The robust and efficient partial factorization of dense symmetric indefinite matrices

John Reid and Jennifer Scott

STFC Rutherford Appleton Laboratory, UK

Householder Symposium 2011, Lake Tahoe

Motivation

- The partial factorization of dense matrices lies at the heart of frontal and multifrontal solvers.
- In particular, we want to robustly and efficiently perform the partial or complete factorization (and subsequent solve) of dense symmetric indefinite matrices.

Motivation

- The partial factorization of dense matrices lies at the heart of frontal and multifrontal solvers.
- In particular, we want to robustly and efficiently perform the partial or complete factorization (and subsequent solve) of dense symmetric indefinite matrices.

Three components/challenges:

- pivoting $\rightarrow$ stability (robustness)
- blocking $\rightarrow$ efficiency (time)
- packing $\rightarrow$ efficiency (memory)

Partial factorization

Dense A of order n is of the form

$$\begin{pmatrix} A_1 & A_{21}^* \\ A_{21} & A_2 \end{pmatrix}$$

where A_1 is order $p \leq n$ and pivots restricted to A_1.
Partial factorization takes the form

$$PAP^* = \begin{pmatrix} L_1 & \\ L_2 & I \end{pmatrix} \begin{pmatrix} D & \\ & S_2 \end{pmatrix} \begin{pmatrix} L_1^* & L_2^* \\ & I \end{pmatrix}$$

- P is a permutation matrix
- L_1 is unit lower triangular of order $q \leq p$
- D is block diagonal of order q, with each diagonal block of size one or two.

Aim

- Exploit symmetry and store only half of A.
- Use standard high level BLAS (note: there is no BLAS 3 for A in packed triangular form).
- Incorporate threshold partial pivoting.

What's in LAPACK?

- Complete factorization only ($p = n$)
- Two subroutines that use Bunch-Kaufman pivoting
  - `_sytrf` stores whole of A (but does use BLAS)
  - `_sptrf` does not use BLAS (and was 10 times slower than `_sytrf` in our tests)

Note on Bunch-Kaufman pivoting ('77)
Ashcraft, Grimes and Lewis ('98): Bunch-Kaufman solvers are norm-wise backward stable but can produce inaccurate results since L is not bounded. Thus LAPACK routines can be unstable.

Pivoting and stability

Choose pivots one-by-one, with aim of limiting size of the entries in L:

$$|l_{ij}| < u^{-1},$$

where $u \in [0, 1]$ is user-set threshold.

Ashcraft et al: bounding l_{ij} plus backward stable scheme for 2×2 linear systems $\Rightarrow$ backward stability of entire process.

Pivoting and stability

Choose pivots one-by-one, with aim of limiting size of the entries in L:

$$|l_{ij}| < u^{-1},$$

where $u \in [0, 1]$ is user-set threshold.

Ashcraft et al: bounding l_{ij} plus backward stable scheme for 2×2 linear systems $\Rightarrow$ backward stability of entire process.

Usual threshold test for 1×1 pivot:

$$|a_{q+1,q+1}| > u \max_{i>q+1} |a_{i,q+1}|.$$

Test for 2×2 pivot (Duff, Gould, Reid, Scott and Turner '91):

$$\left| \begin{pmatrix} a_{q+1,q+1} & a_{q+1,q+2} \\ a_{q+1,q+2} & a_{q+2,q+2} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i>q+2} |a_{i,q+1}| \\ \max_{i>q+2} |a_{i,q+2}| \end{pmatrix} < \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}$$

(absolute value notation refers to matrix of corresponding absolute values).

Notes

- $u = 0$ is interpreted as requiring that the pivot be nonsingular.
- Our default value is $u = 0.1$ (although 0.01 is used within our multifrontal codes).
- Proposed strategy is the symmetric equivalent of rook pivoting in that the pivot is compared with other entries in its rows and columns.
- In our implementation, we found it is beneficial to select 2×2 pivots over 1×1 pivots (BLAS 3).

Simple factorization: no blocking

q is number of pivots chosen so far;
m is index of column to be searched for a pivot.

Input: A, p
Output: q, P, L and D
Initialise: $q = 0$; $m = 0$
do while $(q < p)$
    call find_pivot(m,piv_size)
    if $(piv\_size == 0)$ **exit** ! Failed to find a pivot
    $q = q + piv\_size$
    update the active columns $q + 1$ to m
end do
apply outstanding updates to columns $p+1$ to n

Finding a pivot

This involves incrementing m then

- updating column m with pivots found so far (use `_gemv`)
- searching column m for 1×1 pivot, or
- searching columns $q+1$ to $m-1$ for a partner k to make (k, m) a 2×2 pivot
- then permuting pivot row(s)/col.(s) to positions q+1 (and q+2)

Storage

- intensive activity in columns $q+1$ to m
- want to hold these columns contiguously in memory
- for pivot searching and row/column interchanges, easier and more efficient if each column held contiguously.

