Othello and Parallel Game-tree Search

Dane Johnson and Aaron Handlman

12-21-2020

1 Introduction

The game of Chess has always been captivating to game enthusiasts, and doubly so to computer scientists. Aaron and I were the latest in a laundry list of computer scientists who became interested in the game. Given that our research is in parallel computing, and seeing as we needed a project for this course, it was only natural that we decided to investigate how parallelism could be applied to the great game.

Game-tree search soon became our primary area of concern. It is an inherently serial algorithm, challenging to extend to multiple workers. With the intent to focus on search algorithms, we gravitated away from Chess towards Othello, a simpler game with only one type of piece. With this new interest, we decided our project would be one of implementation and analysis, the prospect of creating an entire Othello engine from scratch was a much more tractable problem.

Thanks in no small part to the Chess Programming Wiki [1], we were able to find a wide variety of Game-tree search algorithms to implement, and benchmark against each other for runtime and scalability.

2 Othello

Othello, or Reversi, is a game played by two players on an 8x8 grid, the same dimensions of a Chess board (and I will therefore use a Chess board to demonstrate the game), with disks that are white on one side and black on the other. The game is played by placing a disk of the player's color on the board, then flipping any disks that are sandwiched between and existing player's piece and the new one. With only a single type of move, Othello is a much easier engine to implement than Chess, but Game-tree search algorithms should be just as, if not more, effective at playing the game.

Figure 1: The standard Othello start position

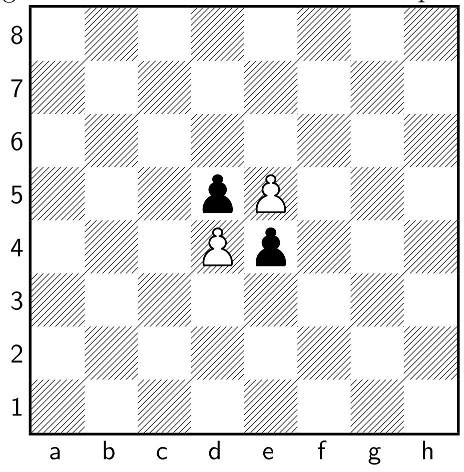
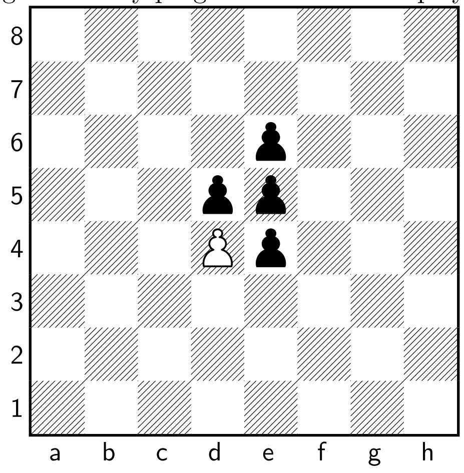


Figure 2: Play progresses after Black plays e6



3 Algorithms

3.1 Game-tree Search (Minimax)

The core of Game-tree search is looking through the search space trying to find the move that maximizes your chances of victory, under the assumption that your opponent will do the same in the opposite direction. Since theses spaces are intractably large, we choose some maximal depth, nodes searched, or time spent looking and then evaluate the board via some criteria (in Othello, how many more moves we have than our opponent).

```
procedure MINIMAX($node, depth$)
    if $depth = 0$ then
        return EVAL($node$)
    else
        $score \leftarrow -\infty$
        for all $c$, children of $node$ do
            $score \leftarrow -$ MAX($score$, MINIMAX($c, depth - 1$))
        end for
        return $score$
    end if
end procedure
```

3.2 Alpha-Beta Pruning

Alpha-Beta pruning improves Game-tree search by keeping track of the highest and lowest values that are guaranteed to each player. If a move is found to be worse than a score guaranteed by these values, the search of that branch terminates early.

```
procedure ALPHABETA($node, depth, \alpha, \beta$)
    if $depth = 0$ then
        return EVAL($node$)
    else
        $score \leftarrow -\infty$
        for all $c$, children of $node$ do
            $score \leftarrow -$ MAX($score$, ALPHABETA($c, depth - 1, -\beta, -\alpha$))
            $\alpha \leftarrow$ MAX($score, \alpha$)
            if $\alpha \geq \beta$ then                    ▷ $\beta$ cutoff
                Break
            end if
        end for
        return $score$
    end if
end procedure
```

3.3 Principal Variation Search

PVS improves further on Alpha-Beta pruning by assuming that the first child of a node is it's "Principal Variation", and that it is likely to be the best choice. It runs a search on the other children with a narrow window (the Scout window), so they will cut off if they generate a value that is smaller than the principle variation. This allows for even more aggressive pruning.

```
procedure PVS($node, depth, \alpha, \beta$)
    if $depth = 0$ then
        return EVAL($node$)
    else
        for all $c$, children of $node$ do
            if $c$ is the "Principal Variation" then
                $score \leftarrow -$ MAX($score$, PVS($c, depth - 1, -\beta, -\alpha$))
            else
                $score \leftarrow -$ MAX($score$, PVS($c, depth - 1, -\alpha - 1, -\alpha$))    ▷ Scout window
                if $\alpha < score$ then
                end if
                $score \leftarrow -$ MAX($score$, PVS($c, depth - 1, -\beta, -\alpha$))
            end if
            $\alpha \leftarrow$ MAX($score, \alpha$)
            if $\alpha \geq \beta$ then                                   ▷ $\beta$ cutoff
                Break
            end if
        end for
        return $\alpha$
    end if
end procedure
```

3.4 Shared Hashtable

Shared hashtable is a naive approach to adding parallelism to Game-tree search. We send of multiple agents to search the space, and cache results in a table. This allows for parallelism without the need for any added implementation complexity.

