Parallelizing a chess search algorithm

Atanas Velchevski Vladimir Zdraveski

February 2023 Skopje, North Macedonia

Faculty of Computer Science and Engineering

1 Introduction

Parallelizing chess is a very interesting task and there are many challenges to overcome. Originally a lot of work went into optimizing the sequential methods, however, they are still not fast enough. This is why we need parallelization. As we know the biggest part of the processing costly operations happens in the game tree searching algorithm, so that is what requires parallelization the most. Practically, the best approach is to parallelize the game tree search itself.

2 Related Work

A good method of parallelization of the chess search algorithm is using a two-processor distributed system [1, 2]. The parallel search algorithm uses two processors which are connected via a distributed local network [1, 2]. Another interesting technique or approach to parallelizing the game tree search is to utilize a concept known as principal variation [3]. This concept uses recursion to check the left-most branch 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 [3].
Parallelizing the chess engine can be done very intuitively by using using a lazy SMP algorithm. The lazy SMP algorithm doubled the engine's search speed using 4 threads on a multicore processor[4].
Everyone remembers when the parallel chess supercomputer Deep Blue defeated the world champion, Garry Kasparov. Deep Blue used custom VLSI chips to parallelize the alpha-beta search algorithm, this is a classic example of AI in the 90s. The system derived its

playing strength mainly from brute force computing power [5].
A great paper from the 90s called massively parallel chess discusses how they managed to engineer a parallel chess program called Socrates, which runs on the NCSA's 512 processor CM-5, tied for third in the 1994 ACM International Computer Chess Championship [6]. Socrates used the Jamboree algorithm to search game trees in parallel and uses the Cilk 1.0 language and run-time system to express and schedule the computation [6].
The Jamboree algorithm is a parallelized version of the Scout search algorithm. The idea is that all of the testings of the children is done in parallel, and any tests that fail are sequentially valued [7]. The search algorithm that was used in Socrates also included some forward pruning heuristics that prune a deep search based on a shallow preliminary search. These forward pruning techniques at the time were extremely useful, and they even allowed programs running on single processors to beat some of the best human grandmasters at chess [7].
Modern chess engines can augment their evaluation by using massive tables containing billions of positions and their memorized solutions [8]. Memorizing positions is a common technique used in modern chess engine development. By saving a board state and a value or optimal "solution" move to disk, a properly-coded engine can ensure that the position is correctly handled in all future cases [8].
In making a chess engine using alpha-beta search there are many ways to reduce its execution time and thereby improve overall performance [9]. One way is by using move ordering heuristics, which are heuristics that reduce the search space by attempting to make a rough estimation about which moves are most promising before starting a search for a board position [9]. Another example is using a transposition table, which can reduce redundant computations [9]. A third example is multithreading, which is an attempt to utilize the several cores found in most modern computers to improve the amount of information found per unit of time [9].
Currently, the highest-scoring chess engine in the world is the engine Stockfish [10]. Stockfish is an open-source engine, that can use up to 1024 CPU threads in multiprocessor systems. The maximal size of its transposition table is 32 TB. Stockfish implements an advanced alpha-beta search and uses bitboards [11]. Compared to other engines, it is characterized by its great search depth. As of July 2022,

Stockfish 15 (4-threaded) achieves an Elo rating of 3540 [11].

3 Solution architecture

So how can we solve this parallelization issue? Well, we can create a parallel version of the chess engine and then compare the parallel to the sequential by letting chess games play out at different depths of the search tree. The decision-making algorithm I have chosen is the minimax algorithm with alpha-beta pruning.
Alpha-Beta pruning is not a new algorithm, but rather an optimization technique for the minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree.
But before we start making this parallel engine, we must find a technique that allows parallelization while still using alpha-beta pruning. Let's look at two good techniques.

1. Lazy SMP

The lazy SMP algorithm is currently the most used and most advanced parallelization technique. Multiple processes or threads search the same root position but are launched with different depths, or varying move ordering at the root node, to statistically improve the gains from the effect of non-determinism, otherwise depending on random timing fluctuations. Today, many chess programs use this easy-to-implement parallel search approach, which scales surprisingly well up to 8 cores and beyond, not only in nodes per second but also in playing strength.

