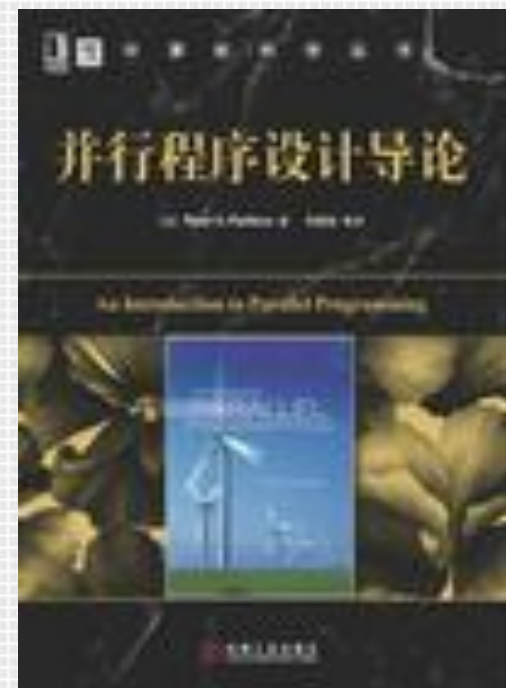


An Introduction to Parallel Programming

Peter Pacheco

Chapter 2

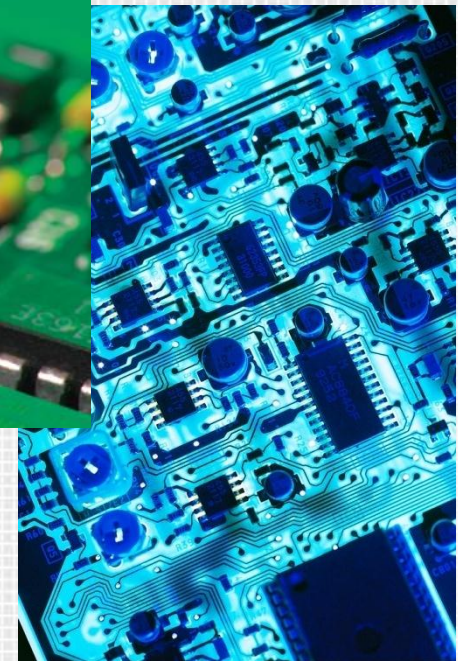
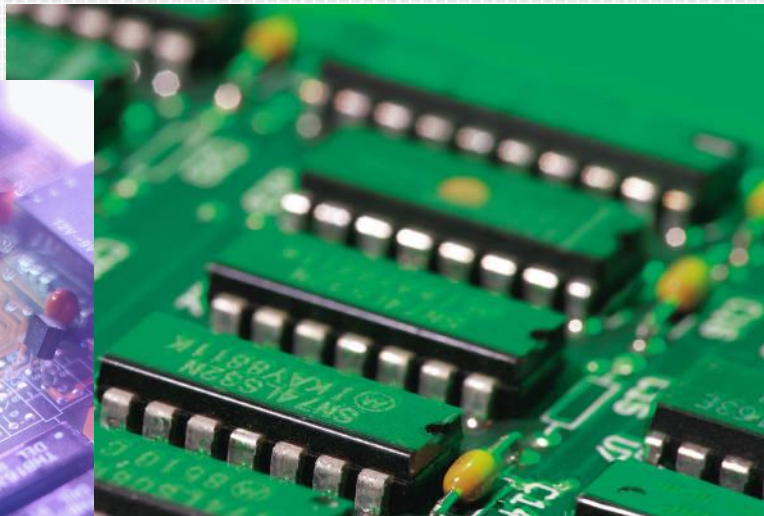
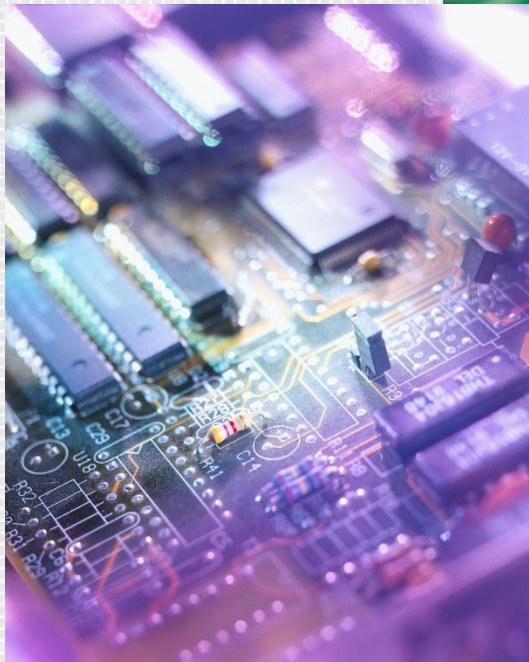
并行硬件和并行软件





Roadmap

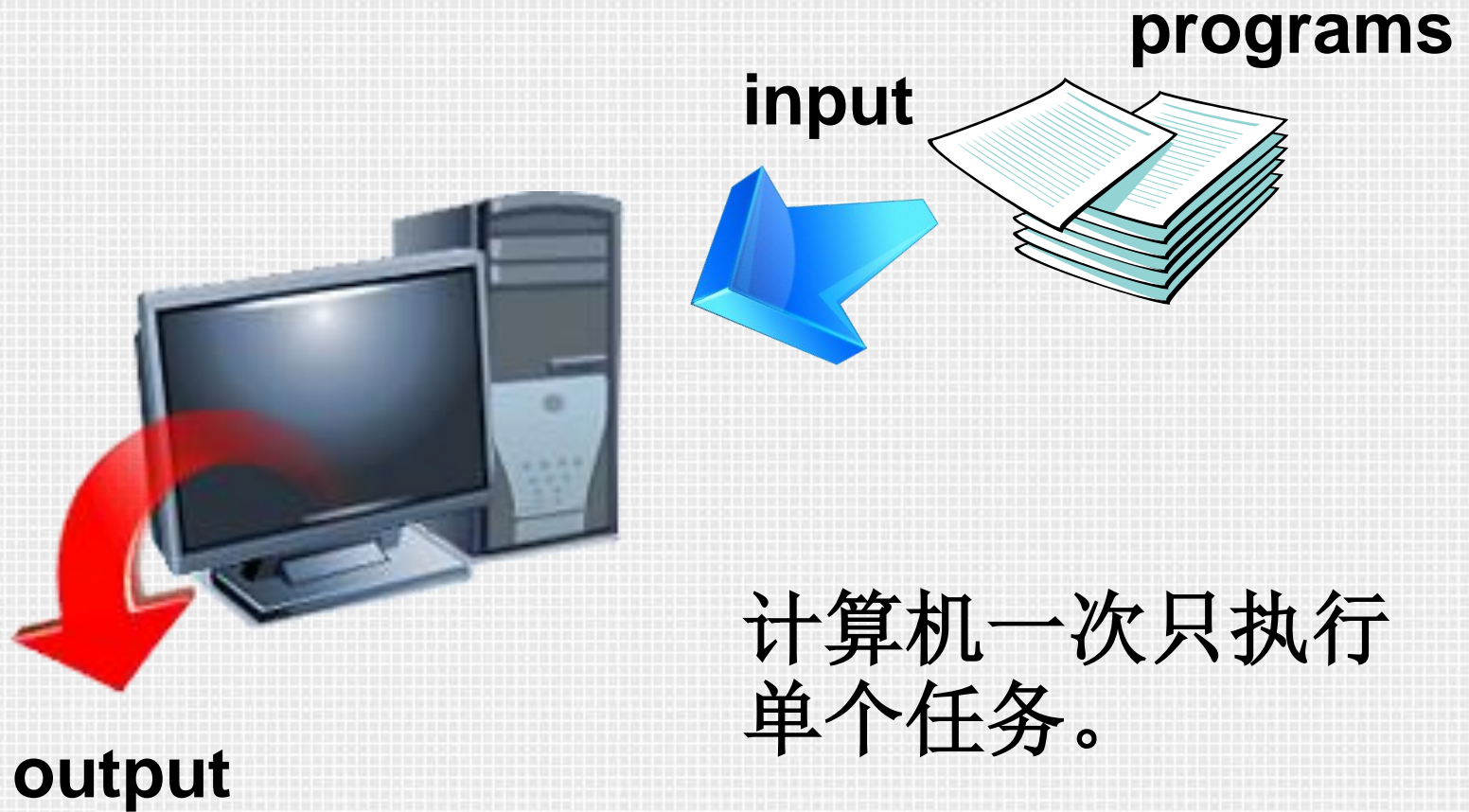
- 背景知识
- 对冯.诺依曼模型的改进
- 并行硬件
- 并行软件
- 输入和输出
- 性能的评价
- 并行程序设计
- 编写和运行并行程序
- 假设和小结



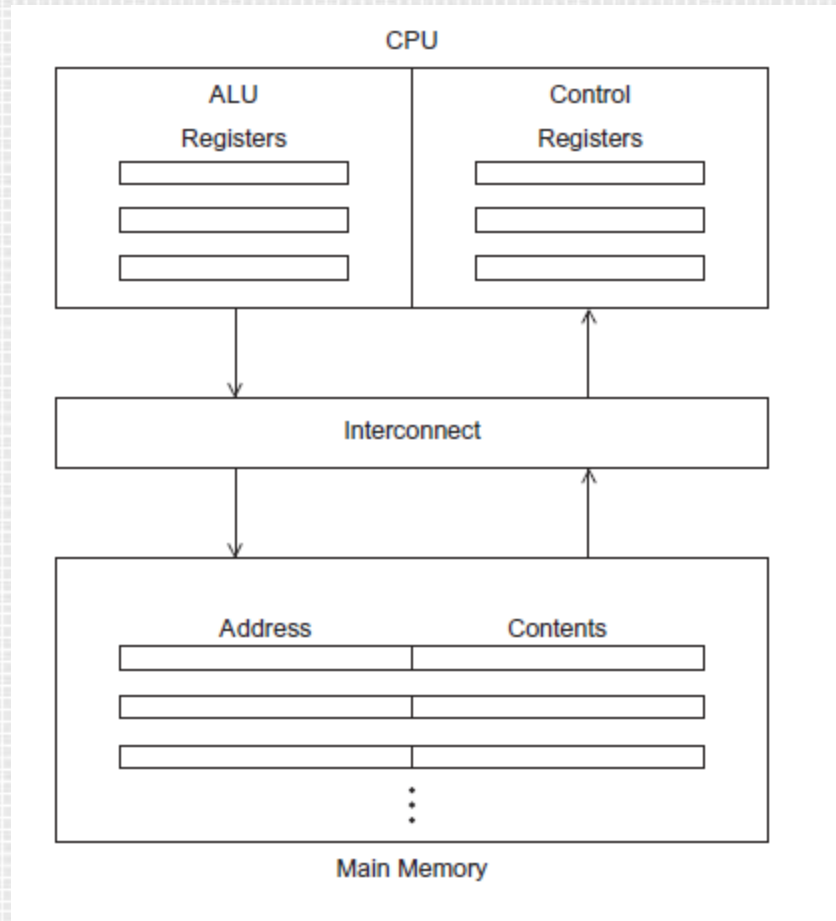
2.1 背景知识



2.1.1 冯.诺依曼体系结构



计算机一次只执行
单个任务。

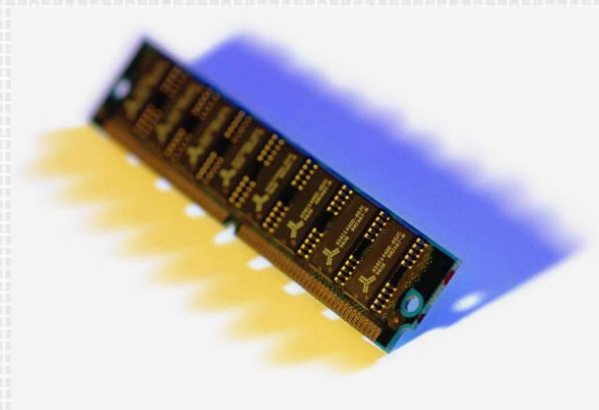


冯.诺依曼体系结构



主存（Main memory）

- 主存中有许多区域，每个区域都可以存储指令和数据。
- 每个区域都有一个地址，可以通过这个地址来访问相应的区域以及区域中存储的数据和指令。





中央处理单元（CPU）

- 分为两个部分：
- 控制单元（**Control unit**）
 - 负责决定应该执行程序中的哪些指令。（*the boss*）
- 算数逻辑单元
（**Arithmetic and logic unit , ALU**）
 - 负责执行命令（*the worker*）



