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

棋盘划分

- 8×8阶矩阵，p=16

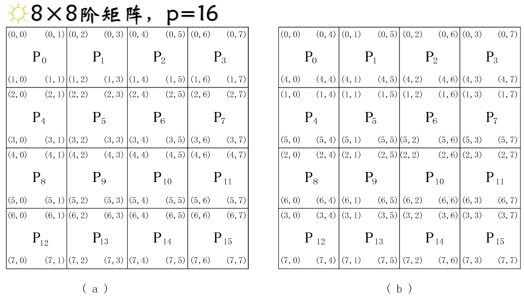
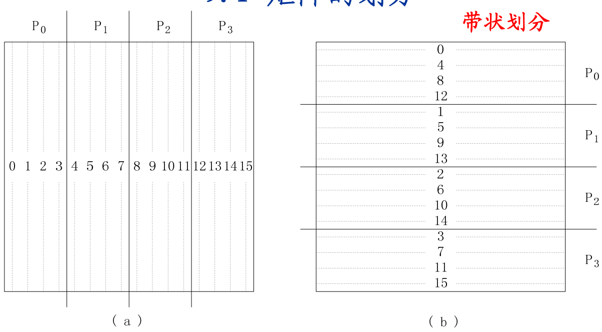


图9.2

块棋盘划分　　循环棋盘划分

9.1 矩阵的划分

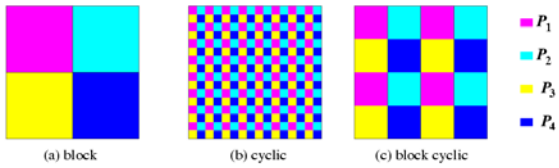
带状划分



- 列块带状划分　　图9.1　　行循环带状划分

棋盘划分

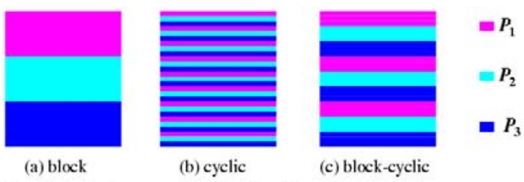
- 示例：p＝4，16×16矩阵的3种棋盘划分



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

带状划分

- 示例：p＝3，27× 27矩阵的3种带状划分



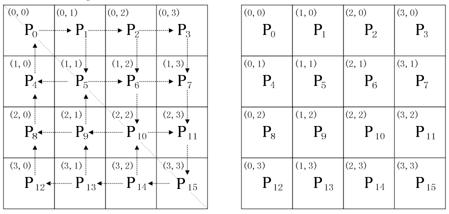
Striped row-major mapping of a 27 × 27 matrix on $p = 3$ processors.

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: $p=n^2$。

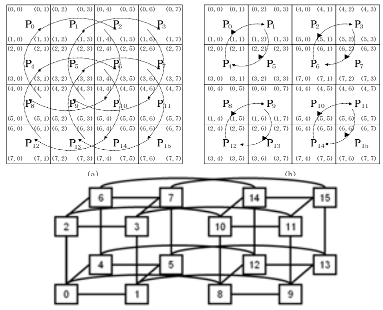


(a) (b)

通讯步 图9.3 转置后

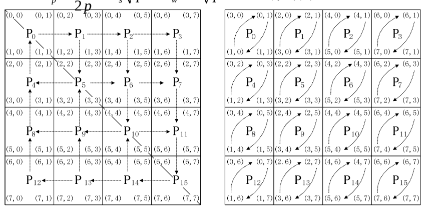
棋盘划分的矩阵转置

- 超立方连接：示例



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) // $2\sqrt{p}(t_s+t_w n^2/p)$⋯通讯
    ②进行块内转置(同一处理器内) // $\frac{n^2}{2p}$……计算
    $T_p=\frac{n^2}{2p}+2t_s\sqrt{p}+2t_w n^2/\sqrt{p}$……运行时间



(a) (b)

通讯步 图9.4 转置后

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

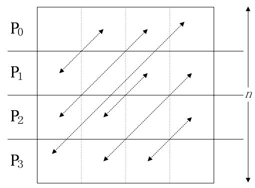


图9.7

- 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

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:

