 is allowed to update the score and best line at the root.

The benefit of running multiple processors comes from the information that they enter into the transposition table. The idea is that processors 1-N will enter information into the transposition table that will be useful to processor 0, such as tighter search bounds (which may lead to cutoffs without searching), exact scores, and suggested moves.

The obvious drawback to this algorithm is the extremely high search overhead.

3.1.2 The Shared Transposition Table Implementation in Prophet

A single data structure, `SharedHashSearchThread`, was added to Prophet to make the Shared Transposition Table algorithm work.

SharedHashSearchThread

Each search thread gets a single `SharedHashSearchThread` structure, which provides stacks for move generation and other search related data specific to a single thread. It also wraps a `pthread_t` object and includes some control flow boolean variables.

```
typedef struct {

        // Each search thread must work independently of the others.  They each operate
        // off their own position, generate moves on their own stack, etc.
        Position pos;
        Move moveStack[MOVE_STACK_SIZE];
        SearchData searchStack[MAX_PLY];
        Undo undoStack[UNDO_STACK_SIZE];
        SearchStats stats;

        // okToStart==true signals the busy-wait loop to let the thread enter the
        // Root() search
        volatile bool okToStart;

        // die==true signals the busy-wait loop to let the thread exit (and terminate)
```

```
        volatile bool die;

        // is this thread idle?
        volatile bool idle;

        // the thread object
        pthread_t t;
} SharedHashSearchThread;
```

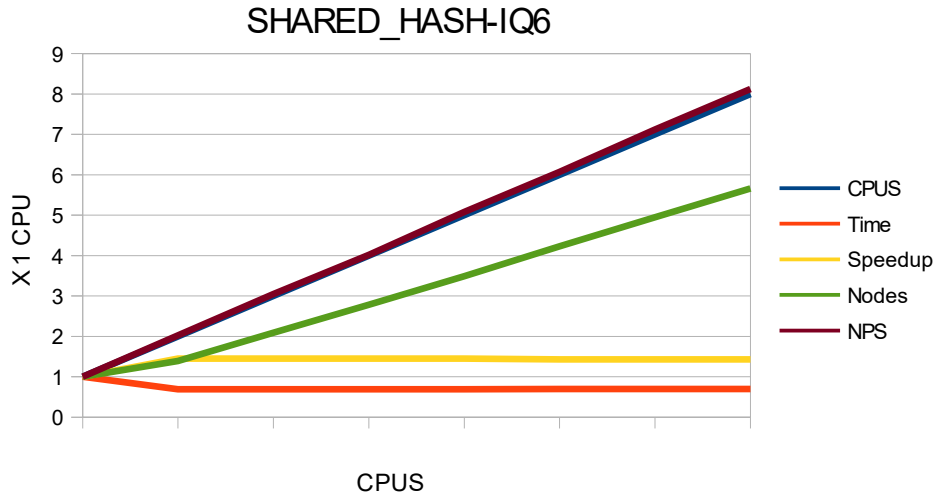
On program initialization, threads 1-N (thread 0 is the main thread) are put into a busy-wait loop. At the root of the search tree, thread 0 generates the move list and prepares the `SharedHashSearchThread` structures for threads 1-N. Once everything is set up, the "okToStart" signal is sent to each search thread, which prompts them to begin searching.

```
void* SharedHashIdleLoop(void* thread) {
        int myThreadID = *(int*)thread;
        SharedHashSearchThread *st = &sharedHashSearchPool[myThreadID];

        while (!st->die) {
                if (st->okToStart) {
                        st->idle=false;
                        Root(myThreadID);
                        st->okToStart=false;
                        st->idle=true;
                }
        }

        return 0;
}
```

Threads 1-N leave `Root()` and return back to the busy-wait loop either as a result of finishing the root move list, or being signaled to stop by thread 0.

3.1.3 Shared Transposition Table Performance in Prophet

To measure performance, a test suite of 188 problems called "IQ6" was run to a fixed depth of 8 ply. The test suite was processed eight different times, each run using one more processor than the last (1-8 processors).



CPUS	Time	Speedup	Nodes	NPS
1	1	1	1	1
2	0.69	1.45	1.39	2.02
3	0.69	1.45	2.09	3.05
4	0.69	1.45	2.78	4.02
5	0.69	1.45	3.49	5.08
6	0.7	1.43	4.23	6.07
7	0.7	1.43	4.95	7.12
8	0.7	1.43	5.66	8.12

Figure 4 – Shared Transposition Table Performance in Prophet

The results show that the average time to solution for two processors was 69% of the average time for a single processor, yielding a speedup of 1.45. Adding additional processors yielded no additional gain. The Nodes column is a measure of search overhead. It shows how many nodes were searched relative to the single processor run. NPS is "nodes per second." As could be expected, each scaled linearly with the number of processors.

3.2 Root Splitting

3.2.1 The Root Splitting Algorithm

The Root Splitting algorithm is similar to the Shared Transposition Table algorithm in the respect that additional processors are used to process the root node. Instead of each move being (potentially) searched by each processor, each move is searched by exactly one processor. The processors take a "divide and conquer" approach to processing the root move list.

When a processor 1-N runs out of work at the root, it exits back to the busy-wait loop. When processor 0 finishes, it must wait until processors 1-N are idle, then it returns back up to `Iterate()` where the iterative deepening driver will either repeat the search with a deeper depth or make the best move over the board.