关键术语

- 寄存器

- 快速存储介质，存储**CPU**中的数据和程序执行的状态信息。

- 程序计数器

- 控制单元中一个特殊的寄存器，用来存放下一条指令的地址。

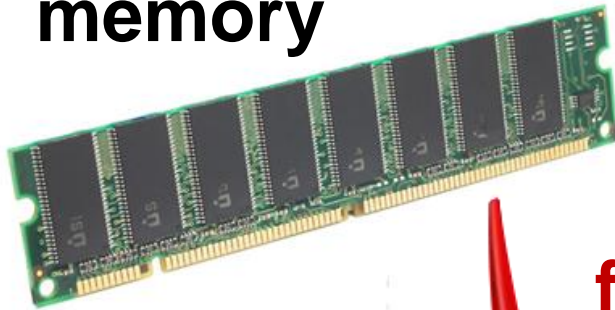
- 总线

- 在**CPU**和主存之间进行数据和指令传输的互连结构。





memory

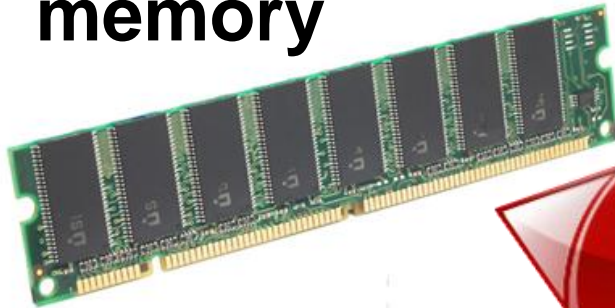


fetch/read



CPU

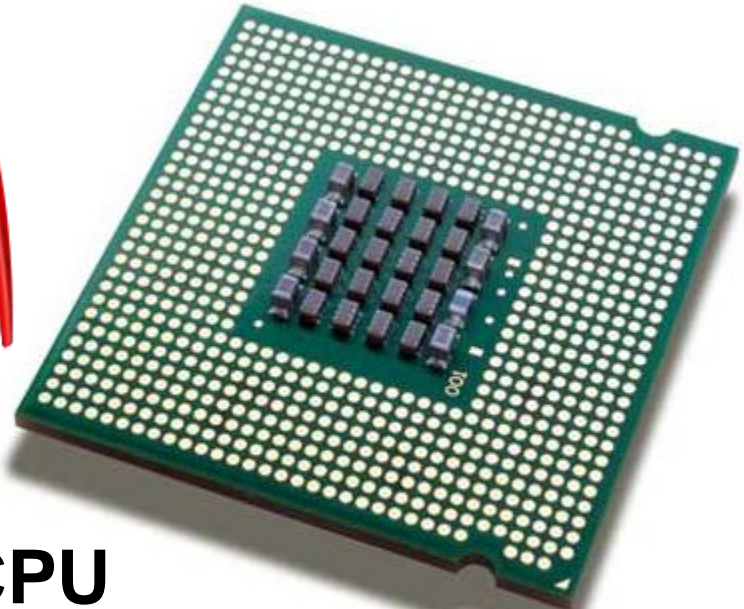
memory

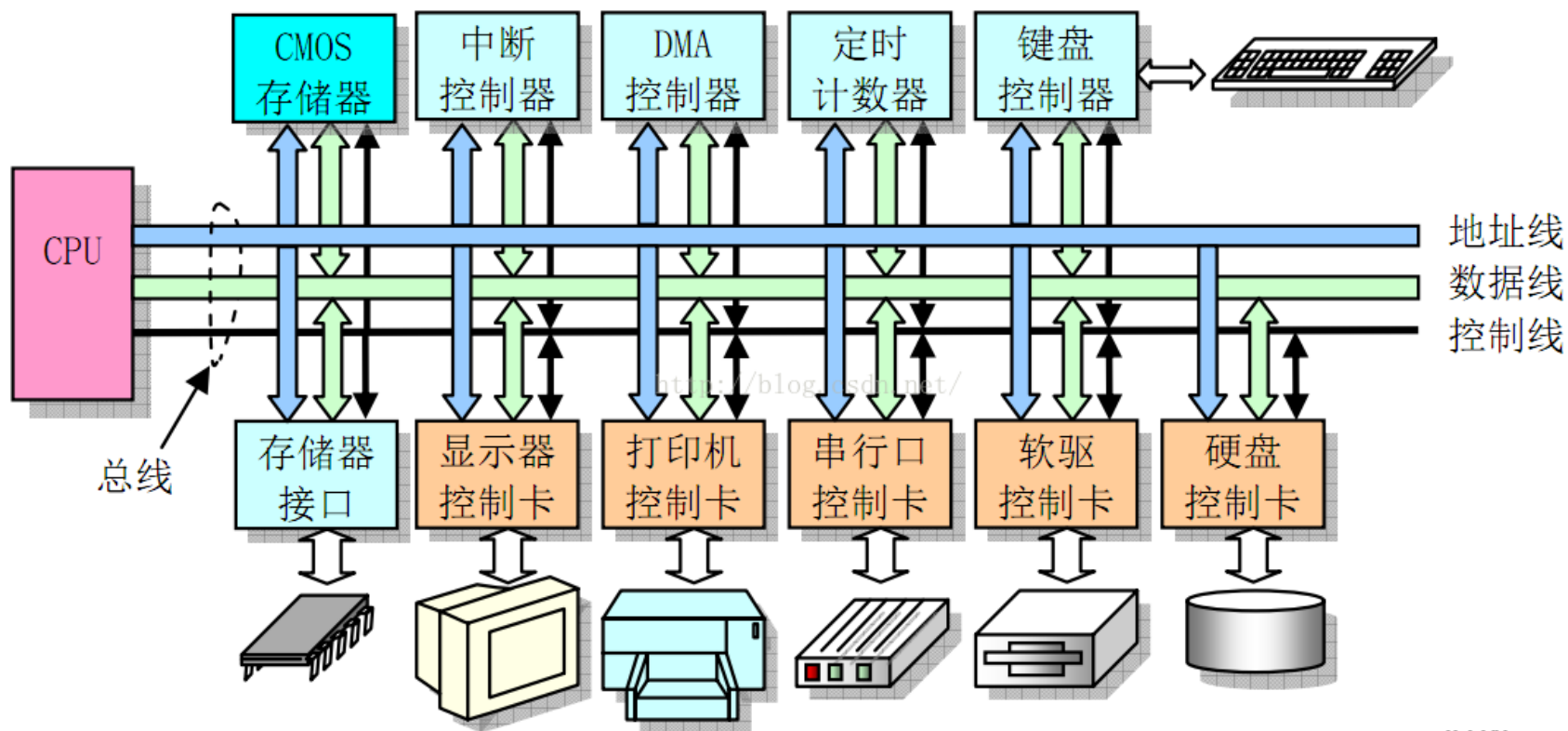


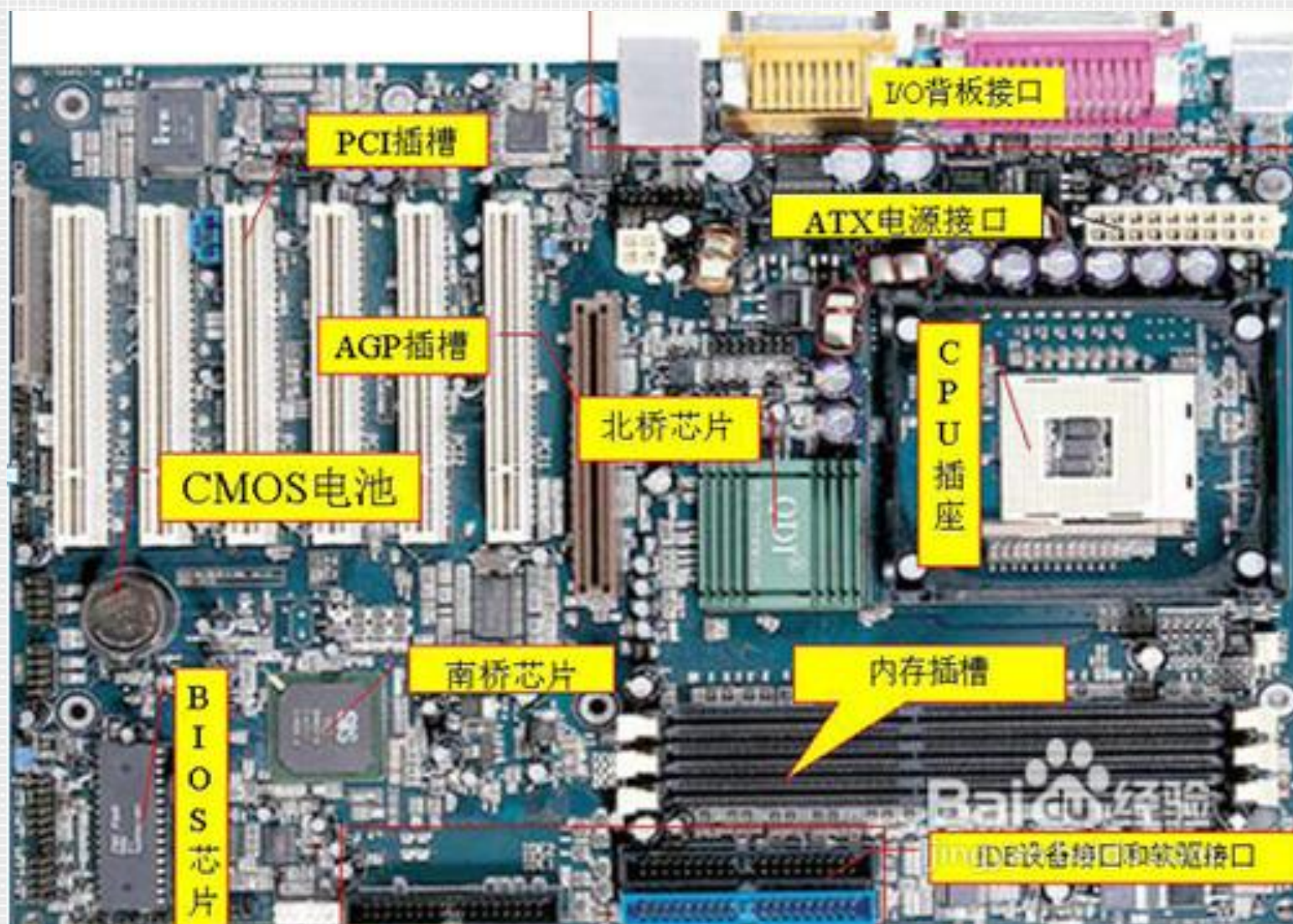
write/store

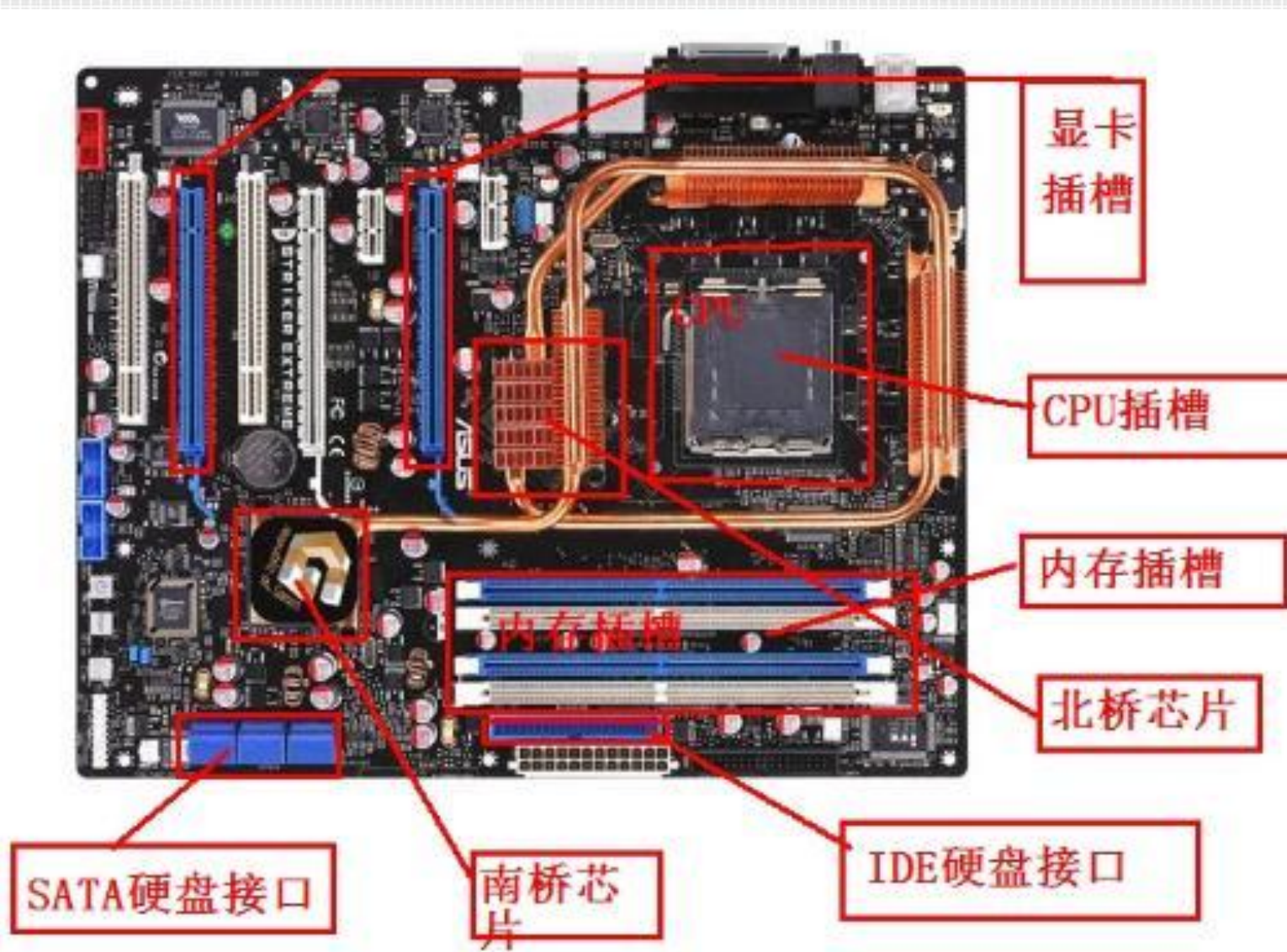


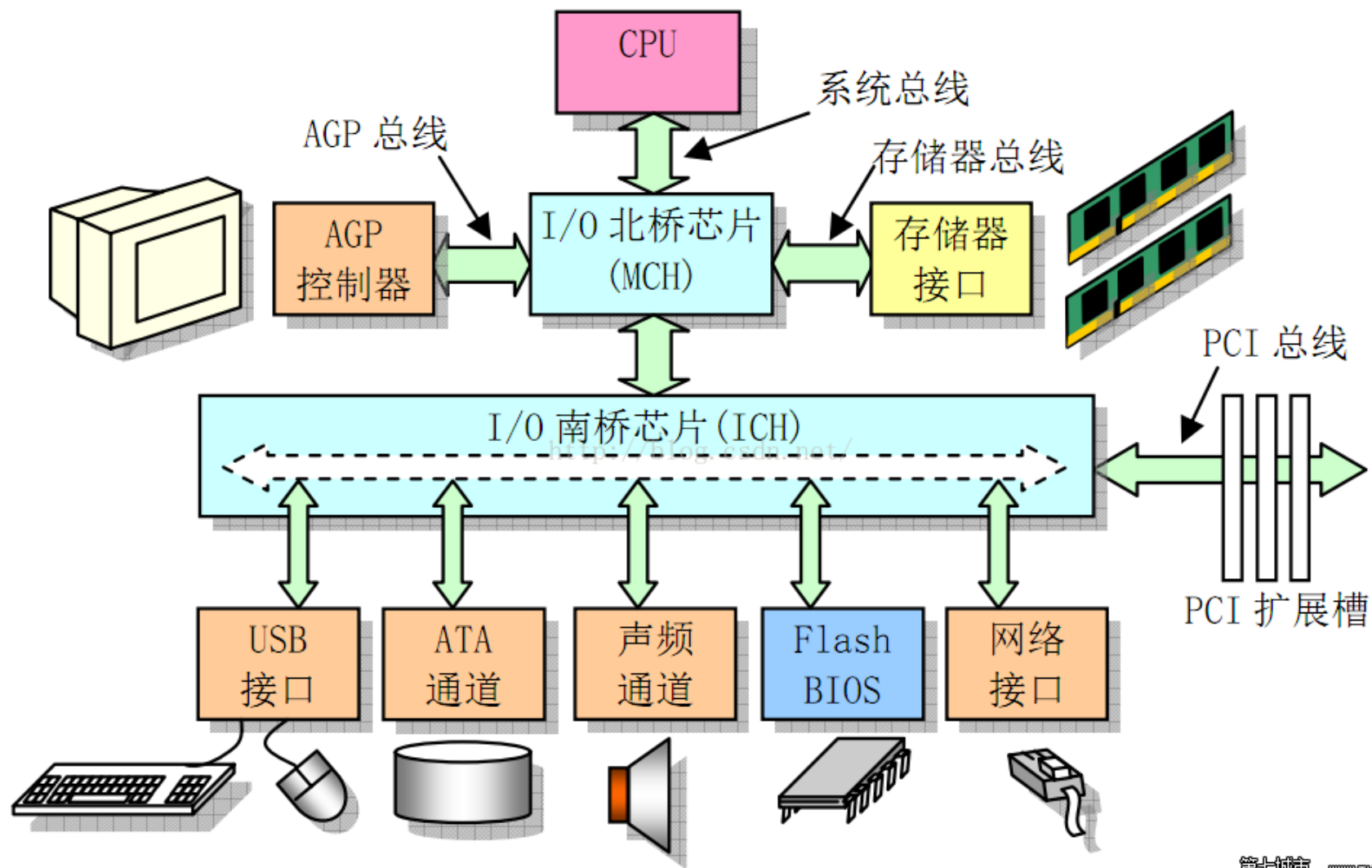
CPU





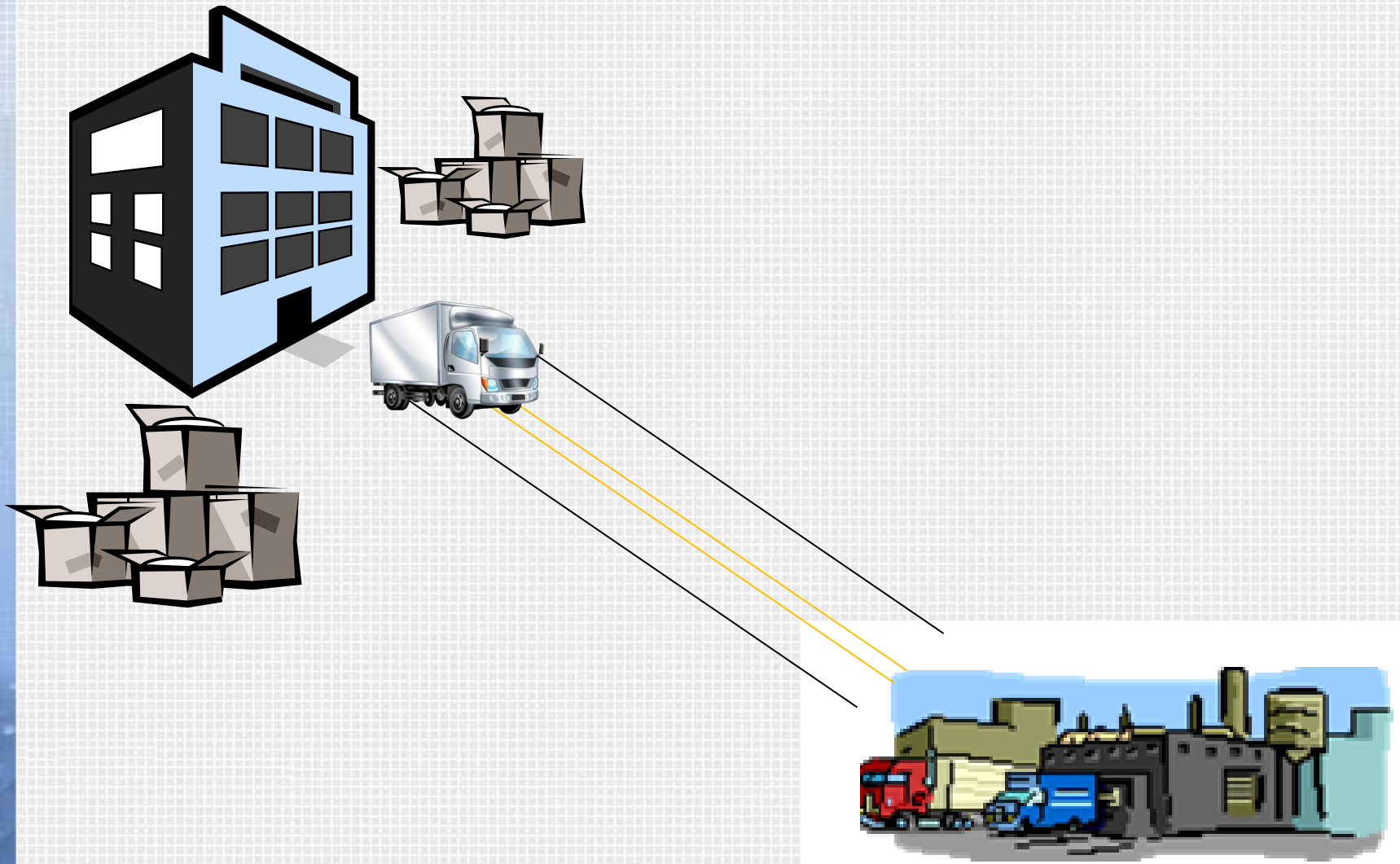








冯.诺依曼瓶颈





2.1.2 进程、多任务及线程

操作系统

- 一种用来管理计算机的软件和硬件资源的主要软件。
- 一个进程包括：
 - 可执行的机器语言程序
 - 一个内存空间
 - 资源描述符
 - 安全信息
 - 进程状态信息



多任务

- 操作系统提供对同时运行多个程序的支持。
 -
- 即使在单核系统中，通过**时间片机制**来实现多个进程的运行。
- 在一个程序执行了一个时间片的时间后，操作系统就阻塞当前程序，切换执行其他程序。

