ECE 459: Programming for Performance
Assignment 2

Sean Byungyoon Kim

February 15, 2019

I verify I ran all benchmarks on an Intel i7-4720HQ CPU @ 2.60 GHz, 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)

To automatically parallelize the code provided in raytrace_auto.c, macros were used for the functions vectorSub, vectorDot, vectorScale and vectorAdd. The original code won't parallelize as is because the critical loop contains functions that the compiler won't normally parallelize (unless the compiler knows that the function is pure). However, when the function is written as a macro, the macro qualifies as a function that can be parallelized, thus the critical loop can be parallelized as well. Tables 1, 2 and 3 present the results for the unoptimized sequential, optimized sequential and automatic parallelization runtimes.

	Time (s)
Run 1	6.969
Run 2	8.822
Run 3	8.939
Average	8.243

Table 1: Benchmark results for raytrace unoptimized sequential execution

	Time (s)
Run 1	5.565
Run 2	5.568
Run 3	5.563
Average	5.565

Table 2: Benchmark results for raytrace optimized sequential execution

	Time (s)
Run 1	0.453
Run 2	0.479
Run 3	0.475
Average	0.469

Table 3: Benchmark results for raytrace with automatic parallelization

Using OpenMP Tasks (30 marks)

To improve the performance of the algorithm to solve the n-queens problem, #pragma omp directives were implemented into the program. The gcc compiler was used with the -fopenmp flag to utilize the omp directives. 4 omp directives were added to improve the performance of this algorithm: #pragma omp parallel for, #pragma omp task, #pragma omp atomic and #pragma omp taskwait. The #pragma omp parallel for directive is used to parallelize the critical for loop that iterates through the number of queens. The #pragma omp task is used to start tasks in parallel when the recursive call to nqueens is called. The #pragma omp atomic directive is used to explicitly state that only one task at a time is allowed to make changes to the variable count and finally, the #pragma omp taskwait directive is used to make sure the task is completed before the malloc'd variable new_config is freed. As well, a cutoff is implemented so that initially, the program will be run sequentially until it becomes worthwhile for parallelization to occur.
To improve the performance of the program even further, a more optimal cutoff could be selected. Currently, the cutoff was selected through trial and error so a better cutoff could lead to an increase in performance. Tables 4 and 5 present the results.

	Time (s)
Run 1	9.924
Run 2	9.080
Run 3	8.720
Run 4	11.247
Run 5	14.445
Run 6	15.021
Average	11.406

Table 4: Benchmark results for n-queens sequential execution with 14 queens

	Time (s)
Run 1	4.782
Run 2	4.749
Run 3	4.517
Run 4	4.668
Run 5	4.727
Run 6	4.434
Average	4.646

Table 5: Benchmark results for n-queens execution with OpenMP tasks with 14 queens

Manual Parallelization with OpenMP (30 marks)

The sequential runtimes for the brute force JWT decoder can be found in the table below.

	Time (s)
Run 1	125.319
Run 2	118.903
Run 3	109.394
Run 4	99.529
Run 5	126.148
Run 6	82.074
Average	110.228

Table 6: Benchmark results for sequential runtimes of brute force JWT decoder with secret length of 5

Within the program, I included the following #pragma directives: #pragma omp parallel for, #pragma omp task, #pragma omp taskwait and #pragma omp critical. The parallel for directive was effective by parallelizing the for loops within the main and brute_sequential function. The pragma task directives were effective in parallelizing the aforementioned for loops and by handling the recursive calls. The critical directive was effective in checking for a valid secret being returned from the decoder, as well as when the secret to be checked for validity is being updated. Finally, the taskwait directive was used to wait for a task to return a secret from the decoder.
I also tried utilizing the cancel and cancellation point directives within the program, but they weren't very effective. Using the cancel and cancellation point directives requires the OMP_CANCELLATION environment variable to be set to true. However, the performance of the program ended up not being as optimal compared to running the program without those directives. As well, setting only the string being adjusted for secret validity was the only private variable within the parallelization. Setting any other variables to private led to incorrect or segmentation faults.
With all annotations applied, here are the results:

	Time (s)
Run 1	24.320
Run 2	26.027
Run 3	39.646
Run 4	37.972
Run 5	34.626
Run 6	34.260
Average	32.809

Table 7: Benchmark results for manual OMP runtimes of brute force JWT decoder with secret length of 5