PVSplit: Parallelizing a Minimax Chess Solver

Adam Kavka

11 May 2015

15-618

Summary

In this project I wrote a parallel implementation of the chess minimax search algorithm for multicore systems. I utilized the principle variation splitting form of the algorithm to reduce search overhead. The result was a 2.5x speedup for search on the Latedays cluster, and a chess program that won 15% more often than its serial counterpart. In addition, the sources inhibiting speedup were quantified.

Background

Computers play full-information strategy games with an algorithm called minimax search. In a minimax search tree, each node is a board state, its children are the results of all possible moves at that board state, and the root is the current board state. The goal is for the active player to pick a move that results in victory under the assumption that the opponent is also picking moves that result in his or her victory. Since tracing moves all the way to the end of the game is computationally intractable, a heuristic is used to estimate which leaves of the search tree are the most favorable board states. Ultimately, we output the best move, and if we used a heuristic, the score for that move.

The bulk of the work here is in the evaluation function on the leaves, and the number of leaves grows exponentially with a branching factor of approximately 32[1]. We reduce the volume of work with a method called alpha-beta pruning. While searching the tree with this method, an alpha value is kept representing the best option so far, and a beta value represents the best value an opponent can guarantee for his or her self. If we find any moves rated better than beta, we need not search any more moves at that junction, because no smart opponent would ever let us reach that point (see figure). It is important to keep in mind that alpha-beta pruning is not an approximation; rather it is an optimization that is guaranteed to return the same result as regular minimax.

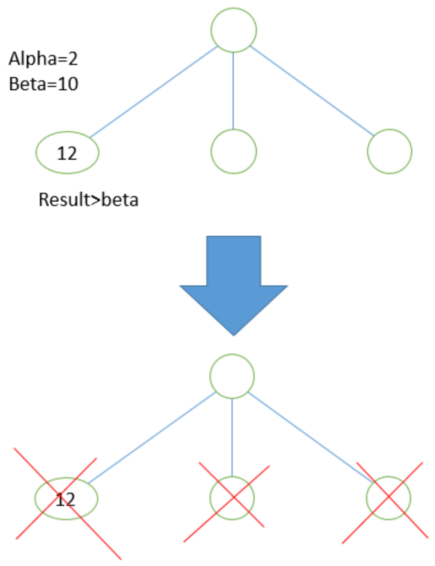


Figure 1 The first subtree returned a result of 12. Since opponent can make choices that guarantee a value of beta=10, he/she will never let us reach this point. Accordingly, we don't waste any more time searching.

Minimax itself is extremely parallelizable. The subtrees can all be searched independently without issue. The evaluation functions at the leaves are similarly independent. There isn't any synchronization in the middle of the search, nor is there any communication in the middle. There also is no inherently serial section. Furthermore, the overhead of allocating subtrees to processors is only done at the top level, so the fraction of work represented by

[1] *Parallelizing a Simple Chess Program*. Brian Greskamp.http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf

overhead is amortized to zero with a deeper tree. The only challenge is a little bit of workload imbalance if some branches are a simpler board state with a smaller branching factor or they reach checkmate.

Unfortunately though, parallelizing minimax with alpha-beta pruning brings many challenges to the table. The challenges include deterministic tie breaking, search overhead, and utilization; I'll discuss each of these in turn.

Challenge: Search Overhead

Recall that in alpha-beta pruning, we use the results of previous subtrees' searches to create a window of possible results, and anything outside those results need not be searched. Furthermore, this window can only decrease in size as we progress. Therefore any information we have on previous searches can only help us eliminate work. This is shown in figure 2.

