CZ4102 – High Performance Computing

Lecture 11:

Matrix Algorithms

- Dr Tay Seng Chuan

Reference:

"Introduction to Parallel Computing" – Chapter 8.

1

Topic Overview

- Parallel Algorithms for
  - Matrix-Vector Multiplication
  - Matrix-Matrix Multiplication
- Performance Analysis

2

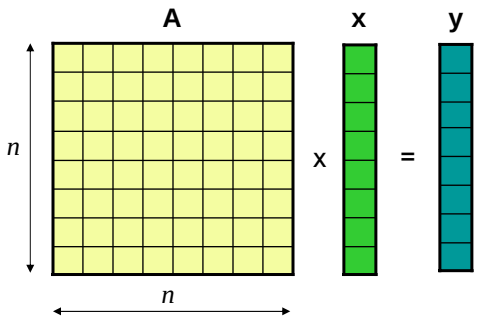
Matrix Algorithms: Introduction

- 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 partitions for parallel processing.
- The run-time performance of such algorithms depends on the amount of overheads incurred as compared to the computation workload.
- As a rule of thumb, good speedup can be achieved if the computation granularity is able to outweigh the overheads such as the communication cost, consolidation cost – algorithm penalty, data packaging, etc.

3

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.

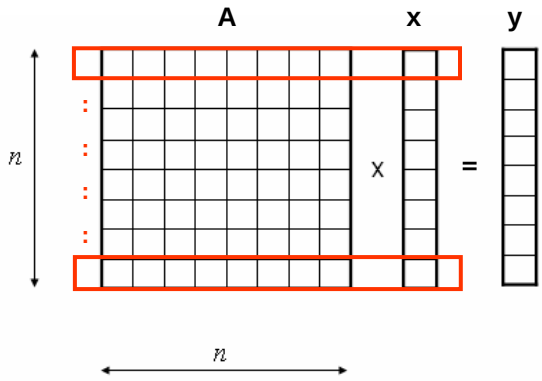


- The total workload is $W = n^2$.

4

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

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

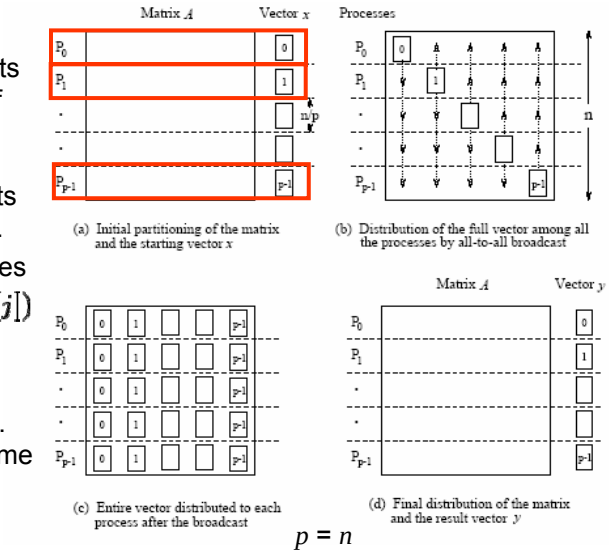


5

Matrix-Vector Multiplication: Rowwise 1-D Partitioning ($p = n$)

$$y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$$

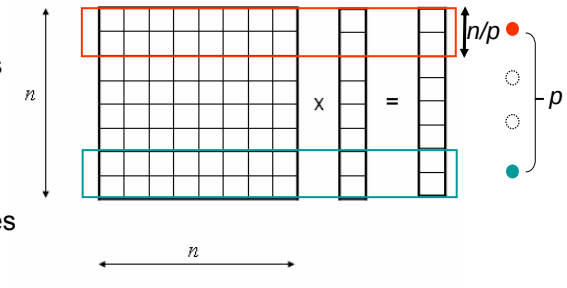
- Since each process starts with only one element of Vector x , an all-to-all broadcast is required to distribute all the elements $x[j]$ to all the processes.
- Process P_i now computes $y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$
- The all-to-all broadcast and the computation of $y[i]$ both take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



