



中科院计算所
INSTITUTE OF COMPUTING TECHNOLOGY, CAS

并行算法设计与案例分析

谭光明 博士

tgm@ncic.ac.cn

中国科学院计算技术研究所

国家智能计算机研究开发中心

计算机体系结构国家重点实验室

- 并行算法的一般设计过程
- 并行数值算法
 - 矩阵乘法
 - PDE



并行算法的一般设计过程

- 设计并行算法的四个阶段
 - 划分 Partitioning
 - 通讯 Communication
 - 组合 Agglomeration
 - 映射 Mapping



PCAM设计方法学

- 划分：分解成小的任务，开拓并发性
- 通讯：确定诸任务间的数据交换，监测划分的合理性
- 组合：依据任务的局部性，组合成更大的任务
- 映射：将每个任务分配到处理器上，提高算法的性能。



划分

- 方法描述
- 域分解
- 功能分解
- 划分判据



划分方法的描述

- 划分：原计算问题分割成一些小的计算任务，以充分揭示并行执行的机会
- 使数据集和计算集互补相交，力图避免数据和计算的复制。
- 划分阶段忽略处理器数目和目标机器的体系结构；
- 域分解：集中数据的分解
- 功能分解：计算功能的分解



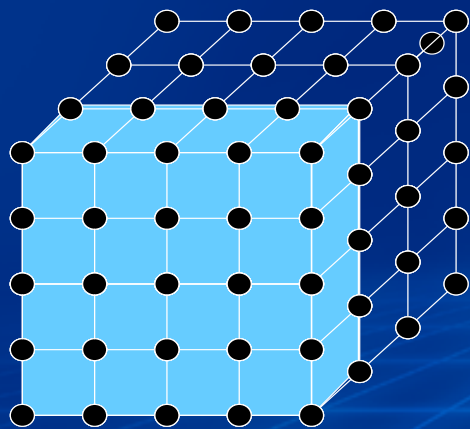
域分解（数据划分）

- 划分的对象是数据，可以是算法的输入数据、中间处理数据和输出数据；
- 将数据分解成大致相等的小数据片；
- 划分时考虑数据上的相应操作；
- 如果一个任务需要别的任务中的数据，则会产生任务间的通讯；

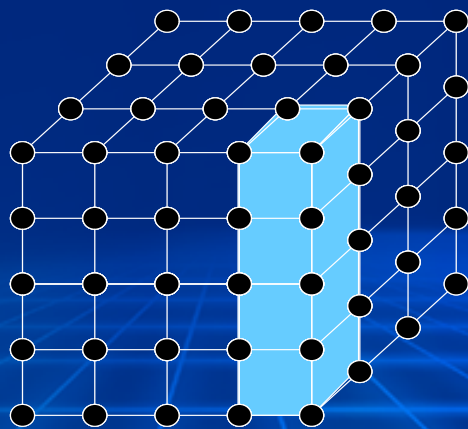


域分解

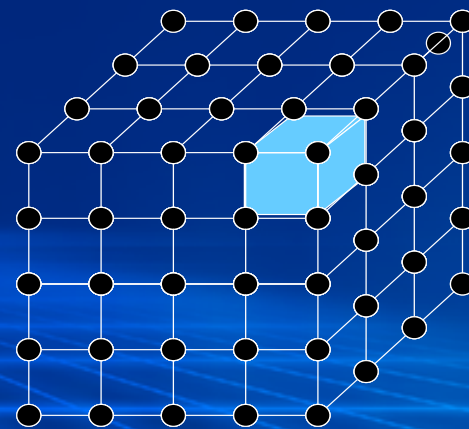
- 示例：三维网格的域分解，各格点上计算都是重复的。下图是三种分解方法：



1-D



2-D

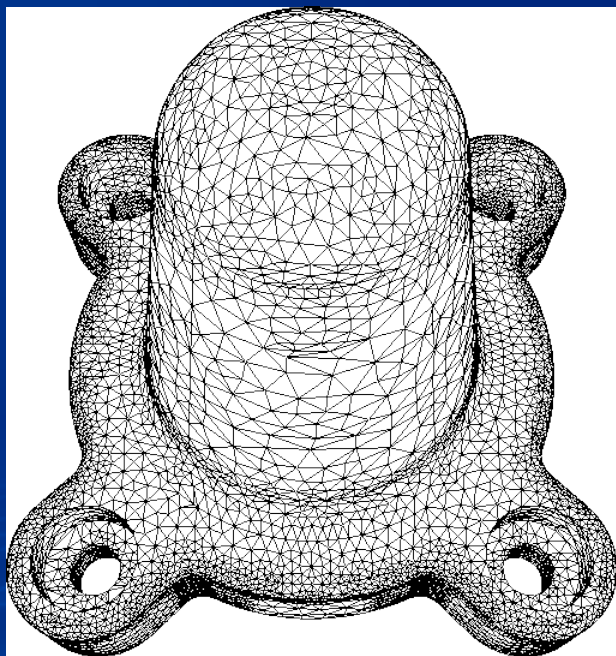


3-D



域分解

- 不规则区域的分解示例：



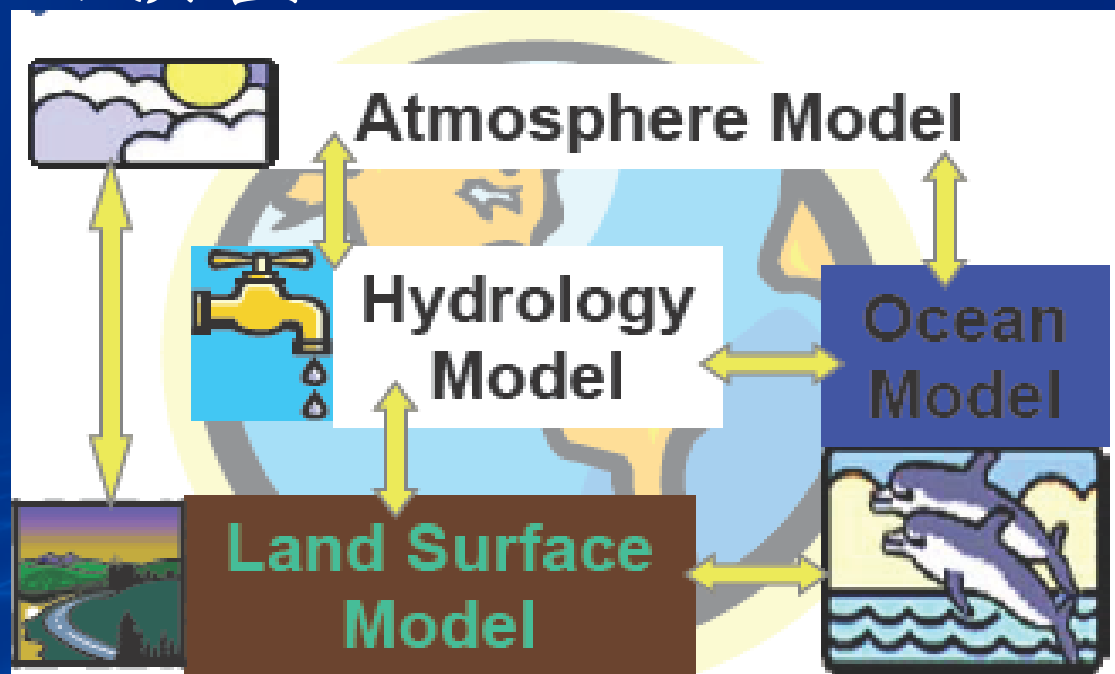
功能分解（计算划分）

- 划分的对象是计算，将计算划分为不同的任务，其出发点不同于域分解；
- 划分后，研究不同任务所需的数据。如果这些数据不相交的，则划分是成功的；如果数据有相当的重叠，意味着要重新进行域分解和功能分解；
- 功能分解是一种更深层次的分解。



功能分解

- 气候模型



划分判据

- 划分是否具有灵活性？
- 划分是否避免了冗余计算和存储？
- 划分任务尺寸是否大致相当？
- 任务数与问题尺寸是否成比例？
- 功能分解是一种更深层次的分解，是否合理？



PCAM设计方法学

- 划分：分解成小的任务，开拓并发性
- 通讯：确定诸任务间的数据交换，监测划分的合理性
- 组合：依据任务的局部性，组合成更大的任务
- 映射：将每个任务分配到处理器上，提高算法的性能。



通讯

- 方法描述
- 四种通讯模式
- 通讯判据



通讯方法描述

- 通讯是PCAM设计过程的重要阶段
- 划分产生的诸任务，一般不能完全独立执行，需要在任务间进行数据交流；从而产生了通讯；
- 功能分解确定了诸任务之间的数据流
- 诸任务是并发执行的，通讯则限制了这种并发性