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

9.3 矩阵-向量乘法

- 9.3.1 带状划分的矩阵-向量乘法
  - y = A*x
- 9.3.2 棋盘划分的矩阵-向量乘法

带状划分的矩阵-向量乘法

- 划分(行带状划分): P_i存放x_i和$a_{i,0},a_{i,1},\cdots,a_{i,n-1}$，并输出$y_i$
- 算法: 对p=n情形
  - ①每个P_i向其他处理器播送x_i(多到多播送);
  - ②每个P_i计算;
- 注: 对p<n情形,算法中P_i要播送X中相应的n/p个分量
  - (1)超立方连接的计算时间

$$T_p=\frac{n^2}{p}+t_s\log p+\frac{n}{p}t_w(p-1) \quad \text{// 前1项是乘法时间，后2项是多到多的播送时间}$$

$$=\frac{n^2}{p}+t_s\log p+nt_w \quad \text{// } p\text{充分大时}$$

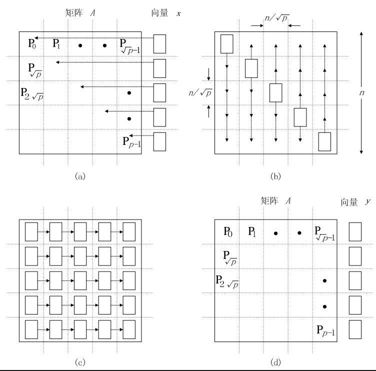
  - (2)网孔连接的计算时间

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

$$=\frac{n^2}{p}+2t_s(\sqrt{p}-1)+nt_w \quad \text{// } p\text{充分大时}$$

棋盘划分的矩阵-向量乘法

- 示例



带状划分的矩阵-向量乘法

- 示例

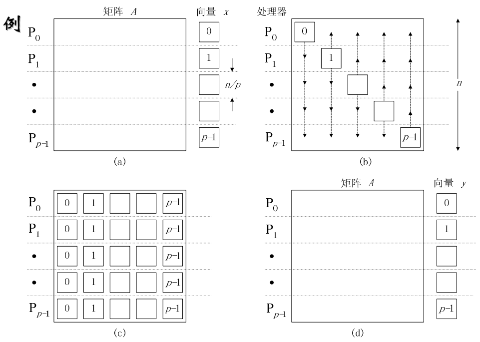


图9. 8

带状与棋盘划分比较

- 以网孔链接为例
  - 网孔上带状划分的运行时间

$$T_p=\frac{n^2}{p}+2t_s(\sqrt{p}-1)+nt_w \qquad (9.5)$$

  - 网孔上棋盘划分的运行时间

$$T_p\approx\frac{n^2}{p}+t_s\log p+\frac{n}{\sqrt{p}}t_w\log p+3t_h\sqrt{p} \qquad (9.6)$$

- 棋盘划分要比带状划分快。

9.3.2棋盘划分的矩阵-向量乘法

- 划分(块棋盘划分): P_{ij}存放$a_{i,j}$, x_i置入$P_{i,i}$中
- 算法: 对$p=n^2$情形
  - ①每个$P_{i,i}$向$P_{j,i}$播送x_i(一到多播送);
  - ②按行方向进行乘-加与积累运算，最后一列$P_{i,n-1}$收集的结果为y_i;
- 注: 对$p<n^2$情形,p个处理器排成$\sqrt{p}\times\sqrt{p}$的二维网孔，
  算法中$P_{i,i}$向$P_{j,i}$播送X中相应的$n/\sqrt{p}$个分量
  - (1)网孔连接的计算时间T_p(CT): $t_s+\frac{n}{\sqrt{p}}t_w+t_h\sqrt{p}$
  - .X中相应分量置入$P_{i,i}$的通讯时间: $(t_s+\frac{n}{\sqrt{p}}t_w)\log\sqrt{p}+t_h(\sqrt{p}-1)$
  - .按列一到多播送时间: $(t_s+\frac{n}{\sqrt{p}}t_w)\log\sqrt{p}+t_h(\sqrt{p}-1)$
  - .按行单点积累的时间: $\therefore\ T_p\approx\frac{n^2}{p}+t_s\log p+\frac{n}{\sqrt{p}}t_w\log p+3t_h\sqrt{p}$

