Parallel algorithms for Scientific Computations
Sparse matrices

Maya Neytcheva

PASC, November 2020

Parallel computer architectures

Architecture	homogeneous / nonhomogeneous - accelerators, GPUs, vector units
Memory organization	shared / distributed / virtual shared
Interconnection topology	mesh, thorus, hypercube, ...
Levels of parallelism	bit-instruction-task level, data level, function level, compiler level...
Granularity	Machine-algorithm match

Parallel computations require to reconsider

- the algorithms,
- the data organization,
- the communications,
- ... also how much we can trade parallelism for numerical efficiency.

Plan of the lecture:

- Sparse matrices - who are those?
- Why are sparse matrices a topic of special interest?
- Handling sparse matrices. Sparse data formats
- Solution methods for sparse matrices
  - Direct methods
    - Fill-ins and can we get rid of them?
    - Reordering strategies
    - Sparse Cholesky factorization
    - Sparse QR, SVD
- Examples

Large matrices

What has been and is considered as large through the years $N(t)$

1970	200
1975	1 000
1980	10 000
1985	100 000
1990	250 000
1995	500 000
2000	2 000 000
since 2005	500 000 000

What is a sparse matrix?

$$A(N \times N), \quad nnz(A) = kN, \quad 2 \leq k \leq logN$$

Where do sparse matrices arise?

acoustic scattering	demography	network flow
air traffic control	economics	oceanography
astrophysics	electrical eng.	petroleum eng.
biochemical	electric nets	reactor modelling
chemical eng.	climate/pollution studies	statistics
chemical kinetics	fluid flow	structural eng
circuit physics	laser optics	survey data
computer simulations	linear programming	signal processing

Thanks to software and powerful computers on our desk, we do not care that much about sparse-dense etc...
until we face large enough problems or we have to repeat a computational task 100, 1000, 10000 times.

This lecture concerns storage book-keeping and programming aspects which will help to

- do the computations faster
- save computer memory

The Top500 list, November 2019

Rank	Manuf.	Computer	Installation Site	# PEs	TFLOPS
1	IBM	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/	Oak Ridge	2 414 592	200795
82		Tetralith - Intel H2204XXLRE, Xeon Gold 6130 16C 2.1GHz, Intel Omni-Path ClusterVision / Hammer	Linköping	64 512	4335

The HPCG Top500 list, November 2019

Rank	Manuf.	Computer	Installation Site	# PEs	TFLOPS
1	IBM	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/	Oak Ridge	2 414 592	2925.75/ 200795

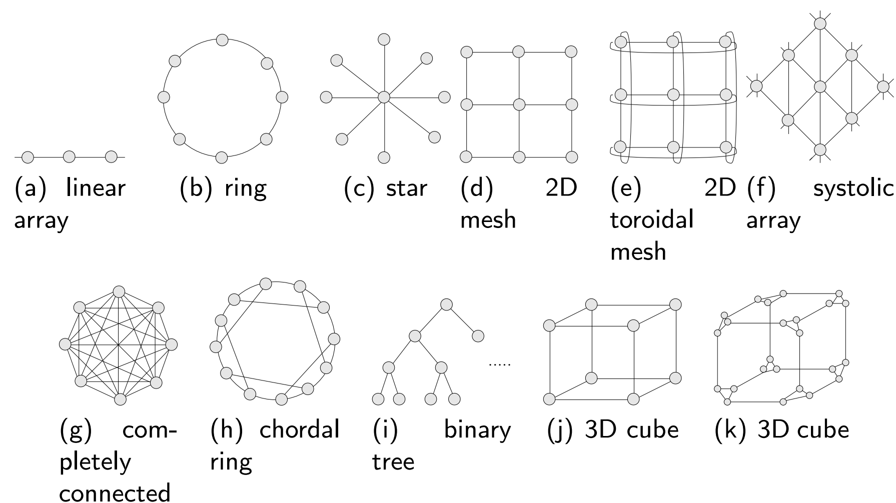
What is HPCG?

HPCG is a complete, stand-alone code that measures the performance of basic operations in a unified code
`https://www.hpcg-benchmark.org/`

- Sparse matrix-vector multiplication.
- Vector updates.
- Global dot products.
- Local symmetric Gauss-Seidel smoother.
- Sparse triangular solve (as part of the Gauss-Seidel smoother).
- Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids.

Reference implementation is written in C++ with MPI and OpenMP support.

Interconnection network topologies



(a) linear array (b) ring (c) star (d) 2D mesh (e) 2D toroidal mesh (f) systolic array

(g) completely connected (h) chordal ring (i) binary tree (j) 3D cube (k) 3D cube

Interconnection network topologies, cont.

