Design of Parallel Algorithms

Parallel Dense Matrix Algorithms

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

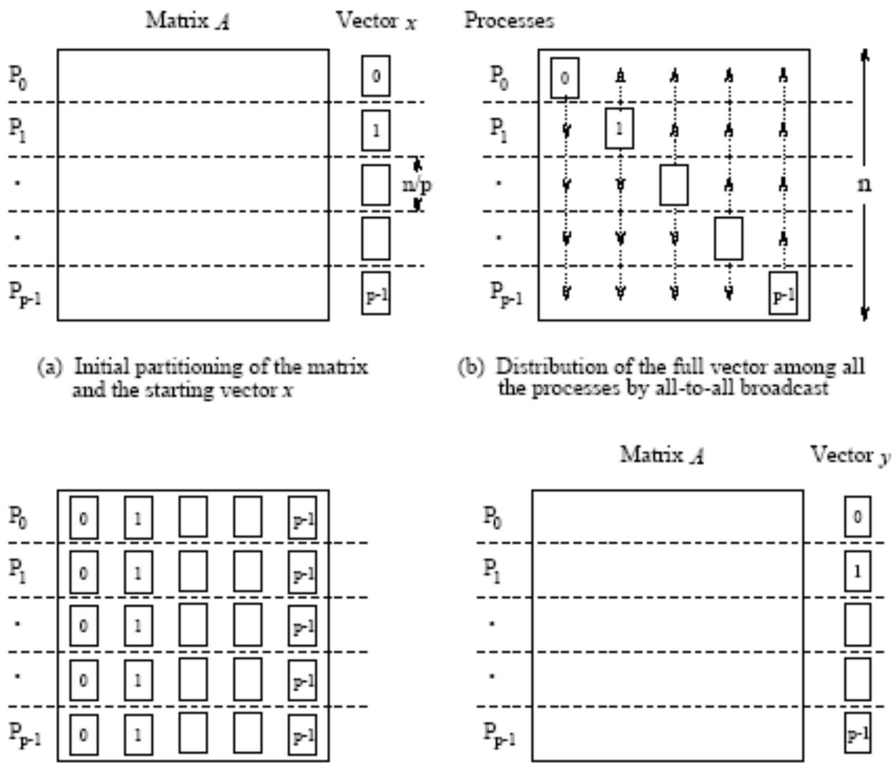
- We aim to multiply a dense n x n matrix A with an n x 1 vector x to yield the n x 1 result vector y.
- The serial algorithm requires n^2 multiplications and additions.

$$W = n^2$$

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

(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. 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 $y[i] = \sum_{j=0}^{n-1} \left(A[i,j] \times x[j] \right)$
- The all-to-all broadcast and the computation of $y[i]$ both take time $\Theta(n)$. 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

$$T_P = \overbrace{\frac{n^2}{p}}^{local\,operations} + \overbrace{t_s \log p + t_w n}^{all-to-all}$$

This is cost-optimal.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

Scalability Analysis:

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

$$T_O = t_s p \log p + t_w np = t_s p \log p + t_w \sqrt{W}\, p$$

- For isoefficiency, we have $W = KT_0$ which the second term gives:

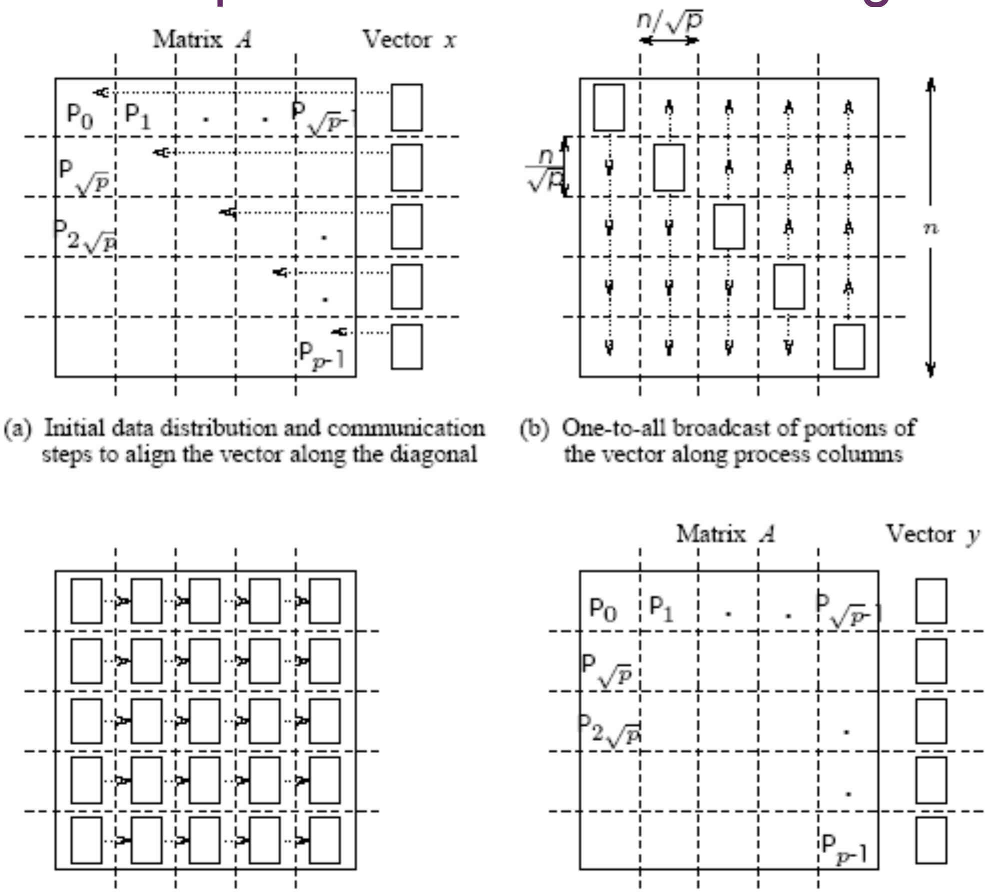
$$W = Kt_w \sqrt{W}\, p \Rightarrow \sqrt{W} = Kt_w p \Rightarrow W = K^2 t_w^2 p^2$$

- 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 n^2 processors such that each processor owns a single element.
- The n x 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. For the one-element-per-process case, $p = n^2$ if the matrix size is $n \times n$.

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 *(one element per processor)*

- Three basic communication operations are used in this algorithm: one-to-one communication $\Theta(1)$ to align the vector along the main diagonal, one-to-all broadcast $\Theta(\log n)$ of each vector element among the n processes of each column, and all-to-one reduction $\Theta(\log n)$ in each row.
- Each of these operations takes at most $\Theta(\log n)$ 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 block of the matrix $(n/\sqrt{p}) \times (n/\sqrt{p})$.
- 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

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

- The broadcast and reductions take time

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

- 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:

$$T_O = pT_P - W = t_s p \log p + t_w \sqrt{W} \sqrt{p} \log p$$

- Equating T_0 with W, term by term, for isoefficiency, we have the dominant term:

$$W = K^2 t_w^2 p \log^2 p$$

- The isoefficiency due to concurrency is $O(p)$.
- The overall isoefficiency is $\Theta(p \log^2 p)$

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 $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) 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 $0 \leq k < \sqrt{p}$.
- Naïve Algorithm:
  - All-to-all broadcast blocks of A along rows and B along columns.
  - Perform local submatrix multiplication.

Matrix-Matrix Multiplication

- The two broadcasts take time $2\left(t_s \log \sqrt{p} + t_w \left(n^2 / p\right)\left(\sqrt{p} - 1\right)\right)$
- The computation requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sized submatrices.
- The parallel run time is approximately

