ECE 459 - Assignment 1

Author:
David Chau

Student Number:
20623345

Date: January 27, 2020

1 Parallelization

The task in this section is to parallelize a program responsible for solving a given set of Sudoku puzzles and then outputting them to file. There are three strategies available in order to accomplish this task, which will be discussed in the following sections. The runtime of the original program given various input sizes is shown below in Table 1.

Table 1: Performance measurements for *sudoku.c*

Input size	**Time (s)**
1	0.024
10	0.24
50	0.943
100	0.818
500	8.01
1000	10.264

1.1 Strategy One

Strategy one is based around solving one puzzle per thread. To do this, each thread is responsible for processing a puzzle from start to finish. This includes: reading the puzzle in from file, solving the puzzle, and finally writing the solution out to file.

Based on the results from Table 2, some minor conclusions can be made. First of all, the overhead from this implementation is insignificant. This can be deduced from the fact that the run-times at lower input sizes are equal to, if not better, than the one presented in Table 1. Next, it can also be said that this strategy was able to do its job of improving the overall performance, which is something that is increasingly important at larger input sizes. This is indicated by the lower runtimes relative to the base `sudoku` program at higher input. For example, with 32 threads and an input size of 1000, `sudoku_threads` was able to run in nearly half the time.

In comparison to the other strategies, this one is the easiest in terms of implementation. It uses the same run function throughout all the threads, which reduces the amount of custom logic and thread management required. Additionally, it runs faster than strategy two and three on the last test case (i.e., 32 threads and input size of 1000). Based on this information, it can be said that this strategy was worth the speed up.

Table 2: Performance measurements for *sudoku_threads.c*

	Number of Threads			
Input size	3	4	16	32
1	0.021	0.017	0.169	0.018
10	0.189	0.214	0.161	0.157
50	0.609	0.678	0.491	0.514
100	0.702	0.62	0.39	0.336
500	7.047	7.385	5.811	5.579
1000	8.684	8.237	6.001	5.039

1.2 Strategy Two

Strategy two is based around assigning threads particular jobs. These jobs consist of: reading puzzles in, solving puzzles, and outputting puzzles. As such, multiple threads will be involved in the process of solving a given puzzle. The allocation of jobs to threads was done such that there was only ever one thread reading and one thread writing. The rest of the available threads perform the solving job. This allocation was determined by profiling the base `sudoku` program to see which parts of the code were taking the longest to run. It turned out that more than 99% of the time was spent solving the puzzle, with the leftover time spent reading, writing, and performing other miscellaneous operations. To note, the queue data structure used in strategy two was provided by https://github.com/raideus/c-data-structures and then modified to suit the requirements of strategy two.

Based on the results from Table 3, some conclusions can be drawn. Similarly to strategy one, there is little overhead based on the runtimes at lower input sizes. Additionally, there is a noticeable performance increase when compared to the base `sudoku` program. For example, at 32 threads and an input size of 1000, strategy two was able to run in 5.993 seconds, relative to the 10.264 seconds of the base program.

Table 3: Performance measurements for *sudoku_workers.c*

	Number of Threads			
Input size	3	4	16	32
1	0.036	0.012	0.022	0.019
10	0.196	0.185	0.201	0.218
50	1.152	0.638	0.518	0.448
100	0.899	0.895	0.429	0.369
500	8.4	7.644	5.898	5.209
1000	10.679	9.527	7.451	5.993

In comparison to the other strategies, this strategy was the second easiest to implement. The speedup increase was worth it for the slight increase in complexity relative to strategy one. In addition, by splitting up the work to be done, it will make it easier to maintain in the future (i.e., separation of responsibility). Additionally, relative to strategy one, the runtime for the last test case is slightly worse. This can be attributed to the fact that the read and write jobs are lightweight. Since strategy two allocates two of its total threads to perform this work, it will be missing out on the parallelization of the heavier work (i.e., solving the puzzles). On the other hand, strategy one will be able to allow all of its threads to perform the work of solving. Compared to strategy three, this strategy is faster on the last test case, however slower across the board for the rest of the test cases.

