



Parallel Algorithms for Dense Matrices

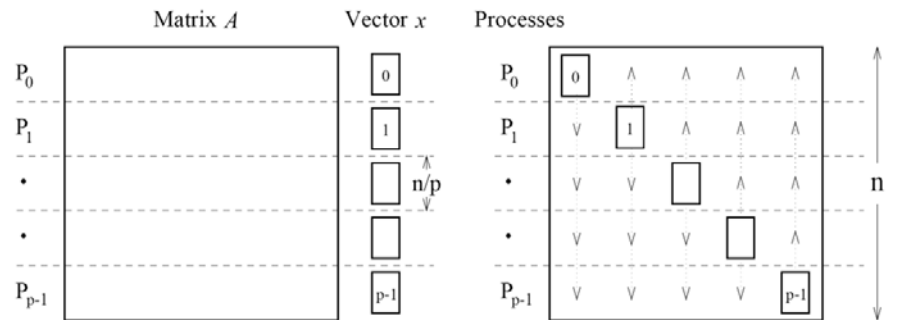
CS121 Parallel Computing
Spring 2021



Dense matrices

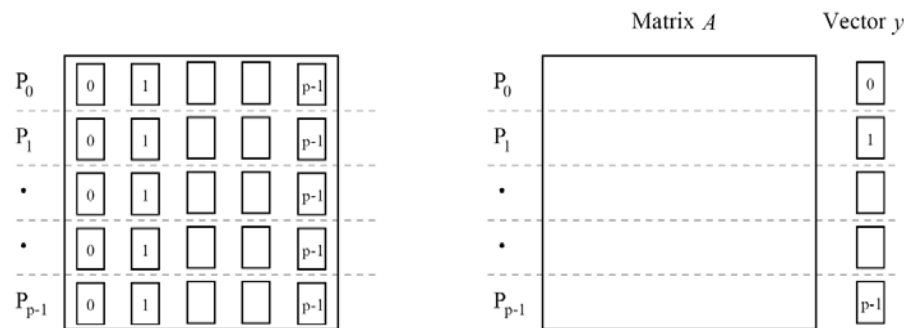
- Dense matrices are ones where most elements are nonzero.
- Dense matrices are used in many optimization problems, and physics and chemistry based simulations.
- Many dense matrix operations are compute bound. This, plus their highly regular structure, allows dense matrix operations to be highly optimized.
- High performance parallel implementations provided by LAPACK, ScaLAPACK, HPL, etc.
- We'll look at algorithms for matrix-vector multiplication, matrix-matrix multiplication and equation solving (Gaussian elimination).

1D matrix-vector multiplication



(a) Initial partitioning of the matrix and the starting vector x

(b) Distribution of the full vector among all the processes by all-to-all broadcast



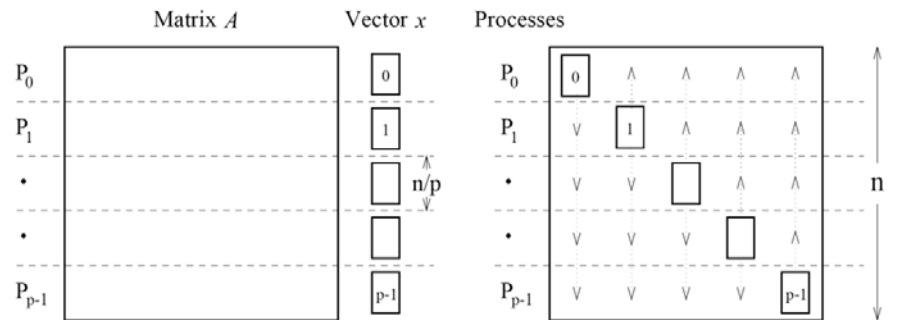
(c) Entire vector distributed to each process after the broadcast

(d) Final distribution of the matrix and the result vector y

Source: Introduction to Parallel Computing,
Grama et al

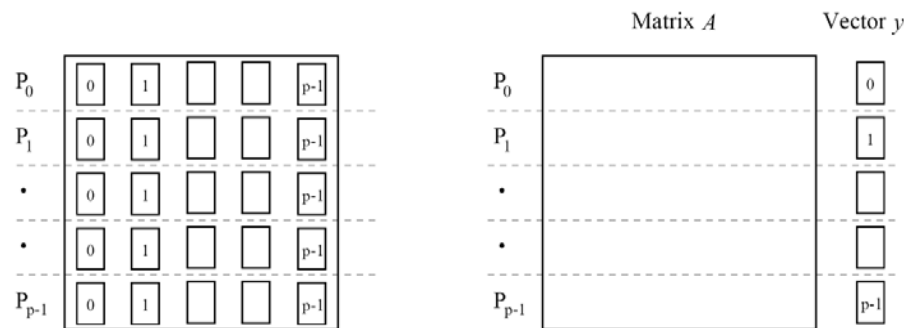
- Consider an $n \times n$ matrix A and an $n \times 1$ vector x partitioned across $p \leq n$ processors.
 - Initially each process stores n/p rows of A and n/p values of x .
- Each process needs the entire vector to multiply by its rows.
 - In step (b), do all-to-all broadcast of the processors' vector segments.
- Each process multiplies the vector by its rows.
 - Each process ends up with n/p values of the output.

1D matrix-vector multiplication



(a) Initial partitioning of the matrix and the starting vector x

(b) Distribution of the full vector among all the processes by all-to-all broadcast

