Direct and Incomplete Cholesky Factorizations with Static Supernodes

AMSC 661 Term Project Report

Yuancheng Luo

2010-05-14

Introduction

Incomplete factorizations of sparse symmetric positive definite (SSPD) matrices have been used to generate preconditioners for various iterative solvers. These solvers generally use preconditioners derived from the matrix system, $Ax = b$, in order to reduce the total number of iterations until convergence. In this report, we investigate the findings of ref. [1] on their method for computing preconditioners from SSPD matrix A. In particular, we focus on their first supernodal Cholesky factorization algorithm designed for matrices with naturally occurring block structures.

1. **begin function** sup_inc_chol 1($\mathcal{S}_{k,m}$)
2. **for** each child $\mathcal{C}$ of $\mathcal{S}$ in the elimination tree **do**
3. sup_inc_chol 1($\mathcal{C}$);
4. **end for**
5. Construct the potential row index set of $\mathcal{S}$ as the union of its original index set and that of all supernodes that have a row index j, $k \leq j < k + m$;
6. Allocate $l \times m$ space for $\mathcal{S}$, where l is the number of potential row indices;
7. Update columns $k \ldots k + m - 1$ by supernodes that have a row index j, $k \leq j < k + m$, using level 3 BLAS;
8. Perform dense panel factorization on the $l \times m$ supernode $\mathcal{S}$;
9. Apply dropping strategy to reduce l to l_{final};
10. Compact and store $l_{final} \times m$ supernode $\mathcal{S}$;
11. **end function** sup_inc_chol 1.

Figure 1: Incomplete Cholesky factorization with static supernodes algorithm [1]

The supernodal incomplete Cholesky algorithm for preconditioner generation is motivated by how the Cholesky factorization accesses column nodes, the overhead from indirect addressing of SSPD matrix A, and the memory advantages obtained from level 3 BLAS routines with dense blocking. We introduce this motivation and explain some priors such as supernodal elimination trees [2] in the background section. In Matlab, we implement the above algorithm along with several comparable to illustrate a proof of correctness and to support the motivating claims. Partial results are shown in the methods section. Last, we experiment with the dropping strategies used in the incomplete factorization for both randomized and structured matrices. Our findings and the analysis are in the experiments section.

Background

Recall that the Cholesky factorization is a special case of the LU decomposition for symmetric positive definite (SPD) matrices where we factor $A = LL^T$ for lower-triangular matrix L.

```
1.  for k = 1 to n do
2.      l_kk = sqrt(l_kk)
3.      for each i s.t. l_ik ≠ 0 do
4.          l_ik = l_ik / l_kk
5.      for each j s.t. l_jk ≠ 0 do
6.          l_*j = l_*j - l_jk * l_*k
```

```
1.  for k = 1 to n do
2.      cdiv(k)
5.      for each j s.t. l_jk ≠ 0 do
6.          cmod(j, k)
```

Figure 2: Pseudo-code for right-looking Cholesky factorization where matrix L is initially the lower triangle portion of matrix A [3]

For SSPD matrix A, we see that the column updates in L depend on if coefficients L_{jk} from previously computed columns are non-zero. In the sparse case where most elements are zero, it is advantageous to create a dependency graph for matrix columns. A specific case of this dependency graph is the elimination tree where nodes represent columns and edges represent a parent/child dependency between two columns. i.e. a child node is a column that must be fully updated before the parent node or column can be updated.