Behind (almost) each topology there was a physical computer:

Topology	Computer	Year	Remark
Linear	ILLIAC IV	1972	up to 64 procs
Mesh	Intel Paragon	1994	No 1 in top500
Torus	Cray T3D/T3E		BlueGene L 5D torus
Systolic	WARP	1985	MISD, tori Carnegie Mellon Univ., 10 Mflops
Fat tree	CM-5	1991	Thinking Machines
N-cube	CM2/200	1985	2^{16} bit-processors
	Ncube	1985	1024 procs

Before discussing sparse matrices...

consider first dense matrices...
because these are easier.

Dense matrix storage schemes

Given a dense matrix $A(m,n)$.
Two main possibilities to store dense matrices:
row-wise and *column-wise*.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

First row

$$DRW: \quad a_{11}\ a_{12}\ \cdots\ a_{1n} \,\big|\, a_{21}\ a_{22}\ \cdots\ a_{2n} \,\big|\, a_{m1}\ a_{m2}\ \cdots\ a_{mn} \,\big|$$

First column

$$DCW: \quad a_{11}\ a_{21}\ \cdots\ a_{m1} \,\big|\, a_{12}\ a_{22}\ \cdots\ a_{m2} \,\big|\, a_{1n}\ a_{2n}\ \cdots\ a_{mn} \,\big|$$

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m,n)$

$$y_i = \sum_{j=1}^{n} a_{ij}x_j, \quad i = 1, \ldots, m.$$

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m,n)$

$$y_i = \sum_{j=1}^{n} a_{ij}x_j, \quad i = 1, \ldots, m.$$

A: stored row-wise (inner product scheme)

```
for i=1:m
    y(i) = 0
    for j = 1:n
        y(i) = y(i) + A(i,j) * x(j)
    end
end
```

Dense matrix storage schemes and computations

What difference does the storage scheme make with respect to computations?

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}, A(m,n)$

$$y_i = \sum_{j=1}^{n} a_{ij}x_j, \quad i = 1, \ldots, m.$$

```
for i=1:m
    y(i) = 0
    y(i) = y(i) + A(i,:)  * x(:)
end
```

Dense matrix storage schemes and computations

Matrix-vector multiplications: $\mathbf{y} = A\mathbf{x}$

A: stored column-wise (outer product scheme)

```
y = 0
for j=1:n
   for i = 1:m
      y(i) = y(i) + A(i,j) * x(j)
   end
end
```

$\mathbf{y} = 0$

for $j = 1:n$

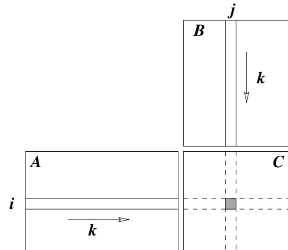
$\quad \mathbf{y} = \mathbf{y} + x(j) * A(:,j)$

end

(vector operation)

Dense matrix-matrix multiply (ijk)

$A(m,n)*B(n,p) = C(m,p)$

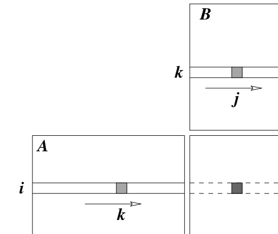



```
C = 0
for  i=1:m
    for  j=1:p
        for  k=1:n
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end
    end
end
```

Scalar-product type of computation

Dense matrix-matrix multiply (ijk)

$A(m,n)*B(n,p) = C(m,p)$

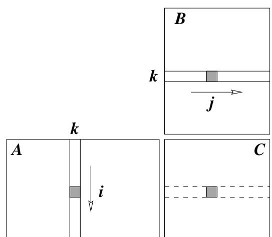



```
C = 0
for  i=1:m
    for  k=1:n
        for  j=1:p
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end
    end
end
```

Outer-product type of computation ('ikj' - row-wise and 'jki' - column-wise)

Dense matrix-matrix multiply (ijk)

$A(m,n)*B(n,p) = C(m,p)$




```
C = 0
for  k=1:n
    for  i=1:m
        for  j=1:p
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        end
    end
end
```

accumulated update form

Dense matrix storage schemes and computations

Bottom line:

- The storage scheme of a dense matrix affects the order how the matrix entries are accessed in the computer memory. This may have a significant effect on the performance of an algorithm since the memory accesses are much slower than arithmetic operations
- One storage scheme is better for some operations and not so preferable for other operations (A^T).