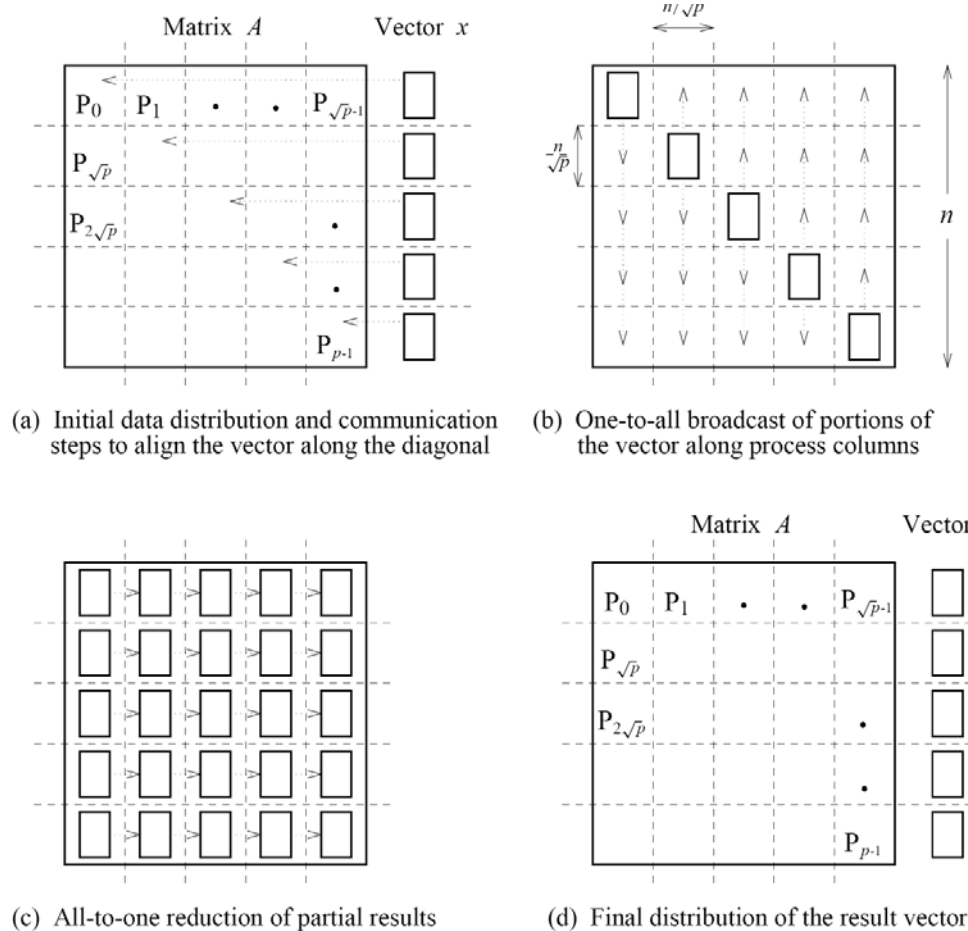


(c) Entire vector distributed to each process after the broadcast

(d) Final distribution of the matrix and the result vector y

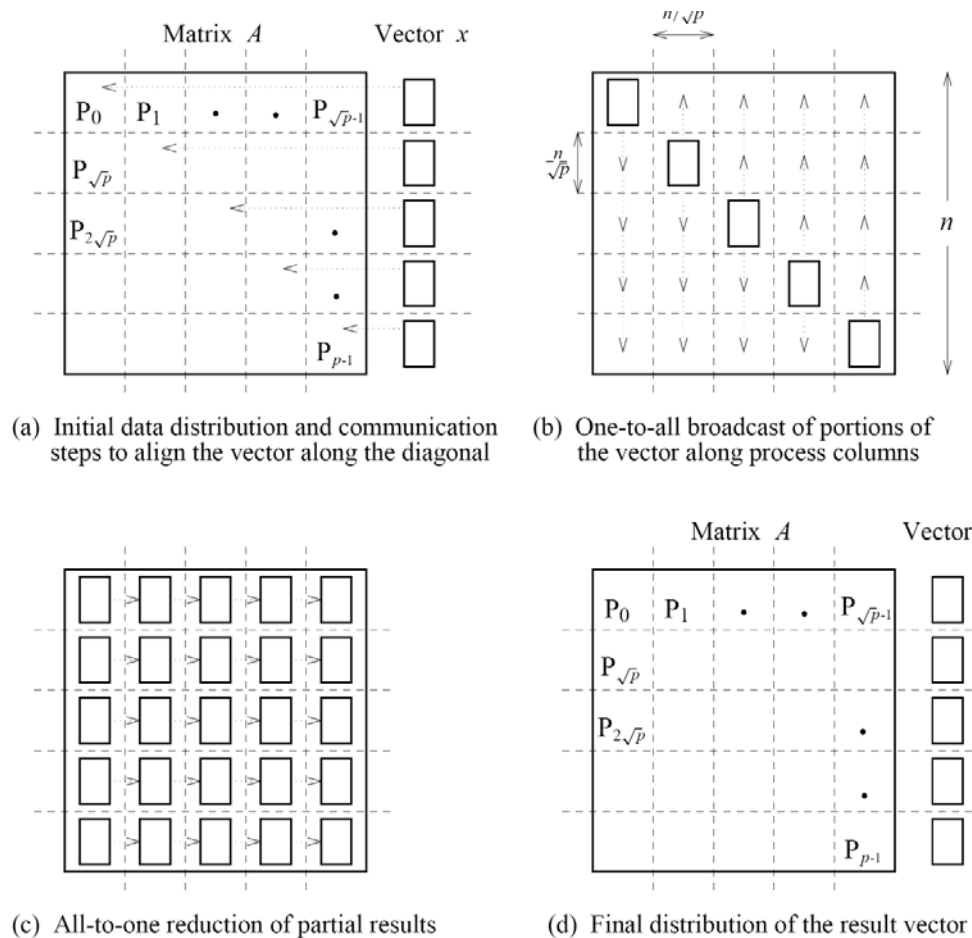
- Assume an underlying hypercube architecture.
- Compute time for each process is $\frac{n^2}{p}$.
- Communication time for all-to-all broadcast is $t_s \log p + t_w n$.
- Overhead by all p processors from communication is $t_s p \log p + t_w n p$.
- Total amount of work is n^2 .
- For isoefficiency, need $n^2 = \Omega(t_s p \log p + t_w n p)$.
 - This is satisfied for $p = O(n)$.

2D matrix-vector multiplication



- Consider a logical 2D mesh of p processes, each initially with a $(n/\sqrt{p}) \times (n/\sqrt{p})$ portion of the matrix.
- The vector is stored only in the last column. Each process in last column has (n/\sqrt{p}) vector values.
- In (a), each process in last column sends its vector elements to a process on the diagonal.
- In (b), each diagonal process does a one-to-all broadcast of the vector chunk.
- In (c), each process multiplies its row chunks by its vector chunks, producing (n/\sqrt{p}) partial values. Then each row of processes does a reduction of the partial values to the last process in the row.
- The output is stored in the last column of processes.

2D matrix-vector multiplication



- Each process stores n^2/p matrix values, and does one multiplication and one addition for each value. So the total computation is $O(n^2/p)$.
- Communication time
 - (a) $t_s + t_w n/\sqrt{p}$.
 - (b) and (c) $(t_s + t_w n/\sqrt{p})\log(\sqrt{p})$.
- Total time per process is $O\left(\frac{n^2}{p} + \log p + \frac{n}{\sqrt{p}}\log p\right)$.
- Communication overhead is $p \log p + n\sqrt{p} \log p$.
- Isoefficiency requires $n^2 = \Omega(\text{overhead})$
 - So $n^2 = \Omega(n\sqrt{p} \log p)$, so $n = \Omega(\sqrt{p} \log p)$, so $n^2 = \Omega(p \log^2 p)$.
 - So need $p = O\left(\frac{n^2}{\log^2 p}\right)$.
- 2D matrix-vector multiplication is more scalable than 1D.

Matrix-matrix multiplication

```

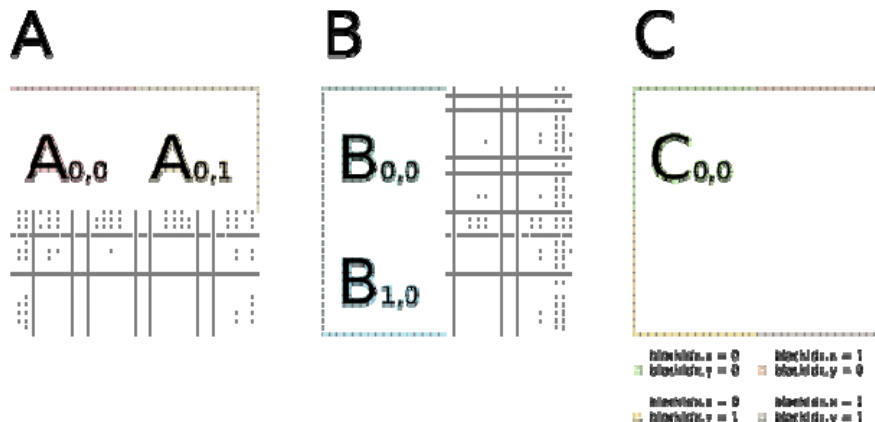
1. procedure MAT_MULT (A, B, C)
2. begin
3.   for i := 0 to n - 1 do
4.     for j := 0 to n - 1 do
5.       begin
6.         C[i, j] := 0;
7.         for k := 0 to n - 1 do
8.           C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.         endfor;
10.  end MAT_MULT

