Greg Loose

15-418 Final Project Report: Chess Engine

Project Summary

For my final project, I implemented a parallel chess engine using MPI. I was able to use it to solve chess puzzles and beat intermediate to advanced bots on Chess.com with only a few seconds of processing time between moves. Relative to a simple sequential implementation with no special optimizations, I was able to achieve over 50x speedup with parallelization across 128 cores on the PSC machines, though a more advanced sequential implementation using alpha-beta pruning decreased overall computation time, but reduced the relative speedup to about 10-20x.

Algorithm Background

The chess engine algorithm is based on recursive minimax evaluation of all legal moves for each "node" of what is commonly thought of as a "game tree". This terminology is somewhat misleading, as, at least in my implementation, no tree data structure is explicitly maintained in memory; rather the tree traversal is implemented as a series of recursively branching function calls which make updates to a single board data structure, essentially consisting of a 2D array representing the pieces on each square. For example, the first call to the "findBestMove" function makes one recursive call for each legal move for White, and each of these calls makes an additional call for each legal response for Black, and so on; thus the "tree" structure is implicit in the call stack. Each function call must update the board state with the latest move, then undo the move after all branches of the "subtree" rooted at this function call are finished.

The leaves of the game tree– that is, the bottom-level function calls, which do not make any further recursive calls– are evaluated based on a board evaluation function relying on simple chess heuristics. My implementation scores positions based on two factors: material and mobility. The former is a common way of evaluating a game of chess at a glance, which assigns a point value to each piece and adds up the points for each player, then subtracts Black's total from White's. If material is positive, then White is considered to have an advantage. Typically, pawns are assigned a value of 1, bishops and knights a value of 3, rooks a value of 5, and queens a value of 9, while the king has infinite (or arbitrarily high) value.

Mobility is a less formal metric without a widely accepted definition; I will use this term to refer to the number of legal moves available to each player. In my implementation, this component of the evaluation function is multiplied by 0.01 so as to break ties between positions with equal material. This is an extremely naive evaluation heuristic, but it is simple to implement and gives some structure to the engine's behavior in the early game, which is more about developing one's own pieces than taking pieces from the opponent, resulting in many moves with no change in material.

The "minimax" aspect of the algorithm refers to the method by which positions are evaluated at non-leaf nodes. For example, consider a game tree with a depth of 2. White's best move is not the one that maximizes his score score immediately, but rather the one that maximizes his score after Black does his best to minimize it (note that since chess is a zero-sum game, it is equivalent to talk about maximizing Black's score and minimizing White's, and vice versa). More formally, if s is the current board state, White and Black choose legal moves from the sets W and B, and $w(s)$ refers to the board state after applying move w, then:

$$\text{Value at depth 2}(s) = \max_{w \in W} \min_{b \in B} \text{score}(b(w(s)))$$

This value is a lower bound on White's raw score after one move from each player if he plays optimally; of course, he may score higher if his opponent does not play optimally.

With this setup, we can more formally define the findBestMove (FBM) function for a given initial board state and lookahead depth (with symmetric definitions for Black):

$$\text{FBM} (s, d, \text{WHITE}) = \text{argmax}_{w \in W} \text{ score} (\text{FBM} (w (s), d - 1, \text{BLACK}) \circ w (s))$$

$$\text{FBM} (s, 1, \text{WHITE}) = \text{argmax}_{w \in W} \text{ score} (w (s))$$

Figure 1 shows the general structure of the algorithm. At the top level, White tests every legal move, each of which recursively tests every legal move for Black from the updated board state, and scores are assigned from the bottom up, with Black taking the score of its lowest-scoring child while White takes the score of its highest-scoring child. In this example, after the call to findBestMove (s, 2, WHITE) terminates, White will play the move Nxe5 (that is, the white knight takes the pawn in the fifth row and column from the bottom left).