$$T_P \cong \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$$

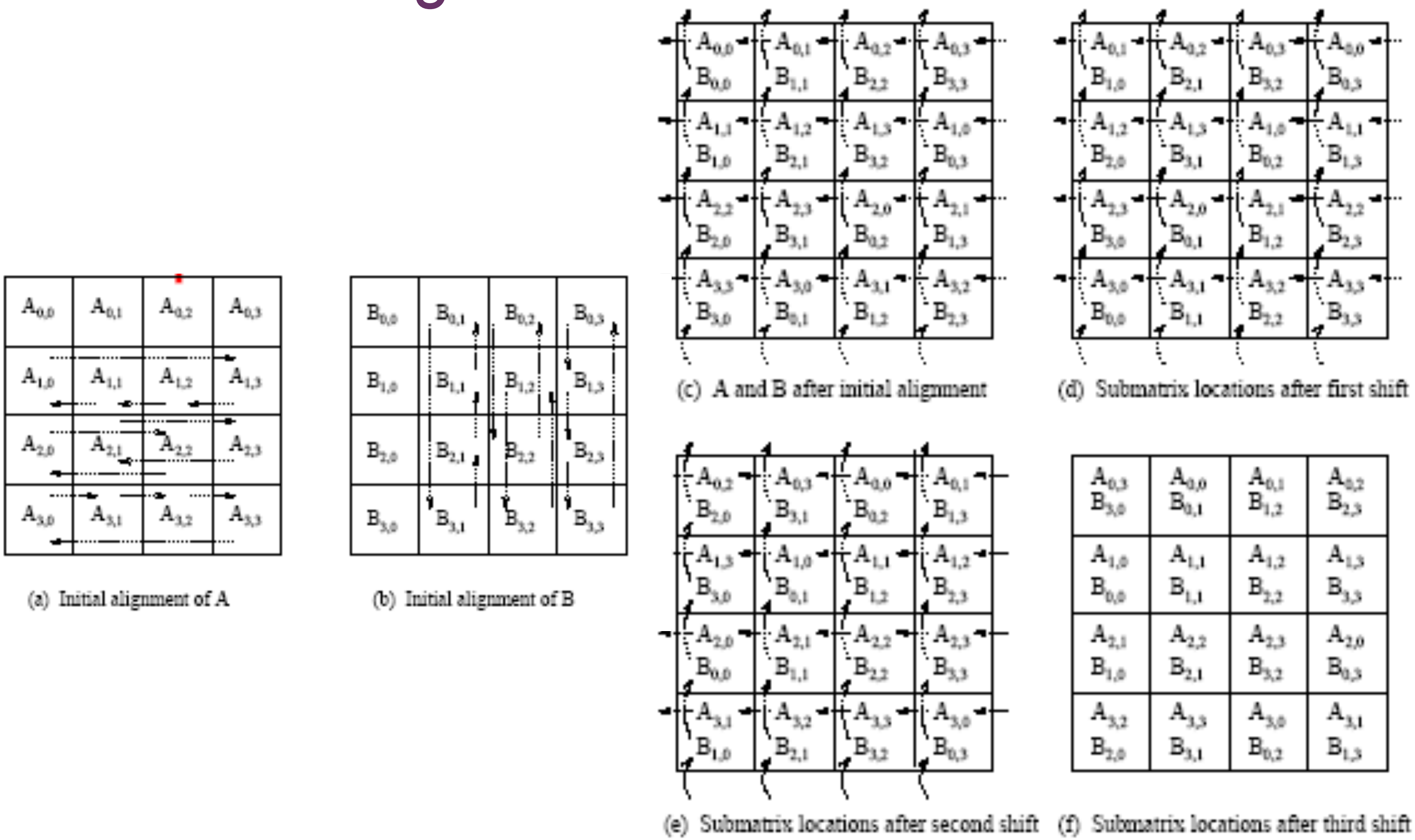
- The algorithm is cost optimal and the isoefficiency is $O(p^{1.5})$ due to bandwidth term t_w and concurrency.
- Major drawback of the algorithm is that it is not memory optimal.

+

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 16 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.

- Do the following for $\sqrt{p}$ 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 blocks have been multiplied.

Matrix-Matrix Multiplication: Cannon's Algorithm

- In the alignment step the two shift operations require a total of time of each processor communicating 1 block:

$$T_{align} = 2\left(t_s + t_w n^2 / p\right)$$

- Each of the single-step shifts in the compute-and-shift phase of the algorithm takes time.

$$T_{shiftCompute} = t_c \frac{n^3}{p^{3/2}} + 2\left(t_s + t_w n^2 / p\right)$$

- 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, although with larger factors on communication time.   This algorithm is memory optimal however!

+

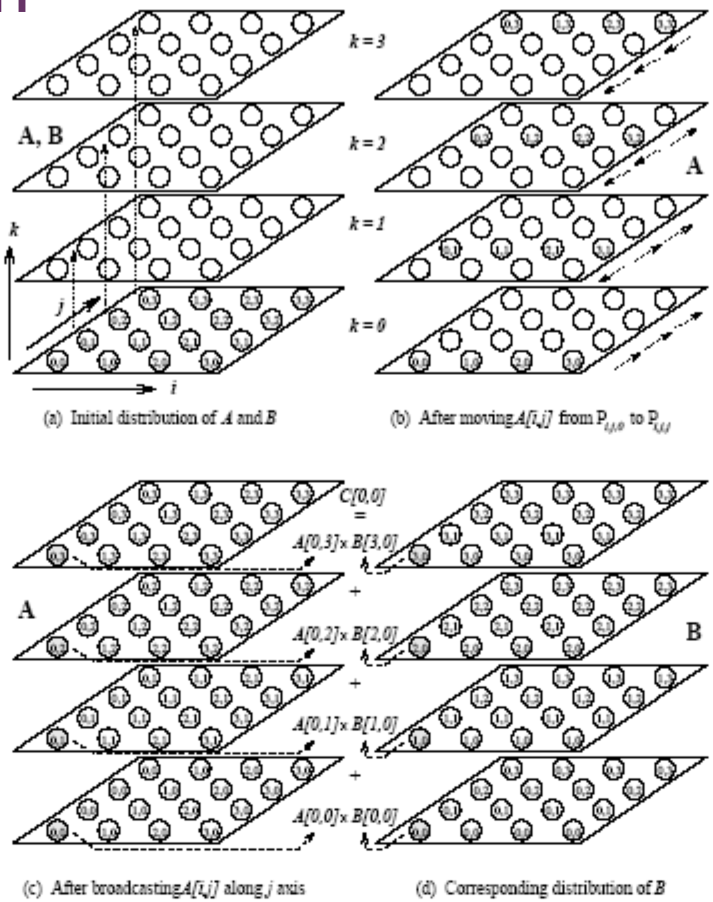
Matrix-Matrix Multiplication: DNS Algorithm

