Parallel Chess Engine with OpenMP

Sebastian Montiel
Carnegie Mellon University
Pittsburgh, PA 15213
smontiel@andrew.cmu.edu

15-618 Final Project Report (2022), Pittsburgh, USA.

Summary

I wrote and parallelized a chess engine. It uses a simple minimax tree search with alpha-beta pruning, parallelized with OpenMP.

1 Background

The core of algorithm of most chess engines is a search through the tree of all possible moves up to a certain depth. The tree grows to be millions or billions of possible positions very quickly, and evaluating each position is what can benefit from parallelization. The only other notable data structure necessary is the board representation, and my implementation uses a 64 character string. Having a lightweight representation is important because when parallelizing the tree, it is helpful to have thread-private copies of the board on which to play candidate moves.

2 Approach

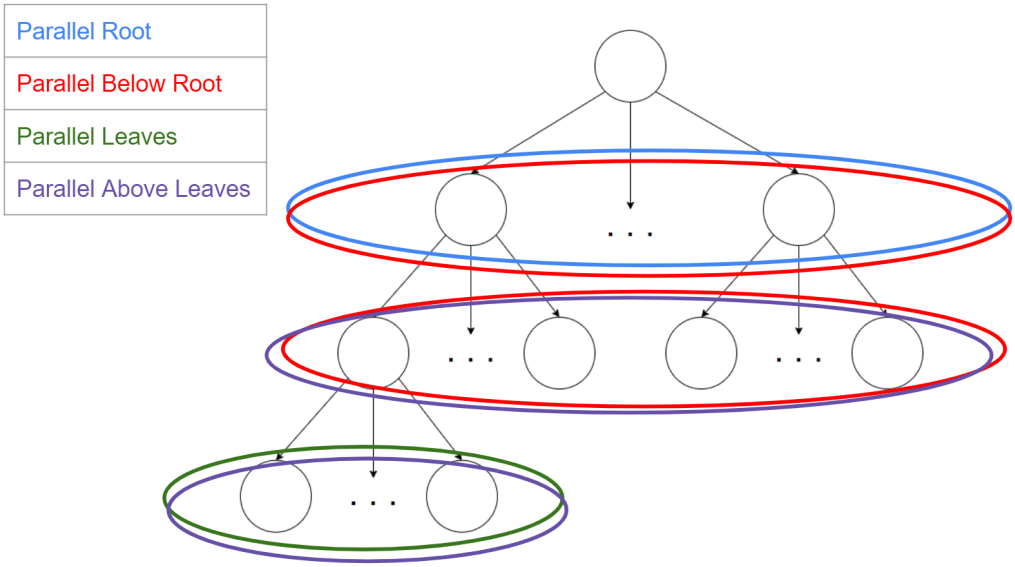
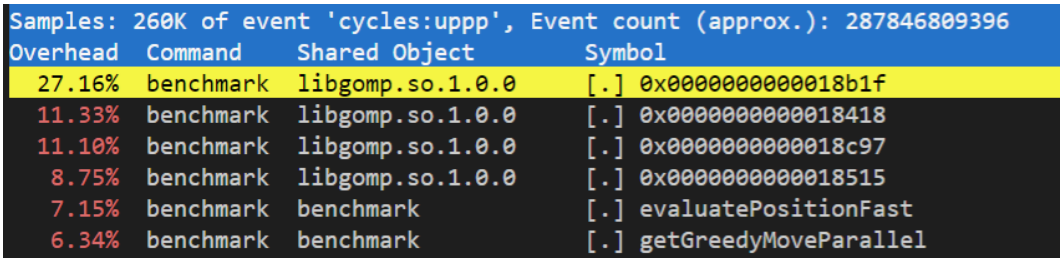


Figure 1: Representation of the chess game move tree, with circles highlighting the different depths at which parallelism occurs in different implementations.