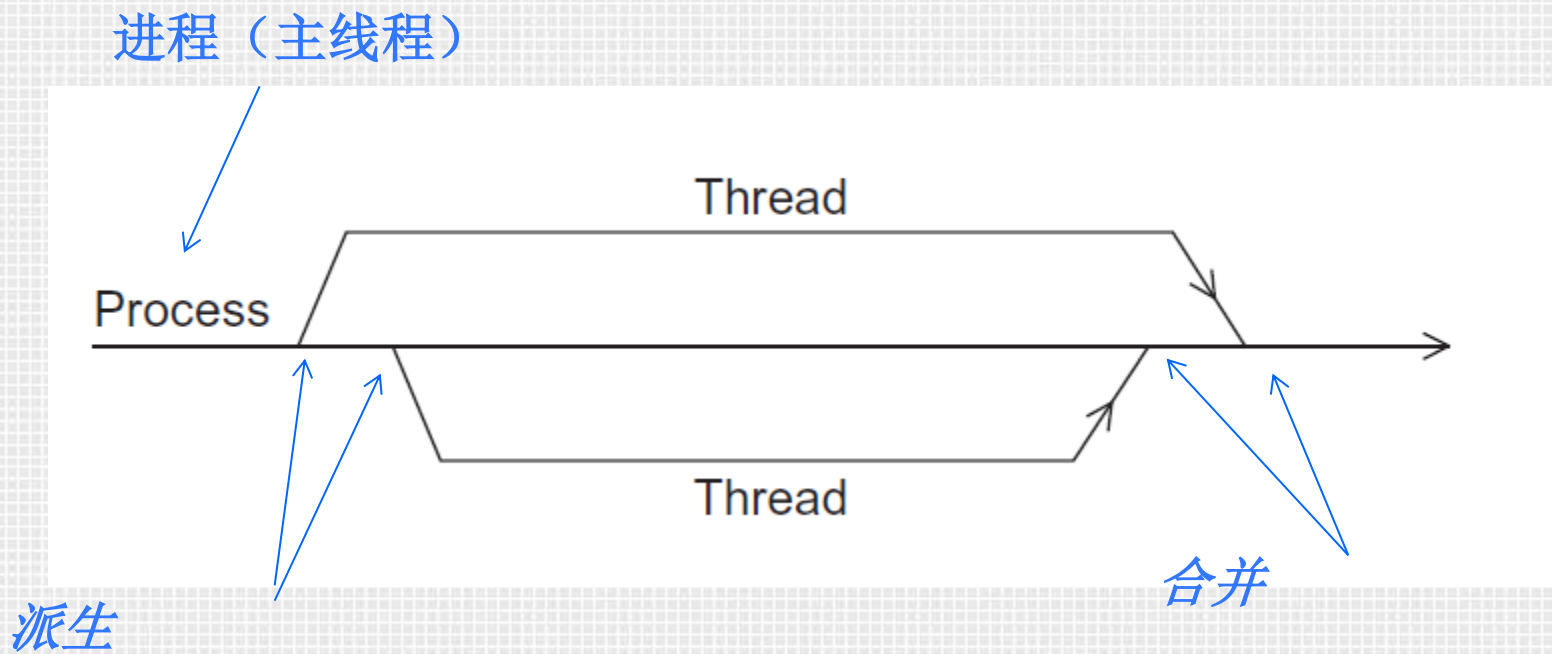


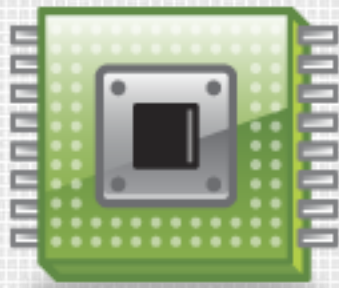
线程

- 一个进程包含一个或者多个线程。
- 将程序划分为多个大致独立的任务，当某个任务阻塞时能执行其他任务。



一个进程与两个线程



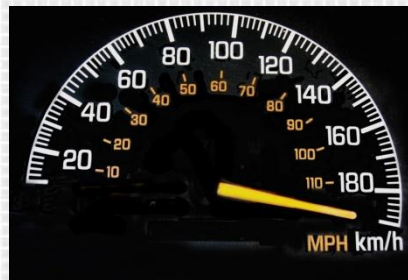


2.2 对冯.诺依曼模型的改进



2.2.1 Cache基础知识

- 高速缓冲寄存器(**CPU Cache**), 是一组相比于主存, **CPU**能更快速地访问的内存区域。
- **CPU cache** 位于与**CPU**同一块的芯片或者位于其他比普通的内存芯片更快的访问的芯片上。





局部性原理

- 在一段时间内，**CPU**总是几种地访问程序中的某一个部分而不是随机地对程序所有部分具有平均访问概率。
 - 空间局部性 – 访问邻近的区域
 - 时间局部性 – 在一段时间内



```
float z[1000];
```

```
...
```

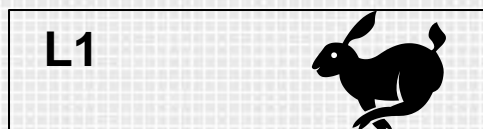
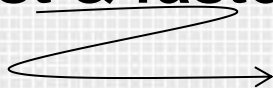
```
sum = 0.0;
```

```
for (i = 0; i < 1000; i++)
```

```
    sum += z[i];
```

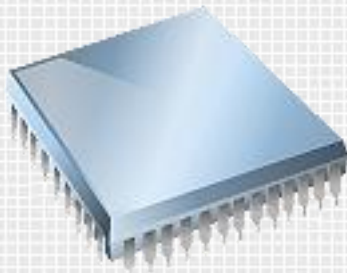
Cache 层

smallest & fastest



largest & slowest

Cache 命中



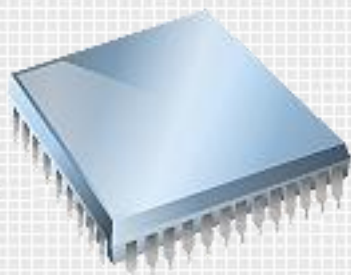
fetch x

| | | |
|----|---|-----|
| L1 | x | sum |
|----|---|-----|

| | | | |
|----|---|---|-------|
| L2 | y | z | total |
|----|---|---|-------|

| | | | | |
|----|------|--------|----|--------|
| L3 | A[] | radius | r1 | center |
|----|------|--------|----|--------|

Cache 缺失



fetch x

L1

y

sum

L2

r1

z

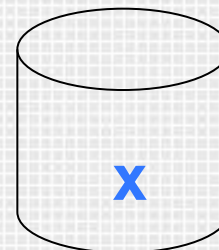
total

L3

A[]

radius

center



x

main
memor
y



Issues?

- 当**CPU**向**Cache**中写数据时，**Cache**中的值与主存中的值就会不同或者不一致。
- 写直达
 - 当**CPU**向**Cache**写数据时，高速缓存行会立即写入主存。
- 写回
 - 数据不是立即更新到主存，而是将发生数据更新的高速缓存行标记成**脏**（**dirty**）。当发生高速缓存行替换时，把标记为**脏**的高速缓存行写入内存中。



2.2.2 Cache 映射

- 高速缓存行应该存储在什么位置？
- 全相联（**Full associative**）
 - 每个高速缓存行能够放置在**Cache**中的任意位置。
- 直接映射（**Direct mapped**）
 - 放置在**Cache**中指定的唯一位置。
- **n**路组相联（***n*-way set associative**）
 - 放置在**Cache**中**n**个不同区域位置中的一个。



Example

| Memory Index | Cache Location | | |
|--------------|----------------|---------------|--------|
| | Fully Assoc | Direct Mapped | 2-way |
| 0 | 0, 1, 2, or 3 | 0 | 0 or 1 |
| 1 | 0, 1, 2, or 3 | 1 | 2 or 3 |
| 2 | 0, 1, 2, or 3 | 2 | 0 or 1 |
| 3 | 0, 1, 2, or 3 | 3 | 2 or 3 |
| 4 | 0, 1, 2, or 3 | 0 | 0 or 1 |
| 5 | 0, 1, 2, or 3 | 1 | 2 or 3 |
| 6 | 0, 1, 2, or 3 | 2 | 0 or 1 |
| 7 | 0, 1, 2, or 3 | 3 | 2 or 3 |
| 8 | 0, 1, 2, or 3 | 0 | 0 or 1 |
| 9 | 0, 1, 2, or 3 | 1 | 2 or 3 |
| 10 | 0, 1, 2, or 3 | 2 | 0 or 1 |
| 11 | 0, 1, 2, or 3 | 3 | 2 or 3 |
| 12 | 0, 1, 2, or 3 | 0 | 0 or 1 |
| 13 | 0, 1, 2, or 3 | 1 | 2 or 3 |
| 14 | 0, 1, 2, or 3 | 2 | 0 or 1 |
| 15 | 0, 1, 2, or 3 | 3 | 2 or 3 |

将**16**行主存映射到**4**行当**Cache**上

n路组相联

- 当内存中的行（多于一行）能被映射到 **Cache** 中的多个不同位置时，替换 **Cache** 中的哪一行？
- 最近最少使用
(least recently used)





2.2.3 Caches和程序

```
double A[MAX][MAX], x[MAX], y[MAX];  
.  
.  
.  
/* Initialize A and x, assign y = 0 */  
.  
.  
.  
/* First pair of loops */  
for (i = 0; i < MAX; i++)  
    for (j = 0; j < MAX; j++)  
        y[i] += A[i][j]*x[j];  
.  
.  
.  
/* Assign y = 0 */  
.  
.  
.  
/* Second pair of loops */  
for (j = 0; j < MAX; j++)  
    for (i = 0; i < MAX; i++)  
        y[i] += A[i][j]*x[j];
```

| Cache Line | Elements of A | | | |
|------------|---------------|---------|---------|---------|
| 0 | A[0][0] | A[0][1] | A[0][2] | A[0][3] |
| 1 | A[1][0] | A[1][1] | A[1][2] | A[1][3] |
| 2 | A[2][0] | A[2][1] | A[2][2] | A[2][3] |
| 3 | A[3][0] | A[3][1] | A[3][2] | A[3][3] |



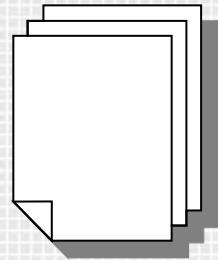
2.2.4 虚拟存储器

- 如果运行一个大型的程序，或者程序需要访问大型数据集，那么所有的指令或者数据可能在主存中放不下。
- 利用虚拟存储器，使得内存作为辅存的缓存。
 - 它利用时间和空间局部性，通过在主存中存放当前执行程序所需要用到的部分。



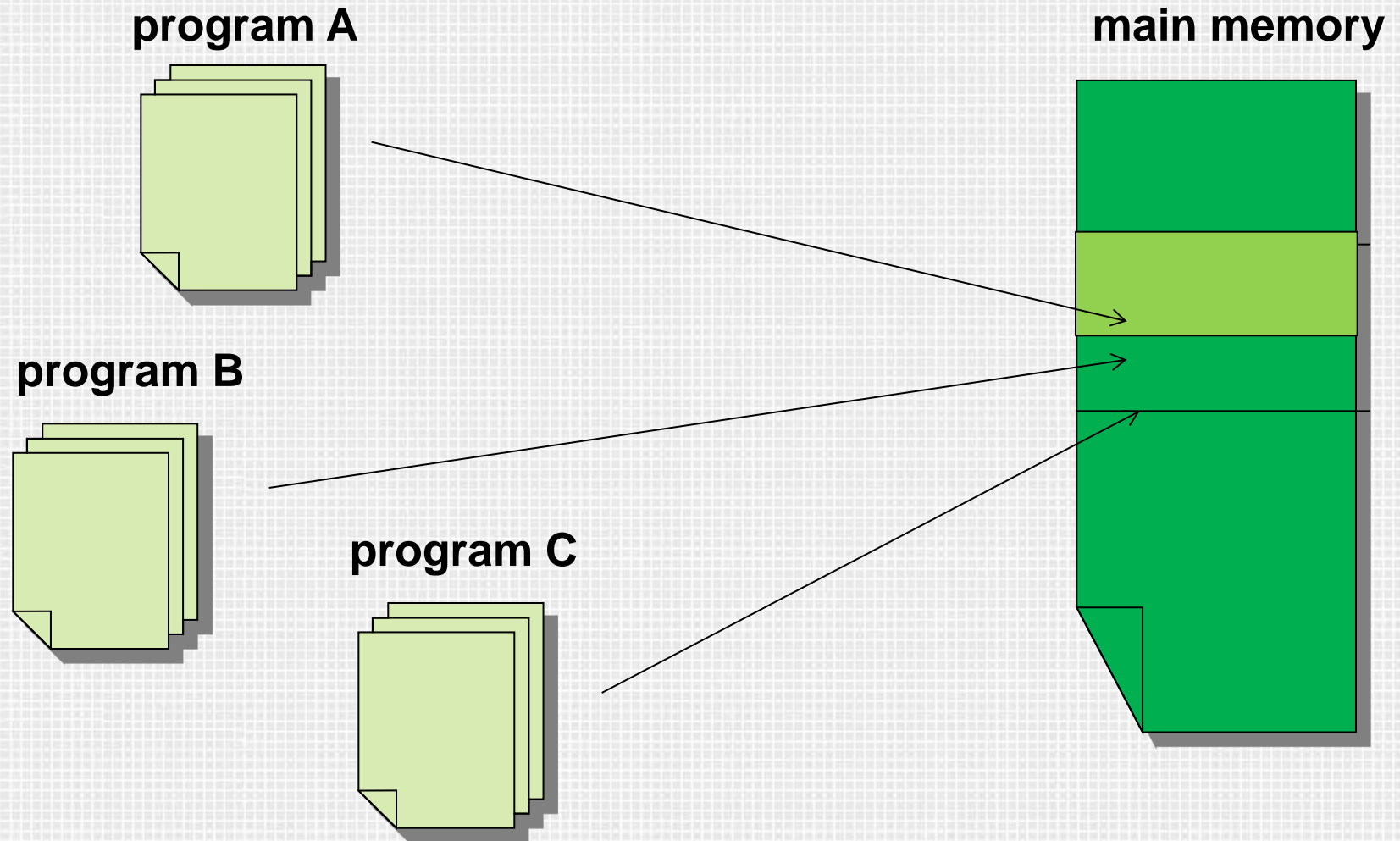
虚拟存储器

- 交换空间 (**Swap space**)
 - 那些暂时用不到的部分存储在辅存的块中
- 页 (**Pages**)
 - 访问辅存比访问内存要慢几十万倍
 - 页通常比较大
 - 采用固定大小，从**4~16K**。





虚拟存储器





虚拟页号

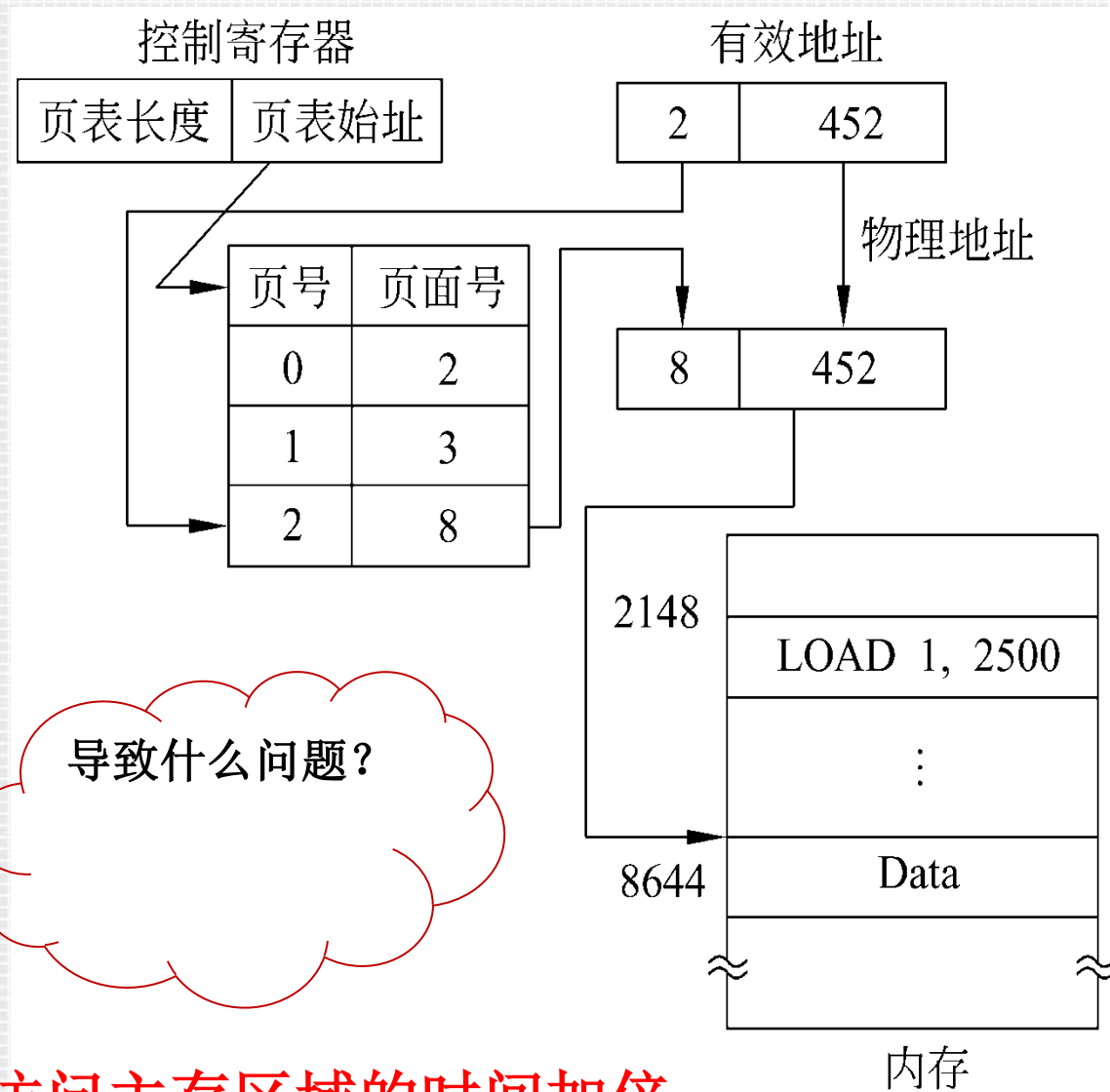
- 在编译程序时，给程序的页赋予虚拟页号。
- 当程序运行时，创建一张将虚拟页号应射程物理地址的表。
- 程序运行时使用到虚拟地址，页表将虚拟地址转换成物理地址。