3.2.2 The Root Splitting Implementation in Prophet

Two data structures were needed to implement Root Splitting. The first, `RootSplitSearchThread`, is identical to `SharedHashSearchThread`.

The second necessary data structure is `RootInfo`. A single `RootInfo` object is shared among all threads. It contains the position at the root, the root move lists, and other search related data at the root.

RootInfo

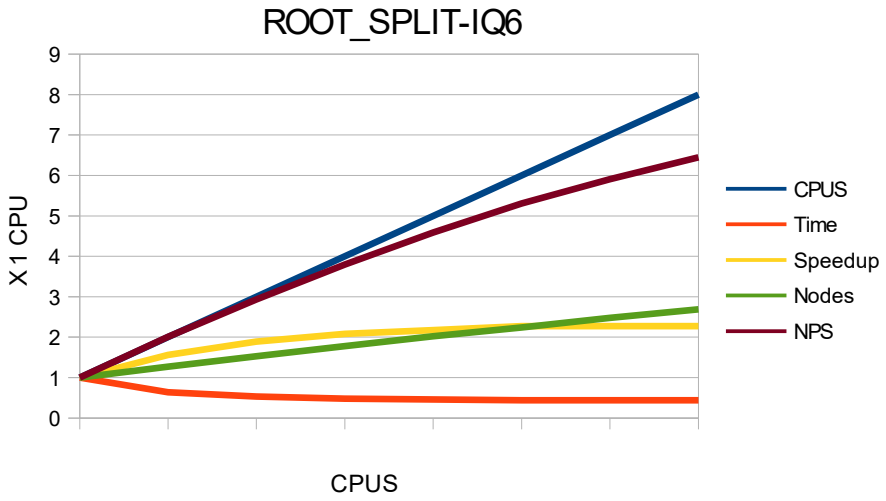
```
typedef struct {

        // The position, move stacks, and search data at the root are shared among
        // all processors.
        Position pos;
        Move moveStack[MOVE_STACK_SIZE];
        SearchData searchStack[MAX_PLY];
        Undo sharedUndoStack[UNDO_STACK_SIZE];
        SearchStats stats;

        // What stage of the game are we in at the root?
        // enum GAME_STAGE { OPENING,MIDDLEGAME,ENDGAME,MATING };
        int gameStage;

        // the alpha and beta bounds at the root are accessible to all processors.
        volatile int alpha;
        int beta;

        // The depth to search.
        int depth;

        // An array recording the number of nodes processed for each root move.
        unsigned long nodes[300];

} RootInfo;
```

With the proper data structures in place the only remaining tasks were to set up the `RootSplitSearchThread` structures before a search, which mainly involved creating copies of the search position for each thread, and modifying `Root()`, the function responsible for processing the root node.

The main modifications to `Root()` were protecting the move selection function and the code responsible for processing scores as they are returned from child nodes with exclusive locks. Additionally, some code was put in place to force thread 0 to wait for all other threads to idle before returning back to the iterative deepening function.

3.2.3 Root Splitting Performance in Prophet

To measure performance, a test suite of 188 problems called "IQ6" was run to a fixed depth of 8 ply. The test suite was processed eight different times, each run using one more processor than the last (1-8 processors).



CPUS	Time	Speedup	Nodes	NPS
1	1	1	1	1
2	0.64	1.56	1.27	2.01
3	0.53	1.89	1.53	2.93
4	0.48	2.08	1.78	3.8
5	0.46	2.17	2.02	4.59
6	0.44	2.27	2.24	5.31
7	0.44	2.27	2.48	5.91
8	0.44	2.27	2.69	6.45

Figure 5 – Root Splitting Performance in Prophet

The performance results for Root Splitting are an improvement over Shared Transposition Tables, but the speedup curve begins to flatten out badly after four processors, with no additional gain after six processors. The nodes searched and NPS appear to scale with the number of processors.

The first branch from the root of an alpha-beta search produces a subtree much larger then the remainder of the branches, when the first branch is best. When the first move is not the best, the "new best" will also produce a subtree much larger than the other moves. [1] With good move ordering, the first branch searched at the root is often the best. It is rarely the case that the best move at the root is searched towards the end of the list, explaining the flattening of the time and speedup curves.

It should also be noted that once the number of processors available exceeds the number of available moves to search (which can be quite low in the endgame), the extra processors will sit idle, making this algorithm unattractive from a scalability perspective.

3.3 Young Brothers Wait (YBW)

The Young Brothers Wait (YBW) algorithm is conceptually simple, elegant, and is not very difficult to implement on top of an existing recursive framework. Rather than just splitting at the root of the tree as Root Splitting does, YBW splits nodes anywhere in the search tree.

3.3.1 The YBW Algorithm

In his thesis "Parallel Alpha-Beta Search on Shared Memory Multiprocessors," [4] Valavan Manohararajah describes two version of Young Brothers Wait. In the first, "weak YBW," there is but a single requirement that before a node can be searched in parallel, the first child must be completely searched. Or, in other words, the "young brothers" must wait until the "eldest brother" has been searched.

