東北林業大學
NORTHEAST FORESTRY UNIVERSITY

计算机系统结构

第6章：多处理机

1.5 计算机系统结构中并行性的发展

6.1 引言

6.2 对称式共享存储器系统结构

6.3 分布式共享存储器体系结构

6.4 互联网络

1.5.1 并行性的概念

1. 并行性：计算机系统在同一时刻或者同一时间间隔内进行多种运算或操作。

只要在时间上相互重叠，就存在并行性。

- 同时性：两个或两个以上的事件在同一时刻发生。
- 并发性：两个或两个以上的事件在同一时间间隔内发生。

2. 从处理数据的角度来看，并行性等级从低到高可分为：

- 字串位串：每次只对一个字的一位进行处理。

  最基本的串行处理方式，不存在并行性。

- 字串位并：同时对一个字的全部位进行处理，不同字之间是串行的。

  开始出现并行性。

- 字并位串：同时对许多字的同一位（称为位片）进行处理。

  具有较高的并行性。

- 全并行：同时对许多字的全部位或部分位进行处理。

  最高一级的并行。

3. 从执行程序的角度来看，并行性等级从低到高可分为：

- 指令内部并行：单条指令中各微操作之间的并行。
- 指令级并行：并行执行两条或两条以上的指令。
- 线程级并行：并行执行两个或两个以上的线程。

  通常是以一个进程内派生的多个线程为调度单位。
- 任务级或过程级并行：并行执行两个或两个以上的过程或任务（程序段）

  以子程序或进程为调度单元。
- 作业或程序级并行：并行执行两个或两个以上的作业或程序。

1.5.2 提高并行性的技术途径

三种途径：

1. 时间重叠

引入时间因素，让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度。

2. 资源重复

引入空间因素，以数量取胜。通过重复设置硬件资源，大幅度地提高计算机系统的性能。

3. 资源共享

这是一种软件方法，它使多个任务按一定时间顺序轮流使用同一套硬件设备。

1.5.3 单机系统中并行性的发展

1. 在发展高性能单处理机过程中，起主导作用的是时间重叠原理。

实现时间重叠的基础：部件功能专用化

- 把一件工作按功能分割为若干相互联系的部分；
- 把每一部分指定给专门的部件完成；
- 然后按时间重叠原理把各部分的执行过程在时间上重叠起来，使所有部件依次分工完成一组同样的工作。

2. 在单处理机中，资源重复原理的运用也已经十分普遍。

- 多体存储器
- 多操作部件
  - 通用部件被分解成若干个专用部件，如加法部件、乘法部件、除法部件、逻辑运算部件等，而且同一种部件也可以重复设置多个。
  - 只要指令所需的操作部件空闲，就可以开始执行这条指令（如果操作数已准备好的话）。
  - 这实现了指令级并行。
- 阵列处理机（并行处理机）

更进一步，设置许多相同的处理单元，让它们在同一个控制器的指挥下，按照同一条指令的要求，对向量或数组的各元素同时进行同一操作，就形成了阵列处理

3. 在单处理机中，资源共享的概念实质上是用单处理机模拟多处理机的功能，形成所谓虚拟机的概念。

- 分时系统

1.5.4 多机系统中并行性的发展

1. 多机系统遵循时间重叠、资源重复、资源共享原理，发展为3种不同的多处理机：

   同构型多处理机、异构型多处理机、分布式系统

2. 耦合度

   反映多机系统中各机器之间物理连接的紧密程度和交互作用能力的强弱。

- 紧密耦合系统（直接耦合系统）：在这种系统中，计算机之间的物理连接的频带较高，一般是通过总线或高速开关互连，可以共享主存。
- 松散耦合系统（间接耦合系统）：一般是通过通道或通信线路实现计算机之间的互连，可以共享外存设备（磁盘、磁带等）。机器之间的相互作用是在文件或数据集一级上进行。

  表现为两种形式：

  - 多台计算机和共享的外存设备连接，不同机器之间实现功能上的分工（功能专用化），机器处理的结果以文件或数据集的形式送到共享外存设备，供其它机器继续处理。
  - 计算机网，通过通信线路连接，实现更大范围的资源共享。

最低耦合：除某种中间介质外，无连接无共享

3. 功能专用化（实现时间重叠）

- 专用外围处理机

  例如：输入/输出功能的分离

- 专用处理机

  如数组运算、高级语言翻译、数据库管理等，分离出来。

- 异构型多处理机系统

  由多个不同类型、至少担负不同功能的处理机组成，它们按照作业要求的顺序，利用时间重叠原理，依次对它们的多个任务进行加工，各自完成规定的功能动作。

4. 机间互连

- 容错系统
- 可重构系统

对计算机之间互连网络的性能提出了更高的要求。高带宽、低延迟、低开销的机间互连网络是高效实现程序或任务一级并行处理的前提条件。

- 同构型多处理机系统

由多个同类型或至少担负同等功能的处理机组成，它们同时处理同一作业中能并行执行的多个任务。

1.5.5 并行机的发展变化

并行机的发展可分为4个阶段。

1. 并行机的萌芽阶段（1964年～1975年）

- 20世纪60年代初期
  - **CDC6600**：非对称的共享存储结构，中央处理机采用了双**CPU**，并连接了多个外部处理器。
- 60年代后期，一个重要的突破
  - 在处理器中使用流水线和重复设置功能单元，所获得的性能提高是明显的，并比单纯地提高时钟频率来提高性能更有效。
- 在1972年，Illinois大学和Burroughs公司联合研制Illiac IV SIMD计算机（64个处理单元构成的）

  在1975年 Illiac IV系统 （16个处理单元构成）

2. 向量机的发展和鼎盛阶段（1976年～1990年）

- 1976年，Cray公司推出了第一台向量计算机Cray-1
- 在随后的10年中，不断地推出新的向量计算机。

  包括：CDC的Cyber205、Fujitsu的VP1000/VP2000、NEC的SX1/SX2以、我国的YH-1等

- 向量计算机的发展呈两大趋势
  - **提高单处理器的速度**
  - **研制多处理器系统**

3. MPP出现和蓬勃发展阶段（1990年～1995年）

- 早期的MPP
  - **TC2000（1989年）、Touchstone Delta、Intel Paragon（1992年）、KSR1、Cray T3D（1993年）、IBM SP2（1994年）和我国的曙光-1000（1995年）等。（分布存储的MIMD计算机）**
- MPP的高端机器
  - **1996年，Intel公司的ASCI Red和1997年SGI Cray公司的T3E900 （万亿次高性能并行计算机）**
- 90年代的中期，在中、低档市场上，SMP以其更优的性能/价格比代替了MPP。

4. 各种体系结构并存阶段（1995年～2000年）

- 从1995年以后，PVP（并行向量处理机）、MPP、SMP、DSM（分布式共享存储多处理机）、COW等各种体系结构进入并存发展的阶段。
- MPP系统在全世界前500强最快的计算机中的占有量继续稳固上升，其性能也得到了进一步的提高。

  如：**ASCI Red**的理论峰值速度已达到了**1Tflop/s**

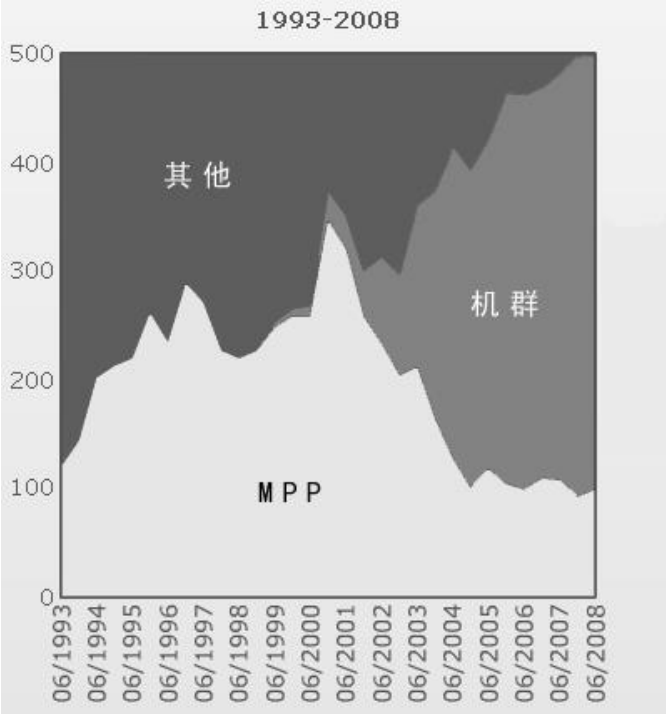
  SX4和**VPP700**等的理论峰值速度也都达到了**1Tflop/s**

5. 机群蓬勃发展阶段（2000年以后）

- 机群系统：将一群工作站或高档微机用某种结构的互连网络互连起来，充分利用其中各计算机的资源，统一调度、协调处理，以达到很高的峰值性能，并实现高效的并行计算。
- 1997年6月才有第一台机群结构的计算机进入Top500排名
- 2003年11月，这一数字已达到208台，机群首次成为Top500排名中比例最高的结构。
- 截至2008年6月，机群已经连续10期位居榜首，其数量已经达到400，占80%。

5. 机群蓬勃发展阶段（2000年以后）

- 机群已成为当今构建高性能并行计算机系统的最常用的结构。
- 1993年至2008年期间，Top500中机群和MPP的数量的分布情况。



6.1 引 言

1. 单处理机系统结构正在走向尽头？

2. 多处理机正起着越来越重要的作用。近几年来，人们确实开始转向了多处理机。

- Intel于2004年宣布放弃了其高性能单处理器项目，转向多核（multi-core）的研究和开发。
- IBM、SUN、AMD等公司
- 并行计算机应用软件已有了稳定的发展。
- 充分利用商品化微处理器所具有的高性能价格比的优势。

3. 本章重点：中小规模的计算机(处理器的个数＜32)

（多处理机设计的主流）

6.1.1 并行计算机系统结构的分类

- Flynn分类法

  SISD、SIMD、MISD、MIMD

- MIMD已成为通用多处理机系统结构的选择，原因：
  - MIMD具有灵活性；
  - MIMD可以充分利用商品化微处理器在性能价格比方面的优势。

    计算机机群系统（cluster）是一类广泛被采用的MIMD机器。

3. 根据存储器的组织结构 ， 把现有的MIMD机器分为两类：

（每一类代表了一种存储器的结构和互连策略）

➢ 集中式共享存储器结构

- 最多由几十个处理器构成。
- 各处理器共享一个集中式的物理存储器。

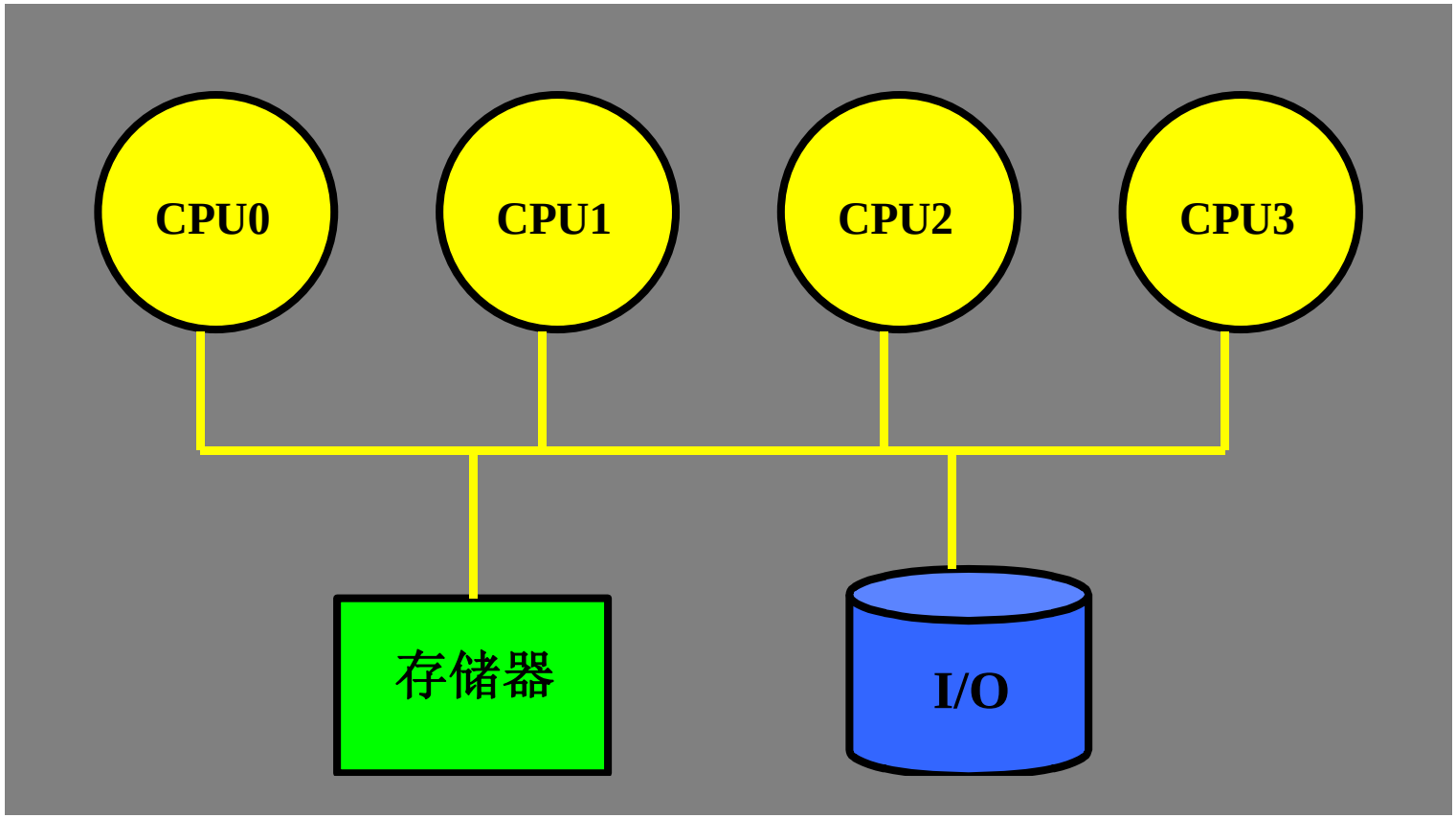
这类机器有时被称为

- SMP机器

（Symmetric shared-memory MultiProcessor）

- UMA机器（Uniform Memory Access）

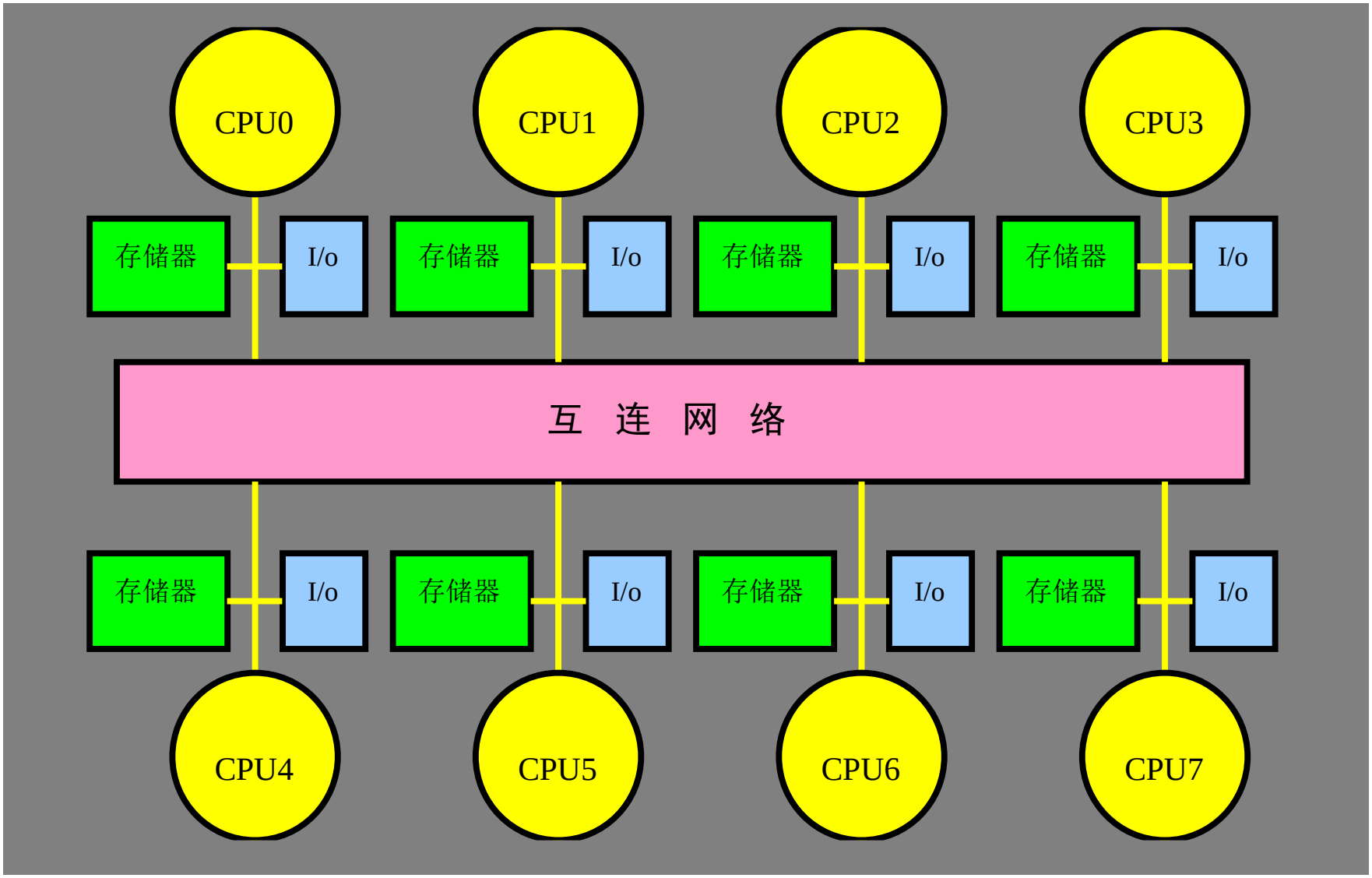
根据多处理机系统中处理器个数的多少，可把现有的MIMD机器可分为两类。



对称式共享存储器多处理机的基本结构

➢ 分布式存储器多处理机

- 存储器在物理上是分布的。
- 每个结点包含：
  - 处理器
  - 存储器
  - I／O
  - 互连网络接口
- 在许多情况下，分布式存储器结构优于集中式共享存储器结构。