Factorizing symmetric marices

Factorize $A = LL^T$, L – lower-triangular
Cholesky factorization

Major Andre-Louis Cholesky (1875-1918)

The mathematician after whom the Cholesky factorisation is named.
He was born in France, and worked in the Geodesic section of the Geographic service to the French army's artillery branch.
At this time the system of triangulation used in France, and based on the meridian line of Paris, was being revised; new methods were needed in order to facilitate what was not yet a quick or convenient process.
Cholesky invented computation procedures based on the method of least squares, for the solution of certain data-fitting problems in geodesy, to be put into practice in his triangulation of the French and British parts of Crete, and in his work in Algeria and Tunisia.
His mathematical work was posthumously published on his behalf in 1924 by a fellow officer, Benoit.

Factorizing symmetric marices

Cholesky factorization ...

```
% Maya's version of Cholesky - to compare execution time
% ---------------------------------------------------------------
function [U]=my_chol(A)
A = triu(A);
n = size(A,1);
for k=1:n,
    A(1:k-1,k) = A(1:k-1,1:k-1)'\A(1:k-1,k);
    A(k,k) = sqrt(A(k,k) - A(1:k-1,k)'*A(1:k-1,k));
end
U = triu(A);
return
```

Cholesky factorization ...

size(A)	chol Matlab	chol mine	Ratio	nnz(A)	nnz(U)
18	0.0137	0.0337	2.46	35	41
50	0.0145	0.0788	5.42	167	406
162	0.0264	0.5941	22.50	743	4 958
578	0.5782	12.61	21.80	3143	75 237
2178	51.4146	892.286	17.35	12935	1 179 520

GAXPY Cholesky

for $k = 1 : n$
  if $k > 1$
    $A(k : n, k) = A(k : n, k) - A(k : n, 1 : k-1) * A(k, 1 : k-1)$
  endif
  $A(k : n, k) = A(k : n, k)/\sqrt{A(k, k)}$
end

Outer Product Cholesky

for $k = 1 : n$
  $A(k, k) = \sqrt{A(k, k)}$
  $A(k+1 : n, k) = A(k+1 : n, k) - A(n : k, k-1)/A(k, k)$
  for $j = k+1 : n$
    $A(j : n, j) = A(j : n, j) - A(j : n, j)A(j, k)$
  end
end

put together

for $k = 1 : n$
  if $k > 1$
    $A(k : n, k) = A(k : n, k) - A(k : n, 1 : k-1) * A(k, 1 : k-1)^T$
  endif
  $A(k : n, k) = A(k : n, k)/\sqrt{A(k, k)}$
end

for $k = 1 : n$
  $A(k, k) = \sqrt{A(k, k)}$
  $A(k+1 : n, k) = A(k+1 : n, k) - A(n : k, k-1)/A(k, k)$
  for $j = k+1 : n$
    $A(j : n, j) = A(j : n, j) - A(j : n, j)A(j, k)$
  end
end

Example of implementing Cholesky factorization

```
for k=1:n
    xeuitb(A(1:k-1,k),A(1:k-1,1:k-1),A(1:k-1,k))
    A(k,k) = sqrt(A(k,k) - A(1:k-1,k)^T*A(1:k-1,k))
end
```

Computes U (which overwrites A).

BLAS `xeuitb(X,U,B)` computes $X = U^{-1}B$

Sparse matrices

Sparse matrix collections

- Matrix Market `https://math.nist.gov/MatrixMarket/`
- Harwell-Boeing sparse matrix collection `http://swmath.org/software/8516`
- SuiteSparse Matrix Collection `https://sparse.tamu.edu/`
- UF Sparse Matrix Collection `https://www.cise.ufl.edu/research/sparse/matrices/list_by_id.html`

Sparse matrix storage schemes

There are more than 20 different sparse storage schemes...

Sparse matrix storage schemes

Coordinate scheme:

$$A = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$$

V :	1	2	3	4	5	6
I :	1	1	2	2	3	4
J :	2	3	1	2	3	1

Advantages and disadvantages

Sparse matrix storage schemes

Diagonal-wise storage scheme:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & a_{25} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & a_{36} \\ 0 & 0 & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \qquad V = \begin{bmatrix} 0 & a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{25} \\ a_{32} & a_{33} & a_{34} & a_{36} \\ a_{43} & a_{44} & a_{45} & 0 \\ a_{54} & a_{55} & a_{56} & 0 \\ a_{65} & a_{66} & 0 & 0 \end{bmatrix}$$

OF : -1 0 1 3

Sparse matrix storage schemes

Sparse compressed schemes: $A = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$

V :	1	2	3	4	5	6
C :	2	3	1	2	3	1
R :	1	3	5	6	7	

(l) CSR

V :	3	6	1	4	2	5
R :	2	4	1	2	3	3
C :	1	3	5	7	7	

(m) CSC

Sparse matrix-vector multiplication using CSR

```
A: IA, JA, VA
for i=1:m
    t = 9;
    for j=IA(i):IA(i+1)-1
         t = t + AV(j)*x(JA(j));
    end
    y(i) = t;
end
```

Used in UMFPACK.

