Sparse Direct Solvers II: The Key Ingredients

Jennifer Scott
STFC Rutherford Appleton Laboratory
and the University of Reading

Woudschoten Conference, 3-5 October 2018

Overview

Having given an introduction to sparse matrices and Sparse Direct Methods, we now look at the different phases of a sparse direct solver:

Step 2: Analyse

Step 3: Scale and Factorize

Step 4: Solve

Recall: Step 1 was reordering the matrix.

Step 2: Analyse

Graph theory used to analyse the sparsity pattern of the reordered matrix (numerical values not generally used).

Sparse data structures needed by the factorization are set up.

Data structure used depends on the factorization algorithm and on the computer architecture.

As architectures have changed, so too have the data structures but we always only manipulate and store the entries that are non zero (or we decide to treat as non zero).

Note: historically, the analyse phase was much faster than the factorization. Considerable effort has gone into parallelising the factorization so that the gap between the times for the two phases has narrowed. Maybe able to reuse for series of problems.

We will assume that A has a symmetric sparsity pattern.

Key objectives of analyse:

- Identify sets of columns of A with the same (or similar) sparsity patterns (supervariables). These are important in applications where there are multiple degrees of freedom at a finite-element node.
- Identify sets of columns of L with same (or similar) sparsity patterns (supernodes).
- Determine an assembly tree that will be used to guide the numerical factorization (recall: multifrontal algorithm).

Other tasks may include:

- handling errors in the user-supplied data (including out-of-range indices and duplicate entries). Important for inexperienced users when input data structures can be complicated.
- modifying the ordering of the assembly tree to minimize memory requirements.
- reordering variables within supernodes to increase cache locality during the factorization.

The package HSL_MC78 (Hogg and Scott 2010) performs these common tasks and can be employed within a sparse direct solver (eg used by HSL_MA97).

Importance of supervariables

If the average number of variables in each supervariable is k, the amount of integer data held during the analyse phase reduces by a factor of about k^2.

Problem	n (10^3)	ne (10^6)	$nsvar$ (10^3)	k	Storage (Mbytes) Original	Condensed
pkustk14	152	14.8	34	4.45	113	6
F1	344	26.8	120	2.85	204	25
af_shell10	1508	52.3	302	5.00	401	16
audikw_1	944	77.6	314	3.00	592	65

nsvar is number of supervariables. The storage is the integer storage for holding the sparsity pattern of A in its original form and in its condensed form.

Node amalgamation in assembly tree

Amalgamate parent node *pnode* with its child *cnode* if

- list of uneliminated variables at *cnode* is same as list of variables at *pnode* (in this case no additional fill in)
- both involve fewer than *nemin* eliminations (*nemin* can be chosen by user)

Results are:

- fewer nodes in the tree
- some zeros are treated as non zeros so increase in number of entries in factors, memory requirements and flop count
- but if *nemin* chosen correctly, improved performance (in terms of time) through greater use of dense linear algebra kernels

Supernodes

Consecutive columns with essentially identical sparsity structure can often be found in a triangular factor (may be relaxed).

A **supernode** corresponds to a group of p consecutive columns $j, j+1, ..., j+p-1$ in a triangular factor such that

- columns $j, j+1, ..., j+p-1$ have a dense diagonal block
- columns $j, j+1, ..., j+p-1$ have identical sparsity pattern below row $j+p-1$

Benefits:

- allows use of level 3 BLAS
- reduces amount of indirect addressing (inefficient)
- allows compact representation of factor sparsity structure

Step 3: Scale and factorize

Key issue: the factorization can become unstable (the size of individual entries in the matrix factors L and U can grow so that the computed solution is not close to the exact solution)

Remedy: incorporate numerical pivoting.

This means checking the size of the entries and performing row (and column) interchanges as the factorization proceeds to limit growth in the size of the entries.

