8 Queens Problem

Ludvig Larsson

February 15, 2022

1 Task

I chose the 8 queens problem for homework 1. To solve the task, all possible combinations of 8 queens placed on an 8x8 board must be generated and then evaluate each one of them to yield the solutions to the problem. A combination of 8 queens is accepted if no two queens share the same row, column or diagonal. Furthermore, there should be multiple workers (threads) running, to speed up the execution time hopefully. The program can handle between 2x2 boards until 8x8 included, based on command line arguments given.

2 Solution

I decided to implement a recursive algorithm to generate all possible combinations according to the binomial coefficient, without ordering without replacement. The recursive procedure uses an array to push to and pop values from (like a stack). Once the array is full after a few recursion steps one combination is ready for evaluation.

The validations for rows, columns and diagonals are simple. Rows and columns are calculated and by marking indices in an array for respective queen both checks can be done in O(n) time. I did not find a linear solution for validating diagonals, but since I'm using `if (rows(...) && columns(...) && diagonals(...))` diagonal check can often be avoided if either rows or columns did not get a green light beforehand.

The programming model used for the multi-threading aspect of this program is producer consumer. There is only one producer, but possibility to have 1 to N consumers. This is due to the fact that productions are really fast and is afforded to be done sequentially to reduce complexity of the program. Validation of combinations has been the bottleneck (learnt from executing sequentially before multi-threading) so I decided to parallelize consumers.

There is a global buffer struct used to share data between threads, i.e. between producer and consumer(s). The struct holds one combination, if a combination is available or not and termination variables. All should use the same lock since the variables are used together.

2.1 Producer Consumer Pattern

Producers and consumers communicate by having a shared variable `buf_empty` that informs the threads if they should run or not at a given time. The producer will create a combination, take the lock for the shared buffer and copy the combination from private memory to shared memory, set that `buf_empty == 0`, release the lock and then signal a producer to wake up. The producer who wakes up makes sure that `buf_empty == 0` before evaluating the combination. The producer will, with the lock, copy the combination from global memory to local memory and then set `buf_empty == 1` and signal the producer. Once the combination is copied, the consumer can safely continue with evaluation without being interfered by other threads. This allows for multiple consumers to be active simultaneously with independent combinations and yield desired parallelism.

3 Benchmarks (mostly problems)

The code works perfectly with one producer and one consumer. When 2+ consumers are introduced, for some reason that I have not figured out for over one week, it appears that consumers will read the same combinations occasionally and therefore create too many consumptions overall (and also duplicate solutions!). I had a look with Klas (TA) and he wasn't sure either about the synchronization issues. I have used multiple thread debugging tools and data races are highlighted, despite the fact that locks "definitely" protect the critical sections.

Possible issues would include too fast execution of the code to make `memcpy(...)` unreliable, `printf(" ")` after memcpy, in both producer and consumer, reduces the excessive amount of consumptions by a lot namely. Also I speculate if the recursion in the producer can cause problems.

On my MacBook Pro (quad-core), one producer one consumer, the 92 correct solutions are found after 11 minutes. It does feel strange, and this might be where multiple consumers would come in handy. Furthermore, the fact that too many consumptions are done due to multiple consumers, other execution times are not valid since the additional consumptions most likely increase the execution time a lot.