IN3200/IN4200: Parallelization of dense matrix algorithms

Selected slides from A. Grama, A. Gupta, G. Karypis, and V. Kumar, *"Introduction to Parallel Computing"*, Addison Wesley, 2003

Topic Overview

How can parallel computing be applied to the following dense matrix algorithms?

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

Dense Matrix Algorithms: Quick introduction

- Due to their regular structure, parallel computations involving matrices and vectors can readily use 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

- Multiply a dense $n \times n$ matrix A with an $n \times 1$ input vector x to yield an $n \times 1$ result vector y.
- Each entry in the result vector y is computed as

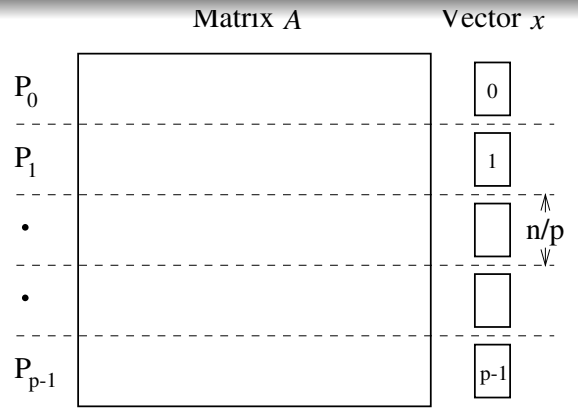
$$y_i = A_{i,0} \cdot x_0 + A_{i,1} \cdot x_1 + \ldots + A_{i,n-1} \cdot x_{n-1}$$

  That is, y_i is computed as a dot-product between row i in matrix A and the input vector x.
- The serial algorithm requires n^2 multiplications and additions.

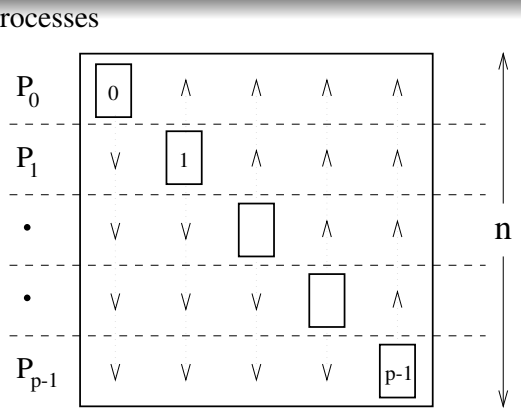
Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Assumption: The number of processes p is the same as n.
- The $n \times n$ matrix A is partitioned among n processes, each process stores one row of the matrix.
- The $n \times 1$ vector x is distributed such that each process owns one of its elements.

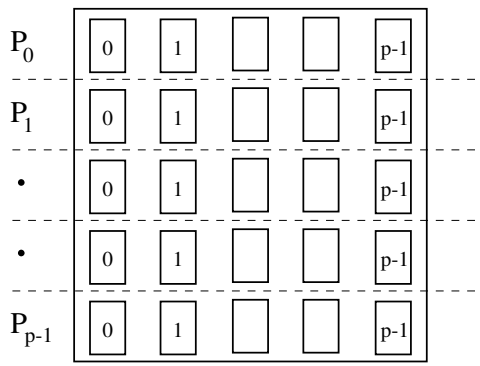
Matrix-Vector Multiplication: Rowwise 1-D Partitioning



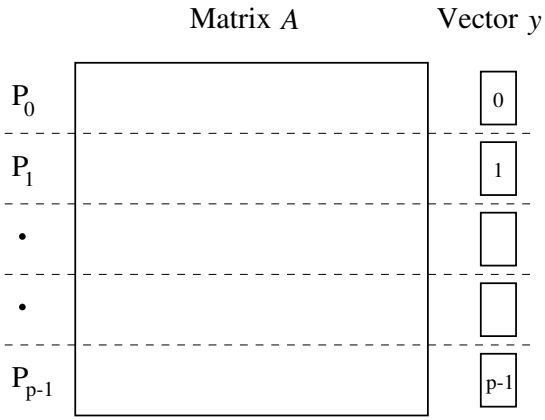
(a) Initial partitioning of the matrix and the starting vector x



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



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



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

Multiplication of an $n \times n$ matrix with an $n \times 1$ vector using rowwise block 1-D partitioning. One row per process: $p = n$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Each process starts with only one element of x, thus an all-to-all broadcast is required to distribute all the elements of x to all the processes.
- Process P_i now computes $y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$.

