Parallel Othello Reflection

Bo Sung Kim

February 27, 2024

1 Othello

Othello, also known as Reversi, is a strategy board game played between two players. The game is played on an 8x8 grid (the board), with the objective being to have the majority of disks flipped to display your color when the last playable empty square is filled. A player must place a disk on the board in such a position that there exists at least one straight (horizontal, vertical, or diagonal) occupied line between the new disk and another disk of the player's color, with one or more contiguous opponent's disks between them. If a player cannot make a valid move, they must pass their turn to the opponent. The game ends when neither player can make a valid move, typically when the board is full or when no further flips are possible. The player with the majority of disks of their color on the board at the end of the game wins.

1.1 My Solution

My solution evaluates and makes the best possible move for a player at any given state of the board. During each turn, the solution explores potential future board states and decides on the move that maximizes the current player's advantage based on a heuristic function's evaluation of the board. The heuristic function assesses the board state based on the number of disks each player has, with more disks indicating a better position.

More specifically, I use a depth-limited search algorithm based on the Negamax variant of the Minimax algorithm, which is a common approach in two-player zero-sum games like Othello. The Negamax algorithm simplifies the implementation of Minimax by leveraging the fact that max(a, b) = -min(-a, -b), thereby reducing the need for separate min and max functions for the two players.

If the search depth is reached or a terminal state of the game (such as a full board or a state where no legal moves are available), the algorithm calculates the board's heuristic value from the perspective of the current player.

Otherwise, we recursively call the algorithm for each possible move, flipping the perspective (player) each time with a copy of the board with the move applied and a decremented search depth. The core of the Negamax algorithm is encapsulated in the formula $value = -\text{Negamax}(child, depth - 1)$, where

child represents the game state after a potential move, and *depth* controls the recursion level. The hueristic for that move is the negation of the returned heuristic value to reflect the current player's perspective.

Once all possible moves have been explored, the best move is one that leads to the highest evaluation score. Furthermore, in the case that there are no possible moves for the current player, their turn is passed to the opponent. Below is the pseudocode for my approach.

```
function NegamaxSearch(board, color, depth)
    if depth == 0 or no legal moves
        return EvaluateBoard(board, color)

    max_value = -Infinity
    for each legal_move in legal moves in the current board state
        copy_board = board
        ApplyMove(copy_board, legal_move, color)
        value = -NegamaxSearch(copy_board, OTHERCOLOR(color), depth - 1)
        if value > max_value
            max_value = value
            best_move = legal_move
    return best_move

function EvaluateBoard(board, color)
    return number of color disks - number of OTHERCOLOR(color) disks
```

1.2 Parallelism

1.2.1 Parallel Search

```
Function ParallelSearch(Board board, Color color, int depth)
    if depth == 0
        return EvaluateBoard(board, color)

    maxValue := -Infinity
    moves := All legal moves for this color on current board

    // Initialize a reducer for maximum value tracking
    var maxReducer = new ReducerMax()

    // Parallel for loop to iterate over all possible moves
    cilk_for each move in moves
        newBoard := ApplyMove(board, move, color)
        value := -ParallelSearch(newBoard, 1-color, depth-1)

        // Thread-safe update to the reducer
        maxReducer.update(value)
    return value from the maxReducer
```

```
End Function
```

The Othello program exploits parallelism primarily through the use of the $cilk\_for$ construct, which allows for the simultaneous evaluation of multiple game states. This is particularly evident in the $ParallelSearch$ function, where each iteration of the loop, corresponding to a different potential move, is executed in parallel. By applying each move to a separate copy of the game board and then recursively exploring subsequent moves from these new board states, the program constructs a parallel search tree. The use of reducers, such as $cilk :: reducer\_max$, further enhances parallelism by enabling thread-safe operations to determine the optimal move among all evaluated moves. This approach effectively divides the problem into independent tasks, each representing a avenue of the game outcomes that can be evaluated independently, thus maximizing the utilization of available processing cores.