```

```

1. procedure BLOCK_MAT_MULT (A, B, C)
2. begin
3.   for i := 0 to q - 1 do
4.     for j := 0 to q - 1 do
5.       begin
6.         Initialize all elements of Ci,j to zero;
7.         for k := 0 to q - 1 do
8.           Ci,j := Ci,j + Ai,k × Bk,j;
9.         endfor;
10.  end BLOCK_MAT_MULT

```



- Matrix multiplication can be done element by element, or by breaking the matrices into blocks and multiplying block by block.
- We partition A and B into $\sqrt{p} \times \sqrt{p}$ blocks, each of size $(n/\sqrt{p}) \times (n/\sqrt{p})$.
- Suppose p processes form a $\sqrt{p} \times \sqrt{p}$ mesh.
- Each process stores the corresponding block from A and B.
- Each $C_{i,j}$ requires $A_{i,k}$ and $B_{k,j}$ for $1 \leq k \leq \sqrt{p}$.
 - So row i processes do all-to-all broadcast of their A blocks, and column j processes do all-to-all broadcast of their B blocks.
- Each process (i,j) ends up with $C_{i,j}$ stored locally.

Matrix-matrix multiplication

```

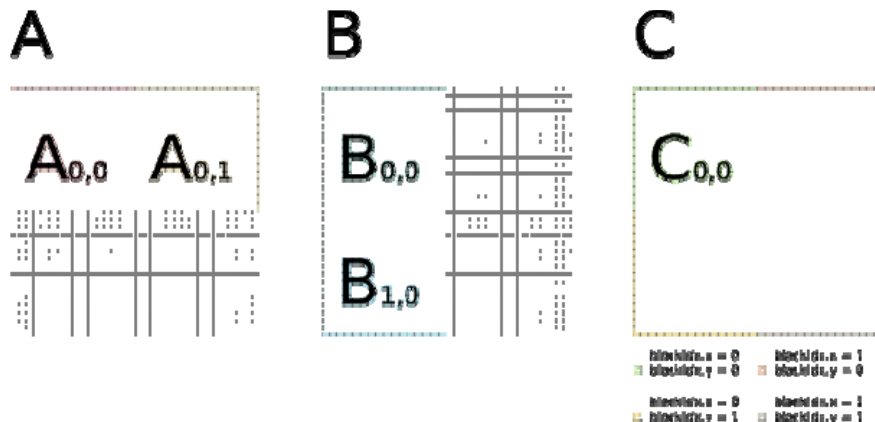
1. procedure MAT_MULT (A, B, C)
2. begin
3.   for i := 0 to n - 1 do
4.     for j := 0 to n - 1 do
5.       begin
6.         C[i, j] := 0;
7.         for k := 0 to n - 1 do
8.           C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.         endfor;
10.  end MAT_MULT

```

```

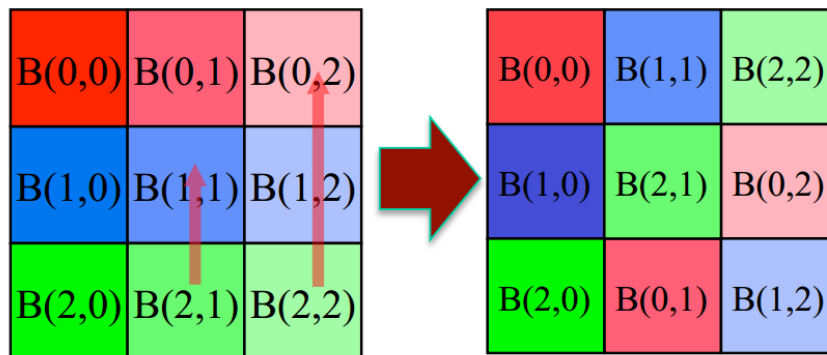
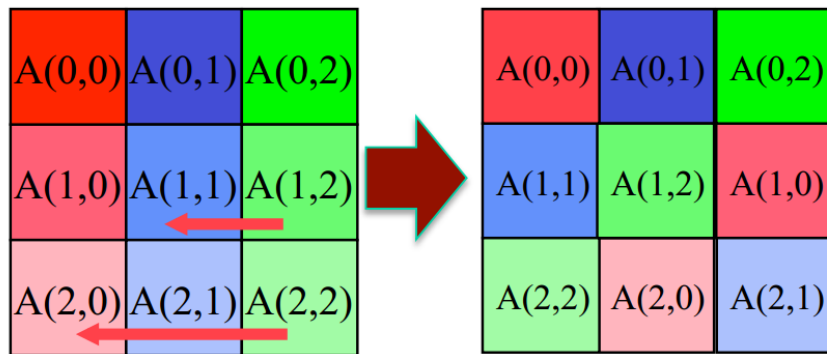
1. procedure BLOCK_MAT_MULT (A, B, C)
2. begin
3.   for i := 0 to q - 1 do
4.     for j := 0 to q - 1 do
5.       begin
6.         Initialize all elements of Ci,j to zero;
7.         for k := 0 to q - 1 do
8.           Ci,j := Ci,j + Ai,k × Bk,j;
9.         endfor;
10.  end BLOCK_MAT_MULT

```



- Each process does \sqrt{p} multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ matrices.
 - Compute cost is $\left(\frac{n}{\sqrt{p}}\right)^3 \sqrt{p} = \frac{n^3}{p}$.
- Each row or column of \sqrt{p} processes does all-to-all broadcast of $\frac{n^2}{p}$ amount of data.
 - Communication time is $O(\log \sqrt{p} + \frac{n^2}{p} \sqrt{p})$.
- Total work is $O(n^3)$, and total overhead is $O(p \log \sqrt{p} + n^2 \sqrt{p})$.
- Isoefficiency requires $n^3 = \Omega(n^2 \sqrt{p})$, so need $p = O(n^2)$.
- One problem with this algorithm is that each process needs to store \sqrt{p} copies of $(n/\sqrt{p}) \times (n/\sqrt{p})$ matrices.
 - So memory use per process is n^2/\sqrt{p} , and total memory use for all processes is $n^2 \sqrt{p}$.
 - This is \sqrt{p} factor more than for the sequential algorithm.

