Dense Matrix Algorithms

Ananth Grama, Anshul Gupta,
George Karypis, and Vipin Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.

Topic Overview

- Matrix-Vector Multiplication
- Matrix-Matrix Multiplication
- Solving a System of Linear Equations

Matix Algorithms: Introduction

- Due to their **regular structure**, parallel computations involving matrices and vectors readily lend themselves to **data-decomposition**.
- Typical algorithms rely on **input**, **output**, or **intermediate** data decomposition.
- Most algorithms use **one-** and **two-dimensional block**, **cyclic**, and block-cyclic partitionings.

Matrix-Vector Multiplication

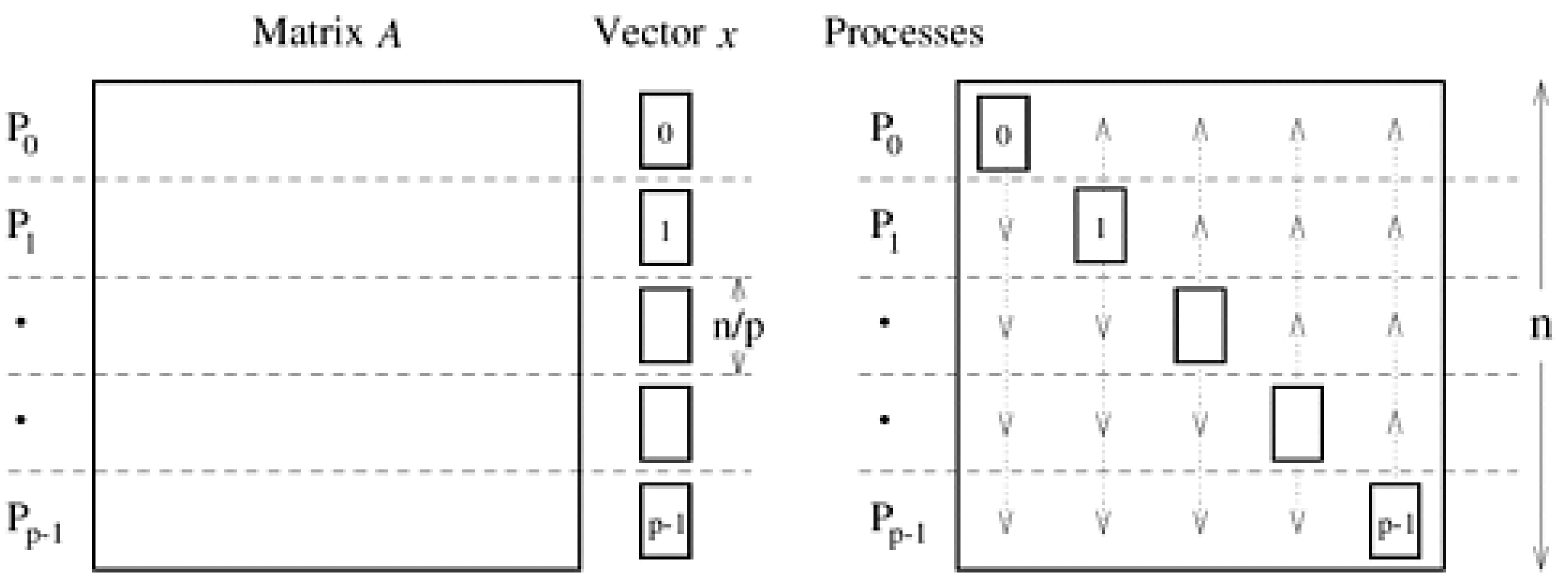
- We aim to multiply a dense n x n **matrix A** with an n x 1 **vector** $\boldsymbol{x}$ to yield the n x 1 **result vector y**.
- The **serial algorithm requires** $\boldsymbol{n^2}$ multiplications and additions.

$$W = n^2.$$

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- The n x n matrix is **partitioned among n processors**, with each processor **storing complete row of the matrix**.
- The n x 1 vector x **is distributed such that each process owns one of its elements**.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