I chose to exploit parallelism in this way because of the inherent nature of the Othello search, where each potential move represents an independent path that can be explored without dependency on others. This characteristic makes this aspect of the algorithm well-suited for parallel computation, as it allows for significant portions of the search space to be evaluated concurrently, leading to a reduction in overall computation time.

1.2.2 Granularity

Furthermore, I implemented a hybrid approach combining parallel and sequential search to optimize performance based on a defined "granularity" threshold. This granularity parameter determines the depth of the game search at which the search algorithm transitions from a parallel to a sequential mode. When the depth of the search exceeds this threshold, the program utilizes the $ParallelSearch$ function, using $cilk\_for$ to concurrently explore multiple possibilities of the game outcomes. As the search delves deeper and approaches the granularity limit, it switches to $SequentialSearch$ to avoid the overhead associated with spawning and managing additional parallel tasks at lower depths, where the computational workload might not justify parallel execution. For my implementation, I selected a threshold of 3 as it demonstrated the best results from cilkview.

This combination of sequential and parallel execution allows for the efficient use of parallelism at early depths of the search for the best move, where the branching factor results in a significant number of possible moves, while maintaining the simplicity and lower overhead of sequential execution at deeper levels. This choice was to balance the computational benefits of parallel execution against its overheads, ensuring that the search algorithm remains efficient across varying depths of the othello search and effectively utilizes available computational resources without the unnecessary overhead from parallel task management at deeper, less computationally sections.

1.2.3 Synchronization

Parallelism is synchronized using Cilk Plus features, notably $cilk\_for$ for parallel loops and reducers for thread-safe data aggregation. The $cilk\_for$ loop enables the program to concurrently evaluate multiple game possibilities, with each iteration running in parallel and independently. This parallel execution is synchronized at the loop's end, where the Cilk runtime ensures that all iterations are complete and synchronized before moving on.

Reducers, particularly $cilk :: reducer\_max$, synchronizes parallel work by allowing threads to update a shared variable without data races and conflicts. Each thread operates on a local copy of the reducer, and upon completion of the parallel region, Cilk Plus combines these local copies into a single, global value. This automatic merging process ensures that the maximum heuristic value found by the parallel evaluations is accurately determined without the need for manual synchronization or locking mechanisms, thus simplifying the code and enhancing performance by minimizing overhead. Because finding the maximum is an associative operation, we are able to effectively use Cilk reducers.

1.2.4 Further Parallelism

- Move Generation: The functions that generate legal moves (ReturnLegalMoves or EnumerateLegalMoves) is parallelizable, as the legality of each move can be determined independently of others. Parallelizing this step by rows or columns could reduce the time taken to generate the set of possible moves.
- Disk Flipping: The function FlipDisks which flips disks after a move is made could also be parallelized. Each direction (horizontal, vertical, diagonal) can be checked in parallel to identify and flip the opponent's disks.
- Board Evaluation: The evaluation of the board (EvaluateBoard) could potentially be parallelized. For instance, evaluating different parts of the board could speed up the process; however, we are using a bit representation of the board which may result in the parallelization overhead outweighing the benefits of parallelism.

2 Results

All results in this section are using the '-O2' optimization.

2.1 Data Races

Cilkscreen is a cilk plus tool that checks for data races by binary rewriting the executable. Running the cilkscreen tool with search depths of 1-7 reported no errors, hence, my program is free of data races.

