Parallel Iterative Matrix Algorithms

CS121 Parallel Computing

Spring 2021

Iterative matrix algorithms

- To solve a dense system of linear equations $Ax = b$, we saw direct methods such as Gaussian Elimination.
- When the A is very large (millions of variables), GE is too slow.
- For structured matrices, e.g. banded matrices, special solution methods can be developed.
- For general sparse matrices, we use iterative algorithms that compute approximate solutions which eventually converge to the true solution.

Iterative matrix algorithms

- Given $A \in \mathbb{R}^{n \times n}$, write $A = M - N$, where M and N are matrices such that M^{-1} is easy to compute (e.g. a diagonal matrix).
  - Let x^* be the solution to $Ax = b$. Then $Mx^* = Nx^* + b$.
- Let $C = M^{-1}N$, $d = M^{-1}b$.
  - So $x^* = Cx^* + d$.
- Starting from an initial x, repeatedly compute $Cx + d$.
  - Denote k'th iterate of x as $x^{(k)}$.
  - Then $x^{(k+1)} = Cx^{(k)} + d$.

Convergence criteria

- We want the iterations to converge, starting from any initial vector $x^{(0)} \in \mathbb{R}^n$.
  - I.e. we want $\lim_{k\to\infty} x^{(k)} = x^*$, so that $x^* = Cx^* + d$.
- Since $x^{(k+1)} = Cx^{(k)} + d$, then subtracting, we get $x^{(k+1)} - x^* = C(x^{(k)} - x^*)$.
  - Also, $x^{(k)} - x^* = C(x^{(k-1)} - x^*)$, etc.
  - So in general $x^{(k)} - x^* = C^k\left(x^{(0)} - x^*\right)$.
- Let $\rho(C)$ be the magnitude of the largest eigenvalue of C.
- Thm The following are equivalent
  - The iterative algorithm converges for any initial $x^{(0)}$.
  - $\lim_{k\to\infty} C^k = 0$.
  - $\rho(C) < 1$.

