Parallel 2048 Solver

Wei-Ting Tang
Carnegie Mellon University, PA
weitingt@andrew.cmu.edu

I. Summary

We implemented a parallelized solver for a 2048 game using OpenMP with various parallel strategies and performed speedup tests for 300 simulation steps on the PSC machine using a different number of threads. Our final solver can reach about 10x speedup than the sequential solver.

II. Background

2048 game (1) is a sliding tile puzzle game on a 4 x 4 board, and the objective of the game is to combine the numbered tiles to create higher tiles as large as possible.

The workload of the 2048 solver (2) is computation-intensive. The prediction steps include sliding the board in four directions, generating new ”2” and ”4” tiles at the empty positions, computing heuristics scores for current positions, and deciding the optimal next move. It creates at most 4 (four directions) * 2 (”2” or ”4”) * 15 (empty positions) = 120 possible searching paths for each step, and the algorithm recursively evaluates the scores for 8 levels, which means we need to go through a huge search space.

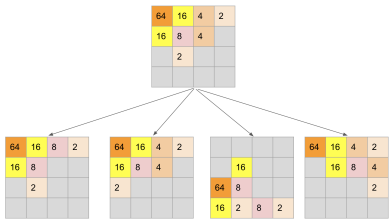

Fig. 1. 2048 solver moving the board

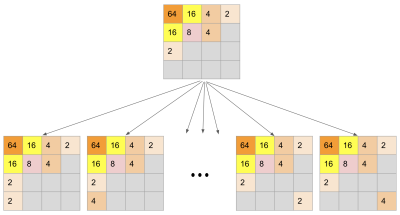

Fig. 2. 2048 solver generating tiles

Our implementation is based on the 2048 solver repo (3) implemented in C++ by nneonneo, and parallelizes the simulation steps using the OpenMP task feature. The board is stored as the type of uint64 to perform fast board operations, and the heuristic scores are float type calculated by the monotonicity and smoothness of the board state.

Since the simulation steps only depend on the previous board state, it can potentially benefit from parallelization. For each recursive call, we pass the board state by value so that all threads can simulate and evaluate the boards independently. The only synchronization needed here is to collect and compare the results after all threads are done.

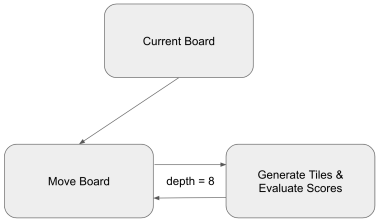

Fig. 3. 2048 solver search process.

III. Approach

The original implementation (3) presents the board simply as a uint64, which minimizes the computational cost for board operations. Therefore, we focused on parallelizing the main evaluation algorithm and kept the data structure and board operators the same.

We used the original implementation as our sequential benchmark, estimating the execution time for 300 moves, and implemented our parallel version in C++, which launches a new OpenMP task when we are going to move the board and generate new tiles, running on the multi-core PSC machine.

In the following sections, we are going to introduce three approaches we have implemented.

A. Approach 1

Our first approach is a **naive** strategy that only parallelizes the most-outer loop. This is the most intuitive way for this specific workflow, which starts at the beginning board state and independently calculates the scores for four different directions. It gives us a good insight into how well the program can perform on a 4-core machine.

Although this approach suffered from significant workload imbalance, it still reached a 2x speedup than the serial benchmark.

To enable high-count threads acceleration, we need to parallelize the recursive evaluation steps further.

B. Approach 2

The number of tasks grows exponentially regarding the simulation depth. In the worst case, there are $4*2*15 = 120$ (sliding four directions, generating 2 and 4 at every empty positions) recursive calls per level. Thus, for depth = 8, it needs $4 * 120^8 \approx 1.72 * 10^{17}$ tasks to simulate in move.

In this approach, we implemented a **dynamic** strategy. We kept tracking the number of tasks to avoid creating too many tasks. We maintained a global variable *task_count* to store the tasks we have launched. The ideal number of tasks should be slightly more than cores count, so we set a threshold of *2 * num_cores* to determine whether we should launch a new task or not.

This approach minimizes the negative effect of producing too many threads and works excellently on 2 to 16 threads. However, the cost of synchronizing starts to dominate the execution time when the number of threads increases.

C. Approach 3

The last approach we have implemented is **static** strategy. Our ultimate goal is to utilize high-count cores to accelerate our program as much as possible, and reduce all other synchronization costs. Therefore, we decided to determine the number of tasks statically.

We only parallelize the first depth of the simulation, and it will create at most 480 threads. Since synchronization costs are very expensive when running on 128 cores, we store the results in an array to reduce the stall effect. There is no critical modification between threads. The parent threads will collect and compute the scores sequentially. This approach improves the performance on 64 and 128 cores.

D. Optimization

While we were testing this approach, the performance became worse when the thread count went higher than 64. So we did some optimizations to fix this problem.

