OpenMP exercise: Cholesky factorization

May 16, 2025

1 The Cholesky factorization

The Cholesky factorization is a well known linear algebra algorithm, formulated by André-Louis Cholesky, which decomposes a **symmetric positive definite** matrix A into the product of a lower triangular matrix L times its transpose, i.e., $A = LL^T$. This decomposition can, for example, be used to compute the solution of a linear system $Ax = b$ in three steps

1.	$A = LL^T$	Factorization	cost: $\mathcal{O}(n^3)$
2.	$z = L^{-1}b$	Forward substitution	cost: $\mathcal{O}(n^2)$
3.	$x = L^{-T}z$	Backward substitution	cost: $\mathcal{O}(n^2)$

If the size of the matrix is n, the cost of these three steps is, respectively, $n^3/3$, n^2 and n^2. The Cholesky factorization has very high **arithmetic intensity** (i.e., the ratio between number of operations and number of memory accesses) and therefore can reach very high efficiency and lends itself very well to parallelization.

2 The Cholesky factorization by blocks

Many variants of the Cholesky factorization exist. In this exercise we will assume that the matrix is split in square blocks of size B as in Figure 1. The matrix has NB block-rows and block-columns and, therefore, its size is $N = B \times NB$.

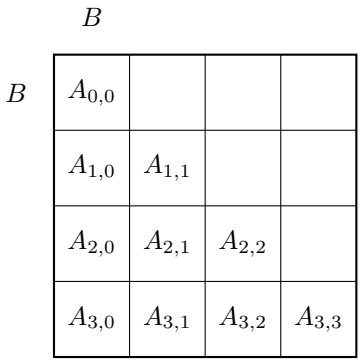


Figure 1: The system matrix split into blocks. We only show the lower half because the upper half is the same (transposed).

Based on this partitioning, the Cholesky factorization is computed as in Algorithm 1, where we assume that $A^{(0)}_{i,j} \equiv A_{i,j}$.
The code that implements the Cholesky factorization in Algorithm 1, is illustrated in Figure 2. Here we have assumed that `A` is a data structure where `A.NB` is the number of block-rows and block-columns and `A.blocks[i][j]` contains the block $A_{i,j}$. One distinctive feature of this algorithm is that it can be computed **in-place**; this means that it doesn't require any additional memory but, instead, the L matrix is stored in the same memory area that used to contain A.

Algorithm 1 Blocked QR factorization with Businger-Golub pivoting (QRCP).

for $k = 0, ..., NB - 1$ **do**
  $A_{k,k}^{(k)} \rightarrow L_{k,k} L_{k,k}^T$
  for $i = k + 1, ..., NB - 1$ **do**
    $L_{i,k} = A_{i,k}^{(k)} L_{k,k}^{-T}$
    for $j = k + 1, ..., i$ **do**
      $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - L_{i,k} L_{j,k}^T$
    end for
  end for
end for

```
for(k=0; k<A.NB; k++){
  /* reduce the diagonal block */
  potrf(A.blocks[k][k]);

  for(i=k+1; i<A.NB; i++){
    /* compute the A[i][k] sub-diagonal block */
    trsm(A.blocks[k][k], A.blocks[i][k]);

    for(j=k+1; j<=i; j++){
      /* update the A[i][j] block in the trailing submatrix */
      gemm(A.blocks[i][k], A.blocks[j][k], A.blocks[i][j]);
    }
  }
}
```

Figure 2: C code for computing the Cholesky factorization for a block-matrix.

The steps of this algorithm are depicted in Figure 3 only for the first iteration of the external loop (i.e. $k = 0$).
The routines in the algorithm above are defined as such:

- `potrf(A.blocks[k][k])`: this routine computes the reduction of a block, i,e, it computes $A_{k,k}^{(k)} \rightarrow L_{k,k} L_{k,k}^T$ which corresponds to the Cholesky factorization of a diagonal block. Note that, after this step, $L_{k,k}$ is stored in the same block that used to contain $A_{k,k}$.

- `trsm(A.blocks[k][k], A.blocks[i][k])`: this routine computes the $L_{i,k} = A_{i,k}^{(k)} L_{k,k}^{-T}$ block. Note that, after this step, $L_{i,k}$ is stored in the same block that used to contain $A_{i,k}$.

- `gemm(A.blocks[i][k], A.blocks[j][k], A.blocks[i][j])`: this routine computes $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - L_{i,k} L_{j,k}^T$. Note that, after this step $A_{i,j}^{(k+1)}$ is stored in the same memory that used to hold $A_{i,j}^{(k)}$ and, consequently $A_{i,j}$.

