Techniques to Parallelize Chess

Matthew Guidry, Charles McClendon

Abstract

Computer chess has, over the years, served as a metric for showing the progress of computer science. It was a momentous occasion when IBM's Deep Blue was able to defeat world champion Garry Kasparov. Many hardware based chess computers have been developed; however in an age of multi-core it will be expected that the progress will continue to advance despite the end of the free lunch given by Moore's Law. We present methods involved in parallelizing a chess engine. We then present a discussion of our implementation of such an engine.

1. Introduction

The area of programming chess has spanned over 50 years and many different kinds of techniques have been utilized. In 1950, information theorist Claude Shannon published a groundbreaking paper entitled *"Programming a Computer for Playing* Chess" that proved pivotal for future work on chess. Chess is a game of "perfect information," which means that all players know the entire state of the game and all previous game moves. This, of course, is different from other games such as poker or bridge. Other games of perfect information include Tic-Tac-Toe and Go. Chess is also a finite game because of the 50-move, 3-move-repetition, and others that force a game to end as a draw.

2. Developing a Chess Engine

There are many factors to consider when building a computer chess engine, a program that performs the computations required to determine the best move. Firstly, one must decide how to represent the chess board. A naïve approach would be to simply create an 8x8 two-dimensional array. However, there are a few problems with this strategy. Firstly, this may utilize more memory than necessary. Secondly, and more importantly, there are other kinds of representations that can allow for very fast operations. The most common optimized representation of a chess board is to use the concept of "bit-boards." The entire state of a chess game could potentially be represented by 12 processor words (this is particularly effective on 64-bit processors). One processor word, or bit-board, could represent the location of white pawns, one for black pawns, etc. By using such a representation we may perform various operations, such as finding the location of all white material by using very fast bit-wise operations such as AND, OR, etc.

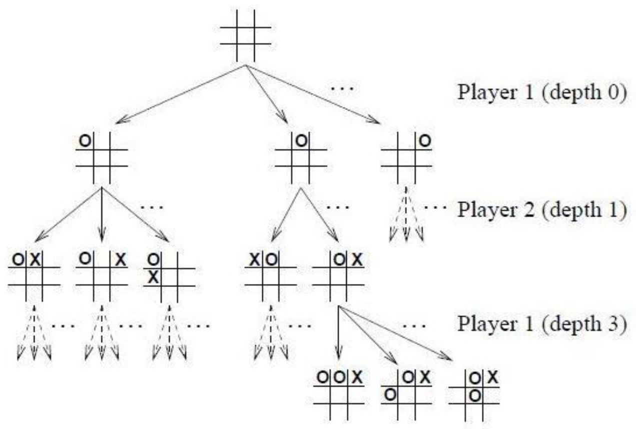
Another key, albeit secondary, factor that chess engine developers must consider is what "interface" they will support to allow others to play matches against their engine. Since there are literally hundreds of different chess engines, a common approach is to design a chess engine to use a common chess playing protocol that may be used by some proxy program that presents a graphical user interface while interacting with the chess engine via standard input and output mechanisms. Two common such protocols are Tim Mann's XBoard protocol and the Universal Chess Interface (UCI). Thus, most chess engines do not include a GUI but rather support one of these protocols and a user would use a program such as Winboard in order to play against a chess engine or to pit two engines against each other.

The primary component of a chess engine, of course, is its "thinking" algorithm. The basic idea utilized by a chess engine, or any game engine that plays a game of perfect information, is to simulate many moves ahead from the current position in order to determine the best next move. A typical chess engine will begin "thinking" once its turn begins and ends once it has determined the "best move." This thinking engenders a certain amount of non-trivial computation therefore some chess engines support "pondering," which is a technique whereby an engine will attempt to determine the next best move while its opponent is determining its best move; this strategy simply gives the engine more time to compute a best move or allows the game to progress more quickly in an untimed match.

3. Minimax

Chess engines perform this simulation by building a game tree of moves. Each node represents the game state or board configuration. At the root of the tree is the current board configuration. The immediate children of the root node are all of the next possible moves. For instance, at the beginning of a game, white's tree would have 20 immediate children from the root node (16 possible pawn moves and 4 possible knight moves). Of course, every child node has some number of other child nodes representing all possible moves from that position. Figure 1 describes an example game tree for Tic-Tac-Toe.

