

CSE 260

Lecture 16

Sorting
Gaussian Elimination

Announcements

- Project Progress report, Today, up to 5% extra credit
- No Office Hours Today, make an appointment for Friday

Today's lecture

- Sorting (application of collectives)
- Matrix Factorization (Linear algebra, block cyclic decomposition)

Parallel Sorting

- Sorting is fundamental algorithm in data processing
 - ◆ Given an unordered set of keys $x_0, x_1, \dots, x_{N-1}$
 - ◆ Return the keys in sorted order
- The keys may be character strings, floating point numbers, integers, or any object for which the relations $>$, $<$, and $==$ hold
- In practice, we sort on external media,
[See: http://sortbenchmark.org](http://sortbenchmark.org)
 - ◆ **TritonSort (UCSD):** 0.725×10^{12} bytes/minute

Rank sorting

- Compute the rank of each input value
- Move each value in sorted position according to its rank
- On an ideal parallel computer, the **forall** loops parallelize perfectly

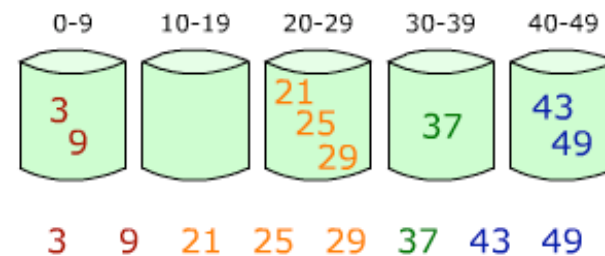
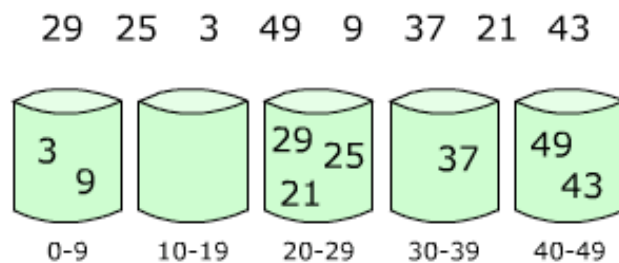
```
forall i=0:n-1, j=0:n-1  
    if ( x[i] > x[j] ) then rank[i] += 1 end if  
forall i=0:n-1  
    y[rank[i]] = x[i]
```

In search of a fast and practical sort

- Rank sorting is impractical on real hardware
- Let's borrow the concept: compute the processor owner for each key
- Communicate data in sorted order in one step
- But how do we know which processor is the owner?
- Depends on the distribution of keys

Bucket sort

- Divide key space into equal subranges and associate a bucket with each subrange
- Unsorted input data distributed evenly over processors
- Each processor maintains p local buckets
 - ♦ Assigns each key to a local bucket: $\lfloor p \times \text{key} / (K_{\max} - 1) \rfloor$
 - ♦ Routes the buckets to the correct owner (each local bucket has $\sim n/p^2$ elements)
 - ♦ Sorts all incoming data into a single bucket

