DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

Parallel Computer Architecture ECEC-622

TITLE: PThreads Assignment 1: Gaussian Elimination

GROUP MEMBERS: Gregory Matthews and Mark Klobukov

INSTRUCTOR: Dr. Kandasamy

DATE SUBMITTED: 2/11/2017

DATE DUE: 2/13/2017

Assignment description:

The first PThread 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] × 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
```

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 parallel code:

1) gauss_eliminate_using_pthreads()

The function that is called from the main() is called **gauss_eliminate_using_pthreads()**. It is shown below. This is now the functions that the threads execute; this is the function that creates the threads and then joins them.

```
/* Write code to perform gaussian elimination using pthreads. */
void
gauss_eliminate_using_pthreads (float *U, float * A, unsigned int
num_elements)
{
    unsigned int i, j, k; // loop counters

    ///COPY CONTENTS OF MATRIX A INTO MATRIX U
    for (i = 0; i < num_elements; i ++)
        for (j=0; j< num_elements; j ++)
            U[num_elements * i + j] = A[num_elements*i + j];
    ///

    pthread_t thread_id[NUM_THREADS]; //data structure to store
thread ID
    pthread_attr_t attributes; //Thread attributes
    pthread_attr_init(&attributes); //initialize thread attributes
to default

    ARGS_FOR_THREAD * args_for_thread[NUM_THREADS];

    for(i = 0; i < NUM_THREADS; i++) {
        args_for_thread[i] = (ARGS_FOR_THREAD *)malloc(sizeof
(ARGS_FOR_THREAD));
        args_for_thread[i]->thread_id = i; //thread ID
        args_for_thread[i]-> num_elements = num_elements;
        args_for_thread[i]->matrixA = A;//can g oaway
        args_for_thread[i]->matrixU = U;
    }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&thread_id[i], &attributes,
gauss_eliminate, (void*) args_for_thread[i]);
    }
```

```
    //DO THE JOIN HERE
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(thread_id[i], NULL);
    }
```

As the pictures above show, this function initializes the matrix that will hold the result, initializes a part of the thread arguments for each thread, creates threads, and lastly joins them after they are done.

2) **gauss_eliminate()**

The following function was executed by each thread. It is called **gauss_eliminate().**

Thread arguments are type casted, then num_elements variable is given a value for each thread (they are all equal to matrix size), and chunk size for each thread is calculated. Since the parallelization is done inside the k-loop from the algorithm on the first page, chunk size needs to be recalculated for every single iteration of this outermost loop.

```
//this function is executed by each thread individually
void * gauss_eliminate(void * args) {
    //type cast the thread's arguments
    ARGS_FOR_THREAD * my_args = (ARGS_FOR_THREAD *)args;
    int k, chunk, i, j;
    int num_elements = my_args->num_elements;
    //algorithm's outermost loop
    for (k = 0; k < num_elements; k++) {
        //printf("Iteration # %d \n", k+1);
        chunk = (int)floor((float)(num_elements - k) / (float)
NUM_THREADS); //recalculate chunk size for each iteration

        //find this thread's individual offset for division
        my_args->offset = my_args->thread_id * chunk;
        //do division
```

Then the two main algorithm steps begin: division and elimination. Division is performed on a single row at a time, so the chunks are defined as a group of consecutive elements of the row (strides are not used to minimize cache misses). Elimination, however, is performed on the (k-1)x(k-1) matrix, so the parallelization was done on rows. That is, each thread was given a group of rows defined by the chunk size and worked on the entire row, from first to last entry.

Both the division and elimination step are wrapped around if-statement that checks the thread's ID. The reason for this is best explained with an example: let 50 be the number of elements left, and 4 be the number of threads. Then the chunk size is floor(50/4) = 12. So threads 1 to 3 will go from 1 to 48, and then the last thread is supposed to go from 48 to 52. But there are only 50 elements, and segmentation fault is then inevitable. Therefore, the last thread needs to stop executing its loop when it detects that the loop counter is equal to the number of rows in the matrix.

Division step is shown below. It is necessary to add a barrier afterwards because the elimination step depends on the modification made during the division step. Computation is exactly like in compute_gold().

```
       if (my_args->thread_id < (NUM_THREADS - 1)) {

           for (j = (k + 1 + my_args->offset); j < (my_args->offset + k + 1 + chunk); j++) {
               my_args->matrixU[num_elements*k + j] = (float)((float)my_args->matrixU[num_elements*k + j] / (float)(my_args->matrixU[num_elements*k + k]));

           }
   } else { //this takes care of the number of elements that the final thread must process
           for (j = (k + 1 + my_args->offset); j < num_elements; j++) {
           my_args->matrixU[num_elements*k + j] = (float)((float)my_args->matrixU[num_elements*k + j] / (float)(my_args->matrixU[num_elements*k + k]));
           } //end for

       } //end else

       barrier_sync(&barrier);
```

Elimination step is shown below. Because of dependencies between each iteration of the outermost k-loop, the barrier is placed at the end of elimination too.