页表

| Virtual Address | | | | | | | | | |
|---------------------|----|-----|----|----|-------------|----|-----|---|---|
| Virtual Page Number | | | | | Byte Offset | | | | |
| 31 | 30 | ... | 13 | 12 | 11 | 10 | ... | 1 | 0 |
| 1 | 0 | ... | 1 | 1 | 0 | 0 | ... | 1 | 1 |

虚拟地址分为两部分：虚拟页号和页内字节偏移量



使访问主存区域的时间加倍



转译后备缓冲区（Translation-lookaside buffer，TLB）

- 高速联想存储器
- **TLB**在快速存储介质中缓存了一些页表的条目，通常**16-512**条。
- **TLB**命中，**TLB**缺失。
- 页面失效
 - 假设想要访问的页不在内存中，即页表中该页没有合法的物理地址，该页只存储在磁盘上。

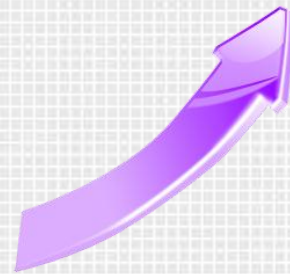
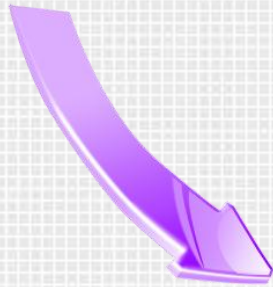
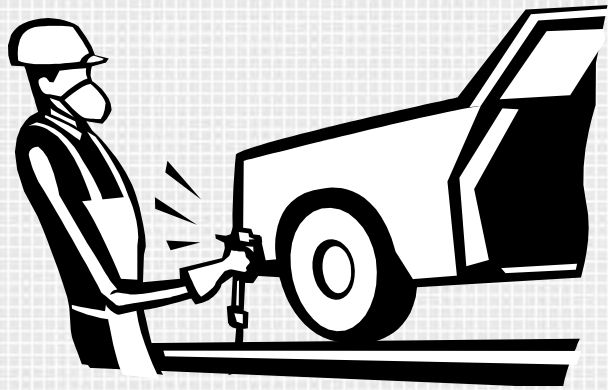


2.2.5 指令级并行Instruction Level Parallelism (ILP)

- 指令级并行，通过让多个**处理器部件**或者**功能单元**同时执行指令来提高处理器的性能。
- 实现方法有两种：
 - 流水线
 - 将功能单元分阶段安排
 - 多发射
 - 让多条指令同时启动



流水线示例1




流水线示例2

将浮点数 9.87×10^4 和 6.54×10^3 相加。

| Time | Operation | Operand 1 | Operand 2 | Result |
|------|-------------------|--------------------|---------------------|----------------------|
| 1 | Fetch operands | 9.87×10^4 | 6.54×10^3 | |
| 2 | Compare exponents | 9.87×10^4 | 6.54×10^3 | |
| 3 | Shift one operand | 9.87×10^4 | 0.654×10^4 | |
| 4 | Add | 9.87×10^4 | 0.654×10^4 | 10.524×10^4 |
| 5 | Normalize result | 9.87×10^4 | 0.654×10^4 | 1.0524×10^5 |
| 6 | Round result | 9.87×10^4 | 0.654×10^4 | 1.05×10^5 |
| 7 | Store result | 9.87×10^4 | 0.654×10^4 | 1.05×10^5 |

- 假设每次操作花费1ns，那么总共花费7ns。



```
float x[1000], y[1000], z[1000];  
.  
.  
for (i = 0; i < 1000; i++)  
    z[i] = x[i] + y[i];
```

- 总共花费**7000ns**



流水线方案

- 将浮点数加法器划分成7个独立的硬件或者功能单元。
- 假设一个功能单元的输出是下面一个功能单元的输入
 - 第一个单元取两个操作数
 - 第二个比较指数
 - 以此类推....



| Time | Fetch | Compare | Shift | Add | Normalize | Round | Store |
|------|-------|---------|-------|-----|-----------|-------|-------|
| 0 | 0 | | | | | | |
| 1 | 1 | 0 | | | | | |
| 2 | 2 | 1 | 0 | | | | |
| 3 | 3 | 2 | 1 | 0 | | | |
| 4 | 4 | 3 | 2 | 1 | 0 | | |
| 5 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 999 | 999 | 998 | 997 | 996 | 995 | 994 | 993 |
| 1000 | | 999 | 998 | 997 | 996 | 995 | 994 |
| 1001 | | | 999 | 998 | 997 | 996 | 995 |
| 1002 | | | | 999 | 998 | 997 | 996 |
| 1003 | | | | | 999 | 998 | 997 |
| 1004 | | | | | | 999 | 998 |
| 1005 | | | | | | | 999 |

流水线加法。表格中的数字表示操作数/结果的下标。



- 一次浮点加法仍然花费7ns。
- 但是1000次浮点加法的时间只花费了1006ns。
- 提高近7倍的速度！！
 - 总的来说k个阶段的流水线不可能达到k倍的性能提高，为什么？

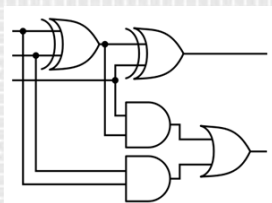


多发射

- 多发射处理器通过复制功能单元来同时执行程序中的不同指令。

for (i = 0; i < 1000; i++)

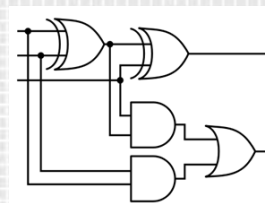
z[i] = x[i] + y[i];



adder #1

← **z[3]**
z[1]

z[4] →
z[2]



adder #2



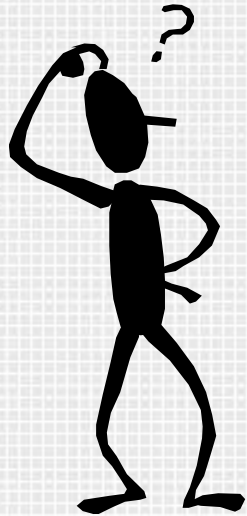
- 如果功能单元是在编译时调度的，则称该多发射系统使用**静态多发射**。
- 如果是在运行时调度的，则称该多发射系统使用**动态多发射**。

超标量**superscalar**



预测（Speculation）

- 为了能够利用多发射，系统必须找出能够同时执行的指令。



- 在预测技术中，编译器或者处理器对一条指令进行猜测，然后再猜测的基础上执行代码。

`z = x + y ;`
`i f (z > 0)`
 `w = x ;`
`e l s e`
 `w = y ;`



如果预测错误，就需要回退机制，执行 $w = y$.



2.2.6 硬件多线程

- 线程级并行（Thread-Level Parallelism, TLP）
 - 尝试通过不同线程来提供并行性。
- 硬件多线程
 - 为系统提供一种机制，使得当前执行的任务被阻塞时，系统能够继续其他有用的工作。



- 细粒度多线程

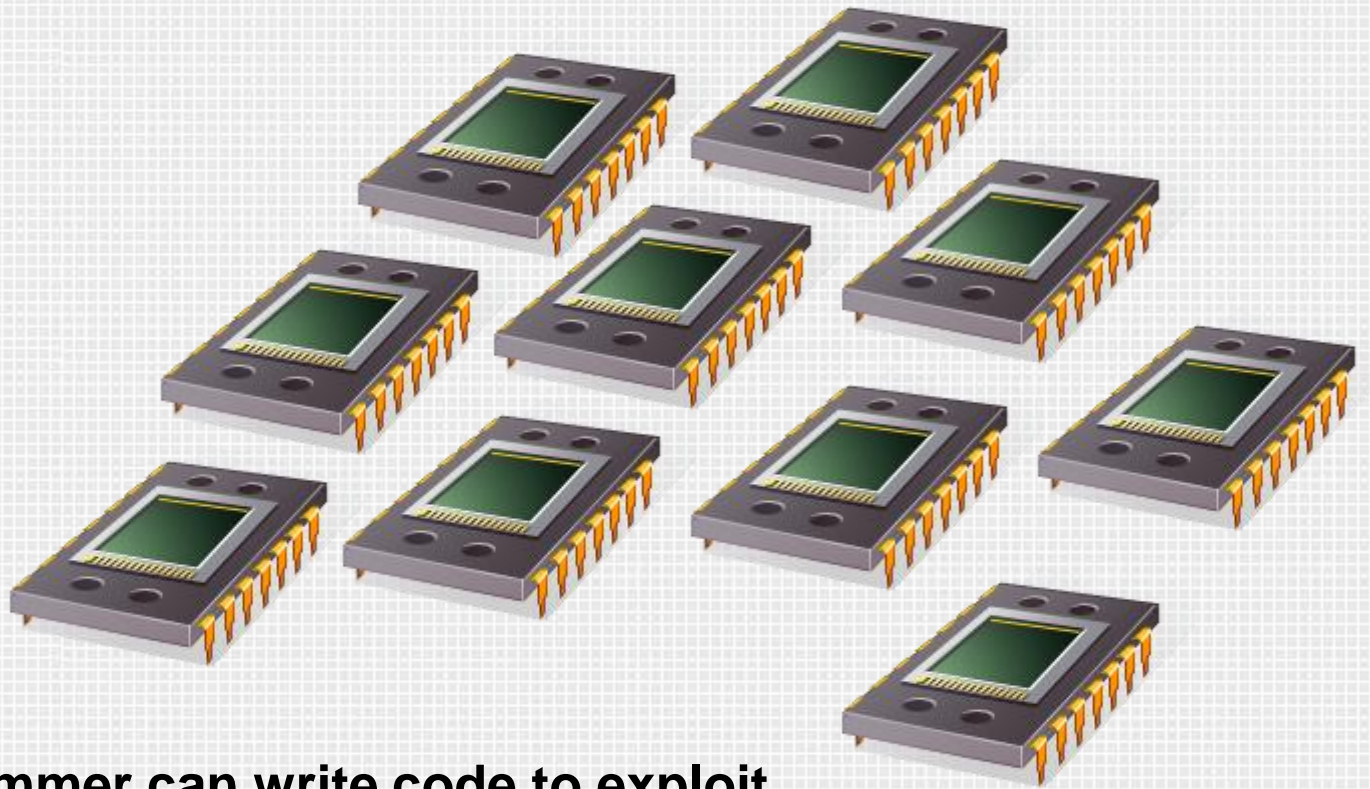
- 处理器在**每条指令**执行完后切换线程，从而跳过被阻塞的线程。
- **缺点：**一个线程包括多条指令，而每条指令的切换都需要等待，导致延迟。

- 粗粒度多线程

- 只切换那些需要**等待较长时间才能完成操作**的线程。



- 同步多线程（**Simultaneous multithreading**，**SMT**）
 - 细粒度多线程的变种。
 - 它通过允许多个线程同时使用多个功能单元来利用超标量处理器的性能。



A programmer can write code to exploit.

2.3 并行硬件



并行计算机系统及结构模型

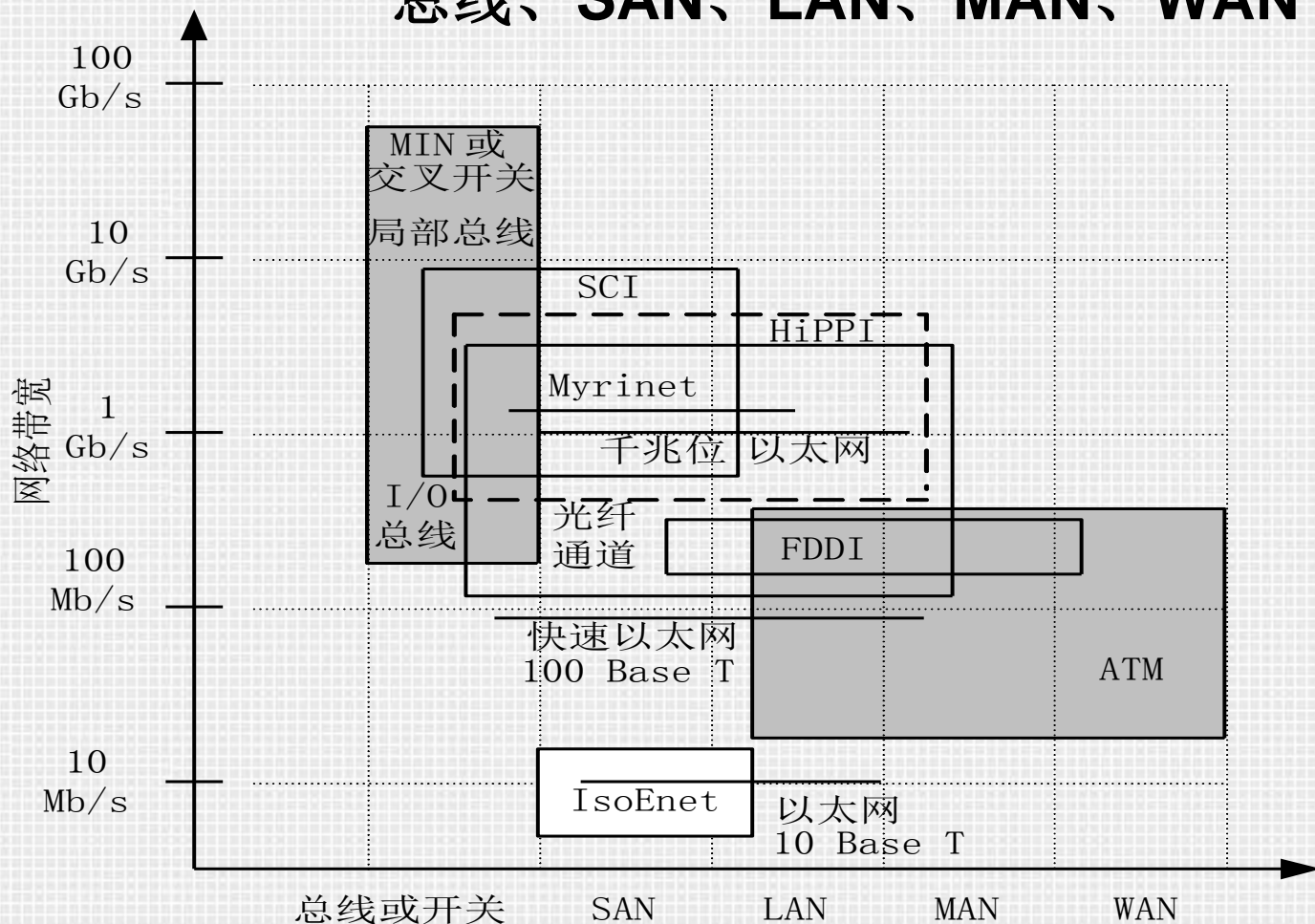
- 一、并行计算机系统互连
 - 1 系统互连
 - 2 静态互联网络
 - 3 动态互连网络
 - 4 标准互联网络
- 二、并行计算机系统结构
 - 1 并行计算机结构模型
 - 2 并行计算机访存模型



