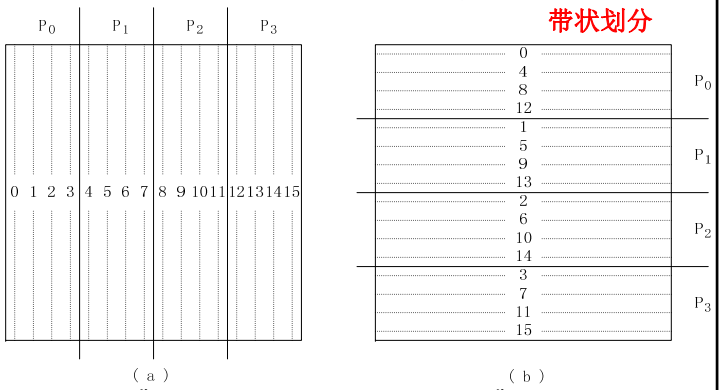
第九章 稠密矩阵运算

- 矩阵的划分
- 矩阵转置
- 矩阵-向量乘法
- 矩阵乘法

9.1 矩阵的划分

带状划分



- 列块带状划分

图9.1

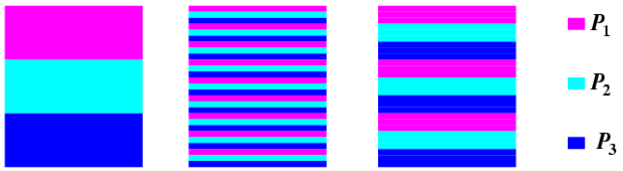
行循环带状划分

特点

- 规则的结构
- 适合于对计算进行静态划分
- 典型的划分基于数据
  - 输入输出和中间数据
- 典型的划分
  - 1D 和2D块状、 循环、块循环
- 多重划分

带状划分

- 示例：p＝3，27× 27矩阵的3种带状划分

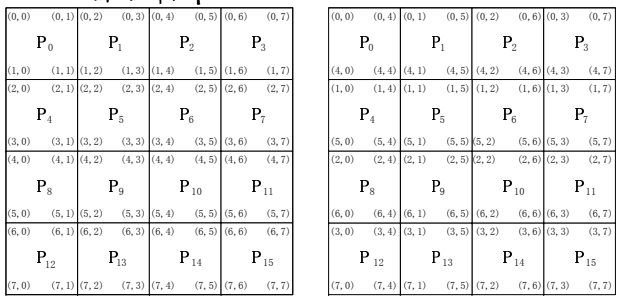


(a) block (b) cyclic (c) block-cyclic

Striped row-major mapping of a 27×27 matrix on $p = 3$ processors.

棋盘划分

- 8×8阶矩阵，p=16



（a） （b）

图9.2

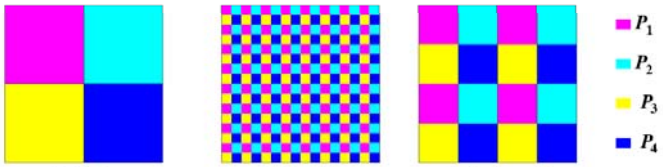
块棋盘划分 循环棋盘划分

9.2 矩阵转置

- 9.2.1 棋盘划分的矩阵转置
- 9.2.2 带状划分的矩阵转置

棋盘划分

- 示例：p＝4，16×16矩阵的3种棋盘划分

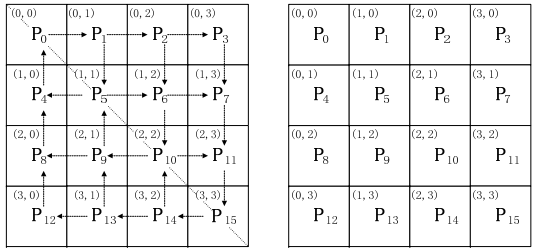


(a) block (b) cyclic (c) block cyclic

Checkerboard mapping of a 16 × 16 matrix on $p = 2 \times 2$ processors.

棋盘划分的矩阵转置

- 网孔连接
  - 情形1: $p=n^2$。



（a） （b）

通讯步 图9.3 转置后

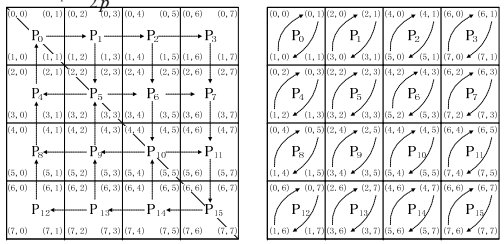
棋盘划分的矩阵转置

❖ 情形2: $p<n^2$。

- 划分: $A_{n\times n}$划分成p个大小为$\frac{n}{\sqrt{p}}\times\frac{n}{\sqrt{p}}$子块
- 算法: ①按mesh连接进行块转置(不同处理器间) $//\ 2\sqrt{p}(t_s+t_w n^2/p)\cdots$通讯

  ②进行块内转置(同一处理器内) $//\ \frac{n^2}{2p}\cdots\cdots$计算

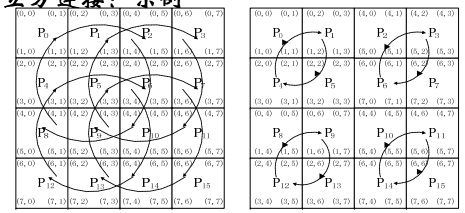
$$T_p=\frac{n^2}{2p}+2t_s\sqrt{p}+2t_w n^2/\sqrt{p}\cdots\cdots\text{运行时间}$$



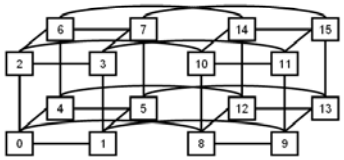
(a) 通讯步 图9.4 (b) 转置后

棋盘划分的矩阵转置

❖ 超立方连接：示例



(a) (b)



棋盘划分的矩阵转置

❖ 超立方连接

- 划分: $A_{n\times n}$划分成p个大小为$\frac{n}{\sqrt{p}}\times\frac{n}{\sqrt{p}}$子块
- 算法:

① 将$A=\begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}$转置为$\begin{pmatrix}A_{11} & A_{21}\\ A_{12} & A_{22}\end{pmatrix}$