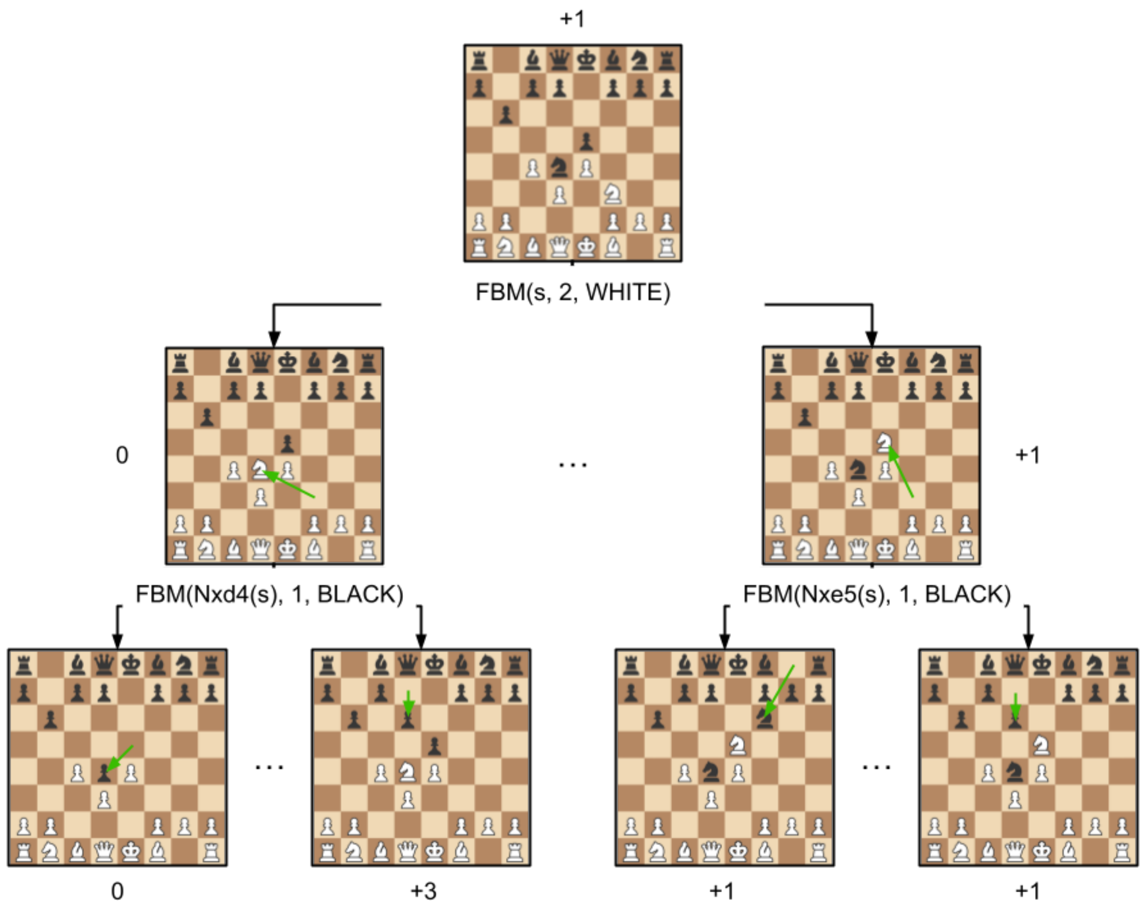


Figure 1: The recursive game tree structure. Arrows represent function calls and numbers represent material score for White. Not all valid moves are shown.

Approach to Parallelism

The high computational cost of a chess engine comes from the sheer quantity of move combinations that must be considered. At the start of a game, each player has 20 legal moves available to them, and this number rises and falls over the course of the game as some pieces are developed into more mobile positions while others are removed from the board. Assuming this number as a low estimate, just to evaluate each move with depth 2 requires the engine to examine 20 * 20 = 400 different board states. This number rises exponentially; at depth 8 (just four moves ahead for each player), we need to consider 25.6 billion different states. Higher depths allow for better strategic decisions, but the time cost becomes prohibitive very quickly, especially for timed games where the clock is a limited resource.

Fortunately, the algorithm as described above is embarrassingly parallel. For a given board state, each legal move can be evaluated independently. In essence, we are assigning to each core one or more subtrees of the game tree to evaluate on its own. This is implemented by evaluating different moves on different MPI processes based on the process ID. After each process has recursively calculated the score for each of the moves assigned to it and found the maximum, we can reduce over all processes to find the global maximum and the move associated with it. If, for example, we have 20 moves per turn and 20 cores, then each core's computation becomes a single traversal of a smaller game tree of depth one less than the main depth, resulting in 20x less work per core. Aside from the reduction at the end, there is no synchronization required.

