Drexel University

Electrical and Computer Engineering Dept.

Introduction to Parallel Computer Architecture, ECEC 413

Title: Gaussian Elimination using OpenMP

Names: Avik Bag, Shannon Miles

TA: Vasil Pano

Section: 001

Date Performed: 1/20/17

Date Due: 1/29/17

Date Received: 1/29/17

Introduction:

The object of this assignment is to take a serial implementation of the Gaussian elimination algorithm and develop a parallel formulation of the gauss_eliminate C-file using OpenMP. This will be accomplished by modifying the gauss_eliminate_using_openmp() function. For matrix sizes of 1024 × 1024, 2048 × 2048, 4096 × 4096, and 8192 × 8192, the parallelized code will be compared to the serial implementation and the speed up will be reported when using 2, 4, 8, and 16 threads. (Please see README on how to run program.)

Graphs and Tables:

Table 1: Execution times

Threads	Matrix Sizes			
	1024	2048	4096	8192
1	0.32s	2.57s	20.51s	179.69s
2	0.21s	1.44s	10.71s	96.86s
4	0.17s	1.11s	8.22s	71.66s
8	0.25s	1.36s	8.26s	70.89s
16	0.49s	1.76s	9.1s	71.08s

Table 2: Speed up calculation in reference to serial implementation

Threads	Matrix Size			
	1024	2048	4096	8192
1	1	1	1	1
2	1.524	1.785	1.915	1.855
4	1.882	2.315	2.495	2.508
8	1.280	1.890	2.483	2.535
16	0.653	1.460	2.254	2.528