2. Tree splitting

However, for our parallel engine, we are going to use a technique called tree splitting. While there are better algorithms, tree splitting is good and simple at the same time. The general process is to make sure that every node (except for the root) in the search tree is computed by only one processor. Keep in mind that we need to retain the effect of alpha-beta pruning while using this technique. We will do it by splitting the children nodes of the root into sections, while each section is computed by only one thread. Each of

these children then becomes the root of a sub-search tree that will be computed sequentially, while again making sure we respect the constraints of alpha-beta pruning. A thread will compute the sub-tree it was assigned to, and then return the result to the root. This result (evaluation) will be compared to the current best result (evaluation). If the new evaluation is better, it will be updated. This general process is basically how minimax works. It is very important to make sure that all of the threads work in mutual exclusion, as multiple threads accessing the root of the tree at the same time will give us the wrong results.

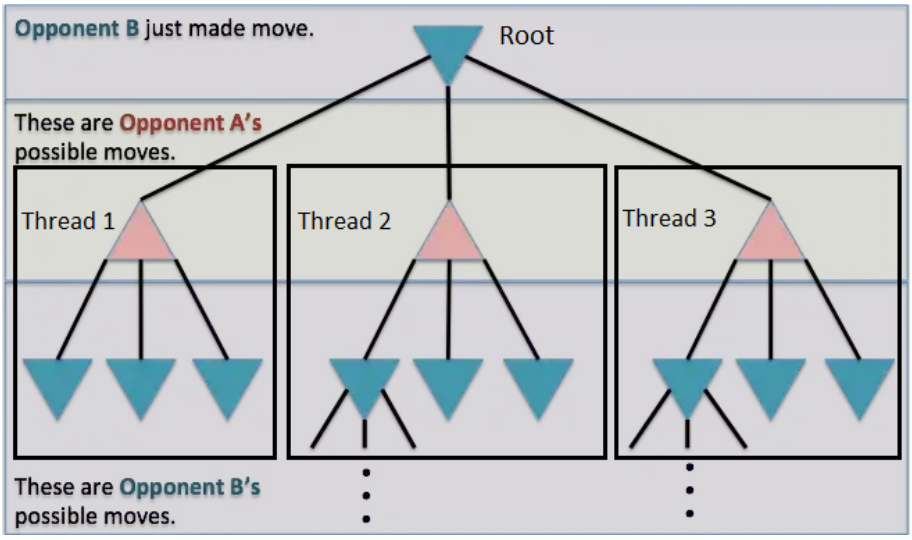

Figure 1: Search Tree Diagram

To start parallelizing at the root of the search tree, we need to parallelize the for loop that iterates over the children of the root and executes the minimax function on them. So we declare the variables that are unique to each thread and then we make sure that when a thread finishes his work, he will be assigned some iterations from another thread. This way the whole process is even faster because the threads will help each other out.
The next step is making the program so that inside the for loop after each minimax function execution returns, the thread enters a critical section to compare and then modify if needed, the best evaluation of the root node.

4 Results

In order to compare the parallel version of the engine to the sequential one, we will let the two engines play chess against each other, and see which one is faster. The results will be different on different processors. My machine has an Intel i3-8350k and 16 gigabytes of ram. We are going to run games at different depths and compare the results. Depth describes how many moves ahead the engine is looking in order to make its decision on what the best move is. First lets run 10 games at depth 2:

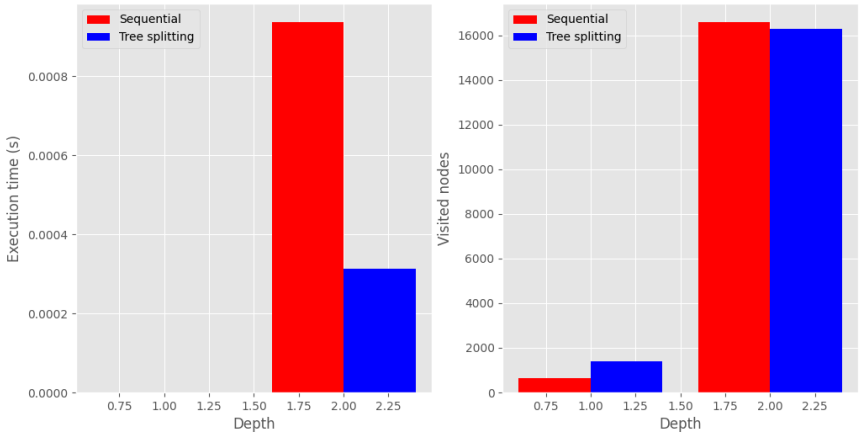

Figure 2: Result of First Battle

We can see on the graph that the parallel engine is 3 times faster than the sequential one. This result is great, even though in the real world a 0.0006 second improvement would make no difference. Also a chess engine that looks only 2 moves ahead is not very accurate.