The second version of YBW is "strong YBW." In strong YBW, nodes are classified as Y-PV, Y-ALL, and Y-CUT, corresponding to the PV, ALL, and CUT classifications described in section 2.3.2. The only difference is between CUT and Y-CUT nodes. CUT nodes, by definition, have but a single child that produces a beta cutoff. In practice search trees are not perfectly ordered, so multiple children may be searched before a beta cutoff is found. In a YBW search, these nodes are labeled Y-CUT nodes. Strong YBW simply states that in addition to the weak YBW constraint that the eldest brother must be searched first, all "promising" nodes of a Y-CUT node must be searched before it can be searched in parallel. It is left to the programmer to decide what constitutes a promising node.

In either version of YBW, once a processor decides it wants to create a split point, it checks to see if there are any available processors to aid in searching the remaining moves. If so, it sets up the appropriate data structures so that its remaining children can be searched in parallel, and "grabs up" any idle processors and puts them to work. The splitting processor becomes the owner (or master), and any idle processors that join in are helpers (or slaves). Note there is no real decision making taking place, other than a processor deciding it wants some help. The idle processors have no choice in the matter – they will be put to work by the first processor that comes along wanting some help.

Once a helper finishes searching a move at a split point and finds there is no remaining work, it returns to an idle state where it is available to help search at other split points. Once an owner finishes searching and finds there is no remaining work, it must determine if any helper processors are still busy searching. If not, it is free to return the score and best line back up to its parent node. If not, it must wait until its helpers have completed their searches before returning back up. Of course, sitting idly by waiting on other processors to finish their work increases synchronization overhead and drags down parallel search efficiency. The "helpful master" concept addresses this issue by putting the owner to work as a helper to one of its remaining helpers.

3.3.2 The YBW Implementation in Prophet

Young Brothers Wait was the first parallel search algorithm implemented in Prophet, with the help of Tord Romstad's "Viper" [7], an instructional computer chess program. There was a bit of groundwork to do to prepare the program to search with multiple threads. This mainly involved modifying the search routines to use pointers to data structures (that are passed in as arguments) rather than global data structures. For example, there is a global `Position` structure `gpos` used to track the current state of the position. Functions that accessed `gpos` directly within the search (move generation routines, move ordering routines, etc.) were modified to take a pointer to a `Position` structure as an argument and operate off that. Similar modifications were made for the move stack, search data stack, and undo stack. The hash tables and history table remained global.

Leaving the hash table global meant information could be shared between search threads. In Prophet, a hash entry is composed of two 64 bit words, the first being a key and the second the value (which includes score, bound type, and a recommended move). This meant that reading and writing were not atomic operations, so the synchronization issue had to be addressed. Specifically, this meant avoiding situations in which thread 1 would read the first word (the key), determine it was a "hit," then thread 2 would write both words to the same memory location, and finally thread 1 would read the now erroneous value word. Initially MUTEX type locks were used to handle this, but were later removed in favor of a lock-less hashing implementation. Lock-less hashing does not prevent the synchronization issue described above, but it does provide a method to detect it [2]. The time saved by not having to lock the hash table before a read or write more

than offsets the occasional “bad data” that has to be tossed out as a result of a synchronization error.

Once the groundwork was complete it was time to create the data structures that would make the algorithm work.

YBWSplitNode

```
typedef struct {

        // The stats structure at the split point.
        SearchStats *splitPointStats;

        // The search data structure at the split point.
        SearchData *splitPointSearchData;

        // The position at the split point.
        Position *splitPointPosition;

        // Each thread gets its own copy of these structures to work with, as well as its
        // own move and undo stacks.
        SearchStats stats[MAX_SEARCH_THREADS];
        SearchData searchStack[MAX_SEARCH_THREADS][MAX_PLY];
        Position spos[MAX_SEARCH_THREADS];
        Move moveStack[MAX_SEARCH_THREADS][MOVE_STACK_SIZE];
        Undo undoStack[MAX_SEARCH_THREADS][UNDO_STACK_SIZE];

        // The ply and depth remaining at the split point.
        int ply;
        int depth;

        // alpha and beta are the bounds at the split point.  Threads may raise the
        // alpha bound as they find better moves.
        volatile int alpha;
        volatile int beta;

        // The master is the thread that begins the split and that will continue
        // searching once processing of the split point is complete.
        int master;

        // For each element in the slaves[] array, TRUE indicates the thread
        // corresponding to this index is a helper/slave thread at this split point.
        bool slaves[MAX_SEARCH_THREADS];

        // How many threads are currently working at this split point?
        volatile int cpus;

        lock_t lock;
} YBWSplitNode;
```

`YBWSplitNode` contains some shared information about the split point itself, such as the position at the split point, the ply and depth remaining at the split point, and the alpha and beta bounds at the split point. It also contains some data structures that are “private” to a specific thread, such as `spos[ ]`, an array of `Position` structures, each one belonging to a different thread.

`YBWSplitNodes` also contain arrays of search stacks, undo stacks, move stacks, and search positions – one element for each thread. The reason these stacks are placed

here, as opposed to the `YBWSearchThread` structure (described below), is that the "helpful master" concept makes it is possible for each thread to participate in multiple split points simultaneously.

As an example, consider a split point in which the the master returns from searching a child node to find there are no more moves to be searched, but a slave thread is still busy searching. The master can't abandon the split point since it's responsible for continuing the search once all slaves are finished. But it can help one of the slave threads process its subtree. So the master is now actively working on two split points.

