6.172 Project 4 Final Write-Up

Kevin Foley, Kevin Shen, Lucas Novak, Juan Ortiz

1 Introduction

The goal of this project was to improve a game-playing AI for the Leiserchess -- a chess-like game with lasers. To play the game, the AI uses the variation of alpha-beta search called Principal Variation Search (PVS) to explore the possible moves it could make and rate them with a given set of heuristics and choose the best available option. For Beta 1 we only sped up our serial code to improve performance, specifically by changing our code to use smaller types, adding a fast path to laser coverage, and precomputing some values (see section 3 for more details). For Beta 2 we continued to speed up our serial code and added a little parallelization (see section 4). For the final we added a little more speed up to our main bottleneck and improved our parallelization by introducing better pruning (see section 5).

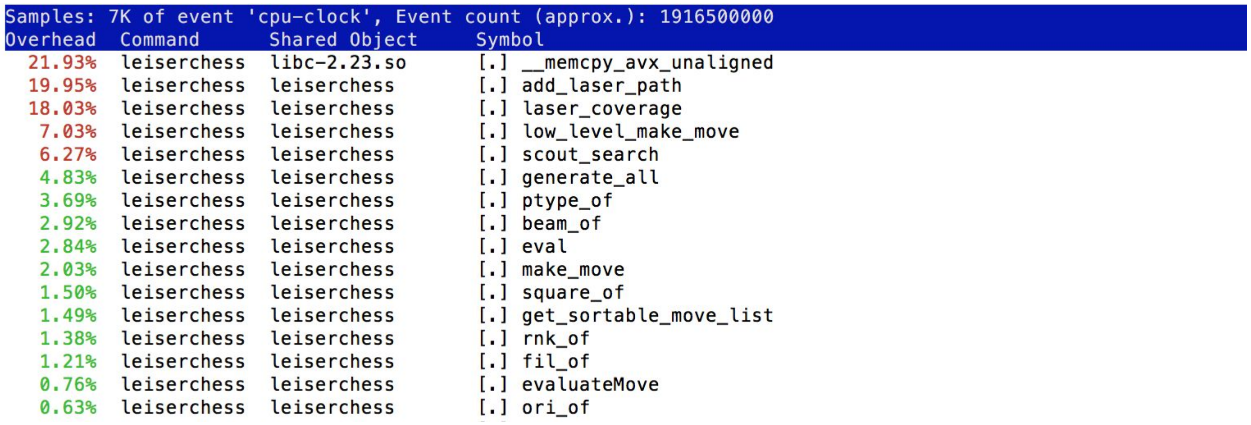
*Note after Beta 1 we started measuring in NPS only. We also made 3 additional input files to measure speed on. These new files are described in section 6. See 5.6&5.7 for an overall change in the performance of serial code.

2 Profiling

2.1 Reference Implementation Profiling

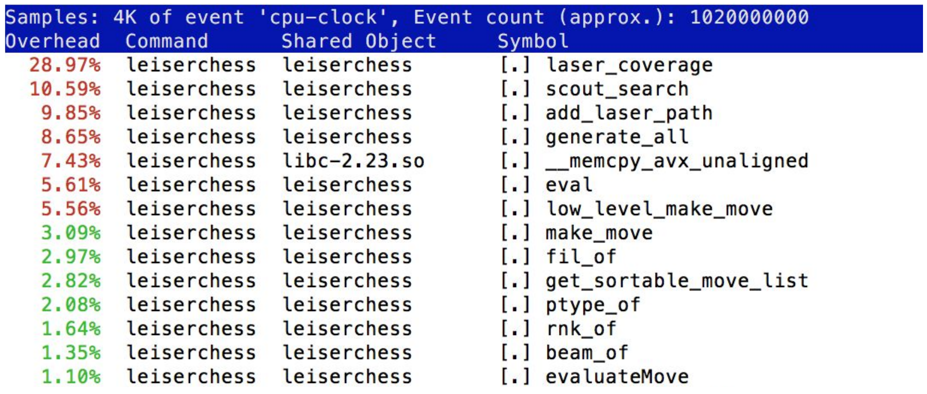
For this beta we tested and profiled the code on the same input.txt file that would set the rng value, search to a depth 8, make a move, search to a depth 8, make a move, search to a depth 6, make a move, and then search to a depth 4.