Jacobi method

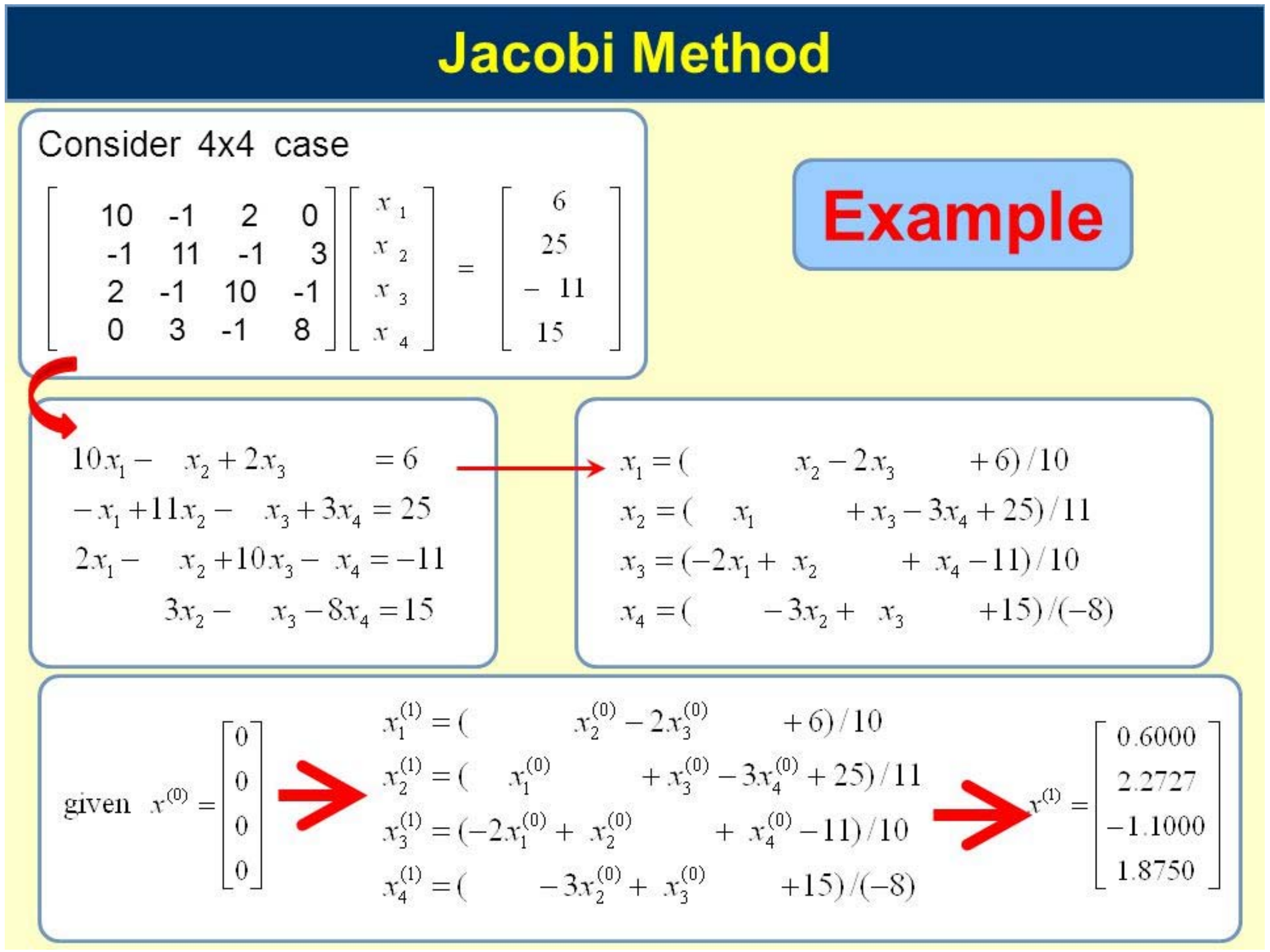
- Write $A = D - L - R$, where D is the diagonal elements of A, $-L$ is the lower triangular part of A without D, and $-R$ is the upper triangular part without D.
- Let $M = D, N = L + R$. Note that M is easy to invert.
- Then $C = D^{-1}(L + R)$, and $d = D^{-1}b$.
  - So $c_{ij} = -a_{ij}/a_{ii}$ if $j \neq i$, and $c_{ii} = 0$ for all i.
  - Also, $d_i = b_i/a_{ii}$ for all i.
- Convergence is guaranteed if the matrix is diagonally dominant, i.e. $|a_{ii}| > \sum_{j=1,j\neq i}^{n} |a_{ij}|$ for all i.
- Recall $x^{(k+1)} = Cx^{(k)} + d$. So the i'th component of $x^{(k+1)}$ is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,j\neq i}^{n} a_{ij}x_j^{(k)}\right).$$

- $x^{(k+1)}$ depends only on $x^{(k)}$, and different components of $x^{(k+1)}$ do not have any dependencies.
  - Thus, all components of $x^{(k+1)}$ can be computed in parallel.

Jacobi method example

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}\right)$$



Jacobi Method

Consider 4x4 case

$$\begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 6 \\ 25 \\ -11 \\ 15 \end{bmatrix}$$

Example

$$\begin{aligned} 10x_1 - x_2 + 2x_3 &= 6 \\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25 \\ 2x_1 - x_2 + 10x_3 - x_4 &= -11 \\ 3x_2 - x_3 - 8x_4 &= 15 \end{aligned} \longrightarrow \begin{aligned} x_1 &= (x_2 - 2x_3 + 6)/10 \\ x_2 &= (x_1 + x_3 - 3x_4 + 25)/11 \\ x_3 &= (-2x_1 + x_2 + x_4 - 11)/10 \\ x_4 &= (-3x_2 + x_3 + 15)/(-8) \end{aligned}$$