First of all, we allocated all threads at the beginning and reused the same threads throughout the whole simulating process, instead of reallocating threads for each move. Secondly, since the cache invalidation is extremely slow for high-count cores, we added paddings to the result array to prevent false sharing. In addition, we use the ”untied” flags for OpenMP tasks, which enables any thread to resume a suspended thread.

Since the workload of each thread varies a lot, and extra costs affect the performance significantly. Those optimization make the execution time faster and more stable.

IV. Results

In this section, we will present the experiment results we conducted on the PSC machine. We measure the time usage of the first 300 moves, running on [2, 4, 8, 16, 32, 64, 128] cores, and use the sequential version as our benchmark. We will only focus on approaches 2 and 3 since the naive approach did not fully utilize the cores.

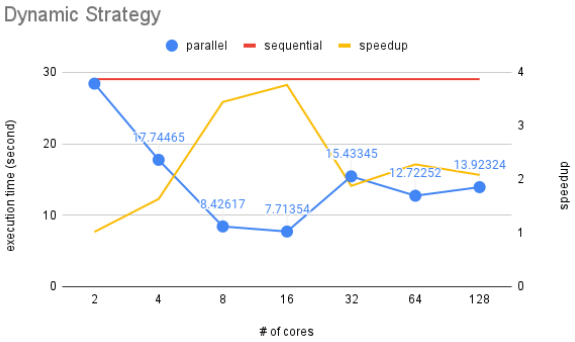

Fig. 4. execution time v.s. speedup v.s. # of threads for dynamic strategy

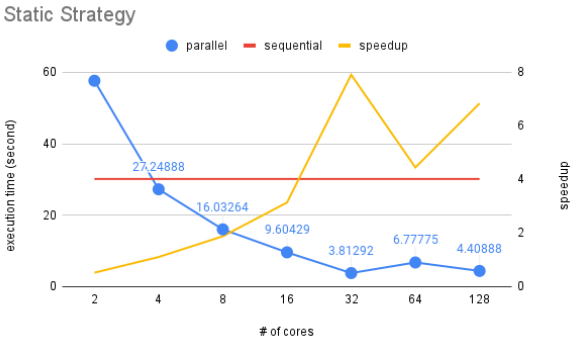

Fig. 5. execution time v.s. speedup v.s. # of threads for static strategy

A. Dynamic Strategy

Our first experiment measures the speedup with different numbers of cores for the parallel approach 2 (dynamic strategy). The result of the experiment is shown in Fig. 4. The approach performs well when running on 16 cores machines, which can reach almost 4x speedup, and scale excellent from 2 cores to 8 cores. This is because we limited the number of threads by *2 * core counts* to utilize the threads better while avoiding extra costs for launching threads. However, the speedup dramatically decreases for 32, 64, and 128 threads. As mentioned in the previous section, the communication costs dominated the running time and decelerated the simulation steps. Currently, OpenMP does not provide any API for controlling the number of tasks, and the synchronization costs are inevitable, if they do support this feature.

B. Static Strategy

This experiment measures the speedup with different numbers of cores for the parallel approach 3 (static strategy). The result of the experiment is shown in Fig. 5. The approach creates a fixed number of threads, regardless of the number of cores. Unsurprisingly, the performance for 2, 4, 8, and 16 cores are slower than approach 2. However, it breaks the ”high-core” limit when running on 32, 64, and 128 cores. It reaches 8x speedup on 32 cores and scales well from 2 cores to 32 cores. It is worth noting that the performance drops for 64 and 128 cores. It is probably because of the

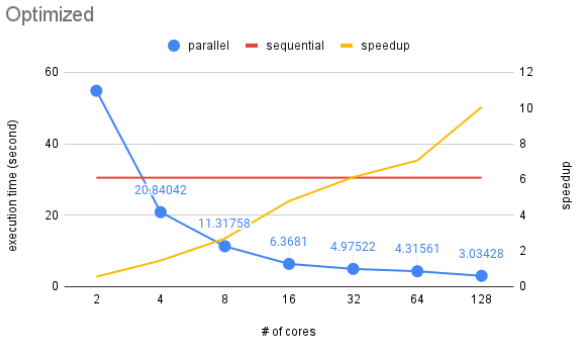

Fig. 6. execution time v.s. speedup v.s. # of threads for optimized static strategy

extra communication costs for allocating threads, invalidating cache lines, and context switches.

C. *Optimized Static Strategy*

This experiment measures the speedup with different numbers of cores for the optimized static strategy, which reusing the threads and adding padding to the array. The result of the experiment is shown in Fig. 6. This version reduces unnecessary initialization and communication costs. It performs best among our approaches and reaches 10x speedup on 128 cores, which means the program actually benefits from the high-count cores.

V. Conclusion

In this project, we implemented different ways to parallelize the 2048 solver, including naive, dynamic, and static strategies. We then analyzed the performance bottlenecks, and further optimized our static approach.

VI. List of Work

This project is implemented by Wei-Ting Tang (weitingt) alone.

References

[1] https://en.wikipedia.org/wiki/2048_(video_game)
[2] https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/2249894022498940
[3] https://github.com/nneonneo/2048-ai