# ECE 459: Programming for Performance Assignment 1

Ryan Qin

January 27, 2021

## 1 Sudoku Solver

The sudoku solver uses backtracking to solve sudoku recursively in a depth-first manner. In each step, it chooses a number from 1 to 9 for a cell and checks whether it's valid. If it is valid, it goes on to the next cell and does so again and again. When it sees a problem, it returns false, which means the current route has come to a dead end, and needs to try something else. Then backtrack happens, it goes to the previous "potentially correct" solution but tries the next cell with other numbers. The correct solution is found when it finds a solution where all the cells are tested to be valid. This happens if the cell is the last one we are checking (bottom-right), at which time true is directly returned without further checking.

In terms of run time, we are traversing through the possible solution tree, with each node having at most 9 children. The minimum number of iterations required is the depth of the tree, which is equal to $N = 81$, the number of cells in the sudoku. Since the size of the input is constant, the final run time of the algorithm is also constant.

I don't believe this algorithm can be further improved significantly because the best case of the run time is going through the sudoku once. However, some minor improvement can be made and potentially increase the performance. For example, instead of trying the number from 1 to 9 in series, we can try a random number within this range excluding the ones that have been tried.

The read_puzzle() function is simpler. It uses a row and column counter to keep track of which position we are reading, and they are incremented accordingly when we read a byte. This counter is then used to assign the Box with the specific index. One note is that when inserting to memory, we have to subtract the value of the byte by 48 and construct it as a NonZeroU8 instead of integer.

The check_puzzle() function is standard as well. It first gather all 9 cells in a row or column and put them in a vector. Then it calls the sort() and dedup() function to remove duplicates before comparing the vector with the correct vector ranging from 1 to 9. Then it does the same but for all the sub-cells. If it doesn't return false in the checks before, true will be returned.

## 2 Nonblocking I/O

For more efficient benchmarking purpose, if number of connections is 0, then we treat it as the blocking-IO case. If number of connections is not 0, then it uses the Multi package for multiple concurrent connections to the host. It calls the set_max_host_connections() function to set the maximum connections it can have when connecting to a single host. Since our application has only one host, calling set_max_total_connections() would also work. It then loops through all the puzzles and creates an easy handler for each one, which is then appended to a list. After all the easy handlers are done, it would get looped again and removed from the list of handlers, at which time the result of the handle becomes available. If the result is 1, that means it's verified and the counter is incremented.

For this part we are only investigating the 10.txt and 100.txt case. For the former, we want to make sure $N = 16$ and $N = 32$ gives the same result because the they are both larger than the total number of required connections. For the latter, we want to see increasing the number of required connections does give better results.

| | Mean time (ms) | Standard deviation (ms) |
|---|---|---|
| Blocking I/O | 1947 | 196 |
| N = 3 | 625 | 100 |
| N = 4 | 505.4 | 51.7 |
| N = 16 | 285 | 26 |
| N = 32 | 287.8 | 17.3 |

Table 1: Result of 10.txt

As shown in Table 1, blocking-IO takes the longest time to finish. When 3 max connections are allowed, the time required reduces significantly by 3 times. This makes sense, since 3 connections should in theory reduce the total time by 3 times. When this number increases to 4 and 16, the time also reduces. However, when it is set to 32, the time is roughly the same as when it is 16. This makes sense, as there is only 10 sudokus, which would only use 10 connections anyways, and the remaining connection are not even used before all puzzles are put in the connection pool. We will make another run with 100.txt which further investigates if time keeps reducing as we increase the number of connections to 32.

| | Mean time (ms) | Standard deviation (ms) |
|---|---|---|
| Blocking I/O | 17943 | 738 |
| N = 3 | 5300 | 244 |
| N = 4 | 4127 | 192 |
| N = 16 | 1113 | 36 |
| N = 32 | 659.8 | 25.4 |

Table 2: Result of 100.txt

Table 2 once again confirms our results from Table 1. If N is the number of connections, it roughly reduces the time required by N times, comparing to the blocking-IO case. As N increases, the time required decreases linearly. It is also important to note that the standard deviation also decreases as N increases. This is because if the program takes less total time to finish, then the difference between the max and min run should also decrease accordingly.