Parallel Optimizations to the Minimax Algorithm for Connect 5

Aleksei Seletskiy, William Wang

Project URL: https://thetazero.github.io/15418-Final-Project/

Summary

We are parallelizing a minimax engine for the game of Connect 5. The 2 main components that we parallelized are the tree search and the board evaluation function. The tree search is parallelized by using OpenMP to search over different moves/branches of our tree in parallel. The evaluation function is a data-parallel task that we implemented in parallel using both ISPC. We evaluated boards in parallel using CUDA. We compared the performances of a sequential engine with sequential evaluations with the different elements of parallelism incorporated into our search.

Background

Connect 5 is a very simple game, where two players play on a NxN board (typically N=19), alternating moves in an attempt to get 5 pieces in a row in either a row, column, or diagonal. The minimax search is a well-known algorithm for searching through game trees of zero-sum games in an attempt to find the best move. It uses an evaluation function that takes in any arbitrary position as an input and outputs the evaluation score of that position. The search branches out over a set of candidate moves until a certain depth, or *plies*, where the resulting positions are evaluated using the evaluation function. From there, the evaluation scores are propagated up the tree. Each depth of the tree alternates between being a minimizing layer, where we select the move that leads to the position with the lowest evaluation, and a

maximizing layer, where we select the move that leads to the position with the highest evaluation. This can be an expensive task, as if we have a branching factor of *b* moves in every position for a depth of *d* plies, we could be searching on the order of O(b^d) moves. Thus, this algorithm could benefit a lot from searching those moves in parallel.

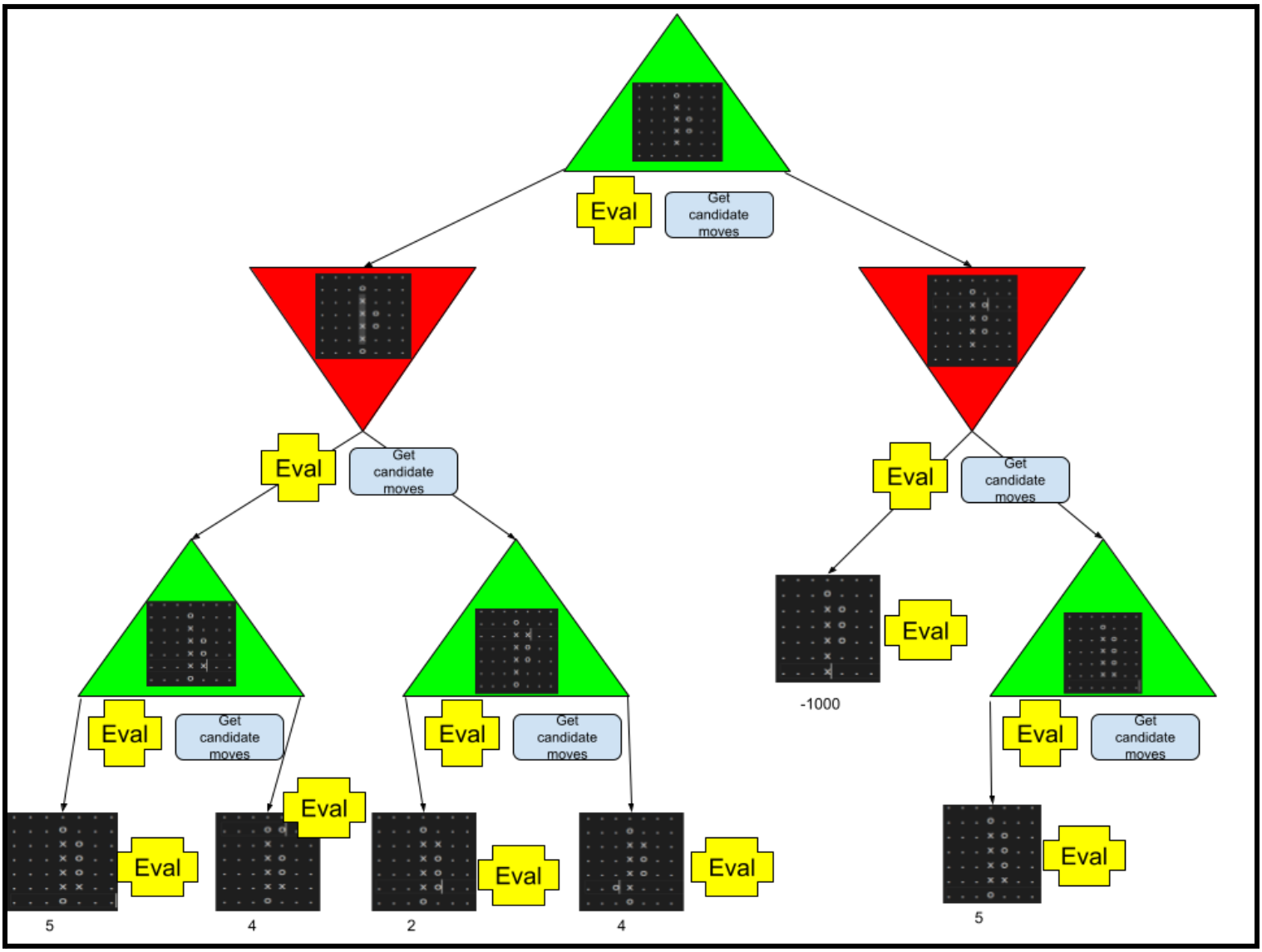
Another element of the minimax search we implemented was alpha-beta pruning. This is a simple algorithm that allows us to prune, or essentially stop searching, branches that are guaranteed to not be relevant in the search. While this helps the sequential version of our search, this would actually reduce the speedup of our parallel search, as pruning has large dependencies on the evaluations of other branches, so when done in parallel, it is likely certain branches that would normally be pruned would not be pruned. However, we can adjust what depth we begin our parallel search to help maximize the pruning in parallel.