For the reference implementation we profiled it using perf and then measured its speed using time command line argument and getting the average nodes per second (NPS). The perf profile is shown below:


```
Samples: 7K of event 'cpu-clock', Event count (approx.): 1916500000
Overhead  Command       Shared Object      Symbol
  21.93%  leiserchess  libc-2.23.so       [.] __memcpy_avx_unaligned
  19.95%  leiserchess  leiserchess        [.] add_laser_path
  18.03%  leiserchess  leiserchess        [.] laser_coverage
   7.03%  leiserchess  leiserchess        [.] low_level_make_move
   6.27%  leiserchess  leiserchess        [.] scout_search
   4.83%  leiserchess  leiserchess        [.] generate_all
   3.69%  leiserchess  leiserchess        [.] ptype_of
   2.92%  leiserchess  leiserchess        [.] beam_of
   2.84%  leiserchess  leiserchess        [.] eval
   2.03%  leiserchess  leiserchess        [.] make_move
   1.50%  leiserchess  leiserchess        [.] square_of
   1.49%  leiserchess  leiserchess        [.] get_sortable_move_list
   1.38%  leiserchess  leiserchess        [.] rnk_of
   1.21%  leiserchess  leiserchess        [.] fil_of
   0.76%  leiserchess  leiserchess        [.] evaluateMove
   0.63%  leiserchess  leiserchess        [.] ori_of
```
The time we got was: real 0m1.869s, user 0m1.844s, sys 0m0.004s. The average NPS was 719281.

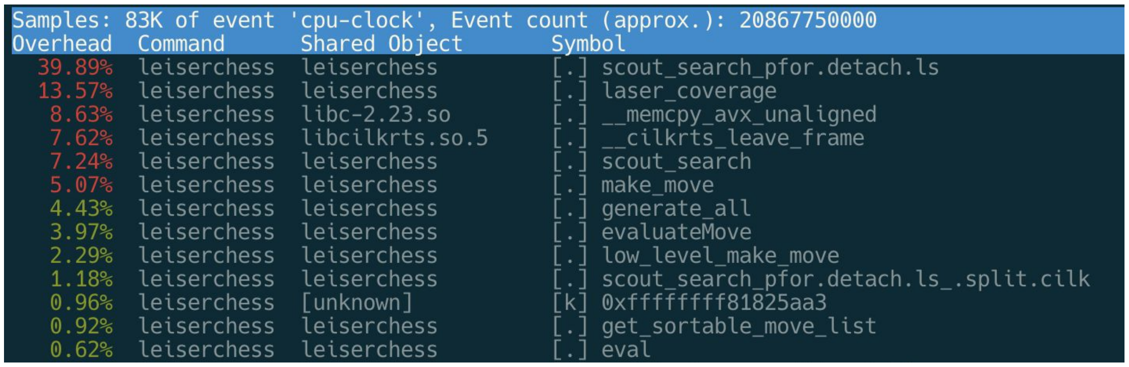
As you can see from the perf report the biggest bottlenecks came from trying to calculate the laser coverage heuristic (either from the function itself or one of its helper functions).

2.2 Beta 1 Perf


```
Samples: 4K of event 'cpu-clock', Event count (approx.): 1020000000
Overhead  Command       Shared Object      Symbol
  28.97%  leiserchess  leiserchess        [.] laser_coverage
  10.59%  leiserchess  leiserchess        [.] scout_search
   9.85%  leiserchess  leiserchess        [.] add_laser_path
   8.65%  leiserchess  leiserchess        [.] generate_all
   7.43%  leiserchess  libc-2.23.so       [.] __memcpy_avx_unaligned
   5.61%  leiserchess  leiserchess        [.] eval
   5.56%  leiserchess  leiserchess        [.] low_level_make_move
   3.09%  leiserchess  leiserchess        [.] make_move
   2.97%  leiserchess  leiserchess        [.] fil_of
   2.82%  leiserchess  leiserchess        [.] get_sortable_move_list
   2.08%  leiserchess  leiserchess        [.] ptype_of
   1.64%  leiserchess  leiserchess        [.] rnk_of
   1.35%  leiserchess  leiserchess        [.] beam_of
   1.10%  leiserchess  leiserchess        [.] evaluateMove
```
The time was real 0m1.000s,user 0m0.984s, and sys 0m0.008s and had an average NPS of 1216928. As you can see calculating the laser heuristic still takes most of the time.