②对A_{ij}递归应用①进行转置，直至分块矩阵的元素处于同一处理器；

③进行同一处理器的内部转置。

- 运行时间:

$$T_p=\frac{n^2}{2p}+2(t_s+t_w\frac{n^2}{p})\log\sqrt{p}\quad //\text{内部转置}\frac{n^2}{2p},\text{选路}:2(t_s+t_w\frac{n^2}{p}),\text{递归步}:\log\sqrt{p}$$

$$=\frac{n^2}{2p}+(t_s+t_w\frac{n^2}{p})\log p$$

9.2.2带状划分的矩阵转置

❖ 划分: $A_{n\times n}$分成p个(n/p)×n大小的带

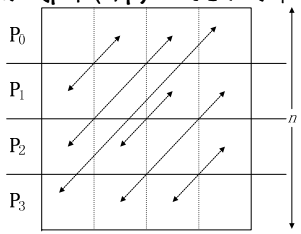


图9.7

❖ 算法:

①P_i有p-1个(n/p)×(n/p)大小子块发送到另外p-1个处理器中；

②每个处理器本地交换相应的元素

矩阵算法: 引言

- 由于结构规则，涉及到矩阵和向量的并行计算会没有困难地采用数据分解.
- 典型的算法依赖于对输入、输出和中间数据进行分解.
- 大部分算法采用一维或二维的块、循环和块循环划分.

矩阵向量乘: 按行 1-D 划分

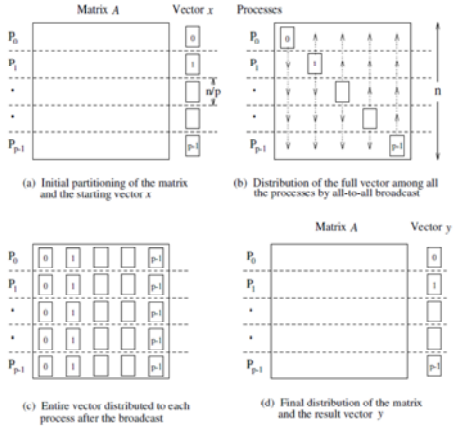
- n x n 矩阵在 n 个处理器间被划分, 每个存储器存储完整的矩阵行.
- n x 1 向量 x 按这样分布，每个进程拥有1个它的元素.

矩阵乘向量

- 首先我们将目标定在稠密的 n x n 矩阵 A 和 n x 1 向量 x 相乘产生 n x 1 结果向量 y.
- 串行算法需要 n^2 个乘法和加法.

$$W = n^2.$$

矩阵乘向量: 按行 1-D 划分



(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

n x n 矩阵与 n x 1 向量相乘，使用按行 1-D 块划分. 对于每个进程一个行的情形, $p = n$.

矩阵乘向量: 按行 1-D 划分

- 由于每个进程开始时只有x的一个元素, 所以需要多到多播送将所有元素分布到所有进程上.
- 进程 P_i 计算:

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

- 多到多播送和计算 $y[i]$ 二者都花费时间 $\Theta(n)$. 因此并行执行时间是 $\Theta(n)$.

矩阵乘向量: 按行 1-D 划分

可扩展性分析:

- 我们知道$T_0 = pT_P - W$, 因此, 我们有,

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

- 对于等效率函数（ isoefficiency）, 我们有 $W = KT_0$, 其中 $K = E/(1 - E)$ 对于期望的效率E.
- 据此, 我们有$W = O(p^2)$ (从t_w 项得来).
- 在 $p < n$ 情况下,等效率函数是有界的，$W = n^2 = \Omega(p^2)$.
- 总之，等效率函数是$W = O(p^2)$.

矩阵乘向量: 按行 1-D 划分

- 现考虑 $p < n$ 时情形，仍然采用块状 1D 划分.
- 每个进程初始时存储 n/p 个完整的矩阵行，同时存储大小为n/p的向量的块.
- 多到多播送在 p 个进程间进行，涉及大小n/p的消息.
- 接下来 n/p 局部点积.
- 因此, 这个过程的并行执行时间是

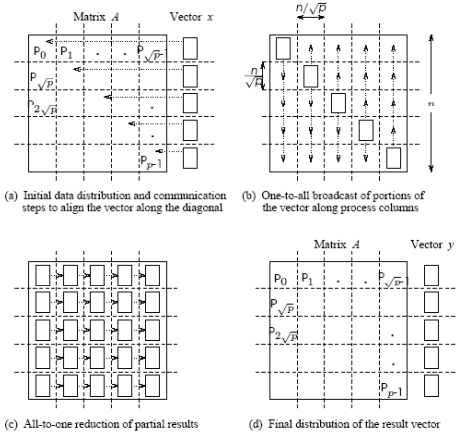
$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

这是成本最优的（cost-optimal）.

矩阵乘向量: 2-D 划分

- n x n在n^2 个进程间划分，结果每个处理器只有一个元素.
- n x 1 向量 x 只分布在n个处理器的最后一列.

矩阵乘向量：2-D 划分



(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

矩阵乘向量用块 2-D 划分. 对于每个进程一个元素的情形, $p = n^2$ 若矩阵大小为 $n \times n$.

矩阵乘向量：2-D 划分

- 该算法中使用了三个基本通信操作: 点到点通信将向量对齐到主对角线； 在每一列的n个进程内实施一到多广播每个向量元素； 以及沿每个列实施多到一单点收集.
- 这三个通信操作花费的时间都是 $\Theta(\log n)$ 并且算法的并行执行时间是 $\Theta(\log n)$.
- 成本 (进程与时间乘积) 是 $\Theta(n^2 \log n)$; 因此, 该算法不是成本最优的.

矩阵乘向量：2-D 划分

- 首先将向量与矩阵恰当对齐.
- 对于2-D划分，在第一个通信步将向量x沿着矩阵主对角线对齐.
- 第二步从每个对角线进程中复制向量元素到相应列的所有进程中，使用 n 个有同时性的列内广播完成.
- 最后, 通过沿着列的多到一单点收集计算出结果向量.

矩阵乘向量：2-D 划分

- 少于 n^2 个处理器时, 每个进程有(1)个矩阵分块.
- 向量按照块大小(2)分布在最后的进程列中.
- 这种情况下, 对齐、广播、收集的消息大小都是(2) .
- 计算的是(1)子矩阵和长度为(2)的向量的乘积.