Sparse matrix storage schemes

Jagged diagonal storage: The Jagged Diagonal Storage format can be useful for the implementation of iterative methods on parallel and vector processors. Like the Compressed Diagonal format, it gives a vector length essentially of the size of the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation.

$$\begin{bmatrix} 10 & -3 & 0 & -1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

col_ind(:,1)	1	2	1	2	5	5
col_ind(:,2)	2	3	3	4	6	6
col_ind(:,3)	4	5	4	5	0	0
col_ind(:,4)	0	0	0	6	0	0

Jagged diagonals, cont.

$$\begin{bmatrix} 10 & -3 & 0 & -1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 10 & -3 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 7 & 5 & 4 \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 10 & -3 & -1 & \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}$$

vals	6	9	3	10	9	5;	7	6	8	-3	13	-1;	5	-2	7	1;	4;
cols	2	2	1	1	5	5;	4	3	3	2	6	6;	5	5	4	4;	6;

perm	4	2	3	1	5	6

jd_ptr	1	7	13	17

Yale data format

The Yale sparse format:
a (sparse) matrix is stored with four elements (vectors), which are:
– (1) the nonzero values row by row,
– (2) the ordered column indices of nonzero values,
– (3) the position in the previous two vectors corresponding to new rows, given as pointers,
– (4) the column dimension of the matrix.

Block-row storage (BSR) format

Intel MKL block compressed sparse row (BSR)- four arrays: values, columns, pointerB, and pointerE.
values: A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
columns: Element i of the integer array columns is the number of the column in the block matrix that contains the i-th non-zero block.
pointerB: Element j of this integer array gives the index of the element in the columns array that is first non-zero block in a row j of the block matrix.
pointerE: Element j of this integer array gives the index of the element in the columns array that contains the last non-zero block in a row j of the block matrix plus 1.

How to multiply two sparse matrices?

- The number of nonzeros in the product can be larger.
- Two-pass algorithm, to determine how much memory to allocate.

LU factorization for sparse matrices

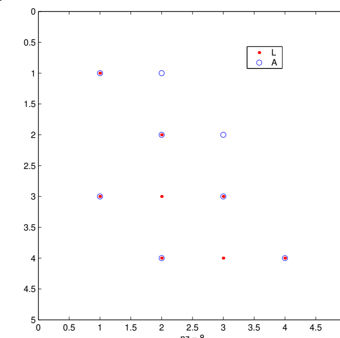
Direct methods: $A = LU, \; LU\mathbf{x} = \mathbf{b}, \; L\mathbf{y} = \mathbf{b}, \; U\mathbf{x} = \mathbf{y}$

The process of triangular factorization (Gaussian elimination) for the case of *sparse* matrices.
Note: In general, during factorization we have to do pivoting in order to assure numerical stability.
The computational complexity of a direct solution algorithm is as follows.

Type of matrix A	Factor	LU solve	Memory
general dense	$2/3n^3$	$O(n^2)$	$n(n+1)$
symmetric dense	$1/3n^3$	$O(n^2)$	$1/2n(n+1)$
band matrix $(2q+1)$	$O(q^2n)$	$O(qn)$	$n(2q+1)$

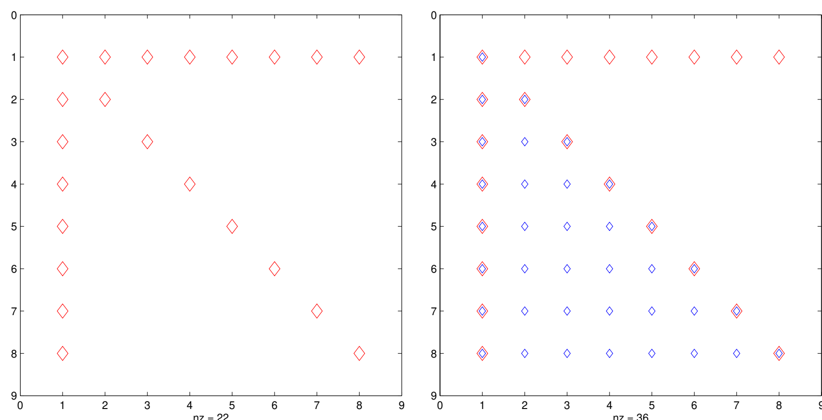
The reason to consider particularly factorizations of sparse matrices

is the effect of *fill-in*, namely, obtaining nonzero entries in the LU factors in positions where $A_{i,j}$ is zero.



