# Iterative Solution of Sparse Equation Systems

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 15/16

# Topics

Solution of tridiagonal and sparse linear equation systems

- Optimal sequential algorithm
- Cyclic reduction
- Domain decomposition
- $LU$ decomposition of sparse matrices
- Parallelisation

# Optimal Sequential Algorithm

- As an extreme case of sparse equation systems we consider

$$Ax = b \qquad (1)$$

with $A \in \mathcal{R}^{N \times N}$ tridiagonal.

$$\begin{pmatrix} * & * & & & & \\ * & * & * & & & \\ & * & * & * & & \\ & & * & * & * & \\ & & & * & * & * \\ & & & & * & * \end{pmatrix}$$

- The optimal algorithm is the Gaussian elimination, sometimes also called Thomas algorithm.

# Optimal Sequential Algorithm

- Gaussian elimination for tridiagonal systems

  // Forward elimination (here solution, not $LU$ decomposition):
  **for** $(k = 0; k < N - 1; k++)$ {
  $l = a_{k+1,k}/a_{k,k}$;
  $a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$
  $b_{k+1} = b_{k+1} - l \cdot b_k$;
  } // $(N-1) \cdot 5$ fp operations
  // Backward substitution
  $x_{N-1} = b_{N-1}/a_{N-1,N-1}$;
  **for** $(k = N - 2; k \geq 0; k--)$ {
  $x_k = (b_k - a_{k,k+1} \cdot x_k + 1)/a_{k,k}$;
  } // $(N-1)3 + 1$ fp operations

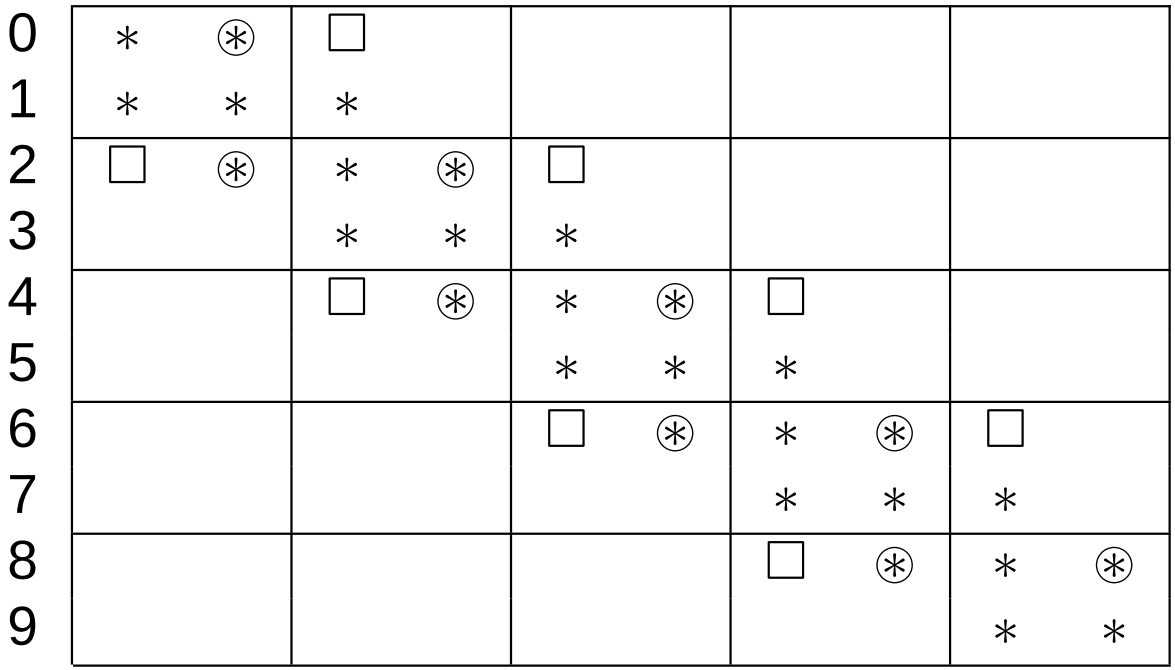
- The sequential complexity amounts to

$$T_S = 8Nt_f$$

  Obviously the algorithm is strictly sequential!

# Cyclic Reduction

- Consider a tridiagonal matrix with $N = 2M$ ($N$ even).
- *Idea:* Eliminate in each *even* row $k$ the elements $a_{k-1,k}$ and $a_{k+1,k}$ with the help of the odd rows above resp. beneath.
- Each even row is therefore only coupled with the second previous and second next; since these are just even, the dimension has been reduced to $M = N/2$.
- The remaining system is again tridiagonal, and the idea can be applied recursively.




⊛ are removed, thereby fill-in (□) is generated.