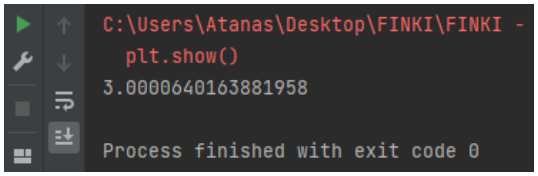

Figure 3: Speedup of First Test

So next let's run 10 games with a depth of 3:

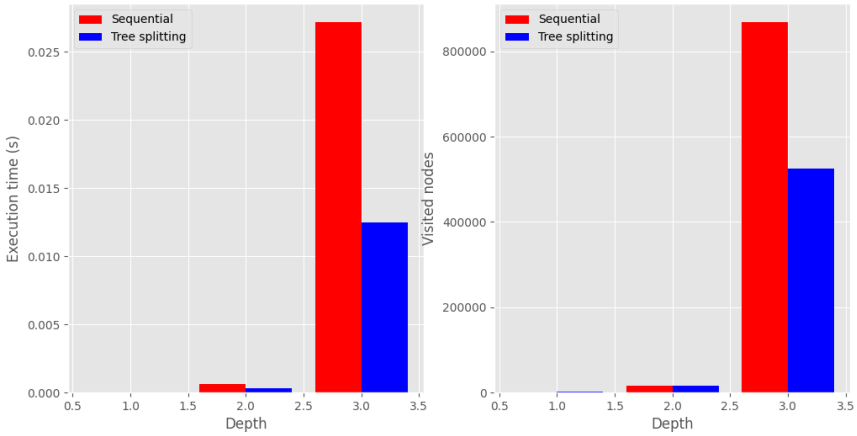

Figure 4: Result of Second Battle

This time the parallel engine is more than 2 times faster, 2.1748 to be exact, a great result. We can also see on the graph that the parallel engine at this depth has visited almost 40% less nodes.

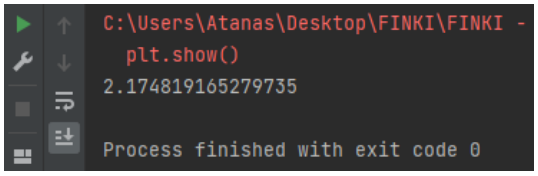

Figure 5: Speedup of Second Test

Next we need to try running at depth 4. Depth 4 and above will make the engines visit a huge number of nodes in the search tree. I doubt we will get better results. However, we have to try:

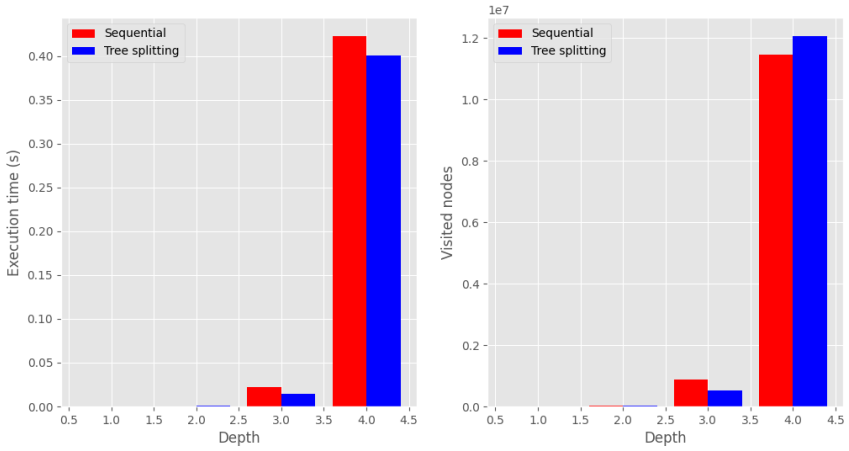

Figure 6: Result of Third Battle

We can see that at depth 4 the parallel engine visits more nodes than the sequential, but still remains faster. However this advantage has now dropped to only 1.0545 times. This is a poor result. The best result seems to be at a depth of 2 and 3.