Pivoting for stability

Recall first step of Gaussian elimination:

$$A = \begin{pmatrix} \alpha^2 & w^T \\ v & B \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ v/\alpha & I \end{pmatrix} \begin{pmatrix} \alpha & w^T \\ 0 & C \end{pmatrix}$$

$$C = B - vw^T/\alpha.$$

If α is small, some entries in B may be lost from the calculation of C. And if $\alpha = 0.0$, we have breakdown.

Pivoting: swap the current diagonal entry with a larger entry.

Goal: control growth of entries in L and U and prevent breakdown if pivot candiadte is zero.

Pivoting for stability

- Pivoting strategies:
  - Total (complete) pivoting: at each stage, choose largest entry in reduced matrix. Expensive in terms of time and fill in.

Pivoting for stability

- Pivoting strategies:
  - Total (complete) pivoting: at each stage, choose largest entry in reduced matrix. Expensive in terms of time and fill in.
  - Partial pivoting: select largest entry in first column of reduced matrix.

Pivoting for stability

- Pivoting strategies:
  - Total (complete) pivoting: at each stage, choose largest entry in reduced matrix. Expensive in terms of time and fill in.
  - Partial pivoting: select largest entry in first column of reduced matrix.
  - Rook pivoting: pivot must be largest in its row **and** column.

Pivoting for stability

- Pivoting strategies:
  - Total (complete) pivoting: at each stage, choose largest entry in reduced matrix. Expensive in terms of time and fill in.
  - Partial pivoting: select largest entry in first column of reduced matrix.
  - Rook pivoting: pivot must be largest in its row **and** column.
- Partial pivoting commonly used (cheap and usually OK).
- Rook pivoting has become more popular in recent years.

Pivoting in the symmetric case

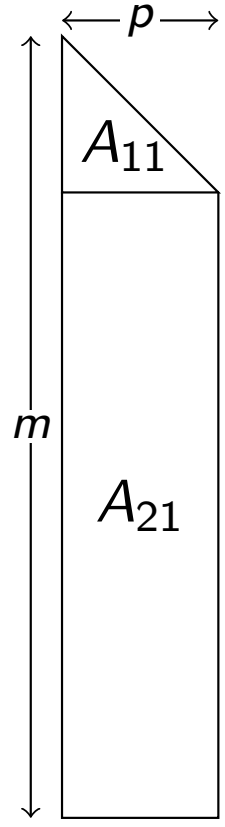
Let $A = A^T$

- Compute $A = LDL^T$. Only store L and D.
- Positive definite case: $A = (LD^{1/2})(LD^{1/2})^T$ with D diagonal and L unit lower triangular. Cholesky factorization
- Indefinite case: Partial pivoting destroys symmetry so
  - either move a large diagonal entry to (1,1) position, or
  - move a large off-diagonal entry to (1,2) position and use 2×2 pivot

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix}.$$

D is block diagonal with 1×1 and 2×2 blocks

Recall supernodes



The major numerical tasks to be performed on each supernode are:

Factor $A_{11} = L_{11} D_{11} L_{11}^T$;

Solve $L_{21} = A_{21} (D_{11} L_{11}^T)^{-1}$;

Form $S = L_{21} D_{11} L_{21}^T$ (Schur complement); and

Scatter S across other supernodes.

Key difference from an otherwise equivalent dense factorization is that pivots can only be selected from within A_{11}.

But when selecting pivots, the factor task needs to take account of the values of the entries in A_{21} as well as those in A_{11}.

If a candidate pivot is found to be unsuitable, it is moved to a later supernode for elimination. Such pivots are said to be delayed.

How do we decide if a pivot is suitable? In the sparse case, selecting the largest entry in the column is too restrictive.