Fig. 1: A Game Tree for Tic-Tac-Toe



Of course, the method used to search the game tree and evaluate each board configuration is the most critical factor in developing a powerful chess engine. In chess, each move by a single player is referred to as a "ply." Most chess engines search a predetermined number of ply (or nodes of the game tree). A major issue with this approach is that clearly we cannot practically examine all possible moves from a given position as the search space becomes exponentially large. A typical chess engine will only search between 4 and 7 ply as the amount of time required to find a "best move" tends to become very large. The idea here is that given more ply, a chess engine can make a better decision so optimizing a chess engine usually involves increasing the amount of ply that can be searched for some time ***t***.

The most common move search algorithm for games of perfect information,

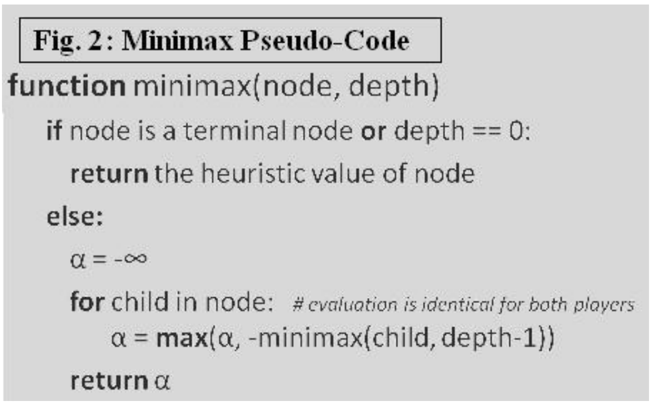


Fig. 2: Minimax Pseudo-Code

```
function minimax(node, depth)
    if node is a terminal node or depth == 0:
        return the heuristic value of node
    else:
        $\alpha = -\infty$
        for child in node:   # evaluation is identical for both players
            $\alpha$ = max($\alpha$, -minimax(child, depth-1))
        return $\alpha$
```

particularly two-player zero sum games such as chess, is the *minimax* algorithm. The minimax algorithm is based on the *minimax theorem*: “*For every two-person, zero-sum game with finite strategies, there exists a value V and a mixed strategy for each player, such that (a) Given player 2's strategy, the best payoff possible for player 1 is V, and (b) Given player 1's strategy, the best payoff possible for player 2 is -V.*” [1]

The main idea behind the algorithm is that one player attempts to minimize the maximum payoff of the other player and in a zero sum game this process will tend to produce positive outcomes for the first player [2]. The crux of the algorithm lies in an evaluation function; this function is how we determine if a potential move is better than others. Besides improving the number of ply an engine can search one can optimize an engine’s play by improving the evaluation function. A simple evaluation function may entail comparing the sums of the relative chess values of all the material for each player, white and black (for instance, 1 point is given for a pawn and 3 points are given for a knight or bishop, etc). To optimize this simple strategy one may then consider positional factors such as only awarding .5 points for an isolated pawn and so forth. Good chess engines include even more complex heuristics for their evaluation functions. The minimax algorithm works by recursively looking at each node in a game tree, to a certain depth, and returning the “value” of each node using the evaluation function. Figure 2 depicts the pseudo code of minimax.

4. Alpha Beta Pruning

One can build a very workable chess engine by simply using minimax and good evaluation function; however the canonical minimax algorithm does not fully exploit the engines potential. Consider that for a given maximum depth (or maximum number of ply) there may be many paths computed that will never actually be exercised. For instance, if you assume that your opponent is at least as good as you then you can eliminate paths that involve the opponent making moves that are too beneficial for you (i.e. giving up a queen for an isolated pawn). By making such eliminations, the search space can be greatly reduced and the chess engine may search more useful plies. This approach is referred to as alpha-beta pruning [3].

To use minimax with alpha beta pruning, we may simply extend the minimax algorithm to accept additional inputs *alpha* and *beta* representing the best score each player, current player and opponent respectively. The initial invocation of the algorithm would entail setting *alpha* = $-\infty$ and beta = ∞. Alpha beta pruning can, on average, reduce the branching factor *b* of a game tree toward a factor approximating O(sqrt(b)) [4].

5. Using Transposition Tables

Another optimization technique one may consider to use is what’s known as a *transposition table*. A transposition table is essentially a kind of cache whereby a chess engine can store the results of previous searches of particular board configuration (game tree nodes). This technique essentially memoizes the recursive search algorithm. This table is essentially implemented as a hash table with the game state as the key and the value being value of the board configuration as determined by search. Thus, we may modify our recursive minimax with alpha beta pruning algorithm by including a step to consult the transposition table. Of course, it would consume too much memory to store the results of every search in our table thus we must determine some method by which to initiate adding an entry to our table. Many chess engines omit this step and instead utilize a pre-populated transposition table and the game tree search algorithm simply consults it but the table itself remains unchanged.

