ECE 459: Programming for Performance
Assignment 1

Yang Yongren

January 27, 2020

1 Parallelization

The table shows the original program without multi-threading

1.txt		10.txt		50.txt		100.txt		500.txt		1000.txt	
real	0.019s	real	0.220s	real	0.950s	real	0.975s	real	8.190s	real	10.321s
user	0.004s	user	0.206s	user	0.941s	user	0.792s	user	8.130s	user	10.295s
sys	0.003s	sys	0.004s	sys	0.000s	sys	0.004s	sys	0.008s	sys	0.005s

1.1 Strategy 1

The table shows performance of strategy 1, one thread per puzzle. The performance benefit of this approach is that it can compeletly parallelize every puzzle, each thread is responsible for its own puzzle's read, write and solve, so puzzles are being processed at the same time. Theoretically speaking, if the number of puzzle is smaller than the number of thread, it only needs time of one puzzle to finish the job, if the number of puzzle is greater than the number of thread, time is ceiling(number of puzzle/number of threads). As we can see from the table, compare to the original data, as the number of puzzles increases, the per formance inprovement is more obvious, and as the number of threads increases, the performance also improves significantly.

To implement this, I open a new file pointer in each thread, and then seek to the location of the puzzle it responsible for, which achive the parallel read. Then write the unsolved puzzle into a temp buffer and use sove function to solve it and write to a temp buffer. The last step is to write the solved buffer into the output file.

One of the difficulty is that in order to write correct output to a file, a lock must be used to lock the output file pointer, which serialize the writing process. The performance is not improved when the number of puzzle is very small, this is because multi-thread needs time to cread and join thread and manipulate the file pointer, which consumes time. So the performance is obviously improved only the number of puzzle is large, and the percentage of the initialize time is small. The program has no memory leak or race condition.

Thread	3		4		16		32	
1.txt	real	0.021s	real	0.016s	real	0.017s	real	0.018s
	user	0.011s	user	0.003s	user	0.009s	user	0.005s
	sys	0.011s	sys	0.013s	sys	0.000s	sys	0.004s
10.txt	real	0.183s	real	0.259s	real	0.249s	real	0.149s
	user	0.219s	user	0.464s	user	0.503s	user	0.298s
	sys	0.011s	sys	0.008s	sys	0.000s	sys	0.000s
50.txt	real	0.439s	real	0.555s	real	0.543s	real	0.486s
	user	1.208s	user	1.781s	user	1.413s	user	0.937s
	sys	0.000s	sys	0.035s	sys	0.009s	sys	0.008s
100.txt	real	0.464s	real	0.330s	real	0.331s	real	0.306s
	user	1.277s	user	1.317s	user	0.955s	user	0.845s
	sys	0.016s	sys	0.004s	sys	0.004s	sys	0.016s
500.txt	real	3.424s	real	2.297s	real	2.652s	real	2.492s
	user	13.233s	user	13.444s	user	8.655s	user	8.713s
	sys	0.020s	sys	0.040s	sys	0.008s	sys	0.008s
1000.txt	real	4.870s	real	3.195s	real	2.561s	real	2.494s
	user	17.636s	user	17.934s	user	11.904s	user	11.893s
	sys	0.012s	sys	0.020s	sys	0.033s	sys	0.020s

1.2 Strategy 2

The table shows performance of strategy 2, workers threads. The performance benefit of this approach is that if the number of each worker is properly divided, and if there are enough number of workers, it only needs time of one puzzle to finish the job, for example, solving consumes more time than reading the puzzle from file or writing the puzzle to file, so I create more solving workers, after the reading worker finish 1 puzzle, solving workers can start, and writing workers can start after 1 puzzle is solved.

To implement this, I use 2 stacks to store the unsolved puzzle and solved puzzle respectively, each reading worker open a new file pointer, and then seek to the location of the puzzle it responsible for, and then push the unsolved puzzle into the stack. The solving worker will keep popping the unsolved puzzle from the stack, then solve it and push it into the stack that stores solved puzzle. The writing worker will keep popping the solved puzzle from the stack, then write it to a file.

One of the difficulty is still a lock must be used to lock the output file pointer, which serialize the writing process. The performance is better than the default program but worse than the strategy 1, when different workers are working together, they must share one same stack pointer to keep track of the first element, the lock and push and pop action will cost some time. Also, when there are only 3 threads, it will act like default program without multi-threading or even worse because of the extra action. If the thread is not properly divided, some workers will wait for other workers, that also makes it slower than strategy 1. The program has no memory leak or race condition.

Thread	3		4		16		32	
1.txt	real	0.016s	real	0.010s	real	0.017s	real	0.022s
	user	0.016s	user	0.008s	user	0.003s	user	0.003s
	sys	0.011s	sys	0.009s	sys	0.000s	sys	0.006s
10.txt	real	0.190s	real	0.216s	real	0.160s	real	0.162s
	user	0.293s	user	0.280s	user	0.198s	user	0.186s
	sys	0.017s	sys	0.013s	sys	0.008s	sys	0.000s
50.txt	real	0.934s	real	0.934s	real	0.551s	real	0.504s
	user	1.401s	user	1.233s	user	0.940s	user	0.942s
	sys	0.059s	sys	0.029s	sys	0.016s	sys	0.008s
100.txt	real	0.868s	real	0.832s	real	0.337s	real	0.335s
	user	1.396s	user	1.414s	user	0.838s	user	0.871s
	sys	0.033s	sys	0.004s	sys	0.016s	sys	0.025s
500.txt	real	8.556s	real	8.499s	real	2.710s	real	2.597s
	user	14.193s	user	12.897s	user	8.813s	user	8.772s
	sys	0.501s	sys	0.236s	sys	0.178s	sys	0.179s
1000.txt	real	10.984s	real	10.967s	real	3.370s	real	2.896s
	user	19.009s	user	17.312s	user	11.536s	user	11.330s
	sys	1.117s	sys	0.552s	sys	0.432s	sys	0.349s

