Final Project: Playing Connect-4 with Parallelism

Howard Chen, Elizabeth Ji
hhchen@andrew.cmu.edu, emji@andrew.cmu.edu
May 13, 2017

1 Summary

We parallelized a state-scoring program for Connect-4, using various parallel frameworks. We created implementations using OpenMP, CUDA, and pthreads, and compared their performance against a sequential baseline, finally using DFS and pthreads to acheive a speedup of roughly 4x.

2 Background

2.1 Introduction

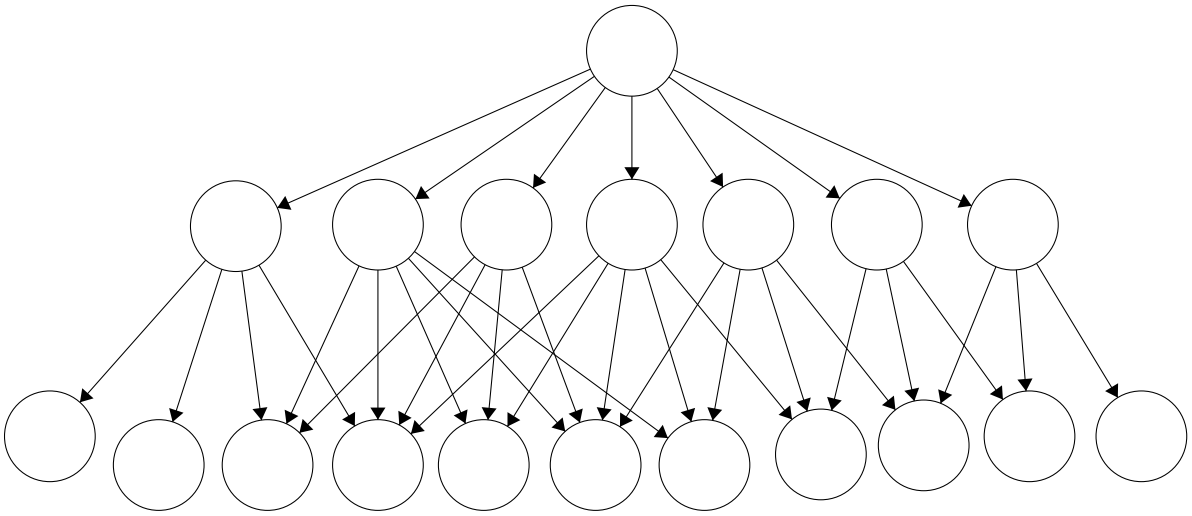
The original program we decided to speed up is a Connect-4 move selection program. Given a starting position, the program selects the best possible move as given by the Minimax algorithm. At a high level, it explores all possible moves to a fixed depth, scores them with a heuristic, and then uses Minimax to score its "ancestor" positions. This creates an implicit graph structure, with all nodes depending on their children's scores. Note that states can be reached via different sequences of moves, so the graph is unfortunately not a tree.

2.2 Data Structures

Ideally, we wish to score each state exactly once, and not recompute its score every time we encounter the node during our traversal. This naturally creates a need to hash states so that we can efficiently look up scores we have already computed. In a parallel setting, we need to have a thread-safe hash table implementation to ensure correctness. However, this particular algorithm only needs to insert each state once, and we never need to delete entries. Thus, we were able to implement a simple lock-free hash table that supports the `get(state)`, `put(state, score)`, and `count(state)` operations in a thread-safe manner. The hash table uses separate chaining, where linked-list insertion is implemented using compare-and-swap operations. This implementation avoids contention in cases where simultaneous insertions hash to different bins. When the load factor is low (i.e. < 1) we achieve very high throughput. Specific numbers are detailed in the results section.

2.3 Workload

A typical state graph will look something like the one below, where nodes represent states and arcs represent possible moves (and Minimax dependencies):



Each level of nodes is completely independent, with the lowest level containing the most nodes. Furthermore, the heuristic functions typically involve fair amounts of arithmetic, and can be computed without communication. Thus, there are natural opportunities for parallelism in computing the lowest level of nodes. However, for nodes further up in the graph, we need to compute all dependent values and be able to look them up efficiently.