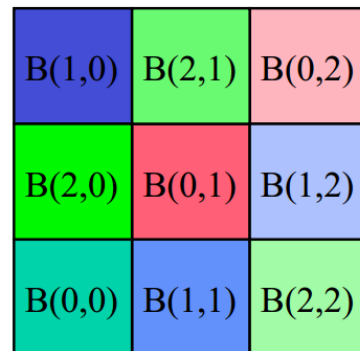
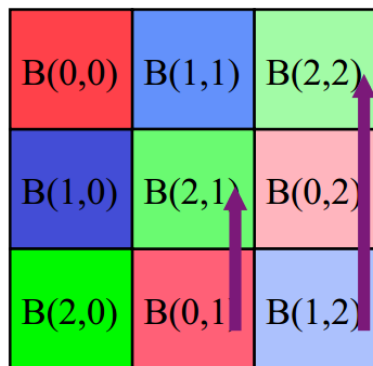
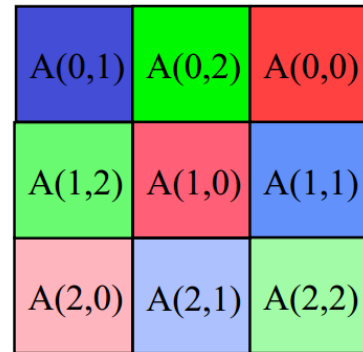
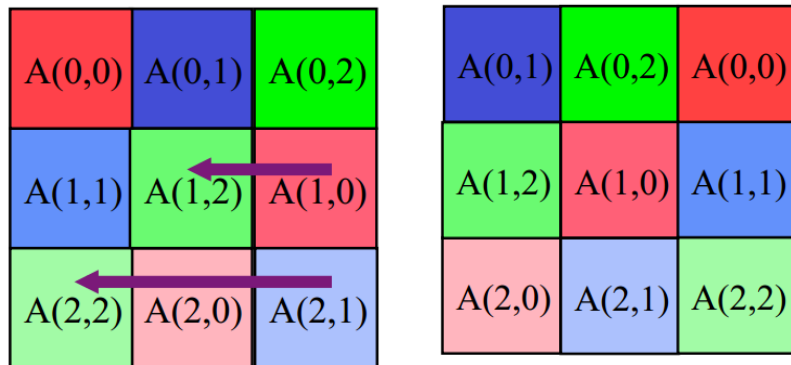
Cannon's matrix-matrix multiplication



- Cannon's algorithm has nearly the same efficiency as the previous algorithm, but uses $O(n^2)$ storage, just like the sequential algorithm.
- It uses the same partitioning as the previous algorithm, but moves some blocks of A and B to other processors.
 - Cyclically shift the i 'th row of A to the left by i , and shift the i 'th column of B up by i .
- Each C value is formed by multiplying like colored blocks from A and B, then adding up the products, one for each color.
 - Ex $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1}$, i.e. we add up the red, blue and green products.
- After shifting, for any i , all blocks with the i 'th color in A and B lie along the i 'th anti-diagonal.
 - So for any (i,j) , like colored blocks from $A_{i,:}$ and $B_{:,j}$ are on processor $P_{i,j}$.
 - When these blocks are multiplied, they make up one of the terms of $C_{i,j}$.

Source: <http://cseweb.ucsd.edu/classes/fa12/cse260-b/Lectures>

Cannon's matrix-matrix multiplication

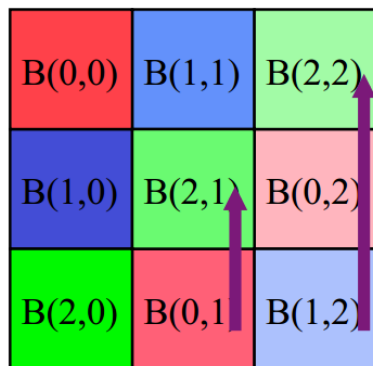
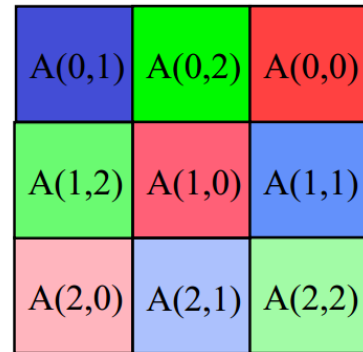
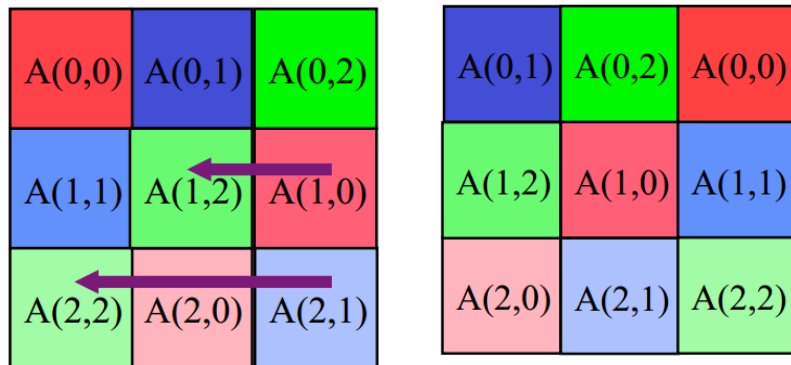


Step 0

Step 1

- After the initial shifting, run for $\sqrt{p} - 1$ more stages.
 - In every stage, shift A blocks left by 1, and B blocks up by 1.
- Each processor $P_{i,j}$ still has like colored blocks from A and B.
- Blocks from A stay in same rows and blocks from B stay in same columns.
- So product of the blocks makes up another term in $C_{i,j}$.
- After \sqrt{p} stages, $P_{i,j}$ contains $C_{i,j}$, for all i and j.

Cannon's matrix-matrix multiplication



Step 0



Step 1

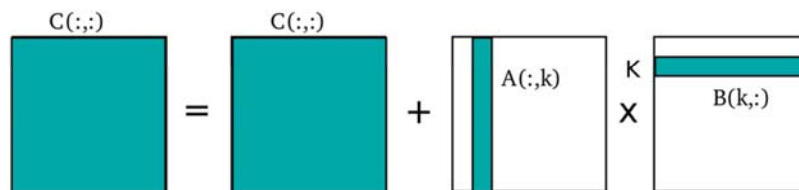
- Each process only stores one block at a time. So total storage at any time is $O(n^2)$.
- Each process does $O\left(\frac{n^3}{p}\right)$ amount of computation.
- Each of the \sqrt{p} shifts costs each processor $O\left(t_s + \frac{t_w n^2}{p}\right)$ communication.
- Total overhead is $O(p^{\frac{3}{2}} + n^2 \sqrt{p})$.
- Isoefficiency requires $p = O(n^2)$.

SUMMA multiplication

```
% inner product approach
for i = 1:I
  for j = 1:J
    for k = 1:K
      C(i,j) = C(i,j) + A(i,k)*B(k,j);
```



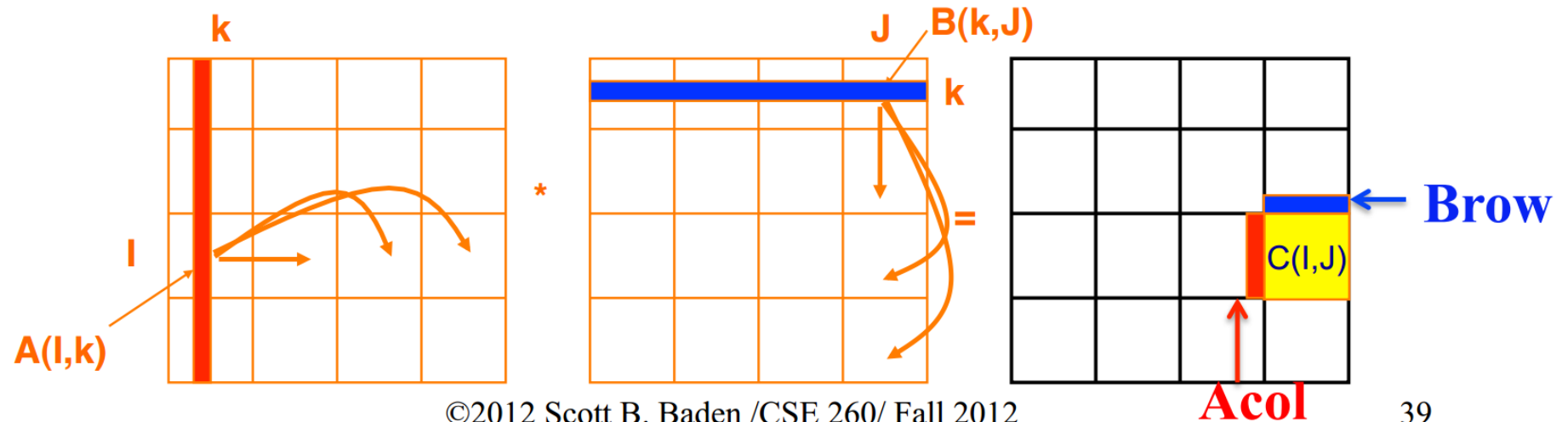
```
% outer product approach
for k = 1:K
  for i = 1:I
    for j = 1:J
      C(i,j) = C(i,j) + A(i,k)*B(k,j);
```



