DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: OpenMP Assignment 1: Gaussian Elimination

NAME: Mark Klobukov

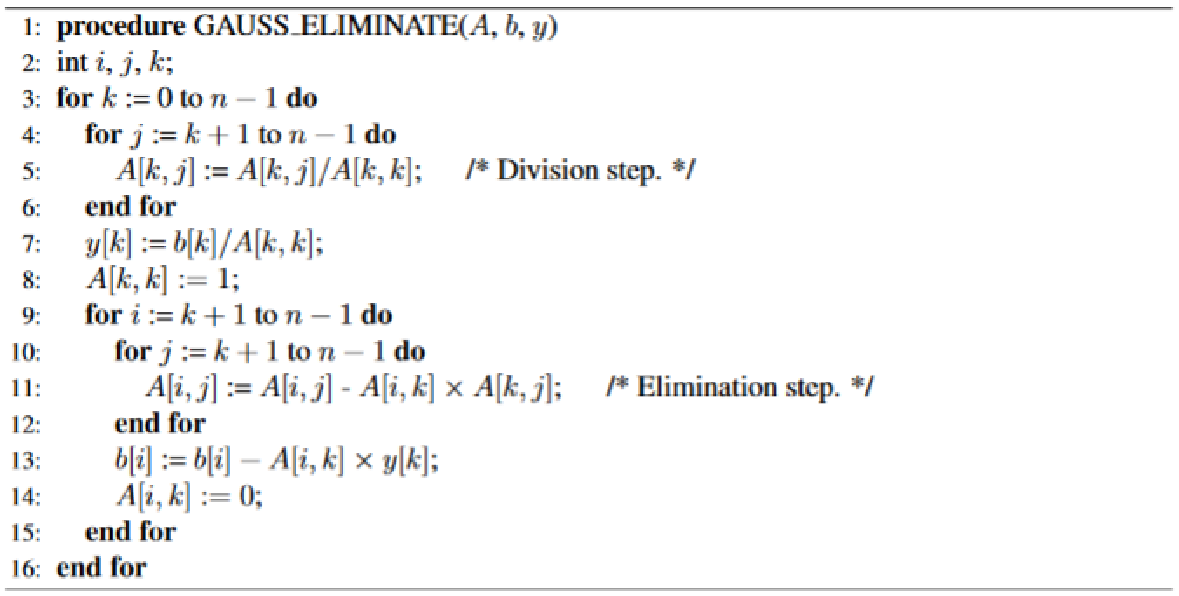
INSTRUCTOR: Vasil Pano

DATE SUBMITTED: 1/28/2017

DATE DUE: 1/29/2017

Assignment description:

The first OpenMP assignment involved parallelizing a given program that performs Gaussian elimination of an NxN system of linear equations. The algorithm is shown in the pseudo-code 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:          $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$;     /* Elimination step. */
12:       end for
13:       $b[i] := b[i] - A[i, k] \times y[k]$;
14:       $A[i, k] := 0$;
15:    end for
16: end for
```

Parallelization approach:

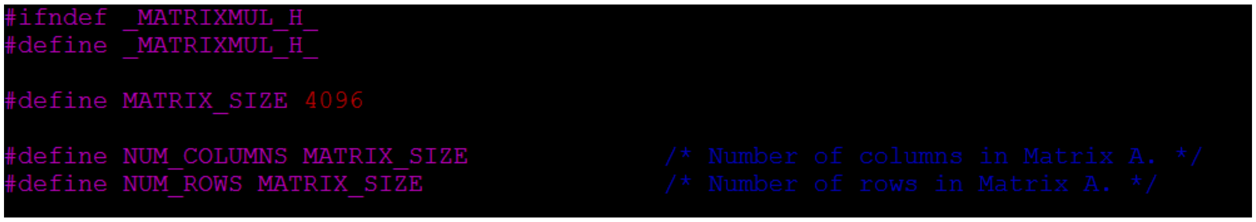
The outermost for-loop (on line 3) cannot be parallelized. The reason for this is that each successive iteration of this loop operates on the values of the matrix A that were updated in the previous iteration. This is a successive procedure.

The for-loops on lines 4 and 11, on the contrary, can be parallelized. The iterations are independent of each other, and are updated with reference to the fixed values of A corresponding to the kth row and kth column.

Explanation of OpenMP additions to the code:

With the assignment, the students received several code files. The file called **compute_gold.c** was not changed in any way, and the serial function for Gaussian elimination written in this file was used for verification of correctness of the parallel version, as well as for the speedup calculation.

The only alteration to the **gauss_eliminate.h** was for the matrix size (third line of code):

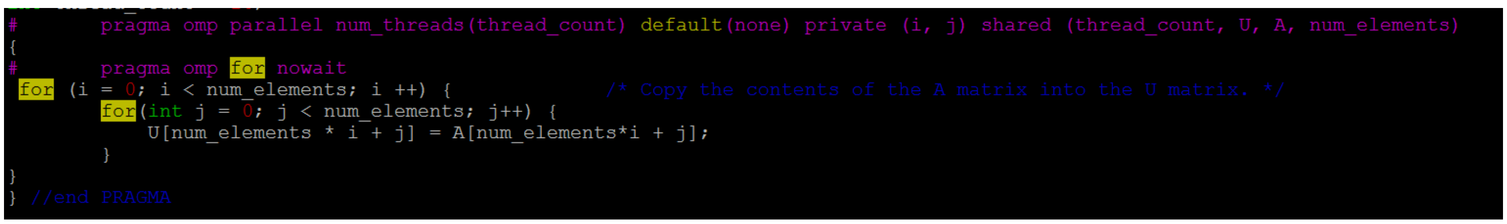


```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_

#define MATRIX_SIZE 4096

#define NUM_COLUMNS MATRIX_SIZE                 /* Number of columns in Matrix A. */
#define NUM_ROWS MATRIX_SIZE                    /* Number of rows in Matrix A. */
```

The work was mainly done in the **gauss_eliminate.c** file. The skeleton of the code left the function called **gauss_eliminate_using_openmp()** blank. I copied the contents of **gauss_eliminate.c** into this function, and then added the necessary **pragma** statements.

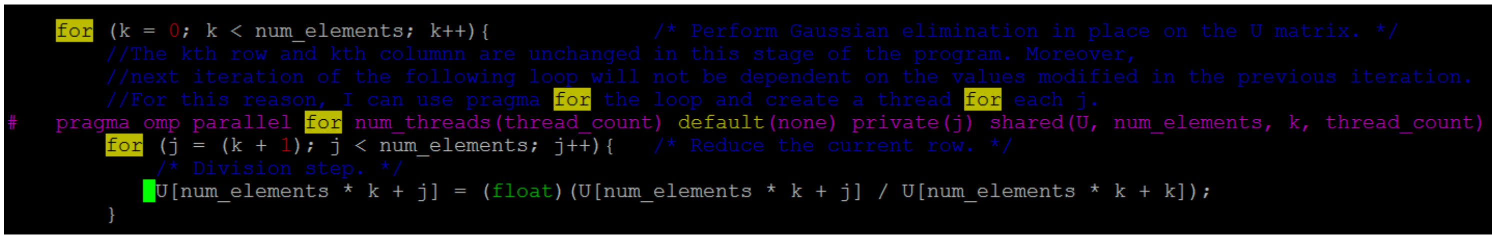
- Parallelization of copying of elements from A to U

First several lines in the function are dedicated to copying all of the elements from matrix A to matrix U. Since the iterations in this for loop are not dependent on one another, it can be parallelized with the **nowait** specification.