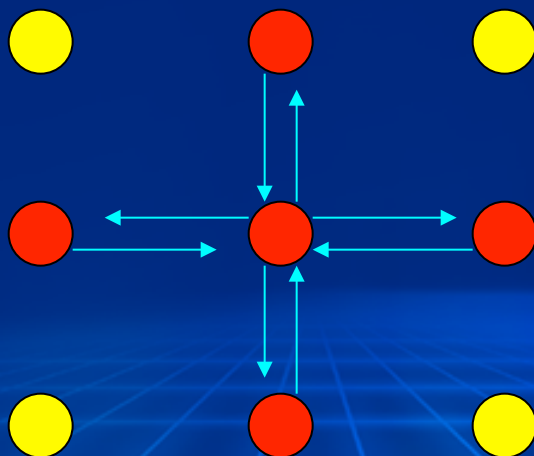
四种通讯模式

- 局部/全局通讯
- 结构化/非结构化通讯
- 静态/动态通讯
- 同步/异步通讯



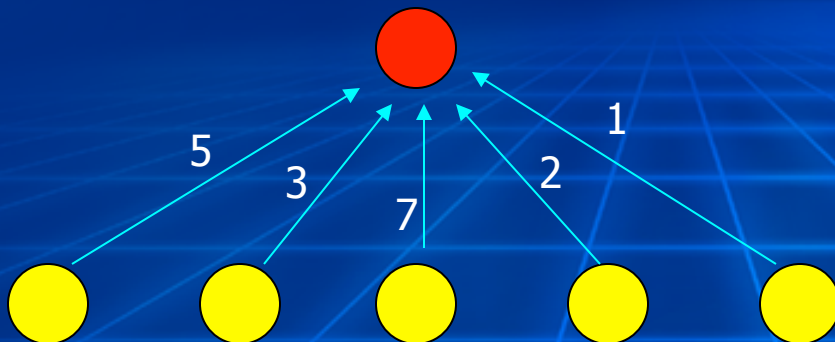
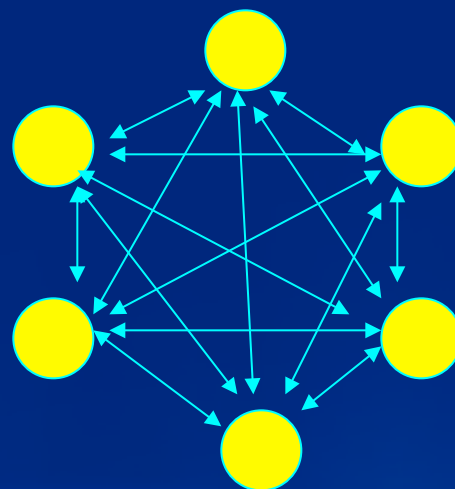
局部通讯

- 通讯限制在一个邻域内



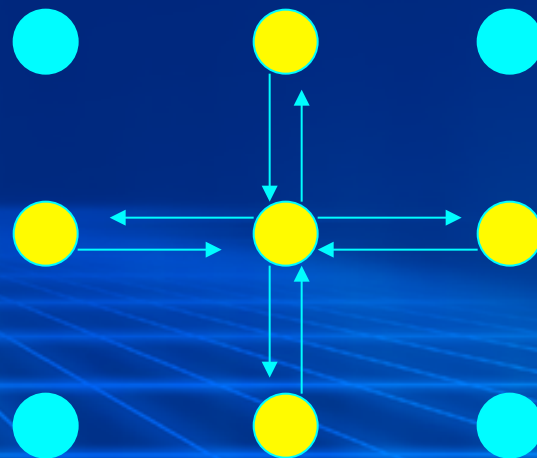
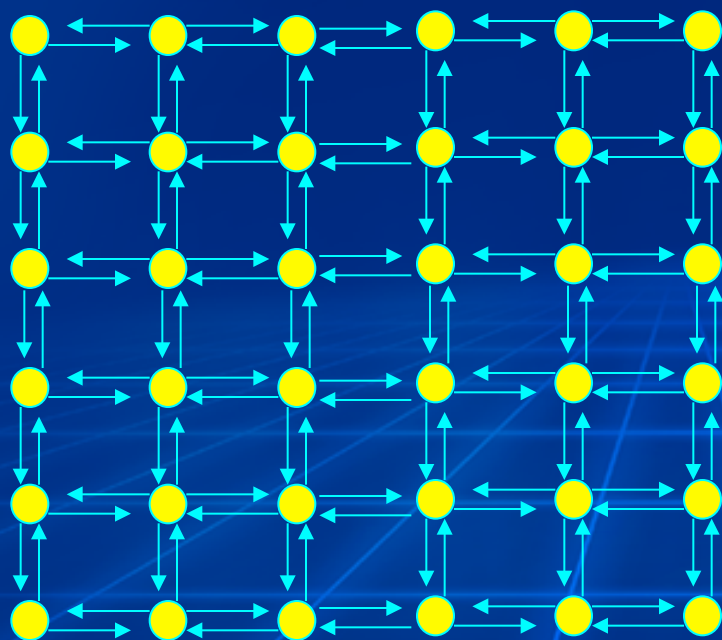
全局通讯

- 通讯非局部的
- 例如：
 - All to All
 - Master-Worker



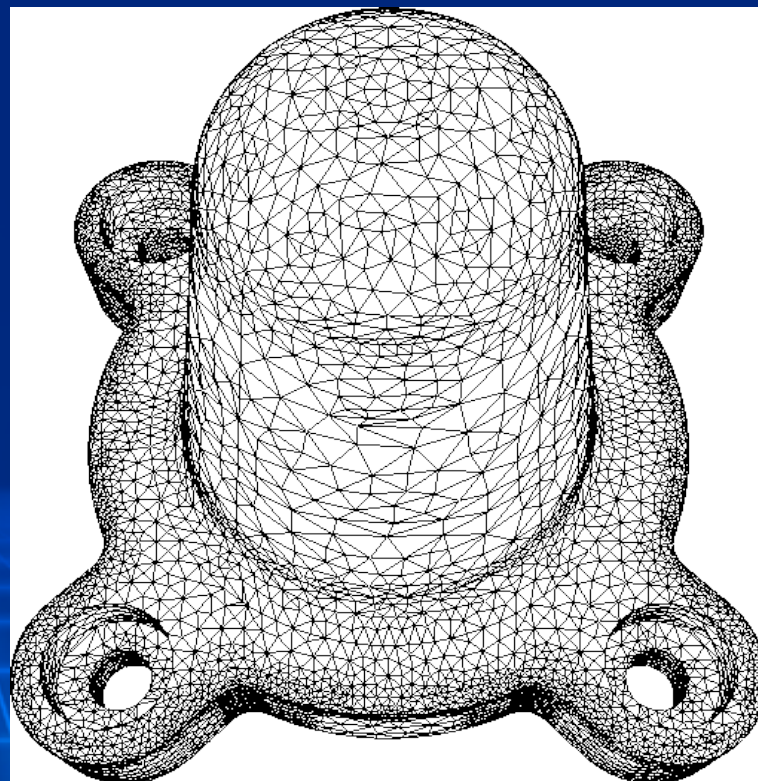
结构化通讯

- 每个任务的通讯模式是相同的；
- 下面是否存在一个相同通讯模式？



非结构化通讯

- 没有一个统一的通讯模式
- 例如：无结构化网格



通讯判据

- 所有任务是否执行大致相当的通讯?
- 是否尽可能的局部通讯?
- 通讯操作是否能并行执行?
- 同步任务的计算能否并行执行?



PCAM设计方法学

- 划分：分解成小的任务，开拓并发性
- 通讯：确定诸任务间的数据交换，监测划分的合理性
- 组合：依据任务的局部性，组合成更大的任务
- 映射：将每个任务分配到处理器上，提高算法的性能。



组合

- 方法描述
- 增加粒度
 - 表面—容积效应
 - 重复计算
- 组合判据



方法描述

- 组合是由抽象到具体的过程，是将组合的任务能在一类并行机上有效的执行
- 合并小尺寸任务，减少任务数。如果任务数恰好等于处理器数，则也完成了映射过程
- 通过增加任务的粒度和重复计算，可以减少通讯成本
- 保持映射和扩展的灵活性，降低软件工程成本



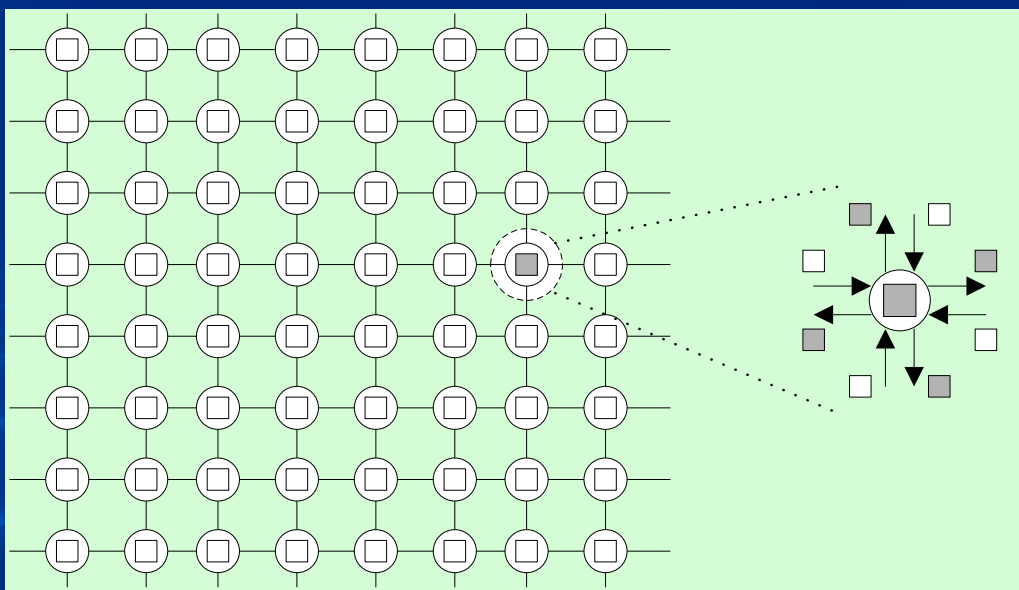
增加粒度

- 在设计过程的划分阶段，致力于定义尽可能多的任务以增大并行执行的机会。
- 大量的细粒度任务不一定能产生一个有效的并行算法。
- 大量细粒度任务有可能增加通信代价和任务创建代价