3 Approach

The structure of the state graph leads to two natural approaches to exploring/scoring the states: breadth-first and depth-first scoring. We implemented sequential versions of both approaches, and then used the above listed parallel frameworks to try to speed them up. All implementations were written in C++.

3.1 BFS

We tackled the parallelization of the BFS solution first, because each frontier provided clear opportunities for parallelization. The structured, iterative nature of BFS prompted us to try using OpenMP and CUDA.

Our OpenMP implementation scored each new frontier in parallel. It used the lockless hash map to ensure that boards were only added to the frontier once, and also as a central lookup table for scores.

We attempted to use CUDA for scoring frontiers in parallel as well. However, we decided that having all CUDA threads modify one shared hash table was not a performant solution, so we focused on scoring the deepest level of states (using a heuristic).

3.2 DFS

DFS is not inherently parallel, but it is still possible to have simultaneous runs of DFS work in parallel, sharing computed results. This sharing is done by updating one thread-safe hash table, storing the scores of already-visited states. We implemented this parallel version both in OpenMP and with pthreads. These two solutions were fairly similar. Both computed a small frontier (i.e. to depth 1 or 2), and scored the frontier in parallel using DFS on each state.

3.3 Other

We also tried a hybrid OpenMP and CUDA solution, using OpenMP to compress the boards before sending them to the GPU. However, this was still slower than just using OpenMP, so we did not pursue it further. We also attempted parallel DFS to a depth of 2, giving 49 starting states. However, this did not show speedup over a depth of 1 (7 starting states), so it was also not pursued further.

4 Results

All of the following results were collected by running on the GHC machines (16 cores), starting with the empty starting board. Overall, DFS-based solutions performed 1.5-2.5 times better (in terms of runtime). This is likely due to the need for BFS-based solutions to materialize and de-duplicate the entire frontier at each level. The parallel solutions achieved noticeable, but non-linear speedup compared to the baselines. We will now discuss the results of the BFS and DFS solutions separately.

4.1 BFS

The OpenMP solution ended up being fairly successful, beating the sequential solution at all depths we tested. The table below summarizes runtimes on various search depths:

Search depth	States	Sequential time (ms)	OpenMP time (ms)
3	294	10.7	6.7
5	6577	147.9	45.2
7	76965	1380.5	376.1
9	825432	14888.1	3830.6

This table shows significant but non-linear speedup. This is likely due to a number of factors. The first is high contention over inserting into the hashmap at each iteration of BFS. All threads will likely be inserting elements at the same time, and little arithmetic is actually being done after the deepest frontier is scored. Additionally, the threads are subject to a fair amount of synchronization overhead due to the iterative nature of BFS, where one frontier is scored at a time.

4.2 DFS

The table below summarizes runtimes on various search depths:

Search depth	States	Sequential time (ms)	OpenMP time (ms)	Pthread time (ms)
3	294	6.0	8.7	11.7
5	6577	83.6	31.6	24.2
7	76965	719.0	2 16.3	159.4
9	825432	7736.9	2046.8	1794.4

We can see that the speedup is also non-linear here. The primary causes of this were workload imbalance and increases in the total work performed. The workload imbalance comes from the fact that every thread handles a different initial move. Since any winning position is considered terminal (and has no children), different initial moves will have different subgraphs rooted at the move. Printing the number of states scored by each thread indicates that the discrepancy can be as large as a 2x difference (varying between runs). The additional work comes from multiple threads computing scores for the same states. Since each state is only added to the hash table when all of its children's scores have been computed, multiple threads can try to compute the score of the same state. This leads to more hash table lookups, more attempted insertions, etc.. On a run to a depth of 7 (76,965 states), the total number of states visited by all 7 threads was 164,724. This is more than a 2x increase in additional work, which explains a good portion of the imperfect speedup.

