Greg Matthews
ECEC 622 Section 01
January 31st, 2017

# Histogram Generation Using OpenMP

In this week's experiment, the objective was to implement a way to parallelize a Histogram generating function by working with the OpenMP API in C to design a parallel function that has better CPU runtime than the original serial version. The function will take an $int*$ pointer to an input data array, an $int*$ pointer to the histogram array, the given number of elements that is supplied by the user from the terminal, and also the predefined histogram size, which is 500 bins.

In the general scope of things, what will need to be done to generate the Histogram array is to initialize the array with 0's to get it ready to be populated by the input data, and then run through a loop and iteratively populate the values into the histogram. Parallelization of these functions is immediately noticeable given the for loop, however, there's a lot of room for error when populating the Histogram when given multiple threads running simultaneously trying to populate the histogram. This is why a good approach is to create a local Histogram array that keeps the scope private.

```
// Write the function to compute the histogram using openmp
void compute_using_openmp(int *input_data, int *histogram, int num_elements, int histogram_size)
{
#pragma omp parallel num_threads(NUM_THREADS)
{
    int i;
    int hist_temp[NUM_THREADS+1][HISTOGRAM_SIZE]; // Temporary local histogram array
    const int current_thread = omp_get_thread_num();

    // Initializing local histogram array, each thread gets its own row on array
    for (i = 0; i < histogram_size; i++){
        histogram[i] = 0;
        hist_temp[current_thread][i] = 0;}

    #pragma omp barrier

    // Populating the local histogram array, stored seperately (different row) by each thread
    #pragma omp for nowait
    for (i = 0; i < num_elements; i++) hist_temp[current_thread][input_data[i]]++;

    // Adding in the populated values of local histogram array into original histogram
    #pragma omp critical
    for (i = 0; i < histogram_size; i++) histogram[i] += hist_temp[current_thread][i];
}
}
```

*Figure 1. Parallelized Histogram Generation Function*

Pictured in Fig 1, a new variable called $hist\_temp$ was made that will serve as a local array for the function to keep the scope private when populating the Histogram in parallel. If one were to populate the input data from the for loop right into the Histogram, the threads would overpopulate the Histogram because each of them would iterate through the for loop $num\_element$ times. Another

approach could've been splitting the for loop into $\frac{num_{elements}}{num_{threads}}$ iterations, and making sure to populate the Histogram in a location that's evenly distributed between the number of threads. In my case however, I believed keeping a local histogram array would be much simpler and more robust.

The first step was to parallelize the entire function as every step was designed to use the full benefit of parallelizability. The first iterative loop was used to initialize the original and local histogram arrays to get them ready to be populated, but first the local histogram will get populated by each individual code in parallel. The second for loop houses a $\#pragma\ omp\ for\ nowait$, the $nowait$ is preferred because of the fact that the variables and the populating steps are independent from one another and don't follow a specific sequence, therefore the implied barrier given from the for directive can be dropped by using $nowait$. Now that the local histogram has been populated, the original histogram can be generated by adding in each of the local array values that are stored in the local histogram array, however this step has to be done serially by calling $\#pragma\ omp\ critical$. This is necessary to remove the possibility of multiple threads trying to add their inputted data to the same index of the histogram array at the same time. This can cause incorrect results, loss of data, and values being overwritten when not taken care of accordingly. The $critical$ directive ensures the protection of the for loop by only allowing the execution to be ran one thread at a time. The $barrier$ statement makes sure that the following $\#pragma\ omp\ for\ nowait$ statement is run once all the threads are done initializing the histogram arrays. If a thread starts adding the input data into the histogram before the histogram is initialized, incorrect indexing problems can occur. The $barrier$ statement will ensure that all the threads are done their process before moving on.

From the results in Table 1, it is apparent that the more threads that are available, the more efficient the parallelization will be. With 2 threads, parallelization doesn't see much performance increase, with only about a 1.2-1.8x speedup. By increasing the number of the threads, there are more computations that can be simultaneously made, and not surprisingly, the better the performance of the parallelized system. Looking at 16 threads, the speedup is now at around 4.5x, a big improvement.

Likewise, the higher the number of elements, the better the CPU performance is. When comparing 16 threads working on 1,000,000 elements, the speedup is only at 2.2x, however when dealing with 100,000,000 elements, the speedup is much greater at 4.5x. The higher number of elements results in there more work available to be distributed among threads, which in turn increased the efficiency of parallelization, and the CPU performance as well.

```
[gm453@xunil-05 Assignment2]$ ./histogram 100000000
Creating the reference histogram.
CPU run time = 0.1050 s.
Creating histogram using OpenMP.
OpenMP CPU run time = 0.0238 s.
Number of histogram entries = 100000000.
Histogram generated successfully.
Difference between the reference and OpenMP results: 0.000000.
```

*Figure 2. Ideal case: High # of threads and elements equates to highest parallelization efficiency*

Table 1: Histogram Parallelization Experimental Results

| Element Size | Number of Threads | Serial Time (s) | Parallel Time (s) | Speedup |
|---|---|---|---|---|
| 1,000,000 | 2 | 1.6 | 1.3 | 1.231 |
| 1,000,000 | 4 | 1.6 | 1 | 1.600 |
| 1,000,000 | 8 | 1.6 | 0.6 | 2.667 |
| 1,000,000 | 16 | 1.6 | 0.7 | 2.286 |
| 10,000,000 | 2 | 13 | 9.2 | 1.413 |
| 10,000,000 | 4 | 13 | 5.2 | 2.500 |
| 10,000,000 | 8 | 12.7 | 3.5 | 3.629 |
| 10,000,000 | 16 | 12.8 | 2.8 | 4.571 |
| 100,000,000 | 2 | 105.7 | 72.6 | 1.456 |
| 100,000,000 | 4 | 104.9 | 56.3 | 1.863 |
| 100,000,000 | 8 | 105 | 31.7 | 3.312 |
| 100,000,000 | 16 | 107.2 | 23.8 | 4.504 |