Pivot test

Given a pivot threshold $u \in [0, 0.5]$, stability criteria:

- a 1×1 pivot on column q is stable if

$$|a(q,q)| \geq u \max_{i>q} |a(i,q)|,$$

- a 2×2 pivot on columns q and $q+1$ is stable if

$$\left| \begin{pmatrix} a(q,q) & a(q,q+1) \\ a(q,q+1) & a(q+1,q+1) \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i>q+1} |a(i,q)| \\ \max_{i>q+1} |a(i,q+1)| \end{pmatrix} \leq \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}, .$$

These imply each entry in L is bounded by u^{-1} and growth between consecutive steps (and hence in D) is at most u^{-1}.

Choosing u is a balance between stability and sparsity.
Default often $u = 0.01$.

Effects of delayed pivots

If a candidate pivot fails the stability test, it must be delayed.
Consequences:

- More memory needed (during the factorization and to hold the factors).
- Slows down the factorization and makes the solve more expensive.
- Restricts the scope for parallelism.
- Makes programming much harder!

So we want to try and avoid delayed pivots.

Significant effort in recent years to limit the need for pivoting.

One way is to prescale A to make its entries "nice".

Scaling: Equilibration

$$A_S = D_R^{-1} A D_C^{-1},$$

where D_R and D_C are diagonal matrices, is an **equilibration** of A if the norms of the rows and columns of A_S have approximately the same magnitude.
Define

$$D_R = \mathrm{diag}\left(\sqrt{\max_j |A_{ij}|}\right) \quad \text{and} \quad D_C = \mathrm{diag}\left(\sqrt{\max_i |A_{ij}}\right).$$

More generally, the process can be applied iteratively. The norms of the rows and columns tend to $+1$

Can prove linear convergence with asymptotic rate of 0.5 in the $\infty-$ norm (in practice, only need a few iterations).

Algorithm is implemented in HSL code `MC77`.

Alternative approach: maximum matchings

Idea: permute A prior to GE to put large entries on the diagonal.

Why?
Would like to pivot down the diagonal (cheaper than searching for off-diagonal pivots and no interchanges that destroy symmetry needed).

Pivoting down diagonal may be more stable if large entries are on diagonal.

Maximum matching algorithm implemented in HSL code `MC64` (Duff and Koster): all diagonal entries are one and off-diagonal entries less than one in absolute value.

There is also a symmetric variant: symmetrically permute large entries to subdiagonal to use within 2×2 pivots.

Example

Optimization augmented system problem `ncvxqp1` ($n = 12,111$).

This is a symmetric indefinite problem; we factorize using multifrontal algorithm with partial pivoting.

Delayed pivots/time (seconds):

- **No scaling:** $1.7 * 10^5$ / 102
- Equilibration `MC77:` $4.0 * 10^4$ / 8.75
- Maximum matching `MC64:` $1.3 * 10^4$ / 2.34

In this example, `MC64` is better than `MC77` but not always the case.

Note: `MC64` can be **expensive** (may cost more to scale than to factorize). Recent work on cheaper versions (such as approximate maximum matchings, parallel variants).
We recommend its use for "tough" symmetric indefinite problems.

Static pivoting

- Scale and put large entries on to diagonal (or subdiagonal) first
- If necessary perturb small diagonal entries as factorization progresses (**static pivoting**) so that pivot sequence not altered. This means $A + \Delta A_D = LU$. The hope is that $A \approx LU$.
- Use iterative refinement (or FGMRES) to recover accuracy

Success **not** guaranteed ie may not obtain required accuracy. But

- can be useful if less accuracy is required
- can be significantly faster (if only a small number of steps of refinement needed)
- factors may be much sparser so less memory required and solve phase is faster.

Communication avoiding sparse LDL^T

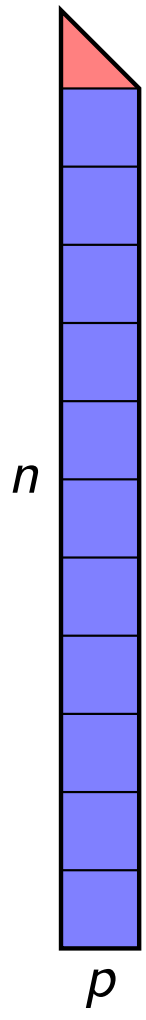
A key problem with pivoting is that it inhibits parallelism (all the entries in a candidate pivot column have to be up-to-date before we can be sure the pivot is ok).

