Level-2 BLAS

- Matrix-Vector operations with $\mathcal{O}(n^2)$ operations (sequentially)
- BLAS-Notation:

S	single precision
G E M	general matrix
V	vector

- defines SGEMV, matrix-vector product: $y = \alpha Ax + \beta y$
- Other Level-2 BLAS: solving triangular system $Lx = b$ with triangular matrix L.

Level-3 BLAS

- Matrix-Matrix operations with $\mathcal{O}(n^3)$ operations (sequentially)
- BLAS-Notation:

S	single precision
G E M	general matrix
M	matrix

- defines SGEMM, matrix-matrix product: $C = \alpha AB + \beta C$

Granularity for BLAS

BLAS level	operation	formula	memory	granularity
BLAS-1	AXPY: $2n$	$\alpha x + y$	$2n + 1$	< 1
BLAS-2	GEMV: $2n^2$	$\alpha Ax + \beta y$	$n^2 + 2n$	2
BLAS-3	GEMM: $2n^3$	$\alpha AB + \beta C$	$4n^2$	$\frac{n}{2}$

BLAS-3 has best operations to memory ratio!

2.2. Analysis of the Matrix-Vector-Product

$$A = (a_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,m}} \in \mathbb{R}^{n\times m}, \qquad b \in \mathbb{R}^m, \qquad c \in \mathbb{R}^n$$

2.2.1. Vectorization

$$\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + \cdots + a_{1m}b_m \\ \vdots \\ a_{n1}b_1 + \cdots + a_{nm}b_m \end{pmatrix} =$$

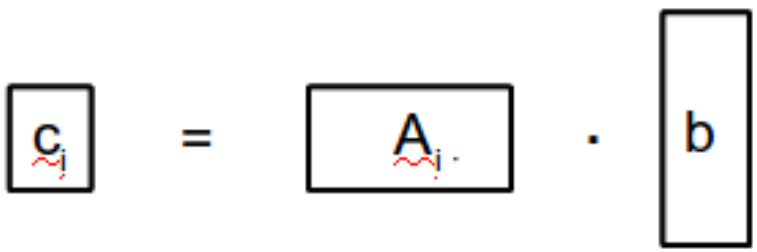
$$= \begin{pmatrix} \sum_{j=1}^{m} a_{1j}b_j \\ \vdots \\ \sum_{j=1}^{m} a_{nj}b_j \end{pmatrix} \qquad = \sum_{j=1}^{m} b_j \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix}$$

n DOT-products of length m — m SAXPYs of length n (GAXPY)

Pseudocode: *ij*-form

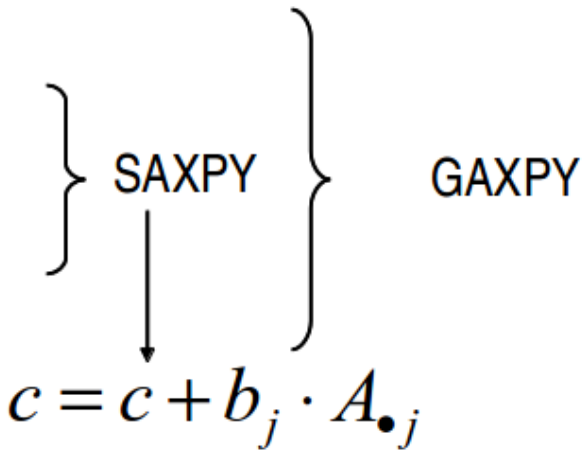
```
c = 0;
for i=1,...,n
    for j=1,...,m
        c_i = c_i + a_ij b_j        } DOT-product
    end
end
```

$c_i = A_{i\bullet} b$, DOT-product of ith row of A with vector b



Pseudocode: *ji*-form

```
c = 0;
for j=1,...,m
    for i=1,...,n
        c_i = c_i + a_ij b_j
    end
end
```



- SAXPY updating vector *c* with *j*th column of *A*
- GAXPY:
  - Sequence of SAXPYs related to the same vector
  - Advantage: vector *c*, that is updated, can be kept in fast memory
- No additional data transfer

GAXPY (repetition)

- SAXPY:

  $y := y + \alpha x$

- GAXPY:

  $y = y_0$

  `for` $i = 1 : n$

  $\quad y := y + \alpha_i x_i$

  `end`

- Series of SAXPYs regarding the same vector y.
- length(GAXPY) $=$ length(y)
- Advantage: less data transfer!

2.2.2. Parallelization by Building Blocks

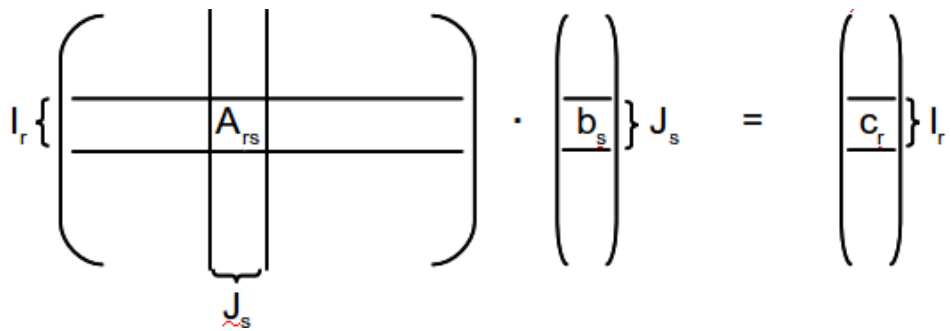
Reduce matrix-vector product on smaller matrix-vector products.

$\{1, 2, \ldots, n\} = \langle 1, n\rangle = I_1 \cup I_2 \cup \ldots \cup I_R$ disjunct: $I_j \cap I_k = 0$ for $j \neq k$

$\{1, 2, \ldots, m\} = \langle 1, m\rangle = J_1 \cup J_2 \cup \ldots \cup J_S$ disjunct: $J_j \cap J_k = 0$ for $j \neq k$

Use 2-dimensional array of processors P_{rs}.

P_{rs} gets matrix block $A_{rs} := A(I_r, J_s)$, $b_s := b(J_s)$, $c_r := c(I_r)$.



$$c_r = \sum_{s=1}^{S} A_{rs} b_s =: \sum_{s=1}^{S} c_r^{(s)}$$

Pseudocode

```
for r = 1,...,R
    for s = 1,...,S
        c_r^(s) = A_rs b_s;
    end
end
```

Small, independent matrix-vector products. No communication necessary during computations!

```
for r = 1,...,R
    c_r = 0
    for s = 1,...,S
        c_r = c_r + c_r^(s);
    end
end
```

Blockwise collection and addition of vectors. Rowwise communication! Fan-in.

