ECE 459: Programming for Performance
Assignment 2

Your Name

February 13, 2020

I verify I ran all benchmarks on a Intel(R) Xeon(R) Gold 5120 CPU with 14 physical cores and `OMP_NUM_THREADS` set to 14 (I double checked with `echo $OMP_NUM_THREADS`)

Automatic Parallelization (15 marks)

	Time (s)
Run 1	8.461
Run 2	8.223
Run 3	8.898
Average	8.527

Table 1: Benchmark results for raytrace unoptimized sequential execution

	Time (s)
Run 1	4.586
Run 2	4.543
Run 3	4.515
Average	4.548

Table 2: Benchmark results for raytrace optimized sequential execution

	Time (s)
Run 1	0.5839
Run 2	0.6124
Run 3	0.5846
Average	0.5936

Table 3: Benchmark results for raytrace with automatic parallelization

1. My change:

```
/* Subtract two vectors and return the resulting vector */
#define vectorSub(v1, v2) (vector) {*v1.x - *v2.x, *v1.y - *v2.y, *v1.z - *
    v2.z }

/* Multiply two vectors and return the resulting scalar (dot product) */
#define vectorDot(v1, v2) (float)(*v1.x * *v2.x + *v1.y * *v2.y + *v1.z * *
    v2.z)

/* Calculate Vector x Scalar and return resulting Vector*/
#define vectorScale(c, v) (vector) {*v.x * c, *v.y * c, *v.z * c }

/* Add two vectors and return the resulting vector */
#define vectorAdd(v1, v2) (vector) {*v1.x + *v2.x, *v1.y + *v2.y, *v1.z + *
    v2.z }
```

   I have converted all the vector operation functions into equivalent macros.

2. Justify my change

   (a) Why the existing code does not parallelize?

   - In the existing code, there are function calls in the double for loop. The problem with calling another function is that the compiler has no idea what that routine might do. Therefore, a loop that contains function calls cannot, in general, be automatically parallelized.

   (b) Why my changes improve parallelization and preserve the behaviour of the sequential version?

   - After vector-operation functions are converted to macros, those functions are handled by the pre-compiler and guaranteed to be inlined. By inlining functions using Macros, the double for-loop containing those function calls are parallelized automatically.
   - My change preserves the behaviour of the sequential version. This is because when changing the vector operation functions to macros and changing the parameter from pointer to value, it preserves the function behaviour.

   (c) Why my changes adversely impact maintainability?

   - Changing a function to a marco is not maintainable. The first reason is that macro does not support namespaces, which means those Macro names will collide with the other variable or function names and cause unwanted behaviour. The second reason is that Macros can't be debugged. A "function-like" macros will act like a single statement, which makes it hard to figure out what is going on. The third reason is that the address cannot be passed as a parameter to a "function-like" macros.

3. Speed up Explanation

   - There are around 14x speedup over bin/raytrace, and 7x speedup over bin/raytrace_opt. This result is as expected. The double for loop was auto parallelized. The work of the for loop is evenly distributed to 14 threads since the OMP_NUM_THREADS environment variable is 14. This confirms the 14x speedup over bin/raytrace. Since bin/raytrace_opt has 2x faster than bin/raytrace, bin/raytrace_auto has 14 / 2 = 7x speedup over bin/raytrace_opt.

Using OpenMP Tasks (30 marks)

	Time (s)
Run 1	1.792
Run 2	1.897
Run 3	1.823
Run 4	1.733
Run 5	1.742
Run 6	1.758
Average	1.791

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

	Time (s)
Run 1	0.3367
Run 2	0.3434
Run 3	0.3426
Run 4	0.3440
Run 5	0.3323
Run 6	0.3394
Average	0.3397

Table 5: Benchmark results for n-queens execution with OpenMP tasks (`n` = 13)

	Time (s)
Run 1	11.318
Run 2	11.815
Run 3	11.259
Run 4	11.687
Run 5	11.565
Run 6	11.453
Average	11.516

Table 6: Benchmark results for n-queens execution (`n` = 14)

	Time (s)
Run 1	1.749
Run 2	1.729
Run 3	1.721
Run 4	1.736
Run 5	1.744
Run 6	1.731
Average	1.735

Table 7: Benchmark results for n-queens execution with OpenMP tasks (`n` = 14)

1. My change:

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int j=0; j<n; j++)
        {
            #pragma omp task firstprivate(j)
            {
                char *new_config;
                new_config = malloc((1)*sizeof(char));
                new_config[0] = j;

                nqueens(new_config, n, 1);

                free(new_config);
            }
        }
    }
}
```

   To parallelize the solving of an n-queens problem, I have divided the n-queens problem into n sub-problems, where n is the possible positions for the first row of the queen matrix. More specifically, each sub-problem is a new n-queens problem which has the first row filled out. Each sub-problems is an OpenMP task, which can be parallelized.

2. Justify my change

   (a) Analyzing the performance of the provided version

   - The provided solution is a sequential version that brutal forced the n-queens problem. The provided sequential version has an $O(n^n)$ time complexity, which is an NP problem. In the recursion tree, the first level has n nodes, because there are n possible positions on the first row. For each possible position on the first row, there are n possible positions on the second row. Therefore, there are n^2 nodes on the second level. Since there are total n rows in the queen matrix, there are n levels. Therefore, the time complexity of the provided version is n^n, and this explains why the performance is so bad.

   (b) Why my changes improved performance?

   - After dividing the n-queens problem into n sub-problems, the n sub-problems are evenly distributed to 14 threads. And each sub-problem has only n - 1 levels and time complexity of $O(n^{n-1})$. Therefore, the total time complexity is optimized to $O((n/14) * (n^{n-1})) = O((n^n)/14)$

   (c) How you could further improve performance?

   - My solution can be further improved by creating new tasks from the running thread and keeping the early exited thread working. More specifically, some sub-problems exit earlier and the corresponding thread becomes idle. Therefore, we can create a new task whenever the omp_set_num_threads() is less than 14. By doing this, we can keep all 14 threads working consistently and better distribute the workload.

3. Speed up explanation

   - For both n = 13 and 14, there is around 7x speedup over the provided sequential solution. This is as expected. When n = 14, the provided solution has $O(14^{14})$ time complexity, and the parallelized solution has $O(14^{13})$ time complexity. The theoretical speedup is 14. However, some tasks are finished early, and the corresponding threads become idle. Even though the sub-problems are distributed to 14 threads, some sub-problems have higher time complexity on the constant level, which means the total workload is not evenly distributed to 14 threads. Therefore, the actual speedup is 7x, which is less than the theoretical speedup.

Manual Parallelization with OpenMP (55 marks)

I placed the following OpenMP directives in my program:

The following directives were effective:

- ```
  #pragma omp parallel for
  ```

  This parallel-for directive works. Inside the first recursion call, there is a for-loop that loop through all the character in the gAlphabet. For each iteration, a new secret is generated and checked if it is a valid secret, and the function recursively calls itself with the new secret. By using the parallel-for directive in front of the for-loop, we can distribute the 36 units of work to 14 threads for first recursion call. And later on, when one thread becomes idle, the parallel-for directive will detect the idle thread and distribute the for-loop workload again to the idle thread, which guarantees that all 14 threads are operating till the end.

- ```
  #pragma omp parallel
  {
      #pragma omp single
      {
          for (int i = 0; i < gAlphabet.size(); i++)
          {
              #pragma omp task
              {
                  std::string new_secret(1, gAlphabet[i]);

                  if (isValidSecret(message, origSig, new_secret)) {
                      cout << new_secret << endl;
                      found = 1;
                  }

                  dfs(gAlphabet, new_secret, gMaxSecretLen);
              }
          }
      }
  }
  ```

  Using OpenMP tasks works. Similar to Part 2, to parallelize the generation of all possible secret, I have divided the problem into n sub-problems, where n is the number of possible characters in the JWT's secret. In my testing, n is 36. More specifically, each sub-problem is a new secret generation problem that has the first character filled out. Each sub-problems is

an OpenMP task, which can be parallelized. I use one thread to create all 36 tasks. And all 36 tasks are queued and will be handled concurrently by 14 threads. Therefore, parallelization is achieved using explicit OpenMP tasks.

I tried these directives and they were not effective:

- `#pragma omp for ordered`

  The Ordered directive does not work. Without the parallel directive, the ordered directive by itself does not have the parallelization effect. In our case, inside the for-loop that loop through all the possible characters, the new secret verification is independent of each other. Therefore, there is no point in using the ordered directive.

- `#pragma omp single`

  The Single directive does not work. Without the parallel directive, the Single directive by itself does not have the parallelization effect. When using the single directive, only a single thread executes the bottleneck for-loop region. Therefore, there won't be any parallelization for using Single directive.

- `#pragma omp master`

  The Master directive does not work. Similar to Single directive, without the parallel directive, the Master directive by itself does not have the parallelization effect. When using the master direction, only the master thread executes the bottleneck for-loop region, which is the same as the sequential version. Therefore, using Master directive by itself does not work.

- `#pragma omp parallel sections`

  The Sections directive does not work, even though the section directives inside the sections directive can be parallelized. This is because the task we want to parallel is a for-loop and create multiple section directions using for loop throws an error. One way to overcome this is to flatten out the for-loop, but this is impractical in programming since the number of possible characters can vary.

With the parallel-for directive applied, here are the results:

	Time (s)
Run 1	0.183
Run 2	0.188
Run 3	0.178
Run 4	0.191
Run 5	0.185
Run 6	0.184
Average	0.185

Table 8: Benchmark results for jwtcracker_sin (sequential version) (`length of secret` = 4)

	Time (s)
Run 1	2.579
Run 2	2.584
Run 3	2.604
Run 4	2.593
Run 5	2.605
Run 6	2.591
Average	2.593

Table 9: Benchmark results for jwtcracker_omp with manual OpenMP (`length of secret` = 4)

	Time (s)
Run 1	7.375
Run 2	7.290
Run 3	7.790
Run 4	7.072
Run 5	7.858
Run 6	7.281
Average	7.444

Table 10: Benchmark results for jwtcracker_sin (sequential version) (`length of secret` = 5)

	Time (s)
Run 1	102.424
Run 2	102.183
Run 3	101.134
Run 4	101.783
Run 5	102.165
Run 6	102.672
Average	102.060

Table 11: Benchmark results for jwtcracker_omp with manual OpenMP (`length of secret` = 5)

	Time (s)
Run 1	313
Run 2	301
Run 3	305
Average	306

Table 12: Benchmark results for jwtcracker_sin (sequential version) (`length of secret` = 6)

	Time (s)
Run 1	4041
Run 2	4118
Run 3	4154
Average	4104

Table 13: Benchmark results for jwtcracker_omp with manual OpenMP (`length of secret` = 6)

Speed up explanation:

- For all secret length of 4, 5 and 6, the jwtcracker_omp has around 14x speedup over the single-thread solution. This is as expected. For the first recursion call, all 36 units of work are distributed to 14 threads. And later on, when one thread becomes idle, the parallel-for directive will detect the idle thread and distribute the workload to the idle thread, which guarantees that all 14 threads are operating till the end. This confirms that there is a 14x speedup over the provided single-thread solution.