This was my initial approach to parallelization, which typically earned me speedups equal to about 60% of the number of legal moves at the top level. Of course, the problem with a naive version of this method is that speedup is strictly limited by top-level move count; if we are only assigning subtrees rooted at top-level moves, and there are 20 first moves available then any cores beyond the first 20 will be doing no useful work. The solution to this is to distribute cores among move subtrees recursively until there are fewer cores than moves, then assign subtrees to each core as before– the difference being that it is now lower-level subtrees being assigned, which are exponentially more plentiful and therefore can be fairly divided among cores.

This assignment of move subtrees to cores is illustrated in Figure 2. The "recursive splitting" strategy is somewhat more complicated to implement, but is well supported by standard features of MPI. Specifically, this feature is implemented using the MPI_Comm_split function, which allows the processes in a communicator group (e.g. MPI_COMM_WORLD) to be divided among multiple subgroups. Each subgroup has its own process ID space, allowing for a perfect abstraction of the top-level behavior. Each recursive function call can simply substitute its own communicator group in place of MPI_COMM_WORLD, and create new groups to further subdivide all of the processes that reach that call. Upon implementing this assignment strategy, I was able to achieve about 50x speedup on 128 cores, which is well beyond the theoretical maximum for my initial approach, limited as it is to the number of legal moves (generally estimated to be about 35 on average).

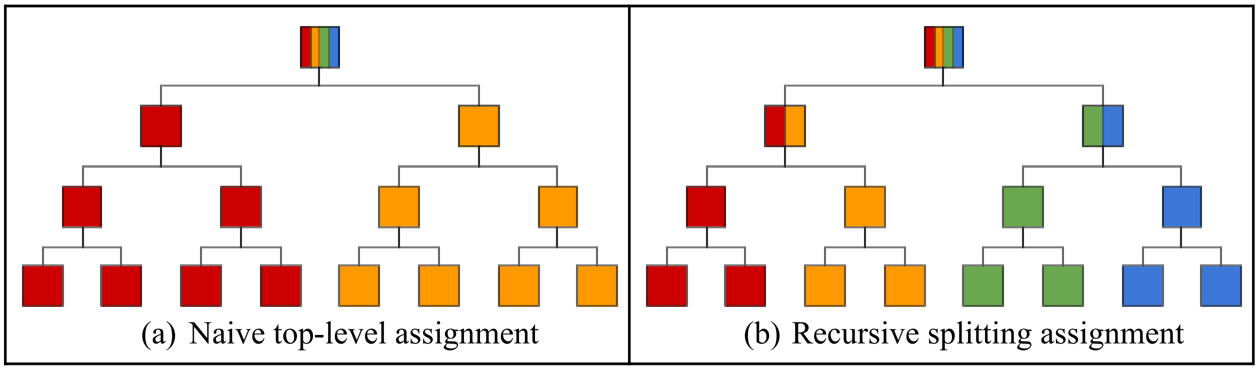

(a) Naive top-level assignment
(b) Recursive splitting assignment

Figure 2: Assignment strategies for two moves per turn divided between four cores. Nodes represent FBM function calls, and edges represent recursive calls after applying a legal move. Nodes are colored according to the cores that must visit that node.

Optimization with Alpha-Beta Pruning

After completing this parallel implementation with high speedup, I implemented a common optimization known as alpha-beta pruning to see how it would affect my sequential and parallel performance. The intuition behind alpha-beta pruning is that not every branch of the game tree has to be fully explored; for example, if, when evaluating a move for White, the first response considered for Black results in a checkmate, then it is safe to say that unless there are no other options, this move for White is not a good choice and we can stop evaluating this subtree early. To see how this is implemented in general, suppose that the root of the game tree is iterating over all legal moves for White in some sequential order. After evaluating the subtree for each move (by applying the move to the current board state and recursively calling findBestMove to get the best response for Black), the value of that move is stored in a “bestValue” variable if it is greater than the value already stored there. The best value is passed to subsequent findBestMove calls as a parameter “alpha”. While evaluating a subtree, if any response for Black results in a score that is less than alpha, then we know that Black can always play that move to ensure that White scores lower than he did with his previous best move (and, as chess is a zero-sum game, it is in Black’s best interest to do so). Therefore, White has no reason to play the move that started this subtree, and so no further responses are evaluated.