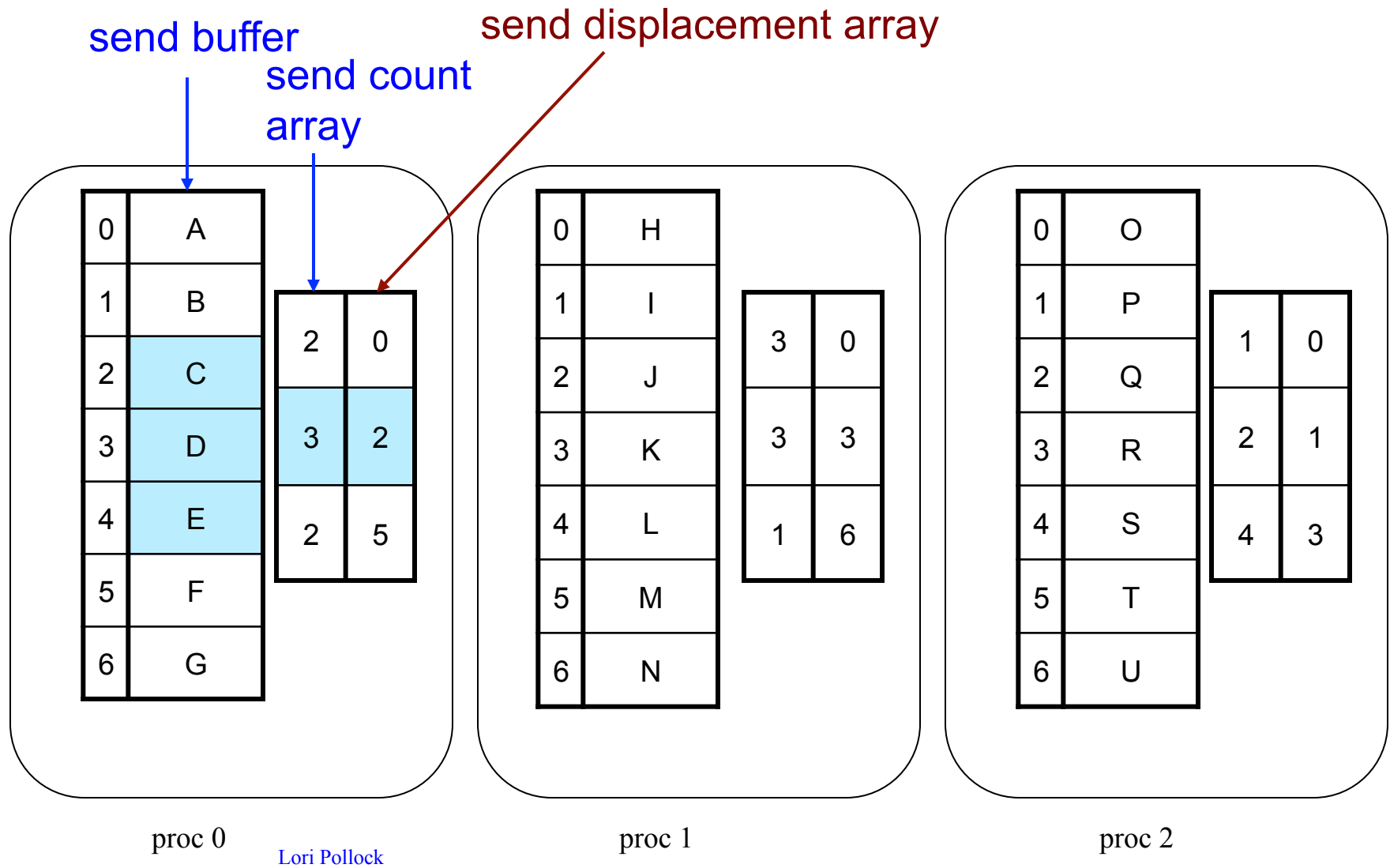


Wikipedia

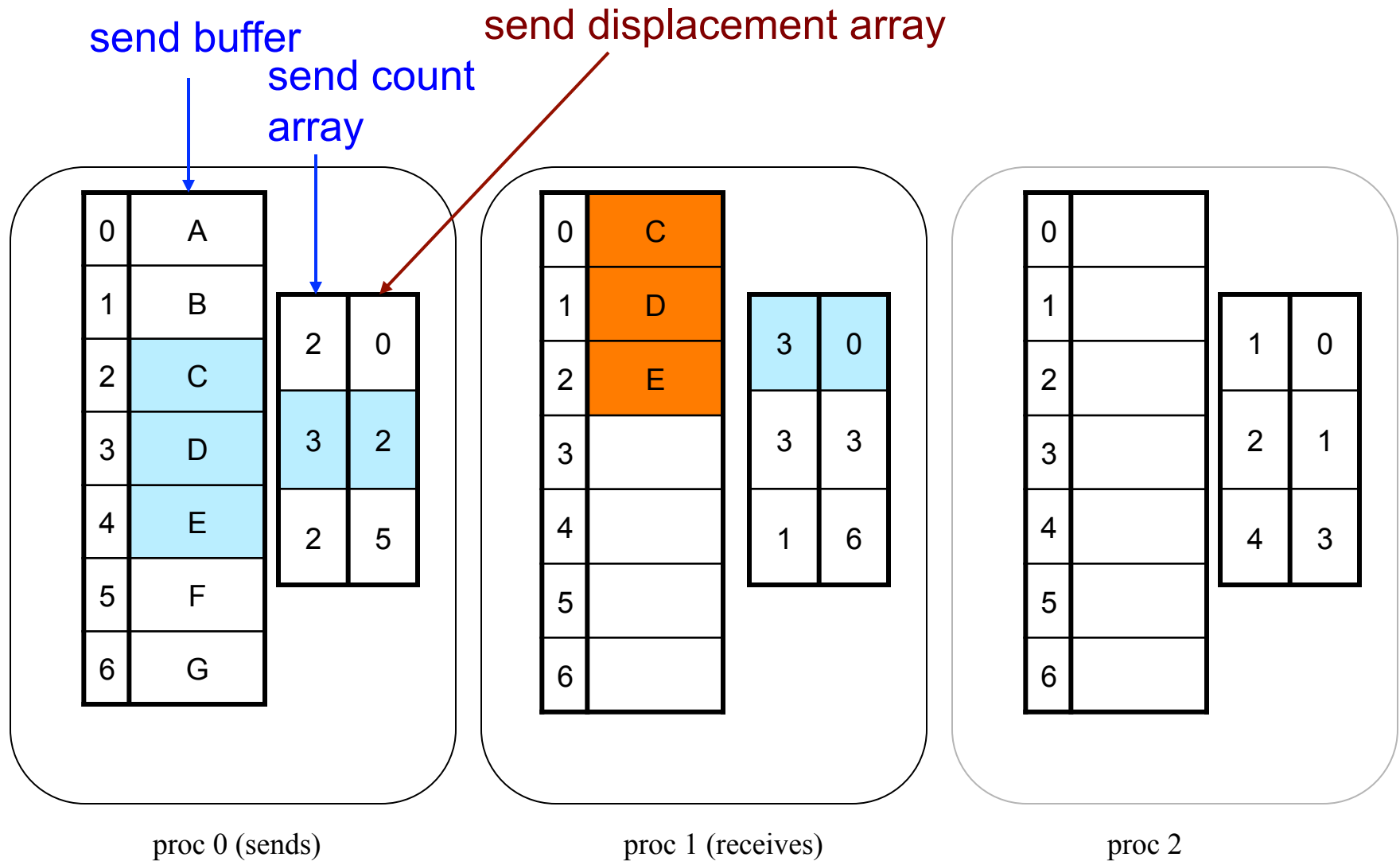
“Vector” All to All routes the data

- Processors maintain send and receive counts
- `MPI_Alltoallv` (
void *sendbuf, int sendcounts[], int sDispl [],
MPI_Datatype sendtype,
void* recvbuf, int recvcnts[], int rDispl[],
MPI_Datatype recvtype, MPI_Comm comm)

Alltoally example



Alltoally example



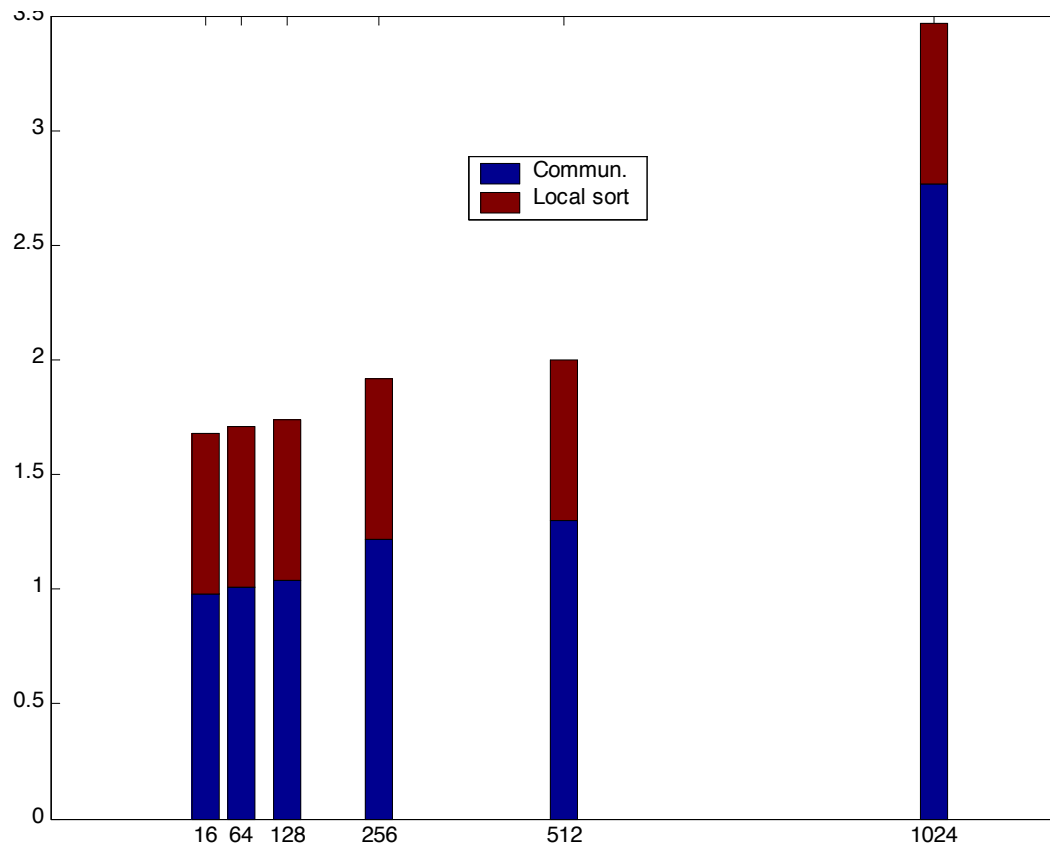
Lori Pollock

Running time

- Assume that the keys are distributed uniformly over 0 to $K_{\max}-1$
- Local bucket assignment: $O(n/p)$
- Route each local bucket to the correct owner
All to all: $O(n)$
- Local sorting : $O(n/p)$
 - ◆ Radix sort
 - ◆ www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix

Scaling study

- IBM SP3 system: 16-way servers w/ Power 3 CPUs
- Weak scaling : 1M points per processor



Local sort: quicksort
 $O(n/p \log(n/p))$

All-to-all
 $O(n)$

Worst case behavior

- What is the worst case?
- Mapping of keys to processors based on knowledge of $K_{\max}$
- If keys are in range $[0, Q-1]$...
... processor k has keys in the range $[k*Q/P : (k+1)*Q/P]$
- For $Q=2^{30}$, $P=64$, each processor gets $2^{24} = 16$ M elements
- What if keys $\in [0, 2^{24}-1] \subset [0, 2^{30}-1]$?
- But if the keys are distributed non-uniformly, we need more information to ensure that the keys (and communication) are balanced over the processors
- Sample sort is an algorithm that collects such information and improves worst case behavior

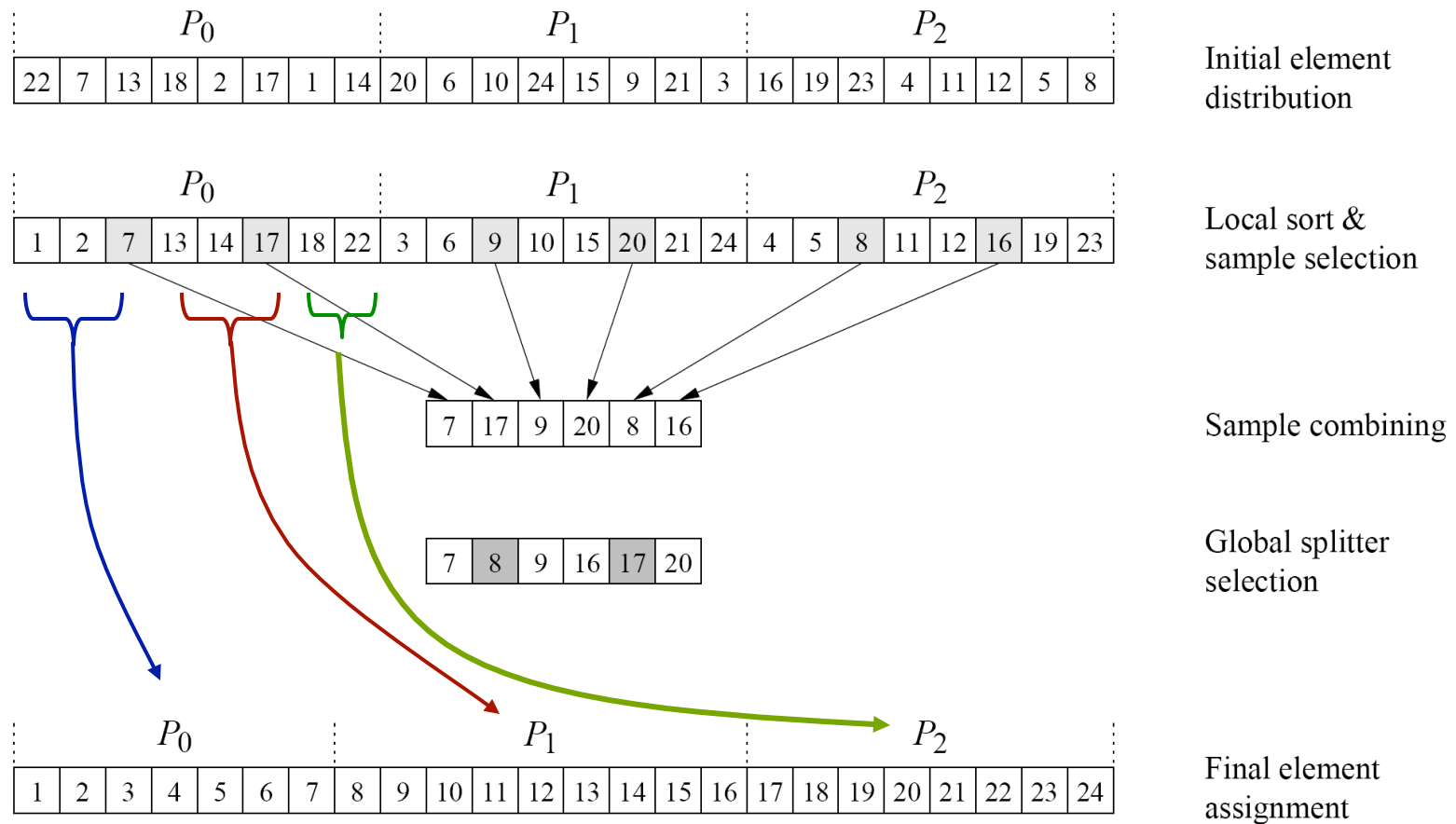
Improving on bucket sort

- *Sample sort* remedies the problem
- “Parallel Sorting by Regular Sampling.” H. Shi and J. Schaeffer. *J. Parallel and Distributed Computing*, 14:361-372, 1992
- “Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study.”
D. R. Helman, D.A. Bader, and J. JáJá,
Proc. SPAA: Annual ACM Symp. on Parallel Algorithms and Architectures (1996)
<http://www.umiacs.umd.edu/research/EXPAR/papers/spaa96.html>

The idea behind sample sort

- Use a heuristic to estimate the distribution of the global key range over the p processors processor so that...
- ...each processor gets about the same number of keys
- Sample the keys to determine a set of $p-1$ **splitters** that partition the key space into p disjoint intervals
[sample size parameter: s]
- Each interval is assigned a unique processor mapped to a bucket
- Once each processor knows the splitters, it can distribute its keys to the others accordingly
- Processors sort incoming keys

Alltoallv used in sample sort



Introduction to Parallel Computing, 2nd Ed., A.Grama, A.I Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