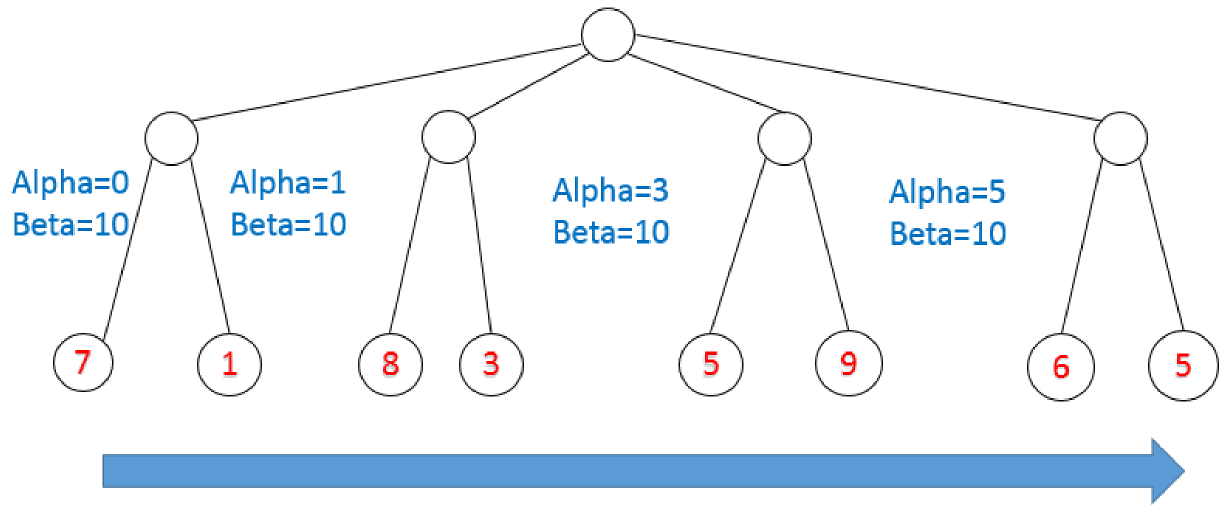


Figure 2 If we search the subtrees from left to right, our window of (beta-alpha) gets progressively smaller.

When performing searches in parallel however, simultaneous searches cannot use each other's results. This means the parallel version will search more nodes and call the evaluation function on more leaves to reach the same result. This effect is called search overhead.

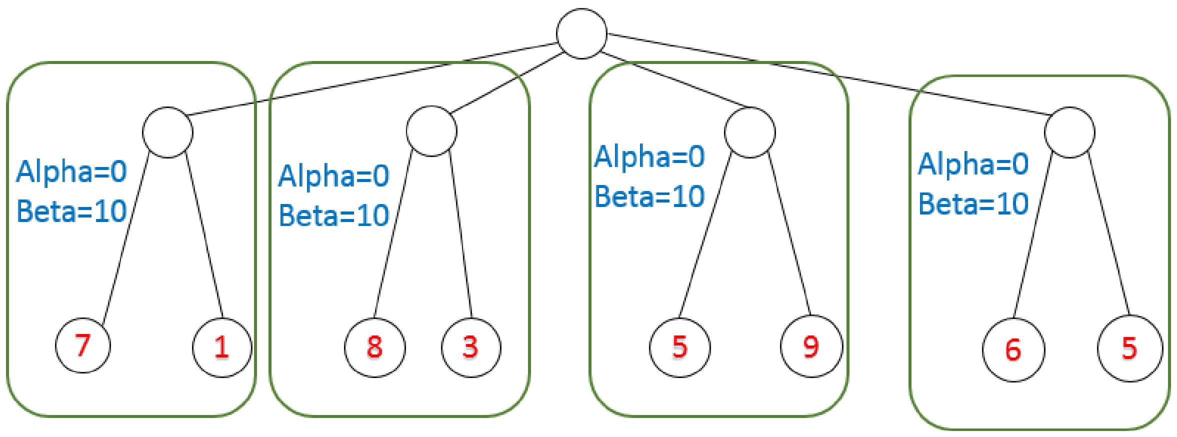


Figure 3 When the subtrees are searched concurrently, they cannot use previous results to narrow the (beta-alpha) window.

A proven way to address search overhead is Principle Variation Search[2]. With Principle Variation Search, the first subtree is searched in its entirety before any of the other subtrees are searched. While it may seem like searching the initial subtree is a huge serial section, we can actually search it in parallel and thus not be inhibited by Amdahl's Law. Once the first subtree is searched we have a baseline alpha and beta, and the other subtrees can be searched in parallel. This method becomes more effective when the first subtree returns a high score, leaving a small search window for the other subtrees.