$\pi(i) := 0$ for all i For $i = 1$ to n $\quad \mathcal{L}_i := \mathcal{A}_i$ $\quad$ For all j such that $\pi(j) = i$ $\quad\quad \mathcal{L}_i := (\mathcal{L}_i \cup \mathcal{L}_j) \setminus \{j\}$ $\quad \pi(i) := \min(\mathcal{L}_i \setminus \{i\})$	• $\mathcal{A}_i$ is the set of non-zero pattern (index) of columns j below the diagonal. • $\mathcal{L}_j$ is the set of non-zero pattern (index) of columns j below and including the diagonal. • min $\mathcal{L}_j$ is the smallest element of $\mathcal{L}_j$ • $\pi(i)$ defines the parent function in the elimination tree

Figure 3: Symbolic factorization via elimination tree[1]

Generating the elimination tree is done in the symbolic factorization phase so that a post-order traversal of the elimination tree in a numerical factorization phase gives the correct ordering of columns to update. Also, $\mathcal{L}_j$ indicates the row indices in column j that will be updated and is indicative of potential fill-ins in the final factorization L.

For the numerical factorization phase, a left-looking Cholesky factorization algorithm is performed during the post-order traversal of the elimination tree. From the row indices in $\mathcal{L}_j$, we can form a dense sub-column j and update it by searching leftwards for columns that have a non-zero L_{jk} element. A more efficient method reuses the elimination tree to search for nodes in the sub-tree of j instead.

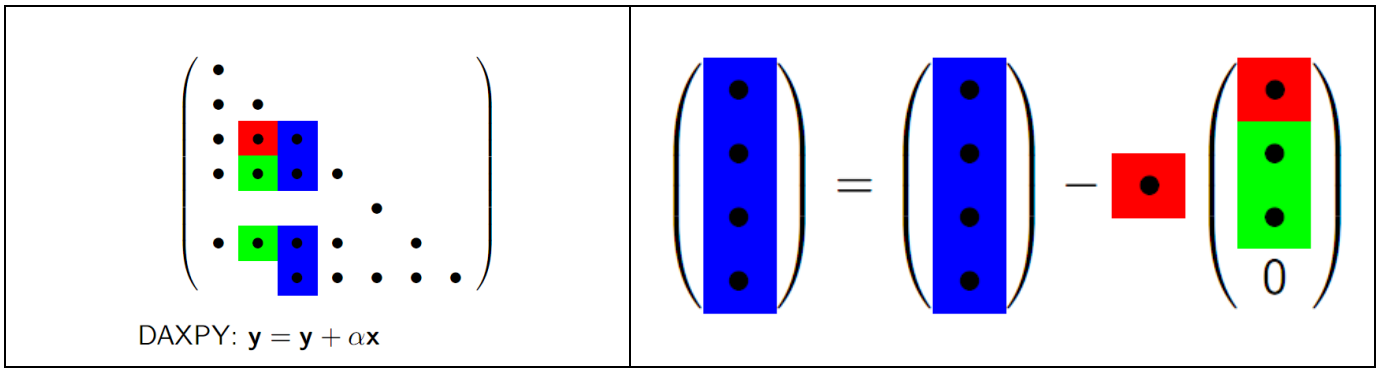

Figure 4: Level 1 BLAS updates (poor cache efficiency) for left-looking Cholesky columns [2]

The supernodal symbolic factorization generalizes the single column structure of the elimination tree. Rather than tracking a single column per node, a supernode may contain multiple adjacent columns that share a similar non-zero pattern structure. The first constraint is that adjacent columns in a supernode must first be parent nodes from the single elimination tree. A second constraint is an overlap criteria which defines a percentage threshold for the number of common non-zero row indices shared between adjacent columns. We construct the supernodal elimination tree by merging sub-trees from the original elimination tree with respect to the above constraints.

In the supernodal numerical factorization phase, a similar left-looking Cholesky factorization algorithm is performed during the post-order traversal of the supernodal elimination tree. We form dense column blocks and update it by searching leftwards or down the supernodal sub-tree for columns that have a non-zero L_{jk} where j is one of the column index in the current supernode.

