Concurrency and Parallelism - Report 1

Francisco Cunha & Ricardo Esteves

October 13, 2017

Abstract

For this assignment, we were given a working, sequential implementation of the game Othello, in C. Our main task was to parallelize this program using **Cilk+**. After studying the sequential version, we measured the execution time and the overall impact caused by each section of the program. With this information in mind, we looked for what not only could, but also should, be parallelized, to improve the program's run-time. After various tests and analysis, we ended up with a solution that reached an absolute speedup of up to 6.8x over the original implementation (in the 16-core test machine).

1 Introduction

As stated above, we were faced with a sequential version of a C program that played Othello (also known as Reversi). This game's rules won't be explained here, but what's fundamental to know is that each player will play alternately, and, in this case, our players were AIs.

The most interesting part of this program is how, in each turn, a move is calculated. The current implementation of the player AI uses the Minimax algorithm. This algorithm is widely used in decision and game theory, and has the goal of minimizing the possible loss for a worst case scenario [5]. Although being implemented with just one level of depth, evaluating every possible move in every single turn still adds up to a considerable amount of computation per game, representing the bulk of our program's execution time.

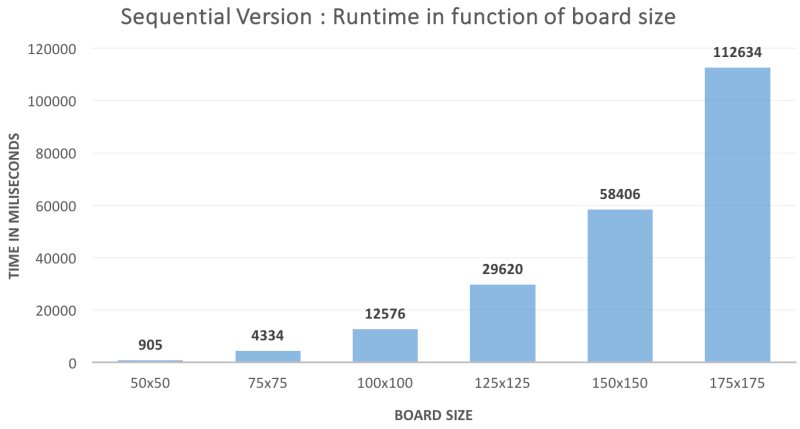

Figure 1: Sequential execution times in function of the board size.

As expected, this grows proportionally to the board's dimensions, effectively defining the latter as our problem size.

In order to make a faster, parallel implementation, we not only analyzed the program's flow but also each function, one by one. Doing this would allow us to detect heavy sequential workloads and parallelizable loops easily. Adding to this information the parallel patterns we've been studying, we ended up with a clearer perception of what made sense (or not) to parallelize.

2 Approach

The first thing we did was to measure the average execution times of every single function in our program. This gave us the first hints on what we should not parallelize - functions that did very light sequential work and executed quickly. We also noticed that the `make_move` function took a considerably longer execution time than the others. However, this was a very rough analysis, so we couldn't conclude much.

To get a better feel for the weight of each function, we resorted to the GNU profiler (*gprof*), focusing on the percentage of time spent on each. After a few measurements, these were the average results we obtained:

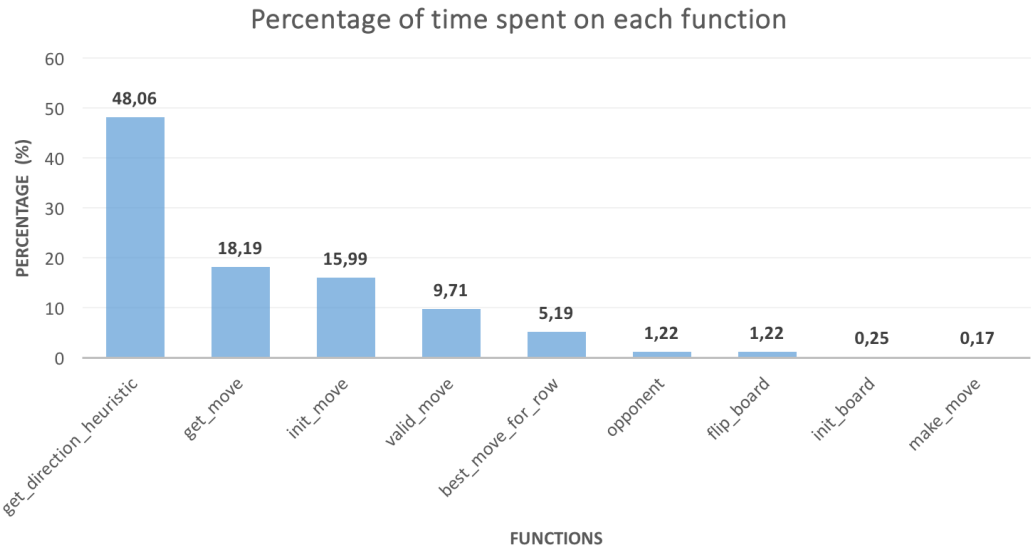

Figure 2: Percentage of time spent on each function.

As expected, the functions where the program would spend most of its time at were **exclusively** involved in the high-level process of making moves. From our very first measurements referenced earlier, we knew that, individually, these functions were actually pretty lightweight, and not a good fit for parallelization. They were, however, being called sequentially many times (directly or indirectly) from the `make_move` function.

If instead of visiting every cell sequentially, one by one, we could split our board into non-overlapping chunks (applying the **Partition** pattern) and have them be treated in the same way by different workers in parallel (using the **Map** pattern), then we'd effectively reduce the execution time of the function - thus greatly decreasing the run time of the program itself.