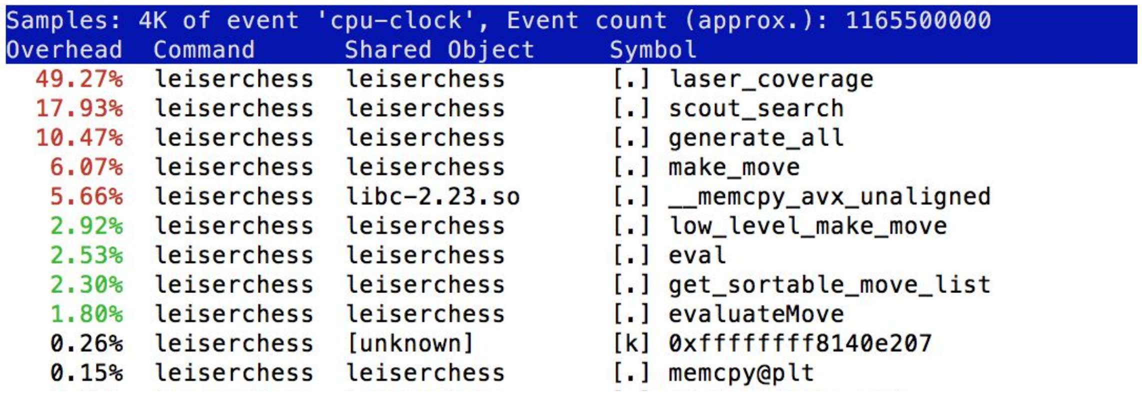
2.3 Beta 2 Perf

2.3.1 Parallel



```
Samples: 83K of event 'cpu-clock', Event count (approx.): 20867750000
Overhead  Command      Shared Object        Symbol
  39.89%  leiserchess  leiserchess          [.] scout_search_pfor.detach.ls
  13.57%  leiserchess  leiserchess          [.] laser_coverage
   8.63%  leiserchess  libc-2.23.so         [.] __memcpy_avx_unaligned
   7.62%  leiserchess  libcilkrts.so.5      [.] __cilkrts_leave_frame
   7.24%  leiserchess  leiserchess          [.] scout_search
   5.07%  leiserchess  leiserchess          [.] make_move
   4.43%  leiserchess  leiserchess          [.] generate_all
   3.97%  leiserchess  leiserchess          [.] evaluateMove
   2.29%  leiserchess  leiserchess          [.] low_level_make_move
   1.18%  leiserchess  leiserchess          [.] scout_search_pfor.detach.ls_.split.cilk
   0.96%  leiserchess  [unknown]            [k] 0xffffffff81825aa3
   0.92%  leiserchess  leiserchess          [.] get_sortable_move_list
   0.62%  leiserchess  leiserchess          [.] eval
```

2.3.2 Serial

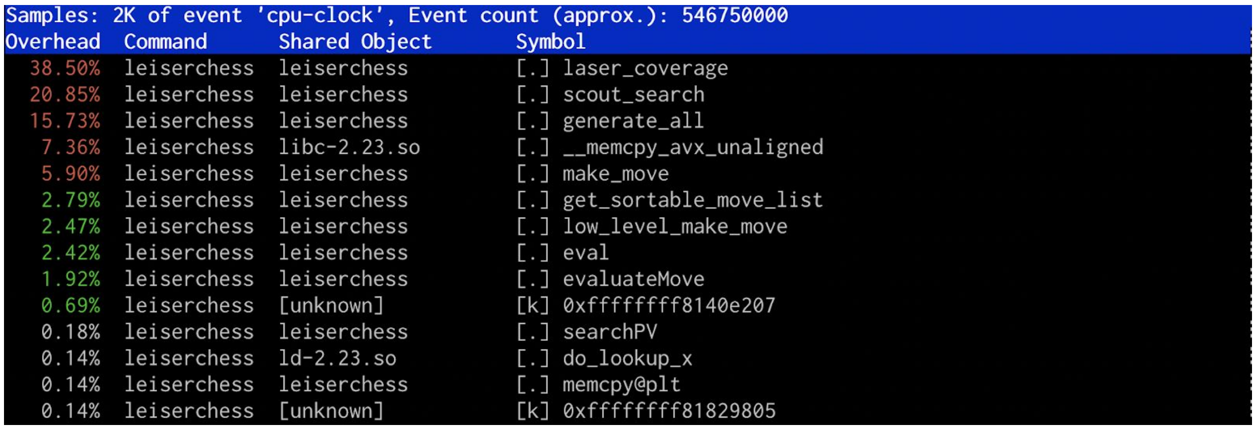


```
Samples: 4K of event 'cpu-clock', Event count (approx.): 1165500000
Overhead  Command      Shared Object      Symbol
  49.27%  leiserchess  leiserchess        [.] laser_coverage
  17.93%  leiserchess  leiserchess        [.] scout_search
  10.47%  leiserchess  leiserchess        [.] generate_all
   6.07%  leiserchess  leiserchess        [.] make_move
   5.66%  leiserchess  libc-2.23.so       [.] __memcpy_avx_unaligned
   2.92%  leiserchess  leiserchess        [.] low_level_make_move
   2.53%  leiserchess  leiserchess        [.] eval
   2.30%  leiserchess  leiserchess        [.] get_sortable_move_list
   1.80%  leiserchess  leiserchess        [.] evaluateMove
   0.26%  leiserchess  [unknown]          [k] 0xffffffff8140e207
   0.15%  leiserchess  leiserchess        [.] memcpy@plt
```