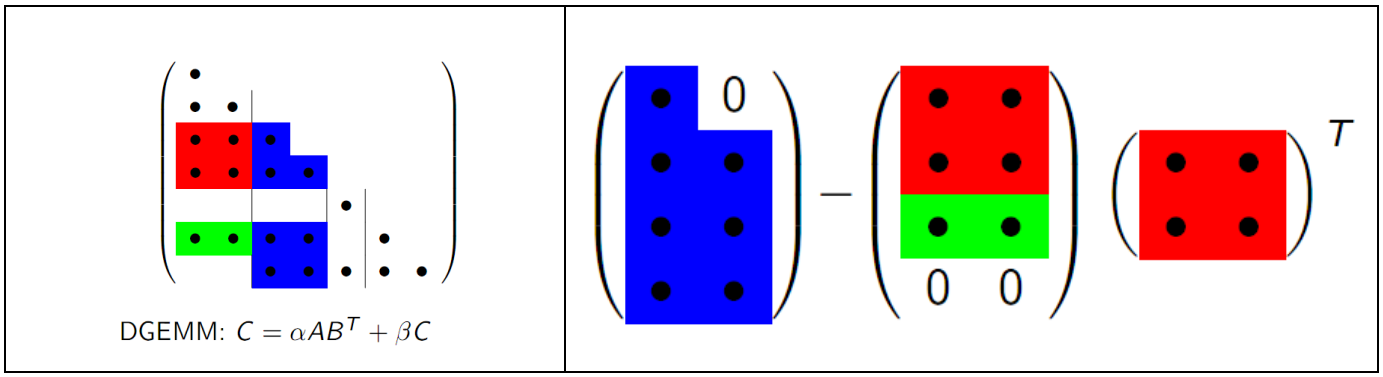

Figure 5: Level 3 BLAS updates (superior cache efficiency) for left-looking Cholesky supernodes [2]

Methods

Adapting the supernodal Cholesky factorization algorithm to an incomplete factorization in the static blocking case is done by employing a dropping strategy for fill-in elements during numerical factorization. In ref. [1], the threshold τ and γ are used to represent a drop threshold and a target number of rows per supernode. The drop score for each row is computed once a supernode is fully updated. If the drop score falls below τ, then the row within the supernode is zeroed or removed. If the number of rows remaining is greater than γ times the number of initial non-zero rows in the supernode prior to any updating, then a less restrictive drop threshold is computed.

$$dscr_{\mathcal{S}}(i) = \frac{1}{m} \sum_{j=k,k+m-1} \left| \frac{L(i,j)}{L(j,j)} \right|, \quad i \geq k+m.$$

$$\tau_{\mathcal{S}} = \frac{d_{max} \times d_{min}}{d_{min} + \frac{\gamma \times l_{initial}}{l_{remain}} (d_{max} - d_{min})}.$$

Figure 6: A drop score (top) is computed for each row i where k = leftmost column index of supernode I, m = number of columns. In the second drop threshold (bottom), d_{max} = maximum drop score <= 1, d_{min} =minimum drop score >= τ

For the Matlab implementation, we wrote both the symbolic and numerical factorization steps for the single elimination tree and supernodal algorithms from scratch (No open source code available before this). To demonstrate proof of correctness, we generate a random 500x500 SSPD matrix *A* for input. For the direct Cholesky factorization, we use Matlab's *chol* command as a base case. For incomplete Cholesky, we use Matlab's *inf-cholinc* for a base-case. The residual $norm_2(A\text{-}LL^T)$ is used to determine accuracy.

Matlab cmds:	A = sprandsym(500, .1 , abs(rand(500,1))); etree_cholsolve_test(A, 500, .1, 1) sn_cholsolve_test(A, 500, .1, .9, 1e-7, 3, 1)

For direct factorizations, we test the following:

- Built-in matlab *chol*
- Naïve left-looking Cholesky
- Single elimination tree Cholesky
- Supernodal direct Cholesky

For incomplete factorizations, we test the following:

- Built-in matlab *cholinc* with *inf* as drop parameter (no fill-ins, handles diagonal)
- Supernodal incomplete Cholesky with decreasing τ and fixed γ

