高性能计算技术

第七讲 并行程序设计基础

v ώ
kjhe@scut.edu.cn

华南理工大学计算机学院

内容概要

- 并行算法的设计例子：矩阵乘法
  - **Canons**
  - **DNS**
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

矩阵的划分

- 划分方法

带状划分(**striped partitioning**):

one dimensional, row or column,

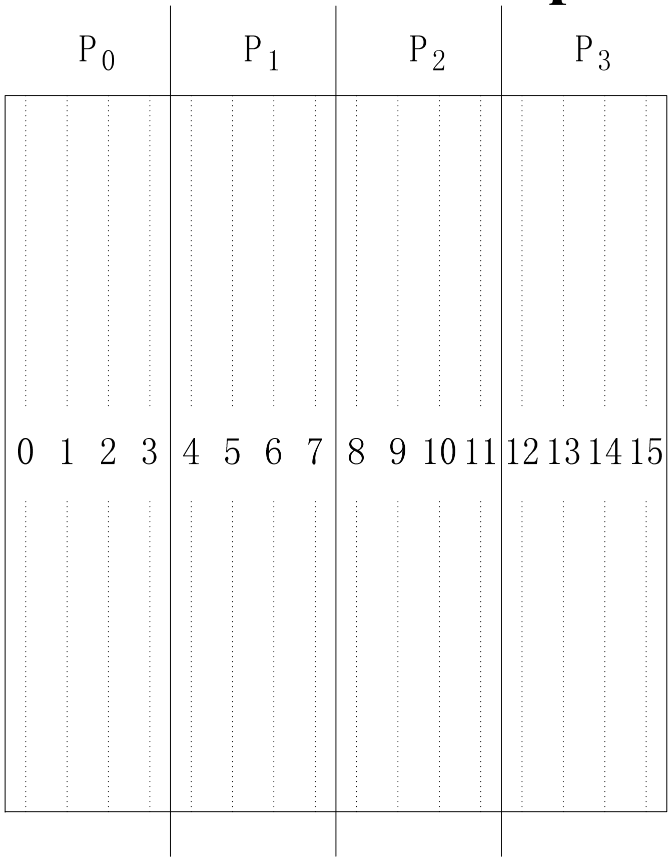
block or cyclic

棋盘划分(**checkerboard partitioning**):

two dimensional, block or cyclic

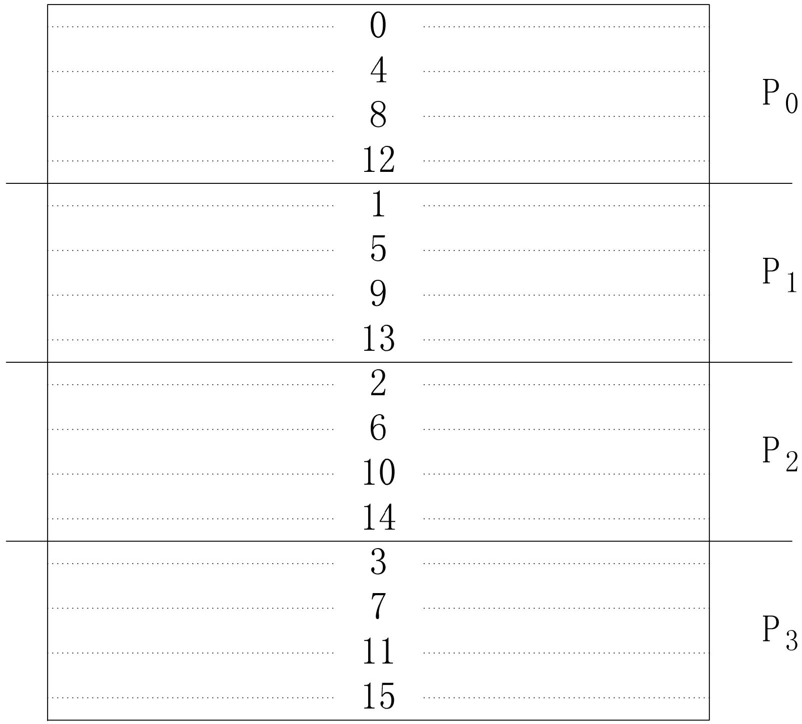
带状划分

- **16×16阶矩阵，p=4**



（a）

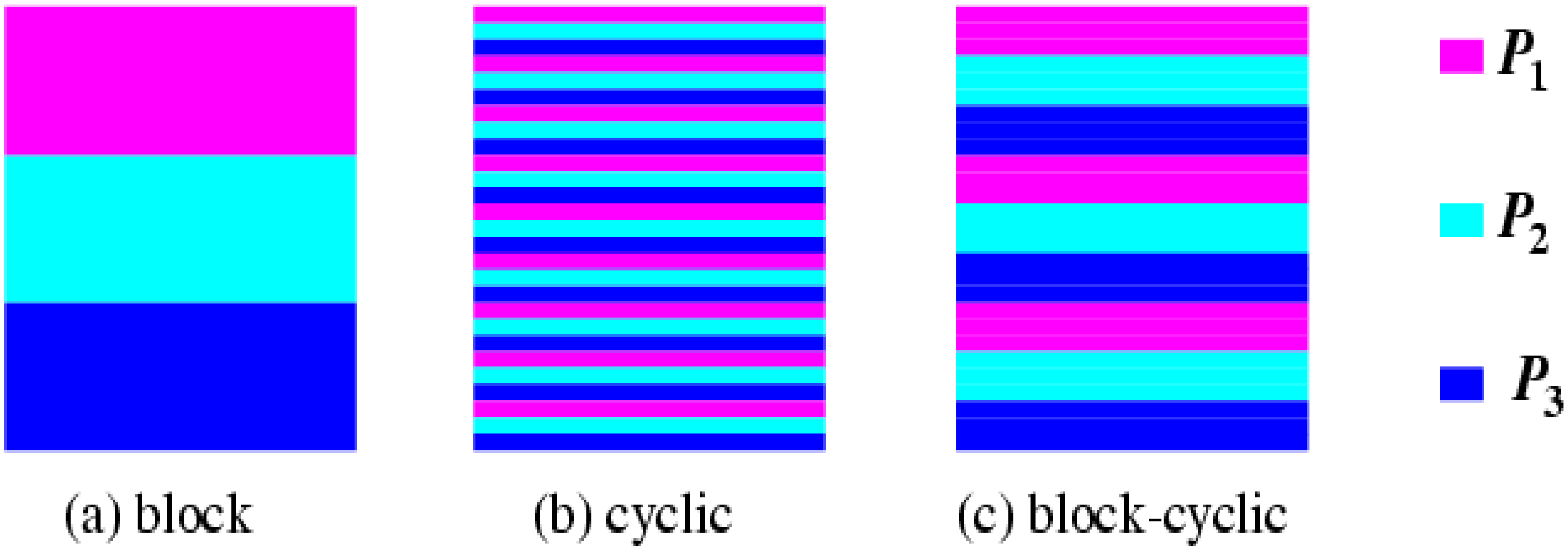
列块（block）带状划分



（b）

行循环（cyclic）带状划分

带状划分（2）



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

棋盘划分

- **8×8阶矩阵，p=16**

(0, 0) (0, 1) P_0 (1, 0) (1, 1)	(0, 2) (0, 3) P_1 (1, 2) (1, 3)	(0, 4) (0, 5) P_2 (1, 4) (1, 5)	(0, 6) (0, 7) P_3 (1, 6) (1, 7)
(2, 0) (2, 1) P_4 (3, 0) (3, 1)	(2, 2) (2, 3) P_5 (3, 2) (3, 3)	(2, 4) (2, 5) P_6 (3, 4) (3, 5)	(2, 6) (2, 7) P_7 (3, 6) (3, 7)
(4, 0) (4, 1) P_8 (5, 0) (5, 1)	(4, 2) (4, 3) P_9 (5, 2) (5, 3)	(4, 4) (4, 5) P_{10} (5, 4) (5, 5)	(4, 6) (4, 7) P_{11} (5, 6) (5, 7)
(6, 0) (6, 1) P_{12} (7, 0) (7, 1)	(6, 2) (6, 3) P_{13} (7, 2) (7, 3)	(6, 4) (6, 5) P_{14} (7, 4) (7, 5)	(6, 6) (6, 7) P_{15} (7, 6) (7, 7)

(a)

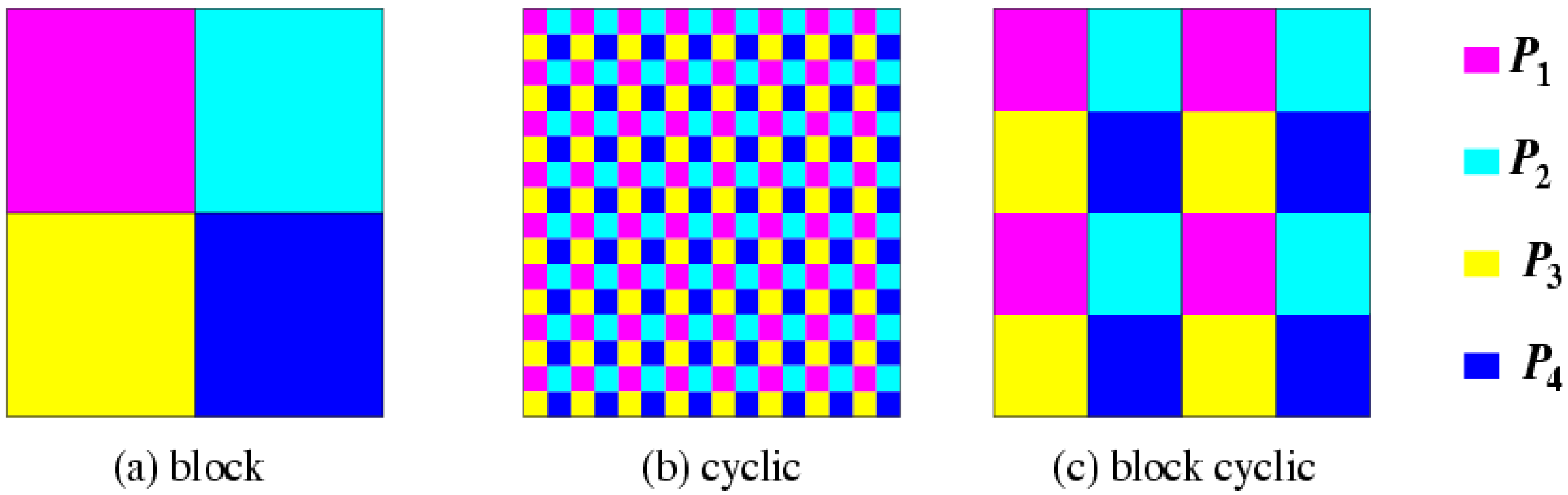
棋盘划分

(0, 0) (0, 4) P_0 (4, 0) (4, 4)	(0, 1) (0, 5) P_1 (4, 1) (4, 5)	(0, 2) (0, 6) P_2 (4, 2) (4, 6)	(0, 3) (0, 7) P_3 (4, 3) (4, 7)
(1, 0) (1, 4) P_4 (5, 0) (5, 4)	(1, 1) (1, 5) P_5 (5, 1) (5, 5)	(1, 2) (1, 6) P_6 (5, 2) (5, 6)	(1, 3) (1, 7) P_7 (5, 3) (5, 7)
(2, 0) (2, 4) P_8 (6, 0) (6, 4)	(2, 1) (2, 5) P_9 (6, 1) (6, 5)	(2, 2) (2, 6) P_{10} (6, 2) (6, 6)	(2, 3) (2, 7) P_{11} (6, 3) (6, 7)
(3, 0) (3, 4) P_{12} (7, 0) (7, 4)	(3, 1) (3, 5) P_{13} (7, 1) (7, 5)	(3, 2) (3, 6) P_{14} (7, 2) (7, 6)	(3, 3) (3, 7) P_{15} (7, 3) (7, 7)

