Some Observations Regarding High-Performance Serial Implementations of Sparse Symmetric Factorizations

Esmond G. Ng
Lawrence Berkeley National Laboratory

Barry W. Peyton
Dalton State College

Sparse Days, June 20–22, 2022

Work supported by the U.S. Department of Energy.

Direct Solution of a Sparse SPD Linear System

- Let $Ax = b$ be a sparse symmetric positive definite (SPD) linear system.
  - To solve the system, we perform sparse Cholesky factorization by computing $A = LL^T$, where L is lower triangular.
- This talk is focused on how indexing can influence the imlementation of sparse Cholesky factorization
- The discussions will be in terms of individual columns but can be generalized to blocks of columns (i.e., supernodes) where the columns in each block have essentially the same sparsity structure.

Some Definitions and Notation

- glbind(j) is the set of *global* row indices of the nonzeros in column j of L, listed in ascending order.
  - That is, $i \in$ glbind(j) if and only if $L_{i,j} \neq 0$.

- cmod(j, k) is the modification of column j by a multiple of column k required during the factorization.
  - j – target column
  - k – source column
- cdiv(j) is the scaling of column j required during the factorization.

- The elimination tree [Liu (1990)]: For $1 \leq j \leq n$, the parent $p(j) := \min\{k > j : k \in \text{glbind}(j)\}$.

Left-Looking Sparse Factorization

for $j := 1$ **to** n **do**
  for $k < j$ **such that** $j \in \text{glbind}(k)$ **do**
    cmod(j, k); /* target j fixed; source k varying */
  cdiv(j);

- It was used in SPARSPAK and YSMP in the 1970's.
- The cmod(j, k)'s come from the left before cdiv(j) is performed.
- The key to its efficient indexing scheme is the fact that $\text{glbind}(k) \cap \{j, j+1, \ldots, n\} \subseteq \text{glbind}(j)$ whenever $j \in \text{glbind}(k)$.
- A floating-point array of size n was used to accumulate the cmod(j, k)'s using scatter-add operations.

The Problem with Right-Looking Sparse Factorization

for $k := 1$ **to** n **do**
  cdiv(k);
  for $j \in \text{glbind}(k) \setminus \{k\}$ **do**
    cmod(j, k); /* source k fixed; target j varying */

- cdiv(k) is performed and then the cmod(j, k)'s are applied to the right.
- Recall that $\text{glbind}(k) \cap \{j, j+1, \ldots, n\} \subseteq \text{glbind}(j)$ whenever $j \in \text{glbind}(k) \setminus \{k\}$.
- Not popular ... It was not known how to take care of the indexing issues efficiently, because straightforward matching of indices in $\text{glbind}(k) \cap \{j, j+1, \ldots, n\}$ with indices in glbind(j) requires searching the larger set glbind(j).

High-Performance Sparse Factorization

- The breakthrough came from the multifrontal method [Duff and Reid (1983)], which provided ...
  - ...the first solution to the indexing difficulties associated with right-looking sparse factorization, and
  - ...the first solution to blocking the factorization to increase factorization speed.

- The multifrontal method opened the door to using the supernodal structure in the sparse Cholesky factor to create block factorization algorithms that can use level-3 BLAS kernels.
  - [Duff and Reid (1983)]
  - [Ashcraft, Grimes, Lewis, Peyton, and Simon (1987)]
  - [Rothberg and Gupta (1991)]
  - [Ng and Peyton (1993)]

Relative Indices

- Relative indices (also known as *local indices*) were first described in [Schreiber (1982)].
- Their use in the multifrontal method was introduced in [Ashcraft (1987)].

- In the set of indices of k relative to its parent $j = p(k)$, namely, relind(k, j), each global index $i \in \text{glbind}(k) \setminus \{k\}$ is represented by the location of i *relative* to the bottom of the list glbind(j).

Relative Indices ... an Example

k	$dist$	$j = p(k)$
$\otimes$		
$\times$	6	$\otimes$
	5	$\times$
$\times$	4	$\times$
	3	$\times$
$\times$	2	$\times$
$\times$	1	$\times$
	0	$\times$

Relative Indices ... an Example

k	$r_{k,j}$	$j = p(k)$
$\otimes$		
$\times$	**6**	$\otimes$
	5	$\times$
$\times$	**4**	$\times$
	3	$\times$
$\times$	**2**	$\times$
$\times$	**1**	$\times$
	0	$\times$

$$r_{k,j} = \text{relind}(k, j)$$

- Among other things, $\text{relind}(k, j)$ enables one to incorporate $\text{cmod}(j, k)$ directly into column j of the factor.

