ECE 459: Programming for Performance
Assignment 2

Pavel Shering

July 12, 2018

I verify I ran all benchmarks on ece-tesla0 with 14 physical cores and `OMP_NUM_THREADS` set to 14 (I double checked with `echo $OMP_NUM_THREADS`)

Automatic Parallelization (15 marks)

The following tables present the data for various optimizations of *ray_trace* $*$ *.c* file.

	Time (s)
Run 1	5.632
Run 2	5.649
Run 3	5.434
Average	5.572

Table 1: Benchmark results for raytrace unoptimized sequential execution

	Time (s)
Run 1	3.466
Run 2	3.058
Run 3	3.143
Average	3.222

Table 2: Benchmark results for raytrace optimized sequential execution

	Time (s)
Run 1	0.520
Run 2	0.548
Run 3	0.549
Average	0.539

Table 3: Benchmark results for raytrace with automatic parallelization

Table 1 represents the true benchmark for sequential execution of the `ray_trace` algorithm. The sequential version runs in average of 5.572 seconds, which will be used as the comparison for the further optimizations. **Table 2** enables compiler optimizations of the sequential version providing

an execution time of 3.222 seconds.

To further optimize the `ray_trace` algorithm the sequential version of the code requires changes. Specifically the calls to functions within the profitable loop (the outside loop of 60,000 iterations on individual rays) need to be removed. In order to successfully parallelize the outside loop the functions calls are converted into `#define` macros. This allows for the compiler to replace the function calls with those macros directly at compile time removing any functions calls and allowing for the compiler to create threads to compute reach ray individually.

It was critical to test that the changes did not effect the behavior of the sequential program. The behavior is preserved verified by comparing the image output from sequential and parallel versions, as well as, the macros perform the same math operations just inline instead of being passed to the functions for computations. However the macros are very difficult to maintain and understand. If the way the vectors are manipulated changes it will be very difficult to change the macros to the match the correct outcome. That being said, the auto parallelized (with `gcc -O2`) version shown in **Table 3** obtains **10.34x** and **5.98x** speedups against the sequential version in **Table 1** and compiler optimized sequential version in **Table 2**, respectively. Thus, the optimization is definitely beneficial, but will cost extra effort on maintaining the code in terms of the macros. However, since its not suspected that the manipulation for 3D vectors will change any time soon, its safe to use the auto parallelized version going forward since the behavior is already preserved.

Using OpenMP Tasks (30 marks)

The following tables illustrate the runtime results for sequential and omp parallelized versions of nQueens algorithm, where n represents the board size and the amount of Queens to be placed.

Run #	**n = 13 Time (s)**	**n = 14 Time (s)**
Run 1	3.197	13.518
Run 2	3.145	13.493
Run 3	2.166	13.484
Run 4	2.119	13.617
Run 5	2.090	13.554
Run 6	2.095	13.880
Run 7	2.097	13.378
Run 8	2.116	15.216
Run 9	2.202	13.562
Run 10	2.316	13.329
Average	**2.3543**	**13.7031**

Table 4: Benchmark results for nQueens sequential execution (`n` = 13 & 14)

Run #	**n = 13 Time (s)**	**n = 14 Time (s)**
Run 1	0.178	1.000
Run 2	0.119	0.855
Run 3	0.211	0.855
Run 4	0.111	0.856
Run 5	0.196	0.852
Run 6	0.109	0.853
Run 7	0.108	0.872
Run 8	0.108	0.887
Run 9	0.095	0.870
Run 10	0.158	0.833
Average	**0.1393**	**0.8733**

Table 5: Benchmark results for nQueens execution with OpenMP tasks (`n` = 13 & 14) resulting in **16.901x** and **15.692x** speedups respectively

Table 4 shows the bench mark for the sequential nQueens algorithm with -02 compiler optimization thus creating a benchmark to compare to for OpenMP parallelization. The required minimum speedup are **4x** with $n = 13$, **1.75x** with $n = 14$. **Table 5** shows the resulting speedups of **4x** with $n = 13$, **1.75x** with $n = 14$.

In order to achieve such speedups the sequential code is modified to be use with OpenMP tasks. The first change is not related to OpenMP, but its optimizing the sequential code in general by moving the malloc of `new_config` to outside of the **for** loop thus allowing overwrites of the same memory but saving the time to allocating and freeing the `new_config` j times. This alone provides a **2.5x** speed up for both $n = 13$ and $n = 14$.

Next, nQueens is a backtracking problem, which means it will have a fixed number of next moves from a given solution state regardless of the node of interest in the solution tree. This makes it a perfect candidate for OpenMP tasks because the task can be spawned from any node and propagated to then combine into a solution count, behaving like a depth first search on a tree. The sequential algorithm is further modified to create tasks in the `nqueens` function when it is safe to place a queen creating the next set of possible solutions in another task which gets the current state in the `new_config` thus allowing for independence from the rest of the solutions. However, this will overload the system with tasks, and thus the task creation must be limited to a depth level in the solution tree. To preserve the sequential behavior each task is given its own `new_config_task` for use to continue the depth search thus avoiding a race condition for using the same `new_config`.

I cannot speculate any further changes to speed this code up further, besides using more cores and processors and thus increasing the OpenMP tasks limit.

Manual Parallelization with OpenMP (55 marks)

I placed the following OpenMP directives in my program:

- `#pragma omp parallel`
- `#pragma omp single`

- `#pragma omp task firstprivate(i, particle_final_pos, particle_init_pos, d, k_forces)`
- `#pragma omp taskwait`

All directives from the list above are effective as that is what parallelizes the program. The `#pragma omp parallel` allows for parallelization of the `huns_method` call, and `#pragma omp single` does the call in a single thread. Next, the `#pragma omp task firstprivate(i, particle_final_pos, particle_init_pos, d, k_forces)` allows to parallelize the calculation of forces and position for each particle. Thus, for each particle a task is created and the task then computes the net force and new position of the i^{th} particle. Finally, the `#pragma omp taskwait` allows to preserve the sequential algorithm behavior and correctness.

The directive `#pragma omp for` did not provide as much of a speed up a the tasks, thus the sequential algorithm was re-factored for optimizing OpenMP tasks.

With all annotations applied, the results are displayed in **Table 6**:

Run #	**Sequential Time (s)**	**OpenMP Time (s)**
Run 1	33.589	4.708
Run 2	50.388	4.593
Run 3	44.655	4.188
Run 4	34.249	4.181
Run 5	37.076	4.269
Run 6	37.973	4.177
Run 7	51.767	4.168
Run 8	36.819	4.28
Run 9	34.819	4.634
Run 10	38.533	4.183
Average	**39.986**	**4.3381**

Table 6: Benchmark results for particle position propagation with 4000 sample size resulting in **8.295x** speedup

As mentioned before a task is created to update the net forces and position of each individual electron. The particle's forces and position calculations can be done separately from another particle, thus allowing for the OpenMP to compute the force and position of 14 electrons in parallel. Thus, speeding up the computation when amount of particles exceeds the overhead of creating tasks. The check for error between the averaged position and initial position propagation cannot be optimized with tasks as the **break** statement already allows for escaping calculating the difference for all particles.