Blocking: Special Cases

$S = 1$: The computation of $A_{i\bullet} b$ is vectorizable by GAXPYs.

$$c = \begin{pmatrix} A_{1\bullet} \\ \hline A_{2\bullet} \\ \hline \vdots \end{pmatrix} \cdot b = \begin{pmatrix} A_{1\bullet} b \\ \hline A_{2\bullet} b \\ \hline \vdots \end{pmatrix}$$

No communication necessary between processor $P_1, \ldots, P_R$

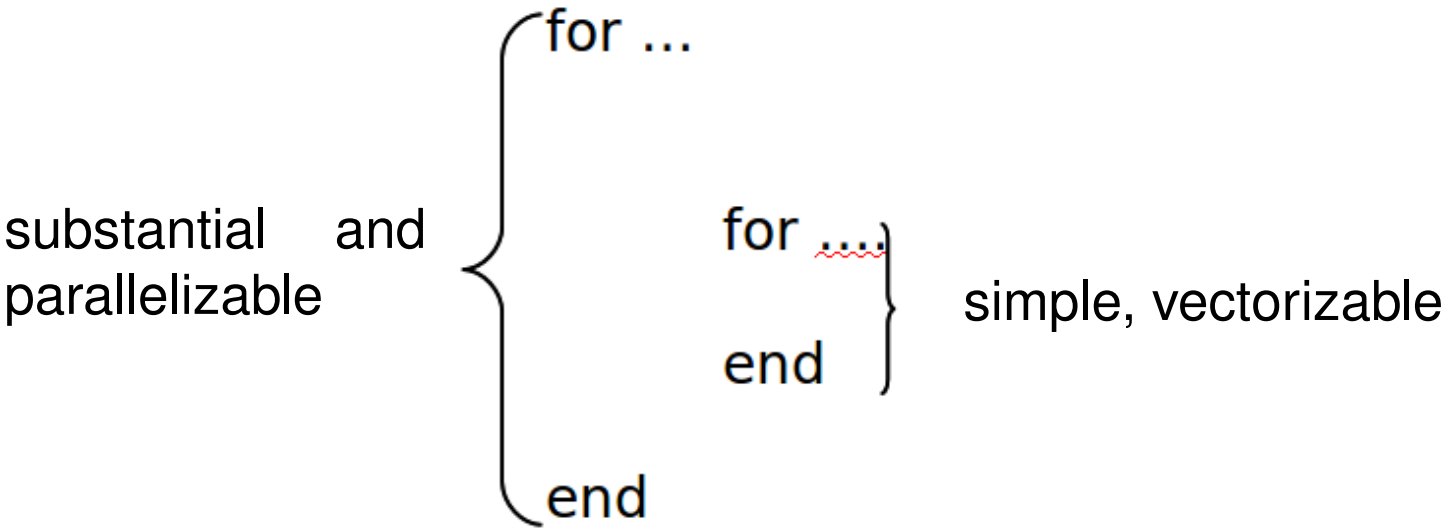
$R = 1$: $A_{\bullet j} b_j$ are independent.

$$c = (A_{\bullet 1} | A_{\bullet 2} | \cdots) \cdot \begin{pmatrix} b_1 \\ \hline b_2 \\ \hline \vdots \end{pmatrix} = A_{\bullet 1} b_1 + A_{\bullet 2} b_2 + \ldots$$

Then collection of partial results from processor $P_1, \ldots, P_S$. Fan-in.
Final sum in one processor: vectorizable by GAXPYs.

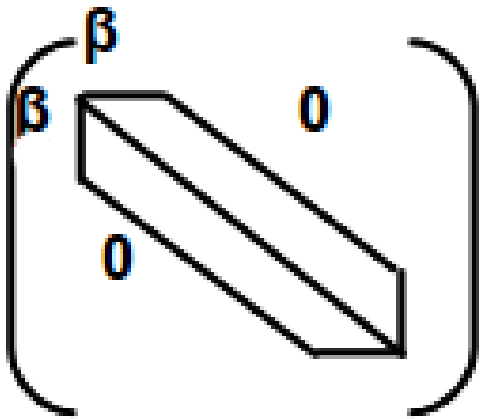
Rules

1. Inner loops of a program should be simple, vectorizable
2. Outer loop of a programm should be substantial, independent, parallelizable.



3. Reuse of data (cache, minimal data transfer, blocking)

2.2.3. $c = Ab$ for Banded Matrix



- Bandwidth β (symmetric)
- 2β+1 diagonals: main diag. + β subdiag. + β superdiag.
- $\beta = 1$: tridiagonal

Notation: Banded Matrices A and $\tilde{A}$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1,\beta+1} & 0 & \cdots & 0 \\ \vdots & a_{22} & \ddots & \ddots & \cdots & \vdots \\ a_{\beta+1,1} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & a_{n-\beta,n} \\ \vdots & \vdots & \ddots & \ddots & a_{n-1,n-1} & \vdots \\ 0 & \cdots & 0 & a_{n,n-\beta} & \cdots & a_{nn} \end{pmatrix} \rightarrow$$

$$\tilde{A} = \begin{pmatrix} \tilde{a}_{10} & \cdots & \tilde{a}_{1,\beta} & 0 & \cdots & 0 \\ \vdots & \tilde{a}_{20} & \ddots & \ddots & \cdots & \vdots \\ \tilde{a}_{\beta+1,-\beta} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \tilde{a}_{n-\beta,\beta} \\ \vdots & \vdots & \ddots & \ddots & \tilde{a}_{n-1,0} & \vdots \\ 0 & \cdots & 0 & \tilde{a}_{n,-\beta} & \cdots & \tilde{a}_{n,0} \end{pmatrix}$$

$c = Ab$ for Banded Matrix

Storing entries diagonalwise: $n(2\beta + 1)$ matrix instead of n^2.

$$\tilde{a}_{i,s} = a_{i,i+s} \quad \text{for row} \quad i = 1, \ldots, n$$

$$1 \leq i + s \leq n \quad \text{and} \quad -\beta \leq s \leq \beta \quad \text{and} \quad 1 \leq i \leq n$$

$$1 - i \leq s \leq n - i \quad \text{and} \quad -\beta \leq s \leq \beta$$

$$\downarrow \quad \textit{in row } i$$

$$s \in [l_i, r_i] = [\max\{-\beta, 1 - i\}, \min\{\beta, n - i\}]$$

$$1 - s \leq i \leq n - s \quad \text{and } 1 \leq i \leq n$$

$$\downarrow \quad \textit{in diag. } s$$

$$i \in [\tilde{l}_s, \tilde{r}_s] = [\max\{1, 1 - s\}, \min\{n, n - s\}]$$

Computation of the mtx-vec product based on storage scheme on vector CPUs

$$\text{For } i = 1, \ldots, n: \; c_i = A_{i\bullet} b = \sum_j a_{ij} b_j = \sum_{s=l_i}^{r_i} a_{i,i+s} b_{i+s} = \sum_{s=l_i}^{r_i} \tilde{a}_{i,s} b_{i+s}$$

- General TRIAD, no SAXPY:

```
for s = -β : β
    for i = max{1 - s, 1} : min{n - s, n}
        c_i = c_i + ã_{i,s} b_{i+s}
    end
end
```

- or, partial DOT-product:

```
for i = 1 : n
    for s = max{-β, 1 - i} : max{β, n - i}
        c_i = c_i + ã_{i,s} b_{i+s}
    end
end
```

- Sparsity $\Rightarrow$ less operations, but also loss of efficiency.