- 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

- 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



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

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.

- Assume running on $p=q^3$ procesors
- 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.
- The reduction takes time $t_s \log q + t_w (n/q)^2 \log q$.
- Multiplication of $(n/q)\times(n/q)$ submatrices is performed serially and takes $(n/q)^3$ time.
  - Note that a 3-D block that is assigned to a given processor represents that matrix of a $(n/q)\times(n/q)$ sub-matrix of A and B (the third dimension represents the k loop of the sub-matrix multiply!)

DNS blocked algorithm

- For parallel running time we assemble the parts to get

$$T_P = t_s + t_w (n/q)^2 + 3\left(t_s + t_w (n/q)^2\right)\log q + (n/q)^3$$

- Recall that $p=q^3$ which we can substitute into the above equation to obtain

$$T_P = \frac{n^3}{p} + \left(t_s + t_w \frac{n^2}{p^{2/3}}\right)(1 + \log p)$$

- This gives a parallel overhead function of

$$T_O = \left(t_s + t_w \frac{W^{2/3}}{p^{2/3}}\right)(p + p\log p) = \Theta\left(W^{2/3} p^{1/3} \log p\right)$$

Computing the Isoefficiency function of the blocked DNS algorithm

- Isoefficiency function is found to be $W = f(O(p\ (\log p)^3)$ as shown below:

$$W = KT_O(W, p)$$

$$W = KW^{2/3} p^{1/3} \log p$$

$$W^{1/3} = Kp^{1/3} \log p$$

$$W = K^3 p (\log p)^3$$

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 $Ax = b$, where A is an n X n matrix with $A[i, j] = a_{i,j}$, b is an n X 1 vector $[\, b_0, b_1, \ldots, b_n \,]^T$, 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:

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

We write this as: $Ux = y$.

