

# Direct Algorithms for Sparse Schur Complements and Inverses

Dr. Ryan Chilton

MyraMath



myracore.com

unknowns (x10<sup>6</sup>)

1 1.5 2

```
...multifrontal/rholsky/SparseCholeskySolver.h>
11 #include <dense/Matrix.h>
12 #include <dense/MatrixRange.h>
13
14 #include <dense/LowerMatrix.h>
15 #include <dense/LowerMatrixRange.h>
16
17 #include <dense/inverse.h>
18 #include <dense/gemm.h>
19 #include <dense/transpose.h>
20 #include <dense/frobenius.h>
21
22 #include <utility/random.h>
23
24 #include <iostream>
25
26 using namespace myra;
27
28 // Layout of unknowns for laplacian2(7,7):
29 // 0 7 14 21 28 35 42
30 // 1 8 15 31 41 51 43
31 // 2 9 . 44
32 // 3 10 . 45
33 // 4 11 . 46
34 // 5 12 . 47
35 // 6 13 20 27 34 41 48
36
37 int main()
38 {
39     // Form a laplacian2() matrix A and a solver for it.
40     SparseMatrix<double> A = laplacian2<double>(7,7);
41     SparseCholeskySolver<double> solver(A);
42
43     // Form matrix B that only has nonzeros on the "top wall" of the graph.
44     SparseMatrixBuilder<double> B(49,7);
45     for (int ns = 0; ns < 30; ++ns)
46     {
47         int i = random_int(7)*7;
48         int j = random_int(7);
49         double b = random<double>();
50         B(i,j) = b;
51     }
52
53     std::cout << "A = " << std::endl << A.pattern() << std::endl;
54     std::cout << "B = " << std::endl << B.pattern() << std::endl;
55
56     // Form B*inv(A)*B using solver.schur()
57     LowerMatrix<double> S1 = solver.schur(B.sparse());
58
59     // Form B*inv(A)*B using dense kernels, compare.
60     Matrix<double> AA = A.dense();
61     Matrix<double> BB = B.dense();
62     Matrix<double> S2 = transpose(BB)*inverse(AA)*BB;
63
64     // Compare.
65     std::cout << "S1 = B*inv(A)*B (sparse) = " << S1 << std::endl;
66     std::cout << "S2 = B*inv(A)*B (dense) = " << S2 << std::endl;
67     double error = frobenius(S1.dense('T')-S2);
68     std::cout << "||S1-S2|| = " << error << std::endl;
69     return 0;
70 }
```

(Click here to view the output of this program)

Generally speaking,  $S$  is dense even when the  $A, B, C$  operands are sparse. The only structure we exploit is just one triangle of  $S=B^* \text{inv}(A) * B$  as a LowerMatrix (costing half the memory and half the time) and returns a fully populated Matrix representing  $S=B^* \text{inv}(A) * C$ .

Computing Entries of  $\text{inv}(A)$



# Outline

## **Examine some less common sparse direct algorithms:**

Partial linear solution.

Schur complements.

Sampling the inverse operator.

## **Apply them as “frontends” for low-rank skeletonization:**

Cross approximation.

Range estimation.

Ritz projection.

**Motivations: fast direct solvers for FE-BI's and FE-DDM's.**

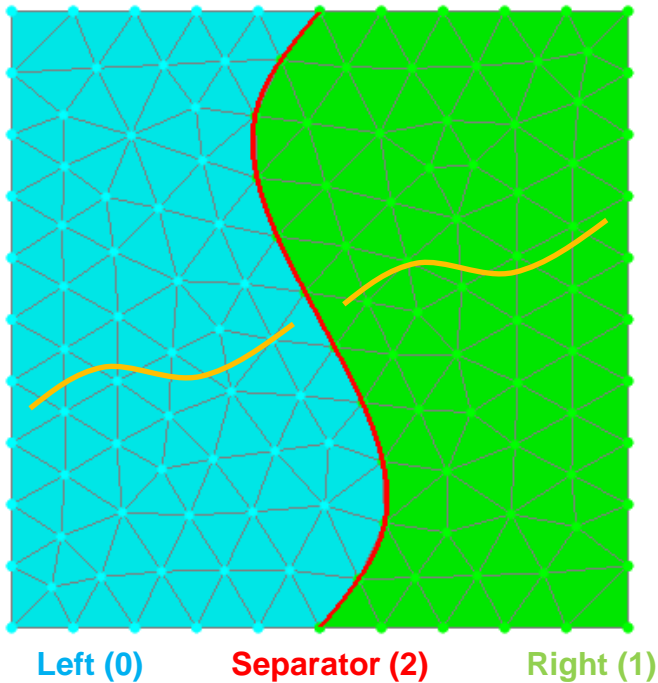


# Refresher: Factor $A=LL^T$

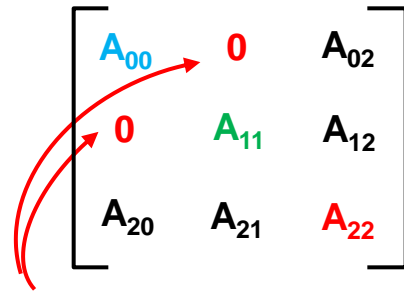
Reorder: **Left(0)**, **Right(1)**, **Separator(2)**.  $A_{01} = A_{10} = \text{all zero!}$

Right looking. Factor  $A_{00}/A_{11}$ , schur downdate  $A_{22}$ , factor.

FEM mesh:

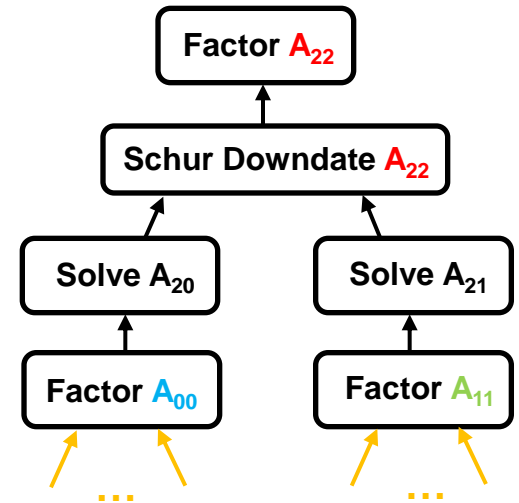


Reordered matrix:



Separator induces these zeroes. They can't fill-in!