Band Ab in Parallel

- Partitioning:

$$\langle 1, n \rangle = \bigcup_{r=1}^{R} I_r, \text{ disjunct}$$

`for` $i \in I_r$

$$c_i = \sum_{s=l_i}^{r_i} \tilde{a}_{is} b_{i+s}$$

`end`

- Processor P_r gets rows to index set $I_r := [m_r, M_r]$ in order to compute its part of the final vector c.

- What part of vector b does processor P_r need in order to compute its part of c?

Band *Ab* in Parallel

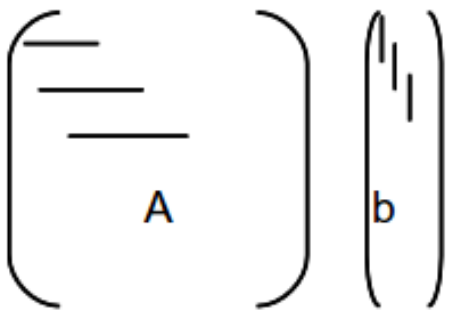
- Necessary for I_r: $b_j = b_{i+s}$:

$$j = i + s \geq m_r + \max\{-\beta, 1 - m_r\} = \max\{m_r - \beta, 1\}$$

$$j = i + s \leq M_r + r_{M_r} = M_r + \min\{\beta, n - M_r\} = \min\{M_r + \beta, n\}$$

- Processor P_r with index set I_r needs from b the indices

$$j \in [\max\{1, m_r - \beta\}, \min\{n, M_r + \beta\}]$$



2.3. Analysis of Matrix-Matrix Product

$$A = (a_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,m}} \in \mathbb{R}^{n\times m}, \quad B(b_{ij})_{\substack{i=1,\dots,m \\ j=1,\dots,q}} \in \mathbb{R}^{m\times q},$$

$$C = AB = (c_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,q}} \in \mathbb{R}^{n\times q}$$

```
for i = 1 : n
    for j = 1 : q
        c_ij = Σ_{k=1}^{m} a_ik b_kj
    end
end
```

$$\begin{pmatrix} * & * & * \\ \boxed{a_{i1} \quad \cdots \quad a_{im}} & & \\ * & * & * \end{pmatrix} \cdot \begin{pmatrix} * & \boxed{b_{1j} \\ \vdots \\ b_{mj}} & * \\ * & & * \\ * & & * \end{pmatrix} = \begin{pmatrix} * & * & * \\ * & \boxed{c_{ij}} & * \\ * & * & * \end{pmatrix}$$

2.3.1. Vectorization

- Algorithm 1: (ijk)-Form:

```
for i = 1 : n
    for j = 1 : q
        for k = 1 : m
            c_ij = c_ij + a_ik b_kj   } DOT-product of length m
        end
    end
end
```

$c_{ij} = A_{i\bullet} B_{\bullet j}$ for all i, j

- All entries c_{ij} are fully computed, one after another.
- Access to A and C is rowwise, to B columnwise (depends on inner most loops!)

Other View on the Matrix-Matrix Product

Matrix A considered as combination of columns or rows

$$A = A_1 e_1^T + \ldots + A_m e_m^T = (A_1 \;\; 0 \;\; \cdots) + (0 \;\; A_2 \;\; 0 \;\; \ldots) + \ldots + (\ldots \;\; 0 \;\; A_m)$$

$$= e_1 a_1 + \ldots + e_n a_n = \begin{pmatrix} a_1 \\ 0 \\ \vdots \end{pmatrix} + \ldots + \begin{pmatrix} \vdots \\ 0 \\ a_n \end{pmatrix}$$

$$AB = \sum_{j=1}^{n} A_j e_j^T \sum_{k=1}^{m} e_k b_k = \sum_{k,j} A_j (e_j^T e_k) b_k = \sum_{k=1}^{m} \underbrace{A_k b_k}_{\text{full } n \times q \text{ matrices}}$$

as a sum of full matrices $A_k b_k$ by outer product of the kth column of A and the kth row of B.

Algorithm 2: (jki)-Form

```
for j=1,...,q
    for k=1,...,m
        for i=1,...,n
            c_ij = c_ij + a_ik b_kj
        end
    end
end
```

SAXPY

GAXPY

- Vector update: $c_{\bullet j} = c_{\bullet j} + a_{\bullet k} b_{kj}$
- Sequence of SAXPYs for the same vector: $c_{\bullet j} = \sum_k b_{kj} a_{\bullet k}$
- C computed columnwise; access to A columnwise. Access to B columnwise, but delayed.

Algorithm 3: (kji)-Form

```
for k=1,...,m
    for j=1,...,q
        for i=1,...,n
            c_ij = c_ij + a_ik b_kj        } SAXPY
        end
    end
end
```

- Vector update: $c_{\bullet j} = c_{\bullet j} + a_{\bullet k} b_{kj}$
- Sequence of SAXPYs for different vectors $c_{\bullet j}$ (no GAXPY)
- Access to A columnwise. Access to B rowwise + delayed. C computed with intermediate values $c_{ij}^{(k)}$ which are computed columnwise.

