第6讲 稠密矩阵算法

并行计算机的消息传递时间参数说明

- 通过网络传输一个消息所需要的总时间由以下组成：
  - 启动时间(t_s)：启动时间是在发送节点和接收节点处理消息所花费的时间。一种延迟时间，含
    - 消息准备时间（添加消息头、尾和校正信息）
    - 执行路由算法时间和本地节点与路由器建立连接的时间
  - 每站时间(t_h)：消息头在两个直接连接的节点间传送所花费的时间，也称节点延迟。
  - 每字传送时间(t_w)：它是由消息长度所决定的时间开销。

对系统的假设

- 系统中的任何处理器之间可以通信
  - 把它们看作是全连接的
- 因此，系统中任意两个进程都可以通信
  - 也它们当做是全连接的
- 在时间复杂度分析时，并行计算机的消息传递时间，一般不考虑进程之间通信的差异，并且**时间以一次算术运算时间为单位**
  - 此时，延迟时间和带宽都是常数
    - 假设延迟时间为t_s，带宽为每字传输时间t_w，每站时间t_h在通常的集群中较小而被忽略，于是：
    - 一次m个字的传输时间= $t_s + m * t_w$
  - 这个假设下的系统是分布式内存系统的一种抽象
    - 共享内存系统可看做分布式的特例
  - 处理器是全连接的

内容提要

1. 矩阵向量乘法
2. 笛卡尔拓扑
3. 矩阵矩阵乘法
   a) 简单的并行算法
      - 性能与可扩展性分析
   b) 笛卡尔拓扑平移操作
   c) Cannon算法
      - 性能分析
   d) 2.3 DNS算法
4. 解线性方程组

矩阵算法介绍

- 本讲讨论对象是稠密矩阵(dense matrix)或满矩阵(full matrix)，这种矩阵没有或很少零元素。
- 由于矩阵和向量的规则结构，该类型的并行计算容易用于数据分解。
- 根据具体的计算，可以对输入、输出或者中间数据进行分解。
- 大多算法使用一维和二维块、循环以及块循环等划分。

1 矩阵向量乘法

- 1.1 一维行划分
  - 每个进程一行
  - 进程数少于n（可扩展性分析）
- 1.2 二维划分
  - 每个进程一个元素
  - 进程数少于n^2（可扩展性分析）
  - 一维与二维划分的比较

矩阵向量乘法

- 一个*n* x*n* 稠密矩阵A乘一个*n* x 1向量 *x* 产生一个*n* x 1结果向量*y*。
  - 常用于迭代法，一次迭代的结果y=Ax要在下一次迭代使用：
    for（...;...;...）{y=Ax； x=y；}
- 假设一对乘加运算需要一个单位时间，则串行运算的时间为：

$$W = n^2$$

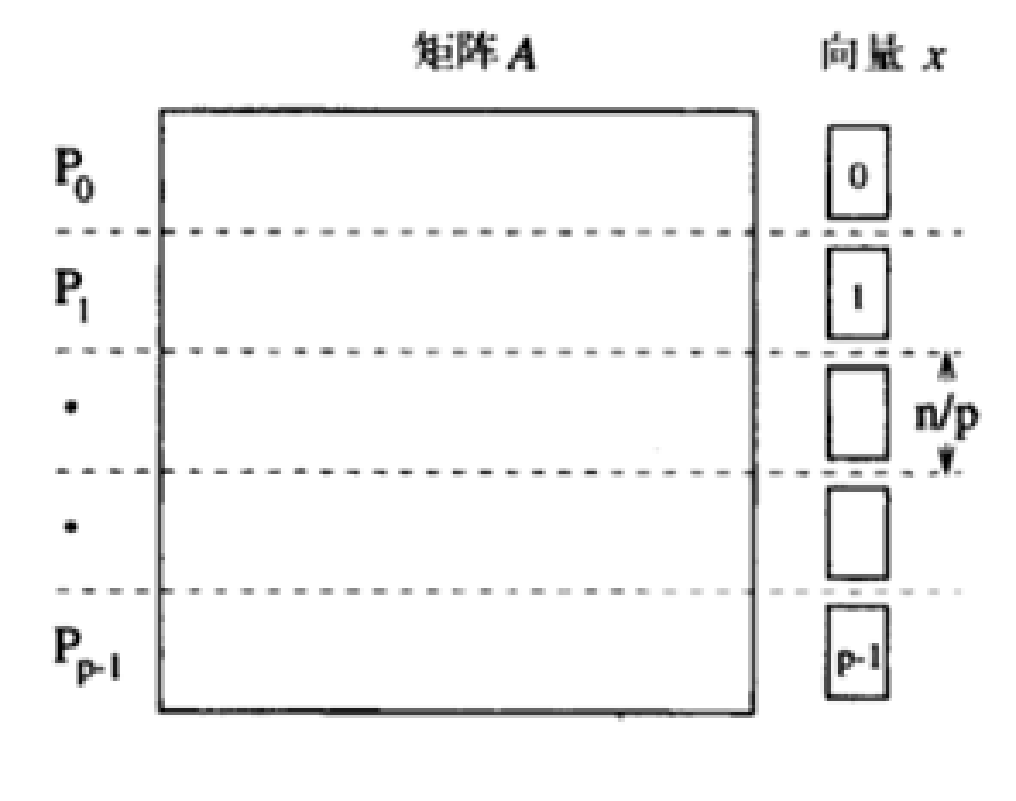
矩阵向量乘法
一维行划分

- 考虑*n* x*n* 矩阵A和*n* x1向量*x* 在p个进程之间划分。
- p=n
  - 每个进程P_i储存矩阵*A* 的一整行*A*[i,*]。
  - 每个进程P_i储存向量*x* 的一个元素*x*[i]。
- p<n
  - 每个进程P_i储存
    - 矩阵*A* 的n/p个整行*A*[k,*]
    - $_i$储存向量*x* 的n/p个元素*x*[k]
    - 其中k=i*n/p,…,(i+1) *n/p-1;
- 向量x为什么要分开存储?
  - 留到后面解释

矩阵向量乘法

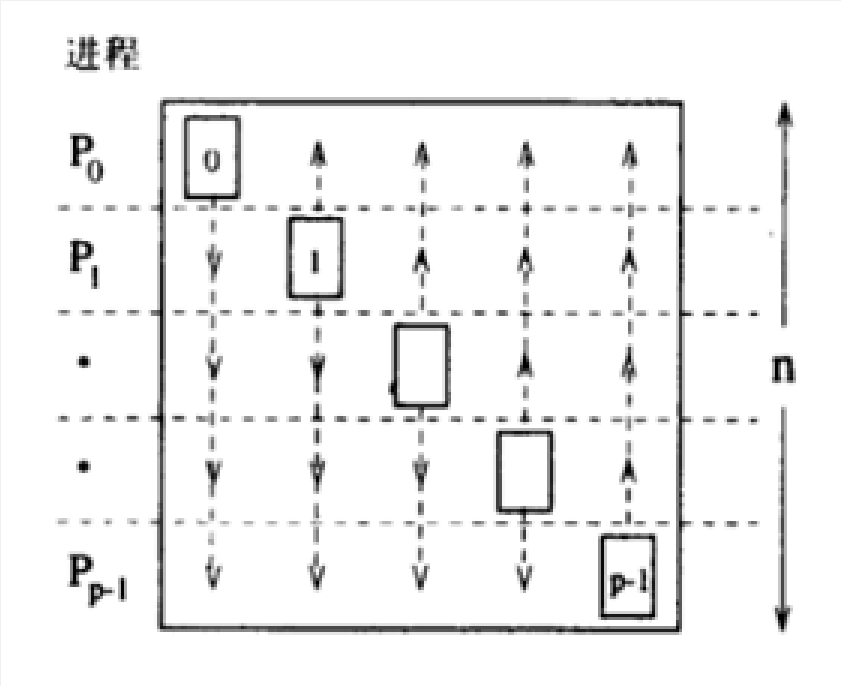
一维行划分

使用一维行划分的*n*x*n*矩阵与*n*x1向量的乘法对于每个进程一行的情况，*p*<=*n*。



初始，每个进程保存向量x的n/p个元素，矩阵A的n/p行。

矩阵向量乘法
一维行划分

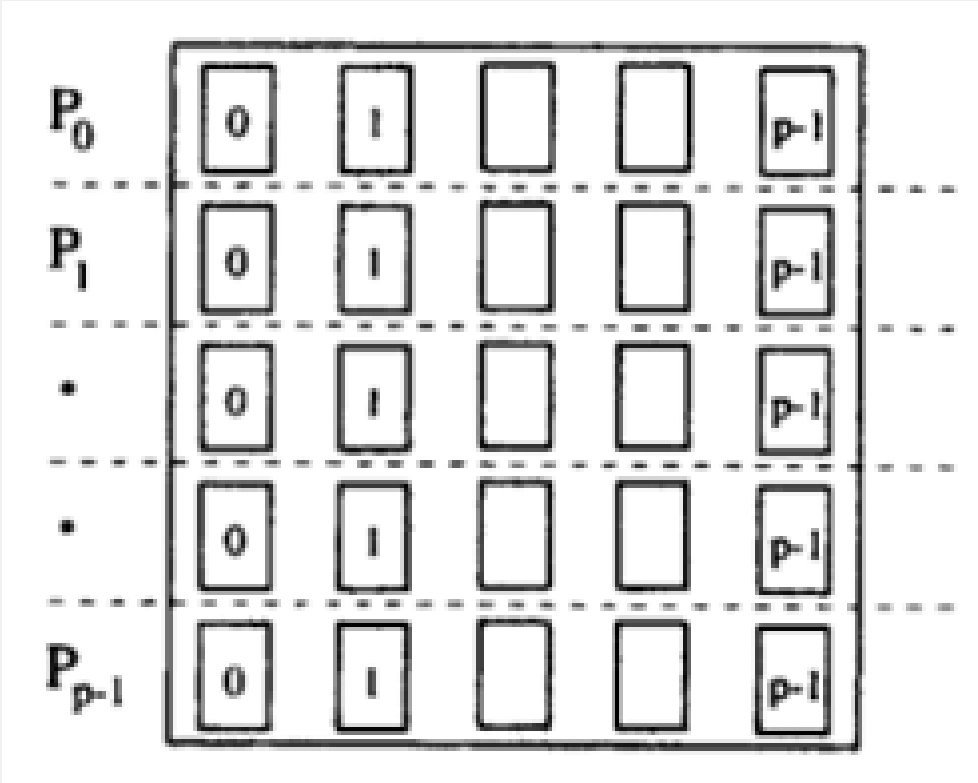


第一步，每个进程将它所保存的那n/p个向量x元素广播给其他进程，是多对多广播或全收集集合通信操作。MPI_Allgather或MPI_Allgatherv

在超立方上时间复杂度为 (其中m=n/p每个进程传输信息数量)

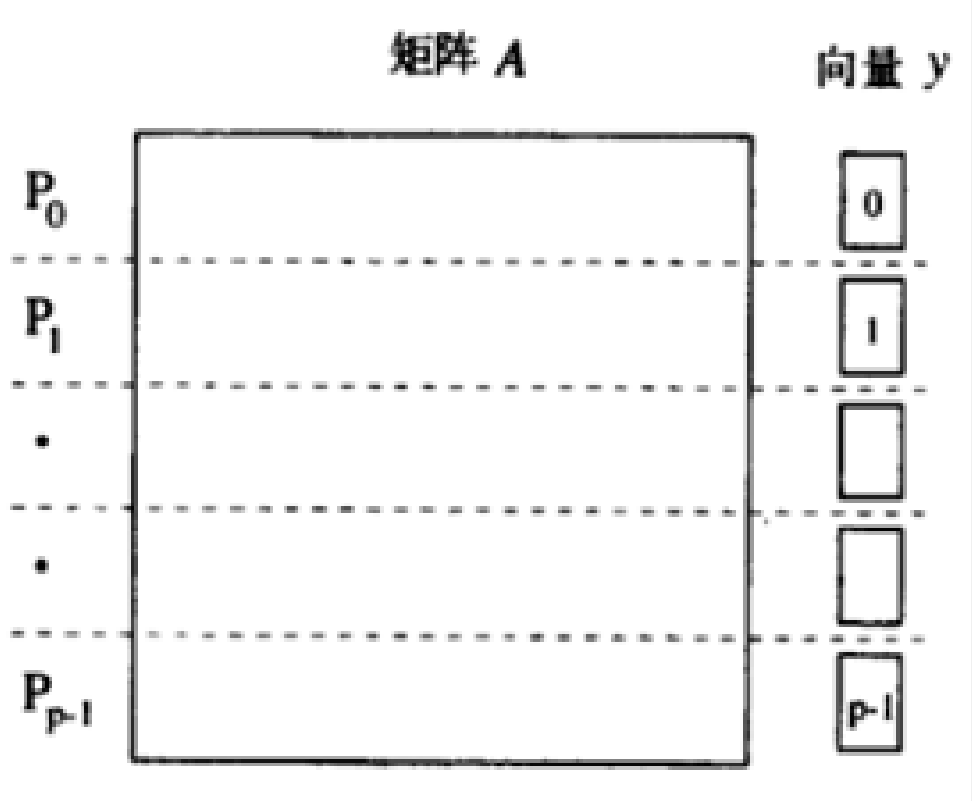
$$T = \sum_{i=1}^{\log p} \left(t_s + 2^{i-1} t_w m\right) = t_s \log p + t_w m(p-1) \approx t_s \log p + t_w n$$