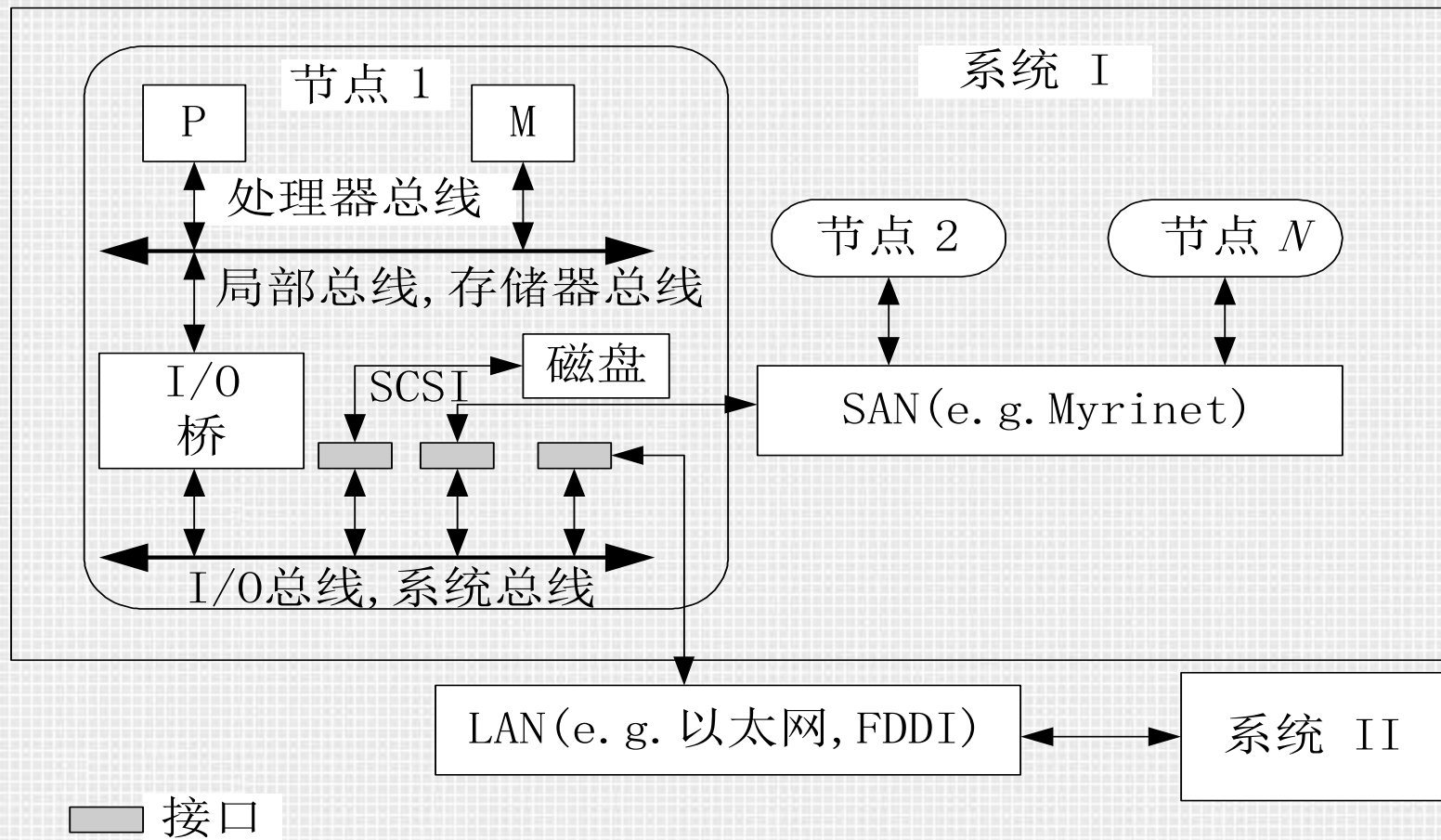
系统互连

- 不同带宽与距离的互连技术:

总线、SAN、LAN、MAN、WAN



局部总线、I/O总线、SAN和LAN





网络性能指标

- 节点度（**Node Degree**）：射入或射出一个节点的边数。在单向网络中，入射和出射边之和称为节点度。
- 网络直径（**Network Diameter**）：网络中任何两个节点之间的最长距离，即最大路径数。
- 对剖宽度（**Bisection Width**）：对分网络各半所必须移去的最少边数
- 对剖带宽（**Bisection Bandwidth**）：每秒钟内，在最小的对剖平面上通过所有连线的最大信息位（或字节）数
- 如果从任一节点观看网络都一样，则称网络为对称的（**Symmetry**）



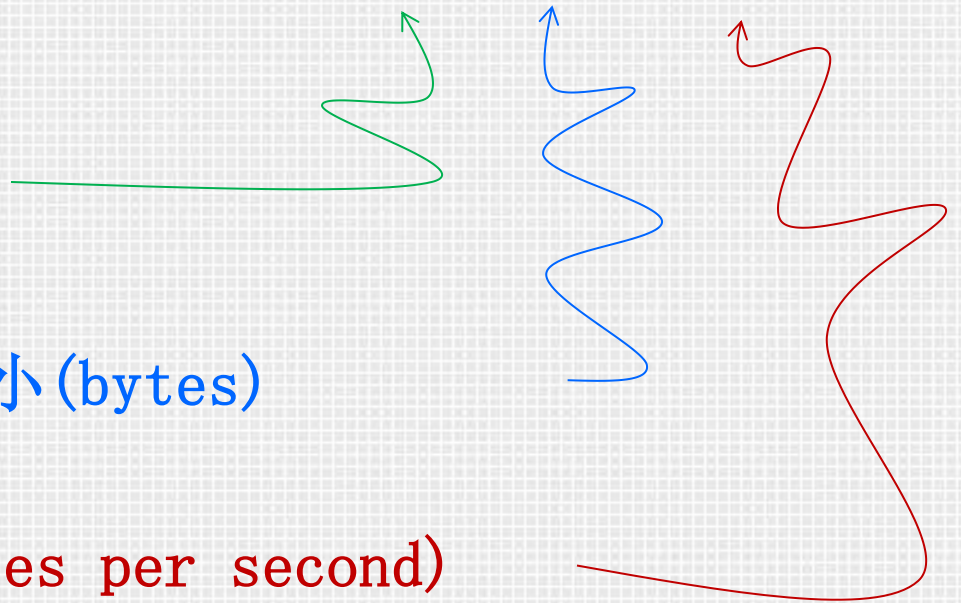
- 用来衡量互联网络的性能指标:
- 延迟
 - 指从发送源开始传送数据到目的地开始接收数据之间的时间
- 带宽
 - 指目的地在开始接收数据后接收数据的速度。


$$\text{Message transmission time} = L + n / b$$

延迟 (seconds)

消息大小 (bytes)

带宽 (bytes per second)





静态互连网络 与动态互连网络

- 静态互连网络：处理单元间有着固定连接的一类网络，在程序执行期间，这种点到点的链接保持不变；典型的静态网络有一维线性阵列、二维网孔、树连接、超立方网络、立方环、洗牌交换网、蝶形网络等
- 动态网络：用交换开关构成的，可按应用程序的要求动态地改变连接组态；典型的动态网络包括总线、交叉开关和多级互连网络等。



静态互连网络（1）

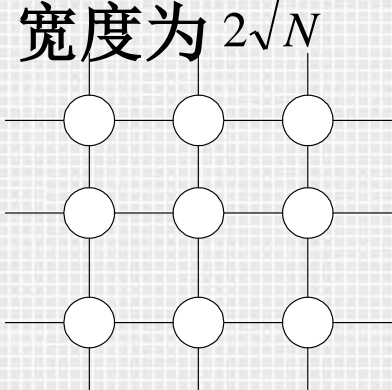
- **一维线性阵列（1-D Linear Array）：**
 - 并行机中最简单、最基本的互连方式，
 - 每个节点只与其左、右近邻相连，也叫二近邻连接，
 - **N 个节点用 $N-1$ 条边串接之，内节点度为2，直径为 $N-1$ ，对剖宽度为1**
 - 当首、尾节点相连时可构成循环移位器，在拓扑结构上等同于环，环可以是单向的或双向的，其节点度恒为2，直径或为 $\lfloor N/2 \rfloor$ （双向环）或为 $N-1$ （单向环），对剖宽度为2



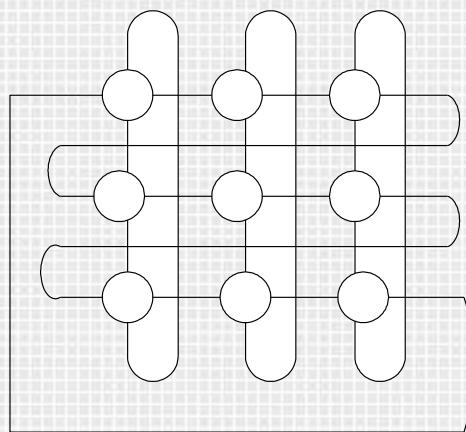
静态互连网络（2）

- $\sqrt{N} \times \sqrt{N}$ 二维网孔（2-D Mesh）：

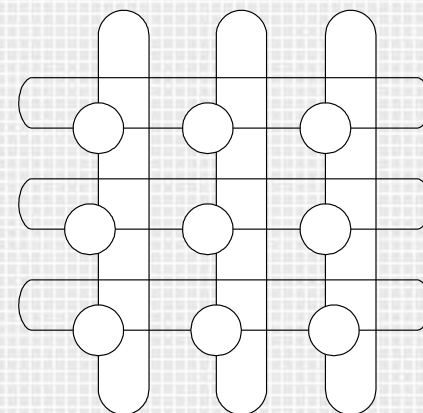
- 每个节点只与其上、下、左、右的近邻相连（边界节点除外），节点度为4，网络直径为 $2(\sqrt{N}-1)$ ，对剖宽度为 \sqrt{N}
- 在垂直方向上带环绕，水平方向呈蛇状，就变成 **Illiac**网孔了，节点度恒为4，网络直径为 $\sqrt{N}-1$ ，而对剖宽度为 $2\sqrt{N}$
- 垂直和水平方向均带环绕，则变成了**2-D环绕（2-D Torus）**，节点度恒为4，网络直径为 $2\lfloor\sqrt{N}/2\rfloor$ ，对剖宽度为 $2\sqrt{N}$



(a) 2-D网孔



(b) Illiac网孔

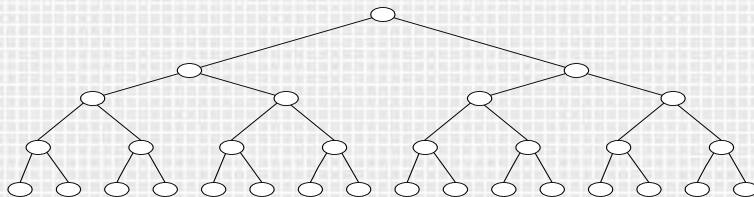


(c) 2-D环绕

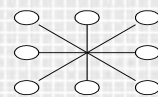


静态互连网络 (3)

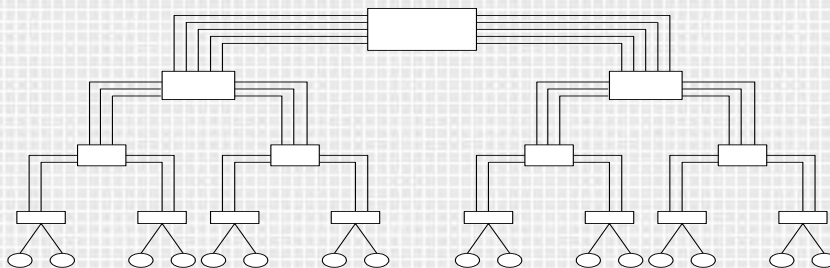
- 二叉树：
 - 除了根、叶节点，每个内节点只与其父节点和两个子节点相连。
 - 节点度为**3**，对剖宽度为**1**，而树的直径为 $2(\lceil \log N \rceil - 1)$
 - 如果尽量增大节点度为，则直径缩小为**2**，此时就变成了星形网络，其对剖宽度为 $\lfloor N/2 \rfloor$
 - 传统二叉树的主要问题是根易成为通信瓶颈。胖树节点间的通路自叶向根逐渐变宽。



(a) 二叉树



(b) 星形连接



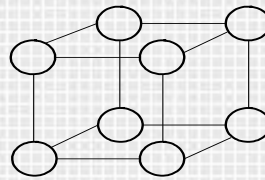
(c) 二叉胖树



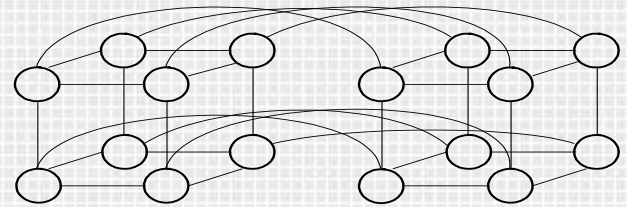
静态互连网络（4）

- 超立方：
 - 一个n-立方由 $N = 2^n$ 个顶点组成，3-立方如图(a)所示；4-立方如图(b)所示，由两个3-立方的对应顶点连接而成。
 - n-立方的节点度为n，网络直径也是n，而对剖宽度为 $N/2$ 。

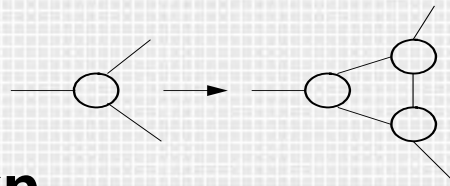
- 如果将3-立方的每个顶点代之以一个环就构成了如图(d)所示的3-立方环，此时每个顶点的度为3，而不像超立方那样节点度为n。



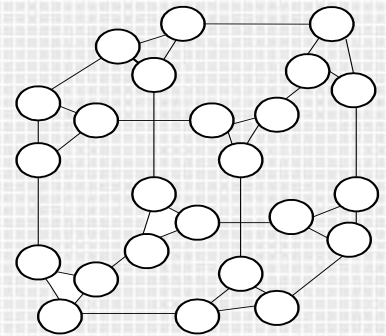
(a) 3-立方



(b) 4-立方



(c) 顶点代之以环



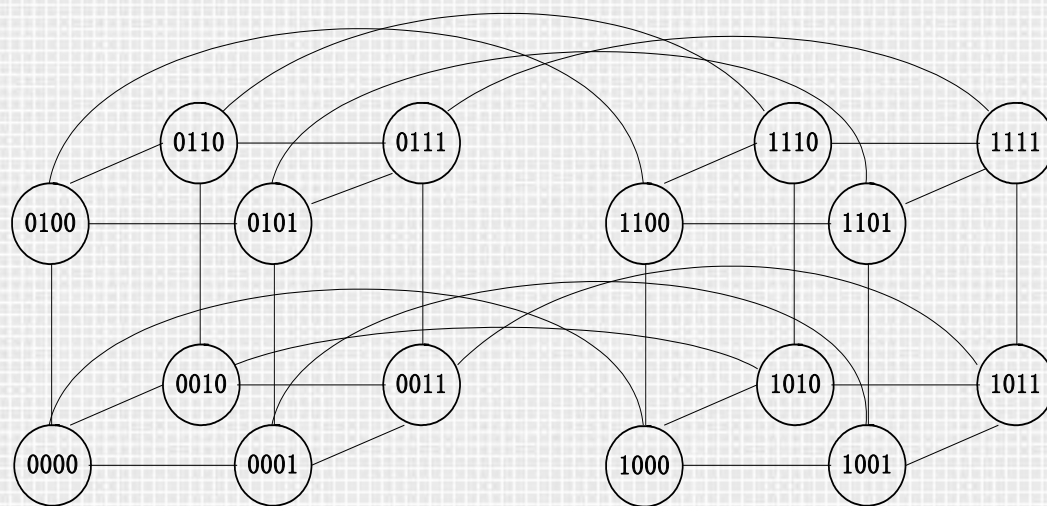
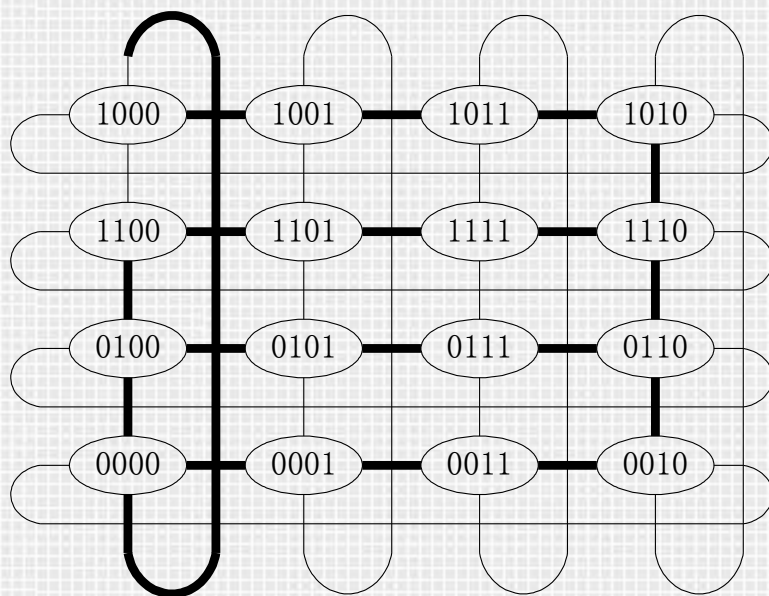
(d) 3-立方环



嵌入

- 将网络中的各节点映射到另一个网络中去
- 用膨胀（**Dilation**）系数来描述嵌入的质量，它是指被嵌入网络中的一条链路在所要嵌入的网络中对应所需的最大链路数
- 如果该系数为1，则称为完美嵌入。
- 环网可完美嵌入到**2-D**环绕网中
- 超立方网可完美嵌入到**2-D**环绕网中