Overview of Different Forms

	ijk Alg. 1	ikj	kij	jik	jki Alg. 2	kji Alg. 3
Access to A by	row	—	—	row	column	column
Access to B by	column	row	row	column	—	—
Comput. of C	row	row	row	column	column	column
Computation of c_{ij}	direct	delayed	delayed	direct	delayed	delayed
Vector operation	DOT	GAXPY	SAXPY	DOT	GAXPY	SAXPY
Vector length	m	q	q	m	n	n

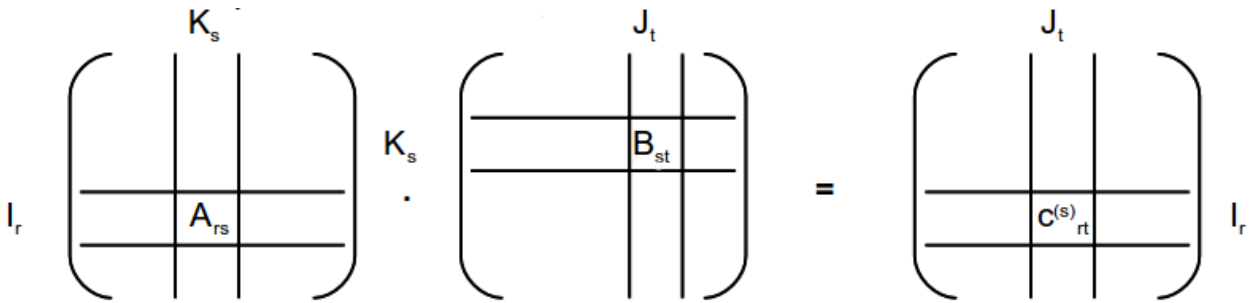
Better: GAXPY (longer vector length).

Access to matrices according to storage scheme (rowwise or columnwise)

2.3.2. Matrix-Matrix Product in Parallel

$$\langle 1, n \rangle = \bigcup_{r=1}^{R} I_r, \quad \langle 1, m \rangle = \bigcup_{s=1}^{S} K_s, \quad \langle 1, q \rangle = \bigcup_{t=1}^{T} J_t$$

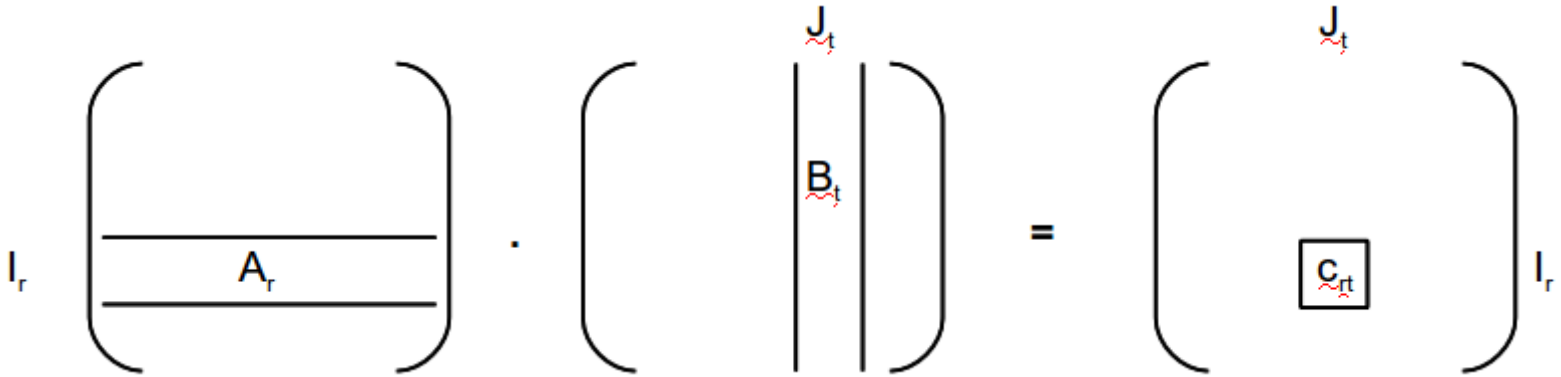
Distribute the blocks relative to index sets I_r, K_s, and J_t to processor array P_{rst}:



1. Processor P_{rst} computes small matrix-matrix product. All processors in parallel: $c_{rt}^{(s)} = A_{rs} B_{st}$
2. Compute sum by fan-in in s:

$$c_{rt} = \sum_{s=1}^{S} c_{rt}^{(s)}$$

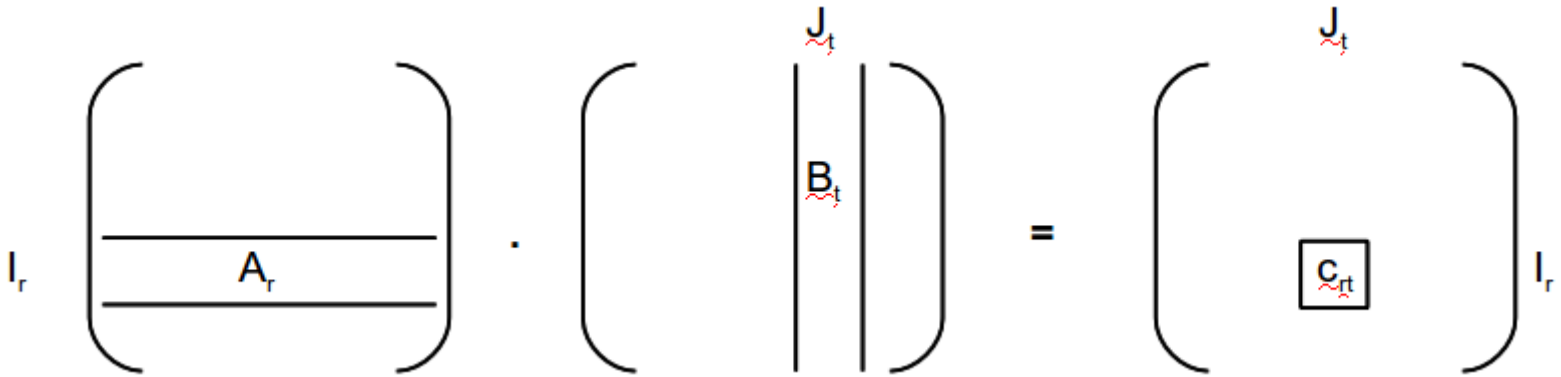
Mtx-Mtx in Parallel: Special Case $S = 1$



- Each processor P_{rt} can compute its part of c, c_{rt}, independently without communication.
- Each processor needs
  - full block of rows of A, relative to index set I_r, and
  - full block of columns of B, relative to index set J_t,

  to compute c_{rt} relative to rows I_k and columns J_t.

Mtx-Mtx in Parallel: Special Case $S = 1$



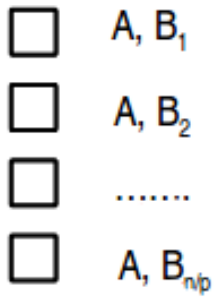
- With $n \cdot q$ processors each processor has to compute one DOT-product with $\mathcal{O}(m)$ parallel time steps.

$$c_{rt} = \sum_{k=1}^{m} a_{rk} b_{kt}$$

- Fan-in by $m \cdot nq$ additional processors for all DOT-products reduces number of parallel time steps to $\mathcal{O}(\log(m))$.

