ME759 Final Project Report
University of Wisconsin-Madison
2017

”Work Faster, Not Just Smarter” – Performing Minimax Parallelization in Othello with OpenMP

Hancock, Derek

December 2017

Abstract

This project evaluates the effectiveness of parallel processing using OpenMP in the minimax algorithm used in a timed adversarial gaming environment. This is accomplished by writing several agents that will use the same algorithm and heuristic board evaluation logic but utilize the hardware in different ways. The agents include a baseline sequential agent, a naive OpenMP agent, and an optimized OpenMP agent. I show that better optimization and hardware translates into a higher winning percentage.

Contents

1 Introduction

The purpose of this project is to explore the application of parallel processing using OpenMP in a tree algorithm. In this project, we consider the minimax algorithm in the game known as Othello. The goals of this project can be broken down as follows:

1. Build the game environment with server and client code to play Othello
2. Use OpenMP to perform the minimax algorithm in parallel
3. Optimize the OpenMP implementation
4. Show how optimized code and parallel processing directly translates into a better win percentage in this competitive gaming environment

In artificial intelligence theory, the performance of an algorithm is generally judged by its logical merits and superior space/time complexity. This course opened my eyes to the importance of optimization in the application of AI algorithms. Through this project, I hope to show that optimized code translates directly wins in adversarial games just as traditional AI logic does.

2 Setup and Game Environment

Othello (sometimes also known as reversi) is a classic and popular board game dating as far back as the 1800s. While it is relatively simple to play and understand, the mechanics of the game mean that a board can have 10^{28} possible states, and it is still mathematically unsolved [2]. The ease of which pieces can easily flip colors also makes it difficult yet important to look ahead at future states of the game. The games are also timed so that either player only has 1 minute total to make all their moves during a game. As such, it is an excellent setup for exploring the effectiveness of parallelization.

2.1 Othello Mechanics

Othello is played on a 8x8 board between 2 players with black and white pieces. Each player takes turn playing a piece, and the goal of the game is finish with the most pieces on the board. At the start of the game, each of the first 4 moves need to reside in the center 4 positions on the board (figure 1).

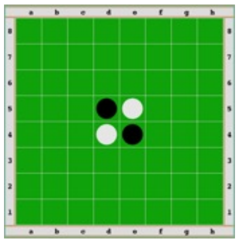
Figure 1: A game after the initial 4 moves

After that, a valid move is made at a position such that you have a piece on either side of 1 or more of your opponent's pieces. When a move is made, all of the opponent's pieces inside the "sandwich" of your pieces are flipped to your color (see figure 2). If there is no valid move, your turn is skipped. The game ends when one player has no more pieces on the board or there are no more valid moves for either player.

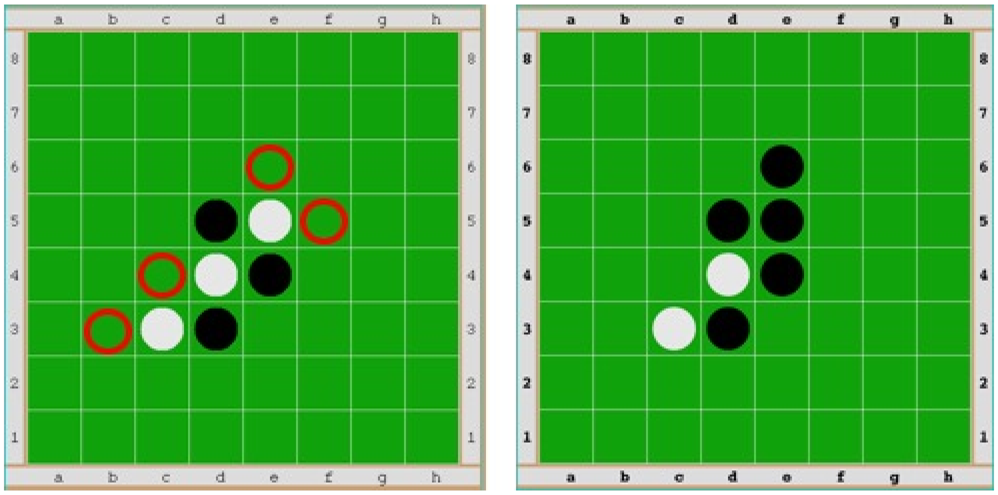

Figure 2: An example showing the possible moves for black during its turn, and then the result of placing a black piece at position 6e

2.2 Client and Server Code

The first step was to get a working game up and running. This required:

- A game server to track overall game state such as board state
- Server player to send and receive move information with the clients
- Clients to logically decide which moves to make

Once I finally finished the server and client infrastructure, I could move on to minimax and then parallelization.