For our evaluation function, we essentially loop through each tile of our board to check if it is empty, and what the maximum number of consecutive tiles would be should the tile be filled by a player. This is a very data-parallel task, as for each tile, we perform the same search in every direction to count how many adjacent pieces of the same color border the specific tile. In our evaluation function, we also keep track of the empty tiles that have the most adjacent like-colored tiles, so that when we perform our search, we search those tiles first. As our board is a simple array, using a data-parallel model is logical as it allows for leveraging of vector instructions.

Approach

We parallelized two aspects of our minimax game search: the tree search, which is almost task-parallel, and our evaluation function, which is data-parallel. Our basic minimax algorithm is an extension of the one found here : (https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/).

The figure below illustrates a simple example of how our version of the minimax tree search works.



At each node of our search tree, we call on our evaluation function, which helps us prioritize which squares we should search first. For example, at depth 1, we would like to prioritize the move to stop player x from winning right away, so the first move player o plays would be to block x's prospective 4-in-a-row. This allows us to prune the bottom right branch of our tree using alpha-beta pruning.

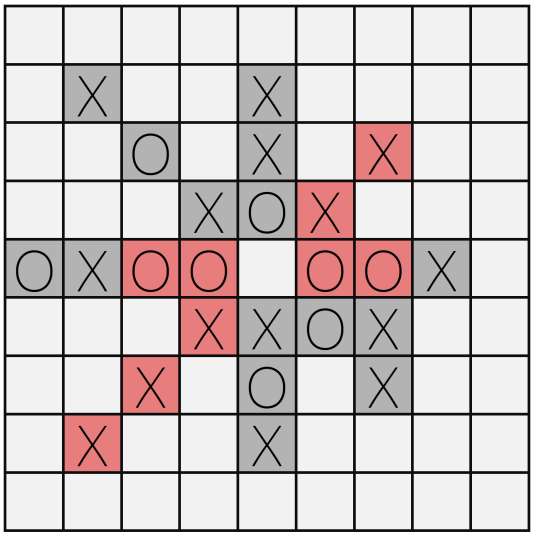
We used a combination of ISPC and CUDA to improve our evaluation function performance, and openMP to improve the tree search performance. We tested our implementations on the Gates machines.

ISPC Evaluation Function:

With so many evaluations during our search, intuitively, it would make sense to parallelize our evaluation function. This is also justified by our basic profiling, which shows that evaluation takes up the majority of our search time (around 70-90% of the time).