(b)

循环棋盘划分

棋盘划分（2）

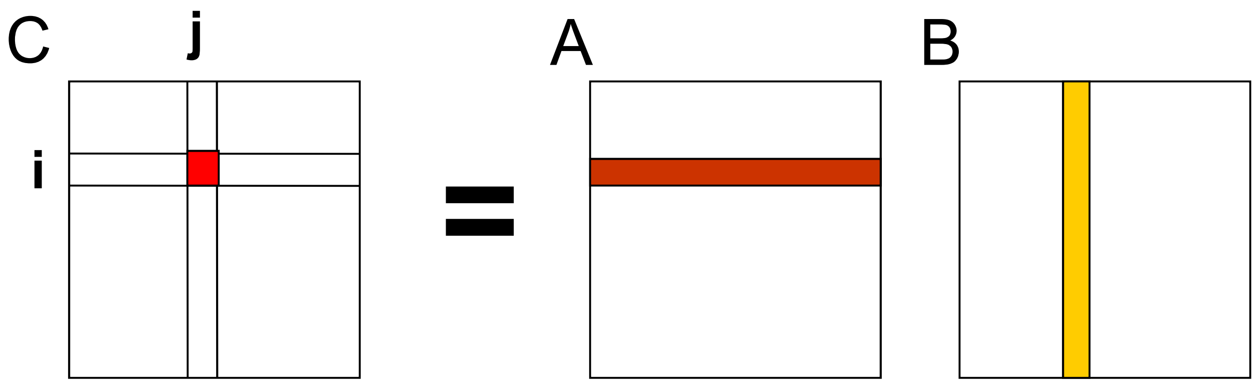


Checkerboard mapping of a 16×16 matrix on $p = 2 \times 2$ processors.

矩阵乘法定义

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**2**下标与**B**中元素的第**1**下标相一致（对准）

矩阵乘法

- 普通串行算法（算法9.3）的运行时间为**O(n^3)**
- 已知串行算法时间复杂度为**O(n^x)**，**$2<x\leq 3$**

Matrix multiplication is the most studied parallel algorithm.

It is a good algorithm to learn because it shows many ideas about parallelism.

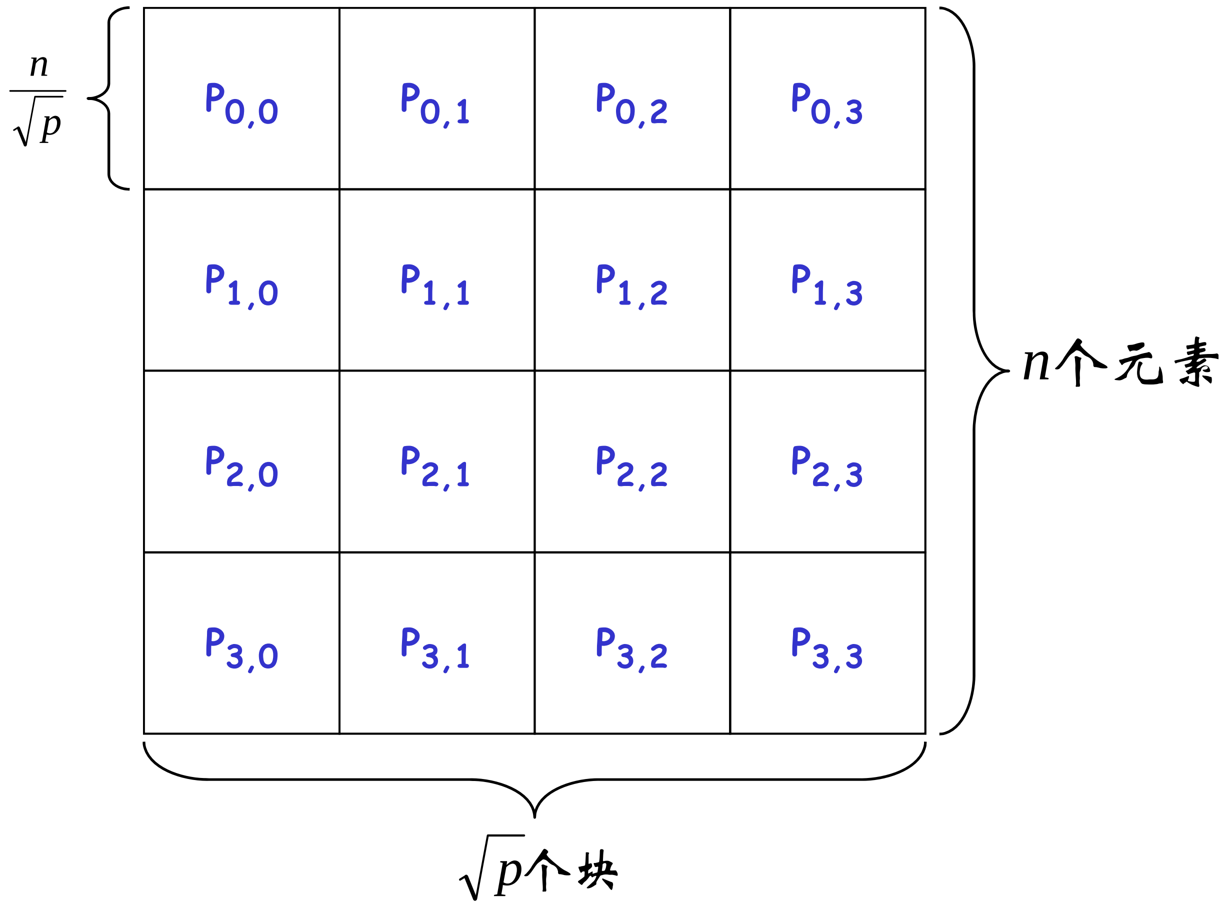
并行矩阵乘法

- 实现方法
  - 计算结构：二维阵列
  - 简单分块并行算法
  - **Cannon (1969年)**
  - **DNS (1981年)**
  - 其他：**Fox, Systolic**（时间对准）

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

矩阵分块

- 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$C_{i,j}$ $(n>>p)$



简单并行分块算法

- 分块: A、B和C分成 $p=p^{1/2}\times p^{1/2}$块大小为$(n/p^{1/2})\times(n/p^{1/2})$的方块阵$A_{i,j}$、$B_{i,j}$和$C_{i,j}$, 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$A_{i,k}$, $k=0\sim p^{1/2}-1$）

  每列处理器进行B矩阵块的多到多播送（得到$B_{k,j}$, $k=0\sim p^{1/2}-1$）

  ②乘-加运算: $P_{i,j}$做 $C_{ij}=\sum_{k=0}^{\sqrt{p}-1}A_{ik}\cdot B_{kj}$
- 计算时间?
- 存在的问题?

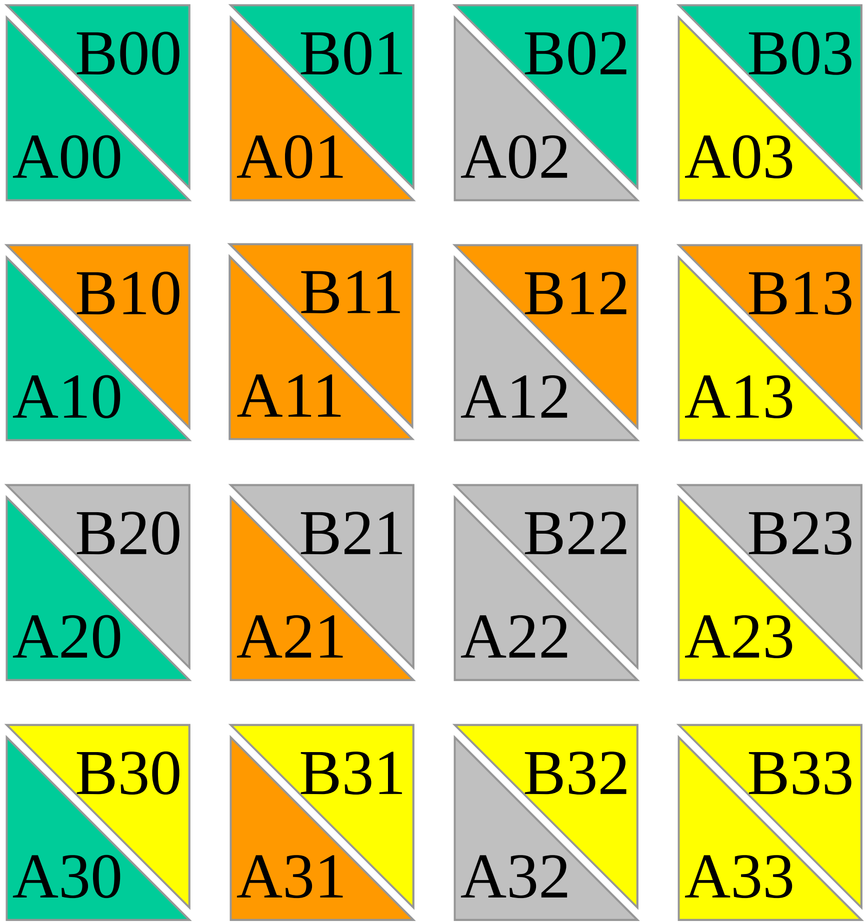
内容概要

- 并行算法的设计例子：矩阵乘法
  - **Canons**
  - **DNS**
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

矩阵分块的颜色表示

每个三角代表一个矩阵块

只有相同颜色的三角可以相乘



块的重排

块 A_{ij}向左循环移动i步

块 B_{ij}向上循环移动j步

Cannon算法描述

- 算法:

  ① 对准:

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

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

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

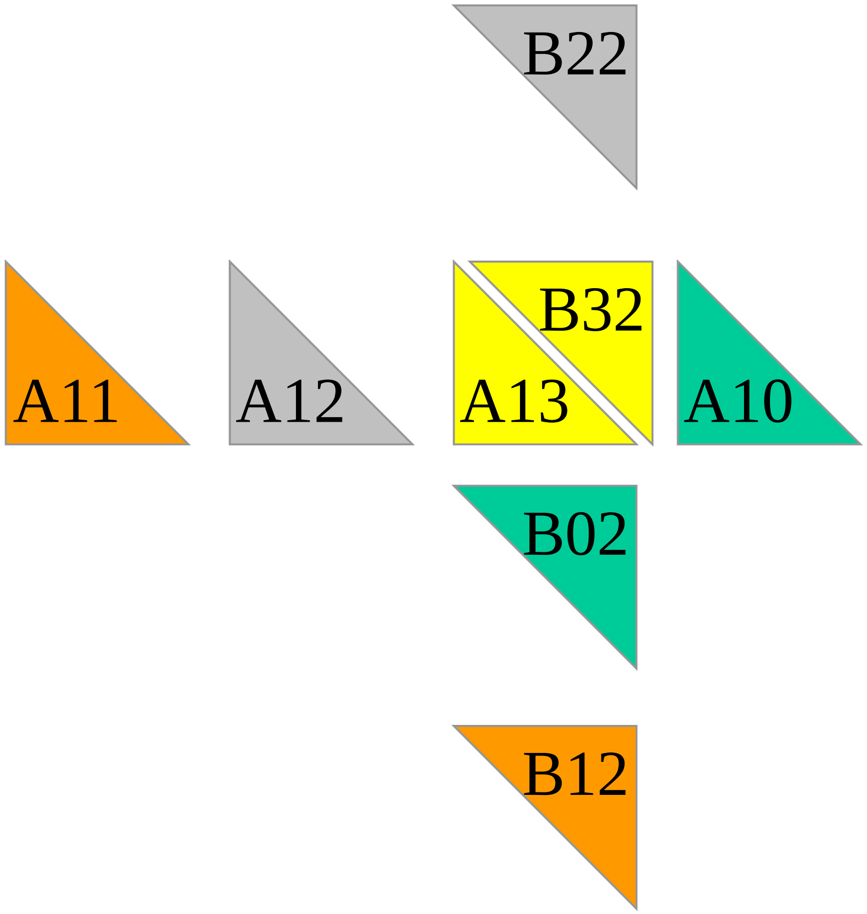
  ③ 移位:

  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;

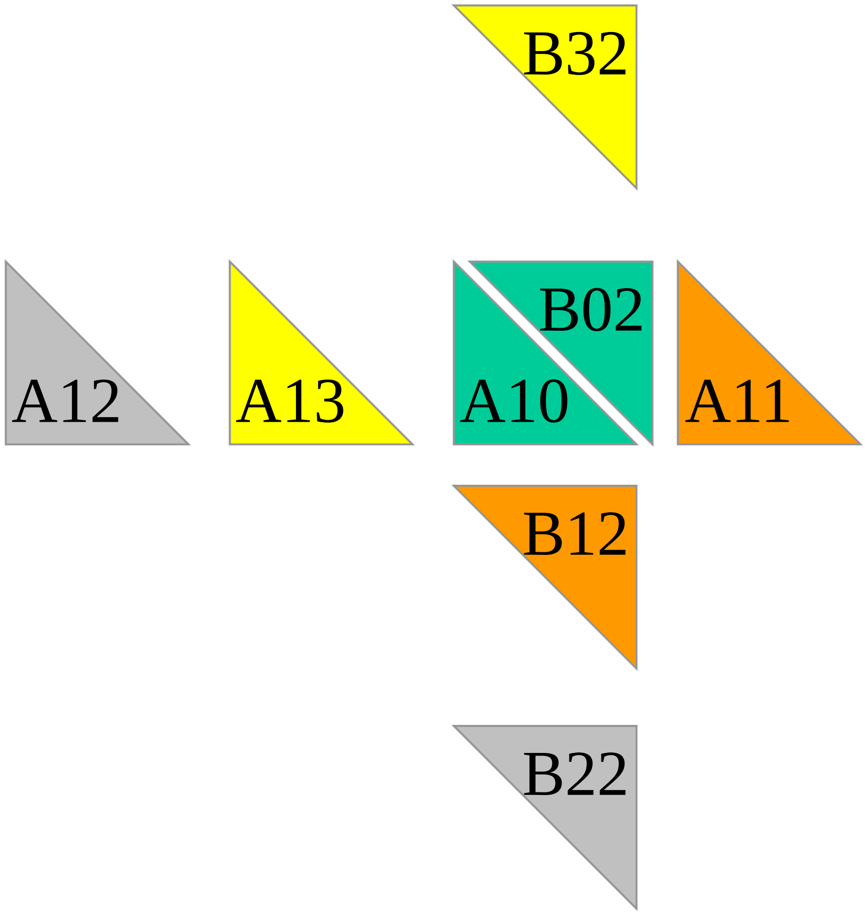
  ④ 转②执行$\sqrt{p}-1$次;

考虑处理器 $P_{1,2}$



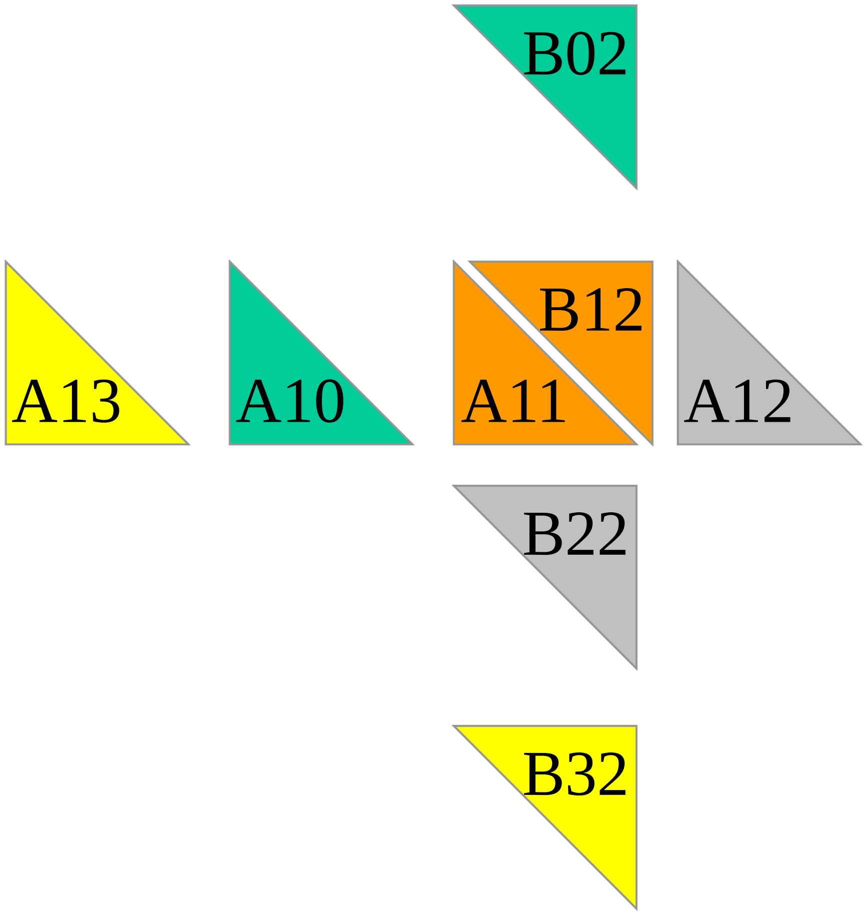
Step 1： 起始后对准

考虑处理器 $P_{1,2}$



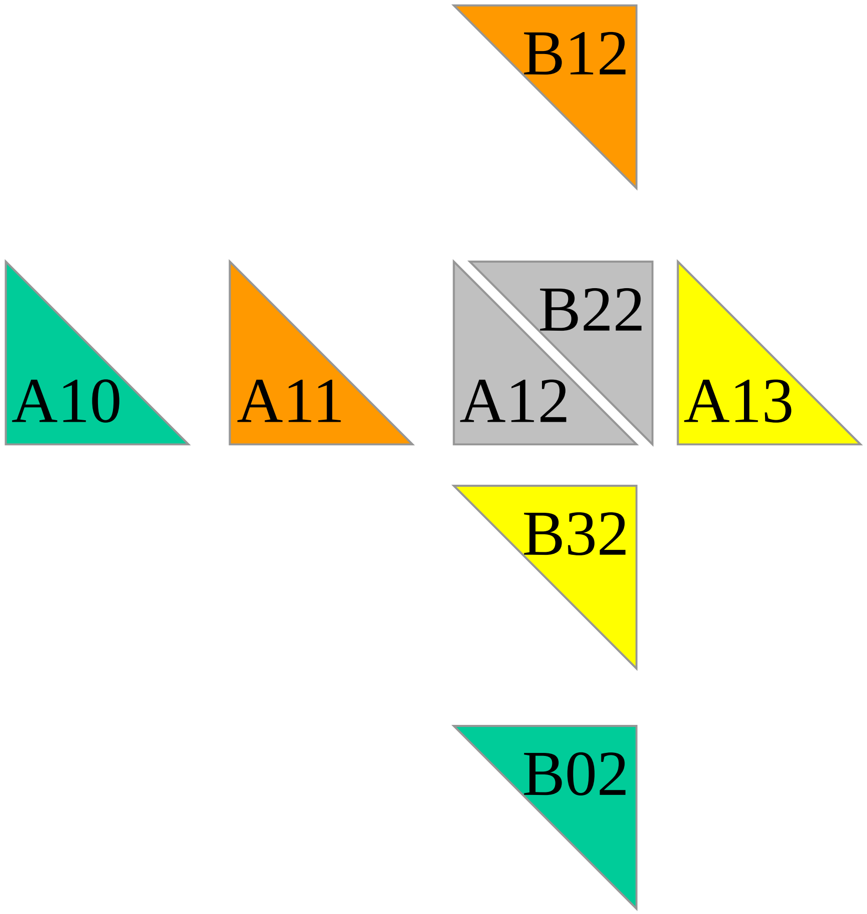
Step 2： 第一次移位后

考虑处理器 $P_{1,2}$



Step 3： 第二次移位后

考虑处理器 $P_{1,2}$



Step 4： 第三次移位后

移位的结果

第一次移位后

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

第二次移位后

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

第三次移位后

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

Cannon分块乘法

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

算法9.5

Begin

(1) for k=0 to $p^{1/2}$-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 $p^{1/2}$-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

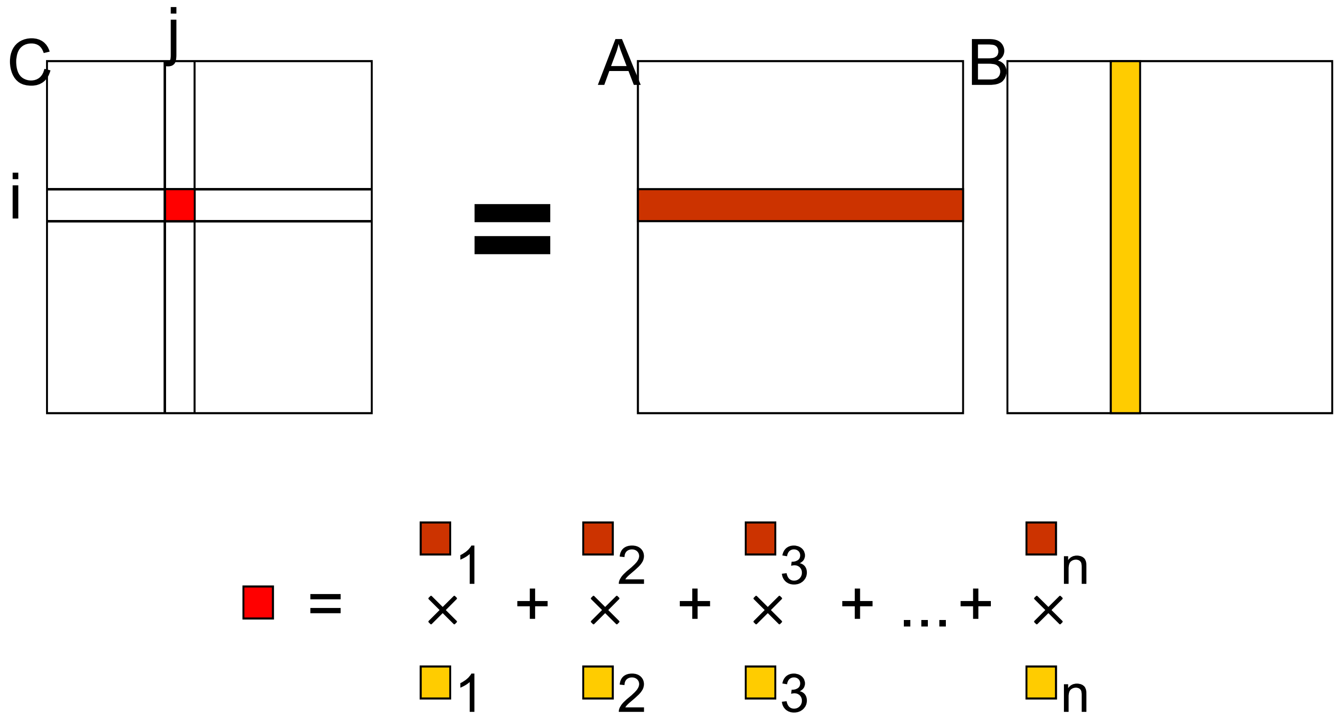
复杂度分析

- 算法有 $p^{1/2}$ 次循环
- 在每次循环，有 $(n / p^{1/2}) \times (n / p^{1/2})$的矩阵乘法： $\Theta(n^3 / p^{3/2})$
- 计算复杂度： $\Theta(n^3 / p)$
- 在每个循环，每个处理器发送和接收两个大小为 $(n / p^{1/2}) \times (n / p^{1/2})$的数据块
  - 每个处理器的通信复杂度： $\Theta(n^2 / p^{1/2})$
- 串行算法： $\Theta(n^3)$
- 并行开销： $\Theta(p^{1/2} n^2)$