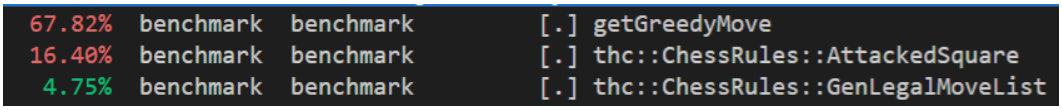
My implementation is written in C++ and leverages a lightweight API that implements the rules of chess (among other extended functionality that I did not use) called THC Chess Library. The most important functionality from the API that I used were the board representation, move input to that representation, and the `GenLegalMoveList` function. I did development on the GHC clusters and evaluated on the 64 core PSC machines. The engine uses a minimax search with alpha-beta pruning. The pruning allows the sequential version to search around two levels deeper in the same amount of time as a search without pruning. Alpha-beta pruning should benefit the sequential algorithm more than the parallel algorithm since branches that could be pruned might be running concurrently with the branch that caused the prune. The best parallel solution uses dynamic scheduling and parallelizes over the first set of candidate moves (i.e. direct children of the tree root). As the results show, parallelism any deeper in the tree resulted in the overhead of setting up threads outweighing the parallelism benefit. To arrive at this final strategy, I tried a few different parallelism strategies. Figure 1 shows the differences between the different parallel implementations. Code for the parallel over root children implementation is included in the appendix.

The main overheads of parallelizing a level of the search tree are creating copies of the game board so worker threads can play and evaluate moves without interference by other threads, atomically



```
Samples: 260K of event 'cycles:uppp', Event count (approx.): 287846809396
Overhead  Command    Shared Object        Symbol
  27.16%  benchmark  libgomp.so.1.0.0     [.] 0x0000000000018b1f
  11.33%  benchmark  libgomp.so.1.0.0     [.] 0x0000000000018418
  11.10%  benchmark  libgomp.so.1.0.0     [.] 0x0000000000018c97
   8.75%  benchmark  libgomp.so.1.0.0     [.] 0x0000000000018515
   7.15%  benchmark  benchmark            [.] evaluatePositionFast
   6.34%  benchmark  benchmark            [.] getGreedyMoveParallel
```

Figure 2: This is the perf capture of the parallel leaves implementation at depth 6 with 8 threads.



```
  67.82%  benchmark  benchmark          [.] getGreedyMove
  16.40%  benchmark  benchmark          [.] thc::ChessRules::AttackedSquare
   4.75%  benchmark  benchmark          [.] thc::ChessRules::GenLegalMoveList
```

Figure 3: This is the perf capture of the parallel root implementation at depth 6 with 8 threads.

setting the best move, alpha, and beta for the level of the tree being computed in parallel, and of course setting up the worker thread contexts by cloning the master thread's state.

Parallel over the leaves of the tree To avoid missing out on the benefit of alpha-beta pruning, I tried parallelizing the leaves of the search tree since the issue of large, pruned branches being computed doesn't exist when the large branches are run sequentially. However, this led to the cost of setting up the context for each worker thread outweighing the benefit of the parallel computation. The parallel algorithm was slower than the sequential algorithm in every case. As seen in Figure 2, most of the execution time is spent setting up worker thread contexts.

Parallel directly above the leaves This strategy uses half of the workers to compute the 2nd to last level of the tree, and 2 workers to compute the leaves for each of the parent workers. This suffers from the same problems as the parallel over leaves strategy, but to a lesser degree since fewer thread contexts need to be setup. The parallel algorithm was slower than the sequential algorithm in every case.

Parallel over the root children This was the first implementation that I tried, and ended up being the fastest solution out of the approaches I tried. The reduced benefits of alpha-beta pruning with parallelism near the root ended up not being an issue since most of the pruning happens a few levels into the tree, which is local to a worker thread in this approach. This approach also doesn't maximize parallelism when the number of threads is less than the number of root children, which was explored by the next solution.

Parallel over the root children and grandchildren This solution attempted to ensure that all worker threads were used by parallelizing over the grandchildren of the root in addition to the direct children, but the speedup was lesser than when only parallelizing over the direct children. This is likely due to the number of thread contexts needing to be setup increasing by an exponential factor, but also because of an atomic step needed to update the best move in the parallelized loop that also happens an exponentially larger number of times.

I additionally considered parallelizing the step of the algorithm where the list of candidate moves is generated, since the API function is sequential. However, a perf report of the parallel engine (shown in Figure 3) showed that this was not a major bottleneck for the algorithm in comparison to scoring each of the board positions.