Splitter selection: regular sampling

- After sorting local keys, each processor chooses p evenly spaced samples
- Each processor “deals” its sorted data into one of p bins
 - ♦ The k^{th} item is placed into position $\lceil k/p \rceil$ of bin $k \bmod p$
 - ♦ When done, each sends bin j to processor j
- This is like a transpose with block sizes $= n/p^2$
- Each processor receives **p sorted subsequences**
- Processor $p-1$ determines the splitters
 - ♦ It samples each **sorted subsequence**, taking every $(kn/(p^2s))^{\text{th}}$ element ($1 \leq k \leq s-1$), where $p \leq s \leq n/p^2$
 - ♦ Merges the sampled sequences, and collects $p-1$ regularly spaced splitters
 - ♦ Broadcasts the splitters to all processors
- Processors route (exchange) **sorted subsequences** according to the splitters (transpose)
- The data are unshuffled

Performance

- Assuming $n \geq p^3$ and $p \leq s \leq n/p^2$
- Running time is $\approx O((n/p) \lg n)$
- With high probability ...
no processor holds more than $(n/p + n/s - p)$ elements
- Duplicates d do not impact performance unless $d = O(n/p)$
- Tradeoff: increasing s ...
 - ♦ Spreads the final distribution more evenly over the processors
 - ♦ Increases the cost of determining the splitters
- For some inputs, communication patterns can be highly irregular with some pairs of processors communicating more heavily than others
- This imbalance degrades communication performance

The collective calls

- Processes transmit varying amounts of information to the other processes
- This is an `MPI_Alltoallv`
(`SKeys`, `send_counts`, `send_displace`, `MPI_INT`,
`RKeys`, `recv_counts`, `recv_displace`, `MPI_INT`,
`MPI_COMM_WORLD`)
- Prior to making this call, all processes must cooperate to determine how much information they will exchange
 - ♦ The *send list* describes the number of keys to send to each process k, and the offset in the local array
 - ♦ The *receive list* describes the number of incoming keys for each process k and the offset into the local array

Determining the send and receive lists

- After sorting, each process scans its local keys from left to right, marking where the splitters divide the keys, in terms of **send counts**
- Perform an all to all to transpose these send counts into **receive counts**

```
MPI_Alltoall(send_counts, 1, MPI_INT,  
             recv_counts, 1, MPI_INT, MPI_COMM_WORLD)
```

- A simple loop determines the displacements

```
for (p=1; p < nodes; p++){  
    s_displ[p] = s_displ[p-1] + send_counts[p-1];  
    r_displ[p] = r_displ[p-1] + recd_counts[p-1];  
}
```

Today's lecture

- Sorting (application of collectives)
- **Matrix Factorization (Linear algebra, block cyclic decomposition)**

Linear systems of equations

- A common task in scientific computation is to solve a system of linear equations
- Often result from discretizing a differential equation
- Example: linear system of 2 equations in 2 unknowns

$$(1) \quad 2x + 3y = 8$$

$$(2) \quad 3x + 2y = 7$$

- Rewriting equation (1)

$$x = (8-3y)/2$$

- Substituting x into the LHS of equation (2)

$$3(8-3y)/2 + 2y = (24-9y)/2 + 2y$$

$$\Rightarrow (24-9y) + 4y = 14 \Rightarrow 10 = 5y \Rightarrow y = 2$$

- Back substituting the value of y into equation (1)

$$x = 1$$

Matrix vector equations

- Our linear system of 2 equations in 2 unknowns ...
$$2x_1 + 3x_2 = 8$$
$$3x_1 + 2x_2 = 7$$
- may be conveniently expressed in 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 solved for $x_1 = (8 - 3x_2)/2$ and substituted the value of x_1 into the 2nd equation, we reduced the matrix to an equivalent form

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

- We multiplied row 1 of A by 3/2 and subtracting the scaled version from row 2 of A and **b**
- We call this a *rank-1 update*

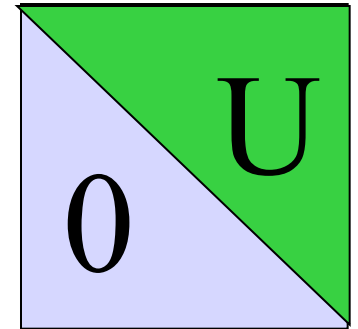
Rank 1 updates

- Multiplying row 1 by 3/2: $[3 \quad 9/2]$
- Subtracting from row 2:
- Similarly for **b**

$$A = \begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}$$
$$A' = \begin{pmatrix} 2 & 3 \\ 0 & -2.5 \end{pmatrix}$$

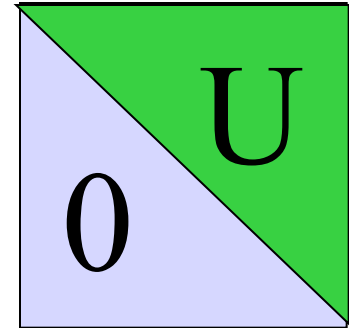
Gaussian Elimination

- 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: an $n \times n$ matrix with zeroes under the main diagonal: an ***upper triangular matrix 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*



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 $\mathbf{y} = U\mathbf{x}$

$$A = L U$$

Cost

- To solve $A\mathbf{x} = \mathbf{b}$
 - ♦ 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 $\mathbf{b}$
- Focus on factorization, which is much more expensive

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

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

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

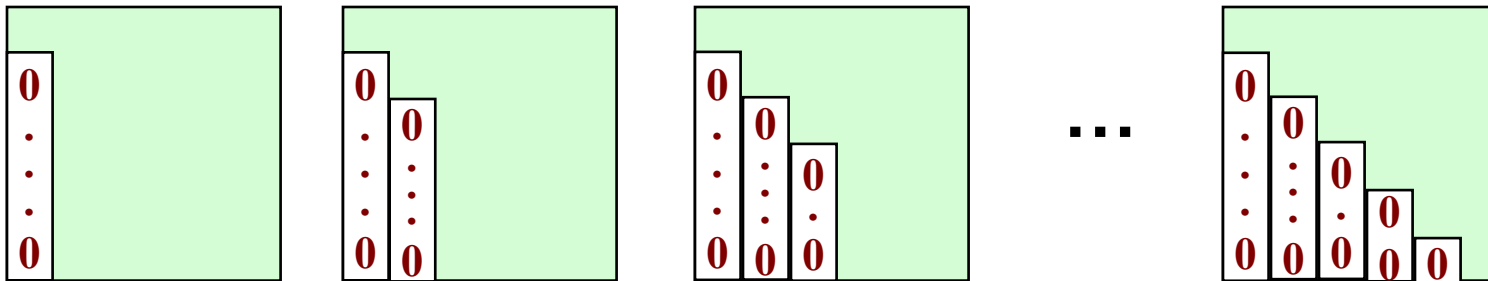
$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & -1 & 0 & -4 \\ 0 & 0 & -5 & 4 \end{array} \right]$$

Gaussian Elimination (GE)