内容概要

- 并行算法的设计例子：矩阵乘法
  - **Canons**
  - **DNS**
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

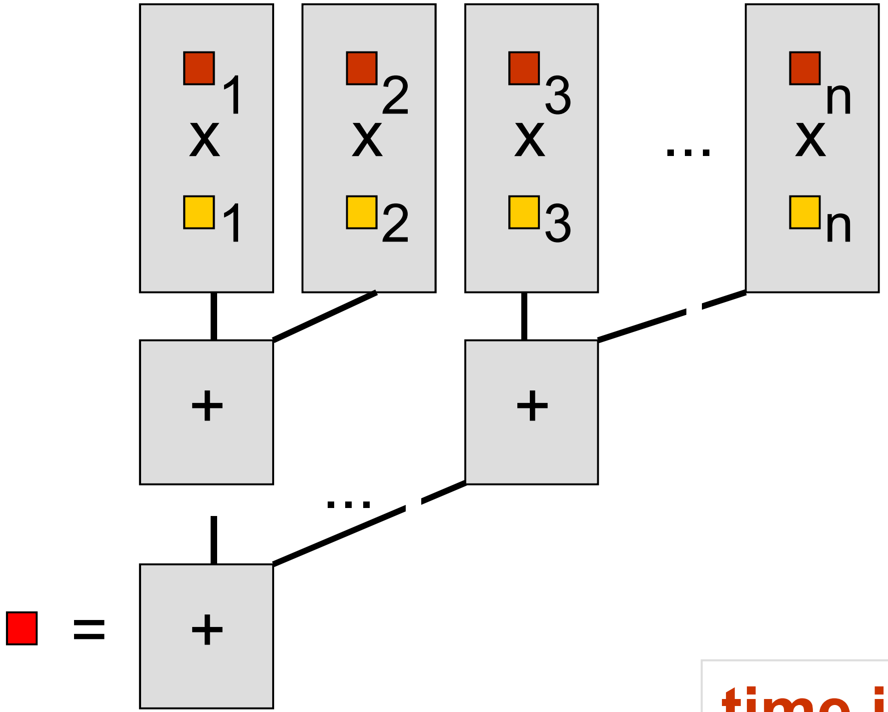
棋盘划分的矩阵乘法



是否可利用更多的处理器达到更高的加速比？

DNS矩阵乘法的思路

- **Motivation: From a good and common idea**



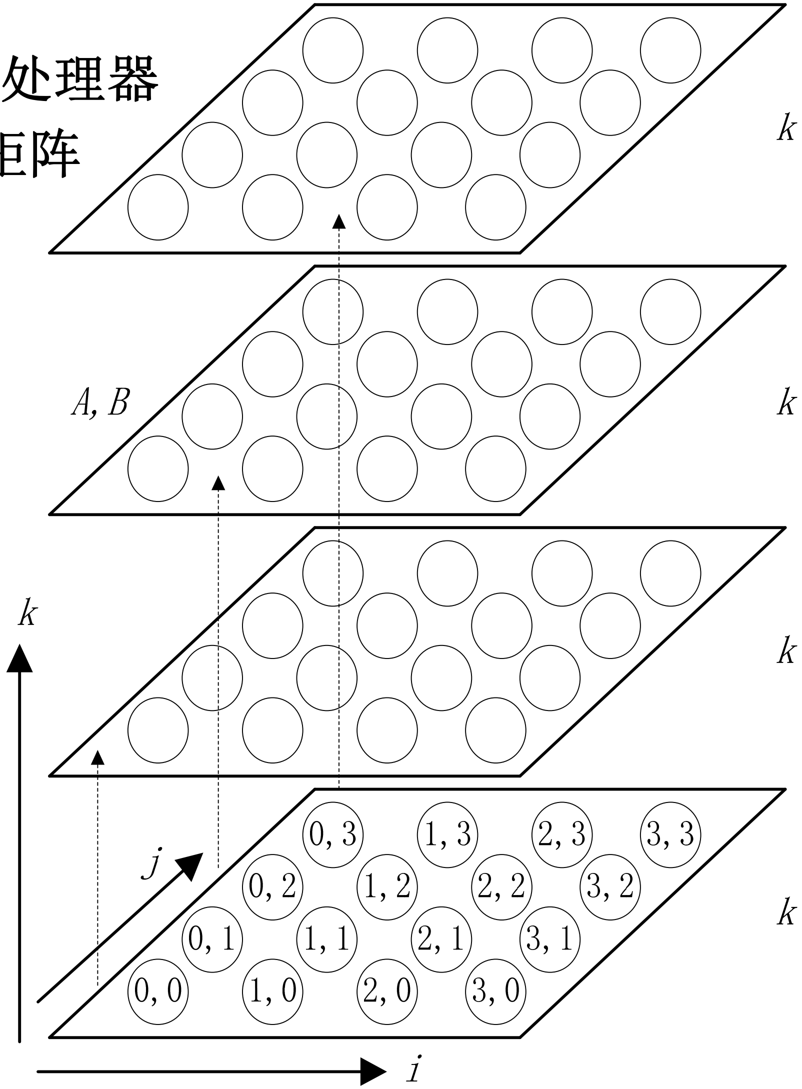
time is $\log_2 n$

DNS矩阵分块

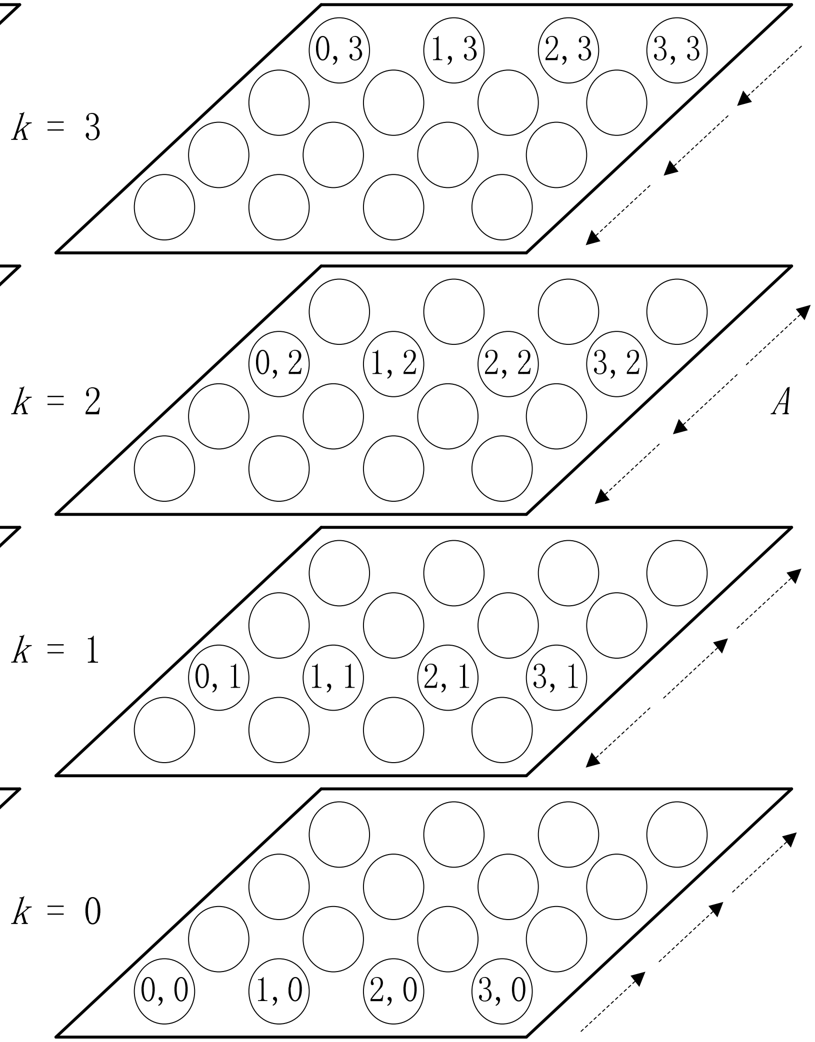
- 背景: 由Dekel、Nassimi和Sahni（1981）提出的SIMD-CC上的矩阵乘法,处理器数目为n^3, 运行时间为O(logn), 是一种速度很快的算法
- 基本思想: 通过一到一和一到多的播送办法，使得处理器(k,i,j)拥有$a_{i,k}$和$b_{k,j}$,进行本地相乘,再沿k方向进行单点积累求和,结果存储在处理器(0,i,j)中
- 处理器编号: 处理器数$p=n^3=(2^q)^3=2^{3q}$, 处理器P_r位于位置(k,i,j), 这里$r=kn^2+in+j$, $(0\le i, j, k\le n-1)$。位于(k,i,j)的处理器P_r的三个寄存器A_r,B_r,C_r分别表示为A[k,i,j], B[k,i,j]和C[k,i,j], 初始时均为0

DNS矩阵乘法示例（1）

64个处理器
4×4矩阵



(a)