3 Package content

The package contains the following files:

- `chol_seq.c`: this file contains a sequential version of the Cholesky factorization by blocks. This is, essentially, the same as the code reported in Figure 2. This file should not be modified and only serves as a reference to compare with the three parallel versions to be developed.

- `chol_par_loop_simple.c`: this file has to be modified to achieve the first parallelization described in Part 1. At the beginning this file is an exact copy of the `chol_seq.c` file and the parallelization is obtained by adding OpenMP directives.

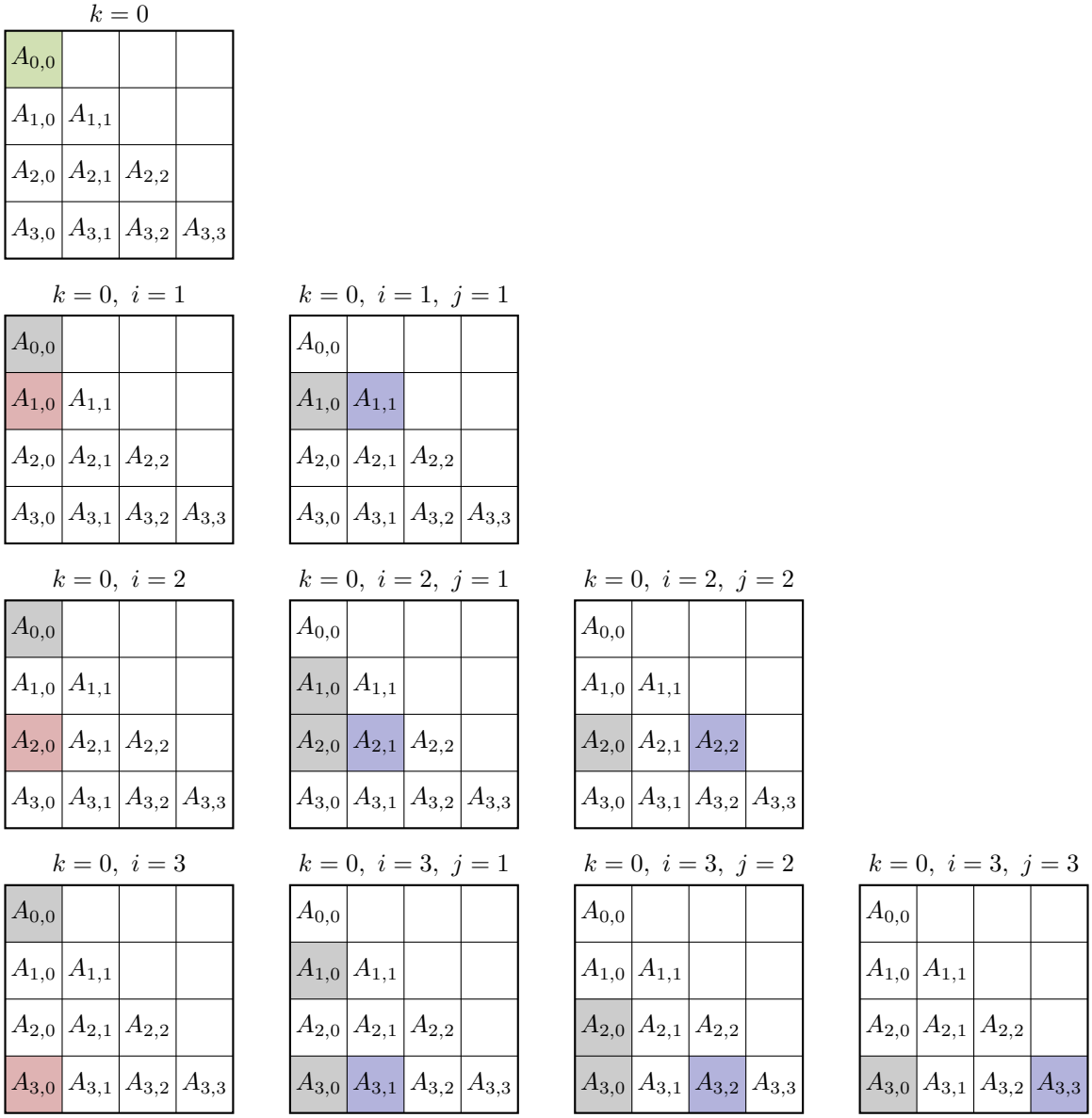


Figure 3: Operations in the the first iteration of the external loop of the Cholesky factorization (i.e. $k = 0$).