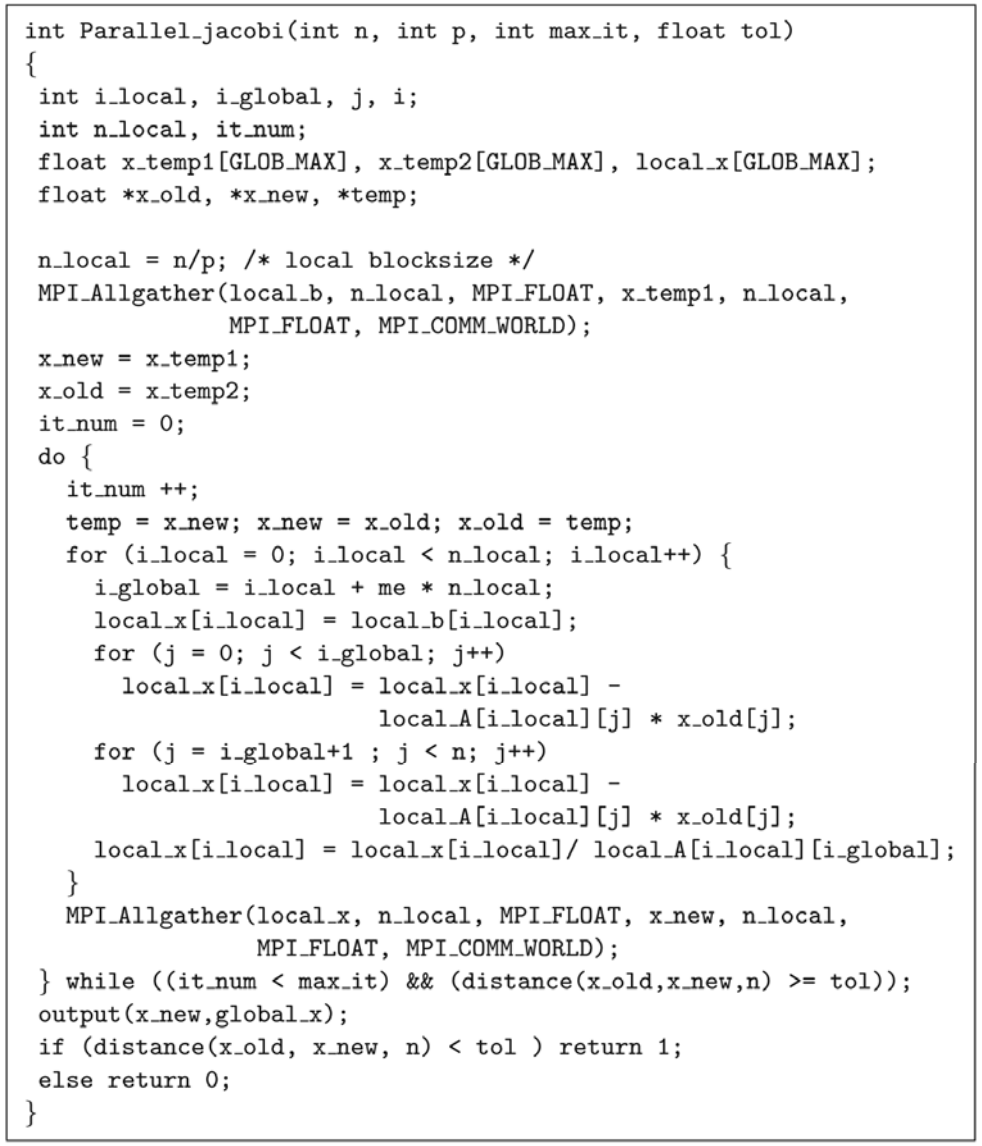
$$\text{given } x^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{aligned} x_1^{(1)} &= (x_2^{(0)} - 2x_3^{(0)} + 6)/10 \\ x_2^{(1)} &= (x_1^{(0)} + x_3^{(0)} - 3x_4^{(0)} + 25)/11 \\ x_3^{(1)} &= (-2x_1^{(0)} + x_2^{(0)} + x_4^{(0)} - 11)/10 \\ x_4^{(1)} &= (-3x_2^{(0)} + x_3^{(0)} + 15)/(-8) \end{aligned} \rightarrow x^{(1)} = \begin{bmatrix} 0.6000 \\ 2.2727 \\ -1.1000 \\ 1.8750 \end{bmatrix}$$

