Eric Rock

ECEC 413 – Lab 2

11/1/15

The implementation of the multithreaded Gaussian elimination computation program is located in src/compute_threaded.c. This source file includes the implementation of Gaussian elimination using OpenMP. We shall not discuss functionality which was provided as part of the assignment in this document.

The project was developed in Linux distribution Ubuntu 12.04, and has been tested on the college of engineering workstation xunil-04.coe.drexel.edu. The program may be compiled from the command line using *make* as follows:

```
> make
```

which will compile the program to operate using a default of 8 threads in the multithreaded implementation. To compile the program using a different number of threads, as well as with a different matrix size, one may compile the program as such:

```
> make NUM_THREADS=X MATRIX_SIZE=Y
```

where X is the number of desired threads and Y is the number of rows / columns in the matrix. To test the program, a *test* makefile target has been provided, which may be used as follows:

```
> make test NUM_THREADS=X MATRIX_SIZE=Y
```

which will recompile the program using X threads on a matrix of size Y^2 and run the output binary.

The program utilizes a macro definition at the top of *compute_threaded.c* to statically determine the number of threads to create at compile time, as well as a macro at the top of gauss_eliminate.h to statically determine the size of the input matrix.

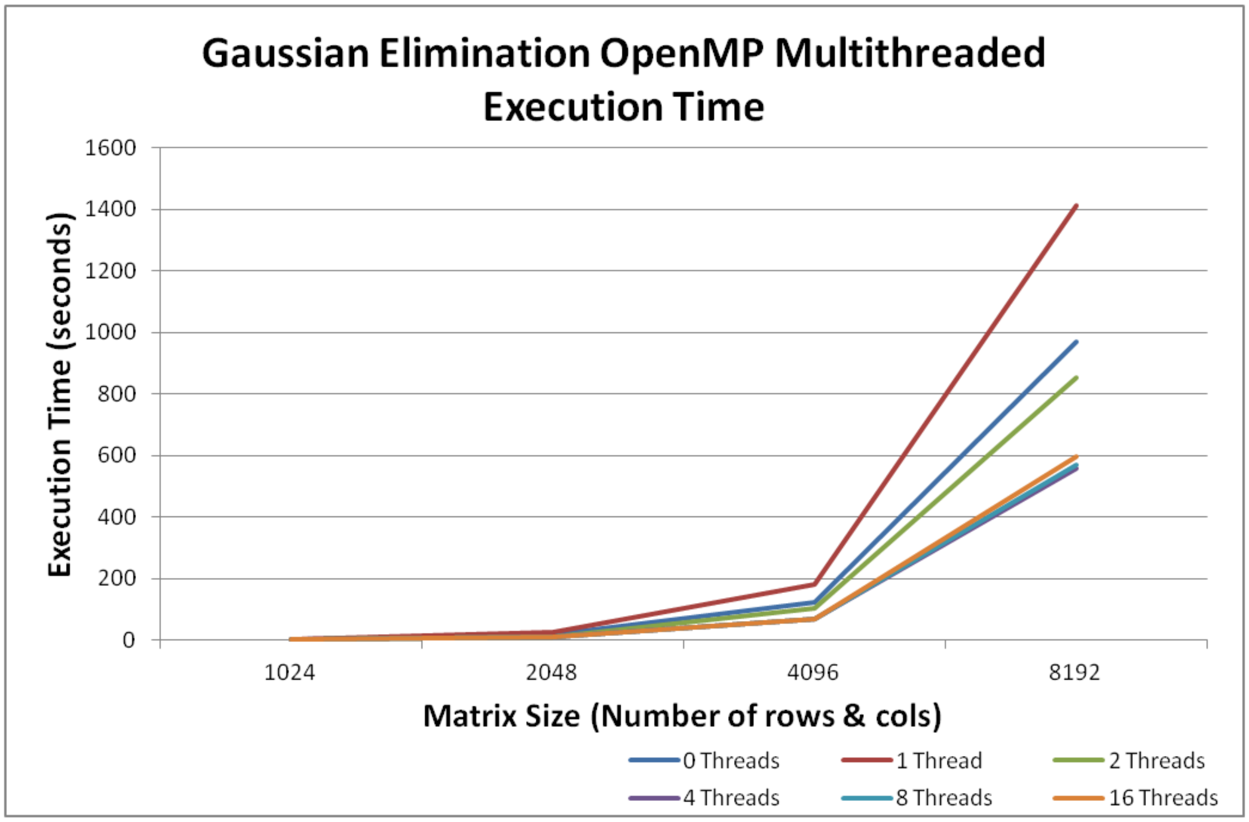
The Gaussian elimination problem is given as follows: we are given a square matrix A of floating point values of some size NxN. We must reduce this matrix into a square matrix B of size NxN such that the upper-left to lower-right diagonal elements are each 1, and the lower left triangular portion is all 0, via the following update rules: A row may be multiplied or divided by a scalar amount, and a row may be added to or subtracted from another row.

