Homework 1 Othello Writeup

Gyuheon Oh

February 2024

1 Introduction

As the efficiency of compute closely follows Moore's law, it is becoming increasingly important to be able to correctly exploit and harness parallelism in porgrams. This homework assignmen, centered on the strategic board game Othello, provided a unique opportunity to explore parallel program design through the development of a shared-memory parallel program using Cilk Plus. The primary objective was to engineer a computer player capable of playing Othello by analyzing future moves to determine the optimal strategy.

The starting point was a sequential program that allowed two human players to engage in the game, which served as a foundation for developing the parallel solution. The task involved transforming this into a shared-memory parallel application that could efficiently analyze multiple future game states simultaneously to determine the best move for the computer player.

This report will outline the steps taken to transform the sequential two human player baseline program, into a parallel program that allowed for the option of up to two computer players. This report will then present the runtime, efficiency, and parallelism data, and analyze the effectiveness of the improved program.

2 Code Design

In order to implement a parallel program that allows two computers to play against each other with variable lookahead depth, we needed to implement two functions, *Negamax*, which implements the Negamax algorithm that we use to score different candiate moves for a certain lookahead depth, and *FindBestMove*, which uses *Negamax* to choose the best possible move to play. Because Othello is a zero sum two person game, we choose the Negamax algorithm as the basis of the algorithm is that the value of a certain move to Player 1 is the negation of that value to Player 2. Therefore, Negamax recursively takes turn for Player 1 and Player 2, attempting to maxmimize the score for Player 1 and minimize the score for Player 2. Here is the pseudocode for my implementation of Negamax.

```
int Negamax(Board board, int depth, int color) {
  if (depth == 0) {
      return the board score for ‘color‘;
  }
  Board legal_moves = GetLegalMoves(color);
  if (legal_moves has no legal moves)) {
    if (GetLegalMoves(OtherColor(color) has no legal moves) {
        return the board score for ‘color‘;
    }
    return -Negamax(board, depth, OtherColor(color));
  }
  int best_value = -64;
  for (each square in legal_moves) {
        if (square is a legal move) {
            Board new_board = board;
            PerformMove(move, new_board);
            int score = -Negamax(new_board, depth - 1, OtherColor(color));
            if (score > best_value) {
                best_val = eval;
            }
        }
    }
  return best_value;
}
```

This is a pretty standard implementation of a Negamax algorithm. For each potential move, we recursively call Negamax with the color swapped to the other color and negate it, finding the best score for the original color that it was called for at the top of the recursion tree.

However, we can see that this algorithm's running time will explode exponentially with depth. If depth is 1, we need to consider 64 moves. If depth is 2, we need to consider 64^2 moves. We need parallelism to tackle this problem.

As explained above, Negamax algorithm is the bulk of the weight of this program. Therefore, we will exploit areas in the runtime of this program where we can parallelize work. One glaring area we can parallize work is in the nested for loops of Negamax, where we iterate over the board in search for potential moves. By using Cilk for instead of usual for loops, we can spawn new processes across the loop, parallelizing our search. Here is a code snippet displaying this.

```
    int max_value;
    cilk_for (int row = 1; row <= 8; ++row) {
        cilk_for (int col = 1; col <= 8; ++col) {
            // Recursively call negamax on these moves to check the
            // the score for that move and update ‘max_value‘
        }
    }
```

This effectively splits up the work across multiple processes in order to speed up our calculations. However, there is an issue here. Multiple processes are accessing the shared memory value, ‘max_value‘. This causes data races, where when multiple processes, attempt to read/write to a shared memory address, leading to non deterministic and unpredictable results. In order to fix this, we use reducers, more specifically, ‘cilk::reducer_max¡int¿‘. Reducers allow for separate processes to keep a thread local copy of the variable, and ‘reduce‘ at the end, avoiding data races and avoiding serialization and deadlocking that comes with lock based concurrency. Here is a code snippet displaying this.

```
cilk::reducer_max<int> max_value(best_value);
cilk_for (int row = 1; row <= 8; ++row) {
    cilk_for (int col = 1; col <= 8; ++col) {
        int eval = -Negamax(new_board, depth - 1, OTHERCOLOR(color));
        max_value.calc_max(eval);
        }
    }
}
return max_value.get_value();
```