- Add multiples of each row to later rows to make A upper triangular
 - ... for each column k
 - ... zero it out below the diagonal by adding multiples of row k to later rows
- ```

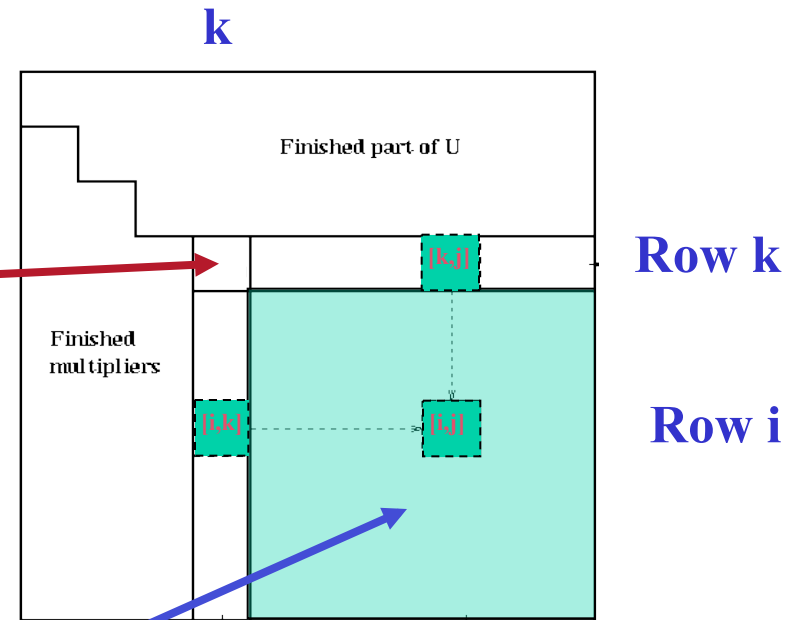
for k = 0 to n-1
 ... for each row i below row k
 for i = k+1 to n-1
 ... add a multiple of row k to row i
 for j = k+1 to n-1
 $A[i,j] := A[i, k] / A[k,k] * A[k,j]$

```



# Eliminating the entries below the diagonal

- For each column  **$k : 0$  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:  
 **$A[k+1:n-1, k]$**
- Update only to the right of & below  **$A[k, k]$**



for  $i = k+1$  to  $n-1$   
 $A[i, k+1:n] - = m_{ik} \times A[k, k+1:n]$

## Roundoff issues

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

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

- We need to be able 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*
- Assume we carry 3 decimal digits of precision
- Consider the following A matrix and RHS b

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

- The correct solution is

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

## Roundoff Error

- Eliminate zero in row 2 by subtracting  $10^4 \times$  row 0

$$\mathbf{L}|\mathbf{b} = \left[ \begin{array}{cc|c} 10^{-4} & 1 & 1 \\ 0 & 1 - 10^4 & 2 - 10^4 \end{array} \right]$$

- But  $1 - 10^4$  rounds to  $-10^4$

$$\mathbf{L}|\mathbf{b} = \left[ \begin{array}{cc|c} 10^{-4} & 1 & 1 \\ 0 & -10^4 & -10^4 \end{array} \right]$$

- Back substituting to solve for  $x_2$  and then  $x_1$

$$-10^4 x_2 = -10^4 \Rightarrow \mathbf{x_2 = 1}$$

- Substituting the value of  $x_2$  into the first equation

$$10^{-4} x_1 + 1 * x_2 = 1 \Rightarrow 10^{-4} x_1 = 0 \Rightarrow \mathbf{x_1 = 0}$$

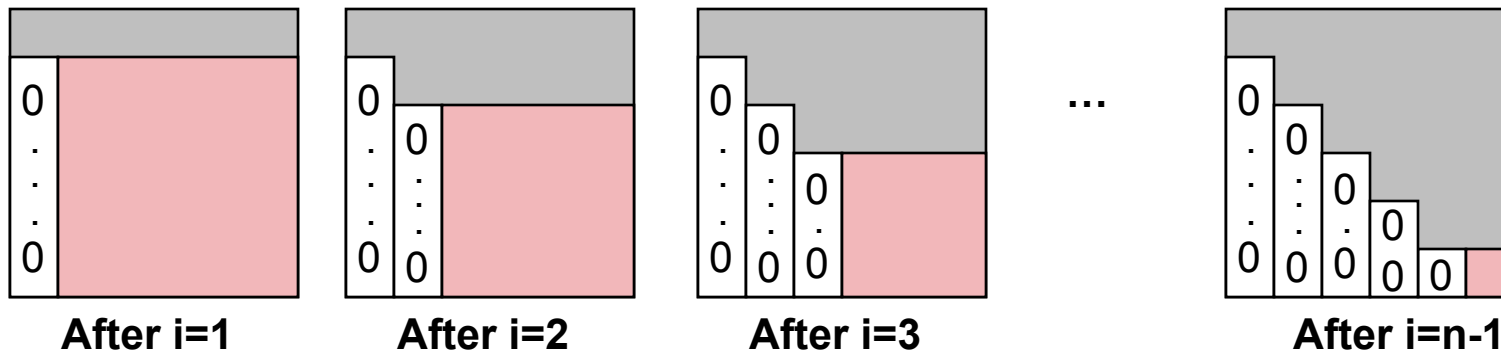
- **But the correct solution is  $x_1 = x_2 = 1$**

# 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 when with partial pivoting, we compute  $\mathbf{PA} = \mathbf{L} \mathbf{U}$ , where  $\mathbf{P}$  is a permutation matrix expressing the rows swaps
- We can also swap columns: *full pivoting*  
 $\mathbf{PAQ} = \mathbf{L} \mathbf{U}$
- But full pivoting is more expensive to implement

# Parallelization

- We'll use 1D vertical strip partitioning
- Each process owns  $N/p$  columns
- Consider the case where  $p=N=6$
- The ■ represents outstanding work in succeeding  $k$  iterations



## Parallelism and data dependencies

- Analyze the code to determine communication requirements
- Assume blocked decomposition on the 2<sup>nd</sup> axis
- Each process in charge of eliminating N/P columns
- Parallelism occurs in *array statements*
- One process chooses pivot row, computes multipliers

```
for k = 0 to n-1 // For each column k
 m[k+1:n-1] = A[k+1:n-1,k] / A[k , k] // Compute Multipliers

 for i = k+1 to n-1 // for each row i > k
 A[i , k+1: n-1] -= m[i] * A[k , k+1:n-1] // Scale row k by mik
 // & subtract from row i
 end for
end for
```