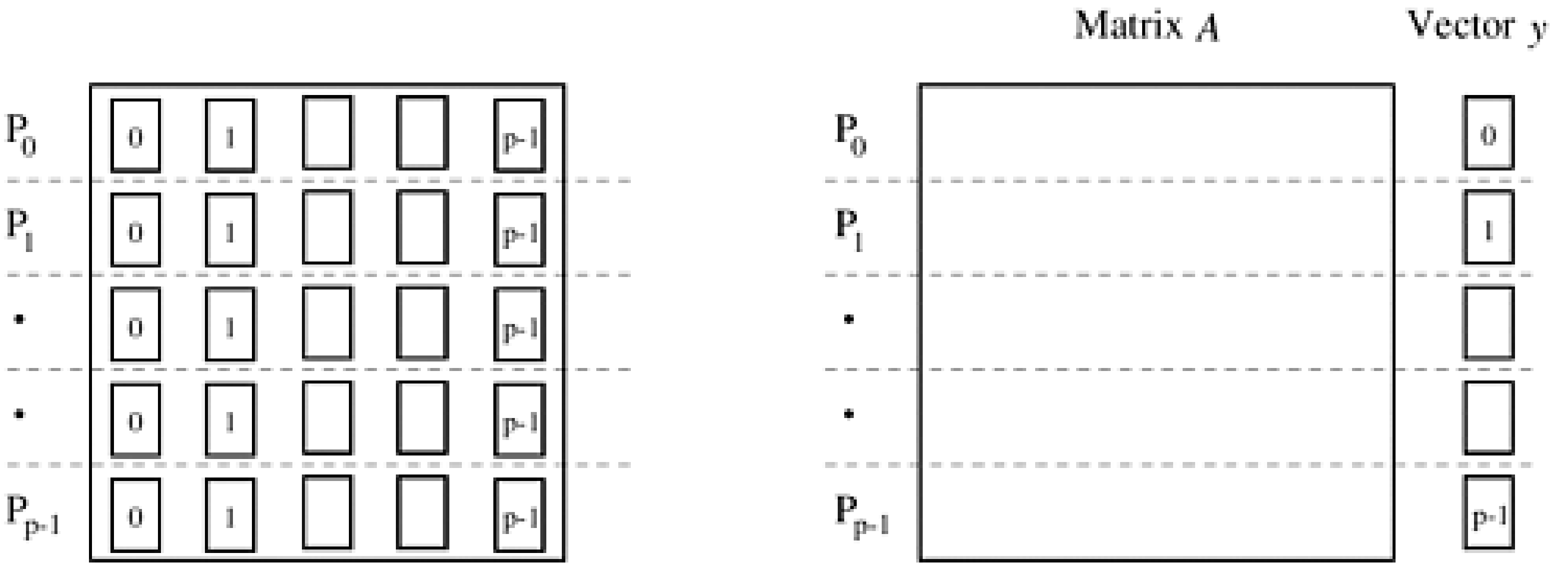


(a) Initial partitioning of the matrix and the starting vector x

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

Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning



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

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

Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Since each process starts with only one element of x , an **all-to-all broadcast** is required to distribute all the elements to all the processes.
- Process P_i now computes $y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$.
- The all-to-all broadcast and the computation of $y[i]$ both take time $\boldsymbol{\Theta(n)}$. Therefore, the parallel time is $\Theta(n)$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Consider now the case **when** $p < n$ and we use block 1D partitioning.
- Each **process initially stores** n/p **complete rows** of the matrix and a portion of the **vector of size** n/p.
- The **all-to-all broadcast** takes place among p processes and involves messages of size n/p.
- This is followed by n/p **local dot products**.
- Thus, the parallel run time of this procedure is

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

This is **cost-optimal**.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

Scalability Analysis:

- We know that $T_0 = pT_P - W$, therefore, we have,

$$T_o = t_s p \log p + t_w np.$$

- For isoefficiency, we have $W = KT_0$, where $K = E/(1 - E)$ for desired efficiency E.
- From this, we have $\boldsymbol{W} = O(p^2)$ (from the t_w term).
- There is also a **bound on isoefficiency because of concurrency**. In this case, $p < n$, therefore, $W = n^2 = \Omega(p^2)$.
- Overall isoefficiency is $W = O(p^2)$.

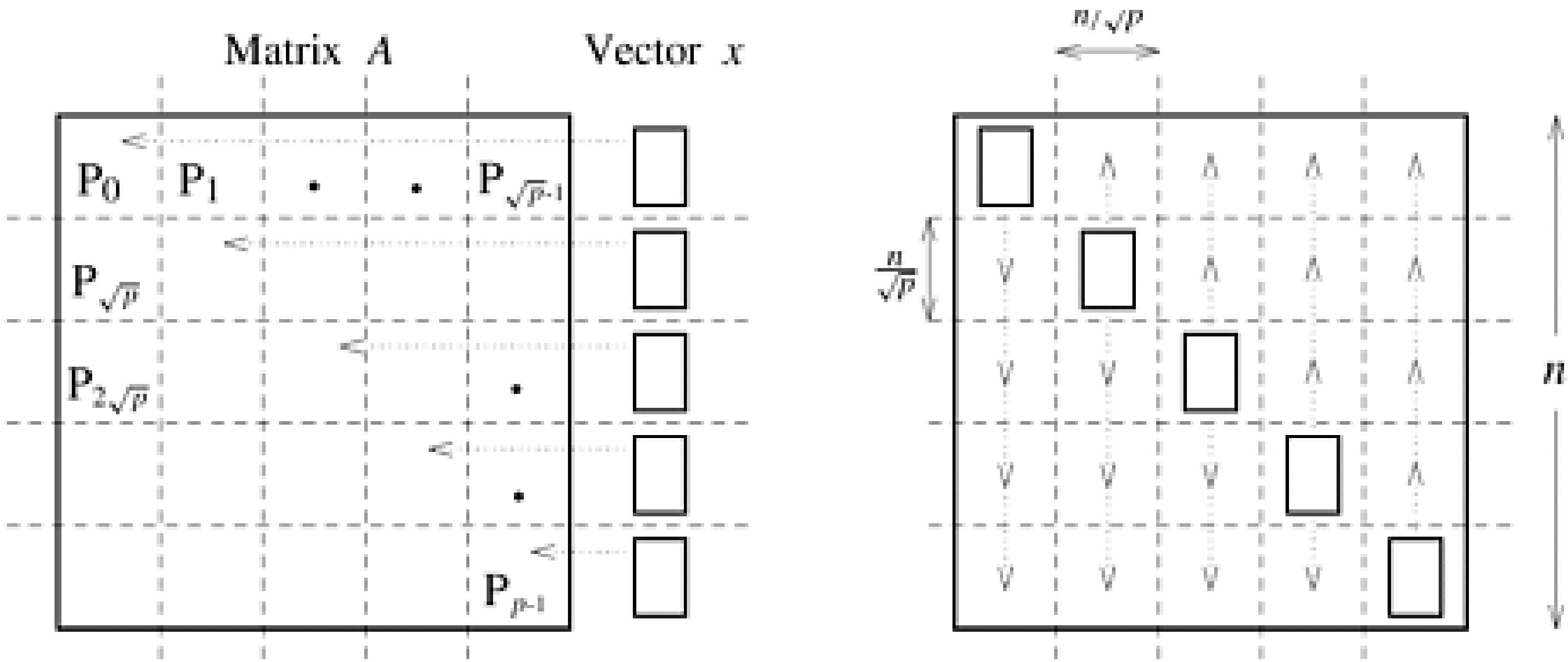
Matrix-Vector Multiplication: 2-D Partitioning

- The n x n matrix is partitioned among $\boldsymbol{n^2}$ **processors** such that each processor owns a single element.
- The n x 1 vector $\boldsymbol{x}$ **is distributed only in the last column** of n processors.

Matrix-Vector Multiplication: 2-D Partitioning

- We must first align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning **aligns the vector** x **along the principal diagonal** of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using n **simultaneous broadcasts among all processors in the column**.
- Finally, the result **vector is computed** by performing an **all-to-one reduction** along the columns.