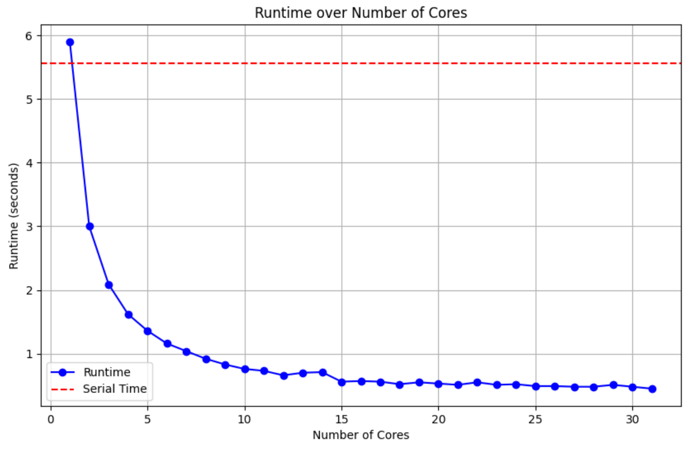
We come across another issue. As Negamax is called for all potentially 64 turns in the game, and recurses ‘depth‘ number of times, with the current implementation, a massive amount of processes are spawned, each bringing spawning and syncing overhead that comes with parallelization. Therefore, it in in our best interest that we do not spawn threads that will not do a lot of work; in other words, we want our spawned parallel threads to compensate for the extra overhead that they bring. In order to tackle this problem, we set thresholds on when we decide to perform Negamax in parallel, and when we want to perform Negamax sequentially. Here is a code snippet illustrating that.

```
    int Negamax(Board board, int depth, int color) {


    if (depth < 3 || num_moves < 5) {

      return SequentialNegamax(board, depth, legal_moves, color);

    } else {

      return ParallelNegamax(board, depth, legal_moves, color);

    }

}
```

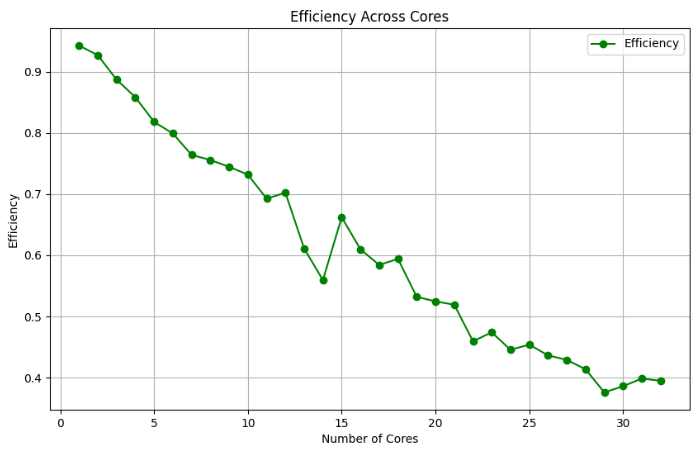
Here, if we are near the bottom of our recursion tree, that is, if our depth is less than 3, or if the number of legal moves to try is less than 5, we call a sequential version of Negamax. Else, we call the Parallel version of Negamax.

This allows us to avoid spawning processes and taking on runtime debt for processes that won't pull their weight.

3 Result Analysis



As we can see from this graphic, we can see that the runtime logarithmically decreases as the number of cores increases. This is a good signal that our implementation of parallelization has a positive impact on the runtime. This behavior can be explained by a combination of factors related to the nature of the task but the factor that is the most suspiscious to me is Amdahl's Law. Amdahl's Law states that the maximum improvement to a system's performance using parallel processing is limited by the portion of the system that must be executed serially. Even if I increase the number of cores, the serial portion of the the Othello process remains a bottleneck, leading to diminishing returns on runtime improvement as more cores are added. Parallel overhead may also be a factor. As mentioned before, parallel computing involves overheads such as communication between cores, synchronization (waiting for all threads to reach a barrier before proceeding), and data distribution and gathering. As the number of cores increases, these overheads can become more significant, reducing the net gain from parallel execution.