CPU0
CPU1
CPU2
CPU3
存储器
I/o
存储器
I/o
存储器
I/o
存储器
I/o
互 连 网 络
存储器
I/o
存储器
I/o
存储器
I/o
存储器
I/o
CPU4
CPU5
CPU6
CPU7

- 将存储器分布到各结点有两个优点
  - 如果大多数的访问是针对本结点的局部存储器，则可降低对存储器和互连网络的带宽要求；
  - 对本地存储器的访问延迟时间小。
- 最主要的缺点
  - 处理器之间的通信较为复杂，且各处理器之间访问延迟较大。
- 簇：超级结点
  - 每个结点内包含个数较少（例如2～8）的处理器；
  - 处理器之间可采用另一种互连技术（例如总线）相互连接形成簇。

6.1.2 存储器系统结构和通信机制

- 两种存储器系统结构和通信机制
  - ➢ 共享地址空间
    - 物理上分离的所有存储器作为一个统一的共享逻辑空间进行编址。
    - 任何一个处理器可以访问该共享空间中的任何一个单元（如果它具有访问权），而且不同处理器上的同一个物理地址指向的是同一个存储单元。
    - 这类计算机被称为

      分布式共享存储器系统

      （DSM: Distributed Shared-Memory）

      NUMA机器 （NUMA: Non-Uniform Memory Access）

- 把每个结点中的存储器编址为一个独立的地址空间，不同结点中的地址空间之间是相互独立的。
  - 整个系统的地址空间由多个独立的地址空间构成
  - 每个结点中的存储器只能由本地的处理器进行访问，远程的处理器不能直接对其进行访问。
  - 每一个处理器-存储器模块实际上是一台单独的计算机
  - 现在的这种机器多以集群的形式存在

2. 通信机制

- 共享存储器通信机制
  - 共享地址空间的计算机系统采用

- 处理器之间是通过用load和store指令对相同存储器地址进行读/写操作来实现的。

➢消息传递通信机制

- 多个独立地址空间的计算机采用
- 通过处理器间显式地传递消息来完成
- 消息传递多处理机中，处理器之间是通过发送消息来进行通信的，这些消息请求进行某些操作或者传送数据。

例如：一个处理器要对远程存储器上的数据进行访问或操作：

- 发送消息，请求传递数据或对数据进行操作；

  远程进程调用(RPC， Remote Process Call)

- 目的处理器接收到消息以后，执行相应的操作或代替远程处理器进行访问，并发送一个应答消息将结果返回。

• 同步消息传递

请求处理器发送一个消息后一直要等到应答结果才继续运行。

• 异步消息传递

数据发送方知道别的处理器需要数据，通信也可以从数据发送方来开始，数据可以不经请求就直接送往数据接受方。

3. 不同通信机制的优点

➢ 共享存储器通信的主要优点

- 与常用的对称式多处理机使用的通信机制兼容。
- 易于编程，同时在简化编译器设计方面也占有优势。
- 采用大家所熟悉的共享存储器模型开发应用程序，而把重点放到解决对性能影响较大的数据访问上。
- 当通信数据量较小时，通信开销较低，带宽利用较好。
- 可以通过采用Cache技术来减少远程通信的频度，减少了通信延迟以及对共享数据的访问冲突。

➢ 消息传递通信机制的主要优点

- 硬件较简单。
- 通信是显式的，因此更容易搞清楚何时发生通信以及通信开销是多少。
- 显式通信可以让编程者重点注意并行计算的主要通信开销，使之有可能开发出结构更好、性能更高的并行程序。
- 同步很自然地与发送消息相关联，能减少不当的同步带来错误的可能性。

- 可在支持上面任何一种通信机制的硬件模型上建立所需的通信模式平台。
  - 在共享存储器上支持消息传递相对简单。
  - 在消息传递的硬件上支持共享存储器就困难得多。所有对共享存储器的访问均要求操作系统提供地址转换和存储保护功能，即将存储器访问转换为消息的发送和接收。

6.1.3 并行处理面临的挑战

并行处理面临着两个重要的挑战

- 程序中的并行性有限
- 相对较大的通信开销

$$\text{系统加速比} = \frac{1}{(1-\text{可加速部分比例}) + \dfrac{\text{可加速部分比例}}{\text{理论加速比}}}$$

- 第一个挑战

有限的并行性使计算机要达到很高的加速比十分困难。

例10.1 假设想用100个处理器达到80的加速比，求原计算程序中串行部分最多可占多大的比例？

解 Amdahl定律为：

$$\text{加速比}=\frac{1}{\dfrac{\text{可加速部分比例}}{\text{理论加速比}}+(1-\text{可加速部分比例})}$$

$$80=\frac{1}{\dfrac{\text{并行比例}}{100}+(1-\text{并行比例})}$$

由上式可得：并行比例＝0.9975

2. 第二个挑战：多处理机中远程访问的延迟较大

- 在现有的机器中，处理器之间的数据通信大约需要50～1000个时钟周期。
- 主要取决于：

通信机制、互连网络的种类和机器的规模

- 在几种不同的共享存储器并行计算机中远程访问一个字的典型延迟

6.1 引 言

机器	通信机制	互连网络	处理机最大数量	典型远程存储器访问时间（ns）
Sun Starfire servers	SMP	多总线	64	500
SGI Origin 3000	NUMA	胖超立方体	512	500
Cray T3E	NUMA	3维环网	2048	300
HP V series	SMP	8×8交叉开关	32	1000
HP AlphaServer GS	SMP	开关总线	32	400

例6.2 假设有一台32台处理器的多处理机，对远程存储器访问时间为200ns。除了通信以外，假设所有其它访问均命中局部存储器。当发出一个远程请求时，本处理器挂起。处理器的时钟频率为2GHz，如果指令基本的CPI为0.5（设所有访存均命中Cache），求在没有远程访问的情况下和有0.2%的指令需要远程访问的情况下，前者比后者快多少?

解 有0.2%远程访问的机器的实际CPI为：

CPI＝基本CPI＋远程访问率×远程访问开销

＝0.5＋0.2%×远程访问开销

远程访问开销为：

远程访问时间/时钟周期时间＝200ns/0.5ns＝400个时钟周期

∴ CPI＝0.5＋0.2%×400＝1.3

因此在没有远程访问的情况下的机器速度是有0.2%远程访问的机器速度的1.3/0.5=2.6倍。

➢ 问题的解决

- 并行性不足： 采用并行性更好的算法
- 远程访问延迟的降低：靠系统结构支持和编程技术

3. 在并行处理中，影响性能（负载平衡、同步和存储器访问延迟等）的关键因素常依赖于：

应用程序的高层特性

如数据的分配，并行算法的结构以及在空间和时间上对数据的访问模式等。

➢ 依据应用特点可把多机工作负载大致分成两类：

- 单个程序在多处理机上的并行工作负载
- 多个程序在多处理机上的并行工作负载

4. 并行程序的计算／通信比率

- 反映并行程序性能的一个重要的度量：

  计算与通信的比率

- 计算／通信比率随着处理数据规模的增大而增加；
- 随着处理器数目的增加而减少。

6.2 对称式共享存储器系统结构

- 多个处理器共享一个存储器。
- 当处理机规模较小时，这种计算机十分经济。
- 近些年，能在一个单独的芯片上实现2～8个处理器核。

例如：Sun公司 2006年 T1 8核的多处理器

- 支持对共享数据和私有数据的Cache缓存

私有数据供一个单独的处理器使用，而共享数据则是供多个处理器使用。

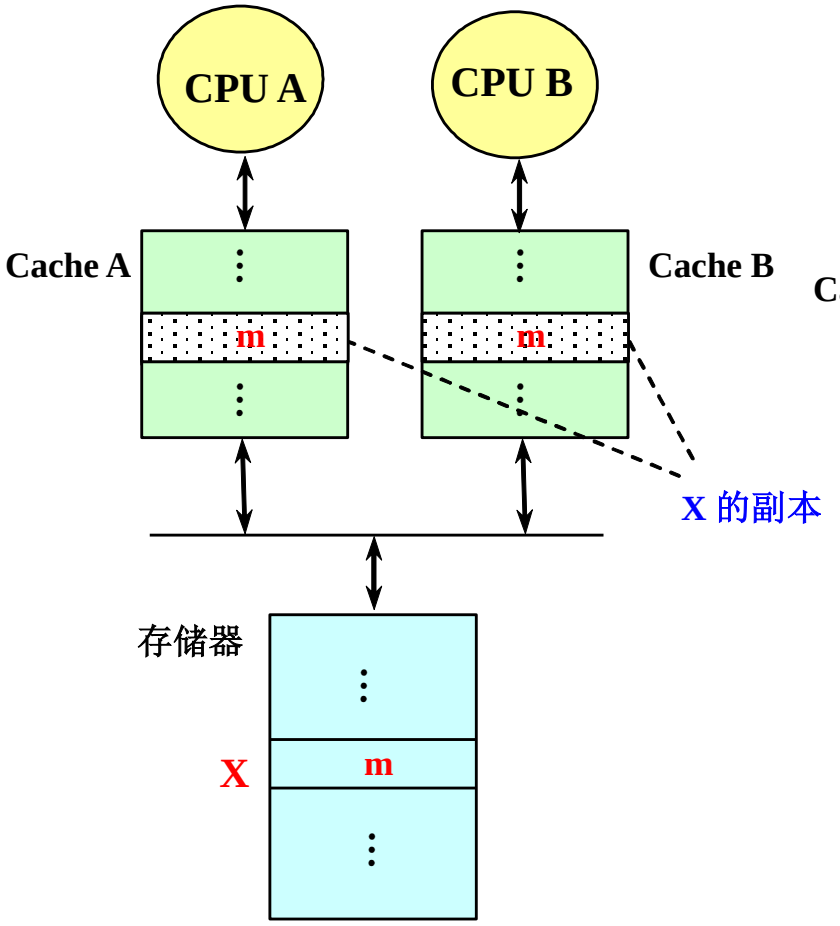
- 共享数据进入Cache产生了一个新的问题

Cache的一致性问题

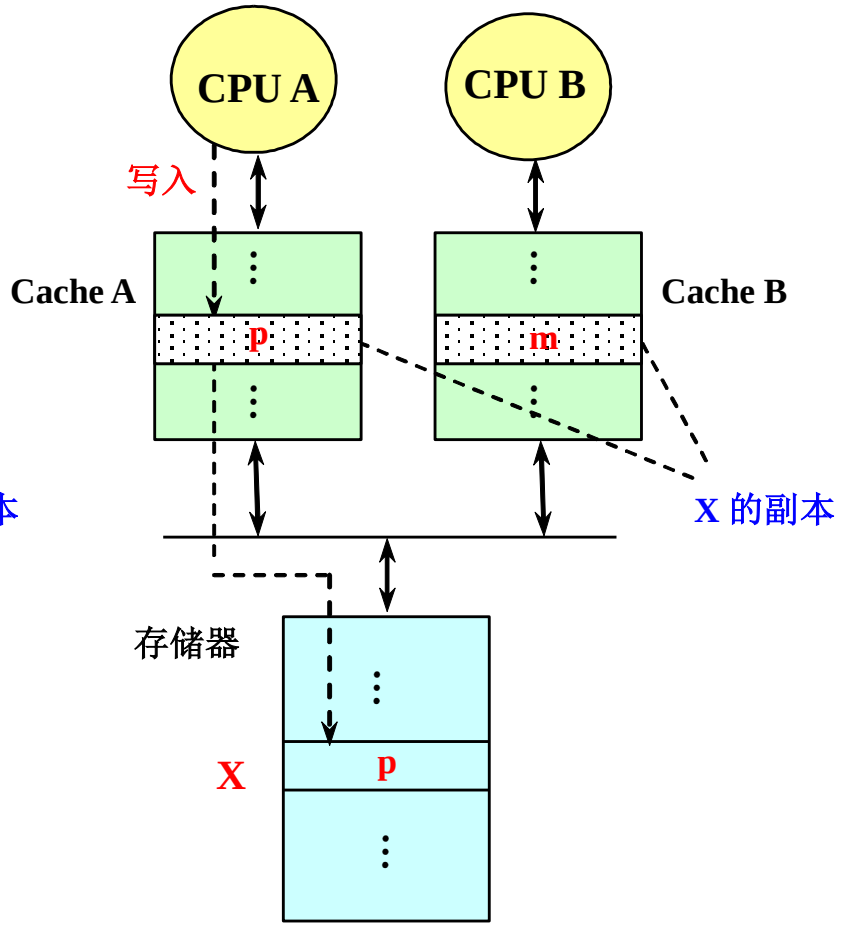
6.2.1 多处理机Cache一致性

- 多处理机的Cache一致性问题
  - 允许共享数据进入Cache，就可能出现多个处理器的Cache中都有同一存储块的副本，
  - 当其中某个处理器对其Cache中的数据进行修改后，就会使得其Cache中的数据与其他Cache中的数据不一致。

例　由两个处理器（A和B）读写引起的Cache一致性问题



（a） CPU A 写入前



（b） CPU A 将 p 写入 X，p≠m

假设初始条件下各个Cache无X值，X单元值为1，并且假设是写直达方式的Cache ， X单元被A写完后，A的Cache和存储器中均为新值，但B的Cache中不是，如果B处理器读Cache它将得到1。

时间 事件	CPU A Cache 内容	CPU A Cache 内容	X单元存储器 内容
0			1
1 CPUA读X	1		1
2 CPUB读X	1	1	1
3 CPUA 将0存入X	0	1	0

2. 存储器的一致性

如果对某个数据项的任何读操作均可得到其最新写入的值，则认为这个存储系统是一致的。

- 存储系统行为的两个不同方面
  - What：读操作得到的是什么值
  - When：什么时候才能将已写入的值返回给读操作
- 需要满足以下条件
  - 处理器P对单元X进行一次写之后又对单元X进行读，读和写之间没有其它处理器对单元X进行写，则P读到的值总是前面写进去的值。

- 处理器P对单元X进行写之后，另一处理器Q对单元X进行读，读和写之间无其它写，则Q读到的值应为P写进去的值。
- 对同一单元的写是串行化的，即任意两个处理器对同一单元的两次写，从各个处理器的角度看来顺序都是相同的。（写串行化 ）

➢ 在后面的讨论中，我们假设：

- 直到所有的处理器均看到了写的结果，这个写操作才算完成；
- 处理器的任何访存均不能改变写的顺序。就是说，允许处理器对读进行重排序，但必须以程序规定的顺序进行写。

6.2.2 实现一致性的基本方案

在一致的多处理机中，Cache提供两种功能：

- 共享数据的迁移

  减少了对远程共享数据的访问延迟，也减少了对共享存储器带宽的要求。

- 共享数据的复制

  不仅减少了访问共享数据的延迟，也减少了访问共享数据所产生的冲突。

一般情况下，小规模多处理机是采用硬件的方法来实现Cache的一致性。

6.2 对称式共享存储器系统结构

- Cache一致性协议

  在多个处理器中用来维护一致性的协议。

  - 关键：跟踪记录共享数据块的状态
  - 两类协议（采用不同的技术跟踪共享数据的状态）

  • 目录式协议（directory）

  物理存储器中数据块的共享状态被保存在一个称为目录的地方。

  • 监听式协议（snooping）

  - 每个Cache除了包含物理存储器中块的数据拷贝之外，也保存着各个块的共享状态信息。

– Cache通常连在共享存储器的总线上，当某个Cache需要访问存储器时，它会把请求放到总线上广播出去，其他各个Cache控制器通过监听总线（它们一直在监听）来判断它们是否有总线上请求的数据块。如果有，就进行相应的操作。