Method	**Residual norm$_2$(A-LL^T)**
chol	7.8076e-016
cholesky_naive	1.5585e-015
etree_cholsolve direct	1.5585e-015
sn_cholsolve direct	6.8895e-016
inf-cholinc	0.5557
sn_cholsolve incomplete (τ = 1e-1, γ = 3)	0.5447
sn_cholsolve incomplete (τ = 1e-3, γ = 3)	0.0226
sn_cholsolve incomplete (τ = 1e-5, γ = 3)	1.3270e-004
sn_cholsolve incomplete (τ = 1e-7, γ = 3)	8.9588e-007

Table 1: Residuals of computed factors of A from various implemented algorithms

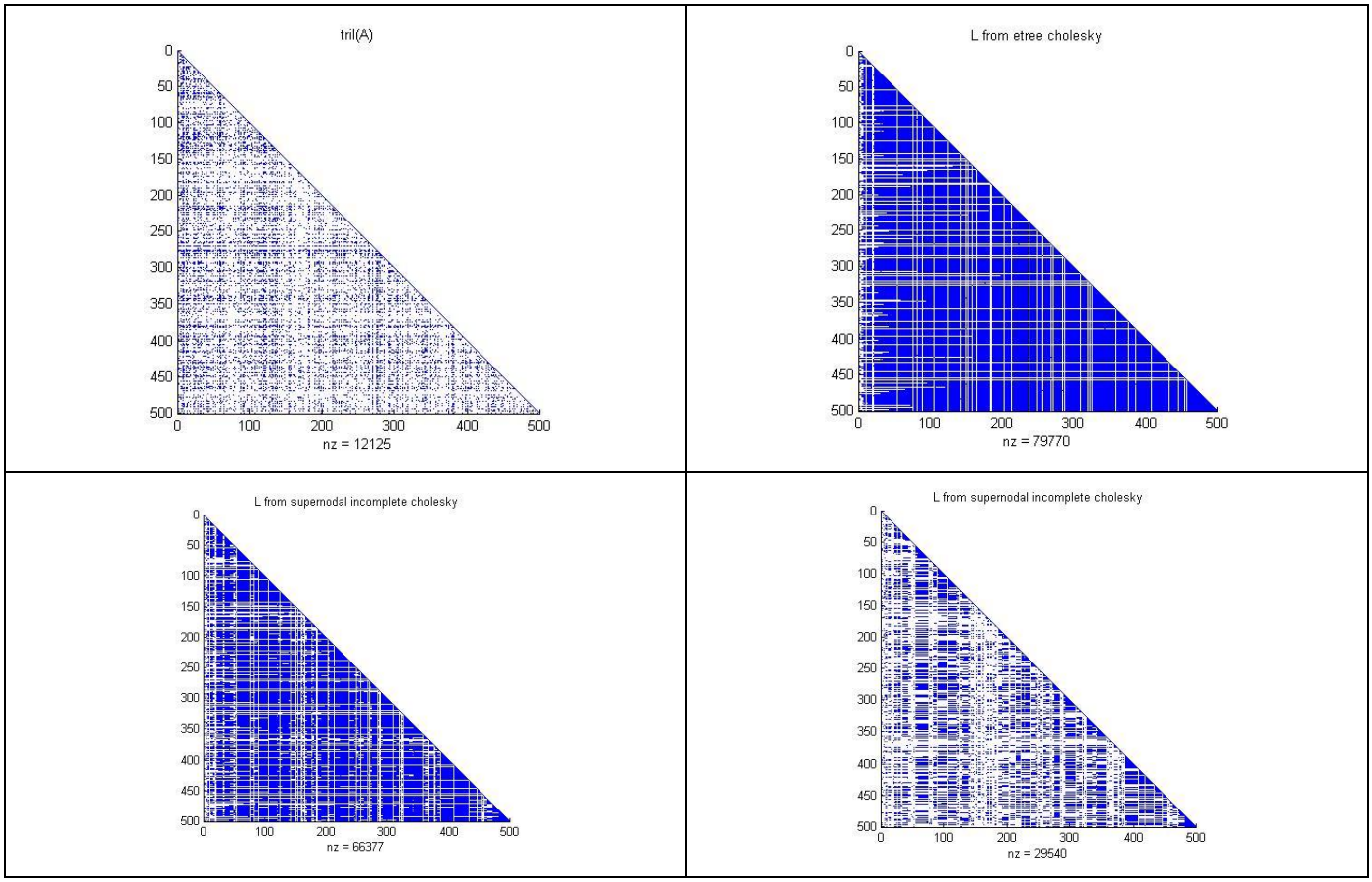

Figure 7: Non-zero pattern in lower triangle of random 500x500 SSPD matrix A (top left), direct factorization L (top right), incomplete factorization τ = 1e-3, γ = 3 (bottom left), incomplete factorization τ = 1e-5, γ = 3 (bottom right)

From the direct factorization results, we see that the naïve and single elimination tree Cholesky algorithms both produced a comparable factorizations with near accuracy to *chol*. For the supernodal direct Cholesky, it obtained an even more accurate factorization than *chol*.

From the incomplete factorization results, we see that as the dropping tolerance decreases, the factorization from the supernodal Cholesky algorithm converges to the direct factorization. This can also be seen in the figure of non-zero elements above.

Experiments

The first experiment is done with the same random 500x500 SSPD matrix. We give a run-time analysis of the symbolic and numerical factorization phases for the various direct algorithms implemented. This is to show improving runtime performances as we make better use of the sparsity structure.

We see that a naïve left-looking Cholesky implementation, which made no use of sparsity and non-zero patterns, completes the factorization with the longest time. Next, a single elimination tree Cholesky factorization that makes no use of dense columns during the numerical factorization step shows that the elimination tree does reduce the search time for column updating. The