$$a_{i,j}^{(k+1)} \longleftarrow a_{i,j}^{(k)} + \frac{a_{i,k}^{(k)} \, a_{k,j}^{(k)}}{a_{k,k}^{(k)}}$$

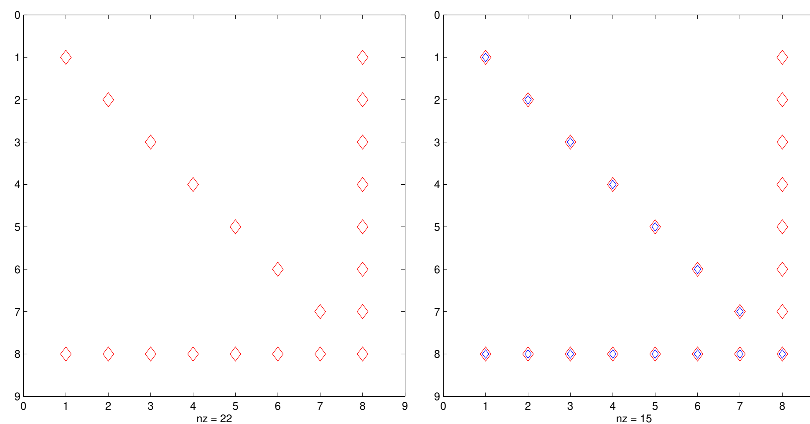
Effect on sparsity structure on factorization:



(n) Arrow matrix (o) The structure of the L-factor

The arrow matrix structure - the L and U factors are full.

Effect on sparsity structure on factorization



(p) Arrow matrix permuted (q) The structure of the L-factor

We can permute the matrix A first and then factorize!

We pose now the question to $\cdots$

find permutation matrices P and Q, such that when we factorize $\widetilde{A} = Q^T A P^T$, the fill-in in the so-obtained L and U factors will be minimal.
The solution algorithm takes the form:

(1) Factorize $Q^T A P^T = LU$
(2) Solve $PL\mathbf{z} = \mathbf{b}$ and $UQ\mathbf{x} = \mathbf{z}$.

How to construct P and Q in general?

The aim of sparse matrix algorithms is to solve the system $A\mathbf{x} = \mathbf{b}$ in time and space (computer memory requirements) proportional to $\boxed{O(n) + O(nnz(A)),}$ where $nnz(A)$ denotes the number of nonzero elements in A.
Even if the latter target cannot be achieved, the complexity of sparse linear algebra is far less than that of the dense case:

Order of A	$nnz(A)$	Time in sec	
		Dense solver	Sparse solver
680	2646	0.96	0.06
1374	8606	6.19	0.70
2205	14133	24.25	2.65
2529	90158	36.37	1.17

Time on Cray Y-MP (results taken from I. Duff)

The strive to achieve complexity $O(n) + O(nnz(A))$ entails very complicated sparse codes.
Some important aspects when implementing the direct solution techniques for sparse matrices in practice:

- sparse data structures and manipulations with those;
- computer platform related issues, such as handling of indirect addressing; lack of locality;
  difficulties with cache-based computers and parallel platforms;

  short inner-most loops.

Extra difficulties come from the fact that···

we have to choose a pivot element and its proper choice may contradict to the strive to minimize fill-in.
Condition $min[(n_i^{(k)}-1)(n_j^{(k)}-1)]$ can be seen as
- choosing a pivot which will modify the least number of coefficients in the remaining submatrix;
- choosing a pivot that involves least multiplications and divisions;
- as a means to limit the fill-in since it will produce at most $(n_i^{(k)}-1)(n_j^{(k)}-1)$ new nonzero entries.
However, in general the entry $a_{i,j}^{(k)}$ has to obey some other numerical criteria also, for example,

$$|a_{i,j}^{(k)}| \geq \tau |a_{i,s}^{(k)}|, i \geq s,$$

where $\tau \in (0,1)$ is a threshold parameter.

```
n=500;
R=sprand(n,n,5/n); I=speye(n); b=rand(n,1); A=I+R; AF=full(A);
tic,x=A\b;toc
Elapsed time is 0.006472 seconds.
tic,x=AF\b;toc
Elapsed time is 0.036819 seconds.

n=5000;
tic,x=A\b;toc
Elapsed time is 0.336134 seconds.
tic,x=AF\b;toc
Elapsed time is 1.666255 seconds.

n=10000;
tic,x=A\b;toc
Elapsed time is 1.881219 seconds.
tic,x=AF\b;toc
Elapsed time is 12.504630 seconds.
```