3 Minimax and Depth

3.1 Minimax

The minimax algorithm is a classic approach in game theory for artificial intelligence agents to decide which move to make. It works by expanding a tree of possible moves. At alternating levels of the tree, it is trying to maximize its possible reward. At the other levels, it assumes its opponent will try to minimize the possible reward from the given possible moves. At the leaf nodes of the tree, the possible future game state is evaluated according to some heuristic function. These values propagate upward through the tree until the agent knows which decision would be optimal (see figure 3).

Minimax will be ineffective if the heuristic function doesn't reasonably capture the true state of the game. For this project, I used a simple heuristic function that calculates its score based off of a weighted version of the board. For example, in Othello it is advantageous to get the corner and side pieces, so the heuristic function gives a high score to corner and side pieces and low/negative scores for positions that allow the opponent to get the corner and side pieces. This simple heuristic works suitably well enough so that minimax is effective.

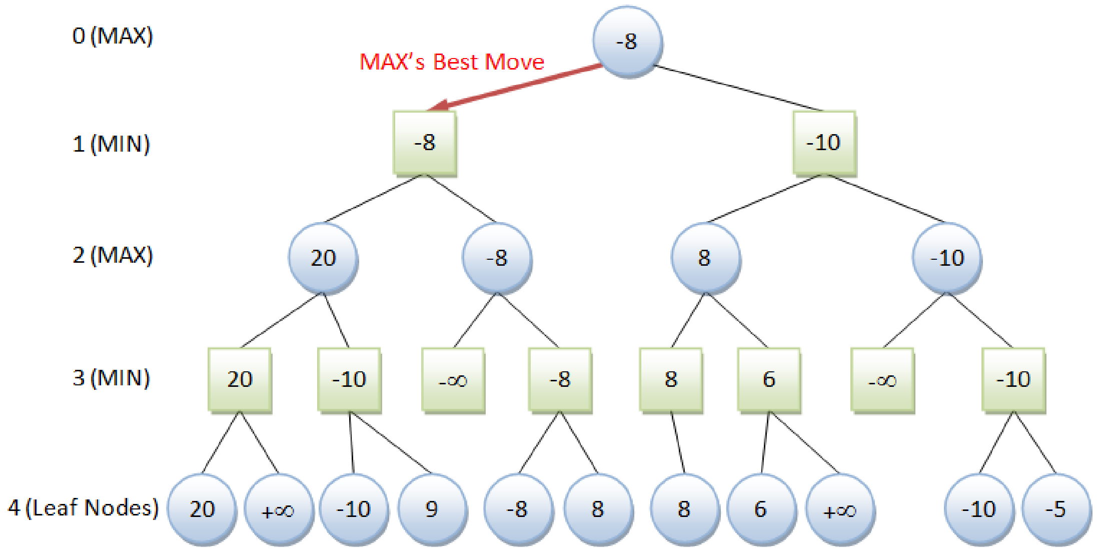

Figure 3: Minimax tree. Used from [1]

3.2 The Importance of Depth

Of course, as the tree gets deeper, the AI player is able to predict further out into the future and have a better understanding of what decision to make. Given similar board evaluation functions, the player that can expand deeper will be more likely to win the game. Of course, the player can only expand the tree so deep given a limited amount of time because the tree grows exponentially. Therefore, a parallelization approach might directly help an agent expand deeper and thus win more.

In this setup, each player has only 1 minute total to make their moves. If a player tries to expand their tree too deep, they might not determine a move in time to finish the game. I added a timing mechanism to the minimax agent so that it takes a maximum of 2 seconds per turn to make a move. If it hits its time limit, it stops building the tree, evaluates it, and returns.