- `chol_par_loop_improved.c`: this file has to be modified to achieve the first parallelization described in Part 2. At the beginning this file is an exact copy of the `chol_seq.c` file and the parallelization is obtained by adding OpenMP directives.
- `chol_par_tasks.c`: this file has to be modified to achieve the first parallelization described in Part 3. At the beginning this file is an exact copy of the `chol_seq.c` file and the parallelization is obtained by adding OpenMP directives.
- `main.c`: this file contains a main program which creates and initializes the matrix and then calls the sequential and the three parallel versions of the factorization. For each of them the program also computed the execution time, the performance rate in Gflops/s (billions of floating-point operations per second) and checks the correctness of the factorization.
- `aux.c`, `auxf.f90`, `common.h`, `kernels.c`, `trace.c` and `trace.h`: this are auxiliary files and should not

be modified.

The main program can be compiled by typing the `make` command; this will generate an executable file `main` that can be run as such:

```
./main B NB
```

where `B` is the size of a block-column and `NB` is the number of blocks-rows and block-columns the matrix is made of. For verifying the correctness of your code, choose moderate values for B and NB (for example B=20 and NB=5). For analyzing the performance and scalability of your parallelization choose bigger values (for example, B=100 and NB=40). The number of threads can be controlled through the `OMP_NUM_THREADS` environment variable, like this

```
export OMP_NUM_THREADS=4
```

for setting the number of threads to 4 (for shells other than bash you should use `setenv OMP_NUM_THREADS 4`).
When executed, the code will produce the following output:

```
Matrix size: 1000

========== Sequential        (1 threads) ==========
Time (msec.) :    21.6
Gflop/s      :    15.4
||Ax-b||     : 1.0226e-17

========== Parallel loop     (1 threads) ==========
Time (msec.) :    19.1
Gflop/s      :    17.5
||Ax-b||     : 9.7849e-18

========== Parallel loop imp (1 threads) ==========
Time (msec.) :    19.1
Gflop/s      :    17.5
||Ax-b||     : 9.9557e-18

========== Parallel tasks    (1 threads) ==========
Time (msec.) :    25.4
Gflop/s      :    13.1
||Ax-b||     : 1.0233e-17
```

For each Cholesky variant, the code will print the execution time, the speed in Gflops/s (billions of floating-point operations per second) and the relative norm of the residual which measures the accuracy of the solution.
By compiling with the command `make main_dbg` instead, the resulting program will also print additional information showing the order in which panel and update operations are executed and which thread executed each of them. This can be very useful to verify that the operations are executed in the correct order.
Running the main program also generates trace files. A trace is an image showing which operations are executed by the threads in time as in Figure 4. Each row shows the operations executed by one thread in time; within each row, a rectangle shows an operation (green for `potrf`, red for `trsm` and blue for `gemm`) and its length is proportional to the operation execution time.

4 Part 1: simple loop parallelization

Can you identify which operations can be performed independently and in parallel? Based on this, modify the `chol_par_loop_simple.c` file to achieve a parallelization using the OpenMP

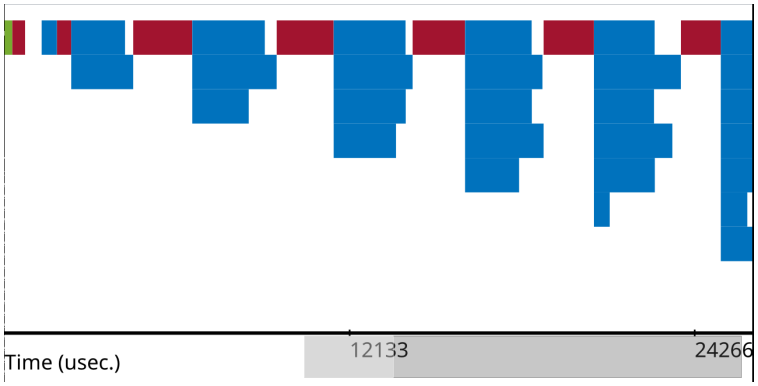


Figure 4: A part of an execution trace with 8 threads

`#pragma omp parallel for`
construct.
Compile the main program and run it. Verify the correctness (the printed residual should be smaller than 10^{-10}) and analyze the performance and scalability of your code using multiple threads. Is the parallel factorization faster than the sequential?
Open the trace file `trace_par_loop_simple.svg` of your parallel code with an image viewer and analyze it. Are all the threads working? is the work fairly distributed among the threads? can you identify inefficiencies?

5 Part 2: improved loop parallelization

5.1 Part 2.1: find more parallelism

The parallelization of the previous section can achieve very little performance and scalability because there is relatively little parallelism in the code.
Think of a way to restructure the code such that more parallelism is exposed. Add OpenMP parallelization directives accordingly in order to take advantage of the added parallelism.