1.3 Strategy Three

Strategy three is based on having multiple threads contribute to solving one puzzle. Essentially, the `solve()` function is parallelized. To do this, the following process was done:

1. Create threads that will do nothing until queue with modified puzzles is populated
2. Given a puzzle, find first two empty spaces
3. Generate modified puzzles with the first two empty spaces filled in with valid values
4. Add modified puzzles to a queue
5. Let main wait on threads to find a solution. They do this by each taking a modified puzzle from the queue and running `solve()`
6. Once a thread finds a solution, write that solution to file and let mains and other threads know that a solution has been found.
7. Repeat from step 2 until no more puzzles are available

To note, the queue data structure used in strategy three was provided by the same source as the one used in strategy two.

Based on the results from Table 4, some conclusions can be made. The first thing being that this implementation seems to have better performance for the wide majority of test cases, relative to strategies one and two and also the base `sudoku` program. When compared to the base program, this is reasonable as the point of this strategy was to increase performance. With respect to the other two strategies, this can be attributed to the fact that there is minimal destroying and recreating of threads in

this implementation. In fact, the same children threads are used throughout the runtime, which differs from the other two strategies where the majority of threads were constantly being created and destroyed, resulting in noticeable overhead.

A notable exception in performance is the last test case (i.e., input size 1000 and 32 threads), which ends up being slower when compared to the other two strategies and the base program (sometimes even taking upwards of 5 minutes). This exception can be tied with the fact that performance seems to degrade when going from 16 threads to 32 threads a majority of the time. A possible justification for this lies within the implementation. Currently, threads are spawned all at once at the beginning of program execution and immediately begin running. There is no mechanism blocking these threads from continually wasting CPU cycles checking if there are puzzles to solve. Similarly, when main is waiting on the children threads to find a solution, it is also continually wasting CPU cycles checking a condition. As more threads are introduced and continually idling, especially at higher input sizes, this degradation in performance becomes more noticeable. Another potential justification could lie with how this methodology is inherently not good (i.e., doing a lot of work for little gain). The main reason why performance decreases are not seen across the board is, as mentioned before, due to the optimization of not destroying threads all the time.

Table 4: Performance measurements for *sudoku_multi.c*

	Number of Threads			
Input size	3	4	16	32
1	0.012	0.017	0.108	0.011
10	0.139	0.083	0.126	0.192
50	0.273	0.363	0.623	0.357
100	0.297	0.199	0.238	0.308
500	4.746	4.561	6.897	9.584
1000	6.052	5.259	5.467	11.523

In comparison to the other three strategies, this strategy was the most complicated to implement. The main reason being that in order to get a decent runtime magnitude and for the runtime to not increase as the number of threads increased, threads needed to be kept alive instead of continually recreated. The logic required to do this, coming from strategy one and two, was noticeably harder to accomplish. It is hard to draw definitive conclusions with regards to performance for this strategy as the other two strategies did not implement the same "keep threads alive" mechanism, which would have greatly improved their speed. In the end though, given the performance metrics on the last test case and even with the optimizations, it is hard to say if this strategy was worth it.

2 Non-blocking I/O

The task in this section is to modify the `verifier` tool so that it uses non-blocking I/O. This involves having multiple concurrent connections to servers open. The runtimes for the base `verifier` can be seen below in Table 5.

Table 5: Performance measurements for *verifier.c*

Input size	**Time (s)**
1	0.068
10	0.0572
50	2.811
100	5.594
500	27.986
1000	55.984

By implementing the solution presented in the assignment manual, the resultant program `verifier_multi` was able to produce the results seen in Table 6.

Table 6: Performance measurements for *verifier_multi.c*

	Number of Connections			
Input size	3	4	16	32
1	0.064	0.063	0.058	0.059
10	0.217	0.163	0.067	0.068
50	0.887	0.680	0.223	0.136
100	1.752	1.3	0.394	0.228
500	8.59	6.414	1.708	0.89
1000	17.102	12.858	3.34	1.759

As expected, by concurrently sending POST requests using the `curl_multi` interface, there is a clear reduction in runtime as the number of threads increases. This difference becomes even more evident at larger input sizes. For example, at an input size of 1000 and 32 connections, the runtime goes from 55.984 seconds to 1.759 seconds. This trend is observed consistently for other combinations of input sizes and connections. Additionally, it should be stated that this makes sense as the bulk of the time in sending a POST request is waiting for a response from the server. Since this portion is essentially happening concurrently for multiple requests, a large increase in performance can be expected.