Algorithm steps:



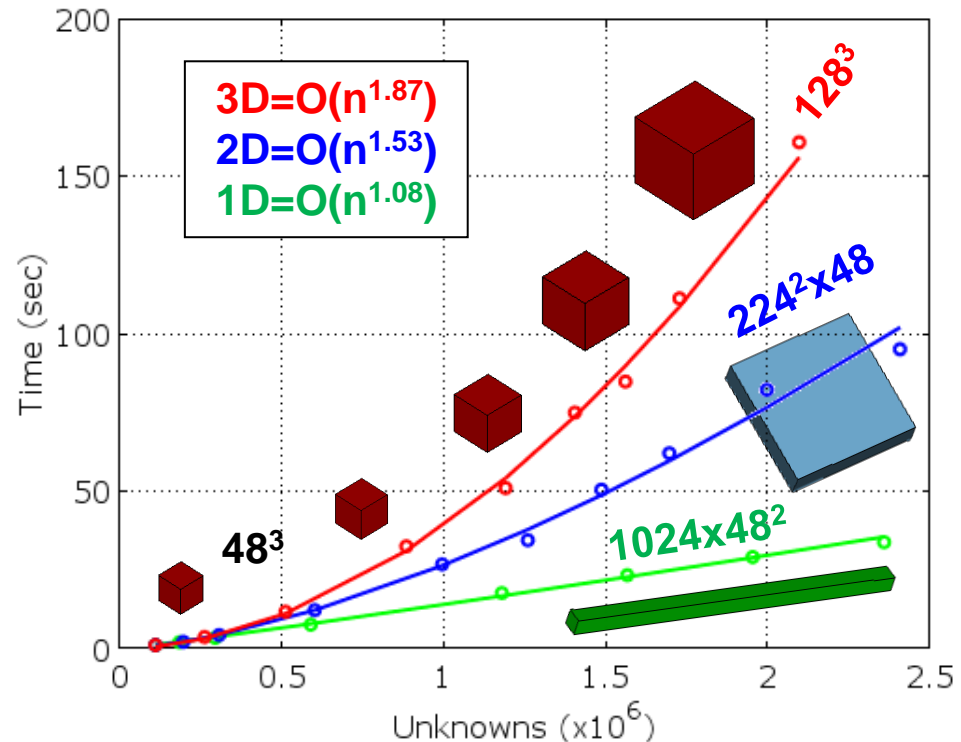
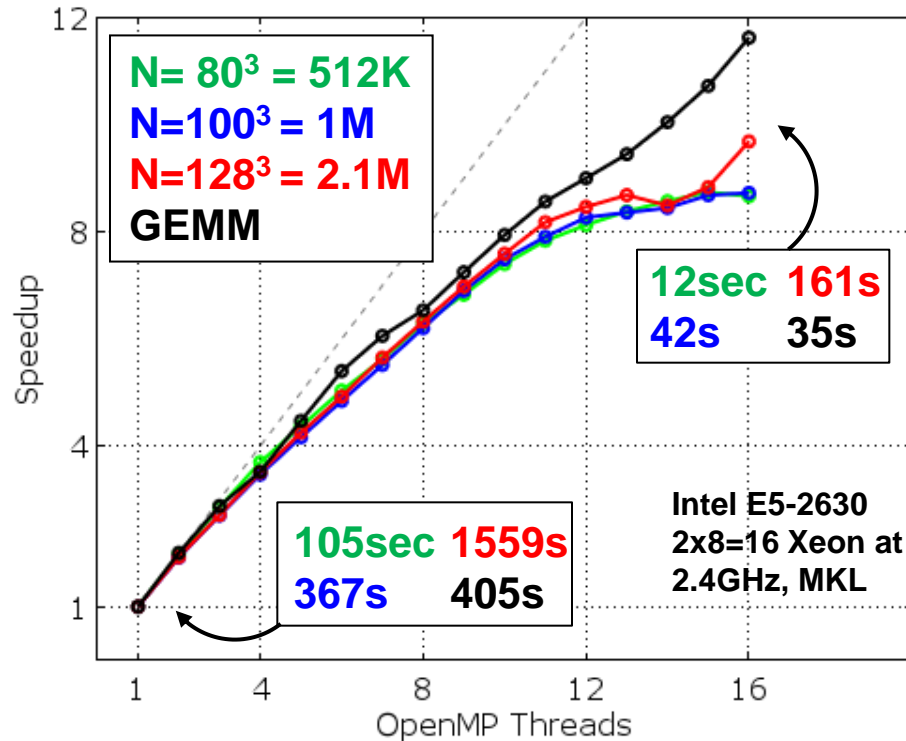
Note  $A_{00}$  and  $A_{11}$  also sparse, apply idea **recursively**.

Leads to a tree of operations, eliminating from bottom up.



# Selected profiling data.

Example problem under study: I x J x K brick (N = IJK)



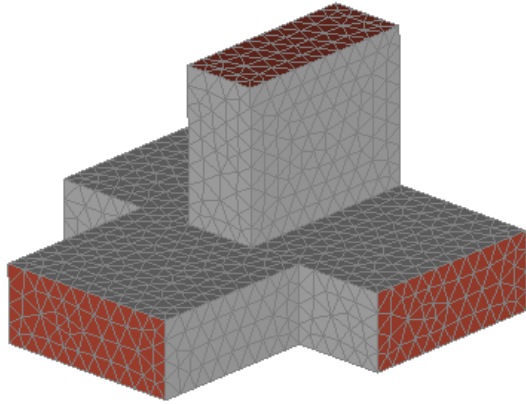
Discrete graph laplacian (7-point): well understood spectrum.

Structured grid: easy to reorder using nested dissection.



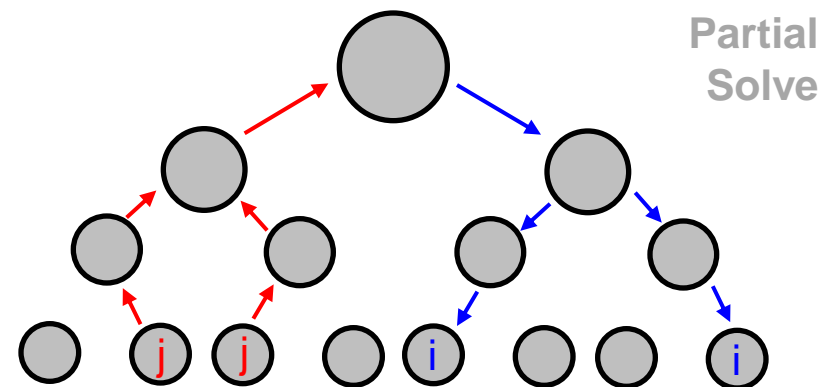
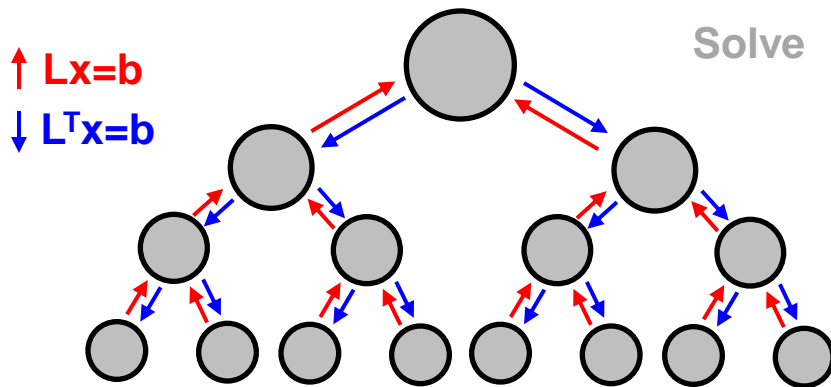
# Partial solution $x=R_i^T A^{-1} R_j b$

In plain english: only  $b(j)$  nonzero, only  $x(i)$  is needed.