Source: https://slideplayer.com/slide/3276935/

Parallel Jacobi method

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1,j\neq i}^{n} a_{ij}x_j^{(k)}\right)$$

- Since all components of $x^{(k+1)}$ are independent, we can use up to n processors.
- In distributed memory, matrix A and vector b are stored in row-wise block format across the processors.
- $x^{(k+1)}$ and $x^{(k)}$ are computed in `x_new` and `x_old`, resp.
- Each processor needs all the values of x.
- After all processors compute their part of `x_new`, the whole vector is distributed to all processors using `MPI_Allgather`.



```
int Parallel_jacobi(int n, int p, int max_it, float tol)
{
 int i_local, i_global, j, i;
 int n_local, it_num;
 float x_temp1[GLOB_MAX], x_temp2[GLOB_MAX], local_x[GLOB_MAX];
 float *x_old, *x_new, *temp;

 n_local = n/p; /* local blocksize */
 MPI_Allgather(local_b, n_local, MPI_FLOAT, x_temp1, n_local,
              MPI_FLOAT, MPI_COMM_WORLD);
 x_new = x_temp1;
 x_old = x_temp2;
 it_num = 0;
 do {
   it_num ++;
   temp = x_new; x_new = x_old; x_old = temp;
   for (i_local = 0; i_local < n_local; i_local++) {
     i_global = i_local + me * n_local;
     local_x[i_local] = local_b[i_local];
     for (j = 0; j < i_global; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     for (j = i_global+1 ; j < n; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     local_x[i_local] = local_x[i_local]/ local_A[i_local][i_global];
   }
   MPI_Allgather(local_x, n_local, MPI_FLOAT, x_new, n_local,
                MPI_FLOAT, MPI_COMM_WORLD);
 } while ((it_num < max_it) && (distance(x_old,x_new,n) >= tol));
 output(x_new,global_x);
 if (distance(x_old, x_new, n) < tol ) return 1;
 else return 0;
}
```

Source: Parallel Programming for Multicore and Cluster Systems, Rauber and Runger

Gauss-Seidel method

- The Gauss-Seidel method usually converges faster than the Jacobi method.
- Again write $A = D - L - R$, but set $M = D - L$ and $N = R$.
  - Thus, we have $C = (D - L)^{-1}R$.
  - Since $D - L$ is lower triangular, it can be inverted by forward substitution.
- Convergence is again guaranteed if the matrix is diagonally dominant, i.e. $|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$ for all i.
- As before, $x^{(k+1)} = Cx^{(k)} + d$. So in components form we have
  $x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right).$
- Unlike Jacobi method, $x_i^{(k+1)}$ depends on all $x_j^{(k+1)}$ for $j < i$ and $a_{ij} \neq 0$.
  - Thus, unless many values of $a_{ij} = 0$, different $x_i^{(k+1)}$ cannot be computed in parallel.
- While Gauss-Seidel converges faster than Jacobi, it has less parallelism, and may not run faster.

SOR method

- Successive over-relaxation modifies the Gauss-Seidel method to obtain faster convergence.
- x is updated as a linear combination of Gauss-Seidel update and its previous value.
- In components form,

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) + (1-\omega)x_i^{(k)}.$$

- Convergence depends on properties of A and ω. E.g. if A is symmetric and positive definite and $\omega \in (0,2)$ then SOR converges.

Parallel Gauss-Seidel method

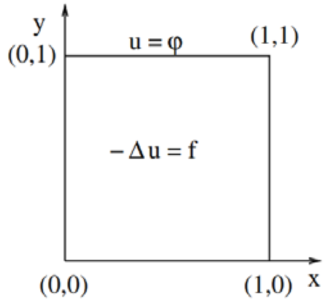
- Since in general $x_i^{(k+1)}$ depends on $x_j^{(k+1)}$ for all j<i, we compute the $x_i^{(k+1)}$ sequentially, for i=0,1,2...
- Each $x_i^{(k+1)}$ is a dot product of $\left(x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}, 0, x_{i+1}^{(k)}, x_n^{(k)}\right)$ with the i'th row of A.
  - This dot product can be split into multiple parts and computed in parallel.
- Use block column-wise decomposition of A and x across the processors.
- Each processor computes part of $x_i^{(k+1)}$.
  - The parts are then summed and distributed to all the processors using `MPI_Allreduce`.
- Repeat the `do` loop until x converges.
- Each processor only does n/p computations for each reduce communication step, so speedup is limited unless $n \gg p$.