algorithm is expanded to include dense column updates during the numerical factorization phase. The results indicate that dense blocking, even along a single dimension, improves performance. Lastly, the supernodal Cholesky algorithm shows how better caching with BLAS 3 routines reduces runtimes.

Experiment 1:	Direct factorizations
Matrix Type:	Random (SSPD) matrix A
Properties:	Dim A = 500, density = .1, overlap criteria = 0-90%
Matlab cmds:	A = sprandsym(500, .1 , abs(rand(500,1))); direct_speedtests(A, 500, .1);

Method	**Symbolic Factorization**	**Numerical Factorization**
cholesky_naive	NA	9.2888
etree_cholsolve w/o dense panel	0.0250	7.3162
etree_cholsolve w/ dense panel	0.0456	3.7949
sn_cholsolve direct average	0.3390	2.7223

Table 2: Runtimes (sec) of 4 direct algorithms on [2]

The second experiment uses the same random 500x500 SSPD matrix above. We give a run-time analysis for the incomplete supernodal Cholesky algorithm with some applications to an iterative solver. In particular, we solve the matrix system Ax=ones(m,1) using the preconditioned conjugate gradient (PCG) solver and the incomplete factorization L as the preconditioner. For some base cases, we run PCG without a precondition and then with *cholinc-inf*. For the supernodal Cholesky factorization, we vary the dropping and target parameters τ and γ.

From the base cases, we see that PCG took 100 iterations to converge without a preconditioner and even longer with the *cholinc-inf*. With the supernodal incomplete Cholesky factorization, the number of iterations drops from 19 to 2 relative to an exponentially decreasing τ. The number of iterations did not vary with respect to the tested γ parameters. This reduction in iterations is compensated by the

[2] Intel Core2 6600, Matlab 2008b

increasing time needed to compute the preconditioner as a smaller dropping threshold τ indicates that the incomplete factorization approaches the direct factorization.