$p = n$

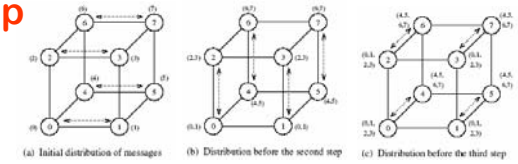
Matrix-Vector Multiplication: Rowwise 1-D Partitioning ($p < n$)

- Consider now the case when $p < n$ and we use blocks in partitions.
- 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 rows of* local dot products. (computation time = n/p x $n = n^2/p$)

Recap: All-to-All Broadcast based on Doubling-up Approach:



$$T = \sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m)$$

$$= t_s \log p + t_w m(p-1).$$

Now $m = \frac{n}{p}$, **we have**

$$T = t_s \log p + t_w \text{ x } \frac{n}{p} \text{ x } \cancel{(p-1)}$$

$$\simeq t_s \log p + t_w \text{ x } n$$

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{\log p - 1} = \frac{1(2^{(\log p - 1)+1} - 1)}{2-1} = p - 1$$

- **Thus, the parallel run time of matrix-vector multiplication based on rowwise 1-D partitioning ($p < n$) is**

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

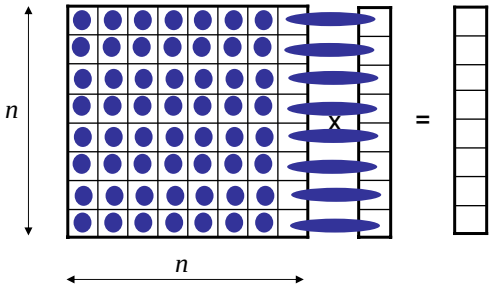
All-to-all broadcast of vector elements.

- This is cost-optimal.

Matrix-Vector Multiplication: 2-D Partitioning ($p = n^2$)

- 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.

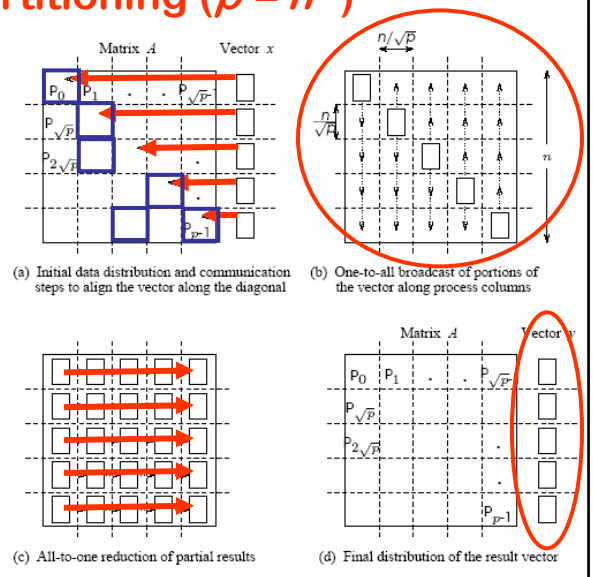
($p = n^2$ processors)



9

Matrix-Vector Multiplication: 2-D Partitioning ($p = n^2$)

- 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.



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

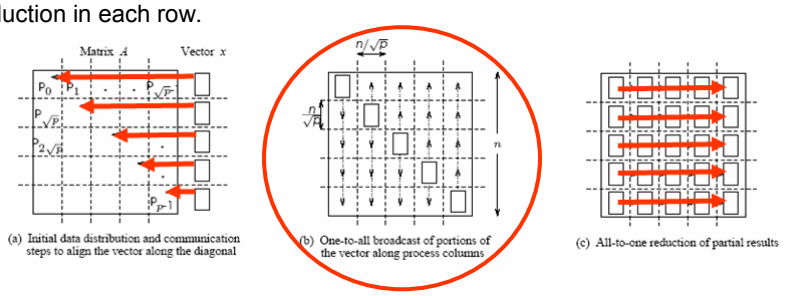
(b) One-to-all broadcast of portions of the vector along process columns

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

(d) Final distribution of the result vector

Matrix-Vector Multiplication: 2-D Partitioning ($p = n^2$)

- 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.