Generalizing Relative Indices

- Let j be an *ancestor* of k in the elimination tree.
- In the set of indices of k relative to j, namely, $\text{relind}(k, j)$, each global index $i \in \text{glbind}(k) \cap \{j, j+1, \ldots, n\}$ is represented by the location of i relative to the bottom of the list $\text{glbind}(j)$.
- Generalized relative indices were used in [Ng and Peyton (1993)] in the implementation of
  - a left-looking algorithm, and
  - a multifrontal algorithm

A New Right-Looking Factorization Algorithm

- It is possible to implement a truly right-looking factorization by using the generalized relative indices.
- Essential to an efficient implementation of the new algorithm is computing (via integer gather operations) relind(k, j) for a *chain* of ancestors j of k for each k.

Right-Looking Factorization – a Small Example

k	j_1	j_2	j_3
$\otimes$			
$\times$	$\otimes$		
	$\times$	$\otimes$	
$\times$	$\times$	$\times$	$\otimes$
	$\times$	$\times$	$\times$
		$\times$	$\times$
$\times$	$\times$	$\times$	$\times$
			$\times$
$\times$	$\times$	$\times$	$\times$
			$\times$
	$\times$	$\times$	$\times$
		$\times$	$\times$
			$\times$
			$\times$

$$j_1 = p(k) \; ; \; j_2 = p(j_1) \; ; \; j_3 = p(j_2)$$

Constructing the Relative Indices

k	r_{k,j_1}	j_1	r_{j_1,j_2}	j_2	r_{j_2,j_3}	j_3
$\otimes$						
$\times$	**6**	$\otimes$				
	5	$\times$	**7**	$\otimes$		
$\times$	**4**	$\times$	**6**	$\times$	**10**	$\otimes$
	3	$\times$	**5**	$\times$	**9**	$\times$
			4	$\times$	**8**	$\times$
$\times$	**2**	$\times$	**3**	$\times$	**7**	$\times$
					6	$\times$
$\times$	**1**	$\times$	**2**	$\times$	**5**	$\times$
					4	$\times$
	0	$\times$	**1**	$\times$	**3**	$\times$
			0	$\times$	**2**	$\times$
					1	$\times$
					0	$\times$

Right-Looking Factorization – After cdiv(k) ...

k	r_{k,j_1}	j_1	r_{j_1,j_2}	j_2	r_{j_2,j_3}	j_3
$\otimes$						
$\times$	6	$\otimes$				
		$\times$	7	$\otimes$		
$\times$	4	$\times$	6	$\times$	10	$\otimes$
		$\times$	5	$\times$	9	$\times$
				$\times$	8	$\times$
$\times$	2	$\times$	3	$\times$	7	$\times$
						$\times$
$\times$	1	$\times$	2	$\times$	5	$\times$
						$\times$
		$\times$	1	$\times$	3	$\times$
				$\times$	2	$\times$
						$\times$
						$\times$

- Perform cmod(j_1, k) and cmod(j_3, k) ... but not cmod(j_2, k).

Right-Looking Factorization – After cdiv(k) ...

k	r_{k,j_1}	j_1	r_{j_1,j_2}	j_2	r_{j_2,j_3}	j_3
$\otimes$						
$\times$	6	$\otimes$				
		$\times$	7	$\otimes$		
$\times$	4	$\times$	6	$\times$	10	$\otimes$
		$\times$	5	$\times$	9	$\times$
				$\times$	8	$\times$
$\times$	2	$\times$	3	$\times$	7	$\times$
						$\times$
$\times$	1	$\times$	2	$\times$	5	$\times$
						$\times$
		$\times$	1	$\times$	3	$\times$
				$\times$	2	$\times$
						$\times$
						$\times$

- Perform cmod(j_1, k) and cmod(j_3, k) ... but not cmod(j_2, k).
  - cmod(j_1, k): need relind(k, j_1), which we have.
  - cmod(j_3, k): need relind(k, j_3), which we don't have.

Right-Looking Factorization – cmod(j_3, k) ...

k	r_{k,j_1}	j_1	r_{j_1,j_2}	r_{k,j_2}	j_2	r_{j_2,j_3}	j_3
$\otimes$							
$\times$	6	$\otimes$					
		$\times$	7		$\otimes$		
$\times$	**4**	$\times$	**6**	**6**	$\times$	10	$\otimes$
		$\times$	5		$\times$	9	$\times$
					$\times$	8	$\times$
$\times$	**2**	$\times$	**3**	**3**	$\times$	7	$\times$
							$\times$
$\times$	**1**	$\times$	**2**	**2**	$\times$	5	$\times$
							$\times$
		$\times$	1		$\times$	3	$\times$
					$\times$	2	$\times$
							$\times$
							$\times$

- Need relind(k, j_3) to perform cmod(j_3, k) ...
  - relind$(k, j_2) =$ relind$(k, j_1) \circ$ relind(j_1, j_2)

