Iterative solution of linear systems

Victor Eijkhout

Fall 2025

Justification

As an alternative to Gaussian elimination, iterative methods can be an efficient way to solve the linear system from PDEs. We discuss basic iterative methods and the notion of preconditioning.

Two different approaches

Solve $Ax = b$

Direct methods:

- Deterministic
- Exact up to machine precision
- Expensive (in time and space)

Iterative methods:

- Only approximate
- Cheaper in space and (possibly) time
- Convergence not guaranteed

Stationary iteration

Iterative methods

Choose any x_0 and repeat

$$x^{k+1} = Bx^k + c$$

until $\|x^{k+1} - x^k\|_2 < \varepsilon$ or until $\frac{\|x^{k+1} - x^k\|_2}{\|x^k\|} < \varepsilon$

Example of iterative solution

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2,1,1)$.
Suppose you know (physics) that solution components are roughly the same size, and observe the dominant size of the diagonal, then

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

might be a good approximation: solution $(2.1, 9/7, 8/6)$.

Iterative example′

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2, 1, 1)$.
Also easy to solve:

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution $(2.1, 7.95/7, 5.9/6)$.

Abstract presentation

- To solve $Ax = b$; too expensive; suppose $K \approx A$ and solving $Kx = b$ is possible
- Define $Kx_0 = b$, then error correction $x_0 = x + e_0$, and $A(x_0 - e_0) = b$
- so $Ae_0 = Ax_0 - b = r_0$; this is again unsolvable, so
- $K\tilde{e}_0 = r_0$ and $x_1 = x_0 - \tilde{e}_0$.
- In one formula:

$$x_1 = x_0 - K^{-1} r_0$$

- now iterate: $e_1 = x_1 - x$, $Ae_1 = Ax_1 - b = r_1$ et cetera

Iterative scheme:

$$x_{i+1} = x_i - K^{-1} r_i \qquad \text{where } r_i = Ax_i - b$$

Takeaway

Each iteration involves:

- multiplying by A,
- solving with K

Error analysis

- One step

$$
\begin{aligned}
r_1 &= Ax_1 - b = A(x_0 - \tilde{e}_0) - b && (1)\\
&= r_0 - AK^{-1}r_0 && (2)\\
&= (I - AK^{-1})r_0 && (3)
\end{aligned}
$$

- Inductively: $r_n = (I - AK^{-1})^n r_0$ so $r_n \downarrow 0$ if $|\lambda(I - AK^{-1})| < 1$
  Geometric reduction (or amplification!)
- This is 'stationary iteration': no dependence on the iteration number. Simple analysis, limited applicability.

Takeaway

The iteration process does not have a pre-determined number of operations:
depends *spectral properties* of the matrix.

Complexity analysis

- Direct solution is $O(N^3)$
  except sparse, then $O(N^{5/2})$ or so
- Iterative per iteration cost $O(N)$ assuming sparsity.
- Number of iterations is complicated function of spectral properties:
  - Stationary iteration $\#\mathrm{it} = O(N^2)$
  - Other methods $\#\mathrm{it} = O(N)$
    (2nd order only, more for higher order)
  - Multigrid and fast solvers: $\#\mathrm{it} = O(\log N)$ or even $O(1)$

Choice of K

- The closer K is to A, the faster convergence.
- Diagonal and lower triangular choice mentioned above: let

$$A = D_A + L_A + U_A$$

  be a splitting into diagonal, lower triangular, upper triangular part, then
- Jacobi method: $K = D_A$ (diagonal part),
- Gauss-Seidel method: $K = D_A + L_A$ (lower triangle, including diagonal)
- SOR method: $K = \omega D_A + L_A$

Computationally

If

$$A = K - N$$

then

$$Ax = b \Rightarrow Kx = Nx + b \Rightarrow Kx_{i+1} = Nx_i + b$$

Equivalent to the above, and you don't actually need to form the residual.

Jacobi

$$K = D_A$$

Algorithm:

for $k = 1, \ldots$ *until convergence, do:*

for $i = 1 \ldots n$*:*

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

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

Implementation:

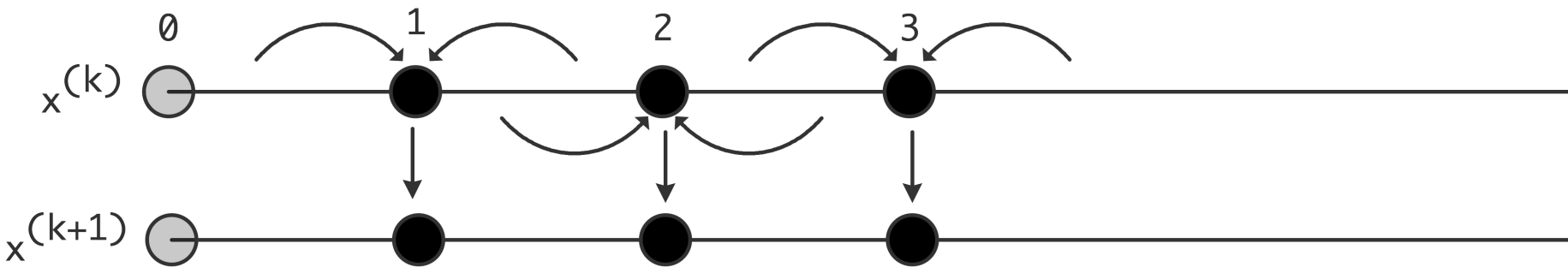
for $k = 1, \ldots$ *until convergence, do:*

for $i = 1 \ldots n$*:*

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

copy $x \leftarrow t$

Jacobi in pictures:



Gauss-Seidel

$$K = D_A + L_A$$

Algorithm:

for $k = 1, \ldots$ *until convergence, do:*

for $i = 1 \ldots n$*:*

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

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

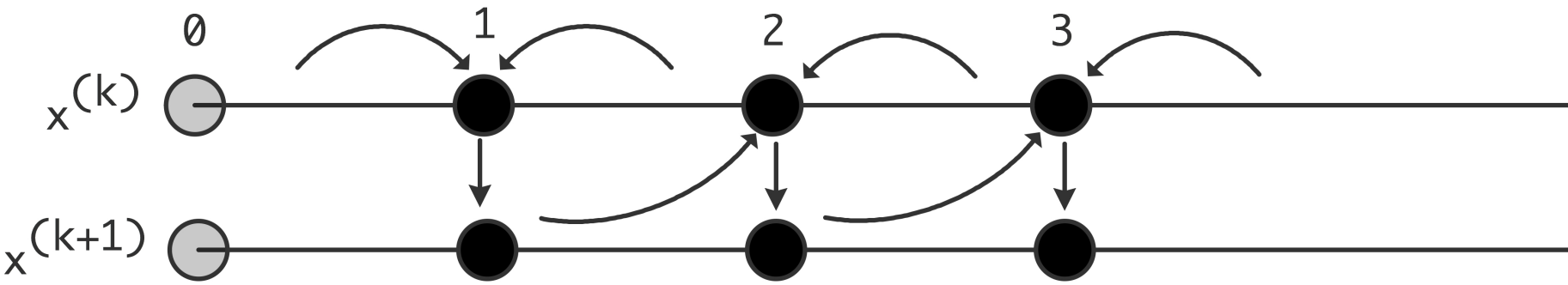
Implementation:

for $k = 1, \ldots$ *until convergence, do:*

for $i = 1 \ldots n$*:*

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

GS in pictures:



Choice of K through incomplete LU

- Inspiration from direct methods: let $K = LU \approx A$

Gauss elimination:

```
for k,i,j:
   a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

Incomplete variant:

```
for k,i,j:
  if a[i,j] not zero:
    a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

$\Rightarrow$ sparsity of $L + U$ the same as of A

Applicability

Incomplete factorizations mostly work for M-matrices:

2nd order FDM and FEM

Can be severe headache for higher order

Stopping tests

When to stop converging? Can size of the error be guaranteed?

- Direct tests on error $e_n = x - x_n$ impossible; two choices
- Relative change in the computed solution small:

$$\|x_{n+1} - x_n\| / \|x_n\| < \varepsilon$$

- Residual small enough:

$$\|r_n\| = \|Ax_n - b\| < \varepsilon$$

Without proof: both imply that the error is less than some other ε'.

Polynomial iterative methods

Derivation by hand-waving

- Remember iteration:

$$x_1 = x_0 - K^{-1} r_0, \qquad r_0 = Ax_0 - b$$

and conclusion

$$r_n = (I - AK^{-1})^n r_0$$

- Abstract relation between true solution and approximation:

$$x_{\text{true}} = x_{\text{initial}} + K^{-1} \pi(AK^{-1}) r_{\text{initial}}$$

- Cayley-Hamilton theorem implies

$$(K^{-1}A)^{-1} = -\pi(K^{-1}A)$$

- inspires us to scheme:

$$x_{i+1} = x_0 + K^{-1} \pi^{(i)}(AK^{-1}) r_0$$

Sequence of polynomials of increasing degree

Residuals

$$x_{i+1} = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0$$

Multiply by A and subtract b:

$$r_{i+1} = r_0 + \tilde{\pi}^{(i)}(AK^{-1})r_0$$

So:

$$r_i = \hat{\pi}^{(i)}(AK^{-1})r_0$$

where $\hat{\pi}^{(i)}$ is a polynomial of degree i with $\hat{\pi}^{(i)}(0) = 1$.
$\Rightarrow$ convergence theory

Computational form

$$x_{i+1} = x_0 + \sum_{j \leq i} K^{-1} r_j \alpha_{ji}.$$

or equivalently:

$$x_{i+1} = x_i + \sum_{j \leq i} K^{-1} r_j \alpha_{ji}.$$

or:

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1} r_i + \sum_{j \leq i} x_j \gamma_{ji} \\ r_{i+1}\gamma_{i+1,i} = AK^{-1} r_i + \sum_{j \leq i} r_j \gamma_{ji} \end{cases}$$

Takeaway

Each iteration involves:

- multiplying by A,
- solving with K

Orthogonality

Idea one:

If you can make all your residuals orthogonal to each other, and the matrix is of dimension n, then after n iterations you have to have converged: it is not possible to have an $n+1$-st residuals that is orthogonal and nonzero.

Idea two:

The sequence of residuals spans a series of subspaces of increasing dimension, and by orthogonalizing the initial residual is projected on these spaces. This means that the errors will have decreasing sizes.

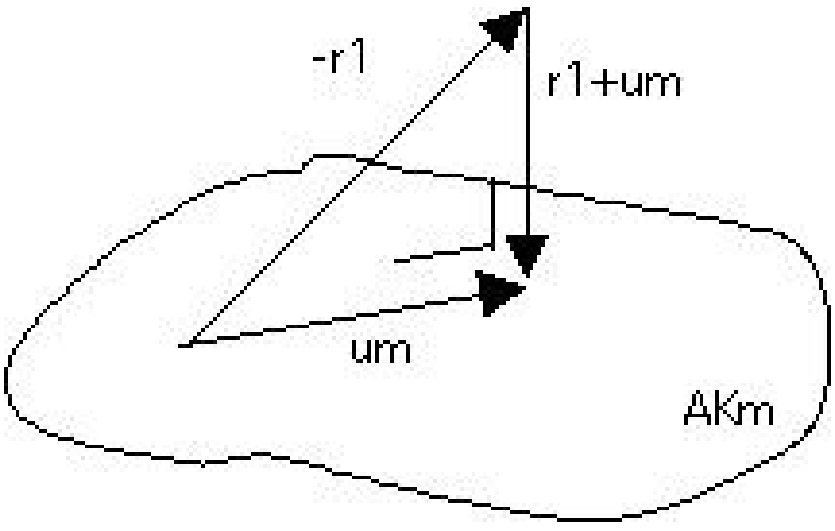
Minimization

Related concepts:

- Positive definite operator

$$\forall_x : \; x^t A x > 0$$

- Inner product
- Projection
- Minimization


-r1
r1+um
um
AKm

Full Orthogonalization Method

Let r_0 *be given*
For $i \geq 0$*:*
    let $s \leftarrow K^{-1} r_i$
    let $t \leftarrow AK^{-1} r_i$
    for $j \leq i$*:*
        let γ_j *be the coefficient so that* $t - \gamma_j r_j \perp r_j$
    for $j \leq i$*:*
        form $s \leftarrow s - \gamma_j x_j$
        and $t \leftarrow t - \gamma_j r_j$
    let $x_{i+1} = (\sum_j \gamma_j)^{-1} s$, $r_{i+1} = (\sum_j \gamma_j)^{-1} t$.

How do you orthogonalize?

- Given x, y, can you

$$x \leftarrow \text{something with } x,\ y$$

such that $x \perp y$?
(What was that called again in your linear algebra class?

