SCA Laboratory
Lab3: Parallelizing a Cholesky Code

C. Álvarez, J. Bosch and D. Jiménez

Fall 2024

Contents

Note: This laboratory is optional and its main purpose is to give you a deeper understanding of the algorithms explained in theory while abridging the work that you will need to develop for the numerical assignment. Correspondingly, if you decide to fill a report with the results of the laboratory it will be graded and it will help to improve your numerical assignment final grade
If you decide to fill a report, deliver it in PDF at the "Racó". There are some **questions** on the document that will guide you to elaborate it. Don't worry if you are not able to answer the questions labeled as "optional". You can do it by pairs.

Part 1

Cholesky Algorithm

The objective of this laboratory is to understand different ways of parallelizing the execution of a Cholesky code and how they influence performance. As it has been explained in theory classes, the execution pattern is similar to other problems like LU factorization and the blocking approach can be applied to a vast majority of the linear algebra algorithms. You should refer to the "Direct methods" set of slides for more information.

In this laboratory you will use the same development environment as in the first laboratory. For connection and execution details please refer to Laboratory 1 documentation. Also, you can find in the first laboratory data some files that you may find useful to complete this laboratory (like `Paraver` configuration files).

All new files necessary to do this laboratory assignment are available in `/scratch/nas/1/sca0/sessions`. Copy `lab3.tar.gz` from that location to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab3.tar.gz"`. Also, remember to process the `environment.bash` file with this other command line `"source environment.bash"` in order to set all necessary environment variables[1].

1.1 Sequential Cholesky Algorithm

The Cholesky algorithm performs the same kind of process as the `LU` algorithm, that is, it decomposes a initial matrix into two triangular matrices (one lower and one upper). As with LU this is mainly used to solve the system of equations that it represents or invert the matrix. The main difference between Cholesky and `LU` decomposition is that Cholesky needs a positive definite matrix, that is, the matrix should be symmetric and have some other convenient characteristics. As a consequence Cholesky decomposition needs only half the operations than `LU` decomposition and decomposes the matrix into a LL^T system where the upper triangular matrix is, in fact, the lower triangular matrix transposed.

Inside the laboratory files you will find a sequential Cholesky algorithm already programmed in file `cholesky.c`. Look at it and try to understand the code, please refer to the "Direct methods" set of slides for more details. As you can see the actual algorithm is in the `omp_potrf` function and the longer parts of the code are in fact the initialization of the matrices and the final check.

Inside the `omp_potrf` function you will see in fact two approaches for solving the cholesky problem. The first one (that is compiled with the `cholesky` target) uses the `potrf MKL` function (as defined in `LAPACK` library). The second one (that is compiled with the `cholesky_seq` target) is manually programmed using the standard algorithm as you can see in the code. As seen in laboratory session 2, the `MKL` approach is already parallelized by the library[2].

Question 1: *Execute the code and time it for different sizes (e.g. 1024, 2048 and 4096 elements) using both approaches. How does the execution time grow with the number of elements in each case?*

[1] **Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in a `.profile` file in your home directory, which is executed when a new session is initiated.

[2] If you prefer you can also use `ATLAS` library instead of `MKL`. To do so, you only have to modify the source code and the Makefile as explained in the second laboratory session. If you use `ATLAS` be sure to indicate it clearly in the report.

1.2 OpenMP parallelization of the sequential algorithm

As you have seen in the previous section (otherwise something is probably wrong) the `MKL` code is significantly faster than the sequential code. Take into account that the `MKL` code is parallelized and in addition uses SIMD instructions and other kinds of "tricks" in order to have a good sequential base (probably faster than the simple one we are using that is only optimized by the compiler). Now you can improve the sequential code.

Copy the sequential code into a new `cholesky_omp.c` file so you can use the provided Makefile. Try to parallelize the sequential code using `OpenMP` pragmas. You can use the "Direct methods" set of slides as a guide to do it. Also you should create a `submit-omp.sh` code to submit it to the queues. You can use one of the `submit-omp.sh` scripts from the previous laboratory as a base. Don't spend a lot of time doing the parallelization as in order to obtain the same performance as MKL you would need to improve the base code also, but you should put at least one `OpenMP` pragma and obtain some speedup over the initial sequential version. If you feel corageous you can also try to improve the sequential code (e.g. using vectors) although it is not necessary.

Question 2: *Copy the parallelized code and the improved code (if you have improved it).*

Question 3: *Time your `OpenMP` parallelized code and find the optimun number of threads in order to execute it with the maximum performance. Which is the optimum number of threads in `boada`? Compare its performance against the previous two versions. Which are your conclusions?*

Part 2

Blocked code

Since BLAS and LAPACK libraries became a de facto standard, blocked implementations have gained popularity as they can be easily adapted to blocked algorithms. Creating a blocked Cholesky code follows the same pattern as creating a blocked `LU` code, as explained in theory classes. In the file `cholesky_blocked.c` you have the code of a blocked implementation of the Cholesky decomposition where each block of code can be computed either using the sequential implementation (if compiled using the `cholesky_blockseq` target) or the MKL implementation (using the `cholesky_blocked` target). Open the file, understand it and compile it.