$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & +1 & +1 \\ 0 & 0 & +1 & -1 \\ +1 & +1 & 0 & 0 \\ +1 & -1 & 0 & 0 \end{bmatrix}$$

Many engineering QoI's use only *boundary-valued*  $b$  and  $x$ .



$O(n^{4/3})$  time, like  $x=A^{-1}b$ . Only  $O(n^{2/3})$  space per RHS, not  $O(n)$ .



# Schur complement $S=B^T A^{-1} B$

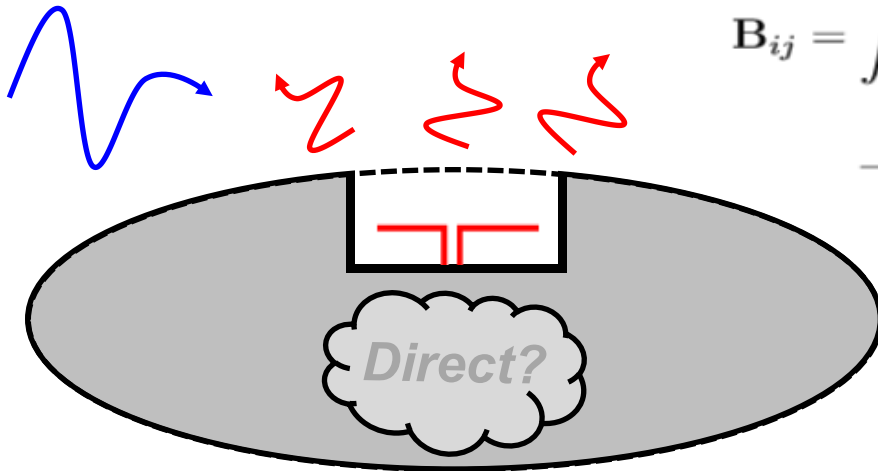
Concept: form “saddle system” of A and B, then “quit” early.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ (\mathbf{L}^{-1}\mathbf{B})^T & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}^T (\mathbf{L}\mathbf{L}^T)^{-1} \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{L}^T & \mathbf{L}^{-1}\mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

Arise from FE-BI hybrids, eg scattering from apertures.

$$\mathbf{A}_{ij} = \int_V (\nabla \times \vec{w}_i \cdot \mu_r^{-1} \nabla \times \vec{w}_j - k^2 \vec{w}_i \cdot \epsilon_r \vec{w}_j) dv$$

$$\mathbf{B}_{ij} = \int_S \nabla \cdot (\hat{n} \times \vec{w}_i) \cdot \int_{S'} \nabla' \cdot (\hat{n}' \times \vec{w}_j) \cdot g(\vec{r}, \vec{r}') dS' dS \\ - k^2 \int_S (\hat{n} \times \vec{w}_i) \cdot \int_{S'} (\hat{n}' \times \vec{w}_j) \cdot g(\vec{r}, \vec{r}') dS' dS$$



$$\begin{bmatrix} \mathbf{A}_{ee} & \mathbf{A}_{em} \\ \mathbf{A}_{me} & \mathbf{A}_{mm} + 2\mathbf{B}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{e} \\ \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{h}_{inc} \end{bmatrix}$$

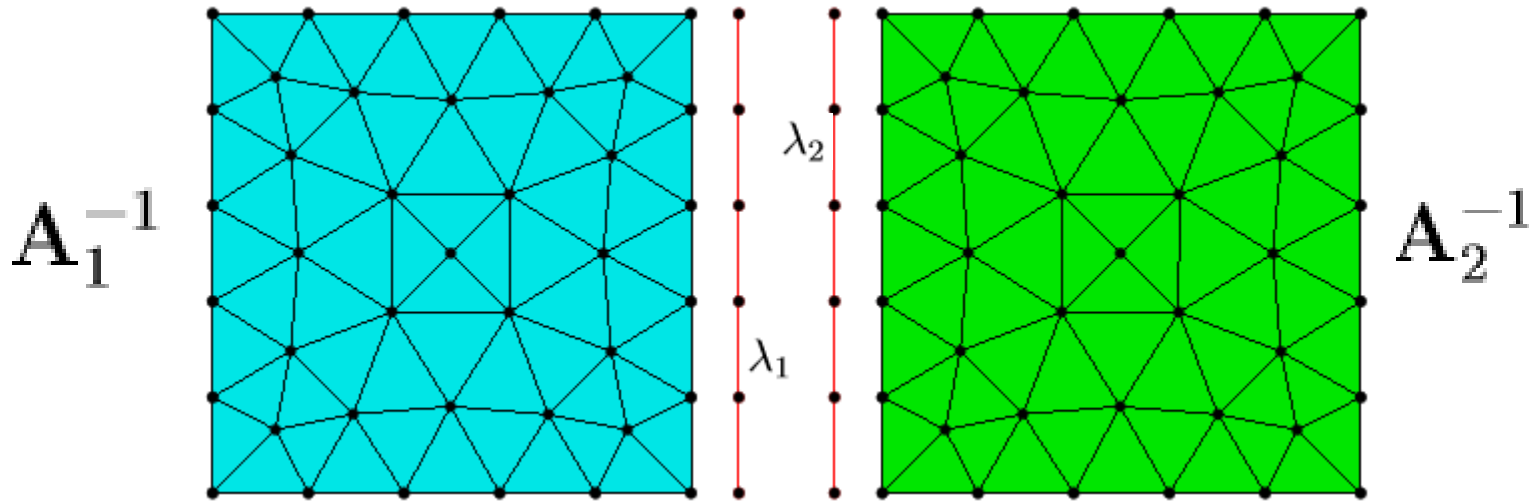
$$\left[ \mathbf{A}_{mm} - \mathbf{A}_{em}^T \mathbf{A}_{ee}^{-1} \mathbf{A}_{em} + 2\mathbf{B}_{mm} \right] \begin{bmatrix} \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{h}_{inc} \end{bmatrix}$$



