CS160 – Winter 2018 – Programming Assignment #6

Due 1159pm, Mar 16, 2018 (Friday)

The goal of this assignment is to use OpenMP to parallelize a specific matrix decomposition, called the Cholesky decomposition.

You will be writing short "reports" for various aspects. All of your timings MUST RUN ON TSCC.

Any reference to floating point or double means to use `double`.

This is team assignment. You may choose one other partner. You are not required to choose a partner

Partnering/Paired programming

When you partner with somebody, the intent is that you code together with one person at the keyboard and the other person commenting. Working together should get to the solution faster. You are NOT required to have a partner, the assignment is doable as a single programmer.

It is not acceptable to "split the assignment" with one student taking on one programming task and another taking a different one. You should expect detailed questions on the final that cover this assignment, if you split the assignment and try to give an excuse of "my partner did that part", that's your problem.

We will be going over the Cholesky decomposition during 9th week lectures. This is not the only place you will see it.

Part 2 is the most time-intensive from a programming perspective. Your goal should be to have part 2 completely coded before Monday of the 10th week of classes.

PART 1

cholesky.c

You are being given a serial code, called cholesky.c, that creates an N x N matrix (Called A in this discussion) that is symmetric, positive definite. Symmetric means that `A(i,j) = A(j,i)` for all `i and j`. Another way to say this is that $A = A^T$ where A^T is the transpose of the matrix `A`. Positive definite means that the Eigenvalues of A or all positive numbers. Eigenvalues are numbers λ, such that $Ax = \lambda x$ (x is called an Eigenvector of A). Cholesky.c computes a Lower Triangular matrix `L` such that $A = L\ L^T$ which is called the Cholesky decomposition of A.

```
./cholesky <N>
```

You need to read and understand the code completely. If you have a partner, work on it together. Look carefully at loop indices. When you compile and link the code, you will need the standard math library, compiling Cholesky can be accomplished with

```
cc -o cholesky -O3 cholesky.c cs160validate.c -lm
```

Task 1.1

Modify cholesky.c such that some key loops are parallelized using OpenMP directives. It's command line arguments should read the number of threads available for use in your OpenMP directives

```
./cholesky <N> <nthreads>
```

The current code measures only the performance of the Cholesky factorization itself. You should also measure the time for initialization and the time for validation. The times should be reported to standard output in a format similar to how the Cholesky factorization time is reported. The specific format is up to you, but you will be using the numbers to report on your findings, in a small report. You probably will find that initialization and verification take longer than the Cholesky factorization itself.

You should explore parallelizing different loops using different schedules. Which loops and how you parallelize them is up to you, but your goal should be improve performance. Start with simple directives and then add more detailed directives. Use this is an opportunity to understand how different strategies in openMP can change performance. Yes, you should attempt to improve performance of all three phases: initialization, factorization, and validation.

Task 1.2

Create a graph with an appropriate title, labeling of axes, that plots the time it takes compute the Cholesky factorization, the time for initialization and the time for verification at each of the dimensions. The format of the graph is up to you. For Matrix sizes use N=512,1024,1536,2048,3072,4096,8192 and threads 1,2,4,8. 8192 sizes may take several minutes to initialize/compute/validate at small thread counts. It is highly recommended that you only run the 8192 case a few times because of time constraints. You can discover many aspects of performance using smaller sizes.

When asking for a node allocation on TSCC use "`-l nodes=1:ppn=8`" to be allocated 8 cores on a node of TSCC. It does not make sense to parallelize with more threads than cores that are allocated.