表面-容积效应

- 表面-容积效应：通讯量与任务子集的表面成正比，计算量与任务子集的体积成正比
 - 增大划分粒度有可能减少通讯量
 - 增加重复计算有可能减少通讯量



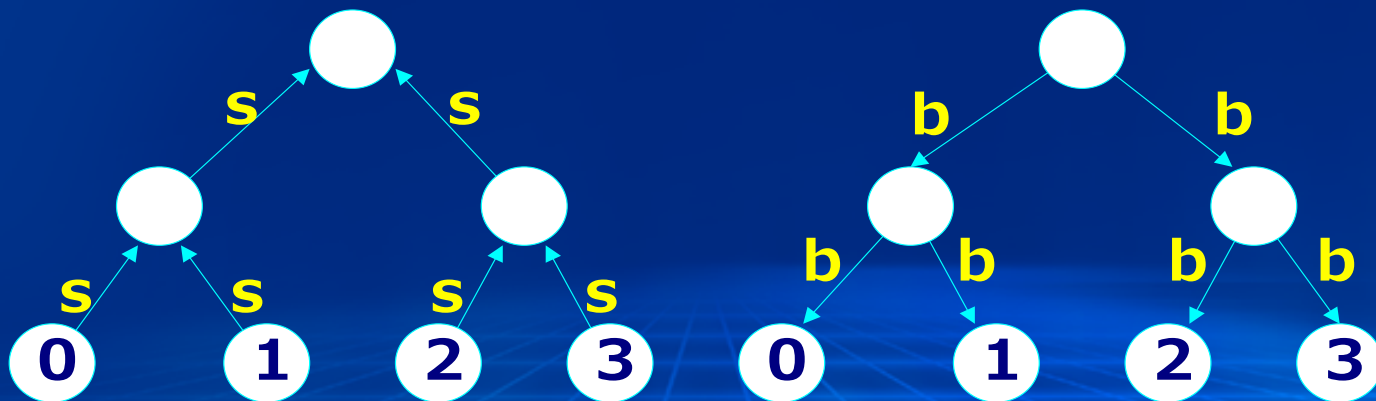
重复计算

- 重复计算减少通讯量，但增加了计算量，应保持恰当的平衡；
- 重复计算的目标应减少算法的总运算时间



重复计算

- 示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和。

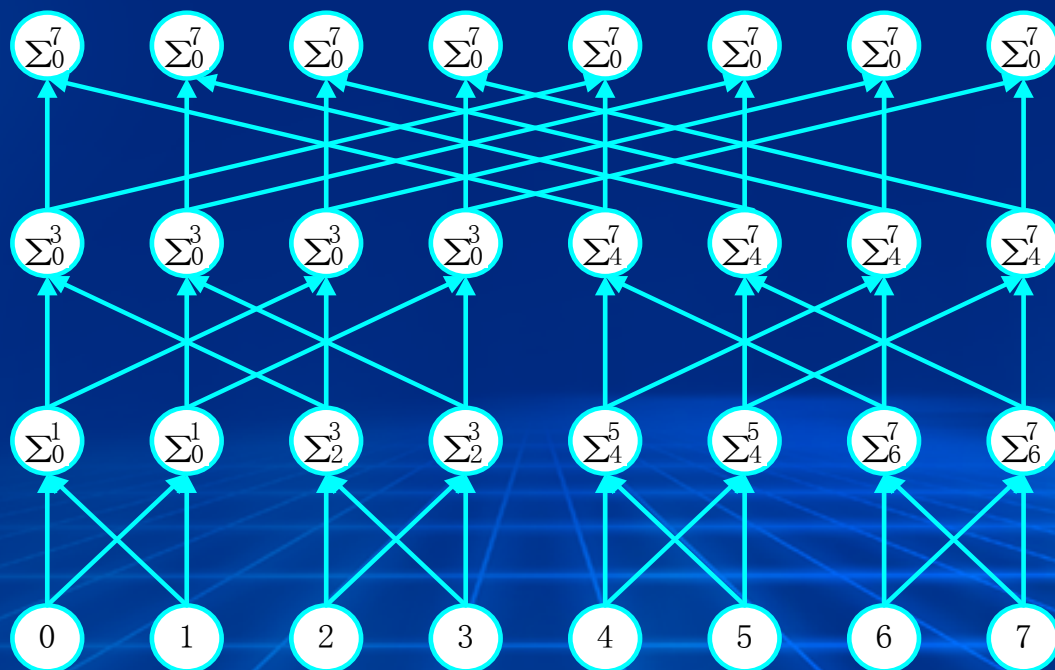


二叉树上求和，共需 $2\log N$ 步



重复计算

- 示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和。



蝶式结构求和，使用了重复计算，共需 $\log N$ 步



组合判据

- 增加粒度是否减少了通讯成本？
- 重复计算是否已权衡了其得益？
- 是否保持了灵活性和可扩展性？
- 组合的任务数是否与问题尺寸成比例？
- 是否保持了类似的计算和通讯？
- 有没有减少并行执行的机会？



PCAM设计方法学

- 划分：分解成小的任务，开拓并发性
- 通讯：确定诸任务间的数据交换，监测划分的合理性
- 组合：依据任务的局部性，组合成更大的任务
- 映射：将每个任务分配到处理器上，提高算法的性能。



映射

- 方法描述
- 负载平衡算法
- 任务调度算法
- 映射判据



方法描述

- 每个任务要映射到具体的处理器，定位到运行机器上；
- 任务数大于处理器数时，存在负载平衡和任务调度问题；
- 映射的目标：减少算法的执行时间
 - 并发的任务 → 不同的处理器
 - 任务之间存在高通讯的 → 同一处理器
- 映射实际是一种权衡，属于NP完全问题；



负载均衡算法

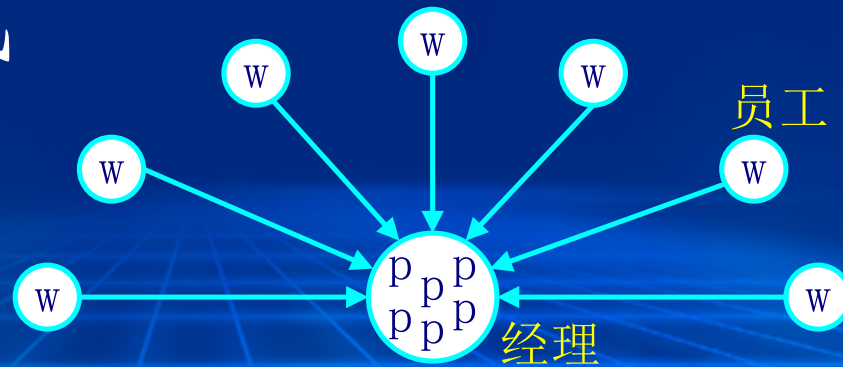
- 静态的：事先确定；
- 概率的：随机确定；
- 动态的：执行期间动态负载；
- 基于域分解的：
 - 递归对剖
 - 局部算法
 - 概率方法
 - 循环映射



任务调度算法

- 任务放在集中的或分散的任务池中，使用任务调度算法将池中的任务分配给特定的处理器。下面是两种常用调度模式：

- 经理/雇员模式



- 非集中模式

映射判据

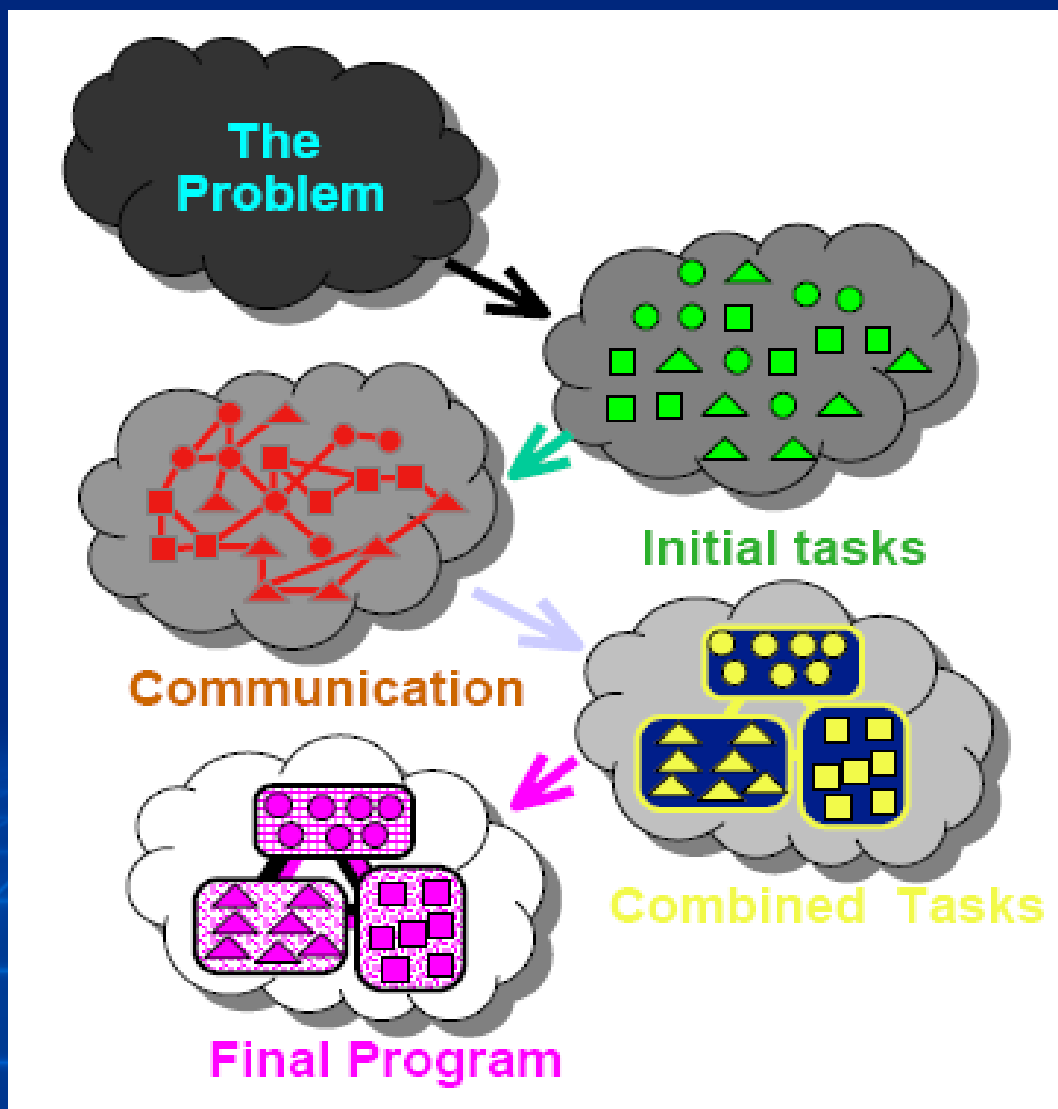
- 采用集中式负载均衡方案，是否存在通讯瓶颈？
- 采用动态负载均衡方案，调度策略的成本如何？