矩阵向量乘法
一维行划分



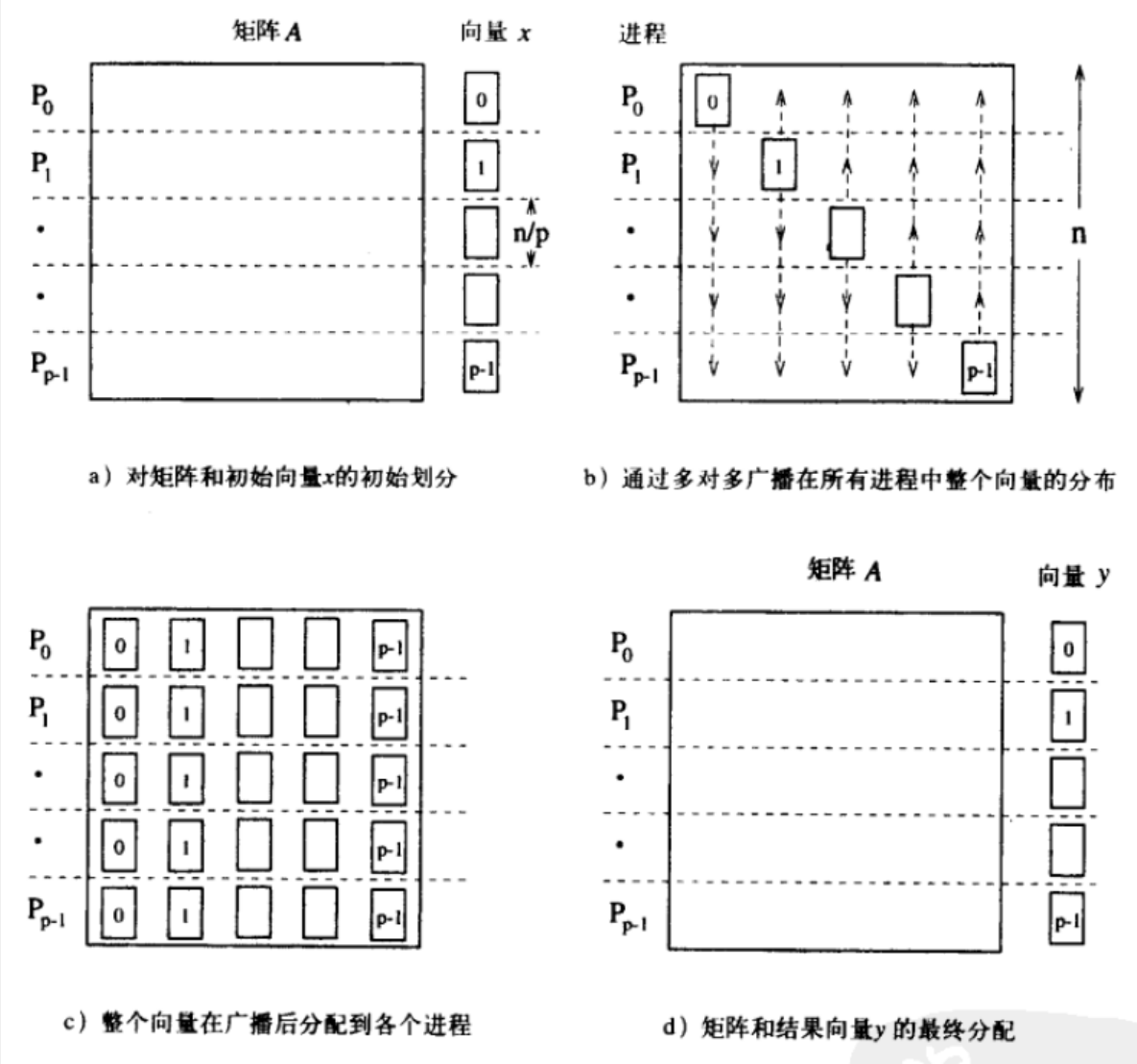
第二步，每个进程将它所获得向量x元素与矩阵A的各行元素内积。计算量为$T=n/p*n=n^2/p$

矩阵向量乘法
一维行划分



第三步，每个进程将它所获得的内积结果存入向量y的相应元素，每个进程存有n/p个y的元素。

矩阵向量乘法
一维行划分



使用一维行划分的nxn 矩阵与nx1 向量的乘法对于每个进程一行的情况，p=n。

向量x为什么要分开存储?

- 如果每个进程都拥有整个x，不就减少通信了？
- 矩阵向量乘的主要应用场景是迭代法
  - 典型的迭代法：

```
while（条件）{
    y=Ax;
    x=y;
}
```

  - 上一次迭代的结果y=Ax要在下一次迭代中使用
    - 对于一维行划分，除初始迭代步外，其他迭代步的x是上一步计算得来：总是分散在各个进程中保存的

矩阵向量乘法
一维行划分

- 设每个进程开始只有x的一个元素，即p=n，在计算开始前要求一次全收集集合通信，用于让每个进程获得完整的向量x。所需时间

$$T = t_s \log p + t_w n = \Theta(n)$$

- 进程P_i计算$y[i] = \sum_{j=0}^{n-1}(A[i,j] \times x[j])$ 所需时间 $T = n/p*n = n^2/p = \Theta(n)$
- 全收集集合通信操作和向量x与矩阵A一行的乘法所需时间均为$\Theta(n)$。
- 因此，并行的时间复杂度也为 $\Theta(n)$。
- 进程时间总和即计算成本为$pT_p = \Theta(n^2)$=串行计算时间$W = n^2$，成本最优。

矩阵向量乘法
一维行划分

- 考虑进程数$p<n$的情况，使用块一维划分方法。
- 每个进程储存矩阵A的n/p个整行和向量x的n/p个元素。
- 全收集通信在p个进程间进行，传输的信息量为n/p。
- 每个进程接着在本地进行n/p次点乘。
- 因此，并行运行时间是：

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

- 当p= Θ(n)时，是成本最优的。

矩阵向量乘法
一维行划分的可扩展性分析——计算强度

- 进程数$p \le n$，使用块一维划分方法。
- 并行运行时间是：

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

- 其中$\frac{n^2}{p} = \frac{W}{p}$是峰值计算时间，$t_s \frac{p \log p}{n^2} + t_w \frac{p}{n}$的倒数——计算通信比为$\frac{n^2}{t_s p \log p + t_w p n}$，也就是计算强度。越大越好。
  - 另外可见，p= Θ(n)时，时间最小。

矩阵向量乘法
一维行划分的可扩展性分析——等效率函数

- 按照第3讲的分析以及上面的推导，我们有总开销：

$$T_o = pT_p - W = t_s p \log p + t_w np$$

- 决定并行算法等效率函数的主要关系是 $W = KT_o$，其中 $K = E/(1-E)$，E 是期望效率。
- 注意p≤n，所以 $W = KT_o < K(t_s + t_w)np$，

即 $W = n^2 < K(t_s + t_w)np$，

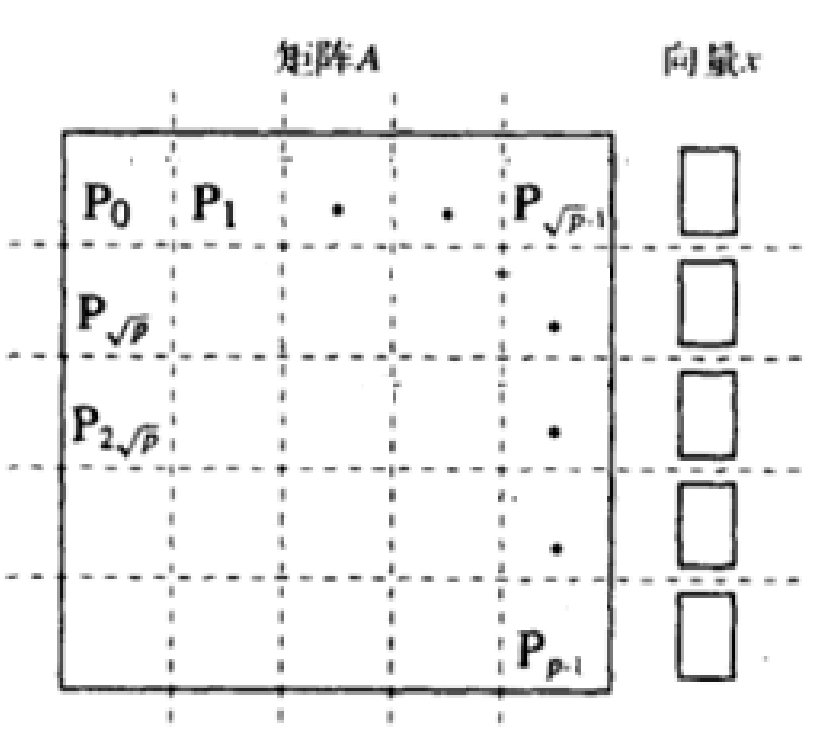
上式约去n后在两边平方，得到 $W = O(p^2)$。

- 又由于 $p \le n$，$W = n^2 = \Omega(p^2)$。
- 综上，渐进等效率函数为 $W = \Theta(p^2)$ 或 $n = \Theta(p)$ 。

矩阵向量乘法
二维划分

- 把n xn 矩阵A 在p（ $p \le n^2$ ）个进程中划分，p个进程组成阵列$\sqrt{p} \times \sqrt{p}$，把n x1向量x 分布在个进程阵列的最后一列
- $p=n^2$
  - 每个进程拥有矩阵A 的一个元素。
  - 把n x1向量x 分布在n 个进程的最后一列。
- $p<n^2$
  - 每个进程P_i储存矩阵A 的具有$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$阶的子矩阵$A[i,j]$。
  - 最后一列进程P_i（$i = \sqrt{p} - 1, 2\sqrt{p} - 1, \dots, p - 1$）储存向量$x$ 的$\frac{n}{\sqrt{p}}$个元素$x[k]$。
- 通过后面的分析会知道，二维划分相比一维有更低的时间复杂度和更好的可扩展性。