Task 1.3

Write a short (~single page including graphic) that shows the performance of Task 1.2. In English, describe the different openMP directives you tried and what kinds of performance improvements you saw from each one. For 1024, 2048, 4096, 8192 sizes compute the speed-up of the Cholesky factorization (not the initialization and verification sets) for 2,4, and 8 threads. Call the report **task1.pdf**

Requirements/Hints/FAQ on Task 1

1. You do not have to check valid type of command-line arguments. We will test only with valid inputs. You do need to check for proper number of arguments.
2. Compile your code explicitly with gcc and use optimization option `-O3`.
3. You will be turning in one Makefile for the entire project for this part you must have a target called "cholesky that compiles the final (Task 1.2 version) of cholesky.
4. You may use any graphic software in which you are comfortable. Excel, OpenOffice, GoogleSheets, Word, etc. to create your graph and mini-report

What you will turn in Task 1

1. Modified cholesky.c
2. Makefile (targets “cholesky” and “clean” are required (no quotes))
3. task1.pdf

Task 2

The goal of this task is to create a block matrix version of cholesky.c This is significantly more involved than task 2. You will want to develop in stages and test the various components of your implementation. The next part of this writeup explains the block version of the Cholesky factorization.

The block version of the Cholesky factorization of an n x n matrix A , using a **block size of n_b**. One can write the factorization as follows

$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}$$

A_{11} is $n_b \times n_b$, A_{21} is $(n - n_b) \times n_b$, and A_{22} is $(n - n_b) \times (n - n_b)$. L_{11} and L_{22} are lower triangular.

First, let’s look at computing L_{11}. The equation is

$$A_{11} = L_{11}L^T_{11}$$

Your serial code can be used to compute this. A11 is n_b x n_b . So is L_{11}. In other words, you use the serial cholesky() function to compute this first block.

Now we can look at computing L_{21}.

(1) $$A_{21} = L_{21}L^T_{11}$$

You just computed L_{11} in the previous step, so that is known. A21 is also known, so (1) is just a set of N equations with N unknowns. We can compute both L_{21} and an UPDATE to the contents of A_{22} (more on this later)

$$L_{21} \leftarrow A_{21}(L_{11}^T)^{-1},$$

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T.$$

You may think that you need to explicitly compute the inverse of L^T_{11}, and then multiply to compute L_{21}, but the structure of L^T_{11} is very special (note that L_{21} is NOT lower triangular).

Let’s look at solving $L_{21}(0,0)$ (the upper-most element). $A_{21}(0,0)$ = L_{21}(row0) o L^T_{11} (col0), where “o” is the vector dot product. But L^T_{11} (col0) is nearly all zero. In fact, only L^T_{11} (0,0) is

non-zero. (the transpose of a lower-triangular matrix is upper triangular) Hence, $A_{21}(0,0) = L_{21}(0,0)\, L^T_{11}(0,0)$, and therefore $L_{21}(0,0)$ can be computed directly. Indeed, any element of $A_{21}(k,0) = L_{21}(\text{rowk}) \circ L^T_{11}(\text{col0})$, and the same equations apply.

The above says that the entire first column of L_{21} can be solved directly. Now, let's look at solving $A_{21}(0,1) = L_{21}(\text{row0}) \circ L^T_{11}(\text{col1})$. Again, the special structure of L^T_{11} is important. The second column is a vector that looks like (x,y,0, 0 …. 0). And you just found $L_{21}(0,0)$ directly in the previous section, So $A_{21}(0,1) = L_{21}(0,0)\, L^T_{11}(0,1) + L_{21}(0,1)\, L^T_{11}(1,1)$, Only $L_{21}(0,1)$ is unknown and therefore, it can now be solved. Once you know the first column of L_{21}, you can compute the second column.

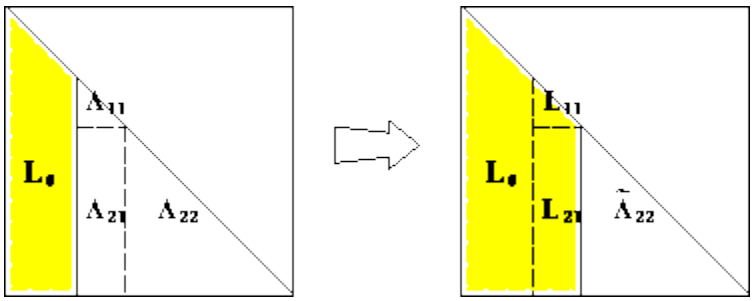
Iteratively, you can find the elements L_{21}, by solving the first column, then the second, then the third, etc. The process is called forward solving.

