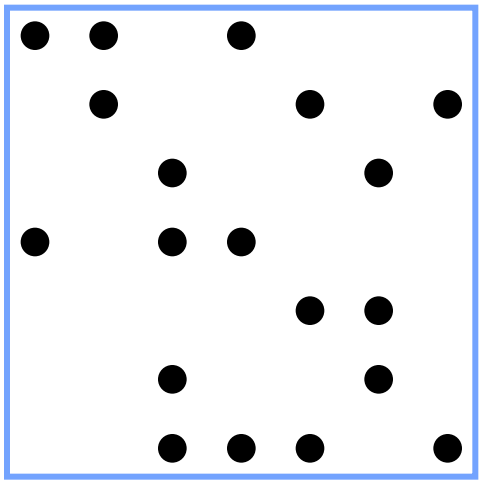
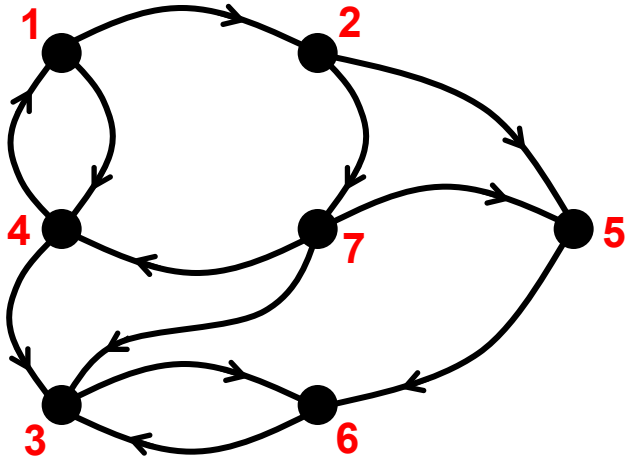
CS 240A: Solving Ax = b in parallel

- Dense A: Gaussian elimination with partial pivoting (LU)
  - Same flavor as matrix * matrix, but more complicated
- Sparse A: Gaussian elimination – Cholesky, LU, etc.
  - Graph algorithms
- Sparse A: Iterative methods – Conjugate gradient, etc.
  - Sparse matrix times dense vector
- Sparse A: Preconditioned iterative methods and multigrid
  - Mixture of lots of things

Matrix and Graph



A

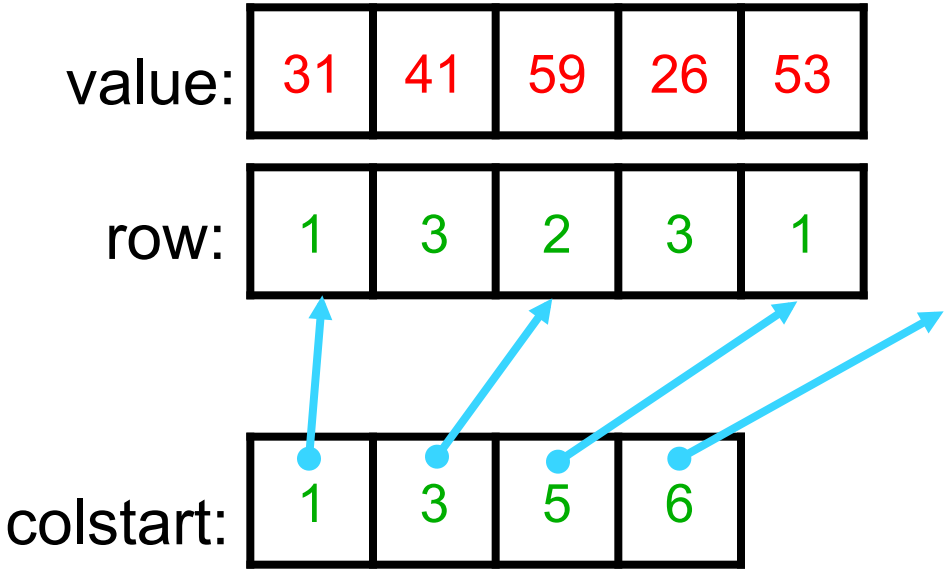


G(A)

- Edge from row i to column j for nonzero A(i,j)
- No edges for diagonal nonzeros
- If A is symmetric, G(A) is an undirected graph
- Symmetric permutation PAP^T renumbers the vertices