PCAM

设计过程

- **划分**: 域分解和功能分解
- **通讯**: 任务间的数据交换
- **组合**: 任务的合并使得算法更有效
- **映射**: 将任务分配到处理器, 并保持负载平衡



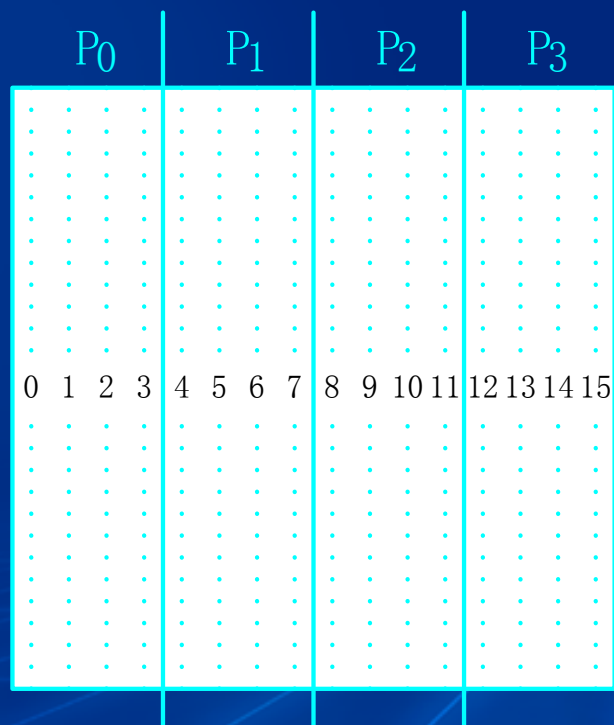
矩阵的划分

- 矩阵运算是数值计算中最重要、最基本的一类运算。
- 科学与工程计算中，矩阵的阶都非常高。为了实现并行，需要将其划分，指派到不同的处理器上。
- 常见的两种划分方法：
 - 带状划分和棋盘划分

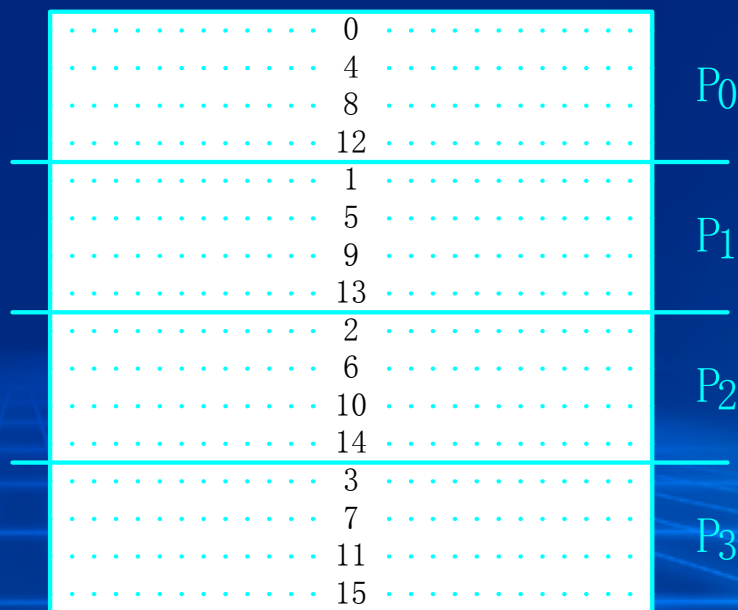


矩阵的带状划分

- 16×16 阶矩阵, $p=4$



列块带状划分

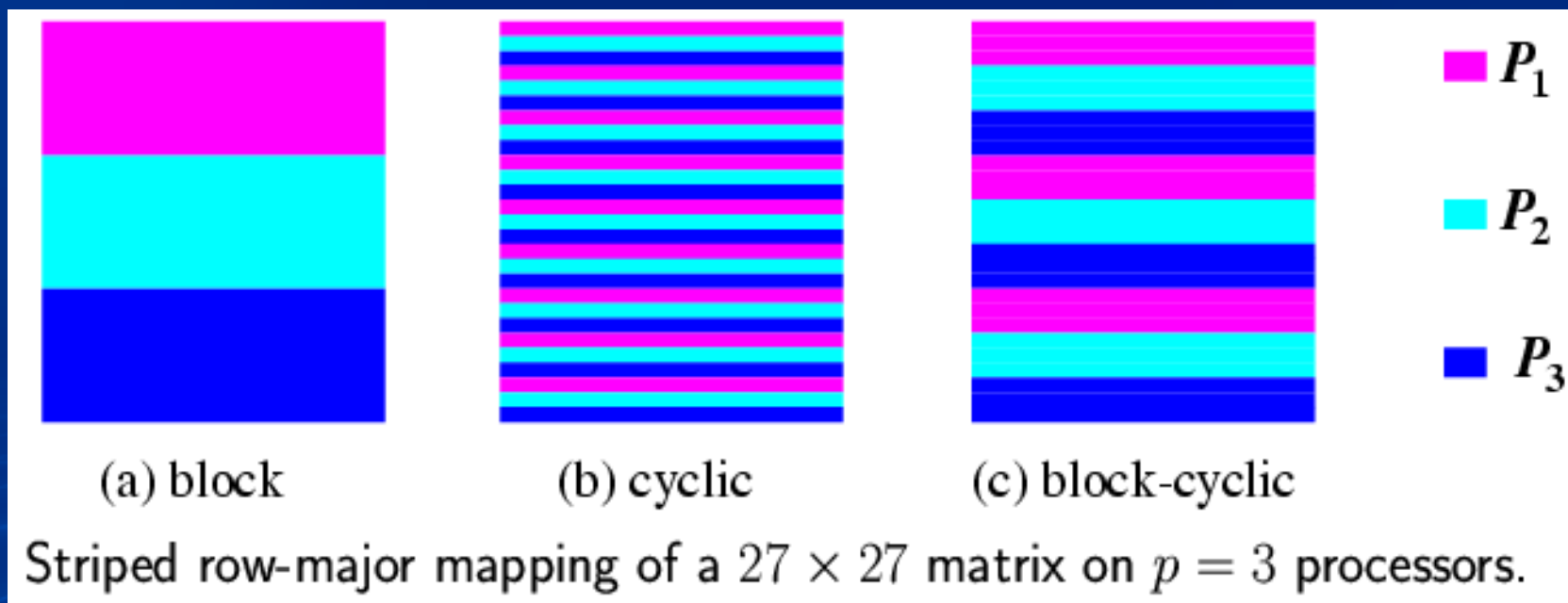


行循环带状划分



矩阵的带状划分

- 示例： $p = 3$ ， 27×27 矩阵的3种带状划分



矩阵的棋盘划分

- 8×8 阶矩阵, $p=16$

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)	(0, 7)
P₀		P₁		P₂		P₃	
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)	(1, 7)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)	(2, 7)
P₄		P₅		P₆		P₇	
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)	(3, 7)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)	(4, 7)
P₈		P₉		P₁₀		P₁₁	
(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)	(5, 7)
(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)	(6, 7)
P₁₂		P₁₃		P₁₄		P₁₅	
(7, 0)	(7, 1)	(7, 2)	(7, 3)	(7, 4)	(7, 5)	(7, 6)	(7, 7)

块棋盘划分

(0, 0)	(0, 4)	(0, 1)	(0, 5)	(0, 2)	(0, 6)	(0, 3)	(0, 7)
P₀		P₁		P₂		P₃	
(4, 0)	(4, 4)	(4, 1)	(4, 5)	(4, 2)	(4, 6)	(4, 3)	(4, 7)
(1, 0)	(1, 4)	(1, 1)	(1, 5)	(1, 2)	(1, 6)	(1, 3)	(1, 7)
P₄		P₅		P₆		P₇	
(5, 0)	(5, 4)	(5, 1)	(5, 5)	(5, 2)	(5, 6)	(5, 3)	(5, 7)
(2, 0)	(2, 4)	(2, 1)	(2, 5)	(2, 2)	(2, 6)	(2, 3)	(2, 7)
P₈		P₉		P₁₀		P₁₁	
(6, 0)	(6, 4)	(6, 1)	(6, 5)	(6, 2)	(6, 6)	(6, 3)	(6, 7)
(3, 0)	(3, 4)	(3, 1)	(3, 5)	(3, 2)	(3, 6)	(3, 3)	(3, 7)
P₁₂		P₁₃		P₁₄		P₁₅	
(7, 0)	(7, 4)	(7, 1)	(7, 5)	(7, 2)	(7, 6)	(7, 3)	(7, 7)

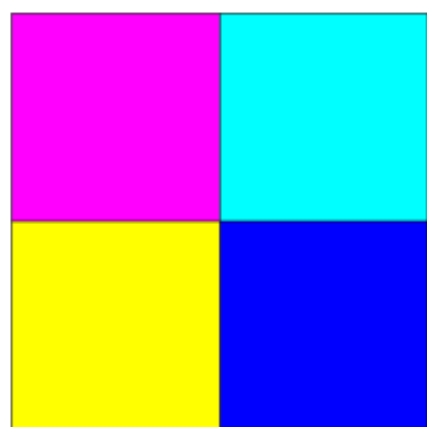
循环棋盘划分



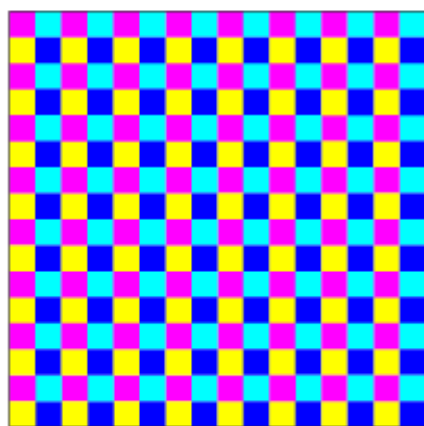
中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

