MATRIX ALGORITHMS

- Linear algebra software and the BLAS
- Matrix-vector & matrix-matrix products
- Solving dense linear systems
- Tridiagonal linear systems

Background & notation: matrices

- An $n \times m$ matrix A is an $n \times m$ array of values in $\mathbb{R}$ or $\mathbb{C}$ (complex matrices)
- Notation: $A \in \mathbb{R}^{n \times m}$ or $A \in \mathbb{C}^{n \times m}$
- Operations with matrices: addition, multiplication, scaling.
- We will consider matrix-vector products, matrix-matrix products, and:
- Linear systems of equations.

Evolution of linear algebra software

- First numerical libraries appeared in 1970s: EISPACK (eigenvalue problems) and LINPACK (mostly linear systems)
- Need for (performance-) optimized functions for basic linear algebra appeared with first supercomputers: BLAS
- Level 1 BLAS (BLAS1) : vector operations (dot products, sums, combination of vectors)...
- Then BLAS2 and BLAS3 followed (in the late 1980s).

Level 2 BLAS (BLAS2) 2 nested loops. Examples:

Rootname	Operation	Matrix type
–GEMV	$y := \alpha A^* x + \beta y$	General A
–TRMV	$y = Tx$	General triangular T
–TRSV	$y = (T^*)^{-1} x$	General triangular

$*$ = one of : no-operation ($A^* = A$), transpose ($A^* = A^T$), or conjugate transpose ($A^* = A^H$)

Level 3 BLAS (BLAS3) 3 nested loops. Examples:

–GEMM	$C := \alpha A^* B^* + \beta C$	General case
–SYR2K	$C := \alpha B A^T + \alpha A^T B + \beta C$ or	Symmetric A
	$C := \alpha A^T B + \alpha B A^T + \beta C$	

BLAS and memory references

Efficiency of programs on high performance computers require to account for memory traffic

Rule: For each memory reference, perform as many floating point operations as possible.

BLAS1: in SAXPY (for example) two vectors are loaded and one is stored: $3n$ memory references for $2n$ flops.

BLAS2: in SGEMV (matrix-vector product) a matrix and the vector are loaded and a vector is stored. $\sim n^2$ memory references for $\sim 2n^2$ flops. (assume a square matrix).

BLAS3: In SGEMM, 3 matrices are loaded and one is stored: $4n^2$ memory references for $2n^3$ flops (assume square matrices).

Matrix-vector products (MATVECS)

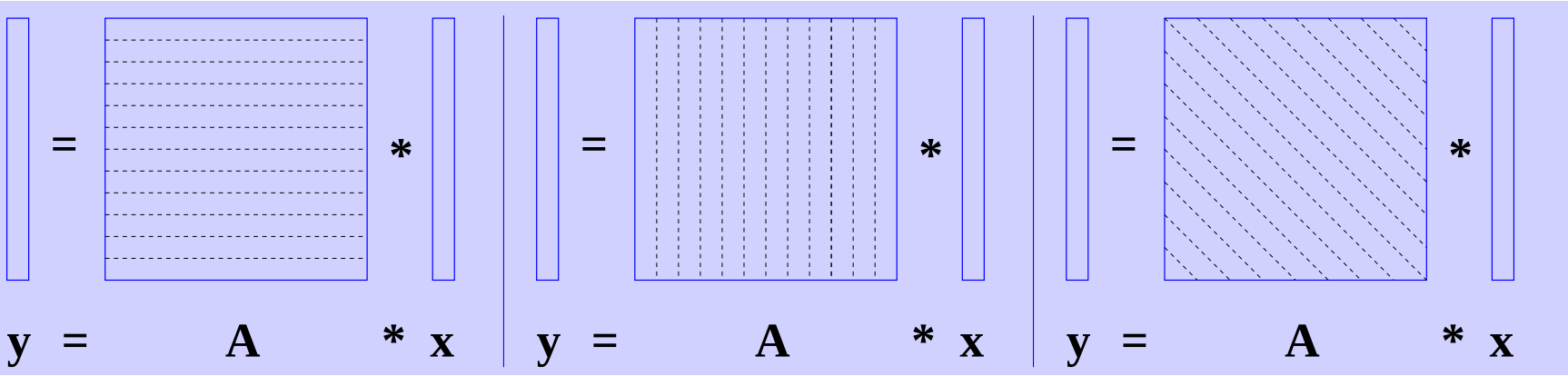
Problem: To compute the product $y = Ax$ where A is a matrix, and x a vector.

➤ The result y should end up with the same mapping to processors as x [think of problem of doing many successive matvecs].

➤ Many different situations:

- Dense case vs sparse case
- Vector processing
- Product with one vector versus several vectors at once.

3 Ways to the matvec



Product can be done by rows, by columns, or by diagonals.

➤ The row-oriented algorithm appears natural – not the best if saxpy's can be done fast (vector processing)

➤ Product by diagonals is useful mainly in the sparse case (e.g., banded matrices)

➤ We will consider two different sequential "base" algorithms.

Algorithm 1

```
for (i=0;  i<n ;  i++) {
     y[i] = 0;
     for (j=0; j<n; j++)
         y[i] += A[i][j]*x[j] ;
    }
```

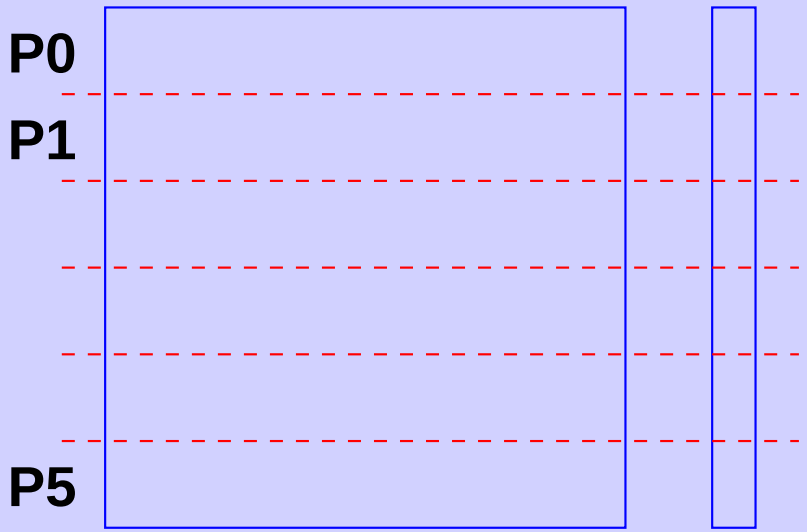
- Uses n dot-products

Algorithm 2

```
 for (j=0; j<n; j++)
    y[j] = 0.0;
 for (j=0; j<n ; j++) {
    for (i=0;  i<n;  i++)
       y[i] += A[i][j]*x[j];
 }
```

- Main operations used SAXPYs: $y \leftarrow y + a * x$

Row-oriented mapping



Algorithm:

1. Do an all-to-all broadcast of the subvectors $\boldsymbol{x}_i$.
2. Compute dot products $\boldsymbol{y}(\boldsymbol{i}) = \boldsymbol{A}(\boldsymbol{i}, :)' \cdot \boldsymbol{x}(:)$ for $\boldsymbol{i}$ in 'myPE'.

✍1 What MPI operation would you use for (1) ?

Timing model

1. Time for communication

$$t_s \log(p) + t_w \times \frac{n}{p} \times (p-1) \approx t_s \log(p) + t_w n$$

2. Time for the local multiplications and additions:

$$2\frac{n^2}{p}$$

3. Total

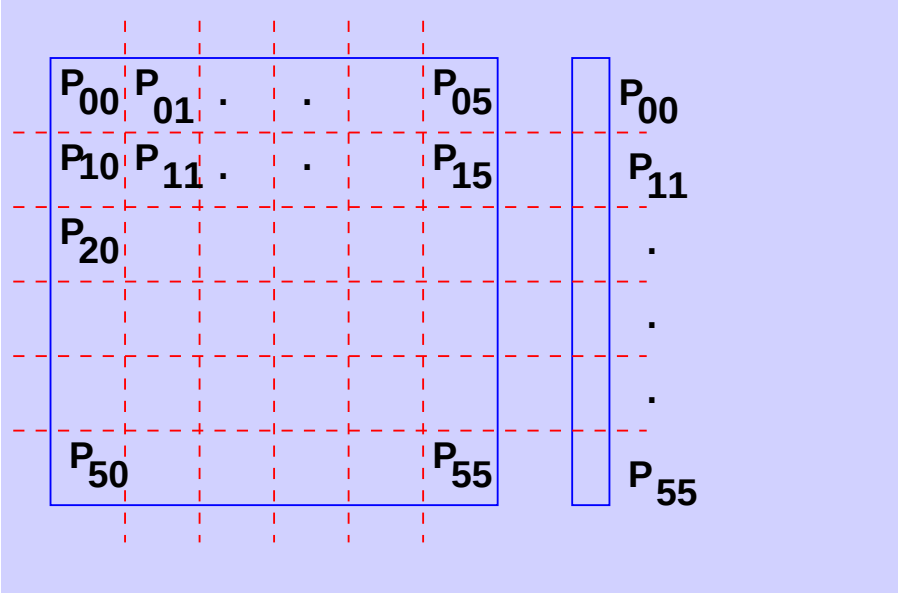
$$T(p) = 2\frac{n^2}{p} + t_s \log(p) + t_w n$$

Questions:

1. Cost-optimal?
2. Standard speed-up & efficiency?
3. What is $T_o(p)$?
4. Scaled speed-up? [time-based]

Matrix- vector products: 2-D mapping

Example of a 2-D mapping for a 6×6 grid



Algorithm: [2-D mesh of processors]

1. Do a broadcast of x_i in each vertical direction of the processor mesh
2. Compute the local products of $A(i,j)$ with $x(j)$
3. Add results in horizontal direction

✍3 Cost model assuming: (1) Hypercube topology - (2) Simple broadcast (no scatter first) (3) Similar simple reduction for the sums.

✍4 Speed-up, efficiency? Speed-up & Efficiency for the case when $p = n$?

✍5 Scaled speed-up?

Matrix-Matrix product

Basic sequential algorithm has three loops

Algorithm 3:

```
for (i=0; i<n ; i++){
    for (j=0; j<n; j++) {
     C[i][j] = 0;
     for (k=0; k<n; k++) ;
       C[i][j] += A[i][k]*B[k][j];
    }
}
```

➤ Important observation: product can be done by blocks:

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$
$$= \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix}$$

➤ In particular, in Algorithm 3 all the $X(i,j)$'s (X one of A, B, or C) can be the individual entries of the matrices involved, or blocks (submatrices), for block algorithms.

```
For (k=0;   k<n;   k++)
    C[i][j] += A[i][k]*B[k][j] // products/sums
                               // of submatrices
```

Consequences

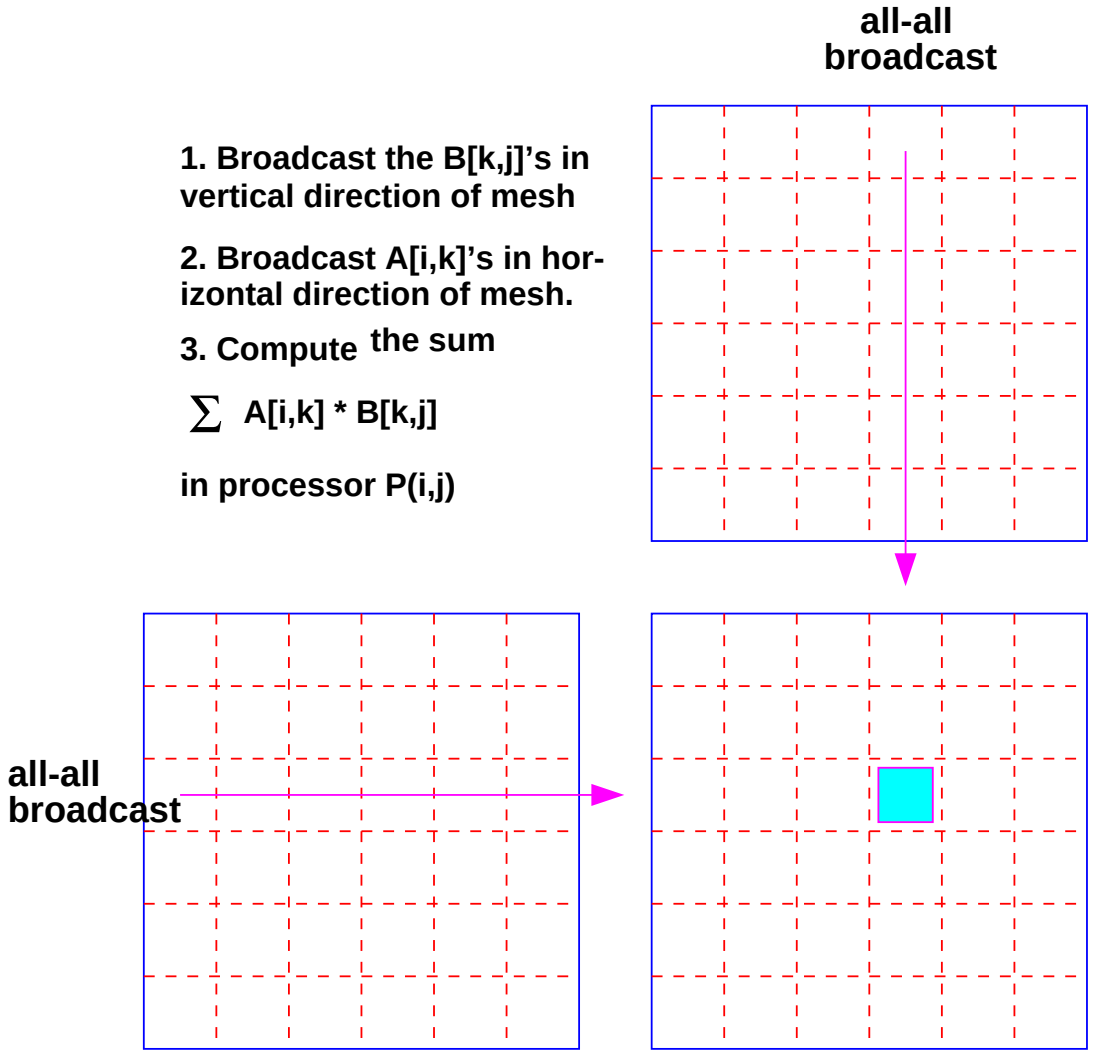
1. Possibility of proceeding recursively

2. Observation can be exploited to devise algorithms with better utilization of cache

3. Can play with block-size to optimize performance [See Atlas package]

Broadcast Algorithm

Assume each $A(i,j)$ and $B(i,j)$ is assigned to processor $P(i,j)$ in a logical mesh of processors. Note: P_{ij} must store all $A_{i,*}$ blocks and all $B_{*,j}$ blocks.

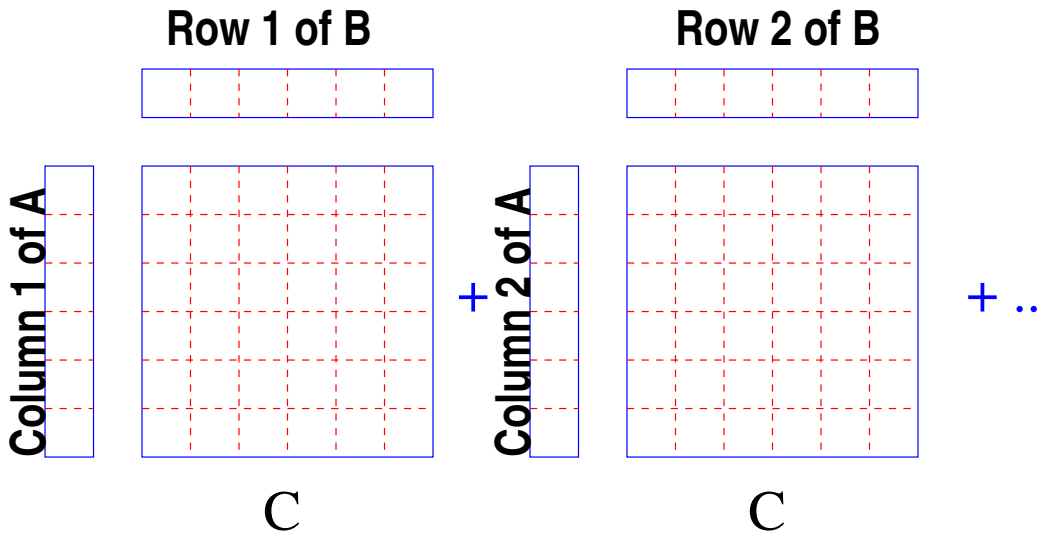


A better implementation:

Use the 'rank-one' updates version of matrix products.

$$C = \sum_{k=1}^{n} A(:,k).B(k,:)$$

- Rank-one updates in block form
- No issue with memory requirement



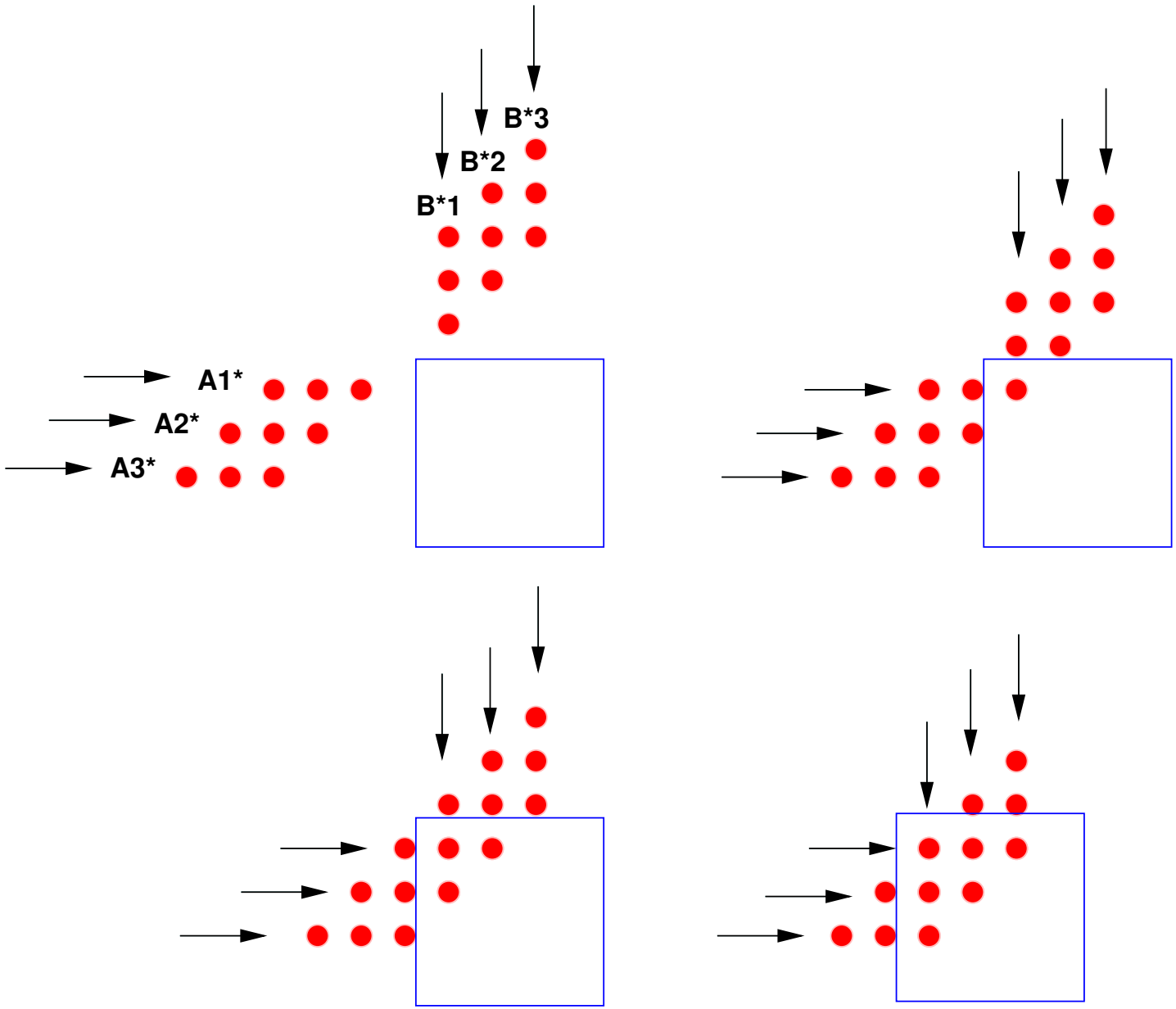
➤ "Row" and "Column" mean block-row and block-column

➤ For k=1:n do : broadcast column $\longrightarrow$ and row $\downarrow$.
Then update: $C(i,j) := C(i,j) + A(i,k) * B(k,j)$.

Cannon's algorithm

➤ Many algorithms were developed with a "systolic" viewpoint: using SIMD simple arithmetic and communication operations.

➤ Algorithm shown next is an example. It computes the product of 2 matrices. Each time cycle, data is moved as noted and product is performed and added to current c_{ij}.


B*3
B*2
B*1
A1*
A2*
A3*

Another organization (no time delays built) : Cannon's algorithm

A11 B11	A12 B22	A13 B33	A14 B44
A22 B21	A23 B32	A24 B43	A21 B14
A33 B31	A34 B42	A31 B13	A32 B24
A44 B41	A41 B12	A42 B23	A43 B34

(a) Initial alignment of submatrices

A12 B21	A13 B32	A14 B43	A11 B14
A23 B31	A24 B42	A21 B13	A22 B24
A34 B41	A31 B12	A32 B23	A33 B34
A41 B11	A42 B22	A43 B33	A44 B44

(b) After the first shift

A13 B31	A14 B41	A11 B13	A12 B24
A24 B41	A21 B12	A22 B23	A23 B34
A31 B11	A32 B22	A33 B33	A34 B44
A42 B21	A43 B32	A44 B43	A41 B14

(c) After the second shift

A14 B41	A11 B12	A12 B23	A13 B34
A21 B11	A22 B22	A23 B33	A24 B44
A32 B21	A33 B32	A34 B43	A31 B14
A43 B31	A44 B42	A41 B13	A42 B24

(d) After the third shift

Linear Systems of equations: Background

The Problem: Given $A =$ an $n \times n$ matrix, $b \in \mathbb{R}^n$

Find $x \in \mathbb{R}^n$ s.t. :

$$Ax = b$$

- x is the **unknown** vector, b the **right-hand side**, and A is the **coefficient matrix**
- Important problem: arises in many engineering and scientific disciplines
- Recall under what conditions the system has a unique solution.
- Recall how to solve a triangular system

Example:

$$\begin{cases} 2x_1 + 4x_2 + 4x_3 = 6 \\ x_1 + 5x_2 + 6x_3 = 4 \\ x_1 + 3x_2 + x_3 = 8 \end{cases}$$

➤ Or, in matrix notation:

$$\underbrace{\begin{pmatrix} 2 & 4 & 4 \\ 1 & 5 & 6 \\ 1 & 3 & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 6 \\ 4 \\ 8 \end{pmatrix}}_{b}$$

$$Ax = b$$

 Solution of above system ?

Triangular linear systems

Example:

$$\begin{pmatrix} 2 & 4 & 4 \\ 0 & 5 & -2 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}$$

- One equation can be trivially solved: the last one. $x_3 = 2$
- x_3 is known we can now solve the 2nd equation:

$$5x_2 - 2x_3 = 1 \;\rightarrow\; 5x_2 - 2 \times 2 = 1 \;\rightarrow\; x_2 = 1$$

- Finally x_1 can be determined similarly:

$$2x_1 + 4x_2 + 4x_3 = 2 \rightarrow \;\ldots\; \rightarrow\; x_1 = -5$$

Back-Substitution algorithm

```
for (i=n-1; i>=0; i--){
  t  = b[i];
  for  (j=i; j<n; j++)
      t -= a[i,j]*x[j];
   x_i = t / a[i,i] ;
 }
```

- We must require that each $a_{ii} \neq 0$
- Operation count?

➤ Column version of back-subsitution – useful for parallel implementation

Algorithm. Back-Substitution algorithm. Column version

```
for  (j=n-1; j>=0; j--){
    x[j] = b[j]/ a[j,j];
    for (i=0; i<j-1; j++)
         b[i] -= x[j] *a[i,j];
}
```

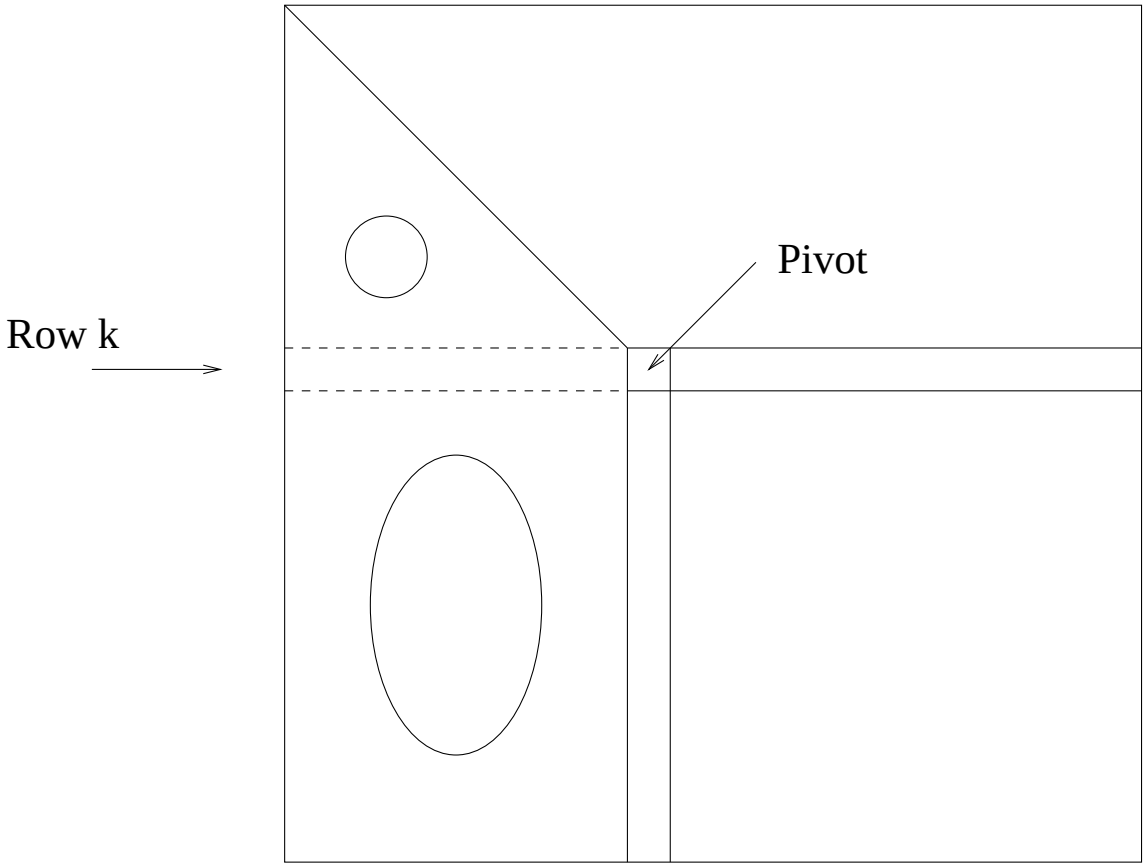
✍7 Justify the above algorithm [Show that it does indeed give the solution]

Principle of the method: We will first transform a linear system into one that is triangular then solve. Main operation: combine rows so that zeros appear in the required locations to make the system triangular.