1D-Parallelization of $A \cdot B$

- 1D: p processors linear, each processor gets full A and column slice of B, computing the related column slice of $C = AB$

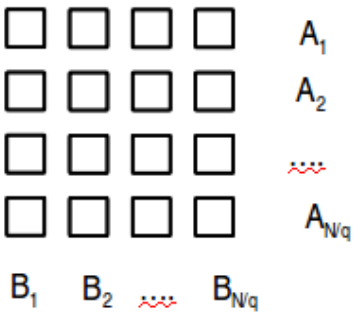


- Communication: $N^2 p$ for A and $(N \cdot \frac{N}{p}) \cdot p = N^2$ for B
- Granularity: $$\frac{N^3}{N^2(1+p)} = \frac{N}{1+p}$$
- Blocking only in i, the columns of B!

```
for i = 1 : n
    for j = 1 : n
        for k = 1 : n
            C_{j,i} = C_{j,i} + A_{j,k} B_{k,i}
```

2D-Parallelization of $A \cdot B$

- 2D: p processors square, $q := \sqrt{p}$, each proc. gets row slice of A and column slice of B computing full subblock of $C = AB$



- Communication: $N^2\sqrt{p}$ for A and $N^2\sqrt{p}$ for B
- Granularity: $$\frac{N^3}{2N^2\sqrt{p}} = \frac{N}{2\sqrt{p}}$$
- Blocking in i and j, the columns of B and the rows of A!

```
for i = 1 : n
    for j = 1 : n
        for k = 1 : n
            C_{j,i} = C_{j,i} + A_{j,k} B_{k,i}
```

3D-Parallelization $A \cdot B$

- 3D: p processors cubic, each processor gets subblock of A and subblock of B, computing part of subblock of $C = AB$.

  Additional fan-in to collect parts to full subblock of C. ($q = p^{\frac{1}{3}}$).

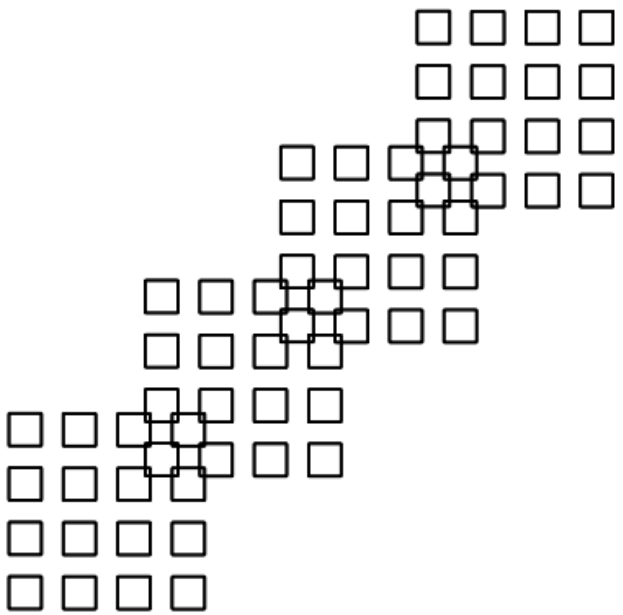
- Communication:

  $$N^2 p^{\frac{1}{3}} \text{ for } A \quad \text{and for } B \left(= p \cdot \frac{N^2}{p^{\frac{2}{3}}} = p \cdot \text{blocksize} \right), \text{ fan-in: } N^2 p^{\frac{1}{3}}$$

- Granularity: $\dfrac{N^3}{3N^2 p^{\frac{1}{3}}} = \dfrac{N}{3p^{\frac{1}{3}}}$

- Blocking in i, j, and k!

```
for i = 1 : n
    for j = 1 : n
        for k = 1 : n
            C_{j,i} = C_{j,i} + A_{j,k} B_{k,i}
```



Parallel Numerics, WT 2013/2014

3 Linear Systems of Equations with Dense Matrices

Contents

3.1. Linear Systems of Equations with Dense Matrices

3.1.1. Gaussian Elimination: Basic Properties

- Linear system of equations:

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &= b_1 \\ &\vdots \quad\quad \vdots \\ a_{n1}x_1 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

- Solve $Ax = b$

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

- Generate simpler linear equations (matrices). Transform A in triangular form: $A = A^{(1)} \rightarrow A^{(2)} \rightarrow \ldots \rightarrow A^{(n)} = U$.

Transformation to Upper Triangular Form

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

row transformations: $(2) \to (2) - \frac{a_{21}}{a_{11}} \cdot (1), \quad \ldots, \quad (n) \to (n) - \frac{a_{n1}}{a_{11}} \cdot (1)$
leads to

$$A^{(2)} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix}$$

next transformations: $(3) \to (3) - \frac{a_{32}^{(2)}}{a_{22}^{(2)}} \cdot (2), \quad \ldots, \quad (n) \to (n) - \frac{a_{n2}^{(2)}}{a_{22}^{(2)}} \cdot (2)$

Transformation to Triangular Form (cont.)

$$A^{(3)} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} \end{pmatrix}$$

next transformations: $(4) \to (4) - \frac{a_{43}^{(3)}}{a_{33}^{(3)}} \cdot (3), \quad \ldots, \quad (n) \to (n) - \frac{a_{n3}^{(3)}}{a_{33}^{(3)}} \cdot (3)$

$$A^{(n)} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}^{(n)} \end{pmatrix} = U$$

Pseudocode Gaussian Elimination (GE)

Simplification: assume that no pivoting is necessary.

$$a_{kk}^{(k)} \neq 0 \quad \text{or} \quad |a_{kk}^{(k)}| \geq \rho > 0 \quad \text{for} \quad k = 1, 2, \ldots, n$$

```
for k = 1 : n-1
    for i = k+1 : n
        l_{i,k} = a_{i,k} / a_{k,k}
    end
    for i = k+1 : n
        for j = k+1 : n
            a_{i,j} = a_{i,j} - l_{i,k} · a_{k,j}
        end
    end
end
```

In practice:

- Include pivoting and include right hand side b.
- There is still to solve a triangular system in U!

Intermediate Systems

$$A^{(k)}, k = 1, 2, \ldots, n \qquad \text{with} \qquad A = A^{(1)} \qquad \text{and} \qquad U = A^{(n)}$$

$$\begin{pmatrix} a_{11}^{(1)} & \cdots & a_{1,k-1}^{(1)} & a_{1,k}^{(1)} & \cdots & a_{1,n}^{(1)} \\ 0 & \ddots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ 0 & \cdots & 0 & a_{k,k}^{(k)} & \cdots & a_{k,n}^{(k)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n,k}^{(k)} & \cdots & a_{n,n}^{(k)} \end{pmatrix}$$

Main part of $A^{(k)}$ that will be used and changed in the following computations.