There are disadvantages to using a transposition table. The primary issue is that since the transposition table does not store information about history, the engine may become more prone to repetitive moves therefore resulting in unnecessary draws because of the 3-move-repetition rule.

6. Iterative Deepening and the Aspiration Window

Good chess engines tend to further improve upon this model by utilizing two other techniques, namely iterative deepening and aspiration windowing. Recall that game tree search algorithms recursively search the game tree until a particular depth (or ply) is reached. Notice, that when utilizing alpha beta pruning, for a given search of depth *d* our search could be made more efficient if we had an estimate of nodes at depth *d-1* and below (we would have, in effect, more cutoffs from our alpha beta pruning). Thus, iterative deepening refers to this process where we begin by performing our search with a maximum depth of 1 and increment the maximum depth by one and search again and we continue this process until we search at our true maximum depth (i.e. 7 ply). Every time the search function is called, certain cost estimates would be set that would inform subsequent searches thereby improving the performance of those searches; the cost estimation steps are referred to as aspiration windows. This means that it becomes more efficient to iteratively deepen our search, particularly since the search space becomes much larger at every depth.

7. A Parallelized Version of Chess

Parallelizing chess presents many interesting challenges. Clearly, a great deal of work has been done in terms of optimizing sequential chess engines. Parallelization techniques are not applicable to some of these optimizations, such as transposition tables, board, representation, and so forth. However, despite these techniques the vast majority of compute intensive operations occur within the tree searching algorithm. A major issue is the standard game tree search algorithm presented, namely minimax with alpha beta pruning, is inherently sequential [5].

First, recall that the concept of iterative deepening is often employed to improve the results of a game tree search. Clearly, this iterative process may be parallelized in such a way that each processor calls the recursive minimax function for a different depth. However, this method would defeat the purpose of iterative deepening, which is to inform subsequent searches of better cost estimates (again, usually by reordering the nodes at each level); the problem here is that to parallelize iterative deepening without engendering this self-defeating property, we would essentially make the process sequential. Therefore, the best approach is to parallelize the game tree search itself.

8. Principle Variation and YBWC

One method to approach the problem of parallelizing the game tree search is to utilize a concept known as principal variation. In principal variation, we may first recursively examine the left most branch in order to establish an alpha bound for the remaining branches. After this alpha bound has been fully established work can then begin on the rest of the tree. Figure 3 depicts the sequential and parallel partitions made by a splitting on the principal variation of a shallow tree for two processors.

By using principal variation we may employ the Younger Brothers Wait Concept [6,9] in which we may "split" after each left-most node has been examined, whereby we

Fig. 3: Principal Variation

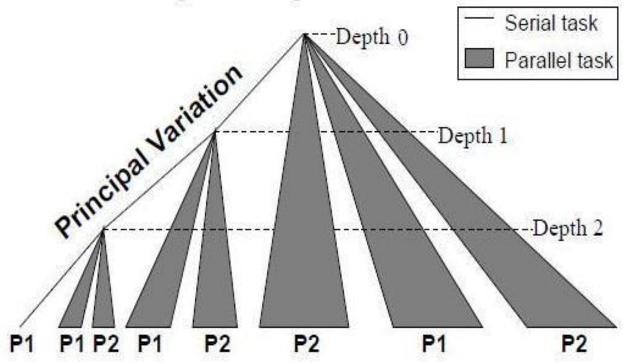
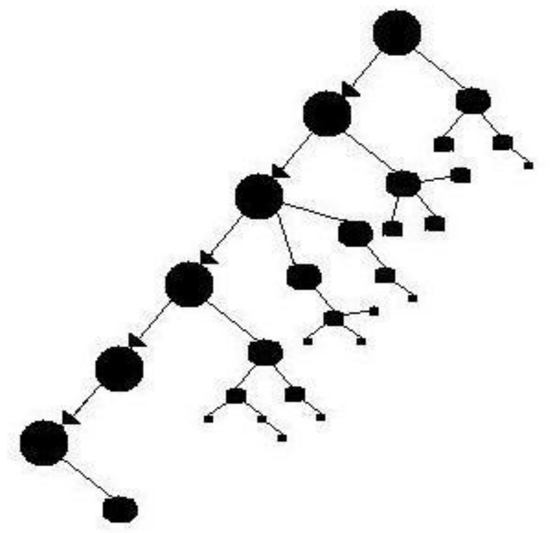


Fig. 4 : An Unbalanced Search Tree



marshal work to new processors. This represents a peer based model in which each of the processors work on a particular level of the initial search tree. Now, it becomes clear that because of the various move possibilities which become available from each board configuration, along with alpha beta pruning, there will not be an equal distribution of work across all processors; in fact some processors may be required to examine many more nodes than others. Figure 4 depicts an example game tree in which splitting after the first left child recursively will yield an unequal distribution of work among each processor.