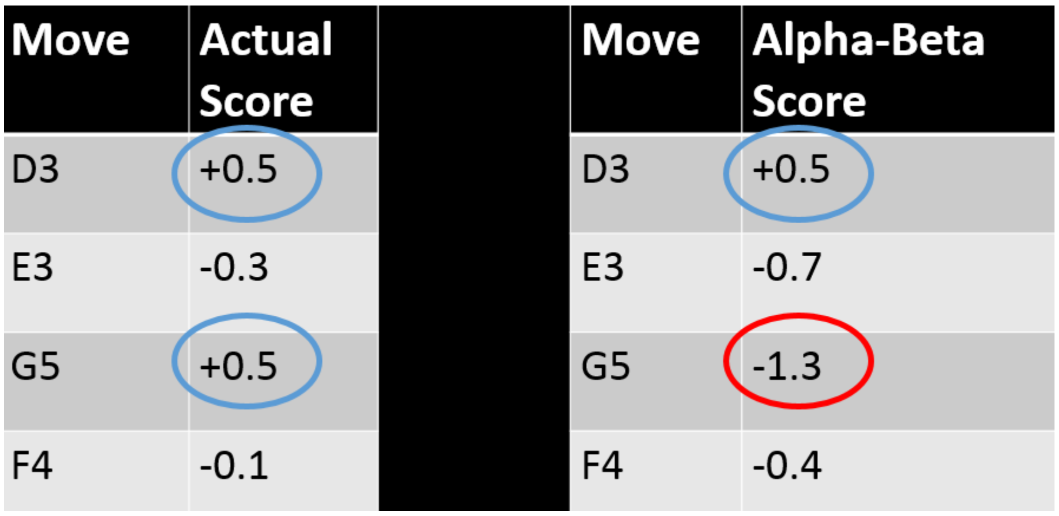
Challenge: Workload Imbalance

Workload imbalance can also inhibit speedup in alpha beta pruning. Once a search passes the beta threshold, that whole subtree stops doing work. If the processor searching that subtree isn't able to steal new work, it'll sit idly, and we won't hit our speedup potential.

Challenge: Non-Deterministic Tie Breaking

This is an interesting and difficult problem that I didn't see in the literature anywhere.

If our evaluation function returns integers, it is possible for two leaves to tie for the best evaluation score. Strategically it is not a big deal which one we choose. However, for debugging it is greatly desirable to have deterministic output; otherwise it is difficult to verify correctness. The first idea for breaking the tie is just choosing whatever comes first. However, in parallel we reach nodes is non-deterministic order. The next idea might be to have a second-order tie breaker, such as lexicographical order of the moves' algebraic names. But this idea has a problem: alpha-beta pruning doesn't guarantee that ties on a move score are seen as ties.



Move	Actual Score
D3	+0.5
E3	-0.3
G5	+0.5
F4	-0.1

Move	Alpha-Beta Score
D3	+0.5
E3	-0.7
G5	-1.3
F4	-0.4

Figure 4 D3 and G5 are tied for the best moves. However, alpha-beta pruning only needs to return the maximum. Once it sees D3, it doesn't have to correctly calculate any values unless they are greater than D3's score.

The figure shows an example. Recall that normally in minimax we get a score for each move, then we choose the maximum. However, alpha-beta pruning does not actually need to get a correct score for each move; once it determines a move won't be the maximum it stops calculating its value. This can include times when a move should be tied for the maximum; alpha-beta stops calculating early and never sees the tie. So a second-tier tiebreaker won't work.

- [2] Parallel Game-Tree Search. T.A. Marsland, senior member, IEEEE, and Fred Popovich. http://ieeexplore.ieee.org.proxy.library.cmu.edu/stamp/stamp.jsp?tp=&arnumber=4767683

There is an elegant solution to this. When we see a new maximum, call it M, we actually record it as M-1. Then any tied values will be above the recorded maximum and will be calculated correctly. At that point a second-order tiebreaker can be used.

Approach

I started with an open source chess program called Marcel's Simple Chess Program[3]. This program was written in C. It stores the board state as a single global array. It had no parallelism to start with but it did have a number of features to augment chess play. These are described in the appendix.

I wrote a parallel version of the search function. I used Cilk for all of the threading. More specifically, I recursively called the main PVSplit recursive function on the first subtree. Once that finished, I used a cilk_for loop to declare each of the remaining subtrees as an independent unit of work. These remaining subtrees were each assigned to only one processor, so there was no point in calling the PVSplit algorithm recursively for them. Instead a simple serial minimax search function was called.

```
PVSplitSearch(alpha, beta, depth):
        bestScore=-inf;
        moves=generateListOfMoves();

        makeMove(moves[0]); //temporarily make move so we can evaluate
        score=-PVSplitSearch(-beta, -alpha, depth-1);//We need the negative signs because our
opponent is picking antagonistic moves

        if (score>bestScore) bestScore=score;

        if (score>alpha) alpha=score;

        if(score > beta) return; //This is where we check for beta pruning

        unmakeMove(moves[0]);//We can unmake the move once we're done evaluating it.

        cilk_for(/*iterate through moves*/){

                makeMove(m); //temporarily make move so we can evaluate
                score=-serialSearch(-beta, -alpha, depth-1);//Deeper recursive calls will not use
Cilk; now that we're on a thread we stay on it

                if (score>bestScore) bestScore=score;

                LOCK(alpha);//Needs to be atomic because we're in cilk_for
                if (score>alpha) alpha=score;
                UNLOCK(alpha)

                if(score > beta) return; //This is where we check for beta pruning

                unmakeMove(m);//We can unmake the move once we're done evaluating it.

        }

}
```

Pseudocode for PVSplit search. Note that for the first move it recursive calls PVSplit, a parallel call. However in the cilk_for loop it calls serialSearch, the serial version of this function.