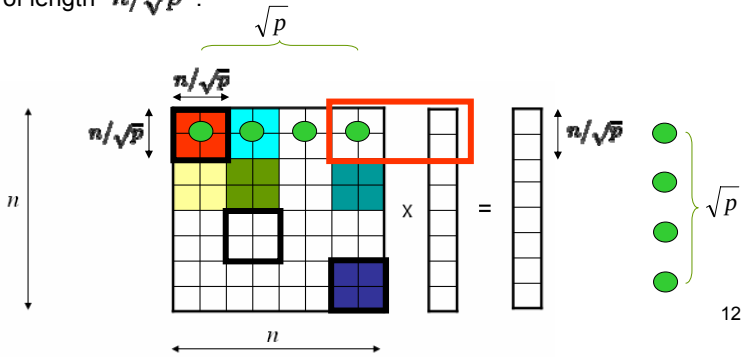


  - These communications take $\Theta(\log n)$ time. Computation time is $O(1)$. The parallel time of this algroithm is $\Theta(\log n) + \Theta(1) = \Theta(\log n)$.
  - The cost (process-time product) is $n^2 \times \log n = \Theta(n^2 \log n) > n^2$; hence, the algorithm is not cost-optimal.

11

Matrix-Vector Multiplication: 2-D Partitioning ($p < n^2$)

- When using fewer than n^2 processors, each process owns an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix. p (ie, $\sqrt{p} \times \sqrt{p}$) processors are used.
- 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 $n/\sqrt{p}$.



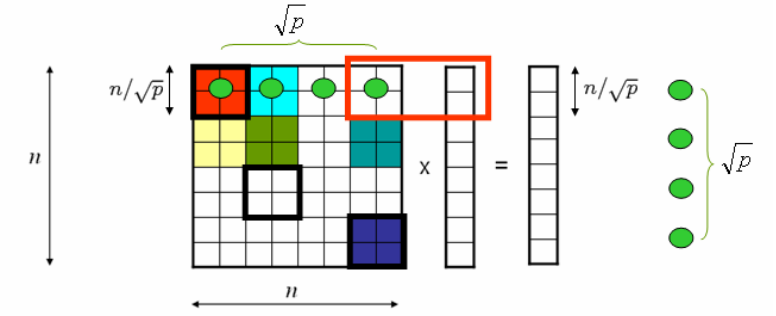
12

Matrix-Vector Multiplication: 2-D Partitioning ($p < n^2$)

- The first alignment step takes time $t_s + t_w n/\sqrt{p}$
- The broadcast and reductions each 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 $T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$

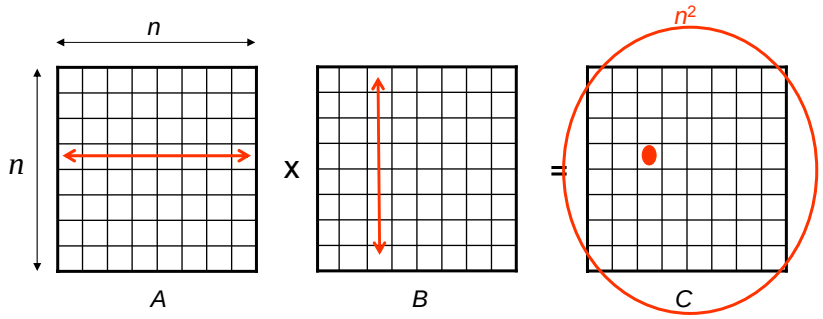
13

Matrix-Matrix Multiplication

- Consider the problem of multiplying two n x n dense, square matrices A and B to yield the product matrix