After the series of optimizations that we performed since our Beta 1 submission, the function whose relative runtime increased the most was laser_coverage. This however can be mostly attributed to the increase in work performed in other areas of the code. This is made evident in the parallel version of our code as the spawns decrease the relative time spent on the function.

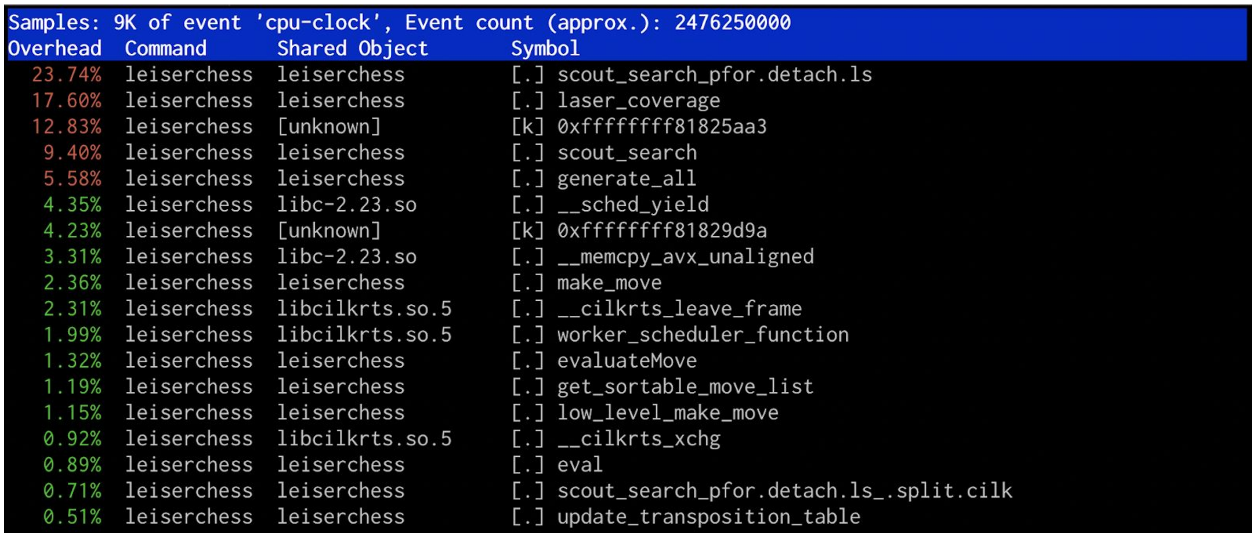
2.4 Final Perf

2.4.1 Serial Execution



```
Samples: 2K of event 'cpu-clock', Event count (approx.): 546750000
Overhead  Command      Shared Object        Symbol
  38.50%  leiserchess  leiserchess          [.] laser_coverage
  20.85%  leiserchess  leiserchess          [.] scout_search
  15.73%  leiserchess  leiserchess          [.] generate_all
   7.36%  leiserchess  libc-2.23.so         [.] __memcpy_avx_unaligned
   5.90%  leiserchess  leiserchess          [.] make_move
   2.79%  leiserchess  leiserchess          [.] get_sortable_move_list
   2.47%  leiserchess  leiserchess          [.] low_level_make_move
   2.42%  leiserchess  leiserchess          [.] eval
   1.92%  leiserchess  leiserchess          [.] evaluateMove
   0.69%  leiserchess  [unknown]            [k] 0xffffffff8140e207
   0.18%  leiserchess  leiserchess          [.] searchPV
   0.14%  leiserchess  ld-2.23.so           [.] do_lookup_x
   0.14%  leiserchess  leiserchess          [.] memcpy@plt
   0.14%  leiserchess  [unknown]            [k] 0xffffffff81829805
```