Gaussian Elimination

1. For k $= 1 : n - 1$ Do:
2. For i $= k + 1 : n$ Do:
3. $piv := a_{ik}/a_{kk}$
4. For j $:= k + 1 : n + 1$ Do :
5. $a_{ij} := a_{ij} - piv * a_{kj}$
6. End
6. End
7. End

A_k =



- Operation count: $\frac{2}{3}n^3$ + low order terms

Solving Dense Linear Systems of equations

- Problem is to solve the linear system of equations:

$$\boldsymbol{Ax = b}$$

where $\boldsymbol{A}$ is a an $\boldsymbol{n \times n}$ matrix, $\boldsymbol{b}$ is the given right-hand side and $\boldsymbol{x}$ is the unknown.

- Method of choice: Gaussian elimination (or variants)

- Can change the order of the loops and obtain a few variants of Gaussian elimination ➤ Can permute the loop variables `i`, `j` and `k` in $\boldsymbol{3! = 6}$ different ways.

```
do ———
    do ———
        do ———
           a(i,j)=a(i,j)-a(i,k)*a(k,j)/a(k,k)
        enddo
    enddo
enddo
```

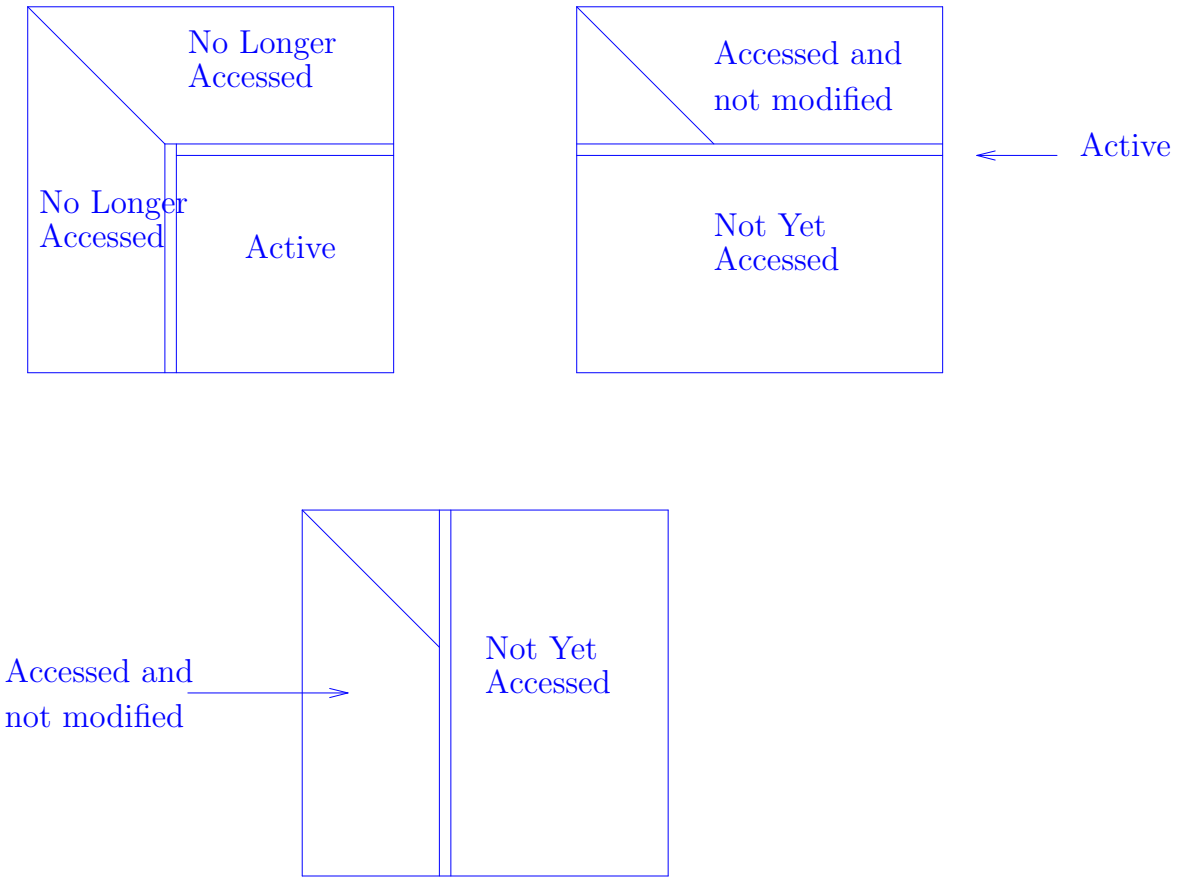
➤ Non-negligible effect on performance. For example, on vector computers, a column variant may be preferable as it involves vector combinations with stride one.

GE - KIJ version

```
do k=1,n-1
    do i=k+1,n
        l(i,k)=a(i,k)/a(k,k)
        do j=k+1, n
           a(i,j)=a(i,j)-l(i,k)*a(k,j)
        enddo
    enddo
enddo
```

GE – KJI variant

```
do k=1,n-1
    // Cdiv:
    do i=k+1, n
        l(i,k)=a(i,k)/a(k,k)
    enddo
    do j = k+1, n
    // Elimination:
        do i=k+1, n
           a(i,j)=a(i,j)-l(i,k)*a(k,j)
        enddo
    enddo
enddo
```



Access patterns for variants of Gaussian Elimination.

Can you recognize which variant is represented by each figure?