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

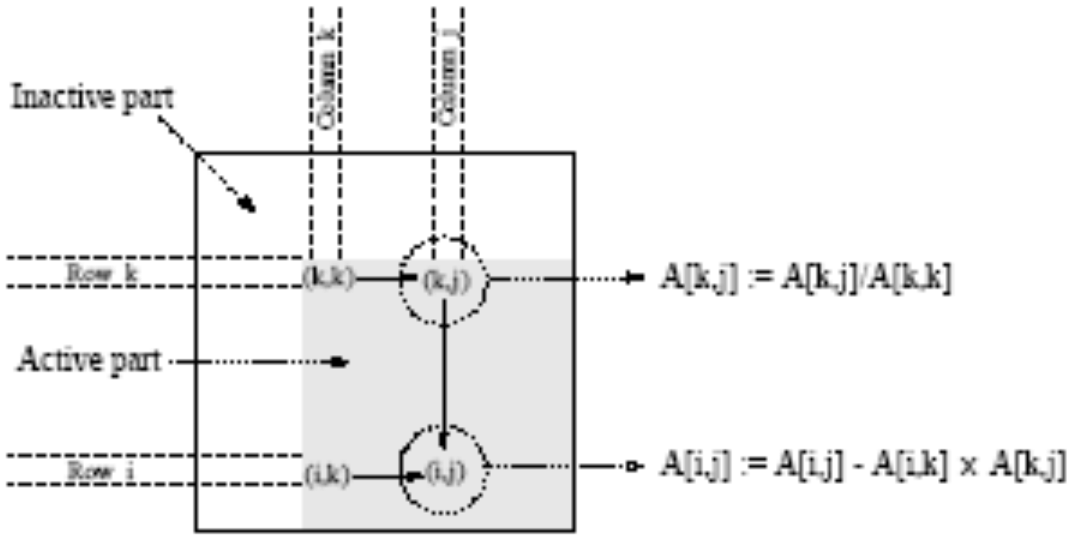
Gaussian Elimination

```
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 $(n-k)^2$ computations. Summing from $k = 1..n$, we have roughly $(n^3/3)$ multiplications-subtractions.



A typical computation in Gaussian elimination.

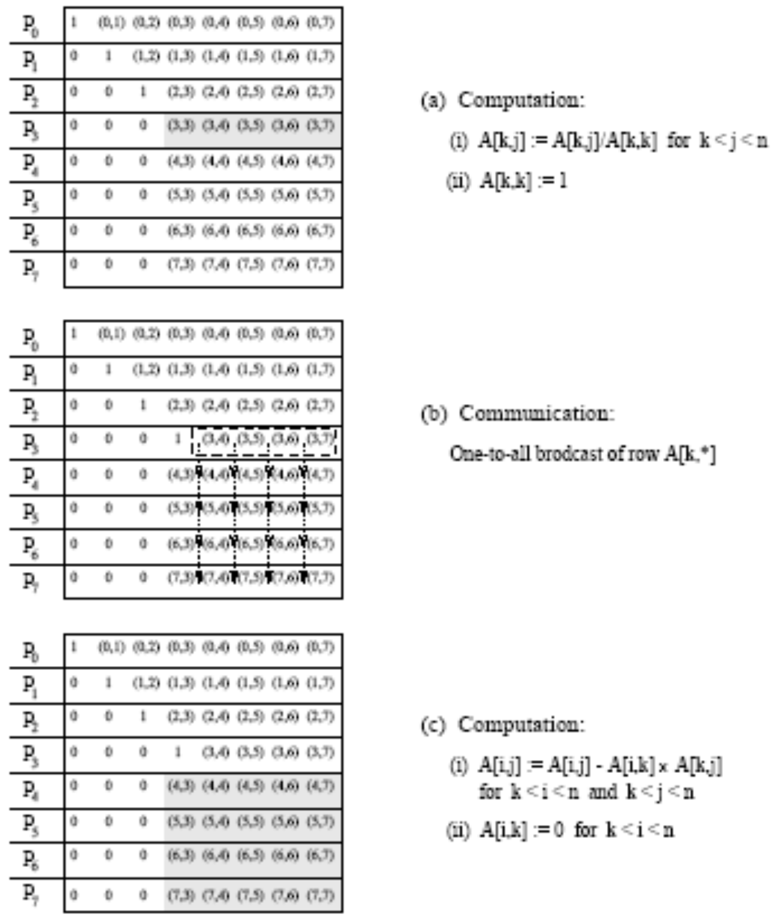
Parallel Gaussian Elimination

- 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 kth 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

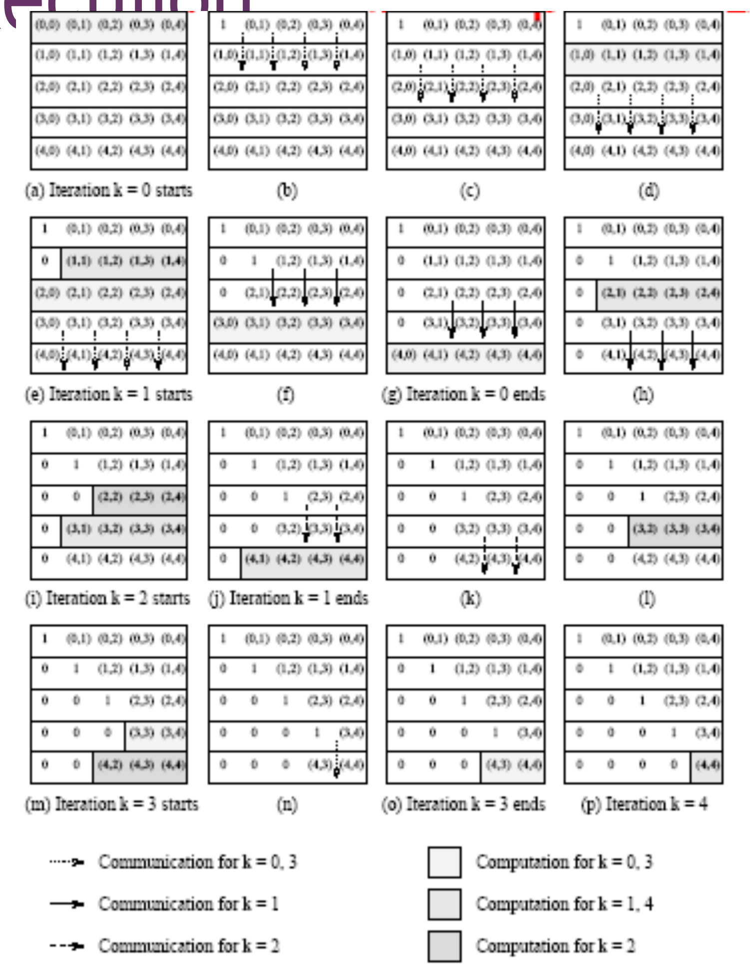


Gaussian elimination steps during the iteration corresponding $k = 3$ for an 8 x 8 matrix partitioned rowwise among eight processes.

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 $Pk+1$.

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 $(n - k - 1) / np$ 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.

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 x 8 matrix on four processes during the Gaussian elimination iteration corresponding to $k = 3$.

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

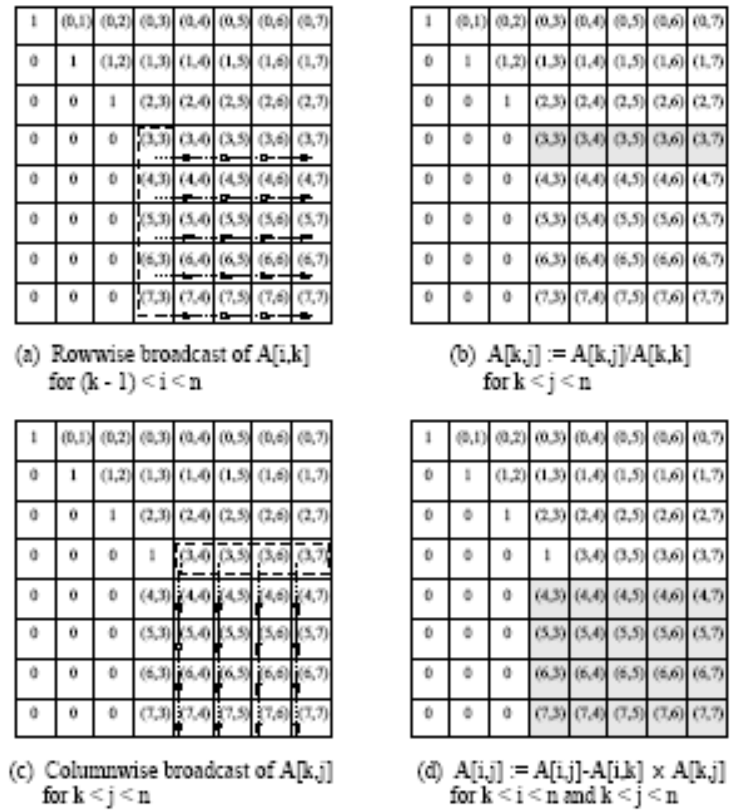
- The load imbalance problem can be alleviated by using a cyclic mapping.
- In this case, other than processing of the last p rows, there is no load imbalance.
- This corresponds to a cumulative load imbalance overhead of $O(n^2p)$ (instead of $O(n^3)$ in the previous case).

+

Parallel Gaussian Elimination: 2-D Mapping

- Assume an n X n matrix A mapped onto an n X n mesh of processors.
- Each update of the partial matrix can be thought of as a scaled rank-one update (scaling by the pivot element).
- In the first step, the pivot is broadcast to the row of processors.
- In the second step, each processor locally updates its value. For this it needs the corresponding value from the pivot row, and the scaling value from its own row.