YBWSearchThread

```
typedef struct {

        // when work is waiting, it's waiting at "splitNode"
        YBWSplitNode* splitNode;

        // If stop == TRUE, this thread should stop searching immediately and "wind
        // back up" to the idle loop.
        volatile bool stop;

        // Is this thread doing anything?
        volatile bool idle;
        volatile bool workIsWaiting;

        // How many split points is this thread processing?
        int activeSplitNodes;

        pthread_t t;
} YBWSearchThread;
```

`YBWSearchThread` objects represent a thread of execution. When the "`workIsWaiting`" flag is true, the thread will transition from the idle loop to YBWSearch( ) where it will begin searching at `splitNode`. `YBWSearchThread` also contains boolean variables indicating whether the thread should stop or if it is idle, and an integer value representing the number of split points the thread is participating in.

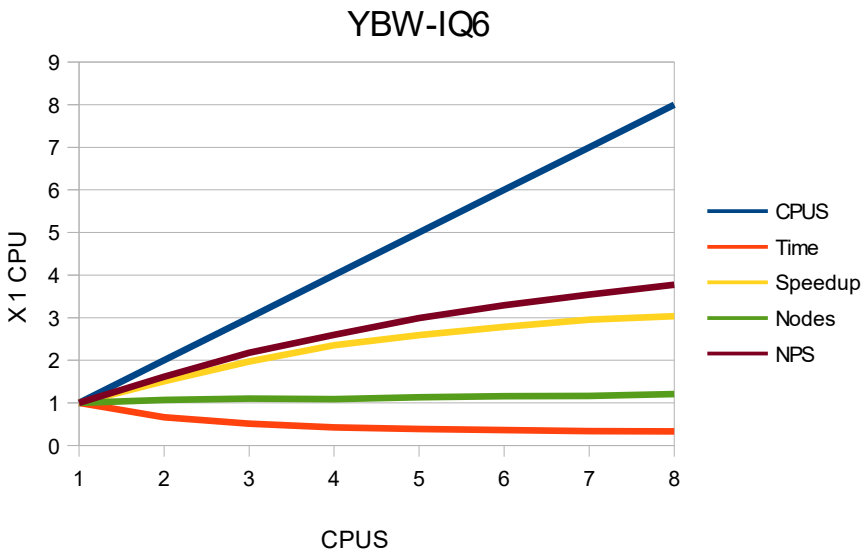
Since YBW works well with a recursive search, the implementation is fairly straightforward. At the bottom of the "move loop" in the search function, a check is done to determine if the node is eligible to be split. Since the test is at the bottom of the move loop it's given that the eldest brother has already been searched, but other constraints can be added as well. If the node is determined to be eligibile, it calls `YBWSplit()`. `YBWSplit()` returns a boolean value indicating it did or did not split the node and search it in parallel. If the return value is true, processing of this node is complete. If false, execution cycles back to the top of the move loop.

`YBWSplit()` checks if any threads are available (idle). If so, it sets up the split point, makes each idle thread a slave, and sends the master to `YBWIdleLoop()`. The

master and each slave will observe the `workIsWaiting` flag and enter `YBWSearch()`, where they will finish processing the split point in parallel.

Once a thread finds there is no more work to do at a split point, it exits back to `YBWIdleLoop()`. Once all slave threads are finished the master thread leaves `YBWIdleLoop()`, back through `YBWSplit()`, and finally back to the search routine. The return value from `YBWSplit()` will be TRUE, indicating processing of this node is complete.

3.3.3 YBW Performance in Prophet



CPUS	Time	Speedup	Nodes	NPS
1	1	1	1	1
2	0.66	1.51	1.07	1.61
3	0.51	1.97	1.1	2.18
4	0.43	2.35	1.09	2.6
5	0.39	2.59	1.13	2.99
6	0.36	2.78	1.16	3.29
7	0.34	2.95	1.16	3.54
8	0.33	3.03	1.21	3.77

Figure 6 – YBW Performance in Prophet

Performance was again measured using the IQ6 test suite. YBW scales significantly better than Root Splitting. Performance continues to improve with additional processors, up to at least eight processors. With eight processors, Prophet processed the test suite in 33% of the time required for the single processor version to complete it. Search overhead is quite low, with only 21% more nodes searched using eight processors than with one. Speedup appears to be tracking very closely to NPS scaling. Unfortunately, NPS scaling seems to be flattening out around 4X, which will seriously inhibit performance with additional processors.

3.4 Dynamic Tree Splitting (DTS)

The Dynamic Tree Splitting (DTS) parallel search algorithm was developed by Dr. Robert Hyatt, with the goal of improving search efficiency by eliminating situations in which one processor must sit idle while another processor finishes its computation. The DTS approach is fundamentally different than the YBW approach, offering more flexibility, at the cost of more complexity.

3.4.1 The DTS Algorithm

Like YBW, in DTS a split point can be created anywhere in the search tree. Unlike YBW, however, the splitting processor does not have to be working at the split ply in order to create the split point. This gives the algorithm a lot of flexibility in creating split points.