(1) $(n/\sqrt{p}) \times (n/\sqrt{p})$

(2) $n/\sqrt{p}$

矩阵乘向量：2-D 划分

- 第一步对齐花费的时间

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

- 广播和收集花费的时间

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

- 局部矩阵向量点积时间

$$t_c n^2/p$$

- 总时间

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

9.4 矩阵乘法

- 简单并行分块乘法
- Cannon乘法
- Fox乘法
- Systolic乘法
- DNS乘法

矩阵乘向量：2-D 划分可扩展性分析

$$T_o = pT_p - W = t_s p \log p + t_w n\sqrt{p} \log p$$

- 为了找出等效率函数，让 T_0 与 W相等, 选占主导的项得（1）. 其中W=n²
- 应归于并发性的等效率函数为 $O(p)$.
- 整个等效率函数为（2）
- (归于为了带宽).
- 为了成本最优, 有（3）.
- 为此有（4）

（1） $W = K^2 t_w^2 p \log^2 p$

（2） $O(p \log^2 p)$

（3） $W = n^2 = p \log^2 p$

（4） $p = O\left(\frac{n^2}{\log^2 n}\right)$

矩阵和矩阵相乘

- 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 $O(n^3)$.
- 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 $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ $(0 \le i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix.
- In this view, we perform q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices.

矩阵和矩阵相乘

- Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ 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.
- Perform local submatrix multiplication.

Matrix-Matrix Multiplication

- **Two broadcasts take time** $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$
- **Computation requires** $\sqrt{p}$ **multiplications of** $(n/\sqrt{p}) \times (n/\sqrt{p})$ **submatrices**
- **Parallel run time is approximately**

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

- **Algorithm is cost optimal**
- **Isoefficiency is** $O(p^{3/2})$
  — **due to bandwidth term** t_w **and concurrency** ($p \le n^2$ **thus** $n^3 \ge p^{3/2}$)
- **Major drawback of the algorithm: not memory optimal**

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

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}. \quad (5)$$

- 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.

时间复杂度t_1+t_2

❖ 超立方连接

$$t_1 = 2(t_s \log\sqrt{p} + t_w \frac{n^2}{p}(\sqrt{p}-1))$$

$$t_2 = \sqrt{p} \times (\tfrac{n}{\sqrt{p}})^3 = n^3 / p$$

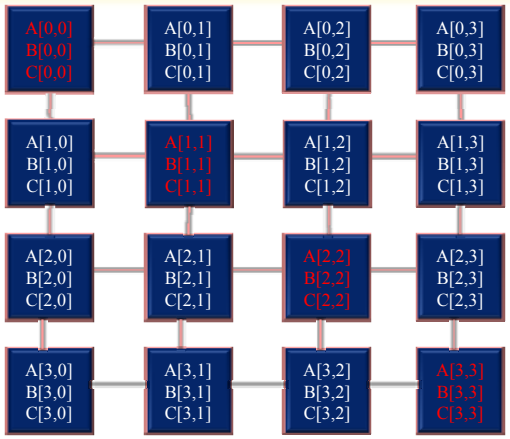
❖ 2D-Mesh连接

$$t_1 = 2(t_s + \frac{n^2}{p} t_w)(\sqrt{p}-1) = 2t_s\sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}$$

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.

