ECE 459: Programming for Performance
Assignment 2

Yongren Yang

February 12, 2020

I verify I ran all benchmarks on a ecetesla0 with at least 4 physical cores and `OMP_NUM_THREADS` set to 14 (I double checked with `echo $OMP_NUM_THREADS`)

Automatic Parallelization (40 marks)

	Time (s)
Run 1	5.930
Run 2	5.926
Run 3	5.902
Average	5.919

Table 1: Benchmark results for raytrace unoptimized sequential execution

	Time (s)
Run 1	3.825
Run 2	4.009
Run 3	3.858
Average	3.897

Table 2: Benchmark results for raytrace optimized sequential execution

	Time (s)
Run 1	0.579
Run 2	0.580
Run 3	0.580
Average	0.580

Table 3: Benchmark results for raytrace with automatic parallelization

Table 3 shows the time of my modified code with automatic parallelization, it has average time of 0.580s over 3 runs, which has 10.2X speedup compared to unoptimized sequential execution and 6.72X speedup compared to optimized sequential execution.

To speed up the program, I used macros to replace the small repeatable vectors calculation functions, this is because that macro acts like directly insert functions into my source code, and there is no overhead involved in using a macro like there is in placing a call to a function. So that the compiler can parallelize the section where the macro is called. I did not use macro to implement complex functions, this is because the functions are hard to implement and the code size will be very large. The behaviour of the program is preserved since the logic is exactly same as the function, instead of passing in and return value, it simply manipulates the parameters. Also the ppm generated is same as the original output.

Using OpenMP Tasks (30 marks)

	Time (s)
Run 1	1.913
Run 2	1.986
Run 3	2.041
Average	1.980

Table 4: Benchmark results for n-queens sequential execution (Q = 13)

	Time (s)
Run 1	11.694
Run 2	11.648
Run 3	12.042
Average	11.795

Table 5: Benchmark results for n-queens sequential execution (Q = 14)

	Time (s)
Run 1	0.420
Run 2	0.409
Run 3	0.416
Average	0.415

Table 6: Benchmark results for n-queens OpenMP execution (Q = 13)

	Time (s)
Run 1	2.528
Run 2	3.163
Run 3	3.150
Average	2.947

Table 7: Benchmark results for n-queens OpenMP execution (Q = 14)

Table 6 shows the time of my modified n-queens (13 queens) with OpenMP execution, it has average time of 0.415s over 3 runs, which has 4.77X speedup compared to sequential execution.
Table 7 shows the time of my modified n-queens (14 queens) with OpenMP execution, it has average time of 2.947s over 3 runs, which has 4X speedup compared to sequential execution.
To speedup the program, I used openMP tasks to parallelized the first 3 lines of queens matrix, I set cutoff value to 3 because the number of thread is limited, only 14 threads, and with the line increases, the probability of failure to test queen position is large, the overhead to push task into queue and get task from queue will be very big. I test 1,2,3,4 cutoff value and 3 gives the best performance.
The speedup is obvious because that after each worker picking up the task, it will keep running recursively until finishing, so the program will have 14 threads searching for the result at the same time.
I also initialize new config in every task instead of only initializing one time, so that each thread can have its own copy of new config to avoid memory error.
Further changes could be using omp parallel for to parallelize the for loop, which divide iterations to threads to improve the performance.

Manual Parallelization with OpenMP (30 marks)

	Time (s)
Run 1	3.827
Run 2	3.526
Run 3	3.644
Average	3.666

Table 8: Benchmark results for JWT sequential execution (secret length = 4)

	Time (s)
Run 1	159.09
Run 2	104.927
Run 3	101.022
Average	121.680

Table 9: Benchmark results for JWT sequential execution (secret length = 5)

	Time (s)
Run 1	more than 1 hour
Run 2	more than 1 hour
Run 3	more than 1 hour
Average	more than 1 hour

Table 10: Benchmark results for JWT sequential execution (secret length = 6)

	Time (s)
Run 1	0.358
Run 2	0.484
Run 3	0.445
Average	0.429

Table 11: Benchmark results for JWT OpenMP execution (secret length = 4)

	Time (s)
Run 1	7.920
Run 2	7.405
Run 3	9.787
Average	8.391

Table 12: Benchmark results for JWT OpenMP execution (secret length = 5)

	Time (s)
Run 1	268.122
Run 2	264.260
Run 3	519.043
Average	350.475

Table 13: Benchmark results for JWT OpenMP execution (secret length = 6)

Table 11 shows the time of my modified JWT (secret length = 4) with OpenMP execution, it has average time of 0.429s over 3 runs, which has 8.55X speedup compared to sequential execution.
Table 12 shows the time of my modified JWT (secret length = 5) with OpenMP execution, it has average time of 8.391s over 3 runs, which has 14.5X speedup compared to sequential execution.
Table 13 shows the time of my modified JWT (secret length = 6) with OpenMP execution, it has average time of 350.475s over 3 runs, which has more than 10X speedup compared to sequential execution.

I placed the following OpenMP directives in my program:
pragma omp atomic
pragma omp parallel for firstprivate(gAlphabet, secret)
The above directives are effective, I also tried these directives and they were not as effective:
pragma omp task

I use (omp parallel for) to parallelize the for loop for the first letter, which divide 36 iterations by 14 and assign them to 14 threads to improve the performance, I set the gAlphabet and secret as first private values, because we need a copy for each thread on start. I only parallelize the first letter, this is because we only have 14 available threads, each thread will recursively searching for the result. Keep creating threads

will cause a lot of overhead. Ideally this will have 14X speedup and the actual result is close to 14X