矩阵的棋盘划分

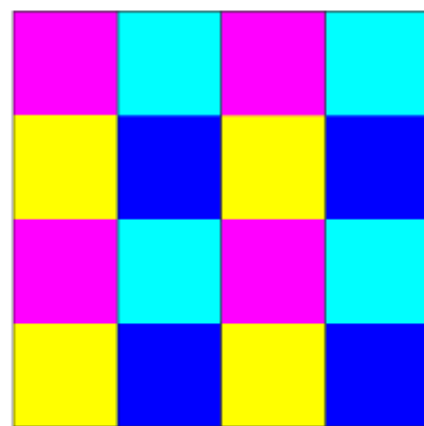
- 示例: $p = 4$, 16×16 矩阵的3种棋盘划分



(a) block



(b) cyclic



(c) block cyclic

■ P_1
■ P_2
■ P_3
■ P_4

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



矩阵乘法

- 行块分解的串行矩阵乘法
- 简单分块并行矩阵乘法
- Cannon矩阵乘法(棋盘划分)



矩阵乘法符号及定义

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

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

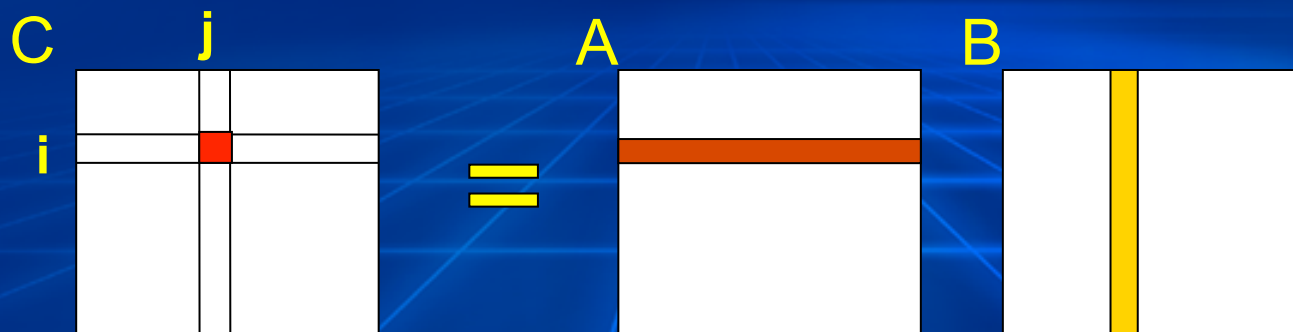
$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}$$



矩阵乘法符号及定义

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

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



基于行的迭代算法

Input: $A_{l \times m}$, $B_{m \times n}$

Output: $C_{l \times n}$

Begin

for ($i = 0; i < l-1; i++$) {

for ($j = 0; j < n-1; j++$) {

$c_{ij} = 0;$

for ($k = 0; k < m-1; k++$) {

$c_{ij} = c_{ij} + a_{ik} * b_{kj};$

}

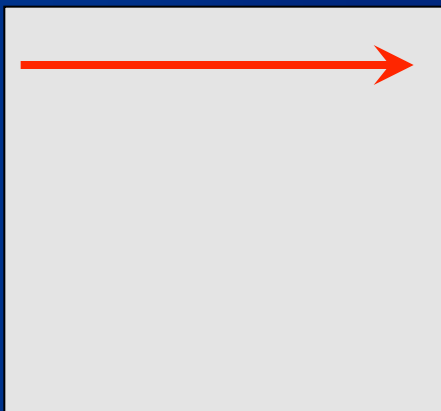
}

End

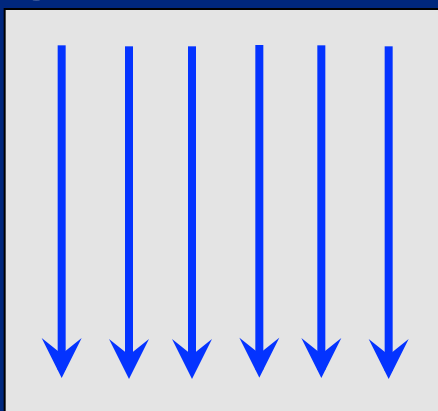


基于行的迭代算法

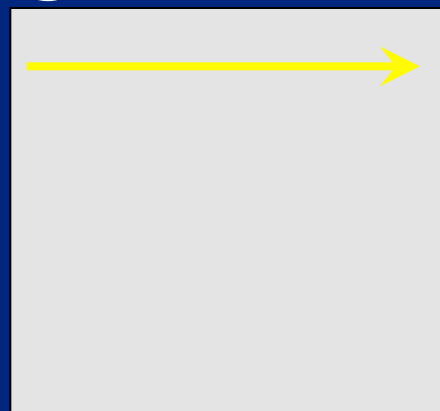
A



B



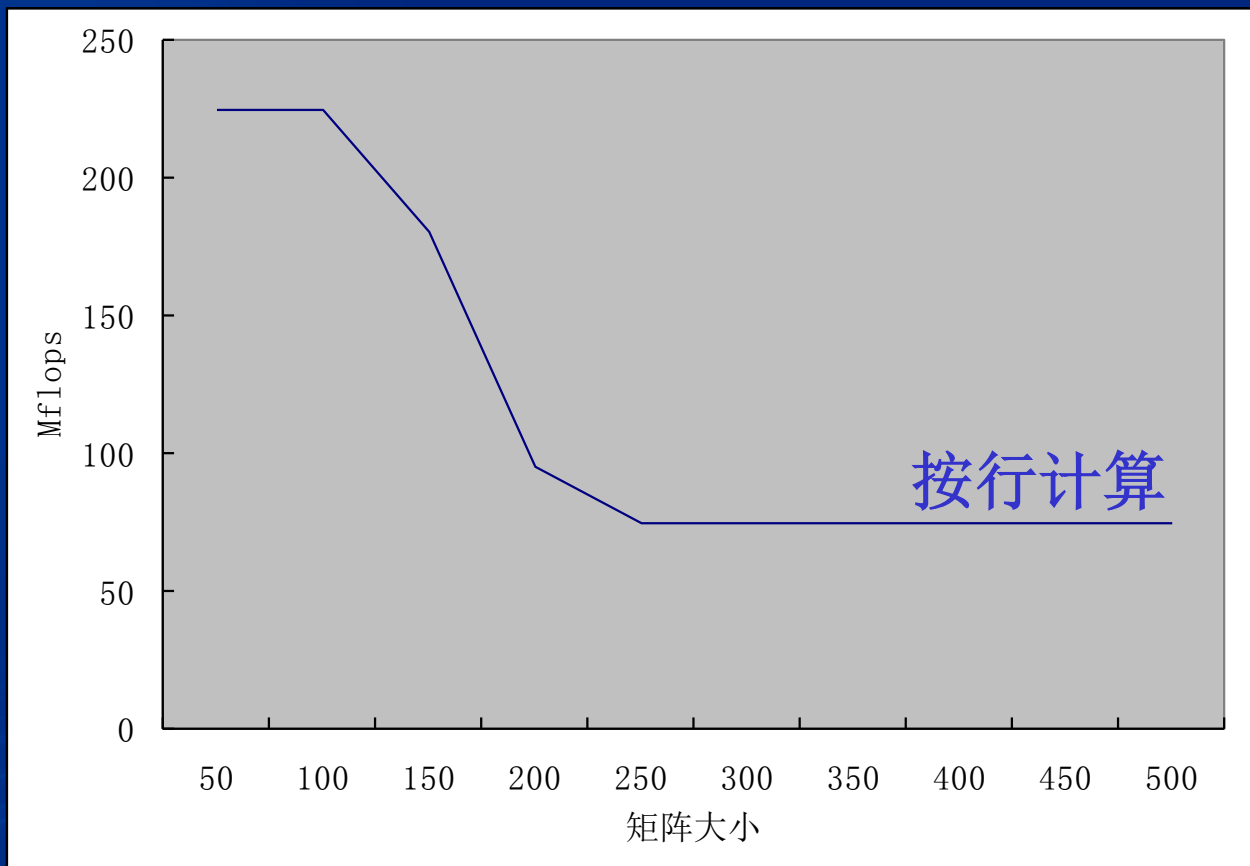
C



每次外层迭代都要读取B的所有元素。对Cache而言，若B太大，后面读入的元素就会覆盖先前已经读入Cache中的元素。这样进行下一次的外层迭代时，需要将B全部重新读入。当矩阵规模达到某个阈值时，Cache命中率急剧下降，从而性能急剧下降。



基于行的迭代算法



Cache: 256K
双精度浮点数

Cache最多可放下32,768 ($\sim 181^2$) 个数, 当 $n \leq 150$ 时, cache命中率远高于 $n > 200$ 时的命中率



基于块的递归算法

- 假设 $A_{l \times m}$, $B_{m \times n}$ 。把 A 分成 4 个小矩阵：

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

同时把 B 分成 4 个小矩阵： $B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$

使得 A_{00} 和 A_{10} 的行数分别等于 B_{00} 和 B_{01} 的列数，那么矩阵的积为：

$$C = \begin{pmatrix} A_{00} * B_{00} + A_{01} * B_{10} & A_{00} * B_{01} + A_{01} * B_{11} \\ A_{10} * B_{00} + A_{11} * B_{10} & A_{10} * B_{01} + A_{11} * B_{11} \end{pmatrix}$$



