# Parallelizing an Othello minimax solver with alpha-beta pruning

Ashwin Srinivasan (ashwins), Miranda Lin (miranda1)

## Summary

In our project, we implemented multiple solvers for the game Othello and parallelized them by using task-level parallelism with OpenMP. We found that alphabeta tended to outperform even parallel minimax due to the decrease in work, and parallelizing alphabeta further improves that performance on some search depths.

## Background

### About the game

Reversi is a two-player board game invented in the late 19th century, in which opponents place white- and black-colored discs on an 8x8 grid. With each move, players try to “outflank” their opponent’s discs by bounding them in a straight line, thus flipping their color. The player with the most discs of their color on the board wins, and the game ends when no more valid moves can be made. We will be studying Othello, a modern variant of the game that has a predefined starting placement of discs.

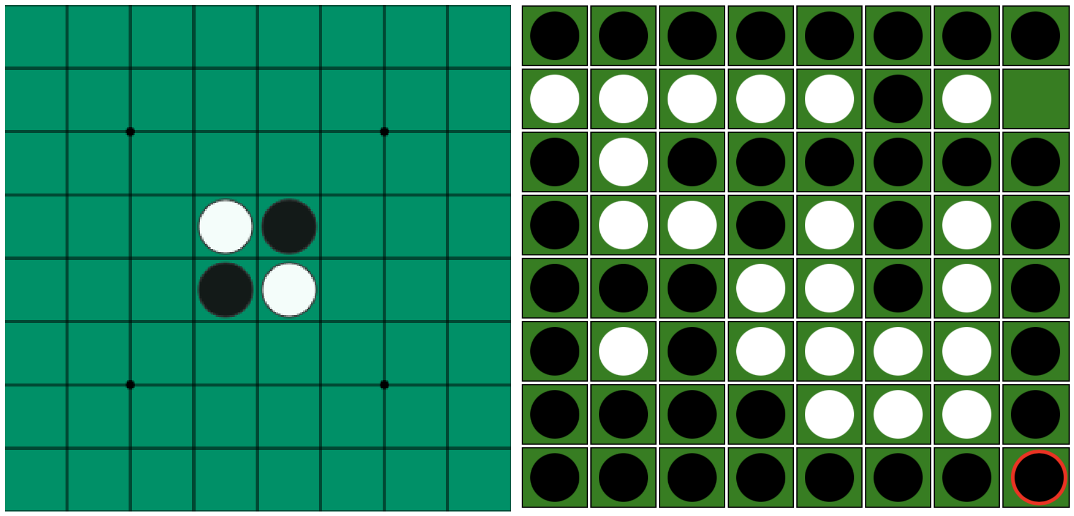

Figure 1: starting (left) and possible ending (right) Othello board states

## Game-solving algorithms

In order to write a computer player for Othello, we explored two game-solving algorithms: the minimax algorithm and the alphabeta algorithm. These algorithms both compute the optimal next move by evaluating the game tree, a data structure that represents all the possible future game states based on the various possible moves. Each level of the game tree represents a player's turn, and each child node represents a different move. However, in most cases we don't want to expand the entire game tree due to space and computation constraints. Hence we will restrict the depth of our search through the tree and use an estimator heuristic function at the lowest level we explore. We will see that there is a tradeoff for the value of the search depth, as a greater search depth results in more information thus possibly better moves, but will take longer to compute.

The minimax algorithm is a recursive decision procedure for choosing the next move in a two-player game, so it can be easily applied to Othello. Each possible resultant state of a move is assigned a value based on a heuristic function, where negative values are chosen by the minimizing player and positive ones are chosen by the maximizing player. At each level, the algorithm gets the results of each of its child nodes then selects the move with the best value according to the estimator.

In effect, minimax searches the entire tree of possible moves from each starting position, so its performance is strongly tied to the branching factor of the game tree. Alpha-beta pruning is a strategy that reduces the search space by eliminating (or "pruning") branches that cannot possibly be the maximum or minimum choice from a particular position (i.e., moves that minimax would never choose anyway). However, the alpha-beta strategy is inherently sequential as branches are pruned based on the results of their neighboring nodes.