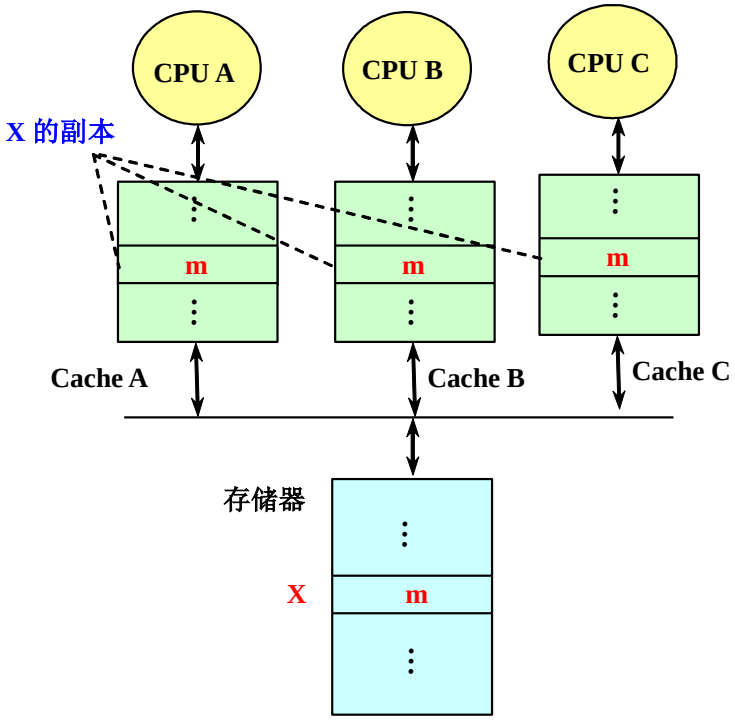
2. 采用两种方法来解决Cache一致性问题。

➢ 写作废协议

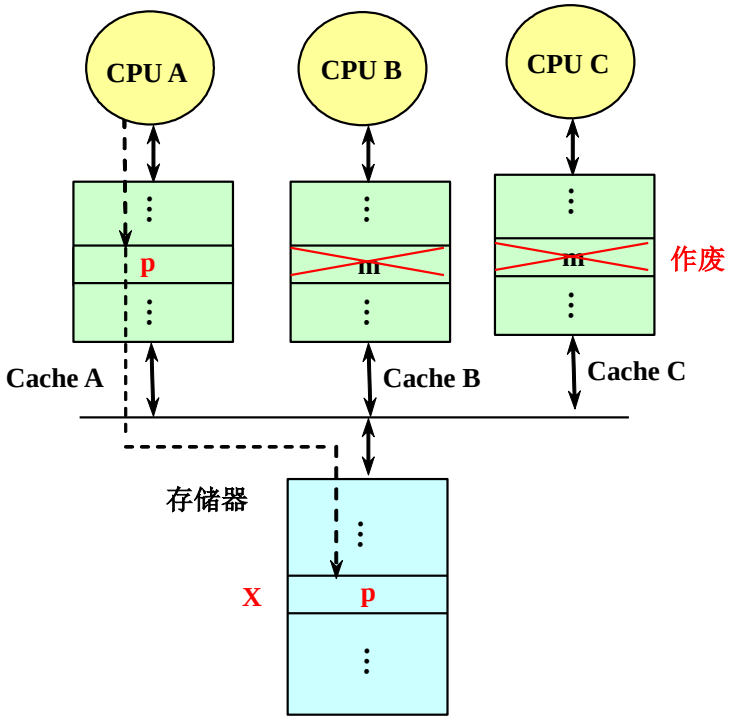
在处理器对某个数据项进行写入之前，保证它拥有对该数据项的唯一的访问权。（作废其它的副本）

例 监听总线、写作废协议举例（采用写直达法）

初始状态：CPU A、CPU B、CPU C都有X的副本。在CPU A要对X进行写入时，需先作废CPU B和CPU C中的副本，然后再将p写入Cache A中的副本，同时用该数据更新主存单元X。



（a） CPU A 写入前



（b） CPU A 将 p 写入 X 后，作废其他 Cache 中的副本

3．两种协议

在写回Cache的条件下，监听总线中写作废协议的实现

处理器行为	总线行为	CPU A Cache 内容	CPU B Cache 内容	主存X单元内容
				0
CPU A 读X	Cache 失效	0		0
CPU B 读X	Cache失效	0	0	0
CPU A 将X单元写1	作废X单元	1		0
CPU B 读X	Cache 失效	1	1	1

6.2 对称式共享存储器系统结构

- 写更新协议

当一个处理器对某数据项进行写入时，通过广播使其它Cache中所有对应于该数据项的副本进行更新。

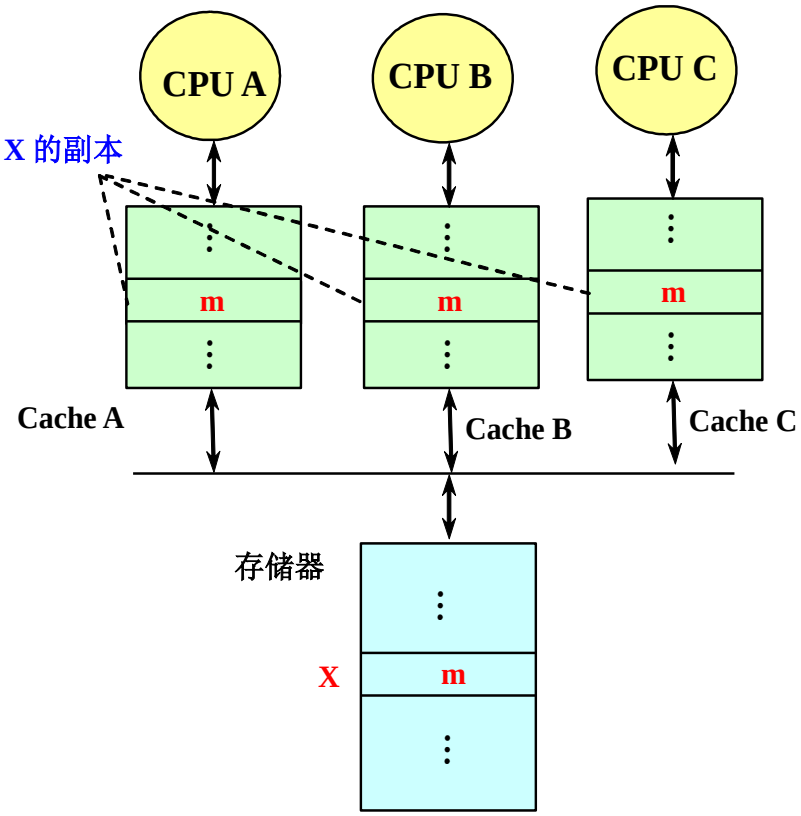
例 监听总线、写更新协议举例（采用写直达法）

假设：3个Cache都有X的副本。

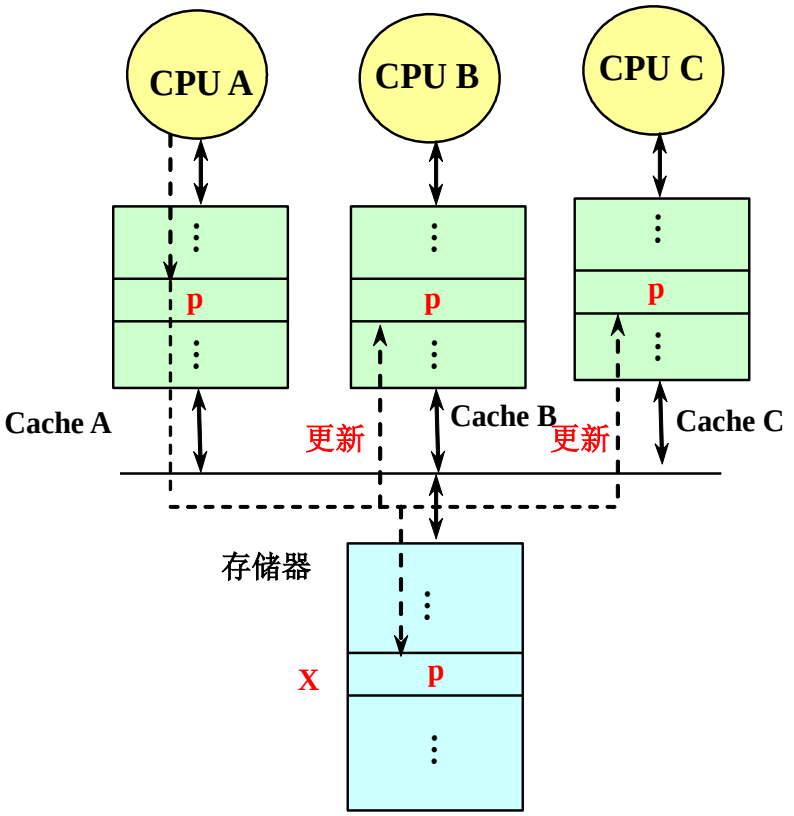
当CPU A将数据p写入Cache A中的副本时，将p广播给所有的Cache，这些Cache用p更新其中的副本。

由于这里是采用写直达法，所以CPU A还要将p写入存储器中的X。如果采用写回法，则不需要写入存储器。

6.2 对称式共享存储器系统结构



（a） CPU A 写入前



（b） CPU A 将 p 写入 X 后，更新其他 Cache 中的副本

3．两种协议

在写回Cache的条件下，监听总线中写作废协议的实现

处理器行为	总线行为	CPU A Cache 内容	CPU B Cache 内容	主存X单元内容
				0
CPU A 读X	Cache 失效	0		0
CPU B 读X	Cache失效	0	0	0
CPU A 将X单元写1	作废X单元	1		0
CPU B 读X	Cache 失效	1	1	1

➢ 写更新和写作废协议性能上的差别主要来自：

- 在对同一个数据进行多次写操作而中间无读操作的情况下，写更新协议需进行多次写广播操作，而写作废协议只需一次作废操作。
- 在对同一Cache块的多个字进行写操作的情况下，写更新协议对于每一个写操作都要进行一次广播，而写作废协议仅在对该块的第一次写时进行作废操作即可。 写作废是针对Cache块进行操作，而写更新则是针对字（或字节）进行。
- 考虑从一个处理器A进行写操作后到另一个处理器B能读到该写入数据之间的延迟时间。

写更新协议的延迟时间较小。

6.2.3 监听协议的实现

小规模多处理机中实现写作废协议的关键是利用总线进行作废操作。

当某个处理器进行写数据时，必须先获得总线的控制权，然后将要作废的数据块的地址放在总线上。

当Cache未命中时，除了将其他处理器相应的Cache数据块作废外，还要从存储器取出该数据块。对于写直达(write --through)Cache ，因为所有写的数据同时被写回主存，则从主存中总可以取到最新的数据值。

对于写回Cache,数据的最新值会困难一些，因为最新值可能在某个Cache中，也可能在主存中。

Cache 中块的标志位可用于实现监听过程

每个块的有效位（valid）使作废机制的实现较为容易。由写作废或其他事件引起的失效处理很简单，只需将该位设置为无效即可。

每个Cache块加一个特殊的状态位来说明它是否为共享。拥有唯一Cache块拷贝的处理器通常称为这个Cache块的拥有者（Owner），处理器的写操作使自己成为对应Cache块的拥有者。当对共享块进行写时，本Cache将写作废的请求放在总线上，Cache块状态由共享（shared）变为非共享（unshared）或者专有（exclusive）。

基于总线一致性协议的实现通常采用在每个结点内嵌入一个Cache状态控制器，该控制器根据来自处理器或总线的请求，改变所选择的数据块的状态。当总线出现写失效时，任何具有该Cache数据块的处理器结点进行作废处理。

将写失效的检测、申请总线和接收响应作为一个单独的原子操作。

因为每次总线任务均要检查Cache的地址位，这可能与CPU对Cache的访问冲突。可通过下列两种技术之一降低冲突：复制标志位或采用多级包含Cache。

复制标志位可使CPU对Cache访问和监听并行进行。当然，Cache失效时，处理器要对Cache标志位进行操作。处理器与监听同时操作标志位发生冲突时，非抢先者将被挂起或被延迟。

多级包含Cache中的Cache由多级构成，处理器的大多数访问针对第一级Cache。如果监听命中第二级Cache，它就必须垄断对各级Cache的访问，更新状态并可能回写数据，这通常需要挂起处理器对Cache的访问。

6.3 分布式共享存储器体系结构

支持共享存储器这类机器中，存储器分布于各结点中，所有的结点通过网络互连。

由结点内的控制器根据地址决定数据是驻留在本地存储器还是驻留在远程存储器，如果是后者，则发送消息给远程存储器的控制器来访问数据。

这些系统都有Cache，为了解决一致性问题，规定共享数据不进入Cache，仅私有数据才能保存在Cache中。这种机制的优点是所需的硬件支持很少，因为远程访问存取量仅是一个字（或双字）而不是一个Cache块。

这种方法有几个主要的缺点：

(1)实现透明的软件Cache一致性的编译机制能力有限。基于编译的软件一致性目前还未实现。

（2）没有Cache一致性，机器就不能利用取出同一块中的多个字的开销接近于取一个字，最基本的困难是基于软件的一致性算法，不能足够准确地预测出实际共享数据块。

(3)诸如预取等延迟隐藏技术对于多个字的存取更为有效，比如针对一个Cache 块的预取。

1．基于目录的Cache一致性

目录协议也必须完成两个主要的操作：处理读失效和处理对共享干净（Clean）块的写（对一个共享块的写失效处理是这两个操作的简单结合）。

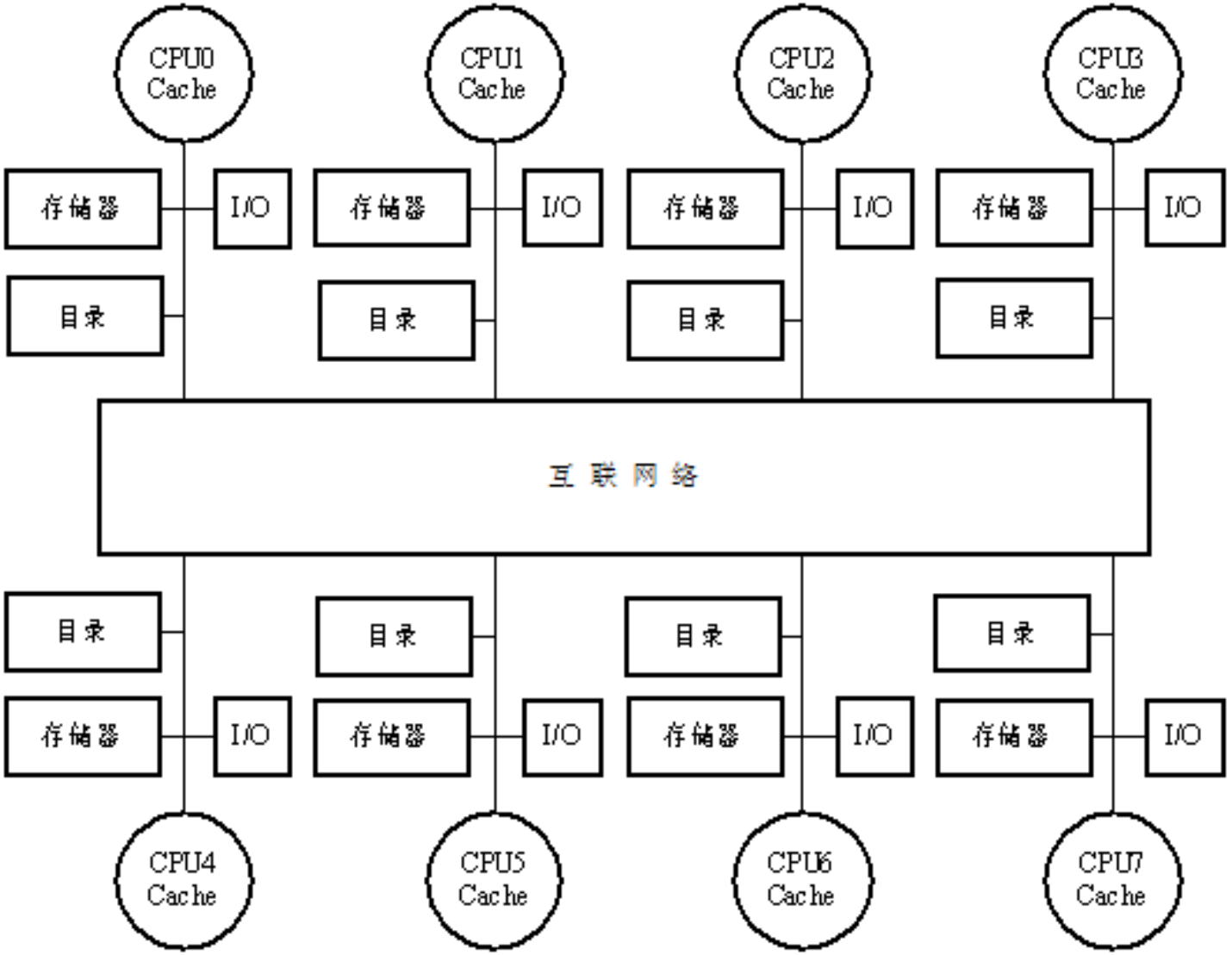
为实现这两种操作，目录必须跟踪每个Cache块的状态。Cache块状态有三种：

·共享（share）--在一个或多个处理器上具有这个块的拷贝，且主存中的值是最新值（所有Cache均相同）；

·未缓冲（uncached）--所有处理器的Cache都没有此块的拷贝；

·专有（exclusive ）--仅有一个处理器上有此块的拷贝，且已经对此块进行了写操作，而主存的拷贝仍是旧的；这个处理器称为此块的拥有者（owner）。

除了要跟踪每个Cache块的状态之外，由于写作废操作的需要，还必须记录共享此块的处理器信息。最简单的方法是对每个主存块设置一个位向量，当此块被共享时，每个位指出与之对应的处理器是否有此块的拷贝。当此块为专有时，可根据位向量来寻找此块的拥有者。


CPU0
Cache
CPU1
Cache
CPU2
Cache
CPU3
Cache
存储器
I/O
目录
互联网络
CPU4
Cache
CPU5
Cache
CPU6
Cache
CPU7
Cache

- （1）当一个块处于未缓冲状态时，对此块发出的请求及处理操作为：

读失效----将存储器数据送往请求方处理器，且本处理器成为此块的唯一共享结点，本块的状态转换为共享。

写失效----将存储器数据送往请求方处理器，此块成为专有，表示仅存在此块的唯一有效拷贝，共享集合仅有该处理器，这标识出了拥有者。

（2）当一个块是共享状态时，存储器中的数据是其当前最新值，对此块发出的请求及处理操作为：

读失效----将存储器数据送往请求方处理器，并将其加人共享集合。

写失效— 将数据送往请求方处理器，对共享集合中所有的处理器发送写作废消息，且将共享集合置为仅含有此处理器，本块状态变为专有。

（3）当某块处于专有状态时，本块的最新值保存在共享集合指出的拥有者处理器中，从而有三种可能的目录请求。

- 读失效----将“取数据”的消息发往拥有者处理器，使该块的状态转变为共享，并将数据送回目录结点写人存储器，进而把该数据返送给请求方处理器，将请求方处理器加人共享集合。此时共享集合中仍保留原拥有者处理器（因为它仍有一个可读的拷贝）。