$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^T = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \\ u_4 v_1 & u_4 v_2 & u_4 v_3 \end{bmatrix}.$$

- One of the drawbacks of Cannon is that it can only deal with square matrices, and n must be divisible by \sqrt{p} .
- SUMMA algorithm overcomes those problems.
 - Our example is still for a square matrix though.
- The basic matrix multiplication algorithm is three nested loops.
 - Fast MM algorithms such as Strassen's work differently.
- The loops can be done in any order.
 - The typical inner product approach is the ijk order.
 - The outer product approach is kij.

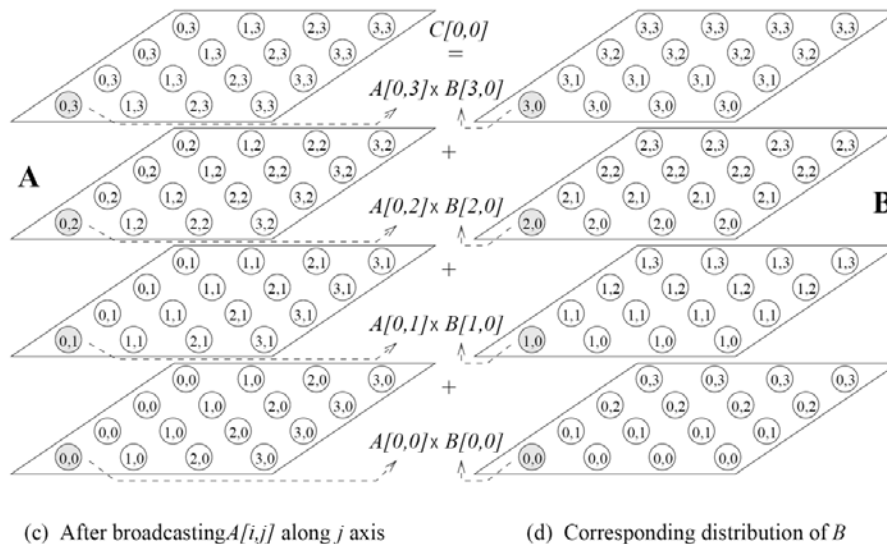
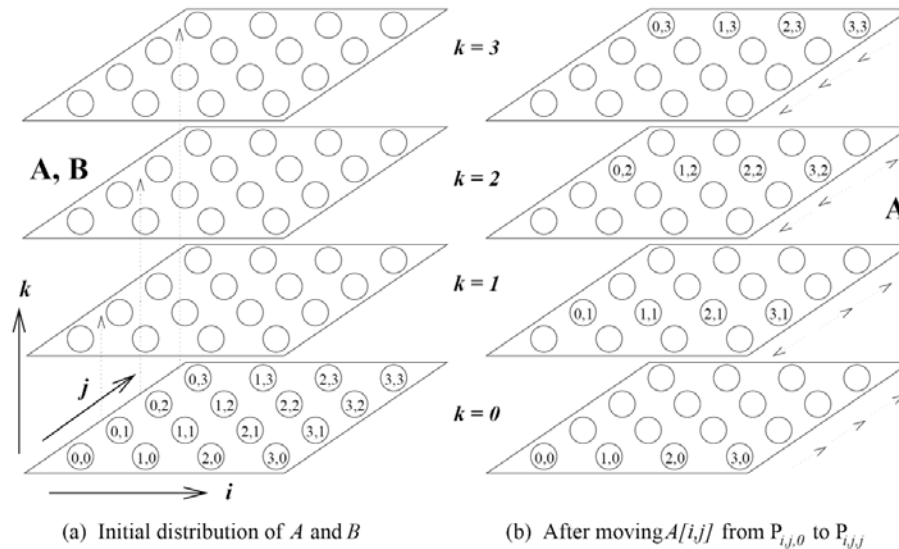
SUMMA multiplication



39

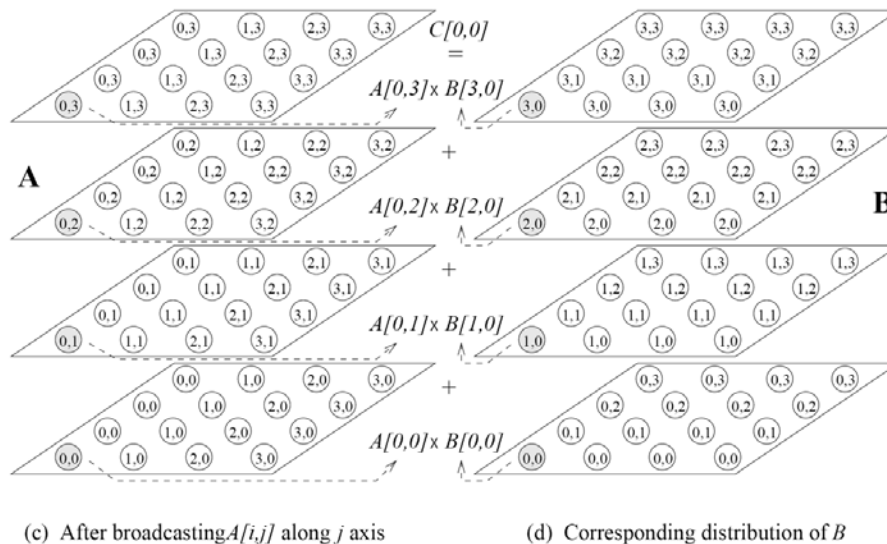
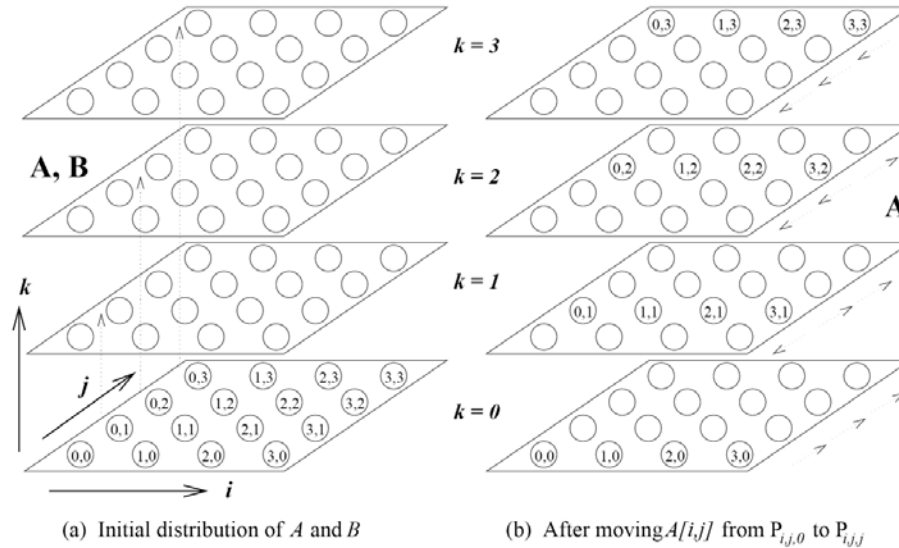
- SUMMA does n outer products.
 - k 'th iteration does outer product of k 'th column of A with k 'th row of B .
 - Processors in k 'th column broadcast their A block to their rows.
 - Processors in k 'th row broadcast their B block to their columns.
 - Processor (i,j) computes $A_{i,k} B_{k,j}$ and accumulates it into $C_{i,j}$.
- Matrices don't need to be square.
- Also allows more flexible mapping of processors to blocks.