3.3 Verifying Functionality

Before parallelizing minimax, I verified the correctness of a baseline minimax algorithm. I played it against a random player to check that it would win almost every game. At first it was winning only 60% of the time against random. After finding and fixing a couple bugs, it was winning 96-98% of the time.

I also verified that increased depth actually translates into more wins. I played the same minimax agent against itself but set different maximum depths. Below is a table showing the record (p1 wins-p2 wins-ties) out of 50 games each. Orange boxes are for player 2 overall more wins and blue for player 1 overall more wins. These results confirm that a deeper depth tends to increase win percentage.

Table 1: Baseline Minimax Results

		Player 2				
	depth	6	5	4	3	2
Player 1	6	19-26-5	31-19-0	42-8-0	48-0-2	50-0-0
	5	16-33-1	28-21-1	37-12-1	28-18-4	33-15-2
	4	12-35-3	22-25-3	22-27-1	29-17-4	42-8-0
	3	17-32-1	21-29-0	28-22-0	29-20-1	50-0-0
	2	0-49-1	13-37-0	4-39-7	25-25-0	18-25-7

4 OpenMP and Minimax

I decided to approach this project because I was interested in parallelizing a tree algorithm like minimax. In the course, most parallelization that we saw was in regards to large matrices or arrays instead of tree-like structures. This section describes how I used OpenMP to parallelize the algorithm. For this project, I limit the number of threads used to a maximum of 20.

4.1 Measuring Performance

In order to verify the effectiveness of parallelization, we need an easy way to estimate its performance. Because each game has different possible moves (and thus tree sizes), it isn't useful to compare the amount of time an agent takes to make a turn or even the number of nodes it explores throughout a game. In order to measure performance, we look at the average number of nodes explored per millisecond.

4.2 OpenMP Steps

An approachable way to add parallelism is to divide the tree and split it up between the threads. Because each branch can be independently explored from the others, the root node can assign a different branch to each thread and choose the best one from among their results. Each branch can work independently because they don't require information or communication from the other branches, which can make this approach very effective. We can do this by using n threads at the root, where n is the number of moves available to the root node. Multiple threads are used only at the first level of the tree so that the number of threads doesn't increase exponentially.

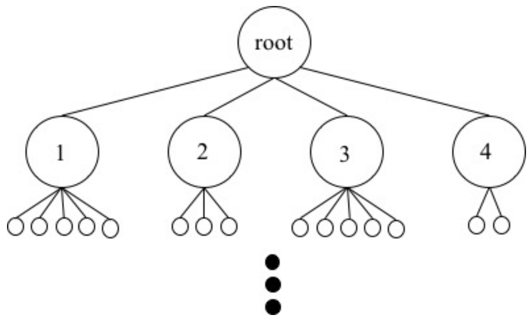

Figure 4: Here 4 threads each evaluates its own subsection of the tree

I compared either approach at the same player position (player 2) on an identical game. The baseline agent with no parallelization is able to expand an average of 715 nodes per millisecond. With OpenMP parallelization of this manner with 20 threads, the OpenMP agent gets only 494 nodes per millisecond on average. With some minor changes, we can increase this to much higher than the baseline. We call this current version of the OpenMP agent the ”naive OpenMP agent.”

4.2.1 Speed up #1: Fewer threads

20 threads is too many in this case. The root node generally only has 6 to 12 moves available to it. By decreasing the number of threads and reducing the overhead of thread management, the OpenMP agent can now expand 812 nodes/ms, which is now higher than the baseline.

4.2.2 Speed up #2: Dynamic Scheduling

The default scheduling approach for OpenMP is to divide up the tasks evenly. This approach isn’t well suited for when the tasks require largely varying amounts of time. This is certainly the case for minimax, where each subtree may have varying numbers of nodes to explore depending on the available moves. By changing the scheduling to dynamic, the rate goes up to 840 nodes/ms.

4.2.3 Speed up #3: Vector Reserve and Critical Section

At first I thought that it would be important to avoid a critical section. In order to avoid it, I reserved the necessary amount of space in a node’s children vector before expanding and then placing any new children at their corresponding index in the vector. However, this reserve is expensive because at the bottom of the tree it reserves space that won’t be used as the agent runs out of time and returns prematurely from its expansion.