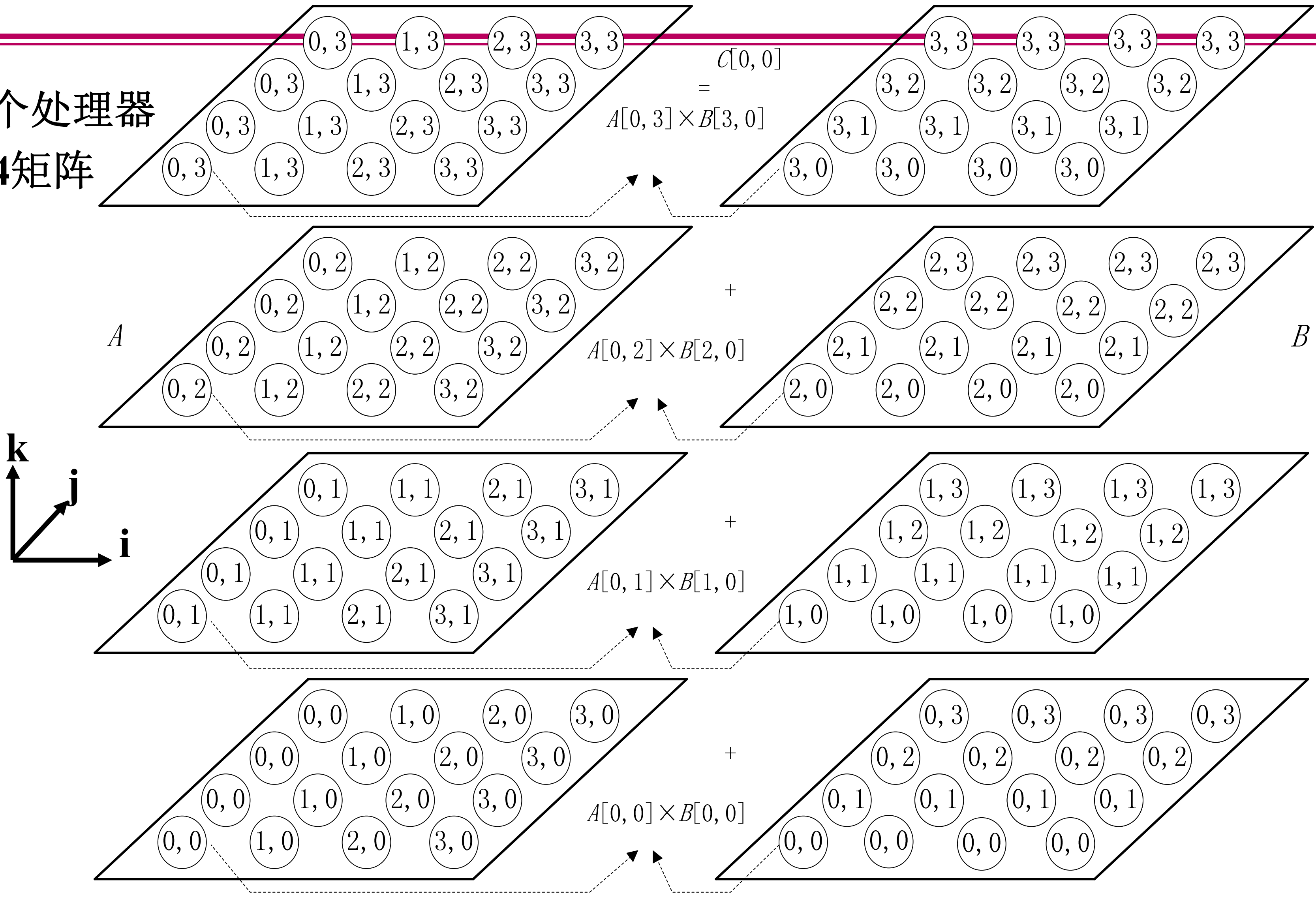


(b)

DNS矩阵乘法示例（2）

64个处理器

4×4矩阵



(c) (d)

DNS矩阵乘法算法描述

- 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

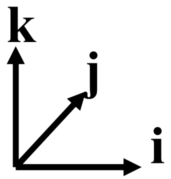
DNS矩阵乘法示例

- 示例

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

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

求$C=A\times B$



$C_{00}=1\times(-5)+2\times 7=9$

$C_{01}=1\times(-6)+2\times 8=10$

$C_{10}=3\times(-5)+4\times 7=13$

$C_{11}=3\times(-6)+4\times 8=14$

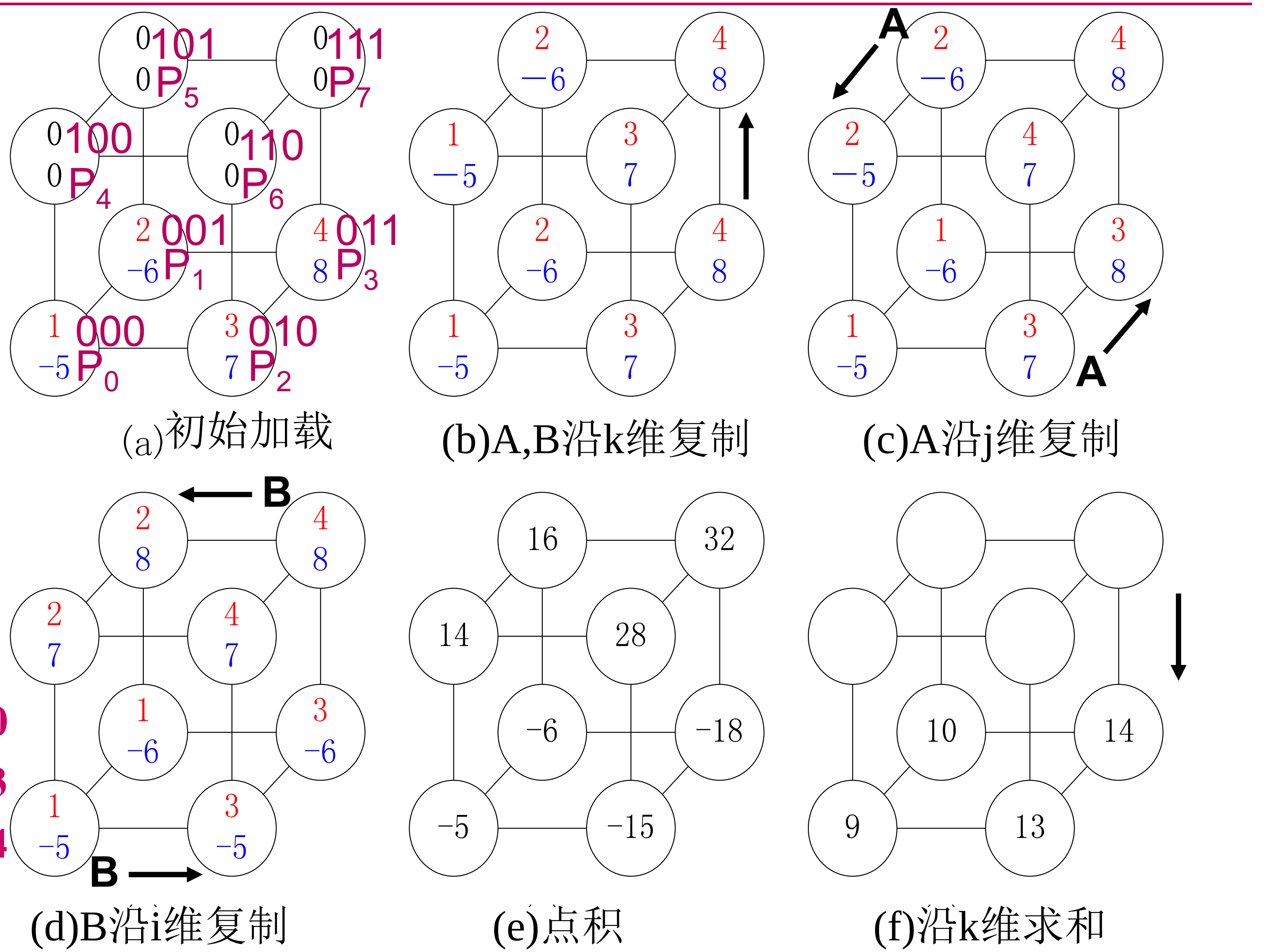


图9.12

DNS矩阵乘法算法（1）

算法9.6

```
//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_2 r_1 r_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
```

DNS矩阵乘法算法（2）

算法9.6

```
(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
```

非成本最优

内容概要

- 并行算法的设计例子：矩阵乘法
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

并行程序设计难的原因

- 技术先行，缺乏理论指导
- 程序的语法/语义复杂，需要用户自己处理
  - 任务/数据的划分/分配
  - 数据交换
  - 同步和互斥
  - 性能平衡
- 并行语言缺乏代码可扩展和异构可扩展，程序移植困难，重写代码难度太大
- 环境和工具缺乏较长的生长期，缺乏代码可扩展和异构可扩展

并行程序构造方法（1）

串行代码段

```
for ( i= 0; i<N; i++ ) A[i]=b[i]*b[i+1];
for (i= 0; i<N; i++) c[i]=A[i]+A[i+1];
```

(a) 使用库例程构造并行程序

```
id=my_process_id();
p=number_of_processes();
for ( i= id; i<N; i=i+p) A[i]=b[i]*b[i+1];
barrier();
for (i= id; i<N; i=i+p) c[i]=A[i]+A[i+1];
```

例子: **MPI,PVM, Pthreads**

(b) 扩展串行语言

my_process_id,number_of_processes(), and barrier()

```
A(0:N-1)=b(0:N-1)*b(1:N)
c=A(0:N-1)+A(1:N)
```

例子: **Fortran 90**

(c) 加编译注释构造并行程序的方法

```
#pragma parallel
#pragma shared(A,b,c)
#pragma local(i)
{
# pragma pfor iterate(i=0;N;1)
for (i=0;i<N;i++) A[i]=b[i]*b[i+1];
# pragma synchronize
# pragma pfor iterate (i=0; N; 1)
for (i=0;i<N;i++)c[i]=A[i]+A[i+1];
}
```

例子：**SGI power C, OpenMP**

并行程序构造方法（2）

三种并行程序构造方法比较

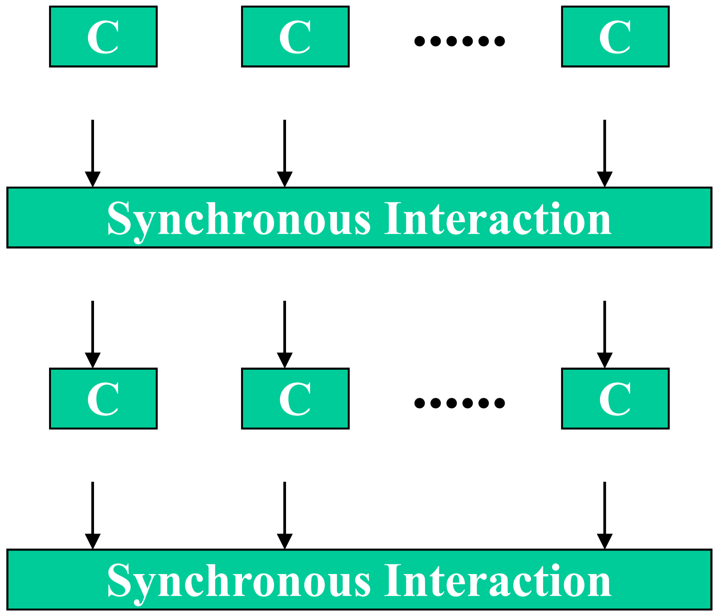
方法	实例	优点	缺点
库例程	MPI, PVM	易于实现，不需要新编译器	无编译器检查，分析和优化
扩展	Fortran90	允许编译器检查、分析和优化	实现困难,需要新编译器
编译器注释	OpenMP, HPF SGI powerC	介于库例程和扩展方法之间，在串行平台上不起作用.	

