Title: Parallel Chess AI Engine

Team Members: Priyanshi Garg, Teddy Liang

Summary
Our project focuses on the development of a parallel chess AI, leveraging the power of C++ and the OpenMP framework to accelerate the decision-making process in chess. The core objective is to significantly reduce the time required for move selection by distributing computations across multiple threads.

URL: https://teddyliang.github.io/15418-final-project/

Background

If we want a chess engine to evaluate the best move by looking 3 or 4 moves deeper into the future, the number of possible board states grows exponentially as the depth increases. The heart of our chess AI lies in its ability to explore a vast array of potential moves and their resulting positions, a task managed by the combination of the minimax algorithm and alpha-beta pruning techniques. The minimax algorithm serves as the foundation, guiding the AI in simulating moves and counter-moves to deduce the most advantageous path. Alpha-beta pruning further refines this process, eliminating less promising branches early on, thereby reducing the computational load and focusing resources on more critical analyses.

The main opportunity for parallelism is in the search for the best move. The game tree, representing potential moves and their outcomes, is inherently expansive. Our approach involves dissecting this tree into multiple branches at the root level, each assigned to a separate thread for exploration. This division allows simultaneous traversal of different parts of the game tree, significantly enhancing the breadth and depth of the search within the same computational time frame. By parallelizing the search, we dramatically increase the number of positions analyzed within a given time, directly contributing to more strategic and informed decision-making by the AI. This can significantly reduce the time needed to make an informed decision. It can also allow the chess AI to look farther into the future in the same amount of time compared to a sequential version, allowing it to choose better moves.

Upon reaching leaf nodes or positions requiring evaluation, the computation of heuristic values for these positions presents another opportunity for parallelism. Given that each position's evaluation is independent, we can allocate subsets of these positions to different threads, thereby accelerating the cumulative assessment process.

The Challenge
Parallelizing a chess AI poses several challenges that stem from both the nature of the game and the intricacies of parallel computing.

One challenge is that the minimax algorithm with alpha-beta pruning relies on a tree structure where each node (board position) depends on the evaluation of its child nodes. This inherent dependency complicates parallelization because the value of a parent node cannot be determined until all its children are evaluated, introducing synchronization points that can limit parallel efficiency. Furthermore, different branches of the game tree can have vastly different complexities and depths, leading to divergent execution paths among parallel threads. Some threads might quickly reach a pruning point or a shallow depth, while others might need to explore deeper, more complex branches, resulting in workload imbalance. This leads to challenges with mapping the workload effectively so that threads have similar amounts of work.

Chess AI involves frequent access to the game state, move history, and potentially transposition tables, which is caching of previously evaluated positions. Ensuring memory locality can be challenging but crucial for performance, especially given the potential for a high volume of random memory accesses. The need for threads to synchronize at various points, especially when using alpha-beta pruning, can lead to a high communication-to-computation ratio. Efficiently managing this ratio is critical to ensuring that the overhead of communication does not outweigh the benefits of parallel computation.

Resources

We will be building our chess AI system from scratch. We plan on using the ghc cluster machines, using varying numbers of cores to test our system. One resource we will use the wikipedia page about alpha-beta pruning: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. We may also want to use machines with thread counts higher than 8 like the PSC machines in order to have a better understanding of how well the parallelism works.

Goals and Deliverables

Plan to Achieve:
- Develop a chess AI using OpenMP and C++ that improves move selection speed by at least 4x compared to a sequential version on 8 cores. This should be achievable since the ideal for 8 cores is 8x speedup
- Ensure the AI uses a parallel version of the minimax algorithm with alpha-beta pruning
- Use standard chess engine test suites to confirm the AI follows chess rules and makes logical moves.

If work goes more slowly, we aim to still have a correct parallel chess AI that can achieve at least 2x speedup.

Hope to Achieve:
- Implementing a parallel board evaluation function, if exploration of it seems good
- Achieve 6x speedup on 8 cores with optimizations to the parallelism
- Incorporate further techniques for more parallelism and/or reduced search space
- Achieve at least 10x speedup with higher core counts

Demo Plan for Poster Session:
- Allow attendees to play against the AI, showcasing real-time decision-making:
- Display speedup graphs comparing the parallel AI to its sequential counterpart.
- Show how the parallel algorithm navigates the game tree.

Analysis Goals:
- Evaluate the scalability of the parallel AI with increasing cores and identify bottlenecks.
- Document the journey of parallelization and explain how we iterated on our code to improve parallelism
- Use chess engine metrics to assess how parallelization and heuristics impact the AI's gameplay strength.

Platform Choice
C++ offers high-performance computing capabilities important for the intensive calculations of a chess AI. OpenMP works well for our project since we can use it to distribute branches to different cores.

Schedule
Week 1:
- Implement basic chess engine components (board setup, piece movements).
- Develop serial versions of the minimax algorithm and alpha-beta pruning. (May go into week 2)

Week 2:
- Implement evaluation function for board positions.
- Parallelize the search algorithm using OpenMP

Week 3:
- Optimize the parallel search for load balancing and efficiency.
- Explore parallelizing the evaluation function
- If have time, explore further techniques to improve parallelism or reduce search space

Week 4: Testing, Debugging, and Final Adjustments
- Conduct thorough testing, and identify and fix bugs
- Final performance optimizations and prepare project documentation.