Right-Looking Factorization – cmod(j_3, k) ...

k	r_{k,j_1}	j_1	r_{j_1,j_2}	r_{k,j_2}	j_2	r_{j_2,j_3}	r_{k,j_3}	j_3
⊗								
×	6	⊗						
		×	7		⊗			
×	4	×	6	**6**	×	**10**	**10**	⊗
		×	5		×	9		×
					×	8		×
×	2	×	3	**3**	×	**7**	**7**	×
								×
×	1	×	2	**2**	×	**5**	**5**	×
								×
		×	1		×	3		×
					×	2		×
								×
								×

- Need relind(k, j_3) to perform cmod(j_3, k) ...
  - relind$(k, j_2) =$ relind$(k, j_1) \circ$ relind(j_1, j_2)
  - relind$(k, j_3) =$ relind$(k, j_2) \circ$ relind(j_2, j_3)

Right-Looking Factorization – cmod(j_3, k) ...

k	r_{k,j_1}	j_1	r_{j_1,j_2}	r_{k,j_2}	j_2	r_{j_2,j_3}	r_{k,j_3}	j_3
$\otimes$								
$\times$	6	$\otimes$						
		$\times$	7		$\otimes$			
$\times$	4	$\times$	6	**6**	$\times$	**10**	**10**	$\otimes$
		$\times$	5		$\times$	9		$\times$
					$\times$	8		$\times$
$\times$	2	$\times$	3	**3**	$\times$	**7**	**7**	$\times$
								$\times$
$\times$	1	$\times$	2	**2**	$\times$	**5**	**5**	$\times$
								$\times$
		$\times$	1		$\times$	3		$\times$
					$\times$	2		$\times$
								$\times$
								$\times$

- Need relind(k, j_3) to perform cmod(j_3, k) ...
  - relind(k, j_2) = relind$(k, j_1) \circ$ relind(j_1, j_2)
  - relind(k, j_3) = relind$(k, j_2) \circ$ relind(j_2, j_3)
  - These involve simple integer **gather** operations.

A New Right-Looking Factorization Algorithm

Compute relind$(k, p(k))$ for every parent-child pair;
for $k := 1$ **to** n **do**
    cdiv(k);
    $j := p(k)$;
    while there exists a column $i > k$ requiring cmod(i, k) **do**
        if $j \neq p(k)$ **then**
            relind$(k, j) :=$ relind$(k, j') \circ$ relind(j', j); /* $j = p(j')$ */
        if cmod(j, k) is required **then**
            Use relind(k, j) to perform cmod(j, k);
        $j' := j$; $j := p(j')$; /* Next ancestor j in the chain */

- We have created a *supernodal* (blocked) version of this algorithm.
  - cdiv() and cmod() can be extended to operate on supernodes.

A New Right-Looking Factorization Algorithm

Some Important Details and Observations

- An obvious inefficiency specific to this algorithm is the computation of relind(k, j) for ancestors j that are *not* modified by column k.
  - Recall relind(k, j_2) in the example used to illustrate the algorithm.
- These "wasted" gather operations are necessary, nonetheless, to traverse the required chain of ancestors.

A New Right-Looking Factorization Algorithm

Some Important Details and Observations

- An obvious inefficiency specific to this algorithm is the computation of relind(k, j) for ancestors j that are *not* modified by column k.
  - Recall relind(k, j_2) in the example used to illustrate the algorithm.
- These "wasted" gather operations are necessary, nonetheless, to traverse the required chain of ancestors.
- The column-based version of this algorithm is of no practical use because there would be far too many "wasted" gather operations.
- The supernodal version of the algorithm is competitive, however, because the number of "wasted" gather operations is small enough to be negligible in practice (as we shall see).

A New Right-Looking Factorization Algorithm

Some Important Details and Observations

- An obvious inefficiency specific to this algorithm is the computation of relind(k, j) for ancestors j that are *not* modified by column k.
  - Recall relind(k, j_2) in the example used to illustrate the algorithm.
- These "wasted" gather operations are necessary, nonetheless, to traverse the required chain of ancestors.
- The column-based version of this algorithm is of no practical use because there would be far too many "wasted" gather operations.
- The supernodal version of the algorithm is competitive, however, because the number of "wasted" gather operations is small enough to be negligible in practice (as we shall see).
- The supernodal version of the algorithm needs a block of floating-point working storage the size of the largest possible update matrix in the multifrontal method – no stack storage required.

