

**Drexel University**

**Electrical and Computer Engineering Dept.**

**Parallel Computer Architecture ECEC-622**

**TITLE: Midterm Problems: Cholesky and Solver**

**GROUP: Mark Klobukov and Gregory Matthews**

**INSTRUCTOR: Prof. Kandasamy**

**DATE SUBMITTED: 2/18/2017**

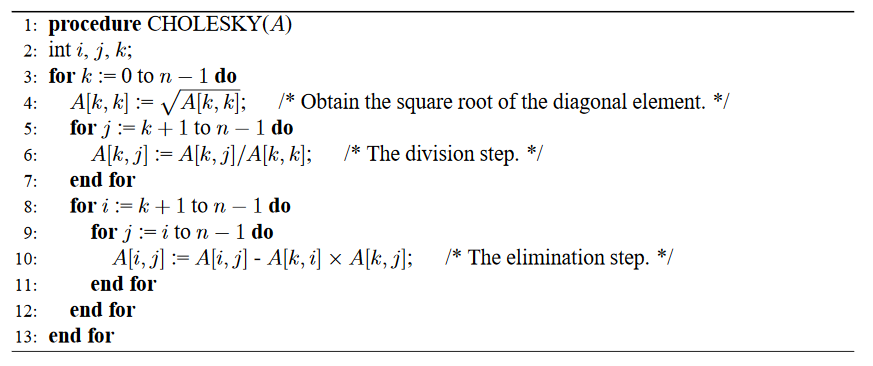
**DATE DUE: 2/19/2017**

**Problem 1: Cholesky Decomposition Method**

**Problem description:**

This problem involved development of multi-core implementation of the Cholesky decomposition method using PThreads and OpenMP. Cholesky decomposition method is a technique for solving a system of linear equations of the form Ax = b if the matrix A is symmetric and positive definite. Numerical stability is ensured since the method does not require pivoting since the diagonal elements are positive definite.

The algorithm for Cholesky method is as follows:



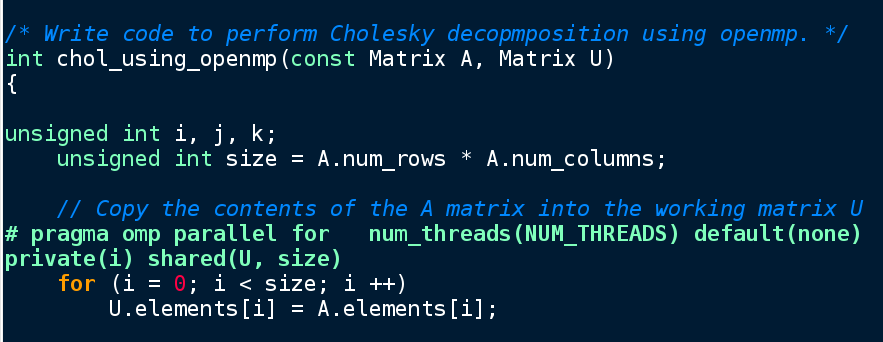
**Parallelization approach:**

In the above algorithm, it can be noticed that there are no data dependencies between iterations of the for-loops on line 5 and on line 8. That is, the values computed during iteration j do not depend on the values computed during iteration (j-1). Therefore, these two loops can be parallelized by breaking them up into chunks of consecutive iterations and allowing one thread work on each chunk.

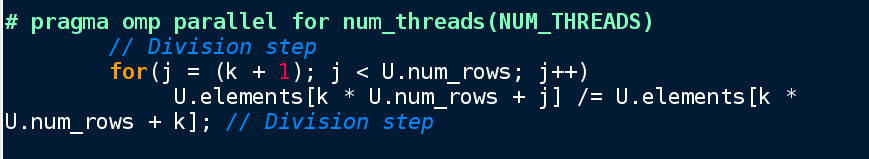
**OpenMP Implementation Description:**

The parallelization was first implemented using OpenMP. The contents of chol\_using\_openmp() function are identical to the chol\_gold() function with the exception of three **pragma** statements.