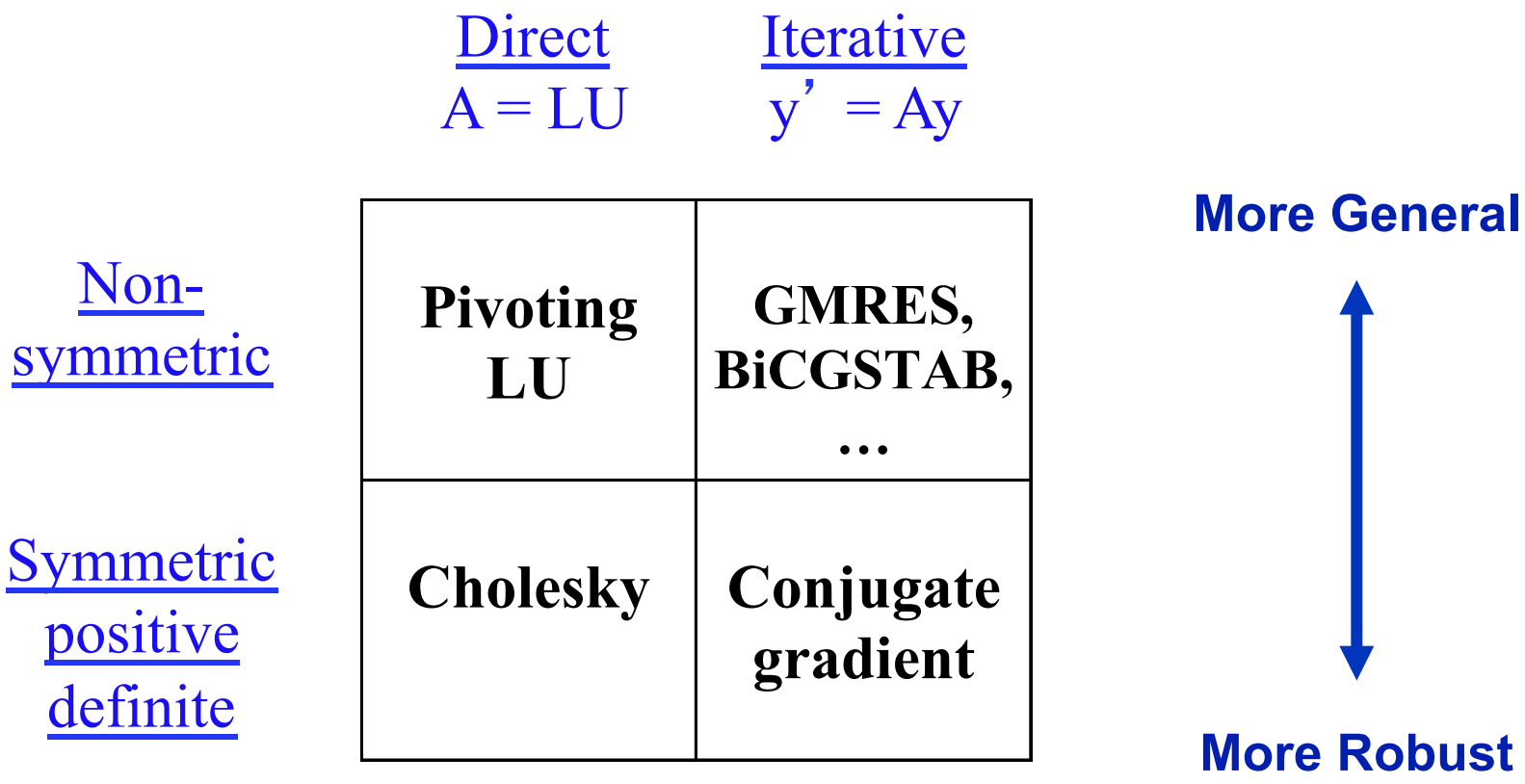
Compressed Sparse Matrix Storage

31	0	53
0	59	0
41	26	0



- Full storage:
  - 2-dimensional array.
  - (nrows*ncols) memory.
- Sparse storage:
  - Compressed storage by columns (CSC).
  - Three 1-dimensional arrays.
  - (2*nzs + ncols + 1) memory.
  - Similarly, CSR.

The Landscape of Ax=b Solvers



CS 240A: Solving Ax = b in parallel

- Dense A: Gaussian elimination with partial pivoting (LU)
  - See April 15 slides
  - Same flavor as matrix * matrix, but more complicated
- Sparse A: Gaussian elimination – Cholesky, LU, etc.
  - Graph algorithms
- Sparse A: Iterative methods – Conjugate gradient, etc.
  - Sparse matrix times dense vector
- Sparse A: Preconditioned iterative methods and multigrid
  - Mixture of lots of things

Gaussian elimination to solve $Ax = b$

For a symmetric, positive definite matrix:

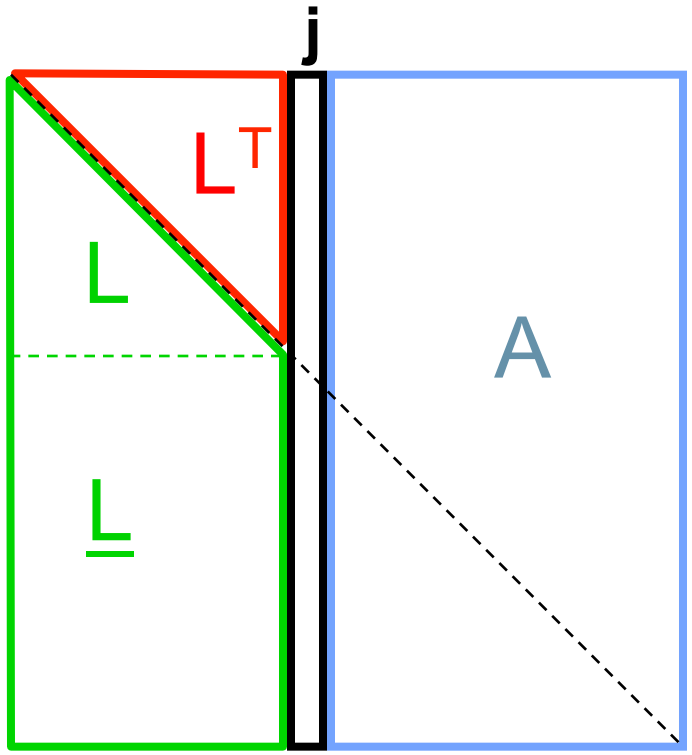
1. Matrix factorization: $A = LL^T$ (*Cholesky factorization*)
2. Forward triangular solve: $Ly = b$
3. Backward triangular solve: $L^T x = y$

For a nonsymmetric matrix:

1. Matrix factorization: $PA = LU$ (*Partial pivoting*)
2. . . .

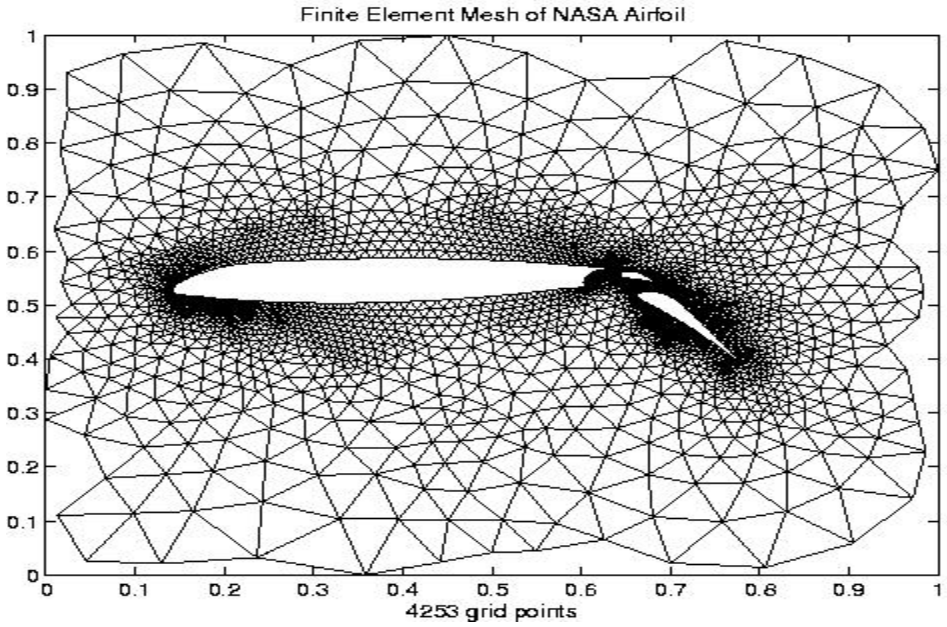
Sparse Column Cholesky Factorization

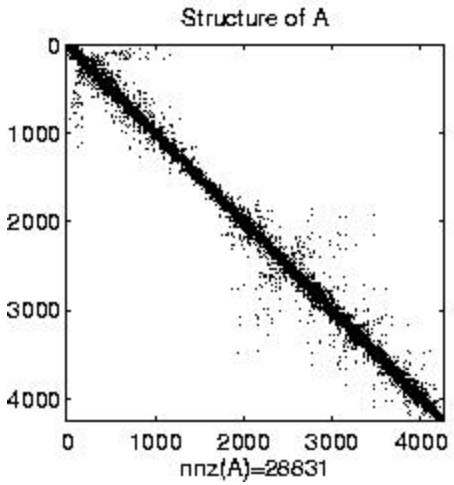
```
for j = 1 : n
   L(j:n, j) = A(j:n, j);
   for k < j with L(j, k) nonzero
      % sparse cmod(j,k)
      L(j:n, j) = L(j:n, j) – L(j, k) * L(j:n, k);
   end;
   % sparse cdiv(j)
   L(j, j) = sqrt(L(j, j));
   L(j+1:n, j) = L(j+1:n, j) / L(j, j);
end;
```