Matrix-Vector Multiplication: 2-D Partitioning

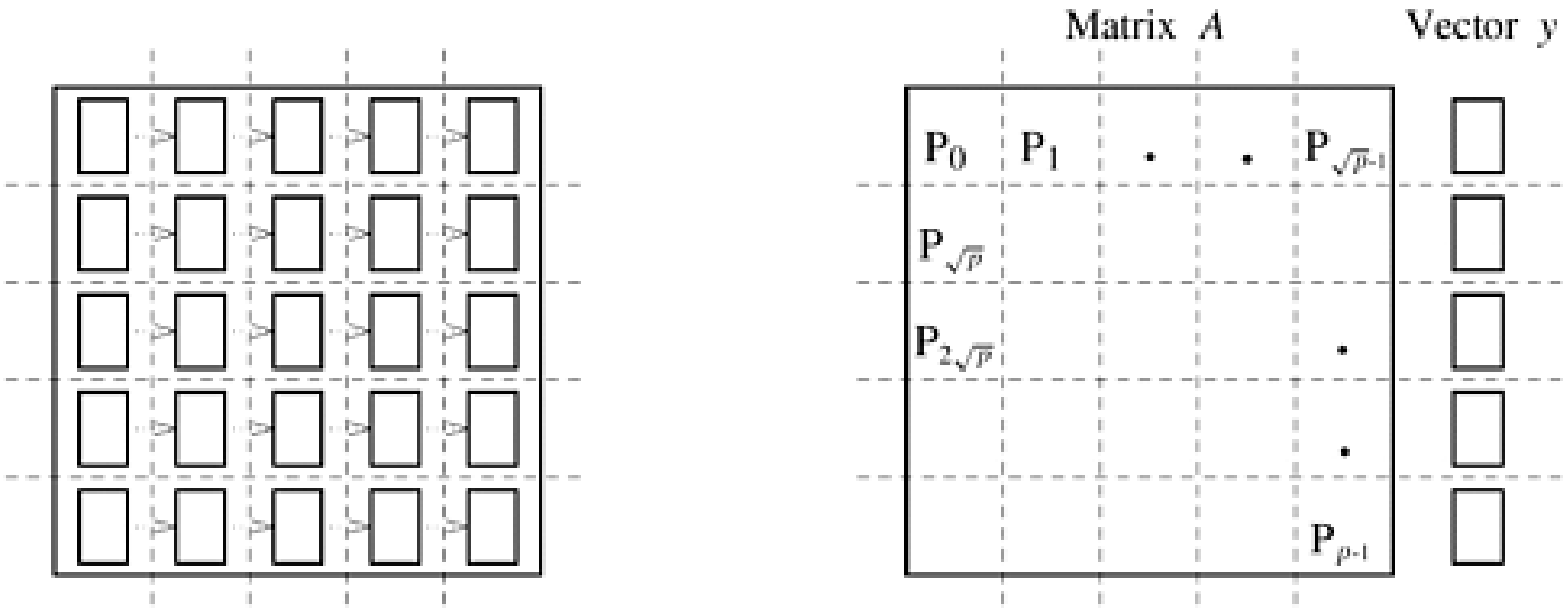


(a) Initial data distribution and communication steps to align the vector along the diagonal

(b) One-to-all broadcast of portions of the vector along process columns

Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is n x n .

Matrix-Vector Multiplication: 2-D Partitioning



(c) All-to-one reduction of partial results

(d) Final distribution of the result vector

Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is n x n .

Matrix-Vector Multiplication: 2-D Partitioning

- Three **basic communication operations** are used in this algorithm: **one-to-one communication to align the vector** along the main diagonal, **one-to-all broadcast** of each vector element among the n processes of each column, and **all-to-one reduction** in each row.
- Each of these operations takes $\Theta(\log n)$ time and the parallel time is $\Theta(\log n)$.
- The cost (process-time product) is $\Theta(n^2 \log n)$; hence, the **algorithm is not cost-optimal**.

Matrix-Vector Multiplication: 2-D Partitioning

- When using **fewer than** ***n^2*** **processors**, each process owns **an** $(n/\sqrt{p}) \times (n/\sqrt{p})$ **block** of the matrix.
- The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only.
- In this case, the **message sizes for the alignment**, broadcast, and reduction are all $n/\sqrt{p}$.
- The computation is a product of an $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrix with a vector of length $n/\sqrt{p}$.

Matrix-Vector Multiplication: 2-D Partitioning

- The first **alignment step** takes time

$$t_s + t_w n/\sqrt{p}$$

- The **broadcast and reductions** take time

$$(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$$

- Local **matrix-vector products** take time

$$t_c n^2/p$$

- **Total** time is

$$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

Matrix-Vector Multiplication: 2-D Partitioning

- Scalability Analysis:

- $T_o = pT_p - W = t_s p \log p + t_w n \sqrt{p} \log p$
- Equating T_0 with W, term by term, for isoefficiency, we have, $W = K^2 t_w^2 p \log^2 p$ as the dominant term.
- The isoefficiency due to concurrency is $O(p)$.
- The **overall isoefficiency** is $O(p \log^2 p)$ (due to the network bandwidth).
- For cost optimality, we have, $W = n^2 = p \log^2 p$. For this, we have, $p = O\left(\frac{n^2}{\log^2 n}\right)$

1-D vs. 2-D Partitioning

	1-D	2-D
Max num. of processors	$p \le n$	$p \le n^2$
T_p	$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$	$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$
isoefficiency	$O(p^2)$	$O(p \log^2 p)$
Max num. of processors (cost-optimally)	$p = O(n)$	$p = O\left(\frac{n^2}{\log^2 n}\right)$