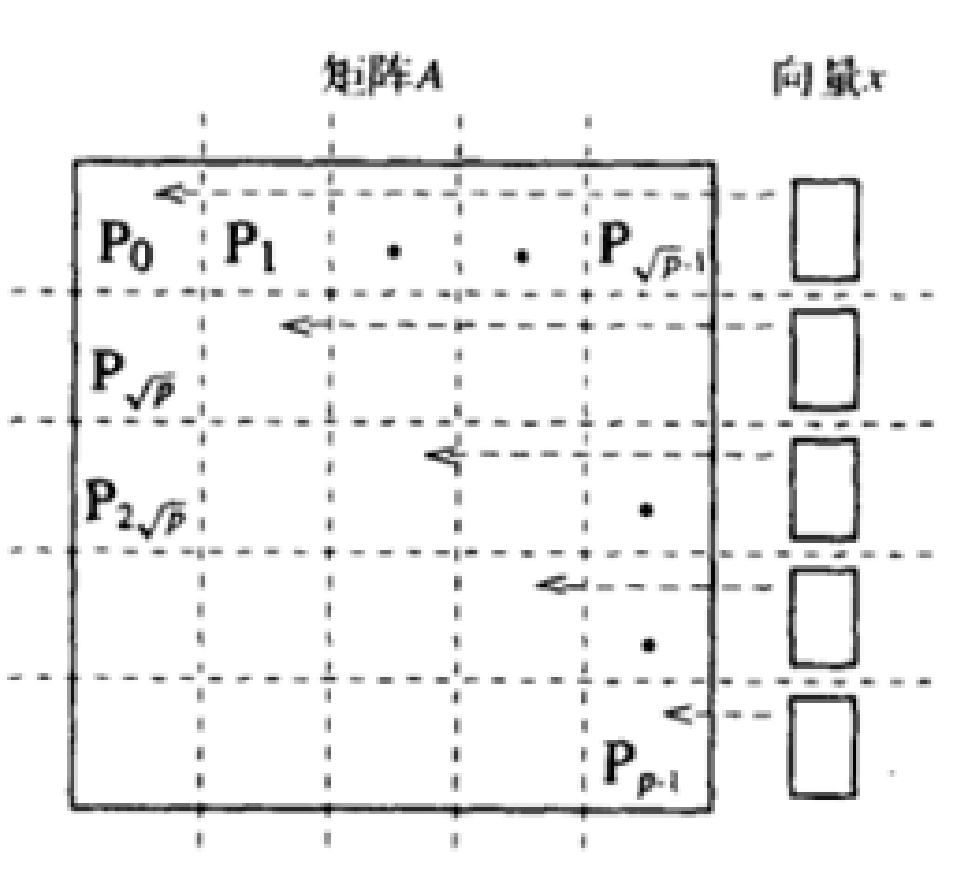
矩阵向量乘法
二维划分



使用二维块划分的矩阵向量乘法，共有$\sqrt{p} \times \sqrt{p} = p$个进程，对于nxn矩阵A有$p \le n^2$。

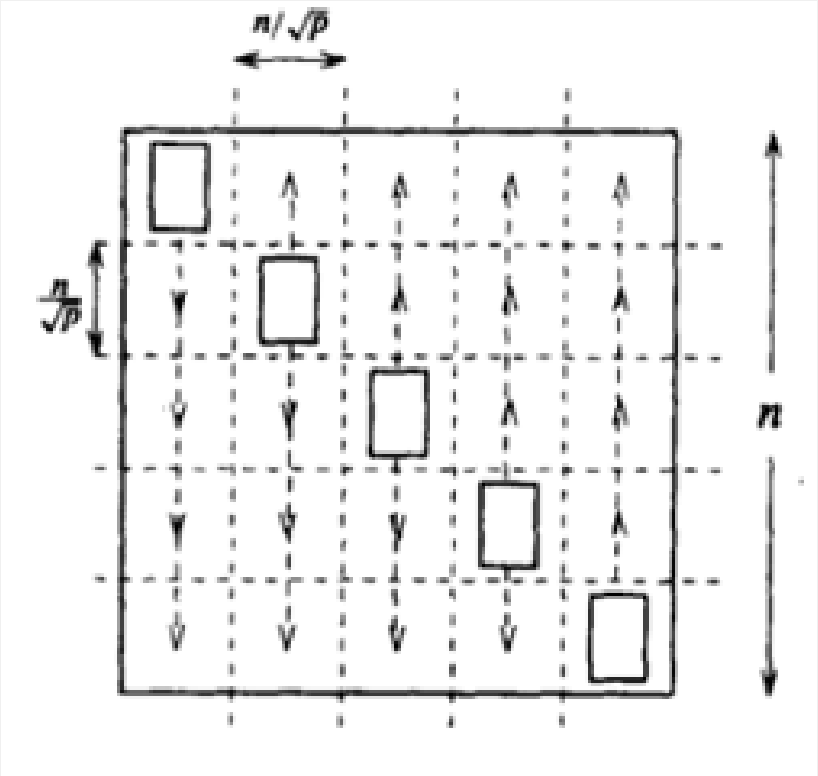
每个进程保存A的$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$个元素。最后一列进程每个保存x的$\frac{n}{\sqrt{p}}$个元素。

矩阵向量乘法
二维划分



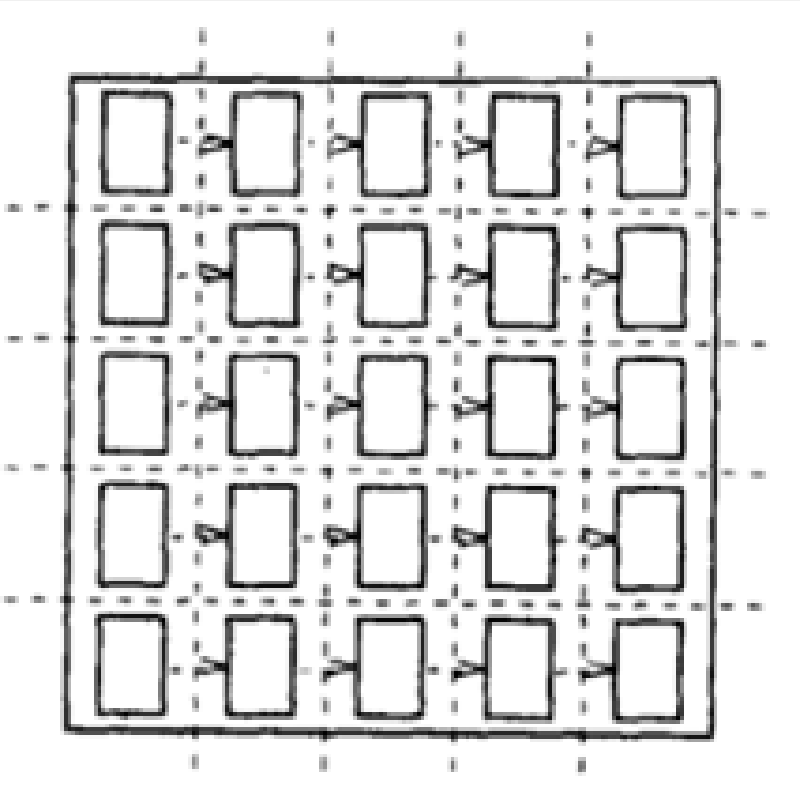
- 第一个通信步将向量x的元素沿着矩阵的主对角线排列。
  - 最后一列进程将其保存的$\frac{n}{\sqrt{p}}$个x元素传给进程所在行的主对角线上的进程

矩阵向量乘法
二维划分



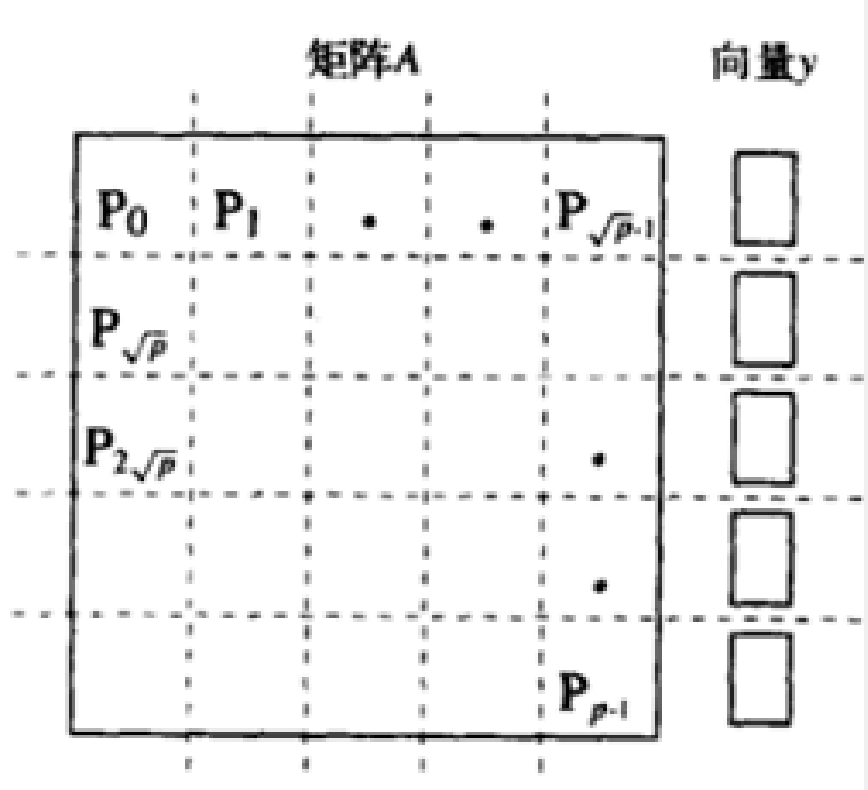
•第二个通信步从主对角线上将这些x的元素传给进程所在列的其他进程，即同时进行$\sqrt{p}$个一对多广播。

矩阵向量乘法
二维划分



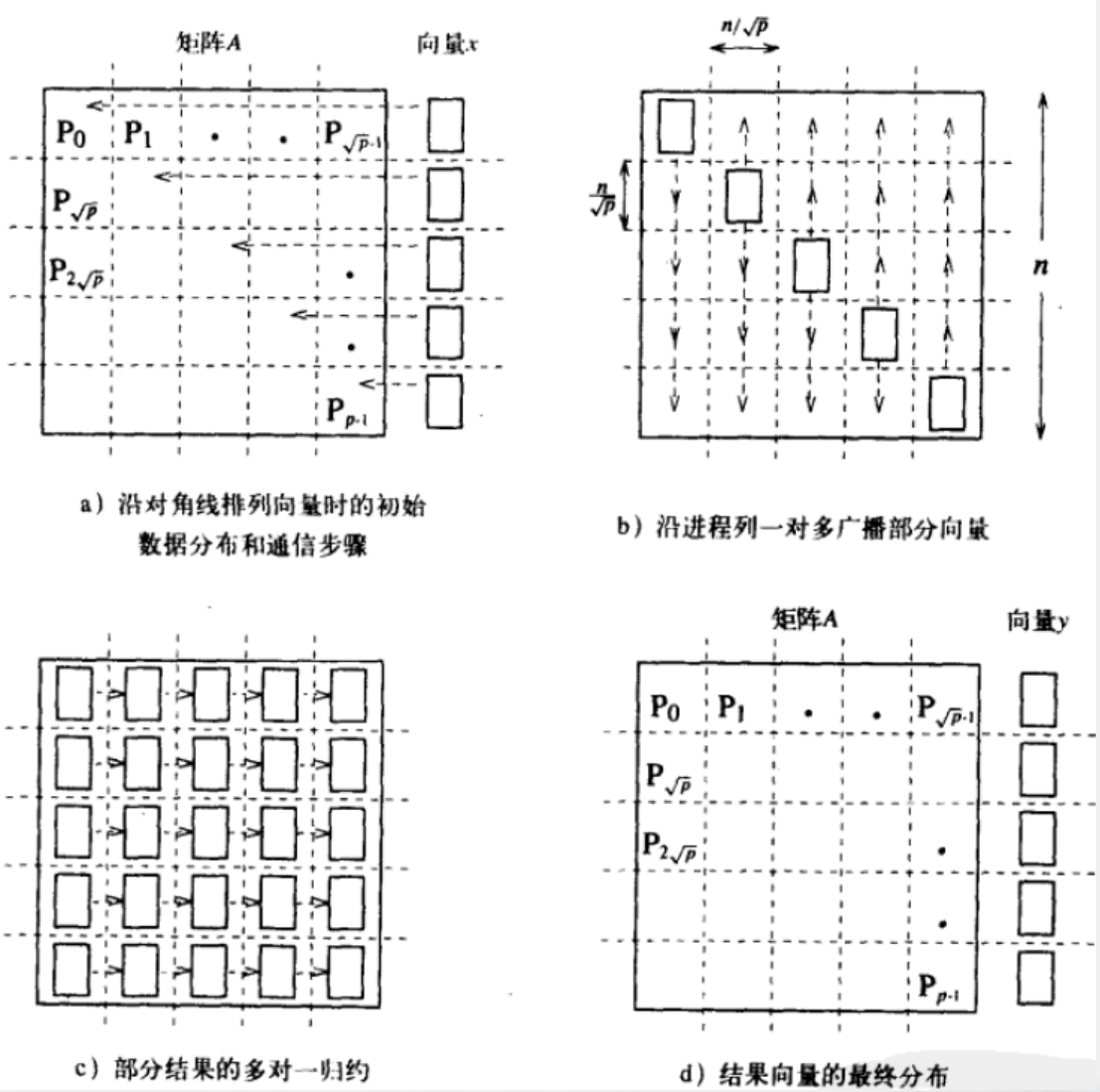
•最后一个通信步对每一行进程进行一次多对一归约，合并计算结果。

矩阵向量乘法
二维划分



- 计算结果在最后一列进程中。

矩阵向量乘法
二维划分



使用二维块划分的矩阵向量乘法。对于每个进程一个元素的情形，对于$n \times n$矩阵有$p = n^2$。

矩阵向量乘法
二维划分计算时间估计

- 当进程数$p \le n^2$，每个进程储存矩阵的一个$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$块。
- 向量以$\frac{n}{\sqrt{p}}$的粒度分布在最后一个进程列。
- 此时，通信步中的消息大小都变为$\frac{n}{\sqrt{p}}$。
- 每个进程计算一个$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$子矩阵和一个$\frac{n}{\sqrt{p}}$向量的乘积。这里有$n^2/p$次乘法运算和$\frac{n}{\sqrt{p}} - 1$次加法。综合，计算时间为$n^2/p$

矩阵向量乘法
二维划分计算时间估计

- 第一步传送向量用时：
$$t_s + t_w\, n/\sqrt{p}$$
- 广播和归约分别用时：
$$(t_s + t_w\, n/\sqrt{p}) \log \sqrt{p}$$
- 本地运算用时：
$$n^2/p$$
- 并行计算时间：$T_p = ?$