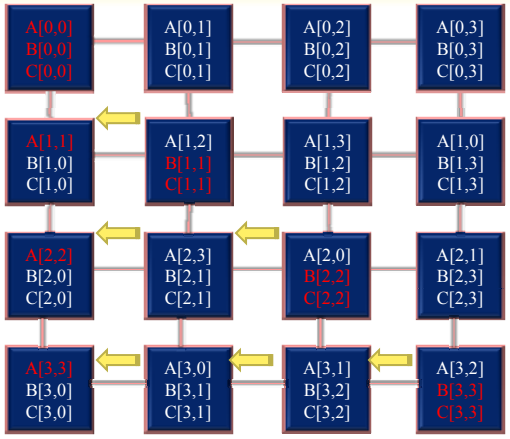
Cannon乘法：初始化



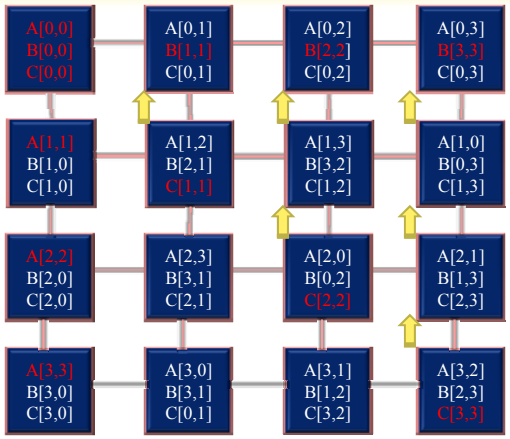
Cannon矩阵乘法

- 适用于网格
- 存储有效
- 规则通信
  - 初始化
  - 对齐
  - 循环移位

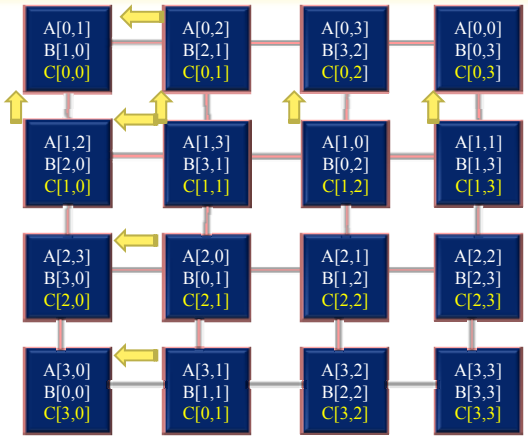
Cannon乘法：对齐A



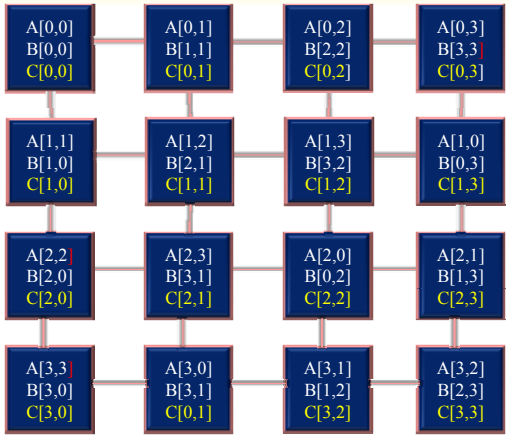
Cannon乘法：对齐B



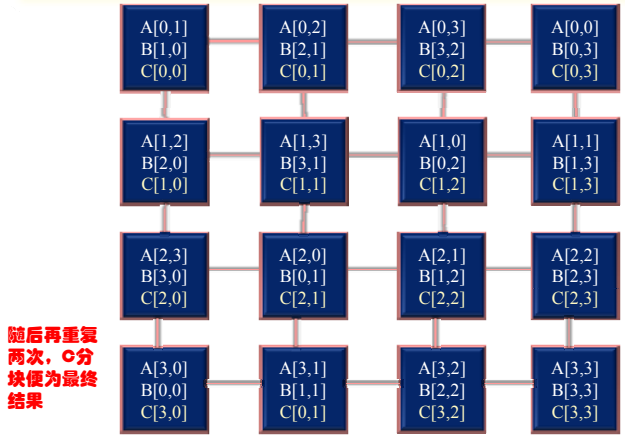
A和B各自循环移位



每个结点并行乘分块得到C分块



各自乘分块并累加到C分块



随后再重复两次，C分块便为最终结果

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.

Cannon乘法

❖ 算法描述: Cannon分块乘法算法

//输入: $A_{n\times n}$, $B_{n\times n}$; 输出: $C_{n\times n}$

Begin

(1)for k=0 to $\sqrt{p}-1$ do

for all $P_{i,j}$ par-do

(i) if i>k then

$A_{i,j} \leftarrow A_{i,(j+1)\bmod \sqrt{p}}$

endif

(ii)if j>k then

$B_{i,j} \leftarrow B_{(i+1)\bmod \sqrt{p},j}$

endif

endfor

endfor

(2)for all $P_{i,j}$ par-do $C_{i,j}$=0 endfor

(3)for k=0 to $\sqrt{p}-1$ do

for all $P_{i,j}$ par-do

(i) $C_{i,j}=C_{i,j}+A_{i,j}B_{i,j}$

(ii) $A_{i,j} \leftarrow A_{i,(j+1)\bmod \sqrt{p}}$

(iii)$B_{i,j} \leftarrow B_{(i+1)\bmod \sqrt{p},j}$

endfor

endfor

End

时间分析:

$$T_P(n) = T_1 + T_2 + T_3$$
$$= O(\sqrt{p}) + O(1) + O(\sqrt{p}\cdot(n/\sqrt{p})^3)$$
$$= O(n^3/p)$$

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:

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}. \quad (6)$$

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

9.4.3 Fox乘法

❖ 分块: 同Cannon分块算法

❖ 算法原理

①$A_{i,i}$向所在行的其他处理器进行一到多播送;

②各处理器将收到的A块与原有的B块进行乘-加运算;

③B块向上循环移动一步;

④如果$A_{i,j}$是上次第i行播送的块，本次选择$A_{i,(j+1)\bmod\sqrt{p}}$向所在行的其他处理器进行一到多播送;

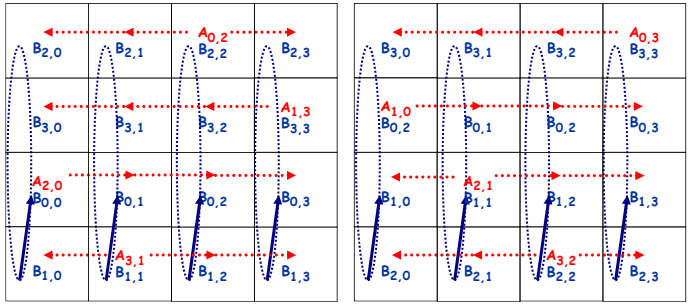
⑤转②执行$\sqrt{p}-1$次;

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{0,1}$	$A_{0,2}$ $B_{0,2}$	$A_{0,3}$ $B_{0,3}$
$A_{1,0}$ $B_{1,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{1,2}$	$A_{1,3}$ $B_{1,3}$
$A_{2,0}$ $B_{2,0}$	$A_{2,1}$ $B_{2,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{2,3}$
$A_{3,0}$ $B_{3,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{3,2}$	$A_{3,3}$ $B_{3,3}$

- Fox (and Cannon) treatments make the following assumptions:
  - The number of processes (p) is a perfect square
  - The matrices to be multiplied are square of order n x n
  - sqrt(p) divides n evenly

Fox乘法

- 示例： $A_{4\times4}$， $B_{4\times4}$，p=16



(a) (b)

Fox乘法

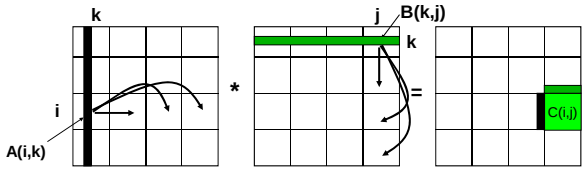
- 示例： $A_{4\times4}$， $B_{4\times4}$，p=16

(c) (d)

Pros and Cons of Cannon

- Local computation one call to (optimized) matrix-multiply
- Hard to generalize for
  - **p not a perfect square**
  - **A and B not square**
  - **Dimensions of A, B not perfectly divisible by s=sqrt(p)**
  - **A and B not "aligned" in the way they are stored on processors**
  - **block-cyclic layouts**
- Memory hog (extra copies of local matrices)

SUMMA

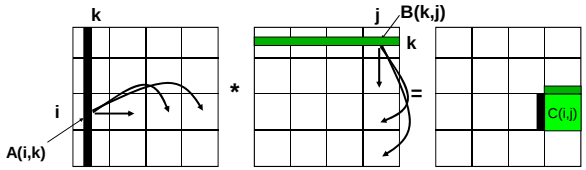


- i, j represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns
- $C(i,j) = C(i,j) + \Sigma_k A(i,k)*B(k,j)$
- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
  - Need not be square