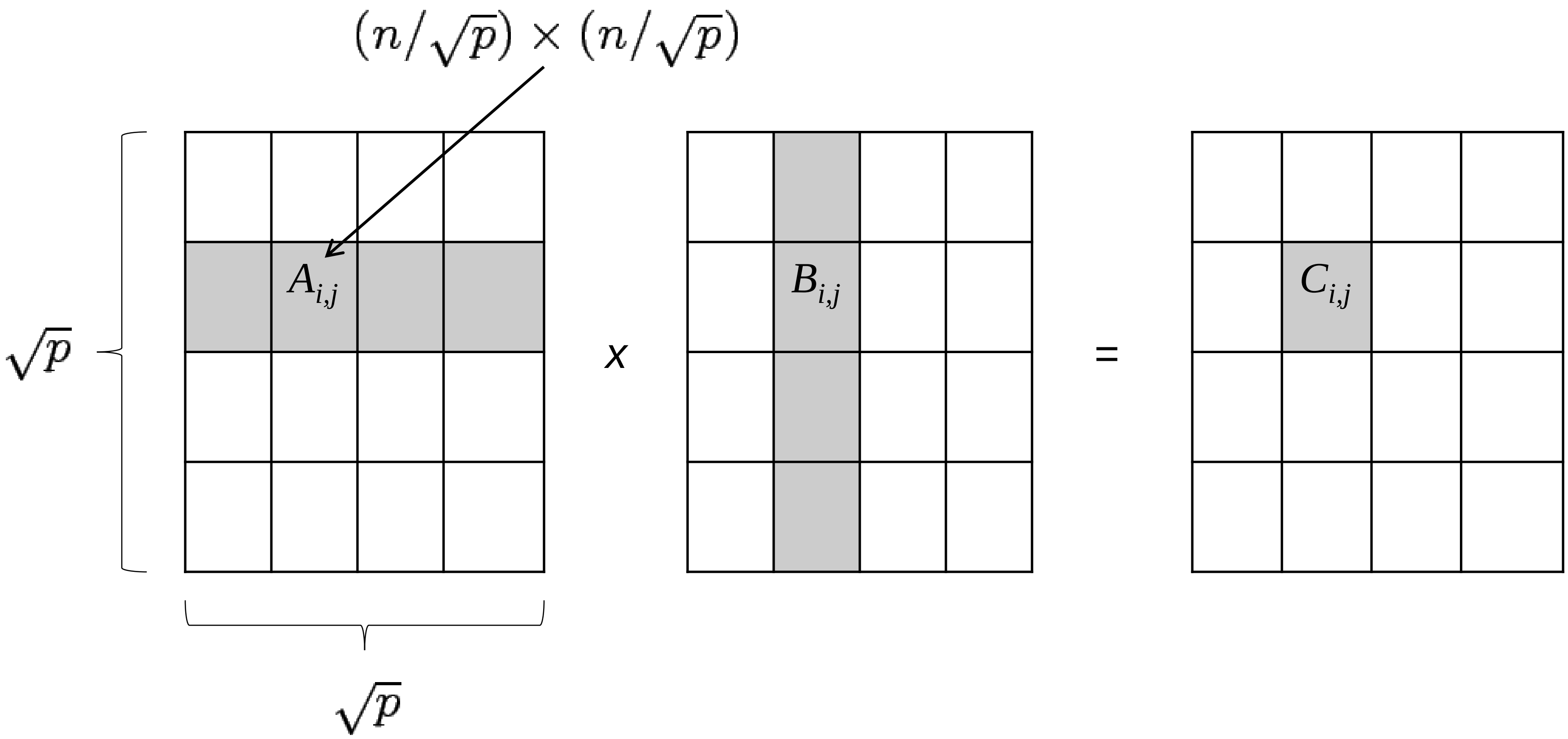
Matrix-Matrix Multiplication

- Consider the problem of multiplying two n x n dense, square matrices A and B to yield the product matrix $C = A \times B$.
- The serial complexity is $O(n^3)$.
- We do **not consider better serial algorithms** (Strassen's method), although, these **can be used as serial kernels in the parallel algorithms**.
- A useful concept in this case is called ***block* operations**. In this view, an n x n matrix A can be regarded as a q x q array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an **(n/q) x (n/q) submatrix**.
- In this view, **we perform q^3 matrix multiplications**, each involving (n/q) x (n/q) matrices.

Matrix-Matrix Multiplication

- Consider two ***n* x *n* matrices *A* and *B* partitioned into *p* blocks** $A_{i,j}$ and $B_{i,j}$ (0 ≤ $i, j < \sqrt{p}$) of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Process $P_{i,j}$ initially **stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$** of the result matrix.
- Computing submatrix $C_{i,j}$ **requires all submatrices $A_{i,k}$ and $B_{k,j}$** for 0 ≤ $k < \sqrt{p}$.
- **All-to-all broadcast** blocks of ***A* along rows** and ***B* along columns**.
- Perform local submatrix multiplication.