基于块的递归算法

Input: $A_{n \times n}$, $B_{n \times n}$, 子块大小为 $(n/q) \times (n/q)$

Output: $C_{n \times n}$

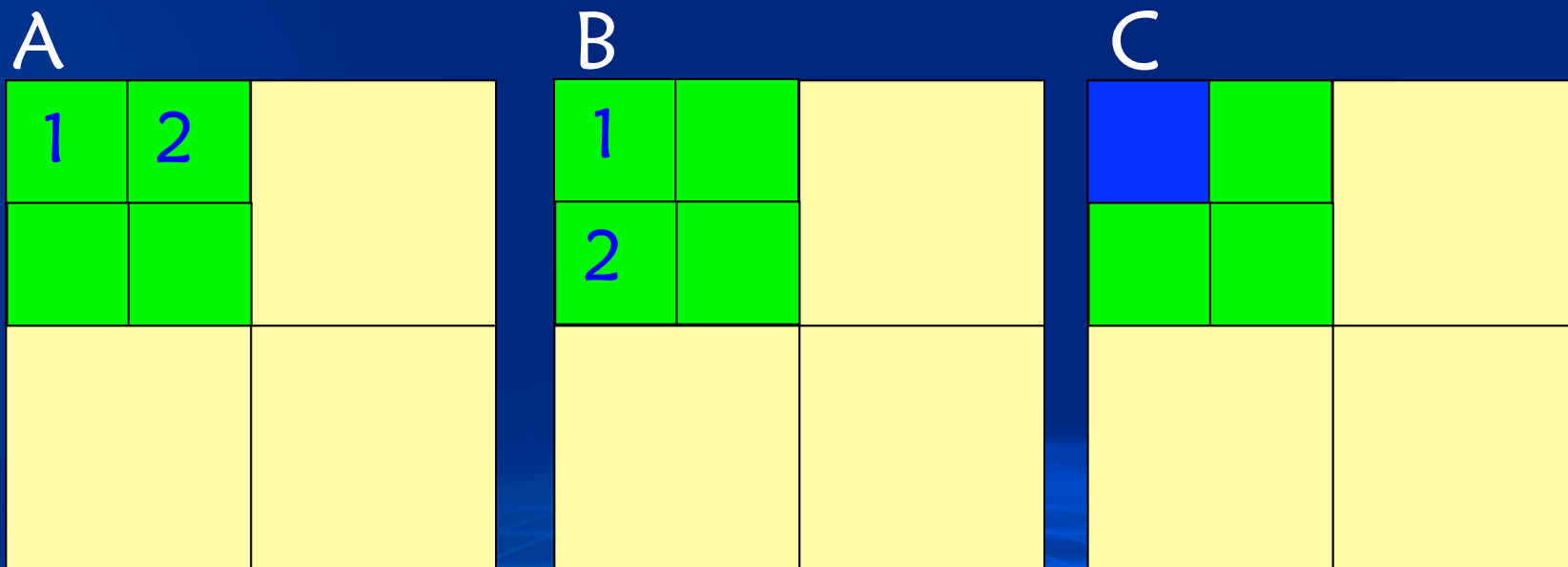
Begin

```
for ( $i = 0; i < q-1; i++$ ) {  
  for ( $j = 0; j < q-1; j++$ ) {  
     $C_{ij} = 0;$   
    for ( $k = 0; k < m-1; k++$ ) {  
       $C_{ij} = C_{ij} + A_{ik} * B_{kj};$   
    }  
  }  
}
```

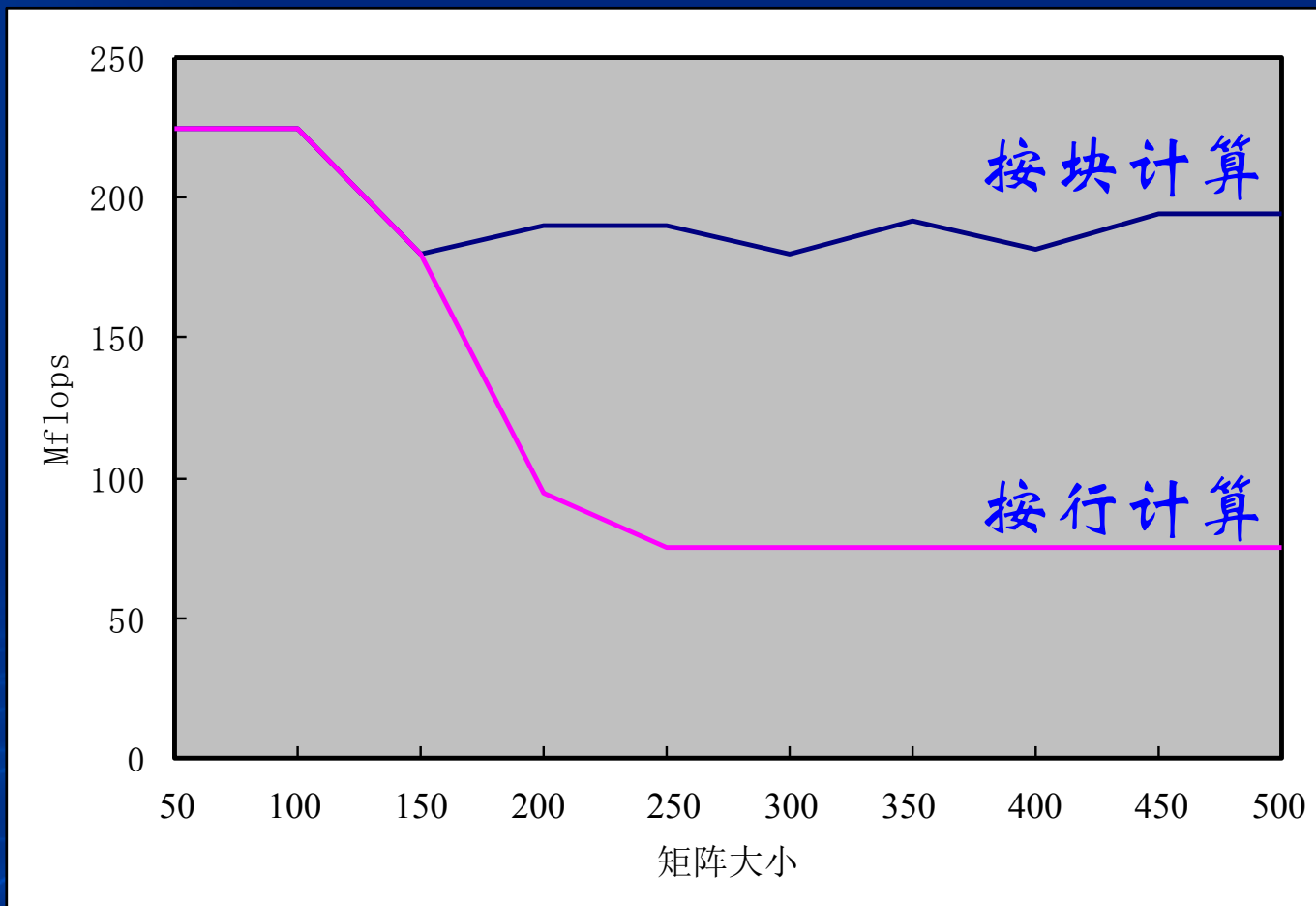
End



基于块的递归算法



基于行迭代和块递归的比较



矩阵乘法

- 行块分解的串行矩阵乘法
- 简单分块并行矩阵乘法
- Cannon矩阵乘法(棋盘划分)



简单分块并行乘法

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

① 通讯: 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 $C_{ij} = \sum_{k=0}^{\sqrt{p}-1} A_{ik} \cdot B_{kj}$



分块

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

通讯

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

$$P_{00}: C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}$$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$
$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$

计算

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



简单并行分块乘法

- 简单并行分块乘法的缺点：
 - 存储要求过大
 - 通信结束时每个处理器拥有 $2\sqrt{p}$ 个块，每块大小为 n^2 / p



矩阵乘法

- 行块分解的串行矩阵乘法
- 简单分块并行矩阵乘法
- Cannon矩阵乘法(棋盘划分)
- Fox矩阵乘法
- Systolic矩阵乘法



Cannon乘法(基于棋盘分解)

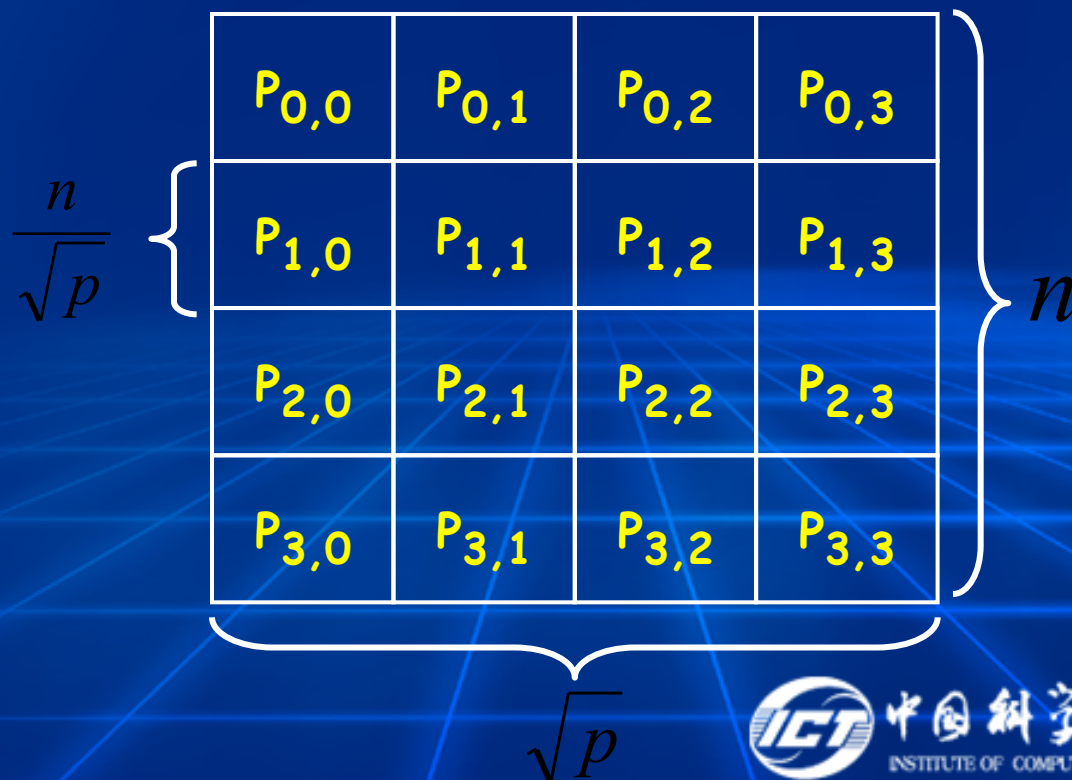
- 算法简介:

- Cannon算法是一种**存储有效**的算法。
- 与并行分块乘法不同，阵列的各行和各列不是施行多到多的广播，而是有目的地在各行和各列实施循环移位，从而降低各处理器的存储量



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 $C_{i,j}$, 大小均为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ p 个处理器编号为 $(P_{0,0}, \dots, P_{0,\sqrt{p}-1}, \dots, P_{\sqrt{p}-1,\sqrt{p}-1})$, $P_{i,j}$ 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 块 $A_{i,j}$ ($0 \leq i, j \leq \sqrt{p}-1$) 向左循环移动 i 步 (按行移位);
块 $B_{i,j}$ ($0 \leq i, j \leq \sqrt{p}-1$) 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;
- ③ A 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: $A_{4 \times 4}$, $B_{4 \times 4}$, $p=16$

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



Cannon乘法

Initial alignment of A Initial alignment of B

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

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

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



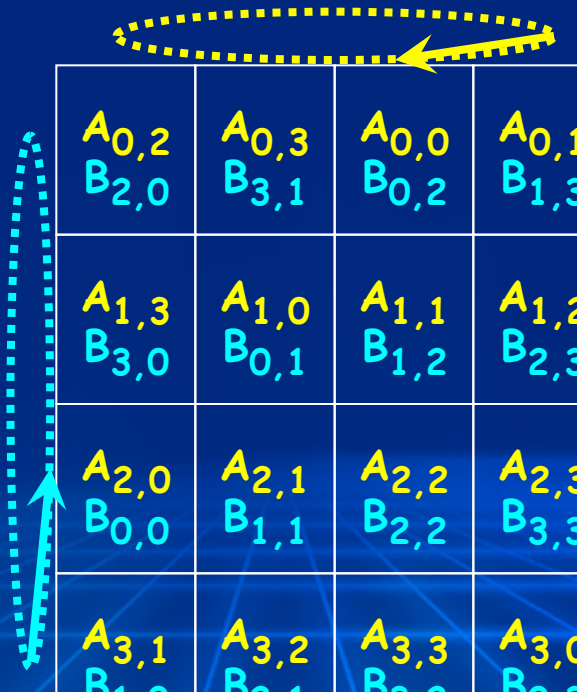
Cannon乘法

After first shift



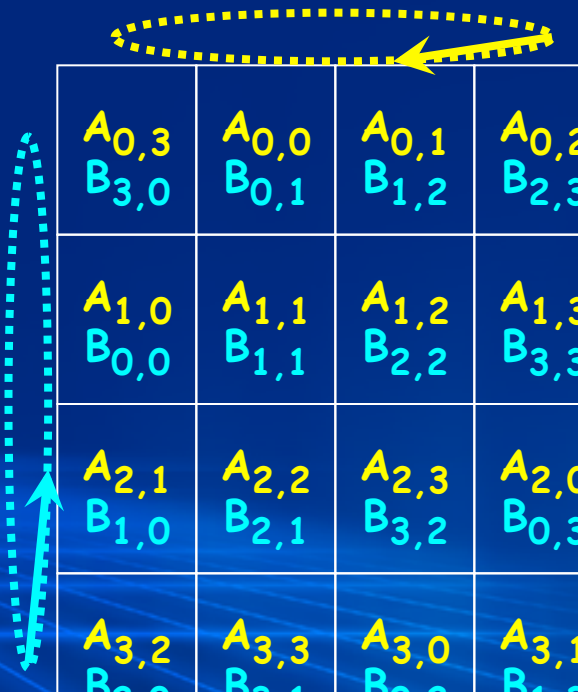
$A_{0,1}$ $B_{1,0}$	$A_{0,2}$ $B_{2,1}$	$A_{0,3}$ $B_{3,2}$	$A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$	$A_{1,3}$ $B_{3,1}$	$A_{1,0}$ $B_{0,2}$	$A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$	$A_{2,0}$ $B_{0,1}$	$A_{2,1}$ $B_{1,2}$	$A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{2,2}$	$A_{3,3}$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

After second shift



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

After third shift



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



Cannon乘法

示例： $P_{1,2}$ 处理器计算 $C_{1,2}$

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

$P_{1,2}$ 的初
始化分块



Cannon乘法： 第一步移位

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

$P_{1,2}$ 计算:

$$A_{1,3} \times B_{3,2}$$



Cannon乘法： 第二步移位

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

$P_{1,2}$ 计算:

$$A_{1,0} \times B_{0,2}$$

$P_{1,2}$ 上的和: $A_{1,0} \times B_{0,2} + A_{1,3} \times B_{3,2}$



Cannon乘法：第三步移位

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

$P_{1,2}$ 计算:

$$A_{1,1} \times B_{1,2}$$

$P_{1,2}$ 上的和: $A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,3} \times B_{3,2}$



Cannon乘法：第四步移位

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

$P_{1,2}$ 计算:

$$A_{1,2} \times B_{2,2}$$

$P_{1,2}$ 上的和: $A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} + A_{1,3} \times B_{3,2}$

Cannon乘法

- 每个进程负载计算大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 的C矩阵的一块
- 每次迭代，每个进程只与其相邻的进程进行点到点通信，不是多到多的广播，算法通信量较小
- 随着矩阵规模和处理器个数的增加，算法的性能保持不变，具有很好的可扩展性
- 计算和通信可以相互重叠



上机作业

- 实现并行矩阵乘法

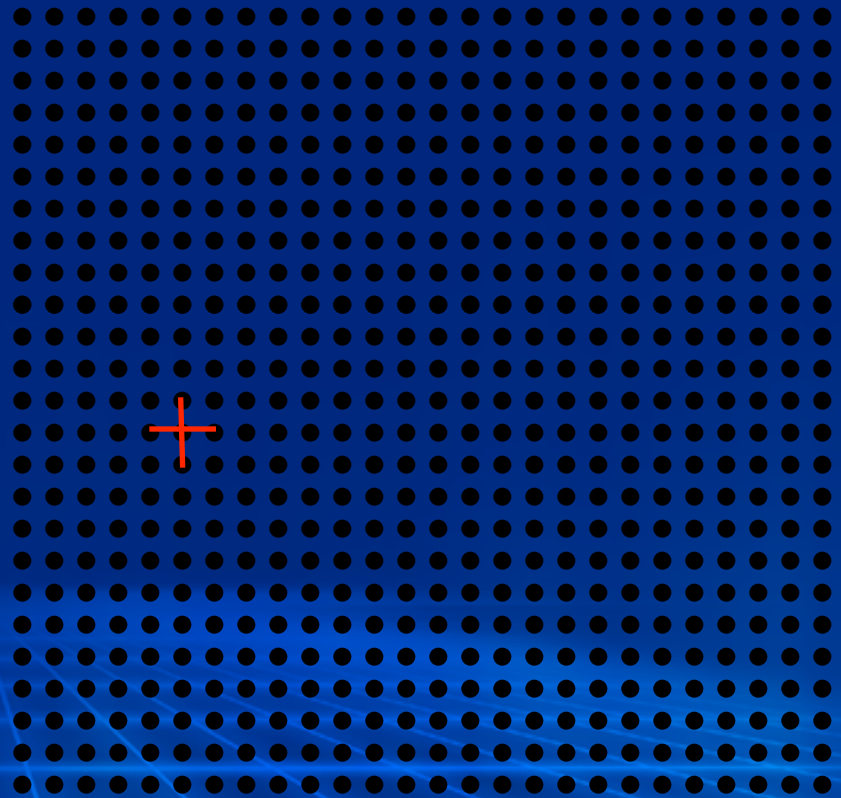


中国科学院计算技术研究所

INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

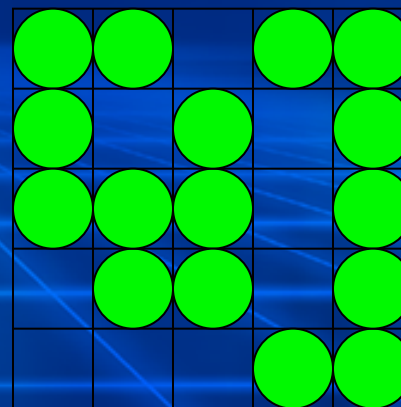
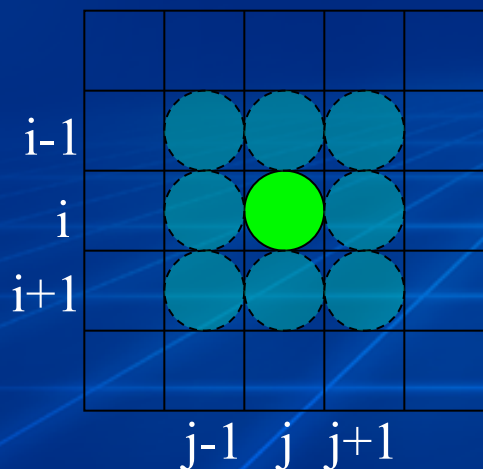
The Mesh

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s *stencil*
- Good numerical algorithms form a matrix equation $Au=f$; solving this requires computing Bv , where B is a matrix derived from A . These evaluations involve computations with the neighbors on the mesh.