# Cyclic Reduction

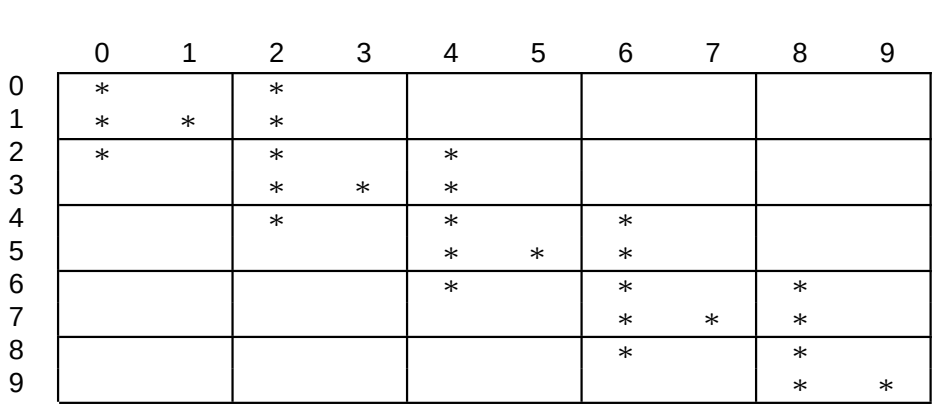
- Algorithm of cyclic reduction

  // Elimination of all odd unknowns in even rows:
  **for** $(k = 1; k < N; k \mathrel{+}= 2)$
  {
  // row $k$ modifies row $k-1$
  $l = a_{k-1,k} / a_{k,k}$;
  $a_{k-1,k-1} = a_{k-1,k-1} - l \cdot a_{k,k-1}$;
  $a_{k-1,k+1} = -l \cdot a_{k,k+1}$; // fill-in
  $b_{k-1} = b_{k-1} - l \cdot b_k$;
  } // $\frac{N}{2} 6 t_f$
  **for** $(k = 2; k < N; k \mathrel{+}= 2)$;
  {
  // row $k-1$ modifies row $k$
  $l = a_{k,k-1} / a_{k-1,k-1}$;
  $a_{k,k-2} = l \cdot a_{k-1,k-2}$; // fill-in
  $a_{k,k} = l \cdot a_{k-1,k}$;
  } // $\frac{N}{2} 3 t_f$

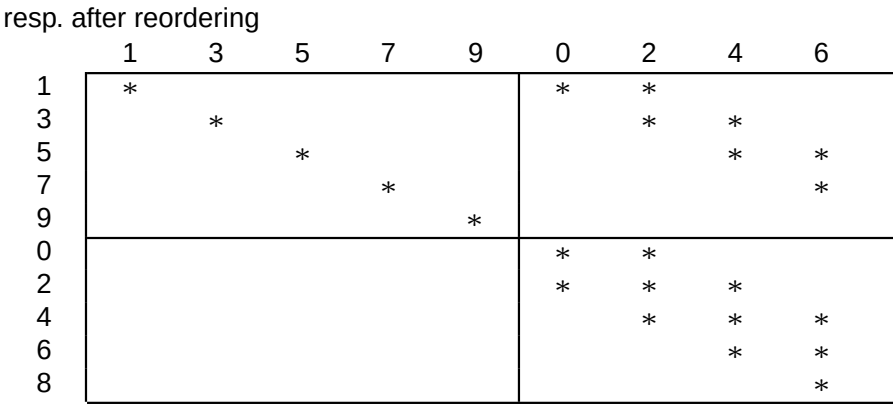
- All traversals of both loops can be processed in parallel (if we assume a machine with shared memory)!

# Cyclic Reduction

- Result of this elimination is



resp. after reordering



- Are the $x_{2k}$, $k = 0, \ldots, M-1$, calculated, then the odd unknowns can be calculated with

  **for** $(k = 1; k < N-1; k \mathrel{+}= 2)$

  $\quad x_k = (b_k - a_{k,k-1} \cdot x_{k-1} - a_{k,k+1} \cdot x_{k+1})/a_{k,k}$;

  // $\frac{N}{2} 5 t_f$

  $x_{N-1} = (b_{N-1} - a_{N-1,N-2} \cdot x_{N-2})/a_{N-1,N-1}$;

  completely in *parallel*.

# Cyclic Reduction

- The *sequential* effort for the cyclic reduction is therefore

$$\begin{aligned} T_S(N) &= (6+3+5)t_f \left( \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots + 1 \right) \\ &= 14Nt_f \end{aligned}$$