The first way we went about parallelizing the evaluation function was by using ISPC code. Our sequential evaluation algorithm consists of a few steps:

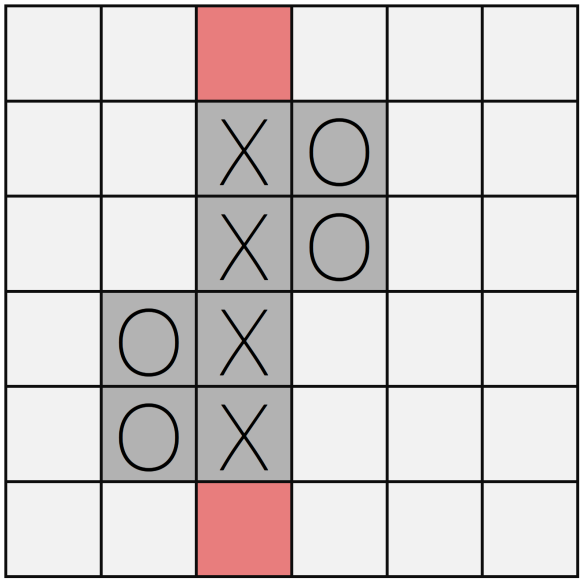
1.) Loop through every tile in the board within a region that is at most T tiles of the outermost filled tiles on the board.
    a.) If the tile is filled, we check each direction to see if there are 4 other like-colored tiles in a row, which would signify the game being in a terminal state as one player wins
    b.) If the tile is empty, we summarize it by its maximum potential, for example if we are evaluating the center tile of the following board, the maximum potential is 6 x's and 5 o's. If we place an x we form a 6 in a row, if we place an o we form a 5 in a row.



c.) For the sequential implementation, as we evaluate, we put the empty tiles that can potentially complete a 5-in-a-row in a hashset. Then, when we call our get_candidate_moves() function, we first add all of those moves to a vector. Then, we add the remaining empty tiles within our region of interest to the vector. That way, we would search the critical moves first before searching the other empty squares, which should improve pruning.

d.) We assign each empty tile summary a score, and then sum them all up for the final board evaluation score.

2.) If in the previous step there are two tiles with the potential to form two 5 or more in a row of the same type, we evaluate the board as inevitably winning. This case is shown below.



This algorithm translates relatively well to ISPC code from our sequential version, but there were a few important changes that increased the speedup of our ISPC version.

First of all, for the step where we add critical moves to a hashset, this was not possible to do with ISPC, as ISPC does not have hash sets. Thus, instead, we implemented an array where we mark every tile in our region with a 1 if it is critical, and else 0. Originally, we would then add

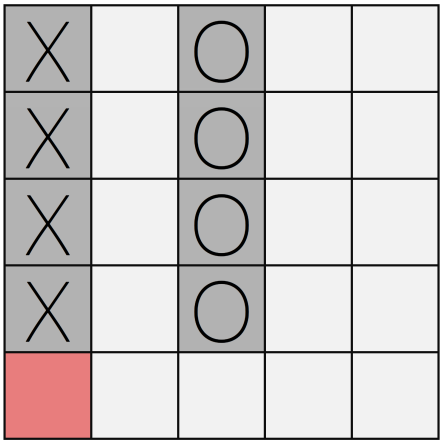
all of the marked tiles to a hashset after the ISPC portion, but that is a sequential element of code, so instead, we eliminated that part and would just loop through our array of marks to determine which moves are critical.

Another change we made in the sequential code was to eliminate some branch divergence between checking a filled tile and an empty tile. In our sequential version, as stated in 1a and 1b above, we have two cases that call two separate functions, which is logical for the sequential version, as in case 1a, checking for 5 in a row can be done faster using its own function. However, in the ISPC version, having an if-else of two long routines would cause branch divergence, and may in fact lead to both of them being called and then being masked. Thus, we modified our code to reduce branch divergence.

In our case, using task-based ISPC across multiple cores was not necessary, as when we integrate our ISPC with OpenMP, each processor will be busy with its own search and its own set of boards to evaluate already, so it made sense for us to just use one processor for ISPC.