[3] Marcel's Simple Chess Program. http://marcelk.net/mscp/

The only shared variables in the algorithm are the alpha-beta values, and all updates to them were made atomic with p_thread mutexes. From what I can tell p_thread mutexes were preferable to Cilk reducers because subtrees in the middle of the cilk_for loop that haven't started yet can read the newest alpha and beta values immediately, whereas Cilk reducers are designed to resolve concurrency at the end of the cilk_for loop. A neat part of this concurrency control is that reading alpha and beta doesn't require any locking. The worst case scenario is reading out of date values, which hurts efficiency, but doesn't hurt correctness.

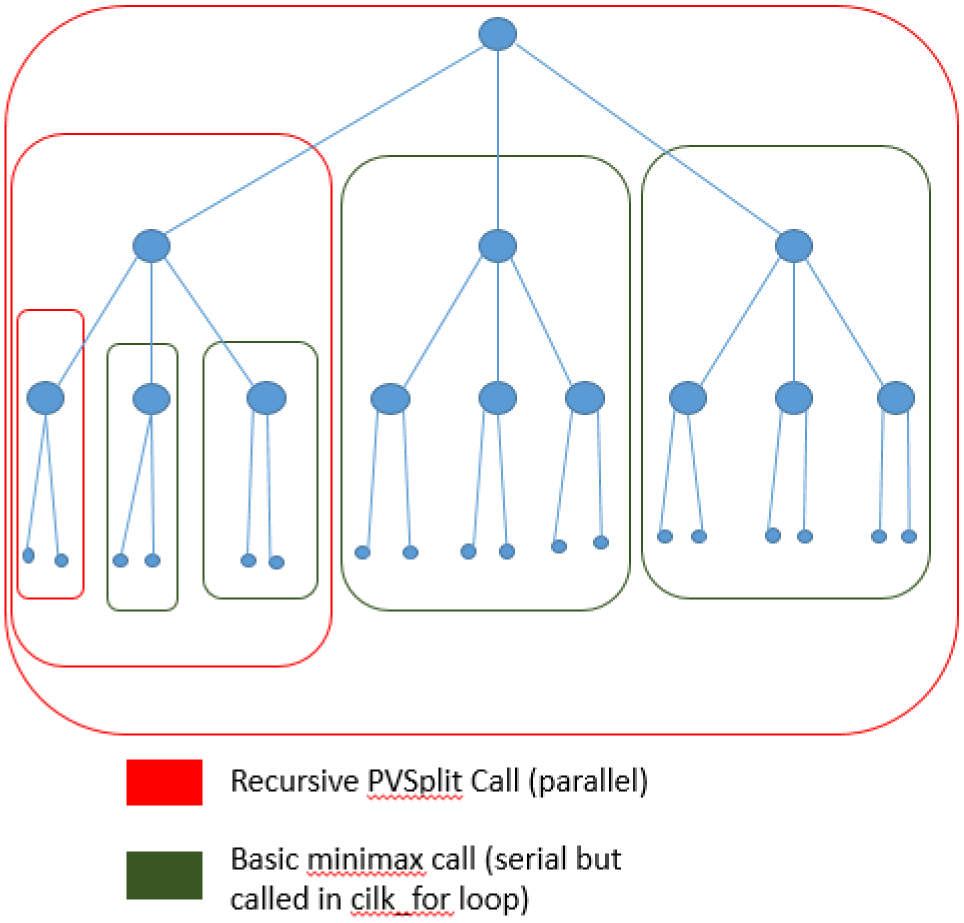

🟥 Recursive PVSplit Call (parallel)
🟩 Basic minimax call (serial but called in cilk_for loop)

Figure 5 An example PVSplit search. The PVSplit function is recursively called on the left-most subtree. Then there is a serial search call on each remaining subtree, but the serial calls are inside a cilk_for loop so there is still parallelism.

The most difficult part of the project was actually getting the parallel code to give correct results. The reason is because chess programs, Marcel's Simple Chess Program included, tend to store their game state (piece locations, turn count, castle rights, etc.) as global variables[4]. In theory this wouldn't be a problem because searching the minimax tree is a read operation; every thread should be able to read from the same variables. However, in implementation, evaluations of moves that are theoretically independent actually make a move that changes the global array holding the board state, evaluate the new state, then unmake said move. This temporarily alters the board data.

In order to solve this, every declaration of independent work needed to be accompanied by a deep copy of the board state. The thread could then alter the copy without incident. In addition, every function call in the program

- [4]Parallelizing a Simple Chess Program. Brian Greskamp. http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf now needed to have a pointer to this board state passed as an argument so it wouldn't modify the global data. This experience was a lesson that "read only" in theory is not always actually "read only" in the code.