Matrix-Matrix Multiplication



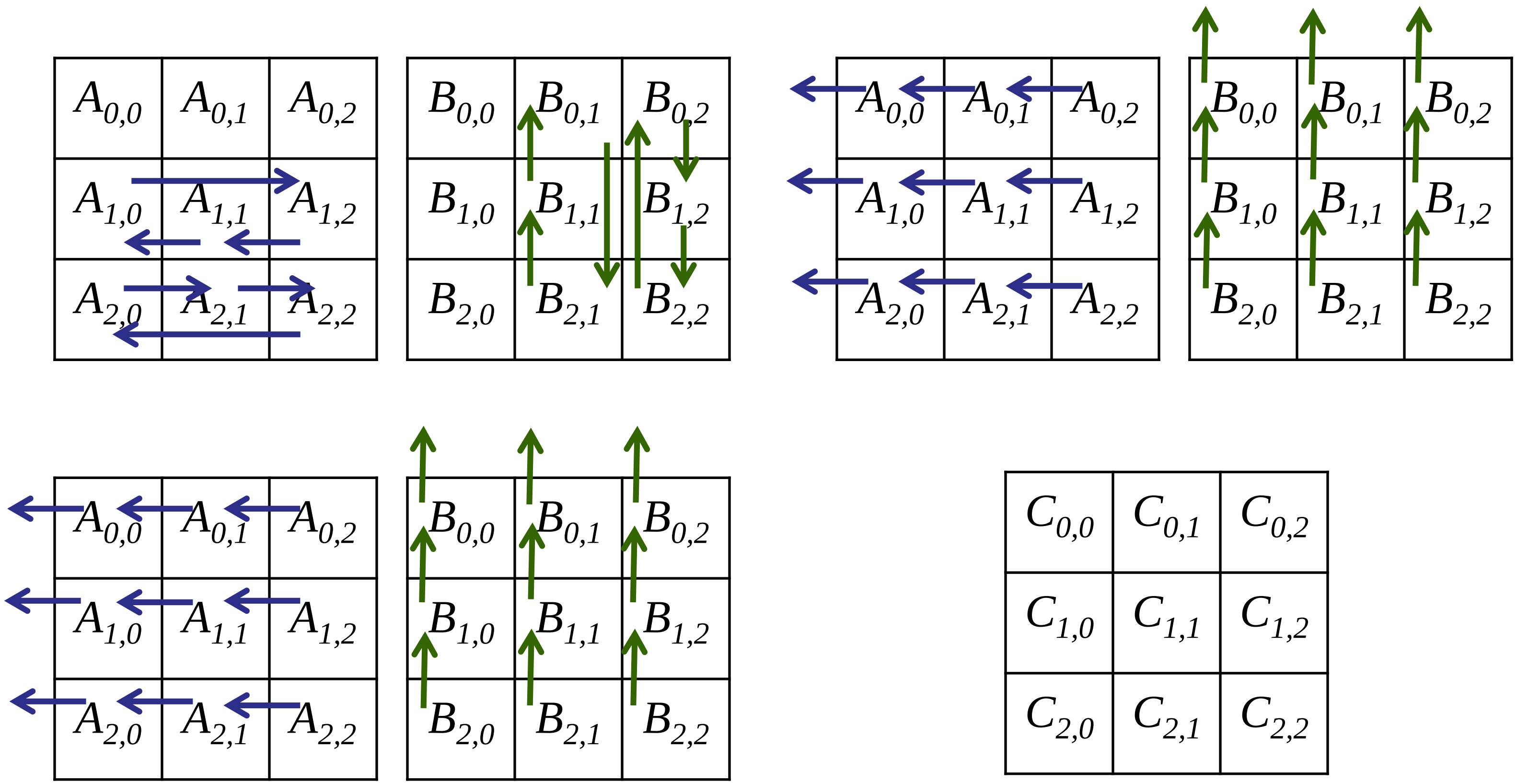
Matrix-Matrix Multiplication

- The two broadcasts take time
$$2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$$
- The computation **requires** $\sqrt{p}$ **multiplications** of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices.
- The parallel run time is approximately
$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}.$$
- The **algorithm is cost optimal** and the **isoefficiency is** ***O(p^1.5)*** due to bandwidth term t_w and concurrency.
- Major drawback of the algorithm is that it **is not memory optimal**.

Matrix-Matrix Multiplication: Cannon's Algorithm

- In this algorithm, we **schedule the computations** of the $\sqrt{p}$ processes of the ith row such that, at any given time, each process is using a different block $A_{i,k}$.
- These **blocks can be systematically rotated** among the processes after every submatrix multiplication so that **every process gets a fresh $A_{i,k}$** after each rotation.

Matrix-Matrix Multiplication: Cannon's Algorithm



Communication steps in Cannon's algorithm on 9 processes.

Matrix-Matrix Multiplication: Cannon's Algorithm

- Align the blocks of A and B in such a way that **each process multiplies its local submatrices**. This is done by **shifting all submatrices $A_{i,j}$ to the left** (with wraparound) by i steps and **all submatrices $B_{i,j}$ up** (with wraparound) by j steps.
- Perform **local block multiplication**.
- Each **block of A moves one step left** and each **block of B moves one step up** (again with wraparound).
- **Perform next block multiplication**, **add to partial result**, repeat until all $\sqrt{p}$ blocks have been multiplied.

Matrix-Matrix Multiplication: Cannon's Algorithm

- In the alignment step, since the maximum distance over which a block shifts is $\sqrt{p}-1$, the **two shift operations** require a total of $2(t_s + t_w n^2/p)$ time.
- Each of the $\sqrt{p}$ single-step shifts in the **compute-and-shift phase** of the algorithm takes $t_s + t_w n^2/p$ time.
- The computation time for **multiplying** $\sqrt{p}$ **matrices** of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ is n^3/p.
- The parallel time is approximately:

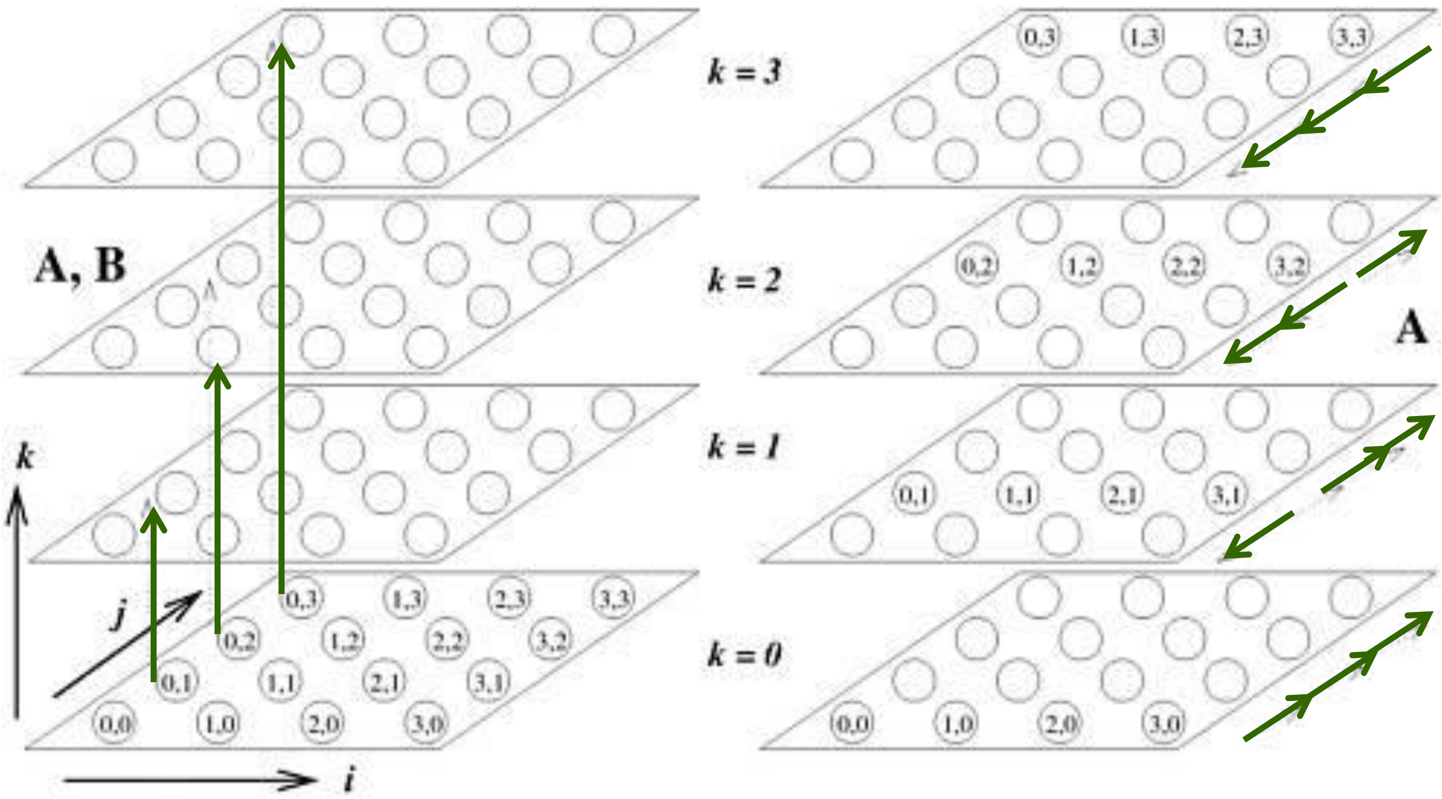
$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}.$$