OpenMP:

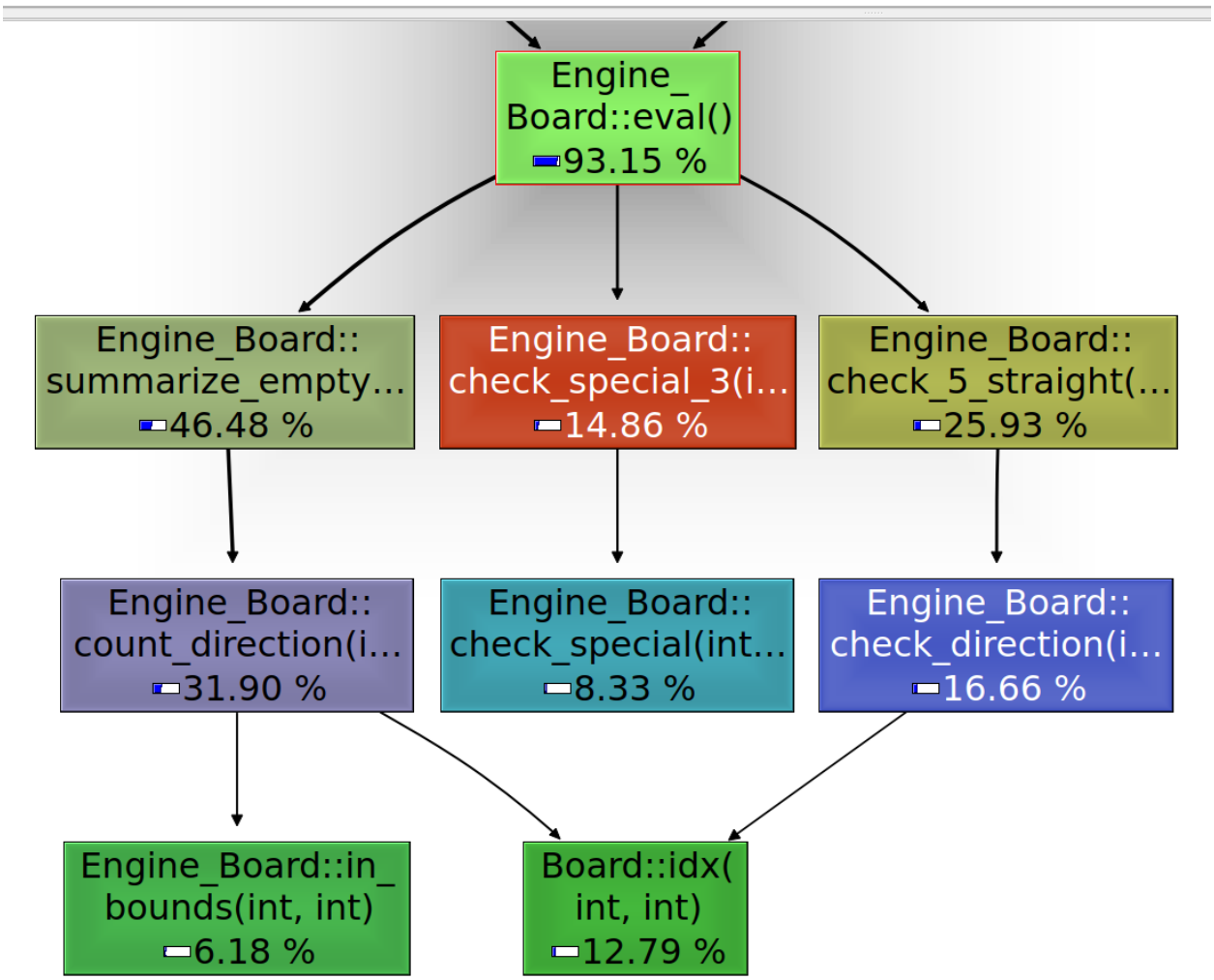
Naively parallelizing over all possible moves does not work very well, for easily winning cases like the following:



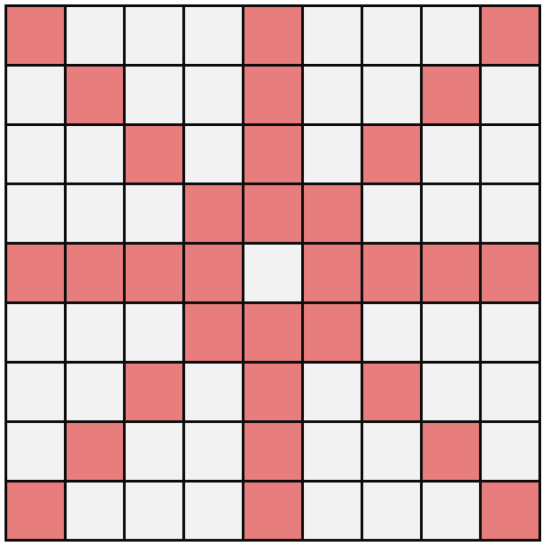
Our sequential implementation would try the winning move early, causing the rest of the moves to not be explored. However, a naive parallel implementation would continue searching for other winning paths, long after it finds the easily winning position. Therefore we still need to implement pruning. We implemented pruning by creating a shared flag that, once set to true, skips the computation. This allowed us to achieve speedup in all cases that were tested, given they weren't on the order of ~10ms (in this case start-up time dominated).