# Determining communication requirements

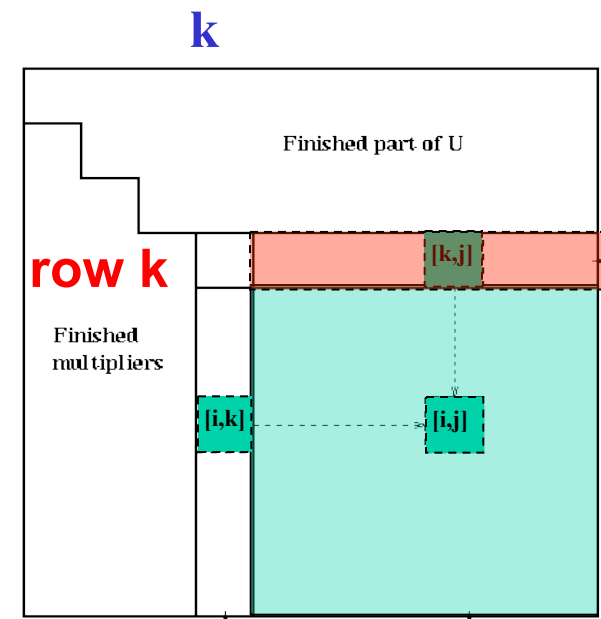
- At each step **k** of the elimination, processor  **$k \div p$**  is in charge: it computes the multipliers
- No communication is needed: all the required data are *owned* by processor  **$k \div p$**

```

for k = 0 to n-1
 m[k+1:n-1] = A[k+1:n-1,k] / A[k , k]

 for i = k+1 to n-1
 A[i , k+1:n-1] -= m[j] * A[k ,k+1:n-1]
 end for
end for

```



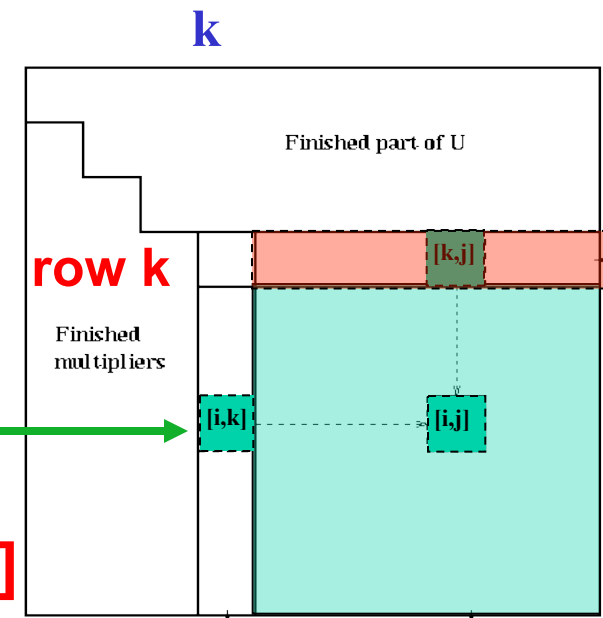
# Commination in trailing matrix update

- Elements in  $A[k, k+1:n]$  (row  $k$ ) have different owners
- Process  $j \text{ div } p$  owns  $A[k,j]$  in  $A[k, k+1:n]$
- Process  $k \text{ div } P$  broadcasts multipliers enabling all to carry out the multiplication  $m[j] * A[k, :]$

```

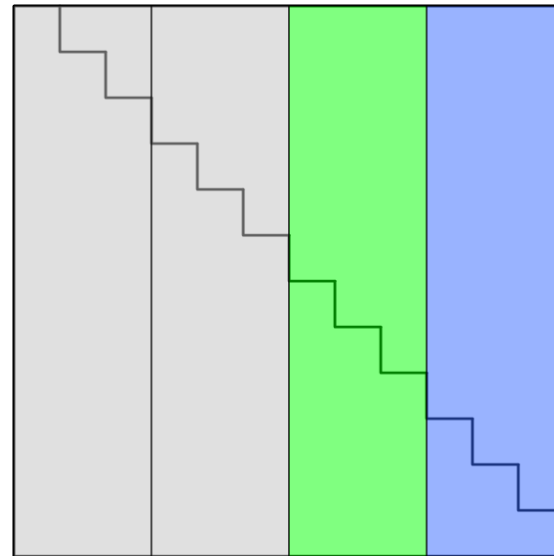
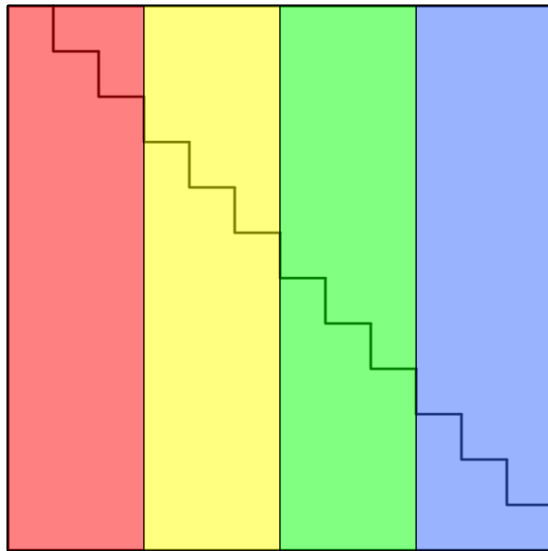
for k = 0 to n-1
 m[k+1:n-1] = A[k+1:n-1,k] / A[k, k]
 // Scale row k by mik and
 // subtract from row i
 for j = k+1 to n-1 // for each row i > k
 A[j, k+1:n-1] -= m[j] * A[k, k+1:n-1]
 end for
end for

