Parallel Numerical Algorithms

Chapter 4 – Sparse Linear Systems
Section 4.3 – Iterative Methods

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

Iterative Methods for Linear Systems

- Iterative methods for solving linear system $\boldsymbol{Ax} = \boldsymbol{b}$ begin with initial guess for solution and successively improve it until solution is as accurate as desired
- In theory, infinite number of iterations might be required to converge to exact solution
- In practice, iteration terminates when residual $\|\boldsymbol{b} - \boldsymbol{Ax}\|$, or some other measure of error, is as small as desired
- Iterative methods are especially useful when matrix $\boldsymbol{A}$ is sparse because, unlike direct methods, no fill is incurred

Jacobi Method

- Beginning with initial guess $\boldsymbol{x}^{(0)}$, *Jacobi method* computes next iterate by solving for each component of $\boldsymbol{x}$ in terms of others

$$x_i^{(k+1)} = \Big(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\Big) / a_{ii}, \quad i = 1, \dots, n$$

- If $\boldsymbol{D}$, $\boldsymbol{L}$, and $\boldsymbol{U}$ are diagonal, strict lower triangular, and strict upper triangular portions of $\boldsymbol{A}$, then Jacobi method can be written

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}\left(\boldsymbol{b} - (\boldsymbol{L} + \boldsymbol{U})\boldsymbol{x}^{(k)}\right)$$

Jacobi Method

- Jacobi method requires nonzero diagonal entries, which can usually be accomplished by permuting rows and columns if not already true
- Jacobi method requires duplicate storage for x, since no component can be overwritten until all new values have been computed
- Components of new iterate do not depend on each other, so they can be computed simultaneously
- Jacobi method does not always converge, but it is guaranteed to converge under conditions that are often satisfied (e.g., if matrix is strictly diagonally dominant), though convergence rate may be very slow

Gauss-Seidel Method

- Faster convergence can be achieved by using each new component value as soon as it has been computed rather than waiting until next iteration
- This gives *Gauss-Seidel method*

$$x_i^{(k+1)} = \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}\right) / a_{ii}$$

- Using same notation as for Jacobi, Gauss-Seidel method can be written

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{D} + \boldsymbol{L})^{-1} \left(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)}\right)$$

Gauss-Seidel Method

- Gauss-Seidel requires nonzero diagonal entries
- Gauss-Seidel does not require duplicate storage for x, since component values can be overwritten as they are computed
- But each component depends on previous ones, so they must be computed successively
- Gauss-Seidel does not always converge, but it is guaranteed to converge under conditions that are somewhat weaker than those for Jacobi method (e.g., if matrix is symmetric and positive definite)
- Gauss-Seidel converges about twice as fast as Jacobi, but may still be very slow

Conjugate Gradient Method

- If $\boldsymbol{A}$ is $n \times n$ symmetric positive definite matrix, then quadratic function

$$\phi(x) = \tfrac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}$$

  attains minimum precisely when $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

- Optimization methods have form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha\, \boldsymbol{s}_k$$

  where α is search parameter chosen to minimize objective function $\phi(\boldsymbol{x}_k + \alpha\, \boldsymbol{s}_k)$ along $\boldsymbol{s}_k$

- For method of *steepest descent*, $\boldsymbol{s}_k = -\nabla\phi(\boldsymbol{x})$

Conjugate Gradient Method

For special case of quadratic problem,

- Negative gradient is residual vector

$$-\nabla\phi(\boldsymbol{x}) = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x} = \boldsymbol{r}$$

- Optimal line search parameter is given by

$$\alpha = \boldsymbol{r}_k^T \boldsymbol{s}_k / \boldsymbol{s}_k^T \boldsymbol{A} \boldsymbol{s}_k$$

- Successive search directions can easily be $\boldsymbol{A}$-orthogonalized by three-term recurrence
- Using these properties, we obtain *conjugate gradient method* (*CG*) for linear systems

Conjugate Gradient Method

```
$\boldsymbol{x}_0 =$ initial guess
$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$
$\boldsymbol{s}_0 = \boldsymbol{r}_0$
for $k = 0, 1, 2, \ldots$
    $\alpha_k = \boldsymbol{r}_k^T \boldsymbol{r}_k / \boldsymbol{s}_k^T \boldsymbol{A}\boldsymbol{s}_k$
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$
    $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{A}\boldsymbol{s}_k$
    $\beta_{k+1} = \boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1} / \boldsymbol{r}_k^T \boldsymbol{r}_k$
    $\boldsymbol{s}_{k+1} = \boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{s}_k$
end
```