矩阵向量乘法
二维划分的计算时间

- $T_p = \overbrace{{n^2}/{p}}^{\text{计算}} + \overbrace{t_s + t_w \frac{n}{\sqrt{p}}}^{\text{传向量到主对角进程}}$
  $+ \overbrace{\left(t_s + t_w \frac{n}{\sqrt{p}}\right)\log\sqrt{p}}^{\text{主对角进程按列广播}} + \overbrace{\left(t_s + t_w \frac{n}{\sqrt{p}}\right)\log\sqrt{p}}^{\text{进程按行归约}}$
- $T_p = {n^2}/{p} + t_s + t_w \frac{n}{\sqrt{p}} + \left(2t_s \log\sqrt{p} + 2t_w \frac{n}{\sqrt{p}} \log\sqrt{p}\right)$

  于是得

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

矩阵向量乘法
二维划分的额外计算总成本

- $T_p = {n^2}/{p} + (t_s + t_w \frac{n}{\sqrt{p}})(1 + \log p)$
- $W = n^2$
- 额外计算总成本：

$$T_o = pT_p - W = t_s p + t_w n\sqrt{p} + (2t_s p \log\sqrt{p} + 2t_w n\sqrt{p}\log\sqrt{p})$$

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

$$= (t_s p + t_w n\sqrt{p})(1+\log p)$$

- 由于$p \le n^2$，所以

$$t_w n\sqrt{p}\log p \le T_o \le (t_s + t_w)n\sqrt{p}(1+\log p) \le 2(t_s + t_w)n\sqrt{p}\log p$$

- 所以

$$T_o = \Theta(n\sqrt{p}\log p)$$

二维划分的可扩展性分析——等效率函数

- 其等效率函数为
  - $W = n^2 = KT_o = K\Theta(n\sqrt{p}\log p)$，两边除以n
  - $n = K\Theta(\sqrt{p}\log p)$，两边平方
  - $W = \Theta(p\log^2 p)$
- 综上，整体渐近等效率函数为$\Theta(p\log^2 p)$。
- 对 $n^2 = \Theta(p\log^2 p)$，两边取对数，
- $2\log n = \Theta(\log p + 2\log\log p)$，得$\log p = \Theta(\log n)$
- 于是，可用logn代换等效率函数的logp，进一步得到成本最优时进程数目的渐近界：

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

矩阵向量乘法
二维划分的可扩展性分析——计算强度

- 进程数$p<=n^2$，使用块二维划分方法。
- 并行运行时间是：

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

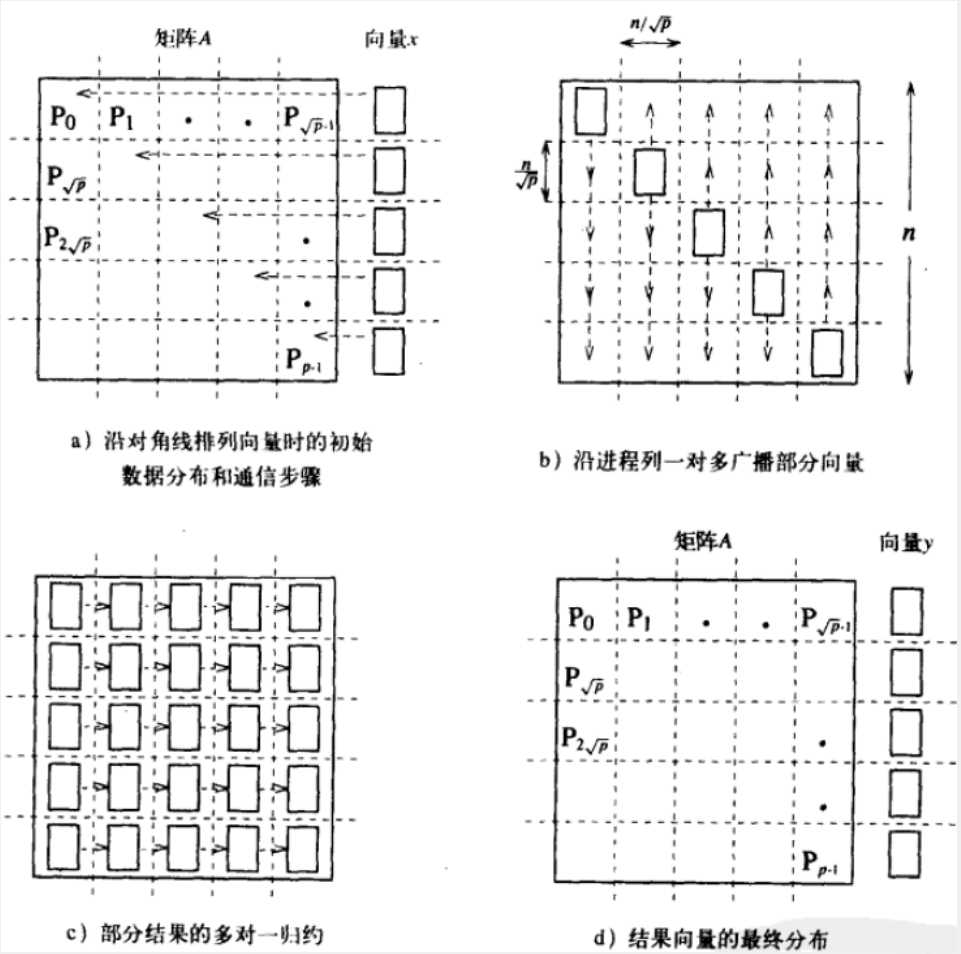
- $= \frac{n^2}{p}\left(1 + \left(t_s \frac{p}{n^2} + t_w \frac{\sqrt{p}}{n}\right)(1 + \log p)\right)$
- 其中$\frac{n^2}{p}$是峰值计算时间，$\left(t_s \frac{p}{n^2} + t_w \frac{\sqrt{p}}{n}\right)(1 + \log p)$的倒数——计算通信比为$\frac{n^2}{(t_s p + t_w \sqrt{p} n)(1 + \log p)}$，也就是计算强度。越大越好。

矩阵向量乘法
一维与二维划分的比较

- 在进程数目相等的情况下，二维划分比一维更快。
- 二维划分有更好的渐进等效率函数：阶低的好
  - $W = \Theta(p \log^2 p)$：二维
  - $W = \Theta(p^2)$：一维
- 计算强度：强度大的好 → 二维好
  - (二维)$\frac{n^2}{(t_s p + t_w \sqrt{p} n)(1 + \log p)} > \frac{n^2}{t_s p \log p + t_w p n}$(一维)
- 并发度：二维$p \leq n^2$，一维$p \leq n$
- 因此，无论进程数多少，二维划分都是更好的选择。

矩阵向量乘法
二维行划分的MPI实现

- 为考虑一般性，假设矩阵A划分为NxN块，相应地，向量x划分为N段，并且每个进程计算1个矩阵块与1个向量段的乘法，于是进程数p=MxN。
- 从右图，整个计算过程有两个主要的集合通信操作：
  1. 每一列块的广播
  2. 每一行块的归约
- MPI实现的关键是
  - 如何按列分别生成通信域
    - 这样就可按列广播
  - 如何按行分别生成通信域
    - 这样就可按行归约



矩阵向量乘法
二维行划分的MPI实现

如何按列分别生成通信域？	如何按行分别生成通信域？
– 先识别进程所在列 ➢ col=rank%N – 用MPI_Comm_split	– 先识别进程所在行 ➢ row=rank/N – 再用MPI_Comm_split

MPI_Comm_split(comm, color, key, newcomm)
功能：color值相同的进程编入同一个新的通信域，key值决定进程在新通信域中的编号

```
int  col, row;
MPI_comm  col_comm, row_comm;
……
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

col=rank%N; row=rank/N;
MPI_Comm_split(MPI_COMM_WORLD,col,row,col_comm);
MPI_Comm_split(MPI_COMM_WORLD,row,col,row_comm);
```

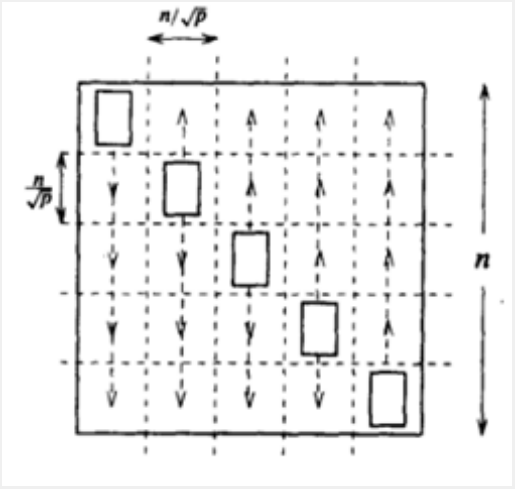
通信域要定义为数组吗？

矩阵向量乘法
二维行划分的MPI实现

- 如何在新生成通信域中使用集合通信？

col列的主对角线上的进程在col_comm中的rank=col
row行的主对角线上的进程在row_comm中的rank=row

MPI_Bcast(x,n/N,MPI_FLOAT,col,col_comm);
局部计算矩阵向量乘，结果存入x;
MPI_Reduce(x,y，n/N,MPI_FLOAT,MPI_SUM,N-1,row_comm);



将主对角线上x的元素传给进程所在列的所有进程——一对多广播。

下面我们介绍另一种实现：使用笛卡尔拓扑的方法

2. MPI进程的笛卡尔拓扑

问题提出

- 在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由基本问题几何和所用的数字算法所决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，我们指这种逻辑进程排列为“虚拟拓扑”。
- 例如，我们对矩阵进行网格划分分块，每个进程存放一块（如下图）；
  - 在全局通讯域上，进程按0，1，2，……，15编号，但对进程进行二维编排？
  - 按二维安排进程后，如何按行或按列进行集合运算或集合通信操作？

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

笛卡尔拓扑

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

- 在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由基本问题几何和所用的数字算法所决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，我们指这种逻辑进程排列为“虚拟拓扑”。

笛卡尔构造子

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

IN comm_old 输入通信子

IN ndims 笛卡尔网格的维数

IN dims 元素个数为ndims的整数数组，说明了每一维的大小

IN periods 元素个数为ndims的逻辑数组，说明了在每一维上网格是否是周期性的。True表示周期性的

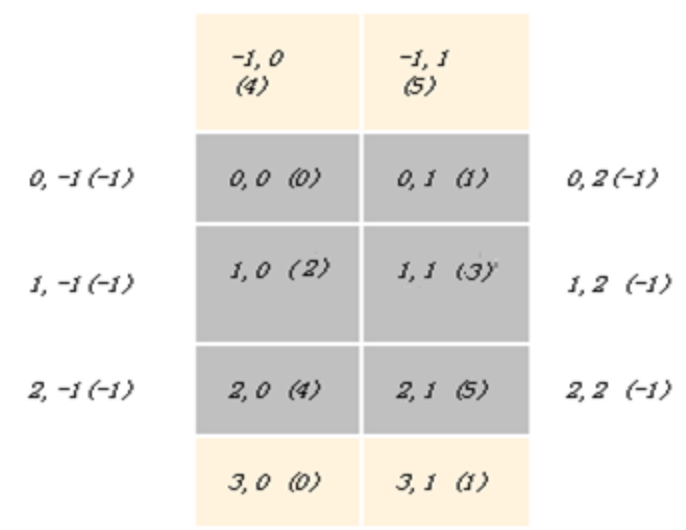
IN reorder 标识数可以重排序（逻辑型）：true表可以

OUT comm_cart 带有笛卡尔拓扑的新通信域

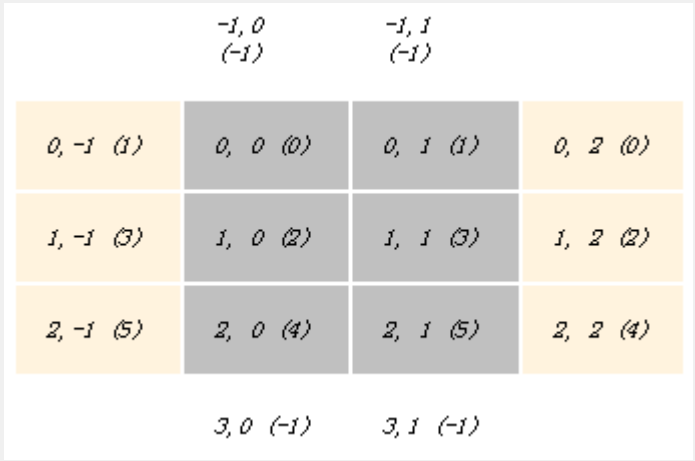
```
#include "mpi.h"
   MPI_Comm old_comm, new_comm;
   int ndims, reorder, periods[2], dim_size[2];

   old_comm = MPI_COMM_WORLD;
   ndims = 2;          /*  维数 */
   dim_size[0] = 3;    /* 行数 */
   dim_size[1] = 2;    /* 列数 */
   periods[0] = 1;     /* 设行周期变化(每一列是环) */
   periods[1] = 0;     /*设列非周期变化*/
   reorder = 1;        /* 为效率，允许进程重排序 */
```