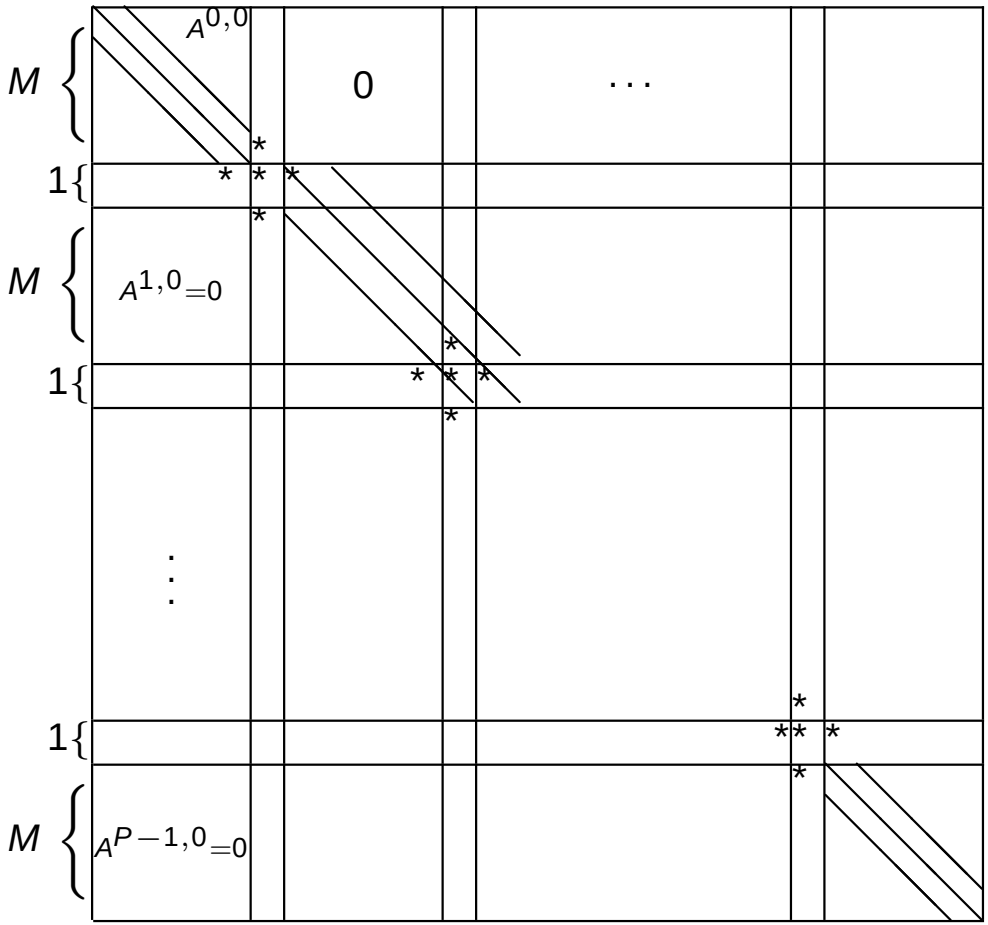
- This is nearly twice as much as the optimal sequential algorithm needs. Therefore the cyclic reduction can be parallelised. The maximal achievable efficiency is however

$$E_{\max} = \frac{8}{14} \approx 0.53,$$

where we have assumed, that all operations are executed optimally parallel and communication is for free (backward substitution needs only $\frac{N}{2}$ processors!). We have not taken into account that cyclic reduction neccesitates more index calculation!

# Domain Decomposition Methods

- Another approach can be in principle be extended to more general problem formulations: domain decomposition methods
- Be $P$ the number of processors and $N = MP + P - 1$ for a $M \in \mathbf{N}$. We subdivide then the $N \times N$ matrix $A$ in $P$ blocks à $M$ rows with a single row between the blocks:



# Domain Decomposition Methods

- The unknowns between the blocks form the *interface.* Each block is at most coupled to two interface unknowns.
- Now we sort rows and columns of the matrix such that the interface unknowns are moved to the end. This results in the following shape:

$$\left(\begin{array}{cccc|c} A^{0,0} & & & & A^{0,I} \\ & A^{1,1} & & & A^{1,I} \\ & & \ddots & & \vdots \\ & & & A^{P-1,P-1} & A^{P-1,I} \\ \hline A^{I,0} & A^{I,1} & \dots & A^{I,P-1} & A^{I,I} \end{array}\right),$$

where $A^{p,p}$ are the $M \times M$ tridiagonal partial matrices from A and $A^{I,I}$ is a $P-1 \times P-1$ diagonal matrix. The $A^{p,I}$ have the general form

$$A^{p,I} = \left(\begin{array}{c|c|c} \dots & \begin{array}{c} * \\ \\ * \end{array} & \dots \end{array}\right).$$

# Domain Decomposition Methods

- Idea: Eliminate blocks $A^{I,*}$ in the block representation. Thereby $A^{I,I}$ is modified, more exact the following block representation is created:

$$\left(\begin{array}{cccc|c} A^{0,0} & & & & A^{0,I} \\ & A^{1,1} & & & A^{1,I} \\ & & \ddots & & \vdots \\ & & & A^{P-1,P-1} & A^{P-1,I} \\ \hline 0 & 0 & \dots & 0 & S \end{array}\right),$$

mit $S = A^{I,I} - \sum_{p=0}^{P-1} A^{I,p} \left(A^{p,p}\right)^{-1} A^{p,I}$.

- $S$ is in general denoted as „Schurcomplement“. All eliminations in $\sum_{p=0}^{P-1}$ can be executed *parallel*.
- After solution of a system $Sy = d$ for the interface unknowns the inner unknowns can again be calculated in parallel.
- $S$ has dimension $P-1 \times P-1$ and is itself sparse, as we can see soon.

# Execution of the Plan

1. Transform $A^{p,p}$ to diagonal shape.
($a_{i,j}$ denotes $(A^{p,p})_{i,j}$, if not stated otherwise):

$\forall p$ parallel

**for** $(k = 0; k < M - 1; k++)$ // lower diagonal

{

$l = a_{k+1,k} / a_{k,k}$;

$a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$;

**if** $(p > 0)$ $a^{p,l}_{k+1,p-1} = a^{p,l}_{k+1,p-1} - l \cdot a_{k,p-1}$;

// fill-in left boundary

$b^p_{k+1} = b^p_{k+1} - l \cdot b^p_k$;

} // $(M-1)7t_f$

**for** $(k = M - 1; k > 0; k--)$ // upper diagonal

{

$l = a_{k-1,k} / a_{k,k}$;

$b^p_{k-1} = b^p_{k-1} - l \cdot b^p_k$;

**if** $(p > 0)$ $a^{p,l}_{k-1,p-1} = a^{p,l} - l \cdot a^{p,l}_{k,p-1}$; // left boundary

**if** $(p < P - 1)$ $a^{p,l}_{k-1,p} = a^{p,l}_{k-1,p} - l \cdot a^{p,l}_{k,p}$; // right boundary, fill-in

} // $(M-1)7t_f$

# Execution of the Plan

2. Eliminate in $A^{l,*}$.

$\forall p$ parallel:

**if** $(p > 0)$

{

$l = a^{l,p}_{p-1,0} / a^{p,p}_{0,0}$;

$a^{l,l}_{p-1,p-1} = a^{l,l}_{p-1,p-1} - l \cdot a^{p,l}_{0,p-1}$; // diagonal in $S$

**if** $(p < P-1)$ $a^{l,l}_{p-1,p} = a^{l,l}_{p-1,p} - l \cdot a^{p,l}_{0,p}$; // upper diag. in $S$, fill-in

$b^{l}_{p-1} = b^{l}_{p-1} - l \cdot b^{p}_{0}$;

}

**if** $(p < P-1)$

{ // right boundary

$l = a^{l,p}_{p,M-1} / a^{p,p}_{M-1,M-1}$;

**if** $(p > 0)$ $a^{l,l}_{p,p-1} = a^{l,l}_{p,p-1} - l \cdot a^{p,l}_{M-1,p-1}$;

// fill-in lower diag of $S$

$a^{l,l}_{p,p} = a^{l,l}_{p,p} - l \cdot a^{p,l}_{M-1,p}$;

$b^{l}_{p} = b^{l}_{p} - l \cdot b^{p}_{M-1}$;

}

# Execution of the Plan

3. Solve Schurcomplement.
$S$ is *tridiagonal* with dimension $P-1 \times P-1$. Assume that $M \gg P$ and solve sequential. $\longrightarrow 8Pt_f$ effort.
4. Calculate inner unknowns.
Here, only one diagonal matrix has to be solved per processor.

$\forall p$ parallel:

**for** $(k = 0; k < M-1; k++)$

$x_k^p = (b_k^p - a_{k,p-1}^{p,l} \cdot x_{p-1}^l - a_{k,p}^{p,l} \cdot x_p^l)/a_{k,k}^{p,p};$

// $M5t_f$

# Analysis

- Total effort parallel:

$$
\begin{aligned}
T_P(N,P) &= 14Mt_f + O(1)t_f + 8Pt_f + 5Mt_f = \\
&= 19Mt_f + 8Pt_f
\end{aligned}
$$

(without communication!)

$$
\begin{aligned}
E_{\max} &= \frac{8(MP+P-1)t_f}{(19Mt_f + 8Pt_f)P} \approx \\
&\underbrace{\approx}_{\text{für } P \ll M} \frac{1}{\frac{19}{8} + \frac{P}{M}} \leq \frac{8}{19} = 0.42
\end{aligned}
$$

- The algorithm needs additional memory for the fill-in. Cyclic reduction works with overwriting of old entries.

# $LU$ Decomposition of Sparse Matrices

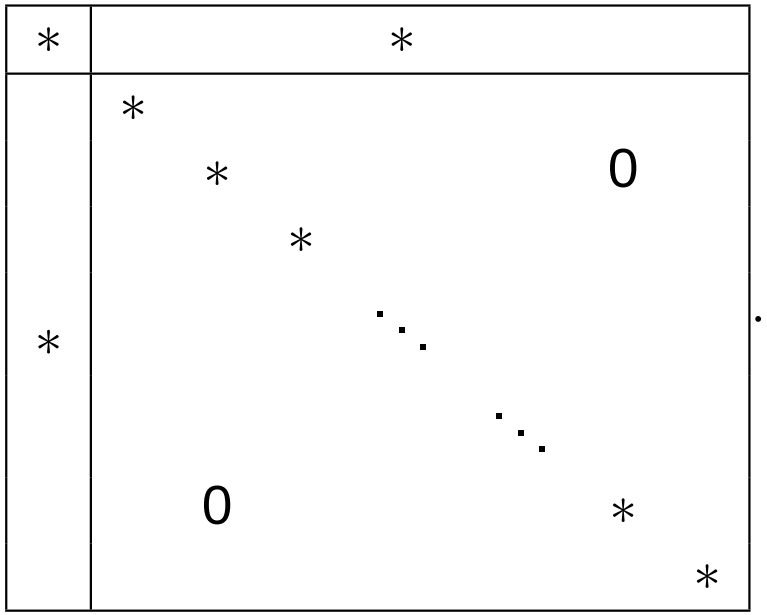
What is a sparse matrix

- In general one speaks of a sparse matrix, if it has in (nearly) each row only a constant number of non-zero elements.
- If $A \in \mathcal{R}^{N \times N}$, then $A$ has only $O(N)$ instead of $N^2$ entries.
- For large enough $N$ it is then advantegeous regarding computing time and memory not to process resp. to store this large number of zeros.