写失效----本块就将有一个新的拥有者。给旧拥有者处理器发送消息，要求其将数据块送回目录结点，从而再送到请求方处理器，使之成为新的拥有者，并设置新拥有者的标志。此块的状态仍旧是专有。

数据写回----拥有者处理器的Cache 要替换此块时必须将其写回，从而使存储器中有最新拷则宿主结点实际上成为拥有者，此块就成为非共享，共享集合为空。

6.4 互连网络

互连网络是一种由开关元件按照一定的拓扑结构和控制方式构成的网络，用来实现计算机系统中结点之间的相互连接。

- 结点：处理器、存储模块或其它设备。
- 在拓扑上，互连网络为输入结点到输出结点之间的一组互连或映象。
- SIMD计算机和MIMD计算机的关键组成部分。
- 3大要素：互连结构，开关元件，控制方式

附加：互联函数

变量x：输入（设x=0，1，...，N－1）

函数f(x)：输出

通过数学表达式建立输入端号与输出端号的连接关系。即在互连函数f的作用下，输入端x连接到输出端f(x)。

- 互连函数反映了网络输入数组和输出数组之间对应的置换关系或排列关系。

（有时也称为置换函数或排列函数）

6.4.1互连网络的性能参数

- 静态网络由点和点直接相连而成，这种连接方式在程序执行过程中不会改变。
- 动态网络是用开关通道实现的，它可动态地改变结构，使其与用户程序中的通信要求匹配。
- 静态网络常用来实现一个系统中子系统或计算结点之间的固定连接。动态网络常用于集中式共享存储器多处理系统中。

6.4.1互连网络的性能参数

- 结点度：与结点相连接的边的数目称为结点度（node degree）。
- 链路或通道（边）是指网络中连接两个结点并传送数字信号的通路。
- 在单向通道的情况下，进人结点的通道数叫做入度（in degree），
- 从结点出来的通道数则称为出度（out degree），结点度是这两者之和。结点度应尽可能地小并保持恒定。

6.4.1互连网络的性能参数

- 网络直径：网络中任意两个结点间路径长度的最大值称为网络直径。网络直径应当尽可能地小。
- 等分宽度：在将某一网络切成相等两半的各种切法中，沿切口的最小通道边数称为通道等分宽度（channel bisection width ）。等分宽度是能很好地说明将网络等分的交界处最大通信带宽的一个参数。

结点间的线长（或通道长度），会影响信号的延迟、时钟扭斜和对功率的需要。

对于一个网络，如果从其中的任何一个结点看，拓扑结构都是一样的话，则称此网络为对称网络。对称网络较易实现，编制程序也较容易。

路由〔routing〕：在网络通信中对路径的选择与指定。

互连网络中路由功能较强将有利于减少数据交换所需的时间，因而能显著地改善系统的性能。

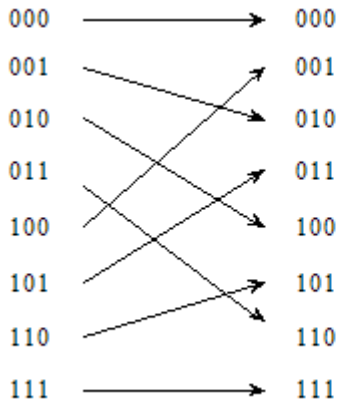
通常见到的处理单元之间的数据路由功能有移数、循环、置换（一对一）、广播（一对全体）、选播（多对多）、个人通信（一对多）、混洗、交换等。这些路由功能可在环形、网络形、超立方体以及多级网络上实现。

1．循环（rotation）-指对象的首尾相连。

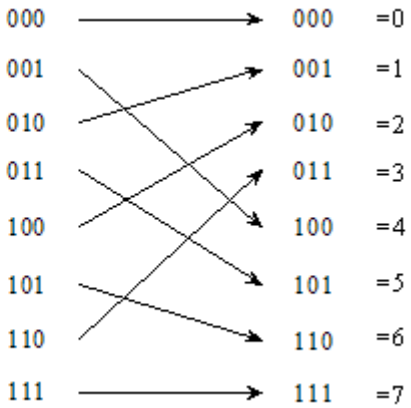
2．置换（permutation ）-指对象的重新排序。

可以用交叉开关来实现置换，也可以用一次或多次通过多级网络来实现某些置换，还可用移数或广播操作实现置换。

3．均匀混洗（shuffle）



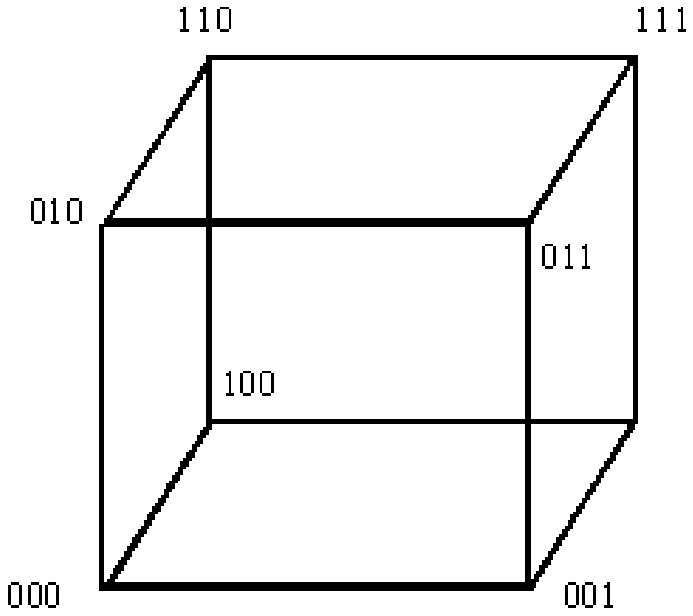
(a) 均匀混洗



(b)逆均匀混洗

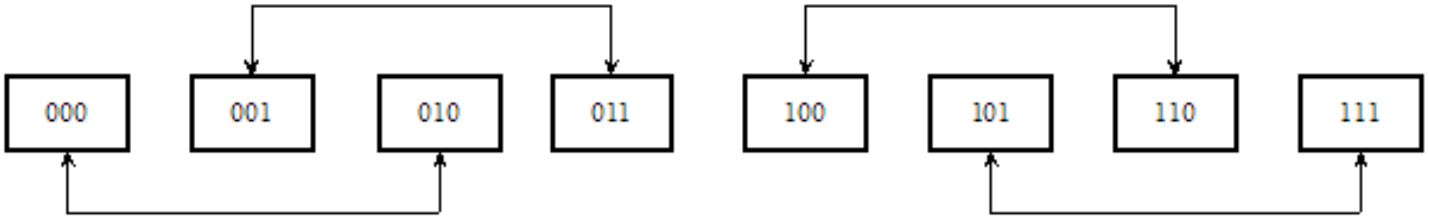
4. 超立方体路由功能

它有三种路由功能，可分别根据结点的二进制地址中的某一位来确定。

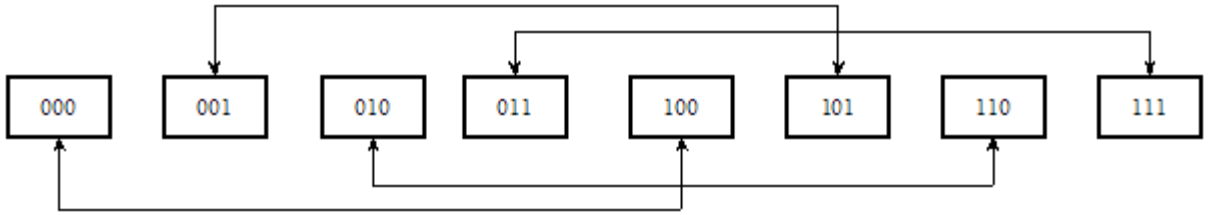


（a）3-立方体(三维二进制立方体网络)

（ b ）根据最低位C_0路由



（c）根据中间位C_1，路由



（d）根据最高位C_2路由

由二进制3-立方体确定的三种路由功能

5．广播和选播

广播是一种一对全体的映射，选播是一个子集到另一子集（多对多）的映射。消息传递型多处理机一般有广播信息机构，广播常常作为多处理机中的全局操作来处理。

影响互连网络性能的因素为：

（1）功能特性---网络如何支持路由、中断处理、同步、请求消息组合和一致性。

（2）网络时延---单位消息通过网络传送时最坏情况下的时间延迟。

（3）带宽---通过网络的最大数据传输率，用MBps 表示。

（4）硬件复杂性---诸如导线、开关、连接器、仲裁和接口逻辑等的造价。

（5）可扩展性---在增加机器资源使性能可扩展的情况下，网络具备模块化可扩展的能力。

6.4.2静态连接网络

1．线性阵列（liner array）这是一种一维的线性网络，其中N个结点用N-1个链路连成一行（图6.6（a））所示。

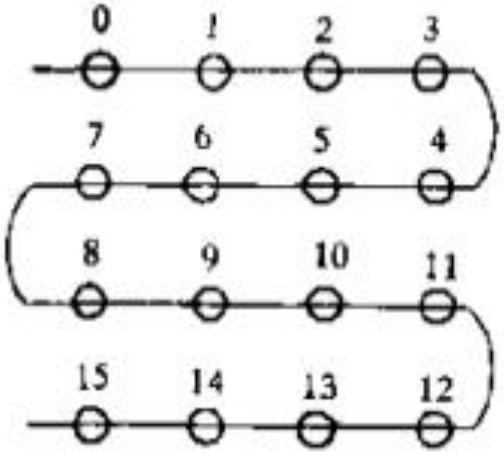
内部结点度为2，端结点度为1，直径为N-1。当N较大时，直径就比较长。等分宽度为1。

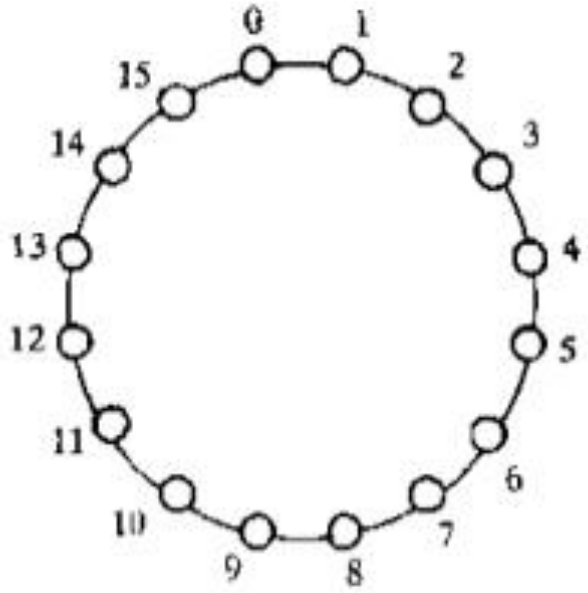
2．环和带弦环（chordal ring）环是用一条附加链路将线性阵列的两个端点连接起来而构成的（图6.6(b)）。

环可以单向工作，也可以双向工作。它是对称的，结点度是常数2的双向环的直径为N/2，单向环的直径是N。

- 对称
- 结点的度：2
- 双向环的直径：N/2
- 单向环的直径：N
- 环的等分宽度b=2

6. 4. 2静态连接网络

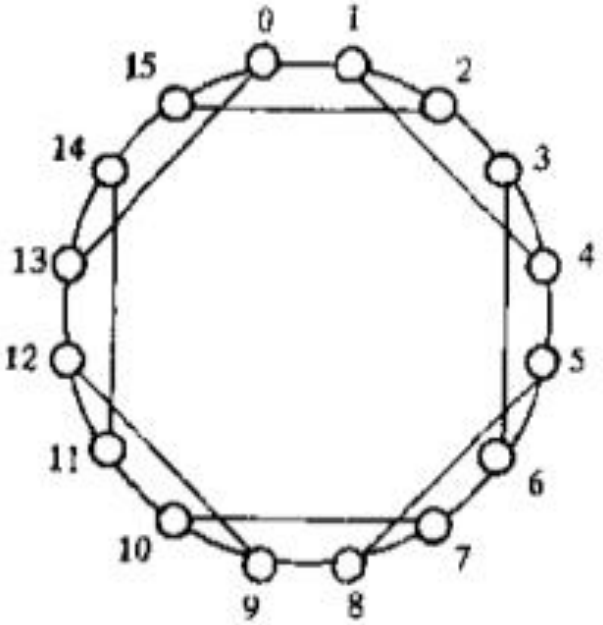


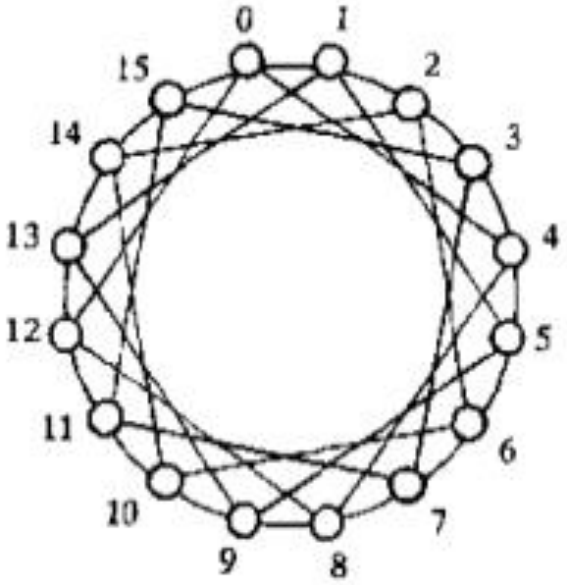


- （a） 线性阵列

（b） 环

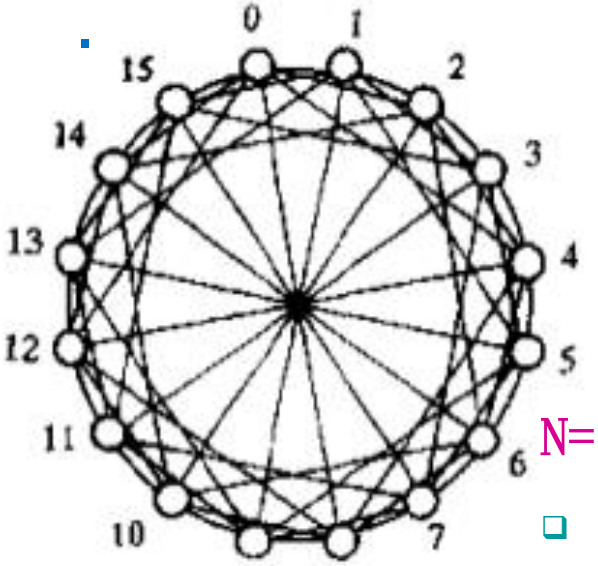
如果将结点度由2提高至3或4，即得到如图6.6(c）和6.6(d）所示的两种带弦环。





- （c） 度为3的带弦环　　（d） 度为4的带弦环(与Illiac网相同)

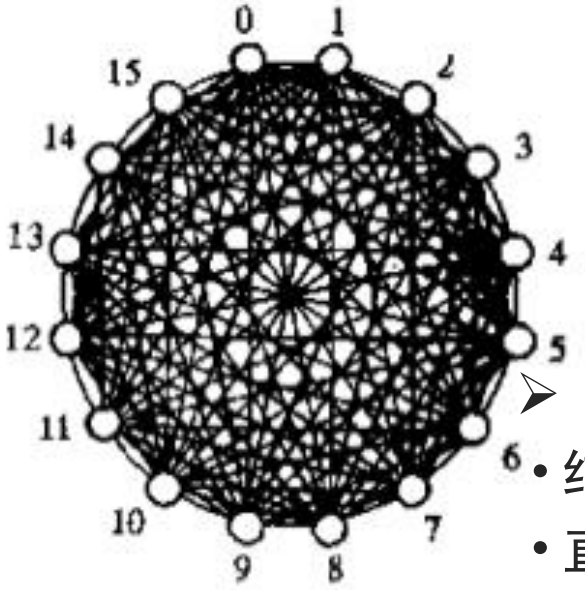
3．循环移数网络（barrel shifter）图6.6（e）所示的是个循环移数网络，其结点数N=16，它是通过在环上每个结点到所有与其距离为2的整数幂的结点之间都增加一条附加链而构成.