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder, &new_comm);



periods[0]=TRUE;periods[1]=FALSE;



periods[0]=FALSE;periods[1]=TRUE;

辅助函数

int MPI_Dims_create(int nnodes, int ndims, int *dims)

- 根据用户指定的总维数ndims和总的进程数nnodes，帮助用户在每一维上选择进程的个数。返回结果放在dims中，它可以作为MPI_Cart_create的输入参数。但是用户也可以根据需要指定特定某一维i的进程数，比如置dims[i]=k>0，则本调用不会修改dims[i]的值，只有对于dims[i]=0的维，本调用才根据合适的划分算法给它重新赋值。dims[i]的初始值不能为负。

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)：将一维线性坐标(即进程标识号)转为进程的笛卡尔坐标

- IN comm 带有笛卡尔结构的通信域
- IN rank 一维线性坐标(即进程标识号)
- IN maxdims 最大维数，数组coords的长度
- OUT coords 返回该一维线性坐标对应的笛卡尔坐标，元素为ndims (maxdims)个的数组

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)：将进程的笛卡尔坐标转为一维线性坐标(即进程标识号)

IN comm 带有笛卡尔结构的通信域

IN coords 说明一个进程的笛卡尔坐标的整数数组(为ndims)

OUT rank 被说明进程的标识数（整数）

maxdims 在调用程序中向量coords的长度

例1.coords与rank的关系

```
int  ndims=2, dims[2]={0}, periods[2] = {0} ,reorder=0, coords[2]={0}, othercomm_sz;
MPI_Comm comm_cart, othercomm=MPI_COMM_WORLD;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
othercomm_sz= comm_sz;
MPI_Dims_create(othercomm_sz, ndims, dims); //计算各维大小
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods,reorder, &comm_cart); //创建笛卡尔拓扑
 MPI_Comm_rank(comm_cart, &my_cartrank); //获取进程在笛卡尔通信域的进程号
MPI_Cart_coords(comm_cart, my_rank, ndims, coords); //将全局进程号转为进程的笛卡尔坐标
MPI_Cart_rank(comm_cart, coords, &my_coords2rank); //将进程的笛卡尔坐标转为笛卡尔通信域的进程号


printf("MPI_COMM_WORLD: %d of %d;  coords: (%d,%d), cart_comm_rank:%d, coords to rank:%d.\n",
my_rank, comm_sz,coords[0],coords[1],my_cartrank,my_coords2rank);
```

```
[lnszyd@login ~]$ mpirun -n 8 ./a.out
MPI_COMM_WORLD: 3 of 8;  coords: (1,1), cart_comm_rank:3, coords to rank:3
MPI_COMM_WORLD: 0 of 8;  coords: (0,0), cart_comm_rank:0, coords to rank:0
MPI_COMM_WORLD: 1 of 8;  coords: (0,1), cart_comm_rank:1, coords to rank:1
MPI_COMM_WORLD: 2 of 8;  coords: (1,0), cart_comm_rank:2, coords to rank:2
MPI_COMM_WORLD: 4 of 8;  coords: (2,0), cart_comm_rank:4, coords to rank:4
MPI_COMM_WORLD: 5 of 8;  coords: (2,1), cart_comm_rank:5, coords to rank:5
MPI_COMM_WORLD: 6 of 8;  coords: (3,0), cart_comm_rank:6, coords to rank:6
MPI_COMM_WORLD: 7 of 8;  coords: (3,1), cart_comm_rank:7, coords to rank:7
```

例1.coords与rank的关系

```
int  ndims=2, dims[2]={0}, periods[2] = {0} ,reorder=0, coords[2]={0}, othercomm_sz;
MPI_Comm comm_cart, othercomm=MPI_COMM_WORLD;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
int color=(my_rank%2);   //按进程号的奇偶分组
int key=(10-my_rank*3<0?my_rank:10-my_rank*3);   //在新通信组的次序
MPI_Comm_split(MPI_COMM_WORLD, color, key, &othercomm);  //创建新通信域
MPI_Dims_create(othercomm_sz, ndims, dims);   //计算各维大小
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods,reorder, &comm_cart);  //创建笛卡尔拓扑
 MPI_Comm_rank(comm_cart, &my_cartrank);     //获取进程在笛卡尔通信域的进程号
MPI_Cart_coords(comm_cart, my_rank, ndims, coords);  //将全局进程号转为进程的笛卡尔坐标
MPI_Cart_rank(comm_cart, coords, &my_coords2rank);   //将进程的笛卡尔坐标转为笛卡尔通信域的进程号
printf("MPI_COMM_WORLD: %d of %d;  coords: (%d,%d), cart_comm_rank:%d, coords to rank:%d,
key=%d.\n", my_rank, comm_sz,coords[0],coords[1],my_cartrank,my_coords2rank,key);
```

```
[lnszyd@login ~]$ mpirun -n 8 ./a.out
MPI_COMM_WORLD: 6 of 8;  coords: (1,0), cart_comm_rank:2, coords to rank:2, key=6.
MPI_COMM_WORLD: 0 of 8;  coords: (1,1), cart_comm_rank:3, coords to rank:3, key=10.
MPI_COMM_WORLD: 2 of 8;  coords: (0,0), cart_comm_rank:0, coords to rank:0, key=4.
MPI_COMM_WORLD: 4 of 8;  coords: (0,1), cart_comm_rank:1, coords to rank:1, key=4.
MPI_COMM_WORLD: 1 of 8;  coords: (1,0), cart_comm_rank:2, coords to rank:2, key=7.
MPI_COMM_WORLD: 3 of 8;  coords: (0,0), cart_comm_rank:0, coords to rank:0, key=1.
MPI_COMM_WORLD: 5 of 8;  coords: (0,1), cart_comm_rank:1, coords to rank:1, key=5.
MPI_COMM_WORLD: 7 of 8;  coords: (1,1), cart_comm_rank:3, coords to rank:3, key=7.
```

划分成笛卡尔的子通信域

int MPI_Cart_sub(MPI_Comm com, int *remain_dims, MPI_Comm *newcomm)

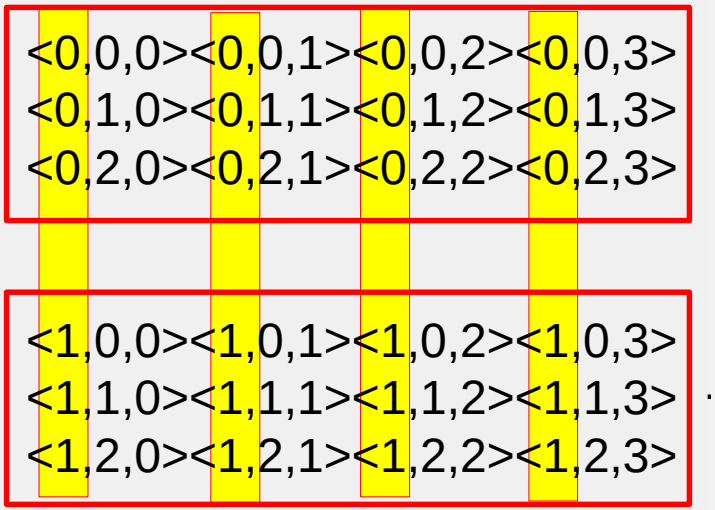
- IN comm 带有笛卡尔结构的通信域
- IN remain_dims 定义保留的维(逻辑数组)
- OUT newcomm 包含子网格的通信域,这个子网格包含了调用进程

➢ 将笛卡尔通信域划分成子笛卡尔通信域, remain_dims指出保留的维。若remain_dims[i]是true，则保留该维；若remain_dims[i]是false，则该维将划分为不同的通信域。

➢ 例：若comm对应的拓扑网格为2x3x4, 而remain_dims=<false,true,true>, 则本调用得到两个子通信域,它们在新旧通信域中笛卡儿坐标的对应关系为:

新旧通信域中笛卡儿坐标的对应关系例

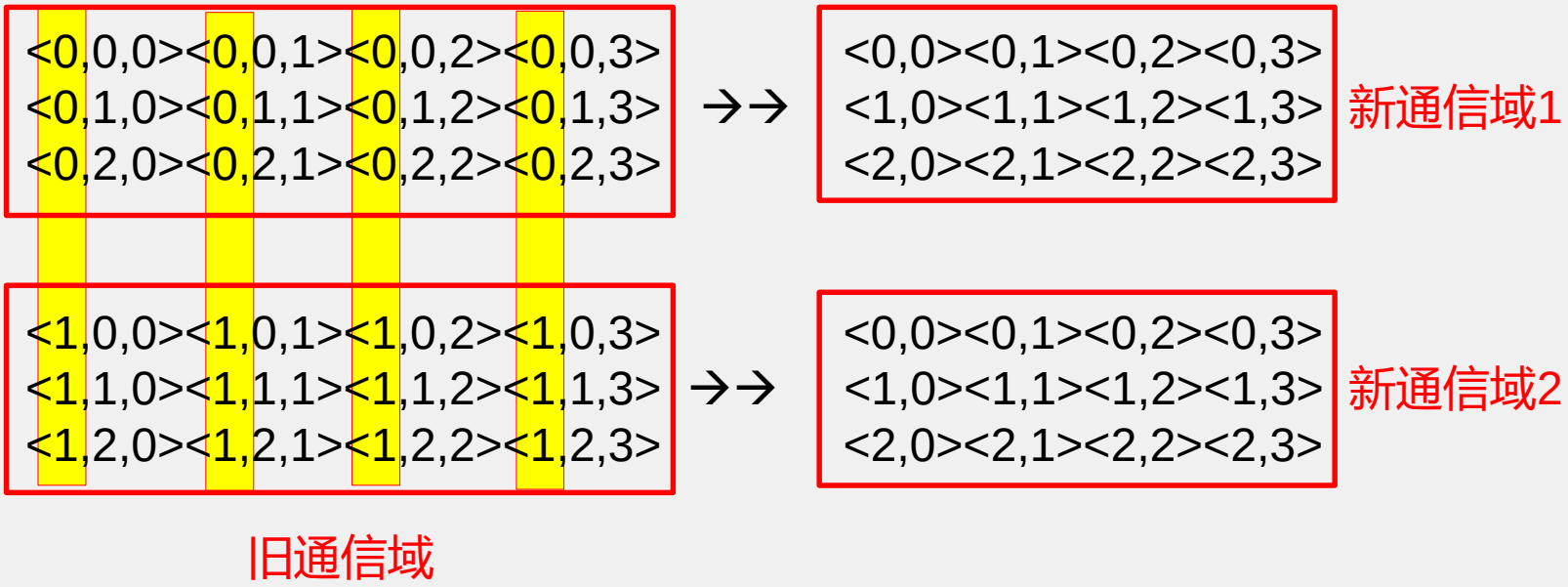
- 若comm对应的拓扑网格为2x3x4
- remain_dims=<false,true,true>
- 本调用得到两个子笛卡尔通信域：新通信域1和新通信域2
  - 由于旧通信域网格有两层：2x3x4
  - 对应remain_dims=<false,true,true>



旧通信域

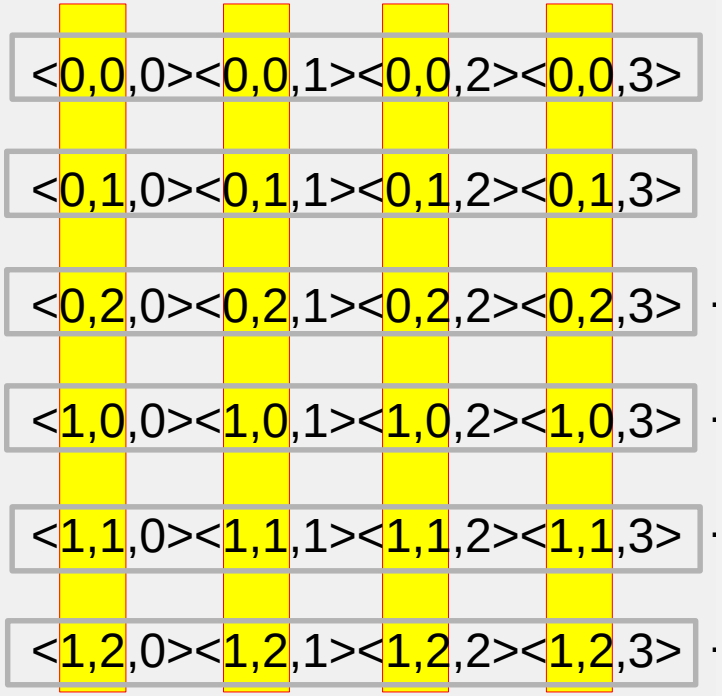
新旧通信域中笛卡儿坐标的对应关系例

- 若comm对应的拓扑网格为2x3x4
- remain_dims=<false,true,true>
- 本调用得到两个子笛卡尔通信域：新通信域1和新通信域2
  - 由于旧通信域网格有两层：2x3x4
  - 对应remain_dims=<false,true,true>



