ECE 459: Programming for Performance
Assignment 2

Jinming Zhang

February 13, 2020

I verify I ran all benchmarks on a ecetesla0(Intel(R) Xeon(R) Gold 5120 (Q3-2017) 14-core 2.2 to 3.2GHz) with at least 14 physical cores and `OMP_NUM_THREADS` set to 14 (I double checked with `echo $OMP_NUM_THREADS`). Note: Since ecetesla0 is highly occupied by other users, the experimental data for this report might fluctuated or even inaccurate.

Automatic Parallelization (15 marks)

	Time (s)
Run 1	7.526
Run 2	8.456
Run 3	7.752
Average	7.911

Table 1: Benchmark results for raytrace unoptimized sequential execution

	Time (s)
Run 1	4.674
Run 2	3.823
Run 3	3.906
Average	4.134

Table 2: Benchmark results for raytrace optimized sequential execution

	Time (s)
Run 1	0.6059
Run 2	0.5868
Run 3	0.6116
Average	0.6014

Table 3: Benchmark results for raytrace with automatic parallelization

The experimental result for three versions of the raytrace is shown in Tables 1, 2, and 3 above(obtained from ”hyperfine”). From the time average, it is clear that the automatic parallelization solution (bin/raytrace_auto) is about **13.2x over bin/raytrace and 6.9x over bin/raytrace_opt**.

The original unoptimized version of the raytrace is the slowest when compared to other implements in the experiment. This is because Oracle's Solaris compiler refused to parallelize the existing code as it contains unsafe dependencies automatically. There are multiple function calls (vectorSub, vectorDot, etc.) in the five main for-loops in the original unoptimized version of the raytrace, which makes the compiler to consider it as unsafe dependencies. The problem with the function call is that the compiler is unsure what that routine might change global data or never return. This will create unsafe dependencies, and because of that, a loop that contains function calls cannot be automatically parallelized by the Oracle's Solaris compiler.

One of the ways to deal with the unsafe dependencies is to change those functions into macro definitions shown in Listing 1 below.

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

/* Multiply two vectors and return the resulting scalar (dot product) */
#define VECTORDOT(v1, v2){(v1).x * (v2).x + (v1).y * (v2).y + (v1).z * (v2).z}
// float vectorDot(vector *v1, vector *v2){
//  return v1->x * v2->x + v1->y * v2->y + v1->z * v2->z;
// }

/* Calculate Vector x Scalar and return resulting Vector*/
#define VECTORSCALE(c, v){(v).x * c, (v).y * c, (v).z * c}
// vector vectorScale(float c, vector *v){
//          vector result = {v->x * c, v->y * c, v->z * c };
//          return result;
// }

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

Listing 1: Macro Functions

The function definitions for vectorSub, vertorDot, vectorScale, and vectorAdd are changed into Macro definitions. In c, macros can accept parameters and return values and become macro functions. With the macro changes, the compiler has pre-compile these macro functions into inline operations. This will eliminate most of the unsafe dependencies from function calls in the loops. Now, Oracle's Solaris compiler can automatically parallelize the loops in the existing code. The behaviour of the sequential version is preserved from this modification because all the alternative macro function is in pure form. These macro functions only use the parameter input, compute the output, and then return. They never change the input parameters or any global variables. Although the input parameters changed from pass by reference to value, these macro functions preserve the functionality of the original function. This means that all the changes leave the sequential version of the code unchanged. The Solaris compiler will automatically parallelize the code without changing the original behaviour of the code. So in the end, the behaviour of the automatic parallelize version is preserved from this modification and the output image.ppm will be the same. When I do a "diff" command on the output of raytrace_opt and raytrace_auto, there are no difference which again proofs the behaviour is preserved.

```
Compiling Part 1 Automatic Parallelization
/opt/oracle/solarisstudio12.3/bin/cc -lm -fast -xautopar -xloopinfo -xreduction
    -xbuiltin q1/raytrace_auto.c -o bin/raytrace_auto
"q1/raytrace_auto.c", line 215: PARALLELIZED
"q1/raytrace_auto.c", line 216: not parallelized, not profitable
"q1/raytrace_auto.c", line 231: not parallelized, loop has multiple exits
"q1/raytrace_auto.c", line 239: not parallelized, unsafe dependence (
    currentSphere t)
"q1/raytrace_auto.c", line 262: not parallelized, not profitable
```