Conjugate Gradient Method

- Key features that make CG method effective
  - Short recurrence determines search directions that are A-orthogonal (conjugate)
  - Error is minimal over space spanned by search directions generated so far
- Minimum error property implies that method produces exact solution after at most n steps
- In practice, rounding error causes loss of orthogonality that spoils finite termination property, so method is used iteratively

Conjugate Gradient Method

- Error is reduced at each iteration by factor of

$$(\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$$

  on average, where

$$\kappa = \operatorname{cond}(\boldsymbol{A}) = \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\| = \lambda_{\max}(\boldsymbol{A})/\lambda_{\min}(\boldsymbol{A})$$

- Thus, convergence tends to be rapid if matrix is well-conditioned, but can be arbitrarily slow if matrix is ill-conditioned
- But convergence also depends on clustering of eigenvalues of $\boldsymbol{A}$

Nonsymmetric Krylov Subspace Methods

- CG is not directly applicable to nonsymmetric or indefinite systems
- CG cannot be generalized to nonsymmetric systems without sacrificing one of its two key properties (short recurrence and minimum error)
- Nevertheless, several generalizations have been developed for solving nonsymmetric systems, including GMRES, QMR, CGS, BiCG, and Bi-CGSTAB
- These tend to be less robust and require more storage than CG, but they can still be very useful for solving large nonsymmetric systems

Parallel Implementation

- Iterative methods for linear systems are composed of basic operations such as
  - vector updates (`saxpy`)
  - inner products
  - matrix-vector multiplication
  - solution of triangular systems
- In parallel implementation, both data and operations are partitioned across multiple tasks
- In addition to communication required for these basic operations, necessary convergence test may require additional communication (e.g., sum or max reduction)

Partitioning of Vectors

- Iterative methods typically require several vectors, including solution $\boldsymbol{x}$, right-hand side $\boldsymbol{b}$, residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$, and possibly others
- Even when matrix $\boldsymbol{A}$ is sparse, these vectors are usually dense
- These dense vectors are typically uniformly partitioned among p tasks, with given task holding same set of component indices of each vector
- Thus, vector updates require no communication, whereas inner products of vectors require reductions across tasks, at cost we have already seen

Partitioning of Sparse Matrix

- Sparse matrix A can be partitioned among tasks by rows, by columns, or by submatrices
- Partitioning by submatrices may give uneven distribution of nonzeros among tasks; indeed, some submatrices may contain no nonzeros at all
- Partitioning by rows or by columns tends to yield more uniform distribution because sparse matrices typically have about same number of nonzeros in each row or column

Row Partitioning of Sparse Matrix

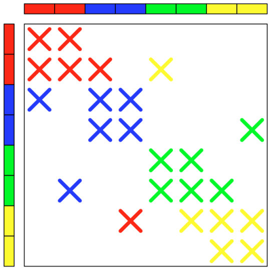
- Suppose that each task is assigned n/p rows, yielding p tasks, where for simplicity we assume that p divides n
- In dense matrix-vector multiplication, since each task owns only n/p components of vector operand, communication is required to obtain remaining components
- If matrix is sparse, however, few components may actually be needed, and these should preferably be stored in neighboring tasks
- Assignment of rows to tasks by contiguous blocks or cyclically would not, in general, result in desired proximity of vector components

Graph Partitioning

- Desired data locality can be achieved by partitioning graph of matrix, or partitioning underlying grid or mesh for finite difference or finite element problem
- For example, graph can be partitioned into p pieces by nested dissection, and vector components corresponding to nodes in each resulting piece assigned to same task, with neighboring pieces assigned to neighboring tasks
- Then matrix-vector product requires relatively little communication, and only between neighboring tasks

Two-Dimensional Partitioning

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} 1 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 9 & 0 & 5 & 0 & 0 & 0 \\ 8 & 0 & 1 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 1 & 8 & 0 & 0 \\ 0 & 4 & 0 & 0 & 3 & 1 & 3 & 0 \\ 0 & 0 & 0 & 6 & 0 & 9 & 1 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$



Sparse MatVec with 2-D Partitioning

- Partition entries of both $\boldsymbol{x}$ and $\boldsymbol{y}$ across processors
- Partition entries of $\boldsymbol{A}$ accordingly

