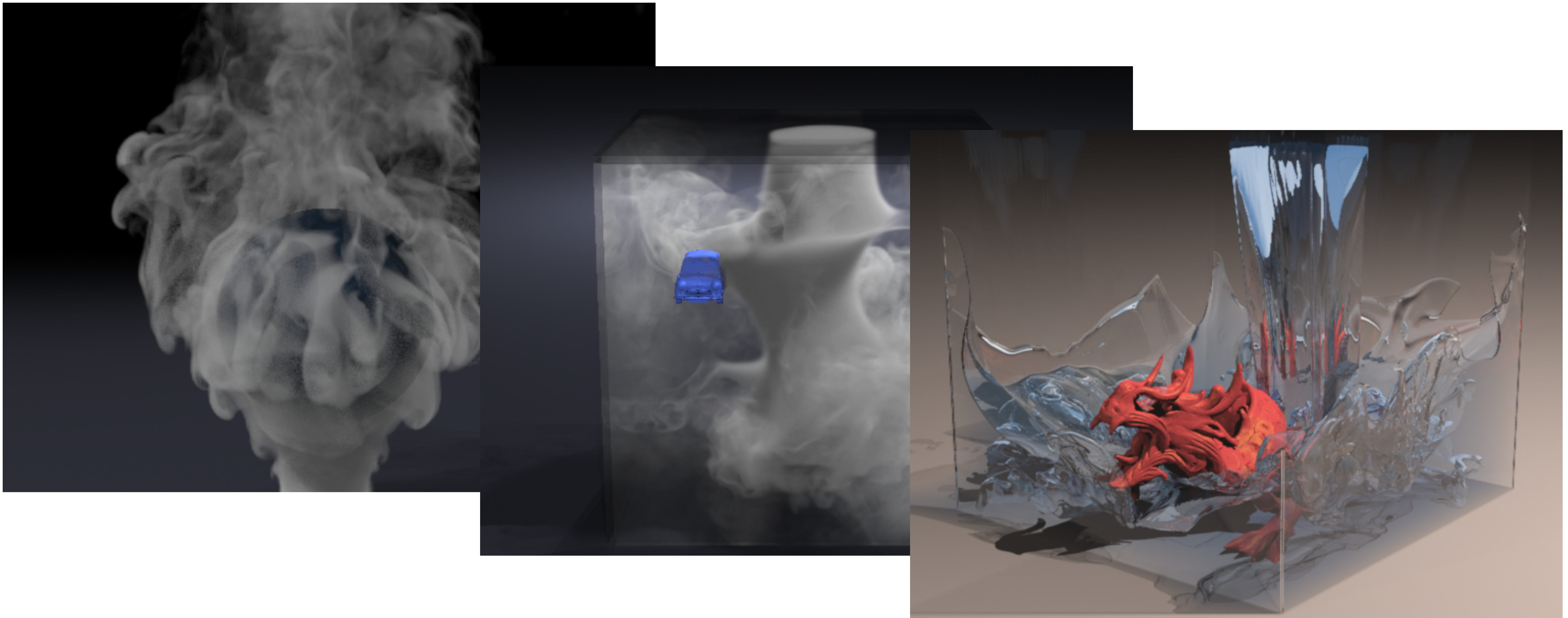


Parallel Sparse Direct Solvers
Performance & design survey of MKL PARDISO

Recap

We did a walk-through of PARDISO, a solver library within Intel MKL.
*PARDISO facilitates the solution of linear systems **Ax=b** for which:*

*- The coefficient matrix **A** is sparse (as opposed to LAPACK and many BLAS Level 3 routines that operate on dense matrices)*

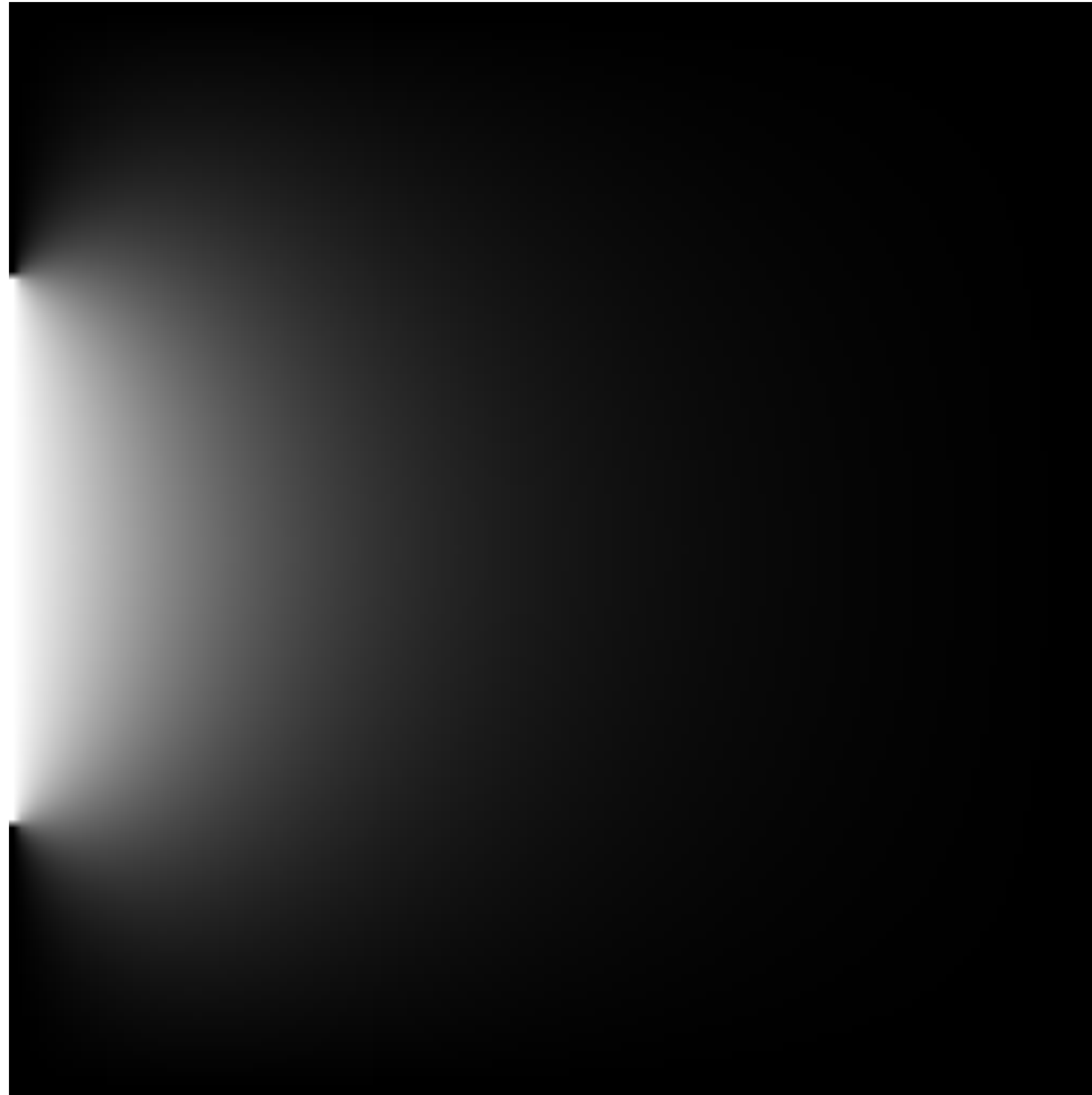
*- The solver works for several different types of matrices, but is particularly efficient for symmetric (and, ideally, positive definite) matrices for which a factorization of **A** is computed once (the "Cholesky" decomposition, when applicable) and re-used at low-cost for solving with different right-hand-sides*

- The solver is "direct", in that it computes the entire solution without the need for iteration

PARDISO operates on CSR-encoded matrices - same as we used before (but when used with symmetric matrices expects to be given just "half" matrix)

Result of direct solver

SparseDirect/LaplacePARDISO_0_0



A deeper look - Solver stages

PARDISO Phase 1 : Reorder the matrix to generate favorable properties
No numerical operations done in this stage - values of matrix entries don’t matter, the only thing that matters is the sparsity pattern
(we’ll see what those “favorable properties” are)

PARDISO solver (DirectSolver.cpp)

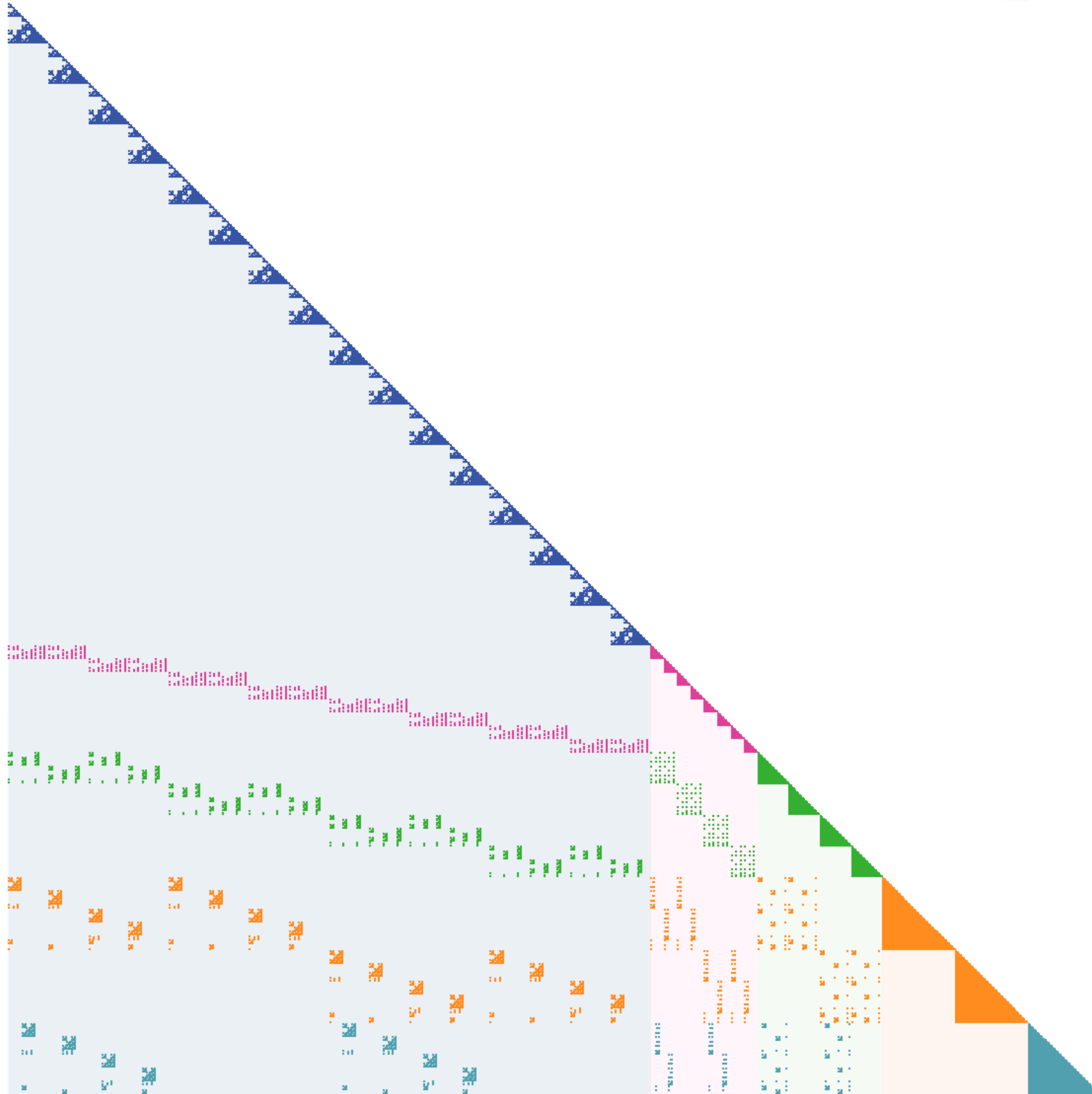
SparseDirect/LaplacePARDISO_0_0

```
}

// Reordering and Symbolic Factorization. This step also allocates
// all memory that is necessary for the factorization
phase = 11;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during symbolic factorization");

std::cout << "Reordering completed ... " << std::endl;
std::cout << "Number of nonzeros in factors = " <<  iparm[17] << std::endl;
std::cout << "Number of factorization MFLOPS = " << iparm[18] << std::endl;

// Numerical factorization
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during numerical factorization");

std::cout << "Factorization completed ... " << std::endl;

// Back substitution and iterative refinement
phase = 33;
iparm[7] = 0;         // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

Laplacian - Initial equation ordering

$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
& 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
& & \ddots & \ddots & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
& 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
& & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
& & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
& \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
& & \ddots & & & & 1 & & & 1 & -6 & 1 \\
& & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$

Laplacian - Pattern after a possible reordering



SparseDirect/LaplacePARDISO_0_0

Execution:

```
Summary: ( reordering phase )
================
Times:
======
Time spent in calculations of symmetric matrix portrait (fulladj): 0.046880 s
Time spent in reordering of the initial matrix (reorder)         : 1.529101 s
Time spent in symbolic factorization (symbfct)                   : 2.171409 s
Time spent in data preparations for factorization (parlist)      : 0.202028 s
Time spent in allocation of internal data structures (malloc)    : 0.498570 s
Time spent in additional calculations                            : 0.455895 s
Total time spent                                                 : 4.903884 s
Statistics:
===========
Parallel Direct Factorization is running on 20 OpenMP
< Linear system Ax = b >
             number of equations:           2097152
             number of non-zeros in A:      8050652
             number of non-zeros in A (%): 0.00018
             number of right-hand sides:    1
< Factors L and U >
             number of columns for each panel: 96
             number of independent subgraphs:  0
             number of supernodes:                    1409897
             size of largest supernode:               16591
             number of non-zeros in L:                2065304266
             number of non-zeros in U:                1
             number of non-zeros in L+U:              2065304267
Reordering completed ...
Number of nonzeros in factors = 2065304267
Number of factorization MFLOPS = 22854214
```

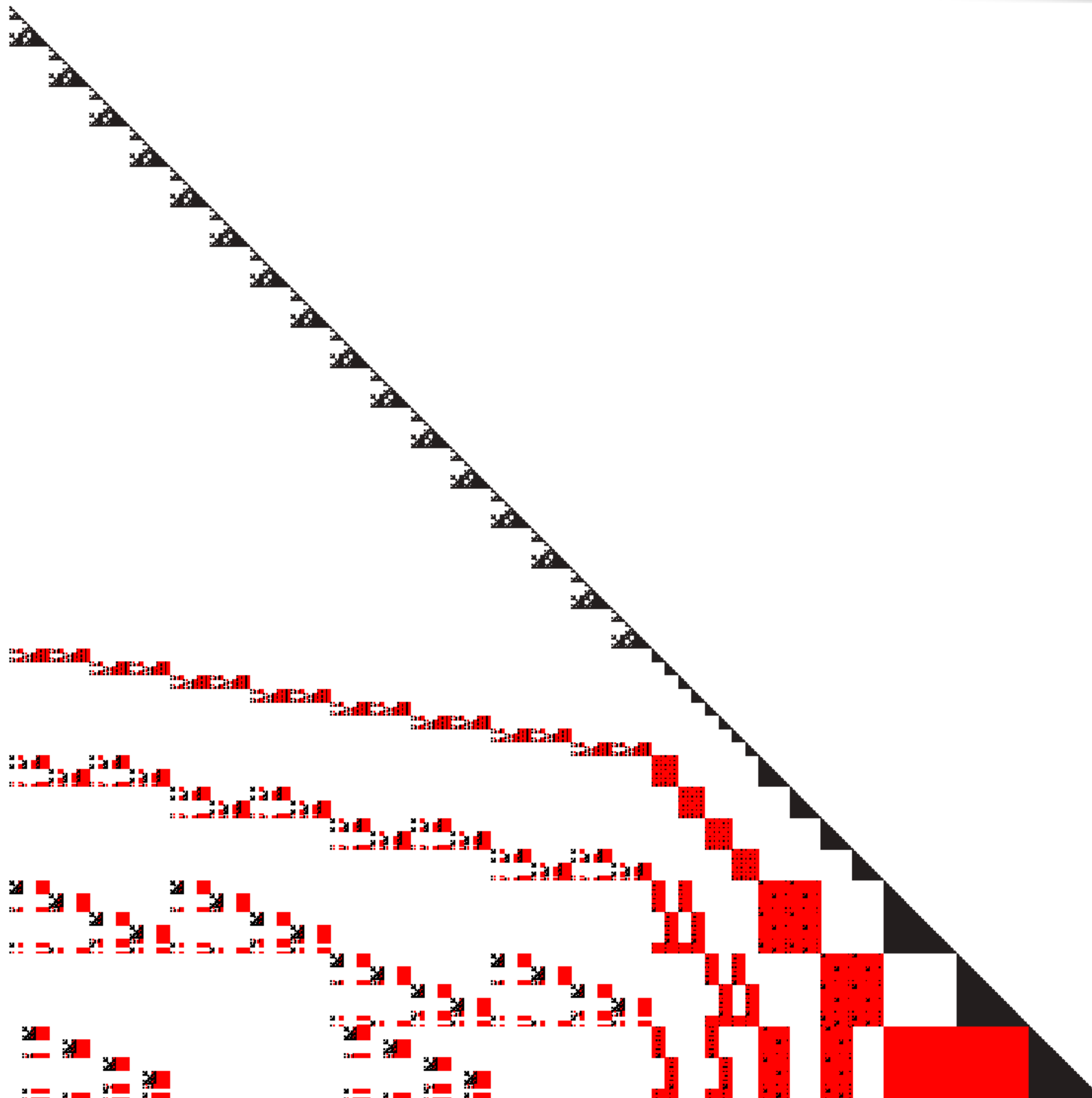
About 10% of overall runtime (typically: at least that much)

A deeper look - Solver stages

PARDISO Phase 1 : Reorder the matrix to generate favorable properties
No numerical operations done in this stage - values of matrix entries don't matter, the only thing that matters is the sparsity pattern
(we'll see what those "favorable properties" are)

PARDISO Phase 2 : Perform the actual Cholesky Decomposition (factorization)
This is the computation-heavy part of the algorithm, and the most expensive part of the execution, for typical (large) matrix sizes.
*Note: In accordance with theory, the Cholesky factor **L** includes all of the entries in the sparsity pattern of **A** in its own, plus some more*
(hopefully as few as possible; reordering influences that)

Sparsity of Cholesky Factor (L) vs. Laplacian Matrix



SparseDirect/LaplacePARDISO_0_0

PARDISO solver (DirectSolver.cpp)

Execution:

```
Summary: ( factorization phase )
================
Times:
======
Time spent in copying matrix to internal data structure (A to LU): 0.000000 s
Time spent in factorization step (numfct)                        : 44.352600 s
Time spent in allocation of internal data structures (malloc)    : 0.022322 s
Time spent in additional calculations                            : 0.000002 s
Total time spent                                                 : 44.374928 s
Statistics:
===========
Parallel Direct Factorization is running on 20 OpenMP
< Linear system Ax = b >
             number of equations:           2097152
             number of non-zeros in A:      8050652
             number of non-zeros in A (%): 0.000183
             number of right-hand sides:    1
< Factors L and U >
             number of columns for each panel: 96
             number of independent subgraphs:  0
             number of supernodes:                    1410153
             size of largest supernode:               16591
             number of non-zeros in L:                2057589566
             number of non-zeros in U:                1
             number of non-zeros in L+U:              2057589567
             gflop   for the numerical factorization: 22775.748047
             gflop/s for the numerical factorization: 513.515503

Factorization completed ...
 PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

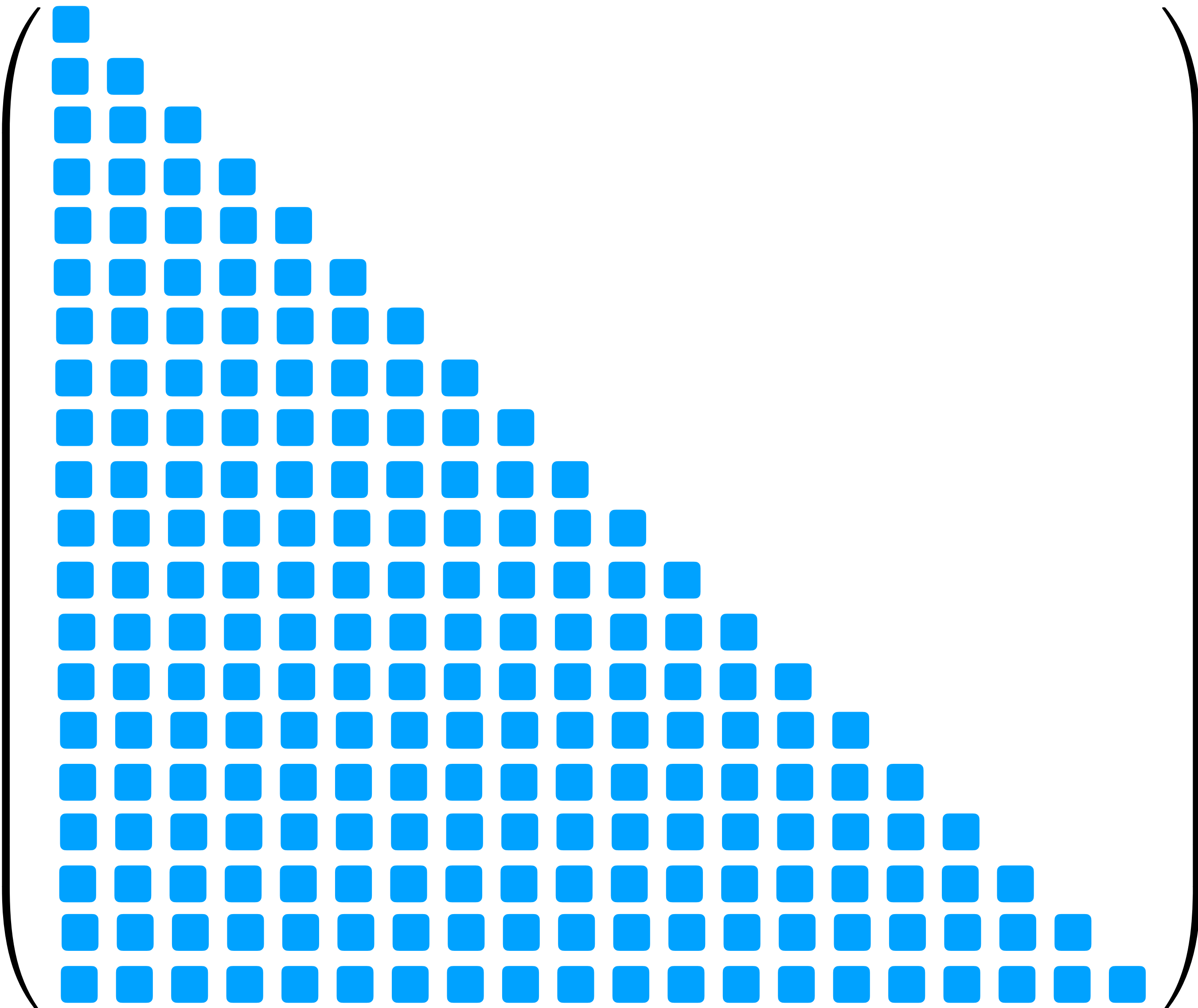
About 90% of overall runtime (sometimes less)

Almost 25% of peak arithmetic utilization

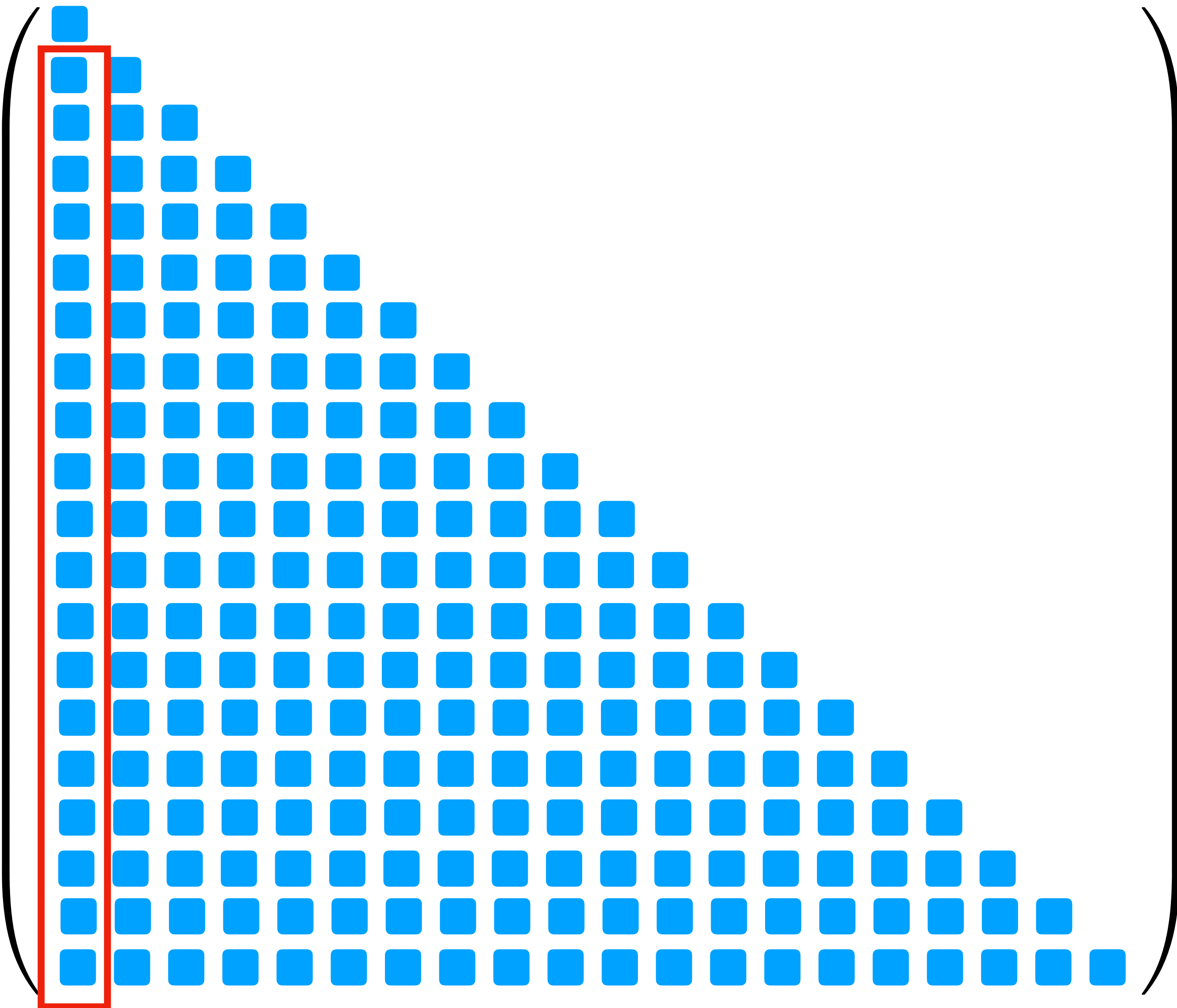
Obstacles to performance & parallelism

*Matrix Density : The number of required operations scale (super-linearly …) with the number of non-zero entries in **L** … thus, ensuring sparser **L** factors has an immediate effect on performance*

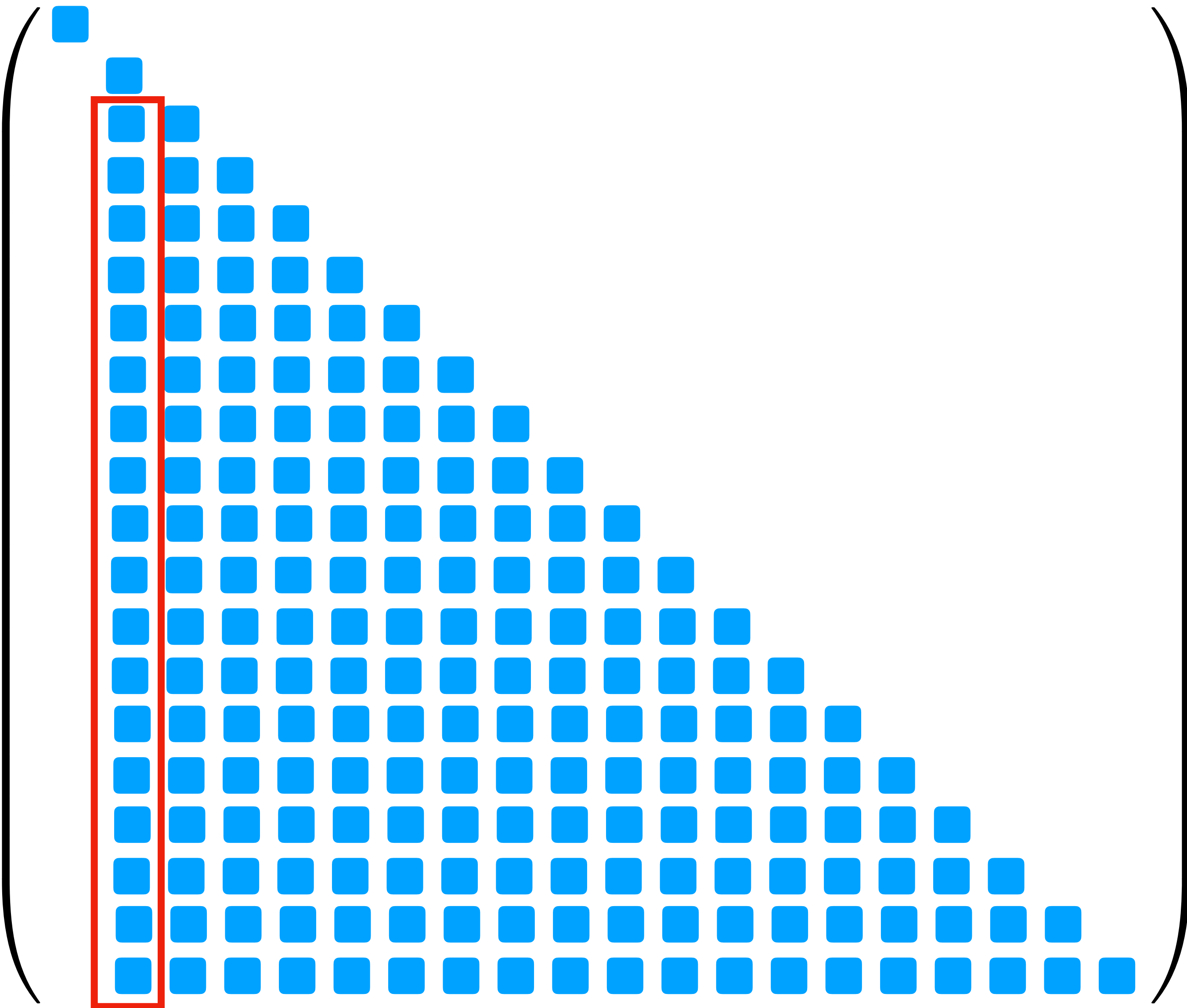
Multithreading : Cholesky, similar to Gauss Elimination, is seemingly a very "serial" algorithm (significant dependencies between steps/loops). We must find some way to cope with this apparent limitation.



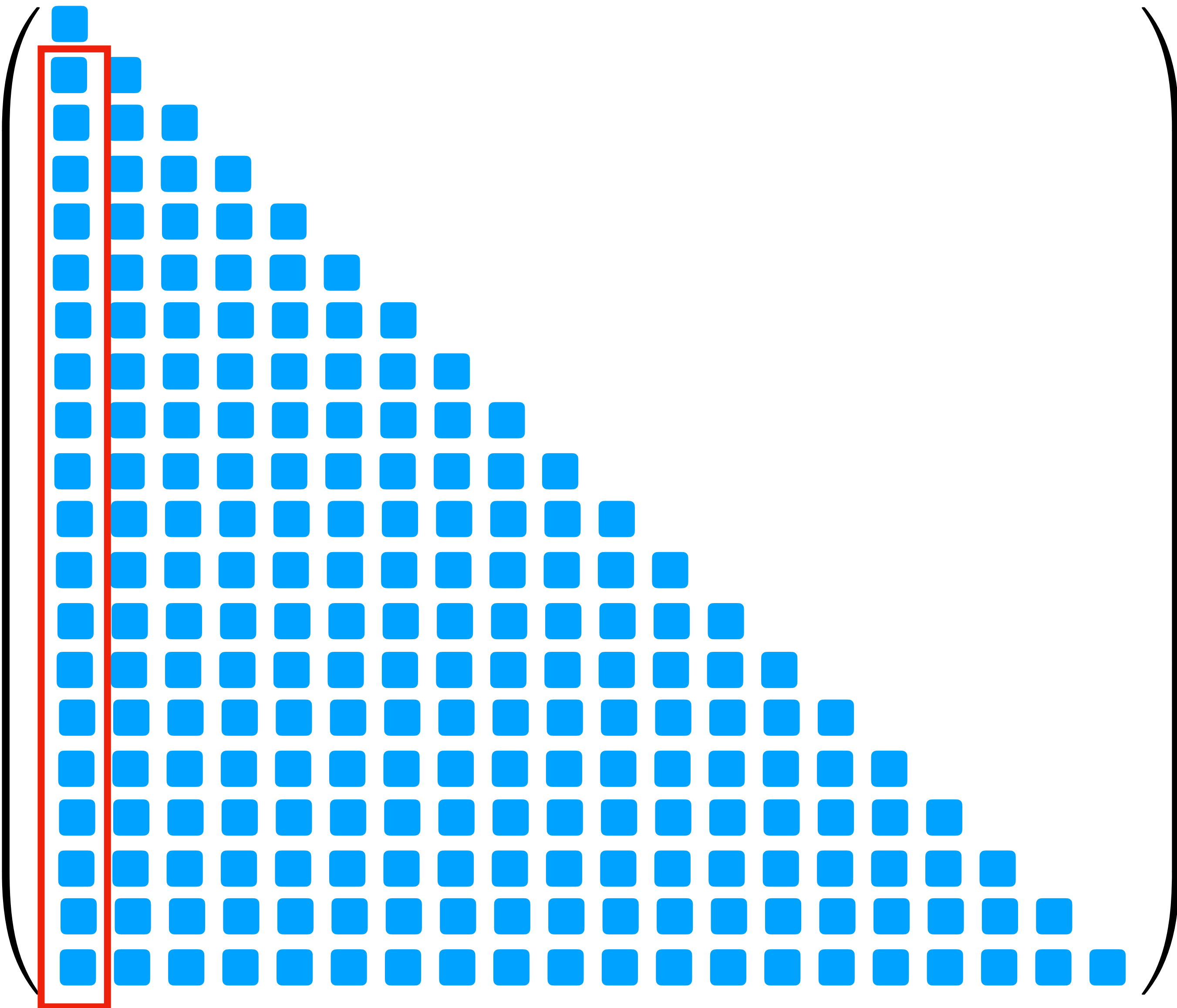
Consider Gauss Elimination …



We need to make all these entries zero …



… and then continue to the next column



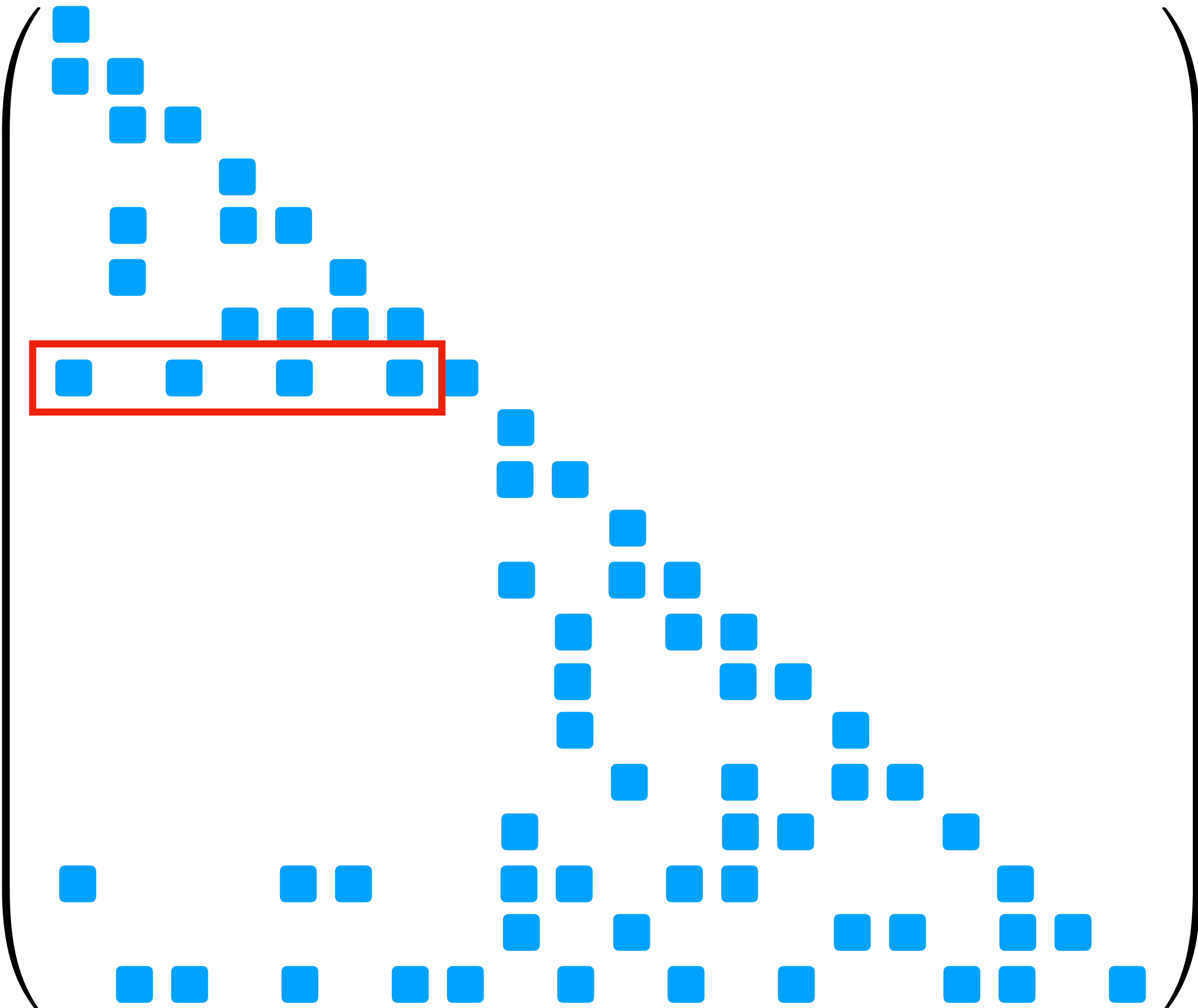
We can do each row of this operation in parallel … but we need to wait for this column before moving to the next (… in principle)

Obstacles to performance & parallelism

*Matrix Density : The number of required operations scale (super-linearly …) with the number of non-zero entries in **L** … thus, ensuring sparser **L** factors has an immediate effect on performance*

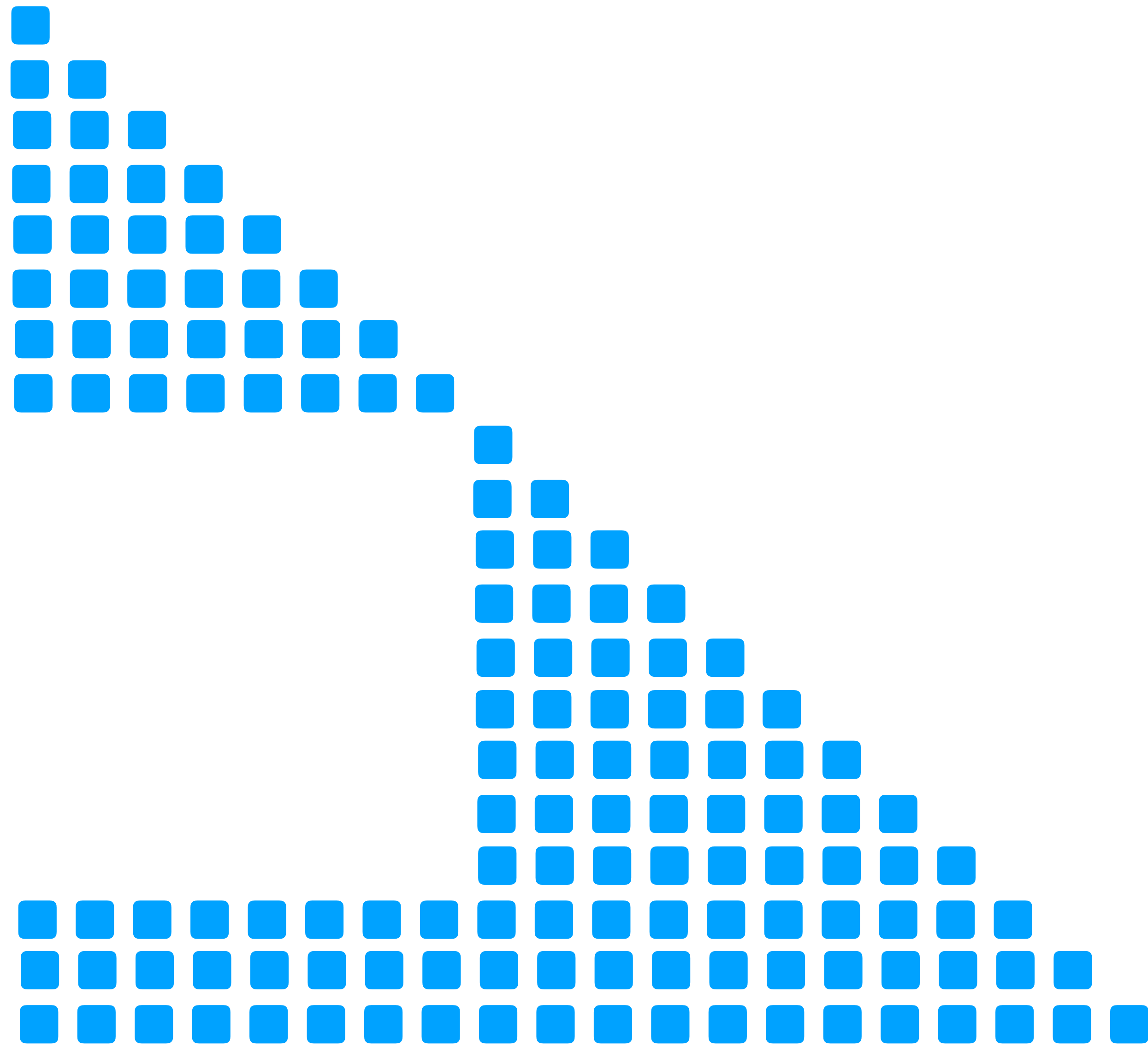
Multithreading : Cholesky, similar to Gauss Elimination, is seemingly a very "serial" algorithm (significant dependencies between steps/loops). We must find some way to cope with this apparent limitation.

Vectorization/SIMD : Sparse matrices don't have the regularity that SIMD operations require; we need to "engineer" such regularity if possible



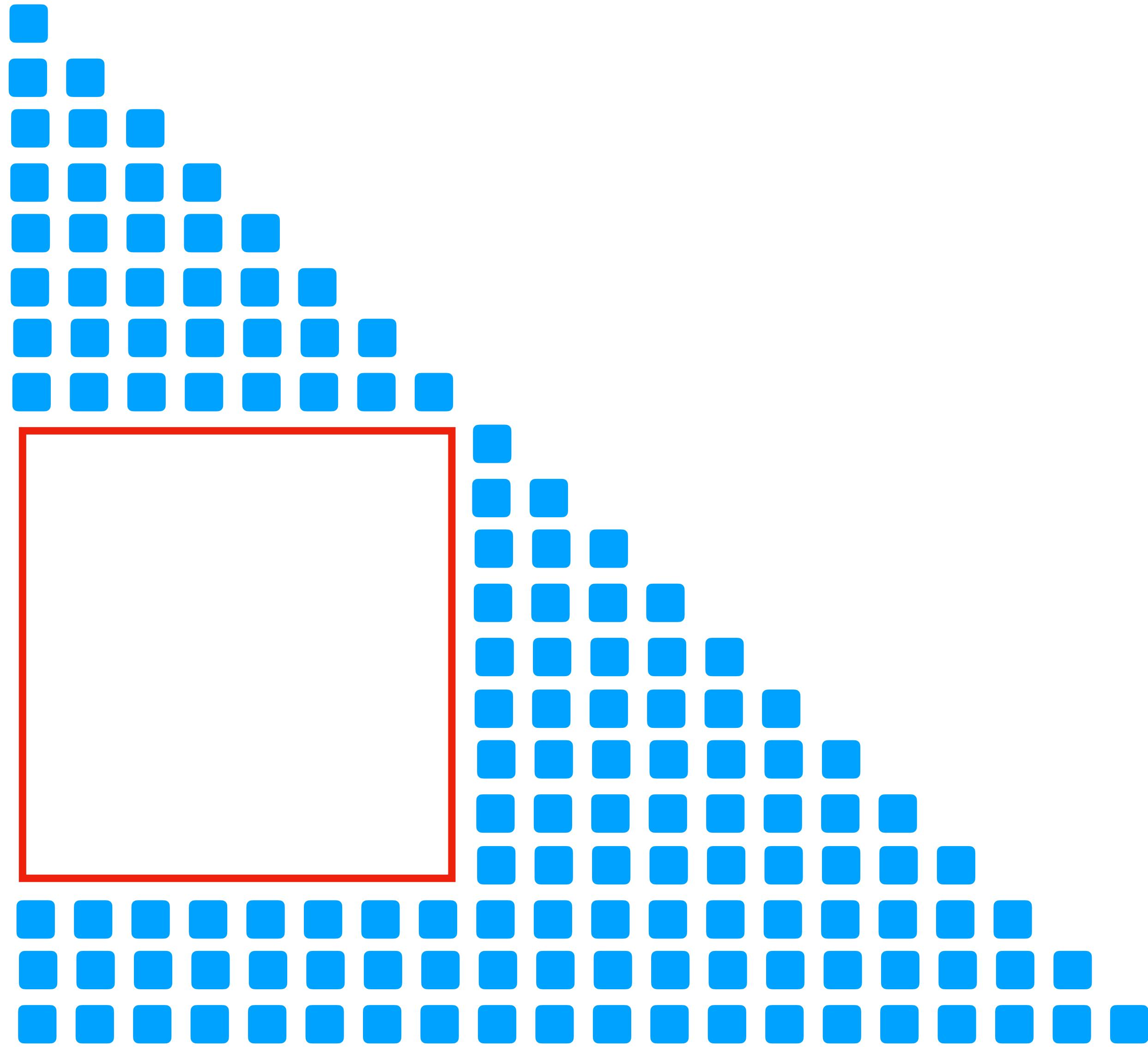
Tasks that we would normally consider candidates for SIMD are not at all regular …

