Greg Loose

15-418 Final Project Milestone Report: Chess Engine

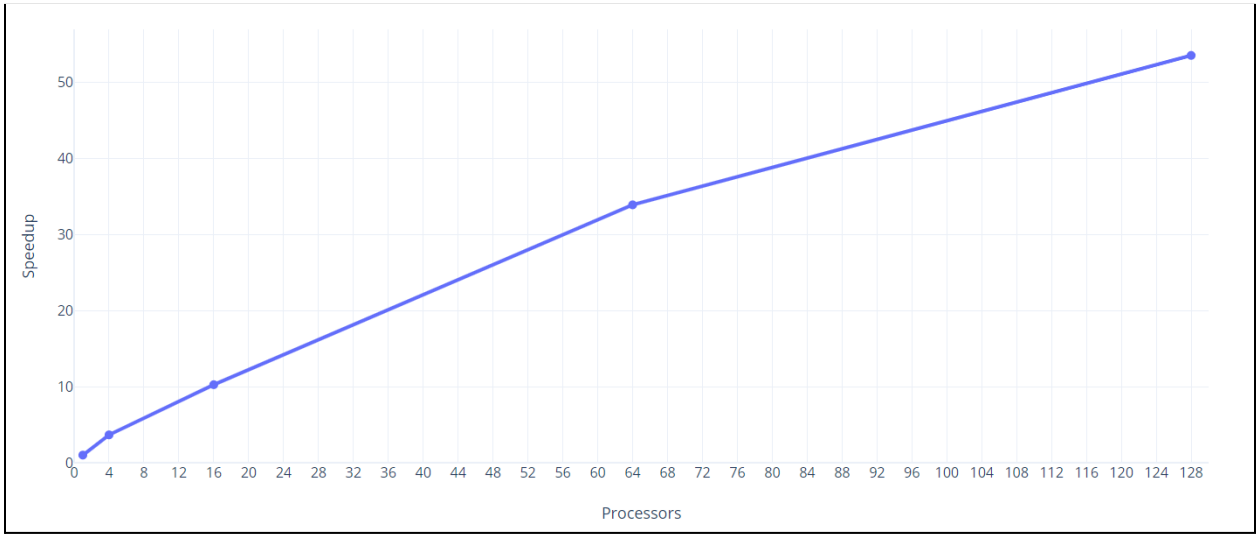
Since I was starting from scratch, my first priority was to implement the rules of chess in standard, sequential C++ code. This went fairly smoothly; currently, I have a "Board" class which represents the current state of the game board as an array of "Piece" objects, with methods in place for finding the moves available to each piece and determining whether they are legal, in the sense that they follow movement rules and don't leave the moving player in check. The user interface is quite minimal at the moment and not particularly user-friendly: the program accepts an initial board state from a plain text file, where the first line indicates which player is to move and the following 8 lines have 8 characters each indicating which piece is on the corresponding square, with white space indicating an empty square. You can call the program as follows:

```
./Board -f input.txt -d 6
```

This indicates that the program should read the board state from input.txt and find the best move with a look-ahead depth of 6 moves (3 moves for each player). It will then evaluate every legal move recursively as the greatest material score after 6 moves, with the other player also choosing its best move with depth d-1, etc. After finding the best move, it prints it to the console and gives the user an opportunity to play that move (i.e. in an online chess game or puzzle), then input the move made by their opponent, at which point the process repeats until the program is killed. I have had good results using this method for unranked online chess puzzles, where the necessary depth is generally no higher than 6.

I mentioned in my project proposal that, while I was leaning toward an OpenMP implementation at the time of writing, I was unable to properly evaluate MPI as a possible alternative since I hadn't started assignment 4 yet and so was not at all familiar with it. After completing the assignment, and writing my sequential implementation for the project, I decided that MPI would actually be better. My initial parallel implementation assigned each top-level move to a different processor to be evaluated independently, then did a reduction over all processors to get the move with the best score. Assuming there are more legal moves than processors, each processor is assigned multiple moves, distributed evenly between them. The moves are evaluated by applying them to the current board state and then choosing the best counter-move for the new board state; thus each processor must have its own separate address space for the game board, which is why I settled on MPI. This initial implementation actually produced fairly high speedup, but of course had no benefit for additional processors beyond the number of top-level moves (usually somewhere around 30). Thus, to get the full benefit from up to 128 processors, I redesigned my implementation to evenly split the processors among top-level moves and create a new MPI communication channel for each move, at which point the process is repeated recursively for each second-level move. This was tricky to implement, but I successfully managed to get

speedup improvements all the way up to 128 processors for high depth (for lower depth, 128 processors actually slowed down performance). To get a measure of my results for a particular use case, I chose a random puzzle from chesspuzzle.net and ran it through my program at depth 6 on the PSC machines, recording the computation time and calculating speedup, resulting in the following plot:



I was able to get about 54x speedup in this case, with a computation time of about 286 seconds sequentially and 5.3 seconds with 128 processors in parallel.

I am still on track with my goals for the project at this point in time, and I believe my early results are quite respectable, but I don't think my engine is ready to play full chess games yet. There remains work to be done in both performance and chess logic. First, even at 128 processors, 6 moves is the current limit for lookahead depth in any reasonable time frame; when you move it up to 7, it takes over two minutes to finish running. There are many resources online for performance optimizations to chess algorithms, such as this website which I found helpful in my preliminary research. Among these, I would like to at least implement alpha-beta pruning and evaluate the performance gains. It is possible that this optimization will have a negative impact on my parallel speedup; one immediate concern is that it will lead to workload imbalance if some moves end up doing less work due to heavy pruning. This may or may not necessitate changes to my parallelization strategy, which is why this optimization is currently my first priority. I plan to implement alpha-beta pruning this week (4/11).

On the chess logic side, there are some remaining flaws in my implementation as well. First, I put off implementing some of the more advanced game mechanics (castling, promotion, en passant) because my priority was to get a working implementation capable of solving simple chess puzzles, and these mechanics do not often come up in this setting. Second, I am currently evaluating moves purely based on material score. This will not have very good results in the

opening of a real game, which is more about positioning than taking pieces. The linked website has some suggestions for alternative metrics to use, such as the number of pieces threatened or protected by your own pieces. I would like to implement these two metrics (or just one, if I am behind schedule) and see how they affect the engine's play in the opening of a game, and its chances at winning against online chess bots. This will be my goal for next week (4/18).

For the poster session, I still feel that a live demo against an online bot opponent or a series of chess puzzles would be a good showcase of my work. Which of these I choose will depend on how well my engine performs in real games (based on the aforementioned chess logic improvements) and how quickly I can get through a game (based on my performance improvements); if I can't consistently win bot matches or they take too long for the five-minute presentation, I will stick to puzzles, with perhaps a brief demonstration of the opening moves of a real game. I will also include graphs (like the one above) and discuss the performance of my parallelization strategies, as well as how my speedup is affected by my performance optimizations (i.e. alpha-beta pruning).

I don't have any serious concerns with my plans moving forward; however, if the TA reading this has any recommendations for simple C++ graphics libraries supported by the PSC machines, please let me know, as improving my user interface would be one way to take my project to the next level if I have extra time.