# Sampling the inverse $Z(i,j)$ , $Z=A^{-1}$

Closely related to Schur complement,  $Z(i,j) = R_i^T \cdot A^{-1} \cdot R_j$

Arise in FETI/DDM, iterate/exchange fields at boundaries.



$$\begin{bmatrix} \mathbf{I} & \mathbf{I} - 2\alpha \mathbf{R}_2^T \mathbf{A}_2^{-1} \mathbf{R}_2 \\ \mathbf{I} - 2\alpha \mathbf{R}_1^T \mathbf{A}_1^{-1} \mathbf{R}_1 & \mathbf{I} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2\alpha \mathbf{R}_1^T \mathbf{A}_1^{-1} \mathbf{f}_1 \\ 2\alpha \mathbf{R}_2^T \mathbf{A}_2^{-1} \mathbf{f}_2 \end{bmatrix}$$

Scatter, solve, gather.

Scatter, solve, gather.

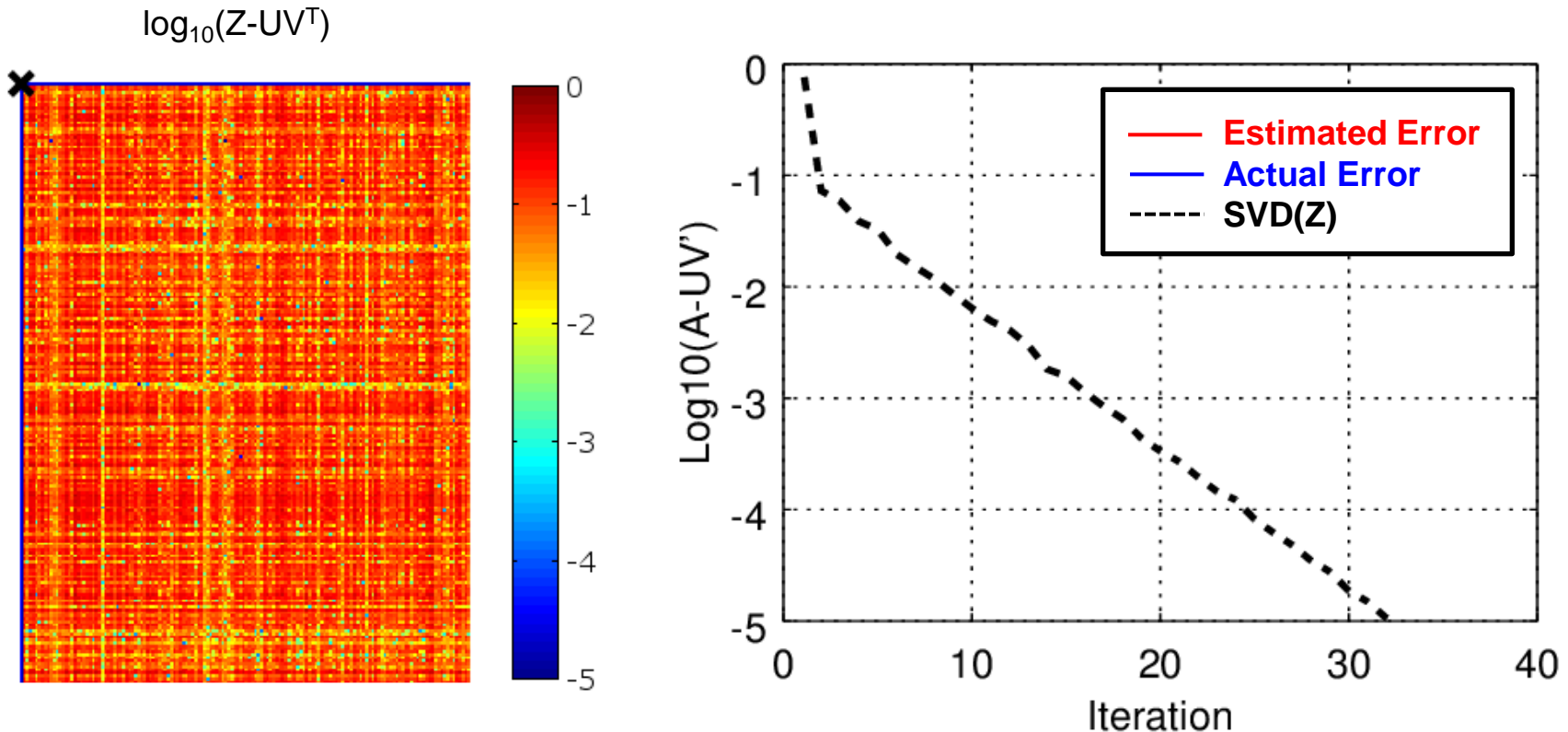
Direct?

Tabulating  $Z(i,j)$  opens up reuse/preconditioning options.



# Cross Approximating $Z(i,j)$ [1/2]

Alternately sample row/column with largest error modulus.