3D matrix multiplication



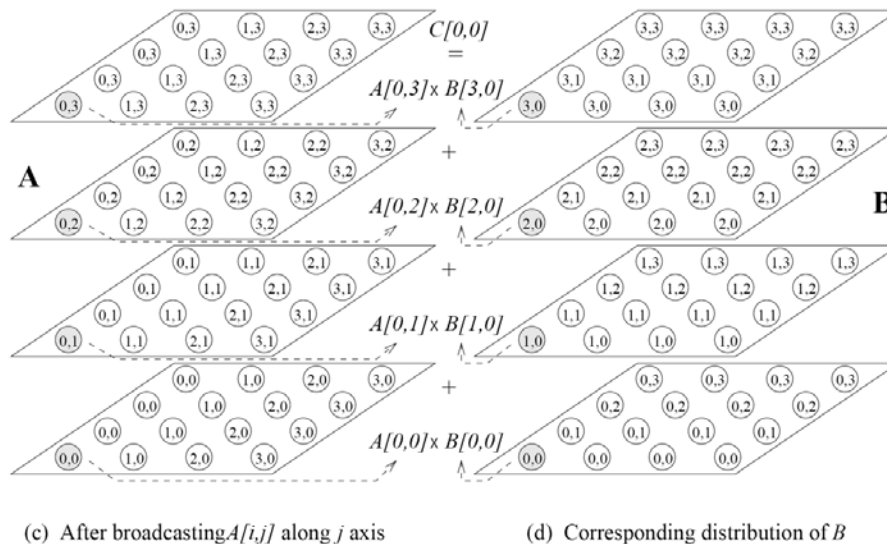
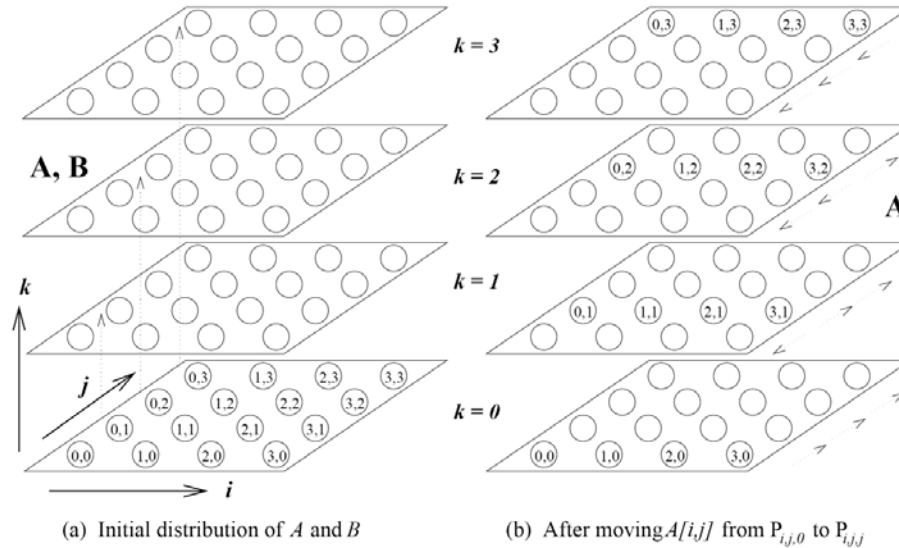
- The 2D algorithms up to now used at most n^2 processors. Since MM has $\Omega(n^3)$ operations, 2D algorithms have $\Omega(n)$ running time.
- We now show the DNS (Dekkel, Nassimi, Sahni) algorithm that can use n^3 processors.
- Arrange the n^3 processors in a $n \times n \times n$ cube.
- For $0 \leq i, j, k \leq n$, processor $P_{i,j,k}$ computes $A_{i,k} B_{k,j}$.
- Then processors in each column $(i, j, :)$ does reduction to collect result onto processor $P_{i,j,0}$.

3D matrix multiplication



- Initially only processors $P_{i,j,0}$, for $0 \leq i, j \leq n-1$, hold the A and B matrices.
- (a) To distribute the data, the j 'th group of processors along the i direction in 0'th (i,j) -plane send their data to corresponding processors in the j 'th (i,j) -plane.
- (b) Then, in each (i,j) -plane, the processors along direction i broadcast in the j direction.
- Effect is that A is replicated in each (i,k) -plane.
- Do similar operations for B , so that B is replicated in each (k,j) -plane.
- Then do reduction in the k direction.
- The broadcast and reduction both take $O(\log n)$ time. So the total computation time is $O(\log n)$.

3D matrix multiplication



- The DNS algorithm isn't optimal using n^3 processors, since the total work is $O(n^3 \log n)$.
- For a cost optimal version, suppose $p=q^3$ for some $q \leq n$. Partition the matrix into $q \times q$ blocks, each of size $(n/q) \times (n/q)$.
- Then apply the same algorithm as before on the blocks.
- The broadcast and reduction both take $t_s \log q + t_w \left(\frac{n}{q}\right)^2 \log q$. The block multiplication takes $\left(\frac{n}{q}\right)^3$.
- Since $q = p^{\frac{1}{3}}$, the total time is $\frac{n^3}{p} + t_s \log p + \frac{t_w n^2}{2 p^{\frac{2}{3}}} \log p$.
- For isoefficiency, we need $n^3 = \Omega(n^2 p^{\frac{1}{3}} \log p)$, which implies $p = O\left(\left(\frac{n}{\log n}\right)^3\right)$.