Updated Version of Evaluation:

We tried running valgrind profiler to determine the bottlenecks in the sequential implementation are almost all in the evaluation function, and determined that a large fraction of the work is essentially counting the number of tiles in a row.



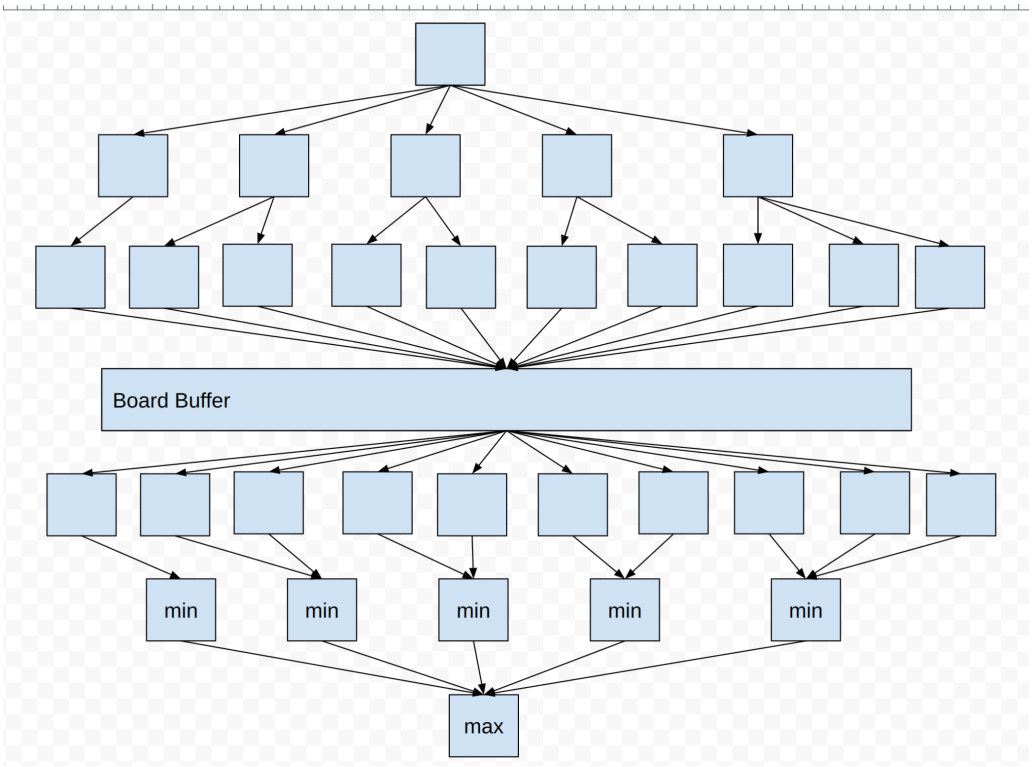
Because of this we created a new version of the evaluation function with a different access pattern. The old pattern was for each tile, search up to five tiles in each direction.



This theoretically means that we can count a tile up to five times, when checking the same direction. Instead we changed the algorithm to scan over the board in each direction once, then combine the results together. Due to time constraints we only created a CUDA implementation of this.

CUDA Evaluation:

The cuda implementation uses the updated scan based evaluation algorithm. This algorithm is far less branch heavy, which should improve performance for CUDA. Due to time constraints, we were unable to integrate the cuda evaluation algorithm into minimax in a reasonable manner. Sending boards one at a time to be evaluated by the GPU is incredibly bad due to kernel startup latency. Instead we search up to a fixed depth for all possible reachable states, and then process all the generated boards at the same time with CUDA. Then we take these results and reduce them using the minimax algorithm.