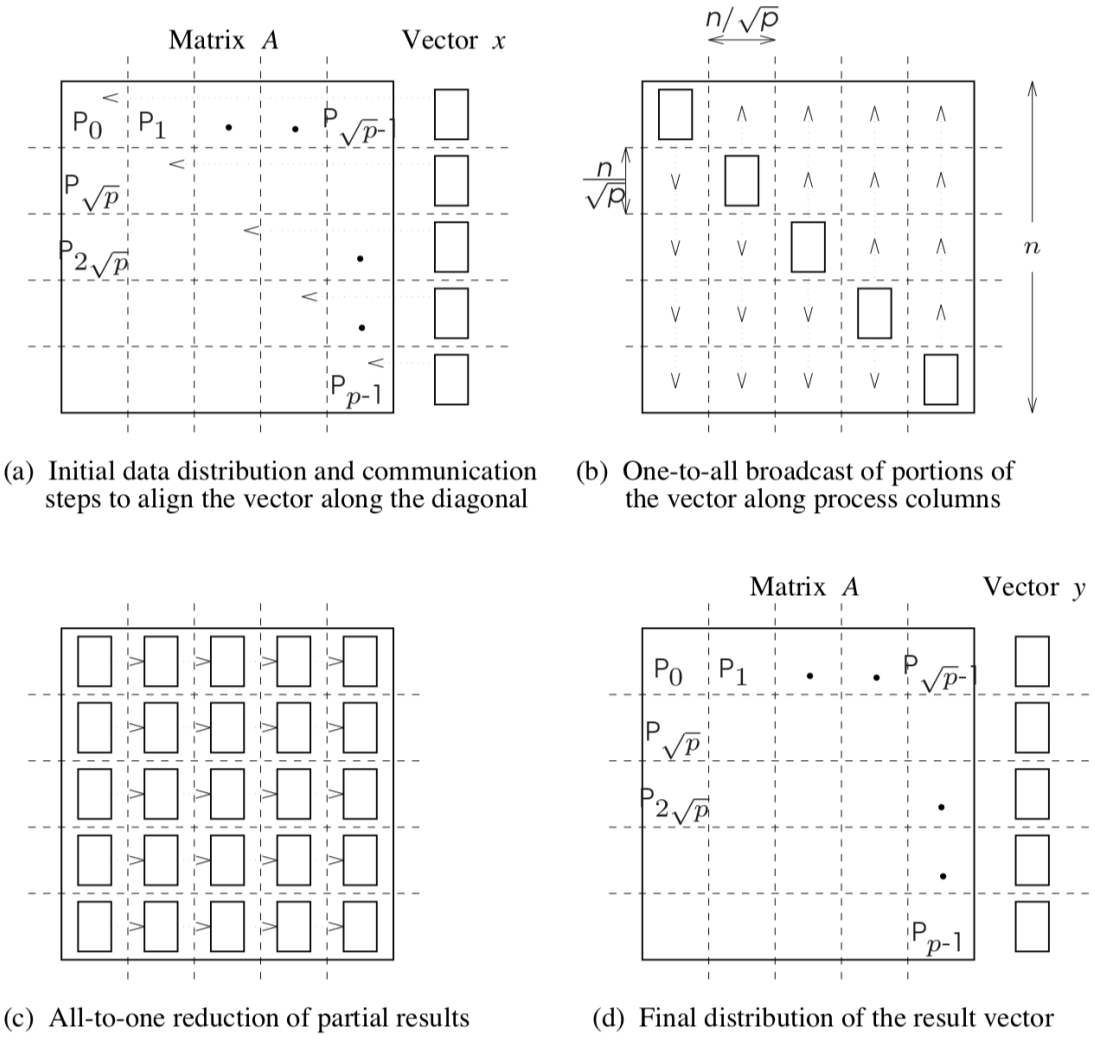
Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Consider now the case when $p < n$ and we use block 1D partitioning.
- Each process initially stores n/p rows of the A matrix and a portion of the x 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.

Matrix-Vector Multiplication: 2-D Partitioning

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

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

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

(d) Final distribution of the result vector

Matrix-vector multiplication with block 2-D partitioning. The case of one element per process: $p = n^2$.

Matrix-Vector Multiplication: 2-D Partitioning

- We must first distribute the vector x appropriately.
- The first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix.
- The second communication 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 each row.