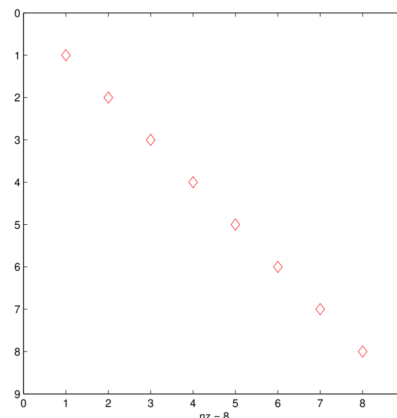
```
n=50000;
R=sprand(n,n,1/n);I=speye(n);b=rand(n,1);A=10*I+0.5*(R+R');
tic,x=A\b;toc
Elapsed time is TOO MANY seconds.


tic,[x,flag,relres,iter,resvec]=pcg(A,b,1e-6,1000);toc
Elapsed time is 0.015673 seconds.
iter   = 5
relres = 4.67 e-07
```

We are most often dealing with *'Given-the-matrix'* case

I.e., the only source of information is the matrix itself and we will try to reorder the entries so that the resulting structure will limit the possible fill-in.

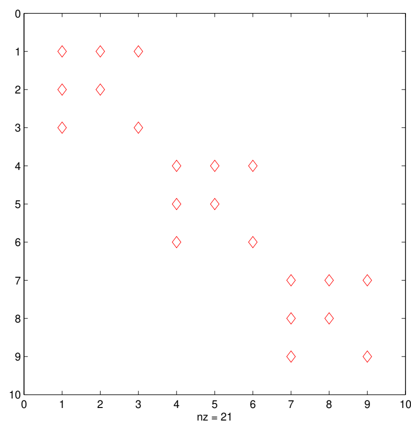
What is the matrix structure to aim at?

Given-the-matrix strategy

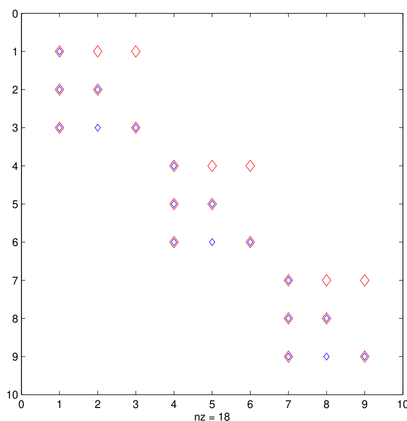


(r) Diagonal matrix

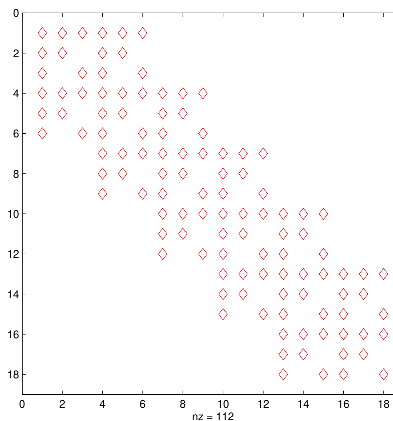
- diagonal
- block-diagonal
- block-tridiagonal
- arrow matrix
- band matrix
- block-triangular