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient, but simpler and easier to generalize
- Presentation from van de Geijn and Watts
  - **www.netlib.org/lapack/lawns/lawn96.ps**
  - **Similar ideas appeared many times**
- Used in practice in PBLAS = Parallel BLAS
  - **Basic Linear Algebra Subprograms**
  - **www.netlib.org/lapack/lawns/lawn100.ps**

SUMMA




```
For k=0 to n-1     … or n/b-1 where b is the block size
                   …  = # cols in A(i,k) and # rows in B(k,j)
  for all i = 1 to pr   … in parallel
      owner of A(i,k) broadcasts it to whole processor row
  for all j = 1 to pc   … in parallel
      owner of B(k,j) broadcasts it to whole processor column
  Receive A(i,k) into Acol
  Receive B(k,j) into Brow
  C_myproc  = C_myproc  + Acol * Brow
```

SUMMA performance

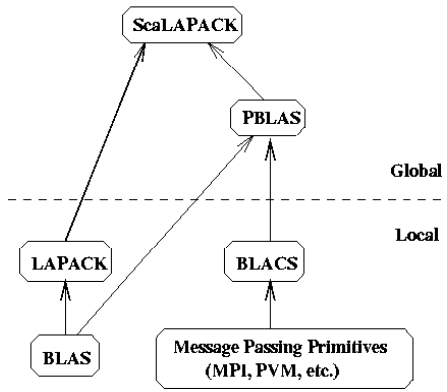
- To simplify analysis only, assume s = sqrt(p)

```
For k=0 to n/b-1
   for all i = 1 to s   …  s = sqrt(p)
      owner of A(i,k) broadcasts it to whole processor row
         … time = log s *( α + β * b*n/s), using a tree
   for all j = 1 to  s
      owner of B(k,j) broadcasts it to whole processor column
         … time = log s *( α + β * b*n/s), using a tree
   Receive A(i,k) into Acol
   Receive B(k,j) into Brow
   C_myproc = C_myproc + Acol * Brow
         … time = 2*(n/s)^2*b
```

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$

ScaLAPACK Parallel Library

ScaLAPACK SOFTWARE HIERARCHY



SUMMA performance

- Total time = $2*n^3/p + \alpha * \log p * n/b + \beta * \log p * n^2/s$
- Parallel Efficiency =
  $1/(1 + \alpha * \log p * p / (2*b*n^2) + \beta * \log p * s/(2*n))$
- ~Same β term as Cannon, except for log p factor
  log p grows slowly so this is ok
- Latency (α) term can be larger, depending on b
  When b=1, get $\alpha * \log p * n$
  As b grows to n/s, term shrinks to
  $\alpha * \log p * s$ (log p times Cannon)
- Temporary storage grows like 2*b*n/s
- Can change b to tradeoff latency cost with memory

Performance of PBLAS

PDGEMM = PBLAS routine for matrix multiply

Observations:
For fixed N, as P increases Mflops increases, but less than 100% efficiency
For fixed P, as N increases, Mflops (efficiency) rises

Speed in Mflops of PDGEMM					
Machine	Procs	Block Size	N = 2000	N = 4000	N = 10000
Cray T3E	4=2x2	32	1055	1070	0
	16=4x4		3630	4005	4292
	64=8x8		13456	14287	16755
IBM SP2	4	50	755	0	0
	16		2514	2850	0
	64		6205	8709	10774
Intel XP/S MP Paragon	4	32	330	0	0
	16		1233	1281	0
	64		4496	4864	5257
Berkeley NOW	4	32	463	470	0
	32=4x8		2490	2822	3450
	64		4130	5457	6647

DGEMM = BLAS routine for matrix multiply

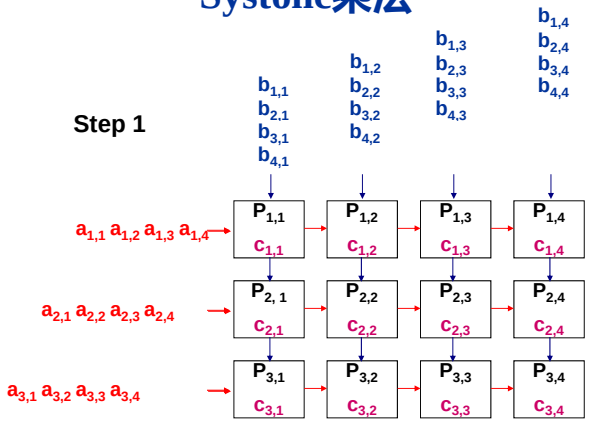
Maximum speed for PDGEMM = # Procs * speed of DGEMM

Observations (same as above):
Efficiency always at least 48%
For fixed N, as P increases, efficiency drops
For fixed P, as N increases, efficiency increases

Efficiency = MFlops(PDGEMM)/(Procs*MFlops(DGEMM))						
Machine	Peak/ proc	DGEMM Mflops	Procs	N = 2000	N = 4000	N = 10000
Cray T3E	600	360	4	.73	.74	
			16	.63	.70	.75
			64	.58	.62	.73
IBM SP2	266	200	4	.94		
			16	.79	.89	
			64	.48	.68	.84
Intel XP/S MP Paragon	100	90	4	.92		
			16	.86	.89	
			64	.78	.84	.91
Berkeley NOW	334	129	4	.90	.91	
			32	.60	.68	.84
			64	.50	.66	.81

