

Lecture 11

Matrix Factorization Programming Heterogeneous Architectures

Announcements

- Office hours cancelled on Thurs
- Project presentations

Results

	1060 N	1060 SM	1060 B	570N	570SM	570L1
SP		348(1K)		113 (4K)	596(1K)	265(4K)
DP		66.7 (1K)		79/80 (1K)	134(1K) 147(4K)	132(4K)

Today's lecture

- Matrix Factorization
- Programming Heterogeneous architectures

Linear systems of equations

- A linear system of 2 equations in 2 unknowns ...

$$2x_1 + 3x_2 = 8$$

$$3x_1 + 2x_2 = 7$$

- Solution is $x=1, y=2$
- We may express the system using matrix notation: $A\mathbf{x} = \mathbf{b}$

$$A = \begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, b = \begin{pmatrix} 8 \\ 7 \end{pmatrix}$$

- When we solve for $x_1 = (8 - 3x_2)/2$ and substitute into the 2nd equation, we reduce the matrix to an equivalent form

$$A = \begin{pmatrix} 2 & 3 \\ 0 & -2.5 \end{pmatrix}, b = \begin{pmatrix} 8 \\ -5 \end{pmatrix}$$

- We multiply row 1 of A by 3/2 and subtracting the scaled version from row 2 of A and b

A 3×3 example

- Consider the following system of equations

$$x_0 + x_1 + x_2 = 3$$

$$4x_0 + 3x_1 + 4x_2 = 8$$

$$9x_0 + 3x_1 + 4x_2 = 7$$

- We usually write the system as an *augmented matrix*

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 4 & 3 & 4 & 8 \\ 9 & 3 & 4 & 7 \end{array} \right]$$

3 × 3 example

- Multiply row 0 by 4,
and subtract from row 1

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 4 & 3 & 4 & 8 \\ 9 & 3 & 4 & 7 \end{array} \right]$$

$$[4 \ 3 \ 4 \ 8] - 4*[1 \ 1 \ 1 \ 3] = [0 \ -1 \ 0 \ -4]$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 9 & 3 & 4 & 7 \end{array} \right]$$

3 × 3 example

- Multiply row 0 by 9,
and subtract from row 2

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 9 & 3 & 4 & 7 \end{array} \right]$$

$$[9 \ 3 \ 4 \ 7] - 9*[1 \ 1 \ 1 \ 3] = [0 \ -6 \ -5 \ -20]$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & -6 & -5 & -20 \end{array} \right]$$

3 × 3 example

- Eliminate second column
- Multiply row 1 by 6,
and add to row 2

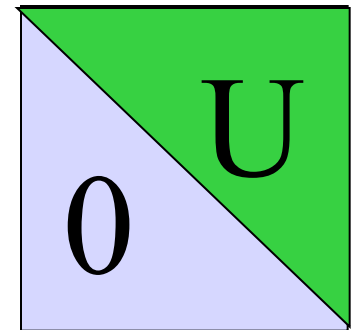
$$\begin{aligned} & [0 \ -6 \ -5 \ -20] + -6*[0 \ -1 \ 0 \ -4] \\ & = [0 \ 0 \ -5 \ 4] \end{aligned}$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & -6 & -5 & -20 \end{array} \right]$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & 0 & -5 & 4 \end{array} \right]$$