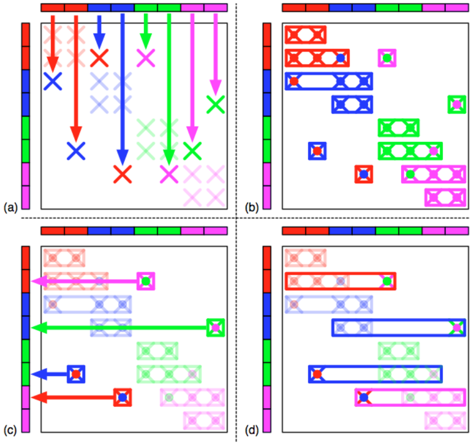
(a) Send entries $\boldsymbol{x}_j$ to processors with nonzero a_{ij} for some i

(b) Local multiply-add: $y_i = y_i + a_{ij}x_j$

(c) Send partial inner products to relevant processors

(d) Local sum: sum partial inner products

Sparse MatVec with 2-D Partitioning



Graph Partitioning Methods Types

Coordinate-based

- Coordinate bisection
- Inertial
- Geometric
- Multilevel

Coordinate-free

- Level structure
- Spectral
- Combinatorial refinement (e.g., Kernighan-Lin)

Graph Partitioning Software

- Chaco
- Jostle
- Meshpart
- Metis/ParMetis
- Mondriaan
- Party
- Scotch
- Zoltan

Coordinate-free partitioning

Partitioning is hard for arbitrary graphs

- Level partitioning – breadth-first-search (BFS) tree levels
- Maximal k-independent sets – select vertices separated by distance k and create partitions by combining vertices with nearest vertex in k-independent set
- Spectral partitioning looks at eigenvalues of *Laplacian matrix* $\boldsymbol{L}$ of a graph $G = (V, E)$,

$$l_{ij} = \begin{cases} i = j & : \text{degree}(V(i)) \\ (i,j) \in E & : -1 \\ (i,j) \notin E & : 0 \end{cases}$$

  the eigenvector of L with the second smallest eigenvalue (*the Fiedler vector*) provides a good partition of G

Coordinate-based partitioning

- Assume graph embedded in d-dimensional space, so we have *coordinates* for nodes
- In a mesh, only neighboring points will be adjacent
- G. Miller, S. Teng, W. Thurston, S. Vavasis (1997) provide a general algorithm for finding an optimal geometric partition
  - points are projected onto surface of a sphere
  - centerpoint (generalized median) of points is computed
  - sphere is rotated and dilated so centerpoint becomes origin
  - points adjacent to any hyperplane containing the centerpoint provides a good partition
- Gives vertex separators of size $O(n^{(d-1)/d})$ for meshes with constant *aspect ratio* – max relative distance of edges in space

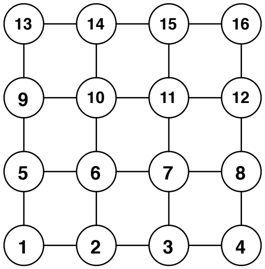
Parallel Jacobi

- We have already seen example of this approach with Jacobi method for 1-D Laplace equation
- Contiguous groups of variables are assigned to each task, so most communication is internal, and external communication is limited to nearest neighbors in 1-D mesh
- More generally, Jacobi method usually parallelizes well if underlying grid is partitioned in this manner, since all components of x can be updated simultaneously

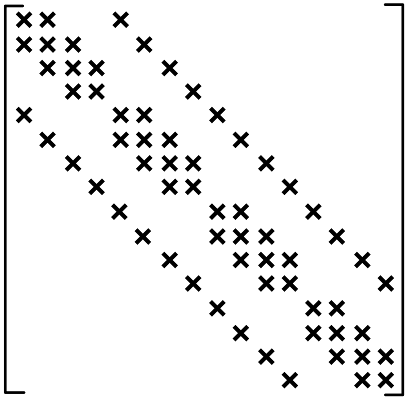
Parallel Gauss-Seidel

- Unfortunately, Gauss-Seidel methods require successive updating of solution components in given order (in effect, solving triangular system), rather than permitting simultaneous updating as in Jacobi method

Row-Wise Ordering for 2-D Grid



***G* (*A*)**

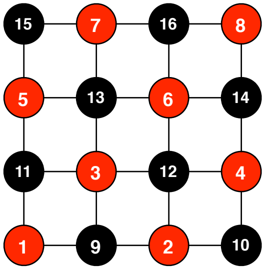


A

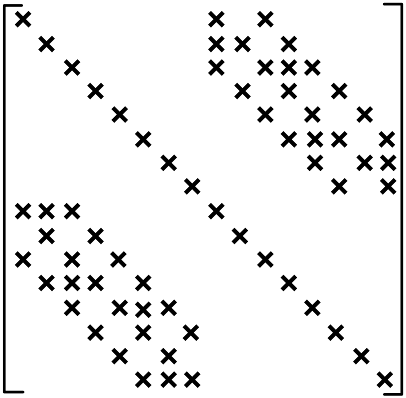
Red-Black Ordering

- Apparent sequential order can be broken, however, if components are reordered according to *coloring* of underlying graph
- For 5-point discretization on square grid, for example, color alternate nodes in each dimension red and others black, giving color pattern of chess or checker board
- Then all red nodes can be updated simultaneously, as can all black nodes, so algorithm proceeds in alternating phases, first updating all nodes of one color, then those of other color, repeating until convergence

Red-Black Ordering for 2-D Grid



***G* (*A*)**



A

Multicolor Orderings

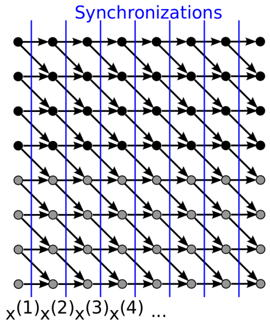
- More generally, arbitrary graph requires more colors, so there is one phase per color in parallel algorithm
- Nodes must also be partitioned among tasks, and load should be balanced for each color
- Reordering nodes may affect convergence rate, however, so gain in parallel performance may be offset somewhat by slower convergence rate

Sparse Triangular Systems

- More generally, multicolor ordering of graph of matrix enhances parallel performance of sparse triangular solution by identifying sets of solution components that can be computed simultaneously (rather than in usual sequential order for triangular solution)

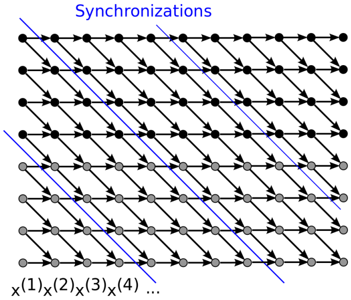
Parallel 1D 2-pt Stencil

Normally, halo exchange done before every stencil application



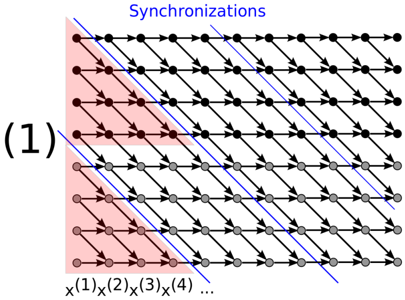
In-time Blocking

Instead apply stencil repeatedly before larger halo exchange



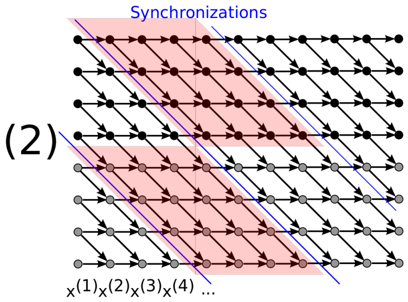
In-time Blocking

Instead apply stencil repeatedly before larger halo exchange



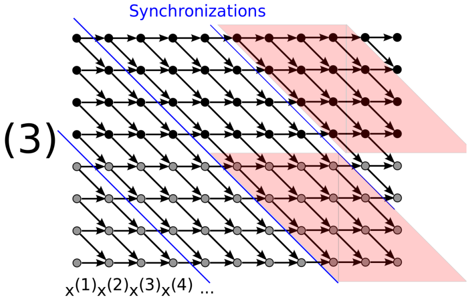
In-time Blocking

Instead apply stencil repeatedly before larger halo exchange



In-time Blocking

Instead apply stencil repeatedly before larger halo exchange



Analysis of in-time blocking for 1D mesh

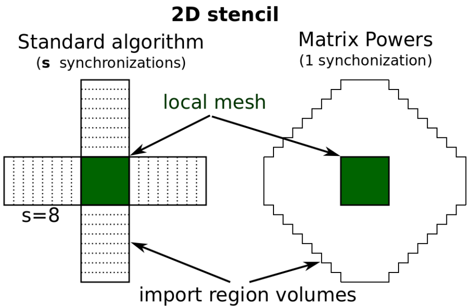
For 1-D 2-pt stencil (3-pt stencil similar)

