Direct Algorithms for Sparse Schur Complements and Inverses

Dr. Ryan Chilton





```
// Form a laplacian2() matrix A and a solver for it.
      SparseMatrix<double> A = laplacian2<double>(7,7);
      SparseRCholeskySolver<double> solver(A);
      // Form matrix B that only has nonzeros on the "top wall" of the gr
      SparseMatrixBuilder<double> B(49,7);
      for (int ns = 0; ns < 30; ++ns)
        int i = random int(7)*7;
       int j = random int(7);
        double b = random<double>();
        B(i,j) = b;
      std::cout << "A = " << std::endl << A.pattern() << std::endl;</pre>
      std::cout << "B = " << std::endl << B.pattern() << std::endl;</pre>
      // Form B'*inv(A)*B using solver.schur()
      LowerMatrix<double> S1 = solver.schur(B.sparse());
54
      // Form B'*inv(A)*B using dense kernels, compare.
5.6
      Matrix<double> AA = A.dense();
      Matrix<double> BB = B.dense();
5.9
      Matrix<double> S2 = transpose(BB) *inverse(AA) *BB;
60
61
      // Compare.
      std::cout << "S1 = B'*inv(A)*B (sparse) = " << S1 << std::endl;</pre>
      std::cout << "S2 = B'*inv(A)*B (dense) = " << S2 << std::endl;
63
64
      double error = frobenius(S1.dense('T')-S2);
65
      std::cout << "|S1-S2| = " << error << std::endl;
66
      return 0;
67
```

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

iris.jo...

Untit...

Untit...



sketc...

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). $A_{01} = A_{10} =$ all zero! Right looking. Factor A_{00}/A_{11} , schur downdate A_{22} , factor.



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

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

e pluribus rapidum

e pluribus rapidum

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.

e pluribus rapidum

Partial solution $x=R_i^TA^{-1}R_j^{-1}b$

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



Many engineering Qol's use only boundary-valued b and x.



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

Schur complement S=B^TA⁻¹B



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.

$$\mathbf{A}_{ij} = \int_{V} \left(\nabla \times \vec{w}_{i} \cdot \mu_{r}^{-1} \nabla \times \vec{w}_{j} - k^{2} \vec{w}_{i} \cdot \epsilon_{r} \vec{w}_{j} \right) dv$$

$$\mathbf{B}_{ij} = \int_{S} \nabla \bullet \left(\hat{n} \times \vec{w}_{i} \right) \cdot \int_{S'} \nabla' \bullet \left(\hat{n}' \times \vec{w}_{j} \right) \cdot g\left(\vec{r}, \vec{r}' \right) dS' dS$$

$$-k^{2} \int_{S} \left(\hat{n} \times \vec{w}_{i} \right) \bullet \int_{S'} \left(\hat{n}' \times \vec{w}_{j} \right) \cdot g\left(\vec{r}, \vec{r}' \right) dS' dS$$

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

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

Sampling the inverse Z(i,j), Z=A⁻¹



e pluribus rapidum

- Closely related to Schur complement, $Z(i,j) = R_i^T \cdot A^{-1} \cdot R_i$
- Arise in FETI/DDM, iterate/exchange fields at boundaries.



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

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



e pluribus rapidum

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]



e pluribus rapidum

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 σ 's, Y probably spans range(Z).



Key idea: partialsolve() can efficiently apply Y=Z(i,j)·X: Y = Z([i],[j])*X = solver.partialsolve([i],[j],X,'Left')

e pluribus rapidum

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)eirarchical 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)



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: $x=R_i^TA^{-1}R_j^{-1}B_i^{-1}B_j^{-1}B_i^{-$

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 FEBI/DDM methods (sparsity+lowrank).

Contact: myracore.com



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

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



Browse the Summary of /test programs.

MvraKL

Home Download Build Tutorial Browse Source Instructions for building on Windows

Overview

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

Below are links to the sections of this page

- Running CMake
- Opening Myral/L in Visual Studio
- Building the library and tests
- Running test executables
- Greating new projects of your own

Running CMake

A native Windows installer for CMake can be found at CMake.org. If yo hold the original source code and the other to hold the products of the labeled "Where is the source code:", type in the path to MyraKL' build folder. Alternatively, you use can the browse buttons on the right 1 "Visual Studio 14 2015 Win64" (for 64 bit systems).

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