Possible idea:

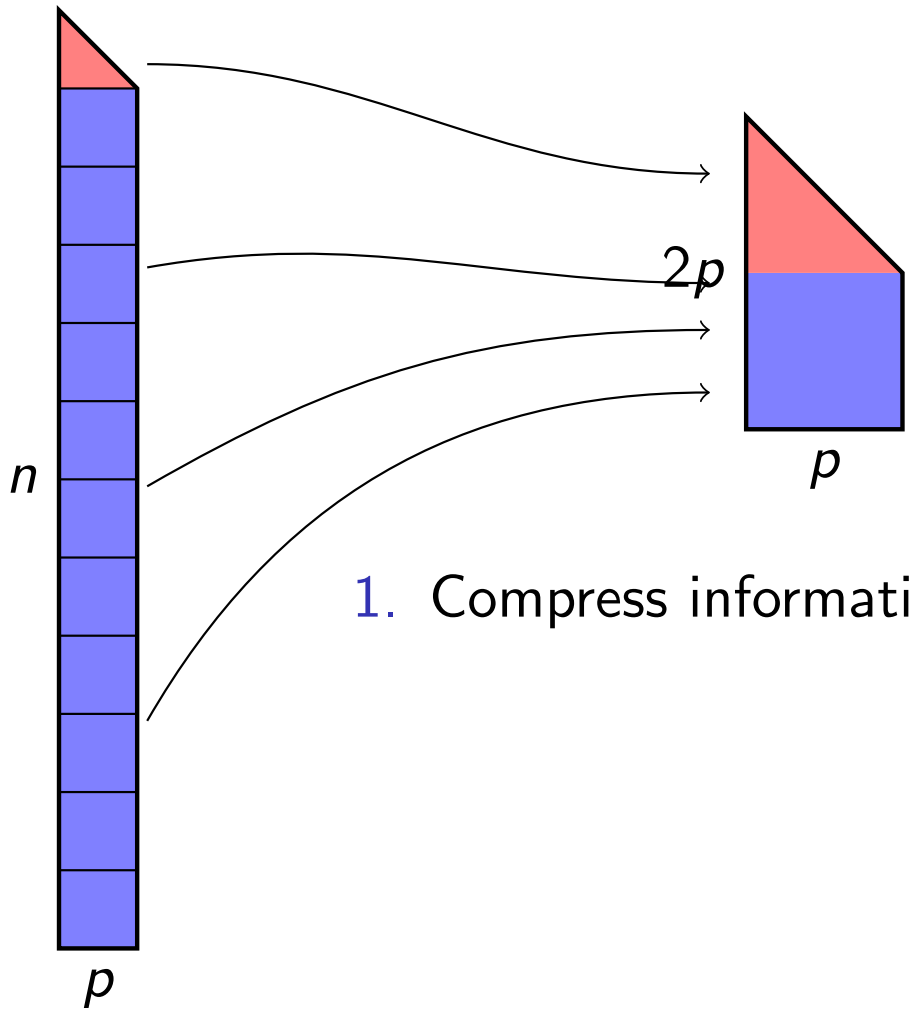
Try-it-and-see pivoting:

- Store a copy before pivoting
- Do numerical test a posteriori
- Back-track if it breaks
- **Still need a fall back plan**

Compressed pivoting (Hogg and Scott 2014)