A broadcast algorithm

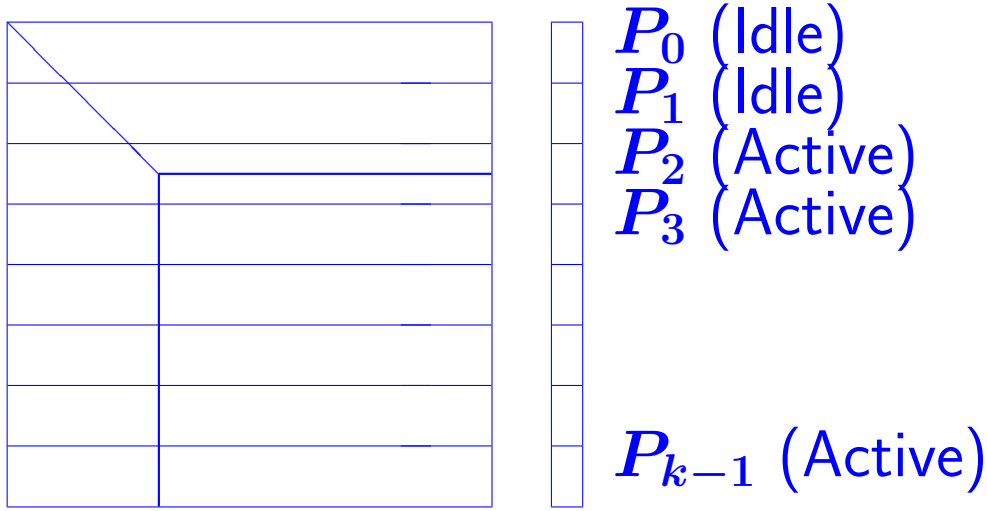
Assume 1-D mapping of data – row-wise (use row-version of algorithm) or column-wise (use column version)

Row version of GE can be written as

```
do k=1,n-1
  do i=k+1,n
     l(i,k) = a(i,k) / a(k,k)
     a(i,k+1:n+1)=a(i,k+1:n+1)-l(i,k)*a(k,,k+1:n+1)
  enddo
enddo
```

➤ Idea : broadcast row $a(k,:)$ to all processors at each step.

➤ Simplest distribution of matrix and RHS to processors: Assign equations $i(n/p)$ to $(i+1)(n/p)-1$ to P_i

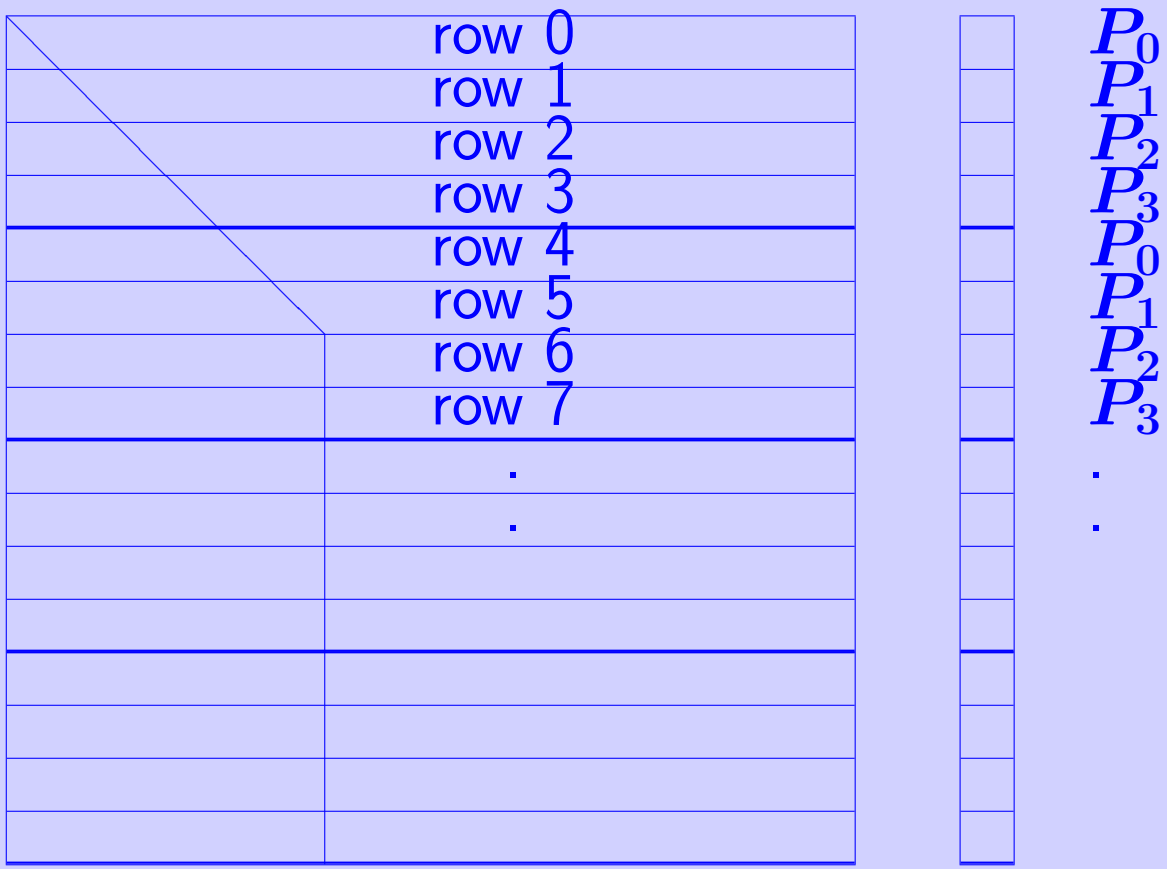


➤ Method: at each step broacast pivot row to all processors as needed.

➤ Difficulty: too much idle time (algorithm is not cost optimal)

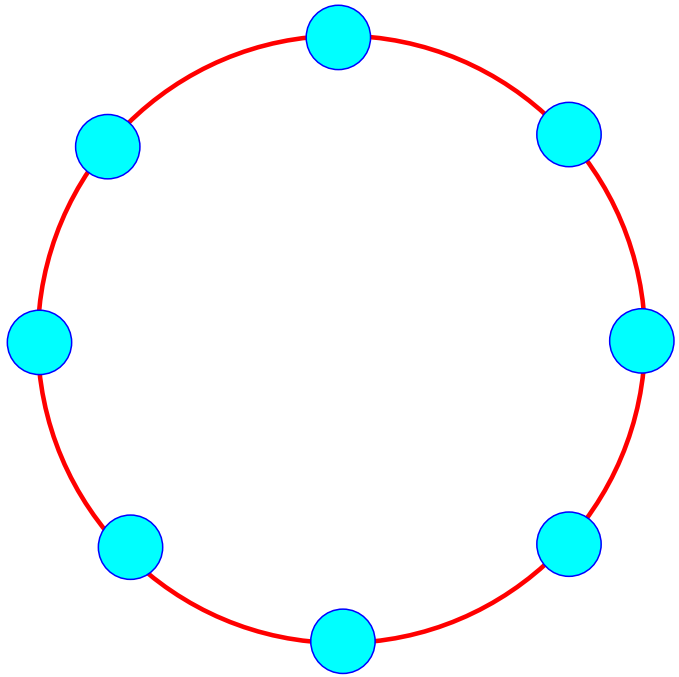
✍9 Cost analysis ?