Listing 2: Compiler Output

As we can see here, the outer most for-loop can be parallelized by the compiler and usually, this will result in the biggest performance boost. The performance boost from the automatic parallelization solution is about 13.2x over the original unoptimized version. From the class, we learned that Under the hood, most parallelization frameworks use OpenMP, which including the Solaris compiler. We can control the number of threads with the OMP_NUM_THREADS environment variable. Since in this assignment, we set OMP_NUM_THREADS to 14, there will be 14 threads used for parallelization. The theoretical speedup is about 14 times, which is close to the experimental value to 13.2 (from table 1 and table 3). There is limited information about the original, optimized version. When comparing the original unoptimized and optimized version, we can see a trend that the optimized version is close to 2 times the performance of the unoptimized version. Theoretically, if the speed up when comparing the unoptimized version and automatic parallelized version is 14 times, then the speed up when comparing the optimized version and automatic parallelized version should be about 7 time. From table 2 and table 3 of the experimental result, it is about 7 times which is close to the theoretical prediction.

The advantage of a macro over an actual function is the speed, as shown in the experiment. However, this change from actual function to macro function will adversely impact maintainability. Firstly, macros can't be debugged. With a debugger, it is easy to see what variables are translated into, but we can never see what the macros are translated into. So you don't actually know what is going on in the code, which makes the code hard to maintain. Secondly, macros are hard to translate into other languages. Some other languages, like python, don't even macros. So if we want to migrate our code from c to python, it is extremely hard to convert macro function to an equivalent version in other languages. Thirdly, we cannot pass a function or an operation into a macro function. Use an operation like a++ or a function return may fail the compilation or lead to unexpected behaviours. This makes macro function hard to use and hard to maintain.

Using OpenMP Tasks (30 marks)

	Time (s)
Run 1	19.252
Run 2	15.247
Run 3	18.595
Run 4	20.335
Run 5	20.340
Run 6	18.714
Average	18.747

Table 4: Benchmark results for nqueens sequential execution (`n` = 14)

	Time (s)
Run 1	3.502
Run 2	5.000
Run 3	3.099
Run 4	4.884
Run 5	4.519
Run 6	3.743
Average	4.125

Table 5: Benchmark results for nqueens execution with OpenMP tasks (`n` = 14)

	Time (s)
Run 1	3.186
Run 2	2.914
Run 3	3.053
Run 4	2.802
Run 5	3.239
Run 6	3.124
Average	3.053

Table 6: Benchmark results for nqueens sequential execution (`n` = 13)

	Time (s)
Run 1	0.480
Run 2	0.568
Run 3	0.579
Run 4	0.764
Run 5	0.425
Run 6	0.647
Average	0.577

Table 7: Benchmark results for nqueens execution with OpenMP tasks (`n` = 13)

The provided version is using a 1D array, ”config,” to store the solutions for the nqueesn problem. It will start with the first row of the n * n space and check for each of the positions (n positions) in the row to see if it is safe for a queen to be placed with the ”safe” function. If that specific position is safe, the ”nqueens” function will start to recursively call itself and traverse through all the possible queen placements. Since there are, n positions in a given row and each position have a depth of n, the run time for the provided version is n^n. The performance for this provided version is slow since the run time grows exponentially. The provided version is trying to solve the nqueens question with brutal force, and there is no parallelization involved in any of the recursive calls on the nqueens function, which makes the performance extremely slow when n is large.