Define Auxiliary Matrices

$$L = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n,1} & \cdots & l_{n,n-1} & 1 \end{pmatrix} \quad \text{and} \quad U = A^{(n)}$$

$$L_k := \begin{pmatrix} 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & l_{k+1,k} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{n,k} & 0 & \cdots & 0 \end{pmatrix}, \quad L = I + \sum_k L_k$$

Elimination Step in Terms of Auxiliary Matrices

$$A^{(k+1)} = (I - L_k) \cdot A^{(k)} = A^{(k)} - L_k \cdot A^{(k)}$$

$$U = A^{(n)} = (I - L_{n-1}) \cdot A^{(n-1)} = \ldots = (I - L_{n-1}) \cdots (I - L_1) A^{(1)} = \tilde{L} \cdot A$$

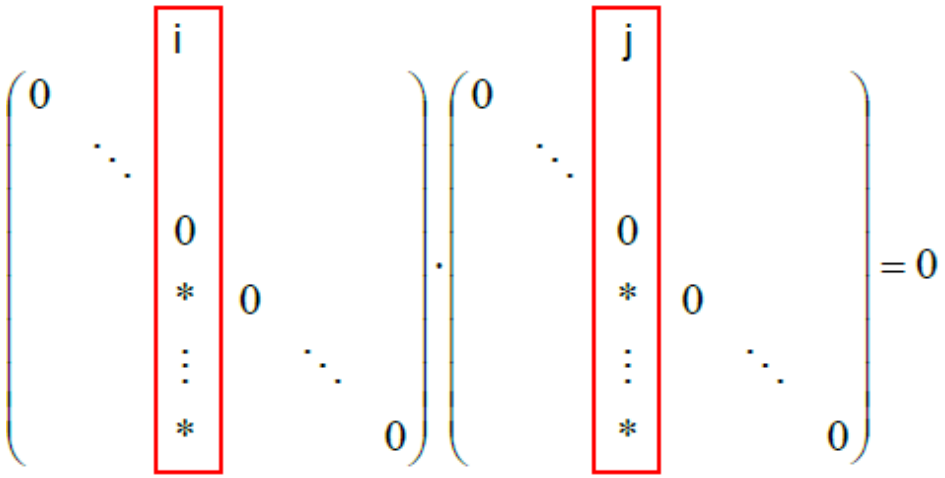
$$\tilde{L} := (I - L_{n-1}) \cdots (I - L_1)$$

$A = \tilde{L}^{-1} \cdot U$ with U upper triangular and $\tilde{L}$ lower triangular

- Theorem 2: $\tilde{L}^{-1} = L$ and therefore $A = LU$.
- Advantage: Every further problem $Ax = b_j$ can be reduced to $(LU)x = b_j$ for arbitrary j.
- Solve two triangular problems $(LU)x = Ly = b$ and $Ux = y$.

Theorem 2: $\tilde{L}^{-1} = L \rightarrow A = LU$

for $i \leq j$: $L_i \cdot L_j =$



$$(I + L_j)(I - L_j) = I + L_j - L_j - L_j^2 = I \Rightarrow (I - L_j)^{-1} = I + L_j$$

$$(I + L_i)(I + L_j) = I + L_i + L_j + L_i L_j = I + L_i + L_j$$

$$\tilde{L}^{-1} = [(I - L_{n-1}) \cdots (I - L_1)]^{-1} = (I - L_1)^{-1} \cdots (I - L_{n-1})^{-1} =$$

$$(I + L_1)(I + L_2) \cdots (I + L_{n-1}) = I + L_1 + L_2 + \cdots + L_{n-1} = L$$

3.2. GE in Parallel: Blockwise

Main idea: Blocking of GE to avoid data transfer between processors.

Basic Concepts:

Replace GE or large LU-decomposition of full matrix by small intermediate steps (by sequence of small block operations):

- Solving collection of small triangular systems $LU_k = B_k$ (parallelism in columns of U)
- $A \rightarrow A - LU$ updating matrices (also easy to parallelize)
- small $B = LU$-decompositions (parallelism in rows of B)

How to Choose Blocks in L/U Satisfying $LU = A$

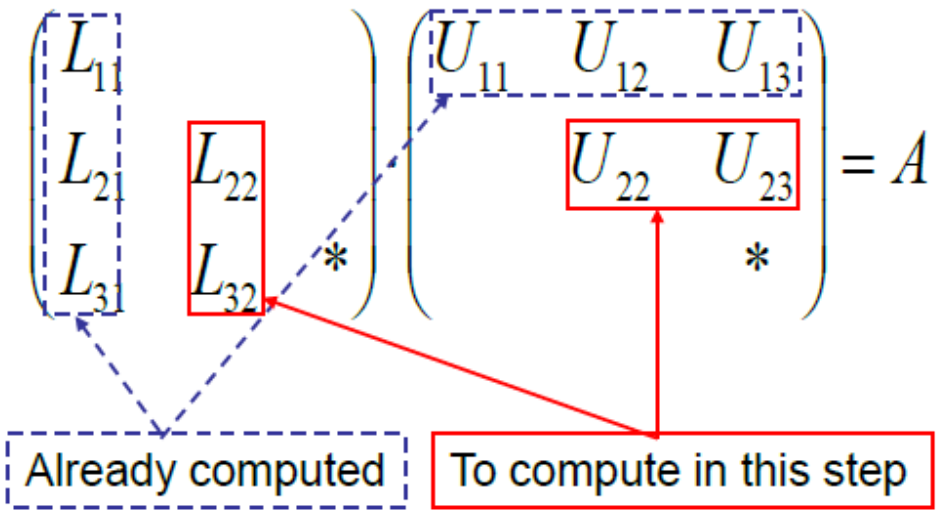
$$\begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} =$$

$$= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} & L_{11}U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} & L_{21}U_{13} + L_{22}U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & * \end{pmatrix}$$

Different ways of computing L and U depending on

- start (assume first entry/row/column of L/U as given)
- how to compute new entry/row/column of L/U
- update of block structure of L/U by grouping in
  - known blocks
  - blocks newly to compute
  - blocks to be computed later

Crout Form



$$\begin{pmatrix} L_{22}U_{22} & L_{22}U_{23} \\ L_{32}U_{22} & * \end{pmatrix} \overset{!}{=} \begin{pmatrix} A_{22} - L_{21}U_{12} & A_{23} - L_{21}U_{13} \\ A_{32} - L_{31}U_{12} & * \end{pmatrix} = \begin{pmatrix} \hat{A}_{22} & \hat{A}_{23} \\ \hat{A}_{32} & * \end{pmatrix}$$

Leads to two subproblems in ⬚ and in ⬚

Crout Form (cont.)

