Project 2 Report

CSC 345-01

Jack Taylor & Luke Kurlandski

Our program effectively implements all of the listed requirements listed for Project 2. First, we take a sudoku puzzle from a file and store it into a two-dimensional array with fscanf. Then, we apply three different methods to check for correctness: check_rows, check_cols, and check_boxes. The purpose of these three methods is to parse through every row, column, and box to check for each required digit. Throughout the process, if multithreading is used, these methods check if another thread has already proven the puzzle to be invalid and will terminate themselves if that is the case.

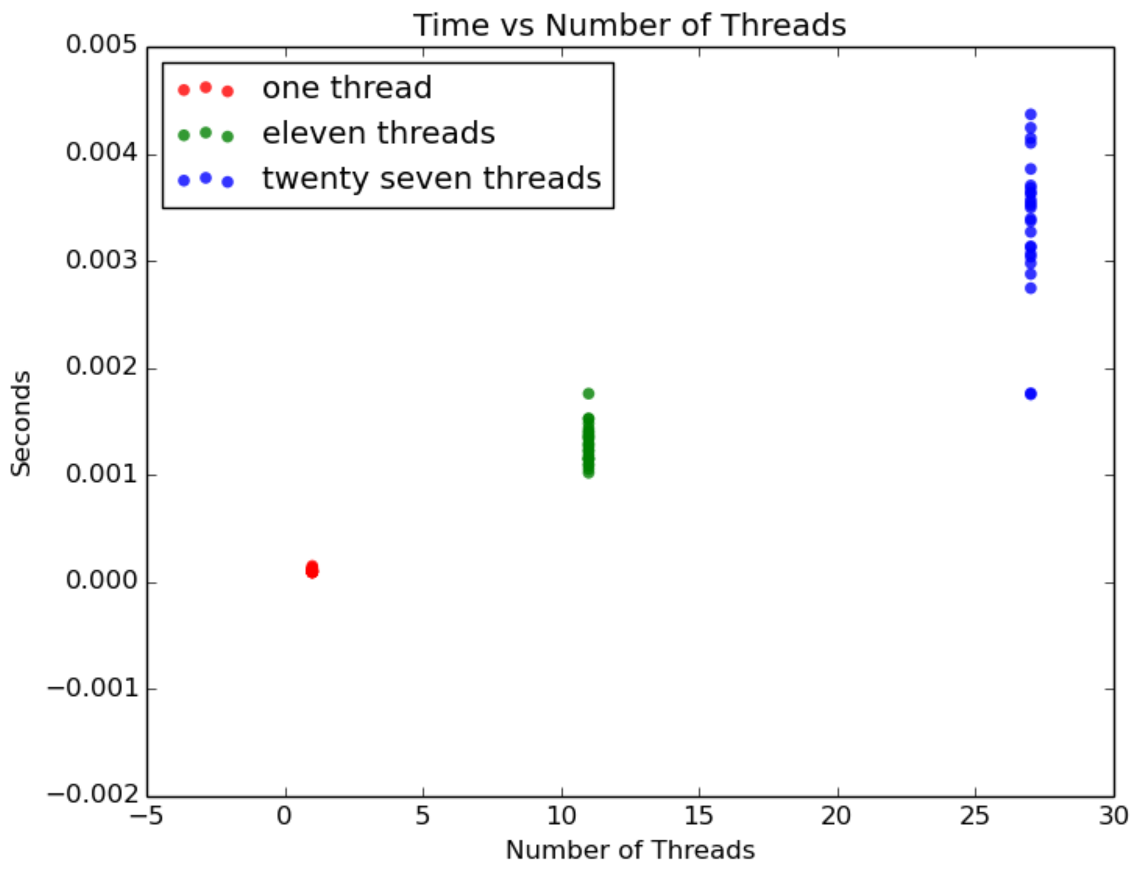
We implement three options for threading. If the user enters a 1, the program runs in a single thread. If the user enters a 2, the program runs in 27 different threads (one for each single row, column, and box). If the user enters a 3, the program is run in 11 different threads (1 to do all rows, 1 to do all columns, and 9 for each single box). After running these methods using the specified number of threads, we stop the clock and use the print_board method in order to print out our 2-dimensional array as well as a YES or NO depending on the final value of the "solution" variable.

We conduct an experiment to determine if a significant difference exists between the variations of thread models. Our null hypothesis is that there is no significant difference between the models. Our alternative hypothesis is that the multithreaded models will outperform the single threaded model, with more threads resulting in greater speed.

After testing this with various different boards, we find that the single-threaded procedure actually runs the fastest, followed by the 11-threaded, and finally by the 27-threaded version.

This is likely due to the laborious task of creating and joining the threads. Any multithreaded application can suffer due to context switches. Since there are only a finite number of CPU cores, the threads cannot all run in parallel. A thread is given a certain amount of time on the core, then a context switch occurs. The thread's state must be saved, the thread goes to the back of a ready queue, and a waiting thread is allocated a CPU. This process does take time, and, in this project, we learned that the cost of threading is not always worth the potential gains.

Therefore, we conclude that the single threaded approach is the fastest in this circumstance. Below is a graphic of the data taken from 75 executions of this program. Each mode was executed 25 times. Clearly, the single threaded version outperforms its multi-threaded cousins.


Time vs Number of Threads
one thread
eleven threads
twenty seven threads
Seconds
Number of Threads