How do you orthogonalize?

- Given x, y, can you

$$x \leftarrow \text{something with } x,\ y$$

  such that $x \perp y$?
  (What was that called again in your linear algebra class?
- Gramm-Schmid method

How do you orthogonalize?

- Given x, y, can you

$$x \leftarrow \text{something with } x,\ y$$

  such that $x \perp y$?
  (What was that called again in your linear algebra class?
- Gramm-Schmid method
- Update

$$x \leftarrow x - \frac{x^t y}{y^t y} y$$

Takeaway

Each iteration involves:

- multiplying by A,
- solving with K
- inner products!

Coupled recurrences form

$$x_{i+1} = x_i - \sum_{j \le i} \alpha_{ji} K^{-1} r_j \tag{4}$$

This equation is often split as

- Update iterate with search direction: direction:

$$x_{i+1} = x_i - \delta_i p_i,$$

- Construct search direction from residuals:

$$p_i = K^{-1} r_i + \sum_{j<i} \beta_{ij} K^{-1} r_j.$$

Inductively:

$$p_i = K^{-1} r_i + \sum_{j<i} \gamma_{ij} p_j,$$

Conjugate Gradients

Basic idea:

$$r_i^t K^{-1} r_j = 0 \quad \text{if } i \neq j.$$

Split recurrences:

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_i = K^{-1} r_i + \sum_{j<i} \gamma_{ij} p_j, \end{cases}$$

Residuals: r_i; search directions: p_i

Symmetric Positive Definite case

Three term recurrence is enough:

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_{i+1} = K^{-1} r_{i+1} + \gamma_i p_i \end{cases}$$

$$\gamma_i = \frac{r_{i+1}^t r_{i+1}}{r_i^t r_i}, \delta_i = \frac{r_i^t r_i}{p_i^t A p_i}$$

Preconditioned Conjugate Gradietns

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1} / p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

takeaway

Each iteration involves:

- Matrix-vector product
- Preconditioner solve
- Two inner products
- Other vector operations.

Three popular iterative methods

- Conjugate gradients: constant storage and inner products; works only for symmetric systems
- GMRES (like FOM): growing storage and inner products: restarting and numerical cleverness
- BiCGstab and QMR: relax the orthogonality

CG derived from minimization

Special case of SPD:

For which vector x with $\|x\| = 1$ is $f(x) = 1/2x^t Ax - b^t x$ minimal? (5)

Taking derivative:

$$f'(x) = Ax - b.$$

Optimal solution:

$$f'(x) = 0 \Rightarrow Ax = b.$$

Traditional: variational formulation
New: error of neural net

Minimization by line search

Assume full minimization $\min_x : f(x) = 1/2 x^t A x - b^t x$ too expensive.
Iterative update

$$x_{i+1} = x_i + p_i \delta_i$$

where p_i is search direction.
Finding optimal value δ_i is 'line search':

$$\delta_i = \operatorname*{argmin}_{\delta} \| f(x_i + p_i \delta) \| = \frac{r_i^t p_i}{p_1^t A p_i}$$

Other constants follow from orthogonality.

Line search

Also popular in other contexts:

- General non-linear systems
- Machine learning: stochastic gradient descent
  p_i is 'block vector' of training set

$$p_1^t A p_i \text{ is a matrix} \Rightarrow \left(p_1^t A p_i\right)^{-1} r_i^t p_i \text{ system solving}$$

Let’s go parallel

Computational aspects of iterative methods

What's in an iterative method?

From easy to hard

- Vector updates
  These are trivial
- Inner product
- Matrix-vector product
- Preconditioner solve

Inner products: collectives

Collective operation: data from all processes is combined.
(Is a matrix-vector product a collective?)
Examples: sum-reduction, broadcast
These are each other's mirror image, computationally.

Naive realization of collectives

Broadcast:

Single message:

$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

Naive realization of collectives

Broadcast:

Single message:

$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

- Single inner product: $n = 1$

Naive realization of collectives

Broadcast:

Single message:

$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

- Single inner product: $n = 1$
- Time for collective?

Naive realization of collectives

Broadcast:

Single message:

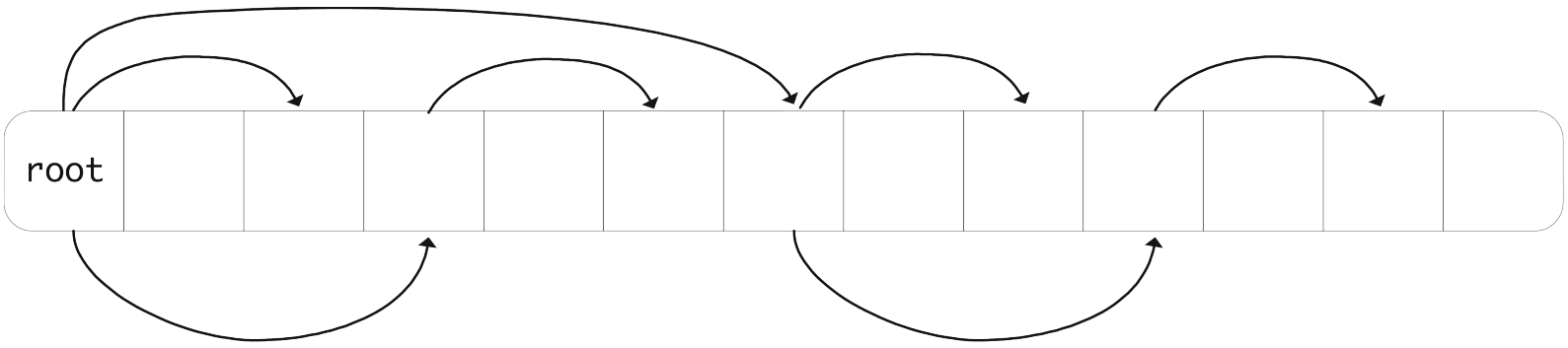
$$\alpha = \text{message startup} \approx 10^{-6}s, \qquad \beta = \text{time per word} \approx 10^{-9}s$$

- Time for message of n words:

$$\alpha + \beta n$$

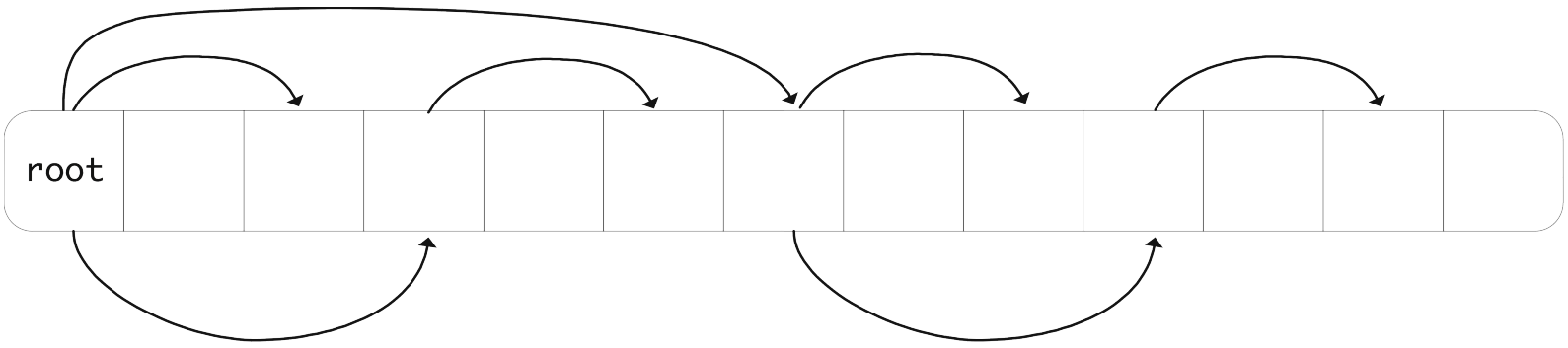
- Single inner product: $n = 1$
- Time for collective?
- Can you improve on that?

Better implementation of collectives



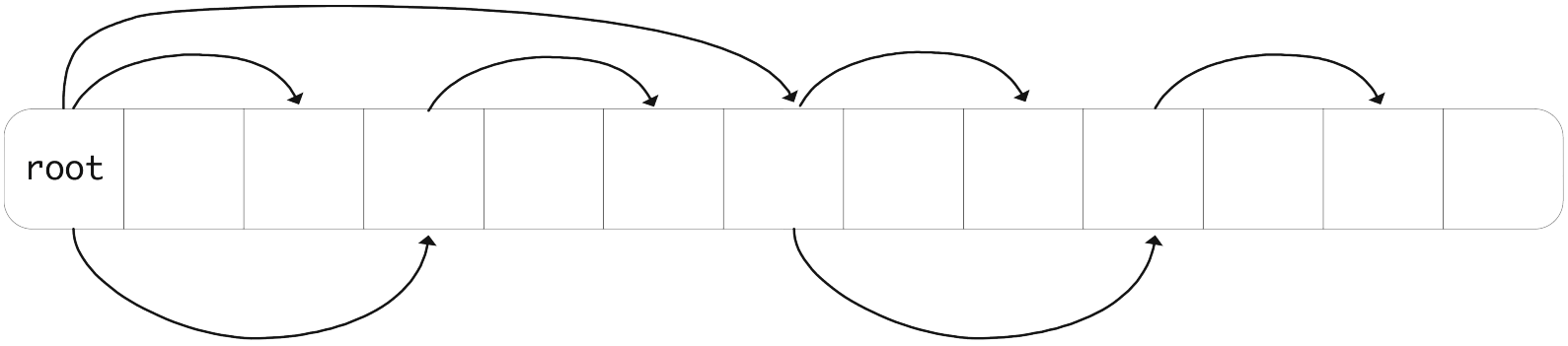
- What is the running time now?

Better implementation of collectives



- What is the running time now?
- Can you come up with lower bounds on the α, β terms? Are these achieved here?

Better implementation of collectives



- What is the running time now?
- Can you come up with lower bounds on the α, β terms? Are these achieved here?
- How about the case of really long buffers?

Inner products

- Only operation that intrinsically has a p dependence
- Collective, so induces synchronization
- $\Rightarrow$ exposes load unbalance, can take lots of time
- Research in approaches to hiding: overlapping with other operations

What do those inner products serve?

- Orthogonality of residuals
- Basic algorithm: Gram-Schmidt
- one step: given u, v

$$v' \leftarrow v - \frac{u^t v}{u^t u} u.$$

  then $v' \perp u$
- bunch of steps: given U, v

$$v' \leftarrow v - \frac{U^t v}{U^t U} U.$$

  then $v' \perp U$.

Gram-Schmidt algorithm

Modified Gram-Schmidt

For $i = 1, \ldots, n$:

let $c_i = u_i^t v / u_i^t u_i$

update $v \leftarrow v - c_i u_i$

More numerical stable

Full Orthogonalization Method

Let r_0 *be given*
For $i \geq 0$*:*
    let $s \leftarrow K^{-1} r_i$
    let $t \leftarrow AK^{-1} r_i$
    for $j \leq i$*:*
        let γ_j *be the coefficient so that* $t - \gamma_j r_j \perp r_j$
    for $j \leq i$*:*
        form $s \leftarrow s - \gamma_j x_j$
        and $t \leftarrow t - \gamma_j r_j$
    let $x_{i+1} = (\sum_j \gamma_j)^{-1} s$, $r_{i+1} = (\sum_j \gamma_j)^{-1} t$.

Modified Gramm-Schmidt

Let r_0 *be given*
For $i \geq 0$*:*
    let $s \leftarrow K^{-1} r_i$
    let $t \leftarrow AK^{-1} r_i$
    for $j \leq i$*:*
        let γ_j *be the coefficient so that* $t - \gamma_j r_j \perp r_j$
        form $s \leftarrow s - \gamma_j x_j$
        and $t \leftarrow t - \gamma_j r_j$
    let $x_{i+1} = (\sum_j \gamma_j)^{-1} s$, $r_{i+1} = (\sum_j \gamma_j)^{-1} t$.