I was able to get all of the game state copying and argument passing working and make a successful PVSplit implementation. That's to say, I have a parallel program implementing PVSplit correctly by giving an entire subtree to each node, and the parallelism helps achieve significant speedup. However, as the Results section will show, workload imbalance was an issue. I made a significant attempt at implementing work stealing on the search calls in the cilk_for loop. Cilk is the ideal tool for stealing work in a recursive function like this, but Cilk isn't much use when the data isn't independent. Thus I was not able to get all the game state copying for work stealing running correctly in the allotted time. To be clear: my program runs quickly in parallel, but dynamic work stealing would help its speedup.

Results

I did all of my test runs on the Latedays cluster. I did two different sets of tests: measuring speedup holding search depth constant and measuring search depth holding speed constant.

For all of the speedup runs, I used semi-random board states as input, then performed many searches in a row and recorded the wall time from the start to the end of each search. When I say "semi-random," I mean started with an opening chess board state then had each side make three random moves. From then on, there was no randomization, just the deterministic results of the search. I typically did 10 starting game states for each test run, with 10 searches per game state. I did this once in parallel and once with the serial program, and I repeated for each core count. The tests were all done with search depth of 6 or 7. Any shallower than this and the startup overhead hurt speedup; any deeper and the tests were intractably long. For reference, a 5-depth search takes about 2 seconds, and a 6 depth search takes about 12 seconds.

Every search was done twice, once with the serial algorithm and once with the parallel algorithm. It was crucial that I made sure all time comparisons were from the same board state; different board states had search times that differed by up to a factor of 50. Also note that all speedup calculations were the parallel code versus the original serial code, and not the parallel code versus other parallel code that happens to only be running with one core.

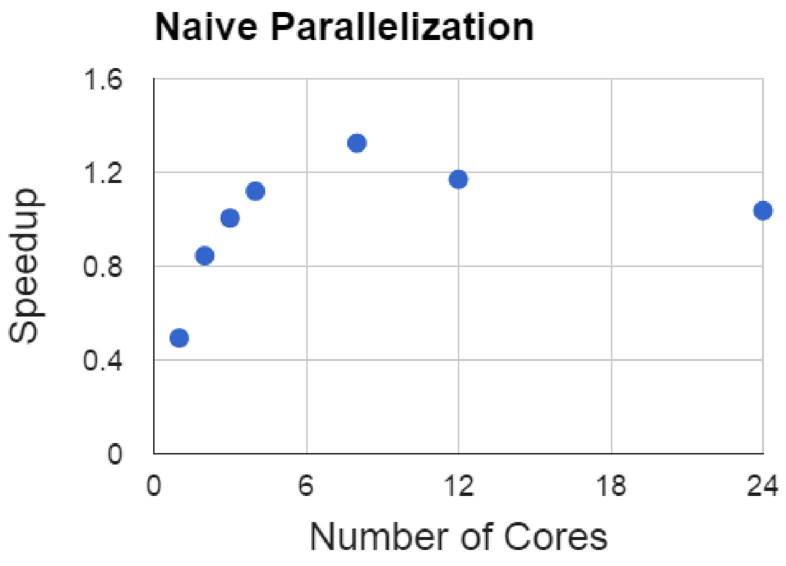


Figure 6 Average speedup for searches using naive parallelization with no PVSplitting and no work stealing. Data gathered with 100 searches, each of depth 6.

The first test I did was speedup vs. number of cores using just the naïve algorithm, no PVSplitting to reduce search overhead and no work stealing. It peaked at 8 cores with a speedup of 1.33. The primary reason for nonlinear speedup here was, unsurprisingly, the search overhead. To measure search overhead, I incremented a counter each time a node was visited, and took the ratio of the serial and parallel counters. With high core count, the parallel search was doing over three times as much work to reach the same result as the serial version.