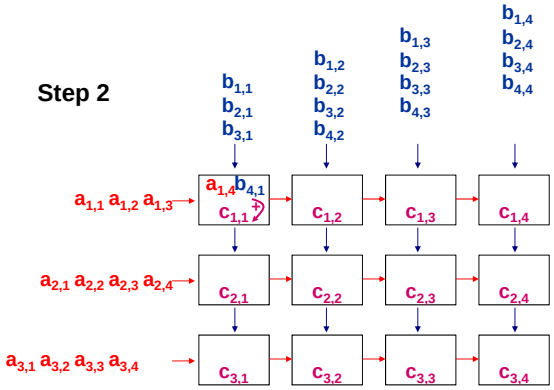
Systolic乘法

Step 1



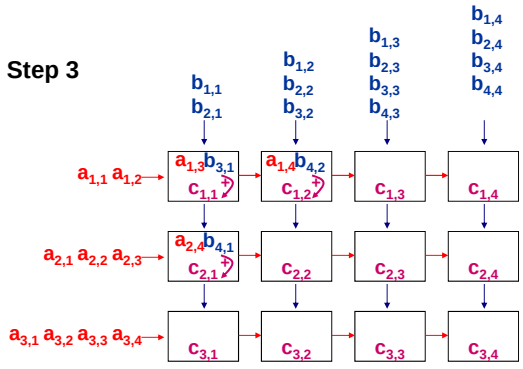
Systolic乘法

Step 2



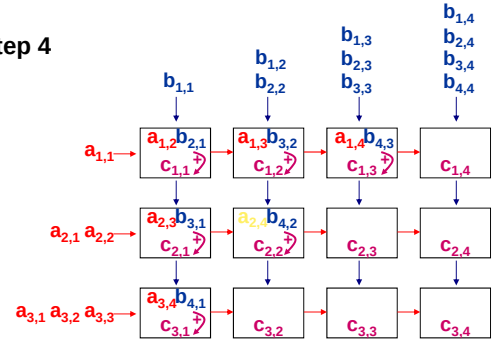
Systolic乘法

Step 3



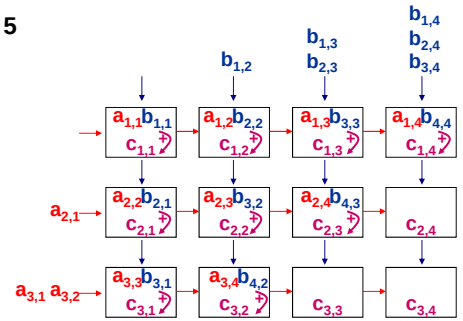
Systolic乘法

Step 4



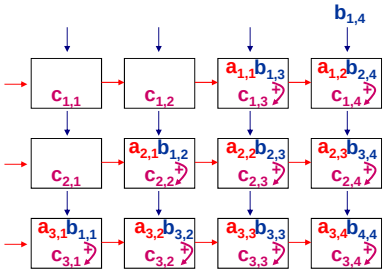
Systolic乘法

Step 5



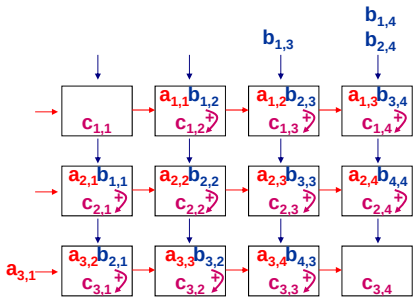
Systolic乘法

Step 7



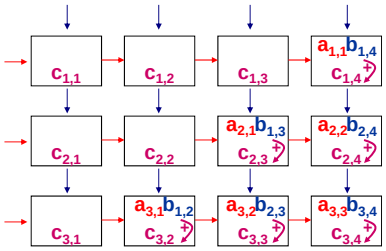
Systolic乘法

Step 6



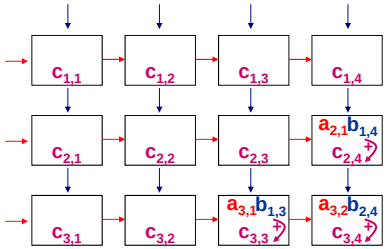
Systolic乘法

Step 8



Systolic乘法

Step 9



Systolic乘法

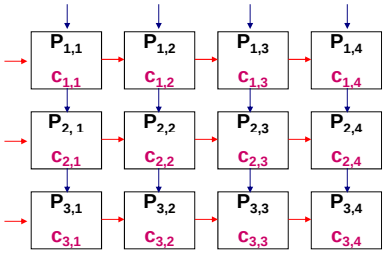
Over

$c_{1,1} = a_{1,1}\, b_{1,1} + a_{1,2}\, b_{2,1} + a_{1,3}\, b_{3,1} + a_{1,4}\, b_{4,1}$

$c_{1,2} = a_{1,1}\, b_{1,2} + a_{1,2}\, b_{2,2} + a_{1,3}\, b_{3,2} + a_{1,4}\, b_{4,2}$

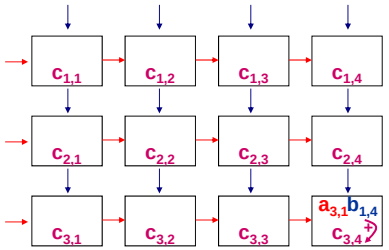
……

$c_{3,4} = a_{3,1}\, b_{1,4} + a_{3,2}\, b_{2,4} + a_{3,3}\, b_{3,4} + a_{3,4}\, b_{4,4}$



Systolic乘法

Step 10



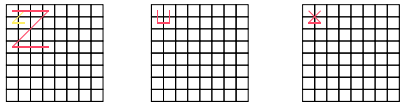
Systolic乘法

- Systolic算法

```
//输入: A_{m×n}, B_{n×k};   输出: C_{m×k}
 Begin
    for i=1 to m   par- do
          for j=1 to k  par-do
            (i) c_{i,j} = 0
          (ii) while P_{i,j} 收到a和b时 do
                c_{i,j} = c_{i,j} +ab
                if i < m then 发送b给P_{i+1,j} endif
                if j < k  then 发送a给P_{i,j+1} endif
              endwhile
          endfor
      endfor
 End
```

Recursive Layouts

- For both cache hierarchies and parallelism, recursive layouts may be useful
- Z-Morton, U-Morton, and X-Morton Layout



- Also Hilbert layout and others
- What about the user's view?
  - **Fortunately, many problems can be solved on a permutation**
  - **Never need to actually change the user's layout**