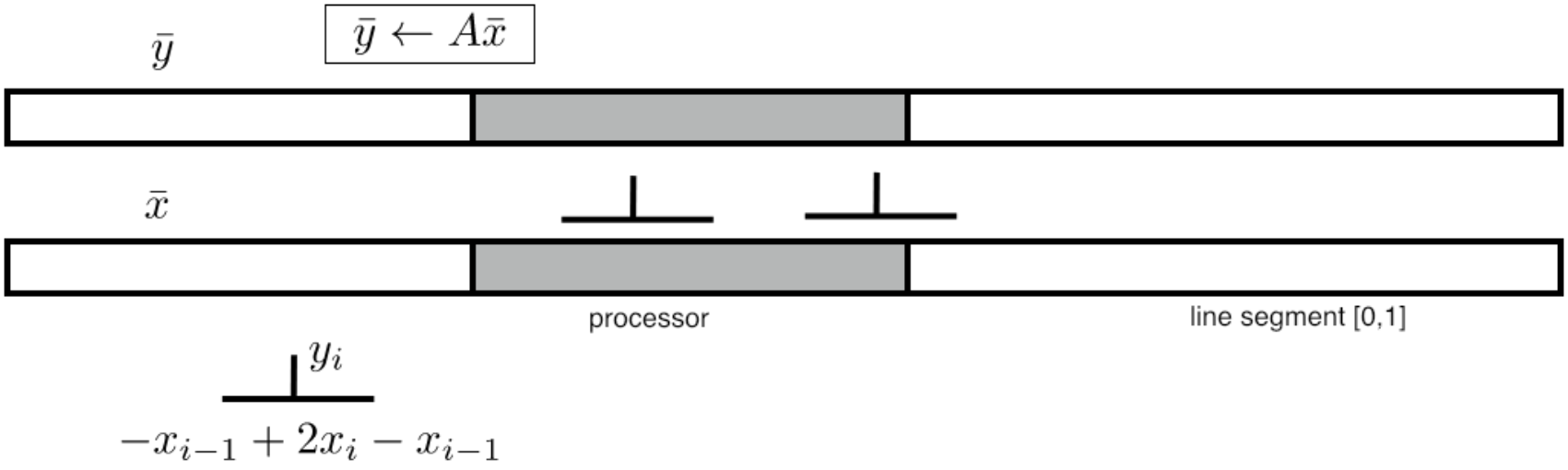
Practical differences

- Modfied GS more stable
- Inner products are global operations: costly

Matrix-vector product

Parallelization

Assume each process has the matrix values and vector values in part of the domain.



Processor needs to get values from neighbors.

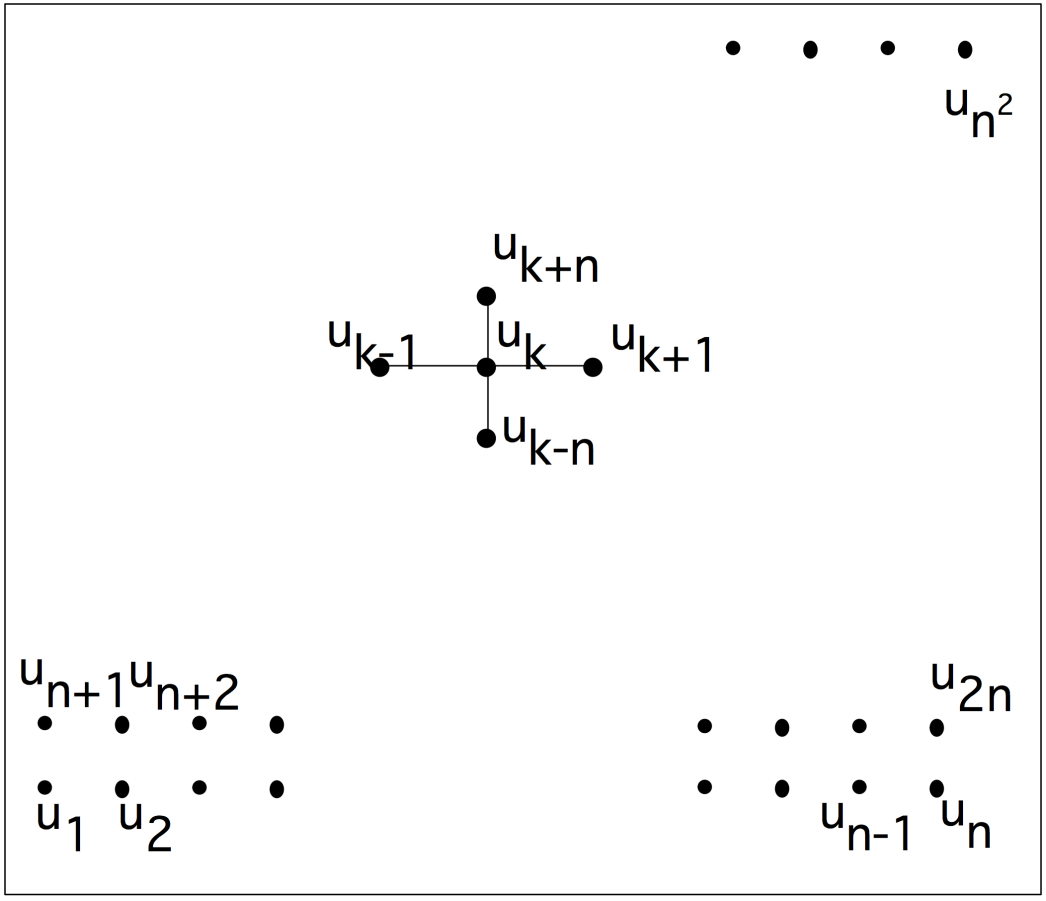
Second order PDEs; 2D case

$$\begin{cases} -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) & x \in \Omega = [0,1]^2 \\ u(\bar{x}) = u_0 & \bar{x} \in \delta\Omega \end{cases}$$

Now using central differences in both x and y directions:

$$4u(x,y) - u(x+y,y) - u(x-h,y) - u(x,y+h) - u(x,y-h)$$

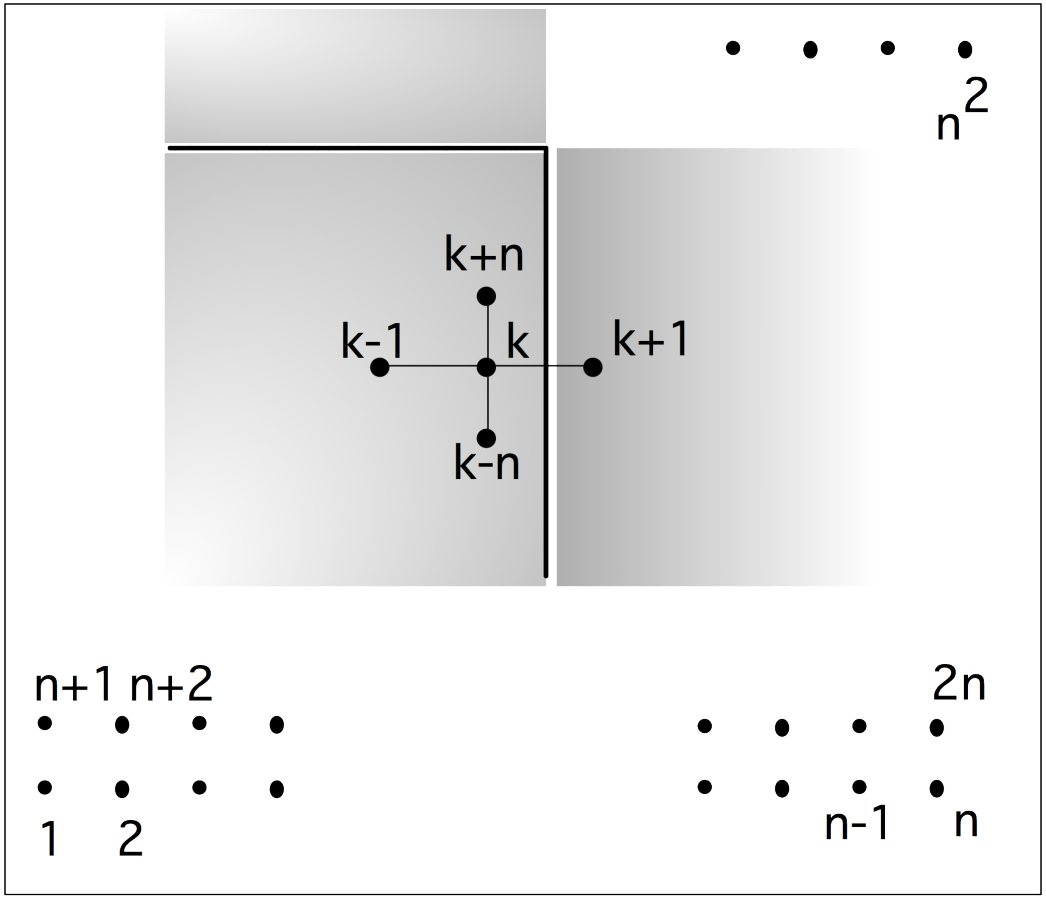
The stencil view of things



$$\begin{array}{ccc} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{array}$$