- The **cost-efficiency and isoefficiency of the algorithm are identical to the first algorithm**, except, this is **memory optimal**.

Matrix-Matrix Multiplication: DNS Algorithm

- Uses a 3-D partitioning.
- Visualize the matrix **multiplication algorithm as a cube**. Matrices A and B come in two orthogonal faces and result C comes out the other orthogonal face.
- Each internal **node in the cube represents a single add-multiply operation** (and thus the complexity).
- DNS algorithm partitions this cube using a **3-D block scheme**.

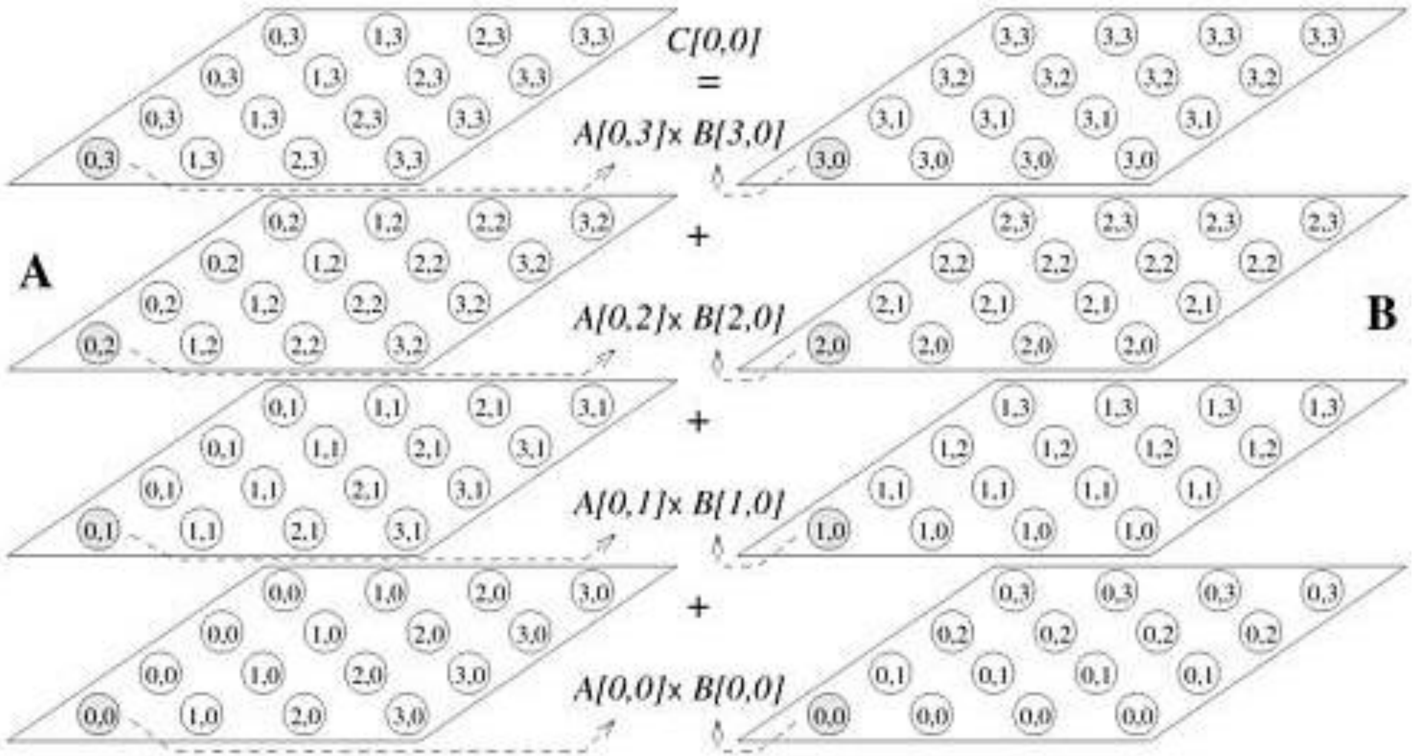
Matrix-Matrix Multiplication: DNS Algorithm



(a) Initial distribution of A and B (b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

The communication steps in the DNS algorithm while multiplying 4 x 4 matrices A and B on 64 processes.

Matrix-Matrix Multiplication: DNS Algorithm



(c) After broadcasting $A[i,j]$ along j axis

(d) Corresponding distribution of B

The communication steps in the DNS algorithm while multiplying 4 x 4 matrices A and B on 64 processes.

Matrix-Matrix Multiplication: DNS Algorithm

- Assume an ***n*** **x** ***n*** **x** ***n*** **mesh of processors**.
- Move the columns of A and rows of B and perform broadcast.
- Each processor computes a **single add-multiply**.
- This is followed by an **accumulation along the C dimension**.
- Since each add-multiply takes constant time and **accumulation and broadcast takes $\log n$ time**, the total runtime is $\log n$.
- This **is not cost optimal**. It **can be made cost optimal** by using $n / \log n$ processors along the direction of accumulation.