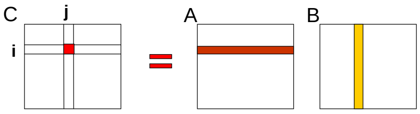
9.4 矩阵乘法

- 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

矩阵乘法符号及定义

设$A=(a_{ij})_{n\times n}$　$B=(b_{ij})_{n\times n}$　$C=(c_{ij})_{n\times n}$，$C=A\times B$

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & & c_{1,n-1} \\ \vdots & \vdots & & \vdots \\ c_{n-1,0} & c_{n-1,1} & \cdots & c_{n-1,n-1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & & b_{1,n-1} \\ \vdots & \vdots & & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{pmatrix}$$



$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

A中元素的第1下标与B中元素的第2下标相一致（对准）

简单并行分块乘法

☼运行时间

(1)

(2)二维环绕网孔连接：

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

②的时间$t_2=n^3/p$

$$\therefore\ T_p = \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}$$

☼注

(1)本算法的缺点是对处理器的存储要求过大

每个处理器有 $2\sqrt{p}$ 个块，每块大小为n^2/p,

所以需要 $O(n^2/\sqrt{p})$，p个处理器共需要$O(n^2\sqrt{p})$ ，

是串行算法的 $\sqrt{p}$ 倍

(2)$p(n)=p$，$t(n)=O(n^3/p)$, $c(n)=O(n^3)$

矩阵乘法并行实现方法

☼计算结构：二维阵列

☼空间对准(元素已加载到阵列中)

Cannon's， Fox's，DNS

☼时间对准(元素未加载到阵列中)

Systolic

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

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**

简单并行分块乘法

☼分块：A、B和C分成$p=\sqrt{p}\times\sqrt{p}$ 的方块阵$A_{i,j}$、$B_{i,j}$和$C_{i,j}$，大小均为 $\frac{n}{\sqrt{p}}\times\frac{n}{\sqrt{p}}$ p个处理器编号为$(P_{0,0},\ldots,P_{0,\sqrt{p}-1},\ldots,P_{\sqrt{p}-1,\sqrt{p}-1})$，$P_{i,j}$存放$A_{i,j}$、$B_{i,j}$和$C_{i,j}$。

☼算法：

①通信：每行处理器进行A矩阵块的多到多播送(得到$A_{i,k}$, $k=0\sim\sqrt{p}-1$)

每列处理器进行B矩阵块的多到多播送(得到$B_{k,j}$, $k=0\sim\sqrt{p}-1$)

②乘-加运算：$P_{i,j}$做 $C_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{ik}\cdot B_{kj}$

☼运行时间

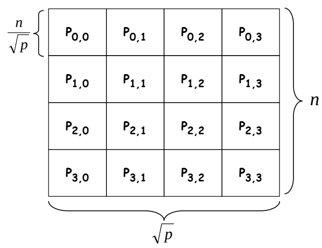
(1)超立方连接：

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

②的时间　$t_2 = \sqrt{p}\times(\frac{n}{\sqrt{p}})^3 = n^3/p$

9.4.2 Cannon乘法

☼分块：A、B和C分成$p=\sqrt{p}\times\sqrt{p}$的方块阵$A_{i,j}$、$B_{i,j}$和$C_{i,j}$，大小 $\frac{n}{\sqrt{p}}\times\frac{n}{\sqrt{p}}$ 均为p个处理器编号为$(P_{0,0},\ldots,P_{0,\sqrt{p}-1},\ldots,P_{\sqrt{p}-1,\sqrt{p}-1})$，$P_{i,j}$存放$A_{i,j}$、$B_{i,j}$和$C_{i,j}$(n > > p)