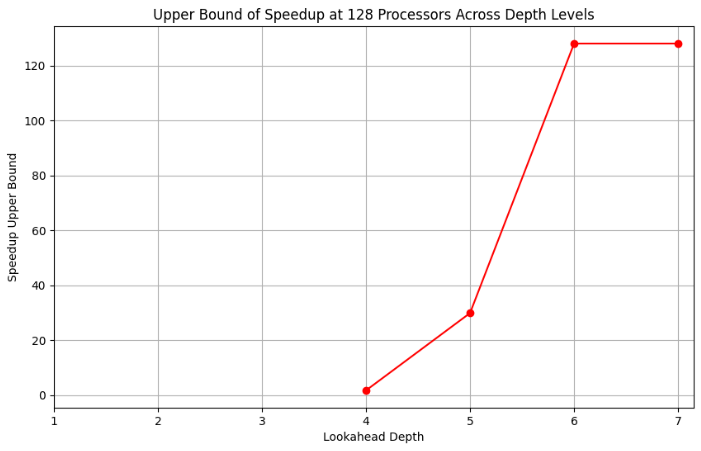


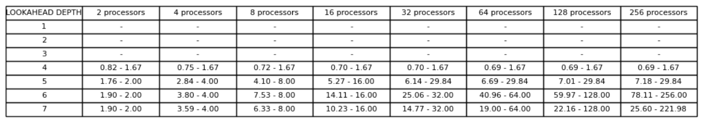
This figure plots the parallel efficiency for our program at lookahead depth of 7 as we increase the number of cores. Parallel efficiency is measures as such,

$$PE = \frac{T_1}{T_p * p}$$

Where T_1 is the runtime on one processor, T_p is the runtime on p processors, and p is the number of processors. The parallel efficiency decreases as we add cores, because we have less and less net gain from increasing cores for the reasons discussed above.

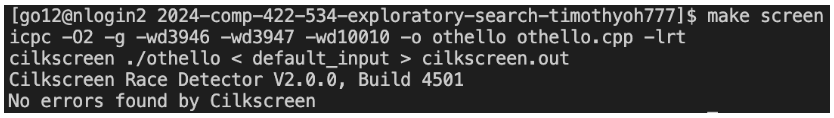
Here are some results from Cilkview and Cilkscreen:





LOOKAHEAD DEPTH	2 processors	4 processors	8 processors	16 processors	32 processors	64 processors	128 processors	256 processors
1	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-
4	0.82 - 1.67	0.75 - 1.67	0.72 - 1.67	0.70 - 1.67	0.70 - 1.67	0.69 - 1.67	0.69 - 1.67	0.69 - 1.67
5	1.76 - 2.00	2.84 - 4.00	4.10 - 8.00	5.27 - 16.00	6.14 - 29.84	6.69 - 29.84	7.01 - 29.84	7.18 - 29.84
6	1.90 - 2.00	3.80 - 4.00	7.53 - 8.00	14.11 - 16.00	25.06 - 32.00	40.96 - 64.00	59.97 - 128.00	78.11 - 256.00
7	1.90 - 2.00	3.59 - 4.00	6.33 - 8.00	10.23 - 16.00	14.77 - 32.00	19.00 - 64.00	22.16 - 128.00	25.60 - 221.98

We can also see that we have synced and correctly made use of reducers, avoiding any data races.



```
[go12@nlogin2 2024-comp-422-534-exploratory-search-timothyoh777]$ make screen
icpc -O2 -g -wd3946 -wd3947 -wd10010 -o othello othello.cpp -lrt
cilkscreen ./othello < default_input > cilkscreen.out
Cilkscreen Race Detector V2.0.0, Build 4501
No errors found by Cilkscreen
```

4 Other Thoughts and Considerations

One area that was under consideration for parallelization was EnumerateLegalMoves. This function iterates through the 8x8 board, checking if that spot is a legal move. We thought this could be easily parallelized, as this was a matrix operation processing rows and columns. However, this demanded much more engineering effort than expected, as there needed to be a way to sync up marking a singular 'Board' object, as this needed to be returned for use in other areas of the program. We chose an approach where instead of each thread marking a spot on the board, which was prone to data races, each thread would store and return an array of legal positions it found, and we would, in serial, create a board with legal moves marked at the end after syncing all the parallel threads. However, the runtime of this was much more significant than the trivial, sequential approach, so this was abandoned.

5 Conclusion

In conclusion, this assignment allowed me to explore implementing parallelism in a non trivial program. I was able to explore the trade-offs between utilizing parallelism and selecting a sequential approach. In the end, I was able to achieve 60% parallel efficiency on 16 cores at a lookahead depth of 7, which signals that my implementation had at least some, non trivial benefits to the overall performance of the program.