3 Results

The chess engine was evaluated by measuring speedup over the sequential algorithm in 3 different board positions: The first move of a chess game, the Najdorf opening, and a puzzle from `lichess.org`. The benchmarks have an increasing number of potential moves in the starting position. The graphical representation of these positions are shown in the appendix. The depth of the search was chosen to be 7 for the benchmarks with pruning and 4 for benchmarks without pruning, since the parallel version of the algorithm would find moves quick enough for a game against a human. The following tables report the runtime for the benchmarks at various thread counts. The charts show the speedup with respect to the sequential implementation. When no scheduler is specified, static scheduling is used to decompose the parallel for loop. Speedup is compared to the sequential implementation, which is identical to the parallel implementation with one thread.

N Threads	At Root	Below Root	At Root Dynamic	At Root Guided
1	1337	1340	1226	1326
4	619	870	420	456
16	337	868	316	310
64	323	868	344	314

Table 1: Runtime (in ms) of the chess engine for benchmark 0 at depth 7.

N Threads	At Root	Below Root	At Root Dynamic	At Root Guided
1	44912	45202	41517	44920
4	16975	26548	12368	17402
16	10108	26310	7868	10259
64	8703	26091	9002	8525

Table 2: Runtime (in ms) of the chess engine for benchmark 1 at depth 7.

N Threads	At Root	Below Root	At Root Dynamic	At Root Guided
1	93924	94609	87054	94615
4	41794	65673	25904	40513
16	22297	64999	20127	22859
64	23556	66081	22105	22987

Table 3: Runtime (in ms) of the chess engine for benchmark 2 at depth 7.

The benchmarks show that the parallel engine using parallelism over the first set of candidate moves (At Root) with guided or dynamic thread scheduling has the best speedup. It is unable to achieve the optimal speedup due to the cost of setting up thread contexts, creating private copies of the game board for worker threads, and the need for synchronization to update the current best move, alpha, and beta.

Interestingly, alpha-beta pruning was not impacted by the final parallelism strategy much. It seems like moves near the leaves were pruned more than those near the root, so all the pruning ended up being local within a worker thread rather than parallel across workers which the algorithm would have missed. This only happend when depth was odd. When depth was even, the pruning did make a slight difference, but this difference disappears when normalizing the runtime by the number of positions considered, as seen in Table 4. The move quality does not change, so I think this difference is due to the fact that when the opponent of the engine makes the last move in a position, evaluation will favor them for moves that sacrifice a piece on the engine's next move, thus forcing the engine to consider branches that would be pruned with even parity depth. However, speedup without pruning is still higher than with pruning, likely due to the shorter depth explored by each worker for the relatively larger problem size.

I believe that the choice of CPU as the target with the OpenMP abstract was a decent choice, although I think that a GPU architecture could possibly have been a better choice for a parallel architecture target. The communication cost of sending moves from the engine opponent to the engine would be higher, but there could be many more worker threads that might more effectively decompose the search tree. It would be very difficult to integrate pruning though.

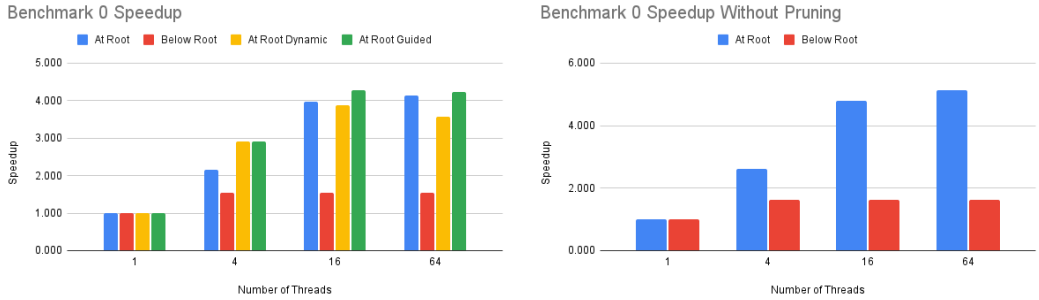


Figure 4: Benchmark 0 speedup of the below root implementation and the at root implementation with different scheduling policies, varying number of threads. Pruning searches to depth 7 and No Pruning searches to depth 4. There are 6,233,973 positions considered at depth 7.