**Key idea: `partialsolve()` can efficiently extract rows/columns:**

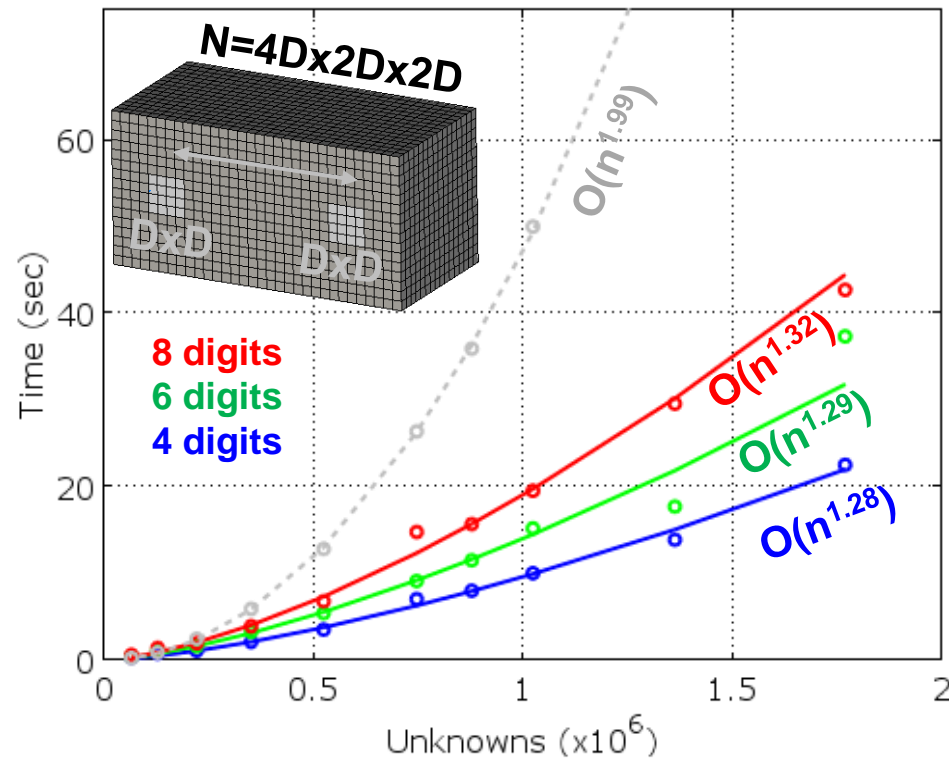
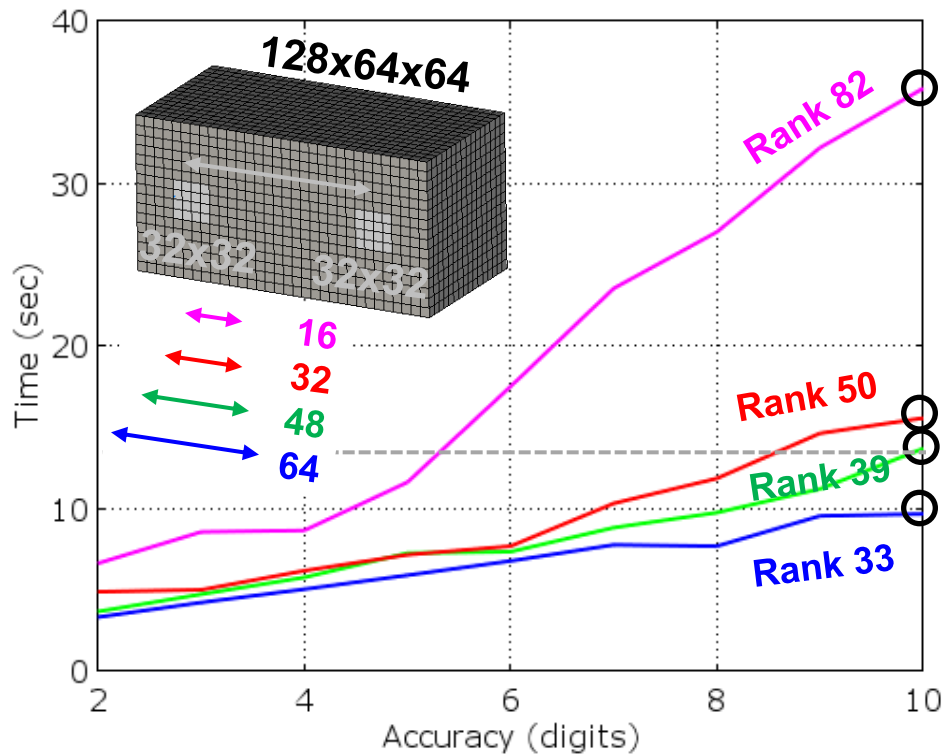
```
c = Z([i],j) = solver.partialsolve([i],j,x=1.0,'Left')
r = Z(i,[j]) = solver.partialsolve(i,[j],x=1.0,'Right')
```





# Cross Approximating $Z(i,j)$ [2/2]

Beats `solver.inverse()` at large  $N$ , especially at low rank/tol.



But in parallel the gap narrows, BLAS3 vs BLAS1 effects.



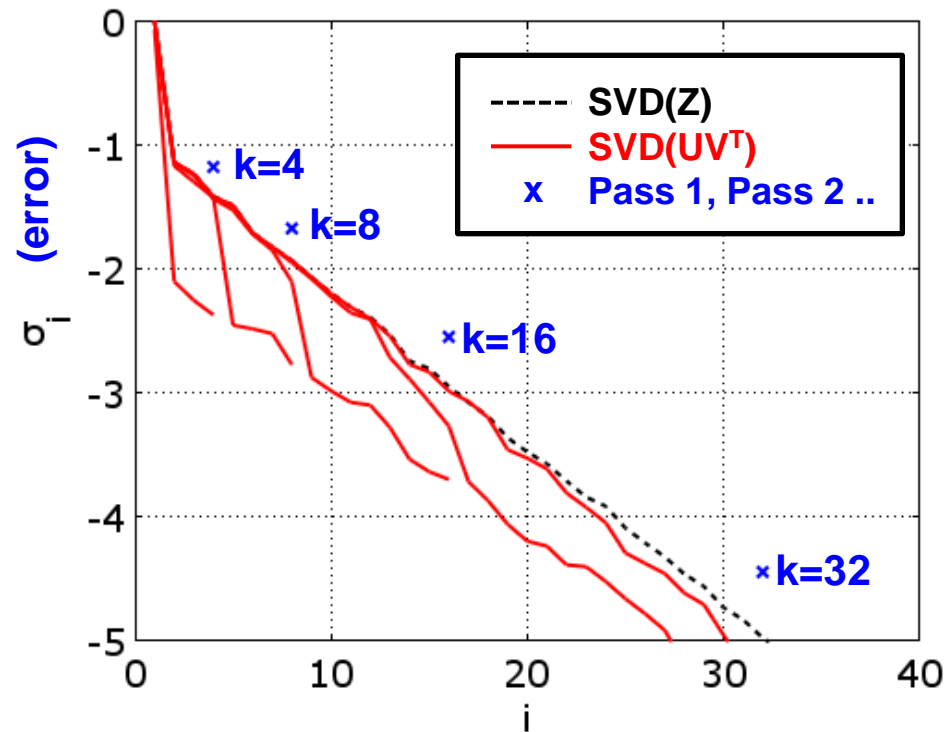
# Range estimation of $Z(i,j)$ [1/2]

Apply action of  $Z$  to random vectors  $X$ , form image  $Y=ZX$ .