$$C = A \times B.$$

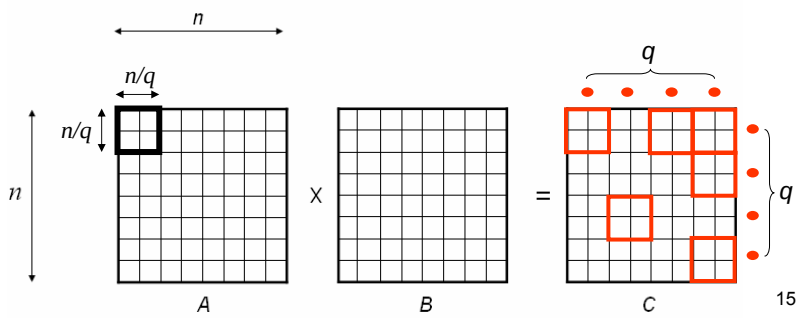
- The serial complexity is $O(n^3)$.



14

Matrix-Matrix Multiplication

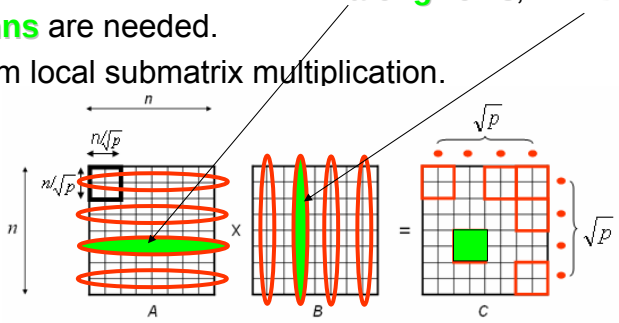
- A useful concept in this case is called *block* operations. In this view, an n x n matrix A can be regarded as a q x q array of blocks $A_{i,j}$ ($0 \le i, j < q$) such that each block is an (n/q) x (n/q) submatrix.
- We perform q^2 matrix multiplications, each involving (n/q) x (n/q) matrices.



15

Matrix-Matrix Multiplication

- Consider two n x n matrices A and B partitioned into p blocks of $A_{i,j}$ and $B_{i,j}$ ($0 \le i, j < \sqrt{p}$) 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 $0 \le k < \sqrt{p}$.
- All-to-all broadcast blocks of **A along rows**, and **B along columns** are needed.
- Perform local submatrix multiplication.

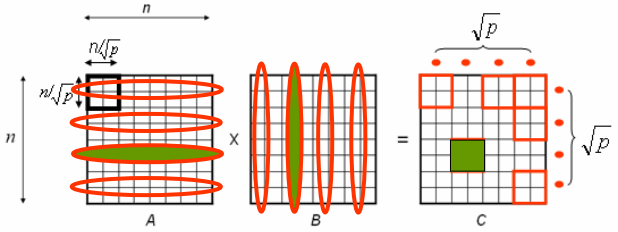


16

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

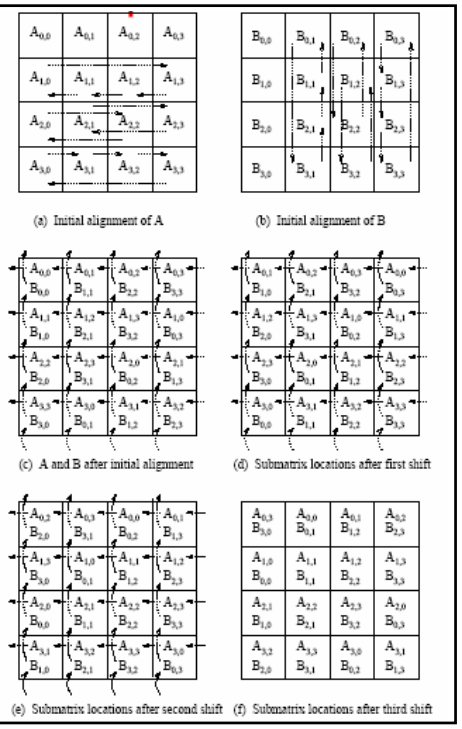
$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}.$$

- Major drawback of the algorithm is that it is not memory optimal.