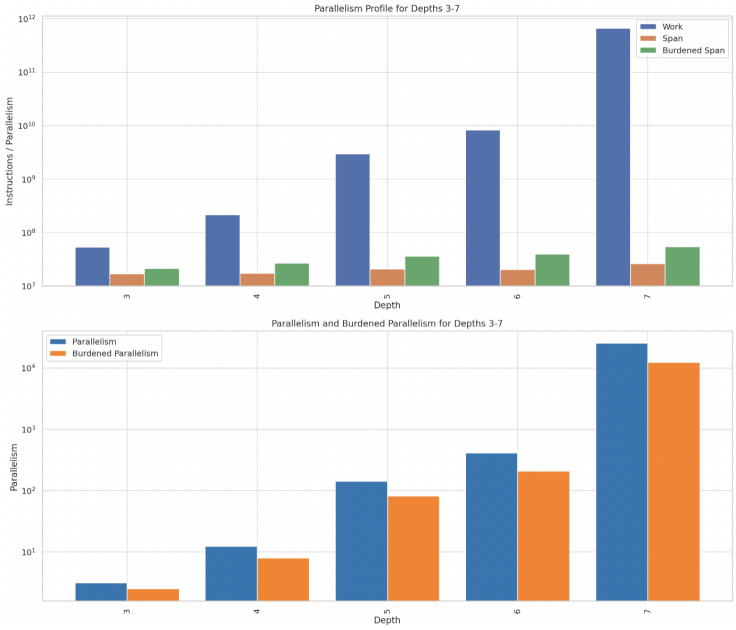

Figure 1: Plot of important parallelism features for depths 3-7

2.2 Cilkview Results

2.2.1 Parallelism Profile

As discussed in section 1.2.2, my implementation only uses parallelism for search depths of 3 and greater. In figure 1, the first graph shows the work (total instructions), span (critical path length), and burdened span (adjusted span considering overheads) for each search depth. The use of a logarithmic scale on the y-axis helps in accommodating the wide range of values. The linear growth we see on the logarithmic scale highlights the exponential increase in work with increasing search depth. The differences between span and burdened span illustrate the impact of overheads on the critical path length. The second graph focuses on the parallelism and burdened parallelism values across different depths. Parallelism quantifies the theoretical maximum speedup with unlimited processors, while burdened parallelism accounts for practical constraints like overheads. The logarithmic scale is again utilized to clearly show the exponential increase in these values, particularly the significant increase at depth 7, which indicates a high potential for parallel execution.

2.2.2 Speedup

This table 3 provides the lower bound of the theoretical speedup estimates over a range of processor counts for each search depth. The estimates illustrate the expected performance improvement with additional processors is subject to

Depth	Work	Span	Burd. Span	Par.	Burd. Par.	Spawns/Syncs	Avg Instr./Strand	Strands/Span	Avg Instr./Span Str	Atomic Instr.	Frame Count
3	53,268,493	16,923,080	21,165,349	3.15	2.52	391	45,373	433	39,083	5,321	782
4	214,388,117	17,316,243	26,891,425	12.38	7.97	2,442	29,260	853	20,300	31,013	5,229
5	2,959,811,687	20,801,645	36,137,395	142.29	81.90	24,995	39,471	1,343	15,488	1,010,326	53,188
6	8,251,738,363	20,210,345	39,675,347	408.29	207.98	83,736	32,848	1,627	12,421	2,144,837	180,208
7	663,827,442,449	26,220,493	53,863,776	25,317.12	12,324.19	3,914,171	56,531	2,393	10,957	291,776,972	8,213,406

Figure 2: Table of parallelism profile for depths 3-7

Depth	2 Processors	4 Processors	8 Processors	16 Processors	32 Processors	64 Processors	128 Processors	256 Processors
3	1.19	1.32	1.4	1.44	1.46	1.47	1.47	1.48
4	1.65	2.44	3.21	3.81	4.2	4.43	4.56	4.62
5	1.9	3.77	6.99	12.2	19.47	27.73	35.2	40.68
6	1.9	3.8	7.57	14.25	25.53	42.25	62.8	83.0
7	1.9	3.8	7.6	15.2	30.4	60.8	121.6	243.2

Figure 3: Table of speedup for depths 3-7

diminishing returns due to overheads and the inherently sequential parts of the othello algorithm. For instance, while the speedup increases as more processors are added, the rate of increase slows down significantly beyond a certain point, highlighting the practical limits of parallel execution efficiency.

2.2.3 Parallel Efficiency

Parallel efficiency is calculates as $S/(p * T(p))$, where S represents the real time of a sequential execution of your program, p is the number of processors and $T(p)$ is the real time of the execution on p processors.