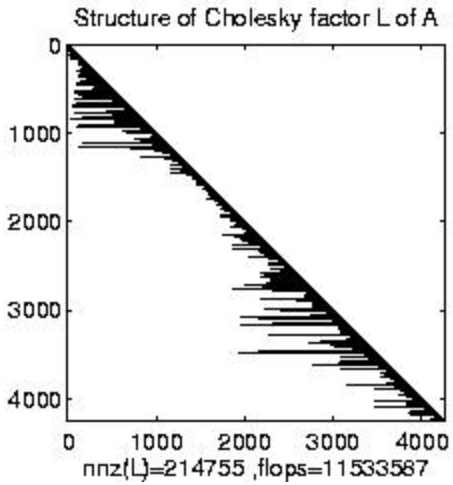


- Column j of A becomes column j of L

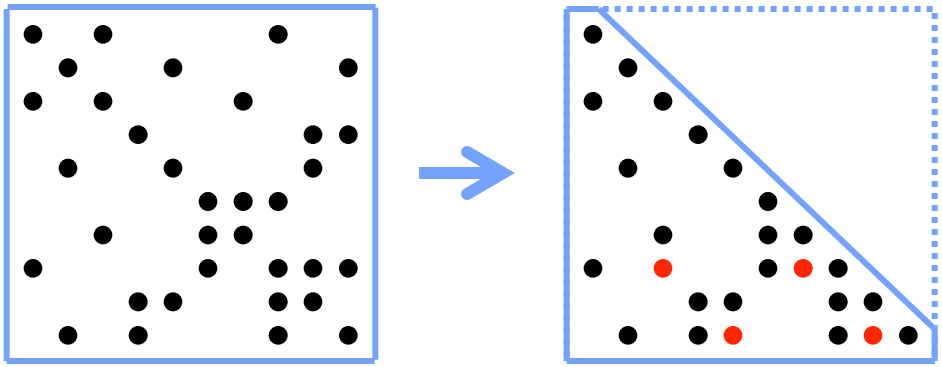
Irregular mesh: NASA Airfoil in 2D



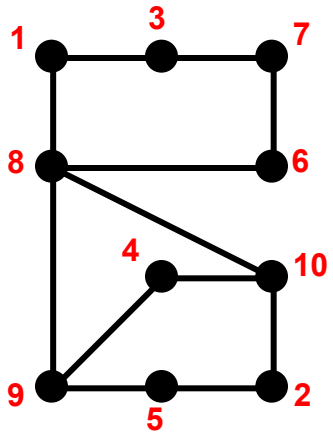




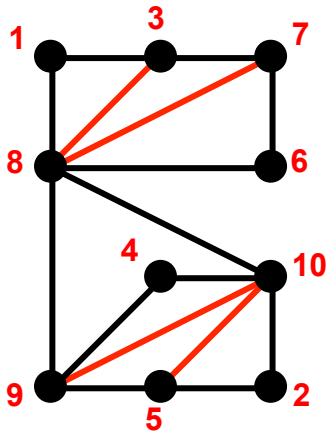
Graphs and Sparse Matrices: Cholesky factorization



Fill: new nonzeros in factor



G(A)



$G^+(A)$
[chordal]

Symmetric Gaussian elimination:
for j = 1 to n
add edges between j's higher-numbered neighbors

Permutations of the 2-D model problem

- Theorem: With the natural permutation, the n-vertex model problem has $\Theta(n^{3/2})$ fill. ("order exactly")

- Theorem: With any permutation, the n-vertex model problem has $\Omega(n \log n)$ fill. ("order at least")

- Theorem: With a *nested dissection* permutation, the n-vertex model problem has $O(n \log n)$ fill. ("order at most")

Nested dissection ordering

- A separator in a graph G is a set S of vertices whose removal leaves at least two connected components.

- A nested dissection ordering for an n-vertex graph G numbers its vertices from 1 to n as follows:
  - Find a separator S, whose removal leaves connected components T_1, T_2, …, T_k
  - Number the vertices of S from n-|S|+1 to n.
  - Recursively, number the vertices of each component: T_1 from 1 to $|T_1|$, T_2 from $|T_1|+1$ to $|T_1|+|T_2|$, etc.
  - If a component is small enough, number it arbitrarily.

- It all boils down to finding good separators!

Separators in theory

- If G is a planar graph with n vertices, there exists a set of at most sqrt(6n) vertices whose removal leaves no connected component with more than 2n/3 vertices. ("Planar graphs have sqrt(n)-separators.")

- "Well-shaped" finite element meshes in 3 dimensions have $n^{2/3}$ - separators.

- Also some other classes of graphs – trees, graphs of bounded genus, chordal graphs, bounded-excluded-minor graphs, …

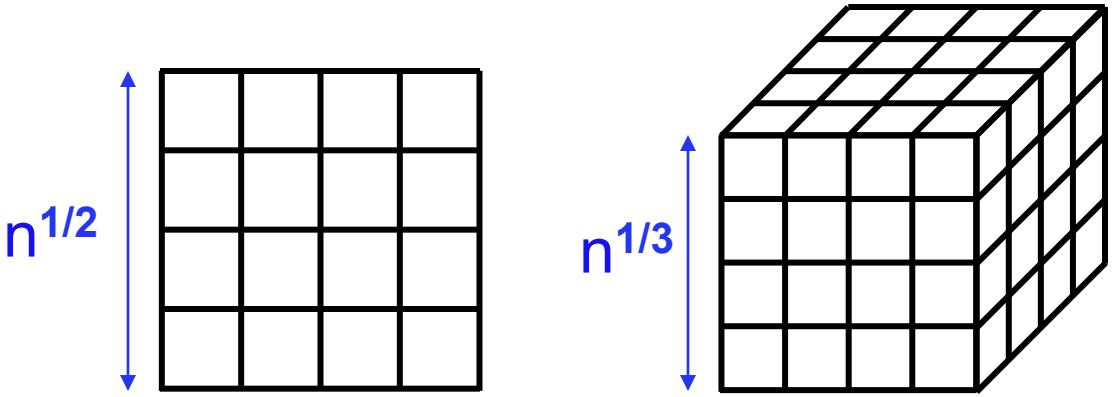
- Mostly these theorems come with efficient algorithms, but they aren't used much.

Separators in practice

- Graph partitioning heuristics have been an active research area for many years, often motivated by partitioning for parallel computation.
- Some techniques:
  - Spectral partitioning (uses eigenvectors of Laplacian matrix of graph)
  - Geometric partitioning (for meshes with specified vertex coordinates)
  - Iterative-swapping (Kernighan-Lin, Fiduccia-Matheysses)
  - Breadth-first search (fast but dated)
- Many popular modern codes (e.g. Metis, Chaco) use multilevel iterative swapping
- Matlab graph partitioning toolbox: see course web page

Complexity of direct methods

Time and space to solve any problem on any well-shaped finite element mesh

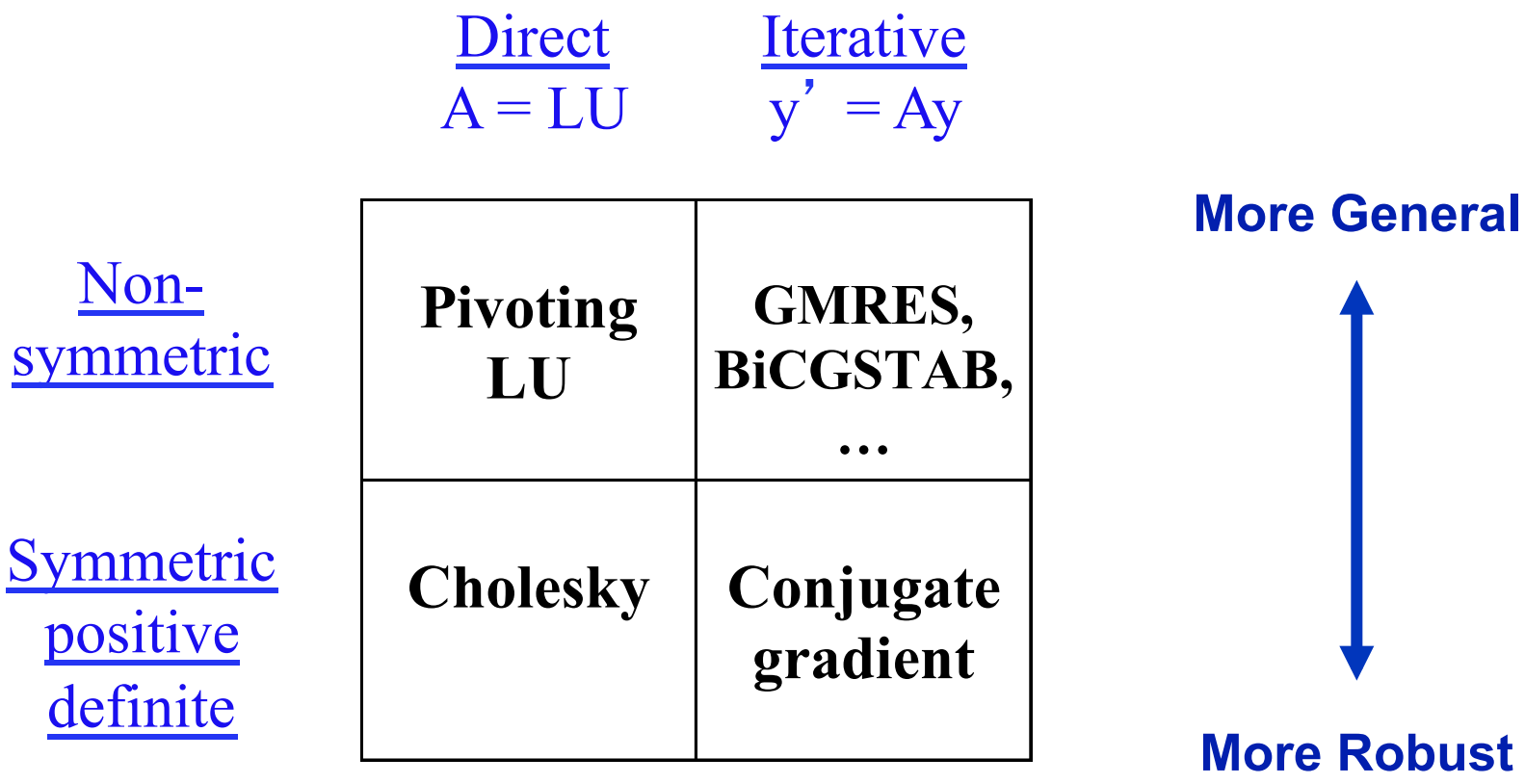


	2D	3D
Space (fill):	$O(n \log n)$	$O(n^{4/3})$
Time (flops):	$O(n^{3/2})$	$O(n^{2})$

CS 240A: Solving Ax = b in parallel

- Dense A: Gaussian elimination with partial pivoting (LU)
  - See April 15 slides
  - Same flavor as matrix * matrix, but more complicated
- Sparse A: Gaussian elimination – Cholesky, LU, etc.
  - Graph algorithms
- Sparse A: Iterative methods – Conjugate gradient, etc.
  - Sparse matrix times dense vector
- Sparse A: Preconditioned iterative methods and multigrid
  - Mixture of lots of things

The Landscape of Ax=b Solvers



Conjugate gradient iteration

$x_0 = 0, \quad r_0 = b, \quad d_0 = r_0$

for $k = 1, 2, 3, \ldots$

$\alpha_k = (r^T_{k-1} r_{k-1}) / (d^T_{k-1} A d_{k-1})$ step length

$x_k = x_{k-1} + \alpha_k d_{k-1}$ approx solution

$r_k = r_{k-1} - \alpha_k A d_{k-1}$ residual

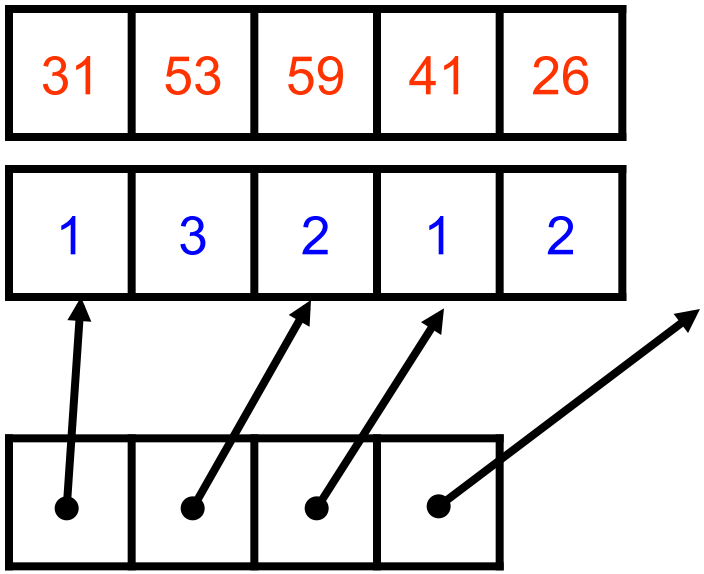
$\beta_k = (r^T_k r_k) / (r^T_{k-1} r_{k-1})$ improvement

$d_k = r_k + \beta_k d_{k-1}$ search direction

- One matrix-vector multiplication per iteration
- Two vector dot products per iteration
- Four n-vectors of working storage

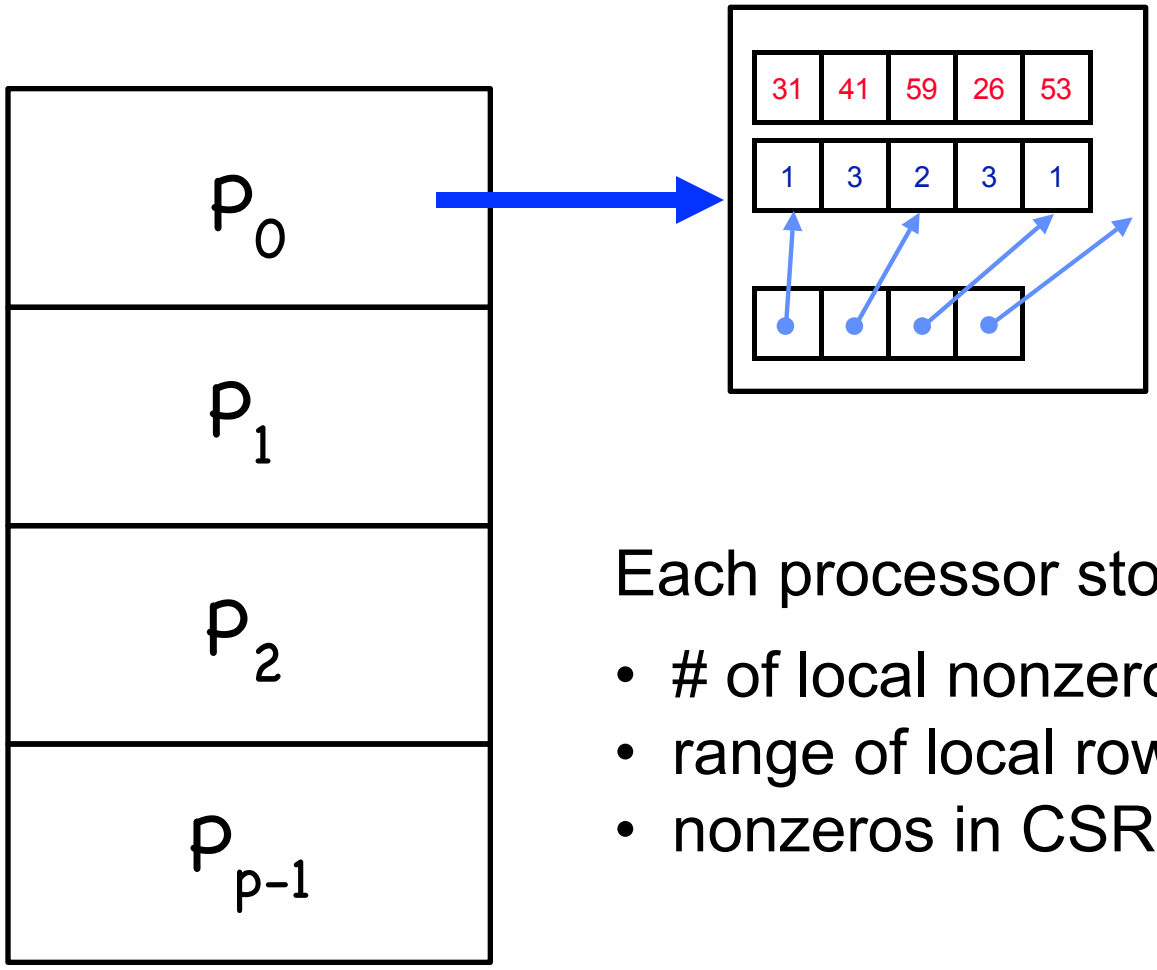
Sparse matrix data structure (stored by rows)

31	0	53
0	59	0
41	26	0



- Full:
  - 2-dimensional array of real or complex numbers
  - (nrows*ncols) memory
- Sparse:
  - compressed row storage
  - about (2*nzs + nrows) memory

Distributed row sparse matrix data structure



Each processor stores:

- # of local nonzeros
- range of local rows
- nonzeros in CSR form

Matrix-vector product: Parallel implementation

- Lay out matrix and vectors by rows

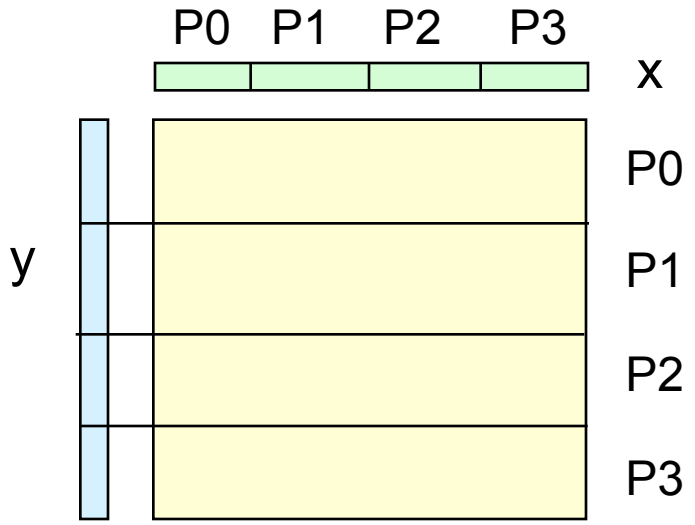
- y(i) = sum(A(i,j)*x(j))
- Skip terms with A(i,j) = 0

- Algorithm

  Each processor i:

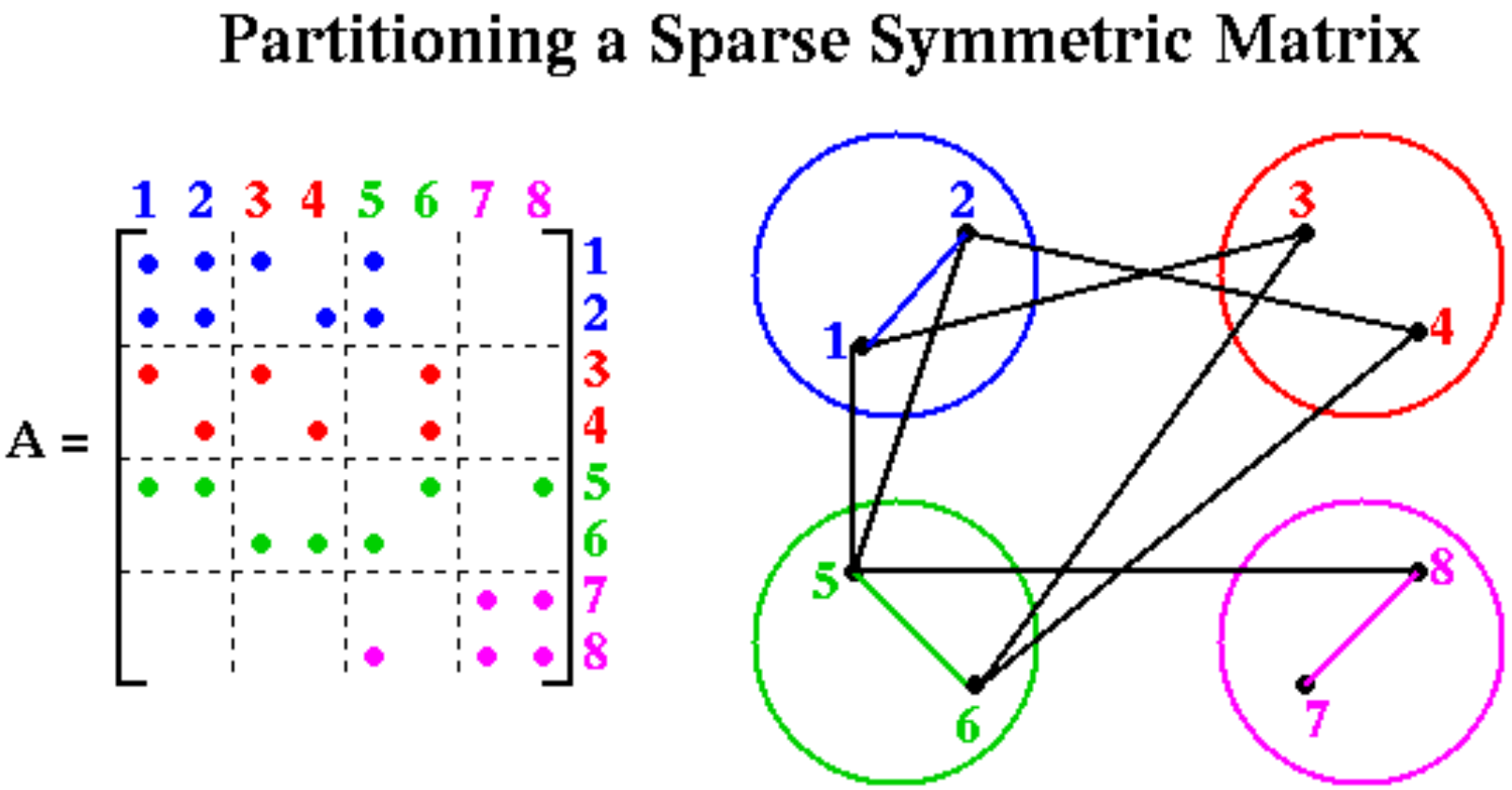
  Broadcast x(i)

  Compute y(i) = A(i,:)*x



- Optimizations: reduce communication by
  - Only send as much of x as necessary to each proc
  - Reorder matrix for better locality by graph partitioning

Sparse Matrix-Vector Multiplication



CS 240A: Solving Ax = b in parallel

- Dense A: Gaussian elimination with partial pivoting (LU)
  - See April 15 slides
  - Same flavor as matrix * matrix, but more complicated
- Sparse A: Gaussian elimination – Cholesky, LU, etc.
  - Graph algorithms
- Sparse A: Iterative methods – Conjugate gradient, etc.
  - Sparse matrix times dense vector
- Sparse A: Preconditioned iterative methods and multigrid
  - Mixture of lots of things

Conjugate gradient: Convergence

- In exact arithmetic, CG converges in n steps
  (completely unrealistic!!)
- Accuracy after k steps of CG is related to:
  - consider polynomials of degree k that are equal to 1 at 0.
  - how small can such a polynomial be at all the eigenvalues of A?
- Thus, eigenvalues close together are good.
- Condition number: $\kappa(A) = \|A\|_2 \|A^{-1}\|_2 = \lambda_{max}(A) / \lambda_{min}(A)$
- Residual is reduced by a constant factor by
  $O(\sqrt{\kappa(A)})$ iterations of CG.

Preconditioners

- Suppose you had a matrix B such that:
  1. condition number $\kappa(B^{-1}A)$ is small
  2. By = z is easy to solve

- Then you could solve $(B^{-1}A)x = B^{-1}b$ instead of $Ax = b$

- Each iteration of CG multiplies a vector by $B^{-1}A$:
  - First multiply by A
  - Then solve a system with B

Preconditioned conjugate gradient iteration

$x_0 = 0, \quad r_0 = b, \quad d_0 = B^{-1} r_0, \quad y_0 = B^{-1} r_0$

for $k = 1, 2, 3, \ldots$

$\alpha_k = (y^T_{k-1} r_{k-1}) / (d^T_{k-1} A d_{k-1})$ step length

$x_k = x_{k-1} + \alpha_k d_{k-1}$ approx solution

$r_k = r_{k-1} - \alpha_k A d_{k-1}$ residual

$y_k = B^{-1} r_k$ preconditioning solve

$\beta_k = (y^T_k r_k) / (y^T_{k-1} r_{k-1})$ improvement

$d_k = y_k + \beta_k d_{k-1}$ search direction

- One matrix-vector multiplication per iteration
- One solve with preconditioner per iteration

Choosing a good preconditioner

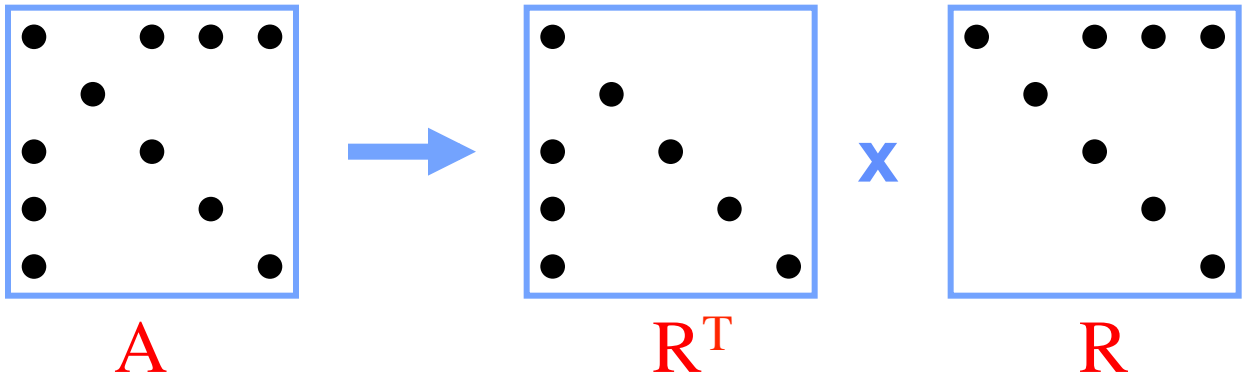
- Suppose you had a matrix B such that:
  1. condition number $\kappa(B^{-1}A)$ is small
  2. By = z is easy to solve

- Then you could solve $(B^{-1}A)x = B^{-1}b$ instead of $Ax = b$

- B = A is great for (1), not for (2)
- B = I is great for (2), not for (1)
- Domain-specific approximations sometimes work
- B = diagonal of A sometimes works

- Better: blend in some direct-methods ideas. . .

