Robert DeLaus, Yianni Giannaris, Cameron Burnett

Beta II Write-up

For the Beta II implementation, we primarily focused on improving our serial implementation as much as possible before beginning to introduce parallelization optimizations. In the submission for Beta I the only optimizations we had made were on the laser coverage heuristic and the board representation. For this submission, we began by taking another look at the board representation. In our position struct we have *board* which is an array of pieces that represents the entire 8 by 8 board plus sentinels which make it 10 by 10. In the board we also have *pieceLocations* which was previously an array of squares of length 16 that represented the locations of all the pieces on the board, but we have now changed it to a 2 by 8 array where the first row is for white pieces and the second row is for black pieces, with the kings always being at index 0 in each of the subarrays*.* The rest of the pawns a scattered throughout the arrays of the appropriate color. The square corresponding to a specific piece with always be found at the same index in *pieceLocations*. In order to get this index when we just have the piece, we store the index in the 3 high order bits of each piece. Next we found that the Piece_t, Square_t, rnk_t and fil_t were all larger than they needed to be so we changed Piece_t to a uint8, Square_t to a int16 and both rnk_t and fil_t to int8. We also made a few serial improvements for the heuristics. We started by creating lookup tables for p-centrality, mult_dist, and harmonic distance so that we could avoid some computation time. Next we noticed that while the code for the project included the implementation of a king mobility heuristic, this heuristic wasn't actually being used in eval. While including this understandably reduced our nodes per second, it significantly improved our ELO in games. The last major serial change we made was implementing an opening book. To do this we made a recursive function that starts off with the board at the starting position. The function uses generateMoves to find all the moves from this position. Then for each of these moves it performs an iterative deepening root search to a given depth from each of the new positions resulting from one of these moves. This computes a score for each of the moves which is then stored in the high order bits of the moves. The function sorts the list of moves using insertion sort and then calls the function recursively on the positions resulting from the N best moves. This process performed recursively to a desired depth to get the opening book entries for that depth. The series of moves are passed through the function and when the desired depth is reached, a deeper root search is done from that position to find the best move after that series of moves has been made.

The performance improvement that our serial optimizations made can be seen in the data at the end of this document.

In order to make parallel optimizations, we first decided to tackle the scout_search routine. We noticed that the for loop over the moves can be easily parallelized if we handled some thread safety issues pertaining to the node_count_serial, the searchNode node, and the transposition lookup table.

Starting with node_count_serial, we utilized the __sync_fetch_and_add C builtin atomic addition function to increment the counter. Moving onto node, we realized that node's current implementation was not thread safe, specifically because of different fields in node, such as best_score, quiesence were written to and read from in the same piece of code that was to be parallelized. In order to overcome this issue, we created node_mutex that would would synchronize accesses to the node. We acquired and released the simple_mutex before calling search_process_score in search_scout because the search_process_score function mutates fields in the node.

Since the a call is made to evaluateMove within the for loop of scout_search, and evaluateMove makes calls to scout_search, multiple threads spawned by the planned cilk_for will access the transposition table, even though the call to update_transposition_table is made outside the for loop. Because of this indirect recursion, as the call to search_scout is hidden behind the call to evaluateMove, we need to provide some kind of locking mechanism to ensure changes to the transposition table were thread safe. Rather than locking on the entire ttHashtable, which would cause excessive lock contention, we decided to lock on individual record sets within the ttHashtable. By locking on a set, threads could asynchronously access records within the ttHashtable as long as those records existed in different sets. We modified the ttHashtable struct to store a p_thread_mutex* as a field. This pointer represented the start of an array p_thread_mutex's. We then defined three functions tt_lock_init, tt_lock_lock, and tt_lock_unlock. tt_lock_init, which is called in tt_make_hashtable, simply initializes the p_thread_mutex array that is pointed to by the ttHashtable struct. The size of this array is equal to the number of sets in the ttHashtable. tt_lock_lock accepts a uint64_t key, finds the corresponding set that contains the record with that key, then indexes into the mutex array to acquire the appropriate lock for that specific set. tt_lock_unlock reverses this process, finding the set to the corresponding key and releasing the mutex.

Once we took appropriate measures to ensure thread_safety, we swapped out the move for loop within search_scout for a cilk_for loop.

The performance improvement that our parallel optimizations made can be seen in the data at the end of this document.

For the final version of this project, we plan to continue parallelizing our code to take full advantage of the parallelism available in this implementation. One of the areas for improvement is searchRoot. Rather than looping through moves and making calls to searchPV and scout_search in a serial fashion, we plan to find the best score in a parallel manner. In order to avoid lock contention, we plan spawn calls to searchRoot in a binary fashion on the move_list. Once we have reach a singular move in the move list, we will then find a score and return that score. The searchRoot function will sync on parallel recursive calls to searchRoot and wait for scores. searchRoot will then compare the returned scores, and the larger of the two will be returned. In order to determine the appropriate move, and to ensure that only scores with valid moves are returned, we need to return a struct that stores both a score and a move associated with the score. We attempted implementing this version for the Beta II submission but did not have enough time to debug.

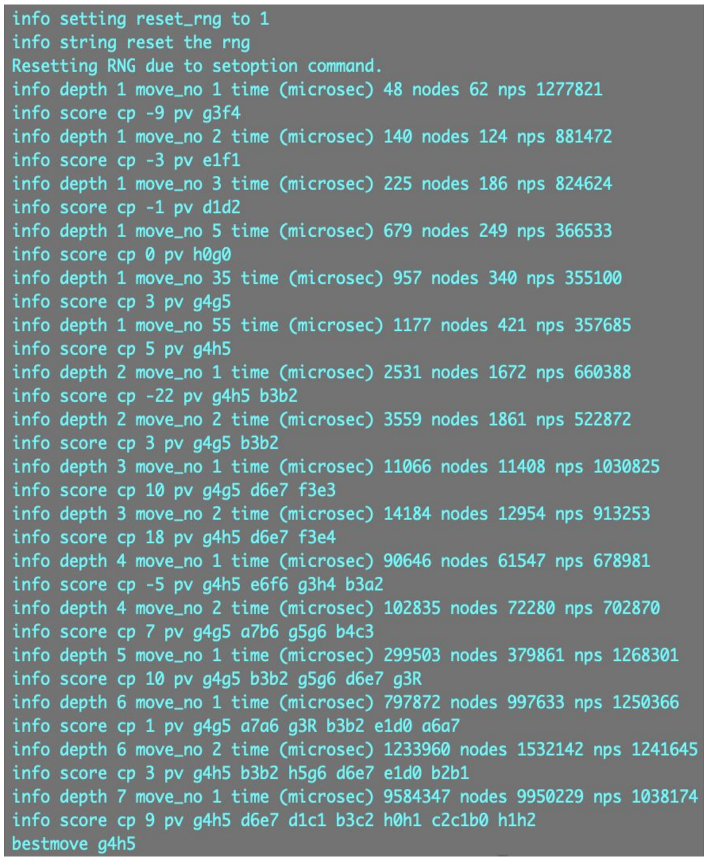
We also plan to implement an end-game book in a similar fashion to how we built our opening book. The opening book provided one of the most significant in-game performance increases of any of the changes we made, and we anticipate the closing book having a similar effect. Since Rob spent so much time with the opening book, we plan to assign him the task of building the end-game one as well.

Also, we plan to improve our opening book further by making it a hash-table. Right now is represented by lists, and with over 10,000 entries at some depths, having to iterate through to find an entry is definitely not optimal.

Work breakdown:

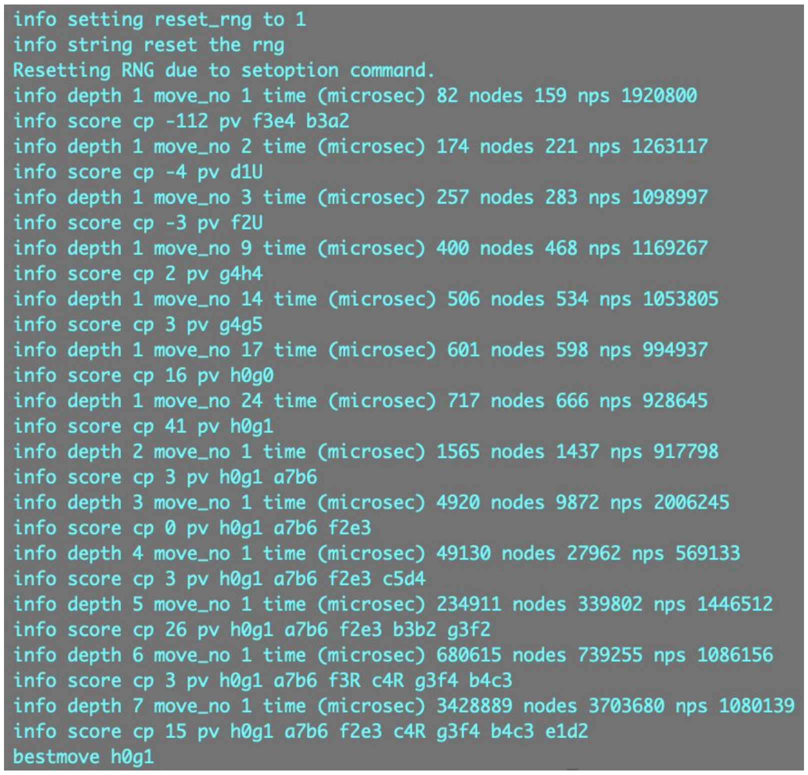
Rob spent about 3 hours working on the board rep and struct sizes, 6 hours working on the opening book, and 4 hours working on parallelization. Cameron spent about 4 hours making lookup tables and implementing the mobility heuristic, 6 hours debugging and further optimizing our serial laser coverage and eval functions, 2 hours attempting other optimizations in eval.c, and 3 hours working on parallelization. Yianni spent about 4 hours adding lookup tables to heuristic functions, testing to see which lookup table optimizations would yield the greatest improvements, 5 hours debugging, and 4 hours attempting to add parallelization optimizations.

Beta 1 submission (reference):



```
info setting reset_rng to 1
info string reset the rng
Resetting RNG due to setoption command.
info depth 1 move_no 1 time (microsec) 48 nodes 62 nps 1277821
info score cp -9 pv g3f4
info depth 1 move_no 2 time (microsec) 140 nodes 124 nps 881472
info score cp -3 pv e1f1
info depth 1 move_no 3 time (microsec) 225 nodes 186 nps 824624
info score cp -1 pv d1d2
info depth 1 move_no 5 time (microsec) 679 nodes 249 nps 366533
info score cp 0 pv h0g0
info depth 1 move_no 35 time (microsec) 957 nodes 340 nps 355100
info score cp 3 pv g4g5
info depth 1 move_no 55 time (microsec) 1177 nodes 421 nps 357685
info score cp 5 pv g4h5
info depth 2 move_no 1 time (microsec) 2531 nodes 1672 nps 660388
info score cp -22 pv g4h5 b3b2
info depth 2 move_no 2 time (microsec) 3559 nodes 1861 nps 522872
info score cp 3 pv g4g5 b3b2
info depth 3 move_no 1 time (microsec) 11066 nodes 11408 nps 1030825
info score cp 10 pv g4g5 d6e7 f3e3
info depth 3 move_no 2 time (microsec) 14184 nodes 12954 nps 913253
info score cp 18 pv g4h5 d6e7 f3e4
info depth 4 move_no 1 time (microsec) 90646 nodes 61547 nps 678981
info score cp -5 pv g4h5 e6f6 g3h4 b3a2
info depth 4 move_no 2 time (microsec) 102835 nodes 72280 nps 702870
info score cp 7 pv g4g5 a7b6 g5g6 b4c3
info depth 5 move_no 1 time (microsec) 299503 nodes 379861 nps 1268301
info score cp 10 pv g4g5 b3b2 g5g6 d6e7 g3R
info depth 6 move_no 1 time (microsec) 797872 nodes 997633 nps 1250366
info score cp 1 pv g4g5 a7a6 g3R b3b2 e1d0 a6a7
info depth 6 move_no 2 time (microsec) 1233960 nodes 1532142 nps 1241645
info score cp 3 pv g4h5 b3b2 h5g6 d6e7 e1d0 b2b1
info depth 7 move_no 1 time (microsec) 9584347 nodes 9950229 nps 1038174
info score cp 9 pv g4h5 d6e7 d1c1 b3c2 h0h1 c2c1b0 h1h2
bestmove g4h5
```

After serial optimizations:



```
info setting reset_rng to 1
info string reset the rng
Resetting RNG due to setoption command.
info depth 1 move_no 1 time (microsec) 82 nodes 159 nps 1920800
info score cp -112 pv f3e4 b3a2
info depth 1 move_no 2 time (microsec) 174 nodes 221 nps 1263117
info score cp -4 pv d1U
info depth 1 move_no 3 time (microsec) 257 nodes 283 nps 1098997
info score cp -3 pv f2U
info depth 1 move_no 9 time (microsec) 400 nodes 468 nps 1169267
info score cp 2 pv g4h4
info depth 1 move_no 14 time (microsec) 506 nodes 534 nps 1053805
info score cp 3 pv g4g5
info depth 1 move_no 17 time (microsec) 601 nodes 598 nps 994937
info score cp 16 pv h0g0
info depth 1 move_no 24 time (microsec) 717 nodes 666 nps 928645
info score cp 41 pv h0g1
info depth 2 move_no 1 time (microsec) 1565 nodes 1437 nps 917798
info score cp 3 pv h0g1 a7b6
info depth 3 move_no 1 time (microsec) 4920 nodes 9872 nps 2006245
info score cp 0 pv h0g1 a7b6 f2e3
info depth 4 move_no 1 time (microsec) 49130 nodes 27962 nps 569133
info score cp 3 pv h0g1 a7b6 f2e3 c5d4
info depth 5 move_no 1 time (microsec) 234911 nodes 339802 nps 1446512
info score cp 26 pv h0g1 a7b6 f2e3 b3b2 g3f2
info depth 6 move_no 1 time (microsec) 680615 nodes 739255 nps 1086156
info score cp 3 pv h0g1 a7b6 f3R c4R g3f4 b4c3
info depth 7 move_no 1 time (microsec) 3428889 nodes 3703680 nps 1080139
info score cp 15 pv h0g1 a7b6 f2e3 c4R g3f4 b4c3 e1d2
bestmove h0g1
```

After parallelization of scout search:

Awsrun8 has been consistently timing out for the last hour, but on earlier runs, we saw that parallelization of scout search resulted in about 4.5 million nps at depth 7.