If we want to improve the performance of the provided version, we can use OpenMP tasks for parallelizing the recursive calls of the nqueens function. Since there are n positions for the first row, the way I chose to boost the performance of the nqueen algorithm is to use n dedicated OpenMP tasks to traverse through n-depth for each of the n positions in the first row. My changes are shown in Listing 3 below.

```c
void nqueens(char *config, int n, int i)
{
    int j;

    if (i==n)
    {
        #pragma omp atomic
        count++;
    }

    /* try each possible position for queen <i> */
    for (j=0; j<n; j++)
    {
        #pragma omp task if(i == 0) shared(config, i, n) firstprivate(j)
        {
            char *new_config;
            /* allocate a temporary array and copy the config into it */
            new_config = malloc((i+1)*sizeof(char));
            memcpy(new_config, config, i*sizeof(char));
            if (safe(new_config, i, j))
            {
                new_config[i] = j;
            nqueens(new_config, n, i+1);
            }
            free(new_config);
        }
    }
    // sync
    return;
}

```

```
#pragma omp parallel
{
    #pragma omp single
    {
        nqueens(config, n, 0);
    }
}

```

Listing 3: Q2 Code Changes

This is similar to the idea of parallelizing the outermost for-loop in n-depth nested for-loops, which will achieve the most beneficial performance boost. The speedup due to my changes is shown in Tables 4, 5, 6 and 7, with 13 and 14 queens presented. From the time average, it is clear that the solution parallelized with OpenMP Tasks is about **5.3x with n=13, 4.5x with n=14**.

My changes have improved performance. This is because my change used OpenMp tasks to parallelized the first layer of the recursive calls on the "nqueens" function. Since there are 14 positions in the first row of the space (take 14 as an example), there will be 14 threads to handle the n - 1 depth of the recursion. At this point, the run time becomes $n^n/14$. This means that there should be a theoretical speedup of 14 times. However, the experimental data is about a 5 times speed. This is due to the unbalanced workload distribution of those 14 threads. Some of the threads will early exit due to the fact that there is no valid solution for that initial solution. This will leave other threads with a larger burden for computing more stuff. This means that my change is not fully parallelized the nqueen algorithm, which results in a speedup that is lower than 14 times. At the same time, I noticed that, although we set the "OMPNUMTHREADS" to 14, we usually don't get 14 CPU due to a big load on ecetesla0. I usually get 2 to 3 full CPU for executing my omp version code. This means that I will only have 4 to 6 working threads to achieve a 4 to 6 times speedup. This quantity of the speed up matches the experimental value I have.

Apart from my changes, other reasonable changes can further boost the performance of the nqueens algorithm. One reason why the speed up for the experimental and theoretical value doesn't match is because of the unbalanced workload for different threads. A reasonable speculation to solve this problem is to balance the workload. Some of the threads will allow early exit due to the fact that there is no valid solution for that initial solution. If some threads early exit, they should create new tasks in order to help other ongoing threads to finish their recursive branches. Code wise, we can keep track of the number of active threads with the opm_set_num_threads()function. If that number is lower than the number of total available threads(in this case 14), new threads should be created to handle leftover tasks for solving the nqueens problem. This will reasonably balance the workloads for threads and boost the performance.

Manual Parallelization with OpenMP (55 marks)

I placed the following OpenMP directives in my program:

- **#pragma omp parallel**
- **#pragma omp single**
- **#pragma omp task**
- **#pragma omp parallel for**
- **#pragma omp ordered**
- **#pragma omp parallel sections**
- **#pragma omp master**
- **#pragma omp barrier**
- **#pragma omp master**

The following directives were effective:

- **#pragma omp parallel for**
  This directive will parallelize the given for-loop. Iterations of the for-loop will be distributed among the current team of threads. This is the directive I used for Q3 shown in Listing 4.

```
#pragma omp parallel for
for (int i = 0; i < strlen(alphabet); i++) {
    string newString;
    newString = subString + alphabet[i];
    // cout << newString << endl;
    findPrintCombAndValidate(alphabet, newString, alphabetLength, message,
   origSig);
}

```

  Listing 4: Manual Parallelization with OpenMP

  The parallel part of the directive will spawn a number of threads (in this case, 14) and the for part of the directive will handle all the iterations of the for loop distributively. Since there are 36 characters in the alphabet string, there will be 36 calls to "findPrintCombAndValidate" function in the first layer of the recursion. However, there are only 14 threads that are available. This means that all the threads will be occupied during the first layer of the recursion. If there are some threads early exits, later on, they will relieve other threads' workloads to achieve parallelization fully. Since the directive separates all tasks into 14 parts and handled by 14 different threads distributively, it will achieve is maximum speed up and hence effective.