Solving linear systems

$$\begin{array}{ccccccc} a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0, \\ a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} & = & b_1, \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}. \end{array}$$

$$\begin{array}{ccccccc} x_0 & + & u_{0,1}x_1 & + & u_{0,2}x_2 & + & \cdots & + & u_{0,n-1}x_{n-1} & = & y_0, \\ & & x_1 & + & u_{1,2}x_2 & + & \cdots & + & u_{1,n-1}x_{n-1} & = & y_1, \\ & & & & & & & & \vdots & & \vdots \\ & & & & & & & & x_{n-1} & = & y_{n-1}. \end{array}$$

$$\begin{aligned} 1) \begin{cases} 5x+3y+2z=-2 \\ x+y=2 \\ 2x-y+z=3 \end{cases} &\Rightarrow \left(\begin{array}{ccc|c} 5 & 3 & 2 & -2 \\ 1 & 1 & 0 & 2 \\ 2 & -1 & 1 & 3 \end{array} \right) \xrightarrow{\text{reorder}} \\ &\Rightarrow \left(\begin{array}{ccc|c} 1 & 1 & 0 & 2 \\ 5 & 3 & 2 & -2 \\ 2 & -1 & 1 & 3 \end{array} \right) \xrightarrow{\begin{matrix} R_1 \\ R_2-5R_1 \\ R_3-2R_1 \end{matrix}} \left(\begin{array}{ccc|c} 1 & 1 & 0 & 2 \\ 0 & -2 & 2 & -12 \\ 0 & -3 & 1 & -1 \end{array} \right) \Rightarrow \\ &\xrightarrow{\begin{matrix} R_1 \\ \text{simplify } R_2 \\ R_3 \end{matrix}} \left(\begin{array}{ccc|c} 1 & 1 & 0 & 2 \\ 0 & 1 & -1 & 6 \\ 0 & -3 & 1 & -1 \end{array} \right) \xrightarrow{\begin{matrix} R_1 \\ R_3+3R_2 \end{matrix}} \left(\begin{array}{ccc|c} 1 & 1 & 0 & 2 \\ 0 & 1 & -1 & 6 \\ 0 & 0 & -2 & 17 \end{array} \right) \Rightarrow \\ &\xrightarrow{\text{independent system}} \begin{cases} x+y=2 \\ y-z=6 \\ -2z=17 \end{cases} \Rightarrow \begin{cases} x=\frac{9}{2} \\ y=-\frac{5}{2} \\ z=-\frac{17}{2} \end{cases} \end{aligned}$$

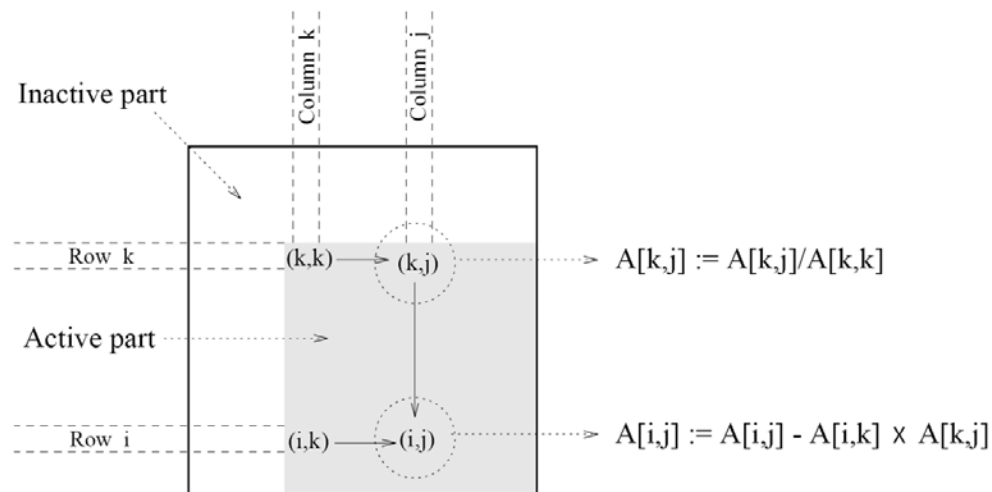
- Given a system of n linear equations, we can reduce it to triangular form using Gaussian Elimination.
- Then the triangular equations can be solved by back substitution.
- As we'll see, Gaussian Elimination takes $O(n^3)$ operations, and back substitution takes $O(n^2)$ operations. So linear systems can be solved in $O(n^3)$ operations.
- Given a matrix A , if $Ax = b$ needs to be solved for multiple b vectors, can also use Gaussian Elimination to compute the LU factorization of A , i.e. $A = LU$, where L is a lower triangular and U is an upper triangular matrix.
 - Then $LUx = b$ can be solved by first solving $Ly = b$ using backwards substitution, then solving $Ux = y$ using backwards substitution.
 - Both steps take $O(n^2)$ time. So solving $Ax = b$ for each b takes $O(n^2)$ instead of $O(n^3)$ time.
 - The initial LU decomposition takes $O(n^3)$ time.

Gaussian Elimination

```

1.  procedure GAUSSIAN_ELIMINATION (A, b, y)
2.  begin
3.    for k := 0 to n - 1 do          /* Outer loop */
4.      begin
5.        for j := k + 1 to n - 1 do
6.          A[k, j] := A[k, j] / A[k, k]; /* Division step */
7.        y[k] := b[k] / A[k, k];
8.        A[k, k] := 1;
9.        for i := k + 1 to n - 1 do
10.         begin
11.           for j := k + 1 to n - 1 do
12.             A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step */
13.           b[i] := b[i] - A[i, k] × y[k];
14.           A[i, k] := 0;
15.         endfor; /* Line 9 */
16.       endfor; /* Line 3 */
17.    end GAUSSIAN_ELIMINATION

```



- Three nested loops of size n , so $O(n^3)$ time.
- Here we assume for simplicity that $A[k, k]$, which we divide by on line 6, is never 0.
 - We also ignore numerical accuracy issues.
 - These are addressed using pivoting.
- Iteration k of the algorithm uses $A[k, k]$ to eliminate all nonzeros in column k .
- If we don't perform lines 7, 8, 13 and 14, the algorithm produces the LU decomposition of A , with L and U stored in the lower and upper triangular parts of A .

Parallel Gaussian Elimination

P ₀	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P ₁	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P ₂	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P ₃	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P ₄	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P ₅	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P ₆	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P ₇	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

P ₀	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P ₁	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P ₂	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P ₃	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P ₄	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P ₅	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P ₆	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P ₇	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

P ₀	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P ₁	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P ₂	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P ₃	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P ₄	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P ₅	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P ₆	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P ₇	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Computation:

- (i) $A[k,j] := A[k,j]/A[k,k]$ for $k < j < n$
- (ii) $A[k,k] := 1$

(b) Communication:

One-to-all broadcast of row $A[k,*]$

(c) Computation:

- (i) $A[i,j] := A[i,j] - A[i,k] \times A[k,j]$
for $k < i < n$ and $k < j < n$
- (ii) $A[i,k] := 0$ for $k < i < n$

■ Consider the k 'th iteration.