Matrix-Matrix Multiplication: DNS Algorithm

Using **fewer than n^3 processors**.

- Assume that the number of processes p is equal to q^3 for some $q < n$.
- The two matrices are partitioned into **blocks of size (n/q) x(n/q)**.
- Each matrix can thus be regarded as a **q x q two-dimensional square array of blocks**.
- The algorithm follows from the previous one, except, in this case, we **operate on blocks rather than on individual elements**.

Matrix-Matrix Multiplication: DNS Algorithm

Using fewer than n^3 processors.

- The first one-to-one **communication** step is performed for both A and B, and takes $t_s + t_w(n/q)^2$time for each matrix.
- The two **one-to-all broadcasts** take $2(t_s \log q + t_w(n/q)^2 \log q)$ time for each matrix.
- The **reduction** takes time $t_s \log q + t_w(n/q)^2 \log q$.
- **Multiplication** of $(n/q) \times (n/q)$ submatrices takes $(n/q)^3$ time.
- The parallel time is approximated by:

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$$

- The **isoefficiency function** is $\Theta(p(\log p)^3)$.

Cannon's vs. DNS Algorithm

	Cannon's	DNS
Max num. of processors	$p \le n^2$	$p \le n^3$
T_p	$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}.$	$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$
W	$O(p^{1.5})$	$\Theta(p(\log p)^3)$
Max num. of processors (cost-optimally)	$p = O(n^2)$	$p = O(n^3/log^3 n)$

Solving a System of Linear Equations

- Consider the problem of solving linear equations of the kind:

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

- This is **written as** $\boldsymbol{Ax} = \boldsymbol{b}$, where A is an n x n matrix with $A[i, j] = a_{i,j}$, b is an n x 1 vector $[\, b_0, b_1, \ldots, b_{n-1} \,]^T$, and x is the solution.

Solving a System of Linear Equations

Two steps in solution are: **reduction to triangular form**, and **back-substitution**. The triangular form is as:

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

We write this as: $Ux = y$.

A commonly used method for transforming a given matrix into an upper-triangular matrix is **Gaussian Elimination**.

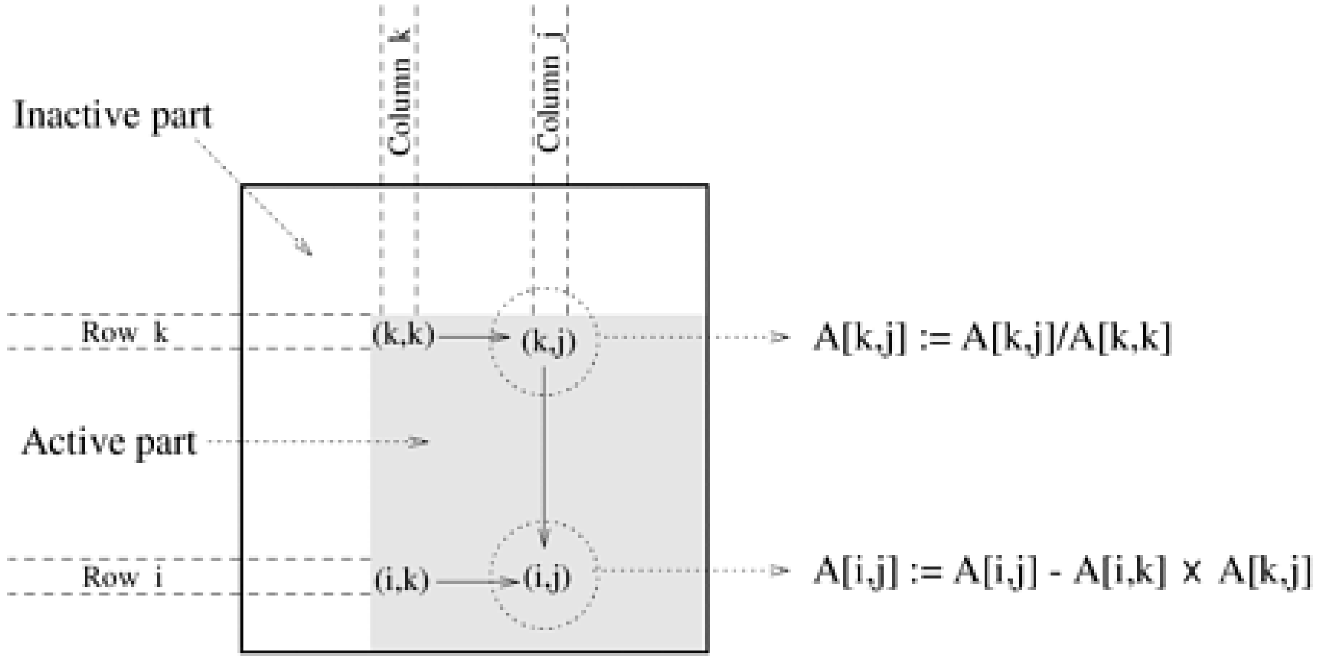
Gaussian Elimination

```
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
```

Serial Gaussian Elimination

Gaussian Elimination

- The computation has **three nested loops** - in the kth iteration of the outer loop, the algorithm **performs** $\boldsymbol{(n-k)^2}$ **computations**. Summing from $k = 1..n$, we have roughly $(n^3/3)$ multiplications-subtractions.



A typical computation in Gaussian elimination.

Parallel Gaussian Elimination

- **Assume $p = n$ with each row assigned to a processor.**
- The first step of the algorithm **normalizes the row**. This is a serial operation and takes time $(n\text{-}k)$ in the k^{th} iteration.
- In the second step, the **normalized row is broadcast** to all the processors. This takes time $(t_s + t_w(n-k-1))\log n$.
- Each processor can **independently eliminate this row** from its own. This **requires $(n\text{-}k\text{-}1)$ multiplications** and subtractions.
- The **total parallel time** can be computed by summing from $k = 1 \ldots n\text{-}1$ as

$$T_P = \frac{3}{2}n(n-1) + t_s n \log n + \frac{1}{2}t_w n(n-1)\log n.$$

- The formulation is **not cost optimal** because of the t_w term.