并行编程风范

- 相并行（**Phase Parallel**）
- 分治并行（**Divide and Conquer Parallel**）
- 流水线并行（**Pipeline Parallel**）
- 主从并行（**Master-Slave Parallel**）
- 工作池并行（**Work Pool Parallel**）

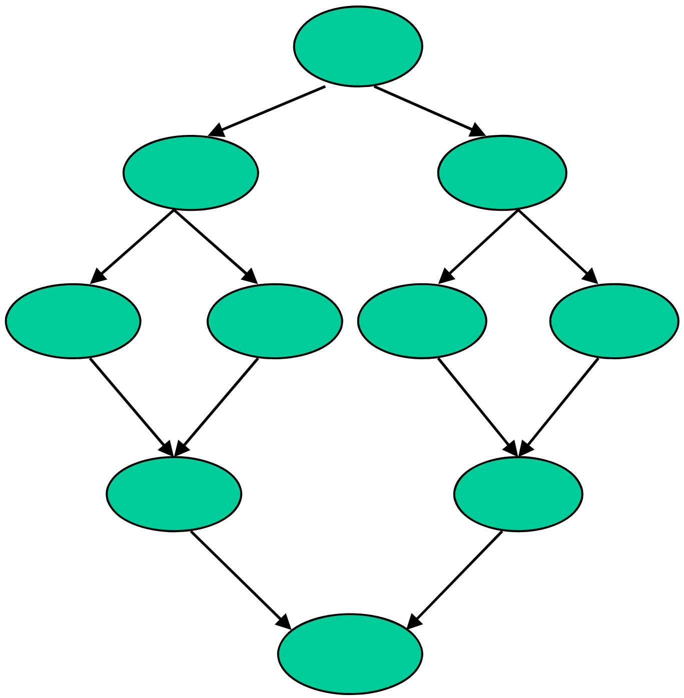
相并行
（Phase Parallel）

- 一组超级步（相）
- 步内各自计算
- 步间通信、同步
- BSP（4.2.3）
- 方便差错和性能分析
- 计算和通信不能重叠



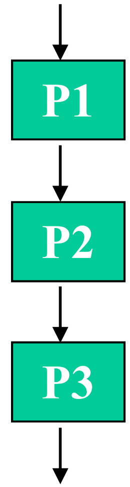
分治并行
（Divide and Conquer Parallel）

- 父进程把负载分割并指派给子进程
- 递归
- 重点在于归并
- 分治设计技术（6.2）
- 难以负载平衡



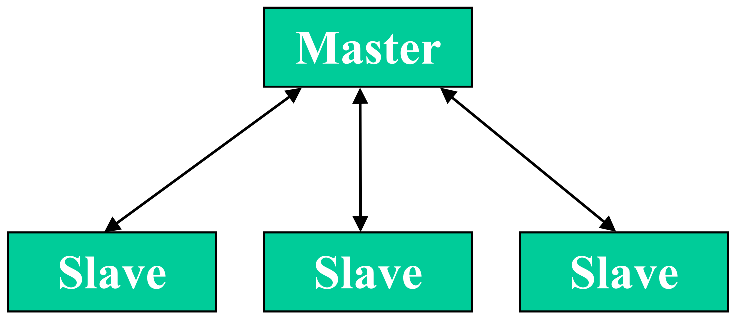
流水线并行
（Pipeline Parallel）

- 一组进程
- 流水线作业
- 流水线设计技术（6.5）



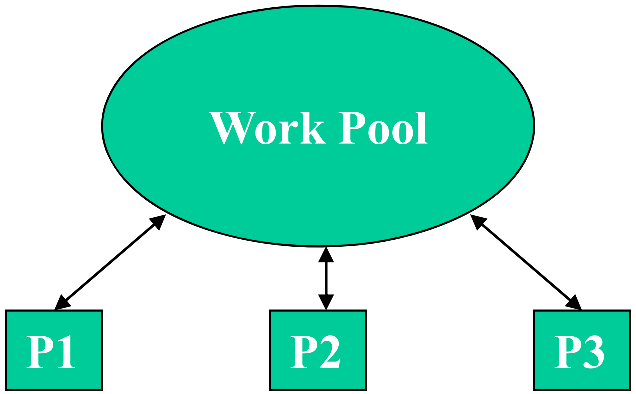
主一从并行
（Master-Slave Parallel）

- 主进程：串行、协调任务
- 子进程：计算子任务
- 划分设计技术（**6.1**）
- 与相并行结合
- 主进程易成为瓶颈



工作池并行
（Work Pool Parallel）

- 初始状态：一件工作
- 进程从池中取任务执行
- 可产生新任务放回池中
- 直至任务池为空
- 易与负载平衡
- 临界区问题（尤其消息传递）



内容概要

- 并行算法的设计例子：矩阵乘法
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

进程的同构性

- 进程：并行程序的基本计算单位
- **SIMD**：所有进程在同一时间执行相同的指令
- **MIMD**：各个进程在同一时间可以执行不同的指令
  - **SPMD**（**Single Program Multiple Data**）：各个进程是同构的，多个进程对不同的数据执行相同的代码（一般是数据并行的同义语）
    - 常对应并行循环，数据并行结构，单代码
  - **MPMD**（**Multiple Program Multiple Data**）：各个进程是异构的， 多个进程执行不同的代码（一般是任务并行，或功能并行，或控制并行的同义语）
    - 常对应并行块，多代码

要为有1000个处理器的计算机编写一个完全异构的并行程序是很困难的

SPMD例子

```
main(int argc, char **argv)
   {
     if(process is to become Master)
         {
                  MasterRoutine(/*arguments*/)
         }
      else /* it is worker process */
      {
                  WorkerRoutine(/*arguments*/)
         }
   }
```

SPMD 对比MPMD

- **SPMD**

  并行循环：当并行块中所有进程共享相同代码时

  - **parbegin S1 S2 S3 …….Sn parend**

    S1 S2 S3 …….Sn是相同代码
  - 可以简化为：

    parfor (i=1; i<=n, i++) S(i)

- **MPMD**

  并行块

  - **parbegin S1 S2 S3 …….Sn parend**

    S1 S2 S3 …….Sn 可以是不同的代码
  - 也可以用 **SPMD**来仿真**:**

```
parfor (i=0; i<3, i++) {
    if (i=0) S1
    if (i=1) S2
    if (i=2) S3
}
```

因此，对于可扩展并行机来说，只要支持SPMD就足够了

静态和动态并行性

- 静态并行性: 程序的结构以及进程的个数在运行之前（如编译时, 连接时或加载时）就可确定, 就认为该程序具有静态并行性.
- 动态并行性: 否则就认为该程序具有动态并行性. 即意味着进程要在运行时创建和终止

静态并行性的例子:

```
parbegin  P, Q, R  parend
```

其中P,Q,R是静态的

动态并行性的例子:

```
while (C>0) begin
      fork (foo(C));
      C:=boo(C);
end
```

动态并行性

- 开发动态并行性的一般方法: 分支/汇合（**Fork/Join**）
  - **Fork:** 派生一个子进程
  - **Join:** 强制父进程等待子进程

```
Process A:
begin
    Z:=1
    fork(B);
    T:=foo(3);
end
```

```
Process B:
begin
    fork(C);
    X:=foo(Z);
    join(C);
    output(X+Y);
end
```

```
Process C:
begin
    Y:=foo(Z);
end
```

进程分配

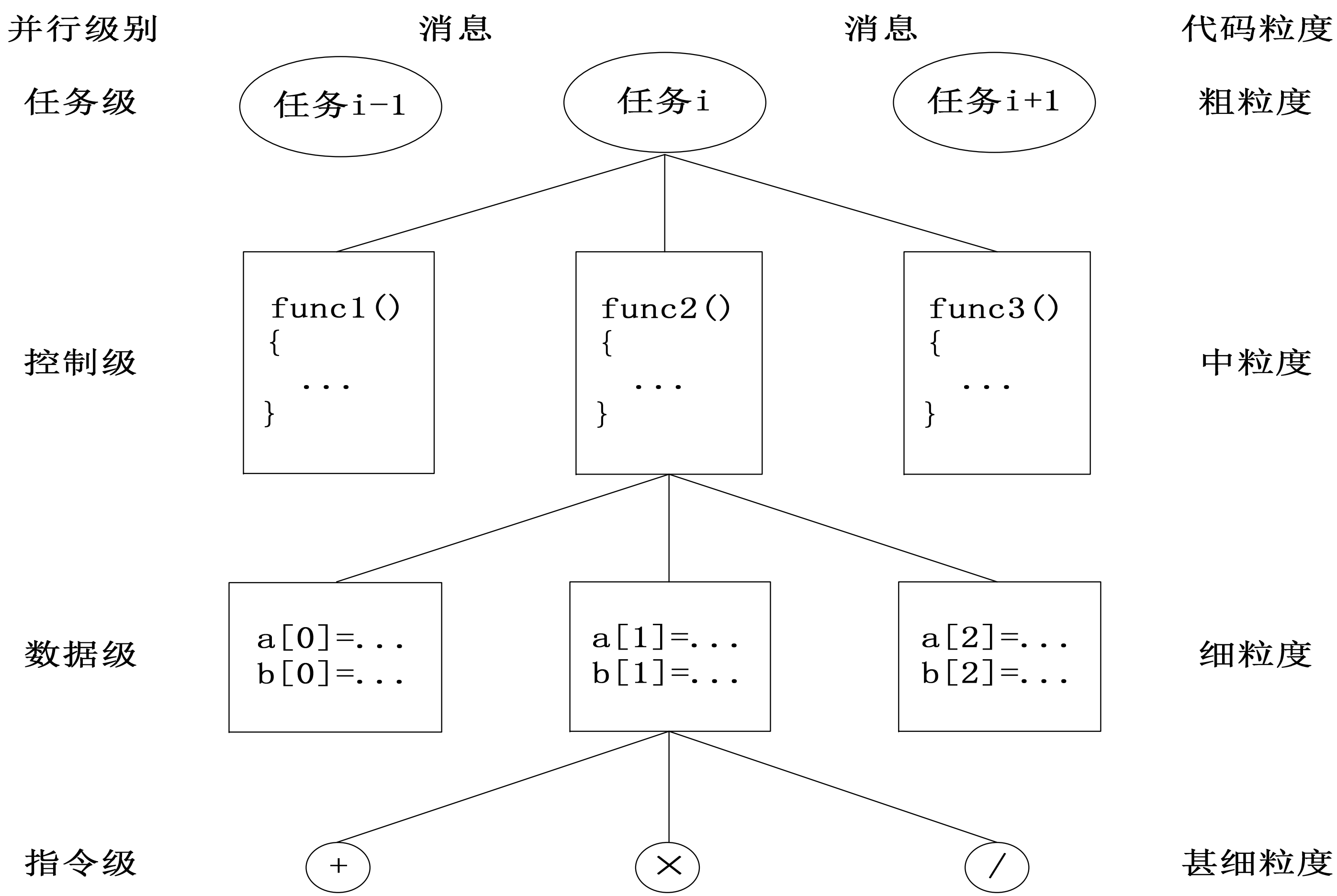
- 进程编组：支持进程间的交互,常把需要交互的进程调度在同一组中
- 一个进程组成员由：组标识符+ 成员序号 唯一确定

- 划分与分配: 使系统大部分时间忙于计算, 而不是闲置或忙于交互; 同时不牺牲并行性（度）
- 划分: 切割数据和工作负载
- 分配：将划分好的数据和工作负载映射到计算结点（处理器）上
- 分配方式
  - 显式分配: 由用户指定数据和负载如何加载
  - 隐式分配：由编译器和运行时支持系统决定
- 就近分配原则：进程所需的数据靠近使用它的进程代码

并行粒度

- 并行度（**Degree of Parallelism, DOP**）： 同时执行的分进程数
- 并行粒度（**Granularity**）： 两次并行或交互操作之间所执行的计算负载
  - 指令级并行
  - 块级（数据级）并行
  - 进程级（控制级）并行
  - 任务级并行
- 并行度与并行粒度大小常互为倒数**:** 增大粒度会减小并行度
- 增加并行度会增加系统（同步）开销