```
       //do elimination
       if (my_args->thread_id < (NUM_THREADS - 1)) {
           for (i = (k + 1) + my_args->offset; i < (k + 1 + my_args->offset + chunk); i++) {

               for (j = k+1; j  < num_elements; j++) {
                   my_args->matrixU[num_elements*i + j] = my_args->matrixU[num_elements*i+j] - (float)(my_args->matrixU[num_elements*i+k] * (float)my_args->
matrixU[num_elements*k+j]);
               }

           my_args->matrixU[num_elements * i + k] = 0;

}
       } else {

           for (i = (k+1) + my_args->offset; i < num_elements; i++) {
               for (j = k+1; j < num_elements; j++) {
                   my_args->matrixU[num_elements*i + j] = my_args->matrixU[num_elements*i+j] - ((float)my_args->matrixU[num_elements*i+k] * (float)my_args->
matrixU[num_elements*k+j]);
                   }

           my_args->matrixU[num_elements * i + k] = 0;
           }
       } //end else

   my_args->matrixU[num_elements*k + k] = 1;
   barrier_sync(&barrier2);


   } //outermost for loop with k
```

3) Barrier function

The function shown below is taken from Dr. Kandasamy's thread synchronization examples on BBLearn. It makes all threads wait until the counter is incremented to NUM_THREADS, thus implementing a barrier.

```
/* The function that implements the barrier synchronization. */
void
barrier_sync(barrier_t *barrier)
{
    pthread_mutex_lock(&(barrier->mutex));
    barrier->counter++;
   // printf("Barrier at %d\n", barrier->counter);
    /* Check if all threads have reached this point. */
    if(barrier->counter == NUM_THREADS){
        barrier->counter = 0; // Reset the counter
        pthread_cond_broadcast(&(barrier->condition)); /* Signal this condition to all the blocked threads. */
    }
    else{
        /* We may be woken up by events other than a broadcast. If so, we go back to sleep. */
        while((pthread_cond_wait(&(barrier->condition), &(barrier->mutex))) != 0);
    }

    pthread_mutex_unlock(&(barrier->mutex));
}
```

Results:

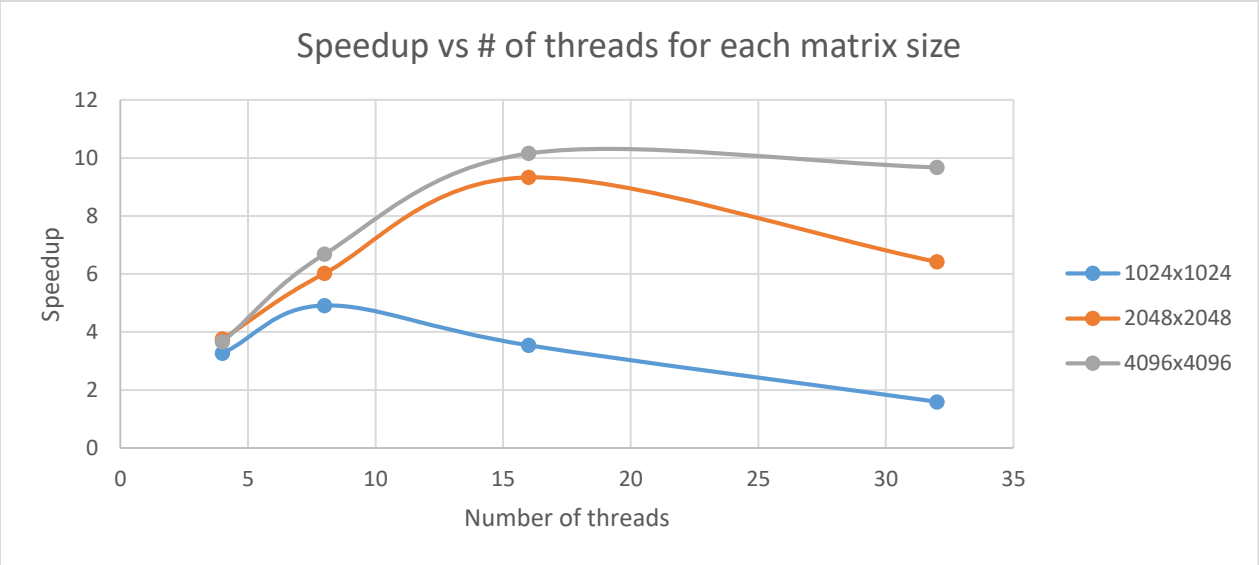
To compile the code, please use the following command:

gcc -pthread -o gauss_eliminate gauss_eliminate.c compute_gold.c -std=gnu99 -lm

The table below summarizes the results of this assignment.

Matrix size	# threads	Time with serial program (sec)	Time with parallel program (sec)	Speedup
1024x1024	4	3.6558	1.12	3.264107
	8	3.44175	0.70101	4.909702
	16	3.45339	0.9765	3.536498
	32	3.43995	2.16242	1.590787
2048x2048	4	30.32058	8.06992	3.757234
	8	29.62007	4.91921	6.021306
	16	28.95639	3.10307	9.33153
	32	29.20924	4.54986	6.419811
4096x4096	4	237.92714	64.97565	3.661789
	8	233.67082	34.95517	6.684872
	16	231.09708	22.75493	10.15591
	32	230.00357	23.77105	9.675785

Table 1: Parallel time, serial time, and speedup for each matrix size/thread number combination



These results conform with Gustafson's law. It did not make much sense to use the maximum number of threads on a 1024x1024 matrix, so the peak performance was achieved with 8. The 2048x2048 matrix elimination performance peaked at 16 threads, and the largest matrix elimination also had the greatest performance at 16 threads, and it decreased only slightly with 32 threads. A reasonable hypothesis, in accordance with Gustafson's law, is that if the matrix size was further increased, using 32 threads would cause the greatest speedup of all trials. This confirms that it is necessary to take into account the fact that problem sizes increase with additional available computing power.

Compile with:

gcc -pthread -o gauss_eliminate gauss_eliminate.c compute_gold.c -std=gnu99 -lm