4.3 Lockless Hashmap

An additional point of interest is the performance of the lockless hashmap. We tested the throughput of the lockless hashmap compared to the standard library's `unordered_map`. Specifically, we inserted large numbers of random keys in parallel (with OMP), and compared to the speed of sequentially adding the keys to the standard map. The results are summarized in a table below:

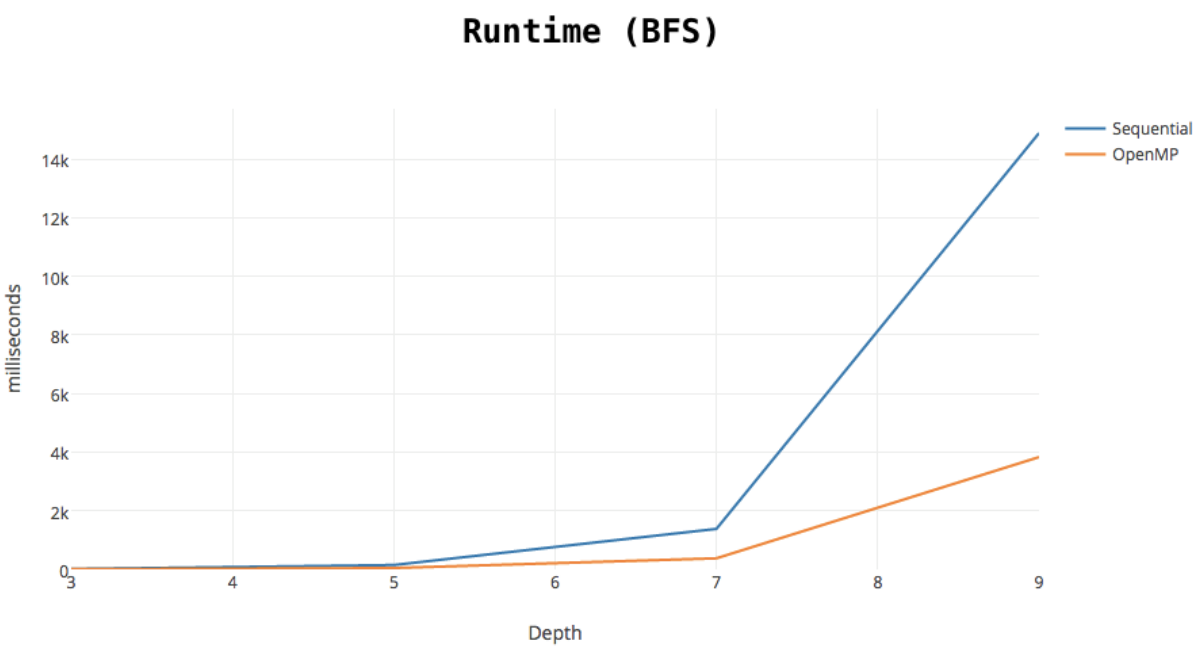
Number of keys	`unordered_map` + Sequential (ms)	Lockless + OpenMP (ms)
1,000	1.0	1.8
10,000	6.3	2.4
100,000	18.9	11.7
1,000,000	173.4	64.5
10,000,000	1000.1	232.3
100,000,000	3373.42	545.78

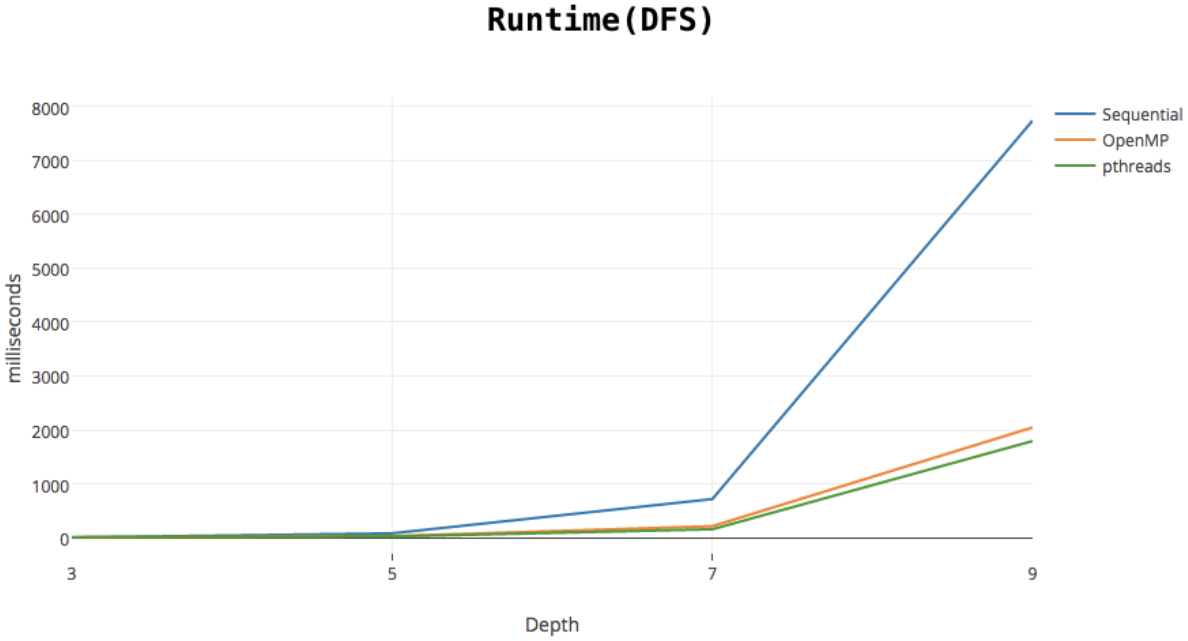
This speedup highlights the potential gain in performance when hashmap insertion is highly contended, as compared to a `std::unordered_map` protected by a critical section (which is at least as slow as completely sequential insertion).