Experimental Setup

- Test matrices
  - 26 matrices of order $n \geq 500{,}000$
- Ordering and symbolic factorization methods
  - METIS nested dissection [Karypis and Kumar (1999)]
  - Fundmental supernode partition [Liu, Ng, and Peyton (1993)]
- Factorization methods
  - Multifrontal (**MF**): the basic code from [Ng and Peyton (1993)]
  - Left-looking block factorization (**LL**) [Ng and Peyton (1993)]
  - Right-looking block factorization (**RL**) [Ng and Peyton]
- Statistics reported
  - Factorization times
  - Floating-point auxiliary storage (e.g., MF stack storage)
  - The number of gathers versus the number of supernodal cmod()'s for RL

Experimental Setup

- Test matrices
  - The matrices were taken from the SuiteSparse matrix collection [Davis and Hu (2011)].
  - We chose symmetric matrices, where each is connected with an actual linear system to be solved.
  - To limit the time required by the experiments, we did not include the largest or densest matrices in the collection.
- Factorization methodology
  - We are aware that our very basic implementation of the multifrontal method lacks many of the standard enhancements needed to make it a fully competitive method.
  - All three factorization codes are built around calls to BLAS3 routines (namely, DPOTRF, DTRSM, DSYRK, and DGEMM).
  - To achieve good performance, the Intel MKL BLAS were used.

Test Matrices

Matrix	N	$\|A\|$	Kind
nd24k	72,000	28,715,634	ND 2D/3D
af_0_k101	503,625	17,550,675	Structural
inline_1	503,712	36,816,342	Structural
af_shell1	504,855	17,588,875	Structural
bundle_adj	513,351	20,208,051	Coputer Vision
parabolic_fem	525,825	3,674,625	CFD
lp1	534,388	1,643,420	Optimization
gsm_106857	589,446	21,758,924	Electromagnetics
Fault_639	638,802	28,614,564	Structural
apache2	715,176	4,817,870	Structural
tmt_sym	726,713	5,080,961	Electromagnetics
PFlow_742	742,793	37,138,461	FEM
CurlCurl_2	806,529	8,921,789	Model Reduction
boneS10	914,898	55,468,422	Model Reduction
audikw_1	943,695	77,651,847	Structural
ldoor	952,203	46,522,475	Structural
bone010	986,703	71,666,325	Model Reduction
ecology1	1,000,000	4,996,000	Ecology
dielFilterV3real	1,102,824	89,306,020	Electromagnetics
dielFilterV2real	1,157,456	48,538,952	Electromagnetics
CurlCurl_3	1,219,574	13,544,618	Model Reduction
thermal2	1,228,045	8,580,313	Thermal
StocF-1465	1,465,137	21,005,389	CFD
Hook_1498	1,498,023	60,917,445	Structural
af_shell10	1,508,065	52,672,325	Structural
G3_circuit	1,585,478	7,660,826	Circuit Simulation

Factorization Time and Floating-Point Auxiliary Storage

- The results are relative to LL.

	Factorization Time			Auxiliary Storage		
	MF	**LL**	**RL**	**MF**	**LL**	**RL**
average	1.216	1.000	1.070	187.132	1.000	4.723
geo. mean	1.184	1.000	1.036	8.851	1.000	2.930
median	1.140	1.000	0.997	5.804	1.000	2.410
minimum	0.879	1.000	0.684	3.321	1.000	1.736
maximum	2.977	1.000	2.814	2650.978	1.000	50.087

- For auxiliary storage . . .
  - The averages and, to a lesser extent, the geometric means are skewed by two very extreme outliers.
  - Consequently, the medians give the most meaningful comparisons.
  - Both of the outliers are anomalously structured extremely sparse matrices that required less than a second to factor.

The Ratio of Gathers to (Supernodal) cmod()'s in RL

Gathers/cmod()'s	
	RL
average	1.792
geo. mean	1.569
median	1.798
minimum	0.031
maximum	2.641

- The maximum ratio is only 2.641.
- For only 6 matrices is the ratio ≥ 2.
- For the remaining 20 matrices, the ratio is < 2.
- For 25 of the 26 matrices, the ratio is > 1.

Summary

- We have introduced a new right-looking block factorization algorithm that exploits generalized relative indexing.
- We have implemented the new algorithm, and we have shown that it has competitive runtimes and competitive floating-point auxiliary storage requirements.
- We have shown that the time that the new algorithm spends on "wasted" gathers is negligible.

Summary

- We have introduced a new right-looking block factorization algorithm that exploits generalized relative indexing.
- We have implemented the new algorithm, and we have shown that it has competitive runtimes and competitive floating-point auxiliary storage requirements.
- We have shown that the time that the new algorithm spends on "wasted" gathers is negligible.

- Extension to factorization of sparse symmetric indefinite matrices:
  - During a right-looking block factorization, the Schur complement is kept up-to-date, which is not the case during either a multifrontal or a left-looking block factorization.
  - Thus, within the framework of right-looking block factorization, we may be able to devise ways of factoring sparse symmetric indefinite linear systems using Bunch-Kaufman.