Another key difference between YBW and DTS is how split points are created. Recall that in YBW idle processors "sit around" until some other processor puts them to work. In DTS, idle processors actively look for work to do by sending HELP messages to busy processors. As the busy processors receive these HELP messages, they send some "state of the search" information to the requesting processor. The idle processor collects and analyzes this information from each of the busy processors, and decides what processor it wants to split with, and where.

Once the idle processor has decided where to split, it alerts the appropriate busy processor. The busy processor sets up the split point (which may be at a ply much higher in the tree than the busy processor is currently at), and the idle processor attaches to the split point and begins searching.

A key feature in DTS is the "peer to peer" design. There is no concept of node ownership. That is, there is no "master." That's not to say a node cannot have multiple

processors working on it, but each is "equal," regardless of which processor initiated the search on that node. When a processor finishes working at a split point and discovers there is no more work to do, one of two things must happen:

Case 1: The processor is not the last processor.

In this case one or more processors are still busy, so the end result of this node is not known. The processor simply merges its results and leaves. At this point the processor is returned to an idle state, where it should look for more work to do.

Case 2: The processor is the last processor.

In this case, no other processors are busy working at this node. This processor is responsible for "returning back up" --- for continuing the search above the split point. It doesn't matter if this processor is the processor that first began processing this node or not, as there is no concept of ownership. It only matters that it is last, and therefore responsible for continuing the search.

Note this is a different approach than that taken by YBW. In YBW, the node that begins processing a node is the owner (or master), and is responsible for processing the results and continuing the search – *regardless of when it finishes*. In DTS, the last processor is responsible, *regardless of when it starts*. The YBW approach is simpler to implement in a recursive framework, but does present the problem of what to do with the owner should it run out of work while helper processors are still busy.

3.4.2 The DTS Implementation in Prophet

The first challenge that presented itself in implementing DTS in Prophet was that Prophet's search routines were recursive, and DTS does not lend itself easily to a recursive framework. Recall that, in DTS, there is no concept of node ownership. The last processor to search at a split point is responsible for backing the score and best line back up to the parent of the split point and continuing the search from there. This would be quite a challenge in a recursive framework, as the call stack for "helper nodes" are not set up to return back up to the parent node. The solution to this issue was to implement an iterative search.

Prophet has three search routines. The first , `Root()`, is designed specifically to process the root node. `Root()` passes control to `Search()`, which handles the "full width" portion of the search. (That is, until the `depth` counter reaches 0.) Once `depth` reaches 0, `QSearch()` is called. When not in check, `QSearch()` evaluates only capturing moves. This helps ensure that the leaf evaluations occur at quiet positions. For the sake of simplicity, a design decision was made to only split the tree within the realm of `Search()`. Splitting in the quiescence stage is probably not a good idea anyway, as the subtrees are too small in most cases to overcome the overhead of doing the split.

`Search()` was renamed `rSearch()` (for "recursive search"), and a new search routine was created called `iSearch()` (for "iterative search") that traverses the same tree and produces the same results as `rSearch()`. Reasoning about an iterative search proved much more difficult than reasoning about a recursive search, especially with search heuristics such as "null move", "principal variation search," and "late move reductions" that involve potential re-searches. To ensure `rSearch()` and `iSearch()` were truly equivalent, a test mode was added that, when enabled, calls both `rSearch()` and `iSearch()` from `Root()` (one after the other), and compares the resulting outputs and data structures.

To aid in the debugging process, Prophet was modified to dump the search tree to file in XML format. An XML viewer called Treebeard was written to display the XML search trees from both the recursive and iterative searches, to help pinpoint where the searches began to diverge.

Unfortunately, the iterative version of the search is about 5% slower than the recursive version. This is most likely due to cache contention and the extra array index required to access the search information in the iterative search.

With the iterative search complete it became time to create some data structures necessary to make the algorithm work. The relevant data structures in Prophet are:

DTSSearchStateInfo

```
typedef struct {

        // ready to be examined?  This information is "incomplete" until ready==true.
        volatile bool ready;

        // the current ply in the search the busy processor is working on.
        int ply;

        // the current depth remaining for the busy processor.
        int depth;

        // the thread that this information belongs to.
        int thread;
} DTSSearchStateInfo;
```

When a thread is idle, it sends the HELP signal to threads that are not idle. Threads within `iSearch()` check for the HELP signal at every node. When a busy thread notices the HELP signal, it responds by populating a `DTSSearchStateInfo`. The idle thread uses this information to determine which thread it would like to split with, and at what ply. Once a busy thread populates the `DTSSearchStateInfo` object, it continues searching normally.

Once the idle thread has chosen a split point, it signals the busy thread and tells it the ply it would like to split at. If the busy thread has not already completed that ply, and

if it does not have another split point below that ply, it will create the split point. Once the split point is created, it signals the idle thread so it can drop back into the idle loop from where it will attach to the split point and begin searching. If the busy thread has already completed the ply the idle thread wanted to split, it simply signals the idle thread to drop back into the idle loop, from where it will try again to find or create a split point.

