ECE 459: Programming for Performance
Lab 1 -- Solving Sudokus

Created by Jeff Zarnett, updated by Bernie Roehl
Due: January 27, 2021 at 11:59 PM Eastern Time

In this lab, you'll write a program that solves Sudoku puzzles and verifies them using an external webservice.

In the first part of the lab, you'll write a Sudoku solver. You can use a verifier (which we provide) to check if your solutions are correct. In the second part of the lab, you'll speed up the verifier by using non-blocking i/o.

Learning Objectives:

- Learn how to use non-blocking i/o to increase performance

Sudoku

Sudoku is a puzzle game. Here's the brief description from Wikipedia: "*it is a number puzzle based on a 9 × 9 grid. The goal is to fill the grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids contain all the digits from 1 through 9. Zero is not a valid value and there can be no repeats within a row, column, or subgrid. The initial condition is always a partially-filled-in matrix that needs to be completed. A well-designed matrix has a single valid solution*".

If you would like to generate more test cases and their solutions, see https://qqwing.com/generate.html

Part 1: Solving Sudokus

In the starter code that we provide, you'll find a README file that describes how to compile and run your program. We suggest you begin by running the starter code.

The "inputfiles" folder contains a number of sample inputs. Each file contains one or more Sudoku puzzles, separated by two blank lines. Each puzzle consists of 9 lines, each 9 characters long, with each character being a digit from 1 to 9 inclusive or a dot (".") to represent an empty cell. For each file in the inputfiles folder, there's a corresponding file in the "solved" folder that contains the solution in the same format (but obviously with numbers replacing the dots).

There are two command-line parameters to your program. The first will be either "`solve`" or "`verify`" and the second will be a filename (e.g. "`100.txt`"). If "`solve`" is specified (as it is for Part 1 of this lab), the program will read a puzzle from the named file in the inputfiles directory (e.g. "`inputfiles/100.txt`"), and read a matching solution from the

corresponding file in the “solved” folder. It will call a solve() function to solve the puzzle, and then compare the result with the completed puzzle that was read from the solved file.

Your job is to implement the solve() function in `lib.rs`. There are many approaches, and you’re free to do it however you like. Regardless of how you implement it, your solve() function should update the puzzle in-place, replacing the dots with the correct numbers.

You will also need to complete the read_puzzle() function. See the comments in `lib.rs` for details.

You can test the correct operation of the program by editing the solution file and changing one of the values. The program should tell you which puzzle did not match the solution. Be sure to change the solution file back again.

Part 2: Nonblocking I/O

In this part, you will specify “`verify`” as the first argument to your program rather than “`solve`”. The second argument is the name of a file in the “solved” directory. Note that there are some files in the “solved” directory that contain a large number of puzzles, and not all of them will have corresponding puzzles in the “inputfiles” directory.

The verifier uses a web service to check whether puzzles are correctly solved.

The verifier reads the named solution file from the “solved” directory, and for each puzzle in that file it creates a JSON object containing the solved puzzle and sends it to the server with an HTTP POST command. The endpoint on the server is “verify”. The endpoint will return HTTP 400 if anything is wrong with the provided data (including if the body is missing). It will return HTTP 200 if it is capable of parsing and understanding the data. If the data is a valid Sudoku solution, then a body of 1 is sent back; otherwise 0 is returned. After doing this for all the provided puzzles, the verifier prints a message saying that *x* of *y* were correct.

The verifier uses a library called “curl”. The curl library uses *callbacks*. If you’re not familiar with callbacks, the idea is pretty simple. You call curl’s perform() function to do some work (i.e. interact with a web service), and it calls functions (which you provide) while carrying out that work. When the work is complete, curl returns from your call to perform(). Basically you’re saying to curl “perform this task, and let me know when you need me to provide some data to you or if you have some data to give to me”.

The provided starter code uses blocking i/o, meaning that it processes one puzzle at a time. It makes a request to the verification server, waits for it to complete, and then makes the next request. The result is that the program spends most of its time waiting for internet traffic, which is often slow.

Your job is to modify the verifier (i.e. the verify.rs file in the provided starter code) so that it uses non-blocking i/o. Your solution should *not* use threading. Instead, it should open multiple concurrent connections to the verification service. This is accomplished with the curl “multi” interface.

For a small number of puzzles, the verifier will run quickly. However, for larger numbers (1000 or 10000) it will take a lot longer. How much longer? The way to find that answer is through benchmarking.

Benchmarking

You can measure how long your program takes to execute by using the `hyperfine` command on Linux. When you build your program, the executable is placed in target/release/lab1, so you here's a typical sequence:

```
cargo build --release

hyperfine -i target/release/lab1 verify 100.txt
```

The -i option to hyperfine tells it to ignore any error status returned from your program. The starter code does not explicitly return an error status from main(), so without the -i option you'll find that hyperfine aborts without giving you useful output.

Report

You should benchmark your program when running in verify mode, and report the results as compared to the baseline (blocking i/o) program for the number of concurrent connections N in {3, 4, 16, 32}. Is the amount of the performance increase as expected? Why or why not?

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are more likely to be recognized as correct. Solutions that do not compile will earn at most 39% of the available marks for that part. Solutions that compile but crash earn at most 49%. Grading is done by building your code, running it, checking the output, and inspecting the code.

Part 1: Solving the puzzles (40 marks) Your code needs to produce correct sudoku output.

Part 2: Nonblocking I/O (40 marks) Your code must properly use curl's "multi" features and have the correct number of concurrent connections.

Part 3: Report (20 marks)

- 8 marks for discussing the results of part 1.
- 8 marks for discussion of part 2.
- 4 marks for clarity.