Parallel CHESS AI

A PREPRINT

Ishaq Yousef Haj-Hasan*
Computer Science Senior
Carnegie Mellon University
Pittsburgh, PA 15217
ihajhasa@andrew.cmu.edu

Sameer Ahmad
Computer Science Senior
Carnegie Mellon University
Pittsburgh, PA 15217
sjahmad@andrew.cmu.edu

December 10, 2019

ABSTRACT

In the following report, we focus on parallelizing aspects of chess programming that help us achieve better speed ups for the AI implementation as compared to sequential versions of search algorithms that already exist. Chess programming is one of the most essential features for testing the of advantages of parallelism since there are so many aspects of chess that allow for run time optimizations due to parallelism which makes such implementations perfect to showcase the abilities of our project. Moreover, some aspects of implementations of chess programming are limited in their compute scalability which allows us to derive the limitations of parallelism. We will discuss the various modules that we incorporated in our implementations and the use of the openmp interface for C++ to create a similar parallel implementation. In our results section, we will highlight aspects of our code that we achieved high levels of parallelism. On top of this, we will discuss reasons as to why we chose the openmp interface to achieve parallelism.

1 Background

One of the pinnacles of the advancements in Computer Chess Programming was the invention of the Deep Blue supercomputer by IBM. On February 10^{th} 1996, Deep Blue defeated a world grand master in chess, namely Gary Kasparov. From this moment onwards, it was evident that the realm of artificial intelligence had received major breakthroughs through the means of chess programming. Chess Programming has been a historical means to assess the capabilities and functionalities of hardware components of a computer. In addition to this, there are countless different avenues that can be explored when examining the possibilities for incorporating advanced search and evaluation heuristics that fundamentally challenge the software capabilities that it is linked to. Due to this very reason, we chose to base our final project around parallelizing chess implementations. We will focus briefly on the foundations of a chess implementation and will delve into various parallelizing approaches that we used to achieve significant speedups in the various modules of our chess implementation.

A computer chess program is derived on four main modules that meld together to create a working chess implementation. These modules are board representations, search space traversal, evaluation and move generation. For this project, we decided to shift the focus of our parallelism from the evaluation and representation modules and we explored the various possibilities or incorporating parallelism into the search space traversal and move generation modules due to their high factors of parallelism. We will also demonstrate the capabilities of our parallelism on our partial boards to investigate the extent of our performance speedups.

*You can view the Project Github on the following LINK

2 Approach

```
Makefile ..... compile tests, timing script, game
src/ ..... contains source code
    board_rep.h ..... chess board interface
    gen_moves.h ..... move generation module
    minimax.h ..... sequential minimax chess AI
    minimax_openmp.h ..... parallel (using openmp) minimax chess AI
    timing.h ..... timing class used for testing only
    board_rep.cpp ..... 2D array implementation of chess board
    gen_moves.cpp ..... sequential/parallel move generation implementation
    minimax.cpp ..... sequential minimax implementation
    minimax_openmp.cpp ..... parallel (using openmp) minimax implementation
    Test Files/ ..... scripts to test rigidity of program
        game.cpp ..... run game session (user/AI vs userAI)
        main.cpp ..... test synonymous MiniMax implementation for 1 move
        test_board.cpp ..... test board implementation
        test_gen_moves.cpp ..... test move generation module
        test_minimax_openmp.cpp ..... test parallel minimax implementation
        timing_ai.cpp ..... timing script for generating results
```

Figure 1: Directory Tree of Github Repository

We will describe the functionality of our implementations based on the different modules we've incorporated. The foundations of our implementation are illustrated in the below diagram:

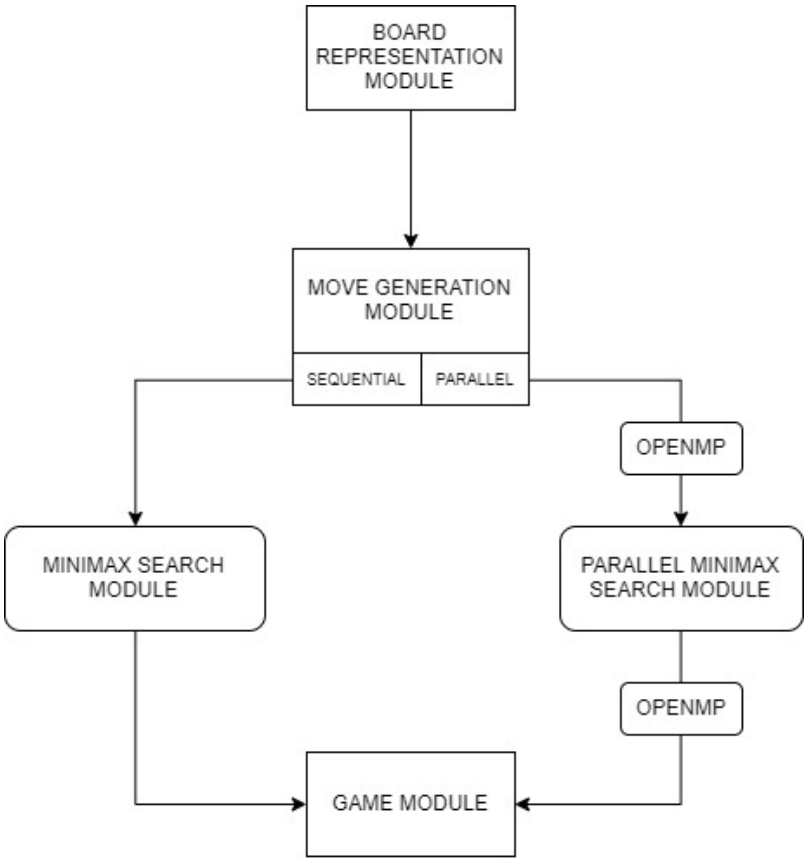


Figure 2: Module Hierarchy of Parallel Chess AI Implementation

2.1 BOARD REPRESENTATION MODULE

The first and most basic module that our implementation is entirely wrapped around is the board representation module. To represent the chess board, we created a **ChessBoard** class that captures all the essential information about the chessboard. First and foremost, each piece is represented by a structure called *Piece*. This structures stores information about the color of the piece and its type (both as integer values to make comparisons and more vigorous tasks easier). Next. our board is represented by a 2-Dimensional array of Dimensions 8 by 8. This is an implementation that we found to be most easy to base our functionality around. In addition to this, we found it necessary to store an additional array of White and Black Pieces that are remaining in the game. Our **ChessBoard** class allows us to copy the board representation, execute a move on an existing board and lookup any cell of the board to see if it contains a board.

2.2 MOVE GENERATION MODULE

The move generation module is designed to sweep through the entire board to analyze the possible moves that are available to perform for any given color. The function *generate_all_next_moves* makes use of multiple helper functions that allows it to sweep the board and discover all legal moves per type of piece. The reason this is so is to enhance the limits of parallelism that can be achieved as generating all legal moves per piece can be achieved in parallel. This was one of the avenues we used to incorporate parallelism into our project. We also knew that sweeping the board could take place in parallel since all cells of the board can be examined separately. Each function sweeps the board to see if any valid spaces are available (not blocked by existing pieces of the same color) to move to pertaining to that particular type.

2.3 MINI-MAX SEARCH MODULE

In this module we implement a class **MiniMax** that our implementation will use to compute optimal moves for the AI. The class has one main function that takes a **ChessBoard** class object and a given color pertaining to whose turn it is at that instant and produces an optimal move that the player in turn should play. As stated, the **MiniMax** class makes use of a mini-max search algorithm to scan the search space and compute the most optimal move. This means that, given a certain depth d, to compute our optimal move our algorithm will search all possible paths that can be created up until d moves have elapsed and pick a move that follows the most optimal path on the basis that the color that is computing the move will maximize his evaluation for every turn and the color that is opposing will try to minimize the current players evaluation by picking the move least favorable to it. Our minimax implementation had potential possibility for parallelism since the search algorthm uses DFS to traverse the search space. Due to this, after we generate the move list, we know that we need to skim each move in this move list to find the most optimal path for the computer to take. Therefore, we this very loop was a means for us to achieve parallelism.

2.4 GAME MODULE

Our game module acts like a glue that incorporates all other modules into one to execute the game. This module is responsible for evaluating the progress of the game.

3 Parallel OpenMP Based MiniMax

The nature of the move generation module makes it hard to statically schedule assignment across all threads. We can predict the workload based on the piece type, however this heuristic can and will fail us as the number of viable moves for a piece is affected by other pieces on the board (as soon as the game starts progressing). Given this information, we found it optimal to use OpenMP interface to parallelize this task because OpenMP provides the option to dynamically schedule work load across threads, and so better workload balance can be achieved. For this same reason, we decided to opt out of incorporating SIMD Vector intrinsics to speed up parallelizable portions of our implementation because under extremely inefficient work load balances across hardware threads, the seemingly significant speed up obtained by such threads can be overshadowed by the latencies that a majority of such threads incur while waiting being idle for the work load dominant tasks to finish executing.