Cannon乘法

☼算法原理（非形式描述）

①所有块$A_{i,j}(0\leqslant i,j\leqslant\sqrt{p}-1)$向左循环移动i步(按行移位)；

所有块$B_{i,j}(0\leqslant i,j\leqslant\sqrt{p}-1)$向上循环移动j步(按列移位)；

②所有处理器$P_{i,j}$做执行$A_{i,j}$和$B_{i,j}$的乘-加运算；

③A的每个块向左循环移动一步；

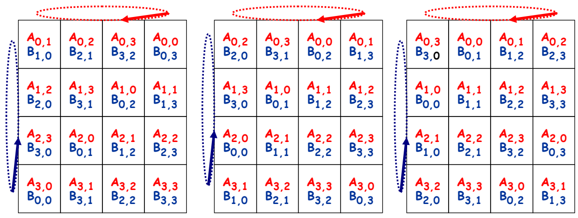
B的每个块向上循环移动一步；

④转②执行$\sqrt{p}-1$次；

Cannon乘法

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

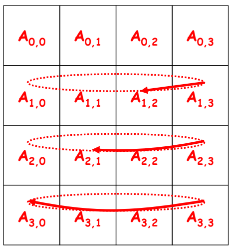
After first shift　After second shift　After third shift

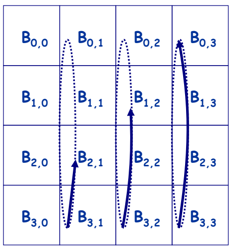


Cannon乘法

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

Initial alignment of A　Initial alignment of B





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

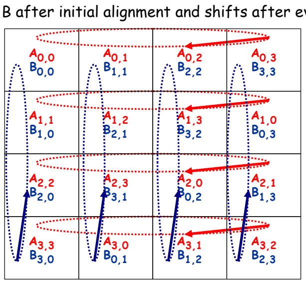
时间分析：

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

Cannon乘法

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

A and B after initial alignment and shifts after every step



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；

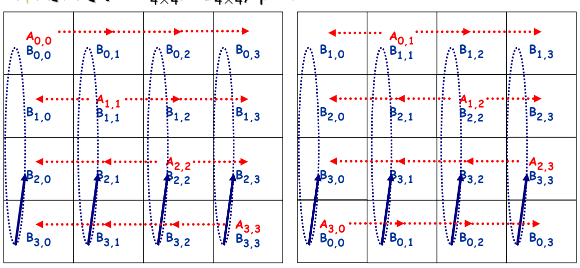
$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{0,1}$	$A_{0,2}$ $B_{0,2}$	$A_{0,3}$ $B_{0,3}$
$A_{1,0}$ $B_{1,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{1,2}$	$A_{1,3}$ $B_{1,3}$
$A_{2,0}$ $B_{2,0}$	$A_{2,1}$ $B_{2,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{2,3}$
$A_{3,0}$ $B_{3,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{3,2}$	$A_{3,3}$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

- 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

- q = sqrt(p) // number of rows, cols in processor grid
- // A is operand 1, B is operand 2 in A * B
- // C is result
- // i,j = process row, column
- // src, dest rows for rotating 'up'
- src = i+1 mod q;
- dest = i-1 mod q;
- for (stage = 0; stage < q; stage++) {
  - k_bar = (i+stage) mod q;
  - broadcast(A[i,k_bar]) to row i;
  - C[i,j] = C[i,j] + A[i,k_bar]*B[k_bar,j]
  - sendrecv(B[k_bar,j],src,dest);
- }

Fox乘法

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



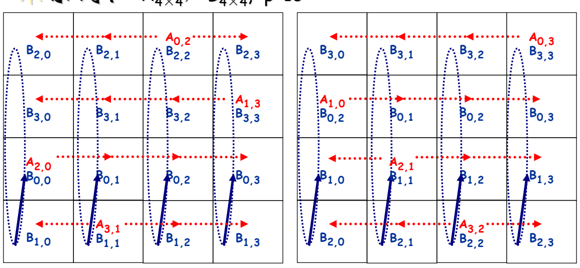
(a) (b)

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

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 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**

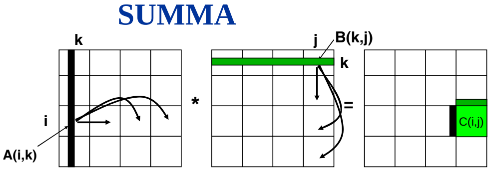
SUMMA performance

- **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 = $2*n^3/p$ + α * log p * n/b + β * log p * n^2 /s**