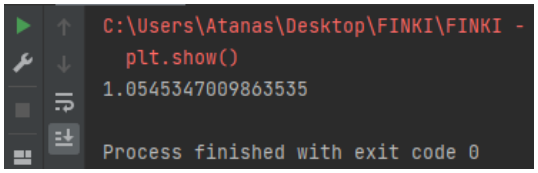

Figure 7: Speedup of Third Test

We will also try depth 5 but not with 10 games as that will take ages to finish on this machine.

Here are the depth 5 results:

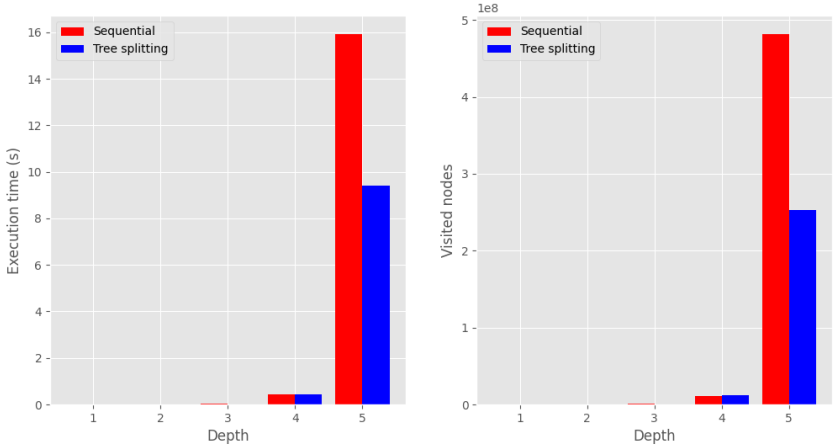

Figure 8: Result of Last Battle

We can see that at depth 5 the parallel engine manages to visit half the number of nodes that the sequential visits. It also manages to finish 1.6896 times faster which is not a bad result. At this depth the sequential visits almost $5 * 10^8$ nodes and the parallel visits half of that, both are huge numbers.

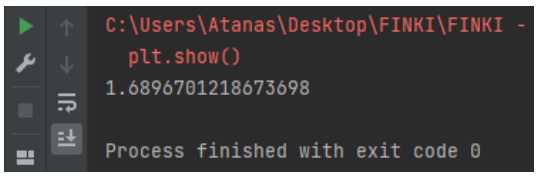

Figure 9: Speedup of Last Test

5 Conclusion

In this paper, we managed to successfully show just how imporant parallel processing is by paralellizing a chess engine using a technique called tree splitting. The decision-making algorithm that we used and respected is the minimax algorithm with alpha-beta pruning. We then tested the parallel engine by letting it play chess against the sequential one. In these tests, we got great results, the best being a 3 times faster execution time for the parallel engine. Our results confirm the results from already existing work i.e. it is possible to greatly improve chess engines by running them on parallel processors.

6 References

[1] - The method of the chess search algorithms parallelization using a two-processor distributed system - VV Vuckovic - SER. MATH. INFORM, 2007 - search.iczhiku.com,

[2] - The Realization of the Parallel Computer Chess Application - V Vuckovic - TELSIKS 2005-2005 uth International Conference . . . , 2005 - ieeexplore.ieee.org

[3] - Techniques to parallelize chess - CMM Guidry, C McClendon - ww2.cs.fsu.edu,

[4] - A complete chess engine parallelized using lazy SMP - EF Østensen - 2016 - duo.uio.no,

[5] - Parallelizing a Simple Chess Program - B Greskamp - ECE412, 2003 - cs.cmu.edu,

[6] - Massively parallel chess - C Joerg, BC Kuszmaul - Proceedings of the Third DIMACS . . . , 1994 - people.csail.mit.edu

[7] - The* Socrates massively parallel chess program - CF Joerg, BC Kuszmaul - Parallel Algorithms: Proceedings of The . . . , 1997 - books.google.com

[8] - The Effect of Endgame Tablebases on Modern Chess Engines - CD Peterson - 2018 - digitalcommons.calpoly.edu

[9] - Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine - H Brange - lup.lub.lu.se

[10] - Stockfish - https://stockfishchess.org/about/

[11] - Stockfish (chess) - Wikipedia - https://en.wikipedia.org/wiki/Stockfish_(chess)