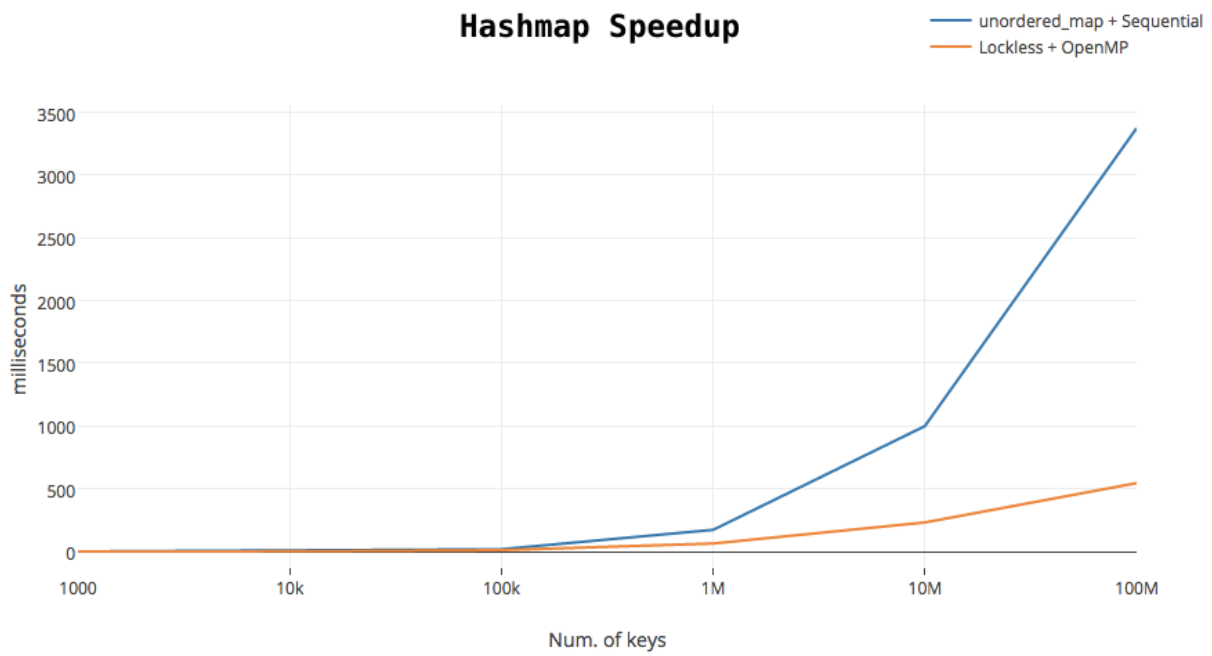
4.4 Graphs

Runtime graphs for our approaches included below:

Howard Chen, Elizabeth Ji
hhchen@andrew.cmu.edu, emji@andrew.cmu.edu







5 List of Work

Equal work was performed by both project members.