Rules for Life

- Matrix values $A(i,j)$ initialized to 1 (live) or 0 (dead)
- In each iteration, $A(i,j)$ is set to
 - 1 (live) if either
 - the sum of the values of its 8 neighbors is 3, or
 - the value was already 1 and the sum of its 8 neighbors is 2 or 3
 - 0 (dead) otherwise



Implementing Life

- For the non-parallel version, we:
 - Allocate a 2D matrix to hold state
 - Actually two matrices, and we will swap them between steps
 - Initialize the matrix
 - Force boundaries to be “dead”
 - Randomly generate states inside
 - At each time step:
 - Calculate each new cell state based on previous cell states (including neighbors)
 - Store new states in second matrix
 - Swap new and old matrices



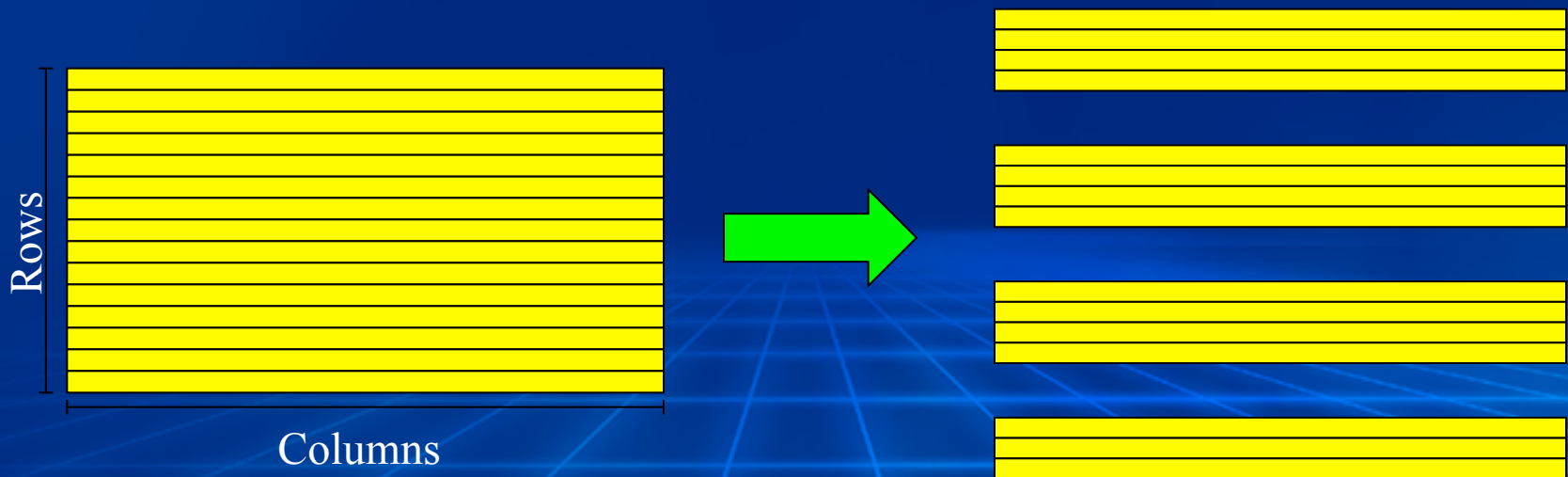
Steps in Designing the Parallel Version

- Start with the “global” array as the main object
 - Natural for output – result we’re computing
- Describe decomposition in terms of global array
- Describe communication of data, still in terms of the global array
- Define the “local” arrays and the communication between them by referring to the global array



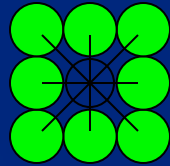
Step 1: Description of Decomposition

- By rows (1D or row-block)
 - Each process gets a group of adjacent rows
- Later we'll show a 2D decomposition

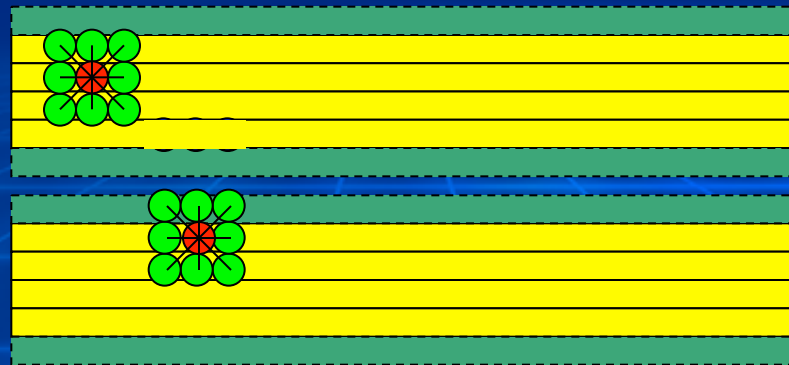


Step 2: Communication

- “Stencil” requires read access to data from neighbor cells



- We allocate extra space on each process to store neighbor cells
- Use send/recv or RMA to update prior to computation



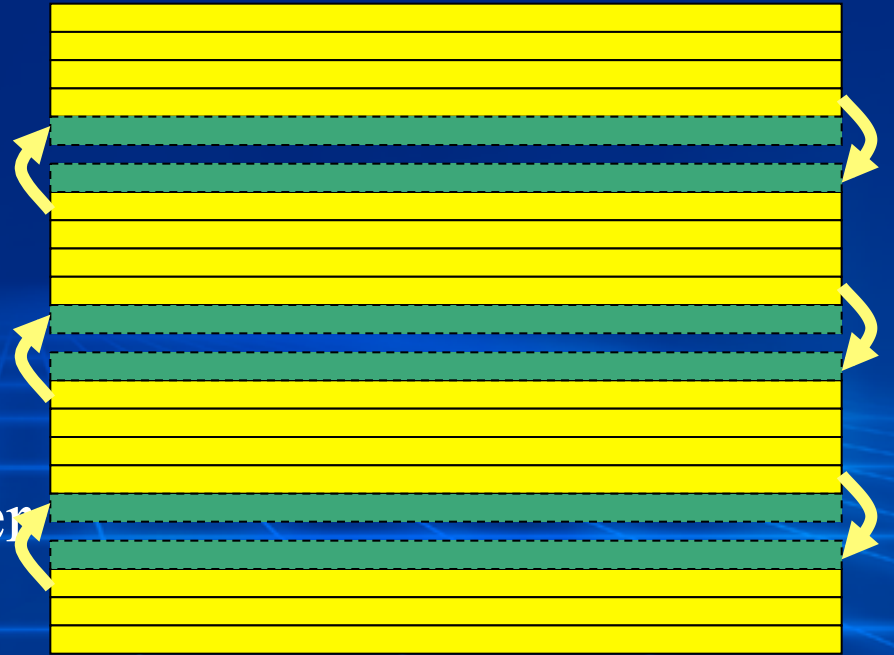
Step 3: Define the Local Arrays

- Correspondence between the local and global array
- “Global” array is an abstraction; **there is no one global array allocated anywhere**
- Instead, we compute parts of it (the local arrays) on each process
- Provide ways to output the global array by combining the values on each process (parallel I/O!)



Boundary Regions

- In order to calculate next state of cells in edge rows, need data from adjacent rows
- Need to communicate these regions at each step
 - First cut: use isend and irecv
 - Revisit with RMA later



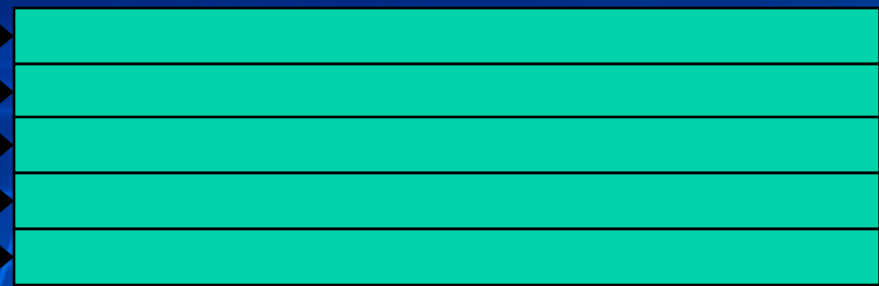
Life Point-to-Point Code Walkthrough

- Points to observe in the code:
 - Handling of command-line arguments
 - Allocation of local arrays
 - Use of a routine to implement halo exchange
 - Hides details of exchange

matrix



mdata



Allows us to use `matrix[row][col]` to address elements



Note: Parsing Arguments

- MPI standard does not guarantee that command line arguments will be passed to all processes.
 - Process arguments on rank 0
 - Broadcast options to others
 - Derived types allow one bcast to handle most args
 - Two ways to deal with strings
 - Big, fixed-size buffers
 - Two-step approach: size first, data second (what we do in the code)



Point-to-Point Exchange

- Duplicate communicator to ensure communications do not conflict
 - This is good practice when developing MPI codes, but is not required in this code
 - If this code were made into a component for use in other codes, the duplicate communicator would be required
- Non-blocking sends and receives allow implementation greater flexibility in passing messages



Describing Data

Need to save this region in the array



- Lots of rows, all the same size

- Rows are all allocated as one big block
- Perfect for MPI_Type_vector

```
MPI_Type_vector(count = myrows,  
                blklen = cols, stride = cols+2, MPI_INT, &vectype);
```

- Second type gets memory offset right (allowing use of MPI_BOTTOM in MPI_File_write_all)

```
MPI_Type_hindexed(count = 1, len = 1,  
                  disp = &matrix[1][1], vectype, &type);
```



See mlife-ic-stout.c pp. 4-6 for code example.

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

vpn

- 选择左边拦扳手图标按钮，打开网络配置，点击左下脚 +
- 选择创建vpn
- 10.1.151.2
- 用户名：rek
- 密码：testrek
- 高级
 - 去掉 EAP 勾
 - 勾上 MPPE



登陆命令:

1. ssh 111.207.107.2
2. mkdir **yourname**
3. ssh ga80 或者 ga81
4. mkdir **yourname**

拷贝文件命令:

1. 在你的虚拟机:

```
scp -r mlife 111.207.107.2:/home/chen/yourname/
```

2. 在远程机器的登陆节点

```
cd yourname
```

```
scp -r mlife ga80:/home/chen/yourname/
```



```
mpicc mlife.c mlife-io-mpiio.c mlife-pt2pt.c -o mlife
```

```
mpirun -np 2 --mca btl tcp,self ./mlife -x 1024 -y 4096
```

```
mpirun -np 2 --mca btl tcp,self ./psor
```



THANKS



中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES