

AA220/CME342/CS238

Parallel Methods in Numerical Analysis

Matrix Computation: Iterative Methods I

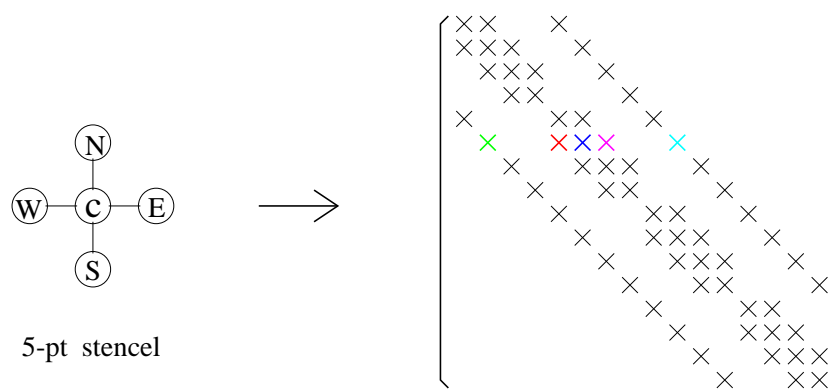
Outline:

- Jacobi, Gauss-Seidel, SOR.
- **Domain partition** (vs matrix partition) computations, & **Multicoloring** technique.
- CG, GMRES, BiCG.
- Parallel Sparse Matrix Algebra.
- Preconditioning.

Source of materials: *Numerical Linear Alg, Introduction to Parallel Computing* by Kumar et al., Demmel's CS267 course.

Sparse Matrices + Direct Methods

The discretization matrix of the heat equation solved by an implicit method on a 2D mesh:

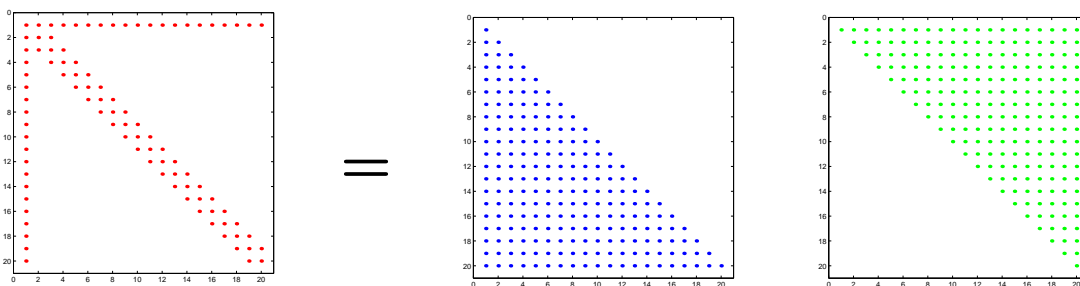


Direct Methods:

- Complexity = $O(n^3)$ \Rightarrow serious computing power challenges when $n \rightarrow \infty$.
- Storage = $O(n^2)$ \Rightarrow Not feasible for large problems.

Sparse Matrices + Direct Methods (cont.)

- Even start with a sparse matrix, the L, U factors can be dense \rightarrow Fill-in occurs during LU factorization:



Observation: Most of the elements are zero \rightarrow no need to store them nor perform any calculation.

Iterative Methods

- Only operate on nonzero elements.
- Typically, max number of nonzeros per row is bounded. Thus,
 - ▷ Work \propto # of unknowns per iteration step.
 - ▷ Storage = $O(n)$.
- No fill-in occurred in iterative methods.
- However, convergence may not be guaranteed.
- Classical relaxation methods: Jacobi, Gauss-Seidel, SOR.
- Krylov subspace methods: Conjugate gradient, BiCG, GMRES, ...
- Krylov subspace methods + Preconditioning \Rightarrow Fast solution methods.

Jacobi Method

- Start with an approximate solution x^k . Then, in general, the residual vector $r^k \neq 0$:

$$r^k \equiv b - Ax^k \neq 0.$$

- Considering the i th component, improve $x_i^k \rightarrow x_i^{k+1}$ by forcing $r_i^k = 0$, i.e.

$$b_i - (a_{ii}x_i^{k+1} + \sum_{i \neq j} a_{ij}x_j^k) = 0.$$

After rearranging terms,

$$x_i^{k+1} = \frac{b_i - \sum_{i \neq j} a_{ij}x_j^k}{a_{ii}}.$$

Repeat for all $i = 1, \dots, n$.

- Note: Jacobi method requires nonzero diagonal entries.

Jacobi Method (cont.)

- Algorithm:

for $i = 1:n$,

$$x_i^{k+1} = (b_i - \sum_{i \neq j} a_{ij} x_j^k) / a_{ii};$$

end;

- **Matrix form:** Write $A = D - L - U$, where
 $D =$ diagonal of A
 $L =$ strict lower triangular part of $(-A)$
 $U =$ strict upper triangular part of $(-A)$

Jacobi iteration:

$$x^{k+1} = x^k + D^{-1}(b - Ax^k).$$

- Thus, Jacobi iteration essentially consists of (sparse) matrix-vector multiplications.
- Convergence: Jacobi converges for diagonal dominant matrices:

$$|a_{ii}| > \sum_{j \neq i}^n |a_{ij}|.$$

Parallel Jacobi Method

- Parallelization of Jacobi method is straightforward:

```
for  $i = localstart : localend$ ,  
     $x_i^{k+1} = (b_i - \sum_{i \neq j} a_{ij} x_j^k) / a_{ii}$ ;  
end;
```

- **Key observations:**

- ▷ x_i^{k+1} is updated using previous $\{x_j^k\}$ only.
- ▷ Update is independent of ordering.

⇒ Can be done in parallel!

- Consider the matrix form:

$$x^{k+1} = x^k + D^{-1}(b - Ax^k).$$

Need to perform parallel (sparse) matrix-vector product. (Will discuss later.)

Parallel Jacobi Method (cont.)

Suppose A is a *general sparse matrix*.

- Distribute A and b by row partitions, and an **entire** x^k among processors (since each row i can have nonzero a_{ij} in any columns):

$$\begin{array}{c}
 \boxed{} \\
 \boxed{x_i^{k+1} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^k}{a_{ii}}} \\
 \boxed{} \\
 \boxed{}
 \end{array}
 \quad
 \begin{array}{c}
 \boxed{} \\
 \boxed{} \\
 \boxed{} \\
 \boxed{}
 \end{array}
 =
 \begin{array}{c}
 \boxed{} \\
 \boxed{} \\
 \boxed{} \\
 \boxed{}
 \end{array}$$

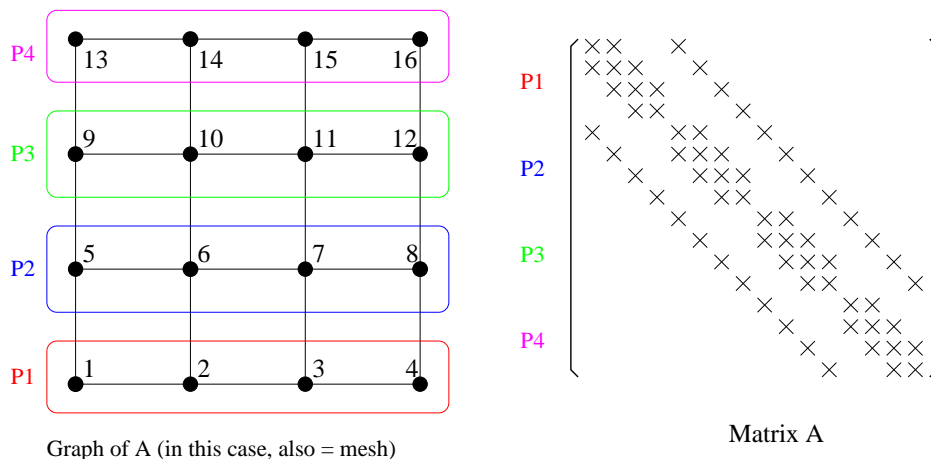
A
 x^k
 b

- Each processor updates its local x_i^{k+1} *without* any communication.
- At the end, each processor broadcasts its x_i^{k+1} 's to one another (MPI Allgather operation).

Parallel Jacobi Method (cont.)

Suppose A is a *planar graph*.

E.g. Discretization of a PDE on a 2D mesh.



- Partition the planar graph associated with A by blocks \Rightarrow Distribute A , b and x^k by block of rows.
- Each processor update its local $\{x_i^{k+1}\}$ with comm. to its *neighboring* procs only.
- Thus, the amount of comm. depends only on the number of boundary nodes. No global communication (Allgather) is needed.

Gauss-Seidel Method

- Start with x^k . Update x_i^{k+1} using the most recent values of x_j , $j \neq i$:

$$x_i^{k+1} = \frac{b_i - \sum_{j < i} a_{ij} x_j^{k+1} - \sum_{j > i} a_{ij} x_j^k}{a_{ii}}.$$

- Note: Gauss-Seidel method also requires nonzero diagonal entries.
- **Matrix form:**

$$x^{k+1} = x^k + (D - L)^{-1}(b - Ax^k).$$

- **Convergence:**
 - ▷ Gauss-Seidel converges for diagonal dominant matrices and symmetric positive definite matrices.
 - ▷ If both Jacobi and GS convergence, GS typically converges **twice as fast as** Jacobi.

SOR Methods

- Weighted average of x^k and x_{GS}^k :

$$x^{k+1} = (1 - \omega) x^k + \omega x_{GS}^k$$

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

- Pre-selected relaxation parameter ω chosen to accelerate convergence:

$$\omega \begin{cases} < 1 & \text{under-relaxation} \\ = 1 & \text{GS} \\ > 1 & \text{over-relaxation} \end{cases}$$

- With optimal value of ω (usually > 1), the convergence rate of SOR can be an order of magnitude faster than GS and Jacobi.

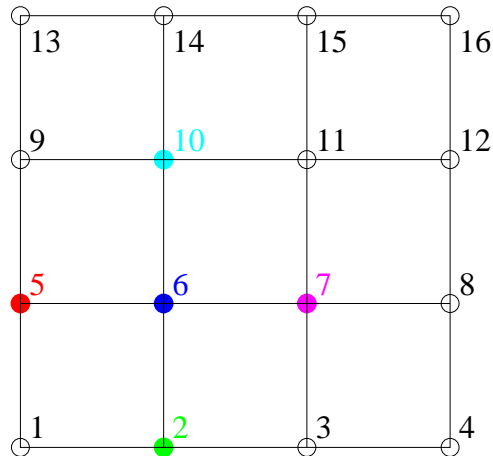
Parallel GS (or SOR)

- Parallelization turns out nontrivial.
- Consider the component equation:

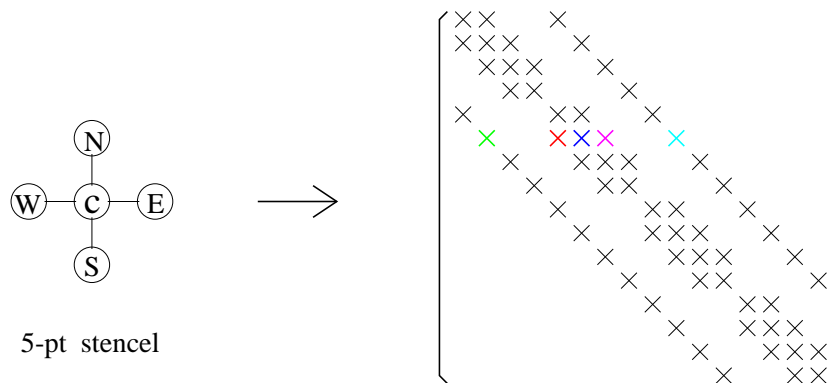
$$x_i^{k+1} = \frac{b_i - \sum_{j < i} a_{ij} x_j^{k+1} - \sum_{j > i} a_{ij} x_j^k}{a_{ii}}.$$

- ▷ Require the updated values x_j^{k+1} , $j < i$, which are not available if the processor does not own them.
- ▷ Every processors have to wait for the updated values from another processors
→ sequential bottleneck.
- **Key observation:**
If $a_{ij} = 0$ for $j < i$, then no communication nor waiting is needed → same scenario as Jacobi method.
- Use multi-coloring technique to order the unknowns such that $a_{ij} = 0$, $j < i$.

2D Mesh: Natural Ordering

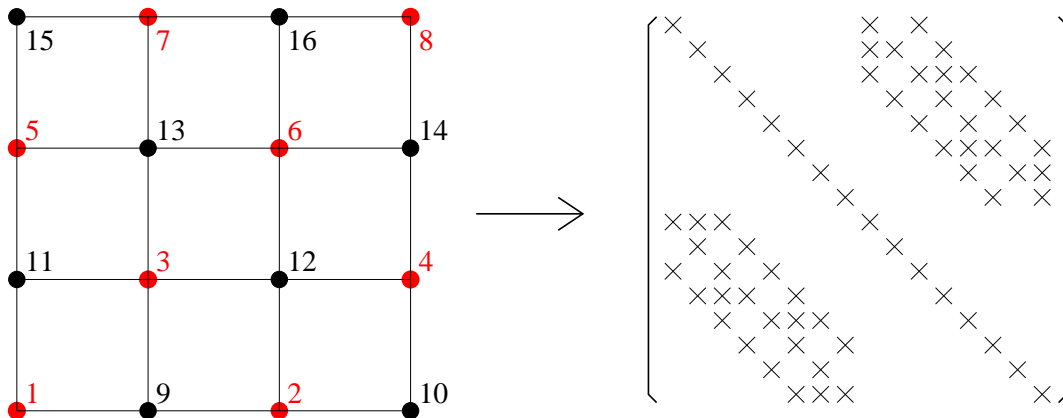


- 5-pt stencil discretization of the Laplacian operator results in a matrix of the form:



- For each node C , there are, in general, 4 neighbors (E, S, W, N). Using natural ordering, the update at C requires the updated values at S and W → sequential bottleneck.

2D Mesh: Red-Black Ordering



- In RB ordering, the update of red nodes (e.g. 3) depends only on black nodes; thus can be done independently in parallel. Afterwards, broadcast the values to their neighbors, and then update black nodes in parallel.
- Parallelization can be realized from the 2×2 block structure in the RB ordered matrix where the diagonal blocks are diagonal matrices.

Parallel Implementation

- Distribute the red nodes equally among p procs; same for the black nodes.
- Algorithm:

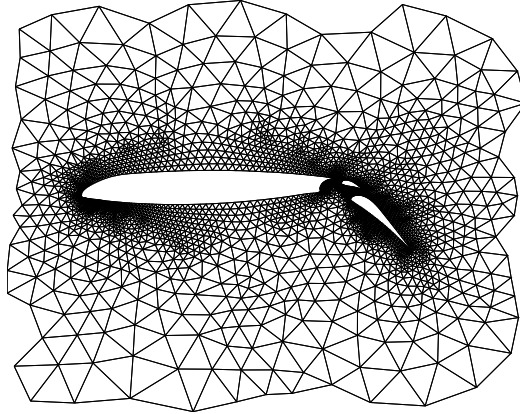
for each color={red, black}

All procs update its local x_i^{k+1} simultaneously:

$$x_i^{k+1} = \frac{b_i - \sum_{j < i} a_{ij} x_j^{k+1} - \sum_{j > i} a_{ij} x_j^k}{a_{ii}};$$

end;

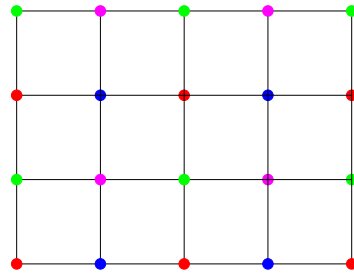
General Mesh: Multicoloring



- For general matrix graph, use multicoloring.
- Nodes for each color can be updated simultaneously, i.e. in parallel.
- The fewer the # of colors, the more parallel the algorithm is.
 - ▷ RB Gauss-Seidel has 2 colors \rightarrow parallel.
 - ▷ Gauss-Seidel has n colors \rightarrow sequential.
- Convergence: The fewer the # of colors, the slower the convergence is.
- Hence tradeoff between parallelism & efficiency.

Multicoloring

- Coloring of the 9-pt stencil graph:



- To color a general graph with min. # of color (for max. parallel efficiency) is NP-hard.
- There are heuristics to color most graphs arising from applications using a small # of colors.
- A sequential algorithm:
(color={0, 1, 2, ...})

```
V = set of vertices;  
for i = 1 to n do  
  Choose vertices  $v_i \in V$  according to an ordering  
  algorithm;  
  Choose the smallest possible color for  $v_i$ ;  
   $V = V \setminus \{v_i\}$ ;  
end
```

Multicoloring (cont.)

- Examples of ordering algorithm:
LFO: order vertices in V such that

$$\deg(v_1) \geq \deg(v_2) \geq \dots \geq \deg(v_n).$$

IDO: v_i is chosen with max incidence degree, i.e. max # of adjacent colored vertices.

- A parallel coloring algorithm:
Given a good partitioning & good assignment of partitions to processors (each process has about the same # of vertices; minimal boundary edges)
 1. Color the global boundary vertices.
 2. Color the local vertices independently on each processor by a sequential algorithm.

Jacobi vs RB Gauss-Seidel

- RB GS converges twice as fast as Jacobi, but requires twice as many parallel steps; about the same run time in practice.
- Parallel efficiency **alone** is not sufficient to determine overall performance.
- We also need fast converging algorithms.

Run Time Complexity: Jacobi

- Assume on a 2D mesh.

- Notations:

$N = \#$ of unknowns

$p = \#$ of processors

$f =$ time per flop

$\alpha =$ startup for a message

$\beta =$ time per word in a message

Time = $\#$ of steps \times cost per step

- Sequential run time:

$$\text{Time(Jacobi)} = O(N) \times O(N) = O(N^2).$$

- Parallel run time:

$$\begin{aligned}\text{Time(Jacobi)} &= O(N) \times [(N/p)f + \alpha + (\sqrt{N}/p)\beta] \\ &= O(N^2/p) f + O(N) \alpha + O(N^{3/2}/p) \beta\end{aligned}$$

Note: $O(N/p)$ flops to update local values, α for the start up of message passing, $O(\sqrt{N}/p)$ boundary nodes communicated to neighbors.

Parallel Run Time: Comparison

Notations:

$N = \#$ of unknowns

$p = \#$ of processors

$f =$ time per flop

$\alpha =$ startup for a message

$\beta =$ time per word in a message

Time = $\#$ of steps \times cost per step

Methods	Parallel run time
Jacobi	$O(N) \times [O(\frac{N}{p})f + \alpha + O(\frac{\sqrt{N}}{p})\beta]$
RB GS	$O(N) \times [O(\frac{N}{p})f + \alpha + O(\frac{\sqrt{N}}{p})\beta]$
RB SOR	$O(\sqrt{N}) \times [O(\frac{N}{p})f + \alpha + O(\frac{\sqrt{N}}{p})\beta]$

Block Methods

- Block Jacobi:

$$x^{k+1} = x^k + \tilde{D}^{-1}(b - Ax^k),$$

\tilde{D} =block diagonal of A .

- E.g. Given $p \times p$ mesh, partition 5-pt stencil matrix A into p block rows where each proc has one line of variables. Then \tilde{D}_j =tridiagonal matrix.
- Suppose subvector x_j^k is the j th portion of vector x^k possessed by proc p_j .

Parallel Algorithm:

for each proc p_j ,

$$x_j^{k+1} = x_j^k + \tilde{D}_j^{-1}(b - Ax^k)_j;$$

end;

Block Methods (cont.)

- A variant of block Jacobi method is to substitute the inversion of \tilde{D}_j by one iteration of GS:

for each proc p_j ,

for $i=\text{localstart}$ to localend ,

$$x_i^{k+1} = (b_i - \sum_{j < i \text{ \& } j = \text{local}} a_{ij} x_j^{k+1} - \sum_{j > i \text{ \textit{or} } j = \text{nonlocal}} a_{ij} x_j^k) / a_{ii};$$

end;

end;

- Similar to GS except that only use the updated x_i^{k+1} which are local.