## Opportunities for parallelism

These algorithms take a game state, essentially a board configuration as well as the current player, as input. The output is the board location of the optimal next move for that player. As described earlier, we see that the computationally expensive part of the algorithm is expanding the game tree, as the game tree grows exponentially as we increase the depth of the tree. In the minimax algorithm, there is ample opportunity for parallelism as the algorithm is independent at each level and thus each child can be computed in parallel. As this is the bulk of the work done, there are few dependencies in this algorithm. In the alphabeta algorithm, the algorithm is inherently sequential, as each child depends on the results of those before it in order to be able to correctly prune.

We decided to explore loosening some of the ordering requirements in order to gain opportunities for parallelism. One approach we considered was to combine minimax and alphabeta, such that at the top level we run the children independently as in minimax, but then at lower levels run alphabeta. However, this approach was too inflexible especially for varying numbers of cores and varying search depths, and would incur overhead. Instead, we ran alphabeta in parallel, but terminated jobs early if another child indicated the “prune” case. Here, we are able to scale to as many cores as we know we can use, but still restrict work when it is not necessary. While there will be some wasted computation here, we anticipated that this would scale well to larger search depths.

Considering other types of parallelism, this algorithm is not data-parallel as the main source of work is computational. Thus, we did not consider data locality to be an important factor in our parallelism approaches. This problem is also not amenable to SIMD execution, because there are large amounts of heterogeneous computation to be done. Thus, our approach was to use task parallelism to speed up the algorithm.

## Methodology

We used C++ for the implementation, OpenMP for parallelism, and targeted the PSC machines to use the large numbers of threads (since our solver would be parallelized over CPU cores). We chose this configuration because we used task level parallelism, and we wanted to have access to machines with many threads that could maximize speedup. We mapped the problem to threads by having each one solve a child node in the game tree, meaning the child nodes would be computed in parallel by multiple threads.

Our implementation involves multiple components. The first component was the game itself. We implemented an Othello game class that takes in arguments for the location of the next piece to place. It checks the legality of submitted moves and determines when the game is over. We considered whether to make our implementation of the game mutable but, we found that making an immutable game state is much more amenable to parallelism as we can pass along the states rather than having to revert moves or work in shared memory.

Next, we implemented a sequential solver that uses the minimax algorithm. To benchmark our algorithm, we ran the entire game using the solver for each move—essentially, pitting the solver against itself. Our parallel and sequential code utilized subclassing, allowing us to switch out game solvers at runtime. We explored whether to structure the algorithm recursively or iteratively, and decided to adopt a recursive approach as the algorithm is inherently recursive. However, we iterated over child nodes in the game tree, which was the key axis of parallelism (and where we used OpenMP). We originally parallelized the estimator function, but found in our experiments that this actually slowed down the solver since the computation is intended to be small and not worth the parallelism overhead.

## Key design decisions

Next, we implemented the alphabeta algorithm to prune game states during the search. Alpha-beta pruning can greatly reduce the amount of work done by the algorithm, particularly if the pruning is done higher up in the tree. This can give us significant speedups even before working on our parallel approach. However, we'd still like to take advantage of parallelism as there is still a significant amount of work to be done by the algorithm. We considered multiple avenues of parallelism with alphabeta—since the algorithm is inherently sequential, we decided to loosen the ordering restrictions in order to achieve better parallelism. We considered combining the minimax and alphabeta algorithms, i.e. ignoring the dependencies and not pruning at certain levels, but found that this would be too difficult to tune based on variable parameters and would not scale well. So, we ran alphabeta in parallel until a node was pruned, then set a flag for all other threads to skip their computation. This wastes some iterations on threads that merely iterate over game tree nodes but skip any work, but frees up resources *as soon as possible* to return the solver result and should overall reduce the amount of work done while still taking advantage of parallelism.