17

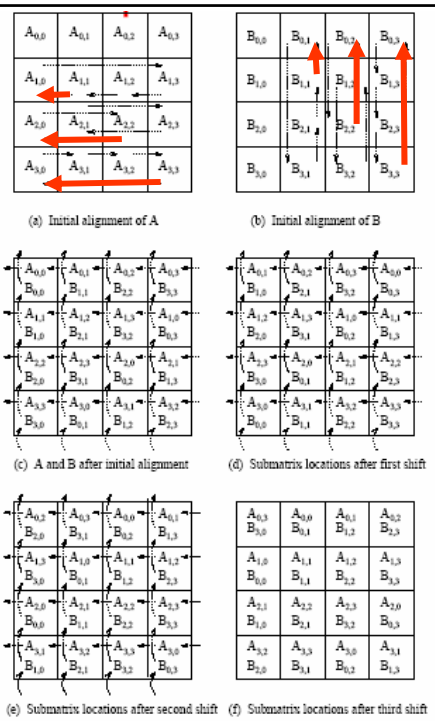
Matrix-Matrix Multiplication: Cannon's Algorithm

- 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

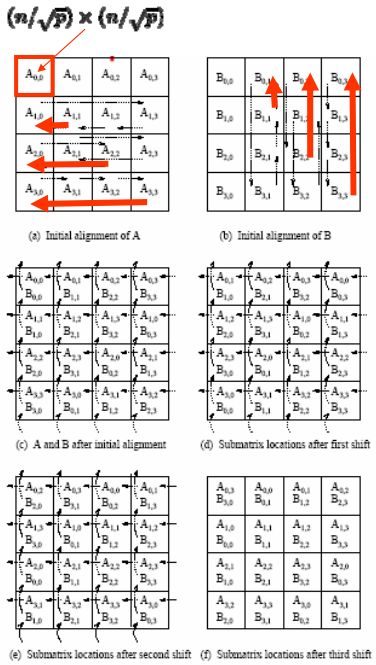
- Align the blocks of A and B in such a way that each process multiplies its local submatrices. This is done by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by i steps and all submatrices $B_{i,j}$ up (with wraparound) by j steps.
- Perform local block multiplication.
- Each block 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

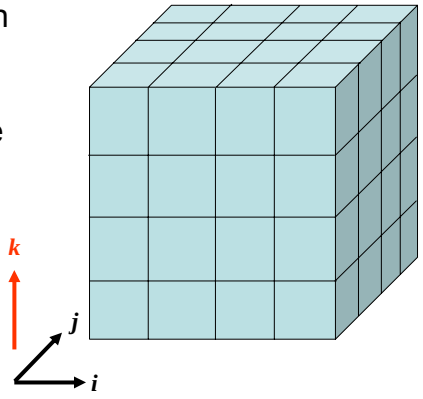
- In the alignment step, since the maximum distance over which a block shifts is $\sqrt{p} - 1$, the two shift operations require a total of $2(t_s + t_w n^2/p)$ time.
- Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase of the algorithm takes $t_s + t_w n^2/p$ time.
- The computation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

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



Matrix-Matrix Multiplication: DNS (Dekel, Nassimi, and Sahni) Algorithm

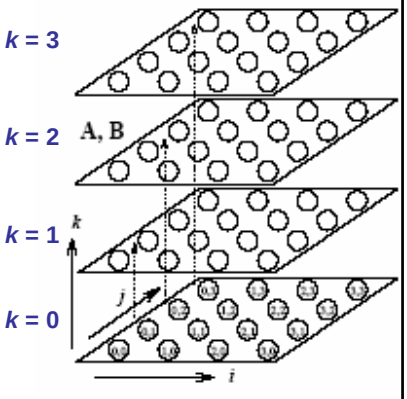
- Uses a 3-D partitioning.
- Visualize the matrix multiplication algorithm as a cube. Matrices A and B come in two orthogonal faces and result C comes out the other orthogonal face.
- Each internal node in the cube represents a single add-multiply operation (and thus the complexity).
- DNS algorithm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



21

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 $P_{i, j, k}$ computes a single multiply: C[i,k] = A[i,k] x B[k,j].
- This is followed by an accumulation along the *k* 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.