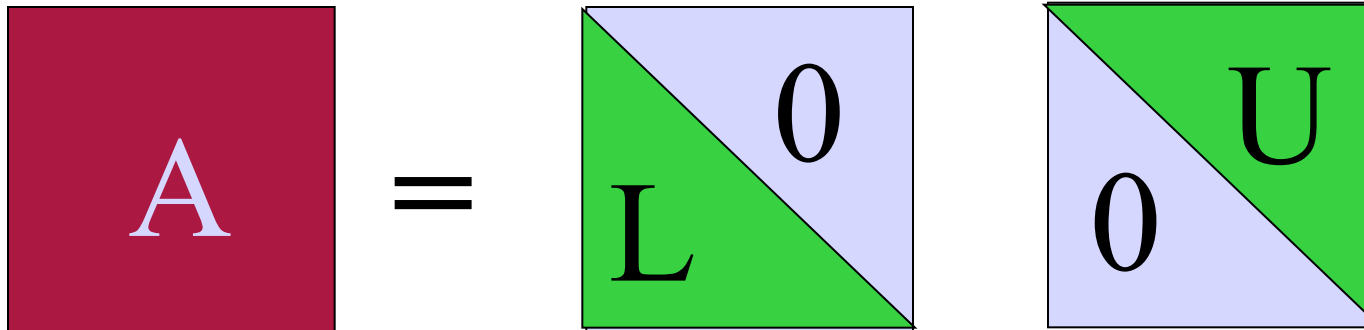
Gaussian Elimination

- The process of eliminating the non-zero values under the main diagonal is called ***Gaussian Elimination***, named after the mathematician *Johann Carl Friedrich Gauss* (1777-1855)
- Input: an $n \times n$ matrix corresponding to a linear system of n equations in n unknowns (must have non-trivial sol'n)
- Output: $n \times n$ matrix with non-zero values above the main diagonal only
upper triangular matrix U



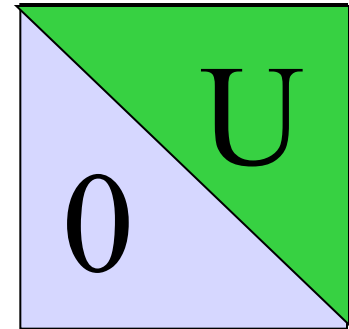
What are we computing?

- GE computes the *LU factorization* $A = L U$, where L is a *lower triangular matrix*
- Plugging LU into the original equation $A\mathbf{x} = \mathbf{b}$
 $A\mathbf{x} = (LU) \mathbf{x} = L (U\mathbf{x}) = L\mathbf{y} = \mathbf{b}$ where $\mathbf{y} = U\mathbf{x}$


$$A = L U$$

Solving the system of linear equations

- Step 1: obtain the upper triangular matrix U ...
- Step 2: solve the corresponding upper triangular system $U\mathbf{x} = \mathbf{c}$ by *back substitution*
- Focus on step 1, which is much more expensive
 - $O(n^3)$ vs $O(n^2)$



Cost

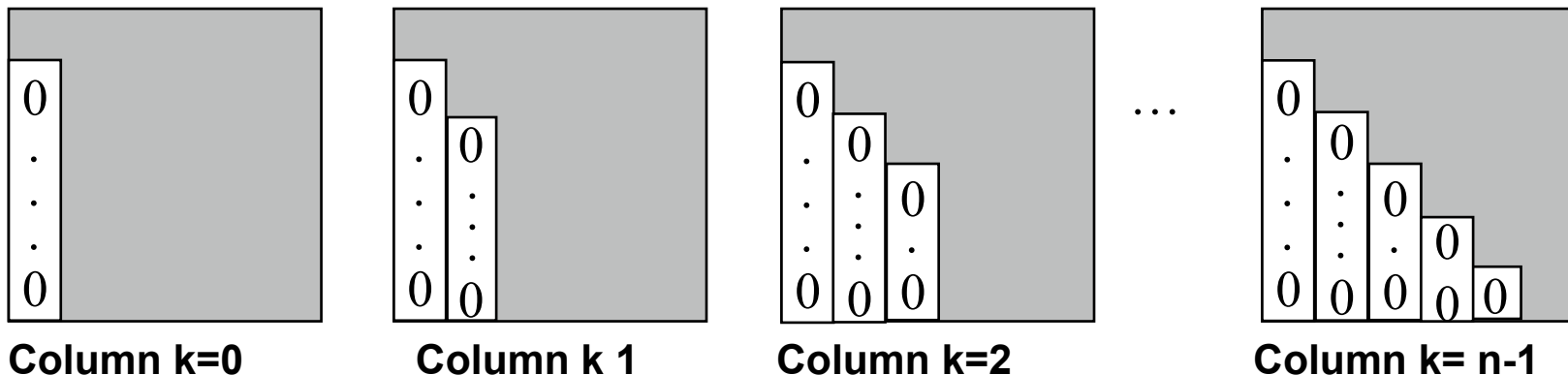
- To solve $A\mathbf{x} = \mathbf{b}$
 - Plug LU into the original equation $A\mathbf{x} = \mathbf{b}$
 $A\mathbf{x} = (LU)\mathbf{x} = L(U\mathbf{x}) = L\mathbf{y} = \mathbf{b}$ where $\mathbf{y} = U\mathbf{x}$
 - Factorize $A = LU$ using GE *($2/3 n^3$ flops)*
 - Solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ using substitution
(n^2 flops)
 - Solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ using back substitution
(n^2 flops)
- We don't compute U explicitly unless we are solving for multiple right hand sides $\mathbf{b}$