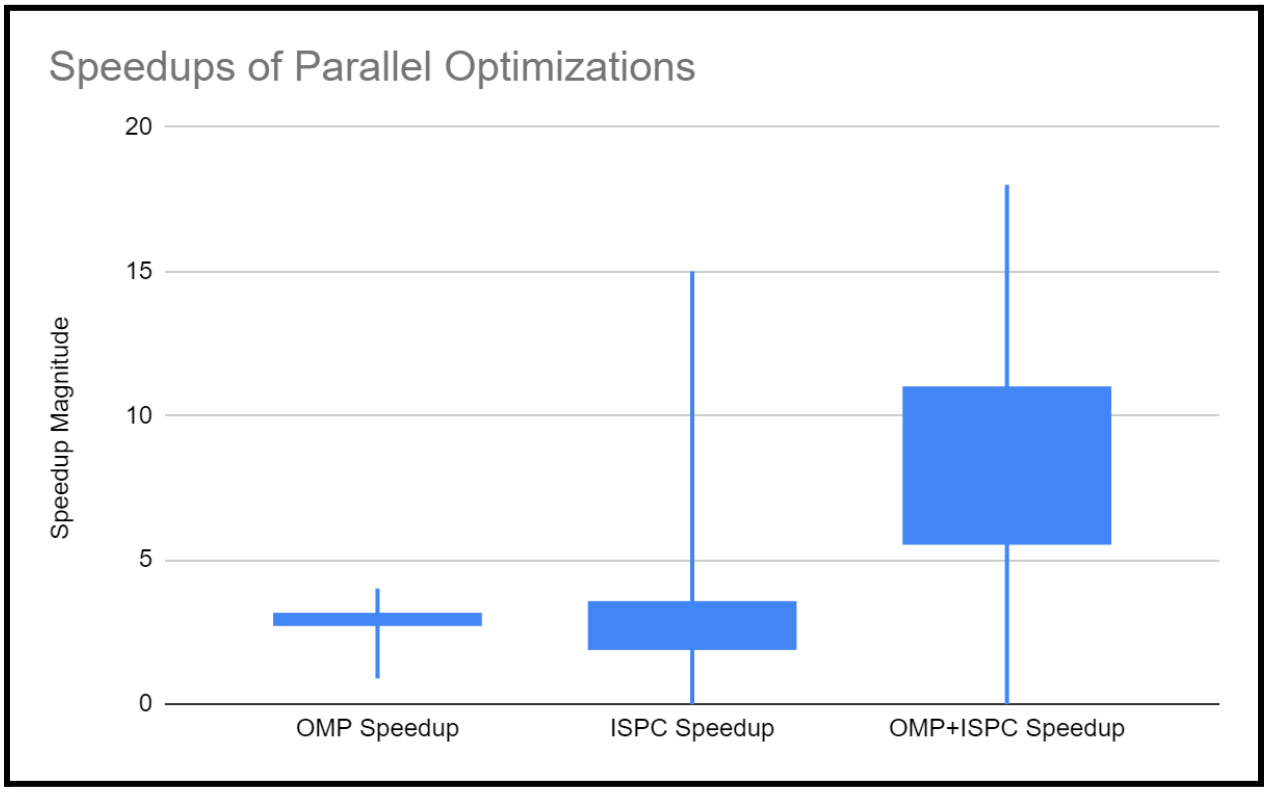


Results

We tested our performance across 60 test positions. These positions were taken from a mix of real and simulated (engine vs. engine) games on 19x19 boards, and range from early in the game, where there are few pieces and the search region is small, up to positions with 40 pieces on the board, where the overall region that is enclosed by pieces is close to half of the board. In addition, we had positions where pieces were sparse and positions where pieces were dense. We tested on a depth of 3, as depths longer than 3 took a really long time to run. For each position, we would run the purely sequential version with one thread, the version with sequential search but parallel evaluation using ISPC, the version with parallel search using OpenMP but sequential evaluation, and then the version with both OpenMP parallel search and ISPC parallel

evaluation. We would time the runtimes of the depth 3 search, and get the relative speedups of the parallelized versions with the purely sequential search/evaluation.

Below, we have the results of our performance profiling across the 3 combinations of adding parallel features.