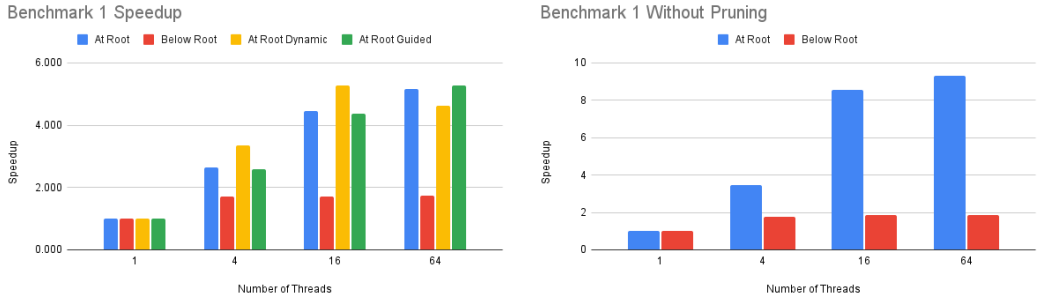


Figure 5: Benchmark 1 speedup of the below root implementation and the at root implementation with different scheduling policies, varying number of threads. Pruning searches to depth 7 and No Pruning searches to depth 4. There are 218,886,619 positions considered at depth 7.

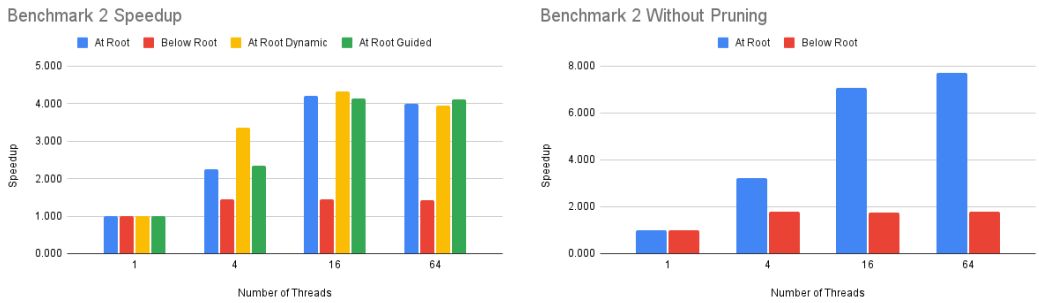


Figure 6: Benchmark 2 speedup of the below root implementation and the at root implementation with different scheduling policies, varying number of threads. Pruning searches to depth 7 and No Pruning searches to depth 4. There are 448,833,017 positions considered at depth 7.

N Threads	At Root	Below Root
1	1	1
4	2.68490414	1.670028511
16	4.893697453	1.705445103
64	4.928354622	1.697139364

Table 4: This table shows the speedup of the parallel algorithm when searching at depth 6, normalized by the number of positions considered by the algorithm.

References

Forster, Bill, https://github.com/billforsternz/thc-chess-library.

https://en.wikipedia.org/wiki/Alpha-beta_pruning.