- This requires two broadcasts, each of which takes $\log n$ time.
- This results in a non-cost-optimal algorithm.

Parallel Gaussian Elimination: 2-D Mapping



Various steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix on 64 processes arranged in a logical two-dimensional mesh.

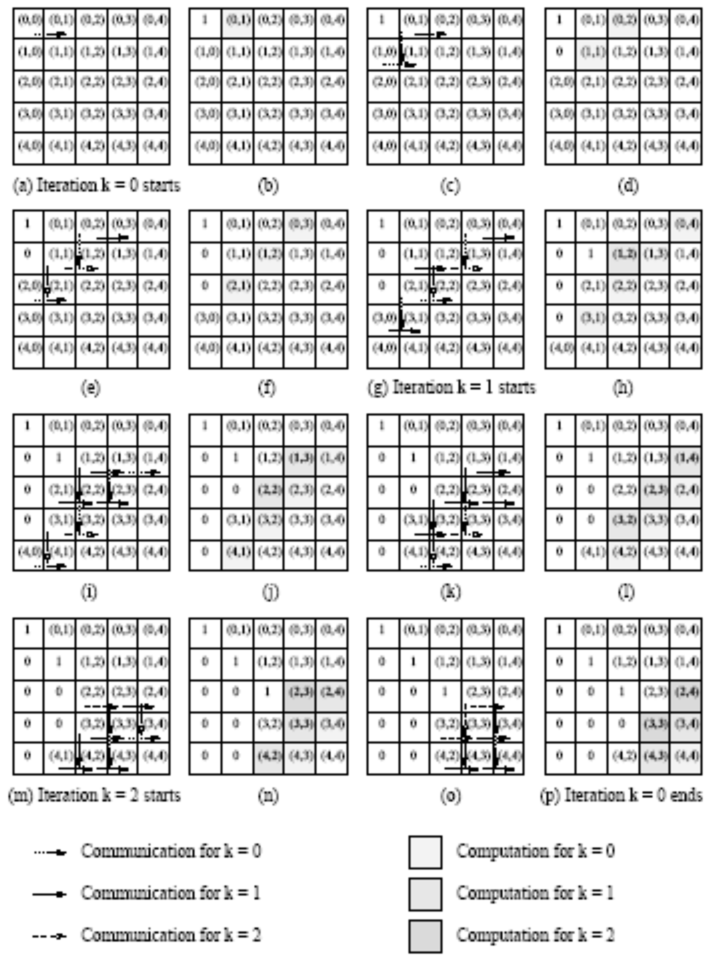
Parallel Gaussian Elimination: 2-D Mapping with Pipelining

- We pipeline along two dimensions. First, the pivot value is pipelined along the row. Then the scaled pivot row is pipelined down.
- Processor $P_{i,j}$ (not on the pivot row) performs the elimination step $A[i, j] := A[i, j] - A[i, k]\, A[k, j]$ as soon as $A[i, k]$ and $A[k, j]$ are available.
- The computation and communication for each iteration moves through the mesh from top-left to bottom-right as a ``front.''
- After the front corresponding to a certain iteration passes through a process, the process is free to perform subsequent iterations.
- Multiple fronts that correspond to different iterations are active simultaneously.

Parallel Gaussian Elimination: 2-D Mapping with Pipelining

- If each step (division, elimination, or communication) is assumed to take constant time, the front moves a single step in this time. The front takes $\Theta(n)$ time to reach $P_{n-1,n-1}$.
- Once the front has progressed past a diagonal processor, the next front can be initiated. In this way, the last front passes the bottom-right corner of the matrix $\Theta(n)$ steps after the first one.
- The parallel time is therefore $O(n)$, which is cost-optimal.

2-D Mapping with Pipelining



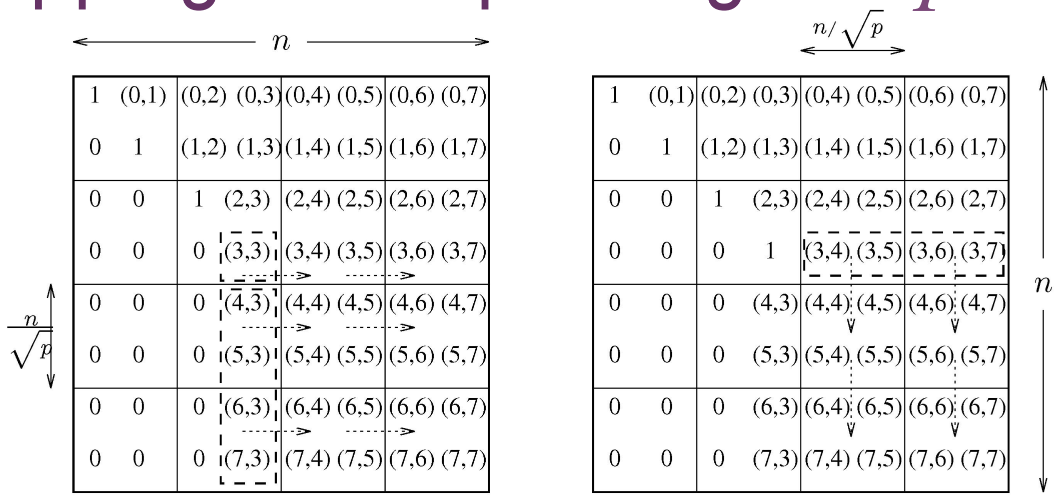
Pipelined Gaussian elimination for a 5 x 5 matrix with 25 processors.

Parallel Gaussian Elimination: 2-D Mapping with Pipelining and $p < n$

- In this case, a processor containing a completely active part of the matrix performs n^2/p multiplications and subtractions, and communicates $n/\sqrt{p}$ words along its row and its column.