The single-threaded algorithm is given as follows:

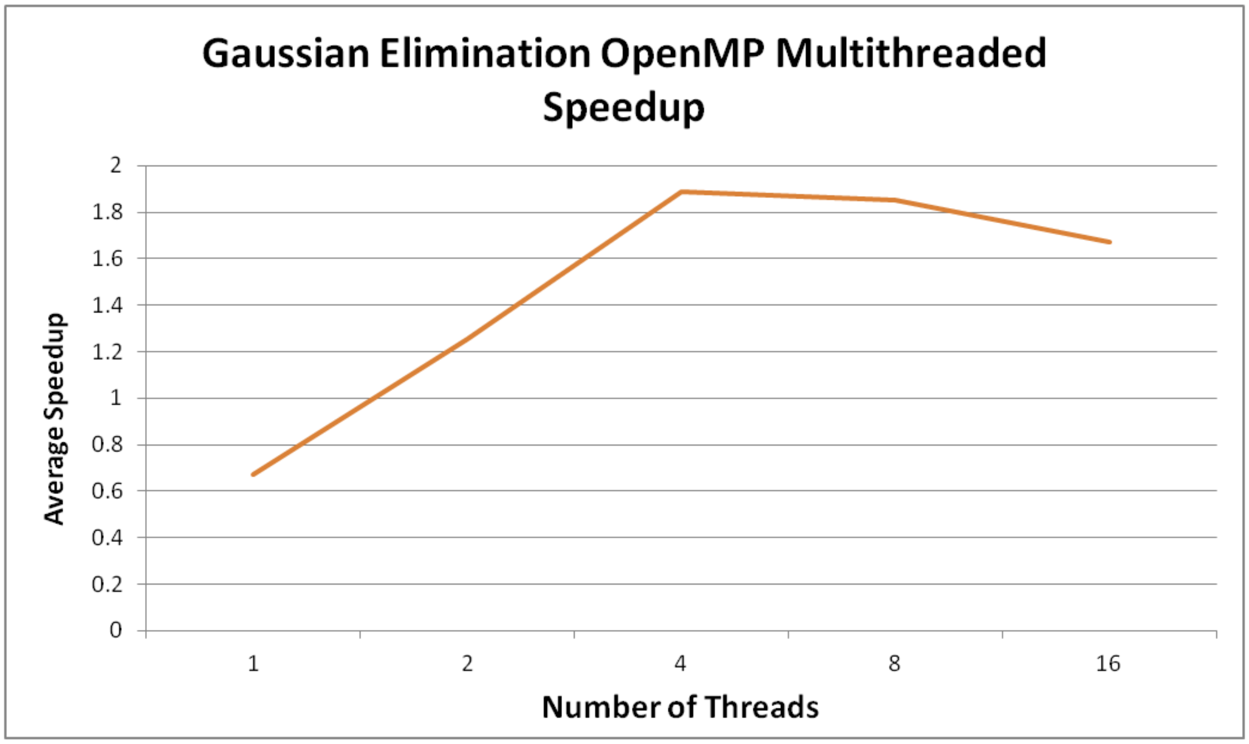
```
procedure GAUSS ELIMINATE(A, b, y)
     int i, j, k;
     for k := 0 to n − 1 do
          for j := k + 1 to n − 1 do
               A[k, j] := A[k, j] / A[k, k]; /* Division step. */
          end for
     y[k] := b[k] / A[k, k];
     A[k, k] := 1;

          for i := k + 1 to n − 1 do
               for j := k + 1 to n − 1 do
                    /* Elimination step. */
                    A[i, j] := A[i, j] - A[i, k] × A[k, j];
               end for
               b[i] := b[i] − A[i, k] × y[k];
               A[i, k] := 0;
          end for
     end for
```

In order to parallelize this algorithm, we simply decompose iterations of several for loops in the above serial code, in order to speed up each iteration of the outer loop. Due to limitations in OpenMP, we are unable to parallelize each for loop in the code, nor would doing so result in an appreciable increase in parallelism without significant numbers of available cores. The for loops chosen for parallelization are the loops in line 4 and line 10.

A graph relating runtime to the input size for 0 (single threaded), 1, 2, 4, and 8 threads are provided below:



Additionally, the average speedup for each class of parallelism is given below:


Gaussian Elimination OpenMP Multithreaded Speedup
Average Speedup
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
16
Number of Threads