ECE 459: Programming for Performance
Assignment 1

Zhidong Zhang

January 28, 2020

1 Parallelization

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

Number of Puzzels = 1000:

Number of Threads	*sudoku*	*sudoku_threads*	*sudoku_workers*	*sudoku_multi*
$N = 3$	8.835s	**3.087s**	8.999s	6.790s
$N = 4$	8.862s	**2.379s**	9.009s	5.995s
$N = 16$	8.903s	**2.144s**	2.246s	5.823s
$N = 32$	8.852s	**2.049s**	2.086s	5.786s

Number of Puzzels = 100:

Number of Threads	*sudoku*	*sudoku_threads*	*sudoku_workers*	*sudoku_multi*
$N = 3$	0.733s	**0.311s**	0.730s	0.579s
$N = 4$	0.893s	**0.270s**	0.754s	0.495s
$N = 16$	0.737s	**0.230s**	0.271s	0.502s
$N = 32$	0.689s	**0.230s**	0.244s	0.485s

Number of Puzzels = 10:

Number of Threads	*sudoku*	*sudoku_threads*	*sudoku_workers*	*sudoku_multi*
$N = 3$	0.178s	**0.143s**	0.166s	0.153s
$N = 4$	0.181s	**0.131s**	0.169s	0.146s
$N = 16$	0.174s	0.128s	0.144s	**0.112s**
$N = 32$	0.170s	0.130s	0.134s	**0.107s**

Here is my discussion of the performance benefit of each approach and the relative difficulty of each.

Sudoku_threads works best when the input size is large. Sudoku_workers works best when the input size is large and the number of threads is large. Sudoku_multi works best when the input size is small and the number of threads is large. All strategy perform better as number of thread increases. After the number of threads reaches 16, the performance saturates.

More specifically, when the input size is 1000, the strategy one (sudoku_threads) works best for any number of threads. When the input size is 100, the strategy one (sudoku_threads) still works best for any number of threads. However, when the input size is 10, the results are different. When the number of threads is small (3 or 4), the strategy one (sudoku_threads) works best. When the number of threads is large (16 or 32), the strategy three (sudoku_multi) works best.

Sudoku_threads works best when the input size is large. In the Sudoku_threads, N threads are created at the very beginning. For each thread, it constantly reads the next puzzle and solve it and write to the output file until there is no more puzzle. When the number of thread is N, there are only N thread-creations, and all N threads are involved in solving the puzzles. Since the puzzle-solving part is the bottleneck, compared to the baseline, Sudoku_threads has at most N times performance increase. However, when N reaches 16, the performance increase saturates around 4, because there is only 4 CPU core in eceubuntu. More interesting, when the input size is small, the performance increase is capped at the input size, horizontal parallelization is not as useful as it is when the input size is large. Therefore, Sudoku_threads works best when the input size is large.

Sudoku_workers works best when the input size is large, and the number of threads (N) is large. In Sudoku_workers, the read, puzzle-solving, write operations are decoupled, and each operation is performed by a number of threads. In my implementation, N threads are evenly distributed, which means, N/3 threads are reading the input file, and N/3 threads are solving the puzzle, and N/3 threads are writing the output. Reading threads, solving threads, and writing threads are communicating using pipe. Since solving the puzzle is the bottleneck, compared to the baseline, Sudoku_workers have at most N/3 times performance increase. More specifically, when N is 3 and 4, N/3 is around 1, which means there is no performance improvement. When N is large, say N = 16, the performance increase is around min(# of CPU, N/3) = min(4, 16/3) = 4 and Sudoku_workers's performance is on par with Sudoku_threads's performance.

Sudoku_multi works best when the input size is small, and the number of threads is large. To parallelize the solving of a single puzzle, the puzzle-solving problem need to be split into 81 sub-problems. Each subproblem is a new puzzle which has the first and the second empty space filled out. To solve 81 sub-problems with k threads, the number of active thread need to be monitored, and new thread can only be created when the number of active threads is smaller than the maximum number of threads. For each puzzle, there are at most 81 threads need to be created. Therefore, when the input size is large (1000), there are at most 81000 thread-creation time, which is a huge overhead and make the Sudoku_multi perform worse than Sudoku_threads and Sudoku_workers. However, when the input size is small, the performance increase of the Sudoku_multi is not capped at the input size. Therefore, sudoku_mult has the best performance when the input size is small and the number of thread is large.

In term of the level of implementation difficulty, sudoku_threads is the easiest to implement, because creating the N threads and joining them are easy. sudoku_workers is the second easiest to implement because it includes creating N threads and using pipe to handle inter-thread communication. sudoku_multi is the hardest to implement because dynamically maintaining N number of threads and splitting the puzzle-solving problem into subproblem are challenging and complex.

In conclusion, sudoku_thread not only has the best overall performance but also is the easiest strategy to implement. Therefore, strategy one (sudoku_thread) is the most recommended strategy.

2 Nonblocking I/O

Then I modified the verifier tool to do nonblocking I/O. Here is my data.

Number of Puzzels = 1000:

$Number\,of\,Concurrent\,Connections$	$verifier\,(baseline)$	$verifier\_multi$
$N = 3$	$55.552s$	**17.403s**
$N = 4$	$55.392s$	**13.057s**
$N = 16$	$55.493s$	**3.330s**
$N = 32$	$55.328s$	**1.745s**

Number of Puzzels = 100:

$Number\,of\,Concurrent\,Connections$	$verifier\,(baseline)$	$verifier\_multi$
$N = 3$	$5.581s$	**1.744s**
$N = 4$	$5.609s$	**1.286s**
$N = 16$	$5.606s$	**0.371s**
$N = 32$	$5.612s$	**0.216s**

Number of Puzzels = 10:

$Number\,of\,Concurrent\,Connections$	$verifier\,(baseline)$	$verifier\_multi$
$N = 3$	$0.558s$	**0.210s**
$N = 4$	$0.558s$	**0.160s**
$N = 16$	$0.558s$	**0.058s**
$N = 32$	$0.558s$	**0.058s**

Here is my analysis of the performance:

It is obvious that verifier_multi performs better than the verifier (baseline) in every scenario. This is as expected. verifier_multi is using asynchronous I/O and is able to make multiple HTTP calls simultaneously. However, verifier (baseline) using blocking I/O and have to make multiple HTTP calls sequentially. Compared to verifier (baseline), verifier_multi reduced the number of round trips by making it parallel and achieves lower latency.

As the number of concurrent connections increases, the latency of the verifier_multi reduces. Especially when the input size is large (1000 or 100), using verifier_multi, the performance increase is linear with respect to the number of concurrent connections. This result is as expected. When the number of concurrent connections increases from k to 2k, the total latency is reduced from (input_size/k) * RTT to (input_size/2k) * RTT, which means the performance is increased by a factor of 2.

It is interesting to see that the performance of verifier_multi does not change after the number of concurrent connections reaches 16 when the input size is 10. This is as expected. The maximum number of active connections is equal to max(input_size, num_connections), which means the performance increase is capped at the input size when the input size is larger than the number of concurrent connections.