Drexel University

Electrical and Computer Engineering Dept.

Introduction to Parallel Computer Architecture, ECEC 413

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

OutputXunil							
Reference Histogram				**Histogram using OpenMP**			
Elements	Threads	CPU Time		Elements	Threads	CPU Time	Speedup
1000000	2	0.004061		1000000	2	0.008851	0.45881821
	4	0.003681			4	0.005084	0.72403619
	8	0.00363			8	0.004371	0.83047358
	16	0.003869			16	0.003976	0.97308853
10000000	2	0.046894		10000000	2	0.072197	0.64952837
	4	0.039164			4	0.047678	0.82142707
	8	0.038279			8	0.021382	1.79024413
	16	0.037583			16	0.016542	2.27197437
1000000000	2	0.38351		100000000	2	0.449509	0.85317535
	4	0.385386			4	0.316371	1.21814578
	8	0.380931			8	0.196837	1.93526116
	16	0.381075			16	0.108969	3.4970955

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

OutputPersonal							
Reference Histogram				**Histogram using OpenMP**			
Elements	Threads	CPU Time		Elements	Threads	CPU Time	Speedup
1000000	2	0.002339		1000000	2	0.001929	1.2125454
	4	0.002569			4	0.001239	2.0734463
	8	0.002185			8	0.001588	1.3759446
	16	0.002431			16	0.001819	1.3364486
10000000	2	0.024853		10000000	2	0.017618	1.4106596
	4	0.023109			4	0.009113	2.5358279
	8	0.023688			8	0.008515	2.7819143
	16	0.025058			16	0.008927	2.80699
1000000000	2	0.254364		100000000	2	0.173886	1.4628205
	4	0.240305			4	0.103195	2.3286496
	8	0.238031			8	0.092845	2.563746
	16	0.237225			16	0.105628	2.2458534

Disclaimer: The test cases are sun on both xunil and my personal machine for comparison purposes. The discussion section talks about the results achieved.