This algorithm is illustrated in Figure 3 below. Observe that, after evaluating the third leaf from the left, we know that the second white subtree will have value at least 5, and so will not be chosen by the black parent tree, which has already seen a subtree of value 3 and wants to minimize its value. Therefore, evaluation of the white subtree is aborted. Similarly, after evaluating the last white subtree, we know that the black parent tree will choose a value of 1 or less and so will not be chosen by the white root, so this subtree is also aborted.

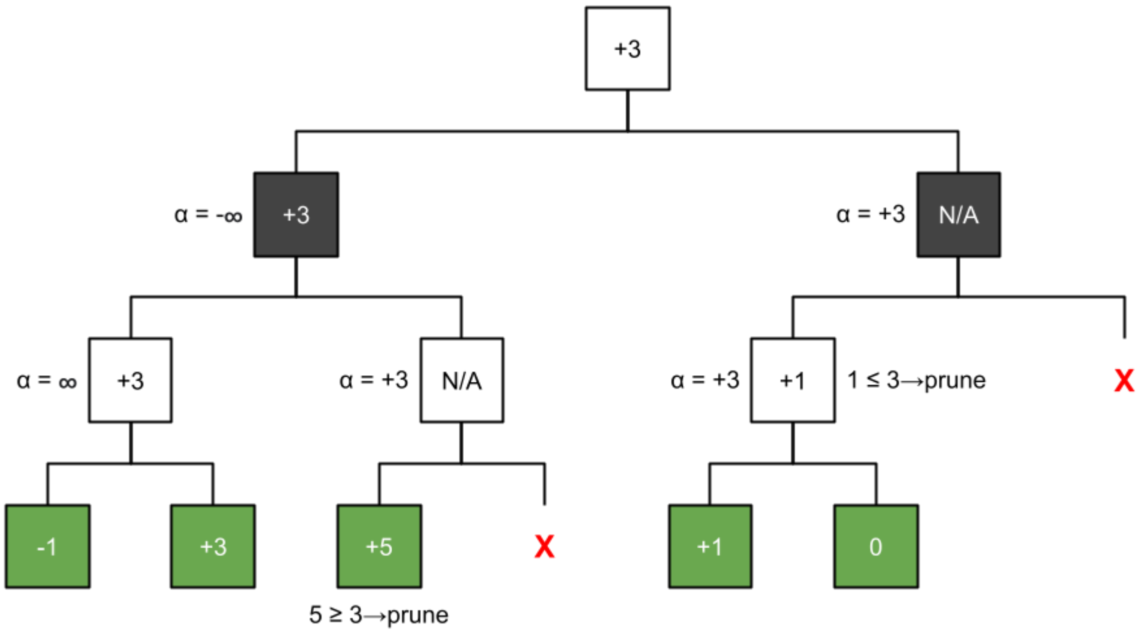


Figure 3: An example of the alpha-beta pruning algorithm. The color of a node signifies which player is choosing a move; White chooses max value subtree, Black chooses min value. Green nodes are leaves, whose value is given by a static board evaluation function.

Alpha-beta pruning has potential for huge time savings. In the best case, suppose that each player has 40 legal moves, and the first move evaluated for White is his best move. Then, every subsequent move considered will only need to check a fraction of the responses for Black, because one of them will result in a lower score for White and cause the subtree to be abandoned. If each subtree also considers Black's best move first, then for 39/40 top-level moves for White we only need to consider 1/40 responses for Black, meaning that we are only doing $1/40 + (39/40)(1/40) \approx 0.05$ times as much work, for about a 20x speedup. Of course, it is unrealistic to assume that the first move considered will always be the best, but we can improve our chances by pre-sorting the moves by some simple heuristics. In my implementation, I sort in descending order of raw score, evaluating each move at depth 1.