SUMMA

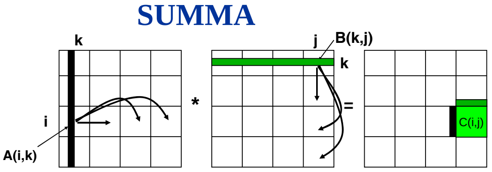


- i, j represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns
- C(i,j) = C(i,j) + Σ_k A(i,k)*B(k,j)
- Assume a p_r by p_c processor grid ($p_r = p_c = 4$ above)
  - Need not be square

SUMMA performance

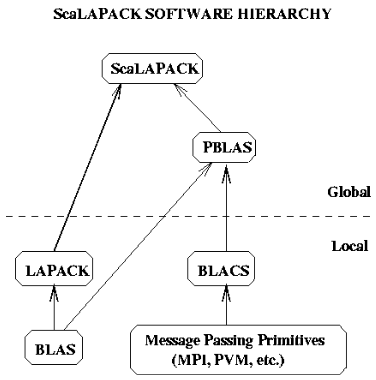
- Total time = $2*n^3/p$ + α * log p * n/b + β * log p * n^2 /s
- Parallel Efficiency =
  - 1/(1 + α * log p * p / (2*b*n^2) + β * log p * s/(2*n))
- ~Same β term as Cannon, except for log p factor
  - log p grows slowly so this is ok
- Latency (α) term can be larger, depending on b
  - When b=1, get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