新旧通信域中笛卡儿坐标的对应关系例

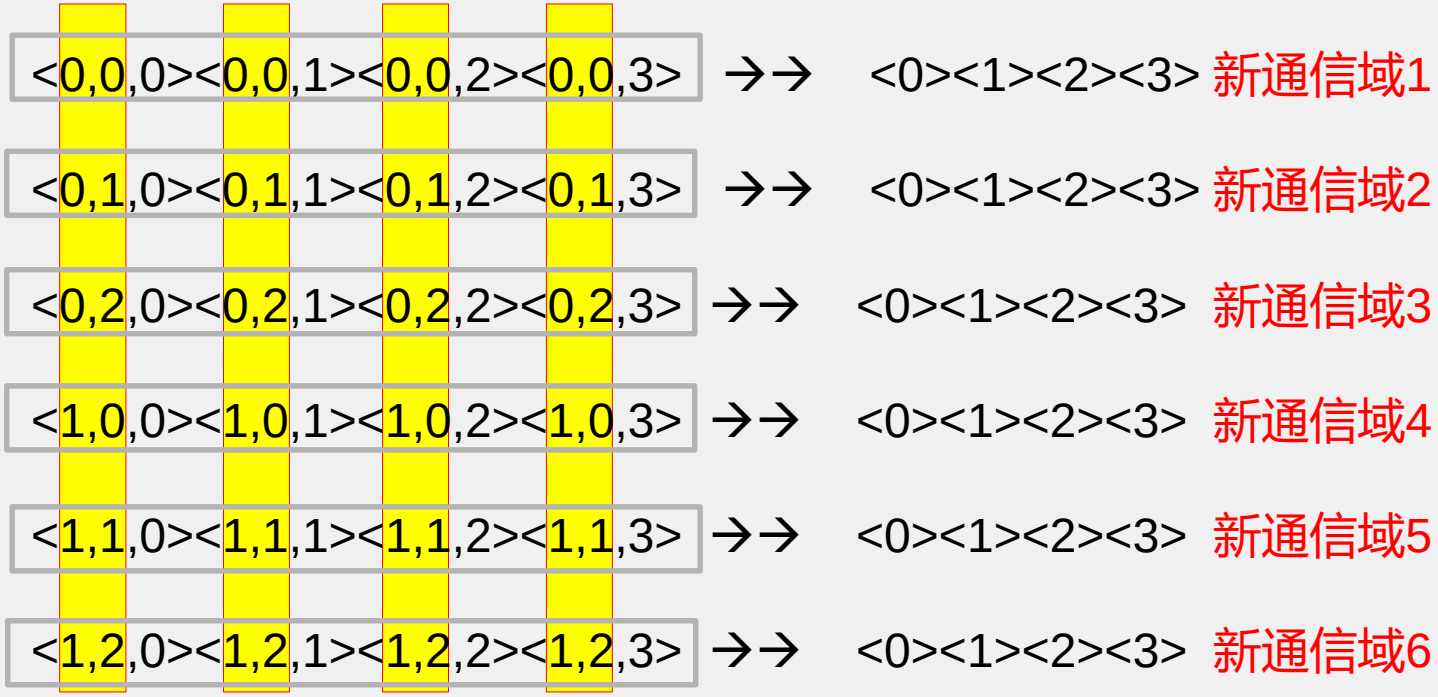
- 若comm对应的拓扑网格为2x3x4
- remain_dims=<false, false,true>
- 本调用得到六个子笛卡尔通信域：6个1维新通信域



旧通信域

新旧通信域中笛卡儿坐标的对应关系例

- 若comm对应的拓扑网格为2x3x4
- remain_dims=<false, false,true>
- 本调用得到六个子笛卡尔通信域：6个1维新通信域



创建笛卡尔行或列通信域

将二维笛卡尔通信域的按行按列分别划分成笛卡尔子通信域。

```
MPI_Comm comm_cart_rows,comm_cart_cols;  //行列通信域变量
int remain_dims[2] = {0,1};   //设定按行划分，保留列号作为进程号
 //创建行通信域
MPI_Cart_sub(comm_cart, remain_dims, &comm_cart_rows);

remain_dims[0] = 1;   //保留行号作为进程号
remain_dims[1] = 0;   //设定按列划分
//创建列通信域
MPI_Cart_sub(comm_cart, remain_dims, &comm_cart_cols);
```

2 矩阵与矩阵的乘法

- 2.1 简单的并行算法
  - 性能与可扩展性分析
- 2.2 Cannon算法
  - 性能分析
- 2.3 DNS算法
  - 进程数少于n^3
  - 性能分析

矩阵与矩阵乘法

- 考虑$n \times n$稠密矩阵乘法$C = A \times B$。
- 简单起见，假定最优串行复杂度为$O(n^3)$。
- 分块矩阵的运算在这里起重要作用。
- $n \times n$矩阵可以看成块$A_{i,j}(0 \le i, j < q)$的$q \times q$矩阵，其中每个块$A_{i,j}$是原矩阵的一个$(n/q) \times (n/q)$子矩阵。
- 在这种观点下，我们执行q^3次矩阵乘法，每个矩阵为$(n/q) \times (n/q)$

矩阵矩阵乘法
简单并行

- 考虑两个n xn 矩阵A 和B，分别划分成p个大小为$(n/\sqrt{p})$x$(n/\sqrt{p})$的块$A_{i,j}$和$B_{i,j}$。
- 进程$P_{i,j}$最初储存$A_{i,j}$和$B_{i,j}$，并计算结果矩阵的块$C_{i,j}$。
- 计算子矩阵$C_{i,j}$需要所有子矩阵$A_{i,k}$和$B_{k,j}$$(0 \le k < \sqrt{p})$。
- 每行进行矩阵A 的块的全收集通信，同时每列进行矩阵B 的块的全收集通信。
- 最后执行子矩阵的乘加运算。

矩阵矩阵乘法
简单并行

- 两个多对多广播操作用时：

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

- 计算包含$\sqrt{p}$ 次($n/\sqrt{p}$)x($n/\sqrt{p}$)子矩阵的乘法，用时：

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

- 总并行时间近似：

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

- 当$p = \Theta(n^2)$（或$n = \Theta(p^{1/2})$）时，成本最优，并且等效率函数是$W = \Theta(p^{3/2})$。
- 算法的主要缺点是内存需求过大。

矩阵矩阵乘法
Cannon算法

- 最初子矩阵$A_{i,j}$和$B_{i,j}$分配给进程$P_{i,j}$。
- 我们调度第i行$\sqrt{p}$个进程的计算，使得每个进程在不同时刻使用不同的$A_{i,k}$。
- 每完成一次子矩阵乘法，这些块在各进程之间进行一次移位，使得每个进程获得下一步需要的新的$A_{i,k}$。
- 对列使用同样的调度。
- 该算法需要的内存总量为$\Theta(n^2)$。

矩阵矩阵乘法

Cannon算法

初始进程$P_{i,j}$分配了子矩阵$A_{i,j}$和$B_{i,j}$。 $C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。

P_{00} A_{00} B_{00} C_{00}	P_{01} A_{01} B_{01} C_{01}	P_{02} A_{02} B_{02} C_{02}	P_{03} A_{03} B_{03} C_{03}
P_{10} A_{10} B_{10} C_{10}	P_{11} A_{11} B_{11} C_{11}	P_{12} A_{12} B_{12} C_{12}	P_{13} A_{13} B_{13} C_{13}
P_{20} A_{20} B_{20} C_{20}	P_{21} A_{21} B_{21} C_{21}	P_{22} A_{22} B_{22} C_{22}	P_{23} A_{23} B_{23} C_{23}
P_{30} A_{30} B_{30} C_{30}	P_{31} A_{31} B_{31} C_{31}	P_{32} A_{32} B_{32} C_{32}	P_{33} A_{33} B_{33} C_{33}

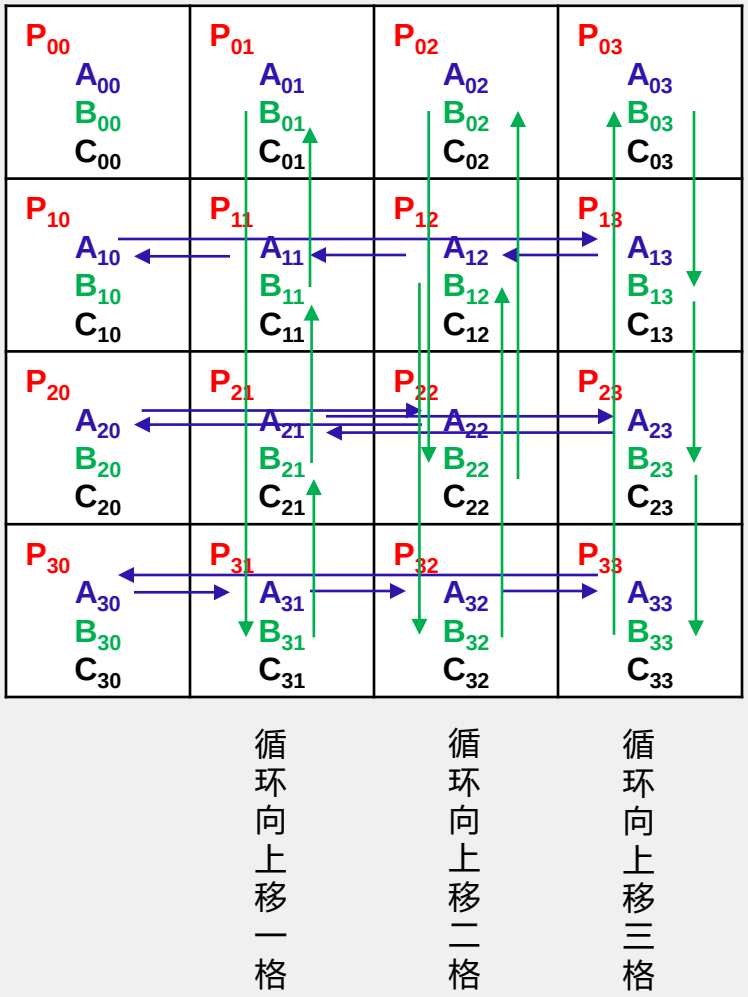
矩阵矩阵乘法

Cannon算法

初始进程$P_{i,j}$分配了子矩阵$A_{i,j}$和$B_{i,j}$。 $C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。



第一步：对子块进行排列：将相应子块传送给相应的进程。

矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。

P_{00} A_{00} B_{00} C_{00}	P_{01} A_{01} B_{11} C_{01}	P_{02} A_{02} B_{22} C_{02}	P_{03} A_{03} B_{33} C_{03}
P_{10} A_{11} B_{10} C_{10}	P_{11} A_{12} B_{21} C_{11}	P_{12} A_{13} B_{32} C_{12}	P_{13} A_{10} B_{03} C_{13}
P_{20} A_{22} B_{20} C_{20}	P_{21} A_{23} B_{31} C_{21}	P_{22} A_{20} B_{02} C_{22}	P_{23} A_{21} B_{13} C_{23}
P_{30} A_{33} B_{30} C_{30}	P_{31} A_{30} B_{01} C_{31}	P_{32} A_{31} B_{12} C_{32}	P_{33} A_{32} B_{23} C_{33}