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 A.
- The vector x 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-Matrix Multiplication

- Consider the problem of multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$.
- The serial complexity is $O(n^3)$.
- A useful concept in this case is called *block* operations. In this view, an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ $(0 \leq i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix.
- In this view, we perform in total q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices.
- For example, to compute every $C_{i,j}$ we need q matrix multiplications:

$$C_{i,j} = A_{i,0}B_{0,j} + A_{i,1}B_{1,j} + \ldots + A_{i,q-1}B_{q-1,j}$$

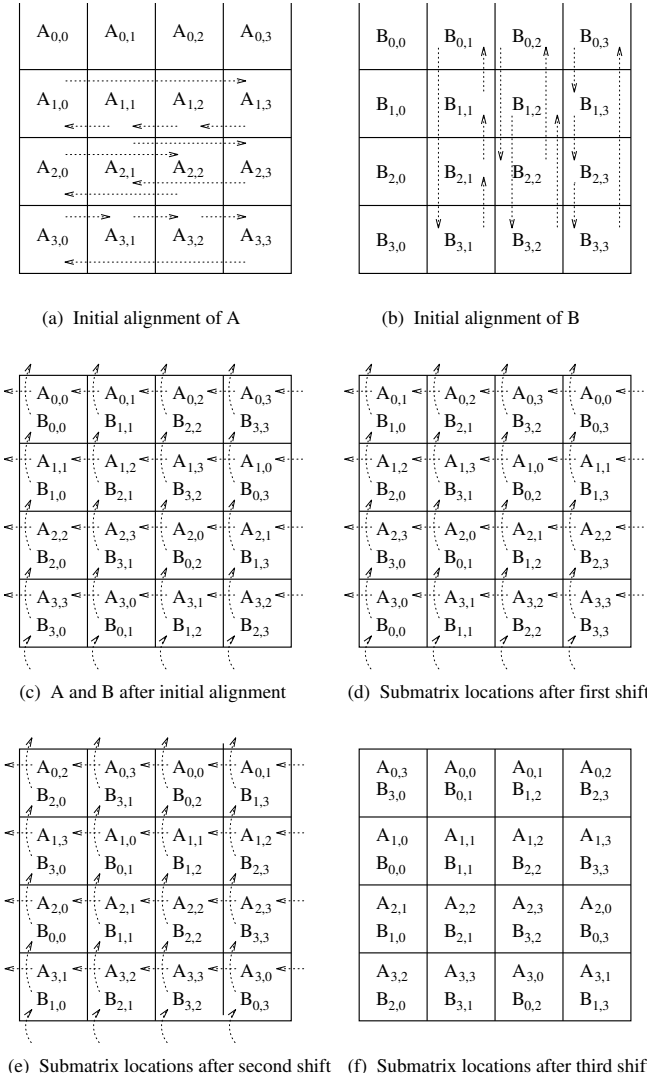
Matrix-Matrix Multiplication: Simple parallelization

- Consider two $n \times n$ matrices A and B partitioned into $p = q \times q$ blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < q)$ of size $(n/q) \times (n/q)$ 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 $0 \leq k < q$.
- All-to-all broadcast blocks of A along rows and B along columns.
- Perform local submatrix multiplication.

Matrix-Matrix Multiplication: Cannon's Algorithm

- In this algorithm, we schedule the computations of the $\sqrt{p} = q$ 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