From figure 4 4, we observe several things. As more threads are introduced, efficiency generally decreases, reflecting the overhead and diminishing returns associated with managing and synchronizing additional parallel tasks. This trend is consistent with Amdahl's Law, which states that the speedup of a program using multiple processors in parallel computing is limited by the sequential portion of the program. From 1 to 5 threads, the parallel efficiency is well above 80% which suggests that parallelism is being utilized efficiently. We can see this reflected in figure 5 5 as the runtime significantly decreases from greater than 100 seconds to close to 20 seconds when using 1 to 5 threads. Between 5 to 15 threads, the efficiency values show a relative stabilization, fluctuating within a narrow range. This stabilization might indicate that the program has reached a balance between the computational workload and the overhead of managing multiple threads. However, beyond 15 threads, efficiency declines more steadily, suggesting that the overhead of managing an increasing number of threads out-

weighs the benefits of parallel execution for this particular workload. We can see this reflected in our runtime as we see marginal decreases in runtime with additional workers which is reflected in figure 4 as the runtime only decreases by 2.5 seconds with double the workers. One possible explanation from this could be there is an average of 15 parallel searches that the algorithm is working on at a given time for a computer vs computer game with a search depth of 7 moves.

2.2.4 Quality of Parallelization

The observed efficiency trend4 indicates that while the parallelization does offer performance benefits, especially in the lower thread count range, there is room for further optimization. Effective parallelization would ideally demonstrate higher efficiency levels across a broader range of thread counts. Potential areas for improvement could include optimizing task granularity and adding parallelization for enumerating legal moves or flipping disks.

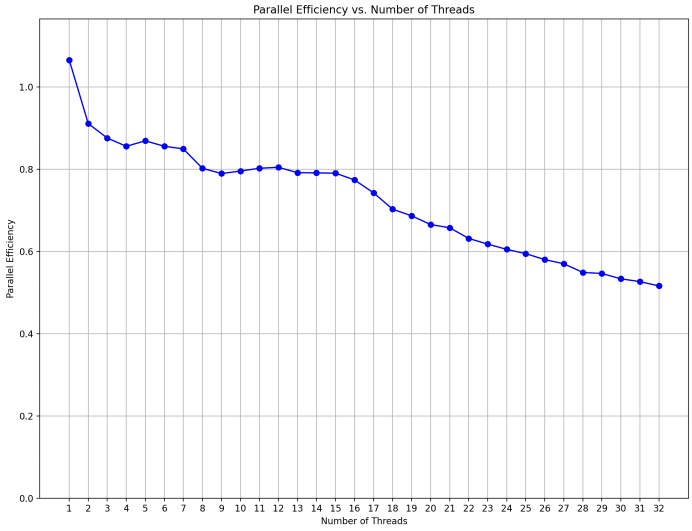

Figure 4: Parallel efficiency for 1-32 cilk workers

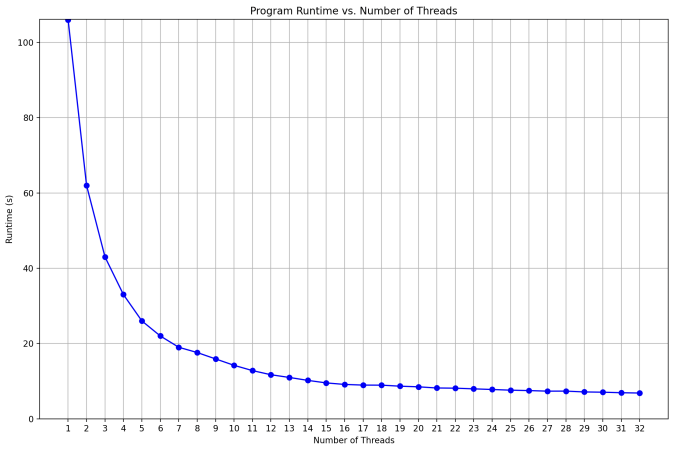

Figure 5: Program runtime for 1-32 cilk workers