Another decision we made in parallelizing our algorithm was whether to use static or dynamic scheduling in our parallel for loops. In our tests, we found that dynamic scheduling performed better, which made sense because there would be workload imbalance between different child nodes, as some subtrees may terminate early due to no legal moves being left. In addition, the above nuance in our alpha-beta pruning meant that some threads will terminate even earlier so dynamic scheduling is most effective in this case.

To get a better understanding of how our solvers were working in the context of the actual game, we created a simple program to visualize the moves of the AI on an imitation game board. It was implemented as a standalone HTML webpage, where you can upload the trace outputted by our benchmarking utility that logs moves taken by each AI player as well as their computation time. The visualization tool replays this trace in real-time, highlighting each move along the way. Not only did this serve as a valuable debugging tool to make sure our solvers were making strong, legal moves, it helped illustrate the relative performance of various solver/search depth/core count combinations since we could see how fast or slow the AIs moved.

## Results

### Experimental setup

We measured performance using a combination of wall-clock time and speedup. Specifically, we measured both the total compute time of running the game (repeatedly making moves until the game ends) as well as the average move time (how long each move took). We used average move time as our primary metric because some games could end in fewer moves than others if the board reaches a configuration where no more legal moves can be made. We found that most games ended in around 60 total moves, i.e. each player makes 30 moves before the terminal state is reached. However, a few games ended in roughly 40 total moves, which reduced the total computation time in those tests.

In our experimental setup, we adjusted three variables: solver algorithm used (sequential minimax, sequential alphabeta, parallel minimax, or parallel alphabeta); number of processors (from a single core to 64, incremented in powers of 2); and search depth (starting from 1 to around 13), We observed that at a search depth of 13, all our solvers would surpass a timeout limit of 10 minutes. This gave us an important tradeoff to consider—at a higher search depth, the algorithm gains more information about future move states and can make a more informed decision on which move to make, but also has to perform exponentially more computation. We want an Othello AI to make decisions speedily but still have good information, so we found that the optimal search depth is between 5 and 9, depending on the use case for the application (e.g., in a casual game mode the search depth could be set to 5).

## Data analysis

Figures 2 - 4 illustrate the average move time over the number of cores, with a fixed search depth for each graph. Our baselines are the minimax algorithm written for a single-threaded CPU as well as a version with alpha-beta pruning but no parallel optimizations. We found that alphabeta consistently outperforms minimax, and even parallel minimax. This aligns with our expectations, because alpha-beta pruning can eliminate large amounts of work and thus greatly improve performance. Due to how pruning operates, it also makes sense that eliminating entire subtrees from the search, particularly early on, can reduce the amount of work more than the benefit of evaluating the entire subtree in parallel. At higher search depths, observe that minimax and even parallel minimax times out completely.