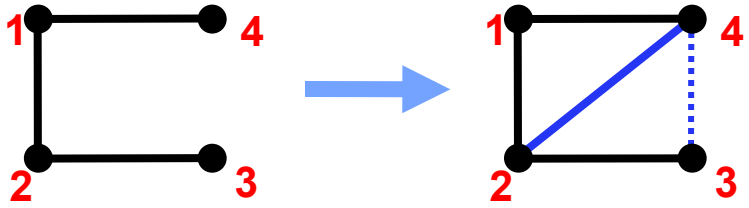
Incomplete Cholesky factorization (IC, ILU)



- Compute factors of A by Gaussian elimination, but ignore fill
- Preconditioner $B = R^T R \approx A$, not formed explicitly
- Compute $B^{-1}z$ by triangular solves (in time nnz(A))
- Total storage is O(nnz(A)), static data structure
- Either symmetric (IC) or nonsymmetric (ILU)

Incomplete Cholesky and ILU: Variants

- Allow one or more "levels of fill"
  - unpredictable storage requirements



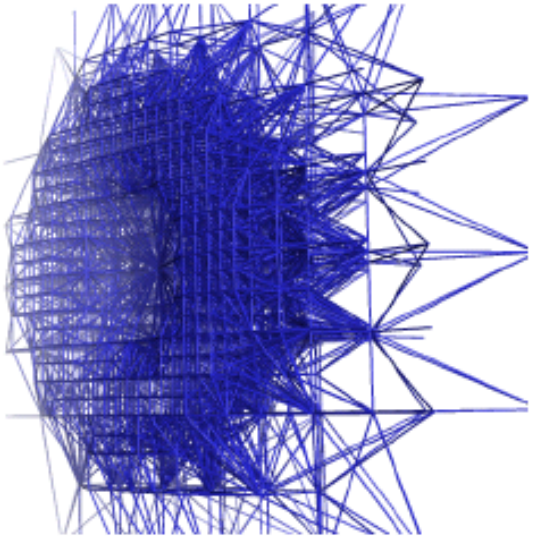
- Allow fill whose magnitude exceeds a "drop tolerance"
  - may get better approximate factors than levels of fill
  - unpredictable storage requirements
  - choice of tolerance is ad hoc
- Partial pivoting (for nonsymmetric A)
- "Modified ILU" (MIC): Add dropped fill to diagonal of U or R
  - A and $R^T R$ have same row sums
  - good in some PDE contexts

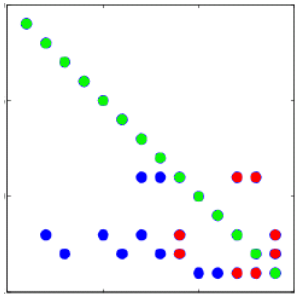
Incomplete Cholesky and ILU: Issues

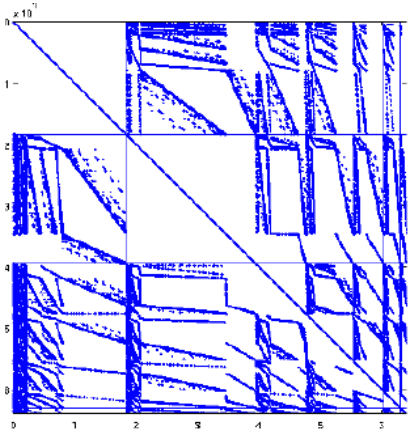
- Choice of parameters
  - good: smooth transition from iterative to direct methods
  - bad: very ad hoc, problem-dependent
  - tradeoff: time per iteration (more fill => more time)
    vs # of iterations (more fill => fewer iters)
- Effectiveness
  - condition number usually improves (only) by constant factor (except MIC for some problems from PDEs)
  - still, often good when tuned for a particular class of problems
- Parallelism
  - Triangular solves are not very parallel
  - Reordering for parallel triangular solve by graph coloring

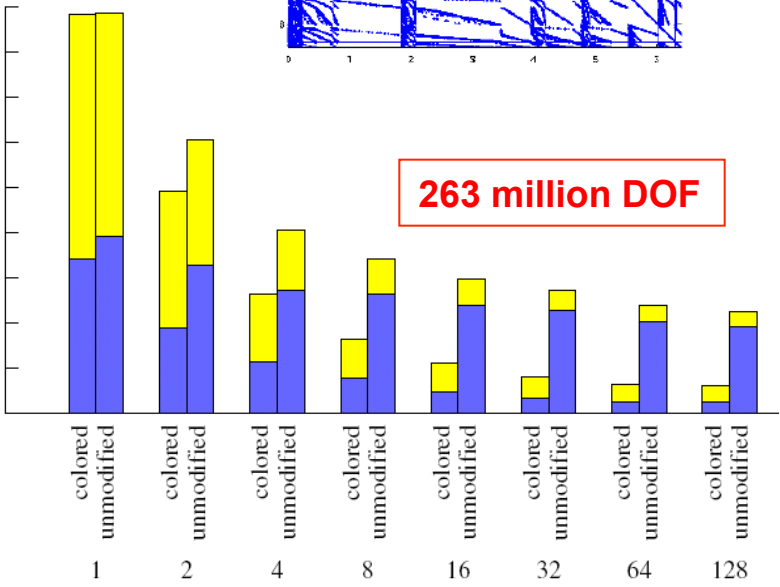
Coloring for parallel nonsymmetric preconditioning

[Aggarwal, Gibou, G]



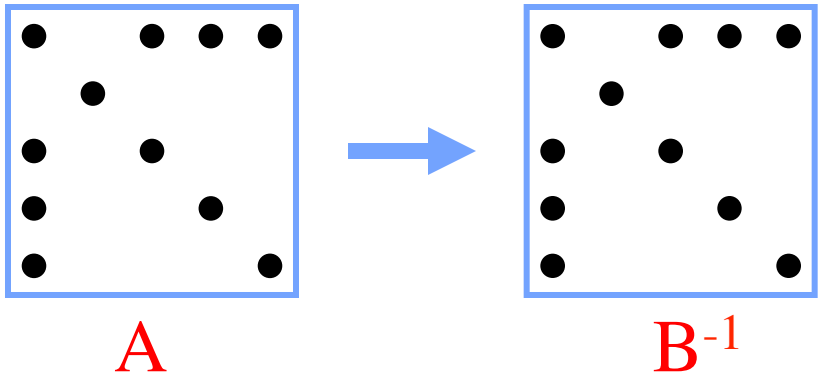






- Level set method for multiphase interface problems in 3D
- Nonsymmetric-structure, second-order-accurate octree discretization.
- BiCGSTAB preconditioned by parallel triangular solves.