If  $Z$  has rapidly decaying  $\sigma$ 's,  $Y$  probably spans  $\text{range}(Z)$ .

```
// Find Q = span(Z)
X = rand(Z.cols, k)
Y = Z.apply(X)
[Q, R, pi] = QR(Y, 0)

// Build k-SVD from Q
W = Z'.apply(Q)
[U, Sigma, V] = svd(W, 0)
Z ≈ (QU) · Sigma · (V)
```



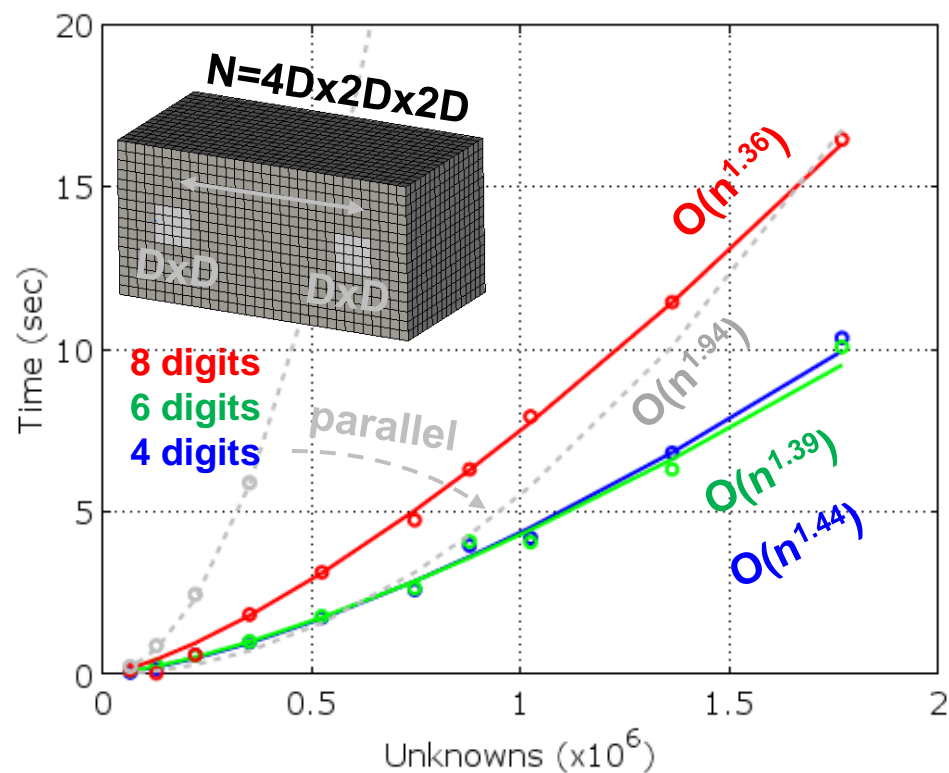
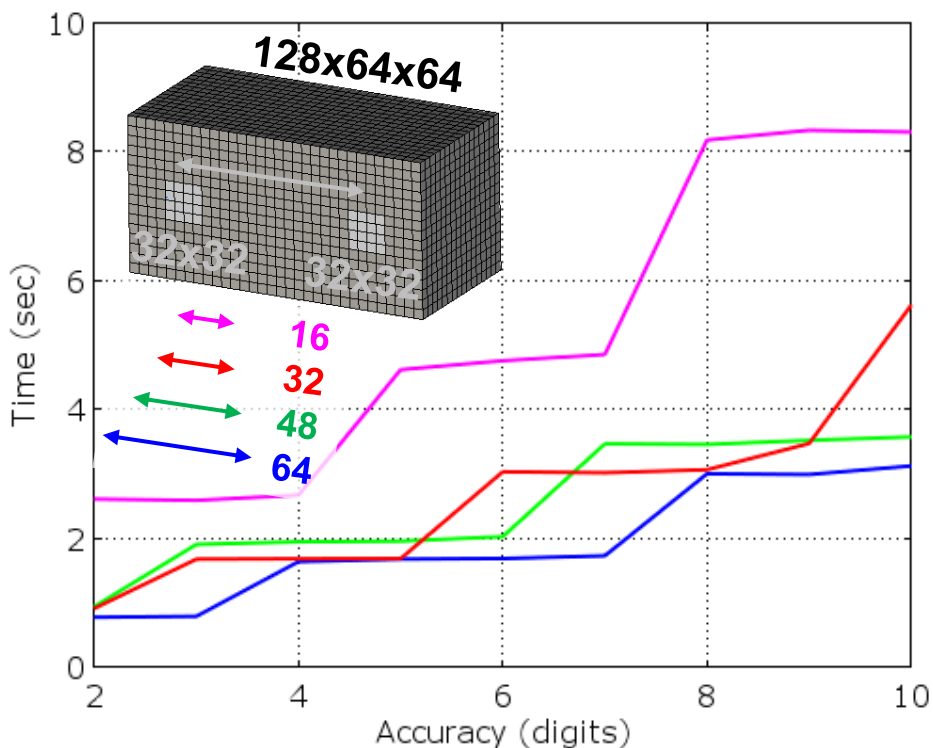
**Key idea: `partialsolve()` can efficiently apply  $Y=Z(i,j) \cdot X$ :**

```
Y = Z([i],[j])*X = solver.partialsolve([i],[j],X,'Left')
```



# Range estimation of $Z(i,j)$ [2/2]

All the same problem instances as before (sizes, shapes).



Availability of all forcing data up front leads to speedup.

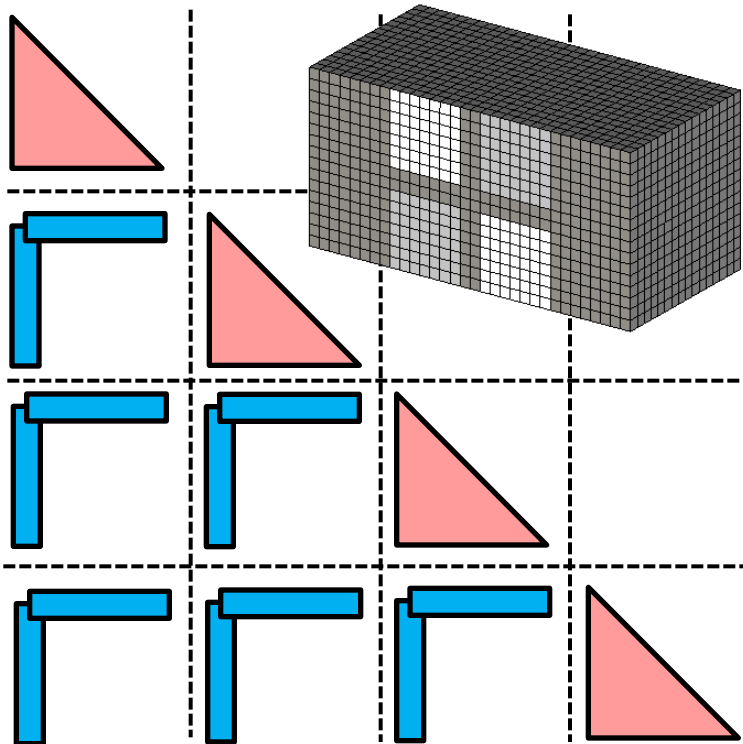
Can be faster than `parallel solver.inverse()`, even at modest  $N$ .