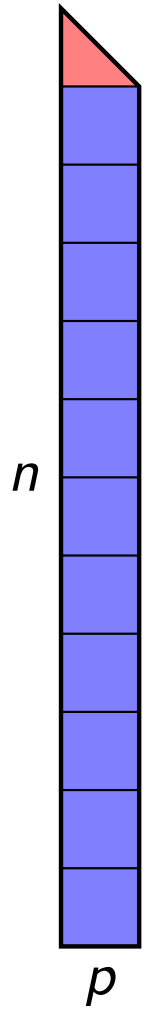


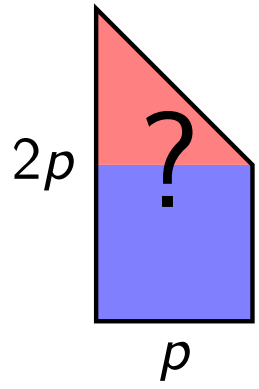
Compressed pivoting (Hogg and Scott 2014)



1. Compress information into small matrix

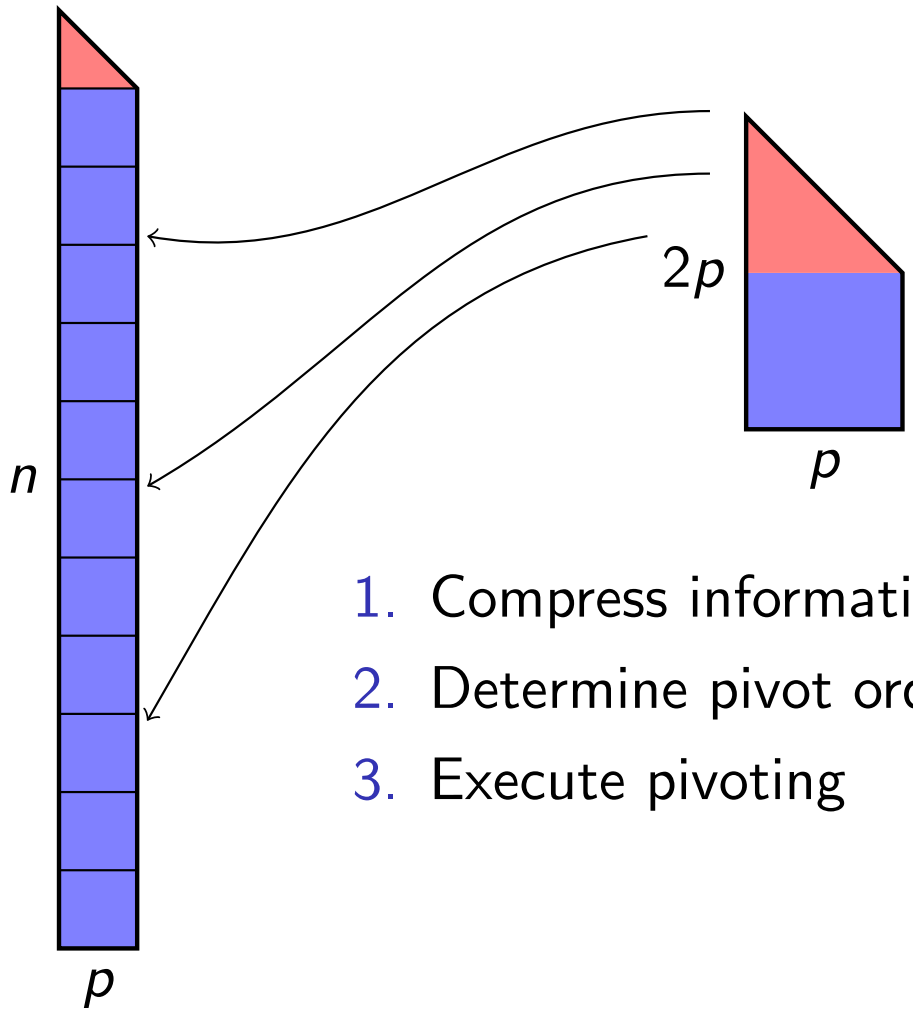
Compressed pivoting (Hogg and Scott 2014)