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 p_r   … in parallel
       owner of A(i,k) broadcasts it to whole processor row
   for all j = 1 to p_c   … 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
```

ScaLAPACK Parallel Library



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

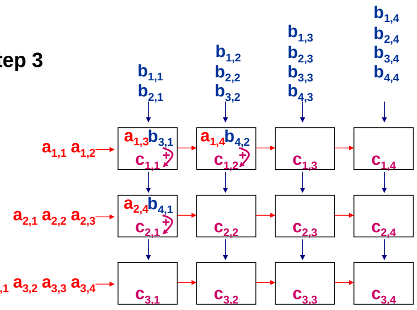
Maximum speed for PDGEMM
= # Procs * speed of DGEMM

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

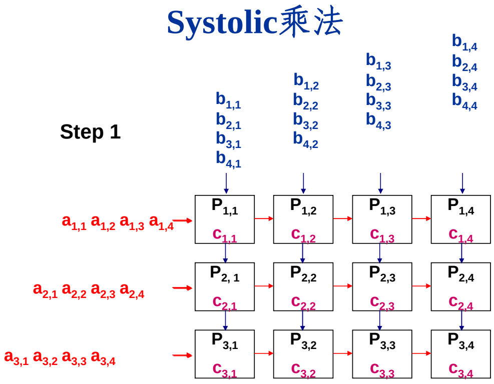
Systolic乘法

Step 3



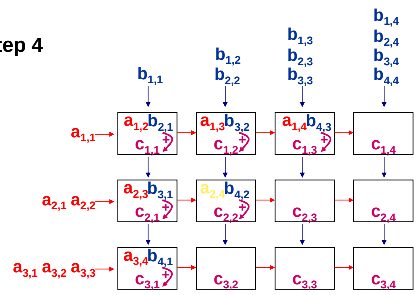
Systolic乘法

Step 1



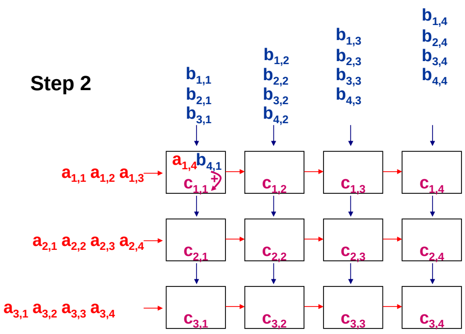
Systolic乘法

Step 4



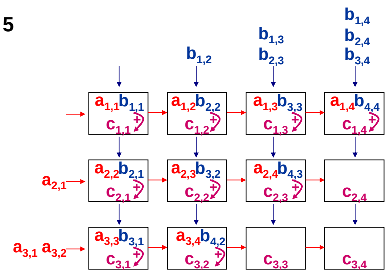
Systolic乘法

Step 2



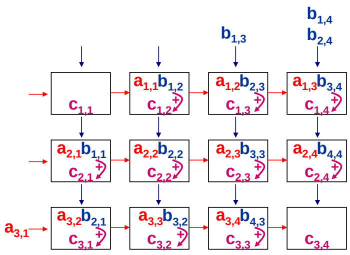
Systolic乘法

Step 5



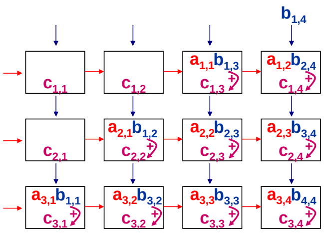
Systolic乘法

Step 6



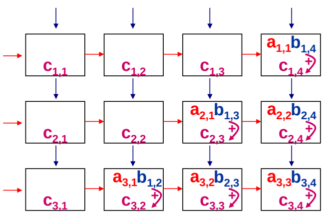
Systolic乘法

Step 7



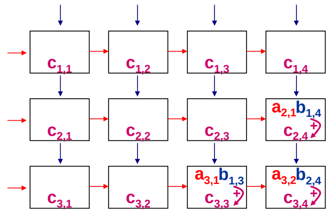
Systolic乘法

Step 8



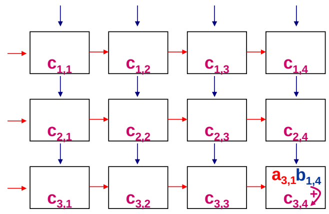
Systolic乘法

Step 9



Systolic乘法

Step 10



Systolic乘法

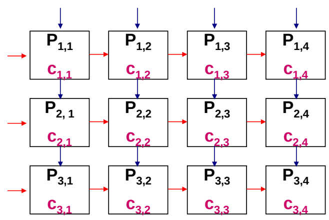
Over

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

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

…………

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



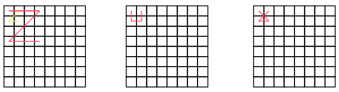
Systolic乘法

- 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**

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；

Summary of Parallel Matrix Multiplication

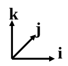
- 1D Layout
  - Bus without broadcast - slower than serial
  - Nearest neighbor communication on a ring (or bus with broadcast): Efficiency = 1/(1 + O(p/n))
- 2D Layout
  - **Cannon**
    - Efficiency = 1/(1+O(a * (sqrt(p) /n)3 +b* sqrt(p) /n))
    - Hard to generalize for general p, n, block cyclic, alignment
  - **SUMMA**
    - Efficiency = 1/(1 + O(a * log p * p / (b*n2) + b*log p * sqrt(p) /n))
    - Very General
    - b small => less memory, lower efficiency
    - b large => more memory, high efficiency
  - **Recursive layouts**
    - Current area of research

- **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

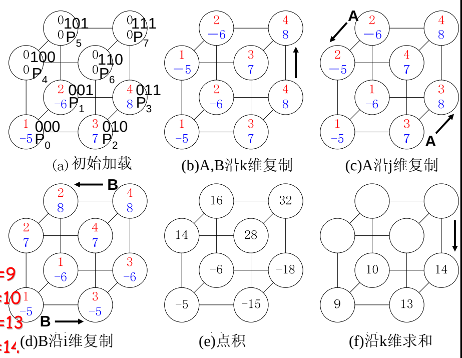


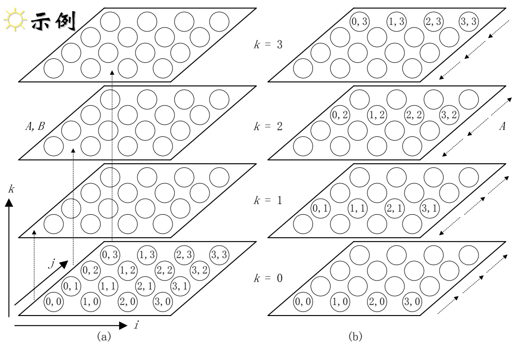
图9. 12

DNS乘法: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=(r2r1r0)_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
```

DNS乘法

示例



DNS乘法