2.4.2 Parallel Execution



```
Samples: 9K of event 'cpu-clock', Event count (approx.): 2476250000
Overhead  Command      Shared Object        Symbol
  23.74%  leiserchess  leiserchess          [.] scout_search_pfor.detach.ls
  17.60%  leiserchess  leiserchess          [.] laser_coverage
  12.83%  leiserchess  [unknown]            [k] 0xffffffff81825aa3
   9.40%  leiserchess  leiserchess          [.] scout_search
   5.58%  leiserchess  leiserchess          [.] generate_all
   4.35%  leiserchess  libc-2.23.so         [.] __sched_yield
   4.23%  leiserchess  [unknown]            [k] 0xffffffff81829d9a
   3.31%  leiserchess  libc-2.23.so         [.] __memcpy_avx_unaligned
   2.36%  leiserchess  leiserchess          [.] make_move
   2.31%  leiserchess  libcilkrts.so.5      [.] __cilkrts_leave_frame
   1.99%  leiserchess  libcilkrts.so.5      [.] worker_scheduler_function
   1.32%  leiserchess  leiserchess          [.] evaluateMove
   1.19%  leiserchess  leiserchess          [.] get_sortable_move_list
   1.15%  leiserchess  leiserchess          [.] low_level_make_move
   0.92%  leiserchess  libcilkrts.so.5      [.] __cilkrts_xchg
   0.89%  leiserchess  leiserchess          [.] eval
   0.71%  leiserchess  leiserchess          [.] scout_search_pfor.detach.ls_.split.cilk
   0.51%  leiserchess  leiserchess          [.] update_transposition_table
```

You can see the changes made to laser coverage (see 5.1,5.2) did help reduce the relative time it took. But is still our serial bottleneck and large portion in parallel.

3 Beta 1 Optimizations

3.1 Using Smaller Types

The first change made was we changed the type of piece_t. Initially the type was an uint_32 but only needed to use 5 bits of information to store ptype, color, and orientation. By changing the type to a uint_8 we now used only 8 bits less space, speeding up and parts of our code that used a piece_t type and especially parts of the code using a position_t that contained an array of piece_ts . With this change our code now: real 0m1.567s, user 0m1.552s, sys 0m0.004s. Our average NPS was 830218.

The next change was changing the type of laser from a float to an int. Similar to above this would decrease the number of bits needed to store a variable. Our code now ran with times: real 0m1.567s, user 0m1.548s, sys 0m0.008s. Our average NPS was 843557.

3.2 Adding Fast Path to Laser Coverage

In the initial code, for each possible move, we perform that move and then run the test for the coverage of that laser path. Both performing the move and testing the coverage of a move is expensive, so we created a fast path so we only need to test a subset of all of the possible moves.

A move will affect the laser coverage heuristic if the king is moving or if the move is moving into or out of the path of the last laser that the king fired. Any other move does not matter because the laser coverage is the same as if no move was made. Therefore, we first compute the current path if the laser was fired and then test if a move changes any part of this path. Our code now runs with times : real 0m1.115s, user 0m1.100s,sys 0m0.004s. Average NPS was 1103942.
*It turned out there was a bug in our fast path implementation, described in 4.2

3.3 Precomputing Board Positions

The next change was updating how we accessed the board (position_t->board). While not directly in our bottleneck, methods that access the board would need to be changed if we wanted to change our board representation (e.g. square_of). For the beta we changed this by precomputing the calculations and storing them in an array. This would prevent us from needing to recalculate their index every time and gave us a quick way to change how we were doing the calculations. At first this array was just for the indexes in the board that contained actual pieces, creating an 8x8 array. With this change the time was now real 0m1.026s, user 0m1.012s, sys 0m0.004s and an average NPS of 1192014. After making 4 seperate arrays to precompute the values of the first row/col of invalid square the time was real 0m1.000s,user 0m0.984s, and sys 0m0.008s and had an average NPS of 1216928.

This precompution only replaced square_of in eval.c as it did not offer any speed up in mov_gen.c. Additionally we used a single 10x10 array instead of one 8x8 and 4 1x10 arrays but due to poor cache performance it did not give any speed up.

