DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: OpenMP Assignment 2: Histogram Generation

NAME: Mark Klobukov

INSTRUCTOR: Vasil Pano

DATE SUBMITTED: 2/4/2017

DATE DUE: 2/4/2017

Assignment description:

Given the skeleton of the histogram generation program (which included a serial version of the calculations), we were asked to write a parallel function to create the histogram. OpenMP was the only parallelization technology used in this assignment.

Approach to parallelization:

The key line in the serial program that determined my approach to parallelization is shown below:

```
// Bin the elements in the input stream
for(i = 0; i < num_elements; i++)
        histogram[input_data[i]]++;
```

Just copying and pasting the same serial function into the parallel code and then adding **#pragma** statements wouldn't work. Multiple threads can be accessing the same element of the histogram. Parallelization on this for loop is done on the loop counter **i** but the histogram elements are incremented using the index input_data[i]. Since we are not guaranteed that the elements in the input_data array are unique, there is a possibility of race condition occurring on that line. An "easy fix" could be to add a **single** keyword to the **#pragma** statement, but that would just do the same thing as running the program serially. For this reason, the parallel computation function had to be significantly different from the serial one.

Explanation computation parallelization:

Instead of making all threads access the same **histogram** array, I created a matrix with dimensions num_threads x histogram_size. Each thread would operate on its assigned row of the matrix and create its own version of the histogram for only a part of the input data. Then, when all threads are done, the matrix results would be summed up vertically and added to the corresponding buckets of the **histogram** array (result).

```
// Write the function to compute the histogram using openmp
void compute_using_openmp(int *input_data, int *histogram, int
num_elements, int histogram_size, int num_threads)
{
    int i, j;
    //Matrix to hold different threads' results
    int this_histogram[num_threads][histogram_size];
```

The code snipped above shows the parallel function declaration and declaration of the aforementioned matrix.

Both the histogram array to hold the result and the partial result matrix needed to be initialized. When an integer array/matrix is declared in C, its elements are not 0 by default. The following picture shows how it was done. This operation could be parallelized because there are no dependencies between iterations of the loop.

```
// Initialize histogram
#pragma omp parallel for num_threads(num_threads) private(i, j)
      for(i = 0; i < histogram_size; i++) {
         histogram[i] = 0;
            for (j = 0; j < num_threads; j++) {
                //Initialize matrix to hold partial results
                this_histogram[j][i] = 0;
            } // for j
        } //for i
```

Then I wrote a **#pragma** statement and surrounded the entire parallel region of computation in {} in order to maintain thread number consistency throughout the parallel region and simplify development.

The following screenshot shows how each thread obtained its own number and only accessed the row of the matrix whose index corresponds to the thread number. This guaranteed that there would be no race conditions in the code.

```
//BEGIN PARALLEL REGION
#pragma omp parallel num_threads(num_threads)
{

    int my_number = omp_get_thread_num();
# pragma omp for
    for (i = 0; i < num_elements; i++) {
        this_histogram[my_number][input_data[i]]++;
    } // end for i
```

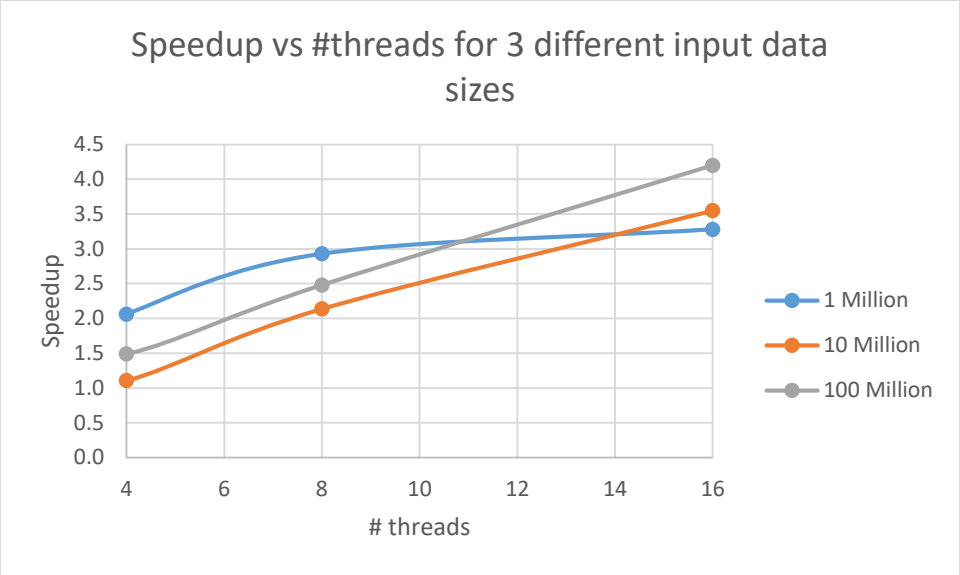
Once all individual threads' results were completed, the matrix was ready to get summed and update the **histogram** array that would hold the final results. No **critical** statement was necessary because no race conditions would arise due to separation of thread calculations into different rows of the matrix.

```
    for (i = 0; i < histogram_size; i++) {
        for(j = 0; j < num_threads; j++) {
            histogram[i] += this_histogram[j][i];
        }// for j
    } // for i
} //end compute_using_openmp
```

Results:

The following table shows the speedups attained by varying the number of elements and the number of threads.

Input data size	# threads	Time serial (s)	Time Parallel (s)	Speedup
1M	2	0.00566	0.00428	1.322
	4	0.00812	0.00394	2.061
	8	0.007	0.00239	2.929
	16	0.00751	0.00229	3.279
10M	2	0.04579	0.04346	1.054
	4	0.03975	0.03589	1.108
	8	0.0368	0.01722	2.137
	16	0.03876	0.01092	3.549
100M	2	0.3666	0.23778	1.542
	4	0.38542	0.25876	1.489
	8	0.38976	0.1571	2.481
	16	0.39013	0.0929	4.199



For each input data size, there is an upward trend in speedup as the number of threads increase, according to the above table and graph. This demonstrates Gustafson's law: the problem size scales with the number of available cores/threads, and so the speedup also increases (with both # threads and problem size).

Conclusion:

Parallelization was performed successfully. The speedup of up to about 4.2 was attained, and the trend characterized by the Gustafson's law was observed.

The program is to be compiled as follows:

gcc -o histogram histogram.c -std=c99 -fopenmp -lm

To run the program: **./histogram #elements**

where **#elements** is an integer.