9.4 矩阵乘法

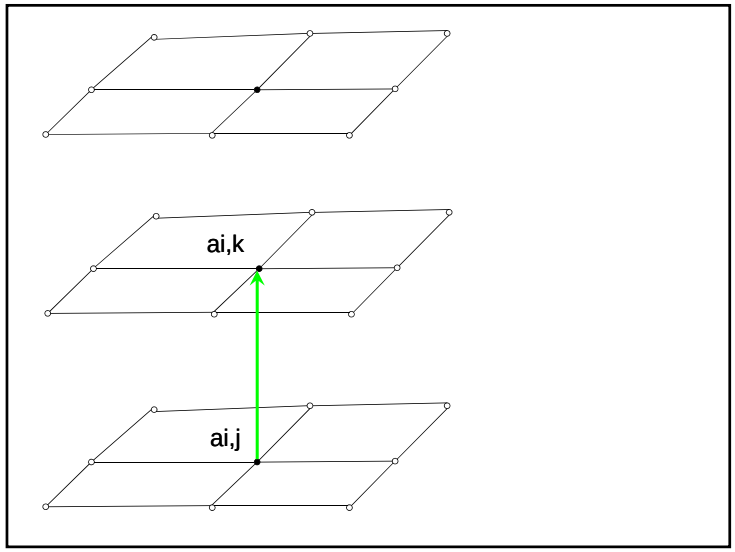
9.4.1 简单并行分块乘法
9.4.2 Cannon乘法
9.4.3 Fox乘法
9.4.4 Systolic乘法
9.4.5 DNS乘法

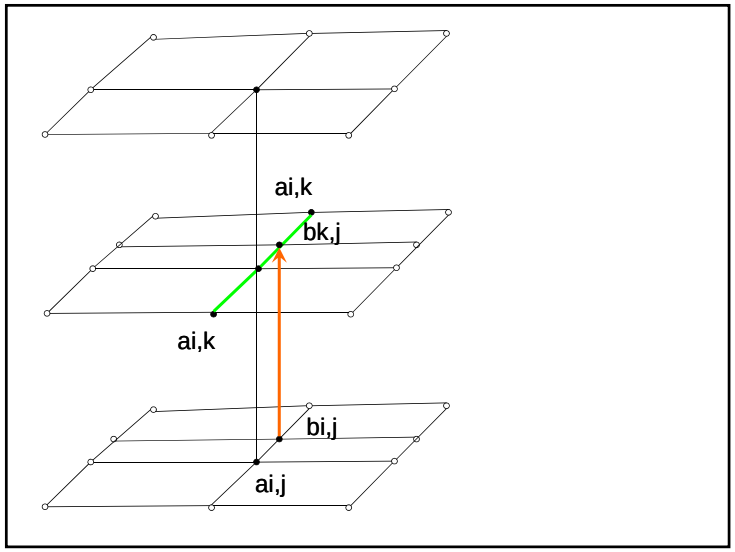
Summary of Parallel Matrix Multiplication

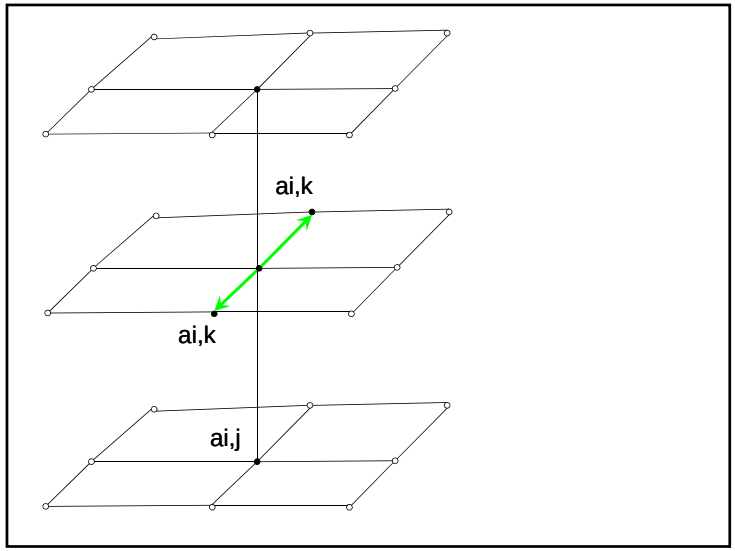
- 1D Layout
  - Bus without broadcast – slower than serial
  - Nearest neighbor communication on a ring (or bus with broadcast): Efficiency = 1/(1 + O(p/n))
- 2D Layout
  - **Cannon**
    - Efficiency = 1/(1+ O(a * (sqrt(p) /n)^3 + b* sqrt(p) /n))
    - Hard to generalize for general p, n, block cyclic, alignment
  - **SUMMA**
    - Efficiency = 1/(1 + O(a * log p * p / (b*n^2) + b*log p * sqrt(p) /n))
    - Very General
    - b small => less memory, lower efficiency
    - b large => more memory, high efficiency
  - **Recursive layouts**