Better data distribution: cyclic mapping or "interleaving"



Gaussian Elimination with Interleaved Rows

A pipelined algorithm



➤ Assumes only a ring (actually a linear array) of p processors

➤ Communication primitives used:

Send(east,x) Recv(west,x)

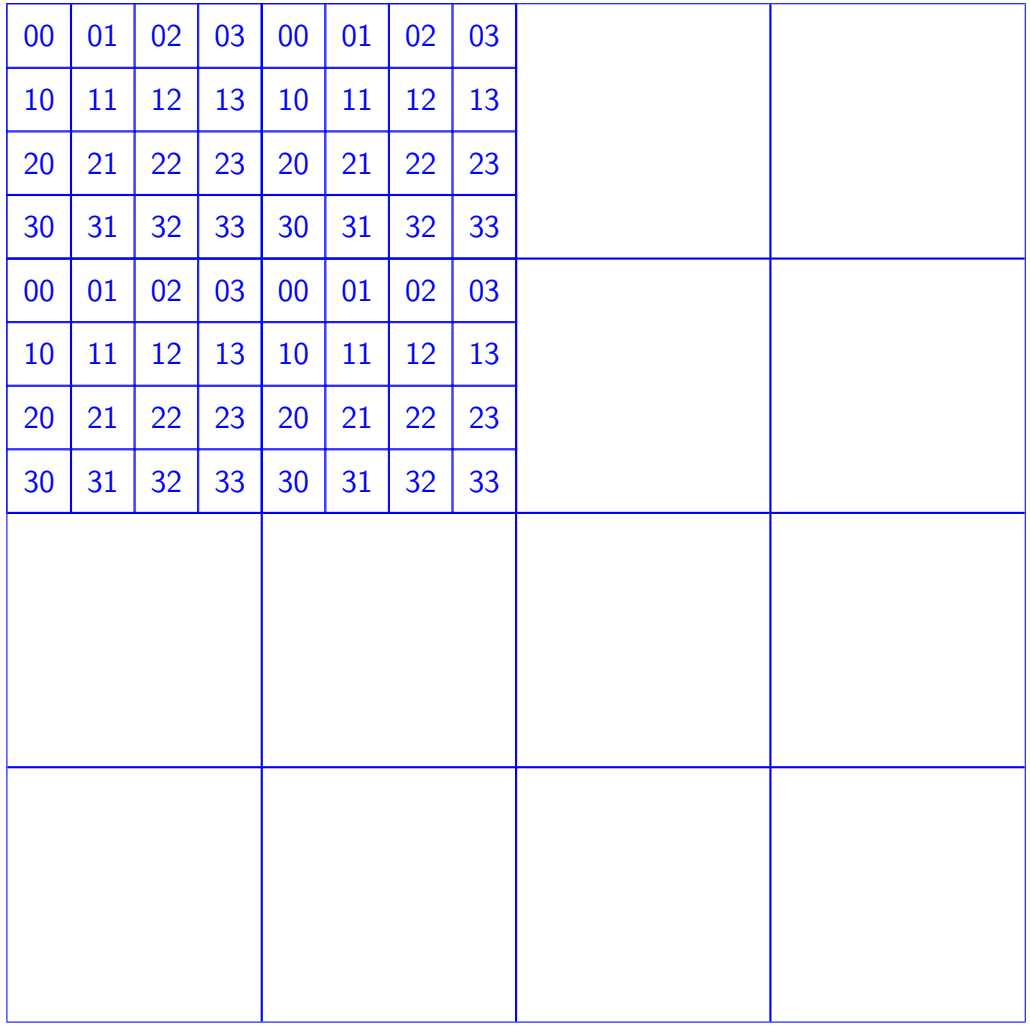
➤ Main idea: do not perform the j-th step of Gaussian at same time. [A processor may be executing the 1st step while another one may be executing the 10th step]. Algorithm driven by availability of pivot row (or column)

➤ Sketch of column-version [KJI] of Algorithm.

```
% Dimensions: n=r*p.
% Columns C(1:r) held in myid are
% A(:,myid+j*p), j=0,1,...,r-1
  j=0;
  do k=1,n
% if (mod(k,p) = myid) then myid holds pivot column
     if mod(k,p)=myid then
        j=j+1;
        cdiv(C(k+1:n,j),C(k,j));
        send(east,C(k+1:n,j),p);
        piv_col(k+1:n)=C(k+1:n,j);
     else
        receive(west,piv_col(k+1:n),counter);
%  counter used to limit the number of times
%  pivot column is sent
        counter=counter-1;
        if counter>1 then
% Immediately forward it to next PE
           send(east,piv_col(k+1:n),counter) ;
        endif
     endif
```

```
% Only now do eliminations
    do jj=j+1,r
       saxpy(C(k+1:n,jj),piv_col(k+1:n),C(k,jj));
    enddo
  enddo
```

2-D mapping



Interleaving of a linear system in a 4×4 processor grid.

Solving Tridiagonal Linear Systems of equations

Suppose that A is tridiagonal:

$$A = \begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{54} & a_{55} \end{pmatrix}$$

- Many operations can be saved by ignoring the zeros.

Storage: only 3 diagonals to be stored + right hand side. [note: one diagonal can be avoided by scaling a_{ii} to 1 during elimination. $\rightarrow$ 'the Thomas algorithm']

Change of notation System $Ax = r$ with

$$a_{i,i-1} = a_i, \quad a_{i,i} = d_i \quad a_{i,i+1} = c_i$$

$$A = \begin{pmatrix} d_1 & c_1 & & & \\ a_2 & d_2 & c_2 & & \\ & a_3 & d_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ & & & a_n & d_n \end{pmatrix} \qquad r = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_n \end{pmatrix}$$

```
function [r] = trid (a, d, c, r)
%% function [r] = trid (a, d, c, r)
%% solves A x = r, where A = tridiag[a, d, c]
n = length(d);
for k=2:n
    piv = a(k)/d(k-1);
    d(k) = d(k) - piv*c(k-1);
    r(k) = r(k) - piv*r(k-1);
end
r(n) = r(n) / d(n);
for j=n-1:-1:1
    r(j) = (r(j) - c(j)*r(j+1))/d(j);
end
```