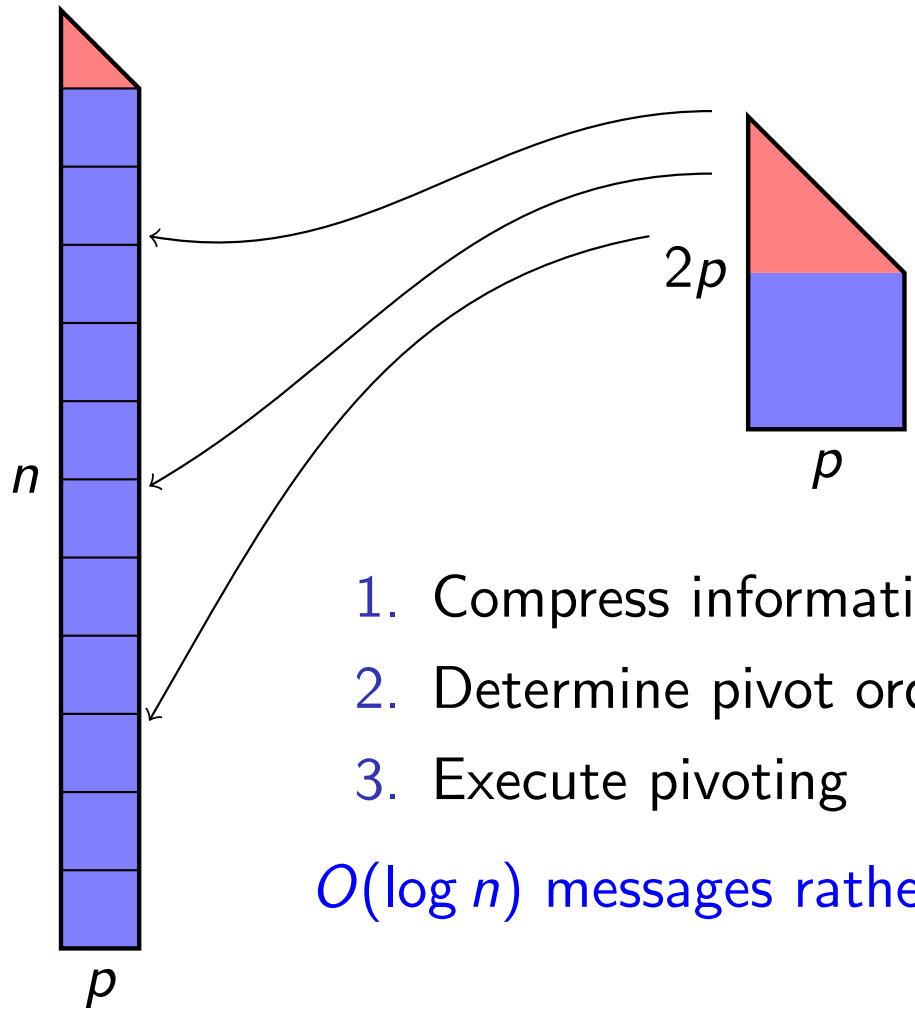
1. Compress information into small matrix
2. Determine pivot order

Compressed pivoting (Hogg and Scott 2014)



1. Compress information into small matrix
2. Determine pivot order
3. Execute pivoting

Compressed pivoting (Hogg and Scott 2014)



1. Compress information into small matrix
2. Determine pivot order
3. Execute pivoting

$O(\log n)$ messages rather than $O(p \log n)$

Alternative approach: avoid pivoting by ordering for stability

`MC64` can be used to combine scaling with ordering.
But while reducing delayed pivots

- it may not eliminate need for numerical pivoting and
- it can lead to more flops and factor fill than we would like.

Alternatives? Lungten, Schilders and Scott (2017) recently proposed an approach for saddle-point systems

$$Kz = b, \quad K = \begin{bmatrix} A & B^T \\ B & -C \end{bmatrix},$$

with A $n \times n$ symmetric positive-definite, B $m \times n$ matrix of full row rank with $m < n$, and C symmetric positive semidefinite.