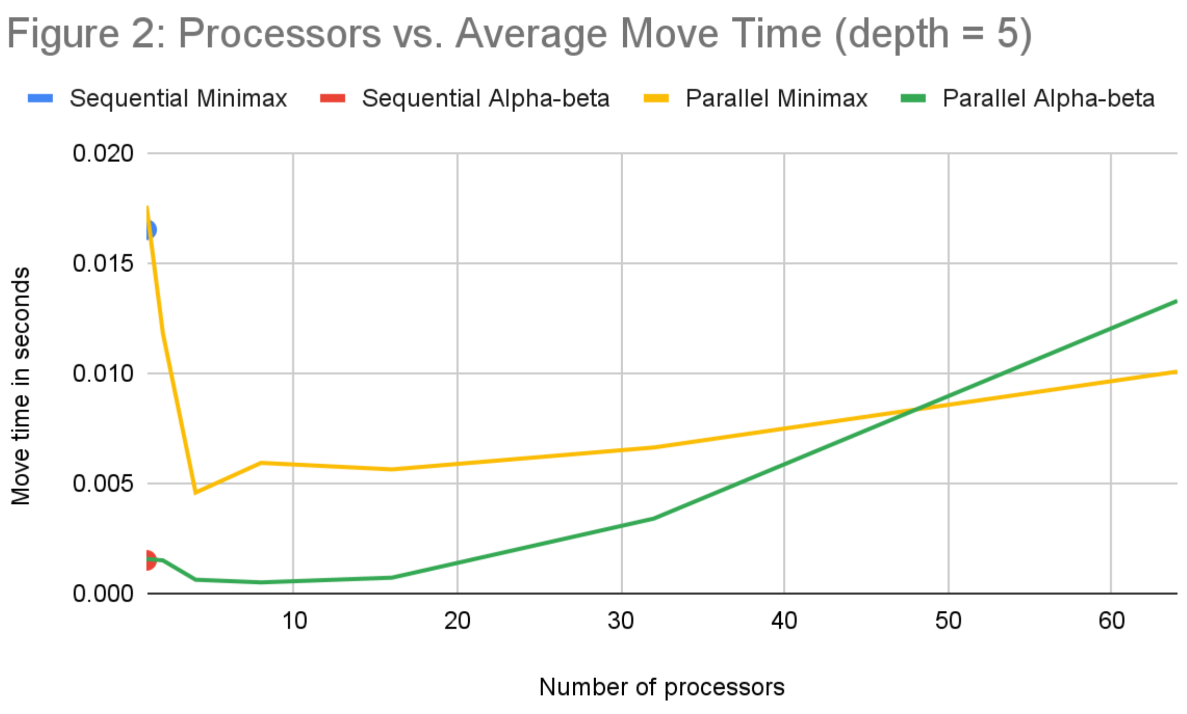

Figure 2: comparing average move times of the four solvers over varying numbers of cores at a search depth of 5

In Figure 2, we see that parallel alphabeta consistently outperforms parallel minimax at lower core counts, but is slower at the highest core count. This is likely due to the overhead incurred by how we structured our parallel alphabeta algorithm—with more cores, more work is being wasted when pruning occurs and there will be more communication as threads must terminate early.

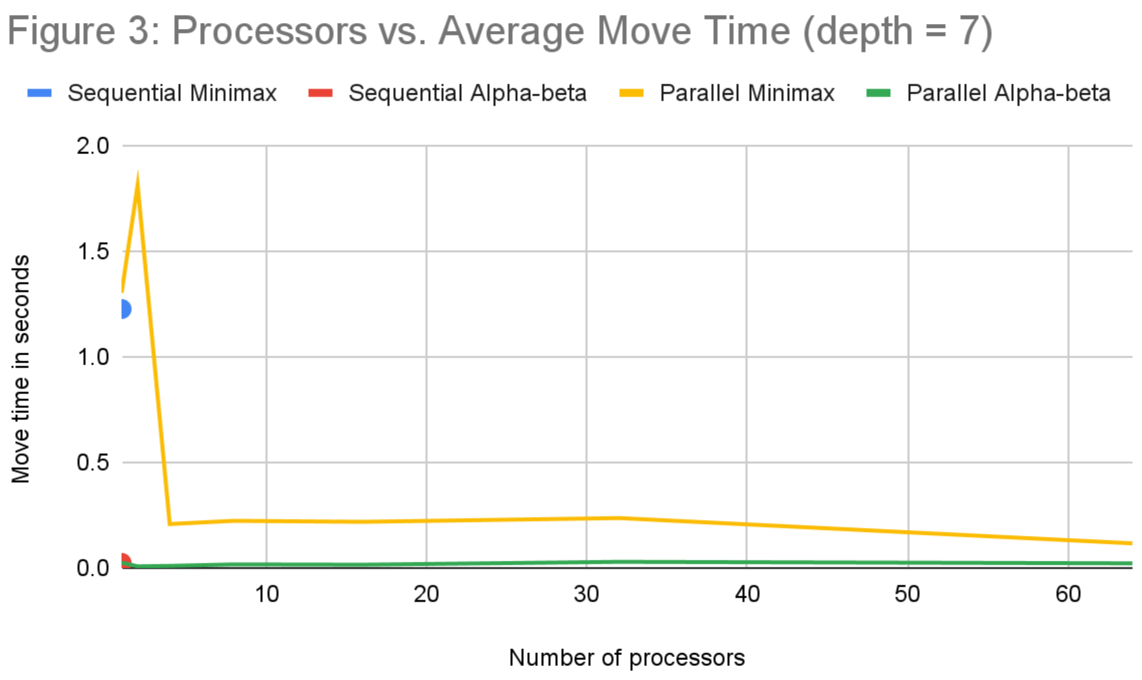

Figure 3: comparing average move times of the four solvers over varying numbers of cores at a search depth of 7