# *LU* Decomposition of Sparse Matrices

## Fill-in

- While *LU* decomposition elements, that initially have been zero, can get non-zero during the elimination process.
- On speaks then of „*Fill-in*“.
- This heavily depends on the structure of the matrix. As a extreme example consider the „arrow matrix“

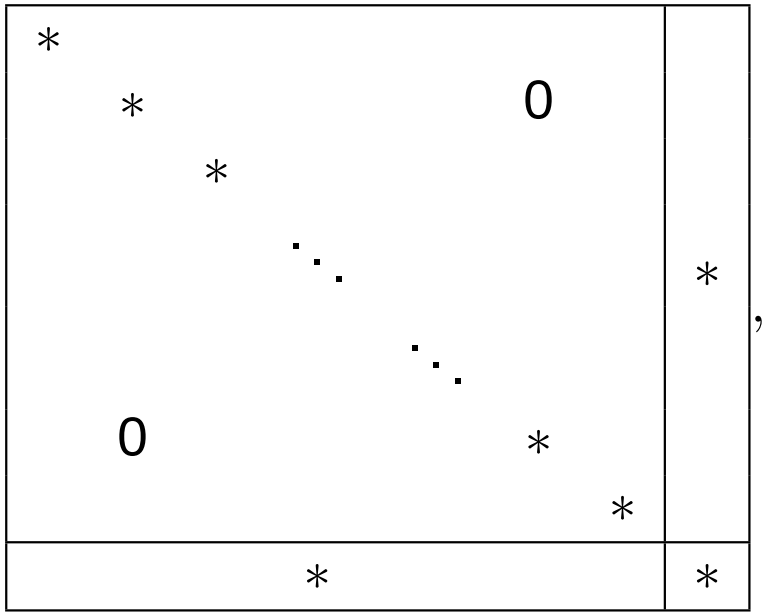



- During elimination in the natural sequence (without pivoting) the whole matrix is „filled-in“.

# *LU* Decomposition of Sparse Matrices

Reordering of the matrix

- If we rearrange the matrix by row and column permutations to the form




- Obviously no fill-in is produced.
- An important point in the *LU* decomposition of sparse matrices is to find a matrix ordering such that the fill-in is minimised.
- Reordering is strongly coupled to pivoting.

# Pivoting

- If the matrix $A$ is symmetric positive definite (SPD), then the $LU$ factorisation is always numerically stable, and *no* pivotng is necessary.

$\rightarrow$ The matrix can thus be reordered in advance, that the fill-in gets small.

- For a general, invertible matrix one will need to use pivoting.

$\rightarrow$ Then during elimination a compromise between numerical stability and fill-in has to be found dynamically.

- Therefore nearly all codes restrict to the symmetric positive case and determine an elimination sequence that minimizes the fill-in in advance.
- The exact solution of the minimization problem is $\mathcal{NP}$ complete.

$\rightarrow$ One therefore uses heuristical methods.

# Graph of a Matrix

Matrix graph

- In the symmetric positive case fill-in can be investigated purely by the zero structure of the matrix.
- For an arbitrary, now not necessarily symmetric $A \in \mathcal{R}^{N \times N}$ we define an undirected graph $G(A) = (V_A, E_A)$ by

$$\begin{aligned} V_A &= \{0, \ldots, N-1\} \\ (i,j) \in E_A &\iff a_{ij} \neq 0 \vee a_{ji} \neq 0. \end{aligned}$$

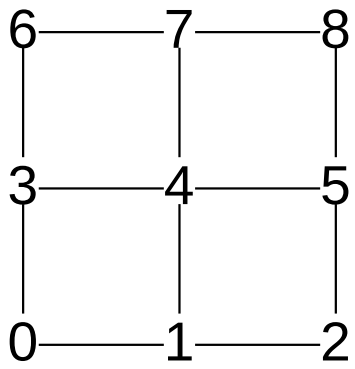
- This graph describes the direct dependencies of the unknowns beneath each other.

# Graph of a Matrix

Example:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | ∗ | ∗ | | ∗ | | | | | |
| 1 | ∗ | ∗ | ∗ | | ∗ | | | | |
| 2 | | ∗ | ∗ | | | ∗ | | | |
| 3 | ∗ | | | ∗ | ∗ | | ∗ | | |
| 4 | | ∗ | | ∗ | ∗ | ∗ | | ∗ | |
| 5 | | | ∗ | | ∗ | ∗ | | | ∗ |
| 6 | | | | ∗ | | | ∗ | ∗ | |
| 7 | | | | | ∗ | | ∗ | ∗ | ∗ |
| 8 | | | | | | ∗ | | ∗ | ∗ |