Once you have fully computed L_{21}, you will update A_{22}, More on this as you keep reading.

Block Cholesky Factorization.

With the above as background, the Cholesky factorization where the full matrix is decomposed into n_b x n_b blocks, can done as follows

A snapshot of the block Cholesky factorization algorithm in Figure below shows how the column panel $L^{(k)}$ (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22} is updated. The factorization can be done by recursively applying the steps outlined above to the $(n - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$.



In the above, L_0 has already been computed. Each iteration through the Cholesky factorization progressively makes whatever is left over smaller and smaller. At the beginning, L_0 is an empty matrix. The vertical panel is n_b wide. In your code, you can copy A into your L matrix, and then update L progressively. This will leave A unchanged. In high-quality library implementations, A (which is symmetric) is stored in the upper triangle, L, which is lower triangular is stored in the lower triangle. We're being a bit less careful and instead allocate two N X N matrices.

To summarize, the steps of the Block Cholesky factorization are as follows where the block size is n_b x n_b, the total size of the complete matrix is N x N, decomposed into K x K blocks of size n_b x n_b . **For this program, we will NOT consider N that is unevenly divided by n_b.**

1. Compute the Cholesky factorization of the diagonal block A_{11}.

   $$A_{11} \rightarrow L_{11} L_{11}^T$$

2. Compute the column panel L_{21},

   $$L_{21} \leftarrow A_{21} (L_{11}^T)^{-1}$$

3. Update the rest of the matrix,

   $$\tilde{A}_{22} \leftarrow A_{22} - L_{21} L_{21}^T = L_{22} L_{22}^T$$

Task 2.1

Create a code called blockCholesky, written in C that implements the block Cholesky factorization. It should have the following invocation

```
blockCholesky <K> <blksize>
```

- <K> is the number of blocks
- <blksize> is the size of block
- The matrix to be factored is (<K>*blksize) X (<K>*blksize)

blockCholesky should create a random, symmetric, positive-definite matrix (just like Cholesky) and then compute the factorization. We need to check the correctness of your solution and the sizes of matrices will be large enough the writing them out into files is not really efficient. Instead, when your factorization is complete, your main method should invoke

```
cs160validate(double * A, double *L, int N, double thresh);
```

A should be your original, randomly-generated symmetric positive definite matrix. L should be the Cholesky factorization of that matrix. This is the same signature as validate in cholesky.c. You may supply any value for thresh, we will use our own value internally. If your block representation does not

store your matrices in this single double array form, then copy your block representation to this format for the validation stage.

We will link your code with our own validator to insure that A is symmetric, positive definite and the $A = LL^T$ has been computed properly. Your starter code provides a header file, "cs160validate.h" and a dummy "cs160validate.c" that you can link in. You can use the validate function in the supplied cholesky.c file to validate your own answers.

Your output from blockCholesky should include timings for initialization, factorization, and validation in your code. You do not and should not time the call to cs160validate

Task 2.2

You are to explore different block sizes to improve the performance of the serial code over the non-blocked, single threaded version in task 1. You should explore block sizes: 16,32,48,64,80,96,112,128 for the matrix sizes used in task 1 (only up to 4096, it will take too long to try all these block sizes at 8192). You should try the best blocksize for 4096 for the 8192 matrix just to get a single measurement. If a blocksize does not evenly divide the matrix size, compute the time for matrix size that is the next integral size above. For example, for matrices near the size of 512x512 the following blksize:matrix dim is given

16:512,32:512,48:528, 64:512, 80:560,96:576,112:560,128:512

Your plots should have matrix size on the X Axis, time on the Y axis. Each blocksize should be its own curve on the same graph. You should also have a curve called "unblocked" from the data collected in task1 at 1 thread for comparison. You do not need the serial version to also compute times for the additional sizes (e.g. 528, 560, 576), the times from just 512, 1024, etc will be sufficient.