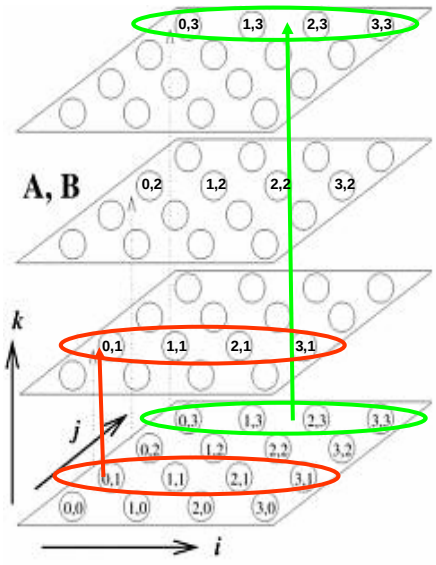


22

Matrix-Matrix Multiplication: DNS Algorithm

The vertical column of processes $P_{i,j,*}$ computes the dot product of row A[i, *] and column B[*, j]. Therefore, rows of A and columns of B need to be moved appropriately so that each vertical column of processes $P_{i,j,*}$ has row A[i, *] and column B[*, j]. More precisely, process $P_{i,j,k}$ should have A[i, k] and B[k, j].

First, each column of A moves to a different plane such that the j th column occupies the same position in the plane corresponding to k = j as it initially did in the plane corresponding to k = 0.

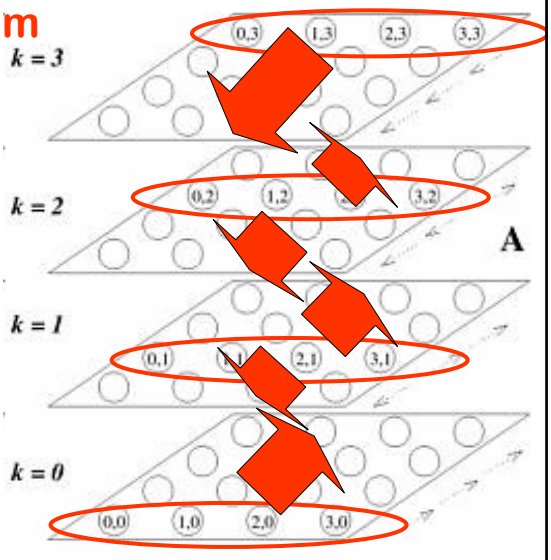


(a) Initial distribution of *A* and *B*

Matrix-Matrix Multiplication: DNS Algorithm

Now all the columns of A are replicated n times in their respective planes by a parallel one-to-all broadcast along the j axis.

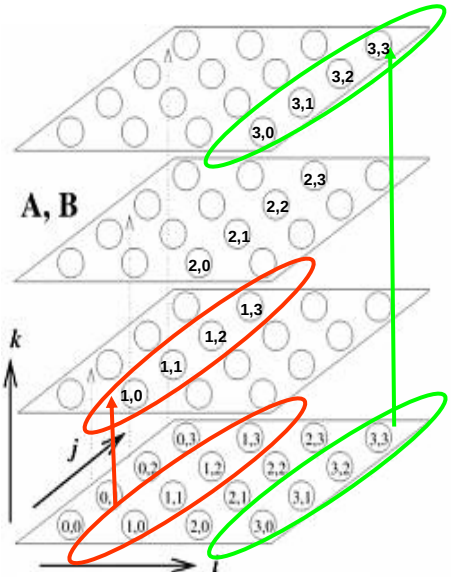
$P_{i,0,j}$, $P_{i,1,j}$, ..., $P_{i,n-1,j}$ receive a copy of A[i, j] from $P_{i,j,j}$. At this point, each vertical column of processes $P_{i,j,*}$ has row A[i, *]. More precisely, process $P_{i,j,k}$ has A[i, k].