Visualizing the algorithm

- Eliminate non-zeros below the diagonal ...
 - One column at a time
 - Scanning from left to right

```

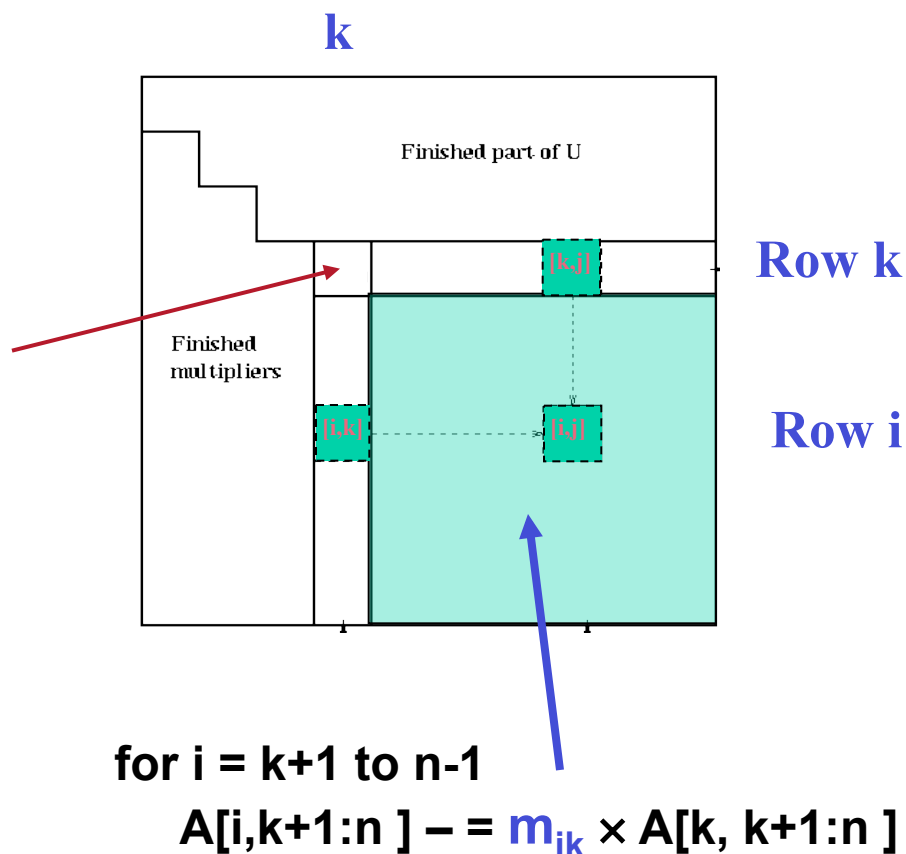
for k = 0 to n-1           // For each column k
    for i = k+1 to n-1     // Eliminate entries below the diagonal:
                           // subtract a multiple of row k
                           // from succeeding rows
         $A[i, k+1:n] \leftarrow (A[i, k] / A[k, k]) * A[k, k+1:n]$  // Rank 1 update
    end for
end for
    
```



Courtesy J. Demmel

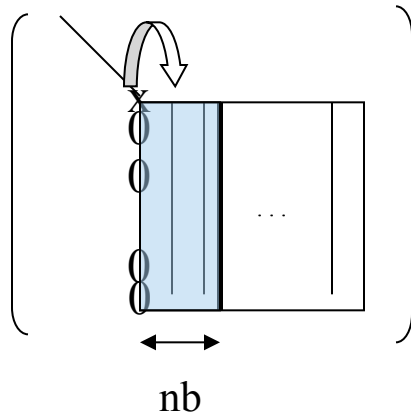
Rank-1 updates

- For each column k in 0 to $n-1$
- ... subtract multiples of row k : $A[k, k+1:n]$
- ... from rows $i = k+1$ to n
 - ▶ Multipliers $m_{ik} = A[i, k] / A[k, k]$
 - ▶ ... cancel the elements below the diagonal: $A[k+1:n-1, k]$
 - ▶ Thus $A[k, k] * m_k - A[j, k] = 0$
- Update only to right and below $A[k, k]$
- Rank 1 updates are costly

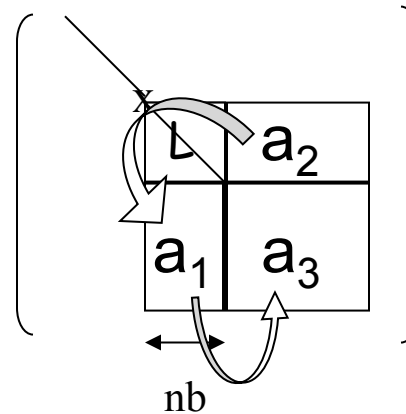


Blocked algorithm

Blocked algorithm eliminates a block column at a time
 Aggregate the trailing submatrix updates from many columns
 Delayed updates uses fast BLAS 3 ops rather than slow BLAS 2 ops (rank 1 update) to update the trailing submatrix



apply sequence to nb

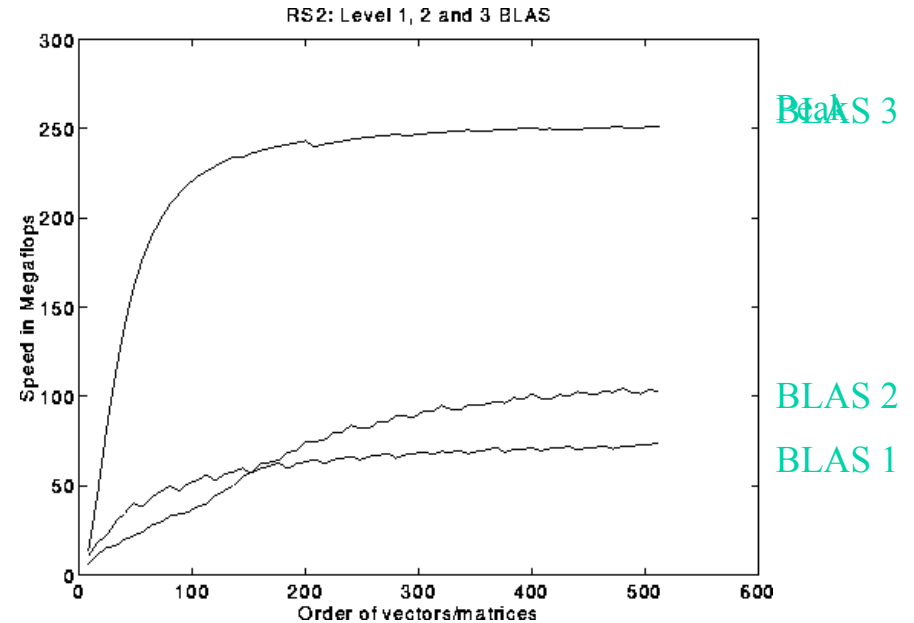


$$a_2 = L^{-1} a_2$$

$$a_3 = a_3 - a_1 * a_2$$

then apply nb to rest of matrix

Courtesy J. Dongarra



Courtesy J. Demmel

Pivoting

- The rank-1 update step uses division ...

$$A[i, k+1:n] -= (A[i,k]/A[k,k]) * A[k,k+1:n]$$

- How to handle vanishing denominators or ones that are very small
- Gaussian elimination will fail with this matrix
- But we can avoid the problem if we swap rows

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Pivoting to avoid stability problems

- We call this process of swapping rows *partial pivoting*
- Also handles tiny pivots
- Assume we carry 3 decimal digits of precision
- Consider the following A matrix and RHS b

$$A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

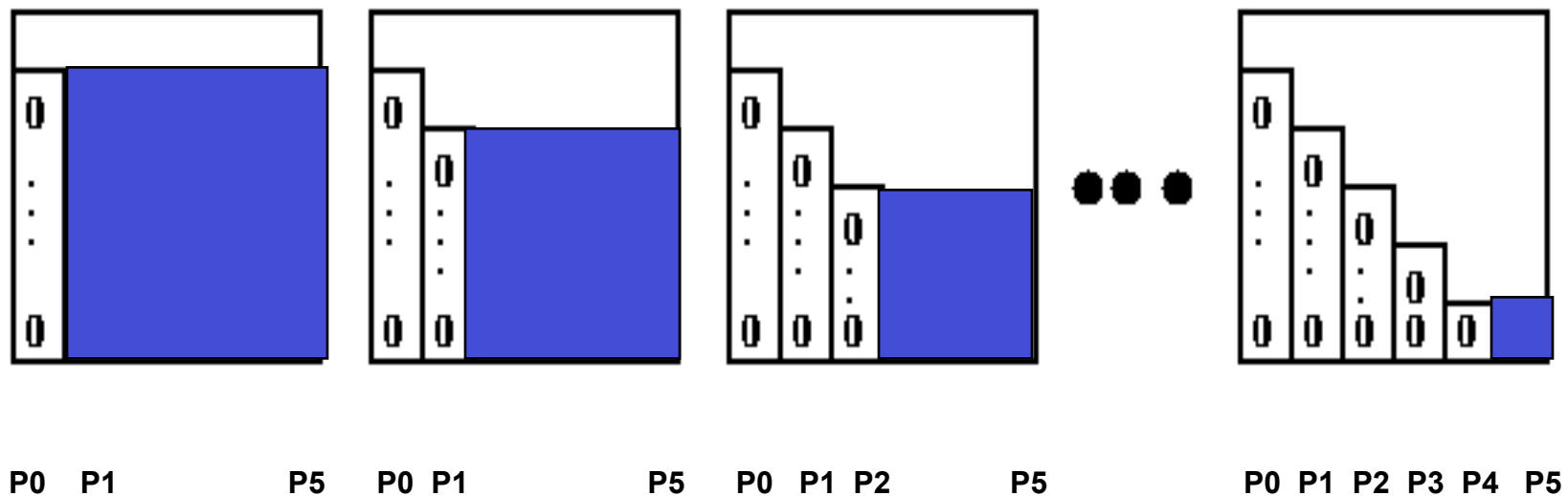
- We compute $x = [1 \ 0]^T$
- The correct solution is $x = [1 \ 1]^T$

Partial Pivoting

- Rule: pick the largest value in the column
- This is called partial pivoting, because only rows are swapped
- It can be shown that with partial pivoting, we compute $\mathbf{A} = \mathbf{P} \mathbf{L} \mathbf{U}$, where $\mathbf{P}$ is a permutation matrix expressing the rows swaps
- We can also swap columns: *full pivoting*
- But full pivoting is more expensive to implement

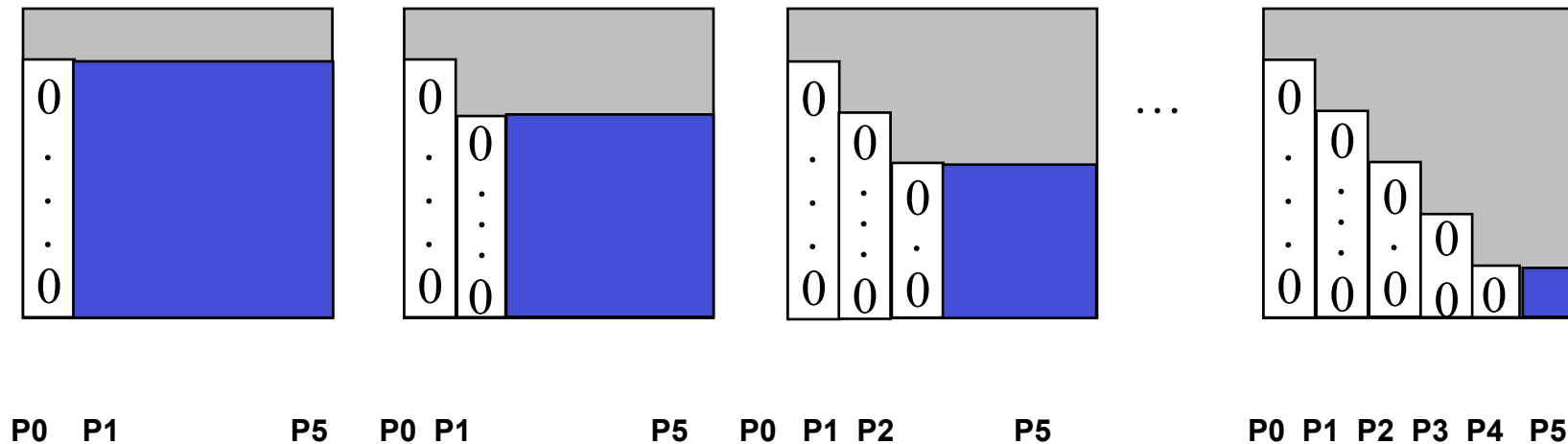
Parallelization

- Consider 1D vertical strip partitioning
- Each thread owns N/P columns
- Let $P=N=6$
- The ■ represents outstanding work in succeeding k iterations



