MPCS 52060 Parallel Programming
Project 3 Report
Parallel Chess Engine Search

Shane McVeigh smcveigh

1 Project Overview

In chess, an **engine** is a program that, given a board input, is able to evaluate each possible legal move by the player, search down a move-tree given the possible moves in response to that first move, and so on, to a given **depth**. Then, the engine returns the best possible move given the position on the board and the results of its exhaustive search of moves. My goal for the final project was to implement a very basic chess engine and implement parallel algorithms to move down the search tree for a given position to suggest the best move.

Since the possibilities for each move grow exponentially, a chess engine should benefit a great deal from parallelized search. My parallelized search focused on the primary bottleneck of the chess engine: evaluating each possible move's "tree" of possible responses and countermoves. Therefore, after inputting a position and generating the list of possible legal moves, which is a relatively trivial sequential portion of the overall engine calculation, the search algorithms I implemented split the work between worker threads, each working on one move's search tree.

Citation: The backend engine logic of the rules, board position, piece definitions, etc. (non-parallel) is thanks in great part to the excellent overview provided by Zserge at this blog post (cited in `engine.go`). There are some key differences in my backend logic in order to more effectively structure the program for the parallel search but the primary implementation and structure are thanks to them.

2 Running Test Script and General Usage

The benchmarking for this project can be run by navigating to the `proj3/benchmark` directory and running either the python or bash script:
`benchmark.py`, or
`benchmark.sh`.

This will run the benchmarking that was used to generate the speedup graphs. It will run a sequential run of the program on depth 5 (5 moves ahead) on the Fischer game position. Then it will run the same for 2, 4, 6, 8, and 12 threads for the parallel threads version and the work-stealing version.

If you wish to use the program more interactively, usage instructions follow:
Navigate to the `proj3/main` directory and run the `main.go` file with the following usage:

```
go run main.go <mode> <numThreads> <start position> <depth>
```

With the following options for each of the 4 arguments:
Mode:

- "s": sequential
- "p": parallel threads
- "w": work-stealing

numThreads: The number of threads to use (integer), for the two parallel implementations. Use 1 for sequential runs.

Start Position:

- "b": beginning chessboard position (first move)
- "f": Fischer's Immortal Game position after 11. Bg5. This position was implemented due to the complexity of the position from a chess perspective and the fact that there are more move trees to filter down than in the beginning position, leading to a better use case for parallelism.

Depth: the number of half-moves (plies) to search down the tree. Depths higher than 6 experience very slow performance due to the exponential nature of possible chess moves by each depth.

For example, to run a depth-5 parallel work-stealing search with 8 threads on the Fischer position, use:

```
go run main.go w 8 f 5
```

3 Parallel Solution Overview

For the Parallel search algorithm, I elected to use a **map-reduce pattern**, by fanning out threads for each possible "first" move, then running down the entire search tree for that move to the given depth within each individual thread. There is a "fully parallel" implementation that spawns a thread for each possible move that I did not include in my benchmarking or speedup analysis, since the total number of possible moves in the average chess position is about 30.

Since the primary function of a chess engine is ultimately to suggest the "best" move based on its search, the "reduce" portion of the map-reduce pattern in my parallel search algorithm consists of checking the evaluated score of the thread's move against the current maximum. Since ultimately, a player does not need to know from the engine what the second best move is, if the thread's move is the current best, it is stored as such and replaces the former "best" move at time of search.

The map-reduce pattern was chosen over a BSP approach because there are not necessarily "supersteps" in a chess search algorithm that should use a condition variable to wait for other threads. Each thread is fully able to search down its own move tree because the moves are fundamentally independent of each other - a thread searching one move's tree cannot impact another's move tree. It doesn't make sense to halt threads at each depth level, because there is nothing fundamentally that they need to "wait" for other threads for in order to proceed with their local search.

Similarly, map-reduce was chosen over a pipelining approach for similar reasons. There is not necessarily a use case for an itermediate channel that receives some information from a thread in this project. The thread should run to completion on its move tree and report the best score it found to be evaluated by the overall engine.

The map-reduce pattern allows for more throughput in this case than either of the other two patterns, because each thread is allowed to run unimpeded down the move tree, which is the vast majority of the overall CPU time of the program. In either of the other patterns, some sort of pause between supersteps or channel syncs would impact the thread's ability to run to completion, leading to worse throughput. Additionally, the map-reduce pattern allows for threads to pick up other moves that have not yet been handled by a different thread, which improves the overall latency, especially compared with the sequential version, as well as the throughput of moves that can be handled in a certain timeframe.