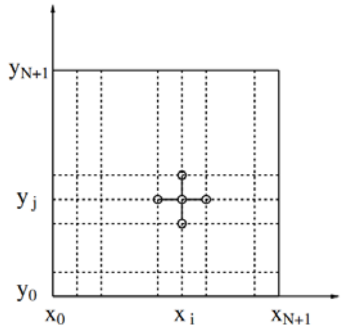
$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

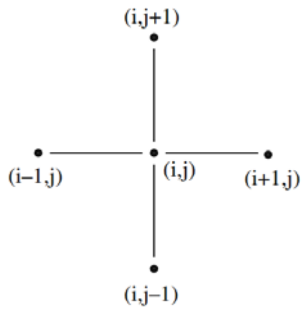
```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

Poisson’s equation

- Poisson’s equation is a partial differential equation (PDE) to describe the potential field caused by a mass or electrostatic density distribution.
  - We’ll look at Poisson’s equation in 2D space.
- Given a function $f(x,y)$, we want to find a function $\phi(x,y)$ with $-\Delta\phi = f$.
  - Here $\Delta\phi = \nabla^2\phi = \frac{\partial^2\phi}{\partial^2 x} + \frac{\partial^2\phi}{\partial^2 y}$.
- Poisson’s equation can be solved numerically by discretizing 2D space.
  - For simplicity, we divide $[0,1] \times [0,1]$ evenly into N+1 points along each axis.
  - Let $h = 1/(N+1)$, and let $u_{ij} = \phi(x_i, y_j)$, and $f_{ij} = f(x_i, y_j)$.
- Then $\frac{1}{h^2}\left(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}\right) = f_{ij}$, for $0 \leq i,j \leq N+1$.
- For simplicity, fix the value of ϕ on the boundary of the square, and divide out both sides by $1/h^2$.
- Look for the value of ϕ in the square’s interior.
  - This leads to a set of N^2 linear equations, one for each $1 \leq i,j \leq N$.

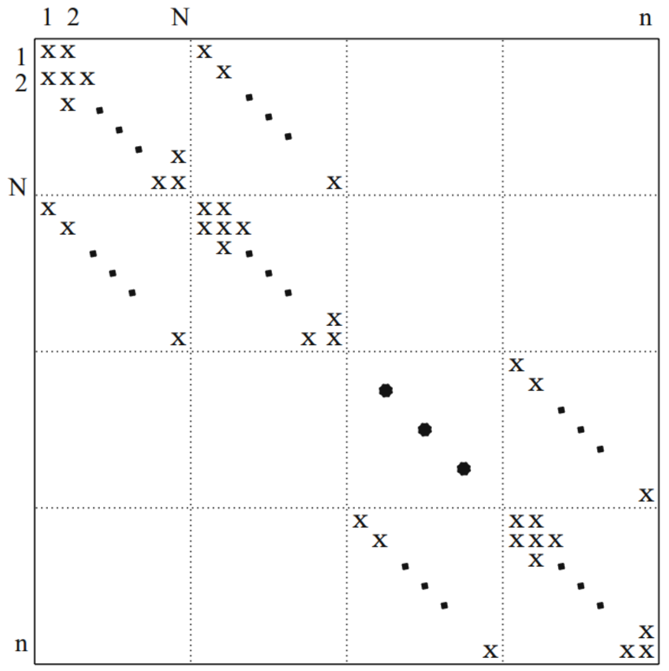






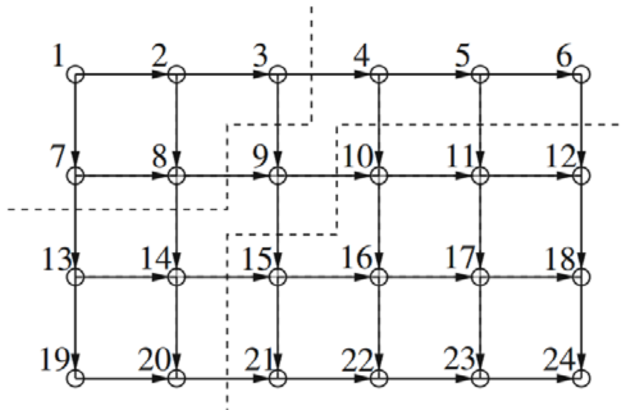
Matrix form of Poisson's equation

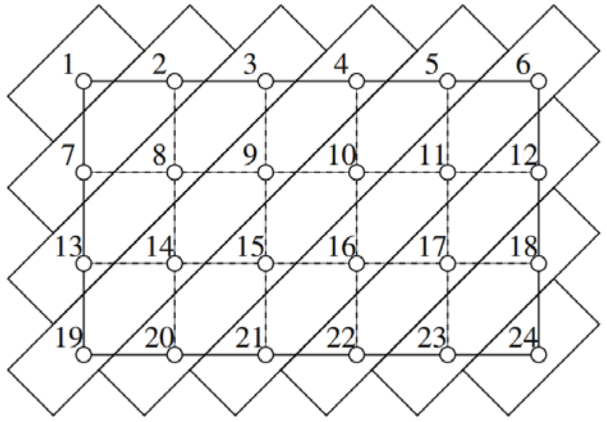
- Let $x_k = u_{ij}$ for $k = i + (j-1)N$, $1 \le i,j \le N$.
- Each equation from the discretization has the form $4x_k - x_{k+1} - x_{k-1} - x_{k+N} - x_{k-N} = b_k$, for some $1 \le k \le N^2$ and b_k.
- Let $n = N^2$, and create an $n \times n$ matrix A for the nonzero coefficients of all the equations.
- A has the following nonzero structure.
  - There are three bands of nonzeros, on the diagonal, and above and below the diagonal.
  - There are two additional bands of nonzeros distance $N = \sqrt{n}$ above and below the diagonal.



Gauss-Seidel for Poisson's equation

- Recall the Gauss-Seidel iteration is $x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$.
- Applied to Poisson's equation, we have $x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - a_{i,i-\sqrt{n}} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i,i-1} \cdot x_{i-1}^{(k+1)} - a_{i,i+1} \cdot x_{i+1}^{(k)} - a_{i,i+\sqrt{n}} \cdot x_{i+\sqrt{n}}^{(k)}\right)$,for $i = 1, \ldots, n$.
- The values $x_{i-\sqrt{n}}^{(k+1)}$ and $x_{i-1}^{(k+1)}$ need to be computed before $x_i^{(k+1)}$.
- Place the x values on the grid in row major order.
- Each x value depends on value directly above and to its left.
  - Ex Point 9 depends on 3 and 8.
- Notice the x values along each diagonal are all independent.
- There are $2\sqrt{n} - 1$ diagonals.
  - Each diagonal has $O(\sqrt{n})$ points, giving a large amount of parallelism.
- The first $\sqrt{n}$ diagonals $l = 1, \ldots, \sqrt{n}$ each contain l points, with indices $i = l + j(\sqrt{n} - 1)$, for $0 \le j \le l$.
- The last $\sqrt{n} - 1$ diagonals $l = 2, \ldots, \sqrt{n}$ contain $\sqrt{n} - l + 1$ points, with indices $i = l\sqrt{n} + j(\sqrt{n} - 1)$, for $0 \le j \le \sqrt{n} - l$.





Gauss-Seidel for Poisson's equation

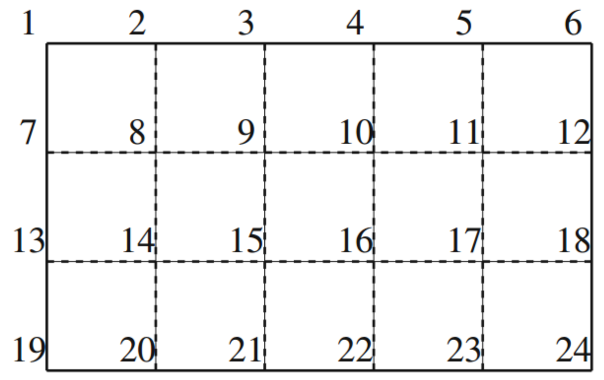
- We parallelize Gauss-Seidel for Poisson's equation by iterating through the diagonals sequentially, and computing all the values in each diagonal in parallel.
- Given p processors, each processor computes every p'th value `x[i]` on the `l`'th diagonal.
- Notice the sizes of the diagonals first increases, then decreases.
  - The two `for` loops compute x during the increasing and decreasing phase, resp.
- The function `collect_elements` sends the x values from the l'th diagonal to neighboring processors to compute the l+1'st diagonal.
- Repeat the `do` loop until x converges.

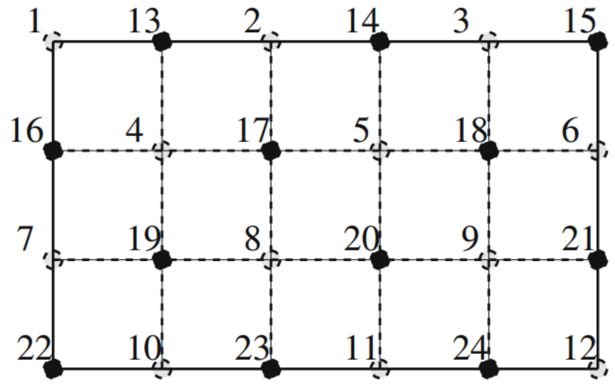
$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - a_{i,i-\sqrt{n}} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i,i-1} \cdot x_{i-1}^{(k+1)} - a_{i,i+1} \cdot x_{i+1}^{(k)} - a_{i,i+\sqrt{n}} \cdot x_{i+\sqrt{n}}^{(k)}$$

```
sqn = sqrt(n);
do {
   for (l = 1; l <= sqn; l++) {
      for (j = me; j < l; j+=p) {
         i = l + j * (sqn-1) - 1; /* start numbering with 0 */
         x[i] = 0;
         if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
         if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
         if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
         if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
         x[i] = (x[i] + b[i]) / a[i][i];
      }
      collect_elements(x,l);
   }
   for (l = 2; l <= sqn; l++) {
      for (j = me -l +1; j <= sqn -1; j+=p) {
         if (j >= 0) {
           i = l * sqn + j * (sqn-1) - 1;
           x[i] = 0;
           if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
           if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
           if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
           if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
           x[i] = (x[i] + b[i]) / a[i][i];
         }
      }
      collect_elements(x,l);
   }
} while(convergence_test() < tol);
```

Red-black ordering

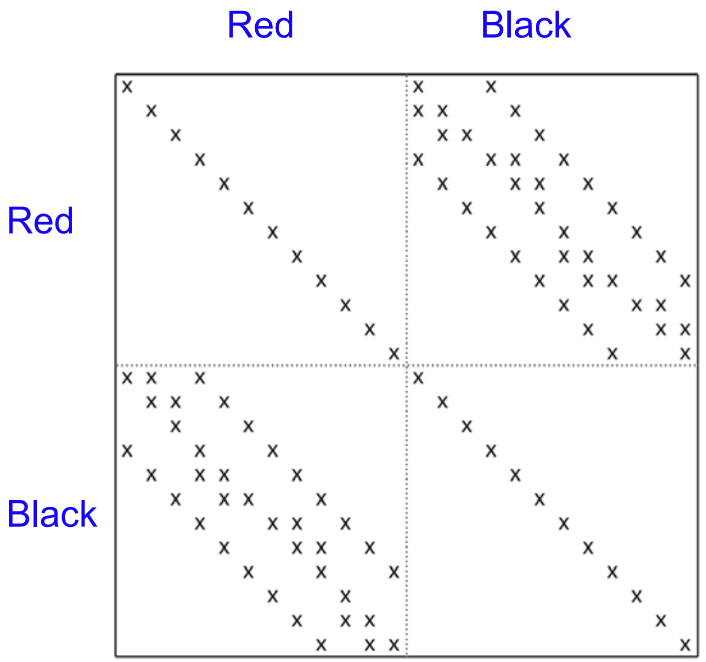
- In Gauss-Seidel, each $x_i^{(k+1)}$ depends on all $x_j^{(k+1)}$ for $j < i$ and $a_{ij} \neq 0$.
- The method is valid for any ordering of the x values, so we can choose an ordering that gives best parallelism.
- The diagonals method used the top ordering. We now consider the bottom red-black ordering.
- Assign each mesh point a color, red or black.
  - For each mesh point (i,j), if i+j is even, it is colored red (grey in the picture on right). Otherwise color it black.
- Since a point only depends on the points above it and to its left, none of the red points depend on each other, and similarly for the black points.
  - So the red and black x values can be computed in parallel.





Red-black ordering

- Matrix A has a different structure after reordering.
- The red points for $x^{(k+1)}$ only depend on the black points for $x^{(k)}$, and the black points for $x^{(k+1)}$ only depend on the red points for $x^{(k+1)}$.
  - Thus, we can compute all $n/2$ red points in $x^{(k+1)}$ in parallel, then compute all the $n/2$ black points in $x^{(k+1)}$ in parallel.
- Write matrix $A = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix}$, where D_R and D_B are diagonal matrices corresponding to the red and black points resp, and E and F are banded matrices.
- Also write $x = (x_R \;\; x_B)$, where x_R, x_B are the set of red and black x values, resp.



Red-black ordering

- Let $D = \begin{pmatrix} D_R & 0 \\ 0 & D_B \end{pmatrix}$, $L = \begin{pmatrix} 0 & 0 \\ -E & 0 \end{pmatrix}$ and $U = \begin{pmatrix} 0 & -F \\ 0 & 0 \end{pmatrix}$.
- Write the Gauss-Seidel iteration as $\begin{pmatrix} D_R & 0 \\ E & D_B \end{pmatrix} \begin{pmatrix} x_R^{(k+1)} \\ x_B^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix}$.
- So $D_R \cdot x_R^{(k+1)} = b_1 - F \cdot x_B^{(k)}$ for $k = 1,2,\dots$, and $D_B \cdot x_B^{(k+1)} = b_2 - E \cdot x_R^{(k+1)}$, for $k = 1,2,\dots$
- Let n_R and n_B be the number of red and black points, resp.
- For each point i, let $N(i)$ be its neighbors in the grid.
- In component form, we have

$$\left(x_R^{(k+1)}\right)_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j \in N(i)} a_{ij} \cdot \left(x_B^{(k)}\right)_j\right) \text{ for } i = 1, \dots, n_R$$

$$\left(x_B^{(k+1)}\right)_i = \frac{1}{a_{i+n_R, i+n_R}}\left(b_{i+n_R} - \sum_{j \in N(i)} a_{i+n_R, j} \cdot \left(x_R^{(k+1)}\right)_j\right) \text{ for } i = 1, \dots, n_B$$

- Thus, we first compute $x^{(k+1)}$ for all the red points, then $x^{(k+1)}$ for all the black points.

Parallel red-black algorithm

- Use a block row-wise decomposition of A and x across the processors.
- Use barrier synchronization between the two loops to compute black values after red ones.
- `collect_elements` sends newly computed values of x that lie on the boundary between two processors to the other processor.

```
local_nr = nr/p; local_nb = nb/p;
do {
   mestartr = me * local_nr;
   for  (i= mestartr; i < mestartr + local_nr; i++) {
       xr[i] = 0;
       for  (j ∈ N(i))
           xr[i] = xr[i] - a[i][j] * xb[j];
       xr[i] = (xr[i]+b[i]) / a[i][i]  ;
   }
   collect_elements(xr);
   mestartb = me * local_nb + nr;
   for (i= mestartb; i < mestartb + local_nb; i++) {
       xb[i] = 0;
       for (j ∈ N(i))
           xb[i] = xb[i] - a[i+nr][j] * xr[j];
       xb[i]= (xb[i] + b[i+nr]) / a[i+nr][i+nr];
   }
   collect_elements(xb);
} while (convergence_test());
```