1. Solve

$$\begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} \cdot U_{22} = \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix}$$

   by small LU-decomposition of the modified part of $A \rightarrow L_{22}, L_{32}$, and U_{22}.

2. Solve

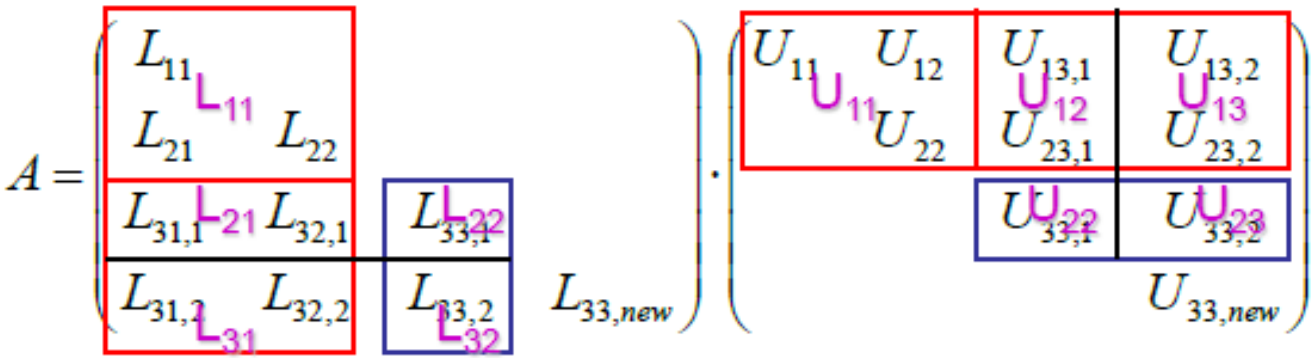
$$L_{22} \cdot U_{23} = \hat{A}_{23}$$

   by solving small triangular systems of equations in $L_{22} \rightarrow U_{23}$.

Initial steps:

$$L_{11} U_{11} = A_{11}, \quad \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{11} = \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix}, \quad L_{11} (U_{12} \;\; U_{13}) = (A_{12} \;\; A_{13})$$

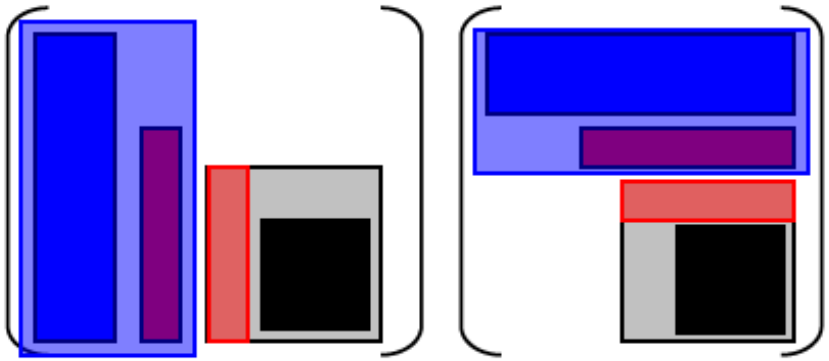
New Partitioning



- Combine already computed parts from second column of *L* and second row of *U* into first column of *L* and first row of *U*.
- Split the until now ignored parts L_{33} and U_{33} into new columns/rows.
- Repeat this overall procedure until *L* and *U* are fully computed.

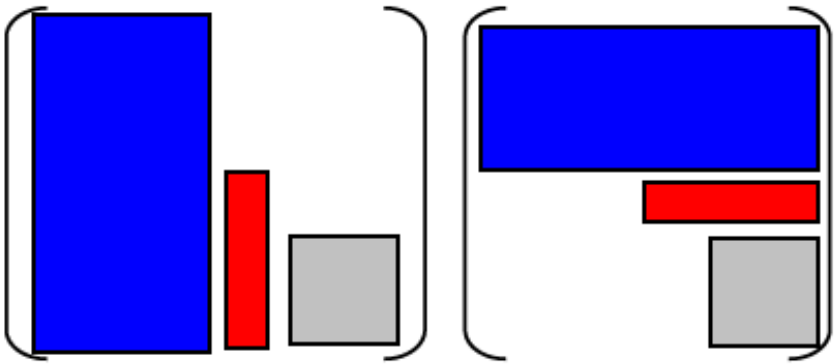
Block Structure

Intermediate block structure:



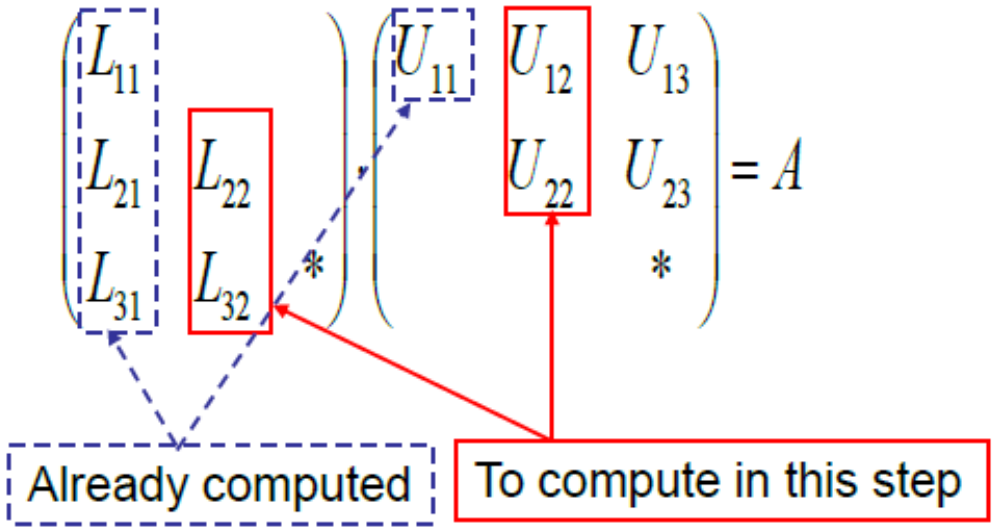
Solve for red blocks.

Reconfigure the block structure:



Repeat until done.