```
#       pragma omp parallel num_threads(thread_count) default(none) private (i, j) shared (thread_count, U, A, num_elements)
{
#       pragma omp for nowait
 for (i = 0; i < num_elements; i ++) {             /* Copy the contents of the A matrix into the U matrix. */
        for(int j = 0; j < num_elements; j++) {
            U[num_elements * i + j] = A[num_elements*i + j];
        }
}
} //end PRAGMA
```

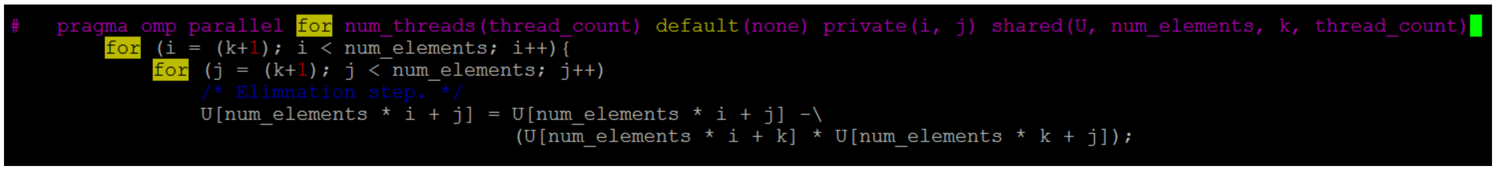
- Parallelization of the division step of the algorithm



```
    for (k = 0; k < num_elements; k++){              /* Perform Gaussian elimination in place on the U matrix. */
        //The kth row and kth columnn are unchanged in this stage of the program. Moreover,
        //next iteration of the following loop will not be dependent on the values modified in the previous iteration.
        //For this reason, I can use pragma for the loop and create a thread for each j.
#   pragma omp parallel for num_threads(thread_count) default(none) private(j) shared(U, num_elements, k, thread_count)
        for (j = (k + 1); j < num_elements; j++){    /* Reduce the current row. */
            /* Division step. */
            U[num_elements * k + j] = (float)(U[num_elements * k + j] / U[num_elements * k + k]);
        }
```

Num_threads directive accepts a previously defined integer variable **thread_count**. The loop counter is declared private to the threads, and the data necessary for all threads are public. Schedule is default **static** so that the assignment of iterations to the threads occurs before runtime.

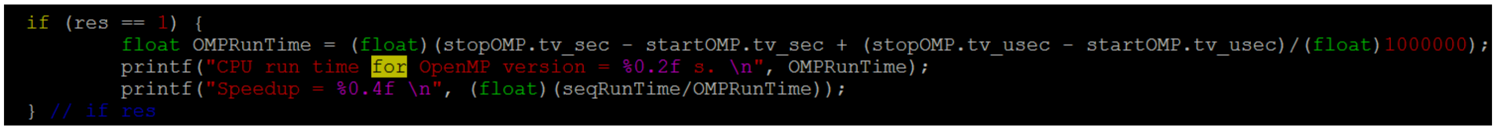
- Parallelization of the elimination step of the algorithm