In order to solve this problem, we may allow more splitting to occur. When there are idle processors, we will force running peers to split on nodes until all processors are working. This process is known as "work stealing." [7] Without this strategy, many parallel game searches would perform only marginally better than sequential searches. Work stealing is used heavily in parallel chess engines and worked particular well in the CilkChess engine from MIT, which was written in the "cilk" language that provides built-in primitives for work stealing [8].

9. Results of Parallel Implementation

For our implementation of a parallel chess engine, of course we did not intend write an entire chess engine from the ground up as that would entail many other factors that have nothing to do with the "thinking" aspect for which we intend to investigate parallelization. Therefore, we have evaluated a few known open source chess engines to determine which would be the best fit for our needs.

Firstly, we narrowed our search to only chess engines that were implemented either in C++ or ANSI C. Next, since we were interested only in the CPU bound aspects of the problem, we wanted to use an engine that made little or no use of memory based optimizations such as transposition tables and so forth. The two chess engines that we spent the most time with are Tim's Simple Chess Program (TSCP) [10] and Marcel's Simple Chess Program (MSCP) [11]. The TSCP chess engine does not utilize transposition tables, is heavily commented, and utilizes an average evaluation function. The MSCP engine does utilize a transposition table, however it was easily disabled. As well, MSCP uses a rather simple evaluation function. We used XBoard in order run our chess simulations. For the same number of ply (or maximum depth), TSCP handily defeats MSCP in every match we simulated. Therefore, we began our work by investigating how to parallelize TSCP and simply utilized MSCP as a baseline and reference for clarifying some of the concepts utilized in the search algorithm when TSCP was not clear.

We started our parallel design by attempting to locate all avenues of parallelization. This included the iterative deepening step, the actual minimax with alpha beta pruning function, and the evaluation function. Once we performed some timings, it was clear, by Amdahl's Law, that parallelizing the evaluation function would yield very little performance improvement as very little time was spent in that routine.

Therefore, as we researched it became clear that parallelizing the minimax with alpha beta pruning was the best way to improve the play of the chess engine for given amount of time. In order to implement the Younger Brothers Wait Concept, we designed a thread pool that would be started when the program began and every call to the "*think()*" function would simply reuse the existing pool; this was done to avoid the overhead of the thread creation. Another component of our design was to only perform the parallel search for a depth (or ply) greater than 3, since the smaller trees may be searched faster sequentially because the overhead of threading tends to dominate.

A major issue that we later realized with TSCP, as well as MSCP, is that they both make extensive use of global variables, after all these programs were not written with parallelization in mind. One possible avenue was to utilize a global lock that protected any access to global variables; however this would have effectively turned the parallel search into a sequential one.

Thus, we realized that in order to gain a speed up we would need to de-globalize the variables so that each thread would be dealing with its own private view of the game states. However, after many trials it became clear that in order to achieve this either with MSCP, TSCP, or a few others we later evaluated, we would be required to effectively re-implement an entire chess engine as the data-structures manipulated by the various functions were explicitly designed with a sequential chess engine assumed. We deemed that such an implementation would go beyond scope and timeline defined for the project.

Bibliography

1. “Minimax”
http://en.wikipedia.org/wiki/Minimax

2. Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.)

3. McCarthy, John (latex2html 27 November 2006). Human Level AI Is Harder Than It Seemed in 1955

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

5. E. Felton and S. Otto, "A highly parallel chess program," in Proc. Int. Conf. Fifth Generation Computer Systems, Tokyo, Japan, 1988.

6. Feldman, R., Mynsliwietz, P., and Manien, B. game tree on a massively parallel system in *Advances in Computer Chess* 7 (1994) University of Limburg pg. 203-215

7. “Parallel Chess Searching and Bitboards”
http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3267/ps/imm3267.ps

8. "Cilk: An Efficient Multithreaded Runtime System"
http://supertech.csail.mit.edu/papers/cilkjpdc96.pdf

9. “Parallelizing a Simple Chess Program”
http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf

10. Tim’s Simple Chess Program
http://www.tckerrigan.com/Chess/TSCP

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