DTSSplitNode

```
typedef struct {
        // the position at the split point.  Move generation should be done using
        // this position, and should be protected by locking.
        Position splitPointPosition;

        // the search stack.  the shared stack is used up to and including the split
        // point.  below that, each thread uses its own.
        SearchData sharedSearchStack[MAX_PLY];

        // The shared moved stack is used to store moves for each ply up to and
        // including the split point.
        Move sharedMoveStack[MOVE_STACK_SIZE];

        Undo sharedUndoStack[UNDO_STACK_SIZE];

        // ply is the "split point".  It should NOT be updated in the search.
        int ply;

        // the depth we want to search to from the split point.  It should NOT be
        // updated in the search.
        int depth;

        // the number of cpus currently working at the split point.
        volatile int cpus;

        // is there any more work to do at this split point?  If not, no other
        // threads should be allowed to join in.
        volatile bool finished;
        volatile bool stop;

        // a lock for serializing access to shared data.
        lock_t lock;
} DTSSplitNode;
```

A `DTSSplitNode` contains all the state information necessary for any thread to continue searching above the split point. This includes move lists, the position at the split point and information to "undo" back to previous positions when backing up the search, alpha-beta bounds, etc.

Note that a `DTSSplitNode` is a bit simpler than its YBW counterpart, the `YBWSplitNode`. In YBW, the helpful master concept makes it possible for threads to be attached to multiple split points simultaneously. In DTS, once a thread runs out of work at a split point, it completely separates from that split point (there is no concept of node ownership) and returns back to the idle loop. So, there is never a time when a single thread is simultaneously working on multiple split points. Thus, a thread does not need multiple stacks (one for each split point) – it needs just one, which has been placed in the `DTSSearchThread` structure, described below.

DTSSearchThread

```
typedef struct {

        DTSSplitNode *splitNode;

        Position pos;

        // when a busy processor supplies state information about its search, it puts
        // the information in the element corresponding to its thread ID in the idle
        // processor's selectionInfo array.
        DTSSearchStateInfo selectionInfo[MAX_SEARCH_THREADS];

        // what split points are this thread working on?  The pointer will be non-null for
        // each ply it is working on a split node in, and null for those it is not.
        DTSSplitNode *splitNodes[MAX_PLY];

        Move *move[MAX_PLY];

        // should this thread stop?
        volatile bool stop;

        // "ready" signals a busy processor is done setting up the split node.
        // (Though the"answer" might be splitNode=NULL, indicating the busy processor
        // doesn't need the help.)
        volatile bool ready;

        // splitThread is the thread ID of some other thread wanting to create a split
        // point somewhere in this thread's search.
        int splitThread;

        // splitPly is where some other thread wants to create a split point in this
        // thread's search.
        int splitPly;

        // the "help signal" is set when a thread is looking for work.
        bool help;

        // is this thread idle?
        bool idle;

        volatile bool okToStart;

        // only respond to the HELP signal when this counter reaches 0!
        int thrashingCounter;

        // how many splits has this thread done?
        volatile int splits;

        Move moveStack[MOVE_STACK_SIZE];
        SearchData searchStack[MAX_PLY];
        Undo undoStack[UNDO_STACK_SIZE];

        // the thread object
        pthread_t t;
} DTSSearchThread;
```

Each search thread has a `DTSSearchThread` object. This data is private to a single thread and contains all the state information for this thread. The `splitNodes[]` array records which plies in the thread's search are split points. The state information above a split point is likely not valid – the correct information will be in the

DTSSplitNode structure. However, should this thread be the last thread at a split point, the state information will be copied from the DTSSplitNode structure to the DTSSearchThread structure as the "Unsplit" operation is performed, enabling this thread to continue the search above the split point.

With the data structures in place it was time to add some auxiliary functions. One of these functions, DTSInit(), is called on program initialization. It is responsible for initializing threads 1 – numSearchThreads. numSearchThreads is initialized to the number of processors detected in the machine, but can be overridden with the command line parameter "threads=N" (N being an integer value from 1 to the number of processors in the machine).

On program initialization each thread (excluding thread 0, the main thread) is sent to the function DTSIdleLoop(), where it sits in a busy wait loop. The following pseudo code describes the operation of DTSIdleLoop() :

```
void DTSIdleLoop(int thread) {
        while (1) {
                if (all threads stop)
                        break;

                if (thread has a split node SN to work on)
                        iSearch(...);

                find active split node SN to join
                if (SN == NULL)
                        send HELP signal to busy processors

                if (thread==0 and all processors idle)
                        return;
        }
}
```

In each pass of the loop, the thread checks to see if it has a split node to work on (DTSSearchThread[thread].splitNode != NULL). If it does, it does some quick checks to ensure the split node is not finished, and if not it enters the iterative search (iSearch) to help search at the split point.

If the thread does not already have a split point to work on, it checks to see if there are any active split points it could help with. If so, DTSSearchThread[thread].splitNode is set and it will begin searching on that

split point in the next pass through the loop. If there are no active split points, the thread sets about creating a new split point. It does this by sending the HELP signal to all busy processors. The busy processors check for the HELP signal at every node, and once detected will share some information about its current state with the idle processor, and then continue searching. The idle processor examines the search state for each busy processor, analyzes them, and decides where it would like to create a split point. It then notifies the thread it selected that it would like to create a split point, and which ply it would like to split at.