Left Looking GE



- Solve $L_{11}U_{12} = A_{12}$ by a couple of parallel triangular solves and

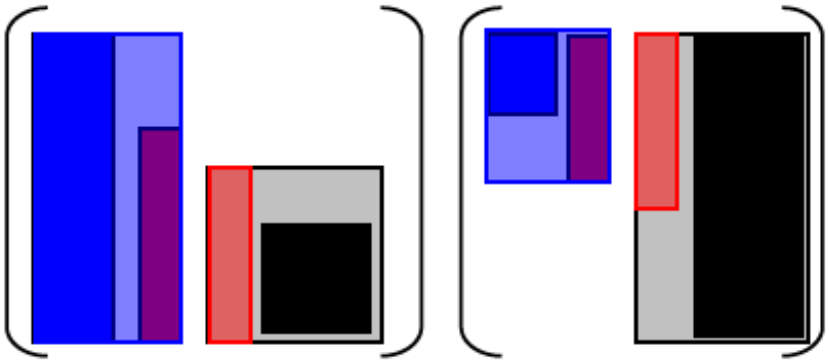
$$\begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22} = \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12} =: \begin{pmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{pmatrix}$$

  update part of A and perform small LU-decompostion.

- Reorder blocks and repeat until ready. Start: $L_{11}U_{11} = A_{11}$, $L_{21}U_{11} = A_{21}$, and $L_{31}U_{11} = A_{31}$.

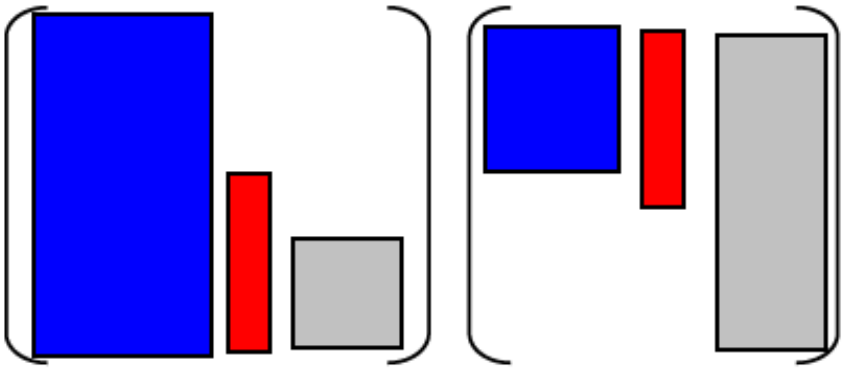
Block Structure

Intermediate block structure:



Solve for red blocks.

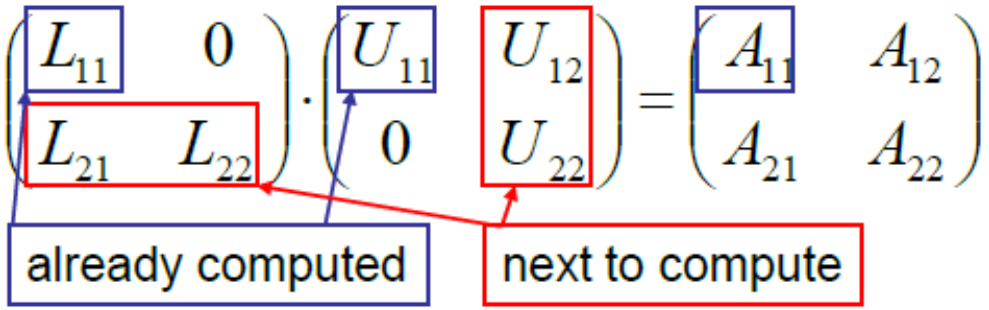
Reconfigure the block structure:



Repeat until done.

Right Looking GE

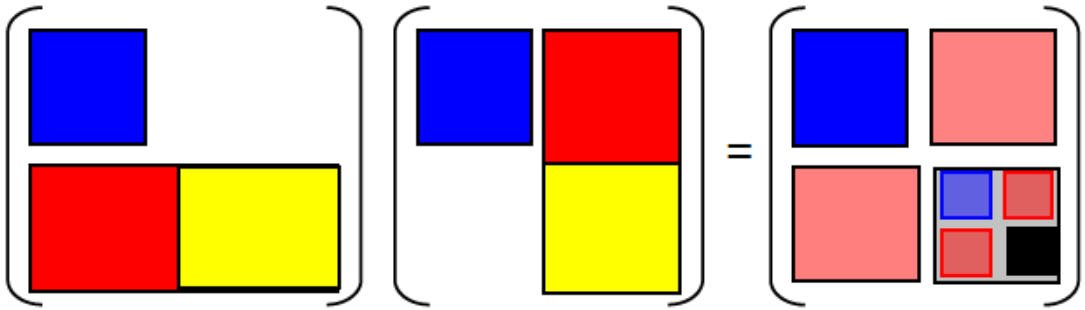
New blocking:



- Start with $L_{11}U_{11} = A_{11}$ (small LU-decomposition).
- Equations $L_{21}U_{11} = A_{21}$ and $L_{11}U_{12} = A_{12}$ by triangular solves gives L_{21} and U_{12}.
- It remains $L_{22}U_{22} = A_{22} - L_{21}U_{12} = \hat{A}_{22}$
- To compute the LU-decomposition of modified A_{22} repeat 2×2-blocking for A_{22} and apply recursively.

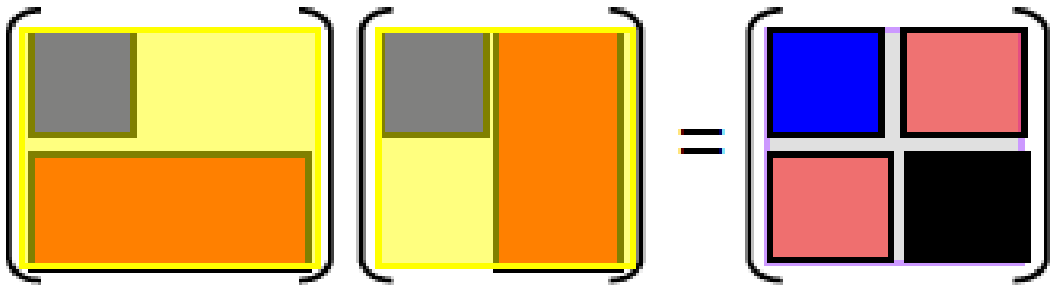
Block Structure

Intermediate block structure:



Solve for blue and both red blocks.

Reconfigure the block structure:



Repeat until done.

Comparison and Overview

- In comparison, all methods
  - have nearly same efficiency in parallel
  - but better performance (in sequential or parallel) than the unblocked variants because they are based on BLAS-3.
- Elementary steps of all blocking methods:
  - Matrix-Matrix product and sum (easy to parallelize)
  - Couple of triangular solves (easy to parallelize)
  - Small LU-decomposition (parallelizable for long rows)
- Crout and right looking slightly better because more flops in matrix-updates and less triangular solves respectively *LU*-decompositions.