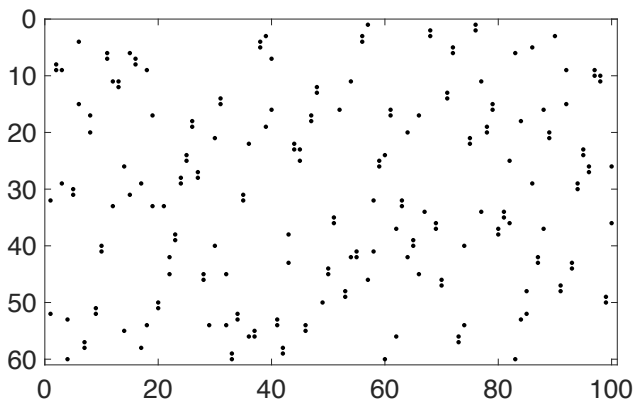
They consider case where for some permutation matrices P_r, P_c

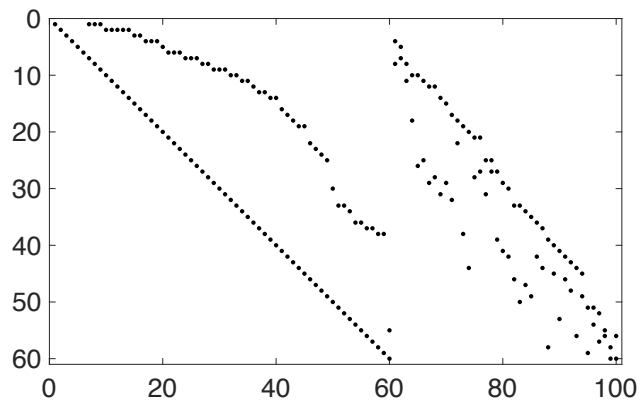
$$P_r B P_c = [B_1 \quad B_2],$$

with B_1 $m \times m$ nonsingular upper triangular matrix.

Many practical applications where this is possible eg network analysis of electronic circuits and water distribution pipe networks.

Lungten et al present algorithms to find suitable permutations.
Example: Original and permuted B





Now have

$$K = \begin{bmatrix} A_{11} & A_{12} & B_1^T \\ A_{21} & A_{22} & B_2^T \\ B_1 & B_2 & -C \end{bmatrix}$$

Choose P to be the permutation matrix with columns

$$P = [e_1 \ e_{n+1} \ e_2 \ e_{n+2} \ \dots \ e_m \ e_{n+m} \ e_{m+1} \ \dots \ e_n],$$

then

$$P^T K P = [K_{ij}],$$

where

$$K_{ij} = \begin{cases} \begin{bmatrix} a_{ii} & b_{ii} \\ b_{ii} & -c_{ii} \end{bmatrix}, \ 1 \le i = j \le m; & \begin{bmatrix} a_{ij} & b_{ji} \\ 0 & -c_{ij} \end{bmatrix}, \ 1 \le j < i \le m; \\ \begin{bmatrix} a_{ij} & 0 \\ b_{ij} & c_{ij} \end{bmatrix}, \ 1 \le i < j \le m; & \begin{bmatrix} a_{ij} \\ b_{ij} \end{bmatrix}, \ 1 \le i \le m < j \le n; \\ \begin{bmatrix} a_{ij} & b_{ji} \end{bmatrix}, \ 1 \le j \le m < i \le n; & \begin{bmatrix} a_{ii} \end{bmatrix}, \ m < i, j \le n. \end{cases}$$

Compute a fill-reducing ordering for K by

1. compressing the adjacency graph of PKP^T by considering each block as a single entity and merging the sparsity patterns of the rows and columns belonging to a 2×2 diagonal block,
2. applying fill-reducing ordering (eg AMD) to compressed graph.