We incorporated a parallel region using openmp interface semantics in two places; sweeping the board squares and generating all immediate legal moves that the player can continue with. However, a restricting factor for the latter that we incurred while incorporating parallelism was that, once we accumulate all possible moves with the help of multiple threads, the accumulation and aggregation of all these moves required for us to introduce a critical section in our code that had to maintain atomicity which reduced our absolute speedup.

```
Best_Move parallel_max(ChessBoard board, int depth, Move move, int color)
{
   if(depth == 0) {
      Best_Move bm;
      bm.move = move;
      bm.score = evaluate(board, color);
      return bm;
   }

   std::vector<Move> moves;
   std::vector<Best_Move> moves_score;
   moves = gen_all_next_moves_parallel(board, color);
   moves_score.resize(moves.size());

   #pragma omp for
   for (int i = 0; i < moves.size(); ++i) {
      ChessBoard bc = *(board.copy());
      Move mv = moves[i];
      bc.move(mv.Old.row, mv.Old.col, mv.New.row, mv.New.col);
      moves_score[i] = parallel_min(board, depth-1, mv,
                                    (color != WHITE) ? WHITE : BLACK);
      bc.free_board();
   }
   Best_Move max_score = fast_get_max(moves_score, 0, moves_score.size());
   return max_score;
    }
}
```

The above snippet of code describes the use of #pragma omp directives to parallelize **maxi** search. This is synchronous with our implementation of our parallel **mini** search which essentially tries to compute the least optimal value for our maximizing algorithm to work with in parallel.

Secondly, we decided to incorporate parallelism into our get_all_next_moves function. We realized that computation of legal moves given any partial board for any player is a highly parallelizable area for our chess implementation. This is due to the fact that sweeping the board for potential pieces that can perform moves and generating all positions that a given piece can move to are independent processes. Therefore, the only limiting factor in terms of processing would be the hardware capabilities. But, as we had explained before, the only additional drawback that came with parallelizing this region of our code was the fact that our final aggregation of all legal moves needed to be added to our main Move vector atomically and therefore, out of all threads that were initialized for this process, only the main thread will perform this accumulation operation.

```
std::vector<Move> gen_all_next_moves_parallel(ChessBoard B, int color)
{
   int idx, row, col;
   Piece *p;
   std::vector<Move> moves;
   std::vector<std::vector<Move>> submoves;
   submoves.resize(64);

   #pragma omp for
   for(idx = 0; idx < 64; idx++) {
      row = idx/8;
      col = idx%8;
      p = B.lookup(row, col);
      if(p != NULL && p->color == color) {
         submoves[idx] = gen_next_moves(B, row, col);
      }
   }

   #pragma omp for
   for(idx = 0; idx < 64; idx++) {
      #pragma omp critical
      moves.insert(std::end(moves), std::begin(submoves[idx]), std::end(submoves[idx]));
   }

   return moves;
}
```