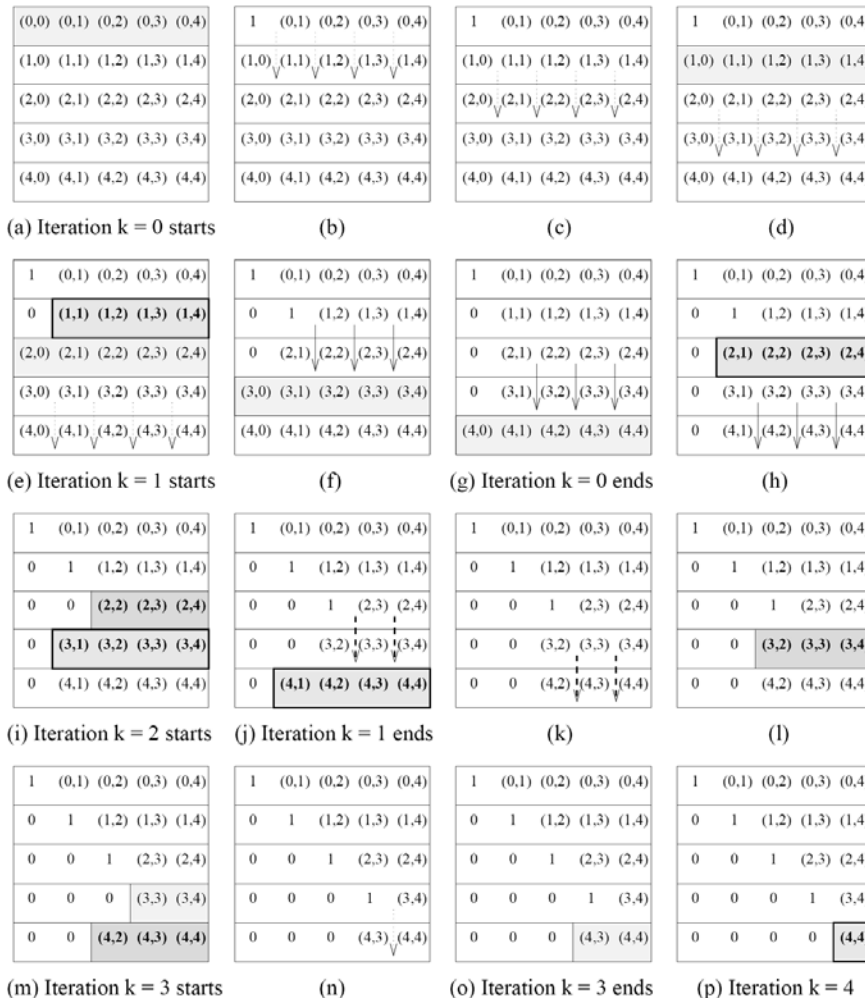
- In step (a), divide the k 'th row by $A[k,k]$.
- In (b), broadcast the k 'th row to the higher rows.
- In (c), each row subtracts the proper multiple of row k from itself.

■ Total time over all iterations k for steps (a) and (c) take $\sum_{k=0}^{n-1} (n - k - 1) = O(n^2)$.

■ For each k , step (b) takes $(t_s + t_w(n - k - 1)) \log n$. Thus, over all n iterations, it takes $O(n^2 \log n)$.

■ Total work is $O(n^3 \log n)$, so this isn't cost optimal.

Pipelined Gaussian Elimination



---> Communication for $k = 0, 3$

—> Communication for $k = 1$

--> Communication for $k = 2$

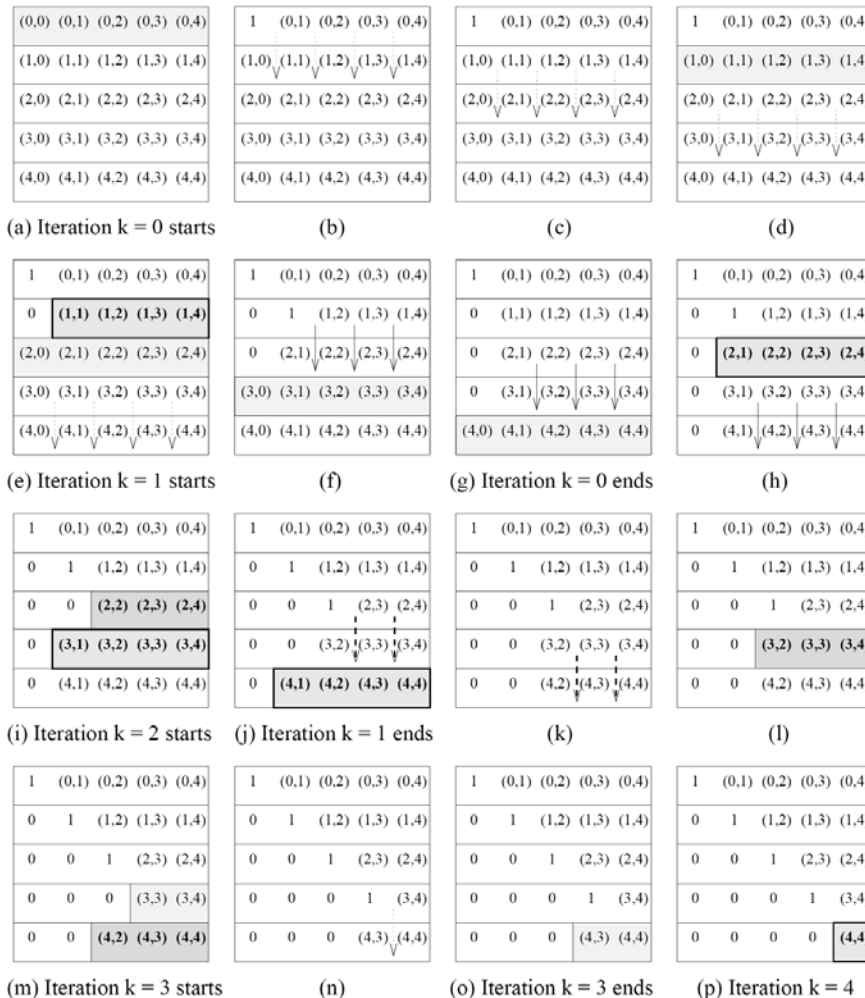
□ Computation for $k = 0, 3$

□ Computation for $k = 1, 4$

□ Computation for $k = 2$

- One problem with previous algorithm is that it waited till iteration k was finished (i.e., the entire k 'th column has been eliminated using the k 'th row) before starting iteration $k+1$.
- We can use a more efficient pipelined algorithm, where each row uses and sends data as quickly as possible.
 - When a row receives data from the previous row, it sends the data to the next row.
 - The row eliminates a variable.
 - Then the row waits for new data from the previous row.
 - When the first nonzero in the row is the diagonal element, it divides out by this row and sends this to the next row.
 - Then the row stops.

Pipelined Gaussian Elimination



---> Communication for $k = 0, 3$

—> Communication for $k = 1$

--> Communication for $k = 2$

□ Computation for $k = 0, 3$

▒ Computation for $k = 1, 4$

■ Computation for $k = 2$

- For each row, call the 3 steps (recv, send, eliminate) a cycle.
 - A row eliminates a variable every cycle.
- The last row receives its last piece of data after n cycles.
 - So the entire algorithm finishes after $O(n)$ cycles.
- Each cycle involves dividing, subtracting or sending $O(n)$ items, so takes $O(n)$ time.
- So algorithm takes total $O(n^2)$ time.
- So total work is $O(n^3)$, and the algorithm is work optimal.

Fewer processors, load balancing

P ₀	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P ₁	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P ₂	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P ₃	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Block 1-D mapping

P ₀	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P ₁	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P ₂	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P ₃	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(b) Cyclic 1-D mapping

- If we have $< n$ processors, we can assign n/p consecutive rows per processor.
 - Each processor does $O(\sum_{i=0}^{n/p-1} (n - i)) = O(n^3/p)$ work.
- However, this leads to the initial processors finishing earlier than the later processors.
 - Total idle work (i.e. idle time x number of idle processors) = $\Theta(n^3)$.
- To prevent idling and achieve better load balancing, can assign the rows in cyclic (round robin) order to the processors.
 - Load difference between different processors in any iteration is then at most one row.
 - Each row contains $O(n)$ work and algorithm runs $O(n)$ iterations, so each process has $O(n^2)$ idle work, and total idle work $O(pn^2)$.