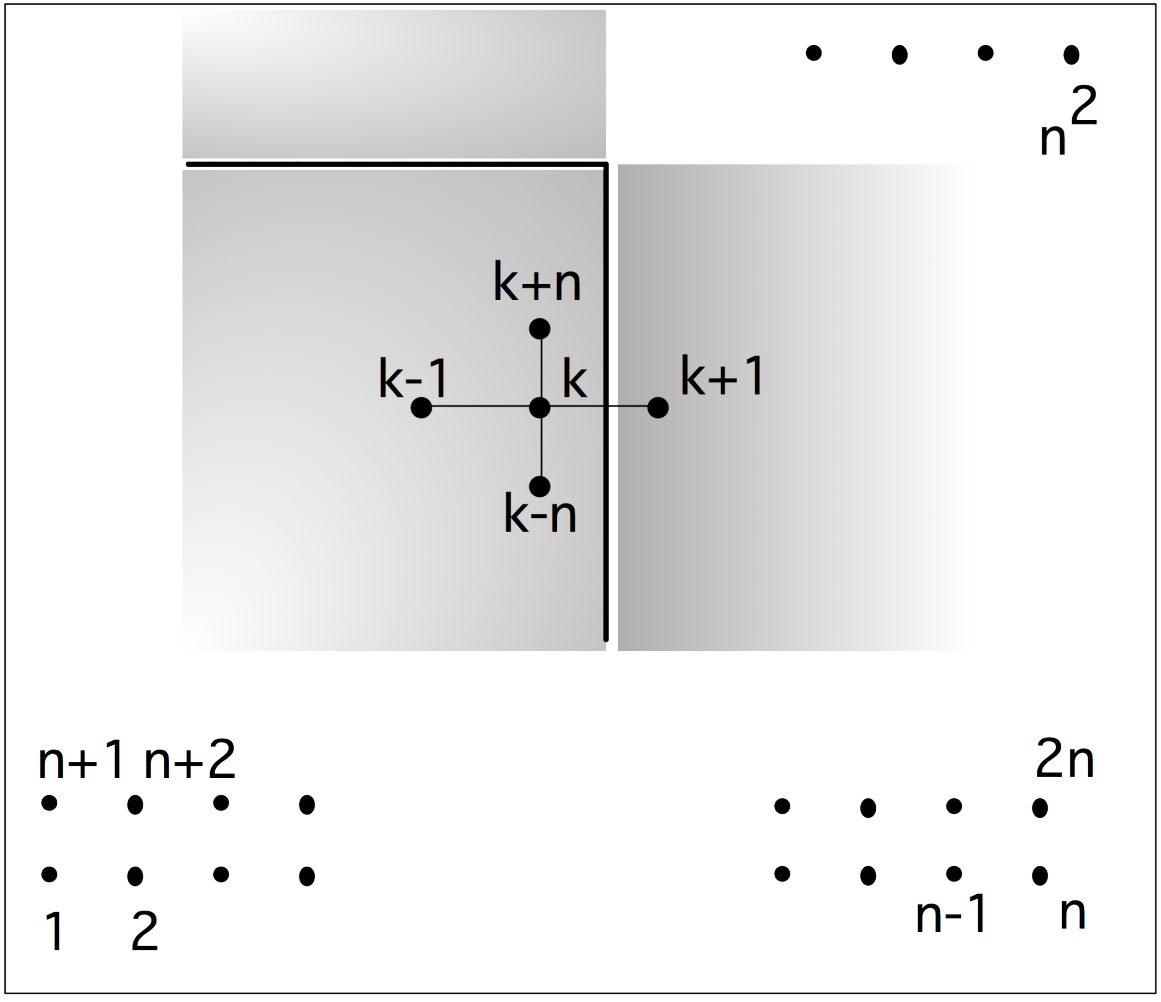
Operator view of spmvp

Difference stencil



PDE, 2D case

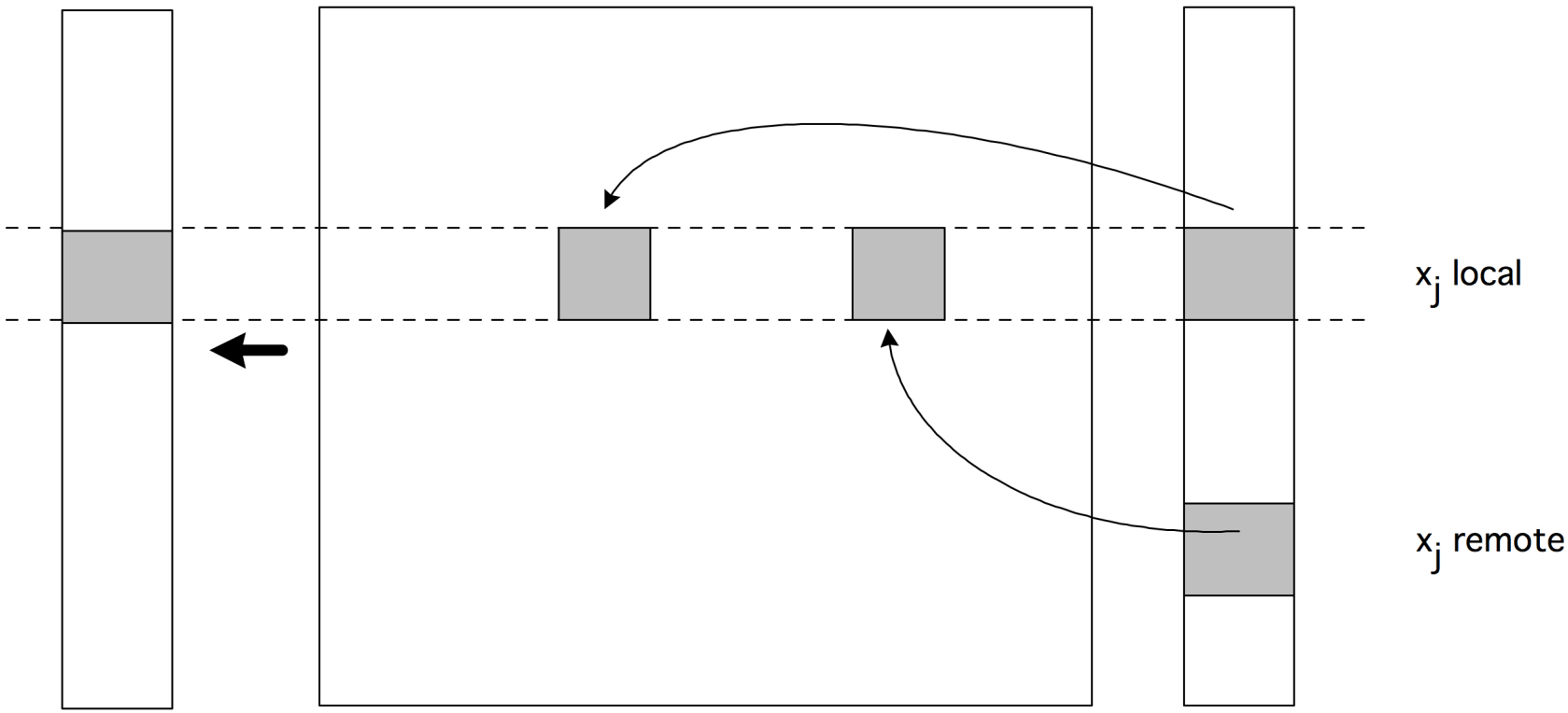
A difference stencil applied to a two-dimensional square domain, distributed over processors. Each point connects to neighbors $\Rightarrow$ each process connects to neighbors.



Matrices in parallel

$$y \leftarrow Ax$$

and A, x, y all distributed:



Matrix-vector product performance

- Large scale:
  - partition for scalability
  - minimize communication (Metis, Zoltan: minimize edge cuts)
  - dynamic load balancing? requires careful design
- Processor scale:
  - Performance largely bounded by bandwidth
  - Some optimization possible

Beware of optimizations that change the math!

Exercise 1

Show that a one-dimensional partitioning of the domain leads to a partitioning of the matrix into block rows with a contiguous block for each process. Show that a two-dimensional partitioning of the domain does not give this structure.

Exercise 2

Machine:

- Network *Latency:* $\alpha = 1\mu s = 10^{-6}s$.
- Network *Bandwidth:* $1Gb/s$ corresponds to $\beta = 10^{-9}$.
- *Computation rate:* A per-core flops rate of $1G$flops means $\gamma = 10^{-9}$.

Case 1. divide the domain, by horizontal slabs of size $n \times (n/p)$. Analyze communication and compuputation complexity and weak scaling efficiency. Show that this does not scale weakly.
Case 2. Divide the domain into patches of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. Show that this case does scale weakly.

Preconditioners

Preconditioners

- There's much that can be said here.
- Some comments to follow
- There is intrinsic dependence in solvers, hence in preconditioners:
  parallelism is very tricky.
  approximate inverses

Parallel LU through nested dissection

Fill-in during LU

Fill-in: index (i, j) where $a_{ij} = 0$ but $\ell_{ij} \neq 0$ or $u_{ij} \neq 0$.
2D BVP: Ω is $n \times n$, gives matrix of size $N = n^2$, with bandwidth n.
Matrix storage $O(N)$
LU storage $O(N^{3/2})$
LU factorization work $O(N^2)$
Cute fact: storage can be computed linear in #nonzeros

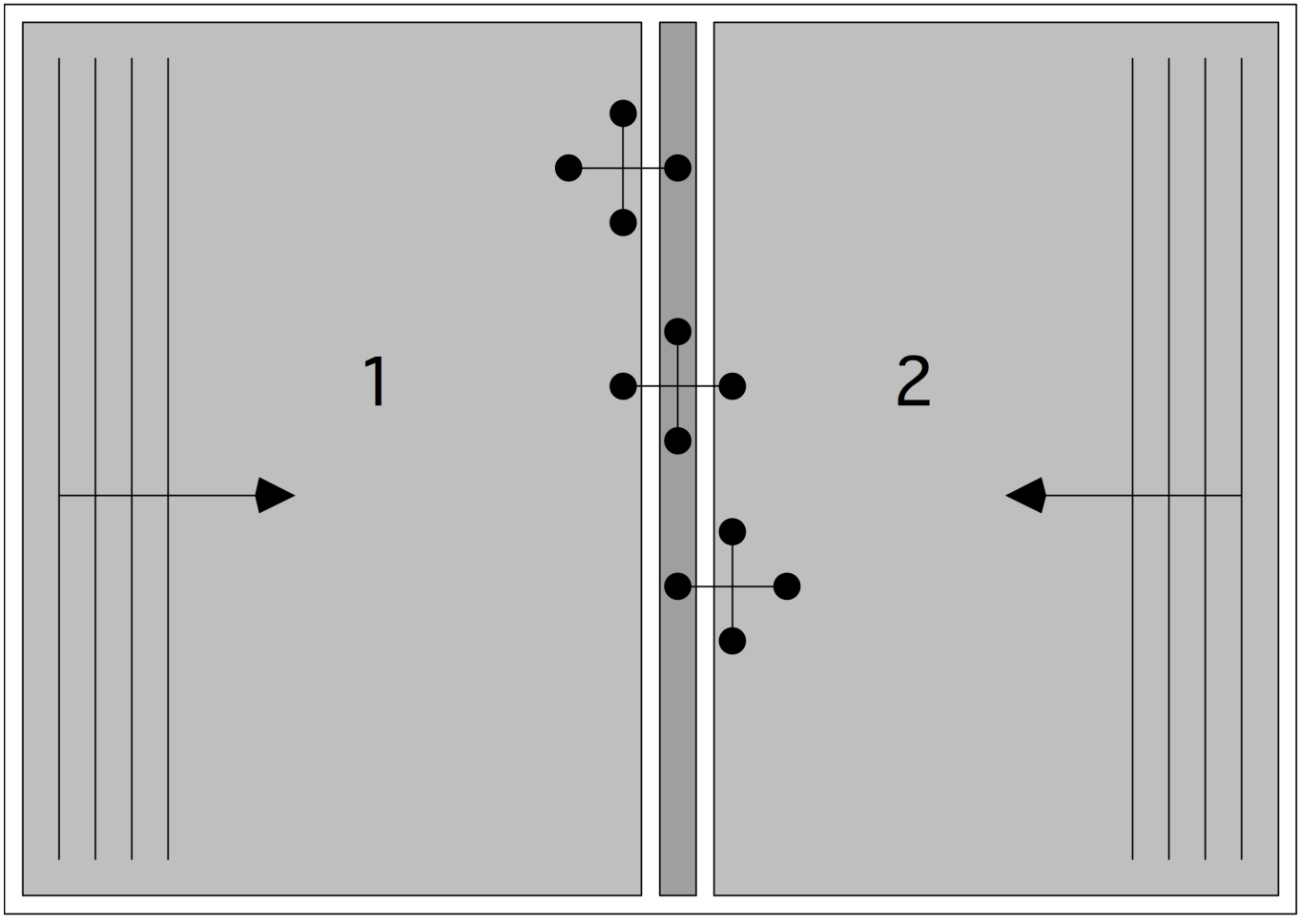
Fill-in is a function of ordering

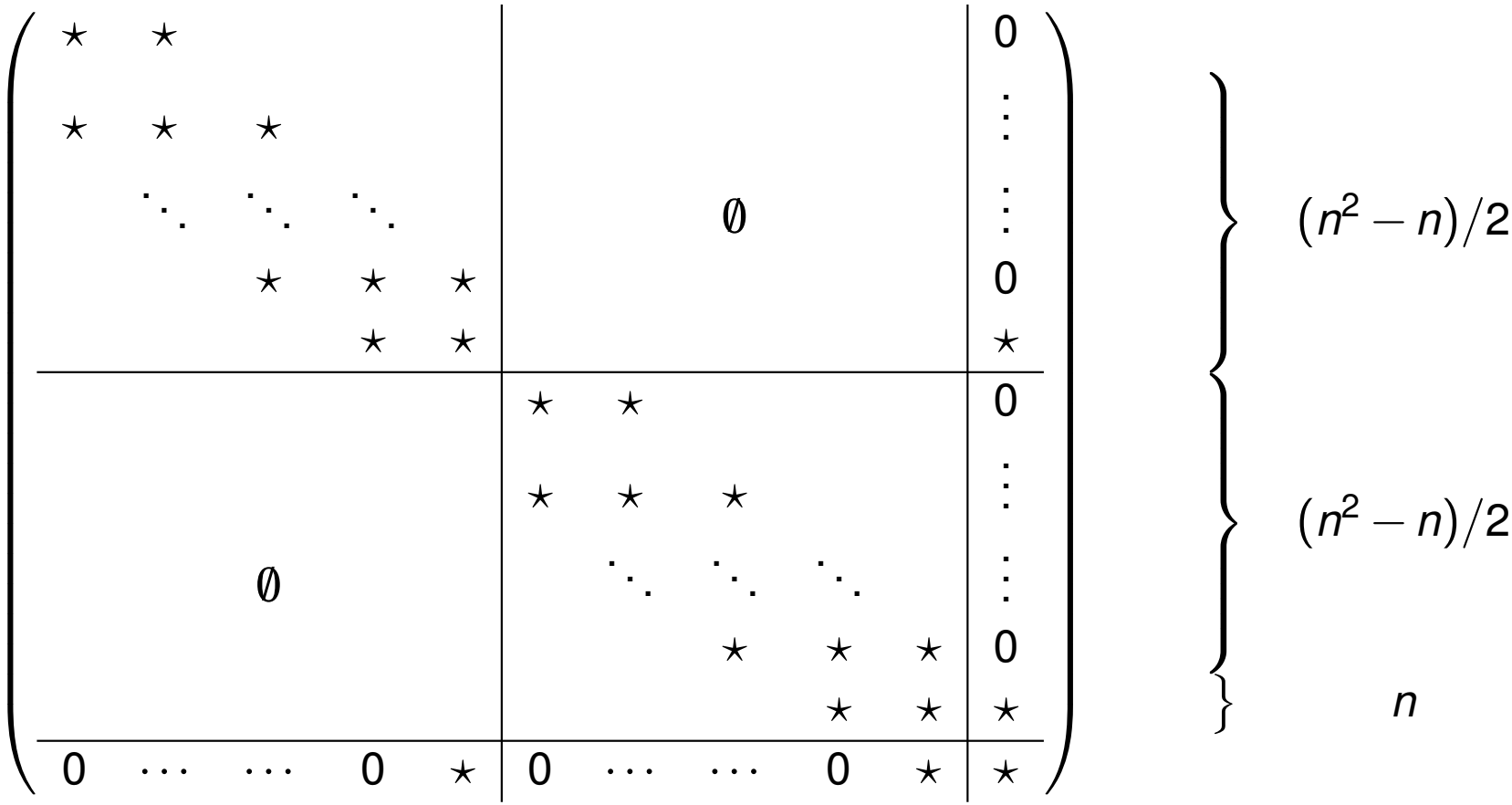
$$\begin{pmatrix} * & * & \cdots & * \\ * & * & & \emptyset \\ \vdots & & \ddots & \\ * & \emptyset & & * \end{pmatrix}$$

After factorization the matrix is dense.