Sparse approximate inverses



- Compute $B^{-1} \approx A$ explicitly
- Minimize $\| B^{-1}A - I \|_F$ (in parallel, by columns)
- Variants: factored form of B^{-1}, more fill, . .
- Good: very parallel
- Bad: effectiveness varies widely

Other Krylov subspace methods

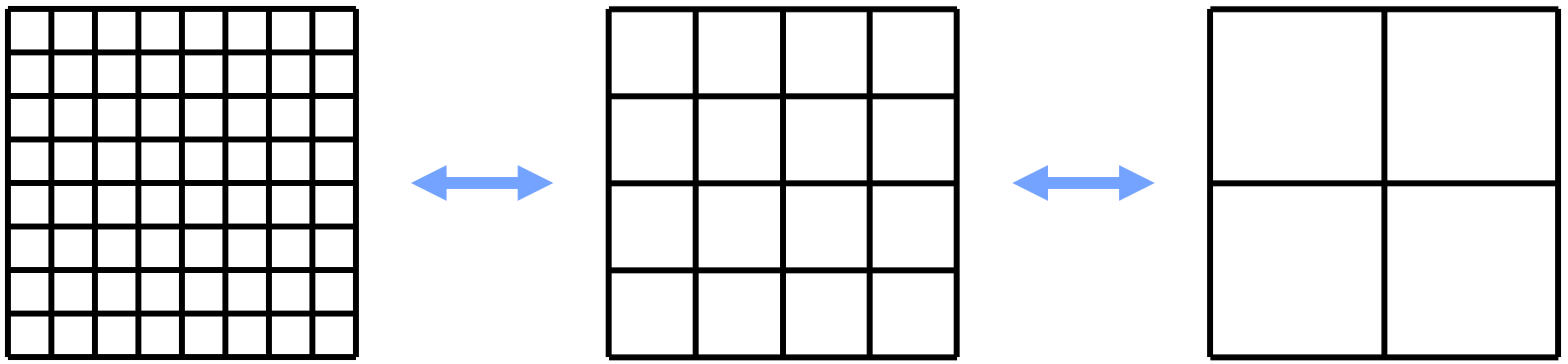
- Nonsymmetric linear systems:
  - GMRES:
    for i = 1, 2, 3, . . .
    find $x_i \in K_i(A, b)$ such that $r_i = (Ax_i - b) \perp K_i(A, b)$
    But, no short recurrence => save old vectors => lots more space
    (Usually "restarted" every k iterations to use less space.)

  - BiCGStab, QMR, etc.:
    Two spaces $K_i(A, b)$ and $K_i(A^T, b)$ w/ mutually orthogonal bases
    Short recurrences => O(n) space, but less robust

  - Convergence and preconditioning more delicate than CG
  - Active area of current research

- Eigenvalues: Lanczos (symmetric), Arnoldi (nonsymmetric)

Multigrid

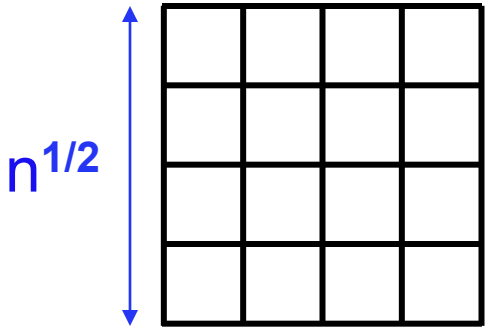


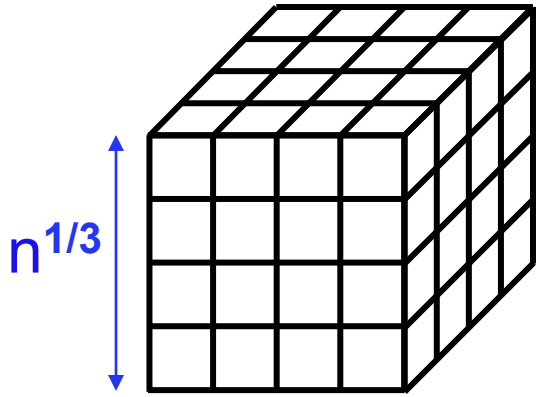
- For a PDE on a fine mesh, precondition using a solution on a coarser mesh
- Use idea recursively on hierarchy of meshes
- Solves the model problem (Poisson's eqn) in linear time!
- Often useful when hierarchy of meshes can be built
- Hard to parallelize coarse meshes well

- This is just the intuition – lots of theory and technology

Complexity of linear solvers

Time to solve model problem (Poisson's equation) on regular mesh

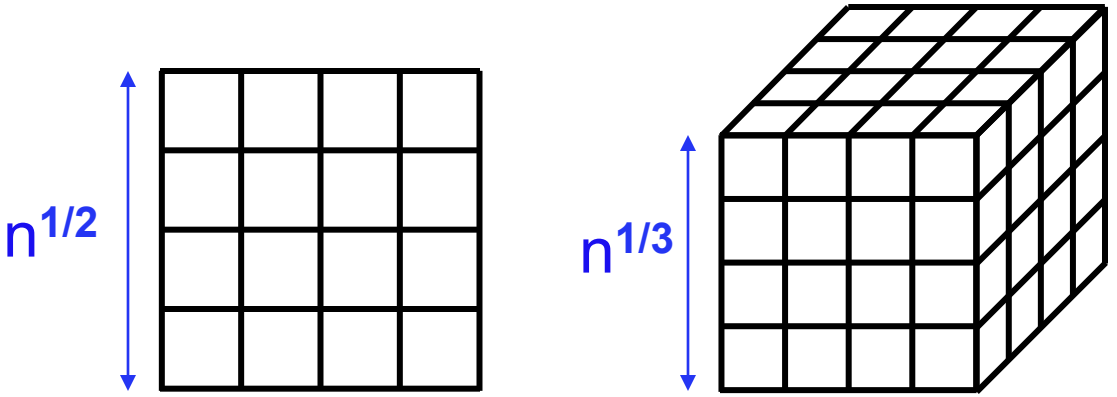




	2D	3D
Sparse Cholesky:	$O(n^{1.5})$	$O(n^{2})$
CG, exact arithmetic:	$O(n^{2})$	$O(n^{2})$
CG, no precond:	$O(n^{1.5})$	$O(n^{1.33})$
CG, modified IC:	$O(n^{1.25})$	$O(n^{1.17})$
CG, support trees:	$O(n^{1.20})$ -> $O(n^{1+})$	$O(n^{1.75})$ -> $O(n^{1+})$
Multigrid:	$O(n)$	$O(n)$

Complexity of direct methods

Time and space to solve any problem on any well-shaped finite element mesh



	2D	3D
Space (fill):	$O(n \log n)$	$O(n^{4/3})$
Time (flops):	$O(n^{3/2})$	$O(n^{2})$