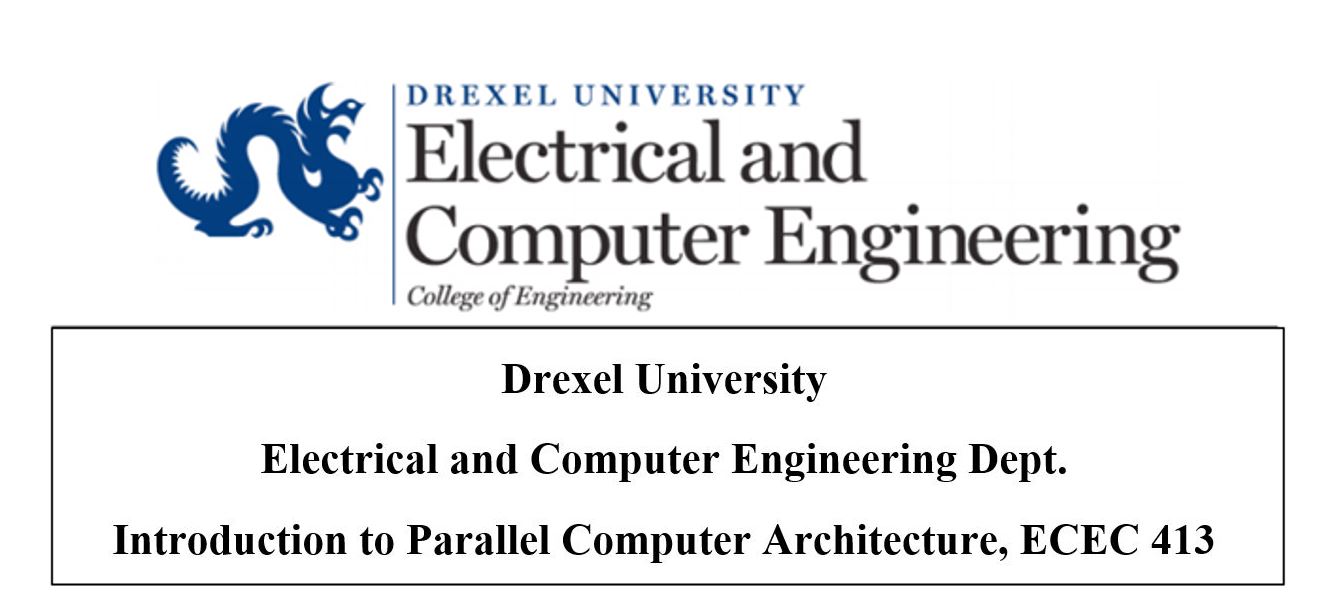
****

**Title:** Histogram Generation using OpenMP

**Names:** Avik Bag, Shannon Miles

**TA:**  Vasil Pano

**Section:** 001

**Date Performed:** 2/8/17

**Date Due:** 2/4/17

**Date Received:** 1/27/17

**Introduction:**

The object of this assignment is to take a serial implementation of the histogram generation algorithm located in the compute\_gold function and develop a parallel formulation of it in the compute\_using\_openmp function. The parallelized code will be compared to the serial implementation and speed up will be reported when using 2, 4, 8, and 16 threads for one million, ten million, and hundred million elements. (Please see README on how to run the program.)

**Graphs and Tables:**

Table : Execution times along with speedups for the various elements and threads used



Table 2: Execution times along with speedups for the various elements and threads used

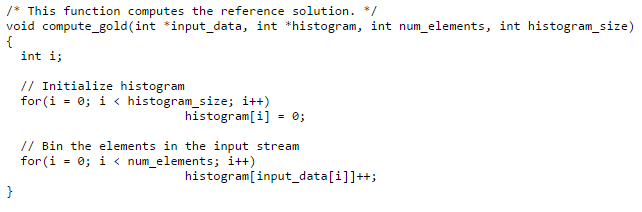


Figure 1: Speedup as seen on personal computer

Figure 2: Speedup as seen on xunil

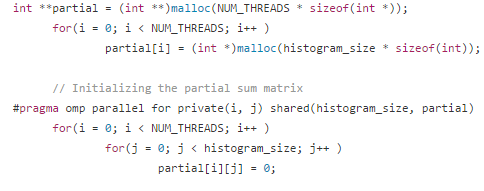
**Discussion**:

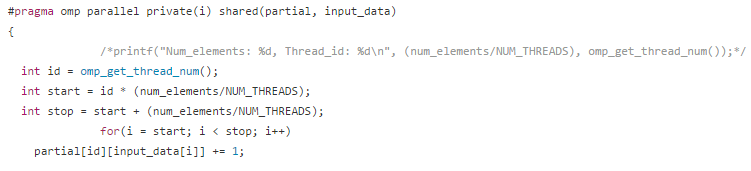
The code below is the serial version of the histogram generation algorithm:



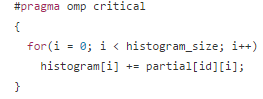
To parallelize the implementation above, the openMP library was used. First the histogram was initialized in parallel with the variable i being private and the variable histogram\_size and array histogram being shared using #pragma omp parallel for default(none) private(i) shared(histogram, histogram\_size)

Next, as seen in figure 3 a partial sum matrix is needed to avoid a race condition. This is done by dynamically allocating memory for this matrix and then for the index i of the matrix the thread id is stored. The second index j refers to the bins. This matrix will allow input\_data to be split to the correct size based on the number of threads chosen. Once input\_data is split, the partial sum is calculated for each chunk of data as seen in figure 4.

  
Figure 3: partial sum setup

  
*Figure 4: partial sum setup*

After the calculation is completed for each thread, the data is merged back into the histogram pointer as seen in figure 5, using #pragma critical to make sure threads do not interfere with each other.

  
*Figure 5: section to be critical to avoid thread collision*

On parallelizing the implementation, speed ups were seen as the number of threads increased. For 2 or 4 threads, there was no speed up; however, when using 8 or 16 threads speed up could be seen with the one million, ten million, and hundred million elements used.

One point to note is that there were differences in the data collected from xunil versus the data collected from my personal machine. This could be because of differences in cache size and even how the operating system organizes the memory spaces. In the xunil dataset, we see that there isn’t a proper increase in speed up, rather it slowed down, until there was a large enough dataset and corresponding number of threads. This could very well be because of the overhead incurred in malloc of the double int pointer matrix and then merging it back. There is also a critical block, which is needed to ensure that no other thread would interfere when the merge process is happening on one thread. The use of the critical, however, affects the parallelization of the implementation.

**Conclusion:**

To conclude, parallelizing processes is beneficial only if speed up can be achieved. For our program that speed up occurred only as the number of threads increased. The size of the data set and the number of threads used will affect the speed up. The overhead needs to be outweighed by the parallelization process for any benefit.