1.3 Strategy 3

The table shows performance of strategy 3, multiple initial solution threads. The performance benefit of this approach is that it can improve the speed of puzzle solving with large amount of threads. To implement this, I assigned each thread a different initial value and try to solve with this value, if the puzzle cannot be solved, it will act like default program, loop all the possible value in each empty space. To make sure the program runs properly, I make a new copy of puzzle every time a new thread is created. The performance is better than default program most of time, but is worst in all three strategies. This is because that read and write process are completely serialized, and even many threads are trying to solve puzzle at the same time, it is still one puzzle being solved at a time, each puzzle can be solved faster if there are enough threads, however, the whole improvment is still not very obvious. Sometimes, the performance is even worse than the single thread program, this is because when the number of thread is less than 9, there is possibility that all the initial value are wrong, then the program will simply run like single thread program, plus extra thread creating and puzzle coping time. Also, if the number of puzzle is too small, the percentage of thread creating and puzzle coping time will be large, can cause the performance worse than the default program.

The program has no memory leak or race condition.

Thread	3		4		16		32	
1.txt	real	0.016s	real	0.011s	real	0.015s	real	0.015s
	user	0.013s	user	0.009s	user	0.000s	user	0.005s
	sys	0.006s	sys	0.008s	sys	0.003s	sys	0.004s
10.txt	real	0.169s	real	0.173s	real	0.177s	real	0.189s
	user	0.848s	user	0.787s	user	0.338s	user	0.335s
	sys	0.004s	sys	0.016s	sys	0.008s	sys	0.000s
50.txt	real	0.983s	real	0.991s	real	0.484s	real	0.533s
	user	2.659s	user	1.930s	user	1.188s	user	1.105s
	sys	0.047s	sys	0.028s	sys	0.008s	sys	0.008s
100.txt	real	0.976s	real	0.887s	real	0.538s	real	0.577s
	user	2.298s	user	1.882s	user	1.180s	user	1.041s
	sys	0.101s	sys	0.048s	sys	0.008s	sys	0.008s
500.txt	real	8.760s	real	8.565s	real	3.324s	real	3.500s
	user	18.178s	user	15.073s	user	13.174s	user	11.524s
	sys	0.487s	sys	0.260s	sys	0.048s	sys	0.024s
1000.txt	real	11.738s	real	10.804s	real	4.661s	real	5.218s
	user	24.963s	user	19.645s	user	17.066s	user	15.895s
	sys	0.898s	sys	0.580s	sys	0.140s	sys	0.052s

2 I/O

The table shows the original program without multi-threading

1.txt		10.txt		50.txt		100.txt		500.txt		1000.txt	
real	0.085s	real	0.572s	real	2.807s	real	5.606s	real	27.980s	real	55.785s
user	0.015s	user	0.000s	user	0.026s	user	0.058s	user	0.238s	user	0.389s
sys	0.011s	sys	0.016s	sys	0.030s	sys	0.047s	sys	0.280s	sys	0.384s

2.1 multi-threading IO

Then I modified the verifier tool to do nonblocking I/O. The table shows the performance of multi-curl. The performance is greatly improved as the number of threads and the number of input increases. This is because I assign each eh handler a puzzle and use the multiPerform api. The program will keep checking if any eh is finished and replace a new one with puzzle.

Thread	3		4		16		32	
1.txt	real	0.075s	real	0.068s	real	0.083s	real	0.083s
	user	0.011s	user	0.012s	user	0.007s	user	0.007s
	sys	0.016s	sys	0.015s	sys	0.002s	sys	0.012s
10.txt	real	0.168s	real	0.118s	real	0.064s	real	0.065s
	user	0.006s	user	0.007s	user	0.006s	user	0.000s
	sys	0.005s	sys	0.003s	sys	0.005s	sys	0.010s
50.txt	real	1.076s	real	0.629s	real	0.220s	real	0.150s
	user	0.033s	user	0.000s	user	0.019s	user	0.019s
	sys	0.027s	sys	0.017s	sys	0.000s	sys	0.005s
100.txt	real	1.959s	real	1.304s	real	0.376s	real	0.219s
	user	0.017s	user	0.020s	user	0.019s	user	0.042s
	sys	0.004s	sys	0.010s	sys	0.011s	sys	0.011s
500.txt	real	8.554s	real	6.439s	real	1.676s	real	0.856s
	user	0.148s	user	0.109s	user	0.085s	user	0.111s
	sys	0.074s	sys	0.069s	sys	0.034s	sys	0.048s
1000.txt	real	17.248s	real	12.807s	real	3.263s	real	1.683s
	user	0.226s	user	0.247s	user	0.161s	user	0.209s
	sys	0.173s	sys	0.147s	sys	0.075s	sys	0.089s