- The computation dominates communication for $n >> p$.
- The total parallel run time of this algorithm is $(2n^2/p)$ x n, since there are n iterations. This is equal to $2n^3/p$.
- This is three times the serial operation count!

+

Parallel Gaussian Elimination: 2-D Mapping with Pipelining and $p < n$

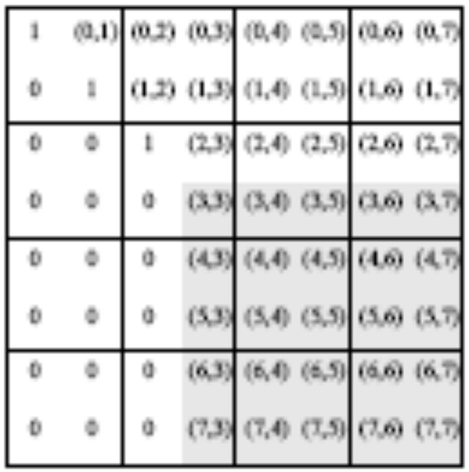


(a) Rowwise broadcast of A[i,k] for i = k to (n - 1)

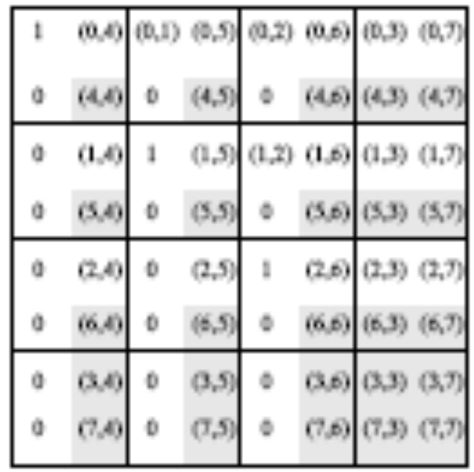
(b) Columnwise broadcast of A[k,j] for j = (k + 1) to (n - 1)

The communication steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 x 8 matrix on 16 processes of a two-dimensional mesh.

+
Parallel Gaussian Elimination: 2-D Mapping with Pipelining and $p < n$



(a) Block-checkerboard mapping



(b) Cyclic-checkerboard mapping

Computational load on different processes in block and cyclic 2-D mappings of an 8 x 8 matrix onto 16 processes during the Gaussian elimination iteration corresponding to $k = 3$.

Parallel Gaussian Elimination: 2-D Cyclic Mapping

- The idling in the block mapping can be alleviated using a cyclic mapping.
- The maximum difference in computational load between any two processes in any iteration is that of one row and one column update.
- This contributes $\Theta(n\sqrt{p})$ to the overhead function. Since there are n iterations, the total overhead is $\Theta(n^2\sqrt{p})$.

Gaussian Elimination with Partial Pivoting

- For numerical stability, one generally uses partial pivoting.
- In the k th iteration, we select a column i (called the *pivot* column) such that $A[k, i]$ is the largest in magnitude among all $A[k, i]$ such that $k \leq j < n$.
- The k th and the i th columns are interchanged.
- Simple to implement with row-partitioning and does not add overhead since the division step takes the same time as computing the max.
