ECE 459: Programming for Performance
Assignment 1

Jeff Zarnett

January 8, 2020 (Due: January 27, 2020 at 11:59PM Eastern Time)

In this assignment, you'll work with a program that solves sudokus (see description below if you're unfamiliar with it) and also verifies against an external webservice. We provide a starter single-threaded implementation. You will modify this program in several ways and discuss what worked well, what didn't, and why. In part 2, you'll use nonblocking I/O to speed up the verification process.

Setup

The course will use `https://git.uwaterloo.ca/` which is the university- provided GitLab service. It is likely at this point you have already used either GitLab or GitHub (they are similar). You may need to set up your account but this is easy to do. If you're reading this and you haven't set up your account, do so immediately. Otherwise we can't give you the assignment starter code.

If you've previously configured your account at `git.uwaterloo.ca`, you should find already that we've created a repository for you for this assignment (we are nice) and it will contain the starter files. Future assignments will work the same way.

You should do this assignment using the ECE-provided Ubuntu servers (eceubuntu). That's the environment that we tested the assignment using. If you are having trouble with your environment or setup the first question I will probably ask is "are you running it locally?". Due to the diversity of possible setups, the course staff cannot spend time figuring out library incompatibilities and personal laptop issues. Sorry. If you are a non-ECE student (e.g., SYDE) you should have access to the ECE Ubuntu servers due to being registered in an ECE course.

Remember that access to most UW server resources is restricted by the firewall and that you may need to enable your VPN connection to do this assignment, especially if you are working from off-campus.

To submit, simply push your fork of the git repository back to `git.uwaterloo.ca`.

Sudoku

Sudoku is a puzzle challenge. Here's the brief description from Wikipedia[1]: 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. It's easier to explain visually, so read the wikipedia page. If you would like to generate more test cases and their solutions, see `https://qqwing.com/generate.html`

This sort of puzzle is pretty easy for computers to solve. There are clever algorithms and strategies for humans but computers are perfectly happy to brute force it. This is a parallelizable task!

[1]`https://en.wikipedia.org/wiki/Sudoku`

Part 1: Parallelization, Three Ways

There's more than one way to parallelize your program, and in this assignment you'll do it in three different ways. Each of these is a strategy discussed in the course. This assignment gives you an opportunity to practice them and reflect on their use. There are three different starter files and you should modify each of them according to their named strategy. In all cases, you will use `pthreads` to implement the functionality. If you need a refresher on how `pthreads` work, there's a pdf in the course repository that should help!

The program provides a makefile that is used to compile the code (and the LaTeX report). The makefile isn't super robust but it means you can easily compile the program using the command `make`. You can make changes to the makefile if you need (but you might get frustrated because makefiles are a bit difficult to work with). But anyway.

Your program takes a parameter `-t N` that specifies the number of threads. It's in the starter code but does not do anything at the start. The program also takes the parameter `-i inputfile` to specify the input file. Output always appears in `output.txt`. It is always required to provide an input file, otherwise the program won't run. A sample invocation then: `./sudoku -t 4 -i inputfile.txt`

Note that you are allowed to modify the code as needed to accomplish your strategy (e.g., add some concurrency control or change a function's signature and/or implementation). Just be sure you both (1) preserve the behaviour (solving puzzles) and (2) use the strategy as specified below.

Strategy One. One thread per puzzle (`sudoku_threads.c`). In this implementation each thread takes an entire puzzle from start to finish: read it, process it, write it to a file. This is pretty straightforward.

Strategy Two. Workers with specific jobs (`sudoku_workers.c`). In this implementation, different threads have different jobs: some threads read the input file, some process puzzles (that is, execute the `solve()` function), and some write the output.

There's no reason why you must have equal numbers of each kind of thread. You can choose whatever distribution of threads makes the most sense. You should probably experiment to find out what works best. In this version, if the argument specifying the number of threads is less than 3 (i.e., you cannot have at least one thread of each type), it is okay to exit with an error message.

Strategy Three. One puzzle, multiple threads (`sudoku_multi.c`). In this strategy, multiple threads contribute to solving a single puzzle. In particular, the `solve()` function is what should be parallelized. Below is a suggestion of how, but you don't have to follow it if you don't want to.

The idea is to parallelize based on different initial conditions. For example, when you get to the first empty space, the first thread tries all possibilities where that first space is filled with 1, the next thread tries all possibilities where it is filled with 2... etc. Because you will want to support more than max 9 threads, this should be done at the second level: the initial state being given to a thread is based on filling the first two empty spaces. Phrased another way, you are splitting solve into the top level and subsequent levels.