Can this be permuted?

Domain decomposition




$(n^2-n)/2$
$(n^2-n)/2$
n

DD factorization

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} =$$

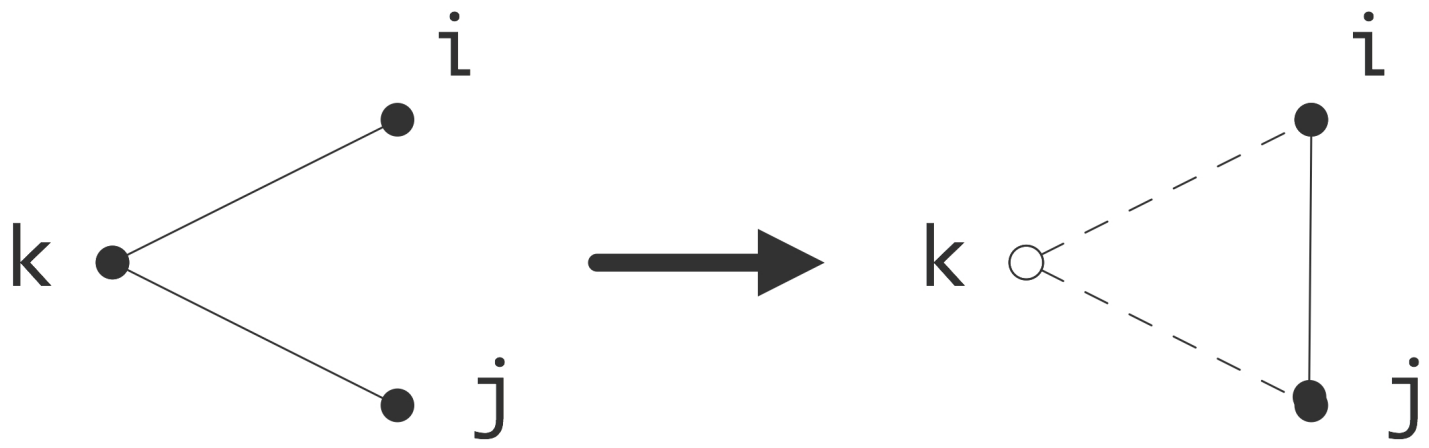
$$\begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ & A_{22} & A_{23} \\ & & S \end{pmatrix}$$

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

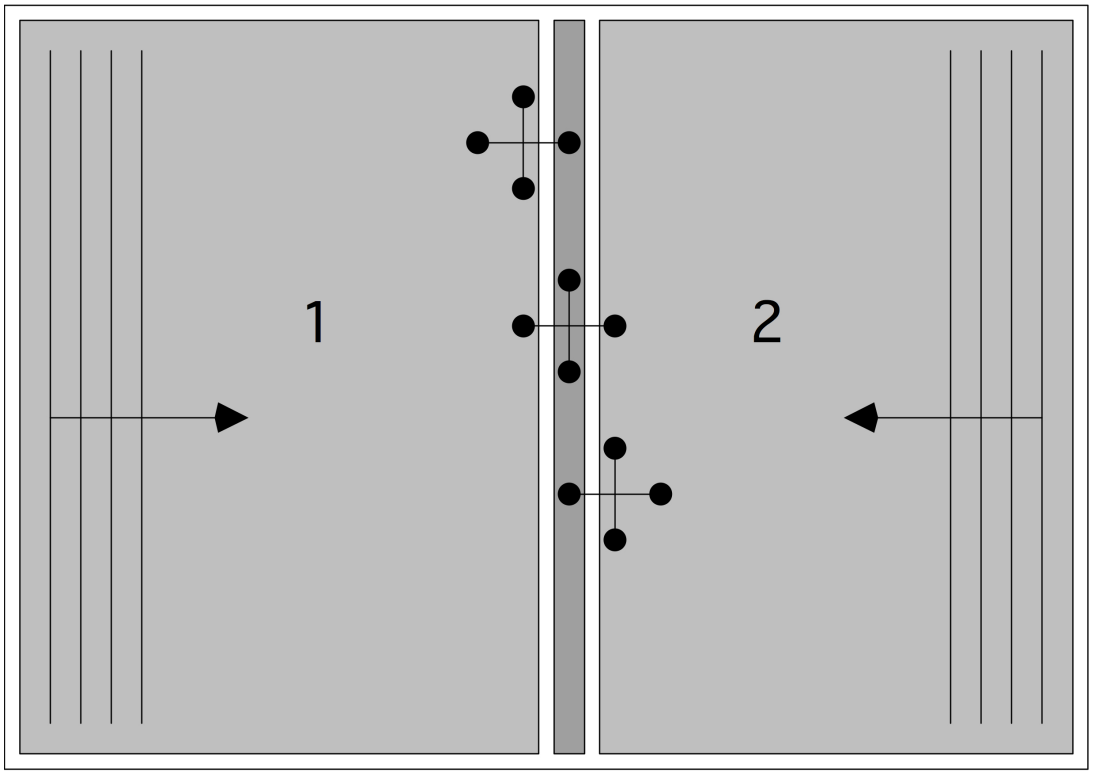
Parallelism...

Graph theory of sparse elimination

$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}$$

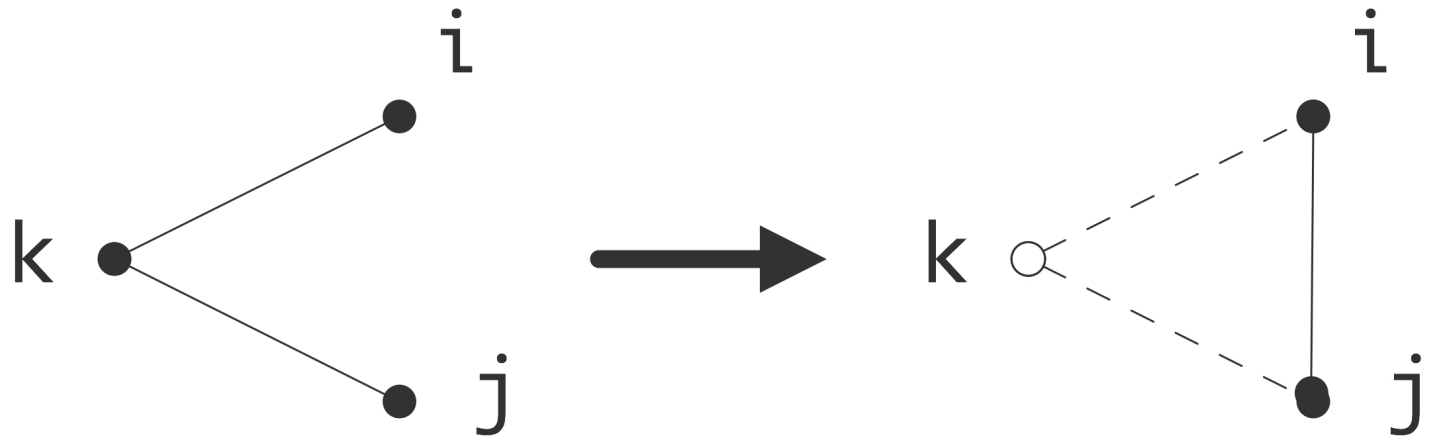


Graph theory of sparse elimination



Graph theory of sparse elimination

$$a_{ij} \leftarrow a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}$$



So inductively S is dense

More about separators

- This is known as ‘domain decomposition’ or ‘substructuring’
- Separators have better spectral properties

Recursive bisection

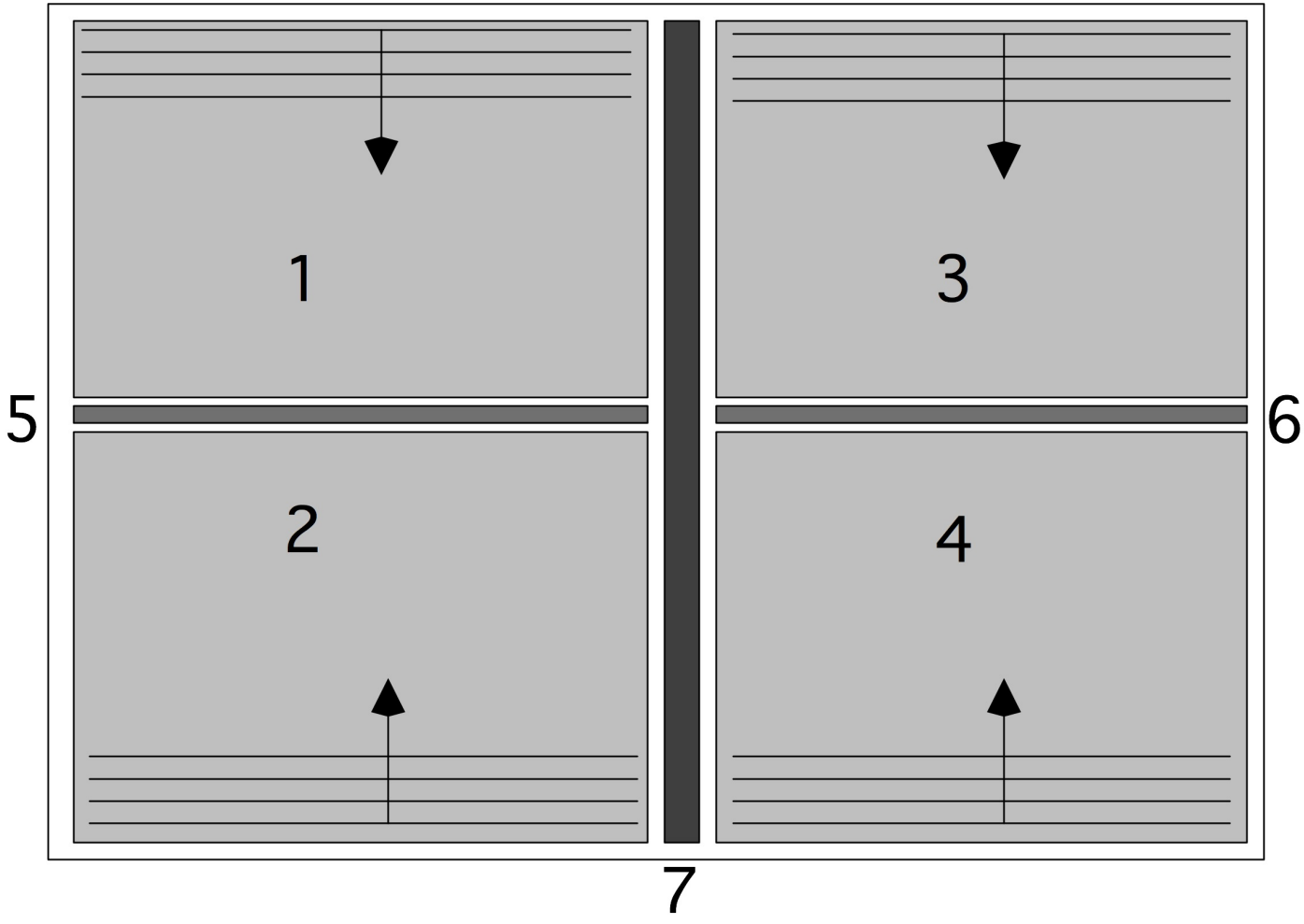
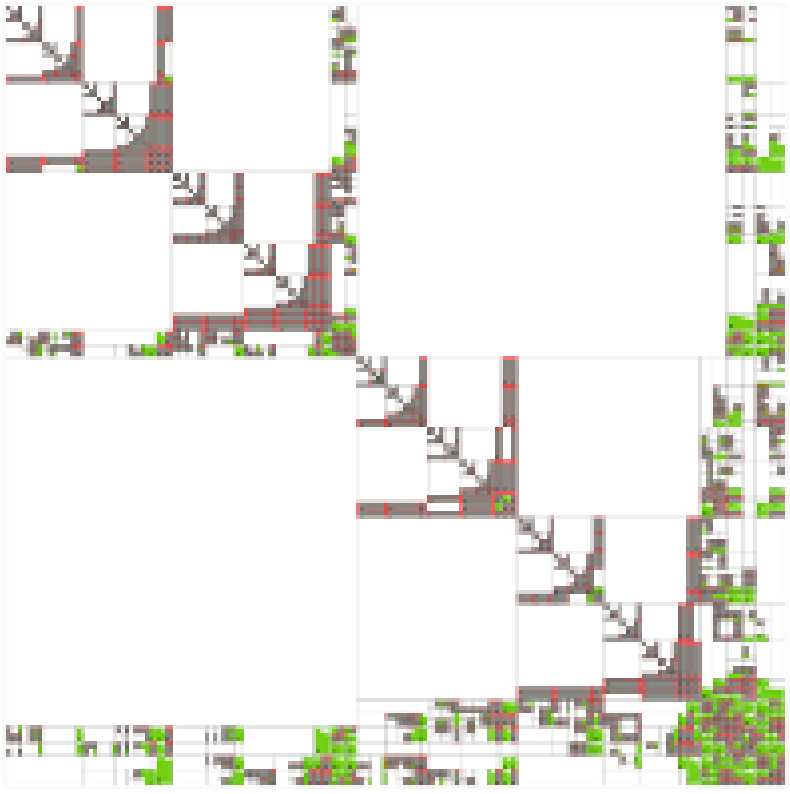


Figure: A four-way domain decomposition.