嵌入





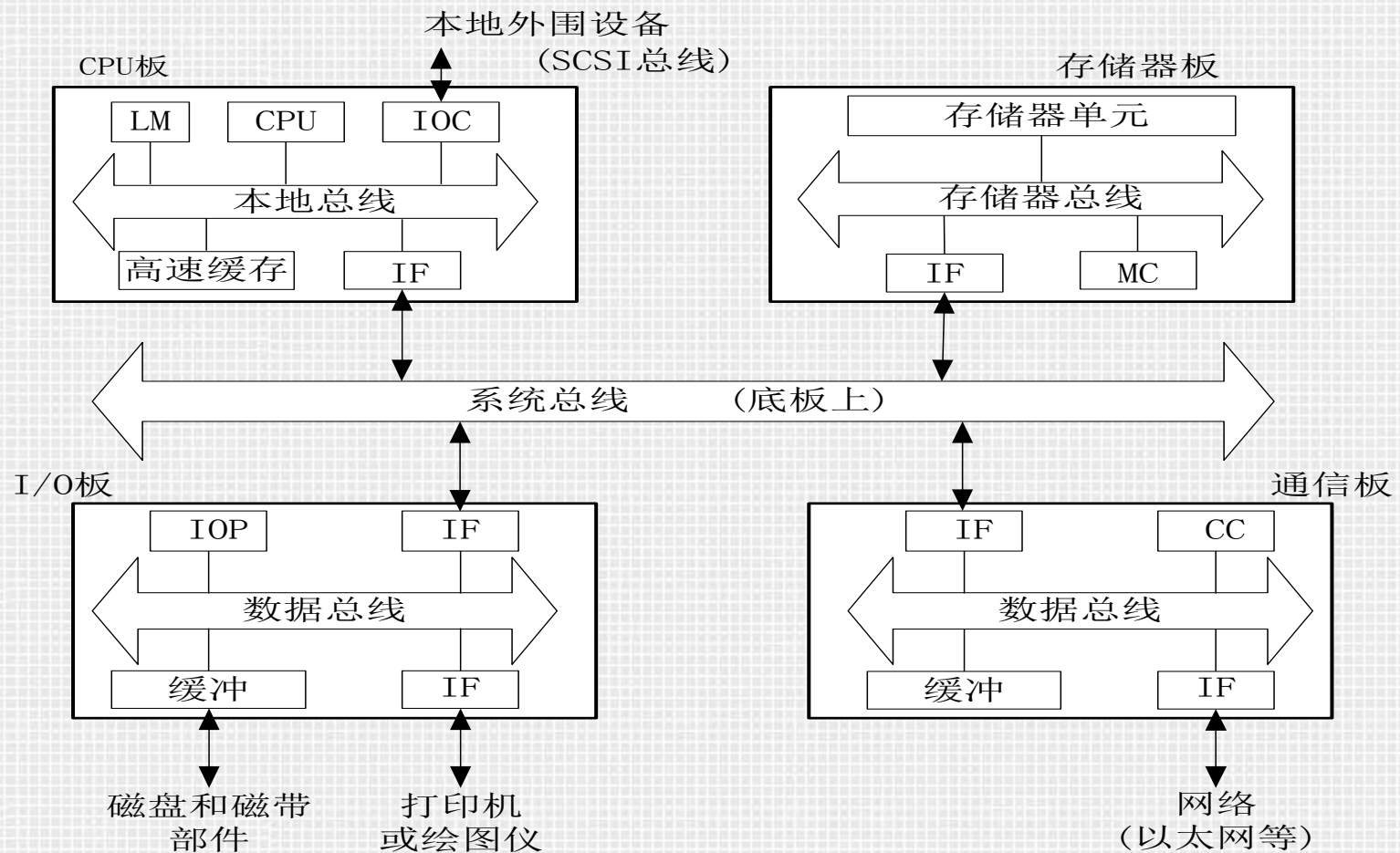
静态互连网络特性比较

| 网络名称 | 网络规模 | 节点度 | 网络直径 | 对剖宽度 | 对称 | 链路数 |
|----------|------------------------------|-------|-------------------------------|-----------------------|----|-----------------|
| 线性阵列 | N | 2 | $N-1$ | 1 | 非 | $N-1$ |
| 环形 | N | 2 | $\lfloor N/2 \rfloor$ (双向) | 2 | 是 | N |
| 2-D网孔 | $(\sqrt{N} \times \sqrt{N})$ | 4 | $2(\sqrt{N}-1)$ | \sqrt{N} | 非 | $2(N-\sqrt{N})$ |
| Illiac网孔 | $(\sqrt{N} \times \sqrt{N})$ | 4 | $\sqrt{N}-1$ | $2\sqrt{N}$ | 非 | $2N$ |
| 2-D环绕 | $(\sqrt{N} \times \sqrt{N})$ | 4 | $2\lfloor \sqrt{N}/2 \rfloor$ | $2\sqrt{N}$ | 是 | $2N$ |
| 二叉树 | N | 3 | $2(\lceil \log N \rceil - 1)$ | 1 | 非 | $N-1$ |
| 星形 | N | $N-1$ | 2 | $\lfloor N/2 \rfloor$ | 非 | $N-1$ |
| 超立方 | $N = 2^n$ | n | n | $N/2$ | 是 | $nN/2$ |
| 立方环 | $N = k \cdot 2^k$ | 3 | $2k-1 + \lfloor k/2 \rfloor$ | $N/(2k)$ | 是 | $3N/2$ |



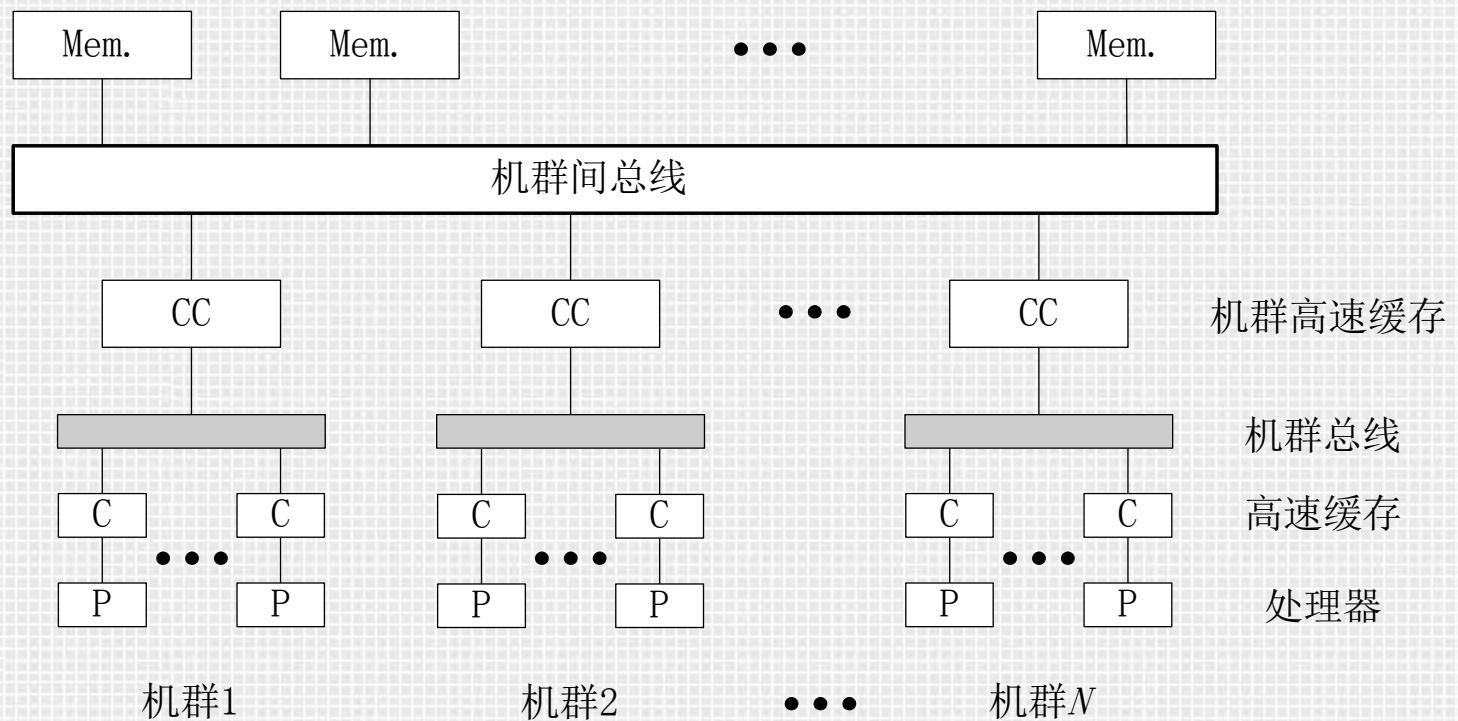
动态互连网络 (1)

- 总线：PCI、VME、Multics、Sbus、MicroChannel
 - 多处理机总线系统的主要问题包括总线仲裁、中断处理、协议转换、快速同步、高速缓存一致性协议、分事务、总线桥和层次总线扩展等





• 层次总线：IEEE Futurebus



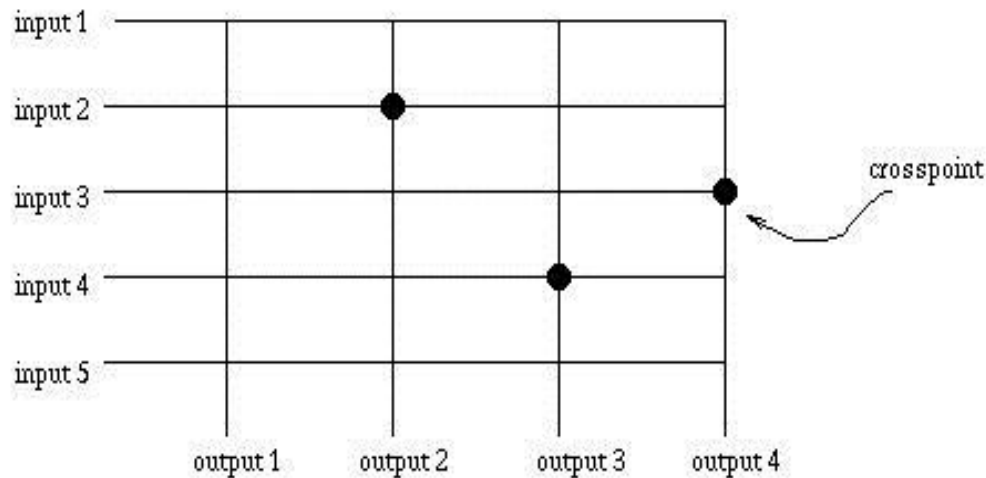


- 总线的优点在于成本低，不随处理器数目的增加而增加
- 总线的缺点在于扩展性不好，总线的带宽固定，随着处理器数的增加，每个处理器带宽减少。
- 可利用程序中的局部性原理减少对总线带宽的需求



动态互连网络 (2)

- 交叉开关（**Crossbar**）网络是单级交换网络，可为每个端口提供更高的带宽。象电话交换机一样，交叉点开关可由程序控制动态设置其处于“开”或“关”状态，而能提供所有（源、目的）对之间的动态连接。
- 在并行处理中，交叉开关一般有两种使用方式：一种是用于对称的多处理机或多计算机机群中的处理器间的通信；另一种是用于**SMP**服务器或向量超级计算机中处理器和存储器之间的存取。



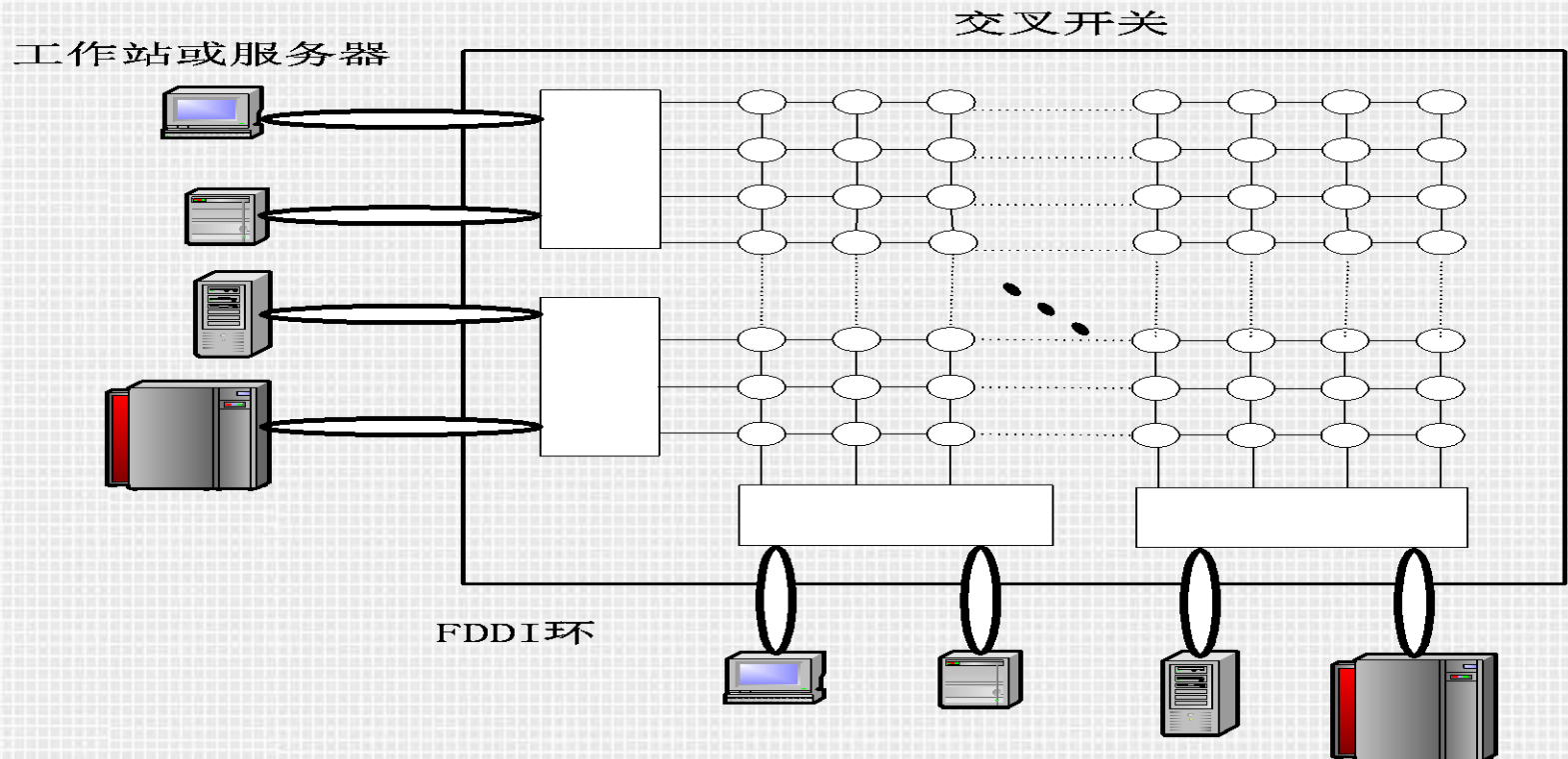
Input 2 is connected to output 2,
input 3 is connected to output 4,
input 4 is connected to output 3.

Note: Connections between free
inputs and outputs are always possible.

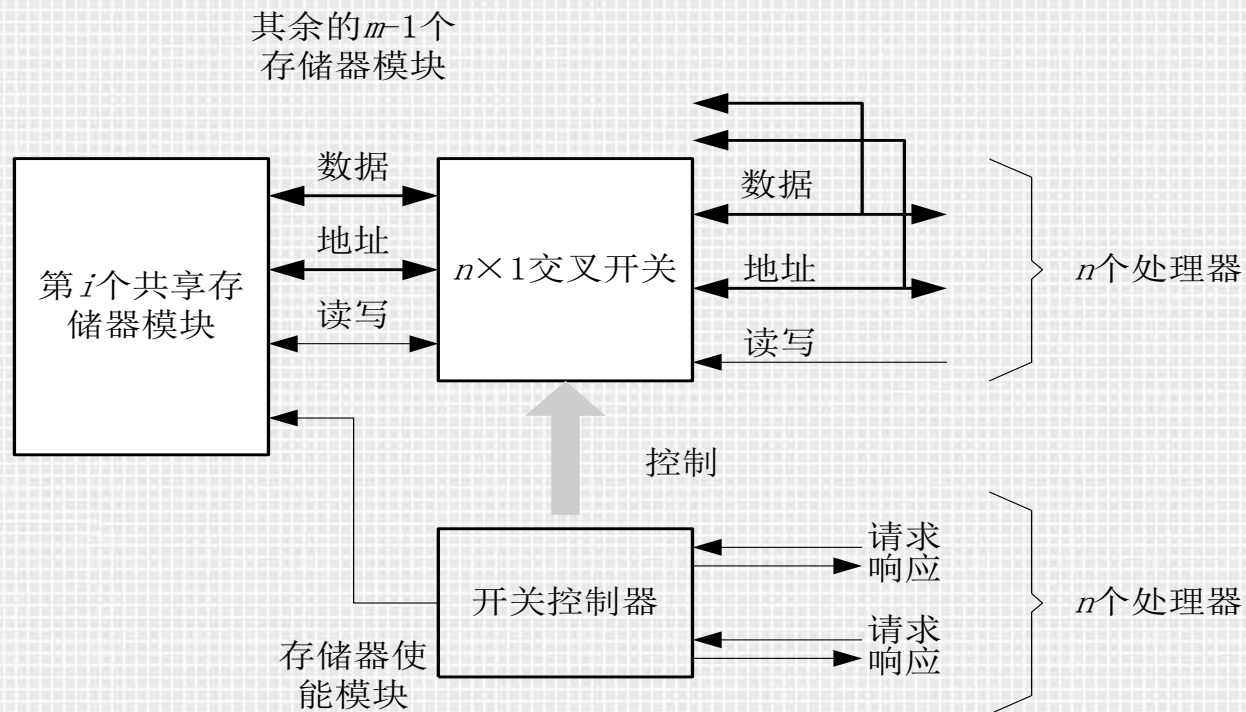
A Small Crossbar Switch



- 千兆开关/FDDI: 一种用于构造Alpha工作站和服务器的互连的交叉开关，带宽3.6Gbps
- Sun Microsystems公司在它们的Ultra Enterprise 10000 (StarFire) SMP服务器中，将Gigaplane总线升级成Gigaplane-XB互连