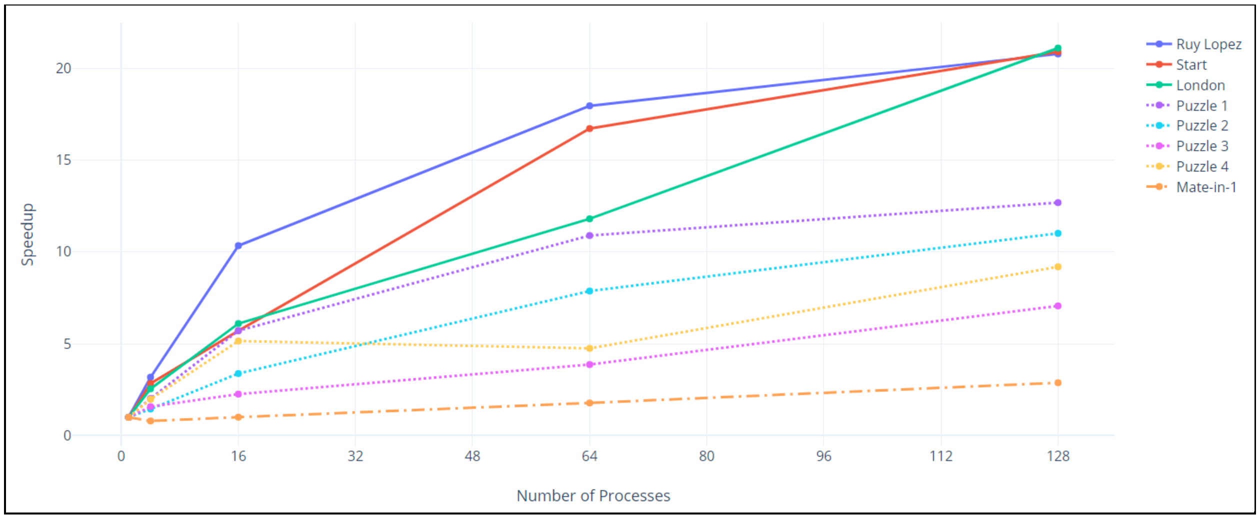
The power of alpha-beta pruning comes from using the knowledge of previously evaluated moves to avoid unnecessary work on later moves. If we evaluate every move at once, then by the time we get a value for alpha the extra work that it would have prevented has already been done. Furthermore, if only one process finds a move that scores above alpha, then it may break out of the subtree early while the others keep going, resulting in significant workload imbalance. Therefore, while alpha-beta pruning does improve performance across the board, there is a negative impact on speedup, as the parallel implementation benefits much less than the sequential version. I attempted to alleviate the workload imbalance issue by adding communication across processes after every move evaluation, so that all processes would break out at the same time; however, if there was any benefit to the change, it proved to be far outweighed by the added communication cost. I also experimented with an alternative parallelization strategy involving sequential processing of moves with alpha-beta pruning for the first few levels of the tree, followed by a switch to the parallel recursive splitting approach described above starting at a depth chosen by the user; however, this proved to be both overly complicated and strictly slower than my earlier approach. My final result ended up with about 10-20x speedup depending on the input, as opposed to the 50x speedup we were seeing on the basic implementation with no pruning.