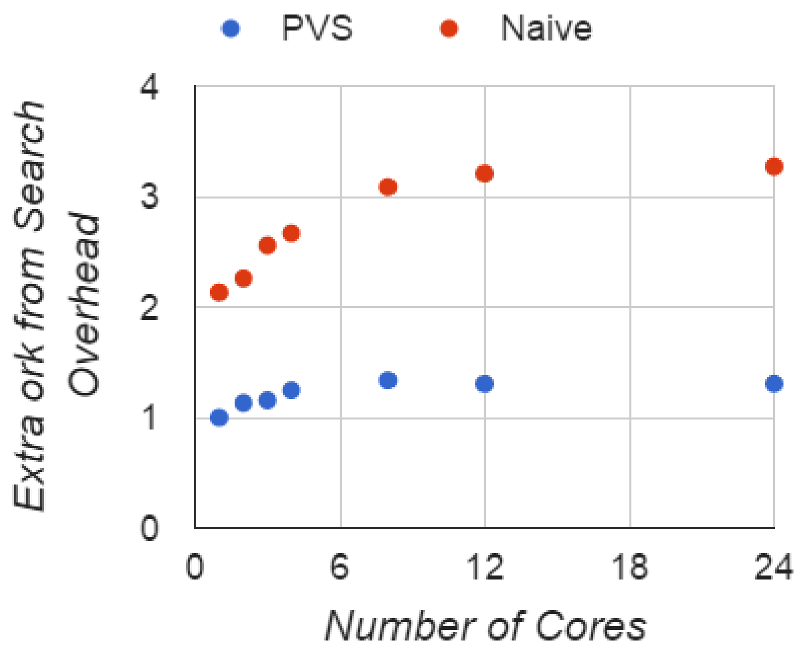


Figure 7 The ratio of the number of nodes visited in parallel program to the number of nodes visited by serial program. AKA search overhead.

I then did test runs with the PVSplitting in place. The point of PVSplitting is to reduce search overhead and it did this. As Fig. 7 shows, where we previously had a 3x search overhead we now only have 1.3x, and it's holding steady even at high core counts. We cannot completely eliminate speedup but this is a big jump, and it has a noticeable impact on speedup. The optimal speedup rose from 1.3x to 2.5x (still at 8x). In fact the speedup was as high as 3.3x on the unix cluster machines (I chose Latedays for my bulk data though because of its persistent queue).

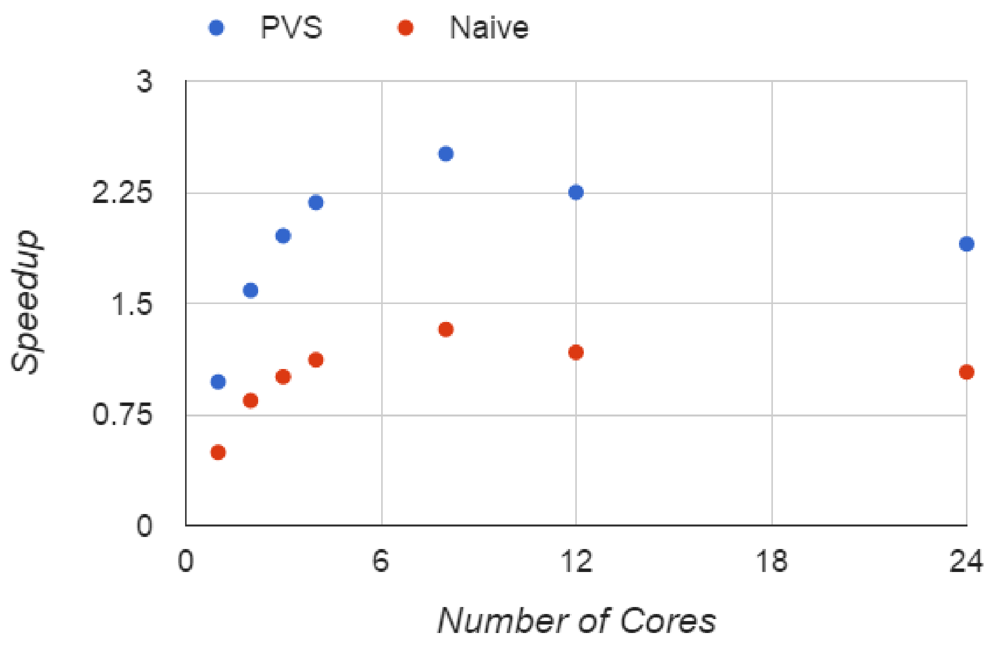


Figure 8 Speedup for the PVS algorithm and naive parallel algorithm relative to serial search. Based on 100 searches of depth 6.

The speedup is still non-linear though. The primary reason for this is workload imbalance. To measure workload imbalance, I recorded the wall time between the start and end of each iteration of the cilk_for loops, call each term t_i. I set T=(total wall time of the search) and n as the number of cores. Then I used this formula:

$$utilization = \frac{\sum t_i}{T * n}$$

Notice the numerator is the time the cores were busy and the denominator is the time they could have been busy.

Plotting the results of these utilization measurements shows that workload imbalance is a big problem at high core counts. Poor utilization reduces our speedup by a factor of 2 at 8 cores and a factor of 4 at 24 cores. This is where work stealing would help greatly.