Block column format ($n = 10$, $nb = 3$)

1	#	#							
2	12	#							
3	13	23							
4	14	24	31	#	#				
5	15	25	32	39	#				
6	16	26	33	40	47				
7	17	27	34	41	48	52	#	#	
8	18	28	35	42	49	53	57	#	
9	19	29	36	43	50	54	58	62	
10	20	30	37	44	51	55	59	63	64

Factorization with blocking

- Prior to looking for pivots, rearrange A to block column format.
- Modify factorization to allow for block columns
  - update a block column when it is needed for pivoting with pivots chosen so far
  - can be done with `gemm`
  - avoids updating single candidate column (reduces use of Level 2 BLAS).

Factor storage with blocking

After partial factorization, rearrange computed factor columns $\begin{pmatrix} L_1 \\ L_2 \end{pmatrix}$

- limit storage (do not waste storage by holding blocks on diagonal as full matrices but pack L_1)
- allow use of BLAS in solve phase by holding columns of L_2 contiguously.

Storage after rearrangement ($n = 10$, $nb = 3$ and $q = 8$).

1							
2	4						
3	5	6					
7	14	21	28				
8	15	22	29	31			
9	16	23	30	32	33		
10	17	24	34	38	42	46	
11	18	25	35	39	43	47	48
12	19	26	36	40	44	49	51
13	20	27	37	41	45	50	52

Inner block

Experiments suggest large block size (eg $nb = 96$) is good choice.
But large block means `gemv` (BLAS 2) used for significant part of computation

- use inner block nbi (typically $nbi = nb/6$)
- whenever multiple of nbi columns of L found, `gemm` used to update the rest of the block column.

Factorization algorithm with 2-level blocking

Input: A, p
Output: q, P, L and D
Initialise: $q = 0$; $m = 0$
do while $(q < p)$
    call find_pivot(m,piv_size)
    if ($piv\_size == 0$) **exit** ! Failed to find a pivot
    $q = q + piv\_size$
    update the active columns $q + 1$ to end of inner block
    if (outer block complete) **then**
        update outer block columns
    else if (inner block complete) **then**
        update inner block columns
    end if
end do
apply outstanding updates to columns p+1 to n

Parallel working

- When n and p are large, most of the work is done by `gemm` when updating trailing matrix by block columns.
- Improve performance by optionally using OpenMP **parallel do**.
- Break each block column into blocks of nb rows and loop over these.

Numerical experiments

Our new partial factorization code for symmetric indefinite dense linear systems is `HSL_MA64`.

- All codes written in Fortran 95.
- Experiments performed using double precision arithmetic on 2-way quadcore Harpertown machine (2×4 cores).
- Compiler is Intel 11.0 with option -fast, Intel BLAS and LAPACK.
- Times are all in seconds.
- Redundant floating-point operations in the upper-triangular part of the matrix are excluded when computing speeds in Gflop/s.

Comparisons with LAPACK (complete factorization)

	Wall-clock Gflop/s					No. 2×2 pivots		
	`dsytrf`	`HSL_MA64`				`dsytrf`	`HSL_MA64`	
	1 thread	1 thread		8 threads				
favour:		2×2	1×1	2×2	1×1		2×2	1×1
n								
1000	5.6	6.1	5.9	10.5	9.1	312	369	87
4000	7.0	7.6	7.5	29.9	28.4	1403	1473	366
8000	7.3	7.8	7.8	38.0	36.1	2902	2910	758
16000	7.2	7.9	7.8	43.2	39.9	6050	5830	1611

Notes:

- On 1 thread, competitive with `dsytrf` in terms of speed.
- Saves half the memory.
- Peak speed of `dgemm` on 1 thread is 9.3 Gflop/s

HSL_MA64 performance (partial factorization)

Speed (wall-clock Gflop/s) and speed-up with $n = 4000$, $nb = 96$ and $nbi = 16$.

	1 thread	4 threads		8 threads	
p	Speed	Speed	Speed-up	Speed	Speed-up
128	6.5	14.5	2.2	17.5	2.7
512	7.6	21.0	2.6	29.7	3.9
2048	7.8	22.4	2.9	32.8	4.2

Multifrontal performance

Recall: key motivation for HSL_MA64 was for partial factorization of the dense frontal matrices within a multifrontal sparse solver.