Performance Evaluation and Results

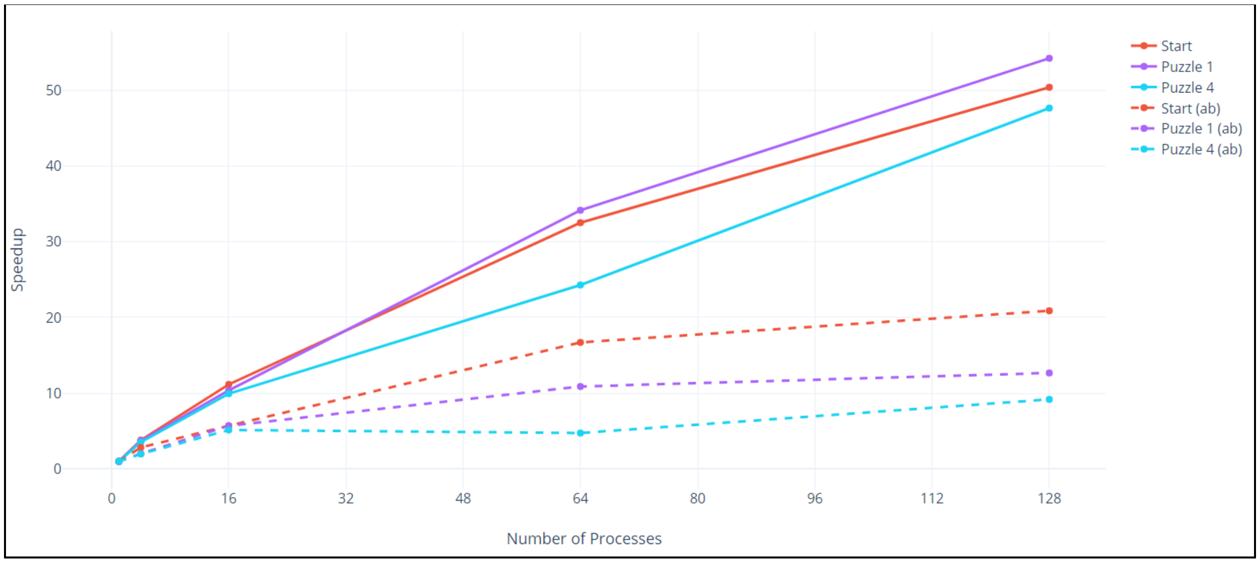
In terms of correctness, my engine has proven to do well in puzzles– where the player is given some initial board state and asked to find the best move from that position– as well as in games against intermediate computer opponents on Chess.com. To test its accuracy, I used my engine on ten consecutive puzzles, generated randomly by ChessPuzzle.net. Of these, it was able to solve seven; of the three puzzles failed, two were due to the fact that I have not implemented draw by repetition– a somewhat advanced game mechanic which I did not consider to be worth the extra complexity– and in the last one my engine's solution obtained the same material score as the intended solution, but was deemed slightly worse by the official puzzle engine due to some deeper positional knowledge. In "real games" against computer opponents, I tested my engine against bots of various levels, and found that it could easily defeat "intermediate" bots, but seemed to meet its match in the "advanced" category; I played two games against "Antonio" (a bot with a competitive rating of 1500), winning the first and losing the second.

I measured the performance of my code by clock time (measured with MPI_Wtime) and speedup (measured as the ratio of clock time for one process to n processes, where n ranges from 1 to 128). Specifically, clock time refers to the time it takes to execute the findBestMove function once. Most of my testing was done with depth 6 (that is, the game tree includes board states up to 6 moves ahead, or 3 moves from each player), as this was the value that I found to be in the sweet spot of reasonable computation time and playing well in puzzles and games.

During performance testing, I executed my program with a variety of initial board states, divided into two categories: puzzles and openings. Puzzles were obtained by generating a random puzzle on ChessPuzzle.net, while openings were selected somewhat arbitrarily from a list of common early game positions. In this category, I primarily tested on the default board state at the start of the game, the Ruy Lopez opening, and the London System (for no particular reason other than that I, in my fairly limited chess theoretic knowledge outside of this project, had heard of them). In dividing my inputs between these two categories, I wanted to examine how my engine performs both when there is an obvious "best move" and when there is less difference between the available options. The results are shown in Figure 4.



# Procs	R.L.	Start	London	Puz. 1	Puz. 2	Puz. 3	Puz. 4	M-in-1
1	57.50	11.51	38.67	21.55	12.24	30.81	2.75	16.96
128	2.85	0.54	1.91	1.81	1.12	4.33	0.31	5.97