- 处理器和存储器间的交叉开关：
 - 交叉开关代替处理器和存储器间的连接总线
 - 提供了多个处理器模块并行存取存储器的可能性
 - 每个时刻每个存储器模块只能由一个处理器进行访问





- 交叉开关具有良好的带宽特性
- **Non-Blocking:** 两个节点之间的通信，不会阻塞其他节点之间的通信。
- 代价不可扩放， $O(P^2)$

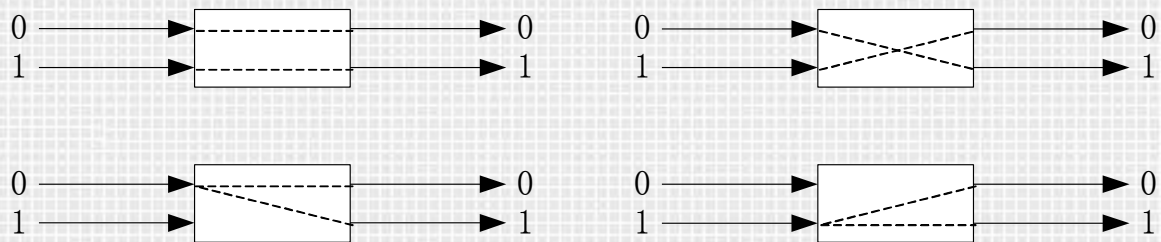


动态互连网络 (3)

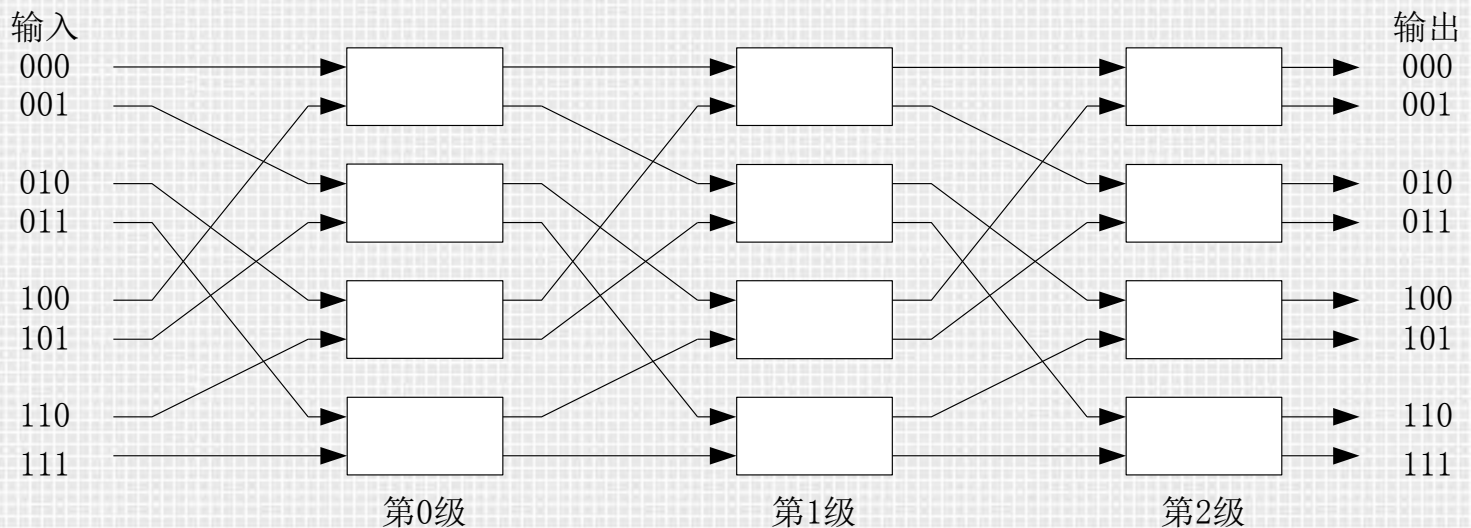
- 交换开关模块：
 - 一个交换开关模块有 n 个输入和 n 个输出，每个输入可连接到任意输出端口，但只允许一对一或一对多的映射，不允许多对一的映射，因为这将发生输出冲突
- 级间互连（**Interstage Connection**）：
 - 均匀洗牌、蝶网、多路均匀洗牌、交叉开关、立方连接
 - n 输入的 Ω 网络需要 $\log_2 n$ 级 2×2 开关，在Illinois大学的Cedar[2]多处理机系统中采用了 Ω 网络
 - **Cray Y/MP**多级网络，该网络用来支持8个向量处理器和256个存储器模块之间的数据传输。网络能够避免8个处理器同时进行存储器存取时的冲突。



- 单级交叉开关级联起来形成多级互连网络MIN
(Multistage Interconnection Network)



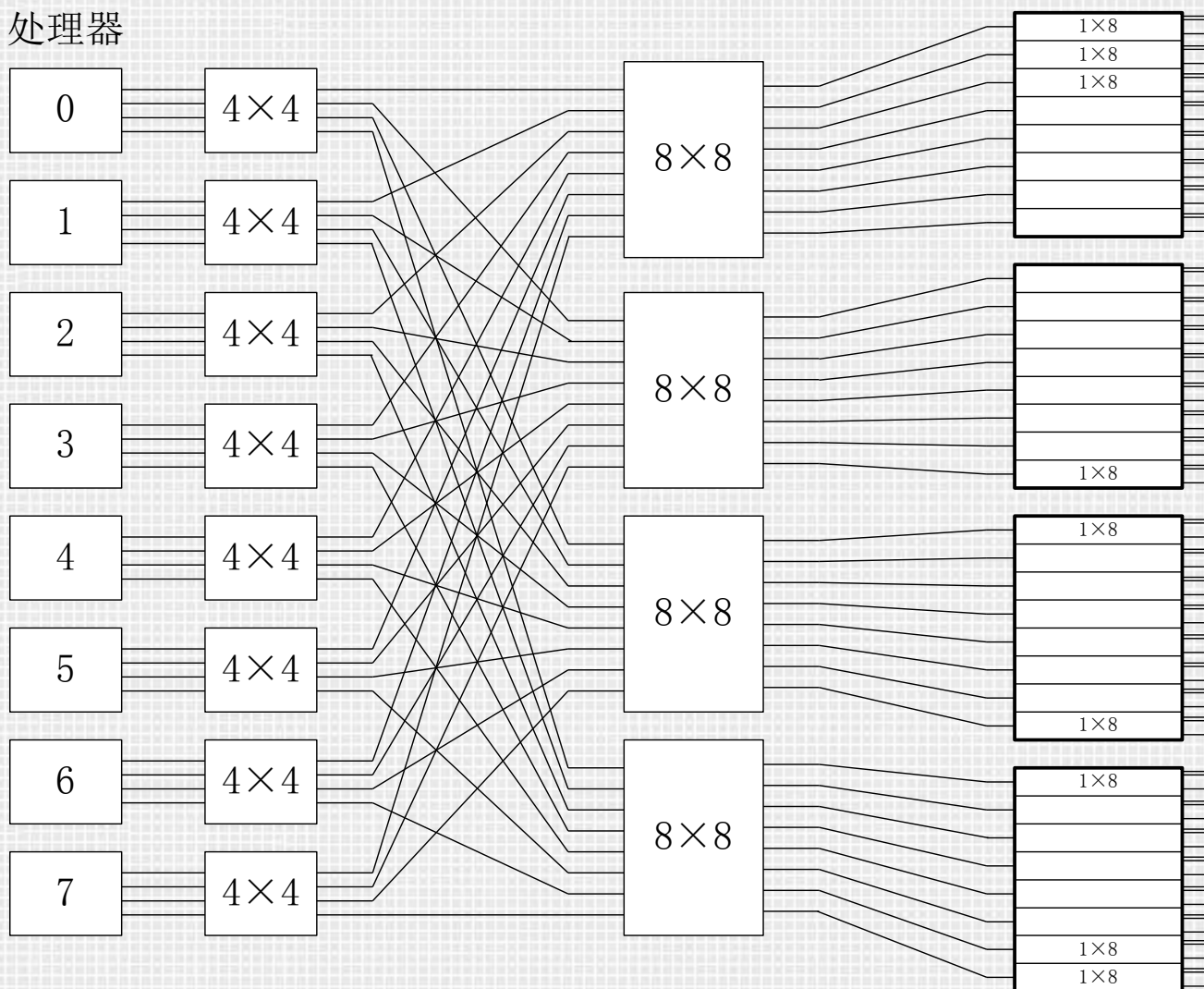
(a) 4种可能的开关连接



(b) 一种8输入的Omega网络



处理器



0, 4, 8, ..., 28
32, 36, 40, ..., 60
64, 68, 72, ..., 92

⋮

224, 228, ..., 252

1, 5, 9, ..., 29

存储器模块

225, 229, ..., 253

2, 6, 10, ..., 30

⋮

226, 230, ..., 254

3, 7, 11, ..., 31

⋮

227, 231, ..., 256



动态互连网络比较

| 动态互连网络的复杂度和带宽性能一览表 | | | |
|--------------------|--|---|-----------------------------------|
| 网络特性 | 总线系统 | 多级互连网络 | 交叉开关 |
| 硬件复杂度 | $O(n + w)$ | $O((n \log_k n)w)$ | $O(n^2 w)$ |
| 每个处理器带宽 | $O(wf / n) \sim O(wf)$ | $O(wf)$ | $O(wf)$ |
| 报道的聚集带宽 | SunFire服务器中的Gigaplane总线: 2.67GB/s | IBM SP2中的512 节点的HPS: 10.24GB/s | Digital的千兆开 关: 3.4GB/s |

- **n**, 节点规模 **w**, 数据宽度

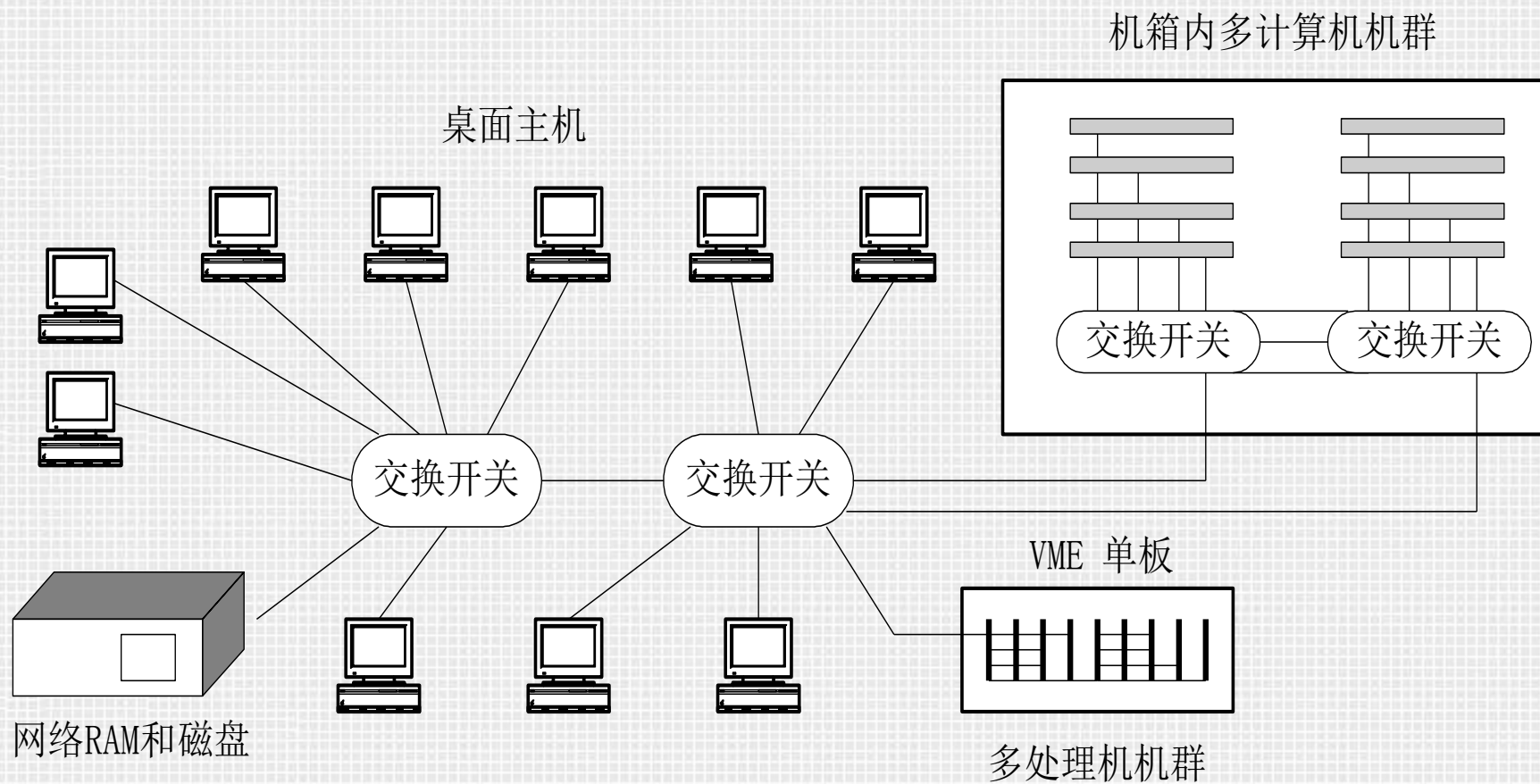


标准互联网络（1）

- **Myrinet:**

- **Myrinet**是由**Myricom**公司设计的千兆位包交换网络，其目的是为了构筑计算机机群，使系统互连成为一种商业产品。
- **Myrinet**是基于加州理工学院开发的多计算机和**VLSI**技术以及在南加州大学开发的**ATOMIC/LAN**技术。**Myrinet**能假设任意拓扑结构，不必限定为开关网孔或任何规则的结构。
- **Myrinet**在数据链路层具有可变长的包格式，对每条链路施行流控制和错误控制，并使用切通选路法以及定制的可编程的主机接口。在物理层上，**Myrinet**网使用全双工**SAN**链路，最长可达3米，峰值速率为（**1.28+1.28**）**Gbps**（目前有**2.56+2.56**）
- **Myrinet**交换开关：**8,12,16**端口
- **Myrinet**主机接口：**32**位的称作**LANai**芯片的用户定制的**VLSI**处理器，它带有**Myrinet**接口、包接口、**DMA**引擎和快速静态随机存取存储器**SRAM**。
- **140 of the November 2002 TOP500 use Myrinet, including 15 of the top 100**

Myrinet连接的LAN/Cluster



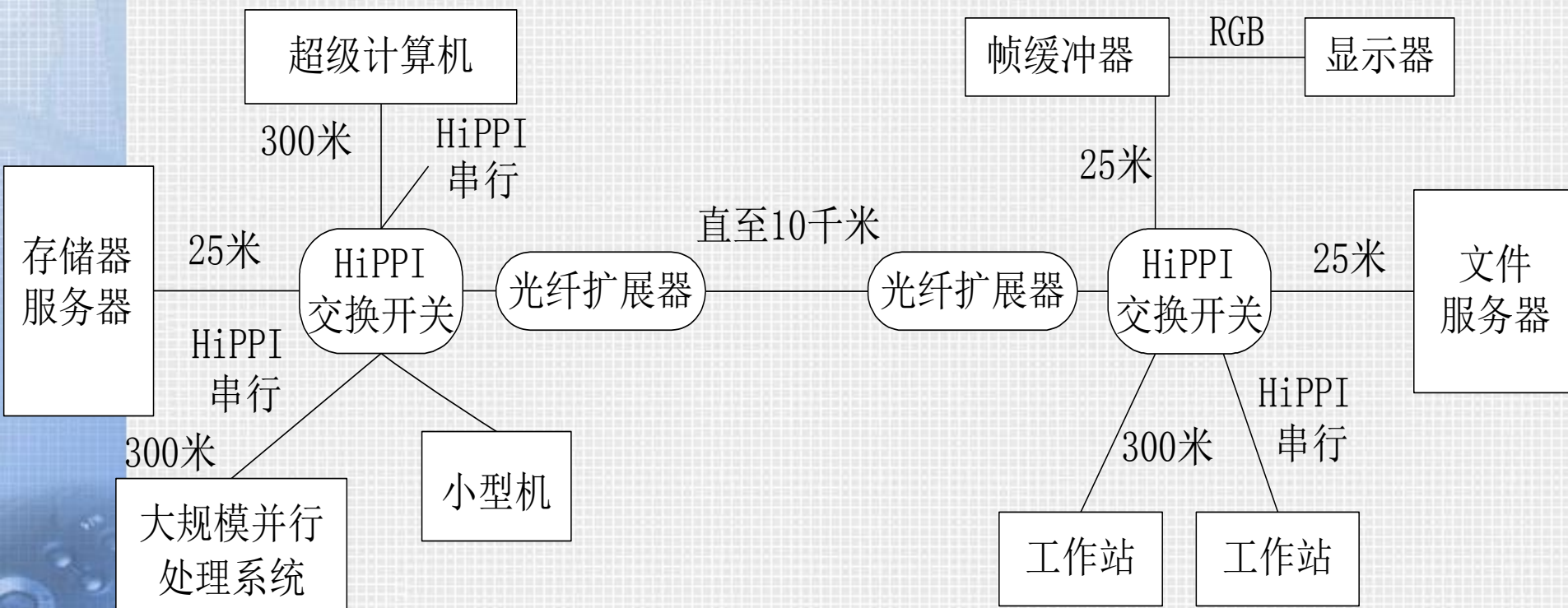


标准互联网络（2）

- 高性能并行接口（HiPPI）

- **Los Alamos**国家实验室于**1987**年提出的一个标准，其目的是试图统一来自不同产商生产的所有大型机和超级计算机的接口。在大型机和超级计算机工业界，**HiPPI**作为短距离的系统到系统以及系统到外设连接的高速**I/O**通道。
- **1993**年，**ANSI X3T9.3**委员会认可了**HiPPI**标准，它覆盖了物理和数据链路层，但在这两层之上的任何规定却取决于用户。
- **HiPPI**是个单工的点到点的数据传输接口，其速率可达**800Mbps**到**1.6Gbps**。
- 开发成功了一种能提供潜在的**6.4Gbps**速率，比**HiPPI**快**8**倍且有很低时延的超级**HiPPI**技术，
- **SGI**公司和**Los Alamos**国家实验室都开发了用来构筑速率高达**25.6Gbps**的**HiPPI**交换开关的**HiPPI**技术。
- **HiPPI**通道和**HiPPI**交换开关被用在**SGI Power Challenge**服务器、**IBM 390**主机、**Cray Y/MP**、**C90**和**T3D/T3E**等系统