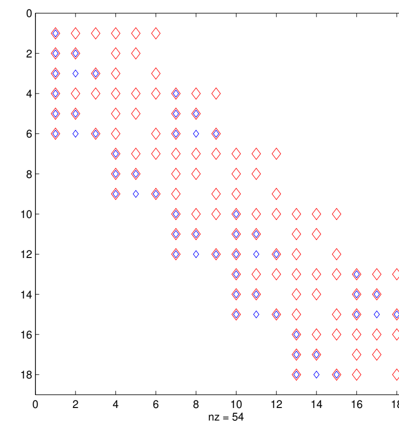
(s) block-diagonal matrix



(t) The structure of the L-factor



(u) Block-tridiagonal matrix

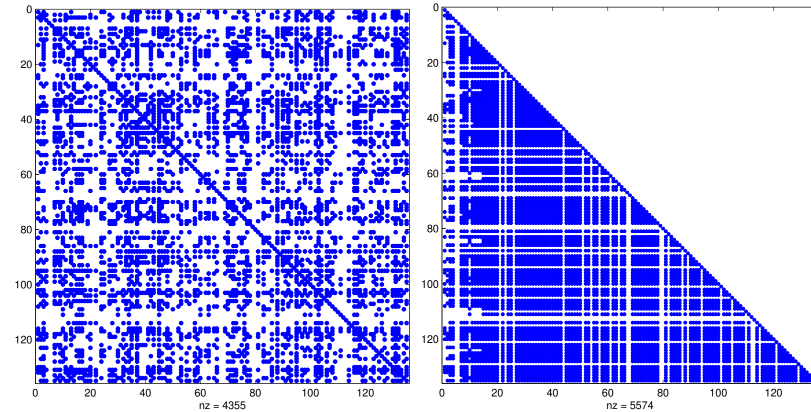


(v) The structure of the L-factor

Consider the case of symmetric matrices ($P = Q$) and three popular methods based on manipulations on the graph representation of the matrix.

- (generalized) reverse Cuthill-McKee algorithm (1969);
- nested dissection method (1973);
- minimum degree ordering (George and Liu, 1981) and variants.

A matrix from somewhere



Generalized Reverse Cuthill-McKee (RCM)

Aim: minimize the envelope (in other words a band of variable width) of the permuted matrix.

1. *Initialization.* Choose a starting (root) vertex r and set $v_1 = r$.
2. *Main loop.* For $i = 1, ..., n$ find all non-numbered neighbours of v_i and number them in the increasing order of their degrees.
3. *Reverse order.* The reverse Cuthill-McKee ordering is $w_1, ..., w_n$, where $w_i = v_{n+1-i}$.

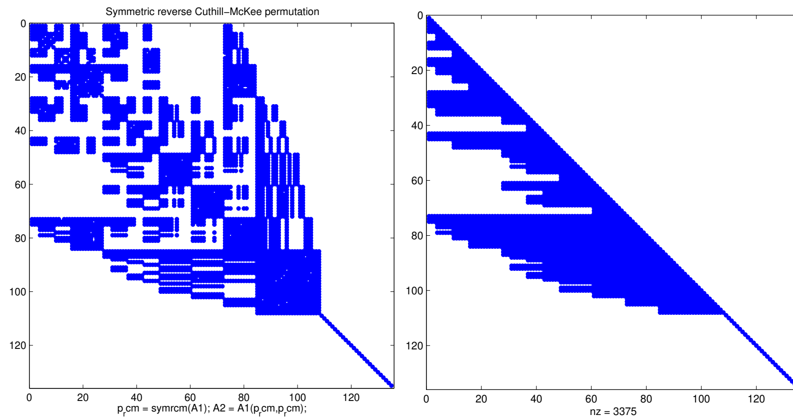
Generalized Reverse Cuthill-McKee (RCM)

One can see that GenRCM tends to number first the vertices adjoint to the already ordered ones, i.e., it gathers matrix entries along the main diagonal.

The choice of a root vertex is of a special interest.

The complexity of the algorithm is bounded from above by $O(m\ nnz(A))$, where m is a maximum degree of vertices, $nnz(A)$ - number of nonzero entries of matrix A.

Generalized Reverse Cuthill-McKee (RCM)

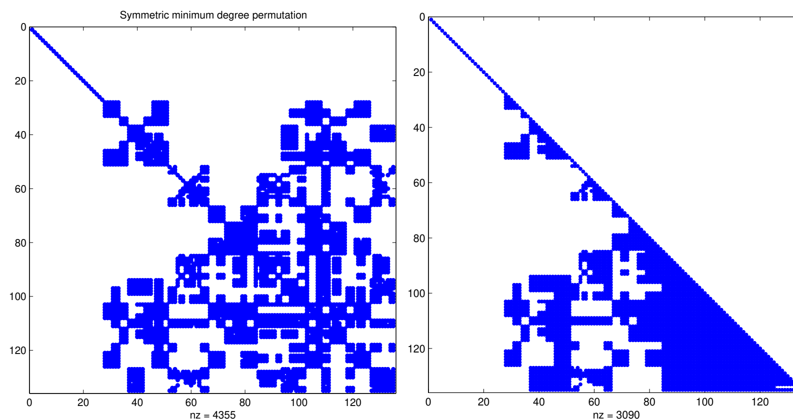


The Quotient Minimum Degree (QMD)

Aims to minimize a local fill-in taking a vertex of minimum degree at each elimination step. The straightforward implementation of the algorithm is time consuming since the degree of numerous vertices adjoint to the eliminated one must be recomputed at each step. Many important modifications have been made in order to improve the performance of the MD algorithm and this research remains still active .