```



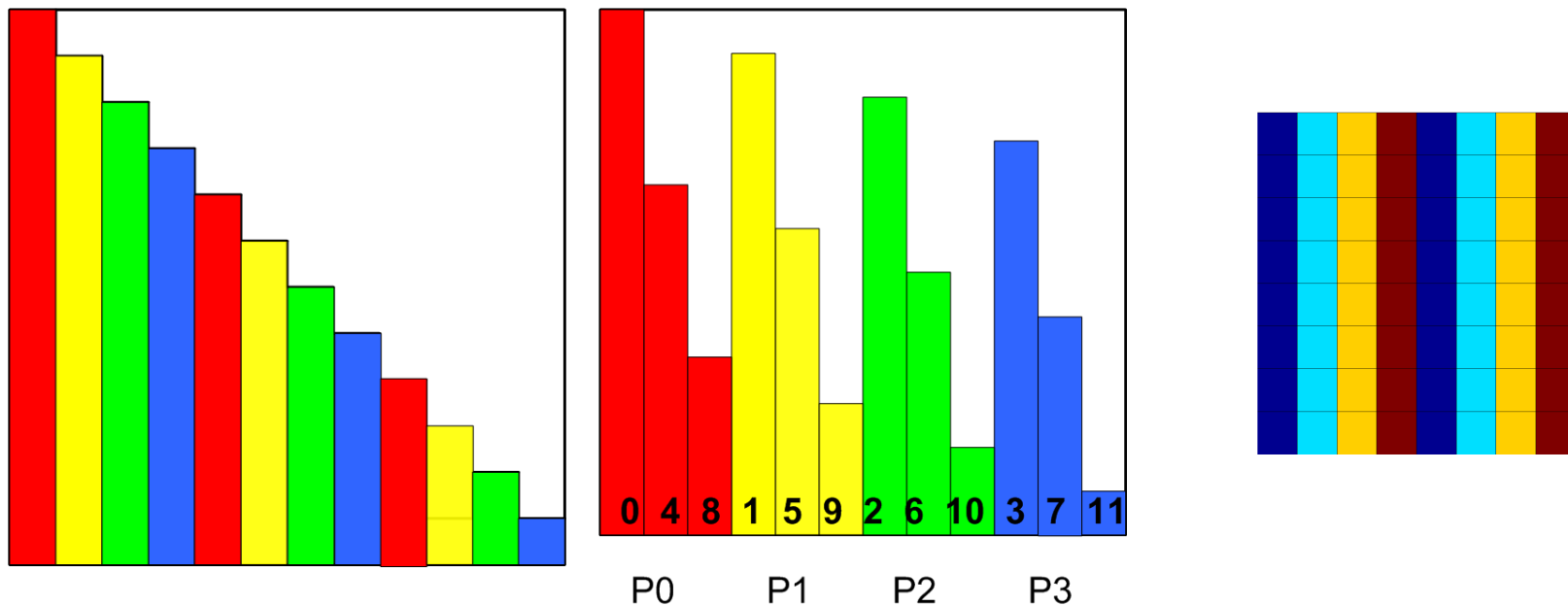
# Performance

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



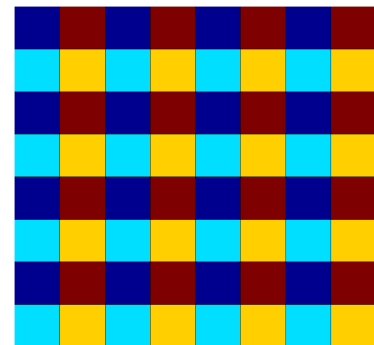
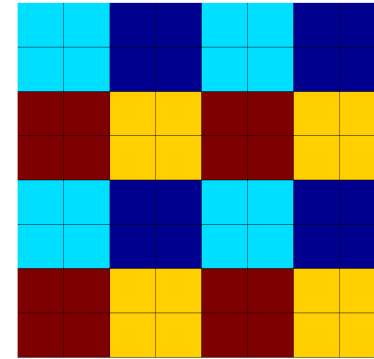
# Cyclic decomposition improves load balance

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



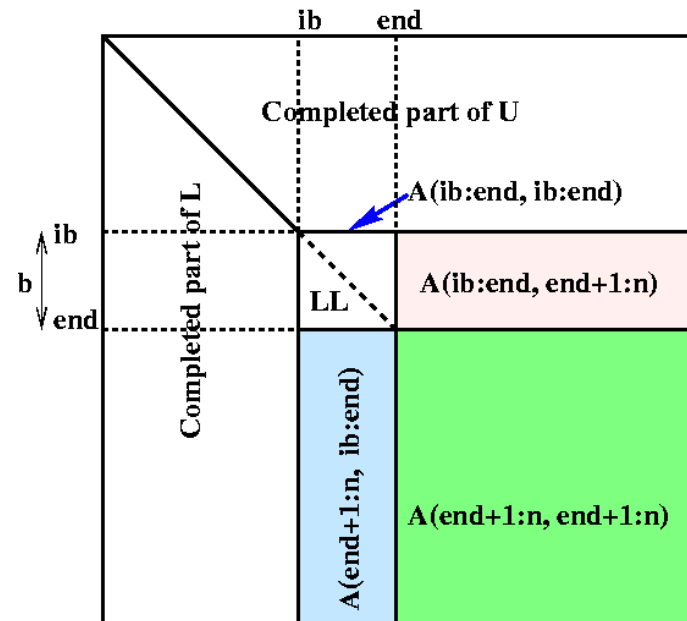
# Data Partitioning

- 2D block cyclic decompositions scale better than 1D
- More complicated since additional communication steps are required
- Improved locality as the block size increases
- The algorithm is blocked as with matrix multiply
- ScaLAPACK is a well known library that performs GE and many other useful operations involving matrices
- See <http://www.netlib.org/scalapack>



# Blocked algorithm

- Our algorithm makes BLAS1 updates, rows & columns only
- Performance is poor: like the *ijk* matrix multiply algorithm
- We can block GE, too, using BLAS level 3, though more complicated than MMPY because of dependencies
- We use “delayed updates” on trailing submatrix
- We update the trailing matrix with the effect of multiple updates in the original algorithm



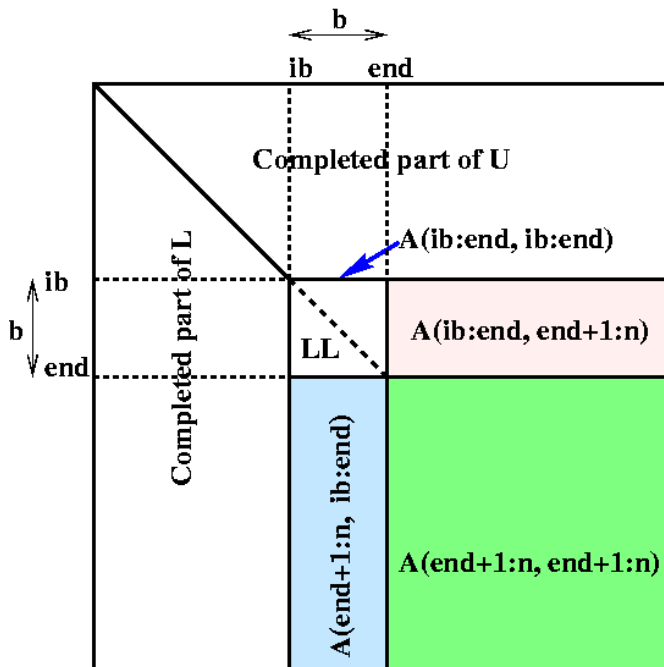
# Blocked algorithm

```

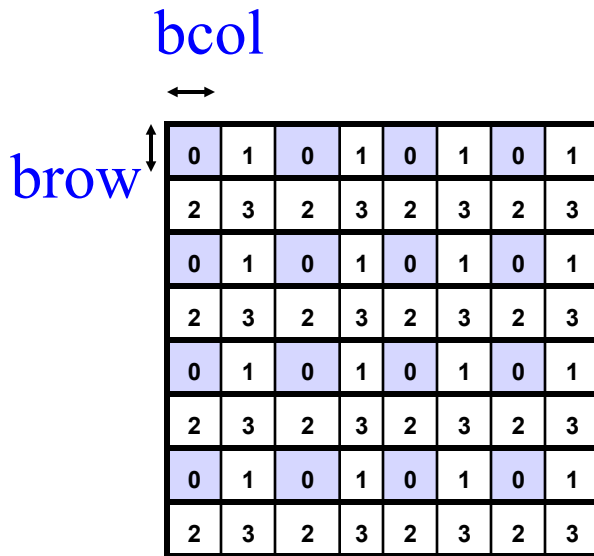
for ib = 1 to n-1 step b ... Process matrix b columns at a time
 end = ib + b-1 ... end of block of b columns
 apply BLAS2 version of GEPP to get $A(ib:n, ib:end) = P' * L' * U'$
 ... $LL = \text{lower triangular part of } A(ib:end, ib:end) + I$
 $A(ib:end, end+1:n) = LL^{-1} * A(ib:end, end+1:n)$... update next b rows of U
 $A(end+1:n, end+1:n) = A(end+1:n, end+1:n)$
 - $A(end+1:n, ib:end) * A(ib:end, end+1:n)$
 ... apply delayed updates with single matrix-multiply
 ... with inner dimension b