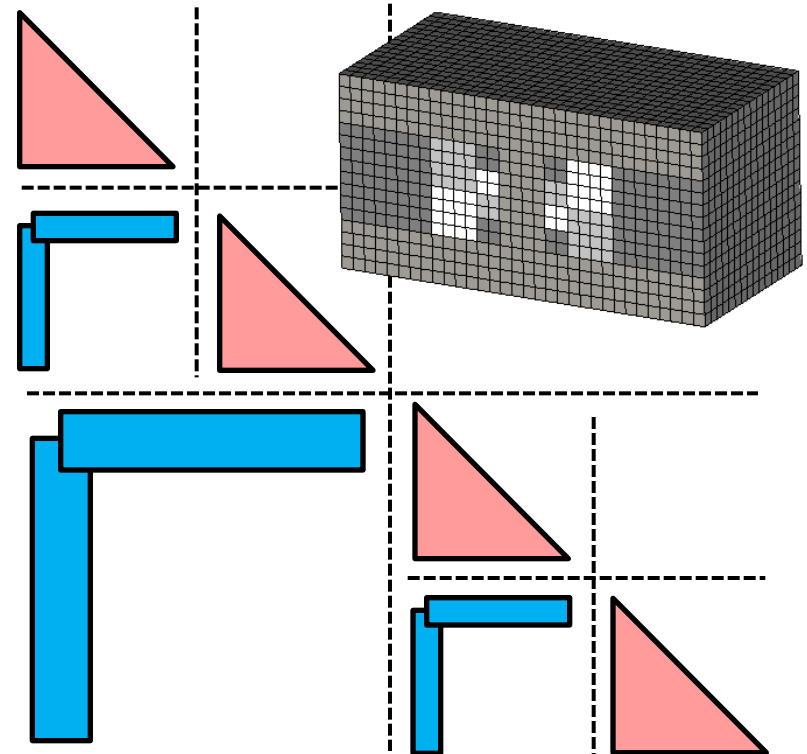
# Ritz Projection of $Z(i,j)$ [1/3]

What about approximating *more than just one block*?

(B)lock (L)ow (R)ank



(H)ierarchical Matrix



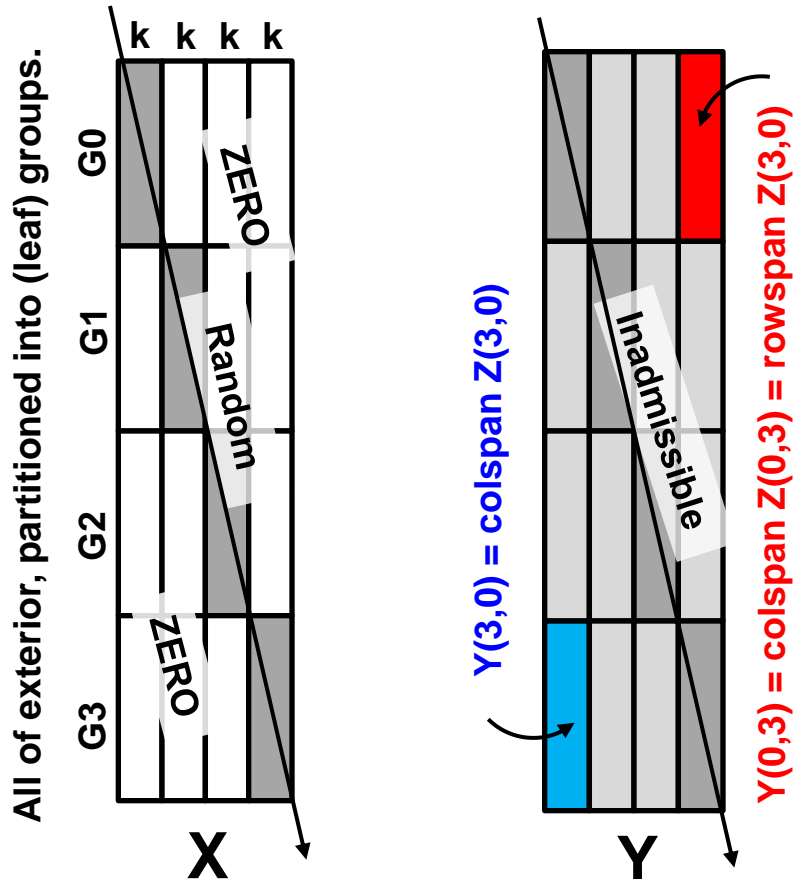
Optimization(BLR)/amortization(H) opportunities do exist.



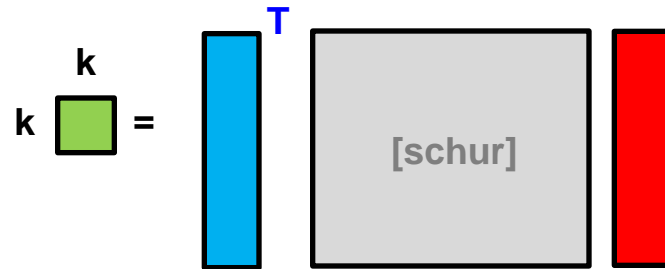
# Ritz Projection of $Z(i,j)$ [2/3]

First pass: find row/column spans using “fat” partialsolve()