Here HSL_MA64 used within the multifrontal code HSL_MA77 (Reid and Scott '09). f_{max} is maximum frontsize.

Problem	$N/10^3$	f_{max}	1 thread			8 threads		
			MA77 Time	MA64 Time	MA64 Gflop/s	MA77 Time	MA64 Time	MA64 Gflop/s
bratu3d	27.8	1521	4.2	3.4	3.7	3.7	2.9	4.2
qa8fk	66.1	2075	4.4	3.5	6.2	3.0	2.0	10.4
Si5H12	19.9	5551	43.7	32.8	4.7	38.1	26.5	5.8
NICE20MC	71.6	11688	945	690	7.6	460	160	32.7
SiO2	15.5	21406	2760	2298	5.8	1498	1033	12.8

Other features of HSL_MA64

- Handles singular systems: when column m is searched, if its largest entry has magnitude less than *small*, the row and column are set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no corresponding pivotal operations are applied to the rest of the matrix.

Other features of HSL_MA64

- Handles singular systems: when column m is searched, if its largest entry has magnitude less than *small*, the row and column are set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no corresponding pivotal operations are applied to the rest of the matrix.

- Option for relaxed pivoting: if no 1×1 or 2×2 candidate pivot satisfies threshold test but the pivot that is nearest to satisfying it would satisfy it with $u = v \geq umin$, the pivot is accepted and u is reduced to v.

Other features of HSL_MA64

- Handles singular systems: when column m is searched, if its largest entry has magnitude less than *small*, the row and column are set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no corresponding pivotal operations are applied to the rest of the matrix.

- Option for relaxed pivoting: if no 1×1 or 2×2 candidate pivot satisfies threshold test but the pivot that is nearest to satisfying it would satisfy it with $u = v \geq umin$, the pivot is accepted and u is reduced to v.

- Option for static pivoting: if no 1×1 or 2×2 candidate pivot satisfies threshold test (even after relaxing the value of u) the 1×1 pivot that is nearest to satisfying the test is accepted. If its absolute value is less than *static*, it is given the value that has the same sign but absolute value *static*.

Static pivoting within HSL_MA77

Entries in **L** (in thousands) and times for factorization and solve phases.

		$nz(\mathbf{L})$		Factor		Solve	
Problem	N	without	with	without	with	without	with
cvxqp3	17500	4884	3131	1.6	0.7	0.1	0.07
dtoc	24993	6701	227	1.3	0.1	0.2	0.02
mario001	38434	746	665	0.1	0.1	0.04	0.03
ncvxqp7	87500	39192	24707	43	13	0.8	0.6

Scaled residuals for HSL_MA77 without and with static pivoting, before and after iterative refinement.

	without		with	
Problem	before	after	before	after
cvxqp3	$1.3 * 10^{-10}$	$2.0 * 10^{-16}$ (1)	$1.7 * 10^{-06}$	$1.9 * 10^{-15}$ (3)
dtoc	$1.1 * 10^{-14}$	$7.4 * 10^{-17}$ (1)	$6.7 * 10^{-13}$	$1.1 * 10^{-16}$ (1)
mario001	$6.1 * 10^{-15}$	$6.1 * 10^{-15}$ (0)	$1.3 * 10^{-10}$	$3.2 * 10^{-16}$ (4)
ncvxqp7	$2.1 * 10^{-09}$	$1.9 * 10^{-16}$ (1)	$1.5 * 10^{-07}$	$3.1 * 10^{-16}$ (5)

Concluding remarks

- Presented HSL_MA64 for the partial or complete factorization and solution of dense symmetric indefinite linear systems.
- HSL_MA64 offers fast execution speeds without compromising stability while minimizing factor storage.
- Key to the design is combining blocking with the use of standard BLAS and incorporating threshold partial pivoting.
- OpenMP allows parallel execution.
- HSL_MA64 is employed within the sparse multifrontal code HSL_MA77 and a modified version is used within the indefinite DAG-base sparse solver HSL_MA86 (Hogg and Scott '10).
- Paper to appear in ACM TOMS.

Concluding remarks

- Presented HSL_MA64 for the partial or complete factorization and solution of dense symmetric indefinite linear systems.
- HSL_MA64 offers fast execution speeds without compromising stability while minimizing factor storage.
- Key to the design is combining blocking with the use of standard BLAS and incorporating threshold partial pivoting.
- OpenMP allows parallel execution.
- HSL_MA64 is employed within the sparse multifrontal code HSL_MA77 and a modified version is used within the indefinite DAG-base sparse solver HSL_MA86 (Hogg and Scott '10).
- Paper to appear in ACM TOMS.

Thank you for your attention!