4 Challenges

The primary challenge I faced was how to determine where to parallelize the search algorithm. I knew there should be some spawning of threads to work on different moves in the move tree, but struggled to identify whether it should be at the top only, or if threads should be able to handle "branches" of a move-tree at a time, and in the work-stealing version, steal branches of other thread's trees.

Ultimately, I settled on parallelizing the search algorithm at the top, i.e. for each first move, spawn a different thread. This allowed for the most effective way to handle communication back to the main engine about the "best" move identified by the thread, while keeping the parallelization simple enough to not invite additional race conditions. Since most chess positions have about 30 possible moves on average, and most parallel implementations we have worked on include only 2-12 threads, I determined it would be sufficient to split threads at the top in this manner.

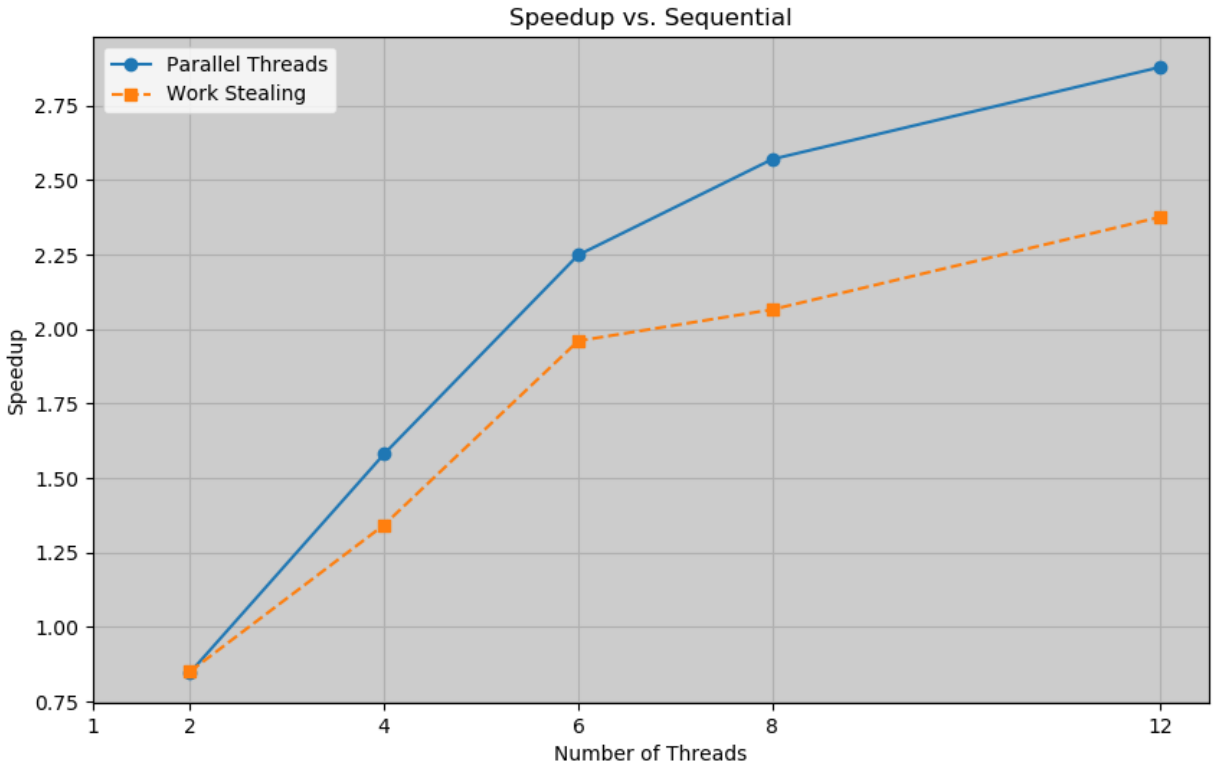
Another key challenge of implementing the search algorithm is that depth increases *vastly* increase the runtime of the search. The best engines today run to a depth of about 23, but I recommend limiting depth to about 6 in this implementation due to the hardware resources available compared to what the best engines have available to them. Even with a lower depth, there was some work I had to do in order to cut down the number of possible moves at each branch of the search tree.

Many chess engines use a variation of a search method called Alpha-Beta Pruning, which automatically cuts off a branch of the search tree if there is a currently better move that the thread has already found elsewhere in the tree (as it doesn't make sense to continue searching a poor move's tree if a better move has already been identified). Some of the performance measurements are evident in the link provided, but this method implemented effectively in a search algorithm can cut down the raw number of moves evaluated by a factor of about 90%, which provides a massive benefit to performance. I was able to implement a basic alpha-beta search method using a negamax function, which allowed my chess engine to ultimately be able to handle higher depths more effectively and provide better parallel performance.