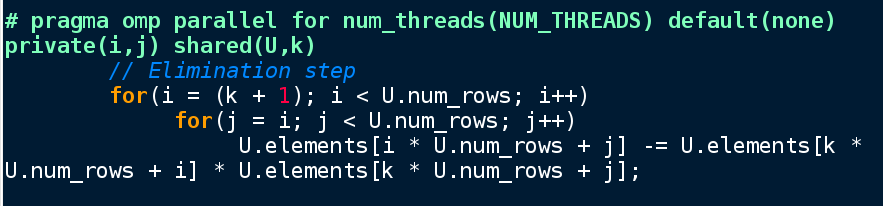
The first pragma statement, shown below, parallelizes copying the contents of matrix A into the working matrix U. Iteration variable is made private, and the constant NUM\_THREADS defines the number of threads to perform this operation.



The next pragma statement was added to parallelize the division step of the algorithm, corresponding to the loop on line 5 of the algorithm description above:



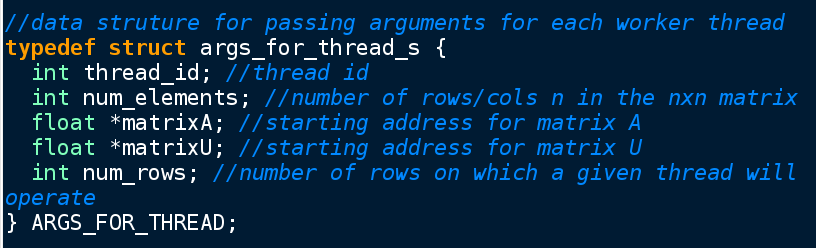
Finally, a pragma was added to the outer loop in the elimination step (line 9 in the algorithm). It was important to keep the iteration variables private but the worker matrix shared.



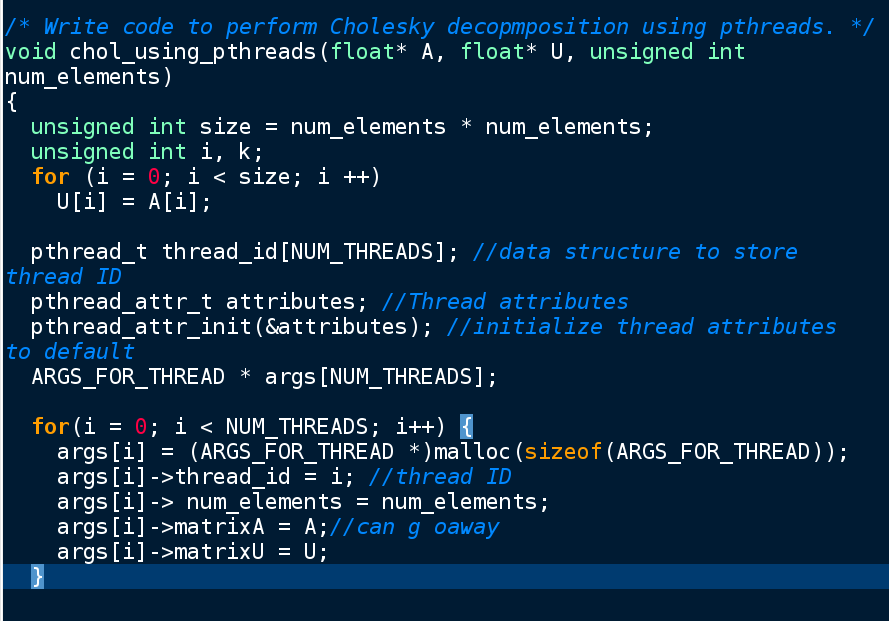
**PThreads Implementation Description:**

For the pthread implementation, two new functions were created. The first one served the purpose of initializing thread arguments, creating the threads, and joining the threads. The second function created was a worker thread function. Both are discussed below.

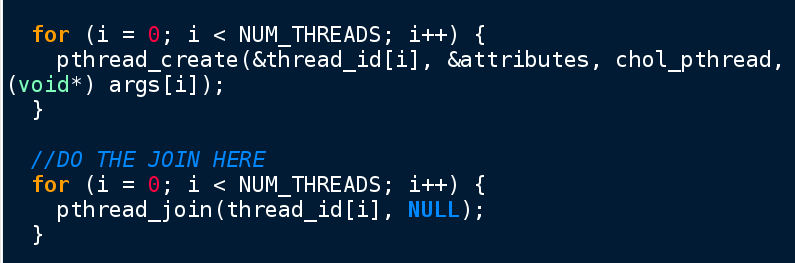
A new data structure to hold thread arguments was defined as follows. It contained all the data necessary for each thread to know to perform its computations.