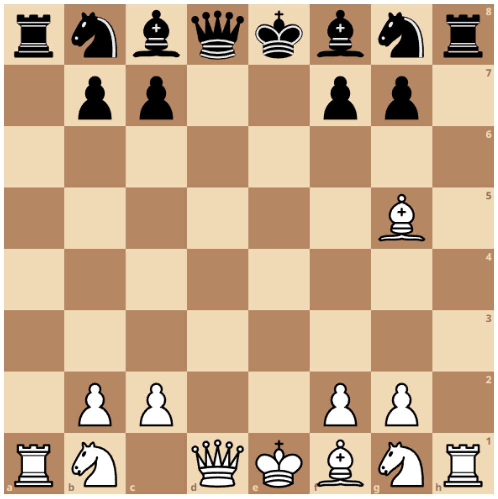


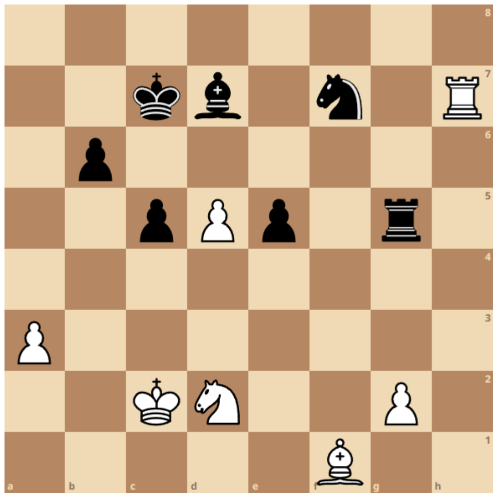
# Procs	Start	Puzzle 1	Puzzle 4
1	352.78	941.39	74.33
128	7.00	17.36	1.56

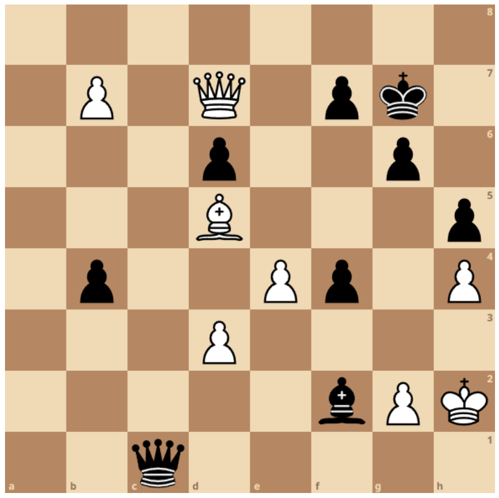
Figure 4: Speedup relative to 1 process and total computation time for multiple inputs. The first figure is for the final implementation, including alpha-beta pruning, while the second is for the unpruned version (with dashed lines showing pruned counterparts for comparison). Because the latter is not the focus of my project and takes a long time to run, only a few inputs were used.

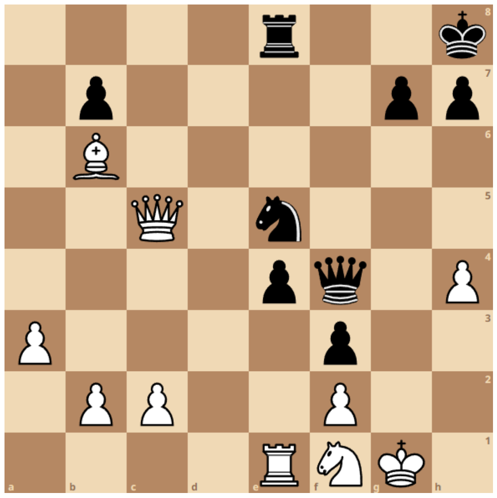
Observe the pattern in Figure 4: in general, we achieve higher speedup for openings than for puzzles. At 128 processes, all of the openings converge around 20x speedup, while the puzzles all have speedups of around 12 and below. The last puzzle, labeled “Mate-in-1”, is a particularly interesting case in which I attempted to engineer a puzzle with low speedup. In this puzzle, White can checkmate in one move by taking Black’s queen with his own. My idea was to give White a best move that would show up early in the sort by score with depth 1 that occurs before alpha-beta pruning, so that the sequential implementation could skip a vast amount of work that the parallel version could not. As can be seen in the figure, the experiment was successful: the speedup curve is almost flat, peaking at just 2.84x speedup at 128 processors. This extreme case provides a non-rigorous, but fairly compelling example of the weakness of my implementation in efficiently solving puzzles, relative to the sequential pruned implementation.

In order to further demonstrate how the inefficiency in my parallel solution is due to performing superfluous work that could be skipped via alpha-beta pruning, I inserted a counter to output the number of times that the main recursive function, “findBestMove”, was called in each process, and aggregated this count over all processes to get a rough estimate of how much work is being done in total. As a control, I first tested this on the version of my implementation without pruning. In the case of Puzzle 4, I observed 2,104,417 calls when running on one processor and 2,104,653 calls when running on 128 processes; very similar totals, as should be expected since this version just divides the work between processes with no special optimizations. However, when running on the same input but with alpha-beta pruning enabled, I observed 499,705 calls on 128 processes but only 138,059 calls on 1 process. In other words, on this input, the parallel solution is doing about 3.62x as much work as the sequential solution in terms of recursive function calls. This explains much of the speedup drop for alpha-beta pruning.