More broadly, as an avid chess player, I hoped to gain a greater understanding of how chess engines are developed, as I had some basic knowledge and understanding that parallelism is a key part of an effective engine's implementation, but wanted to gain a more in-depth and direct understanding of how the search algorithms worked and where engines can best be parallelized. I would hazard a guess after working on my engine that the most effective engines today use some form of work stealing within other thread's branches rather than waiting for branches to finish their tree entirely, as I see that as a possible bottleneck in my engine.

5 Performance

5.1 Speedup Graph



5.2 Analysis

Performance drastically improved for thread counts above 2 in both parallel implementations. Performance improved at each step in thread count, due to the inherently parallel nature of the move search. The Parallel Threads algorithm experienced a maximum speedup of about 2.8 at 12 threads and Work Stealing experienced a speedup of about 2.4 at 12 threads, both excellent. I would expect speedup to continue to increase by thread count up until the point where threads = number of first moves, since there are no dependencies from one move to another.

Parallel threads consistently performed better than work-stealing, for reasons I have outlined below in Section 5.3. In short, this is due to the overhead associated with atomics used for the work-stealing algorithm, as well as the common average runtime for each thread's move tree at a given depth.

Both parallel algorithms experienced a speedup below 1 for 2 threads, indicating they were slower than the sequential algorithm. I suspect this is due to the overhead of parallelizing the algorithms and passing information about the next move to pick up between only two threads, but I am still somewhat surprised that they performed worse than the sequential search.

Overall, the performance improvement and speedup for both algorithms was effective, with an indication that performance would continue to improve with even higher thread counts. The primary limitations for speedup on the Parallel Threads algorithm is a synchronization overhead, due to the short lock on the update to best move and alpha using the alpha-beta search method. Since it does not share a limitation on runtime as prevalent as the use of atomic updates in the work-stealing algorithm and queue, the fact that it consistently experiences better performance is not surprising. Neither algorithm has a significant communication bottleneck, because the threads run almost completely independently of each other and do not communicate thread-to-thread, just with the score update back to the engine after the thread runs to completion.

5.3 Work Stealing

Work stealing did not improve performance against a fixed thread count with no work stealing. I think this is likely due to 2 key factors:

(1) the overhead of the work-stealing queues and algorithm with branch statements and atomics, and
(2) the fact that most threads will run in about the same amount of time.

Item (2) suggests that experiencing performance improvement in the work-stealing algorithm would be difficult since in most runs of the program the threads will be finishing their search trees at around the same time (with some variations due to alpha-beta pruning), which means that there is not much "idle" time for threads to pick up other threads work. By the time threads are completely idle, it means that all the other search trees have been picked up by other threads. Because of this fact and Item (1) with the use of atomics, the work-stealing algorithm is inherently a bit slower than the parallel threads algorithm, which means that it would have to gain performance improvement with threads picking up idle time, which in most cases does not exist in my implementation, unfortunately.

This could be different if each thread searched its move to a given depth, as then there would be a great deal of difference in runtime for each individual thread, leading to more idle time for work-stealing, but that would lead to a less useful result from a chess perspective. With some threads running to a different depth, some moves would inherently be "better" evaluated than others, leading to potential biases to one move that may be refuted given a higher or lower depth value.

5.4 Hotspots and Bottlenecks

In my sequential search algorithm, the primary hotspot is the move tree search for each individual move. The sequential algorithm processes each first move's move tree, one at a time, leading to a significant bottleneck as well. The vast majority of the runtime in the sequential program is spent on the move search. I was able to rectify this in the parallel implementations by spawning threads for each individual move, as discussed above.

There are some other bottlenecks, such as identifying the list of possible moves from the given board position, that I did not parallelize in my parallel algorithms. The reason for this is that it is a small fraction of the overall sequential runtime. There are only about 30 legal moves for the engine to find, and it iterates through each piece to identify those moves. The volume of these moves pale in comparison to the search tree, where the number of possible moves without alpha-beta pruning is approximately 30^n, where n is depth, given the assumption above of 30 legal moves per position.

As discussed in Section 4, I still believe there is a bottleneck existing in the sequential search that I did not implement in parallel due to the complexity, which is threads not being able to "steal" branches of another thread's move tree. I think that, implemented effectively, a response to this bottleneck allowing work-stealing within move branches would be very effective for performance. However, the main drawback is the difficulty of identifying and passing those move branches to other worker routines.

6 Citations

Zaitsev, S. (2020, March 21). Let's write a tiny chess engine in Go. ZSerge. https://zserge.com/posts/carnatus/

Alpha-beta. Alpha-Beta - Chessprogramming wiki. (n.d.). https://www.chessprogramming.org/Alpha-Beta

Depth. Depth - Chessprogramming wiki. (n.d.). https://www.chessprogramming.org/Depth