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix}$$

The domain/operator/graph view is more insightful, don't you think?

How does this look in reality?



Complexity

With $n = \sqrt{N}$:

- one dense matrix on a separator of size n, plus
- two dense matrices on separators of size $n/2$
- $\rightarrow 3/2\, n^2$ space and $5/12\, n^3$ time
- and then four times the above with $n \rightarrow n/2$

$$\begin{aligned}\text{space} &= 3/2 n^2 + 4 \cdot 3/2 (n/2)^2 + \cdots \\ &= N(3/2 + 3/2 + \cdots) \quad \log n \text{ terms} \\ &= O(N \log N)\end{aligned}$$

$$\begin{aligned}\text{time} &= 5/12 n^3/3 + 4 \cdot 5/12 (n/2)^3/3 + \cdots \\ &= 5/12 N^{3/2}(1 + 1/4 + 1/16 + \cdots) \\ &= O(N^{3/2})\end{aligned}$$

Unfortunately only in 2D.

More direct factorizations

Minimum degree, multifrontal,. . .

Finding good separators and domain decompositions is tough in general.

Incomplete approaches to matrix factorization

Sparse operations in parallel: mvp

Mvp $y = Ax$

```
for i=1..n
  y[i] = sum over j=1..n a[i,j]*x[j]
```

In parallel:

```
for i=myfirstrow..mylastrow
  y[i] = sum over j=1..n a[i,j]*x[j]
```

How about ILU solve?

Consider $Lx = y$

```
for i=1..n
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j])
         / a[i,i]
```

Parallel code:

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j])
         / a[i,i]
```

Problems?

Block method

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=myfirstrow..i-1 ell[i,j]*x[j])
         / a[i,i]
```

Block Jacobi with local GS solve

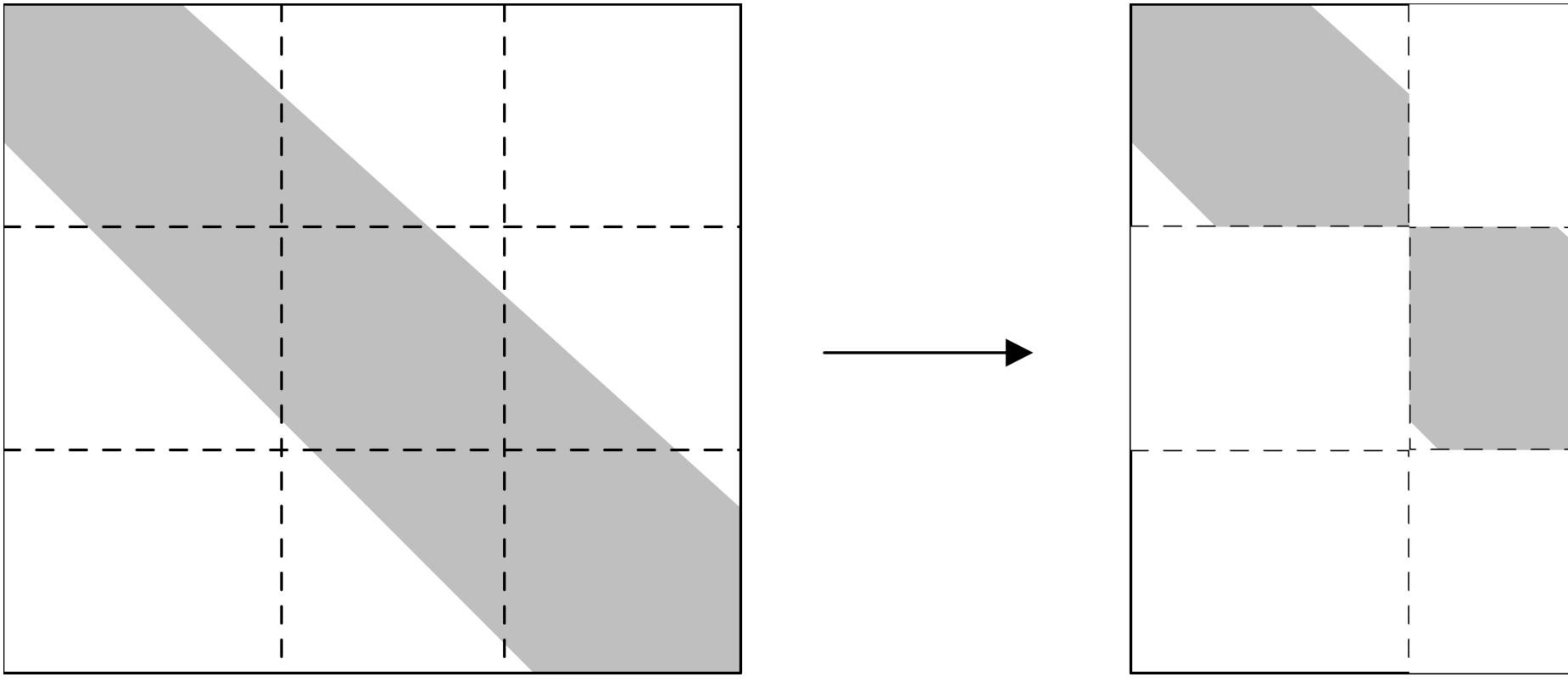


Figure: Sparsity pattern corresponding to a block Jacobi preconditioner.

Variable reordering

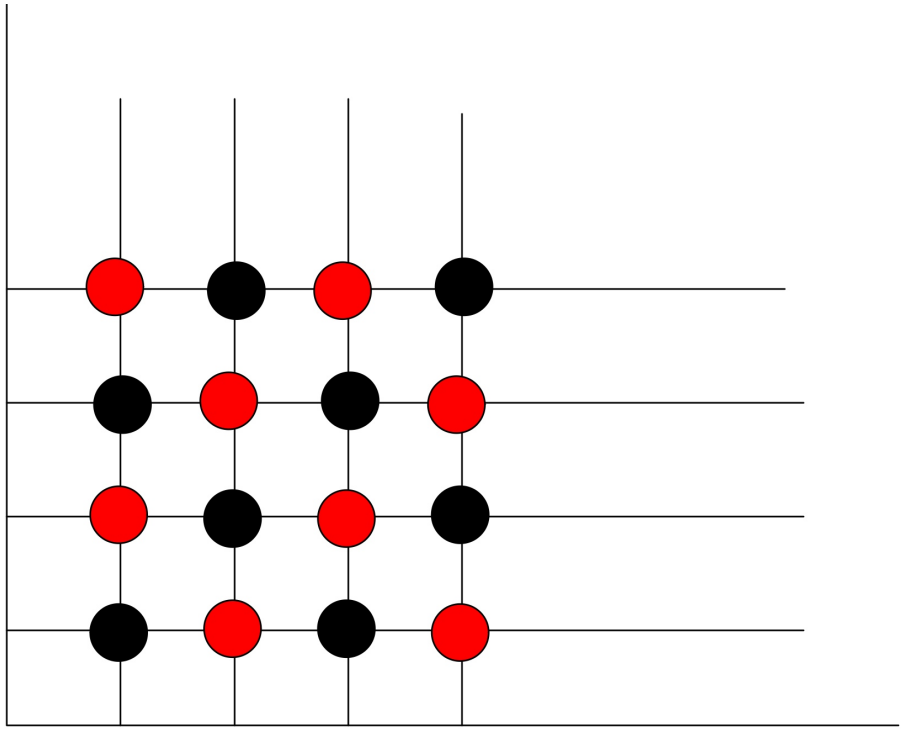
$$\begin{pmatrix} a_{11} & a_{12} & & & & \emptyset \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & \ddots & \ddots & \ddots & \\ \emptyset & & & & & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

with redblack

$$\begin{pmatrix} a_{11} & & & & a_{12} & \\ & a_{33} & & & a_{32} & a_{34} \\ & & a_{55} & & & \ddots & \ddots \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_5 \\ \vdots \\ y_2 \\ y_4 \\ \vdots \end{pmatrix}$$

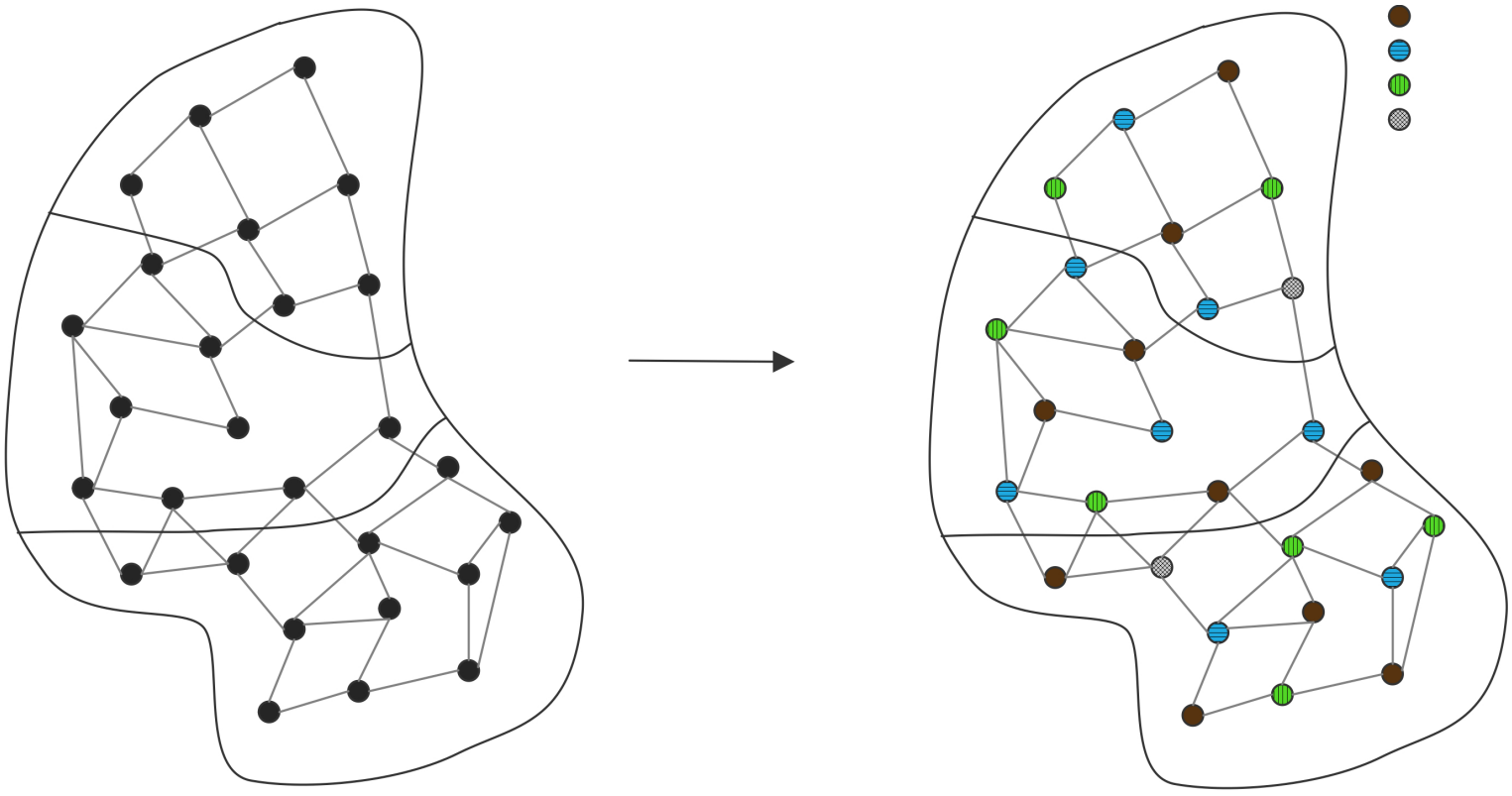
Two-processor parallel Gauss-Seidel or ILU