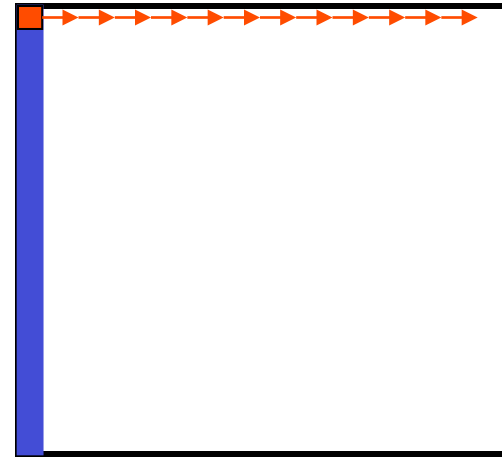
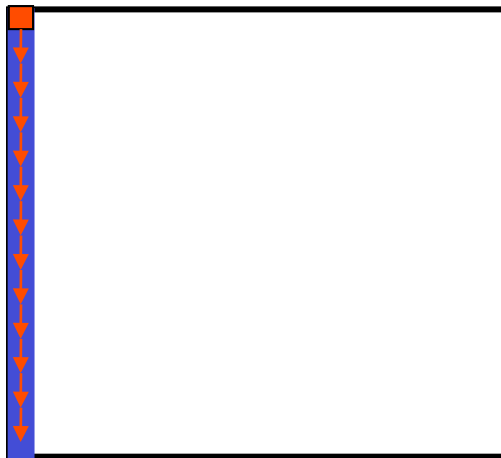
Parallelization

- Consider 1D vertical strip partitioning
- Each thread owns N/P columns
- Let $P=N=6$
- ■ = outstanding work in succeeding k iterations



Communication and control

- Each thread is in charge of eliminating N/P columns
- One thread chooses the pivot row and computes the multipliers
- The other threads share this value
- All threads carry out updates



Analyzing the Parallel Control flow

- All threads carry out updates
- At each step k , thread $k \text{ div } P$ is in charge:
it computes the multipliers $m[\]$
- Elements in $A[k, k+1:n]$ (row k) have different owners
- Thread $j \text{ div } p$ owns $A[k,j]$ in $A[k, k+1:n]$

for $k = 0$ **to** $n-1$ // For each column k

 // Compute Multipliers

$m[k+1:n-1] = A[k+1:n-1, k] / A[k, k]$

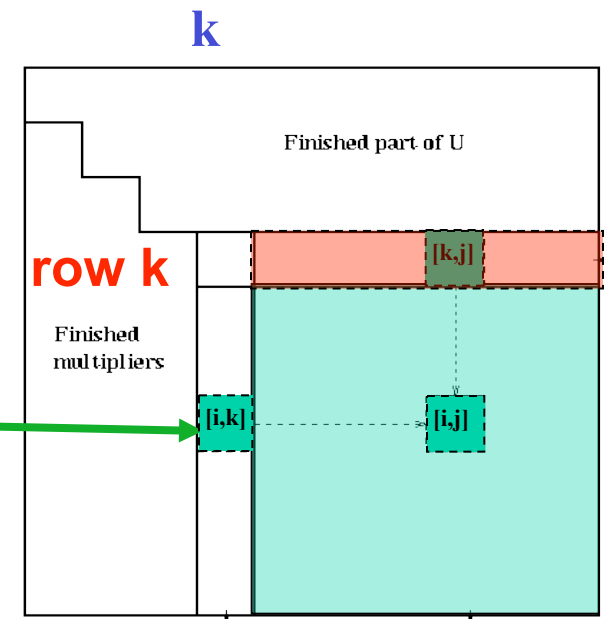
for $i = k+1$ **to** $n-1$ // for each row $i > k$