$A$

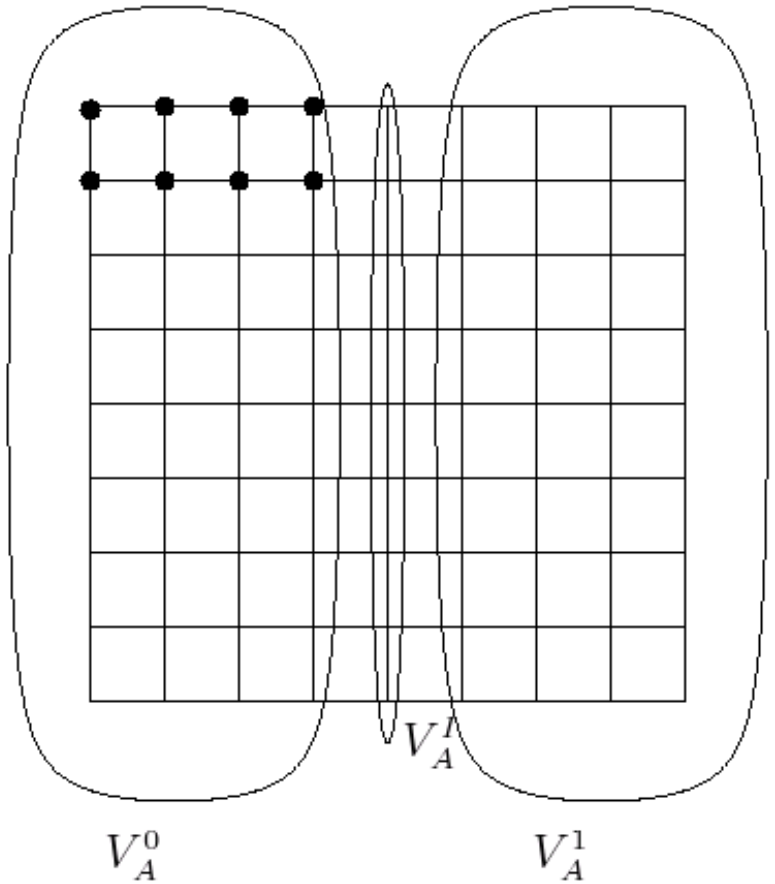



$G(A)$

# Matrix Ordering Strategies

## Nested Disection

- An important method to order SPD matrices for the purpose of fill-in minimisation is the „*nested disection*“.
- Example:
  The graph $G(A)$ of the matrix $A$ be a quadratic grid



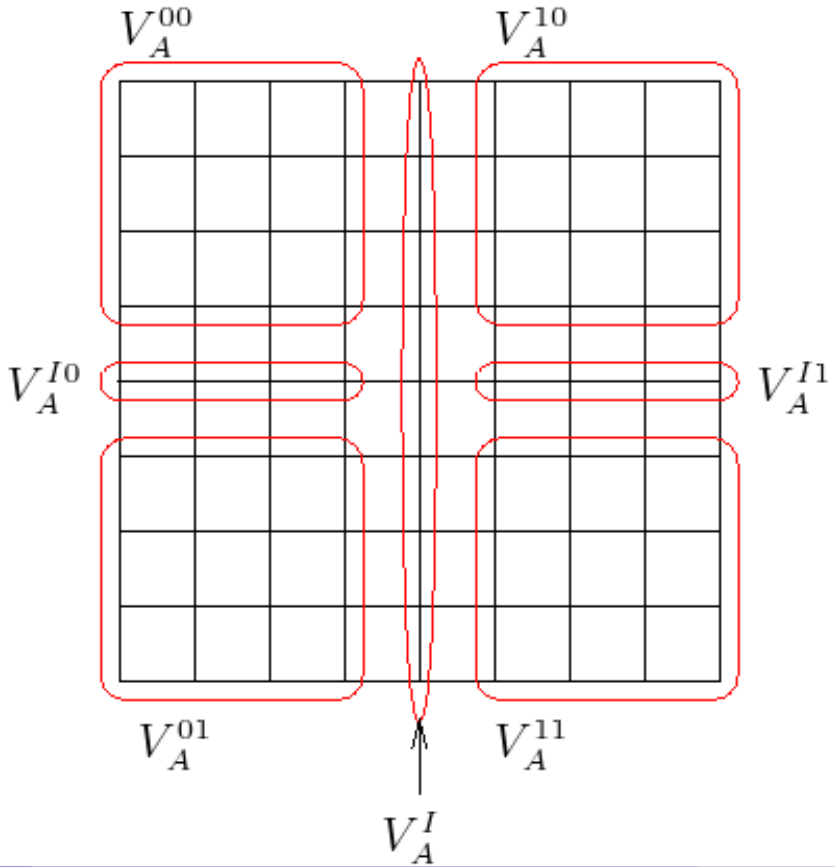
# Matrix Ordering Strategies

Now we divide the node set $V_A$ in three parts: $V_A^0$, $V_A^1$ and $V_A^I$, such that

- $V_A^0$ and $V_A^1$ are as large as possible,
- $V_A^I$ is a separator, this means when $V_A^I$ is removed from the graph this is split into two parts. Thus there is *no* $(i,j) \in E_A$, such that $i \in V_A^0$ and $j \in V_A^1$.
- $V_A^I$ is as small as possible,
- The figure shows a possibility for such a partitioning.

# Matrix Ordering Strategies

- Now one reorders the rows and columns such that first the indices $V_A^0$ are present, then $V_A^1$ and finally $V_A^I$.
- Then we apply the method *recursively* to the partial graphs with the node sets $V_A^0$ and $V_A^1$.
- The method stops, if the graphs has reached a predefined size.
- Example graph after two steps:



# Matrix Ordering Strategies

| | $V_A^{00}$ | $V_A^{01}$ | $V_A^{l0}$ | $V_A^{10}$ | $V_A^{11}$ | $V_A^{l1}$ | $V_A^{l}$ |
|---|---|---|---|---|---|---|---|
| $V_A^{00}$ | $*$ | 0 | $*$ | | | | $*$ |
| $V_A^{01}$ | 0 | $*$ | $*$ | | | | $*$ |
| $V_A^{l0}$ | $*$ | $*$ | $*$ | | | | $*$ |
| $V_A^{10}$ | | | | $*$ | | $*$ | $*$ |
| $V_A^{11}$ | | | | | $*$ | $*$ | $*$ |
| $V_A^{l1}$ | | | | $*$ | $*$ | $*$ | $*$ |
| $V_A^{l}$ | $*$ | $*$ | $*$ | $*$ | $*$ | $*$ | $*$ |

$A$, reordered

Complexity of the nested disection

- For the example above the nested disection numbering leads to a complexity of $O(N^{3/2})$ for the $LU$ decomposition.
- For comparisoon one needs with lexicographic numbering (band matrix) $O(N^2)$ operations.