Engineering/Maximizing Sparsity



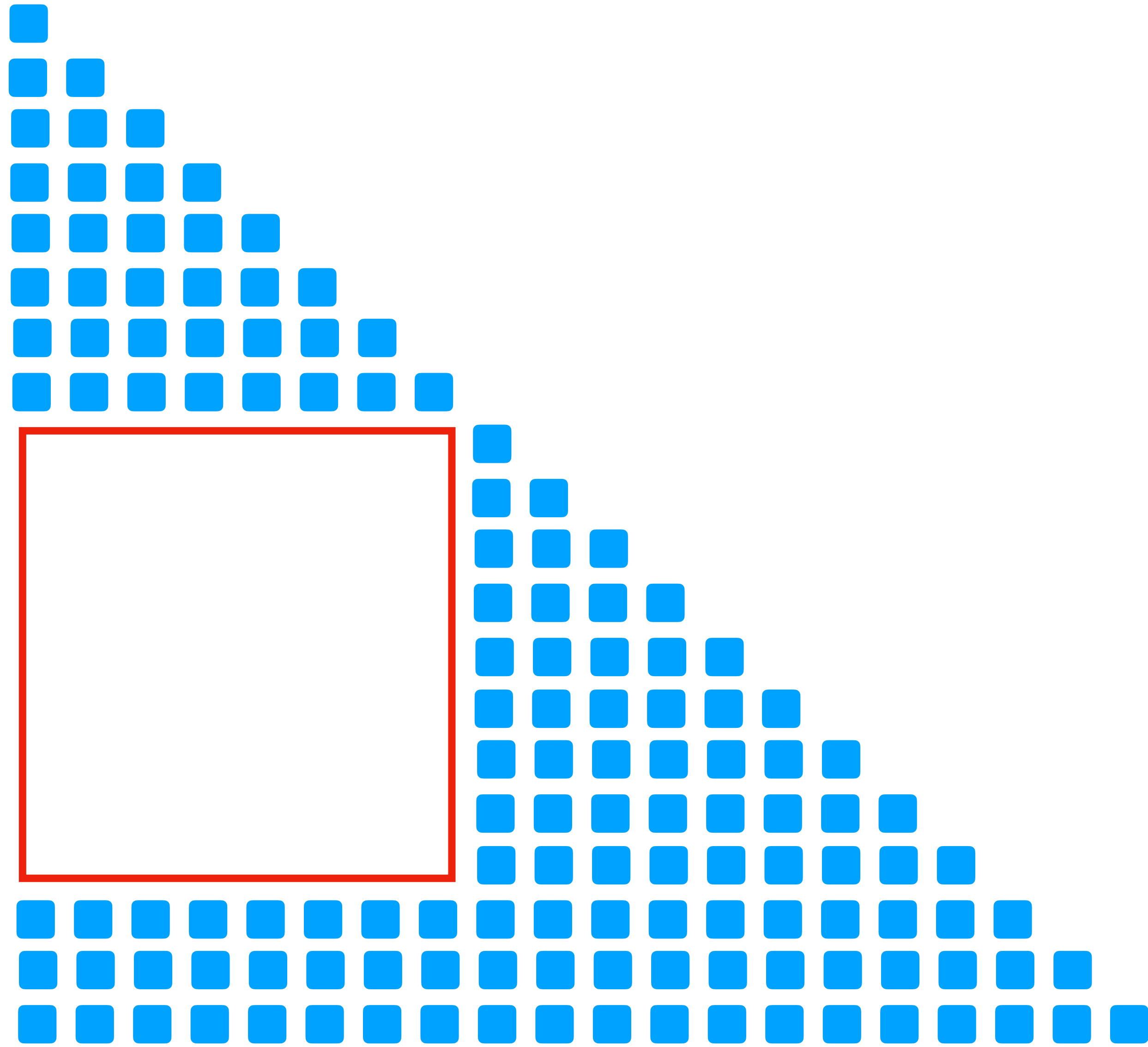
*Sparsity pattern of **A***
(lower-triangular part only)

Engineering/Maximizing Sparsity



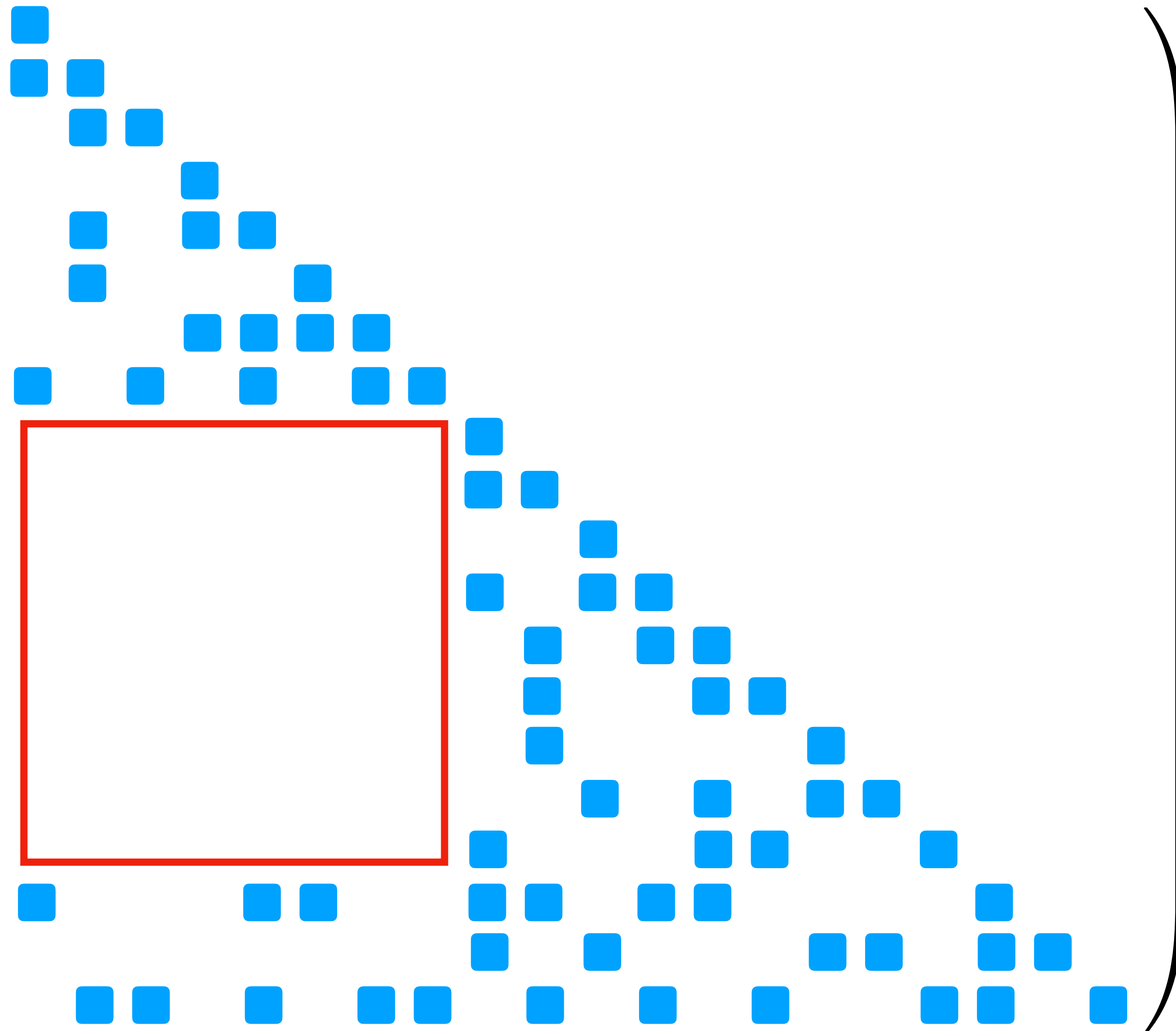
Theory can prove that:
*If there's a rectangular gap in the sparsity pattern of **A** …*

Engineering/Maximizing Sparsity



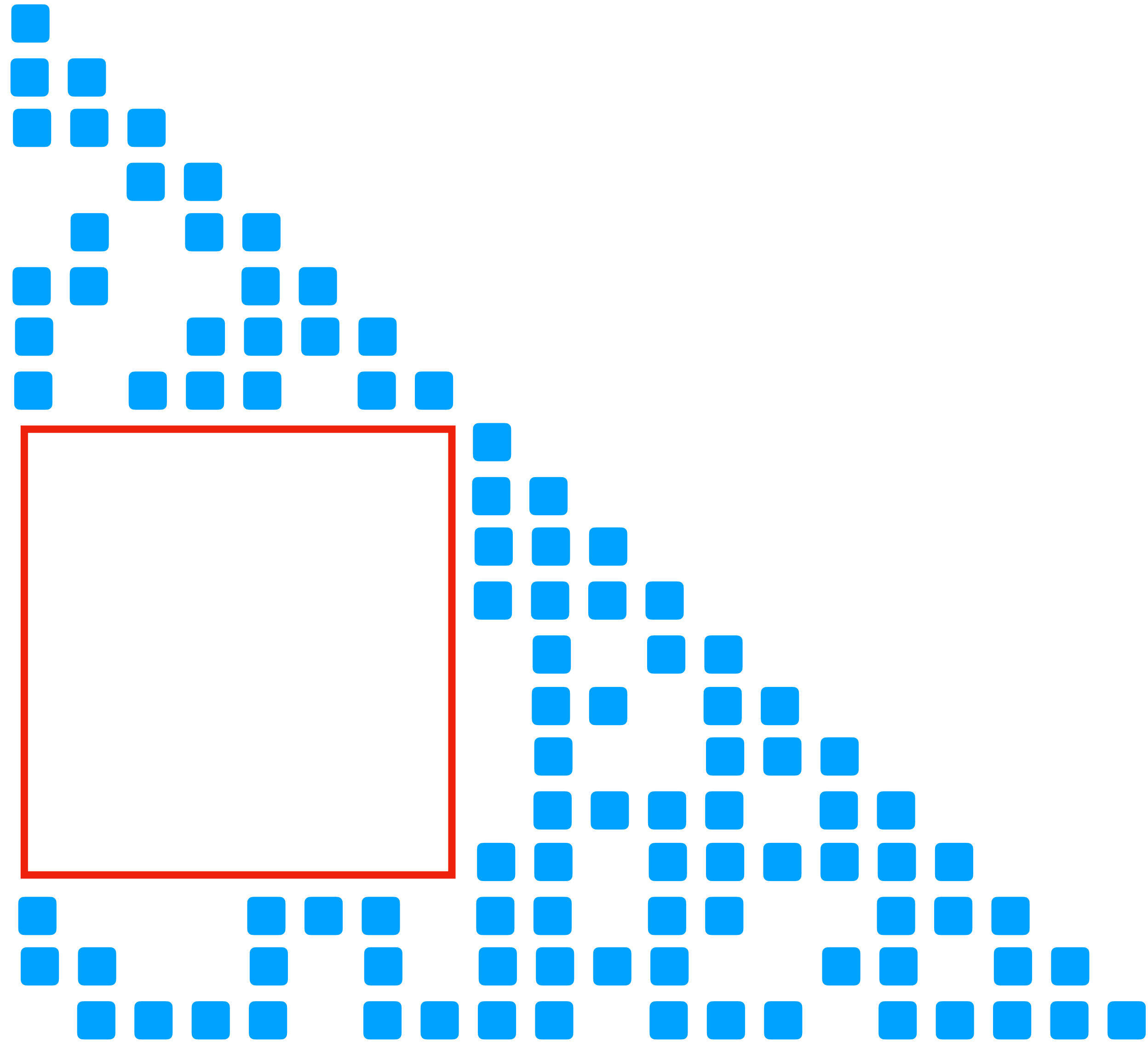
Theory can prove that:
… that gap will also be present
*in the Cholesky factor **L***

Engineering/Maximizing Sparsity

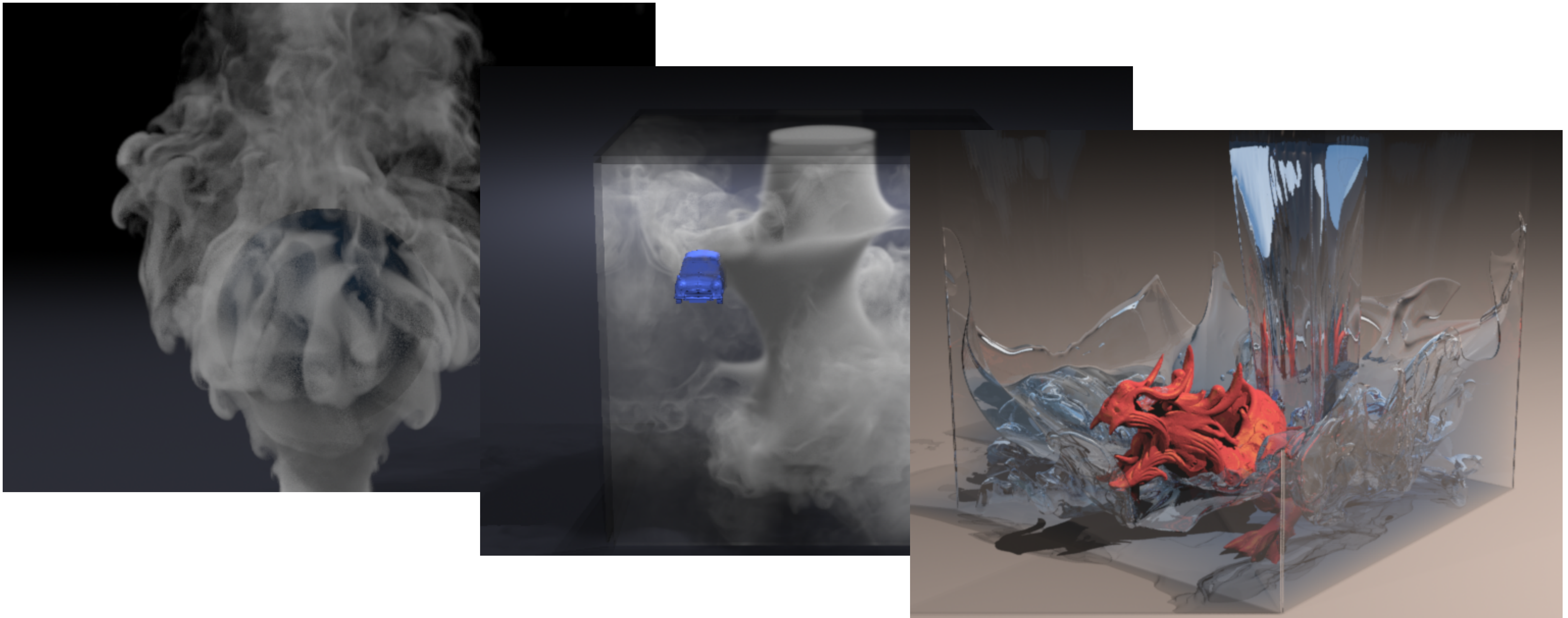


*A sparse matrix **A** can have such gaps without being "dense" elsewhere …*

Engineering/Maximizing Sparsity

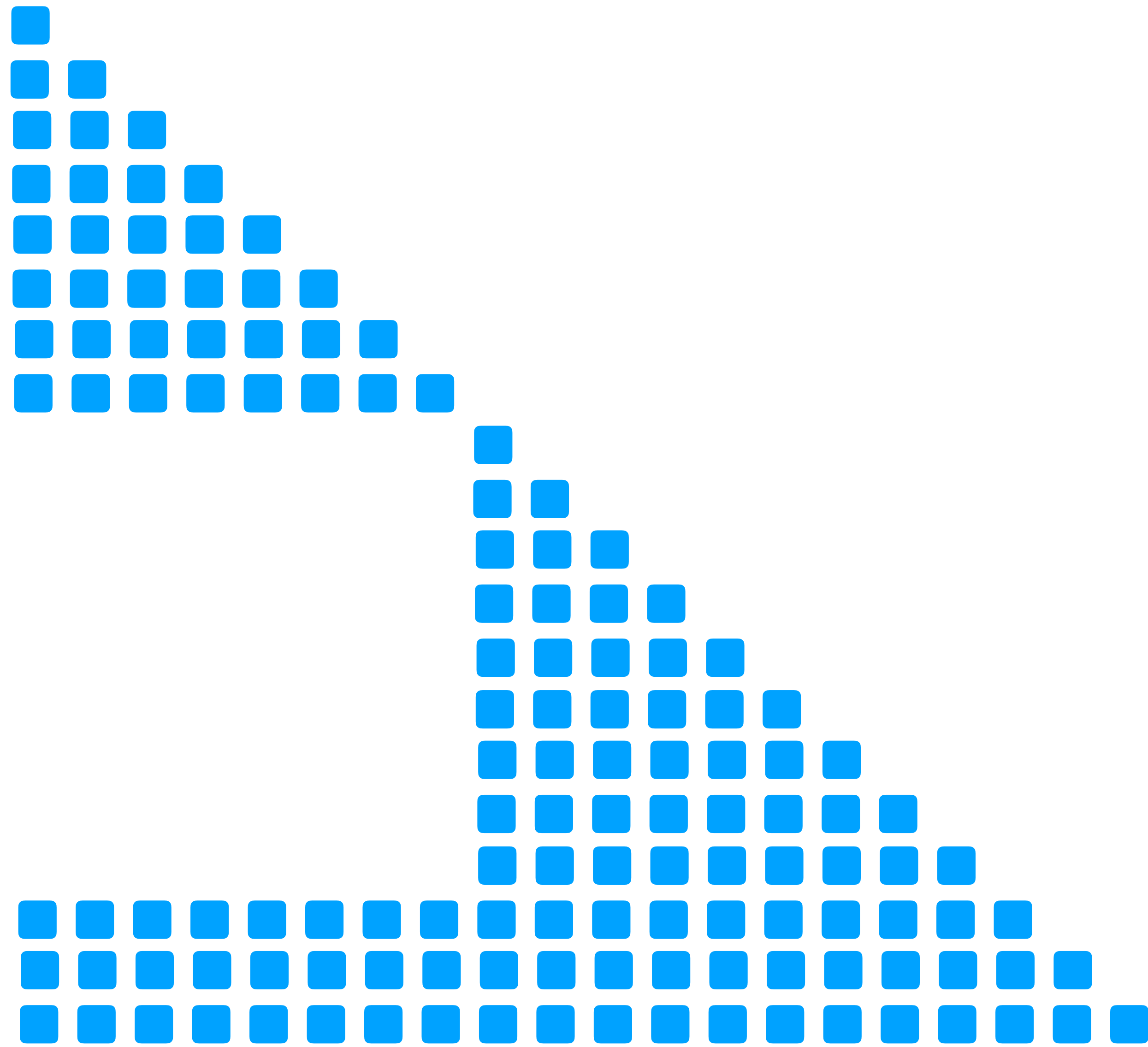


*… and the corresponding factor **L** (even if it becomes denser away from such gaps) does retain these "holes" in its sparsity pattern*



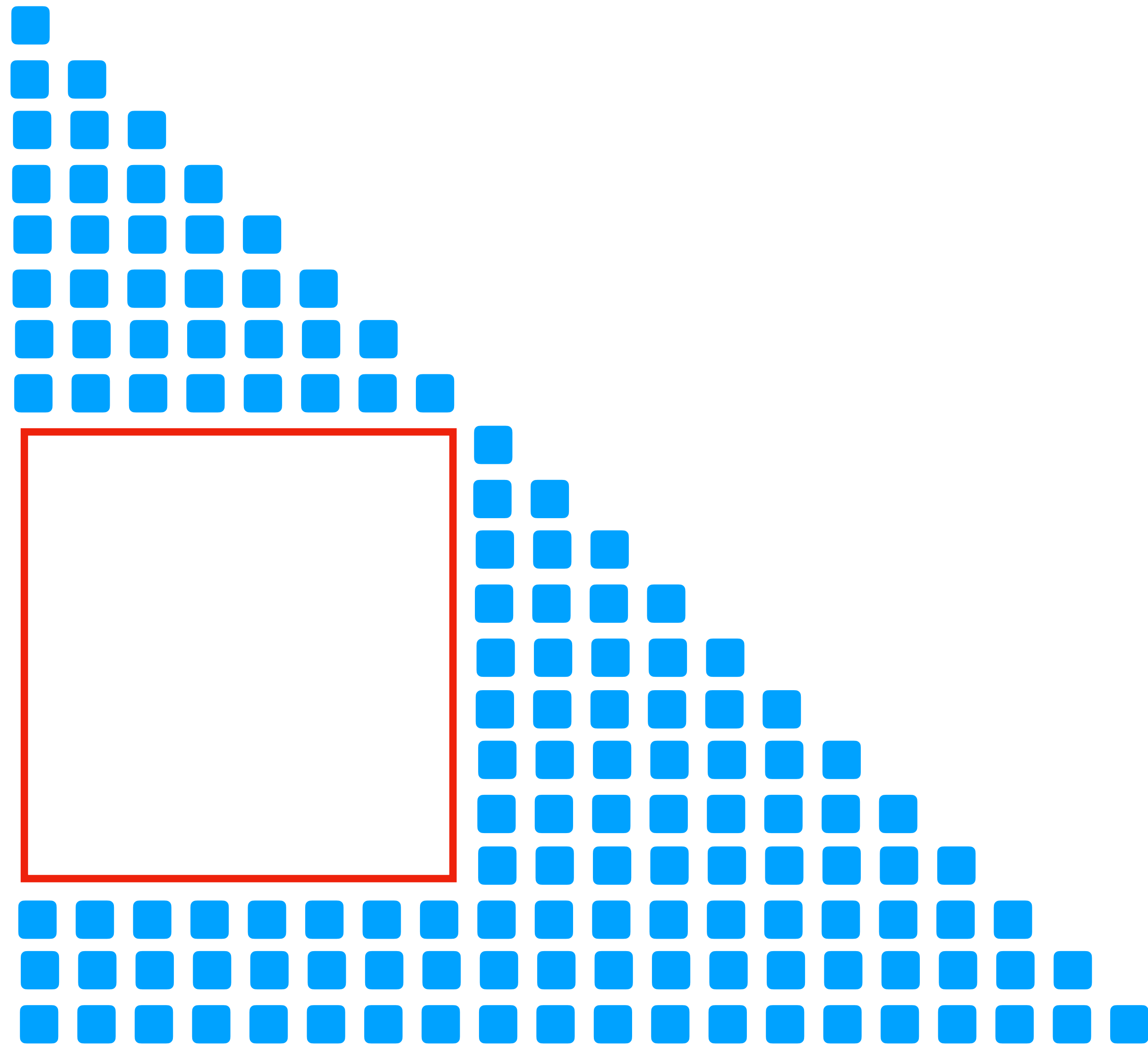
Parallel Sparse Direct Solvers
Performance & design survey of MKL PARDISO

Engineering/Maximizing Sparsity



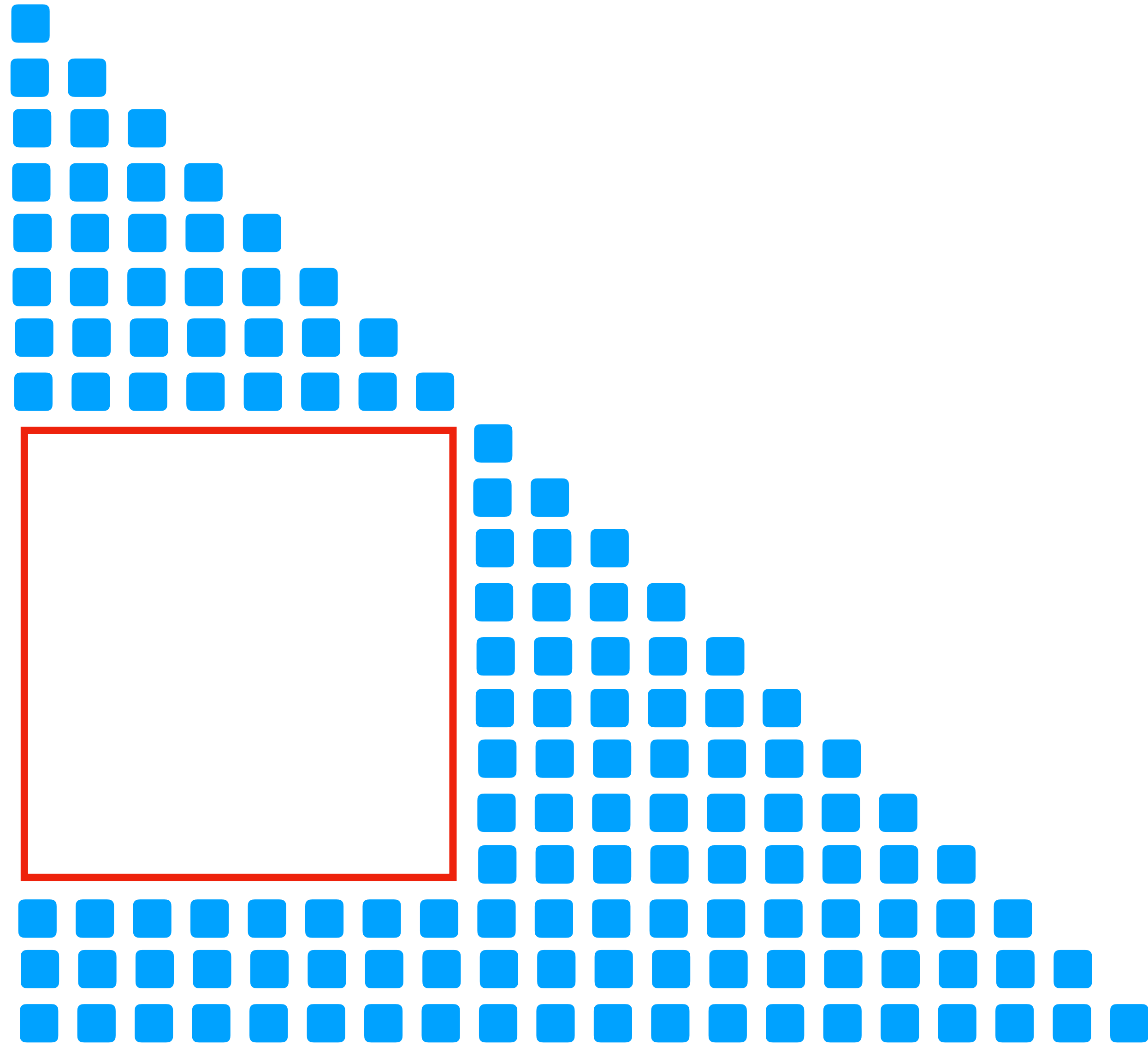
*Sparsity pattern of **A***
(lower-triangular part only)

Engineering/Maximizing Sparsity



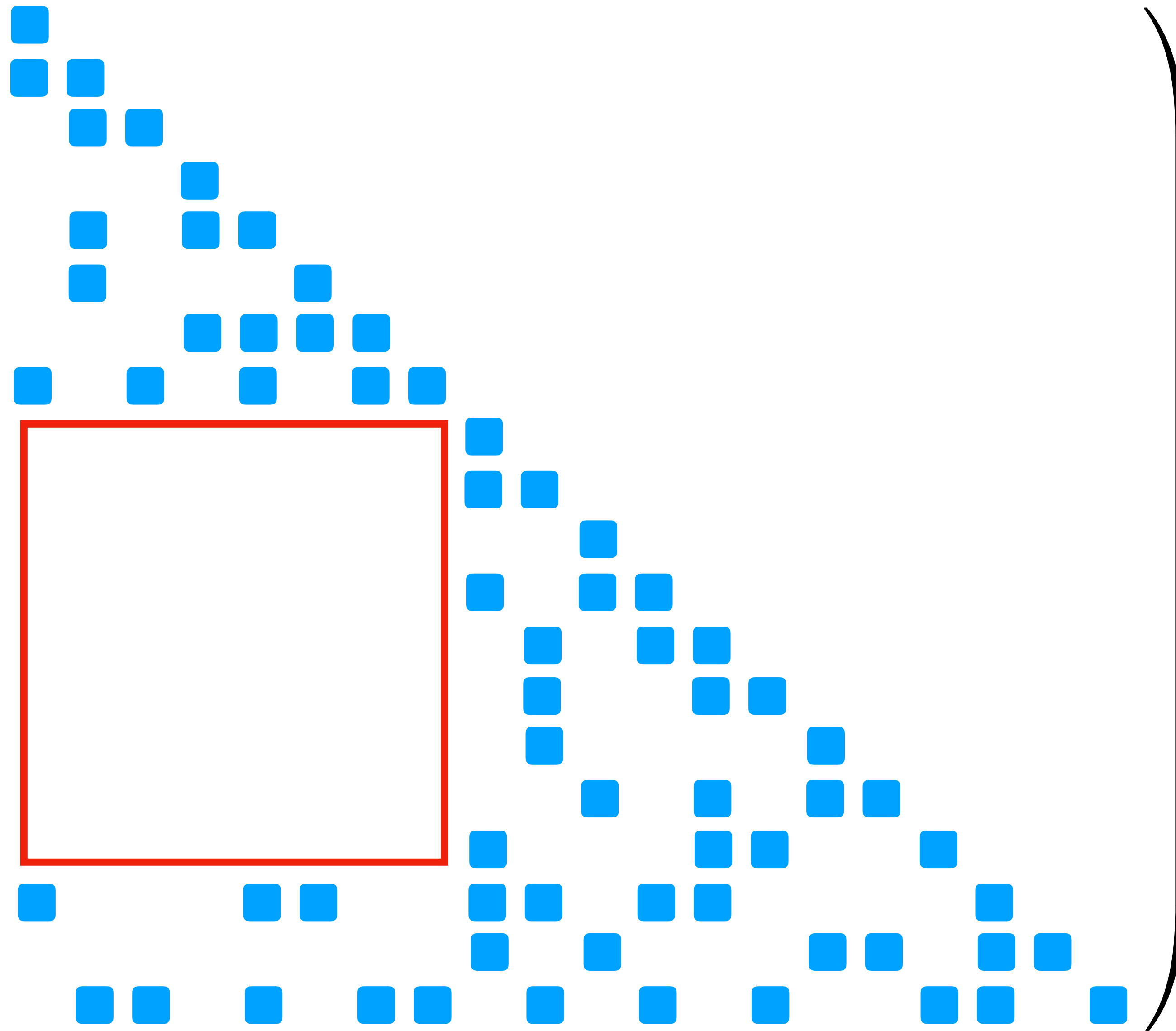
Theory can prove that:
*If there's a rectangular gap in the sparsity pattern of **A** …*

Engineering/Maximizing Sparsity



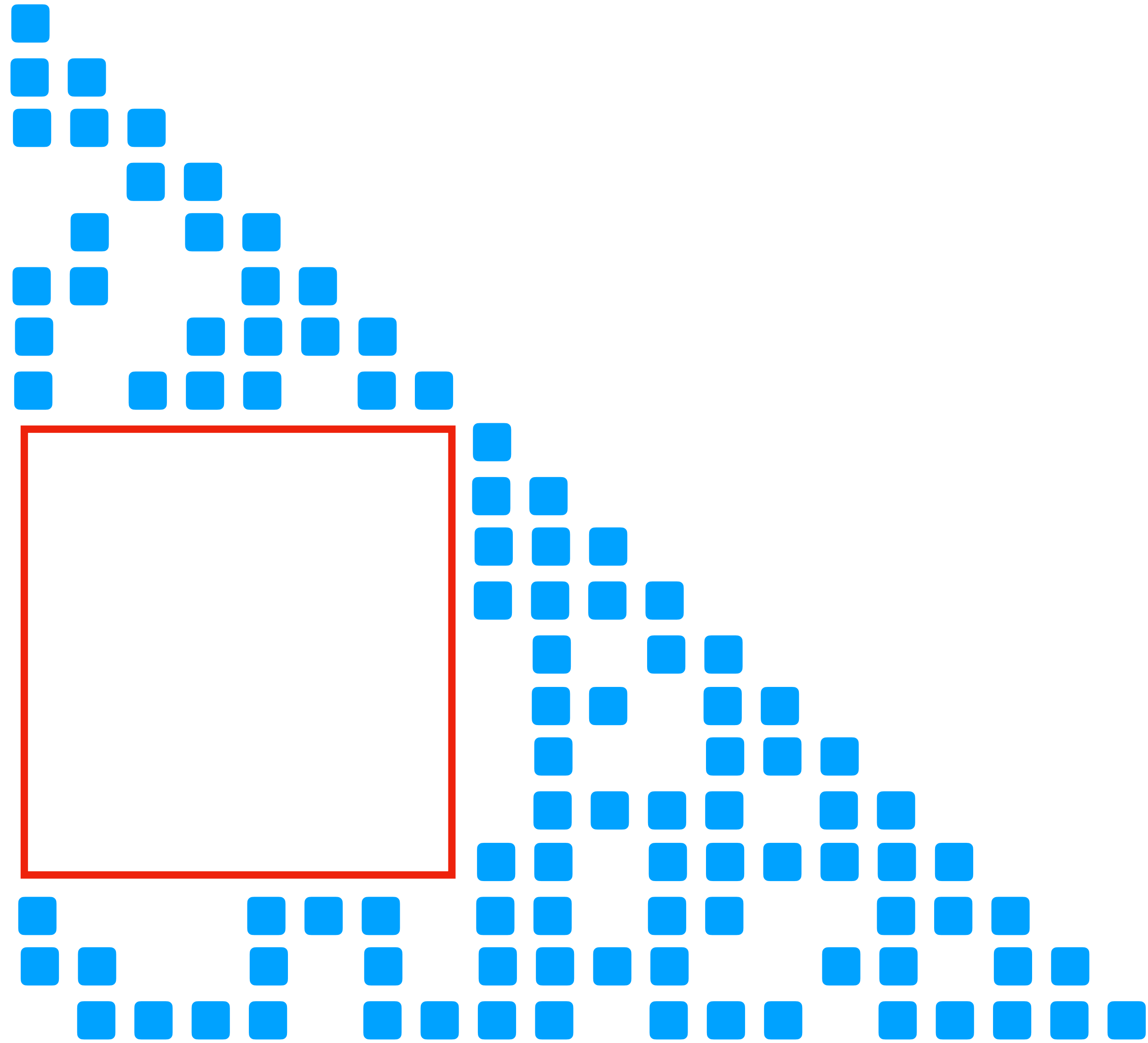
Theory can prove that:
… that gap will also be present
*in the Cholesky factor **L***

Engineering/Maximizing Sparsity

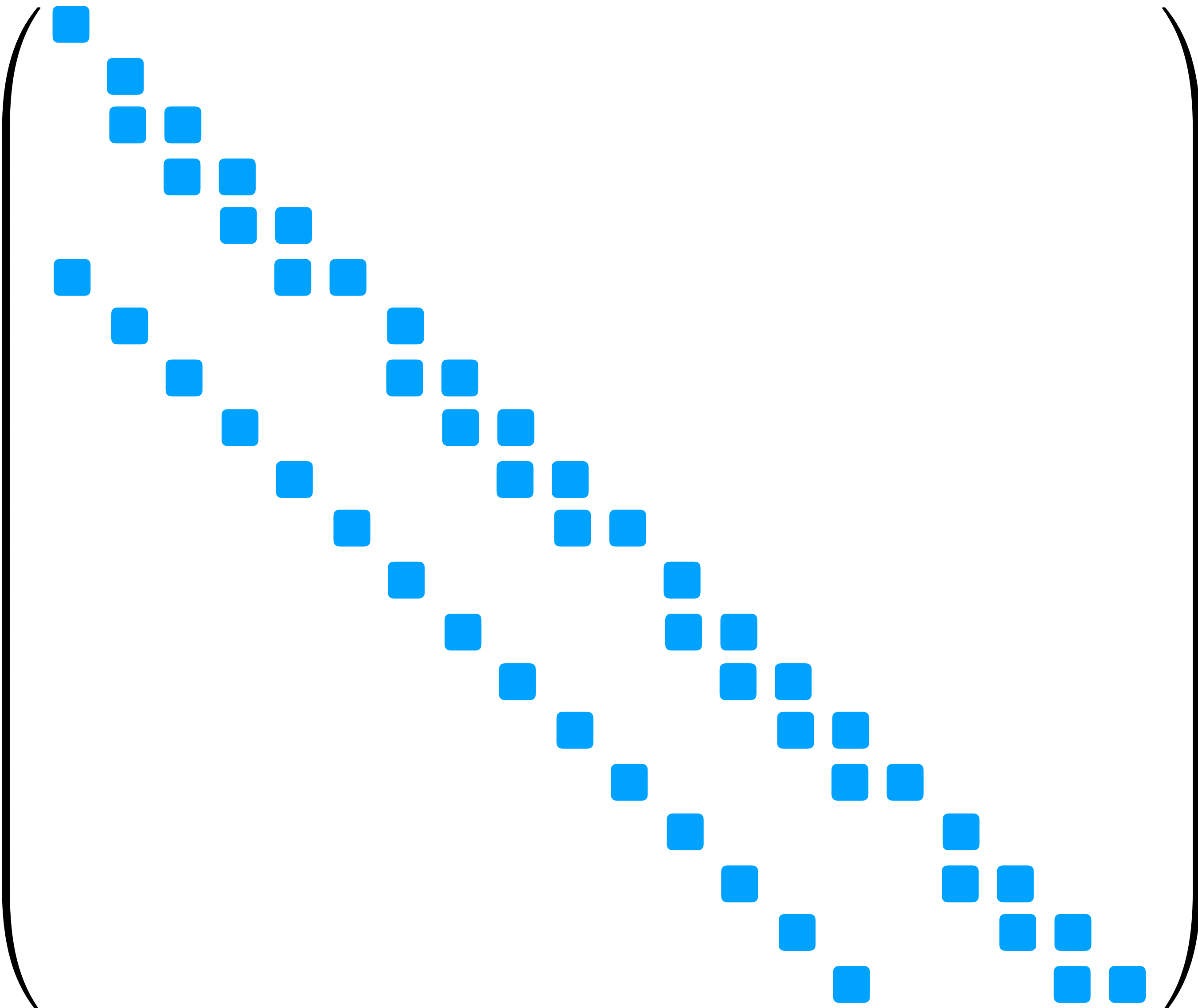


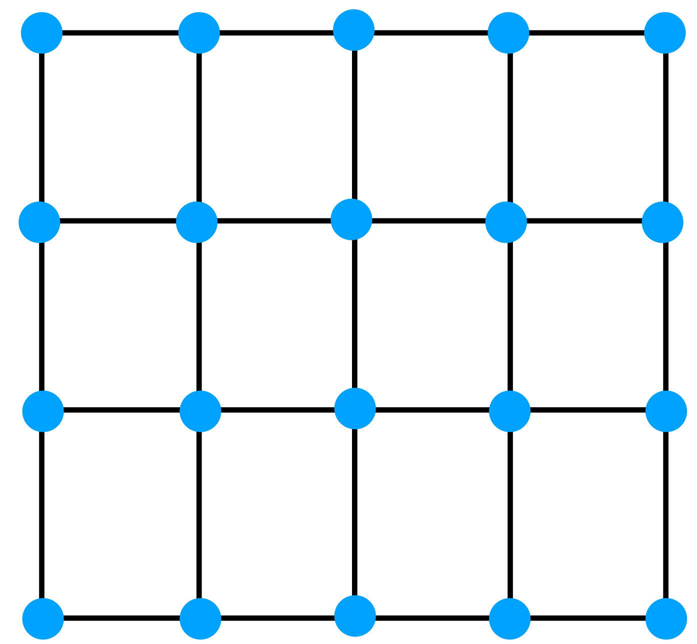
*A sparse matrix **A** can have such gaps without being "dense" elsewhere …*

Engineering/Maximizing Sparsity

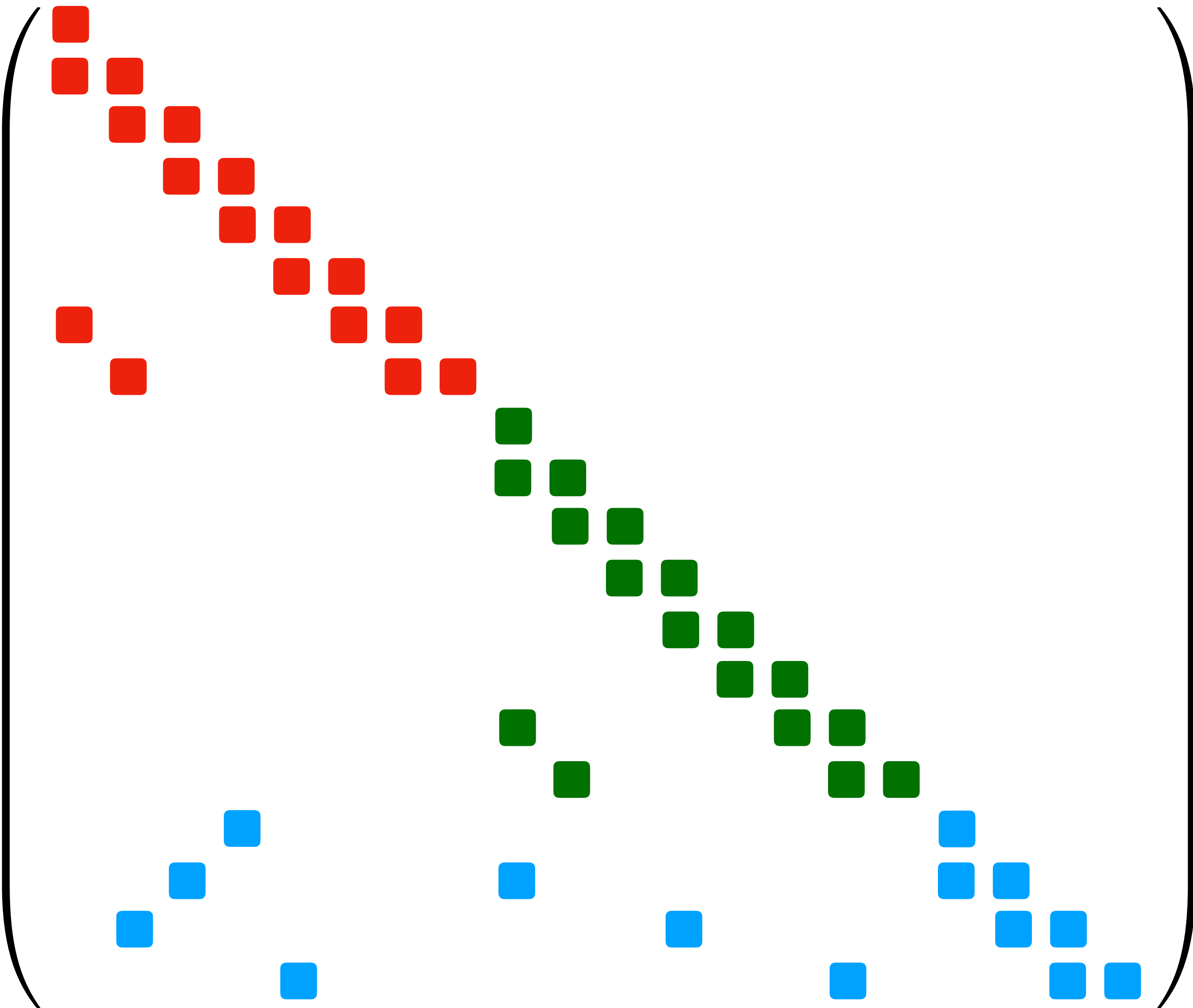


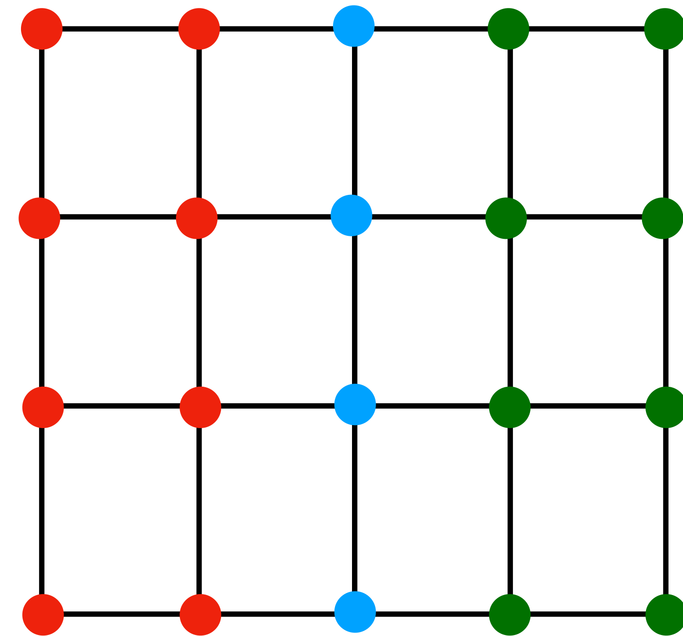
… and the corresponding factor ***L*** *(even if it becomes denser away from such gaps) does retain these "holes" in its sparsity pattern*



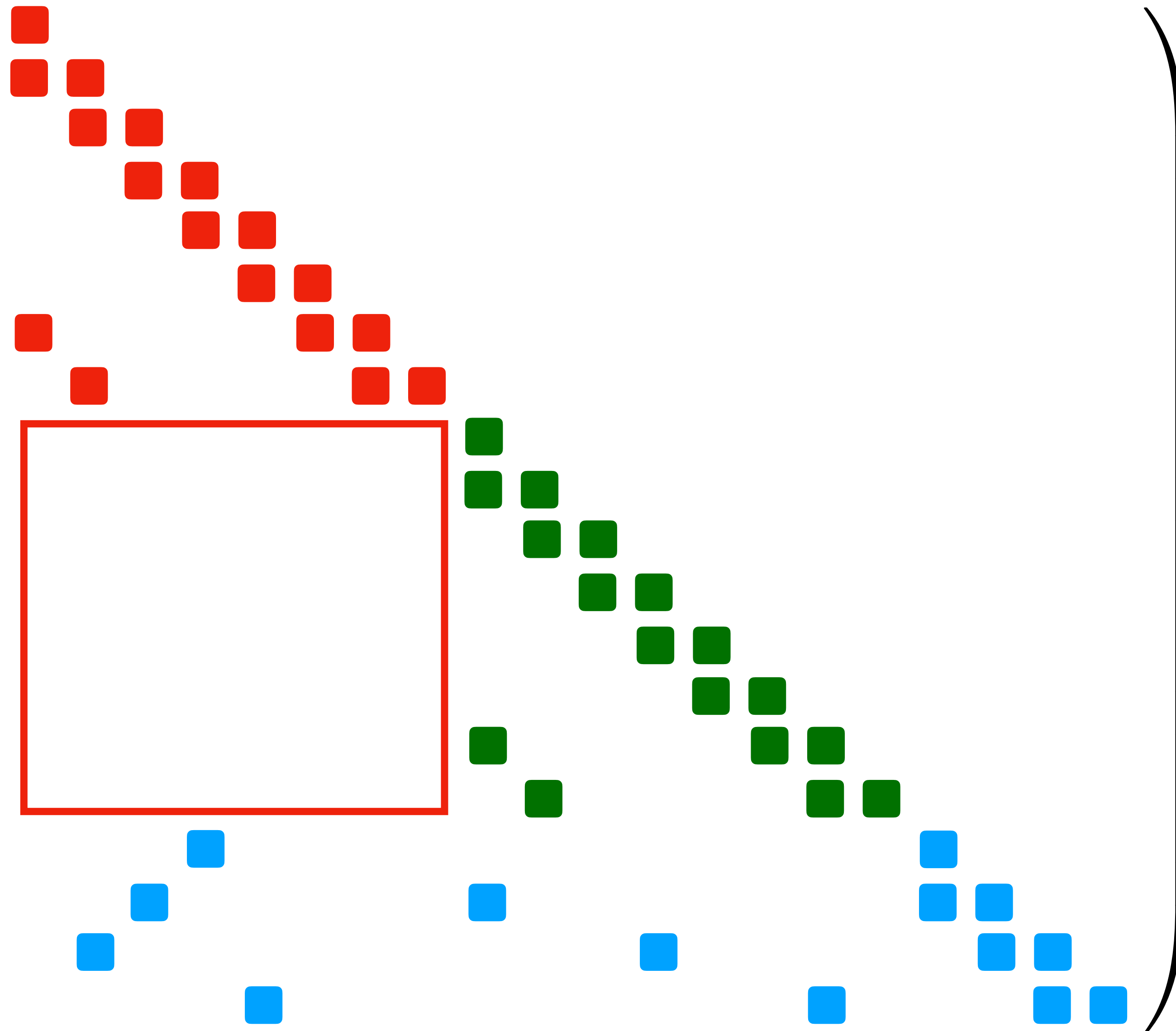


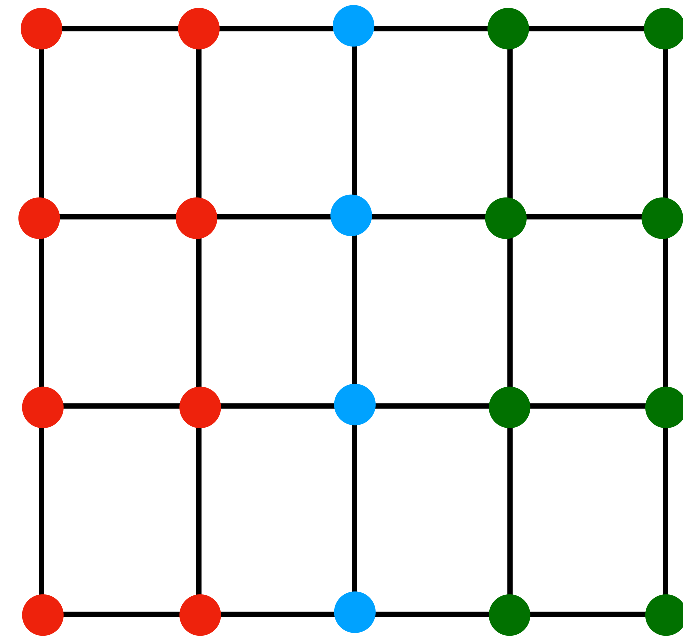
Engineering/Maximizing Sparsity



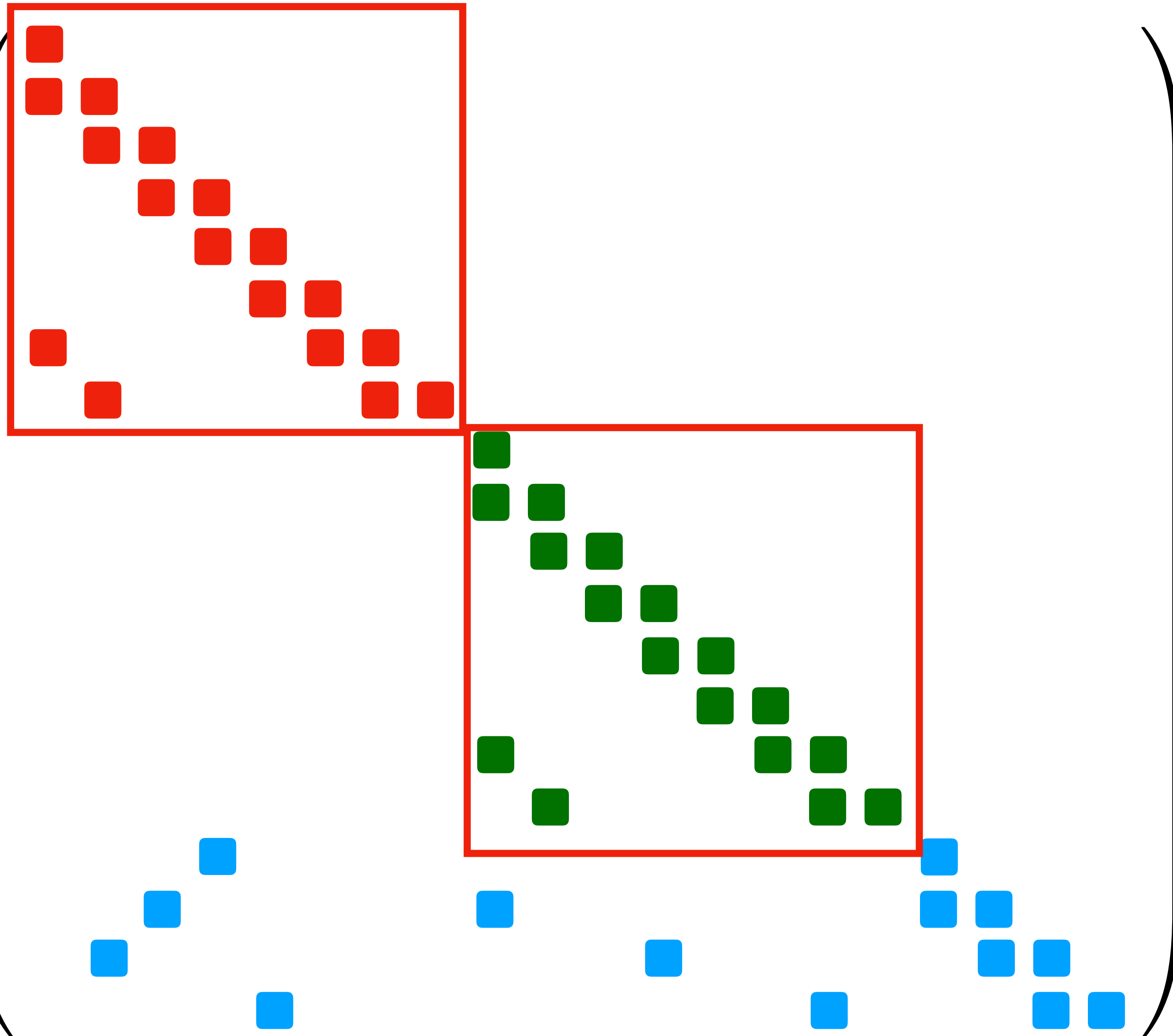


Engineering/Maximizing Sparsity



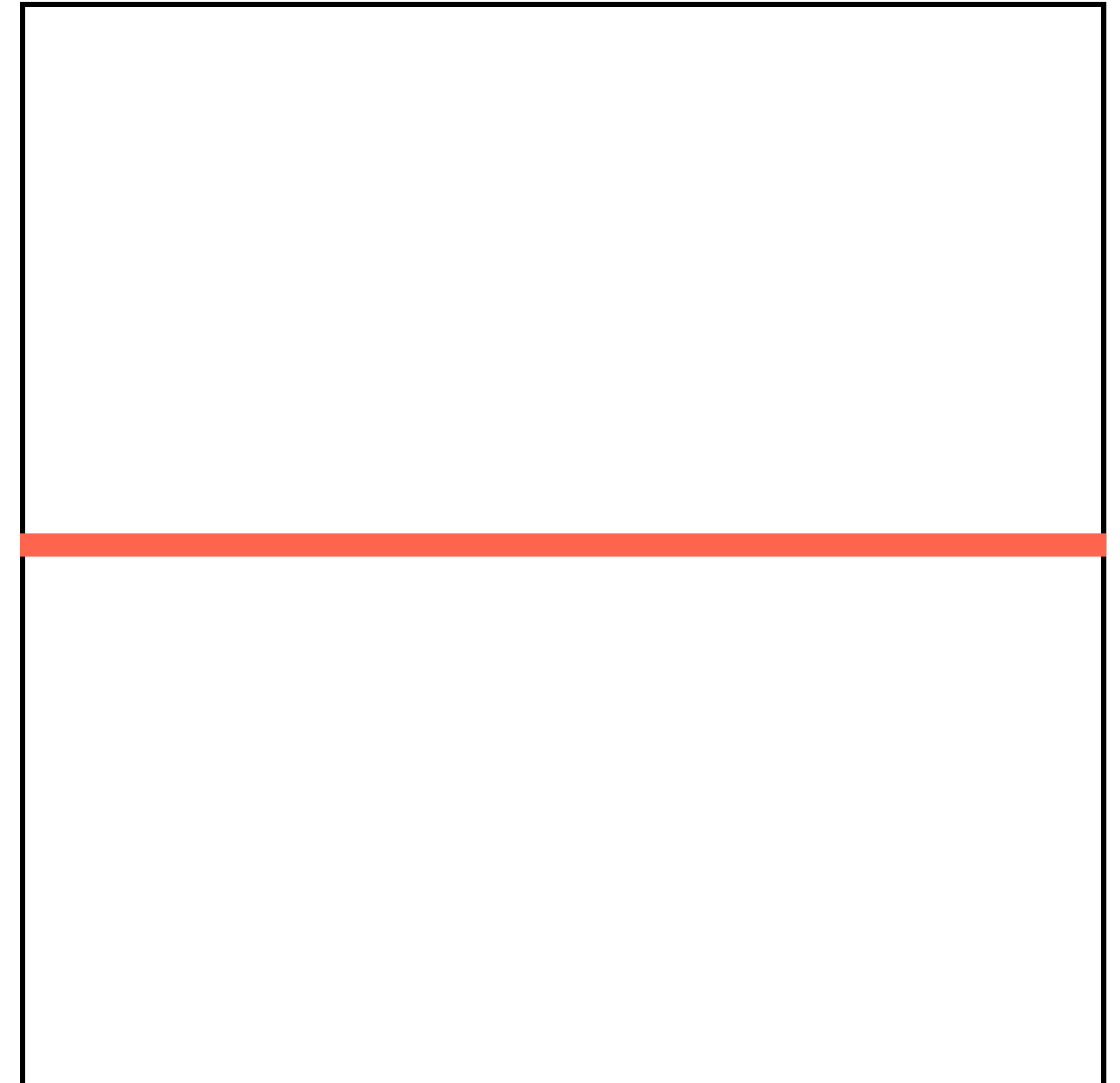


Engineering/Maximizing Sparsity

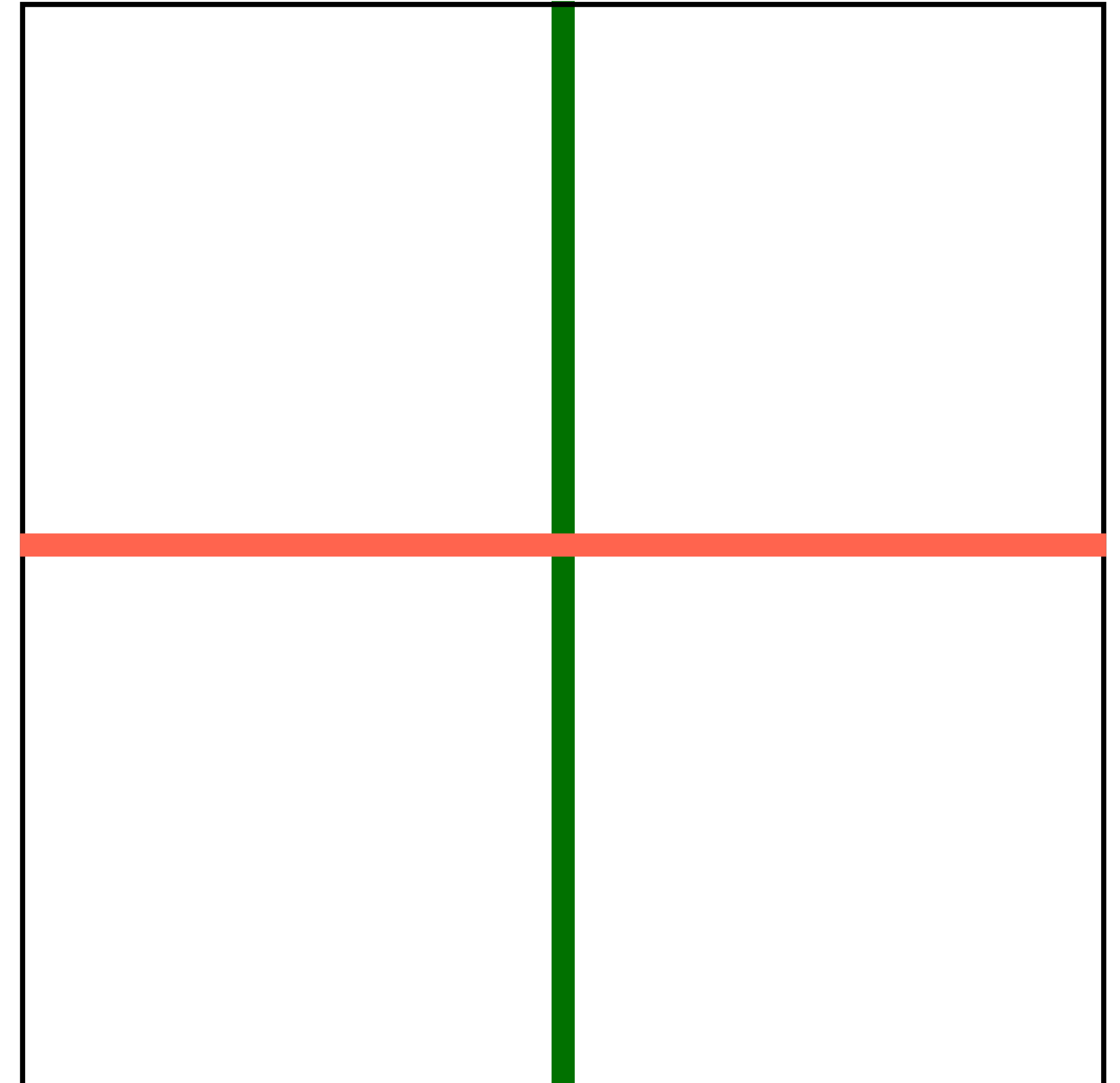


Second benefit:
Cholesky can process each of these two blocks in-parallel!

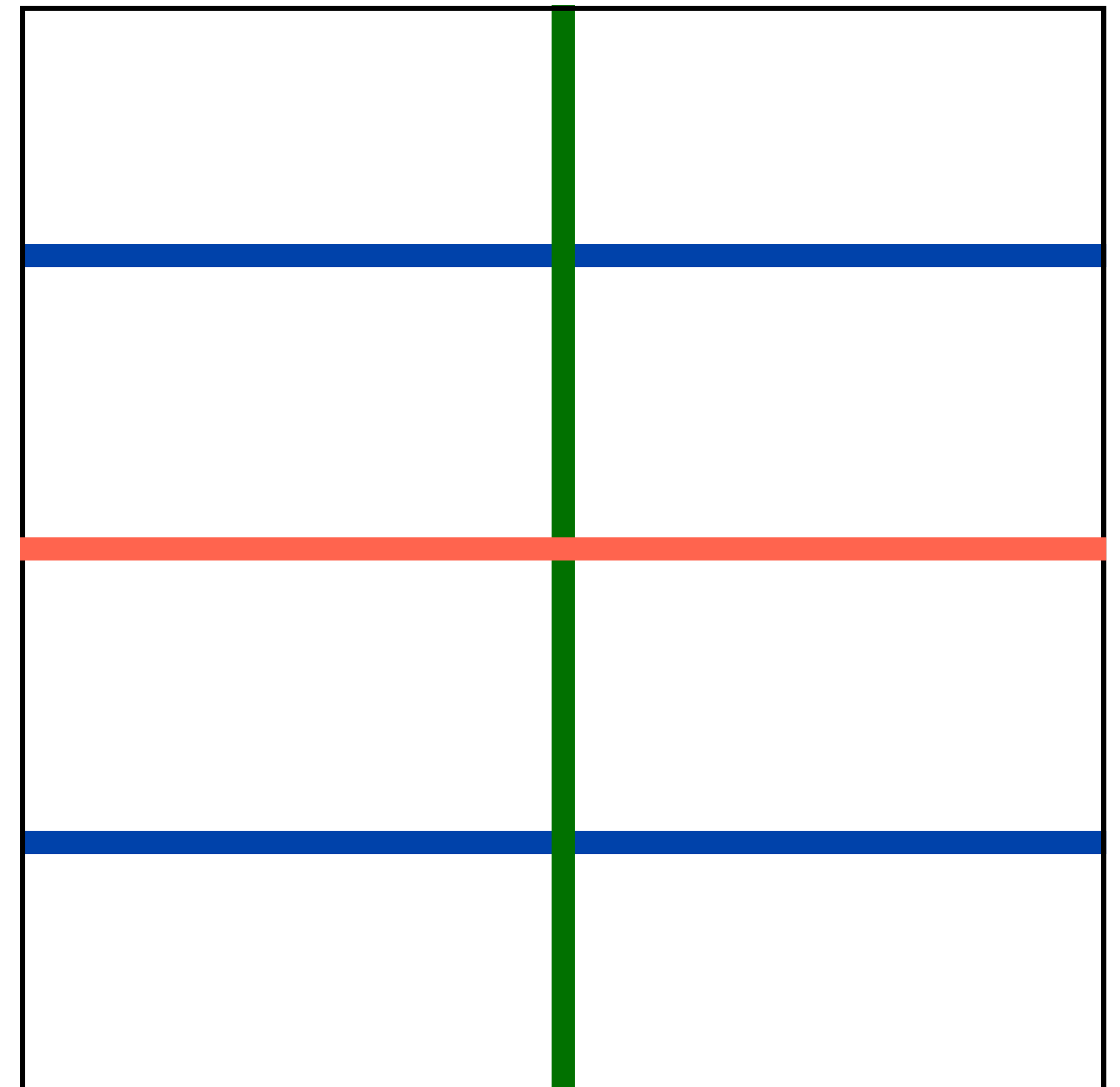
Laplacian - Pattern after a possible reordering



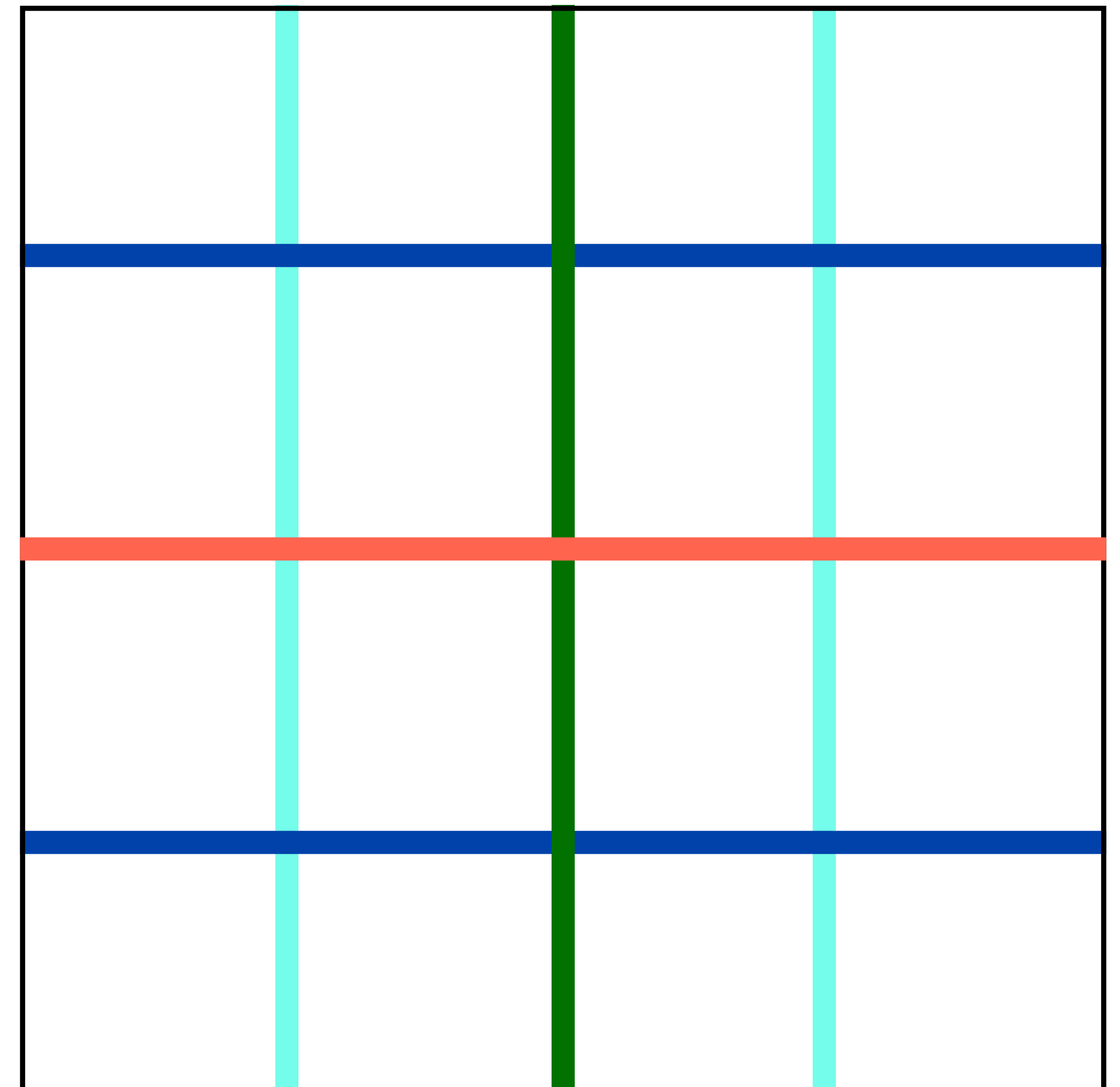
Laplacian - Pattern after a possible reordering



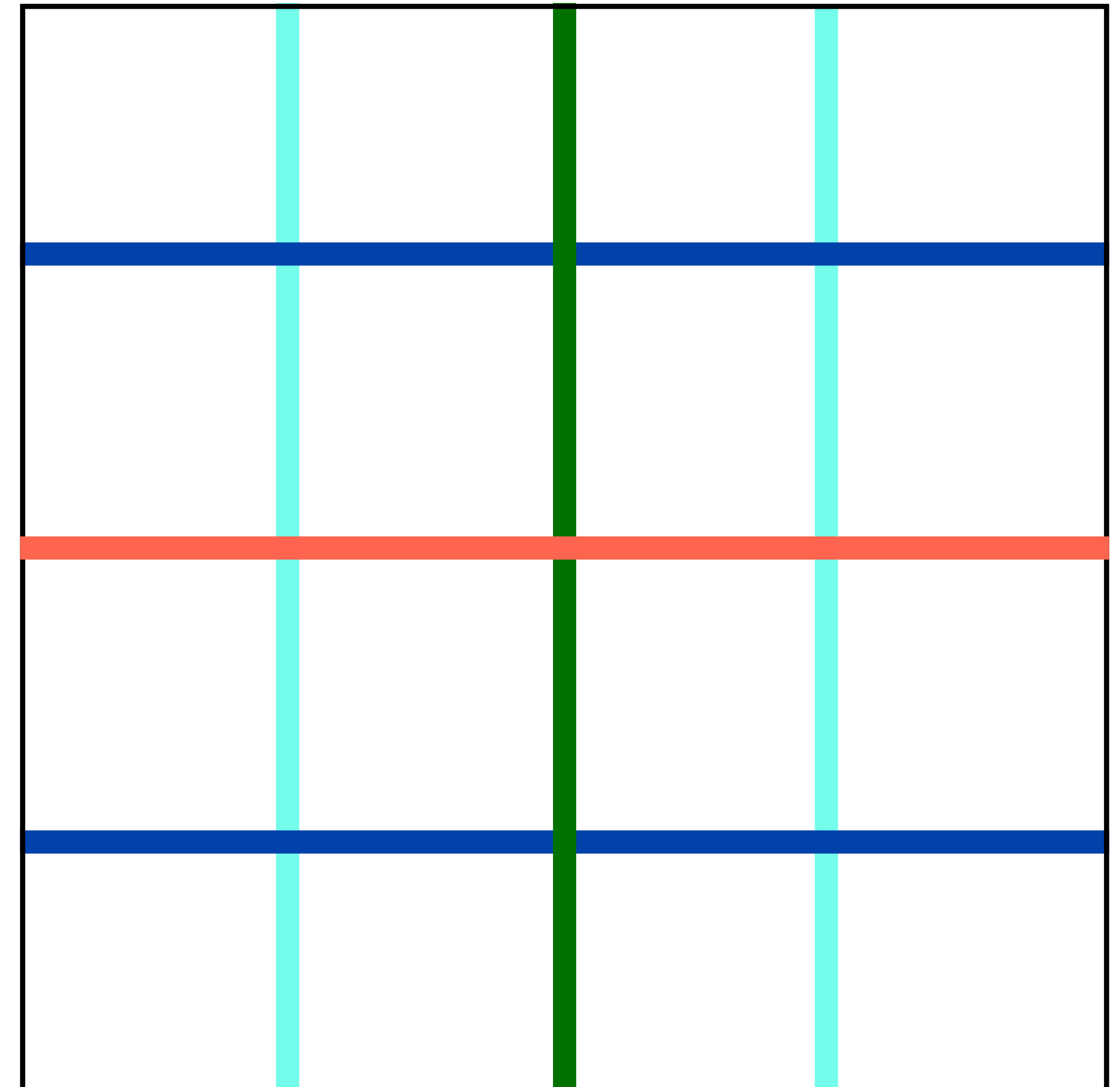
Laplacian - Pattern after a possible reordering



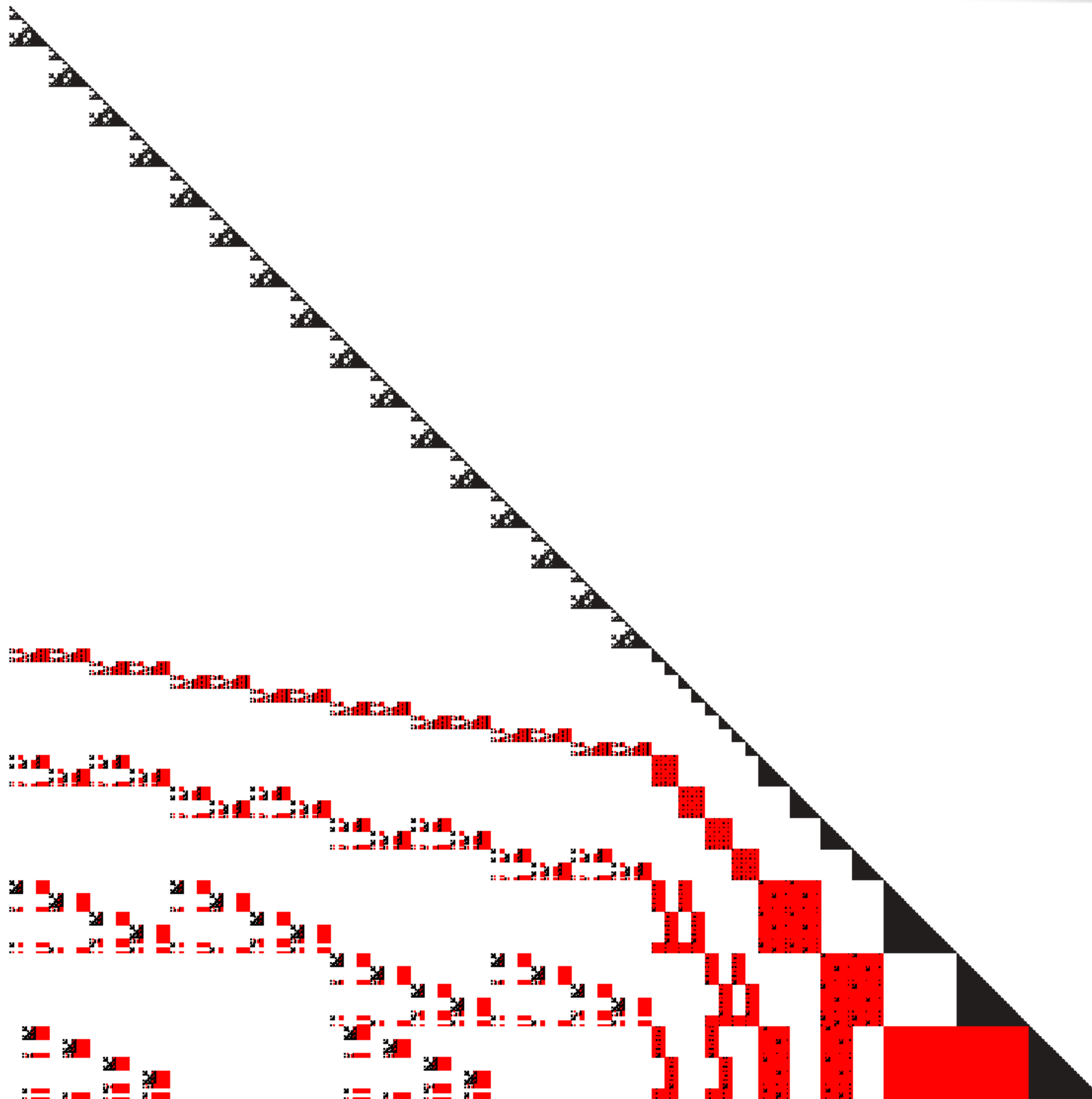
Laplacian - Pattern after a possible reordering



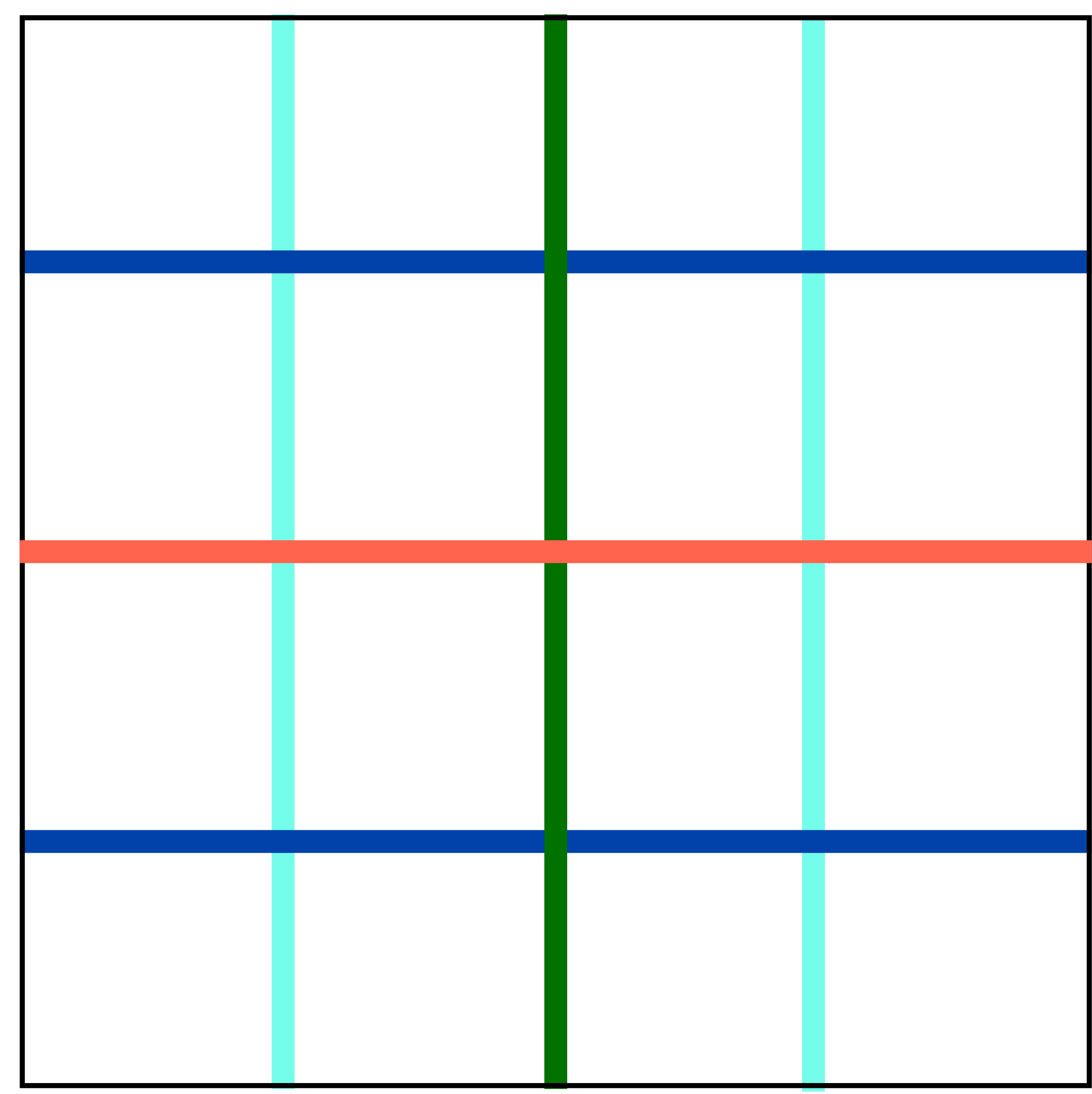
Laplacian - Pattern after a possible reordering



Sparsity of Cholesky Factor (L) vs. Laplacian Matrix



Laplacian - Pattern after a possible reordering



These blocks, too, can be processed in parallel