Drexel University
Electrical and Computer Engineering Department

ECEC 414
High Performance Computing

Lab 4 Assignment
March 3rd, 2014

Julian Kemmerer

Cholesky Decomposition

Design

The PThreads implementation of the Cholesky Decomposition differs significantly from the original code. Originally, the naive choice was made to create and destroy threads on each iteration of the outer ‘k’ (row) loop. This implementation did not improve performance - actually slowing the process by 2-3x.

The second, and final implementation creates threads only once at the start of execution and uses barrier synchronization mechanisms to align execution of various portions of the decomposition. The outer ‘k’ (row) loop must maintain execution order. That is, each ‘k’ iteration must occur in order. Within a single ‘k’ (row) iteration, the decomposition steps must also maintain order. It finally within each step that parallelism can be exploited. Pseudo-code below outlines the threaded function.

```
thread(thread_info):
	//Copy the contents of the A matrix into the working matrix U
	copy(U,A,thread_info)

	sync_threads()

	//Perform the Cholesky decomposition in place on the U matrix
	for(rows in U):
		//Only one thread does diagonal and division steps
		if(thread_info.id ==0):
			// Take the square root of the diagonal element
			U[k,k] = sqrt(U[k,k]);

			//Division step
			for(j = k+1 to rows-1):
				U[k,j] /= U.elements[k,k];

		sync_threads()

		// Elimination step
		range = recalc_range(k,thread_info)
		for(i in range):
			for(j = i to rows-1):
				U[i,j] -= U[k,i] * U[k,j]

		sync_threads()

	sync_threads()

	//Zero out the lower triangular portion of U
	zero_out(U,thread_info)
```

Figure 2. Pseudo-code illustrating the PThread implementation of the Cholesky Decomposition.

Note that the threaded function appears very similar to the original decomposition in total but the loop boundaries are thread specific and execution at a ‘step’ granularity is done in lock-step between threads. To further explain the bolded portions of code we start with the ‘thread_info’ passed into this threaded function. This object contains information regarding iterator boundaries specific to this thread for each step of decomposition. These objects are populated before threads are spawned. This allows the initial copy operation and final zero out operation to also be done in parallel. The largest portion of parallelism comes from the elimination step. At each row iteration a new k value is available and used, along with thread information, to calculate the portion of elimination this thread should handle. Following each step, thread execution is aligned via a barrier structure handled in the ‘sync_threads’ function. In short, areas between ‘sync_threads’ statements are executed in parallel.

Note: source code also contains work for OpenMP implementations- this was for another class. Ignore it.

Performance

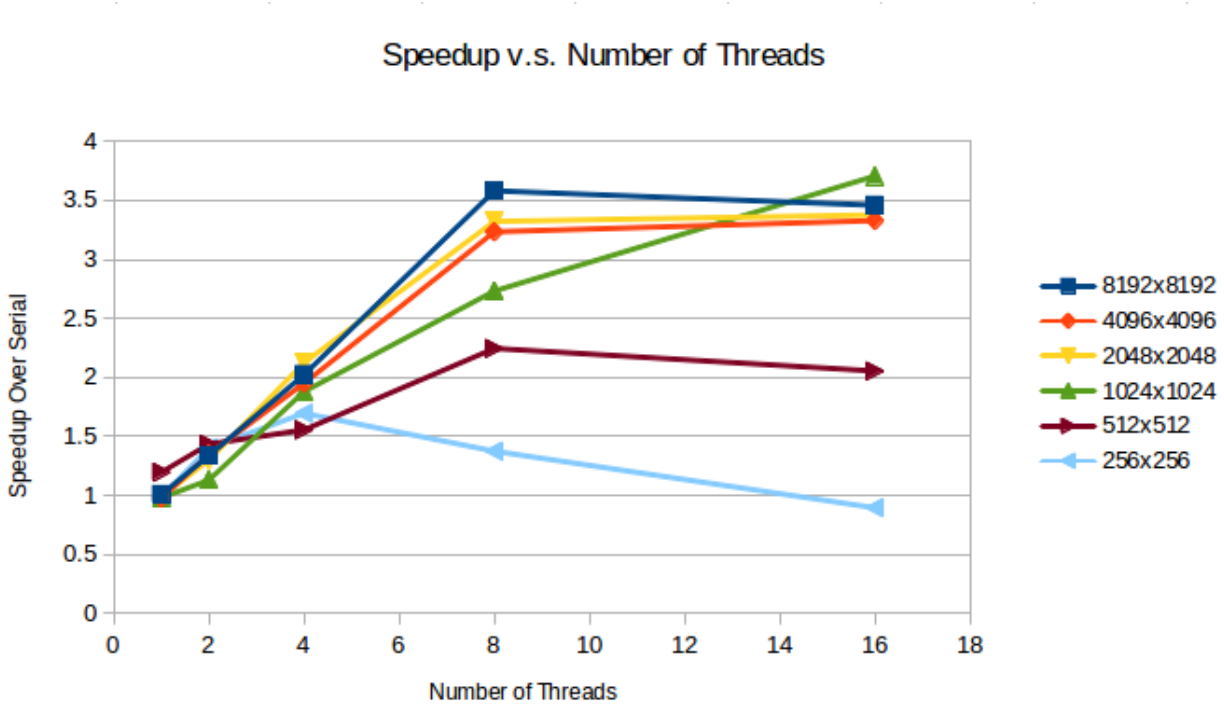

Figure 3. Speedup over serial for 2, 4, 8, and 16 threads
processing 256, 512, 1024, 2048, 4096, and 8192 dimensioned matrices.
(Different matrix sizes also remnant from different class)

In the figure above, it can be seen that PThread implementations do show speedup over serial execution. In addition, several interesting data points and patterns are made clear. First, the largest speedup is seen when using PThreads for matrices of sizes of 1024 or larger (~3.6x speedup). This can be explained by the larger matrix size providing a larger and more continuous data set (in regards to memory accesses, and longer durations between pauses to synchronize execution). The second interesting data pattern is the poor performance of PThreads with matrix size 256. When using a large number of threads for a relatively small amount of data the overall efficiency of the algorithm is quite low. Each thread does only a small portion of work making the forking of threads and communication between them costly relative to the work done by each thread. The final interesting data pattern is the trend towards a *maximum* speedup of 3-4x as the number of threads increases. This reinforces the fact that this problem is not 'embarrassingly' parallel but rather just contains parallelizable components.

For more specifics on this technique and implementation, please see the commented source files.