Appendix

The following code is the implementation of the search for the parallel root implementation. Other implementations are mostly the same except for some conditionals that control the number of threads at each depth, and a version of `getGreedyMove` that uses an OpenMP parallel for construct. A note about the parallel for construct, because breaking out of a parallel for loop is not allowed in OpenMP, the threads set a flag that tells the workers they can continue without work when the branch is pruned rather than explicitly breaking.

```
double minMaxIterationParallel(thc::ChessRules cr, thc::Move &best_move, int curr_depth,
                               double alpha, double beta, int n_threads, bool white)
{
    if (curr_depth == 0)
    {
        return getGreedyMove(cr, best_move, 0, white);
    }

    thc::MOVELIST legal_moves;
    cr.GenLegalMoveList(&legal_moves);
    int best_move_idx = 0;
    if (legal_moves.count > 0)
        best_move_idx = rand() % legal_moves.count;
    double best_score = white ? -1000 : 1000;

    thc::Move local_best_move = best_move;
    volatile bool flag = false;
    if (white) // max
    {
        #pragma omp parallel for default(shared) private(local_best_move) num_threads(n_threads)
        for (int i = 0; i < legal_moves.count; i++)
        {
            if (flag)
                continue;

            thc::ChessRules local_cr = cr;
            local_cr.PushMove(legal_moves.moves[i]);
            double curr_score = minMaxIteration(local_cr, local_best_move, curr_depth - 1,
                                                alpha, beta, !white);
            local_cr.PopMove(legal_moves.moves[i]);
            #pragma omp critical
            {
                if (best_score < curr_score)
                {
                    best_move_idx = i;
                    best_score = curr_score;
                    alpha = (best_score > alpha) ? best_score : alpha;
                }
                if (best_score >= beta)
                    flag = true;
            }
        }
    }
    else // min
    {
        #pragma omp parallel for default(shared) private(local_best_move) num_threads(n_threads)
        for (int i = 0; i < legal_moves.count; i++)
        {
            if (flag)
                continue;
```

```
51
52              thc::ChessRules local_cr = cr;
53              local_cr.PushMove(legal_moves.moves[i]);
54              double curr_score = minMaxIteration(local_cr, local_best_move, curr_depth - 1,
55                                                  alpha, beta, !white);
56              local_cr.PopMove(legal_moves.moves[i]);
57              #pragma omp critical
58              {
59                  if (best_score > curr_score)
60                  {
61                      best_move_idx = i;
62                      best_score = curr_score;
63                      beta = (best_score < beta) ? best_score : beta;
64                  }
65                  if (best_score <= alpha)
66                      flag = true;
67              }
68          }
69      }
70
71      best_move = legal_moves.moves[best_move_idx];
72      return best_score;
73  }
74
75  double minMaxIteration(thc::ChessRules &cr, thc::Move &best_move, int curr_depth,
76                         double alpha, double beta, bool white)
77  {
78      if (curr_depth == 0)
79      {
80          return getGreedyMove(cr, best_move, 0, white);
81      }
82
83      thc::MOVELIST legal_moves;
84      cr.GenLegalMoveList(&legal_moves);
85      int best_move_idx = 0;
86      if (legal_moves.count > 0)
87          best_move_idx = rand() % legal_moves.count;
88      double best_score = white ? -1000 : 1000;
89      double curr_score;
90
91      if (white) // max
92      {
93          for (int i = 0; i < legal_moves.count; i++)
94          {
95              cr.PushMove(legal_moves.moves[i]);
96              curr_score = minMaxIteration(cr, best_move, curr_depth - 1, alpha, beta, !white);
97              cr.PopMove(legal_moves.moves[i]);
98              if (best_score < curr_score)
99              {
100                 best_move_idx = i;
101                 best_score = curr_score;
102                 alpha = (best_score > alpha) ? best_score : alpha;
103             }
104             if (best_score >= beta)
105                 break;
106         }
107     }
108     else // min
109     {
```

```
        for (int i = 0; i < legal_moves.count; i++)
        {
            cr.PushMove(legal_moves.moves[i]);
            curr_score = minMaxIteration(cr, best_move, curr_depth - 1, alpha, beta, !white);
            cr.PopMove(legal_moves.moves[i]);
            if (best_score > curr_score)
            {
                best_move_idx = i;
                best_score = curr_score;
                beta = (best_score < beta) ? best_score : beta;
            }
            if (best_score <= alpha)
                break;
        }
    }

    best_move = legal_moves.moves[best_move_idx];
    return best_score;
}

double getGreedyMove(thc::ChessRules &cr, thc::Move &best_move, int max_depth, bool white)
{
    thc::MOVELIST legal_moves;
    cr.GenLegalMoveList(&legal_moves);
    int best_move_idx = 0;
    if (legal_moves.count > 0)
        best_move_idx = rand() % legal_moves.count;
    double best_score = white ? -1000 : 1000;
    double curr_score;

    int i;
    for (i = 0; i < legal_moves.count; i++)
    {
        cr.PushMove(legal_moves.moves[i]);
        curr_score = evaluatePositionFast(cr);
        cr.PopMove(legal_moves.moves[i]);
        if (white)
        {
            if (best_score < curr_score)
            {
                best_move_idx = i;
                best_score = curr_score;
            }
        }
        if (!white && (best_score > curr_score))
        {
            best_move_idx = i;
            best_score = curr_score;
        }
    }
    best_move = legal_moves.moves[best_move_idx];
    return best_score;
}
```

The following chess positions are the initial positions for each benchmark in the results section.

Figure 7: Position for benchmark 0. White to make the first move of a game.

Figure 8: Position for benchmark 1. White to move in the Najdorf opening

Figure 9: Position for benchmark 2, which is a puzzle from `lichess.org`. White to move and win a pawn with Nxd5.