- Consider t steps, and execute s without messages
- Bring down latency cost by a factor of s, for $t = \Theta(n)$, we improve latency cost from $\Theta(\alpha n)$ to $\Theta(\alpha p)$ with $s = n/p$
- Also improves flop-to-byte ratio of local subcomputations

Analysis of in-time blocking for 2-D mesh

For 2-D mesh, there is more complexity

- Consider t steps and execute $s \leq \sqrt{n/p}$ without msgs
- Need to do asymptotically more computation and interprocessor communication if $s > \sqrt{n/p}$



Asynchronous or Chaotic Relaxation

- Using updated values for solution components in Gauss-Seidel methods improves convergence rate, but limits parallelism and requires synchronization
- Alternatively, in computing next iterate, each processor could use most recent value it has for each solution component, rather than waiting for latest value on any processor
- This approach, sometimes called *asynchronous* or *chaotic* relaxation, can be effective, but stochastic behavior complicates analysis of convergence and convergence rate

Preconditioning

- Convergence rate of iterative methods depends on condition number and can often be substantially accelerated by *preconditioning*
- Apply method to $\boldsymbol{M}^{-1}\boldsymbol{A}$, where $\boldsymbol{M}$ is chosen so that $\boldsymbol{M}^{-1}\boldsymbol{A}$ is better conditioned than $\boldsymbol{A}$, and systems of form $\boldsymbol{M}\boldsymbol{z} = \boldsymbol{A}\boldsymbol{y}$ are easily solved for $\boldsymbol{z}$
- Typically, $\boldsymbol{M}$ is (block)-diagonal or triangular

Basic Preconditioners: $\boldsymbol{M}$ for $\boldsymbol{M}^{-1}\boldsymbol{A}$

- Polynomial (most commonly Chebyshev)

$$\boldsymbol{M}^{-1} = \text{poly}(\boldsymbol{A})$$

  neither $\boldsymbol{M}$ nor $\boldsymbol{M}^{-1}$ explicitly formed, latter applied by multiple SpMVs

- Diagonal (Jacobi) (diagonal scaling)

$$\boldsymbol{M} = \text{diag}(\boldsymbol{d})$$

  sometime can use $\boldsymbol{d} = \text{diag}(\boldsymbol{A})$, easy but ineffective

Block Preconditioners: M for $M^{-1}A$

Consider partitioning

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{E} \\ \boldsymbol{F} & \boldsymbol{C} \end{bmatrix} = \begin{bmatrix} \boldsymbol{B}_1 & & & \boldsymbol{E}_1 & & \\ & \ddots & & & \ddots & \\ & & \boldsymbol{B}_p & & & \boldsymbol{E}_p \\ \boldsymbol{F}_1 & & & \boldsymbol{C}_{11} & \dots & \boldsymbol{C}_{1p} \\ & \ddots & & \vdots & \ddots & \vdots \\ & & \boldsymbol{F}_p & \boldsymbol{C}_{p1} & \dots & \boldsymbol{C}_{pp} \end{bmatrix}$$

- Block-diagonal (domain decomposition)

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{B} & \\ & \boldsymbol{I} \end{bmatrix} \quad \text{so} \quad \boldsymbol{M}^{-1}\boldsymbol{A}\begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_1 + \boldsymbol{B}^{-1}\boldsymbol{E}\boldsymbol{y}_2 \\ \boldsymbol{F}\boldsymbol{y}_1 + \boldsymbol{C}\boldsymbol{y}_2 \end{bmatrix}$$

iterative methods with $\boldsymbol{M}^{-1}\boldsymbol{A}$ can compute each $\boldsymbol{B}_i^{-1}$ in parallel to get and apply $\boldsymbol{B}^{-1}$

Sparse Preconditioners: M for $M^{-1}A$

Incomplete LU (ILU) computes an approximate LU factorization, ignoring fill entries throughout regular sparse LU

- Let $S \in \mathbb{N}^2$ to be a sparsity mask and compute $\boldsymbol{L}, \boldsymbol{U}$ on S
- Level-0 ILU factorization

$$\texttt{ILU[0]}: \quad (i,j) \in S \quad \text{if and only if} \quad a_{ij} \neq 0$$