Experiment 2:	Incomplete factorization with PCG [Ax = ones(m,1)]
Matrix Type:	Random (SSPD) matrix A
Properties	Dim M = 500, density = .1, overlap criteria = 90%, pcg tol=1e-6
Matlab cmds:	A = sprandsym(500, .1 , abs(rand(500,1))); precon_tests(A, 500, .1, .9);

PCG + Preconditioner	**PCG Iterations**
No preconditioner	100
cholinc-inf	150

γ (Row)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**
τ (Column)					
1e-2	19	19	19	19	19
1e-3	6	6	6	6	6
1e-4	3	3	3	3	3
1e-5	2	2	2	2	2

Table 3: PCG Iterations for supernodal incomplete Cholesky with varying (τ, γ) dropping parameters

γ (Row)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**
τ (Column)					
1e-2	0.6036 0.4199	0.5977 0.4181	0.5940 0.4174	0.5978 0.4209	0.6015 0.4157
1e-3	0.5981 0.6059	0.5971 0.6160	0.6038 0.6148	0.5971 0.6157	0.5895 0.6147
1e-4	0.5902 0.9359	0.5904 0.9647	0.6032 0.9640	0.5936 0.9643	0.5966 0.9704
1e-5	0.6032 1.2308	0.5931 1.4040	0.5969 1.4291	0.5965 1.4708	0.6045 1.5038

Table 4: Runtimes (sec) for preconditioner generation from supernodal incomplete Cholesky symbolic (top) & numeric factorizations (bottom)

The third experiment repeats the first experiment with now a structured matrix (plbuckle.mat) [3]. Note that Matlab may give a warning for increasing the recursion limit.

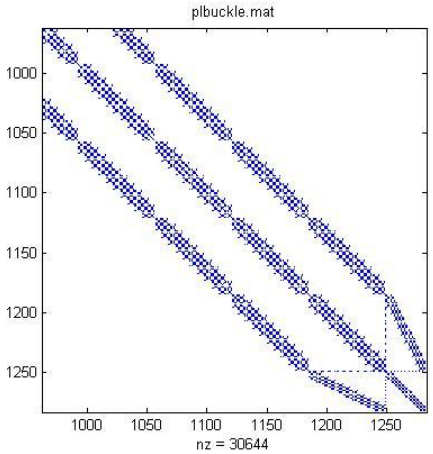

Figure 8: Non-zero pattern of plbuckle.m

[3] University of Florida Sparse Matrix Collection <http://www.cise.ufl.edu/research/sparse/matrices/>

For this matrix with a recurring block structure, the results show a significant improvement (9x) in performance when using an elimination tree over the naïve version. Improvements via dense blocking and supernodes remain similar to the first experiment. A notable difference is the longer symbolic factorization time which may be due to the lack of BLAS routines used in the implementation of this phase.

Experiment 3:	Direct factorizations
Matrix Type:	'Structured' (SSPD) matrix (plbuckle.mat)
Properties:	Dim A = 1282, overlap criteria = 0-90%

Matlab cmds:	load plbuckle;
	set(0, 'RecursionLimit', 1282);
	direct_speedtests(Problem.A, 1282, .1);

Method	**Symbolic Factorization**	**Numerical Factorization**
cholesky_naive	NA	63.6248
etree_cholsolve w/o dense panel	0.2991	7.8606
etree_cholsolve w/ dense panel	0.3373	3.1274
sn_cholsolve direct average	2.0231	2.1677

Table 5: Runtimes (sec) of 4 direct algorithms

The fourth experiment repeats the second experiment with the same structured matrix (plbuckle.mat). Note that Matlab may give a warning for increasing the recursion limit.

For this matrix with a recurring block structure, the results show that the PCG without a preconditioner or with a cholinc-inf preconditioner did not converge within 1000 iterations. When using the preconditioner from the supernodal incomplete Cholesky, the PCG solver converges to the solution within 29 to 5 iterations. We also spot some variability in iterations with respect to the γ parameter. As γ increases, the restriction on the number of remaining rows per supernode is lessened so that fewer rows are dropped. This is reflected by the increasing numerical factorization times with respect to increasing γ as fewer drops or more fill-ins imply a closer factorization to the direct Cholesky method.

