DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

ECEC-413

Gaussian Elimination

Chris Kasper
Prof. Naga Kandasamy
DATE: 04/21/19

Multi-threaded Design

The main parallel tool that is used for extending the serial Gaussian Elimination algorithm to a multi-threaded one, are barriers. They are used to make sure the threads are running the same logical in parallel, and that each thread must wait for the other threads in order to proceed. The logic for the thread setup is as follows:

```
pthread_t *worker = (pthread_t *) malloc (NUM_THREADS * sizeof (pthread_t));
pthread_barrier_init(&divBarrier, 0, NUM_THREADS);
pthread_barrier_init(&elimBarrier, 0, NUM_THREADS);

for (i = 0; i < NUM_THREADS; i++) {
    args_thread = (ARGS_FOR_THREAD *) malloc (sizeof (ARGS_FOR_THREAD));

    args_thread->threadID = i;
    args_thread->U = &U;

    if ((pthread_create (&worker[i], NULL, compute_gold_pthread, (void *) args_thread)) != 0) {
        printf ("Cannot create worker thread \n");
        exit (EXIT_FAILURE);
    }
}

for (i = 0; i < NUM_THREADS; i++) {
    pthread_join (worker[i], NULL);
}

pthread_barrier_destroy(&divBarrier);
pthread_barrier_destroy(&elimBarrier);
```

As you can see, the first thing we do is initialize the barriers to be used in the algorithm. There are two: one for the division step, and one for the elimination step. Each thread is then given an ID, as well as a pointer to the matrix structure. Once all the threads finish, the barriers are destroyed.

The logic for the pthread computation is actually very similar to the serial logic:

```
for (k = 0; k < num_elements; k++) {
    for (j = (k + threadID + 1); j < num_elements; j+= NUM_THREADS) {
        U[num_elements * k + j] = (float) (U[num_elements * k + j] / U[num_elements * k + k]);  /* Division step. */
    }
    pthread_barrier_wait(&divBarrier);

    for (i = (threadID + k + 1); i < num_elements; i+= NUM_THREADS) {
        for (j = (k + 1); j < num_elements; j++) {
            U[num_elements * i + j] = U[num_elements * i + j] - (U[num_elements * i + k] * U[num_elements * k + j]);    /* El
        }
        U[num_elements * i + k] = 0;
    }
    pthread_barrier_wait(&elimBarrier);
}

// Each thread has rows to set the principal diagonal entry
for (k = 0 + threadID; k < num_elements; k += NUM_THREADS) {
    U[num_elements * k + k] = 1;   /* Set the principal diagonal entry in U to be 1. */

}
```

The main idea behind this logic is that each thread goes through the division and elimination computations together. During the division stage, all threads are working on the same row, but are responsible for the columns allocated it (which is done in a round robin fashion). Once a thread finishes its work in this step, it waits at the first barrier to make sure it proceeds once all the other threads are finished.

During the elimination step, each thread works on the rows allocated to it (same round-robin fashion) and does the elimination computation on the elements in their rows. They also set the values under the pivot points in the matrix to be zero. Once the threads finish the elimination step, they wait at the barrier, and then proceed to the next row in the loop to do the division step.

Once both the division and elimination steps are finished for the whole matrix, the principal diagonal entry has to be set to 1 for each row in the matrix. The work for this is allocated equally among all threads.

Speedup

Table 1: Speedup of Gaussian Elimination on xunil-05

Matrix Size	Single Thread	4 Threads	8 Threads	16 Threads	32 Threads
512 x 512	0.09 s	0.05s	0.04s	0.05s	0.07s
1024 x 1024	0.55s	0.18s	0.14s	0.15s	0.16s
2048 x 2048	3.75s	0.98s	0.62s	0.60s	0.57s