A critical section, which generally should be avoided if possible, is actually valuable here. It allows us to avoid reserving unnecessary space. It isn’t costly here because the threads only potentially have to wait for another thread at the root level. By using this critical section instead, the speed goes up to 1346 nodes/ms.

Now after these change, we can look at a scaling analysis of the number of threads to use.
The graph in figure 5 shows that we can settle on using 6 threads at 1538 nodes/ms. This is more than double the baseline’s 715 nodes/ms.

This ability to expand more nodes in a given period of time gives the OpenMP agent a better idea of the future states of the board. In a sense, it makes the OpenMP agent ”smarter” even though both the baseline and OpenMP agent have the same logic and depth limit. The OpenMP improvement translates directly into wins. Playing against each other out of 30 games, the OpenMP agent wins 63.3% of them. The optimized OpenMP agent also wins 73.3% of the games against the Naive OpenMP agent that we started with.

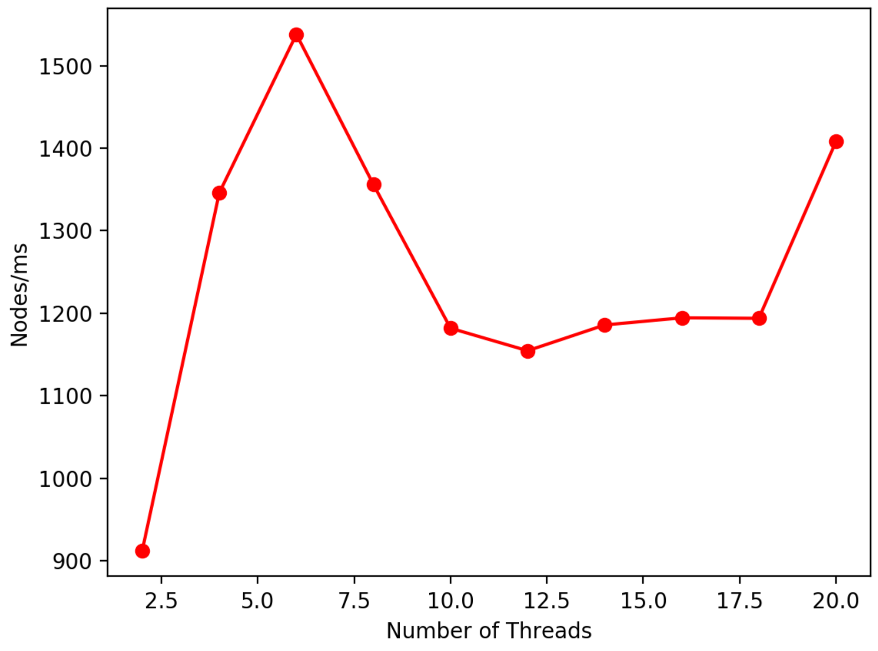

Figure 5

5 Future Work

Given more time, I would have liked to explore different approaches to using OpenMP with the minimax algorithm. Additional approaches might include using multiple threads at depths lower than just the first level or using tasks to split up sections of the tree. Also, there are certainly many optimizations that could still be done to the OpenMP agent that I wasn't able to explore because of time constraints. It also would be interesting to explore this same problem but use parallel techniques that may scale better than OpenMP for deep minimax trees, such as GPU computing or MPI.

6 Conclusion

In this work I have used a timed adversarial gaming environment to test different kinds of agents. I have completed the previously stated goals to build the environment, implement the tree-like minimax algorithm, use OpenMP, and then optimize the use of OpenMP. I showed how simple improvements can lead to a higher winning percentage against less-optimized opponents. The results of this project prove what we have learned all semester in the course–that code optimizations and speed ups are significant and valuable in many domains and applications.

References

[1] Case Study on Tic-Tac-Toe Part 2: With AI. `https://www.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html`. Accessed: 2017-11-30.

[2] Louis Victor Allis. Searching for solutions in games and artificial intelligence, 1994.