Experiment 2:	Incomplete factorization with PCG [Ax = ones(m,1)]
Matrix Type:	'Structured' (SSPD) matrix (plbuckle.mat)
Properties	Dim A = 1282, overlap criteria = 90%, pcg tol=1e-6

Matlab cmds:	load plbuckle;
	set(0, 'RecursionLimit', 1282);
	precon_tests(Problem.A, 500, .1, .9);

PCG + Preconditioner	**PCG Iterations**
No preconditioner	Did not converge > 1000
cholinc-inf	Did not converge > 1000

γ (Row)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**
τ (Column)					
1e-2	29	24	24	24	24
1e-3	14	12	10	9	8
1e-4	10	9	8	6	6
1e-5	10	8	8	6	5

Table 6: PCG Iterations for supernodal incomplete Cholesky with varying (τ, γ) dropping parameters

γ (Row)	1.0	1.5	2.0	2.5	3.0
τ (Column)					
1e-2	2.3723 1.2429	2.4067 1.2645	2.3076 1.2718	2.3152 1.2767	2.3658 1.2674
1e-3	2.3479 1.4496	2.3640 1.5465	2.3635 1.7087	2.3843 1.7229	2.3619 1.7270
1e-4	2.3608 1.7805	2.3566 1.9621	2.3427 1.8973	2.3801 2.0141	2.4088 2.0671
1e-5	2.3754 2.0042	2.3762 2.1068	2.3686 2.2160	2.3897 2.1163	2.3746 2.1369

Table 7: Runtimes (sec) for preconditioner generation from supernodal incomplete Cholesky symbolic (top) & numeric factorizations (bottom)

Conclusions

From our experiments, we see that elimination trees and generalized supernodal trees enhance the runtime performance of both the direct and incomplete Cholesky factorizations for SSPD matrices. In the case of structured SSPD matrices, the use of elimination trees significantly reduces the time necessarily to update columns in the Cholesky algorithm. In the case of random SSPD matrices, the improvements were smaller as the depth of the elimination tree was more dependent on the density parameter for the number of non-zeroes elements in the matrix.

The use of dense blocking in the supernodal structure did show some improvements in performance due to memory caching from higher level BLAS routines. The amount of improvement, as mentioned in ref. [1], is related to the average width (columns) of the supernodes which tends to range from 1 to 10 in their test cases. Thus, for any significant improvements to be made, the matrix length (size) or at least the non-zero patterns along supernodes must be sufficiently large for the level 3 BLAS routines take advantage of. Such a size is not possible in the Matlab implementation of this supernodal algorithm.

The strongest results are seen in the supernodal incomplete Cholesky factorization when applied to the PCG solver. By varying parameters (τ, γ), we obtain a measurable tradeoff between the time taken to compute the preconditioner versus the number of iterations for the PCG solver to converge. The use of a secondary parameter γ also gives an estimate on how close a factorization is from one without any fill-ins to one that is close to the direct factorization. We note that adaptive techniques for altering (τ, γ) during runtime are elaborated in ref. [1] but are not verified in this report.

Open Problems

- Adapting flexible parameter selection for incomplete LU factorization
- Applying machine learning techniques for parameter selection based on features in the input matrix space
- Obtaining optimal time tradeoff such that preconditioner generation time + iterative solver time outperforms direct solver for super large systems (>1 million unknowns)

References

[1] Adaptive Techniques for Improving the Performance of Incomplete Factorization Preconditioning. Anshul Gupta and Thomas George, SIAM, February 8, 2010.

[2] Efficient Sparse Cholesky Factorization. Jonathan Hogg. J.Hogg@ed.ac.uk. University of Edinburgh. August 13, 2006.

[3] A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results, Edward Rothberg and Anoop Gupta, Technical Report STAN-CS-89- 1286, Stanford University, 1989.