Question 4: *Find the four different `BLAS/LAPACK` functions that are used to implement the blocked Cholesky decomposition. Explain what every function implements.*

When you create a blocked algorithm, one of the key parameters is the size of the block (`ts` in your code) and it can greatly influence the performance of your code. Also, although this can be corrected, note that the code provided only accepts matrix sizes that are multiple of the block size.

Question 5: *Change the value of `ts` to 16, 32 and 64 (declared in `cholesky.h`). Recompile the code (modifications to .h files not always trigger the automatic compilation) and time the execution in both blocked implementations. Which is the best block size for each implementation? Be careful to turn off the check when submitting large executions because otherwise the sequential check code will take too much time.*

Question 6: *Why do you think the blocked version has different execution times than the original version?*

Also, you can see that using a blocked code has to take into account the fact that blocked matrices are not stored in the same order as "standard" row matrices in C. In this case, in order to simplify the code, we have opted for transforming the matrix into a blocked matrix before doing the decomposition and transforming it back at the end. In this laboratory, we are not going to time this process as there are ways to either avoid it or hide it with the decomposition computation.

2.1 Improving the sequential blocked code

Now, as before, the `MKL` code is already parallelized as each call to a `MKL` function is already parallelized. However the sequential blocked version can also be improved by improving the base codes or parallelizing the with `OpenMP`. If you have already improved your base version in the previous part, adapt now the improvements to the blocked code. DO NOT parallelize the sequential blocked code as we are going to follow a different approach than before. Only answer the next question if you want to obtain an extra point in this lab report.

Question 7: ***Only do it for an Extra Point at your own risk.*** *Copy here your improvements to the sequential blocked code. Time the results.*

2.2 Parallelizing the blocked code

Now we are going to parallelize the blocked code. With blocked codes, usually a better alternative to `parallel for` is to use tasks. A task, as you may already know, is a code block that the compiler wraps up and makes available to be executed in parallel. With blocked codes, usually it is easy to parallelize them by annotating each block of computations as a task.

In code `cholesky_blocked_omp.c` you have a naive task parallelization of the blocked Cholesky decomposition. As you can see in the code, we have opted for annotating each block as a task. Also, in order to preserve the code correctnes and avoid having two tasks that modify the same data running at the same time, the main Cholesky function has some `taskwait` pragmas that close the parallelism whenever is necessary.

Question 8: *Draw a simple task scheme that shows the executing order of the blocks and the taskwaits place when a 3×3 decomposition is used. Which is the maximum parallelism in this case? How many steps (blocks in the critical path) are with this approach?*

As you can observe the parallelization outside the block granularity is also compatible with the parallelization inside the block granularity, so the parallelization could benefit both the `MKL` approach and the sequential approach.

Question 9: *Time again both approaches choosing the best number of threads for each approach when computing a 4096 matrix.*

As you can see in this case the optimal number of threads differs in each approach as the sequential approach doesn't use any internal parallelism while the `MKL` threads collide with the `OpenMP` threads.

2.3 Using dependences

In order to further improve the parallelism obtained in the latter approach we are going to get rid of the `taskwait` pragmas. You can observe that the `trsm` function is already annotated with some `depend` clauses in the `task` pragma. This clauses that contain the data that is read or written by the task help the runtime execute the tasks in the correct order without the `taskwait` pragma. As a result the tasks that before where executed separately now can be executed in parallel allowing better performance.

Question 10: *Draw a simple task scheme that shows the executing order of the blocks if all of them are annotated with dependences when a 3×3 decomposition is used. Which is the maximum parallelism in this case? How many steps (blocks in the critical path) are?*

Copy the `cholesky_blocked_omp.c` file into a new `cholesky_blocked_deps.c` file. Annotate the remaining tasks of the Cholesky decomposition with `depend` clauses and remove the `taskwait` pragmas (you have to maintain one `taskwait` pragma at the end of the decomposition to wait for all the tasks to finish). Be sure that your approach executes correctly before continuing. If you are stuck you can search Internet for examples of the task decomposition for Cholesky and follow them.

Question 11: *Time both approaches (`MKL` and hand-made) with your dependences code when computing a 4096 matrix. Use all the machine resources (maximum number of threads) and 64 as block size.*

Question 12: *Time again both approaches (`MKL` and hand-made) with your dependences code choosing the best number of threads for each approach when computing a 4096 matrix. Remember to report the number of threads you are using.*

Now try to improve a little bit the hand-made version. In the `omp_gemm` function, change the loop ordering from `j i__ l` to `j l i__`. Be careful to remove the `temp` variable. Now, verify that your code checks with a small matrix and afterwards time the hand-made version again for the 4096 matrix with as many threads as cores.

Question 13: *Copy the resulting code and the results in the report. Be sure to indicate the number threads that you are using. Which is the fastest option? Could the sequential approach be further improved? And the `MKL` one? Explain your conclusions after all the experiments.*