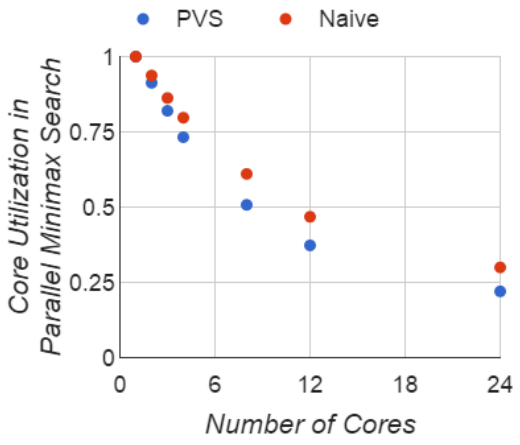


Figure 9 Fraction of time cores are active during 100 searches.

The last effect inhibiting speedup is that the parallel code is coded a little bit less efficiently than the serial code. I verified this by running the parallel code set to one core, then ran the serial code with the same input. The parallel code was always at least 90% as fast as the serial code, and on average around 94% as fast.

These three effects, the remaining search overhead, workload imbalance, and less efficient code, are sufficient to explain the speedup at low core counts. It is worth noting that these three effects do not bottleneck; they compound. For example: with 4 cores we see 0.76 utilization, 1.25x search overhead, and 94% code efficiency. From this we expect 4*0.76/1.25*0.94=2.28x speedup. The actual measured value was 2.18x, so these three factors are accurate predictors within a few percent. They don't predict as accurately at higher core counts. I assume the weaker speedup with 24 workers is because latedays isn't actually 24 cores; it's 12 cores with hyperthreading. This is just speculation though.

One thing that doesn't affect speedup is the overhead of the Cilk calls. The reason for this is the number of Cilk calls grows linearly with search depth (we only make Cilk calls on the left-most node at each tree depth), but the amount of work that we can parallelize grows exponentially. If the overhead hurt speedup, we should see a big boost in speedup when we increase the depth. I saw this effect at low depths, like going from 3 to 4, but once the search depth hit 6 increasing depth did not increase speedup, so Cilk overhead must have been a small fraction at that point.

Analyzing speedup is useful because it is easy to understand. However, in many ways speedup is not the ideal measure of performance. In competitive chess players actually have a finite amount of time per game, which we can approximate as a finite amount of time per move. We really want to see how much we can accomplish in that finite time. Number of nodes might seem like an appealing metric for this, but recall that out-of-date alpha and beta values mean that we can visit nodes without accomplishing anything. A better measurement of meaningful work is search depth.

To measure search depth in a finite amount of time, I simulated more searches from semi-random board states. For each search, I started with a 1 depth search, then incremented the depth and repeated until 1 second had gone by. Once the 1 second time limit was up I recorded the depth of the last search that was completed in its entirety. We need to round down to the last completed depth because alpha-beta pruning does not give meaningful results for non-integer depth. For each number of cores I did 100 searches in parallel with PVS and 100 searches in serial.

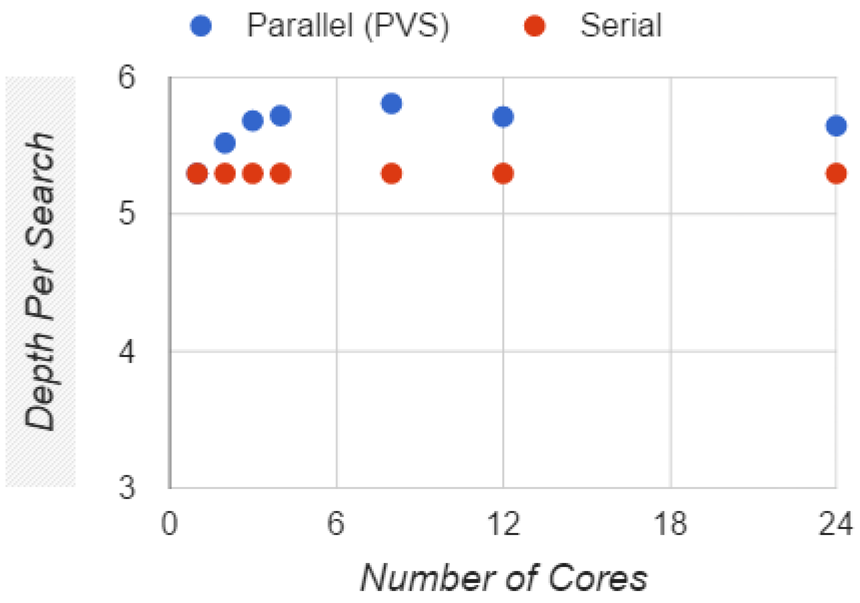


Figure 10 Average search depth for 100 1 second searches

Unsurprisingly search depth is highest when speedup is highest, around 8 cores. Here the parallel program has average search depth of 5.8 while the serial version has only 5.3. Equivalently, we can say that the parallel version looks a whole turn farther into the future about half the time. It's hard to intuit how big a difference this is, so I simulated some games to see if the parallel version actually played better.

