Greg Matthews
ECEC 622 Section 01
January 24th, 2017

Gaussian Elimination Report:

This week's assignment involved parallelizing a Gaussian Elimination function that takes 2 Matrices as arguments, and essentially converting the linear equation $Ax = b$ into an upper triangle system $Ux = y$, which ultimately puts the equation into row reduced echelon form by making all the diagonal entries equal to 1, and whenever $i > j$, then $U[i,j] = 0$. To accomplish this, the Gaussian elimination function will follow through a division step to divide each row out to make the diagonal entry equal to 1. Then, the elimination step will serve to let $A[i,k] = 0$ to row reduce the system. By running this in serial mode, the CPU run time is seen to take about 0.538 seconds:

```
[gm453@xunil-05 Assignment1]$ ./gauss_eliminate
Performing gaussian elimination using the reference code.
CPU run time = 0.538 s.
Gaussian elimination using the reference code was successful.
Test PASSED
```

Figure 1. Serial Execution of Gaussian Elimination Function

However this system is highly parallelizable as a result of its multiple for loops, allowing this execution to be split up among threads to allow for much faster and more efficient performance. The two loops that showed immediate parallelizable features, is the division's step loop, and the elimination step's for loop. The computation of dividing the diagonal elements of the matrix can be split up between each iteration of the for loop, as well as the elimination step, allowing several threads to execute segments of the structured block. The iterations of the for loop are thereby divided among the threads to allow for this parallelization, with the scope mainly being shared, except for the iterating value j in the loop, as seen in Fig 2. The iterations of the loop where scheduled in a dynamic setting to allow for the threads to

```
int gauss_eliminate_using_openmp(const Matrix A, Matrix U)
{
    unsigned int i, j, k;
    int thread_count = 8;

    // copying contents of A to U
    for (i=0; i < A.num_rows; i++)
        for(j=0; j < A.num_rows; j++)
            U.elements[A.num_rows * i + j] = A.elements[A.num_rows * i + j];
    // Gaussian elimination on U, reduce current row
    for (k=0; k < A.num_rows; k++)
    {
#       pragma omp parallel for num_threads(thread_count) default(none) private(j) shared(k,U,thread_count) schedule(dynamic, A.num_rows/thread_count)
        for (j=(k+1); j < A.num_rows; j++)
        {
            // Division
            U.elements[A.num_rows*k+j]=(float)(U.elements[A.num_rows*k+j]/U.elements[A.num_rows*k+k]);
        }
        U.elements[A.num_rows * k + k] = 1;
#       pragma omp parallel for num_threads(thread_count) default(none) private(i,j) shared(k,U,thread_count) schedule(dynamic, A.num_rows/thread_count)
        for (i=(k+1); i < A.num_rows; i++)
        {
            for (j = (k+1); j< A.num_rows; j++)
            // Elimination
                U.elements[A.num_rows*i+j]=U.elements[A.num_rows*i+j]-(U.elements[A.num_rows*i+k]*U.elements[A.num_rows*k+j]);
            U.elements[A.num_rows*i+k] = 0;
        }
    }
    return 1;
```

Figure 2. Modified Gaussian Elimination function will parallelization added using openMP

get assigned iteration values as the loop is executing, rather than with static which would assign the iterations to the threads before the loop executes. This will allow other threads that may have perhaps finished their iterations faster than the rest of the threads, to be scheduled more iterations instead of a static scheduling system in which the thread will have nothing to do once it finishes all its threads. Dynamic scheduling will allow the threads to be scheduled more iterations dynamically as the for loop executes.

Table 1: Gaussian Elimination with OpenMP

Matrix Size	Number of Threads	Serial Time (s)	Parallel Time (s)	Speedup
1024X1024	2	0.573	0.343	1.671
1024X1024	4	0.538	0.265	2.030
1024X1024	8	0.538	0.181	2.972
1024X1024	16	0.555	0.174	3.190
2048X2048	2	3.994	2.678	1.491
2048X2048	4	3.76	1.662	2.262
2048X2048	8	3.746	0.964	3.886
2048X2048	16	3.771	0.877	4.300
4096X4096	2	32.728	22.549	1.451
4096X4096	4	32.607	22.433	1.454
4096X4096	8	33.009	8.925	3.698
4096X4096	16	32.691	7.665	4.265
8192X8192	2	255.381	179.462	1.423
8192X8192	4	260.875	102.016	2.557
8192X8192	8	258.443	59.451	4.347
8192X8192	16	257.911	53.613	4.811

Testing several different matrix sizes, as well as the number of threads used, the data extracted from the Gaussian Elimination function have been added into Table 1. From the results, it is evident that the bigger the matrix size, generally the more efficient the parallelization using openMP will be. This is expected because as the matrix size increases, so does the necessary number of iterations needed in the Gaussian Elimination function, and this more room for parallelization. This is made more evident by the graph in Fig 4 which shows a linear increase in CPU performance using openMP as the size of the matrix is increased. Likewise, the number of threads is shown to have an increase in the CPU time as a result of there simply being more threads to split up the iterations within the for loop, and as such, this allows for more efficient parallelization of the data. As seen in Fig 4, the CPU's performance can be seen to increase with the additional number of threads used in the function execution.