- Column-partitioning, however, requires a global reduction, adding a $\log p$ term to the overhead.
- Pivoting precludes the use of pipelining.

Gaussian Elimination with Partial Pivoting: 2-D Partitioning

- Partial pivoting restricts use of pipelining, resulting in performance loss.
- This loss can be alleviated by restricting pivoting to specific columns.
- Alternately, we can use faster algorithms for broadcast.

Solving a Triangular System: Back-Substitution

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

```
procedure BACK_SUBSTITUTION (U, x, y)
begin
    for k := n - 1 downto 0 do    /* Main loop */
        begin
            x[k] := y[k];
            for i := k - 1 downto 0 do
                y[i] := y[i] - x[k] × U[i, k];
        endfor;
end BACK_SUBSTITUTION
```

A serial algorithm for 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 x n matrix U with vector y distributed uniformly.
- The value of the variable solved at a step can be pipelined back.
- Each step of a pipelined implementation requires a constant amount of time for communication and $\Theta(n/p)$ time for computation.
- The parallel run time of the entire algorithm is $\Theta(n^2/p)$.

+

Solving a Triangular System: Back-Substitution

- If the matrix is partitioned by using 2-D partitioning on a logical mesh of $\sqrt{p} \times \sqrt{p}$ processes, and the elements of the vector are distributed along one of the columns of the process mesh, then only the $\sqrt{p}$ processes containing the vector perform any computation.

- Using pipelining to communicate the appropriate elements of U to the process containing the corresponding elements of y for the substitution step (line 7), the algorithm can be executed in $\Theta(n^2/\sqrt{p})$ time.

- While this is not cost optimal, since this does not dominate the overall computation, the cost optimality is determined by the factorization.