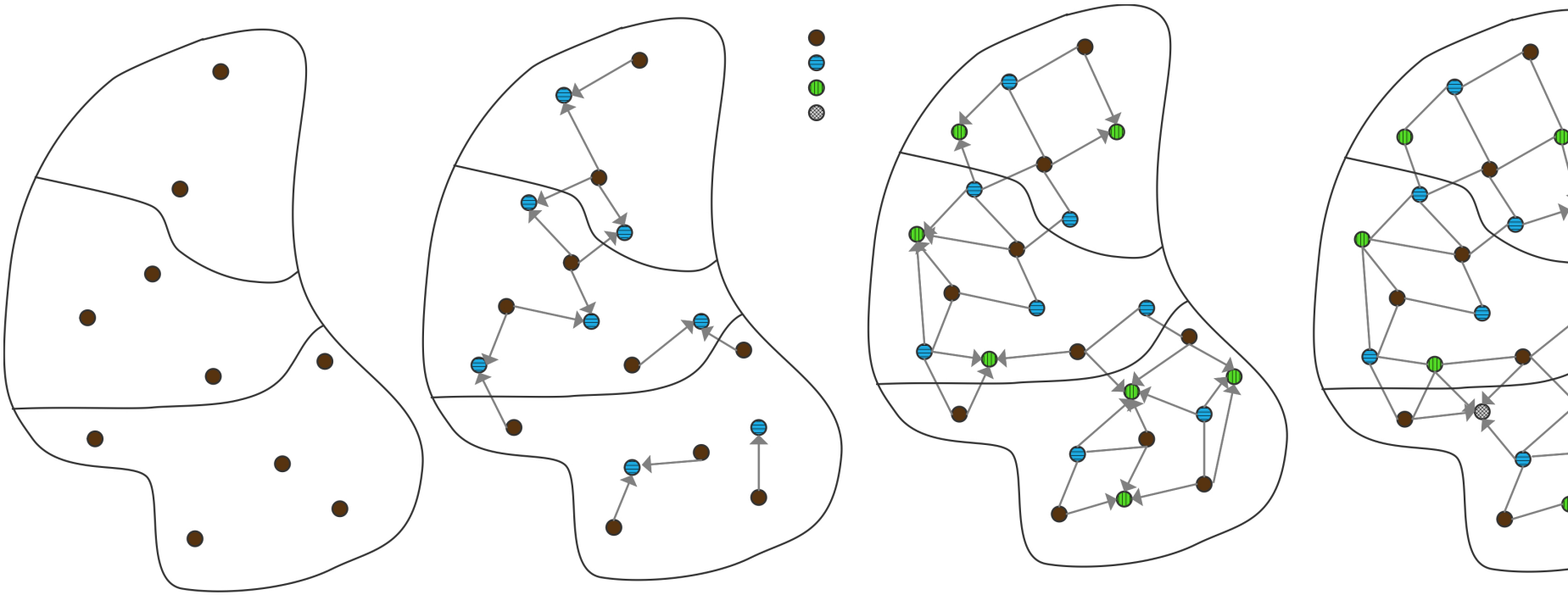
2D redblack



In general, colouring, colour number

Multicolour ILU





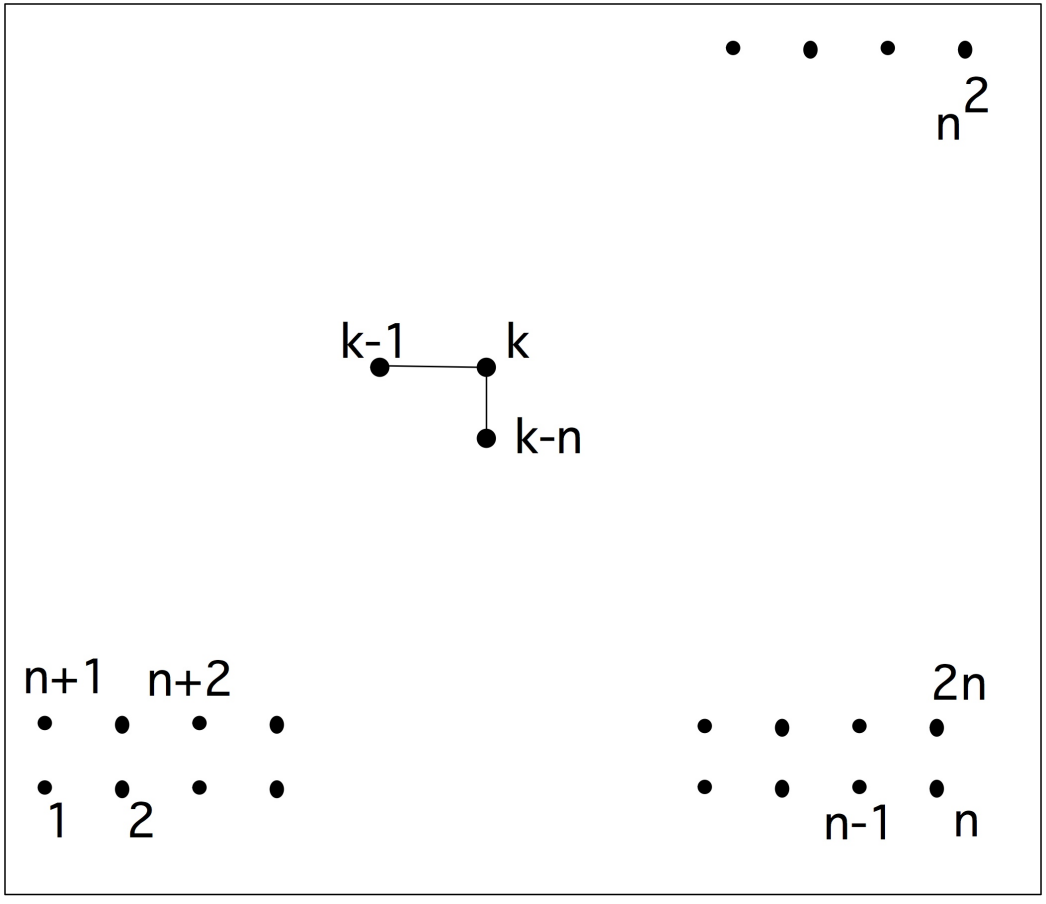
How do you get a multi-colouring?

Exactly colour number is NP-completely: don't bother.
For preconditioner an approximation is good enough:
Luby / Jones-Plassman algorithm

- Give every node a random value
- First colour: all nodes with a higher value than all their neighbors
- Second colour: higher value than all neighbors except in first colour
- et cetera