5.2 Part 2.2: Reduce OpenMP overhead

Note that creating and destroying a parallel region has a cost and should be avoided within a loop. Modify your parallel code in order to create the parallel region only once and then execute multiple parallel loops in it. This can be done by splitting the `#pragma omp parallel for` into the two constructs `#pragma omp parallel` and `#pragma omp for` and placing these two in the right position. Pay special attention to the synchronization between threads. Compile the main program and run it.

6 Part 3: A complex, efficient DAG based parallelization

Analyzing the traces produced by the parallel code developed in Part 1 and 2, you should remark that there are empty gaps. White spaces in the traces mean that some threads are idle (i.e., not working) waiting for some event to happen. Therefore, white spaces represent inefficiencies and should be removed as much as possible. This can be achieved using an **asynchronous execution model** which eliminates (or reduces) useless synchronizations.
Can you draw the dependency graph for the block Cholesky factorization in Figure 2? For a matrix of size 4×4 blocks, it looks like the graph in Figure 5.
Looking at this graph of dependencies, is there any way we can remove some of the white gaps that appear in the execution traces of Part 1 and 2?

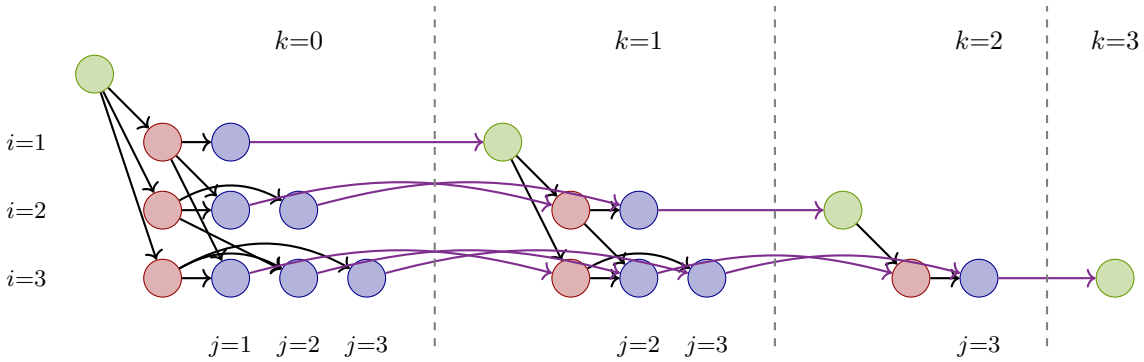


Figure 5: The dependency graph for a matrix of size 4×4 blocks.

6.1 Part 3.1: Task-based parallelization

In the file `chol_par_tasks.c` you will find a copy of the sequential version of the Cholesky algorithm. You have to make it parallel by conveniently adding OpenMP `task` directives. In order to let OpenMP construct the dependency graph, you have to use the `depend` clause.

6.2 Part 3.2: Improving the scheduling

One question that was left unanswered in the previous part is: in case there are multiple operations ready to be executed, which one should OpenMP choose?
Do you believe this is an important question? why? Think about the concept of *critical path* in the graph of dependencies. Because the graph of dependencies of the Cholesky factorization only has one entry point and one exit point, the critical path can simply be defined as the longest path connecting the entry and exit points. Can you identify the critical path in the graph of Figure 5? can you identify operations that always lie along the critical path? Do you believe that these operations should be executed with higher or lower priority?
Modify you code accordingly using the `priority` clause of the `task` directive. Before running the code make sure you set the `OMP_MAX_TASK_PRIORITY` to a suitably large value:

```
export OMP_MAX_TASK_PRIORITY=999
```

Is the resulting code faster? It should.

7 Experimenting on a supercomputer

We have the opportunity to run our code on the Turpan supercomputer installed at the CALMIP supercomputing center of Toulouse. Turpan is made of 15 computing nodes, each equipped with one Ampere Altra Q80-30 processor with 80 cores, two Nvidia A100 GPUs and 512 GB of memory. In our experiments we will only use the CPU cores.
First of all, you must upload your code on the Turpan computer. To do so, execute the following script

```
./upload.sh iritetu1
```