N=16

- 结点度：7
- 直径：2

• （e）循环移数网络



➢ 全连接

• 结点度：15

• 直径最短，为1。

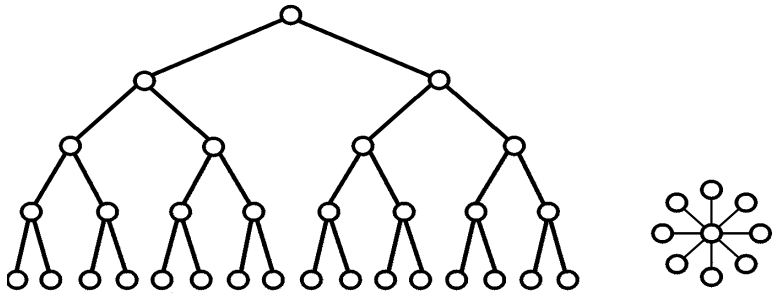
（f）全连接

4．树形和星形（tree and star）一般说来，一棵k层完全平衡的叉二树有$N=2^k-1$个结点。

- 最大结点度：3
- 直径：2(k-1)
- 等分宽度**b=1**

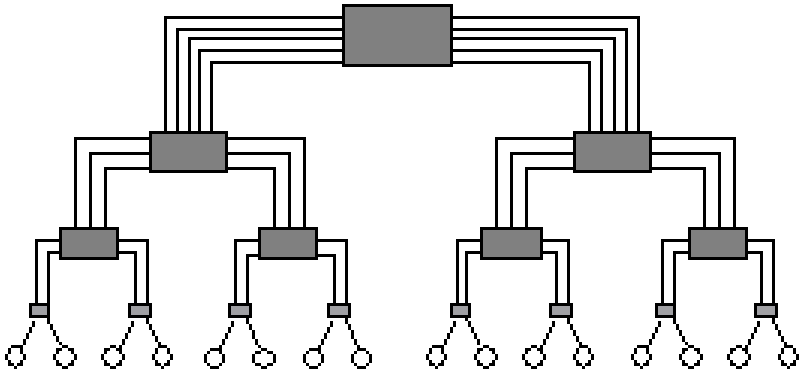
星形是一种2层树：

- 结点度较高，为N－1。
- 直径较小，是一常数2。等分宽度$b=\lfloor N/2 \rfloor$
- 可靠性比较差，只要中心结点出故障，整个系统就会瘫痪。



(a) 二叉树　　(b) 星形

5. 胖树形--胖树的通道宽度从叶结点往根结点上行方向逐渐增宽，它更像真实的树，愈靠近树根的枝杈愈粗。

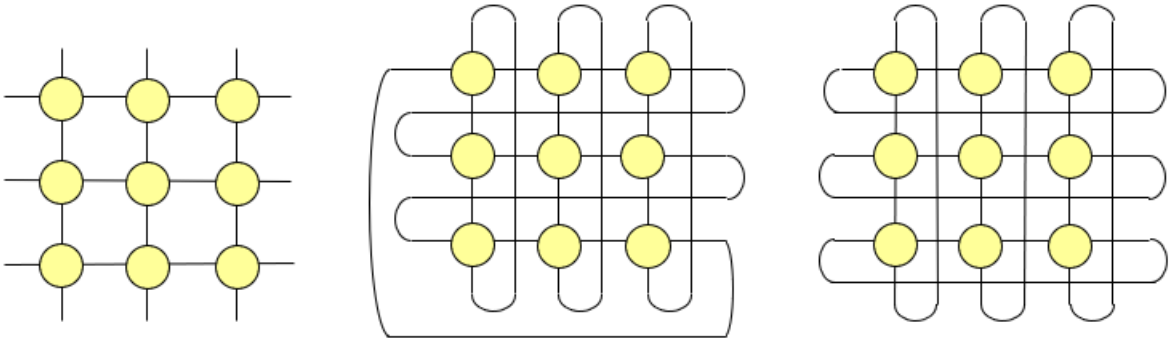


- 胖树形网络

6. 网格形和环形网

网格形网络一种比较流行的结构.

环形网可以看做是直径更短的另一种网格。这种拓扑结构将环形和网格组合在一起，并能向高维扩展。



（a）网格形　（b）Illiac网　（c）环网形

6. 4. 2静态连接网络

➢网格形

- 一个3×3的网格形网络
- 一个规模为N=n×n的2维网格形网络
  - –内部结点的度d=4
  - –边结点的度d=3
  - – 角结点的度d=2
  - –网络直径D=2（n-1）
  - –等分宽度b=n
- 一个由$N=n^k$ 个结点构成的k维网格形网络（每维n个结点）的内部结点度d=2k，网络直径D=k(n-1)。

- 环网形
  - 可看作是直径更短的另一种网格。
  - 把2维网格形网络的每一行的两个端结点连接起来，把每一列的两个端结点也连接起来。
  - 将环形和网格形组合在一起，并能向高维扩展。
  - 一个$n \times n$的环网形网
    - 结点度：4
    - 网络直径：$2 \times \lfloor n/2 \rfloor$
    - 等分宽度$b=2n$

➢Illiac网络

- 名称来源于采用了这种网络的Illiac IV计算机
- 把2维网格形网络的每一列的两个端结点连接起来，再把每一行的尾结点与下一行的头结点连接起来，并把最后一行的尾结点与第一行的头结点连接起来。
- 一个规模为n×n的Illiac网络
  - –所有结点的度d=4
  - –网络直径D=n-1

    Illiac网络的直径只有纯网格形网络直径的一半。
  - –等分宽度：2n

7．超立方体--是一种二元n-立方体结构，它已在nCUBE和CM-2 等系统中得到了实现。

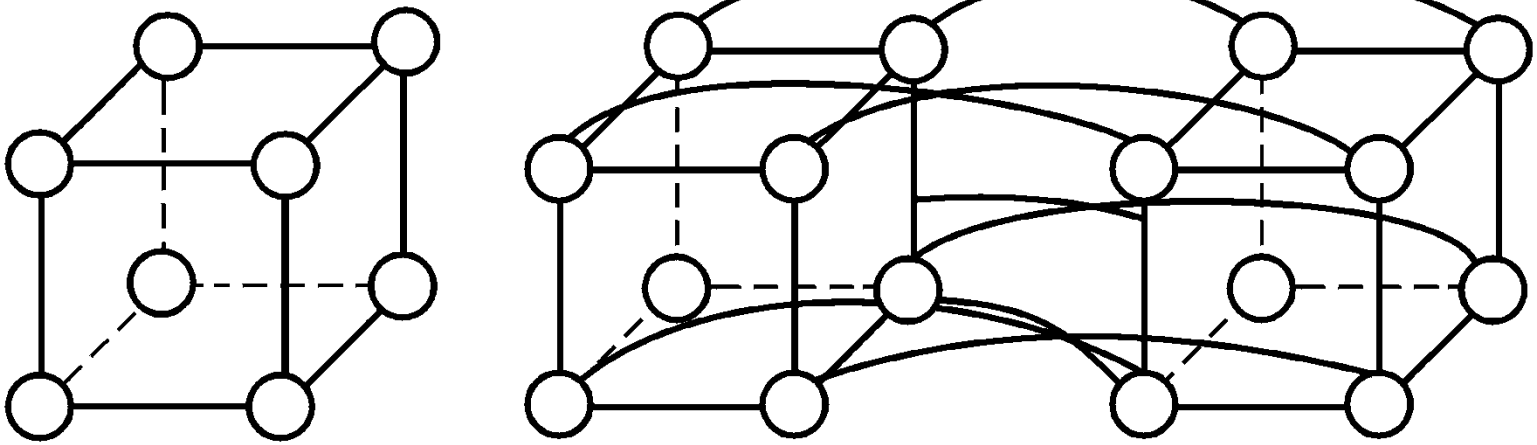
- 一般来说，一个二元n-立方体由$N=2^n$ 个结点组成，它们分布在n维上，每维有两个结点。

  例　8个结点的3维立方体

  4维立方体

- 为实现一个n-立方体，只要把两个（n－1）立方体中相对应的结点用链路连接起来即可。共需要2^{n-1}条链路。
- n-立方体中结点的度都是n，直径也是n，等分宽度为 $b=N/2$ 。

6. 4. 2静态连接网络

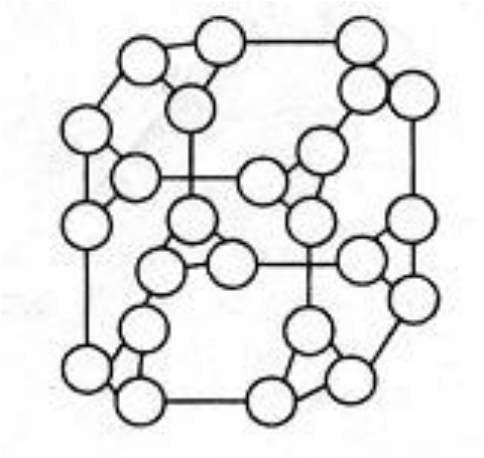


(a) 3–立方体　　(b) 由 2 个 3– 立方体组成的 4– 立方体

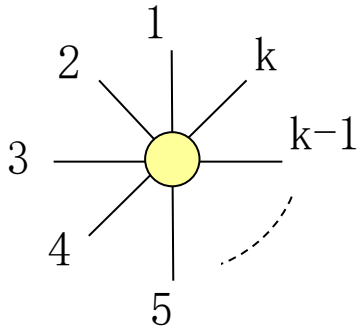
附加：带环立方体 （简称3-CCC）

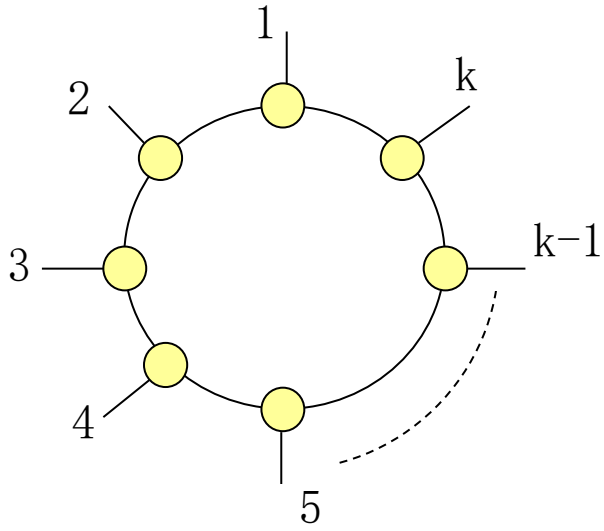
- 把3−立方体的每个结点换成一个由3个结点构成的环而形成的。
- 带环k-立方体（简称k-CCC）

- k-立方体的变形，它是通过用k个结点构成的环取代k-立方体中的每个结点而形成的。
- 网络规模为$N=k\times 2^k$
- 网络直径为$D=2k-1+\lfloor k/2 \rfloor$
  - 比k-立方体的直径大一倍
- 等分宽度为b=N/（2k）



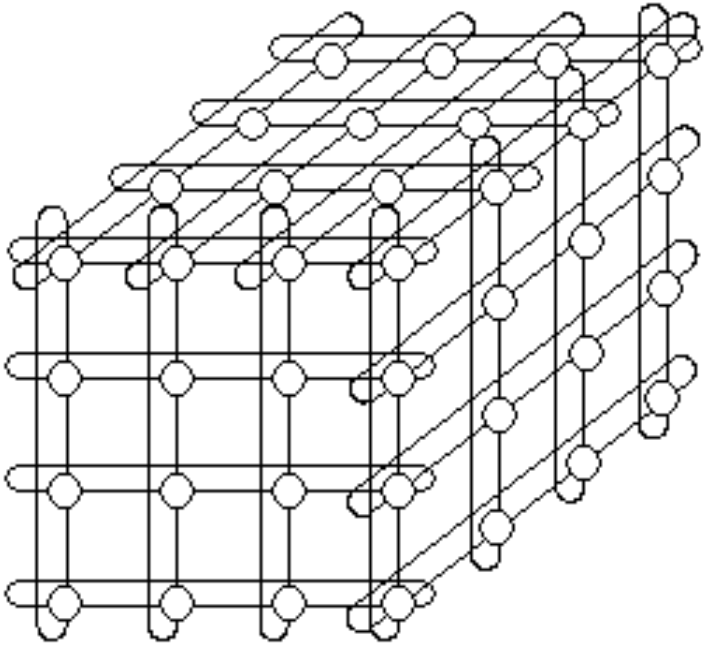
(a) 带环3-立方体





(b) 将k-立方体的每个结点用由k个结点的环来代替，组成带环k-立方体组成

8. k元n-立方体网络--环形、网络形、环网形、二元n-立方体（超立方体）等网络都是k元n-立方体网络系统的拓扑同构体。下图所示就是一种4元3-立方体网络.



k=4 和 *n*=3的 *k*元*n* – 立方体网络

k元n-立方体网络

- 环形、网格、环网形、二元n-立方体（超立方体）和Omega网络都是k元n-立方体网络系列的拓扑同构体。
- 在k元n-立方体网络中，参数n是立方体的维数，k是基数，即每一维上的结点个数。

  $N=k^n$，（$k=\sqrt[n]{N}$，$n=\log_k N$）
- k元n-立方体的结点可以用基数为k的n位地址$A=a_1a_2\ldots a_n$来表示。

  • 其中a_i表示该结点在第i维上的位置
- 通常把低维k元n-立方体称为环网，而把高维k元n-立方体称为超立方体。

静态互连网络特征一览表

网络类型	结点度d	网络直径D	链路数l	等分宽度B	对称性	网络规格说明
线线阵列	2	N-1	N-1	1	非	N个结点
环形	2	[N/2]	N	2	是	N个结点
全连接	N-1	1	N(N-1)/2	$(N/2)^2$	是	N个结点
二叉树	3	2(h-1)	N-1	1	非	树高h=[$\log_2 N$]
星形	N-1	2	N-1	[N/2]	非	N个结点
2D网格	4	2(r-1)	2N-2r	r	非	r×r网格，$r=\sqrt{N}$
Illiac网	4	r-1	2N	2r	非	与 $r=\sqrt{N}$ 的带弦环等效
2D环网	4	2[r/2]	2N	2r	是	r×r环网，$r=\sqrt{N}$
超立方体	n	n	nN/2	N/2	是	N个结点，n=[$\log_2 N$]（维数）
CCC	3	2k-1+[k/2]	3N/2	N/(2k)	是	N=k×2^k结点 环长k≥3
k元n-立方体	2n	n[k/2]	nN	$2k^{n-1}$	是	N=k^n个结点

6.4.3 动态连接网络

采用动态网络的多处理机的互连是在程序控制下实现的。

定时、开关和控制是动态互连网络的三个主要操作特征。

定时可以用同步方式，也可以用异步方式来进行。

单级网络（single-stage network）交叉开关和多端口存储器结构都属于单级网络。

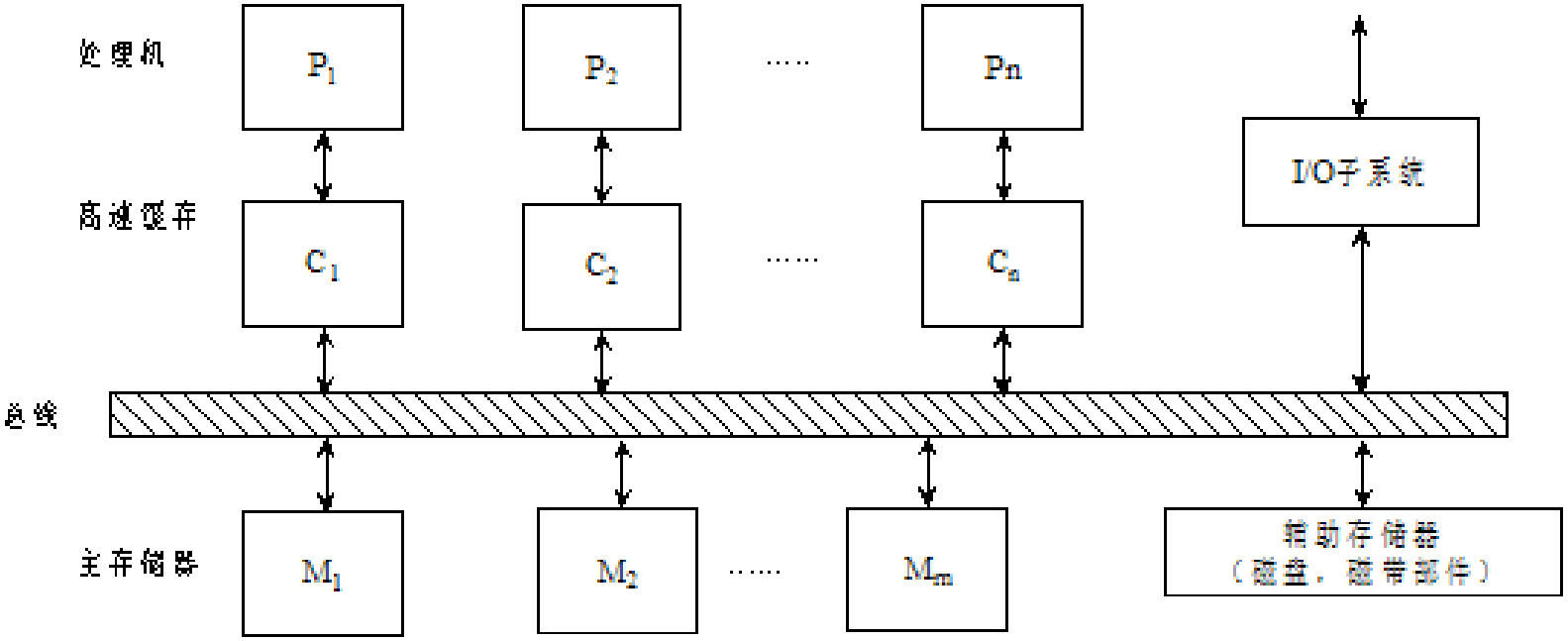
多级网络由一级以上的开关元件构成。

6.4.3动态连接网络

如果同时连接多个输人输出对时，可能会引起开关和通信链路使用上的冲突，这种多级网络称为阻塞网络（blocking network ）。

如果多级网络通过重新安排连接方式可以建立所有可能的输人输出之间的连接，则称之为非阻塞网络（non-blocking network）。

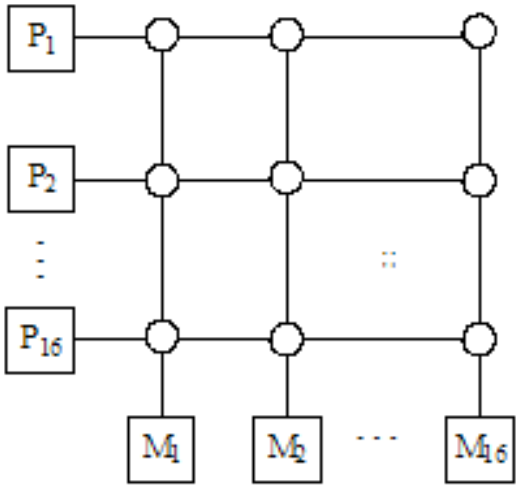
总线系统实际上是一组导线和插座，用于处理与总线相连的处理器、存储模块和外围设备间的数据业务