并行层次与代码粒度（1）


并行级别
消息
消息
代码粒度
任务级
任务i-1
任务i
任务i+1
粗粒度
func1()
{
...
}
func2()
{
...
}
func3()
{
...
}
控制级
中粒度
数据级
a[0]=...
b[0]=...
a[1]=...
b[1]=...
a[2]=...
b[2]=...
细粒度
指令级
+
×
/
甚细粒度

并行层次与代码粒度（2）

并行层次	粒度（指令数）	并行实施	编程支持
甚细粒度指令级并行	几十条，如多指令发射、内存交叉存取	硬件处理器	
细粒度数据级并行	几百条，如循环指令块	编译器	共享变量
中粒度控制级并行	几千条，如过程、函数	程序员（编译器）	共享变量、消息传递
粗粒度任务级并行	数万条，如独立的作业任务	操作系统	消息传递

进程交互

- 交互：进程间的相互影响
- 交互的类型
  - 通信（**communication**）：两个或多个进程间传送数的操作。通信方式：
    - 共享变量
    - 父进程传给子进程（参数传递方式）
    - 消息传递
  - 同步（**synchronization**）：导致进程间相互等待或继续执行的操作
  - 聚集（**aggregation**）：用一串超步将各分进程计算所得的部分结果合并为一个完整的结果，每个超步包含一个短的计算和一个简单的通信或/和同步

交互的模式

- 按交互模式是否能在编译时确定分为:
  - 静态的
  - 动态的
- 按有多少发送者和接收者参与通信分为
  - 一对一：点到点（**point to point**）
  - 一对多：广播（**broadcast**），散播**(scatter)**
  - 多对一：收集（**gather**）, 归约（**reduce**）
  - 多对多：全交换（**total exchange**）， 扫描（**scan**）, 置换/移位（**permutation/shift**）

同步

- 同步方式:
  - 原子同步：不可分的操作
  - 控制同步（路障,临界区）：进程的所有操作均必须等待到达某一控制状态
  - 数据同步（锁，条件临界区，监控程序，事件）：使程序执行必须等待到某一数据状态
- 例子：多进程的计数器操作

原子同步

```
parfor (i:=1; i<n; i++) {
        atomic{x:=x+1; y:=y-1}
        }
```

路障同步

```
parfor(i:=1; i<n; i++){
        Pi
        barrier
        Qi
        }
```

临界区

```
parfor(i:=1; i<n; i++){
        critical{x:=x+1; y:=y+1}
        }
```

数据同步（信号量同步）

```
parfor(i:=1; i<n; i++){
        lock(S);
        x:=x+1;
        y:=y-1;
        unlock(S)
        }
```

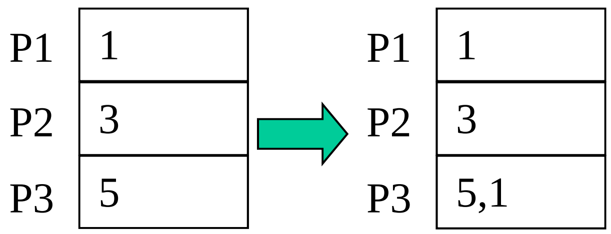
聚集

- 聚集方式:
  - 归约（**reduction**）
  - 扫描（**scan**）

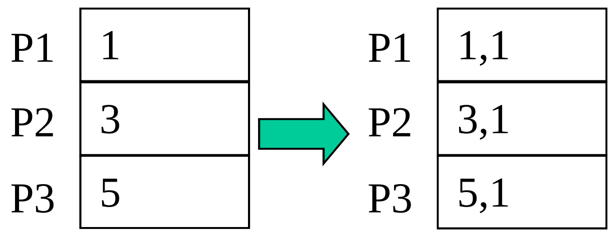
例子：**计算两个向量的内积**

```
parfor(i:=1; i<n; i++){
      X[i]:=A[i]*B[i]
      inner_product:=aggregate_sum(X[i]);
      }
```

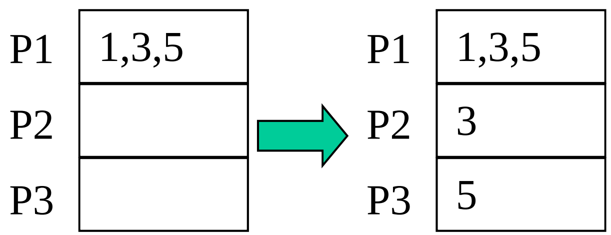
通信模式（1）



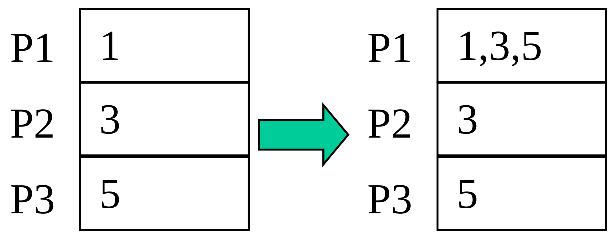
(a) 点对点（一对一）： **P1**发送一个值给**P3**



(b) 广播（一对多）： **P1**发送一个值给全体



(c) 散播（一对多）： **P1**向每个节点发送一个值



(d) 收集（多对一）： **P1**从每个节点接收一个值

通信模式（2）

P1	1,2,3		P1	1,4,7
P2	4,5,6	→	P2	2,5,8
P3	7,8,9		P3	3,6,9

(e) 全交换（多对多）： 每个节点向每个节点发送一个不同的消息

P1	1		P1	1,5
P2	3	→	P2	3,1
P3	5		P3	5,3

(f) 移位（置换, 多对多）： 每个节点向下一个节点发送一个值并接收来自上一个节点的一个值.

P1	1		P1	1,9
P2	3	→	P2	3
P3	5		P3	5

(g) 归约（多对一）： **P1得到和1+3+5=9**

P1	1		P1	1,1
P2	3	→	P2	3,4
P3	5		P3	5,9

(h) 扫描（多对多）： **P1得到1, P2得到1+3=4, P3得到1+3+5=9**

内容概要

- 并行算法的设计例子：矩阵乘法
- 并行程序设计概述
- 并行程序设计的基本问题
- 并行程序设计模型

并行程序设计模型

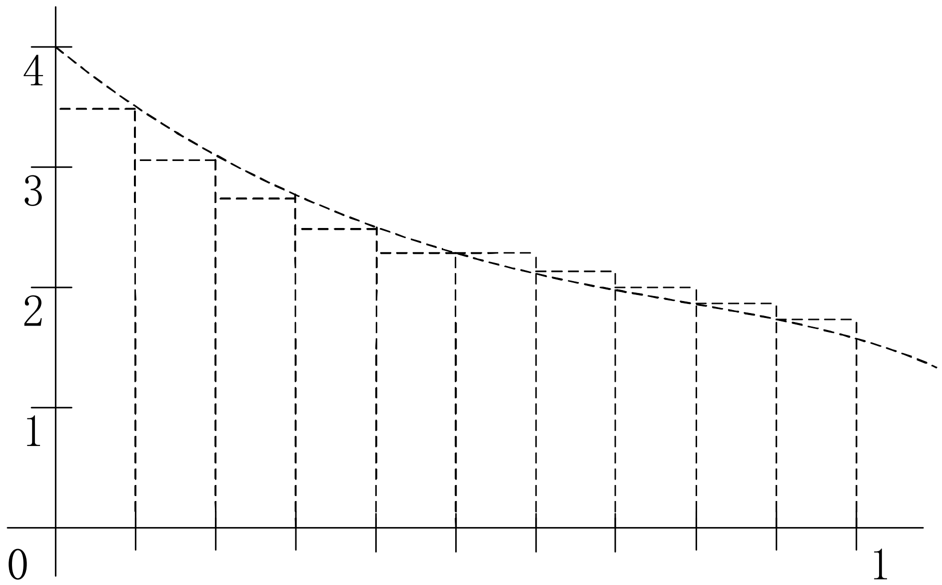
- 隐式并行（**Implicit Parallel**）模型
- 数据并行（**Data Parallel**）模型
- 共享存储（**Shared Memory**）/共享变量（**Shared Variable**）模型
- 消息传递（**Message Passing**）模型

隐式并行和显式并行

- 隐式并行
  - 程序员用熟悉的串行语言编程，编译器或运行支持系统自动转化为并行代码，并实施计算的调度和数据和安排
  - 特点：语义简单，可移植性好，单线程，易于调试和验证正确性，效率很低
- 显式并行
  - 由程序员复杂并行化的主要工作，包括任务分解、映射任务到处理器，通信结构等
  - 类型：数据并行，共享存储，消息传递

计算圆周率的样本程序

$$\pi = \int_0^1 \frac{4}{1+x^2}dx \approx \sum_{0 \le i < N} \frac{4}{1+(\frac{i+0.5}{N})^2} \cdot \frac{1}{N}$$



计算圆周率的c语言代码段

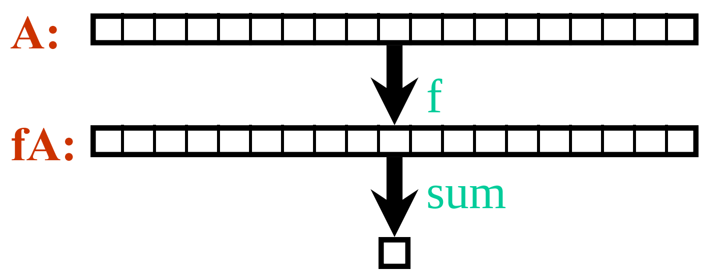
```
#define  N  1000000

main()   {
     double local, pi = 0.0, w;
     long  i;
     w=1.0/N;
     for (i = 0; i<N; i ++) {
             local = (i + 0.5)*w;
             pi = pi + 4.0/(1.0+local * local);
             }
     printf("pi is %f \n", pi *w);
}
```

编程模型：数据并行

- 包含并行操作的单系列线程控制，并行操作应用到全部的数据或其中的子集
- 并行操作的通信是隐含的
- 简单，容易理解
- 缺点：不是所有问题都可以用这种模型解决

A = array of all data
fA = f(A)
s = sum(fA)