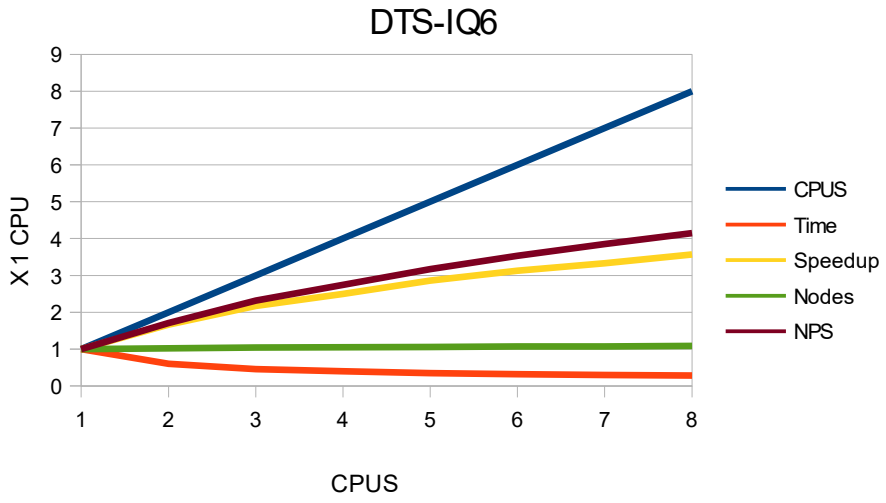
Once the busy processor receives the message that some idle processor wants to split, it sets up a split point and attaches itself to it. The idle processor is returned to the idle loop, where it will notice the active split point and attach to it.

Thread 0, the program's main thread of execution, operates a little differently than the other search threads. Once a move is entered by the user, thread 0 enters `Iterate()`, the iterative deepening driver. From `Iterate()` thread 0 enters `Root()`, the search routine responsible for processing the root node. (The root node is not searched in parallel.) As a move is processed from `Root()`, execution enters the iterative search routine, where it begins processing HELP signals received from idle threads.

Once any thread at a split point runs out of work to do, it checks to see if there are any other processors still working at the split point. If not, it is the last thread and continues the search by "returning back up" to the previous ply. Otherwise, it exits the iterative search routine. For threads 1-`numSearchThread`, this means exiting back to `DTSIdleLoop()`, where it will simply look for another split point to join. For thread 0, things are again a little different. It returns back to a function called `iSearchWrapper()`, a simple function that sits between `Root()` and `iSearch()`. Once back in `iSearchWrapper()`, thread 0 checks to see if any other thread is still busy. If so, it jumps into `DTSIdleLoop()` and acts just like the other threads, looking for work to do. Once all threads become idle thread 0 jumps back out of `DTSIdleLoop()` back to `iSearchWrapper()`, where it processes the final results of the search and finally returns back to `Root()` to process the next root move.

3.4.3 DTS Performance in Prophet

To measure performance, a test suite of 188 problems called "IQ6" was run to a fixed depth of 9 ply. The test suite was processed eight different times, each run using one more processor than the last (1-8 processors).



CPUS	Time	Speedup	Nodes	NPS
1	1	1	1	1
2	0.6	1.67	1.02	1.71
3	0.46	2.17	1.04	2.32
4	0.4	2.5	1.05	2.74
5	0.35	2.86	1.06	3.17
6	0.32	3.13	1.07	3.53
7	0.3	3.33	1.07	3.85
8	0.28	3.57	1.09	4.15

Figure 7 – DTS Performance in Prophet

Performance continues to improve with the addition of each processor, though it does appear the speedup curve is beginning to flatten with additional processors. With eight processors, Prophet processed the test suite in 28% of the time required for the single processor version to complete it, yielding a speedup of 3.57. Search overhead appears quite low, with only 9% more nodes visited using eight processors than with one processor.

The most obvious area for future improvement is NPS scaling (the rate at which nodes are processed relative to the single processor version). Since NPS scaling will bound speedup, it needs to be as high as possible. Further improvement may be obtained by more intelligent split point selection, as a poor choice of split point selection (at a CUT node) causes unnecessary work to be performed.

4 Conclusions

Based on the implementations in Prophet, the DTS algorithm is the most scalable of the four algorithms presented. Using eight processors, Prophet using DTS achieved a speedup of 3.57, processing the IQ6 test suite in 28% of the time it took a single processor version to process the same suite. The YBW implementation achieved a speedup of 3.03, processing IQ6 in 33% of the time it took a single processor. The Root Splitting algorithm failed to scale past six processors, achieving a maximum speedup of 2.27 and processing IQ6 in 44% of the time of a single processor. The Shared Transposition table performed the worst of the four. It achieved a speedup of 1.45 with two processors, processing IQ6 in 70% of the time of a single processor. It failed to gain any additional speedup with more than two processors.

NPS Scaling seems to be the bottleneck for scaling above eight processors for YBW and DTS. It would be interesting to compare the speedups of YBW and DTS with additional processors once NPS scaling is improved.

5 Future Work

5.1 How Far Can We Go?

5.1.1 Linear Scaling as an Upper Bound

Assuming an optimal search algorithm, speedup is bounded overhead as a linear function of the number processors. Superlinear speedups happen often in individual positions, but superlinear speedups *in general* are indicative of a suboptimal search algorithm (i.e. a poorly ordered tree). An informal proof is as follows [8]:

An N processor search can be simulated on a single processor using time slicing. Assuming no overhead, the time slicing search will take N times longer than the parallel search algorithm being simulated.

Suppose a superlinear algorithm P exists. As the algorithm is superlinear, $Tp < Ts / N$, where Ts is the time to solution using a single processor and N is the number of processors.

But then P on N processors can be simulated using the time slicing search, with time $Tt = Tp * N < Ts$. So now the time slicing algorithm running on a single processor runs faster than the single CPU algorithm. Hence, the single CPU algorithm is suboptimal.

5.2 Beyond SMP

5.2.1 Limitations of SMP Architectures

With the exception of very high end computers that use crossbar switches or some other high performance interconnection structure, most shared memory systems use a shared-bus architecture. Shared buses are a cost effective way to add additional processors and memories, but from the standpoint of throughput they do not scale well. As more and more processors are added to the system, competition for bus increases to the point it becomes saturated, processors become starved for data, and performance improvements are no longer possible.

5.2.2 NUMA Architectures

NUMA, or "Non-Uniform Memory Access" addresses the issue of bus contention by partitioning processors and memories into groups. Each group of processors share a memory module and have a dedicated bus. Data is stored with an affinity towards the processors that use it the most. This drastically reduces bus contention when processors from different groups are operating independently on unrelated data. Processors can access memory from different groups by traversing a global interconnect. This causes a performance penalty when accessing memory from other groups.

NUMA architectures offer more scalability than SMP processors, but require more expertise and effort to use properly. Without careful consideration of the effects of the non-uniformity of memory access, performance will suffer.

5.2.3 Distributed Computing

With distributed computing it becomes possible to assign parts of the computation to separate processing units, commonly referred to as nodes (not to be confused with "node" as previously defined in this paper). Systems in this category can be anything from a supercomputer with many nodes using high speed message passing, to clusters of workstations using ethernet.

Data is exchanged between nodes using messages. With message passing, the sharing of information becomes much more expensive than with shared memory, so great care must be taken to use a coarse granularity when partitioning the work.

5.3 Improving Split Point Selection in DTS

The algorithm for selecting split points in Prophet is extremely simple. For each busy processor, all nodes in the path from the "bottom most" split point (or the root in the absence of any split point) and the node the busy processor is currently working on are

considered. Nodes with fewer than four moves searched are excluded from consideration. Of the remaining nodes, the node closest to the root of the tree is given preference. This “four or more moves” constraint increases the likelihood that the node is an ALL node, if move ordering is good. By scoring nodes closer to the root more highly, the program prefers to create split ponts at shallow depths, which represent more work than nodes at deeper depths.

Dr. Robert Hyatt made use of several heuristics to classify node types in his program CRAY BLITZ [1]. These heuristics considered the alpha and beta bounds in relation to the initial search window, the ply a node is at, the number of moves searched at each node, and the node types of predecessor nodes. The idea is that by putting some effort into classifying node types (increasing synchronization overhead), the program is less likely to split at a CUT node, (decreasing search overhead). The net effect was a greater speedup.

Valavan Manohararajah [4] showed that by using neural networks a split point selection algorithm could be constructed that outperformed two other schemes, one very similar to the simple approach taken in Prophet, and the other very similar to the more elaborate approach taken in CRAY BLITZ. The neural network was capable of increasing split point selection accuracy (increasing synchronization overhead and decreasing search overhead), or decreasing split point selection accuracy (decreasing synchronization overhead and increasing search overhead). He found that the need for high split point selection accuracy began to decline as more processors were added. It should be noted, however, that these results are based on simulation, not on real game trees. Nevertheless, it does show the potential for using techniques from Artificial Intelligence to construct adaptive algorithms.

6 Bibliography

6.1 Publications

1. Hyatt, Robert. “The Dynamic Tree-Splitting Parallel Search Algorithm,” The Journal of the International Computer Chess Association, Vol 20, No. 1 (1998), 3-19.

2. Hyatt, Robert and Mann, Timothy, “A Lock-less Transposition Table Implementation for Parallel Search Chess Engines,” Journal of the International Computer Games Association, Vol. 25, No. 2 (2002), 63-72.

3. Knuth, Donald and Moore, Ronald, "An Analysis of Alpha-Beta Pruning,” Artificial Intelligence, Vol. 6, No. 4 (1975), 293-326.

4. Manohararajah, Valavan, "Parallel Alpha-Beta Search on Shared Memory Multiprocessors," Masters Thesis, University of Toronto (2001).

6.2 Books

5. Luger, George F. Artificial Intelligence – Structures and Strategies for Complex Problem Solving (4th Edition). Harlow, Essex: Pearson Education Limited, 2002.

6.3 Software

6. Hyatt, Robert. 2008. "Crafty" <ftp://ftp.cis.uab.edu/pub/hyatt>. The spinlock code in Prophet came from Crafty, with permission from Dr. Hyatt.

7. Romstad, Tord. 2008. "Viper" <http://www.glaurungchess.com/viper/>. An instructional chess program using YBW. Released under the GNU Public License.

6.4 Websites

8. H.G. Muller. "Superlinear Speedups." Online Posting. 01 Nov 2008. Computer Chess Club: Programming and Technical Discussions Forum. 08 Nov. 2008 <http://64.68.157.89/forum/viewtopic.php?topic_view=threads&p=229432&t=24658 >. http://www.talkchess.com : discussion of superlinear speedups