UNIVERSITY OF WATERLOO

ECE 459 - Assignment 2

Author:
David Chau

Student Number:
20623345

Date: February 12, 2020

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

Automatic Parallelization

The task in this section was to make the necessary changes to a raytrace program in order for Oracle's Solaris Studio to parallelize the program. The parallelizations in this case affected the profitable loop, which is the outer loop of 60000 iterations.

The main change was converting the function calls within the profitable loop into `#define` macros. As function calls can have arbitrary side effects, compilers will typically avoid parallelizing loops with function calls. On the other hand, macros are just fragments of code that have been given a name. Whenever the name is used, it is replaced inline by the contents of the macro at compile time. As such, the behaviour intended by the functions can be preserved without actually using functions. This was verified by writing basic test cases in `run_tests()`, as well as by comparing the image output from the sequential and parallel versions of the raytrace program. By making this change, the profitable loop was then parallelized by the compiler.

Since macros can be difficult to understand and have many nuances, these changes also adversely impact the maintainability of the code. This is especially true if any complex logic is required to be added to the existing macros, as they are by nature not as easy to work with when compared to functions in this respect.

Unrelated to enabling the compiler to parallelize, an additional change of writing to `/tmp` was done based on advice given by the instructor. This was at the suspicion that the `/home` directory was being limited by NFS.

Making these changes resulted in a noticeable speedup as indicated by Table 1. As a baseline, the unoptimized sequential version took on average 7.167s to execute, while the optimized sequential version took 3.853s. Through automatic parallelization, the raytrace program was able to run in 0.4743s. This is roughly 15.11x and 8.12x faster than the unoptimized and optimized versions, respectively.

Table 1: Benchmark results for different raytrace executions

Test Case	**Mean (s)**	σ **(s)**	**Min (s)**	**Max (s)**
Unoptimized	7.167	0.710	6.142	7.752
Optimized	3.853	0.208	3.496	4.093
Parallelized	0.4743	0.0155	0.4480	0.4992

Using OpenMP Tasks

The goal of this section was to parallelize the n-queens problem using OpenMP tasks. Specifically, the following OpenMP directives were included:

- `#pragma omp parallel`
  This was used to indicate that parallel execution was starting.
- `#pragma omp single`
  This was used to limit the number of threads calling the initial `nqueens` to just one thread.
- `#pragma omp task firstprivate(j)`
  This was used to split off subsequent calls to `nqueens` as tasks. In order to avoid overloading the system with tasks and to ensure a speedup, this was limited to the first level of recursion using the `RECURSION_DEPTH` variable. Because of this, further calls to `nqueens` after the first level of recursion will result in sequential execution. Additionally, it should be noted that each task operates on their own copy of `new_config` and `j` in order to avoid race conditions.
- `#pragma omp atomic`
  This was used to update the `count` variable in a thread-safe manner once a solution was found.

Given that at the first level of recursion, there will be a guaranteed `n` calls to `nqueens`, making these changes enabled these `n` calls to be run in parallel. As noted prior, subsequent calls to `nqueens` will be run in serial in their respective threads. As a result, a performance increase of 5.93x for `n=13` and 7.07x for `n=14` was achieved. This can be seen in Table 2.

Table 2: Benchmark results for different n-queens executions

Test Case	**Mean (s)**	σ **(s)**	**Min (s)**	**Max (s)**
Sequential, n=13	1.724	0.064	1.675	1.895
Sequential, n=14	10.502	0.064	10.406	10.627
Parallelized, n=13	0.2907	0.0103	0.2787	0.3135
Parallelized, n=14	1.485	0.047	1.435	1.585

One further optimization was made in order to improve performance, which affected cases where `RECURSIVE_DEPTH` ≥ 1 (i.e., sequentially executed portion). This change was moving the `malloc` and `free` of `new_config` out of the sequentially executed for-loop, reducing the number of allocations and deallocations per call to `nqueens` from `j` to 1. This can be done as the index of `new_config` will just be overwritten by further iterations. As indicated by Table 3, this lead to performance increases of 10.53x for `n=13` and 13.69x for `n=14`, when compared to the sequential version.

Table 3: Benchmark results for `malloc` optimized n-queens executions

Test Case	**Mean (s)**	σ **(s)**	**Min (s)**	**Max (s)**
Optimized, n=13	0.1637	0.0094	0.1496	0.1787
Optimized, n=14	0.7669	0.0195	0.7421	0.8074

One final thing to note is that the final parallelized program was checked with `valgrind`. Doing this showed a number of "possibly lost" and "still reachable" blocks, the exact number of which not varying with input size. Based on this and discussions held on Piazza, it was concluded that this was an issue with OpenMP and not the implemented code.

Manual Parallelization with OpenMP

The goal of this section was to manually parallelize a program capable of decoding the secret from a JSON Web Token (JWT). The implementation of the sequential decoder ended up being a back-tracking problem. Because of this, initially a similar methodology to the one used to parallelize the n-queens problem was employed. The following OpenMP directives were used:

- `#pragma omp parallel`
- `#pragma omp single`
- `#pragma omp task firstprivate(i)`

These resulted in the runtimes seen in Table 4. However, this task-based methodology ended up being deemed not as effective, relative to the methodology that will be discussed next. This is potentially attributed to the fact that tasks in this context incur a significant amount of overhead and are not as optimized for these types of computations.

Table 4: Benchmark results for JWT secret decoder executions using task-based methodology

Test Case	**Mean (s)**	**σ (s)**	**Min (s)**	**Max (s)**
Parallelized, `JWT_TOKEN_4`	1.090	0.278	0.795	1.668
Parallelized, `JWT_TOKEN_5`	26.617	4.251	19.851	31.803
Parallelized, `JWT_TOKEN_6`	508.655	63.551	463.717	553.592

What ended up being used in the final solution is a for-based methodology, using the following OpenMP directive:

- `#pragma omp parallel for private(i)`

In this case, the iterations of the loop will be distributed among the given team of threads, without any overhead from task creation. Each thread will therefore be solving different branches of the back-tracking problem at the same time, effectively parallelizing the given program. The performance of this methodology is shown in Table 5. Overall it is evident that runtimes have improved relative to the task-based methodology. Additionally, the decoder on `JWT_TOKEN_4` and `JWT_TOKEN_5` runs 12.57x and 18.84 faster than the sequential version, respectively. Additionally, JWT_TOKEN_6 is now able to run in roughly 257 seconds, instead of running indefinitely in the sequential case.

Table 5: Benchmark results for JWT secret decoder executions

Test Case	**Mean (s)**	σ **(s)**	**Min (s)**	**Max (s)**
Sequential, `JWT_TOKEN_4`	2.853	0.269	2.615	3.400
Sequential, `JWT_TOKEN_5`	86.178	2.353	83.957	90.953
Sequential, `JWT_TOKEN_6`	indefinite	N/A	N/A	N/A
Parallelized, `JWT_TOKEN_4`	0.2269	0.0781	0.1558	0.3759
Parallelized, `JWT_TOKEN_5`	4.574	1.098	2.705	5.589
Parallelized, `JWT_TOKEN_6`	257.061	63.625	143.784	348.999

Similar to what was done for Question 2, the final parallelized program was checked with `valgrind`. Doing this showed a number of “possibly lost” and “still reachable” blocks, the exact number of which not varying with input size. Based on this and discussions held on Piazza, it was concluded that this was an issue with OpenMP and not the implemented code.