`Y = partialsolve([all],[all],X)`



$$R = Y(3,0)^T \cdot Z(3,0) \cdot Y(0,3)$$



$$R = \text{solver.schur}(Y_{30}, Y_{03})$$

$$[U, \Sigma, V] = \text{svd}(R)$$

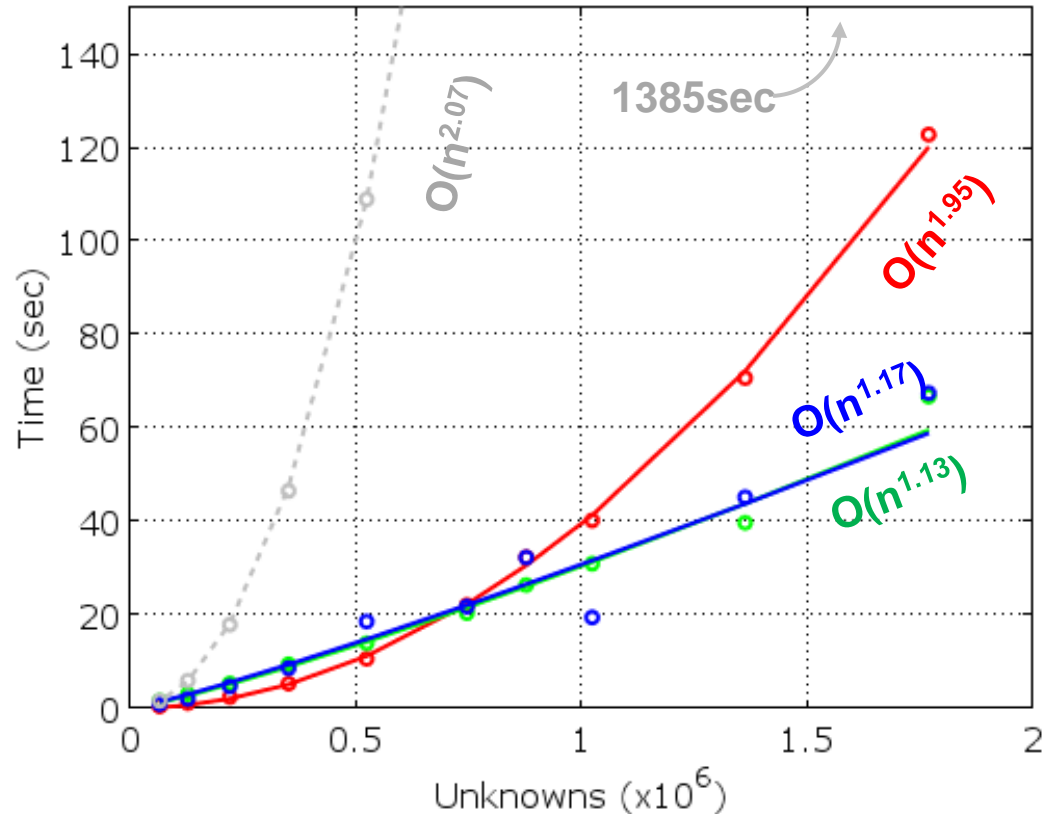
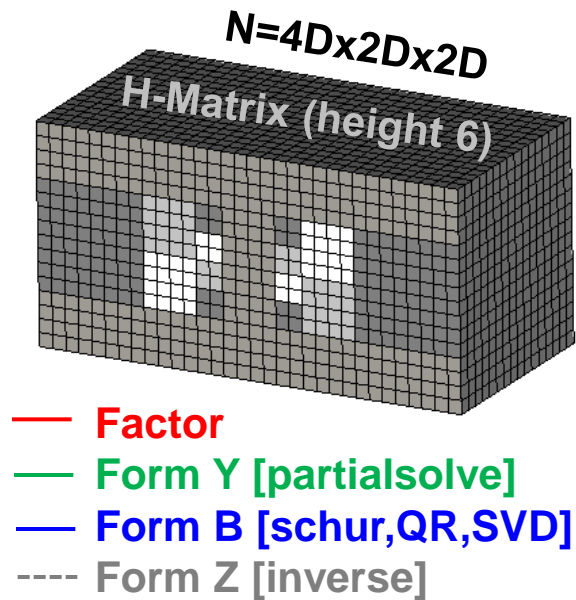
$$Z_{30} \approx (Y_{30} \cdot U) \cdot \Sigma \cdot (V \cdot Y_{03})$$

Second pass: Ritz projection using `solver.schur()`, k-SVD



# Ritz Projection of $Z(i,j)$ [3/3]

Fill an H-matrix representation of  $Z$  restricted to boundary.



Algorithm quickly furnishes all (admissible) blocks.

Can form H-matrix of  $S=B^T A^{-1} B$  with a few minor changes.



# Wrapping Up

## Examined several uncommon sparse direct algorithms:

Partial linear solution:  $x = R_i^T A^{-1} R_j b$  (sparse  $b$ , sifted  $x$ )

Schur complements:  $B^T A^{-1} B$ ,  $B^T A^{-1} C$ , all sparse

Sampling the inverse operator:  $Z(i,j) = R_i A^{-1} R_j$

## Used them as “frontends” for low-rank/skeletonization:

Cross approximation: `partialsolve()` can extract row/column

Range estimation: `partialsolve()` can apply  $Z(i,j)$  quickly

Ritz projection: `schur()+partialsolve()`, amortization over blocks

## Essential tools for FE/BI/DDM methods (sparsity+lowrank).



# Contact: myracore.com

**MyraMath: sparse factor/solve/schur/inverse/partialsolve.**

**MyraKL: BLAS/LAPACK API for MyraMath, or use MKL.**

## MyraMath

Home Download Build Tutorial Browse Source

### API Tutorials

MyraMath provides an ecosystem of algebraic containers, wrappers for BLAS/LAPACK, and other tools to help you deal with this large "API surface", some example-driven tutorials are provided by the most important part of the library, this sequence of tutorials tries to master them.

- Step 1: Read the [Tutorial for /dense routines](#)
- Step 2: Read the [Tutorial for /sparse routines](#)
- Step 3: Read the [Tutorial for /multifrontal solvers \(part 1\)](#)
- Step 4: Read the [Tutorial for /multifrontal solvers \(part 2\)](#)

MyraMath also provides additional algebraic functionality beyond sparse direct solvers but are not essential. Browse them in any order you like:

- Browse the [Tutorial for /iterative routines](#)
- Browse the [Tutorial for /pdense routines](#)
- Browse the [Tutorial for /expression templates](#).
- Browse the [Quick reference for converting Matlab/Octave programs](#).

Finally, there are a few tutorials that discuss general-purpose/non-algebraic containers:

- Browse the [Tutorial for /io routines](#).
- Browse the [Tutorial for /jobgraph routines](#)
- Browse the [Summary of /test programs](#).

## MyraKL

Home Download Build Tutorial Browse Source

### Instructions for building on Windows

### Overview

MyraKL can be built natively on Windows using the Microsoft Visual C++ compiler. The steps for building on Visual Studio are similar. The first step is to generate a `.sln` (solution) file for the library (`myrakl.dll`) and the accompanying test suite is straightforward. For the LAPACK/BLAS tests (within the folders `/dense/tests` and `/dense`).

Below are links to the sections of this page:

- [Running CMake](#)
- [Opening MyraKL in Visual Studio](#)
- [Building the library and tests](#)
- [Running test executables](#)
- [Creating new projects of your own](#)

### Running CMake

A native Windows installer for CMake can be found at [CMake.org](#). If you hold the original source code and the other to hold the products of the build, labeled "where is the source code:", type in the path to MyraKL's build folder. Alternatively, you use can the browse buttons on the right of the "Visual Studio 14 2015 Win64" (for 64 bit systems).

**Free software (GPL), or dual license (info@myracore.com)**