3.4 Sorting

Another optimization we attempted to do was to change the way we sorted the list of moves during scout search. When iterating over the possible moves, PVS requires them to be sorted in descending order based on their key. The original implementation utilizes insertion sort to improve this, so we attempted to change this to quick-sort. This however resulted in an overall slowdown so we reverted to use the given implementation of insertion sort.

4 Beta 2 Optimizations

-for beta times please look at 4.7
-our parallel time is in 4.5

4.1 Precomputation

The first way we decided to speed up our code was to do more precomputation, specifically we precomputed the values for pcentral and mult_dist. For pcentral we realized we were only using rank and file for squares within the board [0,7] so we could precompute an array of length 256 of the values. For mult_dist we needed a rank and file of [-1,8]. We tried 2 approaches to precomputing: make a 10x10 array for every value, 8x8 of every value within [0,7] and a 1x10 array for [-1,8] that we can access to avoid division then multiply with another value of the array. The second option worked better.

4.2 Fixing Fast Path

We realized we had a bug with our fast path in laser coverage (see 3.2). Our laser heuristic did not consider the values of moves that did not modify the lasers path (i.e. a piece moving and being in the path of the laser). This caused us to make several bad moves at the very end of the game. We did not notice this bug as our default input.txt happened to always have a piece move in front of the laser. This fix did not change our time (so was not added to the time table in 4.7) at all but when we ran our code on the scrimmage server it was better at closing games.

4.3 Sorting

One other optimization that we did to search over the possible moves was to change the location in which we sorted them. Previously, there was a call to insertion sort for every move in the move list. This however is unnecessary as the order of the moves will not be affected after the first sort. Thus, a single sort of the move list outside of the iteration is sufficient.

4.4 Make Low_Level_Make_Move in Place

One other optimization that we made was to turn low_level_make_move() into an in place algorithm. Normally low_level_make_move() copies an entire position_t object. Instead we can modify the current board position, add the laser path, then revert the changes.
*This actually created a bunch of race conditions when we added parallelism. We did not catch it until after beta2 (see 5.3)

4.5 Pawn List

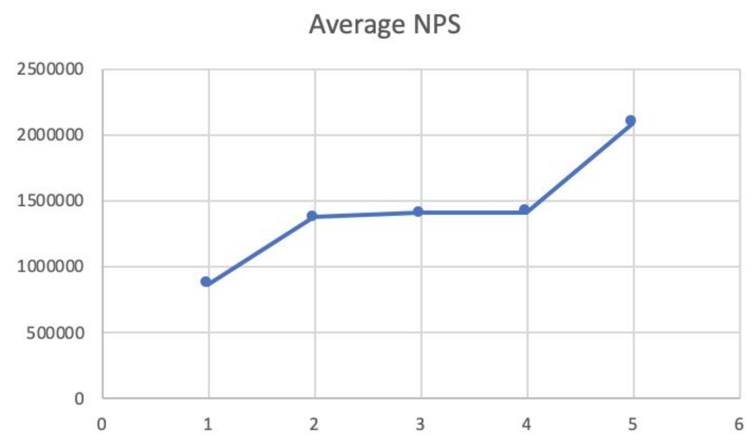
One disadvantage of the square centric board is that for functions in which we need to iterate over the pieces in the board, we waste a lot of work having to check empty squares. One way to mitigate this waste is to further augment our board data structure to also hold a piece-centric board with the locations of the pawns (since we already store the king locations). This, way when we need to iterate over the pieces, such as in the eval.c, we can reduce the number of loops and checks required. We implemented a pawn list for each of the players and used it to speed up the eval function. We wanted to use it for move generation, but we had a bug. It will be fixed for the final project. (see 6.1)

4.6 Parallelization

Another factor that we expect to significantly improve the performance of our implementation is the addition of parallel execution. We began parallelizing by using cilk to run scout search in parallel as suggested in lecture. We removed the possible races during search-process-score and update-best-move-history. We also tried some other parallelizations, but were unable to finish them for the beta 2. We worked on implementing Jamboree Search as it was shown in lecture. We also attempted to parallelize some of the heuristic calculations, but there were not sufficient speed ups. Lastly, we tried to better abort nodes such that we weren't wasting calculations, but that will need to be tested more for the final. In parallel our times in NPS were 4471596, 8753971, 7334607, and 5719770 for normal input, late game 1, late game 2, and swap. (described in 5.2).