```
#   pragma omp parallel for num_threads(thread_count) default(none) private(i, j) shared(U, num_elements, k, thread_count)
        for (i = (k+1); i < num_elements; i++){
            for (j = (k+1); j < num_elements; j++)
                /* Elimination step. */
                U[num_elements * i + j] = U[num_elements * i + j] -\
                                          (U[num_elements * i + k] * U[num_elements * k + j]);
```

This pragma statement is almost identical to the one from the division step, except for an extra private variable **j** from the inner loop. By trial and error process, it was again determined that leaving schedule as **static** will produce the greatest speedup. All threads have an exact same amount of work, so it is a good option in this case.

In addition to the **pragma**s, the following if-statement was added to the body of the main() function. It checks whether the parallel computation produced the same result as the parallel one, and only if the two results agree it displays the time measurement of the parallel computation, as well as the speedup:

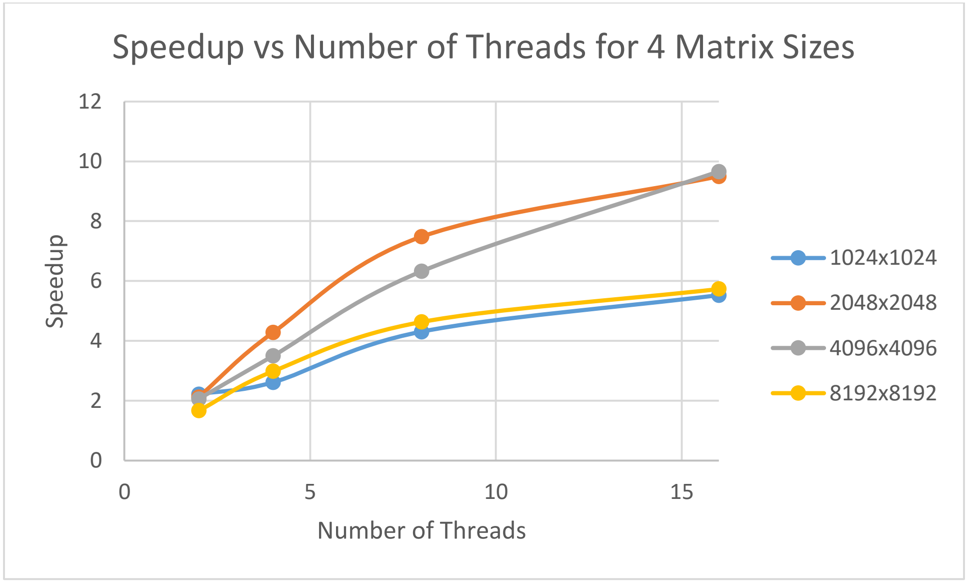


```
  if (res == 1) {
          float OMPRunTime = (float)(stopOMP.tv_sec - startOMP.tv_sec + (stopOMP.tv_usec - startOMP.tv_usec)/(float)1000000);
          printf("CPU run time for OpenMP version = %0.2f s. \n", OMPRunTime);
          printf("Speedup = %0.4f \n", (float)(seqRunTime/OMPRunTime));
  } // if res
```

Results:

The following table shows the serial program time, parallel program time, and the speedup for various matrix sizes and numbers of threads used.

Matrix size	# threads	Time with serial program (sec)	Time with parallel program (sec)	Speedup
1024x1024	2	0.52	0.24	2.2138
	4	0.52	0.2	2.6093
	8	0.54	0.13	4.3037
	16	0.58	0.11	5.5232
2048x2048	2	4.76	2.25	2.1174
	4	4.89	1.15	4.2739
	8	4.76	0.64	7.4794
	16	4.91	0.52	9.4979
4096x4096	2	42.28	20.57	2.0552
	4	51.29	14.68	3.4939
	8	41.6	6.58	6.3223
	16	41.43	4.29	9.6573
8192x8192	2	396.84	238.11	1.6666
	4	435.61	145.91	2.9855
	8	432.06	93.24	4.6339
	16	437.92	76.37	5.7341



In all cases, increase in number of threads caused a program to run faster. Also, speedup increased with the increase in the number of threads. The greatest speedup was achieved for the matrix size 4096x4096 (Speedup = 9.6573) with 16 threads. The maximum speedup for the 2048x2048 matrix was approximately the same.

It appears that the increase in speedup cannot continue indefinitely with increasing matrix size in this case. The program has serial portions too, so the greater the matrix size, the more time the serial portions take up, so the benefits of parallelization are not as dramatic in the case of matrices with the size over 4096x4096. However, the parallelization is still worth it because it decreases the run time of the program by a factor of up to 5 in the case of the largest matrix.

Another possible reason for a lower speedup in a large matrix is that for a larger amount of data (and hence for a longer-running program), there is a higher chance of cache misses. The program in this assignment does not account for the cache memory characteristics of the CPUs. So, I hypothesize that this lack of optimization gets worse with increasing matrix sizes and hinders the growth of parallelization benefits.

Compilation Instructions:

Compile the attached code with the following command.

```
gcc -o gauss_eliminate gauss_eliminate.c compute_gold.c -fopenmp -std=c99 -O3 -lm
```

This compilation command is identical to the one provided in the original gauss_eliminate.c