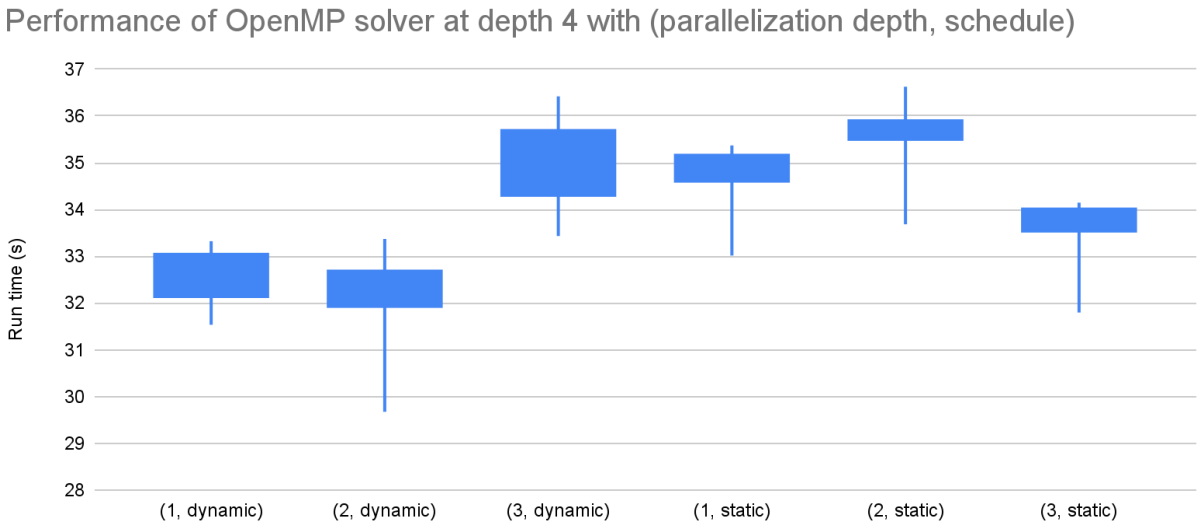
As can be seen, both OpenMP and ISPC led to speedups mostly between 2-4x, and thus, logically, when combining the two, we got speedups between 5-10x mostly.

Bottlenecks to Speedup: ISPC

For ISPC, we compiled it using SSE4, and given that we have 4-wide vectors, our speedups or around 3x meant we had relatively good utilization. The small amounts of divergent execution are probably the main bottleneck to speedup. The divergence mainly occurs when we stop counting in our sequential algorithm when a streak of consecutive pieces ends, but in our parallel version, the variation in when we stop counting leads to divergence.

Bottlenecks to Speedup: OpenMP

To improve our speedup over the sequential implementation we attempted to tweak the parallelization depth and schedule. We found that for searches of depth 3, a parallelization depth of 2 with a dynamic schedule tended to perform better.



The following is the output of running` perf stat` on our benchmarking code with and without open mp. The open mp case is run with the parameters we found to be optimal on the gates machines.

```
Performance counter stats for './Connect5_bench -e omp':

       222,046.52 msec task-clock                #    6.775 CPUs utilized
            2,222      context-switches          #   10.007 /sec
                2      cpu-migrations            #    0.009 /sec
              254      page-faults               #    1.144 /sec
  832,741,095,564      cycles                    #    3.750 GHz
2,056,847,967,333      instructions              #    2.47  insn per cycle
  254,971,221,313      branches                  #    1.148 G/sec
    2,738,630,549      branch-misses             #    1.07% of all branches

     32.775978163 seconds time elapsed
```