The function **chol\_using\_pthreads()** populated this data structure, created threads, and joined them. Initialization of arguments:

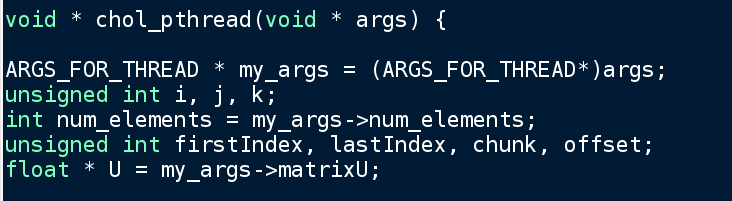


Creating and joining the threads:

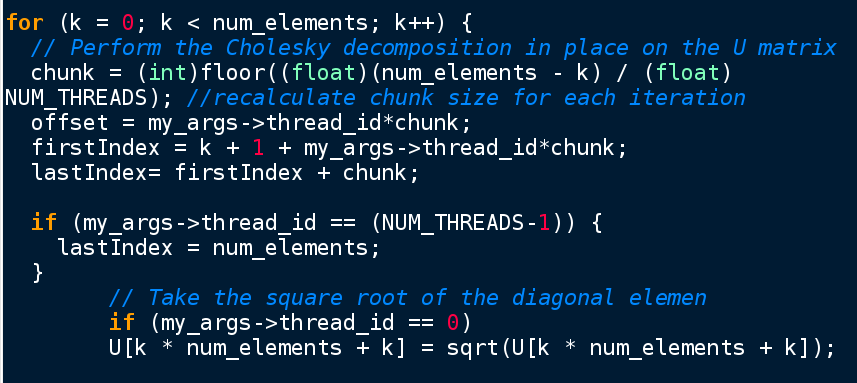


When the threads were created, each one was instructed to execute the **chold\_pthread()** function that is described next.

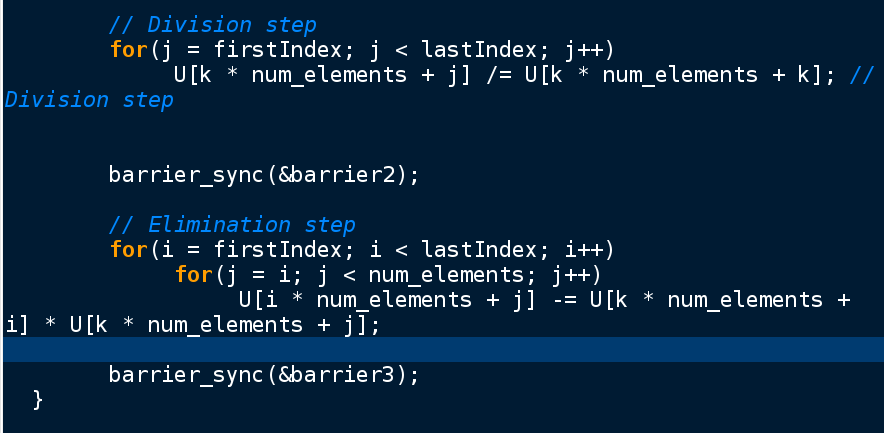
Definition of the worker thread function, type-casting of the input arguments, and initialization/declaration of several variables:



For each iteration of the for loop with the iteration variable k (see line 3 of the algorithm), the chunk size for each thread had to be recalculated because there were fewer and fewer elements whose values had to be computed. Therefore, at the beginning of each iteration, a new computation of chunk, offset, and firstIndex/lastIndex for the division-step and elimination-step loops are performed:

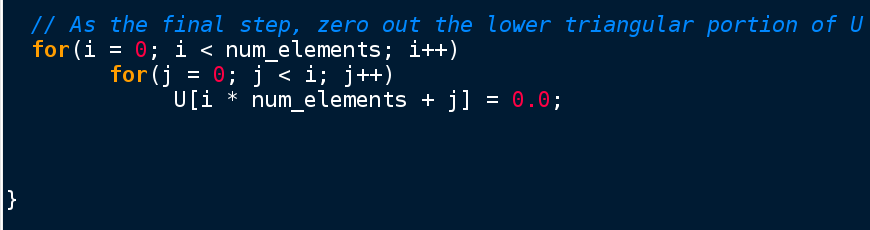


After synchronizing the threads, thus making sure that each one is aware of the update of the diagonal (the square root of which was taken), the threads started their individual division and elimination steps:



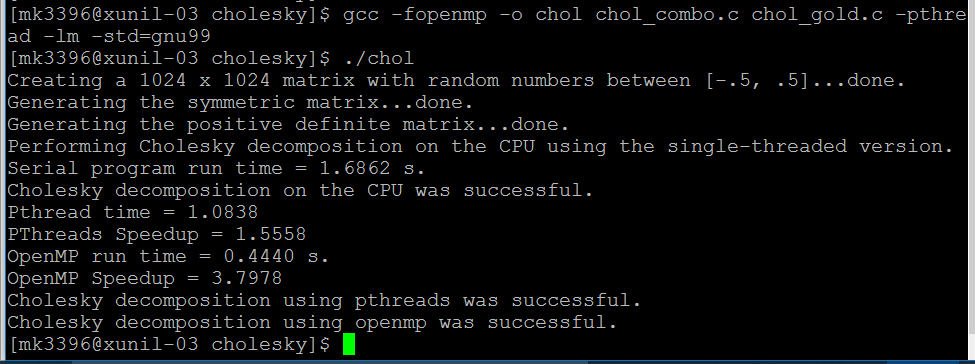
The body of the for-loops is the same as in the serial algorithm. What is different is the loop index boundaries recomputed at the beginning of each k iteration, as well as the synchronization barriers between the steps to ensure that the correct values are used in the next stage of the algorithm. Although there is no data dependency inside the division and elimination steps, there is a data dependency between these two algorithm stages.

When the division and elimination steps are completed, the lower-triangular portion of the matrix had to be zeroed out:



**OpenMP and PThread Implementation Results:**

Sample run of the program for 1024x1024 matrix with 16 threads:



Note the compilation command is **gcc -fopenmp -o chol chol.c chol\_gold.c -pthread -lm -std=gnu99**

The terminal displays serial run time, OpenMP run time, PThread run time, the corresponding speedups, and whether the decomposition using the two parallel methods was successful.

Timing data was collected for different combinations of matrix size and the number of threads, as presented in the following tables. The OpenMP timing was done on xunil-03; the PThread timing was performed on xunil-05.

Table 1: Summary of OpenMP timing results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Matrix size | # threads | Time with serial program (sec) | Time with parallel program (sec) | Speedup |
| 512x512 | 2 | 0.283 | 0.318 | 0.889 |
| 4 | 0.279 | 0.188 | 1.485 |
| 8 | 0.270 | 0.138 | 1.951 |
| 16 | 0.208 | 0.100 | 2.085 |
| 1024x1024 | 2 | 1.698 | 1.755 | 0.967 |
| 4 | 1.631 | 1.037 | 1.574 |
| 8 | 1.622 | 0.641 | 2.530 |
| 16 | 1.681 | 0.464 | 3.620 |
| 2048x2048 | 2 | 13.431 | 14.898 | 0.902 |
| 4 | 12.983 | 8.349 | 1.555 |
| 8 | 13.019 | 4.808 | 2.708 |
| 16 | 13.285 | 3.000 | 4.428 |

Table 2: Summary of PThread timing results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Matrix size | # threads | Time with serial program (sec) | Time with parallel program (sec) | Speedup |
| 512x512 | 2 | 0.177 | 0.139 | 1.269 |
| 4 | 0.177 | 0.102 | 1.734 |
| 8 | 0.217 | 0.101 | 2.159 |
| 16 | 0.215 | 0.200 | 1.078 |
| 1024x1024 | 2 | 1.449 | 1.080 | 1.342 |
| 4 | 1.447 | 0.655 | 2.208 |
| 8 | 1.456 | 0.456 | 3.190 |
| 16 | 1.478 | 0.427 | 3.459 |
| 2048x2048 | 2 | 11.472 | 8.794 | 1.305 |
| 4 | 11.090 | 5.223 | 2.123 |
| 8 | 10.792 | 2.857 | 3.778 |
| 16 | 11.491 | 2.088 | 5.502 |

The following graphs illustrate the relationship between the number of threads and speedup for each matrix size.

An increase in the number of threads caused a greater speedup in the OpenMP implementation. For the PThread version, however, the speedup got worse for the smallest matrix. This is reasonable because after a certain point, the overhead introduced by additional threads is not offset by an equal or greater increase in performance. Such phenomenon was observed in almost all previous programming assignments in this class that involved comparison of speedups between larger and smaller matrices.