 // Scale row k by m_{ik} and
 // subtract from row i

$A[i, k+1:n] -= m[i] * A[k, k+1:n]$

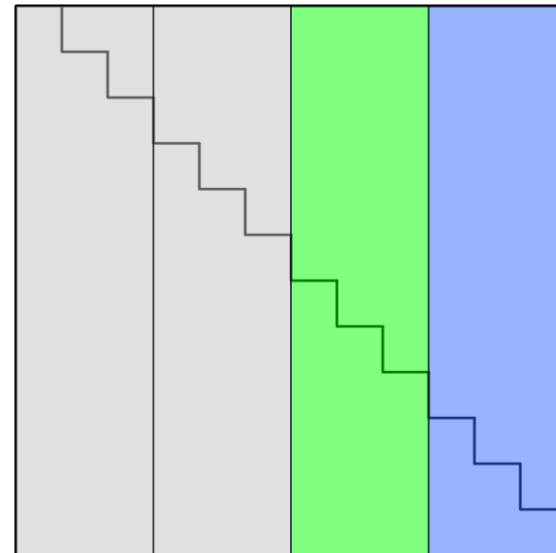
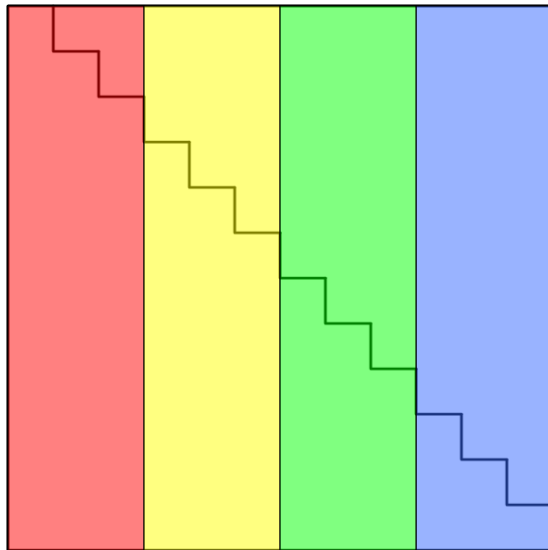
end for

end for



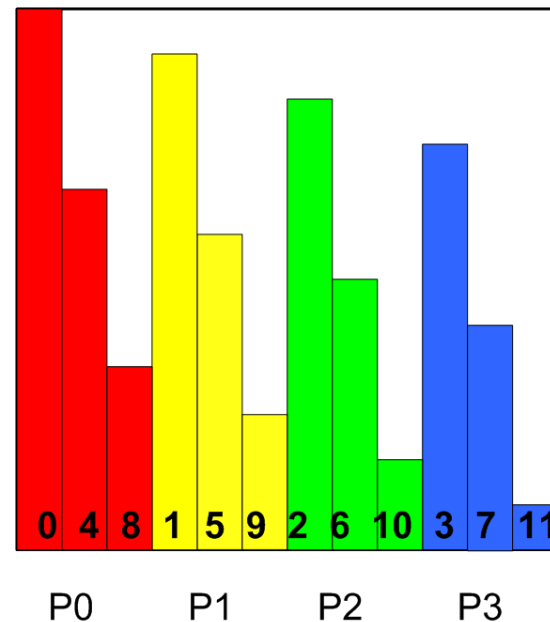
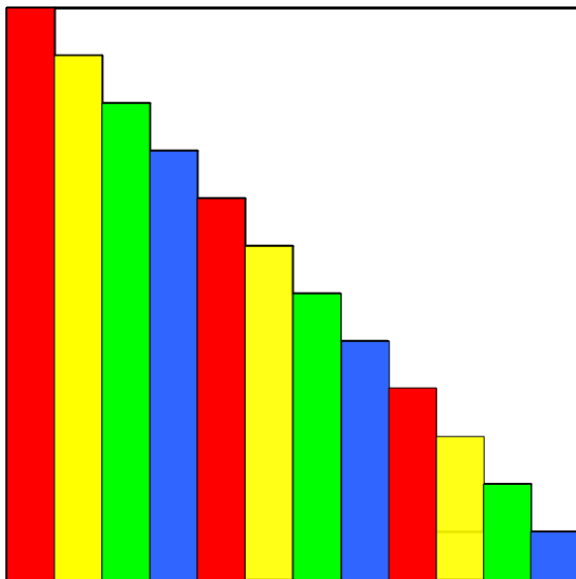
Performance

- Finding the pivot row is a serial bottleneck
 - Only one thread owns the intersecting column
- Another bottleneck is load imbalance
 - When eliminating a column, processors to its left sit idle
 - Each processor is active for only part of the computation



Cyclic decomposition improves load balance

- A *cyclic* decomposition evens out the workload
- A blocked cyclic decomposition improves locality and reduces communication overhead



In practice

- 1D is not scalable; 2D block cyclic decompositions required
- More complicated since additional communication steps are required
- Used in Scalapack
www.netlib.org/scalapack