Setting up the simulation has some subtleties. Obviously we can't just start each simulated game in the regular chess opening (if we did that every game would have the same result), so I gave each side a semi-random start by making the first white move and the first black move completely random. We also can't simply have the parallel version play against a serial version and take the win percentage. If we did this we couldn't control for the situation where one side got lucky and started in an advantageous position. My solution to this was to simulate each starting board state twice: once with the serial program playing both white and black, and once with the parallel program playing white and the serial program playing black. To analyze the results I looked at how often the parallel program did better than the serial version for the same board state. This method thus controlled for getting advantageous random opening moves, as well as for the advantage of being white.

Time Per Move	Games Played	% of games parallel finished better	Change in Win %
0.1 second	50	8%	4%
1 second	40	10%	5%
3 seconds	60	30%	15%
6 seconds	27	18%	9%

Table 1: Simulation of 177 games with semi-random starting boards. % of games finished better refers to games that a parallel program won and the serial version lost or drew OR the parallel version forced a draw and the serial version lost.

I ran 177 of these games for various turn lengths. For all turn lengths the parallel version outperformed the serial version. The table describes "% of games finished better;" this is how often the parallel code forced a draw from a game state the serial version could only tie lose, or the parallel code managed a win from a board state the serial version could only draw or lose. Accordingly the boost in win percentage is half of this because the jump from a draw to a win is only a half game. The parallel version had more of an advantage for longer turn lengths. This is unsurprising because at 0.1 seconds the Cilk overhead is pretty high; it's only a search depth of about 3.

Conclusion

My speedup of 2.5x is less than what has been accomplished in similar student projects[5], but it is enough to get a better result in somewhere from 10% to 30% of games. In the end, speedup was limited primarily by workload imbalance and to a lesser extent from search overhead and less efficient code. The biggest means of improvement would be to implement work stealing, which would have been facilitated by choosing starter code with fewer global variables (if that exists). Having said that, the choice of Cilk is effective in my implementation and would also be effective when adding work stealing. Ultimately, I have adapted a serial chess program into a parallel chess program that is measurably more competitive.

[5] See Robert Carlson's presentation from the parallel competition for another example

Appendix: Chess AI features

My starter code, Marcel's Simple Chess Program contained many features that are useful for intelligent chess play. I successfully adapted all of the features in this first list into my parallel program.

- An evaluation function, critical for the minimax algorithm in complex games.

- Iterative deepening, where we do a search of smaller depth first, and use its result to order the moves for when we do the full-depth search. The move ordering is useful because PVSplit uses a narrow alpha-beta window when the first subtree it searches has a high score.

- Quiscence search, which checks leaf nodes of the search tree to make sure they are not just postponing an inevitable bad event.[6]

- Aspiration window, where we artificially tighten the alpha-beta window in between depth interations in hopes that we've already found the best move. If we're right we eliminate needless work; if we're wrong we need to repeat work. The artificial alpha-beta window tightening was made atomic in my parallel version to prevent concurrency issues on the shared alpha-beta values.

These next features were in Marcel's Simple Chess Program, but I did not have the development time to add them to my parallel program. I thus removed them from the serial program. However, I will describe how one could have implemented them in parallel.

- A history table keeps track of the number of times a given move has been investigated. If it has been investigated a lot, it's probably a good move, so we should evaluate it first. Evaluating good moves first tightens the alpha-beta window in future searches. History tables are interesting because concurrency issues don't cause errors. The worst case scenario is you miss some increments so your optimization isn't as efficient as it could be, but it's likely that that is a price we're willing to pay to avoid locking.[7]

- A transposition table memorizes the result of previous searches. To implement it correctly in parallel, each potential board state needs its own lock. Corrupting the transposition table can lead to spurious results.

[6] Chess Programming Wiki. Qu;iescence Search. https://chessprogramming.wikispaces.com/Quiescence+Search

[7] Parallelizing a Simple Chess Program. Brian Greskamp. http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf

Bibiliography

Marcel's Simple Chess Program http://marcelk.net/mscp/

Parallelizing a Simple Chess Program. Brian Greskamp. http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf

Chess AI Parallelization. Alimpon Shah.
http://www.andrew.cmu.edu/user/arsinha/15418/418_final_report.pdf

Parallel Game-Tree Search. T.A. Marsland and Fred Popovich.
http://ieeexplore.ieee.org.proxy.library.cmu.edu/stamp/stamp.jsp?tp=&arnumber=4767683

Multithreaded Pruned Tree Search in Distributed Systems. Yaoqing Gao and T. A. Marsland.
http://webdocs.cs.ualberta.ca/~tony/RecentPapers/icci.pdf

CilkChess. http://supertech.csail.mit.edu/chess/