(b) After moving *A[i,j]* from $P_{i,j,0}$ to $P_{i,j,j}$

Matrix-Matrix Multiplication: DNS Algorithm

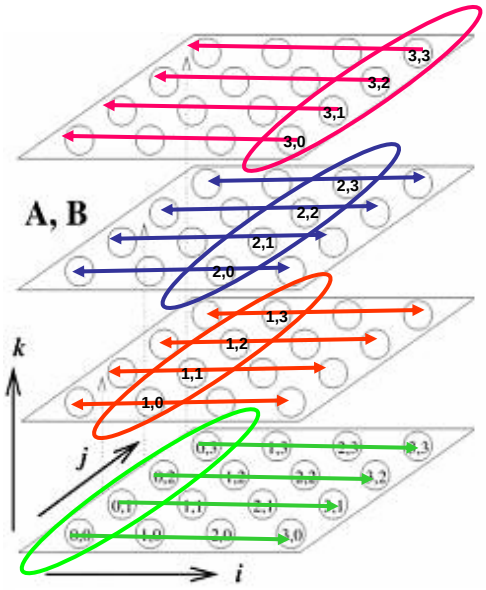
For matrix B, the communication steps are similar, but the roles of i and j in process subscripts are switched. In the first one-to-one communication step, B[i, j] is moved from $P_{i,j,0}$ to $P_{i,j,i}$.



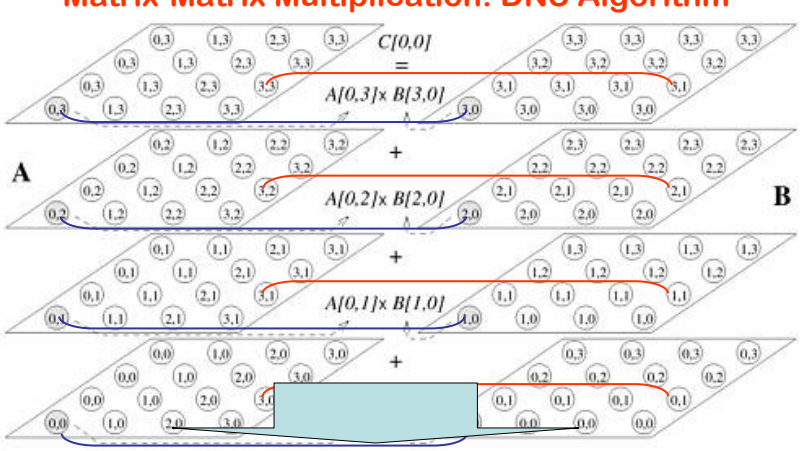
Matrix-Matrix Multiplication: DNS Algorithm

Then it is broadcast from $P_{i,j,i}$ among $P_{0,j,i}$, $P_{1,j,i}$, ..., $P_{n-1,j,i}$.

At this point, each vertical column of processes $P_{i,j,*}$ has column B[*, j]. Now process $P_{i,j,k}$ has B[k, j], in addition to A[i, k].



Matrix-Matrix Multiplication: DNS Algorithm



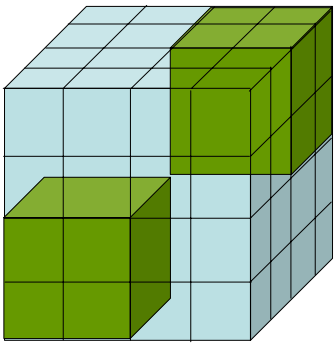
After these communication steps, A[i, k] and B[k, j] are multiplied at $P_{i,j,k}$. Now each element C[i, j] of the product matrix is obtained by an all-to-one reduction along the k axis. During this step, process $P_{i,j,0}$ accumulates the results of the multiplication from processes $P_{i,j,1}$, ..., $P_{i,j,n-1}$.

The DNS algorithm has three main communication steps: (1) moving the columns of A and the rows of B to their respective planes, (2) performing one-to-all broadcast along the j axis for A, and along the i axis for B, and (3) all-to-one reduction along the k axis. All these operations are performed within groups of n processes and take time O(log n). Thus, **the parallel run time for multiplying two n x n matrices using the DNS algorithm on n^3 processes is O(log n).**

27

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.



28

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:

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