# Data Structures for Sparse Matrices

- There are a series of data structures for storage of sparse matrices.
- Goal is an efficient implementation of algorithms.
- Thus one has to watch out for data locality and as few overhead as possible by additional index calculation.
- An often used data structure is *„compressed row storage“*
- If the $N \times N$ matrix has in total $M$ non-zero elements, then one stores the matrix elements row-wise in an one-dimensional field:

  **double** $a[M]$;
- The management of index information happens via three arrays

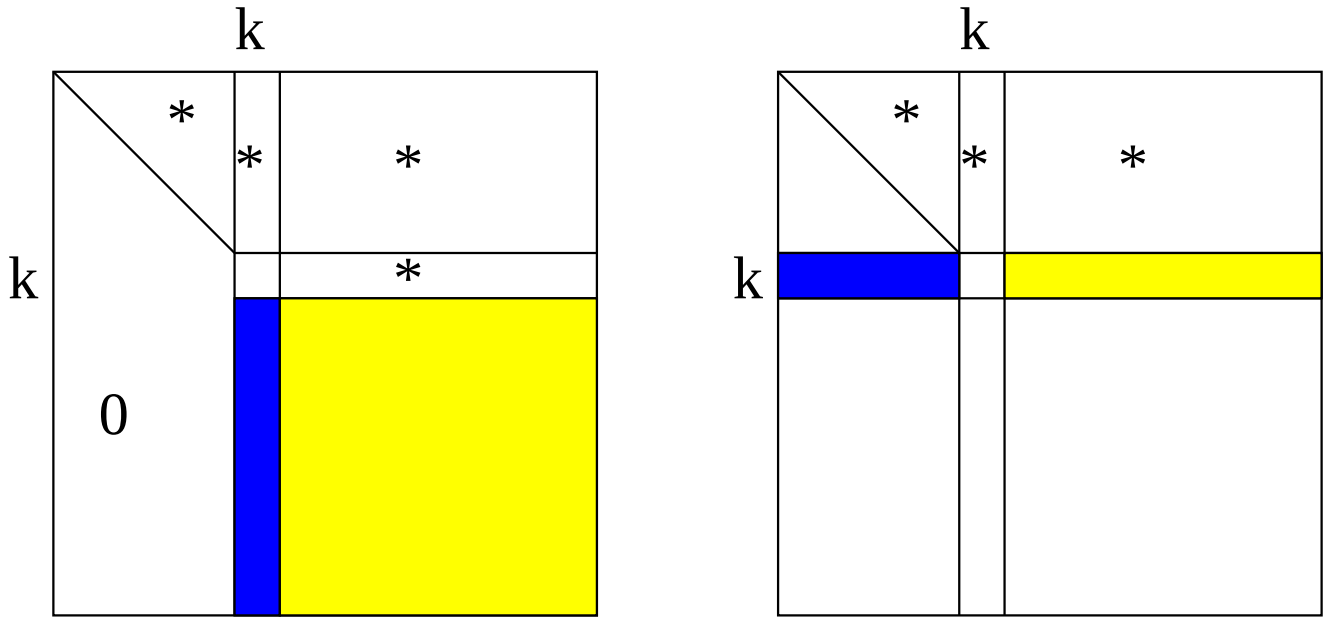
  **int** $s[N], r[N], j[M]$;
- Their meaning shows the realisation of the matrix-vector product $y = Ax$:

```
for (i = 0; i < N; i++) {
    y[i] = 0;
    for (k = r[i]; k < r[i] + s[i]; k++)
        y[i] += a[k] · x[j[k]];
}.
```

- $r$ provides row start, $s$ the row length, and $j$ the column index.

# Elimination Forms

- In the *LU* decomposition of dense matrices we have used the so-called *kij* form of the *LU* decomposition.




- Here in each step $k$ all $a_{ik}$ for $i > k$ are eliminated, which requires a modification of all $a_{ij}$ with $i, j > k$.
- This situation is shown left.
- In the *kji* variant one eliminates in step $k$ all $a_{kj}$ with $j < k$.
- Here the $a_{ki}$ with $i \geq k$ are modified. Wach out, that the $a_{kj}$ have to be eliminated from left to right!
- For the following sequential *LU* decomposition for sparse matrices we are going to start from this *kji* variant.

# Sequential Algorithm

- In the following we assume:
  - The matrix $A$ can be factorized in the given ordering without pivoting. The ordering has been choosen in an appropriate way to minimize fill-in.
  - The data structure stores all elements $a_{ij}$ with $(i,j) \in G(A)$. Because of the definiton of $G(A)$ applies:

$$(i,j) \notin G(A) \;\Rightarrow\; a_{ij} = 0 \wedge a_{ji} = 0.$$

    If $a_{ij} \neq 0$, then in every case also $a_{ji}$ is stored, *also if this is zero.* The matrix does not have to be symmetric.
- The extension of the structure of $A$ happens purely because of the information in $G(A)$. Thus also a $a_{kj}$ is formally eliminated, if $(k,j) \in G(A)$ applies. This can possibly create a fill-in $a_{ki}$, albeit applies $a_{ki} = 0$.
- The now presented algorithm uses the sets $S_k \subset \{0,\ldots,k-1\}$, that contain in step $k$ exactly the column indices, that have to be eliminated.