- Given $[\boldsymbol{L}^{(0)}, \boldsymbol{U}^{(0)}] \leftarrow \texttt{ILU[0]}(\boldsymbol{A})$, our preconditioner will be $\boldsymbol{M} = \boldsymbol{L}^{(0)}\boldsymbol{U}^{(0)} \approx \boldsymbol{A}$
- Level-1 ILU factorization

$$\texttt{ILU[1]}: \quad (i,j) \in S \quad \text{if} \quad \exists k, l_{ik}^{(0)} u_{kj}^{(0)} \neq 0 \text{ or } a_{ij} \neq 0$$

(generate fill only if updates are from non-newly-filled entries)

Parallel Incomplete LU

- When the number of nonzeros per row is small, computing `ILU[0]` is as hard as triangular solves with the factors
- Elimination tree is given by spanning tree of original graph,

$$\text{filled graph} = \text{original graph}$$

- No need for symbolic factorization and lesser memory-usage
- However, no general accuracy guarantees
- ILU can be approximated iteratively with high concurrency (see Chow and Patel, 2015)

References – Iterative Methods

- R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994
- A. Greenbaum, *Iterative Methods for Solving Linear Systems*, SIAM, 1997
- Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, 2003
- H. A. van der Vorst, *Iterative Krylov Methods for Large Linear Systems*, Cambridge University Press, 2003

References – Parallel Iterative Methods

- J. W. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2:111-197, 1993
- J. J. Dongarra, I. S. Duff, D. C. Sorenson, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, SIAM, 1998
- I. S. Duff and H. A. van der Vorst, Developments and trends in the parallel solution of linear systems, *Parallel Computing* 25:1931-1970, 1999
- M. T. Jones and P. E. Plassmann, The efficient parallel iterative solution of large sparse linear systems, A. George, J. R. Gilbert, and J. Liu, eds., *Graph Theory and Sparse Matrix Computation*, Springer-Verlag, 1993, pp. 229-245
- H. A. van der Vorst and T. F. Chan, Linear system solvers: sparse iterative methods, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., *Parallel Numerical Algorithms*, pp. 91-118, Kluwer, 1997

References – Preconditioning

- T. F. Chan and H. A. van der Vorst, Approximate and incomplete factorizations, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., *Parallel Numerical Algorithms*, pp. 167-202, Kluwer, 1997
- M. J. Grote and T. Huckle, Parallel preconditioning with sparse approximate inverses, *SIAM J. Sci. Comput.* 18:838-853, 1997
- Y. Saad, Highly parallel preconditioners for general sparse matrices, G. Golub, A. Greenbaum, and M. Luskin, eds., *Recent Advances in Iterative Methods*, pp. 165-199, Springer-Verlag, 1994
- H. A. van der Vorst, High performance preconditioning, *SIAM J. Sci. Stat. Comput.* 10:1174-1185, 1989
- E. Chow and A. Patel, Fine-grained parallel incomplete LU factorization, *SIAM journal on Scientific Computing*, 37(2), pp.C169-C193, 2015.

References – Graph Partitioning

- B. Hendrickson and T. Kolda, Graph partitioning models for parallel computing, *Parallel Computing*, 26:1519-1534, 2000.
- G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.* 20:359-392, 1999
- G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis, Automatic mesh partitioning, A. George, J. R. Gilbert, and J. Liu, eds., *Graph Theory and Sparse Matrix Computation*, Springer-Verlag, 1993, pp. 57-84
- A. Pothen, Graph partitioning algorithms with applications to scientific computing, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., *Parallel Numerical Algorithms*, pp. 323-368, Kluwer, 1997
- C. Walshaw and M. Cross, Parallel optimisation algorithms for multilevel mesh partitioning, *Parallel Comput.* 26:1635-1660, 2000

References – Chaotic Relaxation

- R. Barlow and D. Evans, Synchronous and asynchronous iterative parallel algorithms for linear systems, *Comput. J.* 25:56-60, 1982
- R. Bru, L. Elsner, and M. Newmann, Models of parallel chaotic iteration methods, *Linear Algebra Appl.* 103:175-192, 1988
- G. M. Baudet, Asynchronous iterative methods for multiprocessors, *J. ACM* 25:226-244, 1978
- D. Chazan and W. L. Miranker, Chaotic relaxation, *Linear Algebra Appl.* 2:199-222, 1969
- A. Frommer and D. B. Szyld, On asynchronous iterations, *J. Comput. Appl. Math.* 123:201-216, 2000