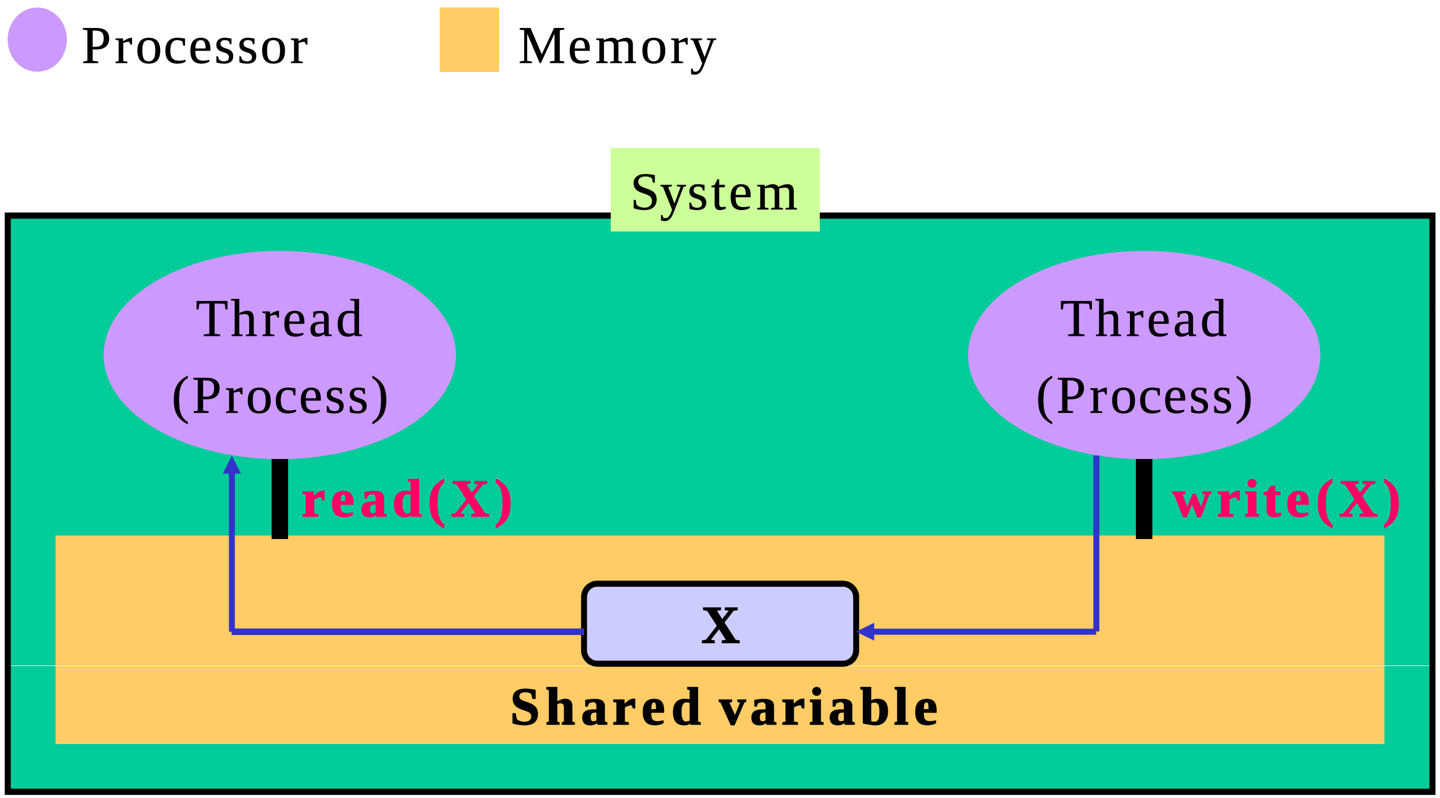
数据并行模型概述

- 概况：
  - SIMD的自然模型，也可运行于SPMD、MIMD机器上
  - 局部计算和数据选路操作
  - 适合于使用规则网络，模板和多维信号及图像数据集来求解细粒度的应用问题
  - 数据并行操作的同步是在编译时而不是在运行时完成的
- 特点：
  - 单线程
  - 并行操作于聚合数据结构（数组）
  - 松散同步
  - 单一地址空间
  - 隐式交互作用，显式数据分布

计算π的数据并行程序代码

```
main( ){
        long i，j，t，N=100000；
          double local [N]，temp [N]，pi，w；
    A：  w=1.0/N；
    B：  forall (i=0；i<N ；  i++){
       P：  local[i]=(i+0.5)*w；
       Q：  temp[i]=4.0/(1.0+local[i]*local[i])；
       }
    C：    pi = sum (temp)；
    D：    printf (“pi is %f \ n”，pi * w )；
   }  / *main( ) * /
```

编程模型：共享存储


Processor
Memory
System
Thread
(Process)
Thread
(Process)
read(X)
write(X)
X
Shared variable

共享存储代码

- 计算和

```
Thread 1

    [s = 0 initially]
    local_s1= 0
    for i = 0, n/2-1
        local_s1 = local_s1 + f(A[i])
    s = s + local_s1
```

```
Thread 2

    [s = 0 initially]
    local_s2 = 0
    for i = n/2, n-1
        local_s2= local_s2 + f(A[i])
    s = s +local_s2
```

What could go wrong?

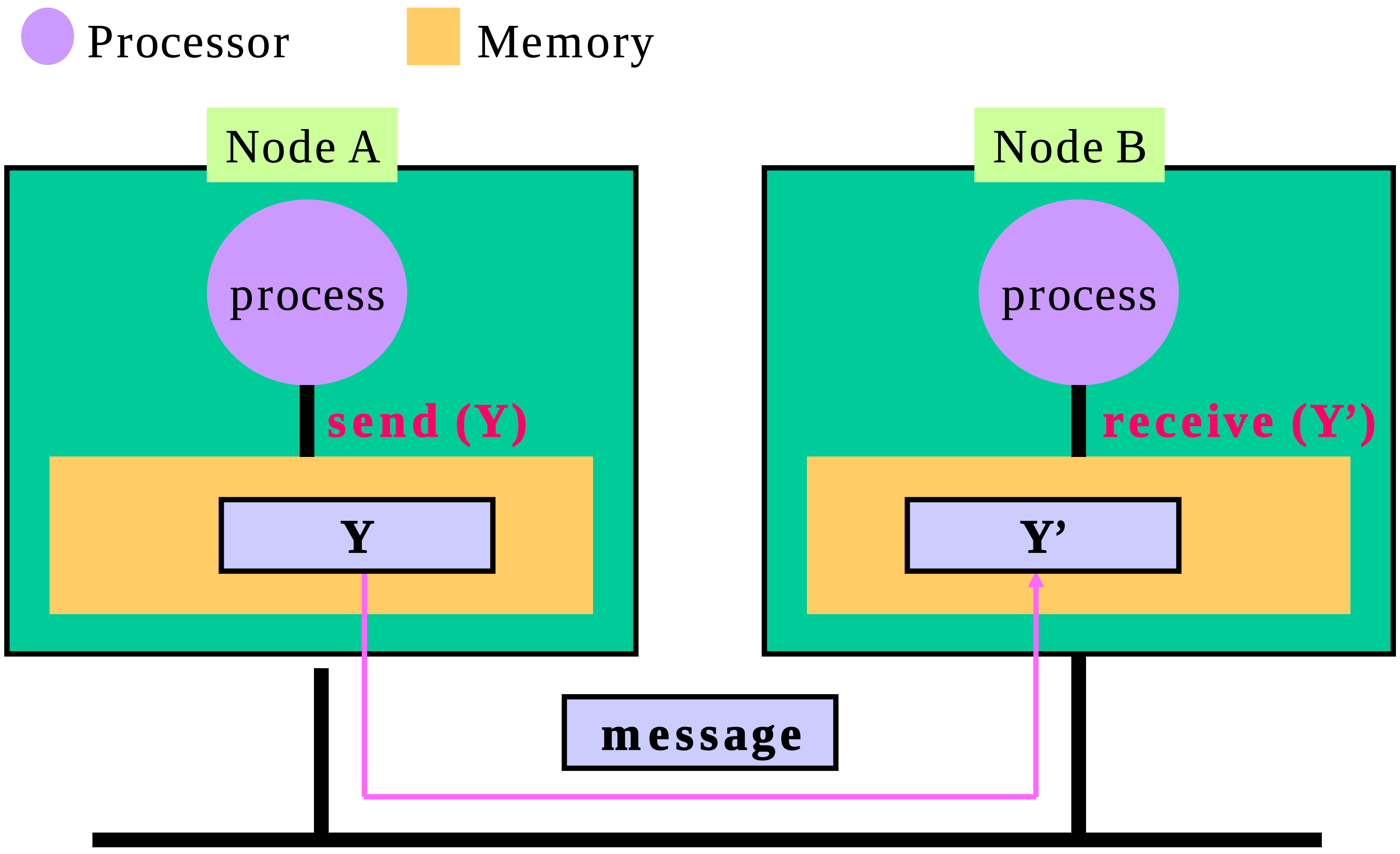
共享存储模型概述

- 概况：
  - **PVP, SMP, DSM**的自然模型
- 特点：
  - 多线程：**SPMD, MPMD**
  - 异步
  - 单一地址空间
  - 显式同步
  - 隐式数据分布
  - 隐式通信（共享变量的读/写）

计算π的共享存储程序代码

```
# define  N  100000
  main ( ){
         double  local，pi=0.0 ， w；
         long  i ；
  A ：    w=1. 0/N；
  B ：    # Pragma Parallel
         # Pragma Shared (pi，w)
         # Pragma Local (i，local)
          {
          # Pragma pfor iterate(i=0；N；1)
          for (i=0；i<N，i++){
          P：   local = (i+0.5)*w；
          Q：  local=4.0/(1.0+local*local)；
          }
  C ：    # Pragma Critical
         pi =pi +local ；
         }
  D：      printf (“pi is %f \ n”，pi *w)；
      }/ *main( ) */
```

编程模型：消息传递


Processor
Memory
Node A
Node B
process
process
send (Y)
receive (Y’)
Y
Y’
message

消息传递编程例子

- 在每个处理器上计算 s = x(1)+x(2)

```
Processor 1
    send xlocal, proc2
        [xlocal = x(1)]
    receive xremote, proc2
    s = xlocal + xremote
```

```
Processor 2
    receive xremote, proc1
    send xlocal, proc1
        [xlocal = x(2)]
    s = xlocal + xremote
```

消息传递模型概述

- 概况：
  - MPP, COW的自然模型，也可应用于共享变量多机系统，适合开发大粒度的并行性
  - 广泛使用的标准消息传递库MPI和PVM
- 特点：
  - 多线程
  - 异步并行性
  - 分开的地址空间
  - 显式相互作用
  - 显式数据映射和负载分配
  - 常采用SPMD形式编码

计算π的消息传递程序代码

```
# define  N  100000
 main ( ){
      double  local=0.0，pi，w，temp=0.0；
      long i ，taskid，numtask；
 A：   w=1.0/N；
      MPI_ Init(&argc，& argv)；
      MPI _Comm _rank (MPI_COMM_WORLD，&taskid)；
      MPI _Comm _Size (MPI_COMM_WORLD，&numtask)；
 B：   for (i= taskid； i< N； i=i + numtask){
      P：  temp = (i+0.5)*w；
      Q：  local=4.0/(1.0+temp*temp)+local；
      }
 C：MPI_Reduce (&local,&pi,1,MPI_Double,MPI_MAX,0,
                      MPI_COMM_WORLD)；
 D：if (taskid = =0) printf(“pi is %f \ n”，pi* w)；
      MPI_Finalize ( ) ；
      } / * main ( )*/
```

三种显式并行程序设计模型主要特性

特 性	数据并行	消息传递	共享存储
控制流（线）	单线程	多线程	多线程
进程间操作	松散同步	异步	异步
地址空间	单一地址	多地址空间	单地址空间
相互作用	隐式	显式	显式
数据分配	隐式或半隐式	显式	隐式或半隐式

课程小结

- 并行算法的设计例子：矩阵乘法
- 并行编程风范
  - ➢ 相并行，分治并行，流水线并行，主从并行，工作池并行
- 并行化问题
  - ➢ 进程同构：**SPMD**
  - ➢ 静态、动态：**fork/join**
  - ➢ 进程编组和并行粒度
  - ➢ 进程交互：通信、同步、聚集
- 并行编程模型
  - ➢ 隐式并行，数据并行，消息传递，共享存储

推荐阅读

- 《并行计算》
  - 第9章：稠密矩阵运算
  - 第12章：并行程序设计基础
- **P.-Z. Lee. Parallel matrix multiplication algorithms on hypercube multicomputers. International Journal of High Speed Computing, 7(3):391-406, Sep. 1995.**
- 十个利用矩阵乘法解决的经典题目
  http://www.matrix67.com/blog/article.asp?id=324

下一讲

- 共享存储编程
  - 《并行计算一结构、算法、编程》第13章