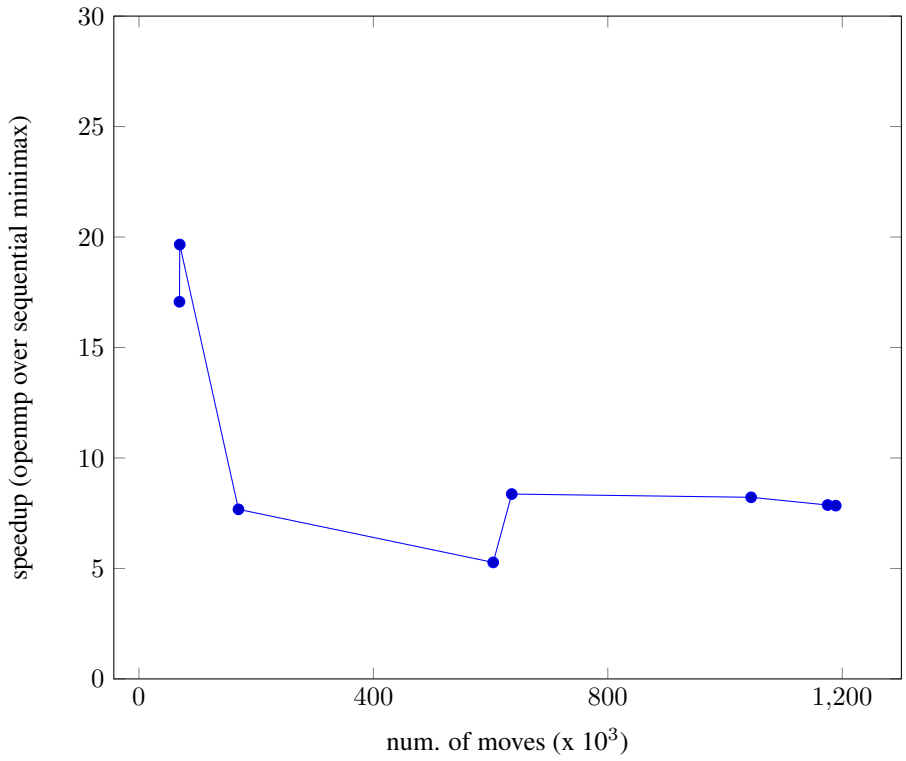
4 Results

Our parallel implementation was able to achieve a substantial speedup over the base sequential implementation for a variety of cases. However, compared to other chess AI solutions that were under years of research and development, it still lacks in the optimal performance it can theoretically achieve. In addition to this, we realized another potential avenue for parallelism when were trying to obtain the maximum or minimum value across a vector of moves.

Program	Depth
Minimax_Parallel_OpenMP	6
DeepBlue	6-8, sometimes 20+
StockFish	40

The Parallel Implementation using OpenMP's API was an optimization of the sequential minimax algorithm we had written. To compare performance and gauge improvements, we timed both implementations against generating moves (based on a certain depth) under different board situations, and measured the achieved speedup.

Results from timing.h



Refer to Appendix A for the raw table of data used to generate graph.

Refer to Appendix B for board statistics (such as number of moves available for each player).

The results from our raw table were achieved by timing the AIs on the GHC machine clusters, as the added parallelism benefits from the high CPU core-count. Low core count devices have shown to not achieve as much speedup.

Though the depth and color parameters change, this is done to achieve different number of moves the AI has to consider. Though workload balance is not maintained across different parameters, the dynamic scheduling nature of Open MP should mitigate the effects of different workload balance and better reflect the efficiency of the AI.

The speedups reflected in the graph are the average speedup over 15 instances of timings.

We expect as the number of moves increase, speedup should increase as the parallel implementation benefits from more parallelism. However, after a certain point, the number of tasks becomes so large that it becomes such an overhead that it reduces the speedup. We can see that trend take place as speedup peaks with 69,632 moves and then drops. What surprised us is this reported increase in speedup from 606K moves to 636K moves. However, speedup plateaus after 636K moves. It would be interesting to further look into the dip and the reasoning behind it as a potential key to improve the parallelism of our AI.

5 Work Distribution

Name	Work %
Sameer Ahmad	50
Ishaq Yousef Haj-Hasan	50

References

[1] Chess Programming. (n.d.). Retrieved November 2019, from https://www.chessprogramming.org/Main_Page.

[2] Guidry, M., McClendon, C. (n.d.). Techniques to Parallelize Chess. Retrieved October 30, 2019, from https://pdfs.semanticscholar.org/8e1c/9f70aa4849475199e26ccd3e21c662e2d801.pd

[3] Steele, K. (1999). Parallel Alpha-Beta Pruning of Game Decision Trees: A ChessImplementation. Retrieved October 30, 2019, from https://students.cs.byu.edu/snell/Classes/CS584/projectsF99/steele/report.html

Appendix

A

Board Num.	Depth	Color	Num Moves	Speedup
1	5	WHITE	1044495	8.22
1	5	BLACK	1175057	7.87
1	4	WHITE	69120	17.07
1	4	BLACK	69632	19.66
2	4	WHITE	636075	8.367
2	4	BLACK	604464	5.273
3	5	BLACK	1188887	7.842
3	4	WHITE	169840	7.675
3	5	WHITE	8322215	7.199

Table 1: Raw Data of Speedup Calculated by OpenMP over sequential in different board situations.

B

Board	White Pcs.	Black Pcs.	Num Moves White (D=5)	Num Moves Black (D=5)
1	2	3	1044495	1175057
2	6	2	-	-
3	6	2	8322215	1188887

Board	White Pcs.	Black Pcs.	Num Moves White (D=4)	Num Moves Black (D=4)
1	2	3	69120	69632
2	6	2	636075	604464
3	6	2	169840	-