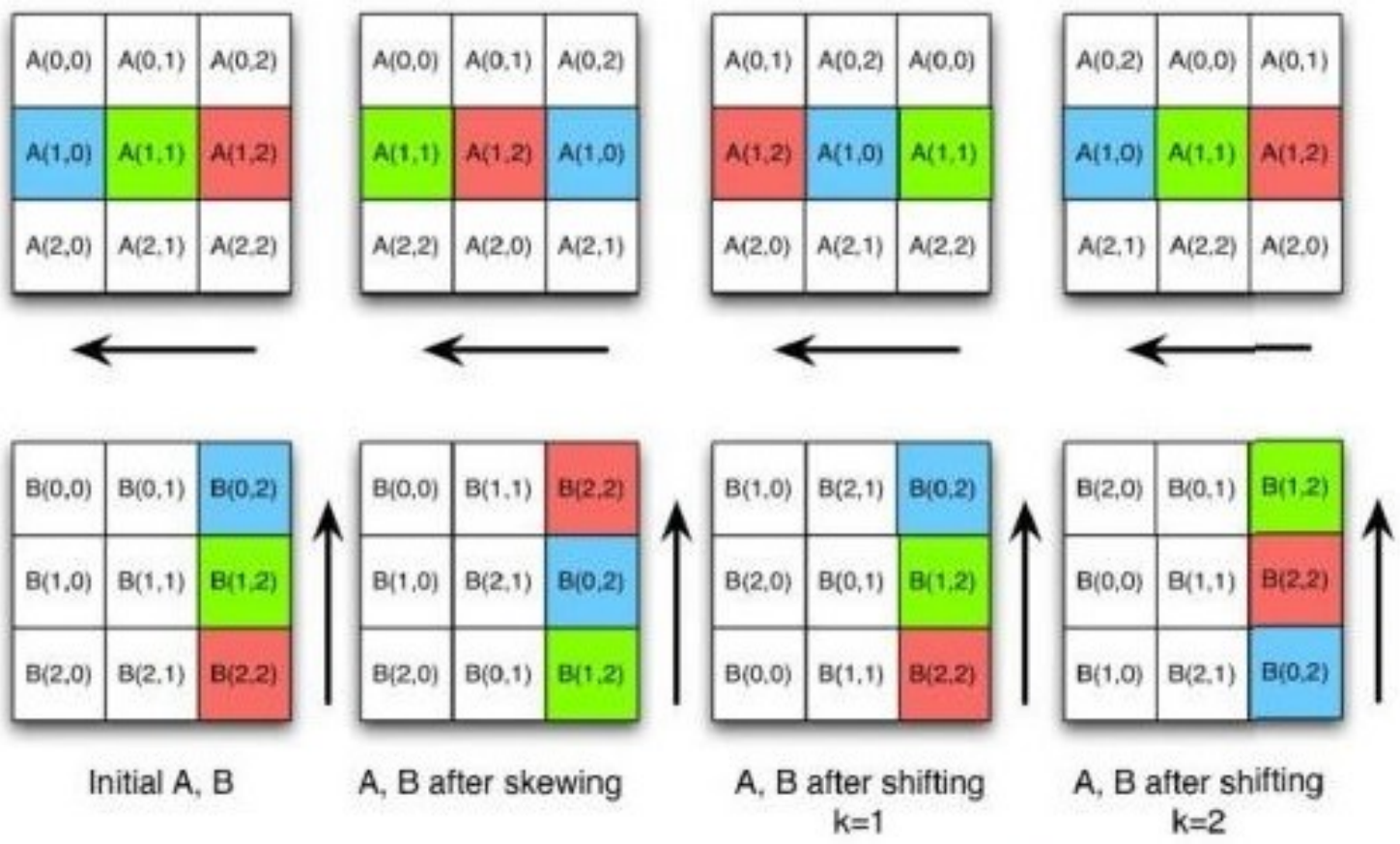
(a) Initial alignment of A (b) Initial alignment of B

(c) A and B after initial alignment (d) Submatrix locations after first shift

(e) Submatrix locations after second shift (f) Submatrix locations after third shif

Communication steps in Cannon's algorithm on 16 processes.

Matrix-Matrix Multiplication: Cannon's Algorithm



Example: Computing $C_{1,2}$ on a 3×3 grid of 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 $\sqrt{p} = q$ blocks have been multiplied.

Solving a System of Linear Equations

Consider the problem of solving linear equations:

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

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

Solving a System of Linear Equations

There are two major steps: (1) Gaussian elimination: row reduction of $Ax = b$ to an upper triangular form, and (2) back-substitution.

The triangular form is as:

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

We write this as: $Ux = y$.