Replace `iritetu1` with the username that the supervisor has provided you and, when prompted, use the provided password. This will copy all your code on the Turpan computer in a directory named after your local username on the TP-rooms machines. For the rest of this document, let's assume your local username is `alcholesky`; if you want to know what your local username is, execute the `whoami` command.
Once your code has been copied, you are ready to connect to Turpan. To do so use the following command:

```
ssh iritetu1@turpanlogin.calmip.univ-toulouse.fr
```

where `iritetu1` has to be replaced with the username provided by the room supervisor. When prompted, use the password associated with the provided username.
Once you are connected on the machine, go to the directory where your code is, initialize your environment and compile the benchmarking programs; to do so, execute the following commands

```
cd alcholesky/TP_Chol
source env_cpuonly.sh
make ARCH=arm bench_weak bench_strong
```

Make sure you replace `alcholesky` with your local username.
Once the code is compiled, you are ready to submit your job to the queueing system. To do so, run the following command

```
sbatch --reservation=tpn7insa dosub.sh
```

Now your job should be in the queue waiting to be executed. You can check this executing the `squeue` command which will show all the jobs in the queue and their status.
The job will execute two benchmarks:

1. **strong scalability**: this will measure the performance of the Cholesky factorization variants for a matrix of fixed size increasing the number of cores from 1 to 80.
2. **weak scalability**: this will measure the performance of the Cholesky factorization increasing the number of cores from 1 to 80 but also increasing the size of the matrix in such a way that the number of operations per core stays constant.

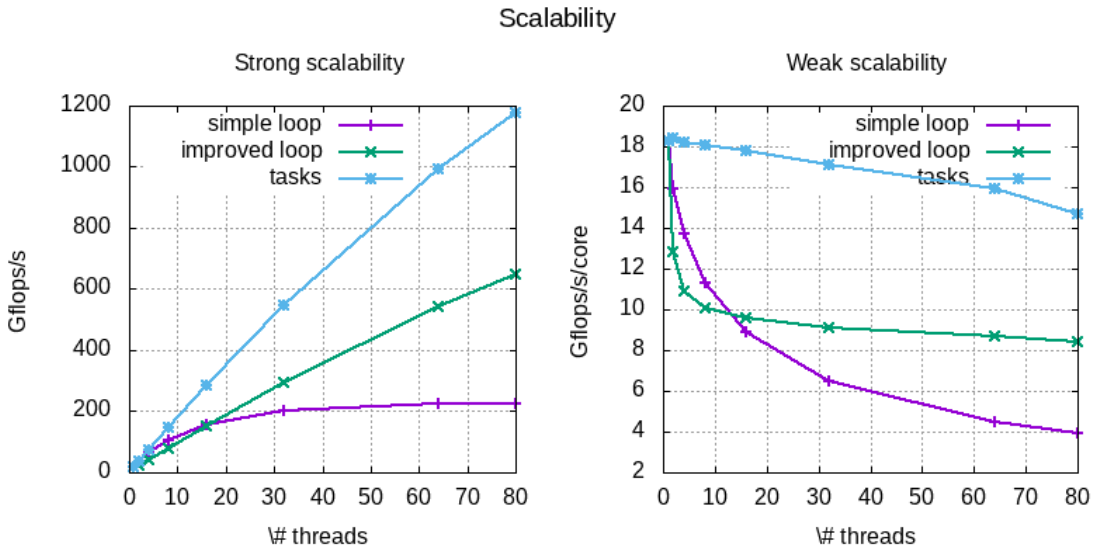
When your job is completed, it will disappear from the queue and the two files `strong_scalability.csv` and `weak_scalability.csv` will be created. You can check their content to make sure everything worked correctly; you should see something like this

```
 4000,    1,   1133.8,      18.8,    1135.6,      18.8,    1132.7,      18.8
 5200,    2,   1427.4,      16.4,    1833.3,      12.8,    1252.3,      18.7
 6400,    4,   1573.1,      13.9,    1995.0,      10.9,    1183.7,      18.5
 8000,    8,   1865.9,      11.4,    2109.3,      10.1,    1168.2,      18.3
10200,   16,   2461.2,       9.0,    2298.0,       9.6,    1230.9,      18.0
12800,   32,   3332.0,       6.6,    2374.6,       9.2,    1270.0,      17.2
16000,   64,   4742.9,       4.5,    2453.2,       8.7,    1339.0,      15.9
17400,   80,   5572.6,       3.9,    2593.3,       8.5,    1453.3,      15.1
```

Now we can log out of Turpan and go back to the TP-room machine; to do so hit `ctrl+d`. Then, download the result of your experiment executing the following command

```
./download.sh iritetu1
```

this will create a figure named `scalability_plot.png` that should look pretty much like this:



For the three developed variants, the left plot shows the overall performance in Gflops/s for the strong scalability case and the right plot shows the performance per core in Gflops/s/core for the weak scaling case. What do you observe in these plots? what is the ideal behavior for both cases?