4.7 Heuristics

When going through the anonymized beta code we noticed that the top elo code added the mobility heuristic. We tried this and it seemed to help us win when scrimaging with our friend so we kept it. **4.7 Graph of beta speed**



Commit / Modification	INPUT	LATE_GAME 1	LATE_GAME 2	SWAP	Timve AVG
Better Debugging Checker & Precomp	789242	1006182	588424	1071758	863901.5
Move sort out of loop	1329412	1468740	909651	1786706	1373627.25
move checker in place	1356964	1514600	919246	1846565	1409343.75
Pawn List	1386102	1541490	937581	1804875	1417512
Compiler Flags	2145920	2308137	1481431	2422204	2089423

-in the commit for our old serial with new inputs the value of input is lower than beta 1. We are not sure

5 Final

-for beta times please look at 5.7
-our parallel time is in 5.4
-for our overall performance over time see 5.6

5.1 Pre Computing Laser Coverage over Sentinels

The laser coverage heuristic determines its weight in two stages: comparing when a laser goes through the 8x8 boards and by comparing the position of the opponent's king to the sentinels that surround the board. For the later we realized that there could be at most 64 different computations needed for this step (one for each location on the board). So for each file and rank of each possible king we precomputed the that part of the heuristic.

5.2 Eval Hash Table

We noticed that there were many board states that were repeated over the course of running eval. To speed this up we created a hash table that would store the heuristic calculated in eval and would use the ZOB hash modulo hash_table_size for the index. This would mean whenever we see a board state we would check if it existed in the hashtable and if it did we could skip the rest of the work of eval. We also needed to add the zob key as a field in the hash table to differentiate between board states that shared the same zob%hash_table_size.

5.3 Fixing Low Level Make Move

We realized that turning low_level_make_move() into an in place algorithm (see 4.4), we had accidentally introduced a race condition. Because we were writing to the board, when our code was run in parallel many different threads were trying to write to the board at the same time and our code then was producing many wrong answers. To fix this we initially copied the boards to a buffer and then modified the buffer in place. We now only needed to copy the board once instead of every time laser_coverage called low_level_make_move and and we did not have a horrible race condition.

5.4 Increased Pruning

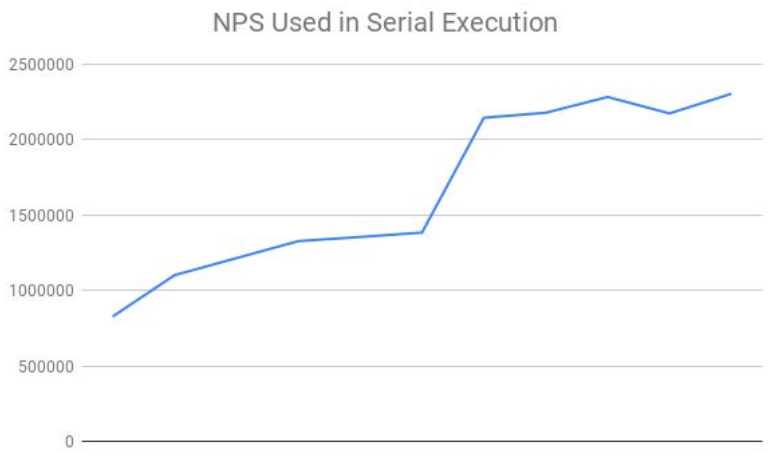
In Beta 2, we parallelized the for loop in scout search. While we could now process different moves on multiple cores, we were not pruning nodes properly. A node would only quit if it's parent aborted and not when the node itself had aborted. Therefore, for the final we made more checks that would prune using node->abort.

We also added coarsening to our parallelism. Before we spawn nodes to evaluate the moves, we calculate the score for the first 3 moves. If any of those moves pass the cutoff we don't need spawn any more calculations. Since spawning has overhead and there's a high chance we find a result in the first few moves, this improves our runtime.

5.5 Common Play Book

An idea we attempted to implement was a common play book. We would go through and scrape the most commonly seen states from the scrimmage server website. Then we would search those states to a deeper depth than we would be able to in game and add the best moves to a precomputed hash table. Finally in games we would check the hash table if we were in one of those common states and make that ideal move. Unfortunately we needed a little more time to implement this. We created the script to scrape the website, got the most common states, implemented a basic hash table, and found a way to add the best states to the table. The problem came from being able to search for the best move. We spent several hours running the best move for the 400 most common states but we had a bug where it did not read the state correctly. By the time we figured it out we did not have enough time to rerun the code on the common states and test to make sure our code work.