Parallel Gaussian Elimination

1)

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)

(a) Computation:

(i) A[k,j] := A[k,j]/A[k,k] for $k < j <$

(ii) A[k,k] := 1

2)

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)

(b) Communication:

One–to–all broadcast of row A[k,*]

3)

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)

(c) Computation:

(i) A[i,j] := A[i,j] – A[i,k]× A[k,j]
for $k < i < n$ and $k < j < n$

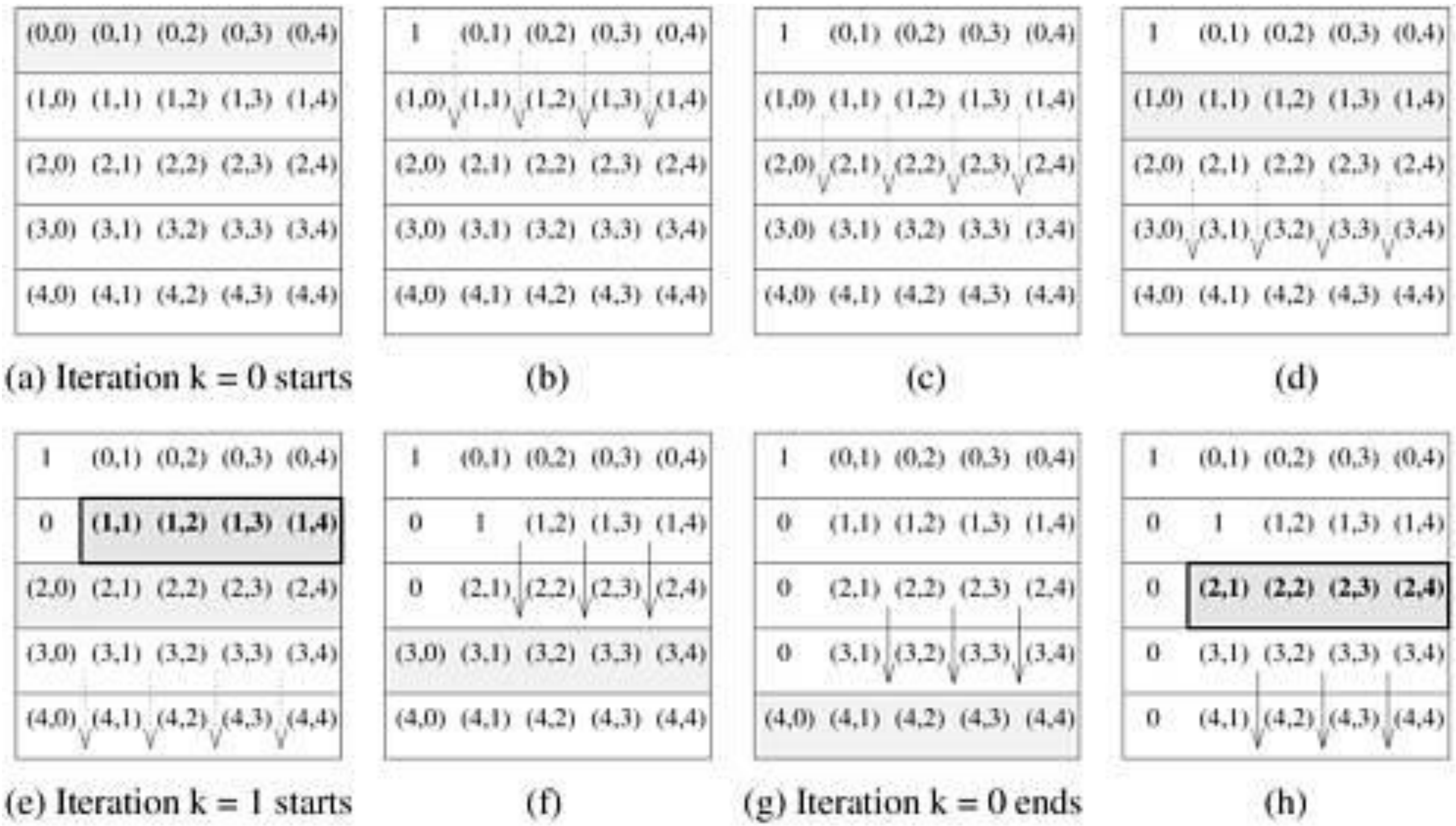
(ii) A[i,k] := 0 for $k < i < n$

Gaussian elimination steps during the iteration corresponding ***k* = 3**

Parallel Gaussian Elimination: Pipelined Execution

- In the previous formulation, **the $(k$+1)st iteration starts only after** all the computation and communication for the **kth iteration is complete**.
- In the **pipelined version**, there are three steps - **normalization** of a row, **communication**, and **elimination**. These steps are **performed in an asynchronous fashion**.
- A processor P_k **waits to receive and eliminate all rows** prior to k.
- Once it has done this, **it forwards its own row to processor** P_{k+1}.

Parallel Gaussian Elimination: Pipelined Execution



Pipelined Gaussian elimination on a 5 x 5 matrix partitioned withone row per process.

Parallel Gaussian Elimination: Pipelined Execution

- The **total number of steps** in the entire pipelined procedure is $\Theta(n)$.
- In any step, either $O(n)$ **elements are communicated between directly-connected processes**, or a division step is performed on $O(n)$ elements of a row, or an **elimination step is performed on $O(n)$ elements** of a row.
- The parallel time is therefore $O(n^2)$.
- This is **cost optimal**.

Parallel Gaussian Elimination: Pipelined Execution

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)

The communication in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix distributed among four processes using block 1-D partitioning.

Parallel Gaussian Elimination: Block 1D with $p < n$

- The above algorithm can be easily adapted to the case when $p < n$.
- In the kth iteration, a processor with all rows belonging to the active part of the matrix performs $(n - k - 1)\, n / p$ multiplications and subtractions.
- In the pipelined version, for $n > p$, computation dominates communication.
- The parallel time is given by: $2(n/p)\Sigma_{k=0}^{n-1}(n-k-1)$
  or approximately, n^3/p.
- While the algorithm is cost optimal, the cost of the parallel algorithm is higher than the sequential run time by a factor of 3/2.