Write up your findings in a file called **task2.pdf.** What can you determine about blocksize vs. matrix size. Is one block size clearly better than all the others? Or is it a "depends on the size of the matrix, too" type of observation. This is another one of those open-ended programs where you are being given the chance to explore the performance space. It might take a few hours to gather all the performance numbers. The blocked version should be faster than the non-blocked version.

What you will turn in Task 2

1. All *.c and *h files you deem necessary to create the blocked solution.
2. Makefile (targets "blockCholesky" , "cs160validate" and "clean" are required (no quotes))
3. Task2.pdf

Task 3.1

Use OpenMP to a parallelize blockCholesky to create an executable called blockCholeskyMP. You should still link with cs160validate so that we can automatically validate your computation.

It should be called as

```
blockCholeskyMP <K> <blksize> <nthreads>
```

Task 3.2

Similar to task 1.3, create a graph of performance for 1,2,4,8 threads and compute speedups. You should use the best block size for each matrix size. (If the block size doesn't integrally divide N, pick the next size matrix closest that does). You will want to plot curves that show your best 1,2,4,8 thread performance. In the instances when the blocksize for single thread performance aligns with best block size for <n> thread performance, you should compute both the efficiency and speedup and place in a separate table.

In your performance writeup you describe your openMP parallelization strategies, why you chose particular strategies, and what kind of performance improvement you saw when comparing different approaches. You do not need to detail every single loop that you parallelized but you should select some representative ones. This means that you will need to compare and contrast the effects of at least two different ways of parallelizing the same loop(s). Do your best at explaining why you believe one strategy should have been better than another. Since openMP pragmas are easy to code, the goal of this writeup is to demonstrate your understanding of what openMP is doing for you. Call this writeup **task3.pdf** . Please attempt to keep this write-up under 3 pages in length

What you will turn in Task 3

1. All *.c and *h files you deem necessary to create the openMP-compatible blocked solution.
2. Makefile (targets "blockCholeskyMP" , "cs160validate" and "clean" are required (no quotes))
3. Task3.pdf

Other Requirements

- One Makefile for all three tasks
- Use optimization -O3 for your final codes
- No large files (e.g. no output files in your repository)
- We're not checking for correct *type* of arguments, but *will check for correct number* of arguments for each programming
- We won't test with "crazy" inputs. Your focus should be on the programming and the performance
- Reasonable commenting, as usual.
- Text in your writeups should be at least 11 point in a readable font (Times Roman, Arial, Calibri) with line spacing no more dense than how this document (the assignment document) is written (in Word using 1.08 spacing). All Margins must be at least 1". Page size must be US standard 8.5" x 11". Readers will be instructed to give you 0 points on any writeup that has spacing/fonts that are similar to:

  > This is very tightly packed text with a too small font. It's considered unreadable. If a reader sees your writeups using this type of spacing and/or small font you will get 0 points on the writeup. NSF/NIH have similar rules for their grants

What to turn in

- See the sections above.

Turning your program

It will be similar to PR5.

What you will be graded on

- Correctness of all of your solutions. We will run them.
- Clarity of explanation of your openMP parallelization and why you chose specific strategies.
- Indentation – Your code must be indented. Choose a style and stick with it.
- Comments – You must comment your code in a reasonable way. At a minimum, your name, ucsd email address, student ID must appear in comments. All your routines should be commented with a brief description of what the routine does, what it returns and a description of the parameters.
- Your task[123].pdf writeups. We'll be looking for reasonable explanations. Your total writeups should not exceed 5 pages. Don't take it as a challenge to put in every detail, but you need to be able to crisply explain what is happening. You won't lose points if a 3 page write-up is only 2 or 2.5 pages and "got the job done". Page counts INCLUDE figures.