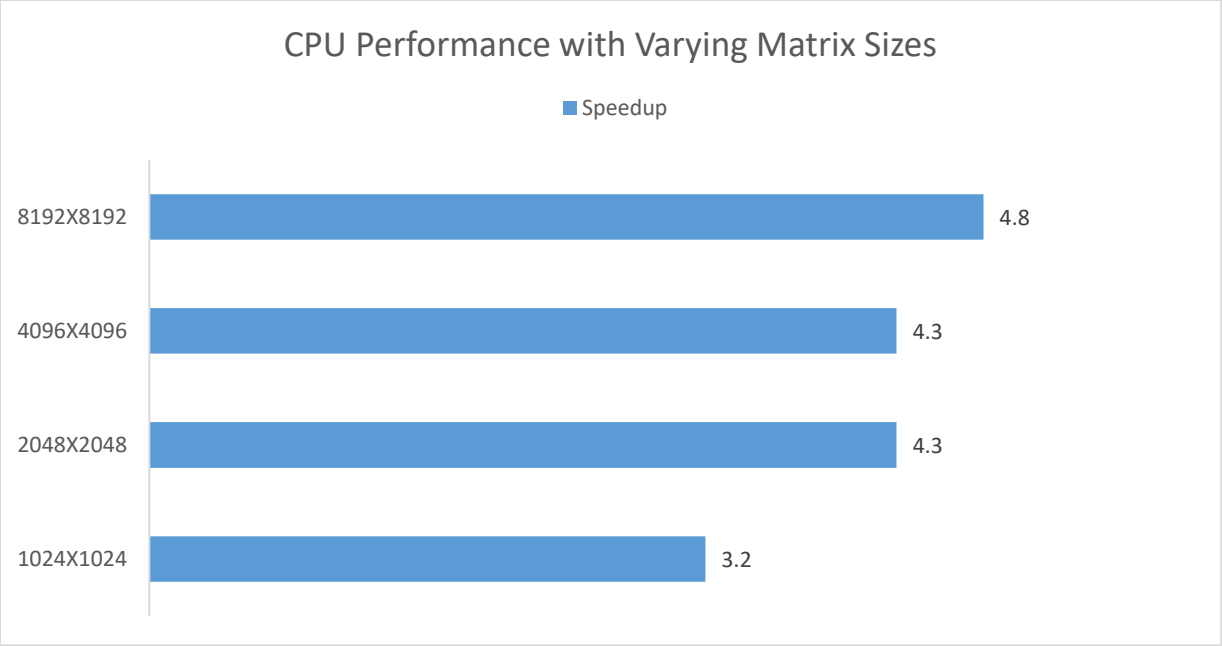

Figure 3. Speedup with varying Matrix Sizes

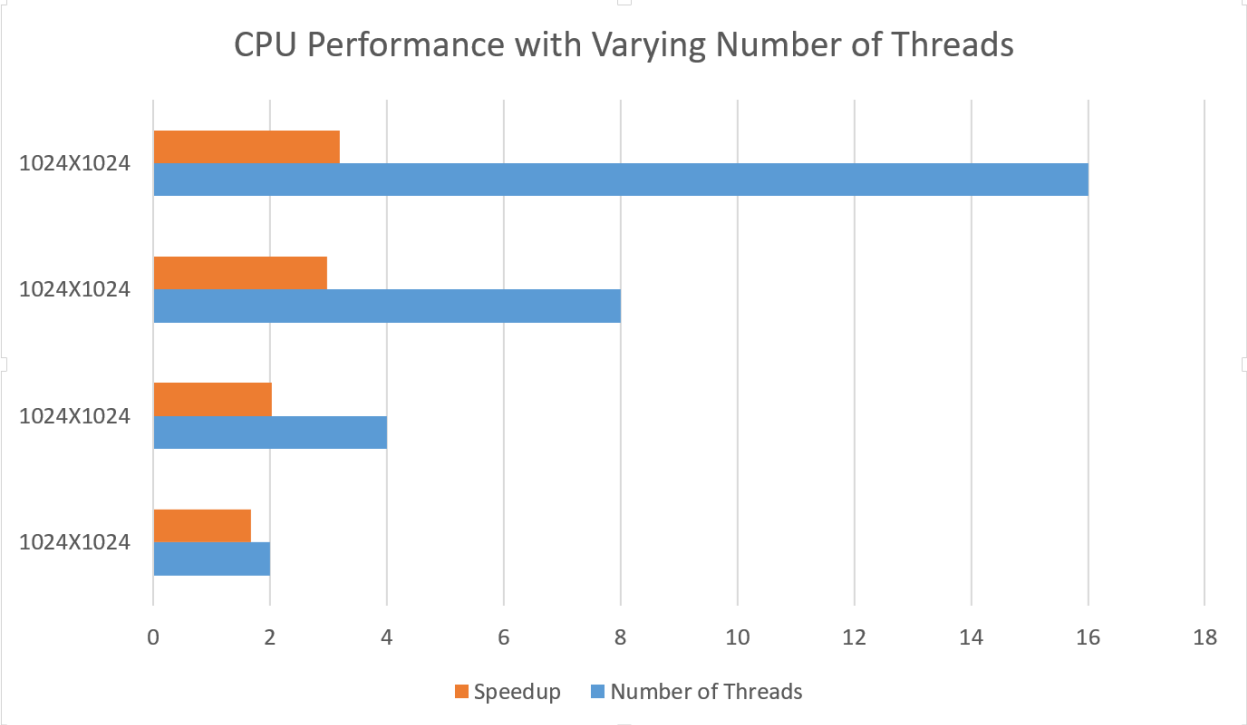

Figure 4. Speedup with varying number of Threads