Lungten et al prove the existence of the resulting LDL^T factorization **without** modifying the pivot sequence (which is **not** the case for the `MC64` matching-based ordering).

Thus this is a possible approach to avoid pivoting and delayed pivots for some tough saddle-point problems.

Another issue: the challenge of bit compatibility

Context: Subsequent runs on same problem give (slightly) different answers Aim: Get the same answer every time.

$$1 + (\epsilon/2 + \epsilon/2) \neq (1 + \epsilon/2) + \epsilon/2$$

Why would we not require bit compatibility?

- If we don't, answers are still equally valid
- More efficient: insisting on bit compatibility restricts parallelism and optimization of the code.
- Compatibility is difficult to achieve.
- Must be achieved by all other software used by the solver.

The challenge of bit compatibility

Context: Subsequent runs on same problem give (slightly) different answers Aim: Get the same answer every time.

$$1 + (\epsilon/2 + \epsilon/2) \neq (1 + \epsilon/2) + \epsilon/2$$

But its very attractive ...

- Hard to debug without it: make it an option?
- Confuses non-expert users.
- Software that uses the solver may behave unexpectedly.
- Some users insist on it (eg financial forecasting, nuclear regulation ...

Achieving bit-compatibility

Must add up in the same order ie enforce ordering on additions:
Choose $((1+2)+3)+4$ or $(1+2)+(3+4)$

Our parallel (multicore) multifrontal direct solver `HSL_MA97` (Hogg and Scott 2011) guarantees bit compatibility.

Step 4: Solve

At this point, we are nearly done!

We have the factorization $PAQ = LU$ and we have triangular systems to solve. $Ly = b$ and $Ux = y$

Traditionally, this was seen as the "cheap and easy" step but with parallel computers it presents challenges. Need faster solve.

Context: The factors L and U must be read from memory

Enemy is: bandwidth

Note: much more economical to solve for several right-hand sides b at once (only read the factors once and BLAS 3 can be used in place of BLAS 2).

In general, the right-hand sides b are dense (or are treated as dense).

In recent years, interest in solving systems where b is sparse (e.g., b could be a column of the identity), although solution x is dense.

In this case, significant savings may be possible.

Some solvers offer an option of inputting b as a sparse vector (e.g., MUMPS).

This is still an active area.

Life is a journey, not a destination (R W Emerson 1920)

O'Connor (1985) Although sparse matrices may seem a rather narrow and specialised topic, it has developed into a very active area of research ... It draws on topics such as linear algebra, numerical analysis, graph theory, combinatorics, data structures and software design ...

Software for sparse matrix problems is not easy to design and develop ...

So we need experts to do it and make it available ...

It is better to travel well than to arrive (Buddha)

Sparse direct solvers have had a relatively short history and have been studied by a small but very active international community of researchers.

Enormous progress has been made as problems are routinely solved that would have been regarded as impossible when I began in this field.

But challenges remain and constantly being developed ....

- Ever larger problems requiring real-time solutions.
- New application areas giving rise to problems with different structures compared to more traditional engineering applications.
- Exploitation of new computer architectures (operations are cheap, memory accesses are expensive).
- Combining direct and iterative solvers to exploit the best of both in hybrid approaches.

And there will be more that we have not yet thought of ...

All journeys have secret destinations of which the traveller is unaware (M Buber, Jewish Philosopher)

Finally ... so what is behind backslash?

It actually depends on the characteristics of A.

If A is unsymmetric, an LU factorization is performed by UMFPACK (Davis).

If A is symmetric and positive definite, a Cholesky factorization is performed using supernodal solver CHOLMOD (Davis).

Otherwise, A is symmetric indefinite and the multifrontal sparse solver `MA57` (HSL, Duff) is employed.

To find out more, the recent book *Direct Methods for Sparse Matrices* by Duff, Erisman and Reid (OUP 2017) is a great place to start.