- **#pragma omp parallel & #pragma omp single & #pragma omp task**
  These three directives are the ones that I used for Q2. It is effective since this directive will generate a task for a thread in the team to handle all the recursion. Q2 already shows the effectiveness of these three directives. The problem with these directives is that the workload might be distributed unevenly and cannot achieve parallelization fully. So these directives cannot achieve maximum speed up, but they are still very effective.

I tried these directives and they were not effective:

- **#pragma omp single**
  The single directive won't parallelize any thing without the parallel directive. It will only spawn one threads to handle all the tasks. So it is basically a sequential version if single is used alone, hence not effective

- **#pragma omp ordered**
  With this directive, OpenMP will ensure that the ordered directives are executed the same way the sequential loop would (one at a time). Since I don't have any shared variables and the order of how the for-loop executes doesn't effect the outcome of the algorithm, with or without this directive makes no different, hence not effective.

- **#pragma omp parallel sections**
  Although this directive will parallel the section it selected, the problem with the parallel sections is that it can't select a region in a for-loop as it will return an error. The code we want to parallelize is inside a for-loop, so this directive is not suitable, hence not effective.

- **#pragma omp master**
  This directive is similar to the single directive, except that the master thread (and only the master thread) is guaranteed to enter this region. It can't go with the parallel directive so it doesn't really help with the parallelization and hence not effective.

- **#pragma omp barrier**
  This directive is used to wait for all the threads in the team to reach the barrier before continuing. However, the region that I want to parallelize doesn't need a synchronization point to proceed for later execution. With or without this directive makes no different, hence not effective.

With all annotations applied, here are the results (**Note:** Since cracking a token with length 6 takes extremely long times and occupies lot of CPUs, there will only be limit amount of trials.):

	Time (s)
Run 1	5.011
Run 2	5.183
Run 3	5.308
Run 4	5.779
Run 5	5.168
Run 6	5.277
Average	5.288

Table 8: Benchmark results for JWTCracker execution unoptimized (`gMaxSecretLen`=4)

	Time (s)
Run 1	0.4094
Run 2	0.3912
Run 3	0.6622
Run 4	0.3851
Run 5	0.5340
Run 6	0.4441
Average	0.4710

Table 9: Benchmark results for JWTCracker execution with manual OpenMP (`gMaxSecretLen`=4)

	Time (s)
Run 1	89.363
Run 2	92.103
Run 3	90.924
Run 4	92.271
Run 5	91.637
Run 6	92.010
Average	91.385

Table 10: Benchmark results for JWTCracker execution unoptimized (`gMaxSecretLen`=5)

	Time (s)
Run 1	6.159
Run 2	6.738
Run 3	6.661
Run 4	6.768
Run 5	6.830
Run 6	7.289
Average	6.741

Table 11: Benchmark results for JWTCracker execution with manual OpenMP (`gMaxSecretLen`=5)

	Time (s)
Run 1	4292.320
Run 2	4132.029
Average	4212.175

Table 12: Benchmark results for JWTCracker execution unoptimized (`gMaxSecretLen`=6)

	Time (s)
Run 1	314.328
Run 2	318.032
Average	316.18

Table 13: Benchmark results for JWTCracker execution with manual OpenMP (`gMaxSecretLen`=6)

The run time for the JWT cracker is similar to the nqueen algorithm. Since there are n alphabets and each alphabet will have a recursion of n-depth, the run time will again be n^n. My change used #pragma omp parallel for. Since there are 36 characters in the alphabet string, there will be 36 calls to "findPrintCombAndValidate" function in the first layer of the recursion. However, there are only 14 available threads. This means that all the threads will be occupied during the first layer of the recursion, and they will only parallelize the first layer of the recursive calls on the "findPrintCombAndValidate" function. Since there are 14 threads to handle the n - 1 depth of the recursion, the run time becomes $n^n/14$. This means that there should be a theoretical speedup of 14 times. When comparing the results from tables 8, 9, 10, 11, 12 and 13, we can see a trend that the speed up for optimized and unoptimized is about **x12 to x13 time**. This result is close to the theoretical prediction. It doesn't achieve the full parallelization of 14 times, and it might due to overheads on thread creations or high loads on ecetesla0.