15-418 Final Project Report

Jason Huang (jasonh1)

Abel Tesfaye (atesfaye)

SUMMARY: We parallelized different chess AI algorithms using CILK, and benchmarked the speed-up against the sequential version of minimax and negamax. Additionally, we have also implemented an interactive game where other students and professors can play against our chess AI.

BACKGROUND: We focused on two specific AI algorithms for our project, namely minimax and negamax. These are common algorithms that involve searching through every single possible moves that each player could make for the next *n*-moves, and based on a heuristic that I designed through online research about chess rules, the algorithm would pick out the best move the current player can make. Each piece has a certain value assigned to it, and a board state in check or checkmate will greatly increase the heuristic value of the player not in check. Of course, generating all possible moves for the next *n*-moves will take a lot of computation time since there can be so many possible moves (each player can have up to sixteen pieces), we parallelized the moves generator, since each move can be made independently.

For the bare bones of our chess game, we decided to implement the game from scratch because we thought it would be easier to parallelize our own code rather than an existing baseline code. This also meant that we could make optimizations to algorithms that were specific to our implementation of the chess game. We created three classes, Piece, Player, and BoardState, and below is a detailed explanation of what each class consisted of and how it contributed to the implementation.

The “Piece” class consists of six values:

- player: keeps track of which player owns this piece
- row: keeps track of which row this piece is on the board
- col: keeps track of which col this piece is on the board
- piece: the letter representing this specific piece, used for printing the board
- first: keeps track of whether the piece has moved yet (this is used only for rooks, kings, and pawns; pawns can move forward two spaces only on their first move, and rooks and kings can castle only if they have never been moved)
- value: a heuristic value assigned to the piece that is useful for chess AI algorithms

Of course, each piece has different valid moves, heuristic value, row/col, etc. Therefore, I have created six subclasses (Pawn, Knight, Bishop, Rook, Queen, King), each subclass inheriting the basic functionality of the Piece class, but with their own constructors and valid moves.

The Piece class itself also has a few all-encompassing functions, such as initializing a Piece on the board (put_on_board), moving a Piece from its current position to a new position on the board (move_on_board), and determining whether said Piece can attack the King for check detection (put_on_check_board). Additionally, Piece has a virtual function, find_legal_moves, that is specified within each subclass, and a deepcopy function that creates a new class with the same values but ensures that there is no aliasing.

The “Player” class consists of four values:

- player: player number (0 for white, 1 for black)
- king_row: keeps track of which row the king is on the board
- king_col: keeps track of which column the king is on the board
- pieces: keeps track of all the pieces that this player still has on the board; starts off with sixteen pieces (eight pawns, two rooks, two knights, two bishops, one queen, one king)

The Player class has a deepcopy function that creates a new class with the same values but ensures that there is no aliasing.

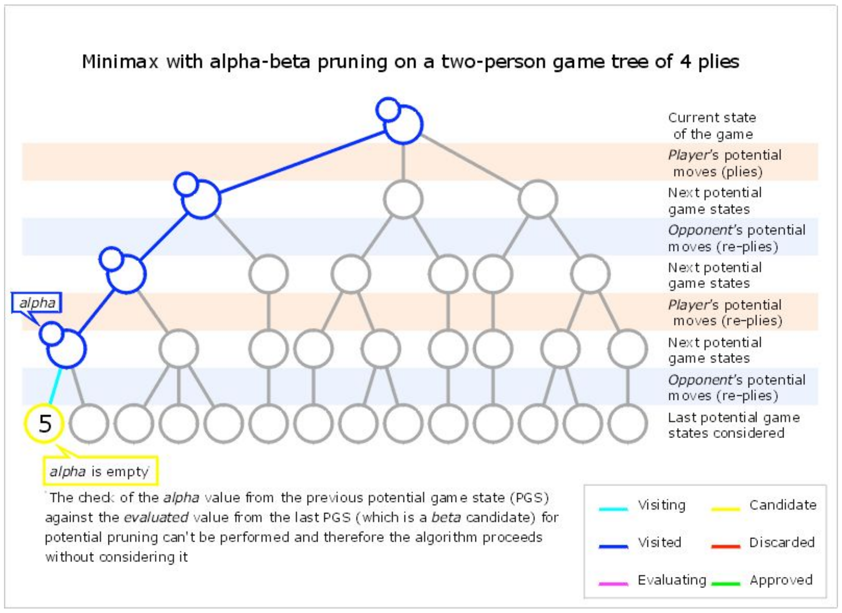
The “BoardState” class consists of three values:

- curr_player: the player that is currently in play
- players: keeps track of both Player classes and all their information
- Board: 2-D board that represents the chessboard

The BoardState class has a deepcopy function that creates a new class with the same values but ensures that there is no aliasing. In terms of other functions, it has a check detection function. We initially also had a checkmate detection, but realized that it could easily be detected by whether or not a certain board state had any valid moves. If not, it was in checkmate. We also included a function called castle, that specifically handled castling moves. Within the BoardState class, we are also able to find all legal moves from all pieces of the current player by calling find_legal_moves.

One major dependence the code has is determining the best move. No matter now we calculate the value for a single move, we have to wait until all values are computed before we can determine the best move. All the moves must be completed and this poses a barrier for our algorithm's control flow. It is data parallel because the moves can be considered independently and then compared at the end. There is not much cache locality because chess moves can access indices that are not near each other. It is not amenable to SIMD execution because we have multiple instructions operating on multiple data.

APPROACH:


Minimax with alpha-beta pruning on a two-person game tree of 4 plies
Current state of the game
Player's potential moves (plies)
Next potential game states
Opponent's potential moves (re-plies)
Next potential game states
Player's potential moves (re-plies)
Next potential game states
Opponent's potential moves (re-plies)
Last potential game states considered
alpha
5
alpha is empty
The check of the *alpha* value from the previous potential game state (PGS) against the *evaluated* value from the last PGS (which is a *beta* candidate) for potential pruning can't be performed and therefore the algorithm proceeds without considering it
Visiting
Candidate
Visited
Discarded
Evaluating
Approved

We parallelized minimax and negamax with the CILK API on C++. We targeted the latedays cluster on the Andrew network. We mapped the moves the the threads on the cluster machine.

Each thread will calculate the value of its move by attempting the move and then using recursion. If we had more moves than threads, the CILK scheduler will queue the moves. The CILK scheduler also handles work stealing to improve the allocation of work through dynamic scheduling. After we implemented the sequential chess game with the alphabeta and minimax algorithms, we attempted to parallelize both of the algorithms. We spent a lot of time trying to make alphabeta parallel, but this algorithm is inherently sequential. We struggled to improve the performance because a lot of computation time was spent communicating data between the threads. With alphabeta, we need to keep track of alpha and beta and allow each thread to properly update them. This fact made parallelizing the algorithm very challenging. The performance our attempt was so meager that we ended up scrapping our alphabeta implementation and replacing it with minmax.

Minmax and negamax have the potential to benefit from parallelism because each move can be computed independently of each other. However, after both algorithms compute the value , they need to communicate the value and find the min or max. To do this, we created an array of ints for each move and each thread would modify the index of the move it computed. Once all the threads finished, we used a reducer to find the max or min of the array we allocated. All of are code was programmed from scratch and CILK was that only outside library we used.

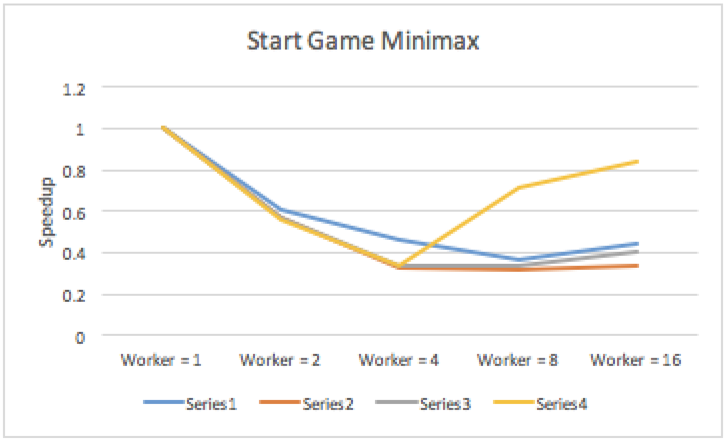
RESULTS:

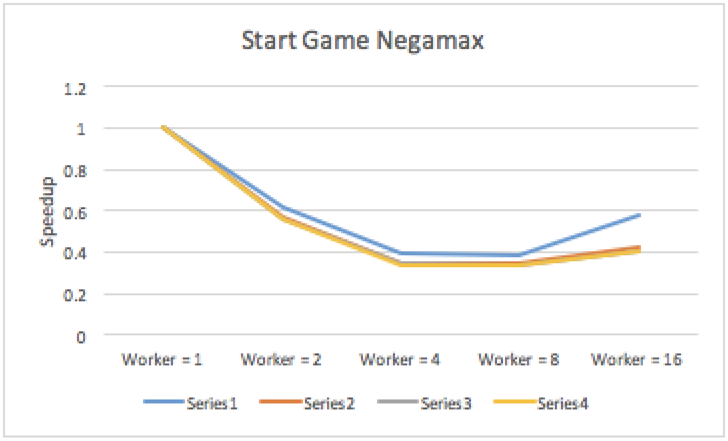
We were mostly successful at achieving our goals at parallelizing the chess AI algorithms. We measured the performance of our code by calculating the speedup of the parallel algorithm vs the sequential algorithm. We tested our code on depths with size 2, 3, 4 and 5. Our worker thread count was 1, 2, 4 and 8. We tested the time for the AI to find the move on 3 different preselected board states. The board states modeled early, middle and late game chess states. We generated the requests by running our program on the latedays machine on the Xeon Phi. Our baseline is the parallel code running with only 1 thread. The problem size in our analysis was the depth of the algorithm search. The number of possible game states scales exponentially with depth and this was origin of all of the computation. Exploring greater depths produces much more work.

We are not very sure what limited the speedup of our program. Our workload seems to be very parallel because of the independent nature of the work. However, our speedup decreases once we have more than 8 threads. This is somewhat confusing since the amount of communication our program does scales with the number of moves not the number of threads. We speculate that this might be happening because of the work stealing nature of cilk. We believe that with many threads, work stealing causes the program to execute in a non optimal order.

We are unable to break our parallel algorithm into multiple parts since we are measuring the amount of time it takes the AI to find the optimal move. There does not seem to be a smaller portion we can break the time measurement into.

We chose the CPU as our target and believe that it is a better choice than the GPU for our workload. GPUs excel at data parallel tasks where each task is easy to compute and there are many little tasks to compute. The independent tasks in our AI algorithm are large and would be better computed by CPUs. A single unit must consider all of the possible moves after making some given move.





Series1 is depth 2. Series2 is depth 4. Series3 is depth 8. Series4 is depth 16

REFERENCES:

- https://www.youtube.com/watch?v=6ib1Kf44KR0
- https://www.youtube.com/watch?v=l-hh51ncgDI

DISTRIBUTION OF TOTAL CREDIT:

Jason Huang (jasonh1):

- Wrote the baseline code from scratch, created classes for Piece, Player, and BoardState, along with a main function that served as a way to test correctness for our algorithms

- Implemented a function that found all the legal moves of a given BoardState and stored it in a vector; parallelized using CILK
- Wrote sequential version of minimax and negamax algorithms, and tested it for correctness
- Created the Makefile to compile sequential code
- Provided pseudocode for how to parallelize each algorithm
- Debugged the parallel versions of each algorithm to ensure correctness
- Implemented part of the interactive chess game

Abel Tesfaye (atesfaye):

- Made improvements to the baseline code, and made the code more C++ friendly (replacing malloc with new, using delete instead of free)
- Discovered negamax algorithm and provided pseudocode for the algorithm
- Updated the Makefile to run parallel code
- Implemented more obscure rules of chess, such as castling and pawn promotion
- Updated the parallel code for finding all moves given a BoardState, synchronizing properly and improving performance slightly
- Fully parallelized the minimax and negamax algorithms using CILK
- Came up with the decision to use a shared variable to keep track of the minimum and the maximum of each state for the implementation
- Implemented part of the interactive chess game

Based on the work performed by each partner, it is reasonable to say that both partners put in equal amount of work on the project. Jason Huang should receive 50% of the credit, and Abel Tesfaye should receive 50% of the credit.