You will want to give each thread a new copy of the puzzle that it can use, without interfering with other threads' attempts. That new copy of the puzzle will be the initial state for that thread to solve. As a well-designed Sudoku puzzle has only one correct solution, only one thread will find the correct answer. You may need to copy it back.

Sometimes, your number of threads is less than 9. In that case, you won't try all possibilities at once. That's fine! When a thread finishes trying the solution where the space is filled with x, it can do the next set of possibilities where the space is filled with y.

General Notes. A quick way to get a look at how long your code has executed is the `time` command, which you put in front of the executable you would like to run. This command tells you the user, system, and real time. The user time is how much CPU time your code was executing; system is how much was spent in system calls; real is the wall-clock time between the start of execution and the end of execution. In this case, you are trying to reduce the real time value (and this may be done by increasing the user time).

For all strategies, your code should be free of race conditions and other concurrency problems. It should also not have memory leaks. Use Valgrind to check things (both the memory check and the helgrind check). Make sure any library calls you use are thread-safe (you can find out by looking at the man pages). Memory that is "still reachable" should be deallocated where possible. Remember that Valgrind may report issues that you cannot fix because they are in a library not under your control. You may also have benign race conditions. If either of those is the case, just make a note of that in your report for the marker to read.

As an **optional** way that you can make your solution even more optimal, you can avoid repeatedly creating & destroying threads. In that case, you have a thread pool: threads are created once and take work out of a queue or other data container. A quick outline of that strategy: imagine putting a data structure (a puzzle or a data structure containing a puzzle) on a queue. Access to the queue can be controlled with a semaphore: a thread wanting to take work waits on the queue semaphore; something putting work on should post on the semaphore. When a thread takes an item, it works on it, and reports results before it is ready to take the next item.

Also in the repository is a verifier tool. This tool can be used to validate output files against an external service that checks the solution. You can also of course compare the single-thread version's output against the output of your modified code, but you might not write all puzzles in the same order in the output file as they were in the input file. That is okay, you do not have to maintain the order. For that reason, there is the checking tool which would allow you to validate all of them efficiently.

Report, Number One. In your report, you should show timing data for different runs of each version of the solver, comparing against the provided starter code for different sizes of input (see the provided files). Test with the number of threads $N \in \{3, 4, 16, 32\}$ Your data will make it clear which strategy or strategies work best under what circumstances; you should explain a bit about **why** a strategy works better or worse than the others. You should also include some discussion of the level of difficulty of implementing each of the strategies and whether it was worth it for the speedup over an easier strategy's speedup.

The report is made automatically by the makefile. LaTeX might seem scary but it's really not. It is very much like programming a document and it's a nice skill to learn. For the most part, you can just type in some text and it compiles to a nice pdf. If you're having trouble with something, google is good at telling you the answer. Also, we mark it for content, not for formatting, so it doesn't have to be the prettiest document ever.

Part 2: Nonblocking I/O

In this part, you will modify the verifier tool so that it uses non-blocking I/O. Its current (simple) implementation uses blocking I/O to verify the output file from the previous part against an external service. Your solution should *not* use pthreads. However, it should have multiple concurrent connections to servers open. This is accomplished with the `curl_multi` interface. In this case, the `-t` commandline option indicates the number of concurrent connections to the server. The `-i` option is still used to specify the input file.

The verifier reads the provided file and for each puzzle, it creates a JSON object and sends it to the server with an HTTP POST command. The endpoint on the report 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 it is a valid Sudoku solution then a body of 1 is sent back; otherwise 0 is returned. The verifier counts the number of successes and tells you that x of y were correct.

The implementation has a 50 ms delay built into it to simulate sending data over the actual internet. It is likely that for the assignment all the servers and clients will be in the same network (or at least geographically nearby) and the delay makes the scenario more "realistic".

You should also use valgrind on part two. See the general notes of part one for guidance about that.

Report. Again, benchmark your work and report results as compared to the baseline (unmodified) 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 crash earn at most 49%. Grading is done by compiling, running it, checking the output, and code inspection.

Part 1: Parallelization (60marks, 3x20) 20 marks for each of the three strategies for a total of 60. Each of the three is considered separately in the compilation/crash clauses of the general grading notes. Your code needs to produce correct sudoku output (although the order of the output can vary) and use the strategy specified.

Part 2: Nonblocking I/O (20 marks) 20 marks for implementation; it must properly use `curl_multi` and have the correct number of concurrent connections.

Part 3: Report (20 marks)
12 marks for discussing the strategies of part 1.
5 marks for discussion of part 2.
3 marks for clarity.