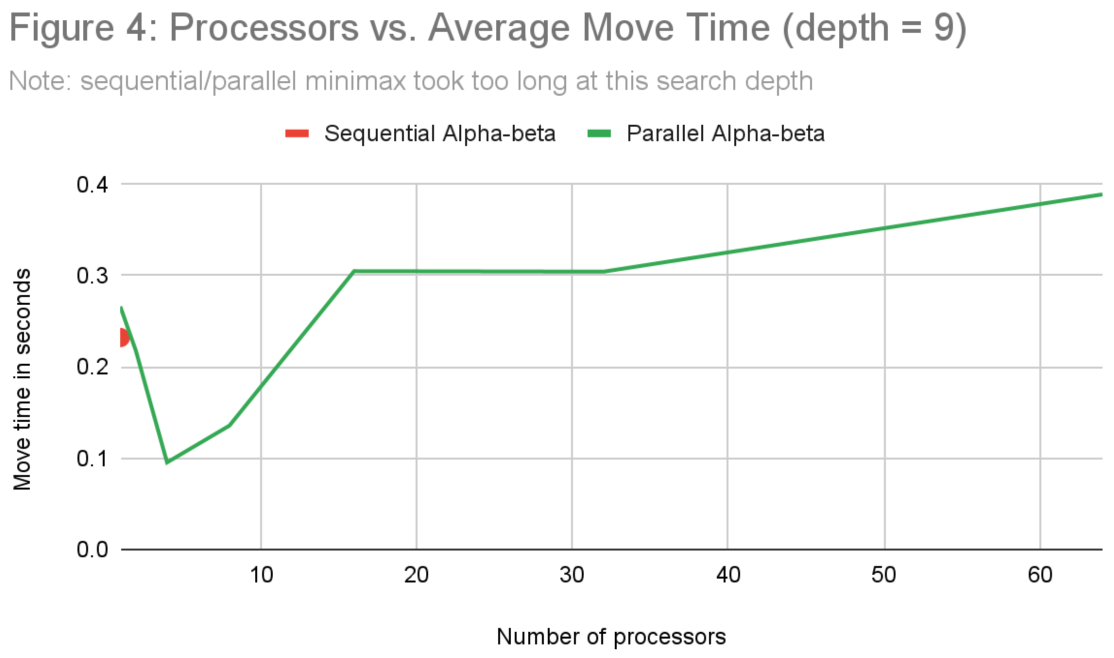

Figure 4: comparing average move times of the four solvers over varying numbers of cores at a search depth of 9; minimax algorithms time out and thus are not graphed

Taking a look at the differing search depths, we see that problem size can certainly affect our results. In Figure 3 and 4, we now see that with a larger search depth, the benefits gained from parallel alphabeta outweigh the overhead as parallel alphabeta consistently outperforms parallel minimax. This indicates that with a larger problem size, parallelism with no pruning can only provide so much in reducing the computation times, while pruning work can greatly improve the computation time.

One thing to observe here is that at higher core counts, sometimes parallel alphabeta performs worse than sequential alphabeta. Again, this is likely due to the communication costs and wasted work done, meaning the overhead will outweigh the slight benefits that parallelism may provide in this case. We have only shown one way of parallelizing the alphabeta algorithm in our project, but in the future more work can be done to test other ways of relaxing the algorithm constraints to perhaps provide more opportunities for parallelism based on the data we have gathered here.

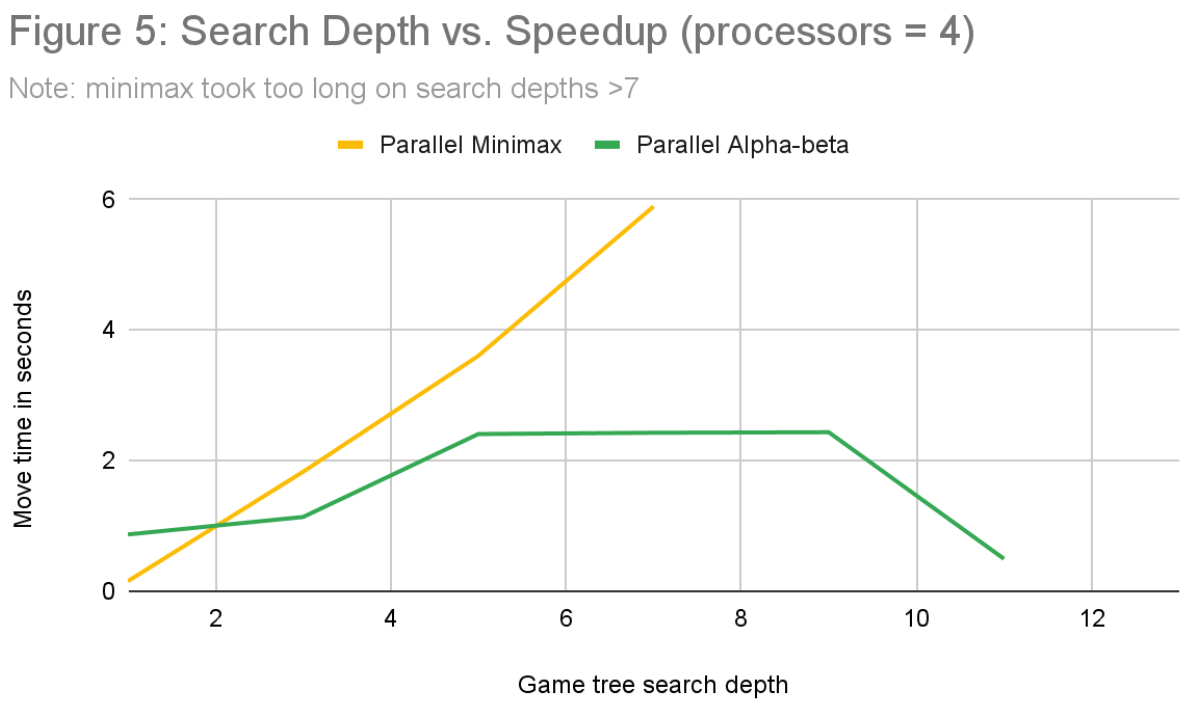

Figure 5: comparison of speedup of parallel algorithms from their sequential counterparts at 4 cores (no data for minimax past search depth 7 due to timeouts)

Next we will analyze how search depth impacts speedup with respect to the sequential version of each algorithm. As seen in Figure 5, when we fix the number of processors at 4, speedup increases linearly with search depth for parallel minimax. This is exactly in line with our hypothesis and the inherent structure of the minimax algorithm—recursing through a tree is highly parallelizable, especially since the work of each thread is independent.

On the other hand, we see more mixed results for parallel alphabeta with respect to its sequential implementation. Speedup isn't as high as it was for minimax, and tails off beyond a search depth of 5. This is a result of the drastic performance improvement brought about by pruning the game tree at any level of the search, which is something that the parallel algorithm cannot fully exploit since threads assigned to pruned nodes stay alive until the entire iteration is complete. As mentioned previously, the reduction of speedup is due to the fact that this algorithm is not inherently parallelizable. There are dependencies between the child nodes that mean what we can parallelize is limited, even if we loosen some of the ordering constraints. At even higher search depths, the parallel version actually takes longer than its sequential counterpart (i.e., speedup is less than 1) due to this issue and the overhead of spawning threads in so many recursive calls.

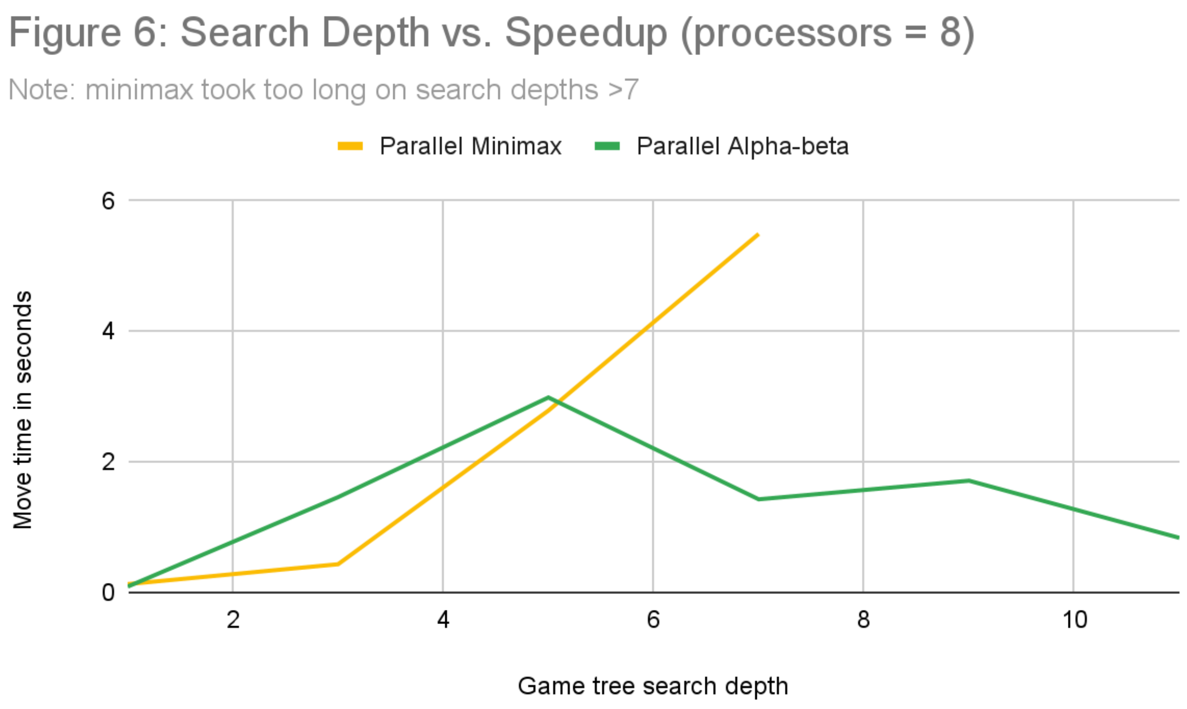

Figure 6: comparison of speedup of parallel algorithms from their sequential counterparts at 8 cores (no data for minimax past search depth 7 due to timeouts)

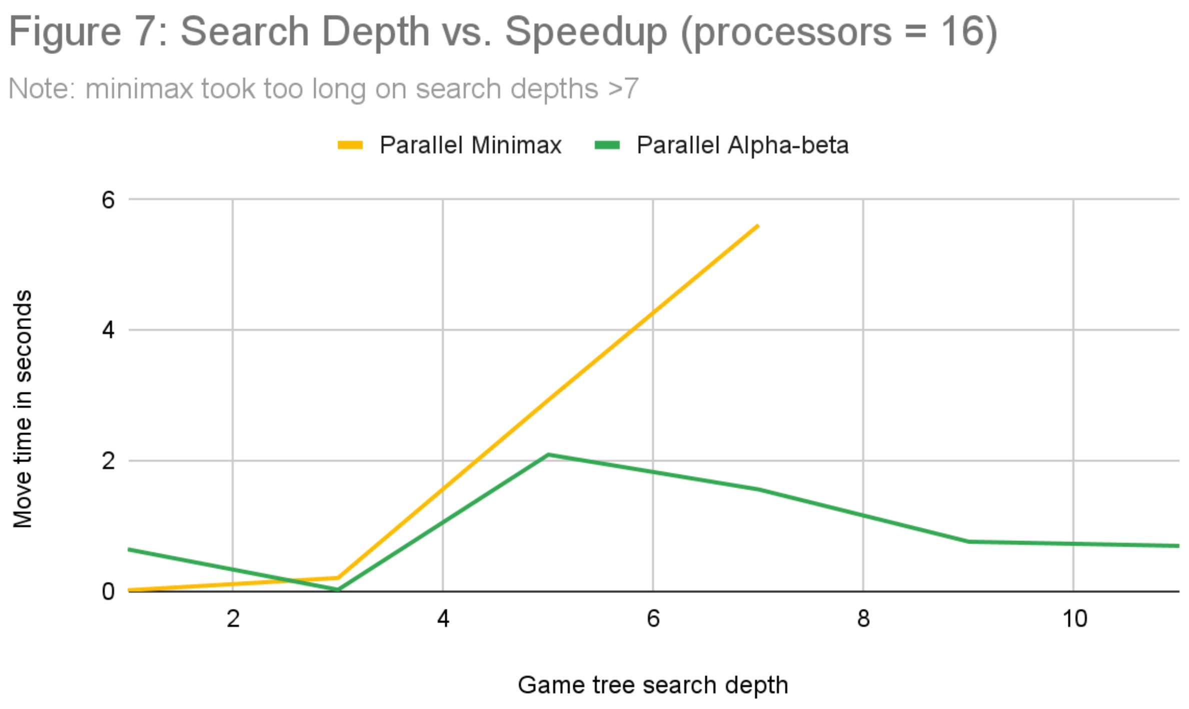

Figure 7: comparison of speedup of parallel algorithms from their sequential counterparts at 16 cores (no data for minimax past search depth 7 due to timeouts)

As we increase the number of processors in Figures 6 and 7, the same trend continues for the alphabeta algorithm: speedup drops off after a search depth of 5. We are unsure what explains the slight uptick at a search depth of 9 with 8 processors (perhaps this was just due to PSC system conditions when testing), but the overall pattern of the overhead and wasted iterations in the parallel implementation slowing down the solver over a protracted game tree search persists.

One other interesting observation emerges with more processors: at low search depths, we see that the parallel minimax algorithm is actually slower than the sequential version. This is likely due to the overhead of having to spawn threads to evaluate game tree nodes in parallel, but the computation itself is trivial due to the low search depth, making the overhead dominate the overall computation time.

## Closing thoughts

Overall, the results we collected were largely in line with our hypotheses given the implementation of both the minimax and alphabeta algorithms. We saw that nearly the entirety of the execution time for our benchmarking setup was taken by the solver computing the best next move for each game state. In particularly, there wasn't a lot of variance in the time taken to make each move since search depth was fixed. However, the average move time itself was strongly correlated with a combination of the number of processors and search depth we selected. We observed the best speedup at medium core-counts and reasonable search depth, with the overhead of parallelism becoming a dominating factor that resulted in subpar performance in other scenarios. Although we believe our choice of a task-parallel approach of CPU threads was the right one for this application, specifically due to the setup of the problem and the unknown nature of the problem size upon embarking, we want to explore how other technologies such as Cilk and CUDA could change our results.

## References

Othello rules:
https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english
Minimax: https://en.wikipedia.org/wiki/Minimax
Alphabeta: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
Othello heuristic ideas: https://www.cs.cornell.edu/~yuli/othello/othello.html

### Work distribution

| **Ashwin** | **Miranda** |
|---|---|
| Background research<br>Game implementation<br>Solver implementation<br>Parallel solver implementation<br>Benchmarking<br>Visualization<br>Report (Results section) | Background research<br>Game implementation<br>Solver implementation<br>Parallel solver implementation<br>Report (all sections)<br>Presentation |
| 50% total | 50% total |