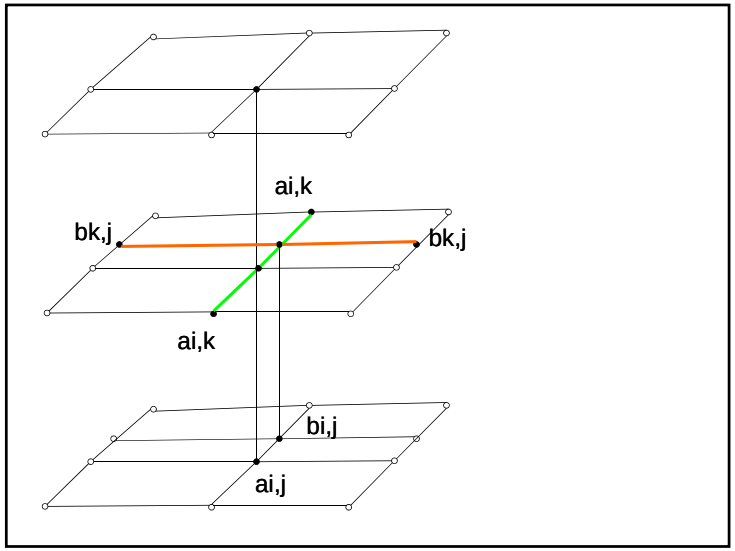
DNS乘法

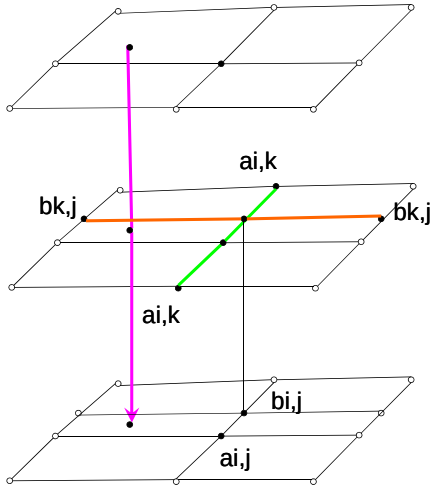
- 背景：由Dekel、Nassimi和Sahni提出的SIMD-CC上的矩阵乘法，处理器数目为n^3，运行时间为O(logn)，是一种速度很快的算法。
- 基本思想：通过一到一和一到多的播送办法，使得处理器(k,i,j)拥有$a_{i,k}$和$b_{k,j}$，进行本地相乘，再沿k方向进行单点积累求和，结果存储在处理器(0,i,j)中。
- 处理器编号：处理器数$p=n^3=(2^q)^3=2^{3q}$，处理器P_r位于位置(k,i,j)，这里$r=kn^2+in+j$，($0\leqslant i, j, k\leqslant n-1$)。位于(k,i,j)的处理器$P_r$的三个寄存器$A_r$,$B_r$,$C_r$分别表示为A[k,i,j]，B[k,i,j]和C[k,i,j]，初始时均为0。
- 算法：初始时$a_{i,j}$和$b_{i,j}$存储于寄存器A[0,i,j]和B[0,i,j]；
  ①数据复制：A,B同时在k维复制(一到一播送)；
  A在j维复制(一到多播送)；B在i维复制(一到多播送)；
  ②相乘运算：所有处理器的A、B寄存器两两相乘；
  ③求和运算：沿k方向进行单点积累求和；


ai,k
ai,j


ai,k
bk,j
ai,k
bi,j
ai,j

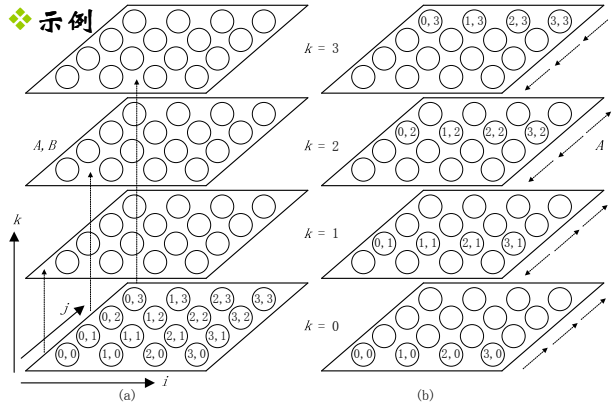

ai,k
ai,k
ai,j


ai,k
bk,j
bk,j
ai,k
bi,j
ai,j



DNS乘法

❖ 示例

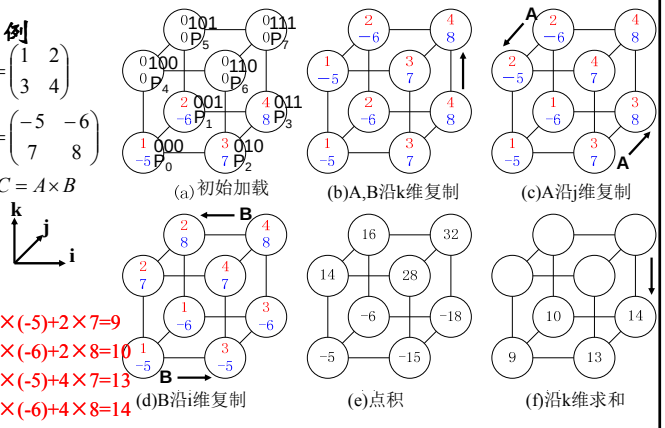


❖ 示例

$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$

$B=\begin{pmatrix}-5 & -6\\ 7 & 8\end{pmatrix}$

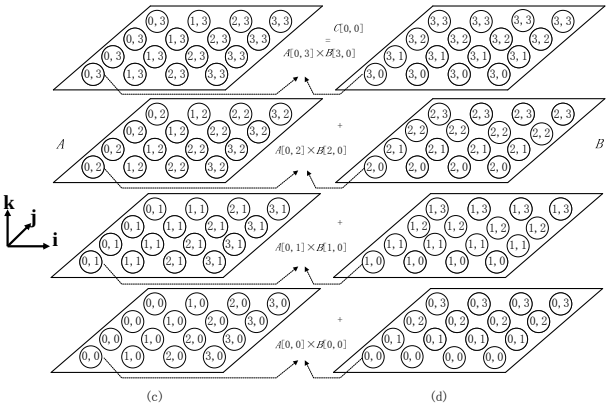
求$C=A\times B$



C00=1×(-5)+2×7=9
C01=1×(-6)+2×8=10
C10=3×(-5)+4×7=13
C11=3×(-6)+4×8=14

图9. 12

DNS乘法



DNS乘法:算法描述

```
//令r^(m)表示r的第m位取反;
//{p, r_m=d}表示r(0≤r≤p-1)的集合,
//这里r的二进制第m位为d;
//输入: A_n×n, B_n×n;    输出: C_n×n
begin  //以n=2, p=8=2^3举例, q=1, r=(r_2r_1r_0)_2
 (1)for m=3q-1 to 2q do  //按k维复制A,B, m=2
      for all r in {p, r_m=0} par-do //r_2=0的r
        (1.1) A_r(m) ← A_r    //A(100)←A(000)等
        (1.2) B_r(m) ← B_r    //B(100)←B(000)等
      endfor
    endfor
 (2)for m=q-1 to 0 do      //按j维复制A, m=0
      for all r in {p, r_m= r_2q+m} par-do //r_0=r_2的r
        A_r(m) ← A_r  //A(001)←A(000),A(100)←A(101)
      endfor          //A(011)←A(010),A(110)←A(111)
    endfor
 (3)for m=2q-1 to q do  //按i维复制B,m=1
      for all r in {p, r_m= r_q+m} par-do
                                   //r_1=r_2的r
        B_r(m)← B_r  //B(010)←B(000),B(100)←B(110)
      endfor         //B(011)←B(001),B(101)←B(111)
    endfor
 (4)for r=0 to p-1 par-do     //相乘, all P_r
        C_r=A_r×B_r
    endfor
 (5)for m=2q to 3q-1 do     //求和,m=2
      for r=0 to p-1 par-do
        C_r=C_r+C_r(m)
      endfor
    endfor
end
```