(a) Mate-in-1 Puzzle


(b) Puzzle 1


(c) Puzzle 2


(d) Puzzle 3

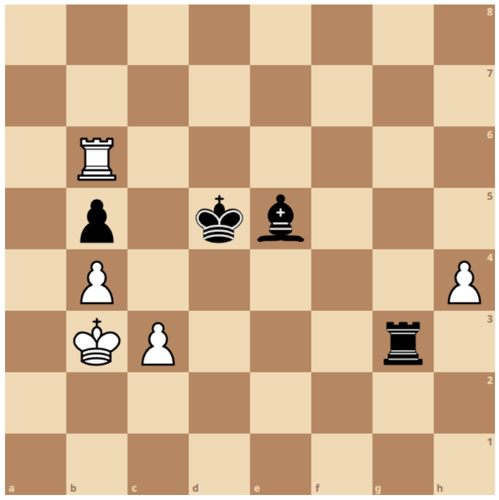

(e) Puzzle 4

Figure 5: All puzzles used in performance testing.

Another factor limiting speedup, unrelated to alpha-beta pruning, is communication between processes. This communication occurs in the form of an MPI_Allreduce between processes in the same findBestMove call. Because each process can evaluate a different subset of the available moves, they must communicate with each other to determine which move actually maximizes the player's score. This must happen for every recursive call in which more than one process is involved, and takes a significant amount of time. In order to get a sense of the impact of communication on total time, I inserted timing code around all MPI_Allreduce calls in findBestMove and ran the program on various inputs. Predictably, the time spent on communication was negligible for the single-process case. However, for Puzzle 1 with 128 processors, I found that of 1.77 seconds elapsed, 1.39 was spent on communication, or almost 80%. Upon repeating this process on other inputs, 80% seems to be a fairly common figure.

Conclusion

The chess engine I have implemented over the course of these past few weeks is very simplistic compared to much of what is currently available online, as was plainly apparent when I tested my own engine against the more advanced bots on Chess.com and watched them make better moves than I could, in a shorter amount of time. Aside from the use of alpha-beta pruning, what I have done is essentially the most basic possible realization of the broad class of minimax decision tree algorithms. However, I never expected or intended to create an engine competitive with the state of the art in just over a month's time, as the tricks used to get the maximum possible performance out of a chess engine are well beyond the scope of this course. I was ultimately impressed with how even such a simple algorithm, when combined with the parallel hardware and techniques that I learned about in this course, could solve real chess puzzles with high consistency and put up a respectable fight against mid-level computer opponents in just a few seconds of computation time– well within the reasonable amount of time for a human player to do the same analysis with much more sophisticated heuristics and game knowledge.

For me, the most interesting takeaway from this project is how much the optimizations that make advanced chess engines so fast can actually get in the way of parallel speedup. Given more time, I think it would have been interesting to see how this observation extends to other common optimizations, or if there were any particularly clever parallel strategies I could have used to make alpha-beta pruning more compatible with parallelization. In my current understanding, it seems unlikely that a pruned implementation can ever match the relative speedup obtainable without pruning, and none of my ideas on this subject have yielded any improvement, but in the final days of the project this question has proven to be an interesting, if presently unfruitful exercise.

References

Hercules, Andrew, *How Does A Chess Engine Work? A Guide To How Computers Play Chess*, Hercules Chess, https://herculeschess.com/how-does-a-chess-engine-work/

- This site gave me some basic background information on chess engines while I was coming up with the plan for this project.

Cornell University, AI Chess Algorithms, https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html

- This site was a useful reference for algorithmic details while working on the project.

Chess Programming Wiki, https://www.chessprogramming.org/Alpha-Beta

- This site gave me some more details about alpha-beta pruning.

ChessPuzzle.net

- I used this site to generate inputs during performance testing.

Chess.com

- I played against the computer opponents on this site to test my engine in a “real game”.