In many references the MD algorithm is recommended as a general purpose fill-reducing reordering scheme. Its wide acceptance is largely due to its effectiveness in reducing fill and its efficient implementation.

The Quotient Minimum Degree (QMD)



The Nested Dissection algorithm

A recursive algorithm which on each step finds a separator of each connected graph component. A separator is a subset of vertices whose removal subdivides the graph into two or more components. Several strategies how to determine a separator in a graph are known. Numbering the vertices of the separator last results in the following structure of the permuted matrix with prescribed zero blocks in positions $(2, 1)$ and $(1, 2)$

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}.$$

The Nested Dissection algorithm

Under the assumption that subdivided components are of equal size the algorithm requires no more than $\boxed{\log_2 n}$ steps to terminate.

ND is optimal (up to a constant factor) for some class of model 2D problems originating from discretized PDEs. The Cholesky factor contains $O(m^2 log_2 m)$ nonzero entries. This is the *best low order bounds* derived for direct elimination methods.

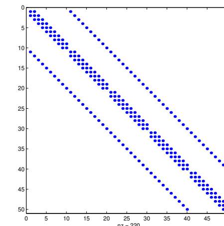
"Given-the-problem" strategy

Given-the-problem strategy

Assume we know the origin of the linear system of equations to be solved.
Example: the problem originates from a numerically discretized (system of) PDEs, and we know the domain of definition of the problem (Ω), its geometrical properties, the discretization method (finite differences (FD), finite elements (FE), finite volumes (FV), boundary integral (BE) method). In such cases the system matrix enjoys a special structure.
This information can be utilized while computing the matrix so that it will be constructed in (almost) favourable form.

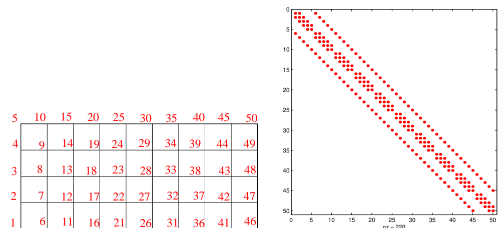
In the PDE world and not only...

(w) Row-wise ordering



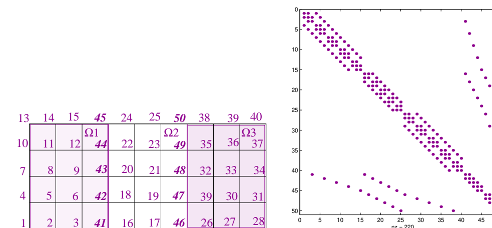
(x) The structure of the matrix A

In the PDE world and not only...



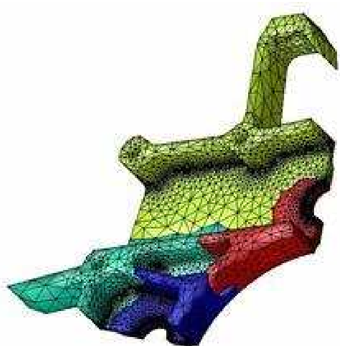
(y) Column-wise ordering (z) The structure of the matrix A

In the PDE world and not only...



() Domain-decomposition ordering () The structure of the matrix A

Parallel mesh partitioner PARMETIS (based on METIS)



() From PARMETIS galery

Sparse QR

$$A = LL^T$$
$$A = QR, \text{ then } A^T A = R^T Q^T Q R = R^T R!$$

Thus, if we know R, we have factorized $A^T A$!
But: if A is sparse, we want that R is as sparse as possible.
This is achieved by performing symbolic factorization of $A^T A$.

Sparse SVD, Matlab

The Matlab SVD implementation follows that of LINPACK, which is for general dense matrices.
To find some of the singular values (largest or smallest) of a large sparse matrix, one can use svds.
svds(A,k) uses eigs to find the k largest magnitude eigenvalues and corresponding eigenvectors of

$$B = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$$

.

Sparse SVD, Matlab

Demo:

```
load 20.dat
A=X20+tril(X20,1)';
S =svd(A);         0.27 s
AS=sparse(A);
SS=svds(AS,10);   0.09 s
```

```
S=svds(AS,5);
  S=svds(AS,5,0.51);
    S=svds(AS,5,0.01);
```

Summary:

- There is no 'best' method!
- The best code in any situation will depend on
  - the solution environment;
  - the computing platform;
  - the structure of the matrix.