We found that the cleanest way to split our data without messing with Cilk+'s internal grain-size optimization's was to make different threads responsible for different rows. However, attempting to parallelize the outer for loop at this point would likely result in **race conditions**, because we'd be reading/writing onto shared variables in the body of the loop.

To make the loop's iterations independent, we avoided locks and resorted to **local state**. We replaced `make_move's best_move` with an array, and created a **pure function** that would calculate the best move for a single given row, by traversing through every cell on that row. Our outer loop could now be parallelized: On each iteration, we'd simply calculate the best move for the relevant row and assign the result to the matching index of our array.

Once all rows were analyzed (in parallel), we would pick the overall best possible move by traversing through our array of best moves per row, in a serial reduction fashion.

At the time, we wanted to see if we could also parallelize the column traversing process on our pure function. However, it suffered from the same problem as the original `make_move` function - reading/writing onto shared variables -, and so the race conditions would result in segmentation faults.

In our analysis, we figured that no other part of the program would benefit enough from parallelization. Most had very little impact on the program's execution time and would suffer from the spawn/communication overhead introduced. In addition, we believed that compromising readability by speeding up tiny parts of the program was not a worthy trade-off, and so we decided to keep it simple.

3 Validation

The primary way for us to validate our solution was to effectively run our program under varying conditions (different numbers of threads, different board sizes, different machines...) and see if the results observed would match our expectations. For evaluating our implementation, we established the following benchmarks:

1. When running with a single thread, our solution should not be slower than the sequential version.
2. The speedup should keep increasing until the number of threads used reached the maximum number of physical ones (for each machine).
3. Our program should be totally free from race conditions or other concurrency problems.
4. The program should follow all of the game's rules.

Once these conditions were observed, we were convinced on the correctness of our implementation. This effectively happened, as explained more in detail in the upcoming section.

4 Evaluation

In order to hit the benchmarks we defined earlier, we started by measuring the run-time of our parallel implementation when using a single thread. It matched the original sequential version's run-time very closely, witch confirmed our expectations.

Now, one of the most crucial parts of testing was to see if our program's speedup kept increasing until the number of threads used reached the maximum number of physical ones for our testing machines. To do this, we ran similar batches of tests in different machines. For the sake of the report, we'll stick to the results we obtained on the provided 16-core machine.

We picked relatively big board sizes (100x100 and 150x150), and, for each, we ran our program with 1, 2, 4, 8, 16, 32 and 64 threads. Repeating these tests five times for each number of threads, we averaged the execution times obtained (in milliseconds), which can be seen here:

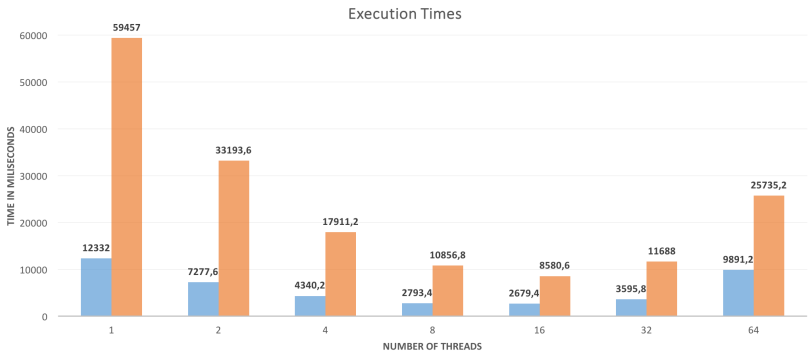

Figure 3: Execution times in milliseconds (100x100 Blue , 150x150 Orange).

Based on this, we also computed the relative speedups achieved:

Figure 4: Relative Speedups (100x100 Blue , 150x150 Orange).

This matched our expectations: Our test machine had 16 cores, and so the relative speedups kept consistently increasing as we used 1 to 16 workers. Following the same logic, when we started to rely on more threads than we had physical cores, the context switching started to cause an ever-increasing overhead, which translated into higher execution times and lower speedups, as we can see from the results we got when using 32 and 64 threads.

As stated in the validation section, we did a lot of testing under different conditions, and not once we were faced with abrupt terminations from concurrency problems. Even though we didn't manipulate the heuristic code directly (and didn't implement the multi-level Minimax), we payed attention to the scores on each game played, and no early endings or inconsistent scores were detected, leading us to consider our implementation as respectful of the game rules.

5 Conclusions

Due to the nature of the program, we can't say exactly what our maximum possible (theoretical) speedup would be. We approached the problem with realistic expectations, and so we think that reaching absolute and relative speedups of **6.8x** and **6.93x** (respectively) on a 16-core machine *whilst* keeping our code simple and clean was a pretty big achievement.

By developing some sensibility for this type of problems during the project, we believe that we balanced optimization and complexity fairly well; even though we could have improved execution times ever so slightly here and there, we followed the KISS principle and concluded it wouldn't be worth it to sacrifice readability for complicated, over-engineered solutions, ending up with an elegant but efficient implementation.

6 Bibliography

[1] Why is Cilk+ not speeding up my program?
https://software.intel.com/en-us/articles/why-is-cilk-plus-not-speeding-up-my-program-part-1

[2] McCool M.,Robison A. ,Reinders J.,(2012), *"Patterns for Efficient Computation"*, Morgan Kaufmann,ISBN: 978-0-12-415993-8

[3] The Official Cilk+ Tutorials,
`https://www.cilkplus.org/cilk-plus-tutorial`

[4] The Official Cilk+ Tutorials,
`https://software.intel.com/en-us/forums/intel-cilk-plus`

[5] Minimax (Wikipedia),
`https://en.wikipedia.org/wiki/Minimax`