第一步完成后各进程所拥有的相应子块：初始排列。
这时进程$P_{i,j}$可计算一个子块乘法：

$$C_{ij} = A_{ik} B_{kj}$$

（按反对角线方向，k=0, 1, 2, 3, 0, 1, 2）

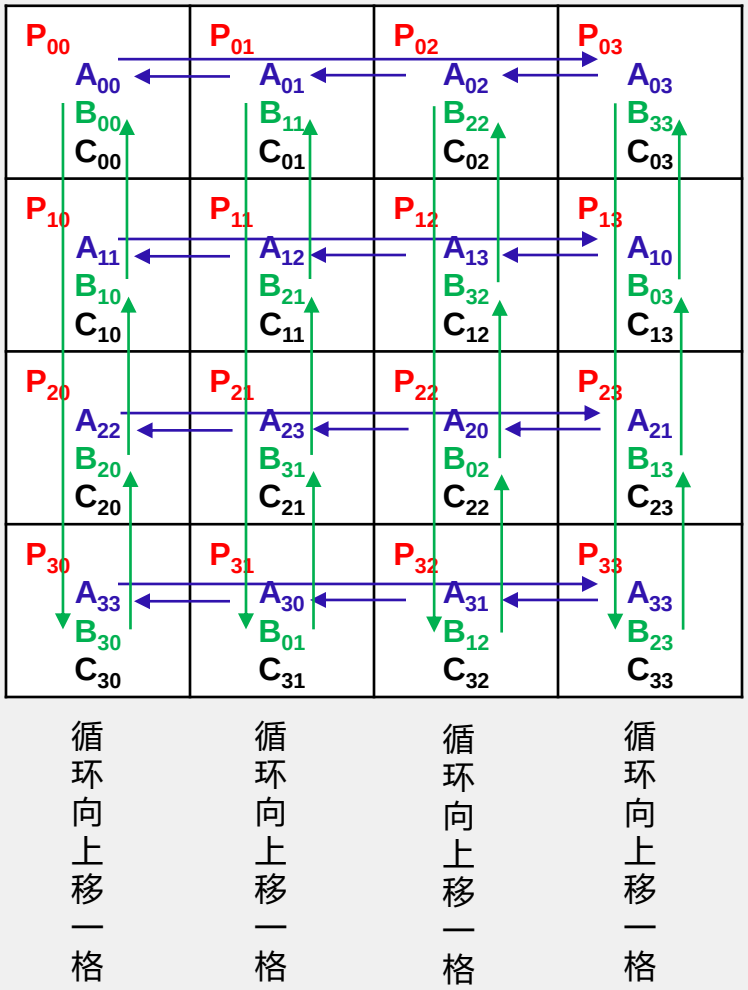
矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。



第二步：将A的子块按行左移一格，同时将B的子块按列上移一格

矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。

P_{00} A_{01} B_{10} C_{00}	P_{01} A_{02} B_{21} C_{01}	P_{02} A_{03} B_{32} C_{02}	P_{03} A_{00} B_{03} C_{03}
P_{10} A_{12} B_{20} C_{10}	P_{11} A_{13} B_{31} C_{11}	P_{12} A_{10} B_{02} C_{12}	P_{13} A_{11} B_{13} C_{13}
P_{20} A_{23} B_{30} C_{20}	P_{21} A_{20} B_{01} C_{21}	P_{22} A_{21} B_{12} C_{22}	P_{23} A_{22} B_{23} C_{23}
P_{30} A_{30} B_{00} C_{30}	P_{31} A_{31} B_{11} C_{31}	P_{32} A_{32} B_{22} C_{32}	P_{33} A_{33} B_{33} C_{33}

第二步完成后的结果。这时进程$P_{i,j}$可再计算一个子块乘法：

$$C_{ij} \mathrel{+}= A_{ik} B_{kj}$$

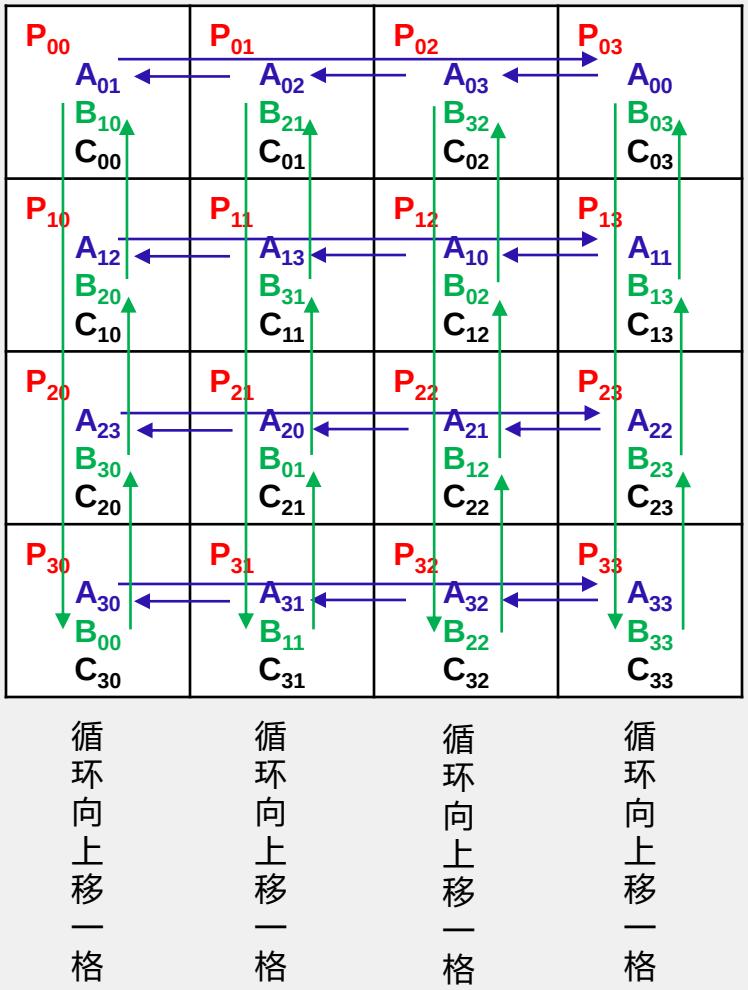
矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。



第三步与第二步一样：将A的子块按行左移一格，同时将B的子块按列上移一格

矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。

P_{00} A_{02} B_{20} C_{00}	P_{01} A_{03} B_{31} C_{01}	P_{02} A_{00} B_{02} C_{02}	P_{03} A_{01} B_{13} C_{03}
P_{10} A_{13} B_{30} C_{10}	P_{11} A_{10} B_{01} C_{11}	P_{12} A_{11} B_{12} C_{12}	P_{13} A_{12} B_{23} C_{13}
P_{20} A_{20} B_{00} C_{20}	P_{21} A_{21} B_{11} C_{21}	P_{22} A_{22} B_{22} C_{22}	P_{23} A_{23} B_{33} C_{23}
P_{30} A_{31} B_{10} C_{30}	P_{31} A_{32} B_{21} C_{31}	P_{32} A_{33} B_{32} C_{32}	P_{33} A_{30} B_{03} C_{33}

第三步完成后的结果。这时进程$P_{i,j}$又可再计算一个子块乘法：

$$C_{ij} \mathrel{+}= A_{ik} B_{kj}$$

然后，同样再进行第四步，即可完成矩阵乘法。

矩阵矩阵乘法
Cannon算法

- 用以下方式排列矩阵A和B的块，使得每个进程对其本地子矩阵相乘。
  ① 一开始，对矩阵A的所有子矩阵$A_{i,j}$，进行j-左移位（带回绕）；对矩阵B的所有子矩阵$B_{i,j}$，进行i-上移位（带回绕）。
  ② 执行一次本地的子矩阵乘法。
  ③ 所有的$A_{i,j}$向左移一位，$B_{i,j}$上移一位。
  ④ 执行下一次乘法并累积结果。
  ⑤ 跳至③, 重复上面的移位和乘法直至$\sqrt{p}$个块都完成。
- 算法相当于将简单并行乘法中的MPI_Allgather算法分解使用。

矩阵矩阵乘法
Cannon算法

- 初始排列包括一次按行和一次按列移位，需要的总时间为$2(t_s + t_w n^2/p)$
- 后续需要$\sqrt{p} - 1$次移位。
- $\sqrt{p}$次($n/\sqrt{p}$)x($n/\sqrt{p}$)子矩阵的乘法需要n^3/p的乘加运算。
- 因此算法的总并行时间为

$$T_p = \frac{n^3}{p} + 2\sqrt{p}(t_s + t_w n^2/p) = \frac{n^3}{p} + 2\sqrt{p} \cdot t_s + 2t_w \frac{n^2}{\sqrt{p}}$$

- Cannon算法的成本最优条件和等效率函数与前面提到的简单方法一致，但更节省空间。

利用笛卡尔坐标平移定位操作实现Cannon算法

笛卡尔坐标平移定位

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)：取得调用进程在图拓扑中的邻居

- IN comm 带有笛卡尔结构的通信域
- IN direction 平移的坐标维
- IN disp 偏移量(＞0：向右/下轮换，＜0：向左/上偏移)
- OUT rank_source 源进程的标识数
- OUT rank_dest 目标进程的标识数

笛卡尔拓扑对各维有个自动编号，分别从第0维到第ndims-1维。Dircction指定要轮换移动的维。坐标平移可同时适应有周期的和无周期的维。对无周期的维，平移到末端后再平移将得到MPI_PROC_NULL。

以二维情形为例。Comm是如右图的笛卡尔通信域。
MPI_Cart_shift(comm, 1, 1, &rank_src, &rank_dest);
在进程5中→ rank_src=4, rank_dest=6
在进程11中→ rank_src=10, rank_dest=-1

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,3)	15 (3,3)

笛卡尔坐标平移定位例

以二维情形为例。comm是如右图的笛卡尔通信域, 按列号周期环绕，按行号非周期。

```
 int periods[2] ={0,1}, dins[2]={4,4};
MPI_Cart_create(oldcomm, 2, dims, periods,
     0, &comm);
```

```
MPI_Cart_shift(comm, 1, -1, &ranksrc,
&rankdest);
```

- 在进程5中→ ranksrc=6, rankdest=4
- 在进程12中→ ranksrc=13, rankdest=15

```
MPI_Cart_shift(comm, 0, 1, &ranksrc,
&rankdest);
```

- 在进程5中→ ranksrc=1, rankdest=9
- 在进程12中→ ranksrc=8, rankdest=-1

	-1 (-1,0)	-1 (-1,1)	-1 (-1,2)	-1 (-1,3)	
3 (0,-1)	0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)	0 (0,4)
7 (1,-1)	4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)	4 (1,4)
11 (2,-1)	8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)	8 (2,4)
15 (3,-1)	12 (3,0)	13 (3,1)	14 (3,3)	15 (3,3)	12 (3,4)
	-1 (4,0)	-1 (4,1)	-1 (4,2)	-1 (4,3)	

笛卡尔坐标上执行数据轮换

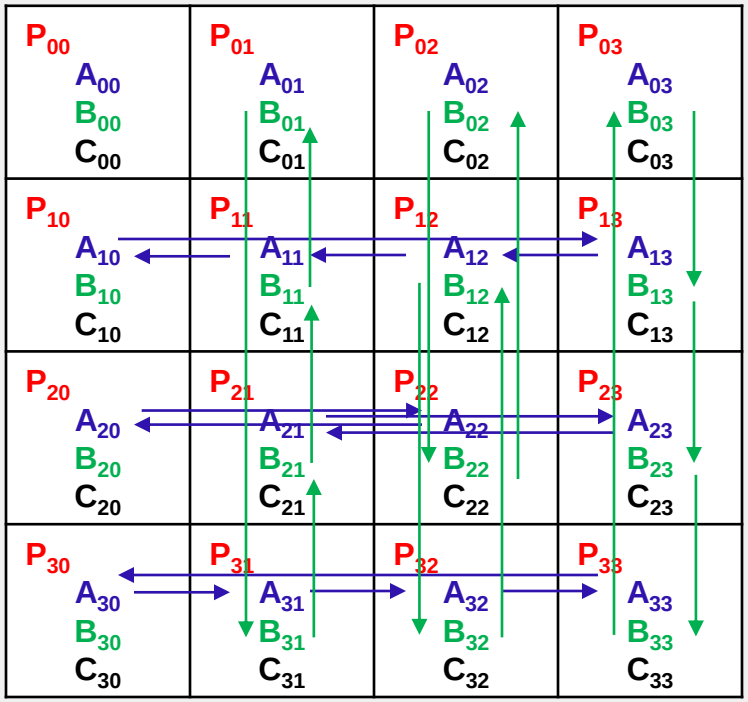
- 如果进程拓扑是笛卡尔结构，并且其中每个进程调用函数MPI_Cart_shift，那么此函数将根据用户说明的坐标方向和步长（正数或负数），通过调用进程返回其相邻的前后/上下两个进程号。
- 将这些相邻进程号传送给MPI_Sendrecv，用于在坐标方向上执行数据轮换。作为输入，MPI_Sendrecv将源进程的标识数作为接受，目标进程的标识数作为发送。

矩阵矩阵乘法
Cannon算法

初始进程$P_{i,j}$分配了子矩阵$A_{i,j}$和$B_{i,j}$。 $C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。



循环向左平移1格

循环向左平移2格

循环向左平移3格

```
int periods[2] ={1,1}, dins[2]={4,4}, ndims=2, cords[2]={0},rowSrc,rowDest,colSrc,colDest;
MPI_Cart_create(oldcomm, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &my_cartrank);
MPI_Cart_coords(comm, my_cartrank, ndims, coords);
if(cords[0]>0){MPI_Cart_shift(comm, 1, -cords[0], &rowSrc, &rowDest);
                MPI_Sendrecv(…,rowDest,…,rowSrc,…);}
if(cords[1]>0){MPI_Cart_shift(comm, 0, -cords[1], &colSrc, &colDest);
                MPI_Sendrecv(…,colDest,…,colSrc,…);}
```