Parallel algorithm: Substructuring

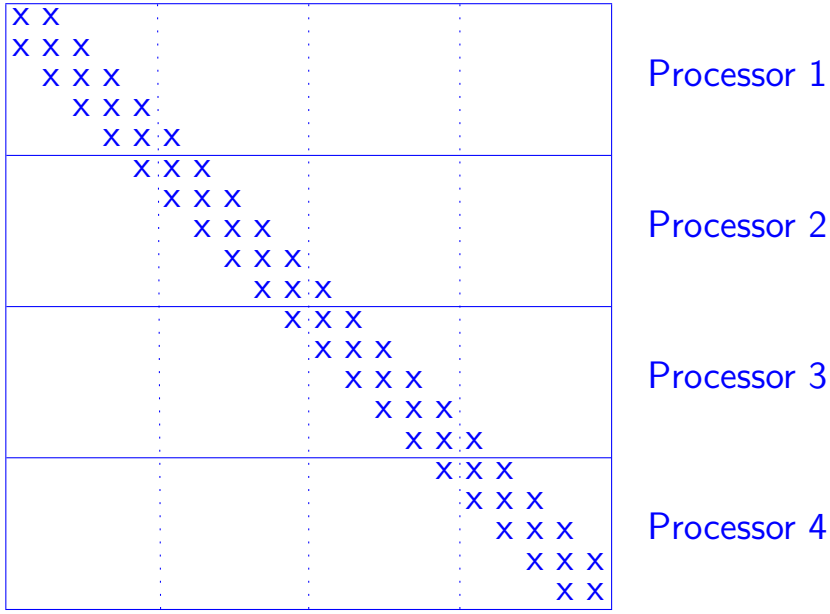
➤ Algorithm described next is due to Wang – a variant of an earlier method by Kuck and Sameh. In what follows j is the processor number, and $m = n/p$. See following figures.

Forward elimination: In Proc. $j = 2, \cdots, p$: Forward-Eliminate unknowns $(j-1)*m+1$, ... jm, to get rid of subdiagonal entries

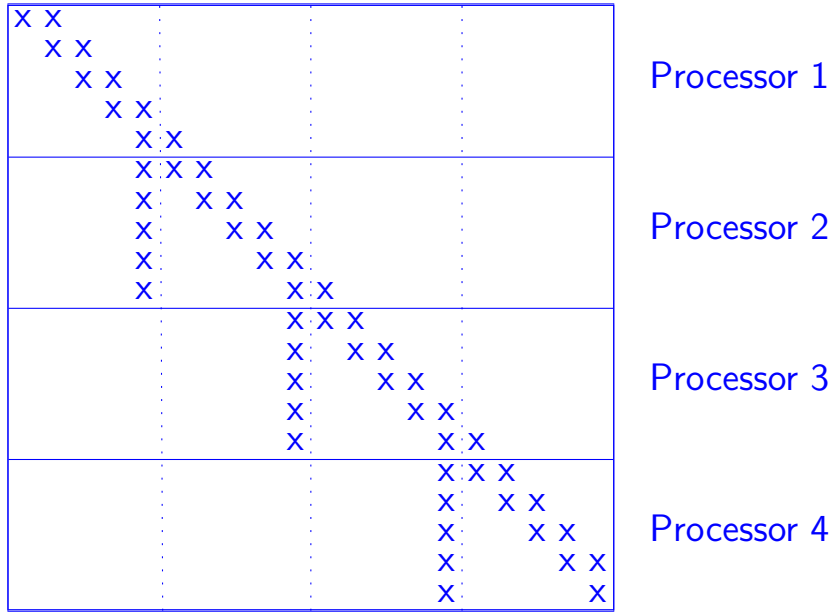
Backward elimination: In Proc $j = 1, \cdots, p-1$ Backward-Elimination to zero out super-diagonal entries in rows $j \times m - 2$ – $(j-1) \times m + 1$

Solve Residual System: Observe that unknowns $jm-1, jm$ for $j = \{1, 2, ..., p-1\}$, satisfy an independent tridiagonal system of size $((2(p-1) \times (2(p-1)))$. ➤ Solve this system

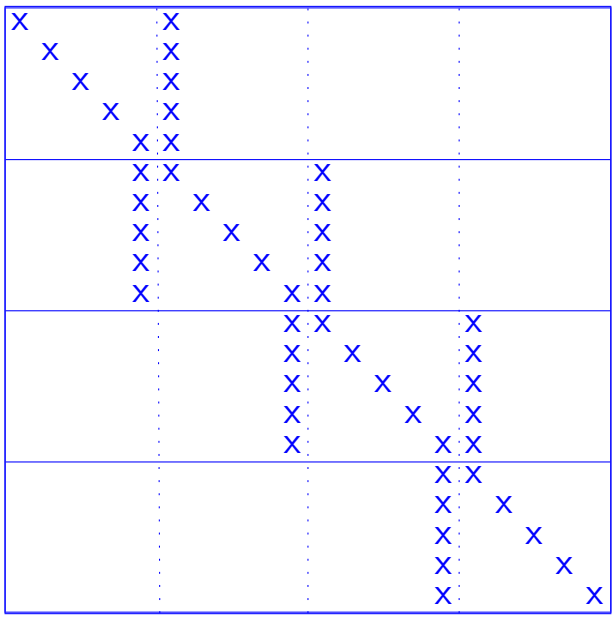
Back-substitute: compute other unknowns by substitution



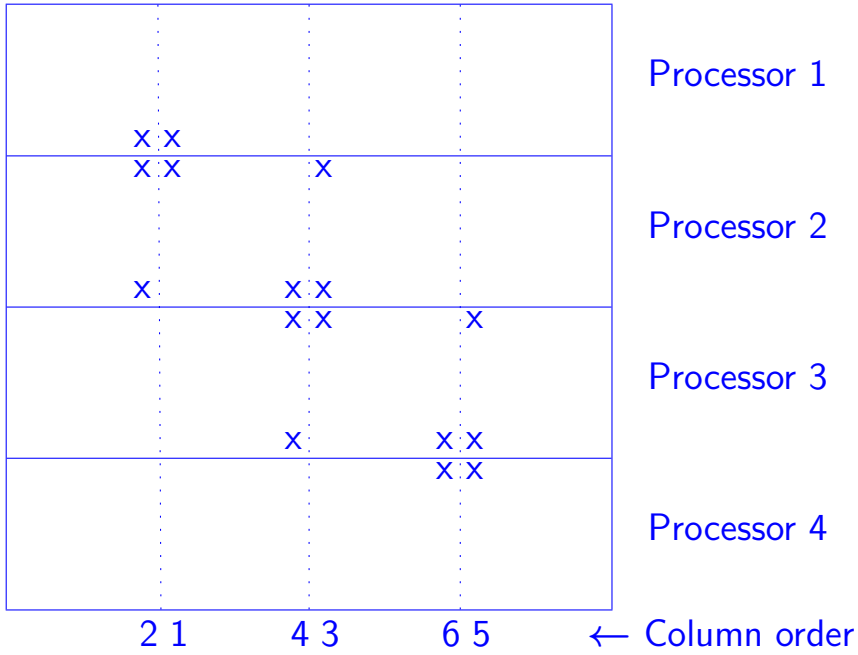
A tridiagonal matrix of dimension 20.



Result of forward elimination



Result of backward elimination



Final independent tridiagonal system to solve