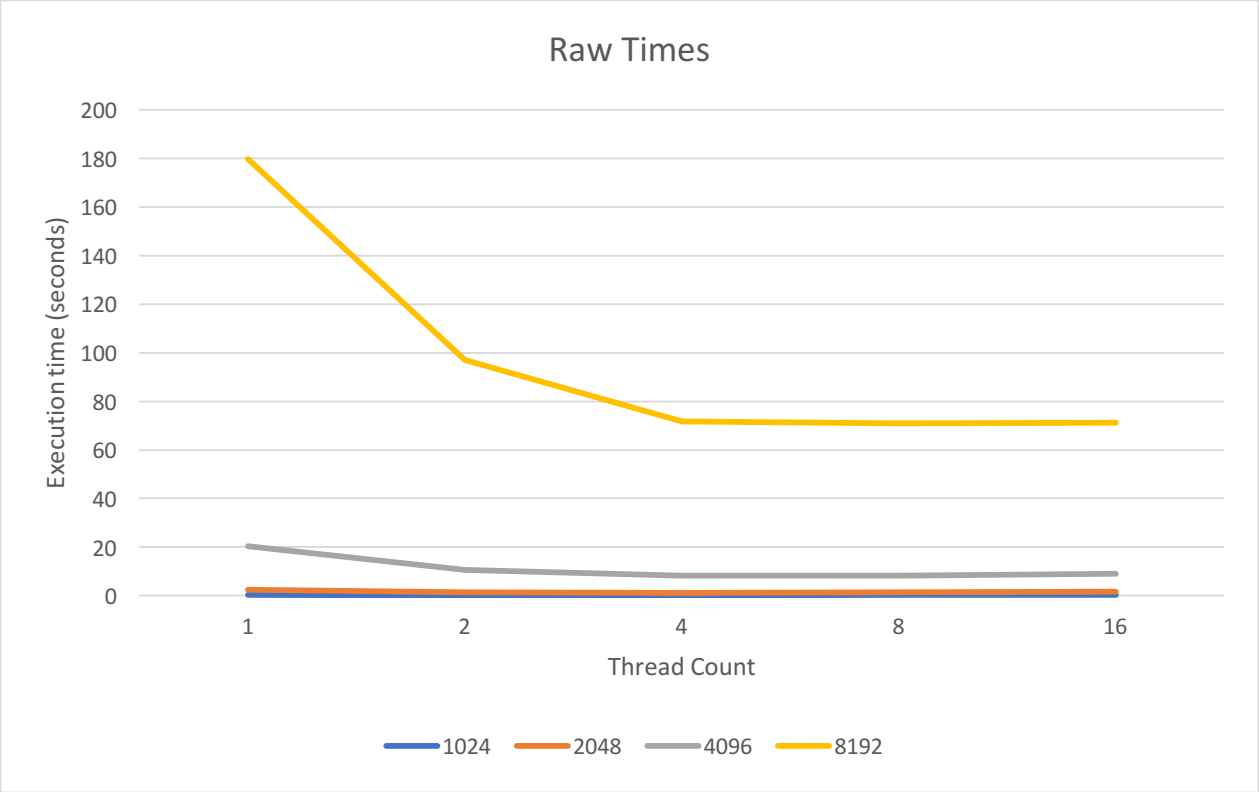

Figure 1: Execution time vs thread count for each matrix size

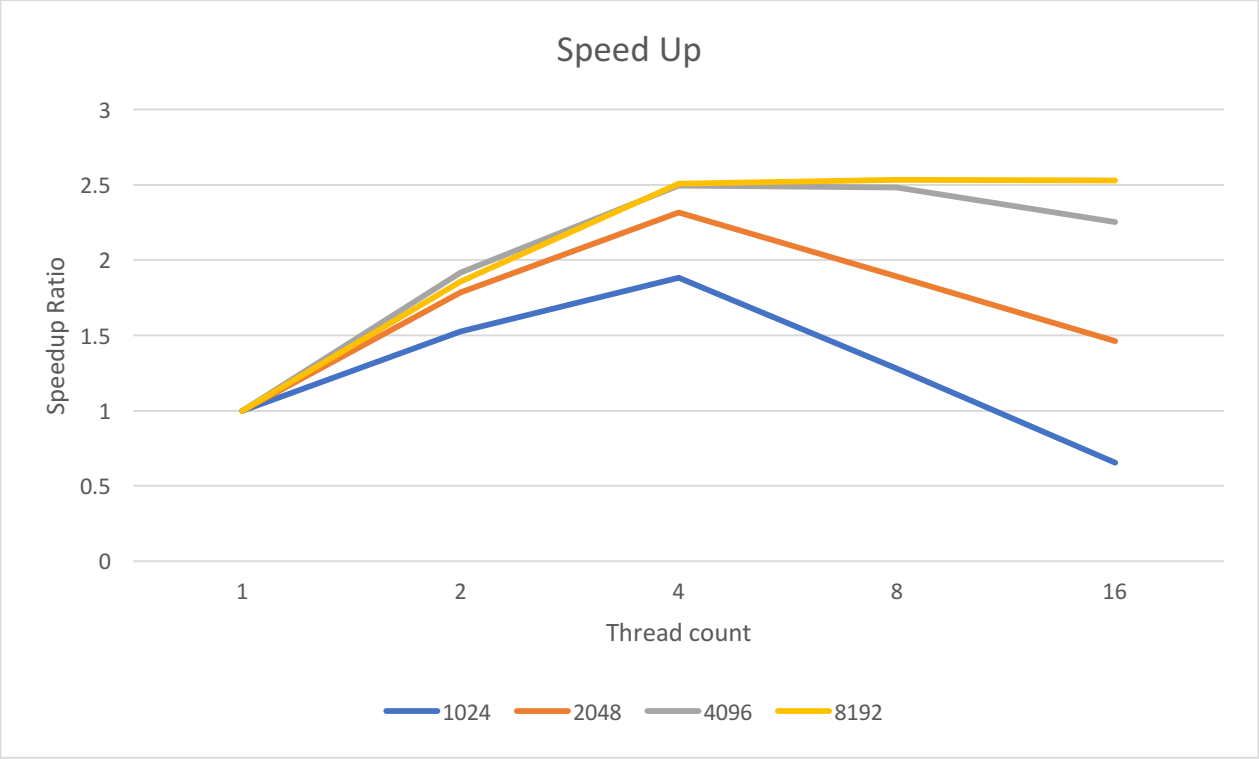

Figure 2: Speed up plot

Discussion:

The Gaussian elimination code to be parallelized is shown below:

```
1:  procedure GAUSS_ELIMINATE(A, b, y)
2:  int i, j, k;
3:  for k := 0 to n − 1 do
4:     for j := k + 1 to n − 1 do
5:        A[k, j] := A[k, j]/A[k, k];      /* Division step. */
6:     end for
7:     y[k] := b[k]/A[k, k];
8:     A[k, k] := 1;
9:     for i := k + 1 to n − 1 do
10:       for j := k + 1 to n − 1 do
11:          A[i, j] := A[i, j] - A[i, k] × A[k, j];      /* Elimination step. */
12:       end for
13:       b[i] := b[i] − A[i, k] × y[k];
14:       A[i, k] := 0;
15:    end for
16: end for
```

To parallelize the implementation above, the openMP library was used. The for-loops were essentially created in the multi thread implementation as will be shown in the source code. The idea was that for nested for-loops, the inner for-loop would spawn a new thread process. For every parallelization, a set of variables were chosen to be private, which meant that these variable would not communicate with the variables of the same name with other threads, in other words, isolating them. There were variables that were however shared. This depended on the implementation of the parallelization. On parallelizing the implementation, there were some severe speed ups.

One thing to note though is that there is a limit as to how many threads can be assigned to the process. Adding too many can actually cause the execution to slow down. There also seems to be a correlation between the size of the data versus the amount of threads used. If there are too many threads for a small dataset, it could actually make execution slower than the serial version. This can be observed in the data gathered and visualized in figure 2.

Conclusion:

To conclude, parallelizing processes is beneficial, but there is a limit to the amount of parallelization that can be carried out. The size of the dataset depends greatly on the number of threads that are needed to maximize the performance. There is a fine balance between size of data set and the number of threads used.