Project 3 Report - Ishani Mhatre

MY EXPERIMENTAL RESULTS

Tests using word → flower & ilab machine used → crayon.cs.rutgers.edu
*** table data is in terms of milliseconds (ms)

Test	spell_t2_singleloop.c	spell_t2_fastest.c	spell_t4_singleloop.c	spell_t4_fastest.c
1	0.058528	0.059198	0.030791	0.031476
2	0.060391	0.059301	0.031076	0.032284
3	0.061244	0.059619	0.031570	0.031579
4	0.060066	0.060052	0.031932	0.031511
5	0.060303	0.061713	0.032245	0.032253
6	0.059558	0.061777	0.031744	0.031995
7	0.061628	0.060233	0.032764	0.031576
8	0.062038	0.061810	0.032243	0.031870
9	0.060485	0.059513	0.030736	0.031656
10	0.060576	0.060255	0.031040	0.032707
STATS	**MEAN = 0.0604817 MEDIAN = 0.060438 STANDARD DEVIATION = 0.001011556**	**MEAN = 0.0603471 MEDIAN = 0.0601425 STANDARD DEVIATION = 0.0010434753**	**MEAN = 0.0316141 MEDIAN = 0.031657 STANDARD DEVIATION = 0.00069148752**	**MEAN = 0.0318907 MEDIAN = 0.031763 STANDARD DEVIATION = 0.00041244853**

DISCUSSION

I ran 10 different test cases across the four files using the same word: “flower.” I chose this word because it was long enough to be able to differentiate any variations in run times and compare the effectiveness of parallelizing different loops. The chart above shows the output run times (in milliseconds) for the four files as well as the statistics for mean, median, and standard deviation (variance). For all four files, the standard deviation was less than 0.002, which means the majority of the data points came close to the mean. Therefore, the standard deviation was low and not significant.

It is important to analyze differences in runtimes of my solutions with the given solutions. Overall, my solution times when I ran the four files compared to the given solution times were just as efficient, some being slightly faster. When running spell_t2_singleloop.c, the mean was approximately 0.0605 milliseconds (ms), which was less than the solution’s (spell_t2_singleloop-sol) mean of approximately 0.0630 ms. For spell_t2_fastest.c file, the mean was about 0.0603 ms and the mean for spell_t2_fastest-sol was about 0.0609 ms. The mean for spell_t4_singleloop.c was about 0.0316 ms and the mean for spell_t2_fastest-sol was about 0.0321 ms. Lastly, for spell_t4_fastest.c, the mean was approximately 0.03189 ms and the mean for spell_t2_fastest-sol was about 0.03196 ms. In all four cases, the average run time of my solutions was just as fast, if not faster than the given solutions’ average run time for each test case. By comparing these statistics, I was able to confirm that my solution worked and was acceptable. There were some overhead costs after parallelization. For example, for the spell_t2 files, in a perfect world, the time should be reduced to ½ of the original run time and for spell_t4 implementations, the time should be shortened to ¼ of original run time. However, there were a few milliseconds

(<0.01 ms) additional when we ran the trials. This may be due to a variety of reasons including execution time or wasted computation. Nonetheless, the overhead costs were minimal and the solutions were effective overall.

Methodology → Looking at the first program, spell_t2_singleloop.c, we see that after the code segment which loads the word list and creates the bit vector, there are two nested for loops to fill the bit vector (bv). I decided to express loop-level parallelism on a single loop level to this section of the code by applying the OpenMP #pragma directives. I wrote a line that states **"#pragma omp parallel for schedule(guided) private (j,hash)**" before the two for loops. At first, I tried implementing the single directive to add an order constraint and execute one thread at a time. However, I realized that the more effective and best method would be to use the OpenMP #pragma do/ for directive. This work-sharing construct specifies that immediate following iterations of the loop must be executed in parallel and assumes a parallel region has already been specified. Therefore, it would make the most sense to use this directive. The first element which is "schedule (guided)" describes how iterations of the loop will be divided. Compared to schedule(static) and schedule(dynamic), schedule(guided) was more efficient. I decided to use guided since it is similar to dynamic where iterations are divided into pieces and dynamically assigned to threads. However, it saves space as the block size will decrease each time a block is assigned to a thread. Subsequent blocks will be proportional to "number_of_iterations _remaining / number_of_threads," making the guided schedule option more efficient. The "private (j,hash)" is used to specify the hash list and will be private to each thread. I decided to make these two variables private, rather than shared, so each thread will have its own unique copy and will not interfere with other threads. This way any modifications made to the two variables will not be visible to other threats.

For the second program, spell_t2_fastest.c, we had no restrictions for parallelizing loops and could implement as many as we wanted. Therefore, I decided to add two #pragma omp parallel variations, one statement before the two nested for loops that fills the bit vector and a second statement before the spell checker section of the code which includes a for loop to check if the word is spelled correctly. For the bit vector for loop, the #pragma statement was, "**#pragma omp parallel for private(i,j,hash) shared(wl_size, num_hf, wl, bv_size) schedule(guided)**." I declared three variables in the private parameter to ensure they would not interfere with other threats. There were three variables added in a shared context, meaning the variables associated with word length size, size of hash function, and bit vector size will be visible to all threads across their parallel regions. I found by adding the three shared variables, the program was faster since the run time of the for loop heavily relies on these variables. I decided to keep the schedule the same and use "schedule(guided)" which was effective earlier. The second #pragma statement was "**#pragma omp parallel for private(j,hash) shared(bv_size, word) schedule(guided)**" which implemented a similar methodology. There were two private variables declared. Since the runtime in the spell checker for loop relies primarily on the word input and bit vector size which references the length of the word, I decided to add the shared variables "bv_size" and "word." Through trial and error by adjusting the private and shared variables as well as the schedule types, I decided on these specific OpenMP directives and #pragma omp parallel variations described above.

Afterwards, for the remaining two programs, I used similar implementations as spell_t2 since the same methods could be applied. For the third program, spell_t4_singleloop.c, I used a similar statement in order to parallelize the nested for loop that fills the bit vector: "**#pragma omp parallel for private(j, hash) schedule(guided)**." I confirmed the same OpenMP directive worked after running the code. For the last file, "spell_t4_fastest.c" I added the two following statements: "**#pragma omp parallel for shared(wl_size, num_hf, wl, bv_size) private(i,j,hash) schedule(guided,500)**" before the bit vector nested for loop and "**#pragma omp parallel for shared(bv_size, word) private(j,hash) schedule(guided)**" before the spell checker for loop. It was the same as I did for the spell_t2_fastest.c. For the bit vector nested for loop, I initially tried to parallelize by using "schedule (static, 100)" using a chunk size of 100 each time. However, I realized this could give me errors for some test cases and run slower, so I changed it to "schedule (guided, 500)" with a smallest chunk size of 500. Bounded by the smallest chunk size, this can reduce potential overlap. This was my thought process as I was implementing my solutions.