ECE 459: Programming for Performance
Assignment 1

Jinming Zhang

January 29, 2020

1 Parallelization

I parallelized the Sudoku program three different ways. Here is my data for each of the strategies.

Table of Summary for all the Strategies

The values in this table consist all the average time (averaging from 3 trials) for different strategies with respect to the number of threads used. These are summary tables for the data presented above.

With 10 puzzles:

Number of Threads	Orignal Strategy(s)	Strategy 1(s)	Strategy 2(s)	Strategy 3(s)
3	0.236	0.157	0.157	0.213
4	0.190	0.164	0.167	0.216
16	0.185	0.147	0.155	0.153
32	0.209	0.159	0.156	0.141

With 1000 puzzles:

Number of Threads	Original Strategy(s)	Strategy 1(s)	Strategy 2(s)	Strategy 3(s)
3	10.075	4.099	5.486	12.438
4	10.094	2.807	3.654	10.924
16	10.227	2.478	2.705	7.775
32	10.104	2.469	2.638	7.167

Note: All the data above is taken from the server eceubuntu1. The accuracy of the data might be inconsistent due to the fact that the server load might vary as the number of users change.

Here is my discussion of the performance benefit of each approach and the relative difficulty of each. Since we are analysing using the benchmarks of {3, 4, 16, 32}, 50, 100, 500, 1000 puzzles is bigger than all of the benchmarks, so the data for 50, 100, 500, 1000 puzzles should have similar behaviors as the one with 1000 puzzles. 1 puzzle takes very short amount of time to execute, and all the strategies will have the same execution time. These are the reasons why I only took the data for 10 puzzles and 1000 puzzles.

As for the performance benefit, from the 1000 puzzle table above, it is very clear that all three strategies are benefiting from the multi-threading. However, each of the new strategies has different advantage and disadvantage when it comes to multi-threading.

For the first strategy, it is a very straightforward way to solve the Sudoku with multi-threading. The way I implemented it is to use threads to loop through the process of reading the puzzle from input file, solve the puzzle and, write the puzzle to the output file and each thread can handle multiple puzzles. When we consider the data from the 10 puzzle table, Strategy 1 shows that it has lowered the execution time for all the benchmarks when compare to the baseline but the execution is saturated at around 0.155s. It is due to the fact that there is only 10 puzzles to solve. When there are 3 threads presented, it is likely that each thread is read, solve and write around 3 puzzles concurrently which results the lower time than benchmark. And same thing happens when there are 4 threads and the only difference is that each handles less. But when there are more than 10 threads (16 or 32), there are more threads than puzzles. In this case, some of the 16 or 32 threads are not handling anything. This results the time saturation in execution.

When we look at the 1000 puzzles tables, we can have more information about strategy 1. When look at the 16, 32 threads case, the time seems also saturated. More importantly, 2.469s is close to 1/4 of the baseline value of 10.104s (16 threads have the similar case). This might due to the fact that there are 4 cores in the eceubuntu1's processor (I believe that is the case). Although threads runs in parallel when multi-threading, in reality threads are still running sequentially with a very small time slice and context switching. How many concurrent threads at a giving time is still depends on the number of cores in the processor. Since there are 4 cores in eceubuntu1, the speed up from multi-threading should be close to 4 times. As a result of that, the first strategy have the greatest performance benefit when there are lots of puzzle to solve, large amount of available processor core.

For the second strategy, it have spilt different part of the puzzle solving process to different works. My implementation for strategy 2 is that read workers communicate with solve worker with an input c pipe. When reader finishes reading, it puts the puzzle on the input pipe. Same thing happens between solvers and writes with an output pipe. It have similar performance benefit as the the first strategy. The difference is that since there are 3 different type of worker threads handling different processes, the speed up factor should to equal to number_of_threads/3. Again, there is a 4 core limit in this case, so the maximum speed up fact is min(4, number_of_threads/3). This is shown from the 1000 puzzle table. When there are 16 threads or more, the execution time saturates. As a result of that, the second strategy have the greatest performance benefit when there are lots of puzzle to solve and large amount of available processor core and lots of available threads.

For the third strategy, it splits a puzzle into 81 jobs and all the available threads handles this 81 jobs in batches. If there are 10 threads available, these 10 threads handles 81 job in batches of 10 jobs 8 times. The one reminder is handled by whatever threads available after that 10 batches. The will make solving one puzzle faster especially when that puzzle is extremely big. Look at the 10 puzzles table and the 1000 puzzles table, when there are less amount of threads available, the performance of the strategy is performance worse than the baseline due to the threads creation

overheads and extra calculation for one single puzzles. When there are a small amount of puzzles that need to be solved, this strategy will be a good choice. As a result of that, the third strategy have the greatest performance benefit when there is an extremely complex puzzles to solve and there are only a small amount of presented puzzles.

2 Non-blocking I/O

Then I modified the verifier tool to do non-blocking I/O. Here is my data.
With 10 puzzles:

Number of connections	Unmodified Version(s)	Unmodified Version(s)
3	0.562	0.214
4	0.560	0.251
16	0.560	0.249
32	0.560	0.180

With 1000 puzzles:

Number of connections	Unmodified Version(s)	Unmodified Version(s)
3	55.499	17.509
4	55.500	13.126
16	55.500	3.356
32	55.499	1.749

Here is my analysis of the performance. Since the Non-blocking I/O will not waste time to wait on callback result, the performance will be way better than the baseline version of the verifer. This is shown in both puzzle tables. Without non-blocking CURL requests, the verifer needs to make HTTP requests sequentially which slows down the verification. This non-blocking I/O is like a CURL version of multi-threading. When concurrent connections increases, the overhead of the Non-Blocking version of the verifer reduce which is again shown in both puzzle tables. As a result of that, the Non-blocking strategy have the greatest performance benefit when there are lots of connections presented at a time.