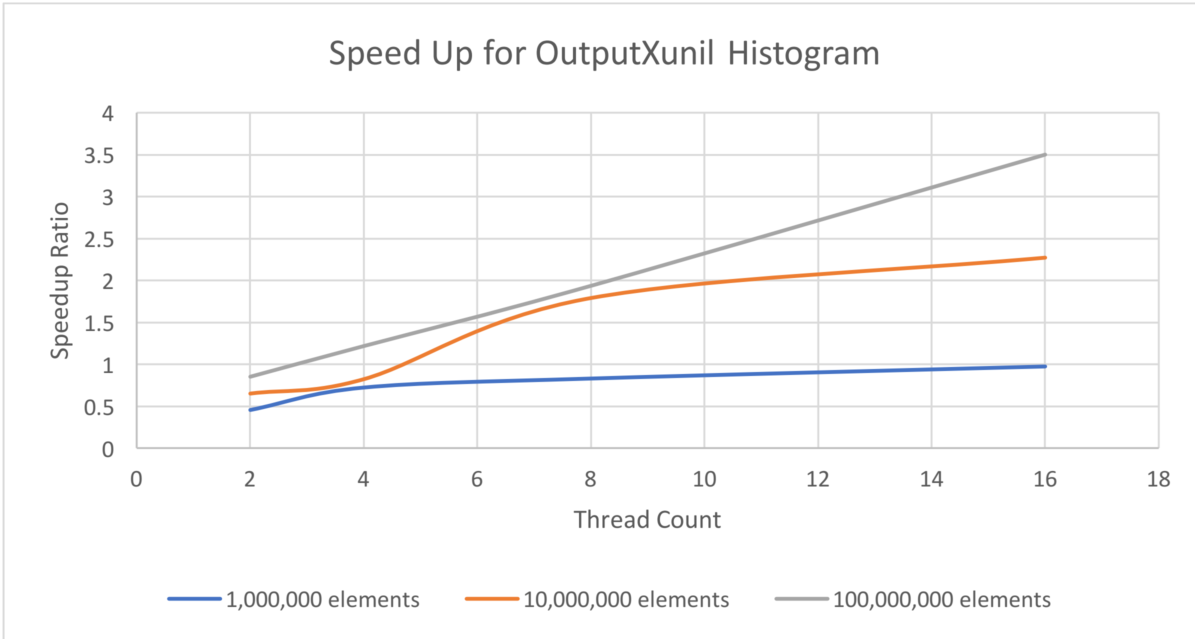

Figure 1: Speedup as seen on personal computer

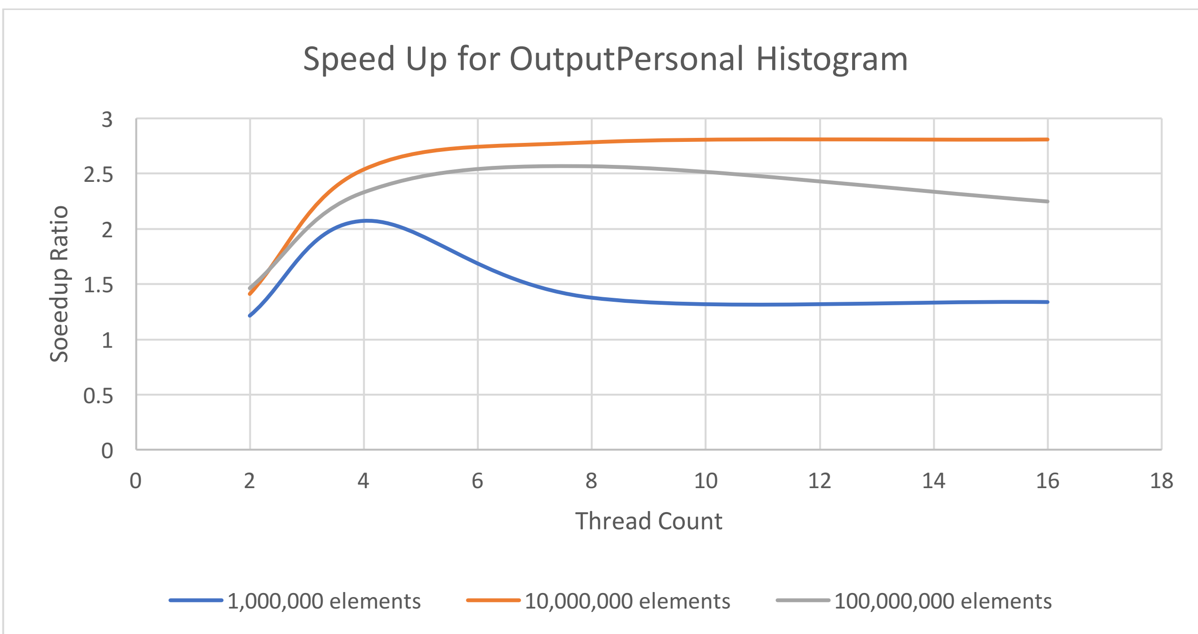

Figure 2: Speedup as seen on xunil

Discussion:

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

```
/* This function computes the reference solution. */
void compute_gold(int *input_data, int *histogram, int num_elements, int histogram_size)
{
  int i;

  // Initialize histogram
  for(i = 0; i < histogram_size; i++)
                        histogram[i] = 0;

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

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.

```
int **partial = (int **)malloc(NUM_THREADS * sizeof(int *));
      for(i = 0; i < NUM_THREADS; i++ )
              partial[i] = (int *)malloc(histogram_size * sizeof(int));

      // Initializing the partial sum matrix
#pragma omp parallel for private(i, j) shared(histogram_size, partial)
      for(i = 0; i < NUM_THREADS; i++ )
              for(j = 0; j < histogram_size; j++ )
                      partial[i][j] = 0;
```

Figure 3: partial sum setup

```c
#pragma omp parallel private(i) shared(partial, input_data)
{
		/*printf("Num_elements: %d, Thread_id: %d\n", (num_elements/NUM_THREADS), omp_get_thread_num());*/
  int id = omp_get_thread_num();
  int start = id * (num_elements/NUM_THREADS);
  int stop = start + (num_elements/NUM_THREADS);
		for(i = start; i < stop; i++)
    partial[id][input_data[i]] += 1;
```

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.

```c
#pragma omp critical
{
  for(i = 0; i < histogram_size; i++)
    histogram[i] += partial[id][i];
}
```

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 when the data size isn't large enough and the number of threads are not enough, rather it slowed down, until there was a large enough dataset and corresponding number of threads for there to be a beneficial speed up. This could very well be because of the overhead incurred in malloc of the double int pointer matrix, initializing it 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.