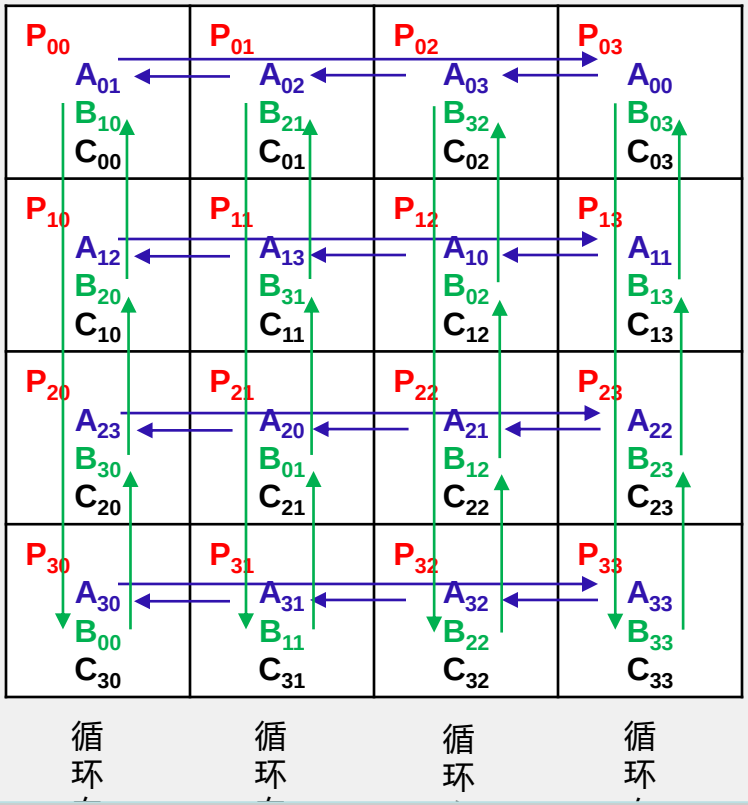
矩阵矩阵乘法

Cannon算法

$C_{i,j}$由$P_{i,j}$计算。

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

C_{ij}的计算要用到A的第i行全部子块，B的第j列全部子块。




```
MPI_Cart_shift(comm, 1, -1, &rowSrc, &rowDest);    //循环向左平移1格
MPI_Sendrecv(…,rowDest,…,rowSrc,…);
MPI_Cart_shift(comm, 0, -1, &colSrc, &colDest);     //循环向上平移1格
MPI_Sendrecv(…,colDest,…,colSrc,…);
```

第三步与第二步一样：将A的子块按行左移一格，同时将B的子块按列上移一格

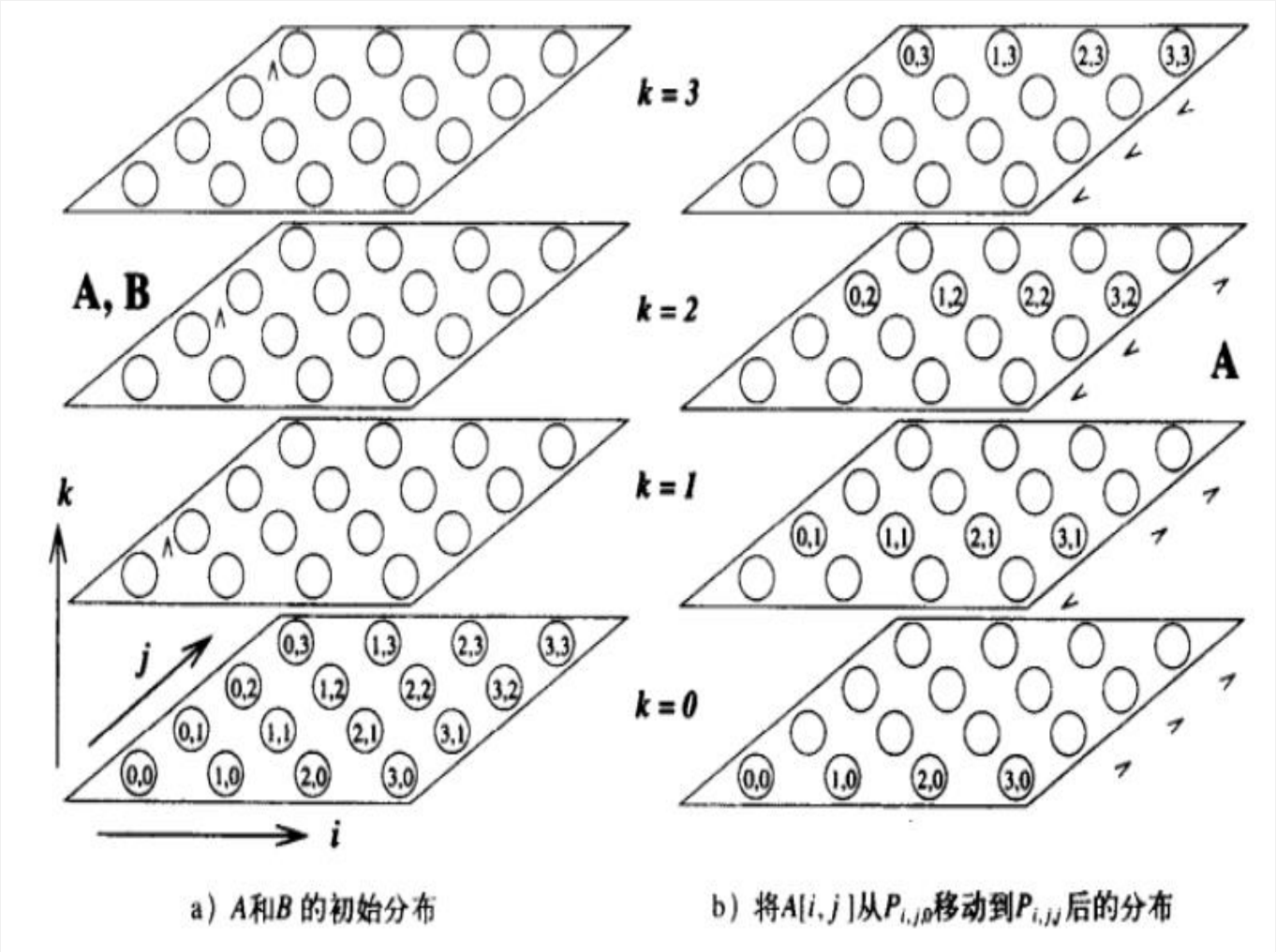
矩阵矩阵乘法
DNS算法

- 使用三维划分。
- 我们知道结果矩阵C中的每个元素实际上是由A的行和B的列进行一个点乘得到的。把点乘操作的多次乘法运算展开成第三维。
- 因此，在进程立方中每个进程只负责了一次乘加操作。而二维算法中负责一次点乘操作。
- DNS算法最多允许n^3个进程参与计算。

矩阵矩阵乘法
DNS算法

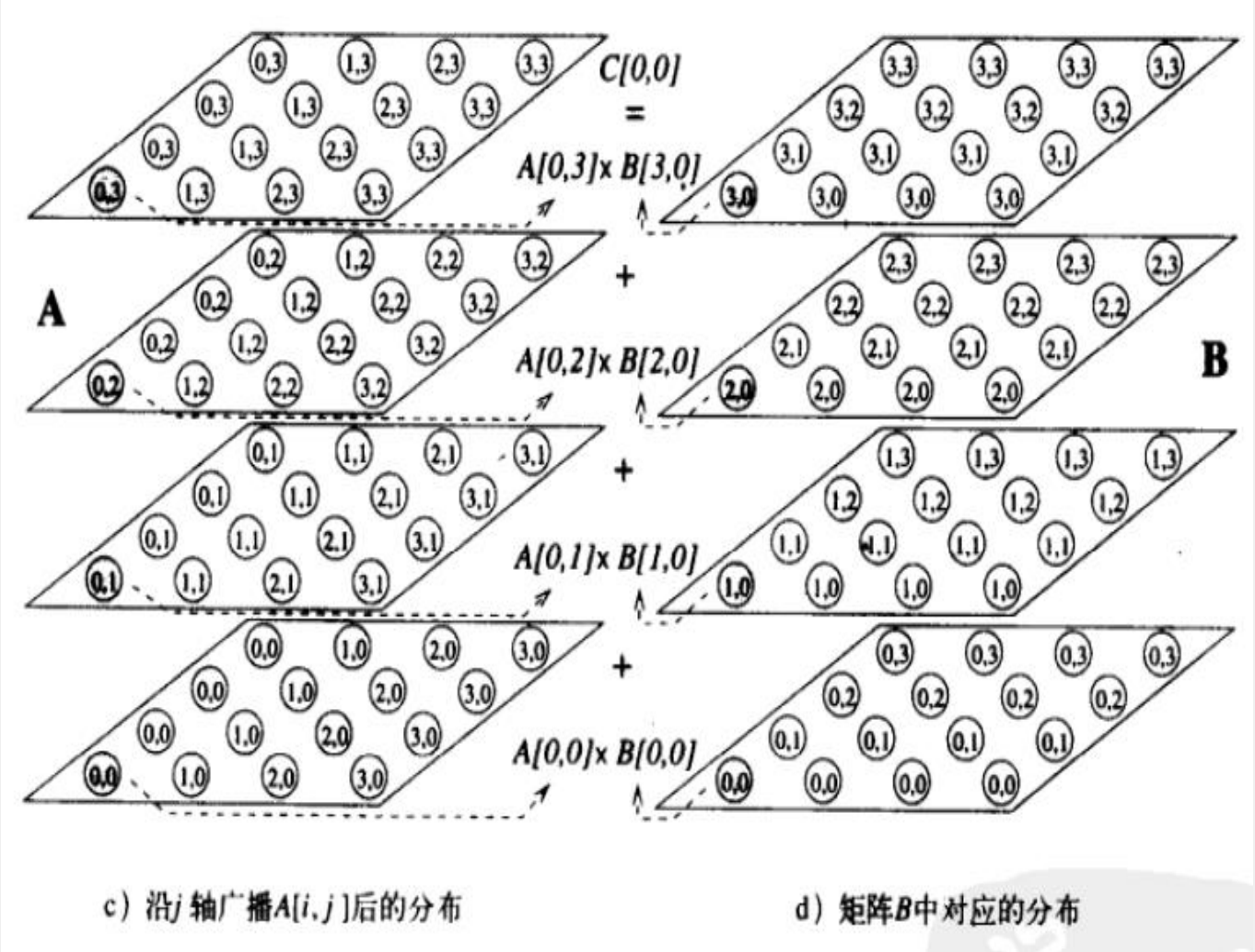
- 假设有一个$n \times n \times n$的进程阵列。
- 将A的第j列发送到第j层的某一列进程中，将B的第i行发送到第i层的某一行进程中，然后对同层进行广播。
- 每个处理器计算一个单独的乘加操作。
- 沿着第三维做加法归约，算出具体对应的矩阵C的每一个元素。
- 一个乘加操作运行时间是一个常数，而归约和广播复杂度为$\Theta(\log n)$，因此总体需要的并行运行时间为$\Theta(\log n)$。
- 使用n^3个进程是并不是成本最优的。

矩阵矩阵乘法
DNS算法



在64个进程上的4x4矩阵乘法的DNS算法，c中带阴影的进程储存A的第一行元素，d中带阴影的进程储存B的第一列元素。

矩阵矩阵乘法
DNS算法



在64个进程上的4x4矩阵乘法的DNS算法，c中带阴影的进程储存A的第一行元素，d中带阴影的进程储存B的第一列元素。

矩阵矩阵乘法
DNS算法的可扩展性分析

- 假设进程的数量是完全立方数$p=q^3$。
- 两个输入矩阵被划分成(n/q)x(n/q)的块。
- 因此矩阵可以被看做是由这些块排成的qxq二维方阵。
- 算法和前面所述是完全类似的，唯一不同的是每个进程现在对子块操作而不是对矩阵的单个元素操作。

矩阵矩阵乘法

DNS算法的可扩展性分析

- 第一步*A*和*B*都执行一对一通信，消息长度是$(n/q)^2$的块，每一次用时 $t_s + t_w(n/q)^2$。
- 两个广播和归约每个用时 $t_s \log q + t_w(n/q)^2 \log q$。
- 进程内的乘法用时$(n/q)^3$。
- 因此，总的并行时间约为

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

- 等效率函数为$\Theta(p \log^3 p)$，当$p=O((n/\log n)^3)$时，是成本最优的。