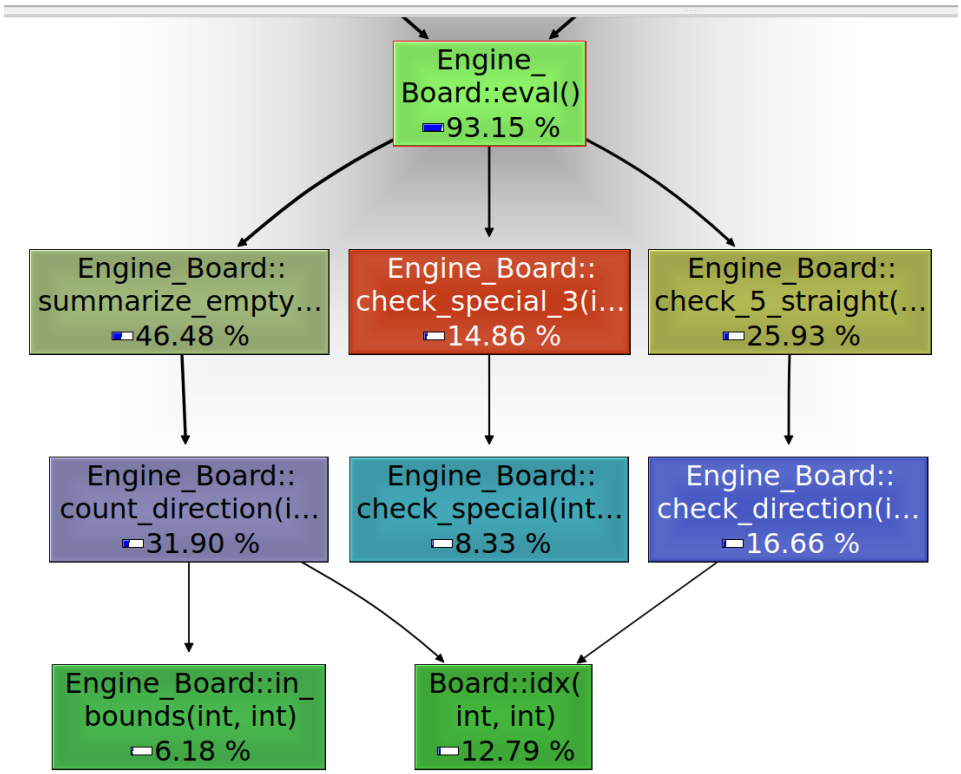
```
Performance counter stats for './Connect5_bench':

       140,472.85 msec task-clock                #    1.000 CPUs utilized
              760      context-switches          #    5.410 /sec
                0      cpu-migrations            #    0.000 /sec
              160      page-faults               #    1.139 /sec
  658,239,470,448      cycles                    #    4.686 GHz
1,671,435,139,809      instructions              #    2.54  insn per cycle
  206,389,684,756      branches                  #    1.469 G/sec
    2,211,895,731      branch-misses             #    1.07% of all branches

    140.476209319 seconds time elapsed
```

From this we see that despite solving the same workload, the open mp implementation used 25% more cycles. This is due to the overhead of parallelism (doing unnecessary work, cost of synchronization). Furthermore due to workload imbalance, only ~6.8 cpus were utilized. The speedup we'd expect to get from this is 6.8 * .75 = 5.1 which is pretty close to the actual 4.3x speedup.

By using valgrind on the sequential implementation we were able to see that the time spent in eval dominates the computation at around 93% of the time spent.



Overall, using the CPU with ISPC was a sound choice, and we got a substantial speed up from this. The GPU implementation struggled due to the inherent trade of between increasing the amount of kernel start ups and amount of pruning, we unfortunately ran out of time to explore this..

References

- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

Work done by each student/distribution of credit

Overall: Roughly 50/50 distribution

Aleksei Seletskiy

- Minimax implementation
- OpenMP implementation of minimax
- CUDA board evaluation function implementation (this does work)
- CUDA implementation of a heavily modified “minimax” (still having minor issues preventing it from working)
- Evaluation function design
- Common case for evaluation function
- Minor Board/Engine_Board updates: undo_move, constructor for inline instantiation, minor bug fixes.
- Testing infrastructure
    - Verifies consistency of various minimax implementations
    - Verifies that evaluation function is acting as expected
    - Verifies engine_board class behavior on a variety of cases
- Benchmarking program where you can pass engine variant and depth to search at
- Profiling sequential implementation with valgrind
- Explanation of evaluation function
- Benchmarking OpenMP implementation with different parameters to choose the optimal ones.

William Wang

- Implementation of overall architecture of code base
    - the OOP design of the board, engine, etc…
    - Infrastructure to gather metadata about our performance, such as timing our overall search time, time spent on the evaluation function, and number of branches pruned
- Implementation of sequential Evaluation Function
- Testing/debugging of sequential function to see if function evaluates various boards correctly
- Integration of Sequential Evaluation and Minimax Implementation
- Testing/Debugging of Sequential version of minimax on various test boards to ensure algorithm
- Profiling to determine bottleneck of sequential program (shows it was much of evaluation)
- Benchmarking program where you can pass engine variant and depth to search at
- Design/Implementation/Debugging of ISPC Evaluation Function
    - Included having to modify algorithm to reduce amount of code that was sequential
    - Modified code to reduce divergence in the ISPC program
    - Tuned various parameters such as board type from int8 to int32 to increase speedup per the ISPC user guide
    - Experimented with for loops with uniform int, foreach loops, and foreach_tiled loops
    - Profiled performance to evaluate the speedup
- Integrated ISPC code with OMP implementation