3.5 Jamboree

Jamboree is a parallel version of PVS. It utilizes the "Young Brothers Wait" concept, the idea that parallelism should be saved until it is needed and that searching the principal variation is inherently serial [**?**, jamb] Parallelism is used for scouting once the principal variation is searched, and any later variations that fail the scout are searched in serial as well.

```
procedure JAMBOREE($node, depth, \alpha, \beta$)
    if $depth = 0$ then
```

```
3:          return EVAL($node$)
4:      else
5:          $score \leftarrow -$ MAX($score$, JAMBOREE($pv, depth - 1, -\beta, -\alpha$))  ▷ Search the "Principal Variation"
6:          for all $c$, children of $node$ do                ▷ In parallel
7:              $score \leftarrow -$ MAX($score$, JAMBOREE($c, depth - 1, -\alpha - 1, -\alpha$))  ▷ Scout window
8:              if $\alpha < score$ then
9:                  Wait for all previous parallel iterations to finish
10:                 $score \leftarrow -$ MAX($score$, JAMBOREE($c, depth - 1, -\beta, -\alpha$))
11:             end if
12:             $\alpha \leftarrow$ MAX($score, \alpha$)
13:             if $\alpha \geq \beta$ then                     ▷ $\beta$ cutoff
14:                 Break
15:             end if
16:         end for
17:         return $\alpha$
18:     end if
19: end procedure
```

4 Implementation Details

Our implementation was written in C++ using the Cilk runtime and compiled with the Tapir Cilk compiler. We ran our benchmarks to test both Shared Hashtable and Jamboree, as well as a naive parallel implementation of Alpha-Beta and PVS which simply parallelized the root node. In order to reduce spawn overhead, we limited parallelization for Jamboree to the first $\log(P)$ levels as our experimental results showed that parallelization past that caused significant slowdown due to the exponential nature of the tree. We focused our performance engineering on the evaluation of the leaf nodes, as this had a far greater impact on the overall performance. For our evaluation function, we chose to use the number of legal moves for the current player minus the number of moves for the opposing player. This is very effective for a simple evaluation function that doesn't require significant tuning, as in Othello having a large number of moves typically means that you have the ability to take control of move of the board. Unlike many games, simply counting the number of pieces is not effective as there can be very large swings in this value from turn to turn. Another advantage of this evaluation function is that it tends to reduce the width of the search tree allowing for deeper searches. This is because both players are trying to minimize each others moves, even if they are trying to maximize their own. This leads to a trend of preferring lines that have a smaller branching factor. Our code can be found at `https://github.com/dane-johnson/othello`.

5 Evaluation

We ran our benchmarks on Adams, which has an Intel(R) Xeon(R) CPU E5-4620 processor, 500GB of RAM, and a 16384K L3 cache. Additionally, it has 4 NUMA nodes, although we did not make use of this as we were executing on vanilla Cilk and therefore didn't have an effective way of binding memory to NUMA nodes. Our results were generated by running each algorithm 5 times as both black and white at each depth and taking the average of the running time for these games. The games were played against an agent which simply picked a random move out of the available moves. Although this is not a very good method for playing the game, we found that it generated more interesting games for comparing the different algorithms because it lead to a higher branching factor. Between the random agent and grouping games played as white and black, our standard deviations were higher than we would otherwise prefer, however we were still able to get a good idea of the trends. In order to count the number of nodes checked we chose to use an agent which always picked the first move instead of the random agent, in order to get more consistent results. We did this differently than the timing because we wanted to get a sense of an average game for timing whereas we simply wanted to see how much the different algorithms would trim for the number of nodes searched.

Experimental Results

We saw that at a depth of 11, SerialPVS searched an average of 221793 nodes per turn, ParallelPVS search an average of 2081209 nodes per turn. This is due to the drastically reduced trimming based on less information gained. AlphaBeta searched an average of 602914 nodes, while ParallelAlphaBeta searched 5743813 nodes. This corresponds to a roughly 10x increase in the number of nodes searched by parallelizing the algorithm. Our benchmarks compare the various parallel algorithms.

6 Future Work

For future work we'd like to start by solving some of our performance issues for Shared Hashtable. With better performance we could add in additional algorithms that rely on a form of shared hashtable, such as ABDADA. We'd also like to investigate how we could better tune the shared hashtable to be NUMA aware, although this is a far more difficult problem than we initially thought. Even using a locality based runtime, we would still need a heuristic to figure out which boards are likely to collide on the hashtable and put those moves on the same NUMA node. We'd also like to do further performance tuning, and implement a way to test the performance of different evaluation functions.

References

[1] Various Authors, *The Chess Programming Wiki*, `www.chessprogramming.org`

[2] C. F. Joerg and B.C. Kuszmaul. Massively Parallel Chess. *DIMACS '94* October 1994.