一种总线连接的多处理机系统

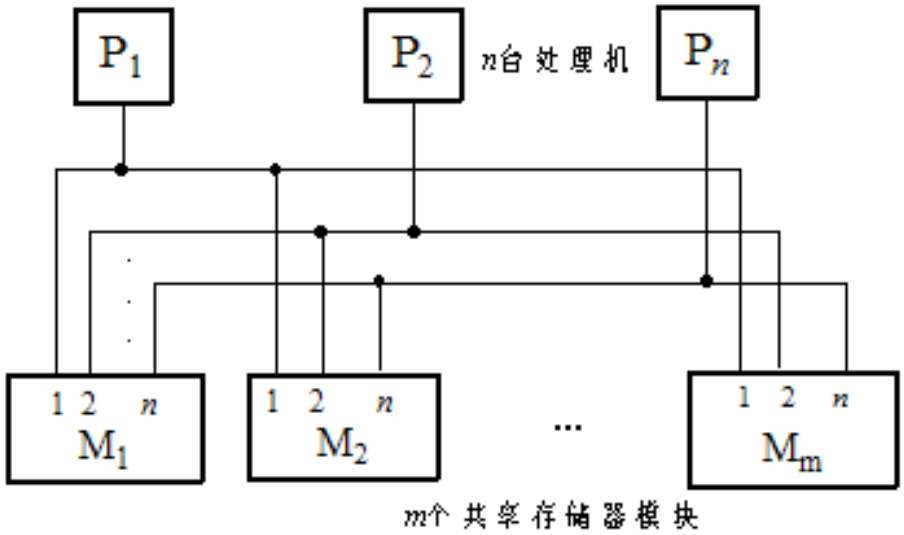
2. 交叉开关网络

在交叉开关网络中，每个输人端通过一个交叉点开关可以无阻塞地与一个空闲输出端相连。交叉开关网络是单级网络，它由交叉点上的一元开关构成。交叉网络主要用于中小型系统。



交叉开关网络是单级无阻塞置换网络。

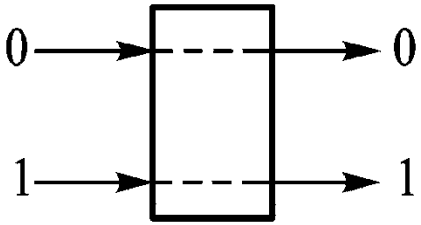
由于大型系统使用交叉开关网络的成本无法承受，所以许多大型的多处理机系统都采用多端口存储器结构。其主要思想是将所有交叉点仲裁逻辑和跟每个存储器模块有关的开关功能移到存储器控制器中



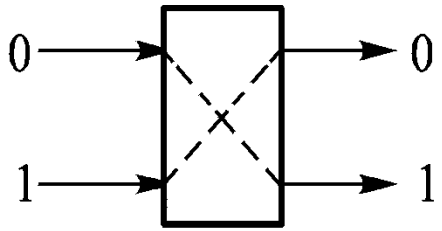
用于多处理机系统的多端口存储器结构

- 最简单的开关模块：2×2开关

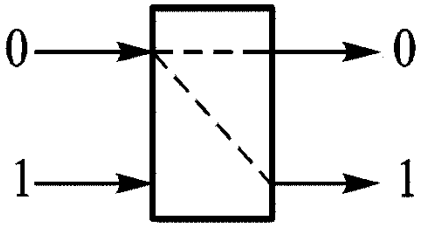
  2×2开关的4种连接方式



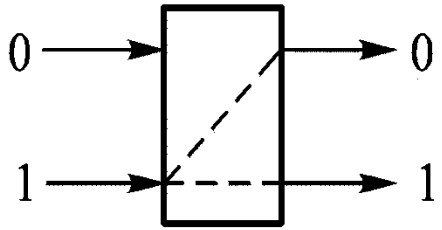
(a) 直送



(b)交叉

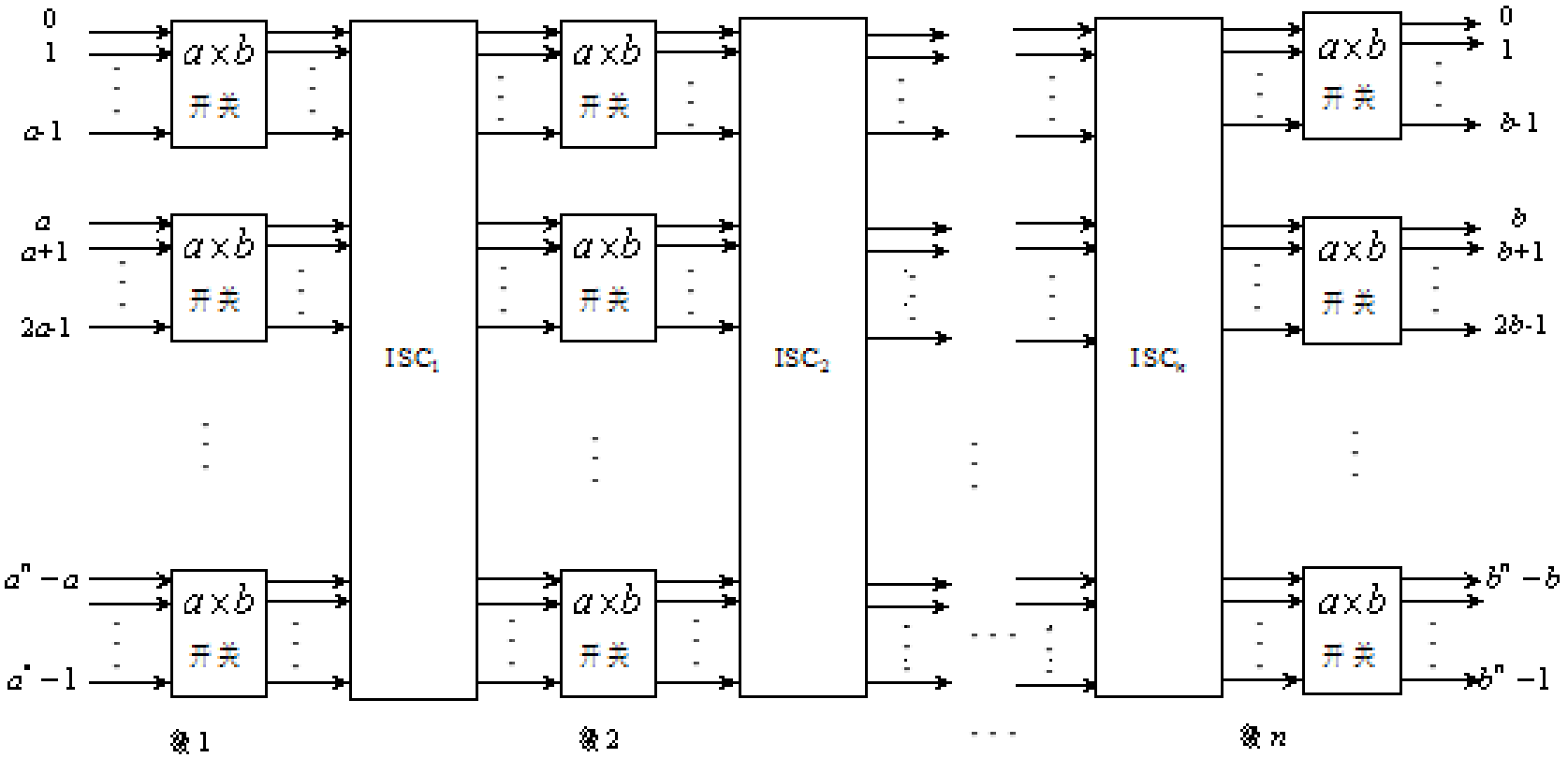


(c)上播



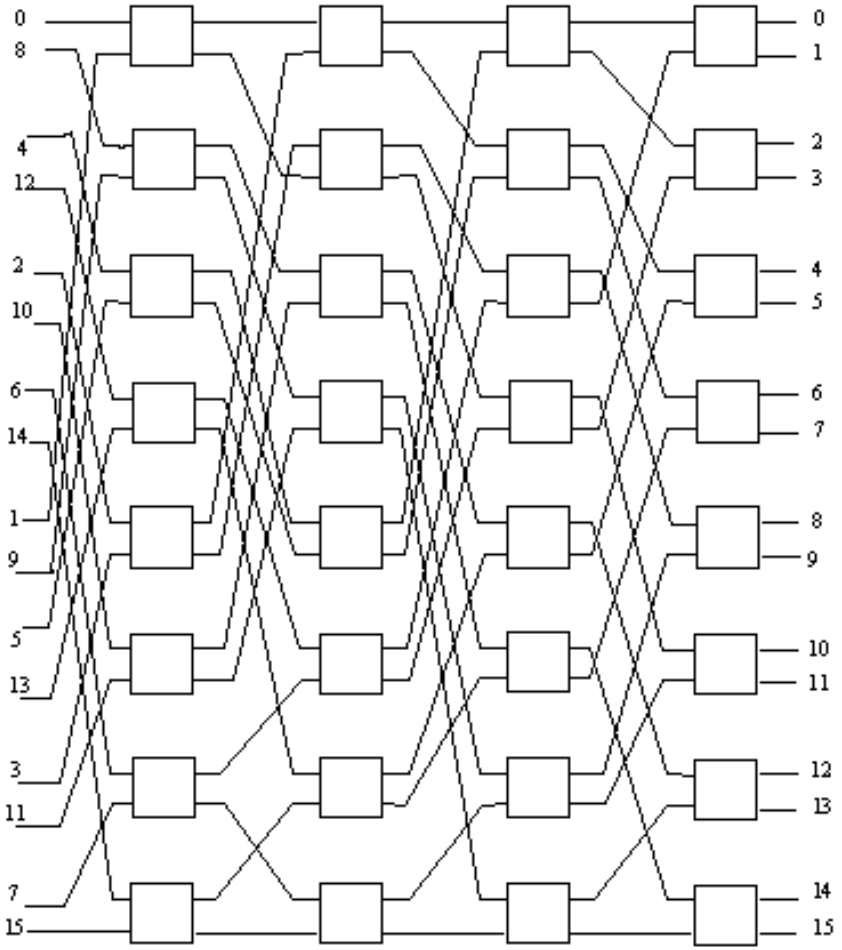
(d)下播

4. 多级网络



- 开关模式和级间构成的通用多级互联网络结构

4. 多级网络



(e) 16×16 Omega网络

- 用2×2开关和均匀混洗构成的16×16 Onega网络

6.5 基于多核的并行程序设计

- 受CPU主频、功耗、散热和超标量等技术复杂度的限制，以及多线程应用软件需求的驱动，微处理器架构发展到多核成为一种必然的趋势.
- 多核架构也是摩尔定律驱动的结果，出现多核处理器最根本的原因是人们对计算能力永无止境的追求。
- 尽管这些年来，处理器从来没有停止过前进的脚步，但每一次性能的突破，换来的只是对更高性能的需求，特别是在油气勘探、气象预报、虚拟现实、人工智能等高度依赖于计算能力的场合，对性能的渴求更迫切。

6.5 基于多核的并行程序设计

6.5.1 多核的需求

- 受CPU主频、功耗、散热和超标量等技术复杂度的限制，以及多线程应用软件需求的驱动，微处理器架构发展到多核成为一种必然的趋势.
- 多核架构也是摩尔定律驱动的结果，出现多核处理器最根本的原因是人们对计算能力永无止境的追求。
- 尽管这些年来，处理器从来没有停止过前进的脚步，但每一次性能的突破，换来的只是对更高性能的需求，特别是在油气勘探、气象预报、虚拟现实、人工智能等高度依赖于计算能力的场合，对性能的渴求更迫切。

6.6 基于多核的并行程序设计

- 多核给我们提供了更经济的计算能力。但是，这种能力能否善加利用，还要取决于软件。
- 如果不针对多核进行软件开发，不仅多核提供的强大计算能力得不到利用，相反还有可能不如单核CPU好。
- 针对多核和多线程的软件开发将是未来十年软件开发的主要挑战，即基于多核的并行程序设计：
  - 多核处理器的基本目的是通过多个任务的并行执行提高应用程序的性能；
  - 尽量分解成多个独立任务，每个任务实现为一个线程，从而将多个任务分布到多个计算核上执行，减少程序的执行时间。

6.6.1 并行编程模型概述

目前几种最重要的并行编程模型：

- 数据并行模型：编程级别比较高，编程相对简单，但它仅适用于数据并行问题；
- 消息传递模型：编程级别相对较低，但消息传递编程模型可以有更广泛的应用范围；
- 共享存储模型：采用多线程的方式，非常适合SMP共享内存多处理系统和多核处理器体系结构。

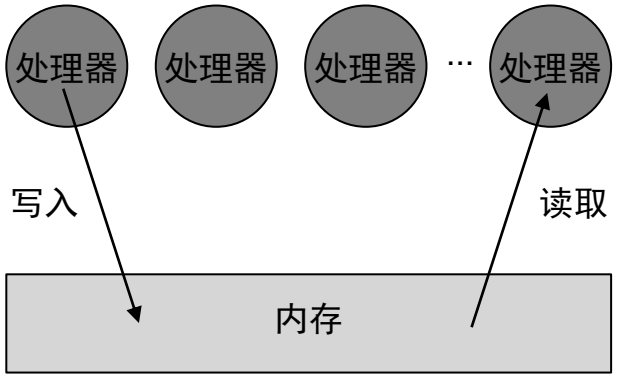
6.6.2 并行编程模型

数据并行和消息传递编程模式的对比

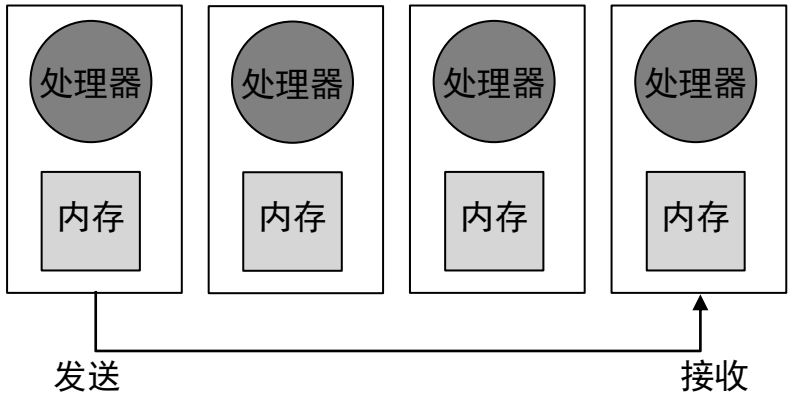
对比内容	数据并行	消息传递
编程级别	高	低
适用的并行机类型	SIMD/SPMD	SIMD/MIMD/SPMD/MPMD
执行效率	效率依赖于编译器	高
地址空间	单一	多个
存储类型	共享内存	分布式或共享内存
通信的实现	编译器负责	程序员负责
问题类	数据并行类问题	数据并行任务并行
目前状况	缺乏高效的编译器支持	使用广泛

6.6.2 共享存储模型和消息传递模型

共享存储和消息传递是两种被广泛应用的并行编程模型。其中，消息传递模型更多地用于较大型系统（数百到数千核），而共享存储模型用于较小型系统。



共享存储编程模型

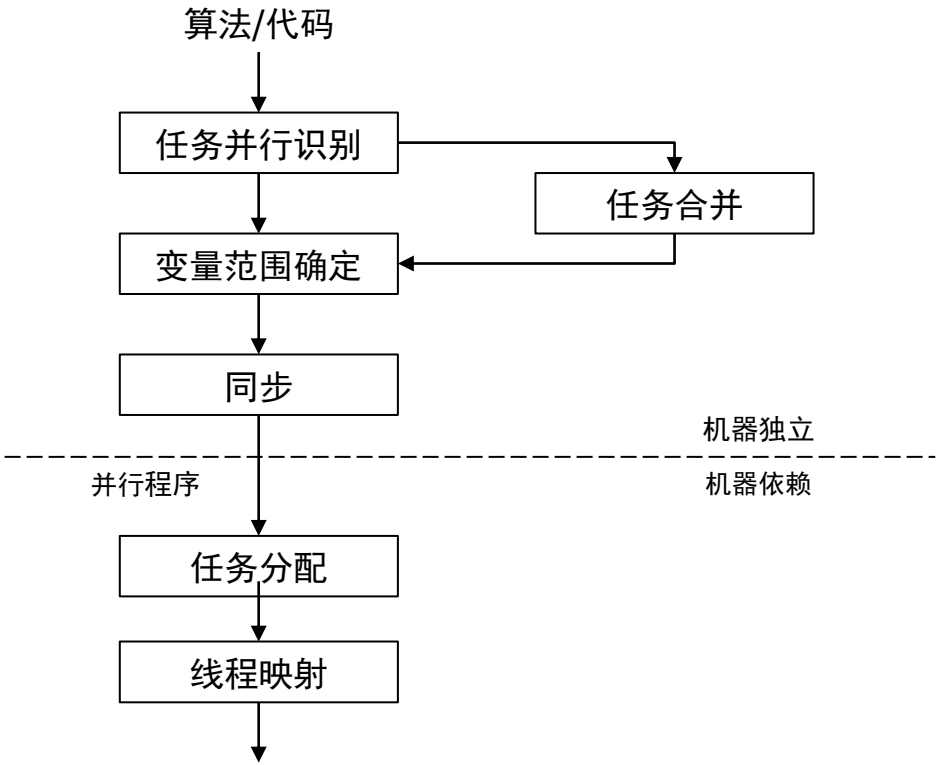


消息传递编程模型

6.6.2 共享存储模型和消息传递模型

共享存储并行编程包括六个步骤，分别是：

- 任务并行识别；
- 任务合并；
- 变量范围确定；
- 同步；
- 任务分配；
- 线程映射。



共享存储并行编程的步骤

6.6.3 并行语言

并行程序是通过并行语言来表达的，并行语言的产生主要有三种方式：

- 设计全新的并行语言；
- 扩展原来的串行语言的语法成分使它支持并行特征；
- 不改变串行语言仅为串行语言提供可调用的并行库。

6.6.4 并行算法

并行算法是给定并行模型的一种具体、明确的解决方法和步骤。

- 根据运算的基本对象的不同：
  - 数值并行算法（数值计算）
  - 非数值并行算法（符号计算）
- 根据进程之间的依赖关系
  - 同步并行算法（步调一致）
  - 异步并行算法（步调、进展互不相同）
  - 纯并行算法（各部分之间没有关系）