5.6 Speed-ups Over Time



Commit / Modification	INPUT
Smaller Types	830218
Fast Path	1103942
Precomputing Board Positions	1216928
Move sort out of loop	1329412
move checker in place	1356964
Pawn List	1386102
Compiler Flags	2145920
Laser Coverage Sentinel Precomputing	2179138
Eval Hash Table	2283618
Inplace Low Level Make Move	2174810
Better Pruning	2304742

* We decided to keep the low-level make move because it was faster on average for different inputs than its predecessor.

5.7 Speed of Serial Execution Over Time

Commit / Modification	INPUT	LATE_GAME 1	LATE_GAME 2	SWAP	Timve AVG
Smaller Types	830218	N/A	N/A	N/A	N/A
Fast Path	1103942	N/A	N/A	N/A	N/A
Precomputing Board Positions	1216928	N/A	N/A	N/A	N/A
Move sort out of loop	1329412	1468740	909651	1786706	1373627.25
move checker in place	1356964	1514600	919246	1846565	1409343.75
Pawn List	1386102	1541490	937581	1804875	1417512
Compiler Flags	2145920	2308137	1481431	2422204	2089423
Laser Coverage Sentinel Precomputing	2179138	2132487	1477774	2230883	2005070.5
Eval Hash Table	2331052	2218530	2218530	2314693	2270701.25
Inplace Low Level Make Move	2174810	2296567	1589455	2359295	2105031.75
Better Pruning	2342407	2339952	1604268	2323044	2152417.75

6 Testing

6.1 Checker Script

The checker script uses a simple text parser to take the output from running the bot using input.txt and comparing it to that of the output produced from the binary of the original bot. We first run each binary on the same input.txt (which the location of is passed into as an argument of the bash script). Next, we have a python script parse the output line by line and determine the number of nodes searched at each point, the moves made by the bots, and the nps. We compare the number of nodes and moves made by the bot to detect bugs / differences in the code.

6.2 New Inputs

We added 3 additional inputs. One was a board with a lot of pawns that could swap with each other. The other 2 were from the late stages of games where we made bad moves at the end of the game.

6.3 Playing Against Itself

One method we came up with to test improvements to the bot was to play newer versions of the bot against older versions. If the newer versions won more games, then we could more confidently conclude that the changes improved performance. Throughout our development process, we used this strategy several times to test iterations of our bot. To run these self-matches we utilized the provided python script as well as the java autotester.

7 Optimizations If We Had More Time

7.1 Use Piece Set for generate_all()

Currently, move generation loops over the entire board to find the possible moves. As mentioned in 4.4 we have a pawn list that should theoretically allow us to only loop through the pawns. For beta 2 we had a bug in our implementation of this so for the final we planed to fix this bug for the final. Unfortunately we could not figure out what was wrong and it was not the main bottleneck to fix. From our perf reports you can see that in serial this takes up 10% of our time and 4.4% of our time in parallel for beta 2 and in the final 15% in serial and 5.6% in parallel so this change should add some speed up though we don’t expect the amount to be too large .

7.2 Laser Coverage Search

Another optimization that we would have considered doing if we had more time would be to search for possible moves along the current laser path. Since the only moves that affect the laser path heuristic are those that modify the path in which the laser traverses the board, we can use the fact that a piece adjacent to the current path must be involved in the move in order for that to happen. In the case of a simple move, this will involve the pawn piece moving into the current path and redirecting the beam. In the case of a swap move, a friendly piece would first be moved from the positions surrounding the current laser path and then a simple move would take the swapped piece into the laser path. Additionally, we need to consider the moves for which the king moves or rotates as that will also change the laser path.

This would greatly decrease the number of moves that we have to generate and check and thus make our execution significantly faster.

7.3 More Common Book

One idea we had was to scrape the collection of games played on the scrimmage server, aggregate this data, and find the most common positions played by the class bots. With this data, we could potentially precompute the best move with a larger depth than we would normally be able to. Once we have the best move, we could construct a hash table to store all the common positions and the corresponding best moves. While we were able to successfully scrape this data, we did not have enough time to precompute the best moves and construct the hash table to store these moves.

7.4 Optimize Heuristics

Currently, we are using the default settings for weighting the values produced by eval's heuristics. We think that we could perform better if we did a search for the best heuristics by playing games using different heuristics and seeing which ones give us better performance.