# Sequential Algorithm

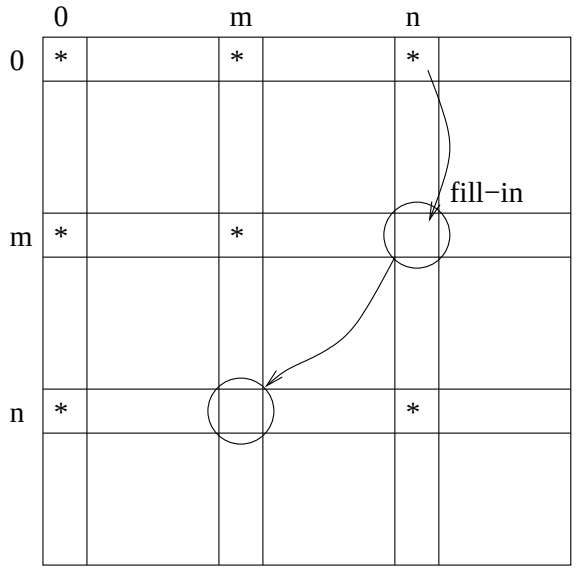
```
for (k = 0; k < N; k++) S_k = ∅;
for (k = 0; k < N; k++)
{
    // 1. extend matrix graph
    for (j ∈ S_k)
    {
        G(A) = G(A) ∪ {(k, j)};
        for (i = k; i < N; i++)
            if ((j, i) ∈ G(A))
                G(A) = G(A) ∪ {(k, i)};
    }

    // 2. Eliminate
    for (j ∈ S_k)
    {                                   // eliminate a_{k,j}
        a_{k,j} = a_{k,j} / a_{j,j};    // L factor
        for (i = j + 1; i < N; i++)
            a_{k,i} = a_{k,i} − a_{k,j} · a_{j,i};
    }

    // 3. update S_i for i > k, holds because of symmetry of E_A
    for (i = k + 1; i < N; i++)
        if ((k, i) ∈ G(A))              // ⇒ (i, k) ∈ G(A)
            S_i = S_i ∪ {k};
}
```

# Sequential Algorithm

- We consider in an example, how the $S_k$ are mapped.




- At start $G(A)$ shall contain the elements

$$G(A) = \{(0,0),(m,m),(n,n),(0,n),(n,0),(0,m),(m,0)\}$$

- For $k = 0$ is set in step 1 $S_m = \{0\}$ and $S_n = \{0\}$.
- Now, next for $k = m$ the $a_{m,0}$ is eliminated.
- This generates the fill-in $(m,n)$, which again has in step 3 for $k = m$ as consequence the instruction $S_n = s_n \cup \{m\}$.
- Thus applies at start of traversal $k = n$ correctly $S_n = \{0,m\}$ and in step 1 the fill-in $a_{n,m}$ is correctly generated, *before* the elimination of $a_{n,\mathbf{0}}$ is performed. This is enabled because of the symmetry of $G_A$.

# Parallelisation

*LU* decomposition of sparse matrices has the following possibilites for a parallelisation:

- *coarse granularity:* In all $2^d$ partial sets of indices, that has been created by nested disection of depth $d$, one can start in parallel with the elimination. First for the indices, that correspond to the separators, communication is necessary.
- *medium granularity:* Single rows can be processed in parallel, as soon as the according pivot row is locally available. This corresponds to the parallelisation of the dense *LU* decomposition.
- *fine granularity:* modifications of an individual row can be processed in parallel, as soon as the pivot line and the multiplicator are available. This is used for the two-dimensional data distribution in the dense case.

# Parallelisation: Case $N = P$

## Program ($LU$ decomposition for sparse matrices and $N = P$)

```
parallel sparse-lu-1
{
    const int N = ...;
    process Π[int k ∈ {0, ..., N − 1}]
    {// (only pseudo code!)
        S = ∅;                                          // 1. set S
        for (j = 0; j < k; j++)
            if ((k, j) ∈ G(A)) S = S ∪ {j};                 // the start
        for (j = 0; j < k; j++)                         // 2. process row k
            if (j ∈ S_k)
            {
                recv(Π_j, r);                           // wait, until Π_j sends the row j
                // extend pattern
                for (i = j + 1; i < N; i++)
                {
                    if (i < k ∧ (j, i) ∈ G(A))          // info is in r
                        S = S ∪ {i};                    // processor i will send row i
                    if ((j, i) ∈ G(A) ∪ {(k, i)})       // info is in r
                        G(A) = G(A) ∪ {(k, i)};
                }
                // eliminate a_{k,j} = a_{k,j} / a_{j,j};   // info is in r
                for (i = j + 1; i < N; i++)
                    a_{k,i} = a_{k,i} − a_{k,j} · a_{j,i};  // info is in r
            }
        for (i = k + 1; i < N; i++)                     // 3. send away
            if ((k, i) ∈ G(A))                              // local info!
                send row k at Π_i;                      // k knowns, that i needs row k.
    }
}
```

# Parallelisation for $N \gg P$

For case $N \gg P$

- Each processor has now a complete block of rows. Three things have to happen:
- *Receive pivot rows* of other processors and store them in the set $R$.
- *Send finished rows* from the send buffer $S$ to the target processors.
- *Local elimination*
  - choose a row $j$ from the receive buffer $R$.
  - eliminate with this row locally all possible $a_{k,j}$.
  - if a row is going to be ready, put it into the send buffer (there may be several target processors).