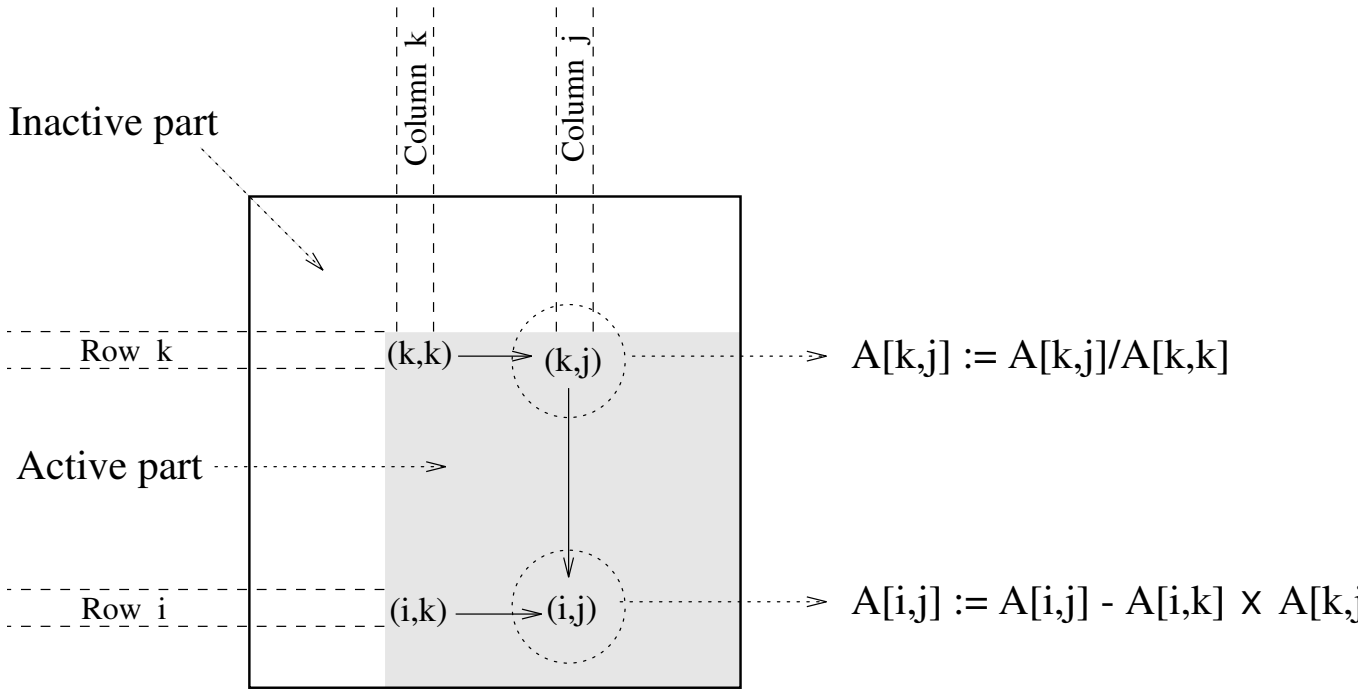
A commonly used method for transforming a given matrix into an upper-triangular matrix is **Gaussian Elimination**.

Gaussian Elimination (serial algorithm)

```
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
```

Gaussian Elimination

The computation has three nested loops – in the kth iteration of the outer loop, the algorithm performs $(n-k)^2$ 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 $(n - k)$ in the kth iteration.
- In the second step, the normalized row is broadcast to all the processors.
- Each processor can independently eliminate this row from its own. This requires $(n - k - 1)$ multiplications and subtractions.

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)

(a) Computation:

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

(ii) A[k,k] := 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	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 brodcast of row A[k,*]

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] x 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 to $k = 3$ for an 8×8 matrix partitioned rowwise among eight processes.

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

- The above algorithm can be easily adapted to the case when $p < n$.
- The rows of matrix A are partitioned evenly among p processors.
  - Blocking 1D partitioning
  - Or, alternatively, cyclic 1D partitioning
- 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.

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

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)

(a) Block 1-D mapping

1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	P_0
0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	
0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	P_1
0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	
0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	P_2
0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	
0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	P_3
0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

(b) Cyclic 1-D mapping

Computation load on different processes in block and cyclic 1-D partitioning of an 8×8 matrix on four processes during the Gaussian elimination iteration corresponding to $k = 3$.

Parallel Gaussian Elimination: Cyclic 1D Mapping

- The load imbalance problem can be alleviated by using a cyclic mapping.
- In this case, other than processing of the last p rows, there is no load imbalance.

Solving a Triangular System: Back-Substitution

The upper triangular matrix U undergoes back-substitution to determine the vector x.

```
1.  procedure BACK_SUBSTITUTION (U, x, y)
2.  begin
3.   for k := n − 1 downto 0 do   /* Main loop */
4.       begin
5.           x[k] := y[k];
6.           for i := k − 1 downto 0 do
7.               y[i] := y[i] − x[k] × U[i, k];
8.       endfor;
9.  end BACK_SUBSTITUTION
```

Solving a Triangular System: Back-Substitution

- The algorithm performs approximately $n^2/2$ multiplications and subtractions.
- Since complexity of this part is asymptotically lower, we should optimize the data distribution for the factorization part.
- Consider a rowwise block 1-D mapping of the $n \times n$ matrix U with vector y distributed uniformly.
- The value of the variable solved at a step can be pipelined back.