6.6.4 并行算法

3. 根据并行计算任务的大小：
   - 粗粒度并行算法（包含较长程序段和较大计算量）
   - 细粒度并行算法（包含较短程序段和较小计算量）
   - 介于二者之间的中粒度并行算法

从本质上说，不同的并行算法是根据问题类别的不同和并行机体系结构的特点产生出来的，一个好的并行算法要既能很好地匹配并行计算机硬件体系结构的特点，又能反映问题内在并行性。

6.7 多核编程实例

程序开发人员开发实际的并行程序主要方法是串行语言加并行库的扩展，其中比较典型的方法有两种：

- 共享存储的方法主要是采用多线程的方式，其主要程序开发环境就是已经成为事实工业标准的OpenMP，目前主要是商业编译器提供对该语言的支持；
- 消息传递开发则包括MPI和PVM等开源开发环境。

本小结重点介绍基于OpenMP的多核编程环境。

6.7 多核编程实例

1.OpenMP介绍

- OpenMP（Open Multi-Processing）是一套支持跨平台共享内存方式的多线程并发的编程API，使用C，C++和Fortran语言，可以在大多数的处理器体系和操作系统中运行。
- OpenMP采用可移植的、可扩展的模型，为程序员提供了一个简单而灵活的开发平台。
- OpenMP提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的pragma来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。

OpenMP — 共享变量编程标准

- OpenMP源于ANSI X3H5,是共享存储系统编程工业标准，它是将基本串行语言（C或Fortran）使用制导指令、运行库和环境变量，按照标准将其并行化；制导指令提供对并行区域、工作共享、同步构造、数据共享和私有化支持，其添加过程就类似于进行显式并行程序设计
- OpenMP是基于线程的并行编程模型，它使用Fork-Join方式：主线程一直串行执行，遇到Fork时创建并行线程，执行完后进入Join又回到主线程继续执行。
- 事实上，所有OpenMP的并行化，都是通过使用嵌入到C或Fortran源代码中的编译制导语句来达到的。

- 编译指导语句
  - 在编译器编译程序的时候，会识别特定的注释，而这些特定的注释就包含着OpenMP程序的一些语义。

    #pragma omp <directive> [clause[[,] clause]…]

    其中directive部分就包含了具体的编译指导语句，包括parallel, for, parallel for, section, sections, single, master, critical, flush, ordered和atomic。
  - 将串行的程序逐步地改造成一个并行程序，达到增量更新程序的目的，减少程序编写人员一定的负担。
- 运行时库函数
  - OpenMP运行时函数库原本用以设置和获取执行环境相关的信息，它们当中也包含一系列用以同步的API。
  - 支持运行时对并行环境的改变和优化，给编程人员足够的灵活性来控制运行时的程序运行状况。

2.OpenMP特点

- 提供一组与平台无关的编译指导（**pragmas**）、指导命令（**directive**）、函数调用和环境变量
- 显式地指导编译器如何以及何时利用应用程序中的并行性。
  - 循环：在开始之前插入一条编译指导，使其以多线程执行。
  - 开发人员：认真考虑哪些循环应以多线程方式执行。

引例

```
#pragma omp parallel
{
  for  (i=0;  i<numPixels;i++)
    {
       psum[i]=
   (pRed[i]*0.2+pBlue[i]*0.3+pYellow[i]*0.5);
    }
}
```

- 说明：
  - #pragma omp parallel：生成多个线程执行{}内的代码。即：执行相同的代码
  - Open机制决定要创建多少个线程，以及如何管理
  - 隐藏细节

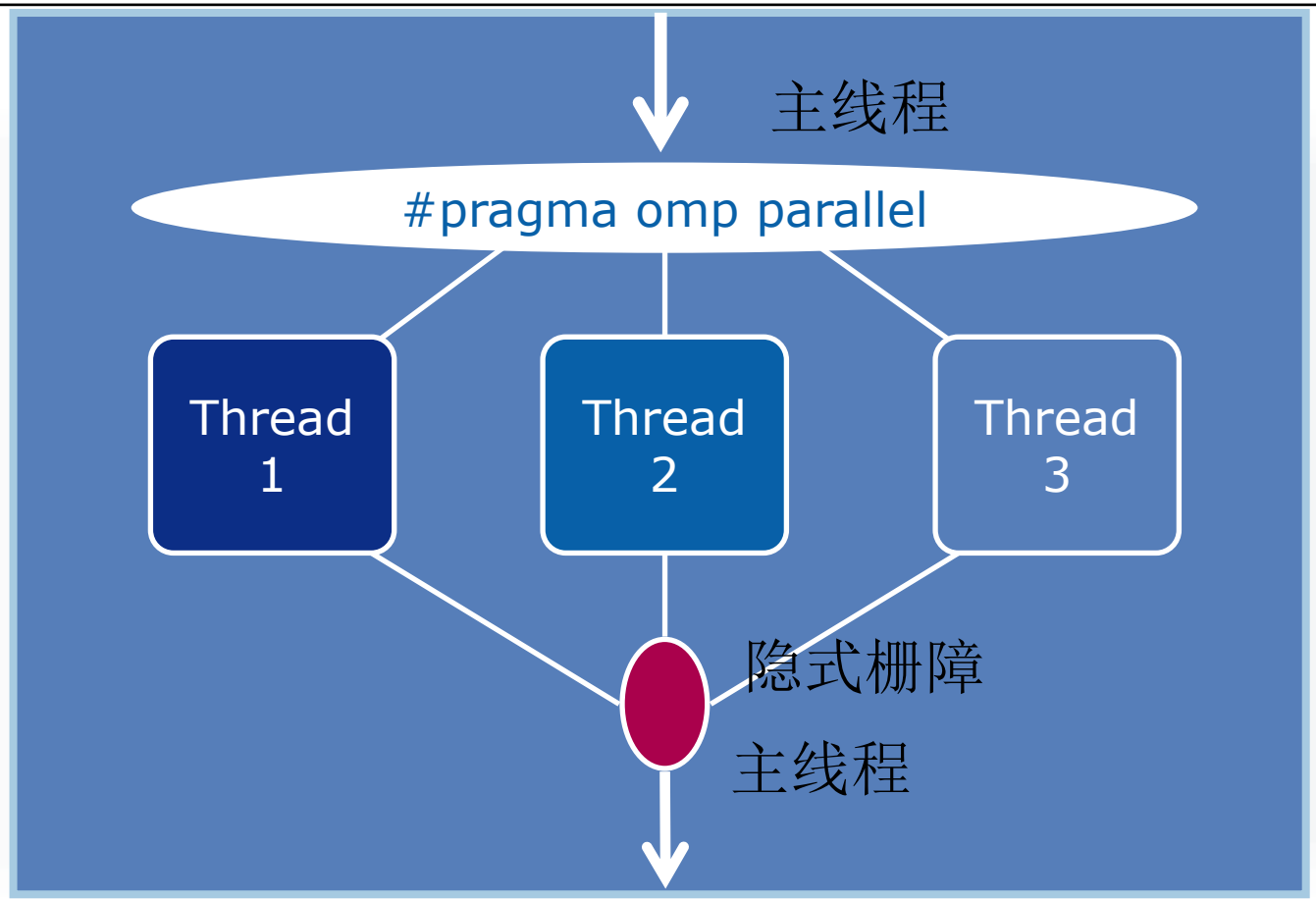
3.OpenMP 基本语法

- 在c/c++中，语法格式：

```
#pragma omp construct [clause [clause]…]
```

-

3.1parallel——并行执行



3.2 parallel for ——并行执行for

```
#pragma omp parallel for
    for  (i=0;  i<numPixels;i++)
    {
        psum[i]=
   (pRed[i]*0.2+pBlue[i]*0.3+pYellow[i]*0.5);

     }
```

说明:

- for结构使用任务分配机制（work-sharing），循环的各次迭代将被分配到多个线程上。

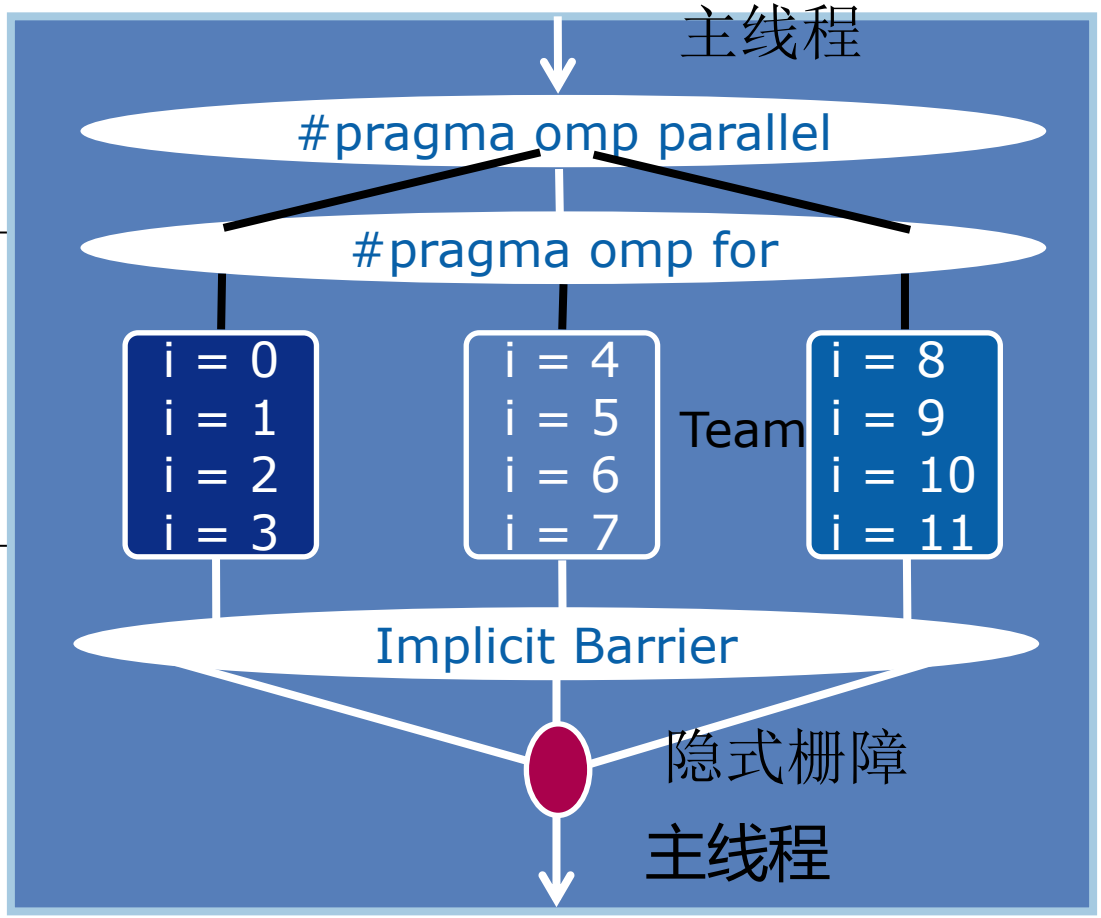
对于循环的五点约束

- **循环语句中的循环变量必须是有符号的整型。注：将在OpenMP3.0版本中取消**
- **循环语句中比较操作必须是这种形式：**loop_variable<,<=,>,>=loop_invariant_integer。
- **循环语句中的第三个表达式（for循环的步长）必须是整数加或整数减操作，加减的数值必须是一个循环变量（**loop_invariant_value)
- **如果比较操作为<,<=，那么循环变量的值在每次迭代时都必须增加；相反，为>,>=，那么循环变量的值在每次迭代时都必须减少。**
- **循环必须是单入口、单出口。循环内不允许有能够到达循环外的跳转语句，循环外不允许有能够进入到循环内的跳转语句。**

任务分配机制

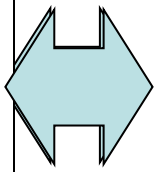
- 将工作分配给多个线程执行.

```
#pragma omp parallel
#pragma omp for
      for (i=0; i<12; i++)
            c[i] = a[i] + b[i];
```



pragmas的合并

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i< MAX; i++)
   {
     res[i] = huge();
   }
}
```



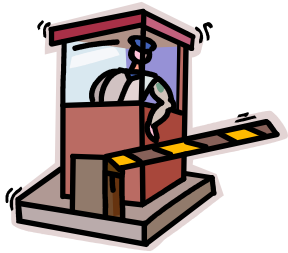
```
#pragma omp parallel for
for (i=0; i< MAX; i++)
{
  res[i] = huge();
}
```

- **代码段 1**
- **代码段 2**

功能是一样的，在什么（不是仅仅有for循环的语句）情况下必须用代码段 1

4.不共享的数据

- 除以下情况外，默认为共享:
  - 栈内数据不共享，每个线程内声明
  - 并行代码内定义的数据不共享
  - 循环变量不共享，迭代变量



5.数据共享

- 默认情况下，并行区中所有的变量都是多线程共享的，但上述情况例外。
- 修改数据的共享属性可以用*private*和*shared*子句

5.1 私有化

- private(variable_name,…)

```
void* work(float* c, int N)
{
        float x, y;
        int i;
    #pragma omp parallel for private(x,y)
        for (i=0; i<N; i++)
        {
                x = a[i];
                y = b[i];
                c[i] = x + y;
        }
}
```

5.2 共享化

shared(variable_name,…)

```
float dot_prod(float* a, float* b, int N)
{
        float sum = 0.0;
        #pragma omp parallel for shared(sum)
        for (int i=0; i<N; i++)
          {
                sum+ = a[i] * b[i];
           }
        return sum;
}
```

练习：

将下面代码用OpenMP语句并行化

```
for(k=0;k<80;k++)
{
    x=sin(k*2.0)*100+1;
   if(x>60)    x=x%60+1;
   printf(  “x %d =%d\n”  ,k,x);
}
```

练习：

将下面代码用OpenMP语句并行化

```
for(k=0;k<100;k++)
{
    x=sin(k*2.0)*100+1;
    if(x>60)  x=x%60+1;
    y=array[k];
    array[k] = do_work(y);
}
```

存在的问题

- private(x,y)
- **在并行段外的x,y的值不会被传到并行段内。**

问题

```
float dot_prod(float* a, float* b, int N)
{
        float sum = 0.0;
        #pragma omp parallel for
        for (int i=0; i<N; i++)
        {
                sum+ = a[i] * b[i];
        }
        return sum;
}
```

What is Wrong?

5.3 OpenMP Critical 结构

临界代码的定义

```
#pragma omp critical [(lock_name)]
```

```
float dot_prod(float* a, float* b, int N)
{
        float sum = 0.0;
        #pragma omp parallel for
        for (int i=0; i<N; i++)
        {
                #pragma omp critical
                sum+ = a[i] * b[i];
        }
        return sum;
}
```

5.3 OpenMP Critical 结构

```
float R1, R2;
#pragma omp parallel
{
        float A, B;
        #pragma omp for
        for (int i=0; i<niters; i++)
        {
                B = big_job(i);
                #pragma omp critical (x)
                  consum(B,&R1);
                A = bigger_job(i);
                #pragma omp critical (y)
                  consum(A, &R2);
        }
}
```

5.4 OpenMP Reduction 结构

```
float dot_prod(float* a, float* b, int N)
{
        float sum = 0.0;
        #pragma omp parallel for
        for (int i=0; i<N; i++)
        {
                #pragma omp critical
                sum+ = a[i] * b[i];
        }
        return sum;
}
```

reduction(operator:variable_list)

5.4 OpenMP Reduction 结构

```
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
{
    sum+ = a[i] * b[i];
}
```

- 为每一个线程创建一个变量sum的副本
- 并行循环结束后，每一个sum的副本将会被合并
- 更新主线程的全局变量sum的值

其他常用 OpenMP pragmas

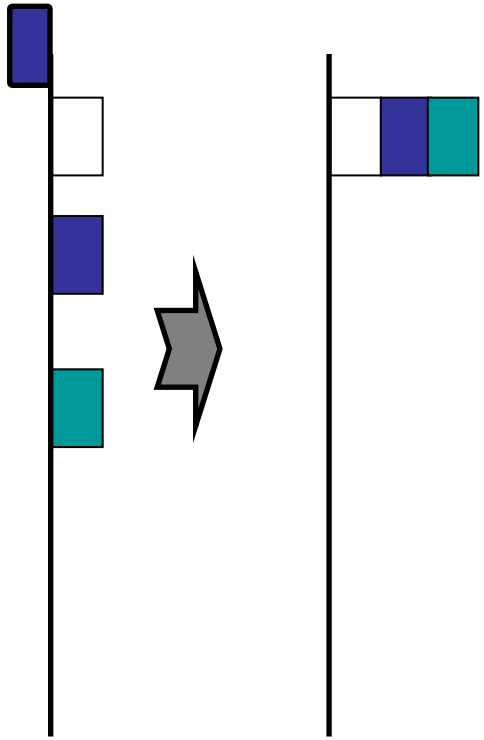
- parallel sections
- The *single* construct
- The *master* construct
- The *nowait* clause
- The *barrier* construct
- The *atomic* construct

6.1 parallel sections

Parallel sections 可以并行不同的代码段

语法结构:

```
#pragma omp parallel sections
{
   #pragma omp section
   phase1();
   #pragma omp section
   phase2();
   #pragma omp section
   phase3();
}
```



• Parallel Sections

并行化下面代码

```
#pragma omp parallel
{  #pragma omp  for
        for(k=0;k<m;k++){
            fn1(k);fn2(k);
        }
   #pragma omp  sections private(y,z)
   {
        #pragma omp  section
      {y=sectionA(x);fn7(y);}  //只执行一次
        #pragma omp  section
   }    {z=sectionB(x);fn8(z);} //只执行一次
}
```

6.2 single 结构

- 定义只有一个线程执行代码段
- 结构:

```
#pragma omp single
```

```
#pragma omp parallel
{
        DoManyThings();
        #pragma omp single
        {
                ExchangeBoundaries();
        }       // threads wait here for single
        DoManyMoreThings();
}
```

6.3 master 结构

定义只被主线程执行的代码段:

```
#pragma omp master
```

```
#pragma omp parallel
{
        DoManyThings();
        #pragma omp master
        {       // if not master, skip to next statement
                ExchangeBoundaries();
        }
        DoManyMoreThings();
}
```

6.4 nowait 结构

- 取消隐式栅障
- 结构：

```
#pragma construct nowa
{[...]};
```

```
#pragma omp for nowait
for (...)
        {...};
```

```
#pragma omp for schedule(dynamic,1) nowait
for (int i=0; i<n; i++)
        a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for (int j=0; j<m; j++)
        b[j] = bigFunc2(j);
```

分析

```
#pragma omp parallel
{  #pragma omp  for   nowait
                for(k=0; k<m; k++){
                    fn10(k);fn20(k);
                  }
   #pragma omp  sections private(y,z)
   {
                #pragma omp  section
                  {y=sectionD();fn70(y);}   //只执行一次
                 #pragma omp  section
   }              {z=sectionC();fn80(z);} //只执行一次

}
```

6.5 barrier 结构

- 设置显示栅障
- 结构:

```
#pragma omp barrier
```

```
#pragma omp parallel shared(A,B,C)
{
        DoSomeWork(A,B);
        printf( “Processed A into B\n” );
#pragma omp barrier
        DoSomeWork(B,C);
        printf( “Processed B into C\n” );
}
```

分析

```
#pragma omp parallel shared(x,y,z) num_threads(2)
{     int tid = omp_get_thread_num();
      if(tid==0){
            y = fn70(tid);
          }
       else{
            z = fn80(tid);
          }
    #pragma omp  barrier
    #pragma omp  for
                    for(k=0; k<m; k++){
                        x[k]=y+z+fn10(k)+fn20(k);
                    }
}
```

综合练习


```
#pragma omp parallel shared(x,y,z) num_threads(2)
{    int tid = omp_get_thread_num();//每个线程都调用
          #pragma omp  for  nowait
           for(k=0; k<100; k++)  x[k]=fn1(tid); //不等待
          #pragma  omp  master
          y = fn_input_only();  //只有主线程调用
          #pragma omp  barrier
          #pragma omp  for  nowait
    for(k=0; k<100; k++)  x[k]=y+fn2(x[k]);  //不等待
      #pragma omp  single
     fn_single_print(y);  //只需一个线程执行
      #pragma omp  master
     fn_print_array(x);  //主线程执行
    }
```

6.6 atomic 结构

- 定义一个微型临界区，速度比临界区快
- 结构：

```
#pragma omp atomic
```

```
#pragma omp parallel for shared(x,y,index,n)
for (i=0; i<n; i++)
{
        #pragma omp atomic
          x[index[i]] += work1(i);
        y[i] += work2(i);
}
```

7.调度优化： 分配任务的原则优化

原因:

- 任务共享
- 描述如何划分循环给组内的线程

- 目的:
  - 保证执行核尽可能地在大部分时间内保持忙碌状态
  - 将调度开销、上下文切换开销和同步开销降到最低。

7.调度优化

几种调度策略：

- *static 静态*
- *dynamic 动态*
- *guided 指导*

7.1静态平均调度策略（默认不指定块大小

语法格式:

```
schedule(static [,chunk])
```

描述:

- 将迭代分割为 *chunk*大小
- 以轮转的方式由系统分配

例

```
#pragma omp parallel for schedule(static)
    for(k=0;k<1000;k++)
        do_work(k);
```

7.2 动态调度策略

语法格式:

```
schedule(dynamic [,chunk])
```

描述:

- 将迭代分割为 *chunk*大小
- 线程完成一个任务数据块后再申请下一个数据块
- 默认情况下*chunk=1*

7.3 指导调度策略

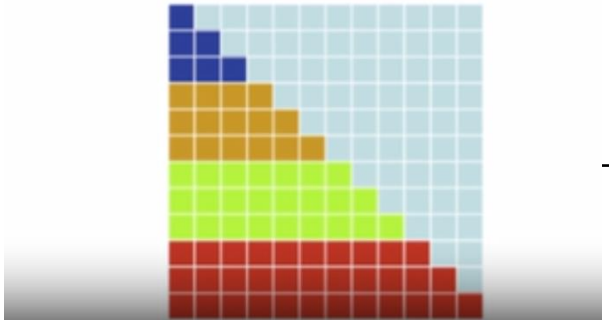
格式:

schedule(guided [,chunk])

描述:

- 开始分配的迭代块的迭代次数比*chunk*的值大
- 开始调度，逐渐减小块的量
- *chunk=1*

```
#pragma omp parallel for
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



7.3 指导调度策略

- 计算公式：$\Gamma_k=[\beta_k/(2N)]$
- 其中N是线程个数，Γ_k代表第K块的大小，从第0块开始，在计算第k块的时，β_k代表剩下的未调度的循环迭代次数。
- 如果Γ_k值太小，那么该值由指定的块大小S取代，如果没有指定块大小，则取默认值1。
- 循环分块方法取决于线程个数（N），迭代次数（β_0）和块大小（S）。

例：

- 给定一个循环， $ß_0$=800，N=2，S=80，则循环划分为：

{200，150，113，85，80，80，12}

调度策略的应用

调度策略	应用
Static	可以预料的等量的划分
Dynamic	不可预知的，易变的任务划分
Guided	减少调度开销

例子

静态调度:

```
#pragma omp parallel for schedule(static,8)
for (int i=start; i<=end; i+=2)
{
        if (TestForPrime(i))
                gPrimesFound++;
}
```

当 *chunk* = 8， *start* i= 3时, 第一个迭代块的i值是:

i = {3,5,7,9,11,13,15,17}

性能分析

```
#pragma omp parallel for
        for(k=0;k<m;k++){
            fn1(k);fn2(k);
        }

#pragma omp parallel for
        for(k=0;k<m;k++){
            fn3(k);fn4(k);
        }
```

性能分析

```
#pragma omp parallel
 {
  #pragma omp  for
      for(k=0;k<m;k++){
          fn1(k);fn2(k);
        }
     #pragma omp  for
      for(k=0;k<m;k++){
          fn3(k);fn4(k);
        }
}
```

8.OpenMP库函数

- omp_get_num_procs, 返回当前可用处理器个数。
- omp_get_num_threads, 返回当前并行区域中的活动线程个数。
- omp_get_thread_num, 返回当前线程号，值在0（主线程）到线程总数-1之间
- omp_set_num_threads, 进入并行区前，将要使用的线程个数

MPI—消息传递编程标准

- 库调用并行编程：一组用标准串行语言编写的进程程序，在并行执行时可以调用消息（函数）库以发送/接收消息。所以在使用MPI标准时，计算由一组重量级进程组成，他们可以采用调用库例程的方法进行通信
- 通信包括基本通信（点到点）和整体通信（播送，归约）
- 最基本的MPI程序可以只使用6个函数；虽然MPI库可包含多达200多个函数，但最经常使用的也就是20多条
- MPI可以绑定C语言或Fortran语言

MPI编程基础

- 简单的MPI程序示例

```
#include <stdio.h>
#include "mpi.h"

int main（ int argc, char *argv[] ） {
    int rank;
    int size;

    MPI_Init（ argc, argv ）；
    MPI_Comm_rank（MPI_COMM_WORLD, &rank）；
    MPI_Comm_size（MPI_COMM_WORLD, &size）；
    printf（ "Hello world from process %d of %d\n", rank, size ）；
    MPI_Finalize（）；
    return 0;
}
```

```
Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
```

MPI程序的四个基本函数

- MPI_Init和MPI_Finalize
  - MPI_Init初始化MPI执行环境，建立多个MPI进程之间的联系，为后续通信做准备。而MPI_Finalize则是结束MPI执行环境。
- MPI_Comm_rank
  - 来标识各个MPI进程
- MPI_Comm_size
  - 用来标识相应进程组中有多少个进程

MPI的点对点通信

- 两个最重要的MPI函数MPI_Send和MPI_Recv。
- int MPI_SEND(buf, count, datatype, dest, tag, comm)
  - 这个函数的含义是向通信域comm中的dest进程发送数据。消息数据存放在buf中，类型是datatype，个数是count个。这个消息的标志是tag，用以和本进程向同一目的进程发送的其他消息区别开来。
- int MPI_RECV(buf,count,datatype,source,tag,comm,status)
  - MPI_Recv绝大多数的参数和MPI_Send相对应，有相同的意义，很好理解。唯一的区别就是MPI_Recv里面多了一个参数status。status主要显示接收函数的各种错误状态。

消息管理7要素

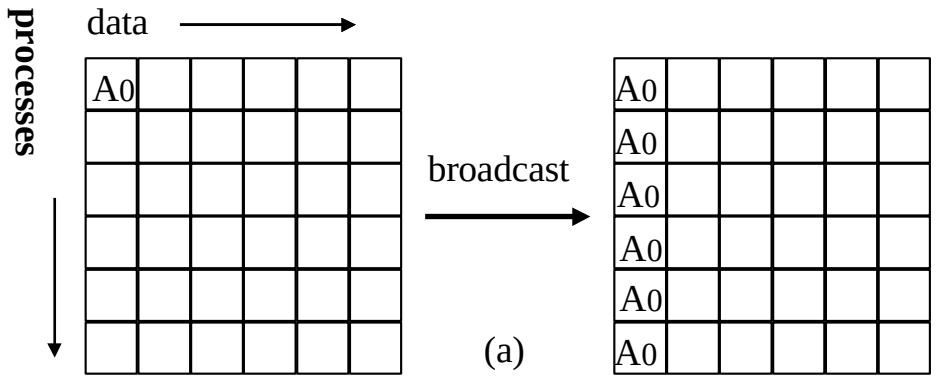
- MPI最重要的功能莫过于消息传递。正如我们先前看到一样，MPI_Send和MPI_Recv负责在两个进程间发送和接收消息。总结起来，点对点消息通信的参数主要是由以下7个参数组成：
  - 发送或者接收缓冲区buf；
  - 数据数量count；
  - 数据类型datatype；
  - 目标进程或者源进程destination/source；
  - 消息标签tag；
  - 通信域comm；.
  - 消息状态status，只在接收的函数中出现。

错误管理

- MPI在错误管理方面提供了丰富的接口函数，这里我们介绍其中最简单的部分接口。
  - 用status.MPI_ERROR来获取错误码。
  - MPI终止MPI程序执行的函数MPI_Abort。
  - int MPI_Abort（MPI_Comm comm, int errorcode）
    - 它使comm通信域的所有进程退出，返回errorcode给调用的环境。通信域comm中的任一进程调用此函数都能够使该通信域内所有的进程结束运行。

MPI群集通信

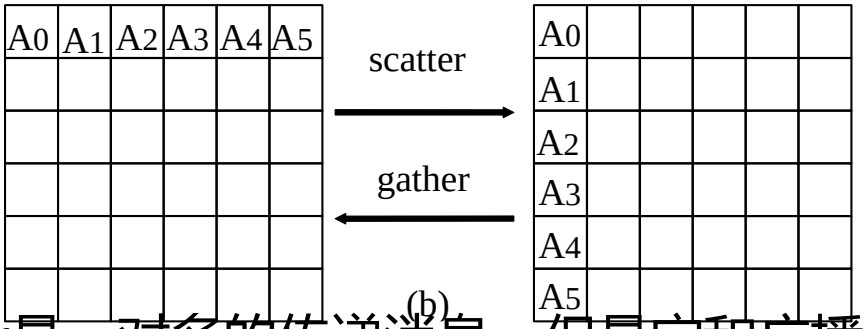
- 同步
  - 本函数接口是：int MPI_Barrier（MPI_Comm comm）
  - 在操作中，通信子comm中的所有进程相互同步，即它们相互等待，直到所有进程都执行了他们各自的MPI_Barrier函数，然后再各自接着开始执行后续的代码。
- 广播
  - 从一个root进程向组内所有其他的进程发送一条消息。接口是：int MPI_Bcast（ void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm ）



MPI群集通信(续)

- 聚集
  - 聚集函数MPI_Gather是一个多对一的通信函数。其接口为：

int MPI_Gather（void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm）

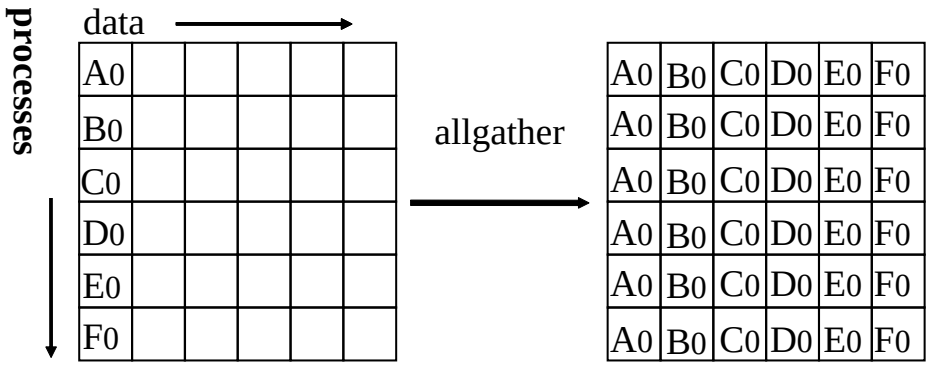


- 播撒
  - MPI_Scatter是一对多的传递消息。但是它和广播不同，root进程向各个进程传递的消息是可以不同的。Scatter实际上执行的是与Gather相反的操作。

MPI群集通信(续2)

- 扩展的聚集和播撒操作
  - MPI_Allgather的作用是每一个进程都收集到其他所有进程的消息，它相当于每一个进程都执行了MPI_Gather执行完了MPI_Gather之后，所有的进程的接收缓冲区的内容都是相同的，也就是说每个进程给所有进程都发送了一个相同的消息。

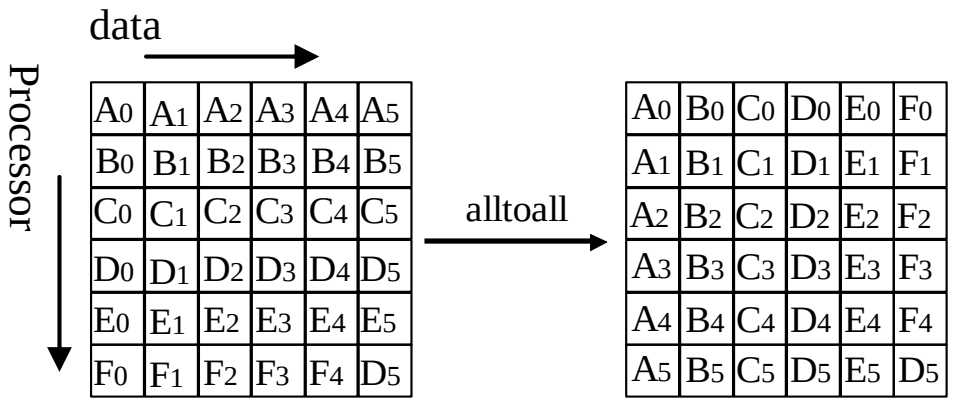
int **MPI_Allgather**（void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm）



MPI群集通信(续3)

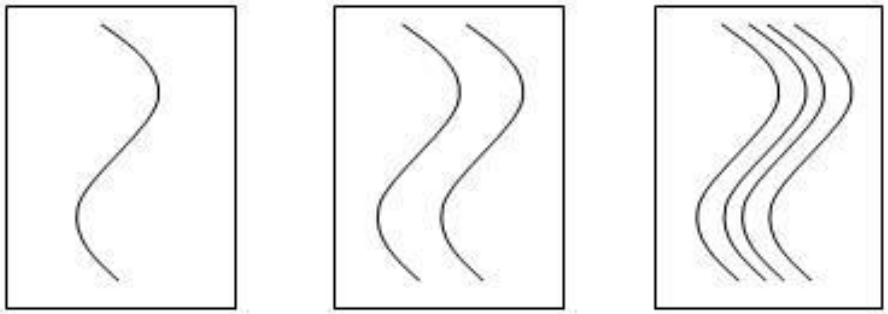
- 全局交换
  - MPI_Alltoall的每个进程可以向每个接收者发送数目不同的数据，第i个进程发送的第j块数据将被第j个进程接收并存放在其接收消息缓冲区recvbuf的第i块，每个进程的sendcount和sendtype的类型必须和所有其他进程的recvcount和recvtype相同，这也意谓着在每个进程和根进程之间发送的数据量必须和接收的数据量相等。

int MPI_Alltoall（void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm）



Thread/pThread

- 进程不适合细粒度的共享存储并行程序设计。
- 线程 （threads）又被称作轻量级进程。
- 进程可由单个线程来执行，即通常所说的串行执行；或者，进程也可由多个线程来并行执行，此时，多个线程将共享该进程的所有资源特征，并可以使用不同的CPU，对不同的数据进行处理，从而达到提高进程执行速度的目的。



(a)单进程单线程　(b)单进程双线程　(c)单进程多线程

并行程序性能优化

- 减少通信量、提高通信粒度
- 全局通信尽量利用高效集合通信算法
- 挖掘算法的并行度，减少CPU空闲等待
- 负载平衡
- 通信、计算的重叠
- 通过引入重复计算来减少通信，即以计算换通信

- 详见
- 实验四~计算机系统结构实验上机指导20210526.pdf

实验四
用 MPI 完成 Cannon 算法的并行计算

東北林業大學
NORTHEAST FORESTRY UNIVERSITY

THANK YOU!