Parallelism and implicit operations: wavefronts, approximation

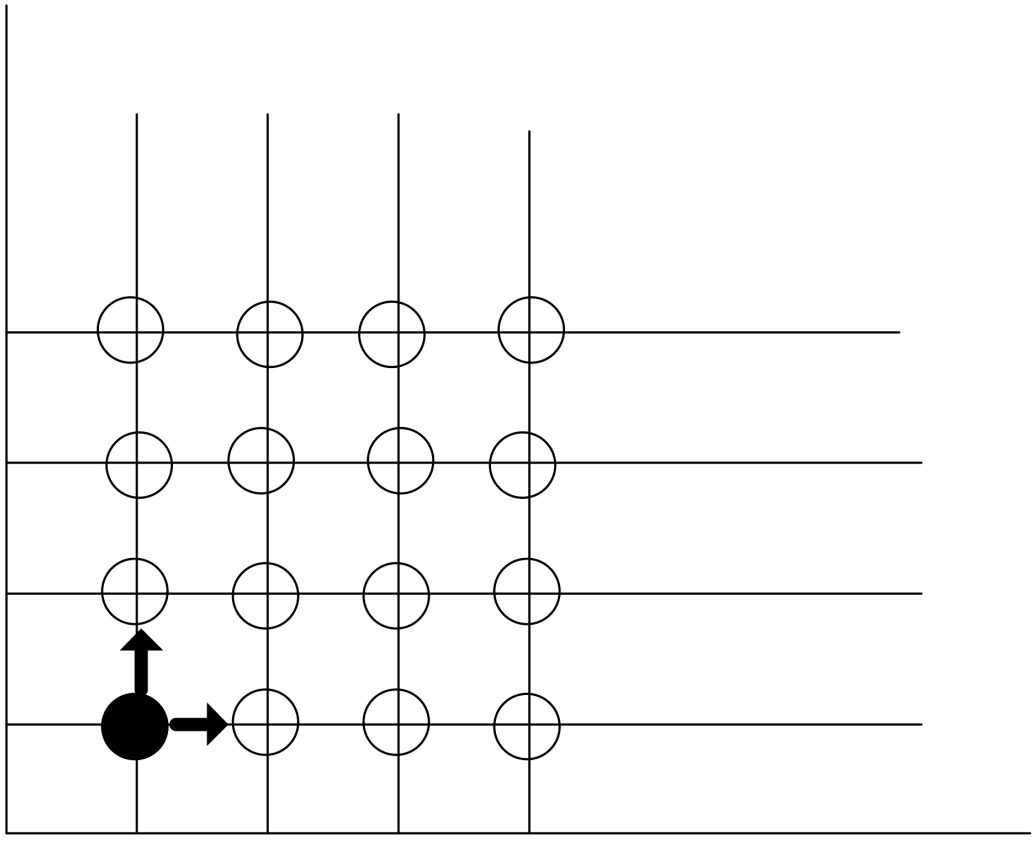
Recurrences

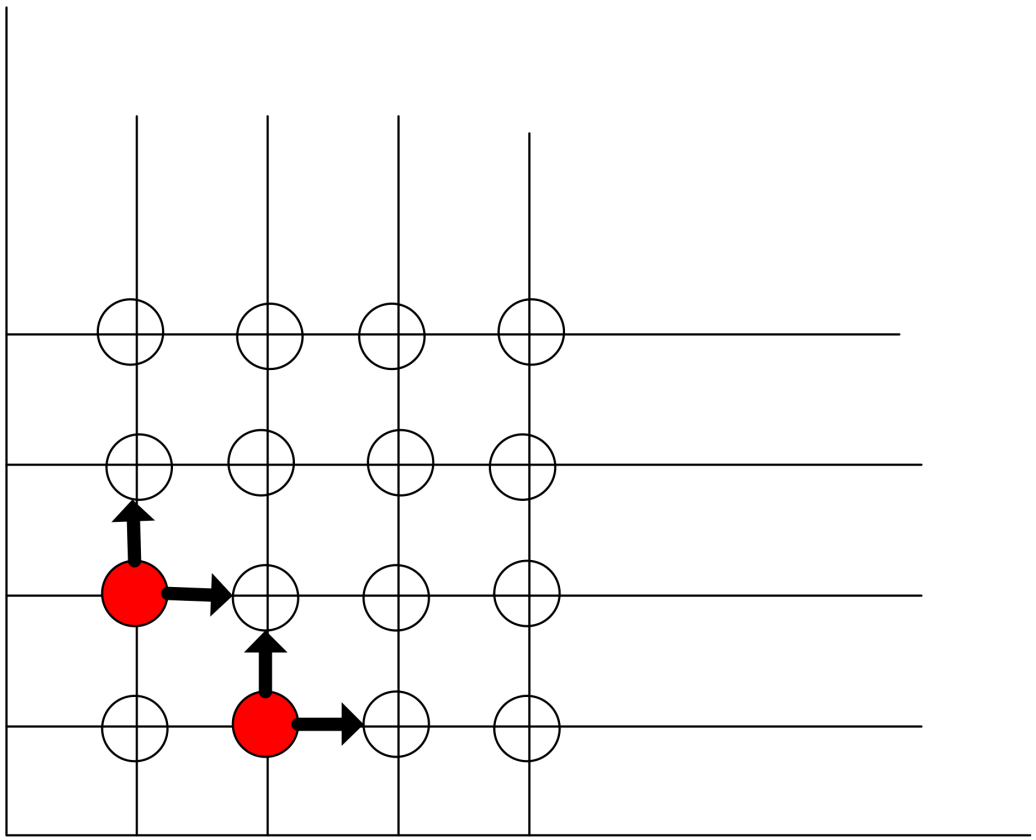


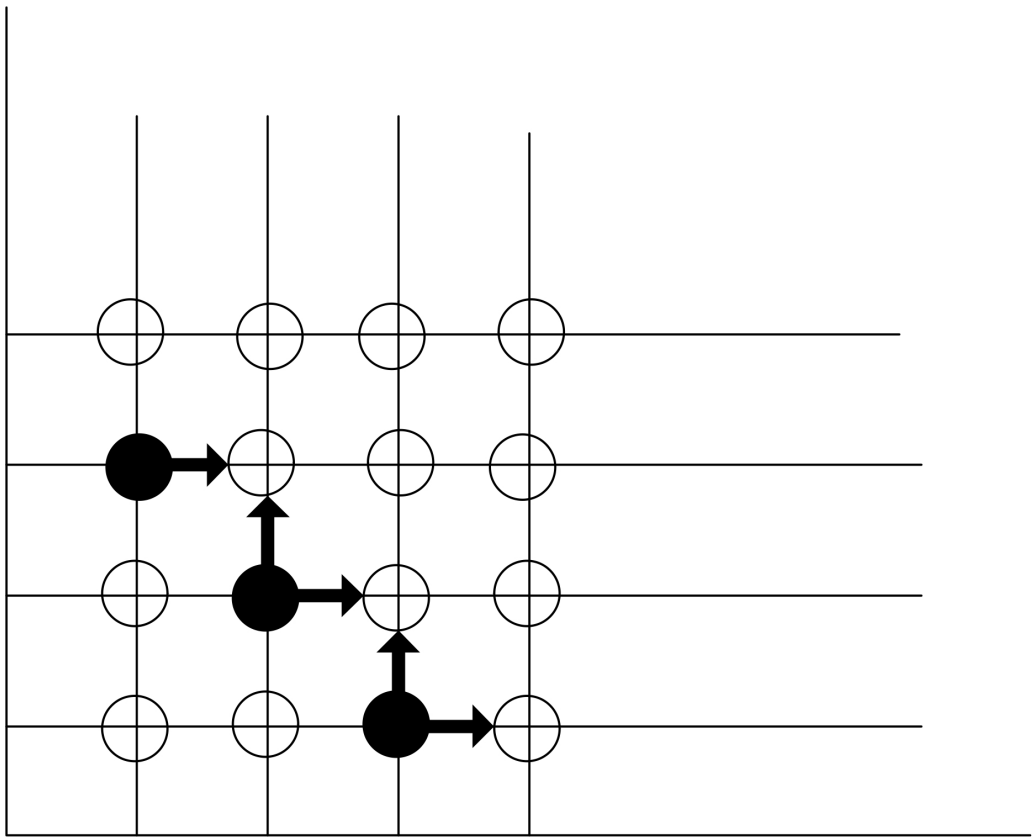
$$x_{i,j} = f(x_{i-1,j}, x_{i,j-1})$$

Intuitively: recursion length n^2

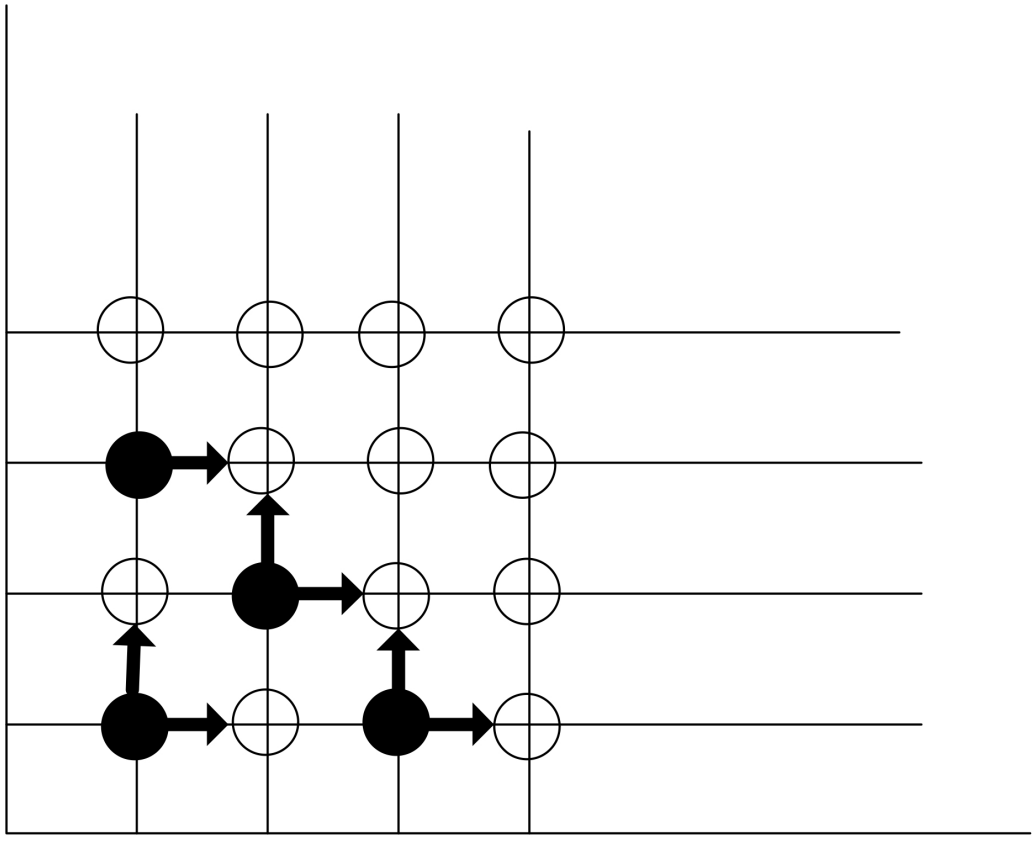
However. . .



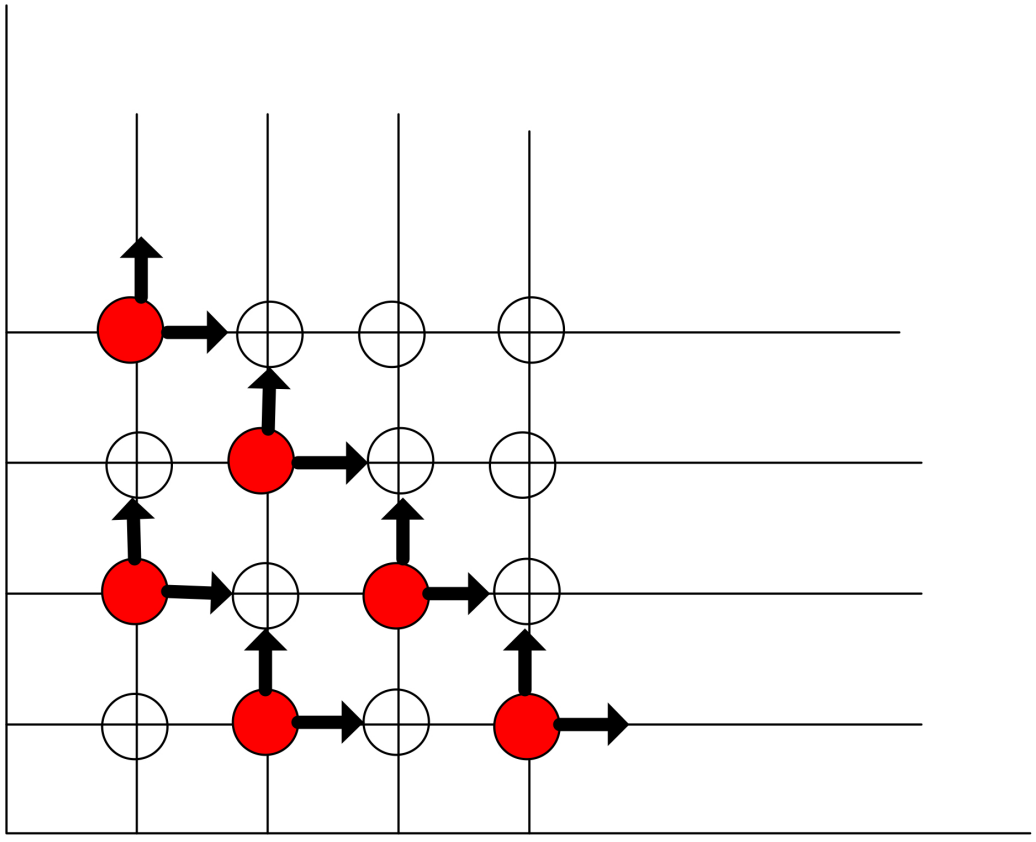




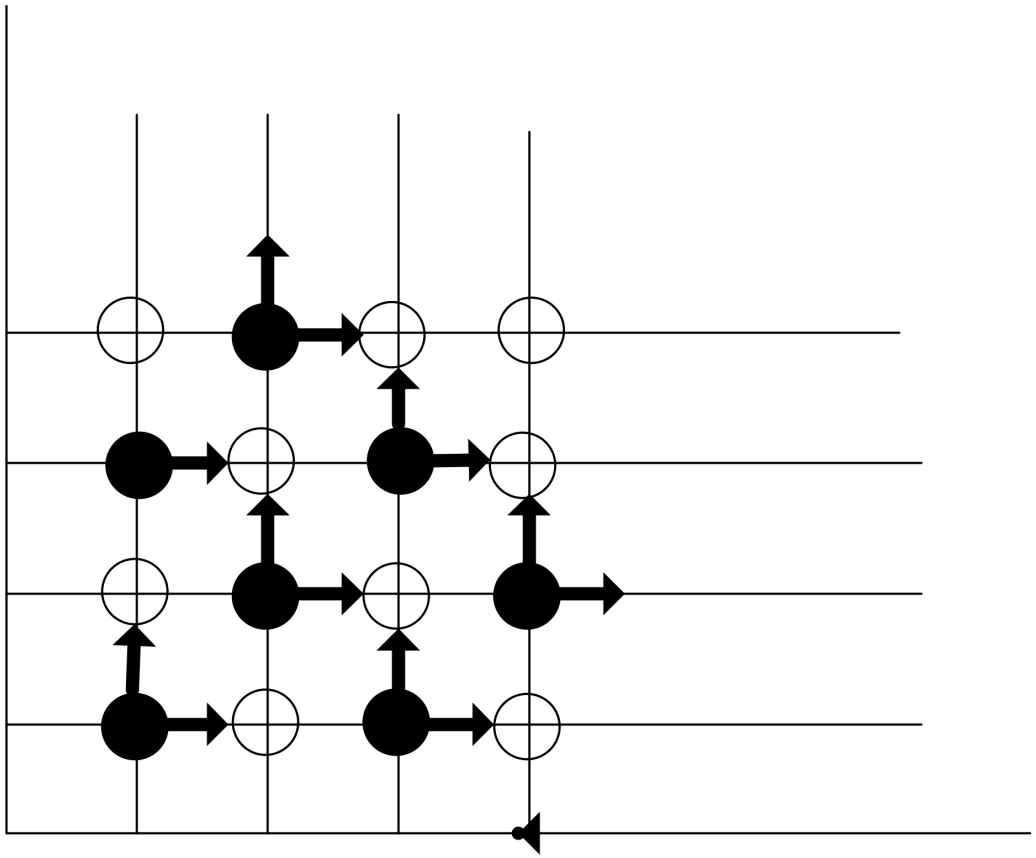
And in fact



But then too



And



Conclusion

1. Wavefronts have sequential length $2n$, average parallelism $n/2$
2. Equivalency of wavefronts and multicolouring

Recursive doubling

Write recurrence $x_i = b_i - a_{i-1} x_{i-1}$ as

$$\begin{pmatrix} 1 & & \emptyset & \\ a_{21} & 1 & & \\ & \ddots & \ddots & \\ \emptyset & & a_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

for short: $A = I + B$

Transform

$$\begin{pmatrix} 1 & & & & & & & \emptyset \\ 0 & 1 & & & & & & \\ & -a_{32} & 1 & & & & & \\ & & 0 & 1 & & & & \\ & & & -a_{54} & 1 & & & \\ & & & & 0 & 1 & & \\ & & & & & -a_{76} & 1 & \\ & & & & & & \ddots & \ddots \end{pmatrix} \times (I+B) =$$

$$\begin{pmatrix} 1 & & & & & & & \emptyset \\ a_{21} & 1 & & & & & & \\ -a_{32}a_{21} & 0 & 1 & & & & & \\ & & a_{43} & 1 & & & & \\ & & -a_{54}a_{43} & 0 & 1 & & & \\ & & & & a_{65} & 1 & & \\ & & & & -a_{76}a_{65} & 0 & 1 & \\ & & & & & \ddots & \ddots & \ddots \end{pmatrix}$$

- Recurrence over half the elements
- Parallel calculation of other half
- Now recurse...

Turning implicit operations into explicit

Normalize ILU solve to $(I - L)$ and $(I - U)$
Approximate $(I - L)x = y$ by $x \approx (I + L + L^2)y$
Convergence guaranteed for diagonally dominant