```

Gaussian Elimination using BLAS 3

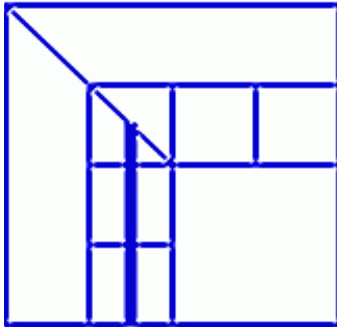


# Row and Column Block Cyclic Layout

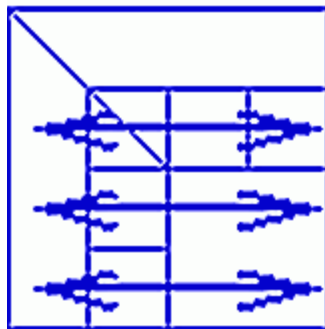


- Processors and matrix blocks are distributed in a 2d array
  - $\text{prow} \times \text{pcol}$  array of processors
  - $\text{brow} \times \text{bcol}$  matrix blocks
  - adjust block size to ensure that  $b$  columns of  $A$  fit in cache
- pcol-way parallelism in a column
- calls to BLAS2 and BLAS3 on matrices of size  $\text{brow} \times \text{bcol}$
- Reduces serial bottleneck
- $\text{prow} \neq \text{pcol}$  and  $\text{brow} \neq \text{bcol}$  possible, possibly desirable

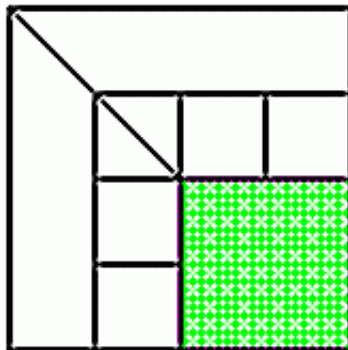
# Distributed GE with a 2D Block Cyclic Layout



Choose pivot, compute multipliers,  
make swap decision



Broadcast swap decision and multipliers  
so each block column can participate



Carry out the elimination

Follows Demmel's discussion

## Distributed Gaussian Elimination with a 2D Block Cyclic Layout

for  $ib = 1$  to  $n-1$  step  $b$

end =  $\min(ib+b-1, n)$

for  $i = ib$  to end

(1) find pivot row  $k$ , column broadcast

(2) swap rows  $k$  and  $i$  in block column, broadcast row  $k$

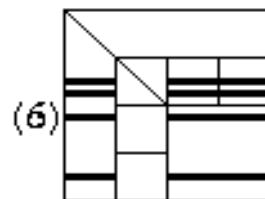
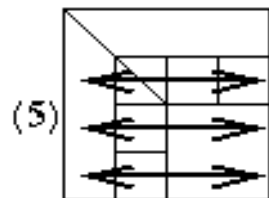
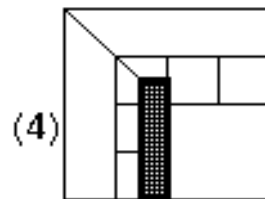
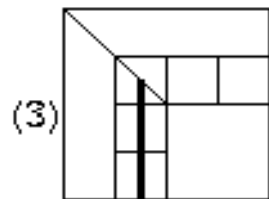
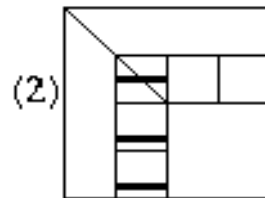
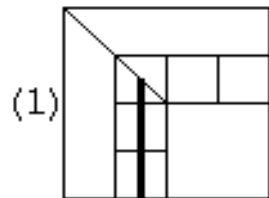
(3)  $A(i+1:n, i) = A(i+1:n, i) / A(i, i)$

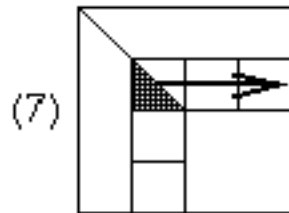
(4)  $A(i+1:n, i+1:end) -= A(i+1:n, i) * A(i, i+1:end)$

end for

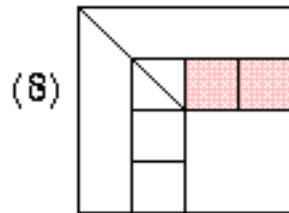
(5) broadcast all swap information right and left

(6) apply all rows swaps to other columns

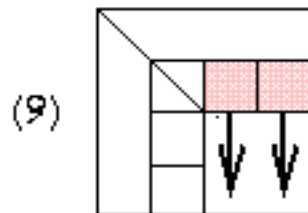




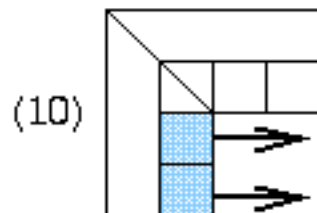
(7) Broadcast LL right



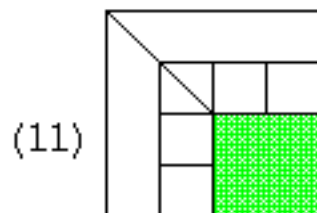
(8)  $A(ib:end, end+1:n) = LL \setminus A(ib:end, end+1:n)$



(9) Broadcast  $A(ib:end, end+1:n)$  down



(10) Broadcast  $A(end+1:n, ib:end)$  right



(11) Eliminate  $A(end+1:n, end+1:n)$

Matrix multiply of  
 $green = green - blue * pink$