使用HiPPI通道和开关构筑的LAN主干网



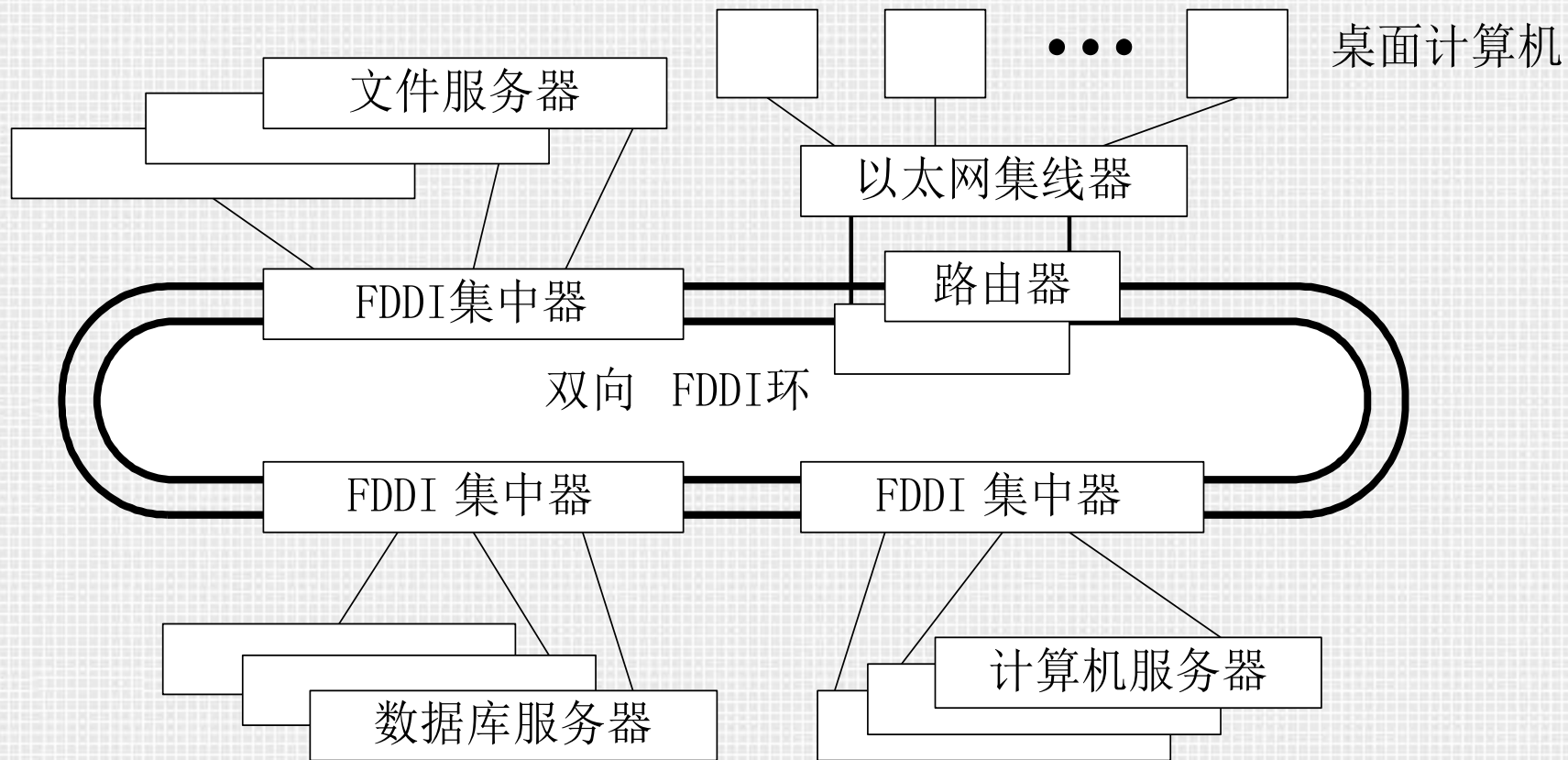


标准互联网络（3）

- **光纤通道FC（Fiber Channel）：**
 - 通道和网络标准的集成
 - 光纤通道既可以是共享介质，也可以是一种交换技术
 - 光纤通道操作速度范围可从**100到133、200、400和800Mbps**。**FCSI**厂商也正在推出未来具有更高速度（**1、2或4Gbps**）的光纤通道
 - 光纤通道的价值已被现在的某些千兆位局域网所证实，这些局域网就是基于光纤通道技术的
 - 连网拓扑结构的灵活性是光纤通道的主要财富，它支持点到点、仲裁环及交换光纤连接
- **FDDI：**
 - 光纤分布式数据接口**FDDI（Fiber Distributed Data Interface）**
 - **FDDI**采用双向光纤令牌环可提供**100-200Mbps**数据传输速率
 - **FDDI**具有互连大量设备的能力
 - 传统的**FDDI**仅以异步方式操作



双向FDDI环作为主干网

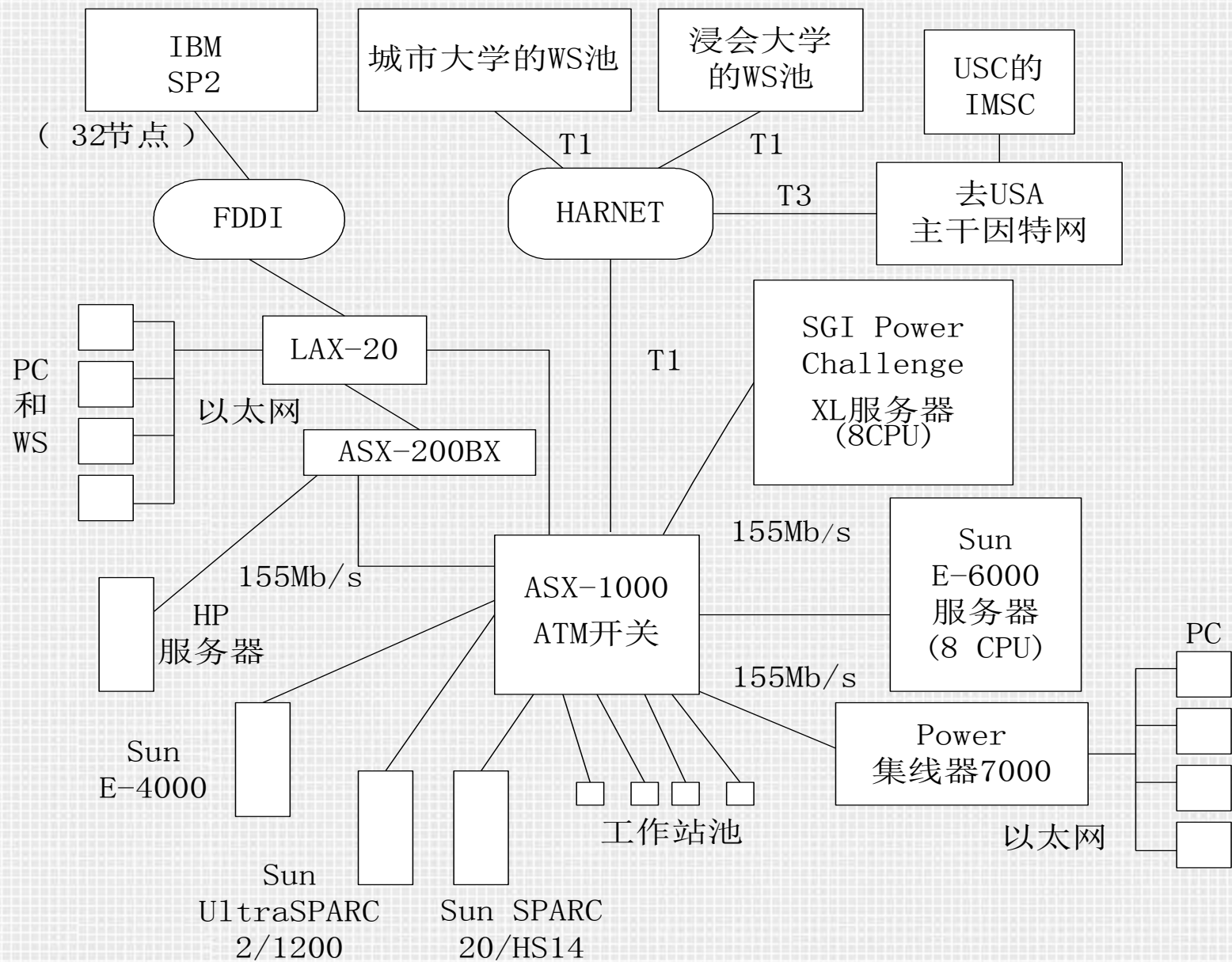




标准互联网络（4）

- **ATM（Asynchronous Transfer Mode）：**
 - 由成立于**1991**年的**ATM**论坛和**ITU**标准定义。
 - **ATM**是一种独立于介质的消息传输协议，它将消息段变成更短的固定长度为**53**字节的报元进行传输。
 - 这种技术是基于报元交换机制。**ATM**的目的是将实时和突发数据的传输合并成单一的网络技术。
 - **ATM**网络支持从**25**到**51**、**155**和**622Mbps**不同的速率，其速率越低**ATM**交换器和使用的链路价格越低。

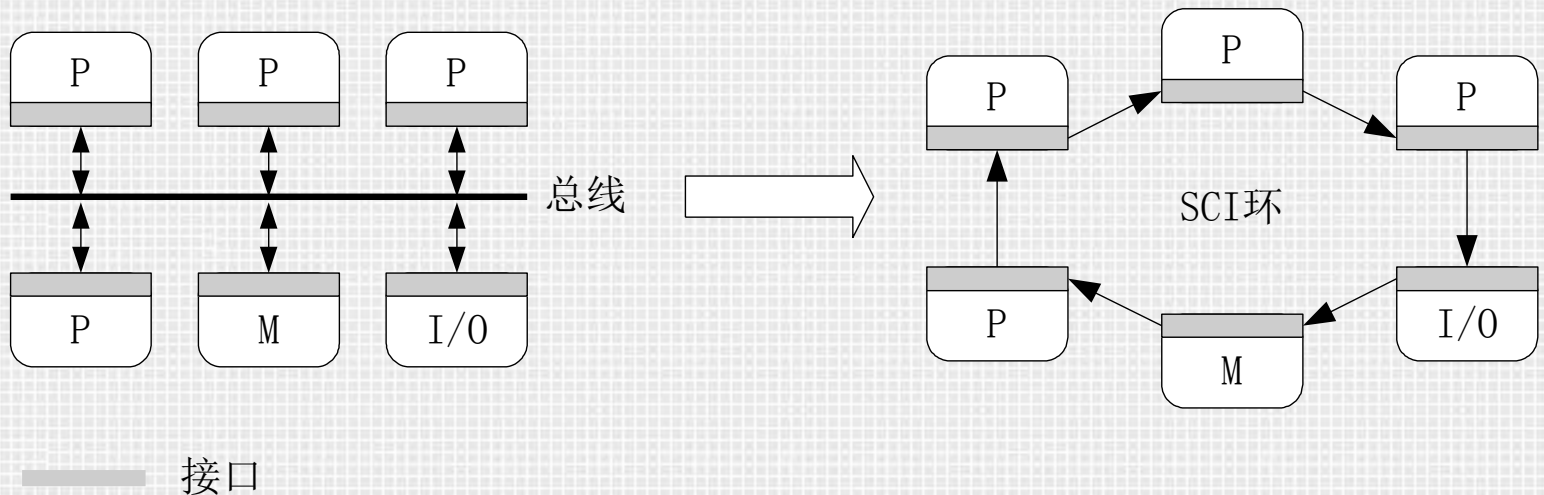
香港大学开发的Pearl机群





标准互联网络（5）

- **SCI（Scalable Coherence Interface）**可扩展一致性接口
 - 既能保持总线的优点又具有传统网络空间的可扩展性
 - 通常的底板总线扩展成全双工、点到点的互连结构
 - 提供分布共享存储器一致的高速缓存映象
 - **SCI**被设计用来提供低时延（小于1微秒）和高带宽（高至**8GB/s**）的点到点互连。
 - 一旦**SCI**得到充分开发，就可连接多至**64K**个节点。





标准互联网络（5）

| 代别 类型 | | 以太网 10BaseT | 快速以太网 100BaseT | 千兆位以太网 1GB |
|----------|--------------------|----------------|------------------------------|--|
| 引入年代 | | 1982 | 1994 | 1997 |
| 速度（带宽） | | 10Mb/s | 100Mb/s | 1Gb/s |
| 最大 距离 | UTR（非屏蔽双扭对） | 100m | 100m | 25—100m |
| | STP（屏蔽双扭对） 同轴电缆 | 500m | 100m | 25—100m |
| | 多模光纤 | 2Km | 412m（半双工） 2Km（全双工） | 500m |
| | 单模光纤 | 25Km | 20Km | 3Km |
| 主要应用领域 | | 文件共享， 打印机共享 | COW计算， C/S结构， 大型数据库存取等 | 大型图像文件， 多媒体， 因特网， 内部网， 数据仓库等 |



Flynn分类法：指令流和数据流数目

典型的冯.诺依曼系统

SISD

单指令流
单数据流

(SIMD)

单指令
多数据流

MISD

多指令流
单数据流

(MIMD)

多指令流
多数据流

不在考虑行列



并行计算机分类

- **Flynn分类（1966年）**

- (1)单指令流单数据流机**SISD**，即传统的单处理机

- (2)单指令流多数据流机**SIMD**

- (3)多指令流单数据流机**MISD**，实际中不存在的机器

- (4)多指令流多数据流机**MIMD**

- **并行机的结构模型—实际的机器体系结构**

- SIMD (Single Instruction Multiple Data**, 单指令流多数据流机)

- PVP (Parallel Vector Processor**, 并行向量机)

- SMP (Symmetric Multiprocessor**, 对称多处理机)

- MPP (Massively Parallel Processor**, 大规模并行处理机)

- COW (Cluster of Workstation**, 工作站机群)

- DSM (Distributed Shared Memory**, 分布共享存储多处理机)

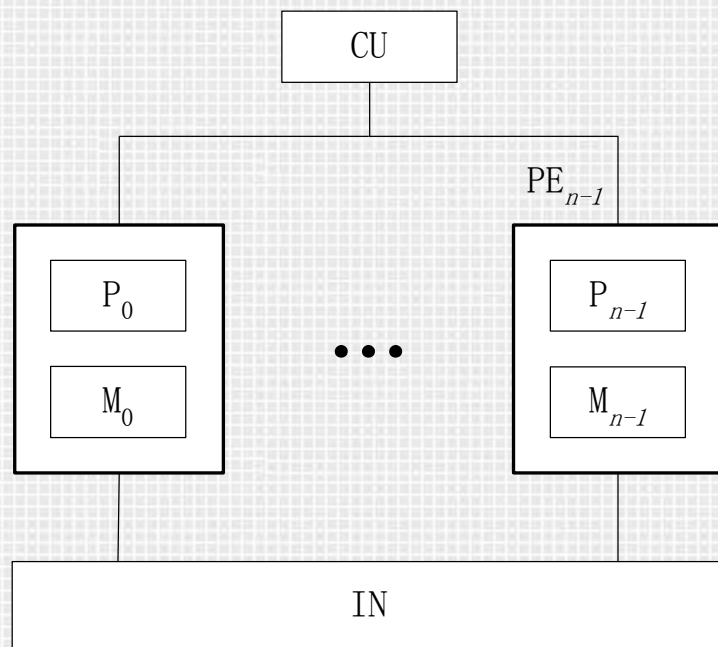
注：**SIMD**是专用并行机，后5种属于**MIMD**并行机。



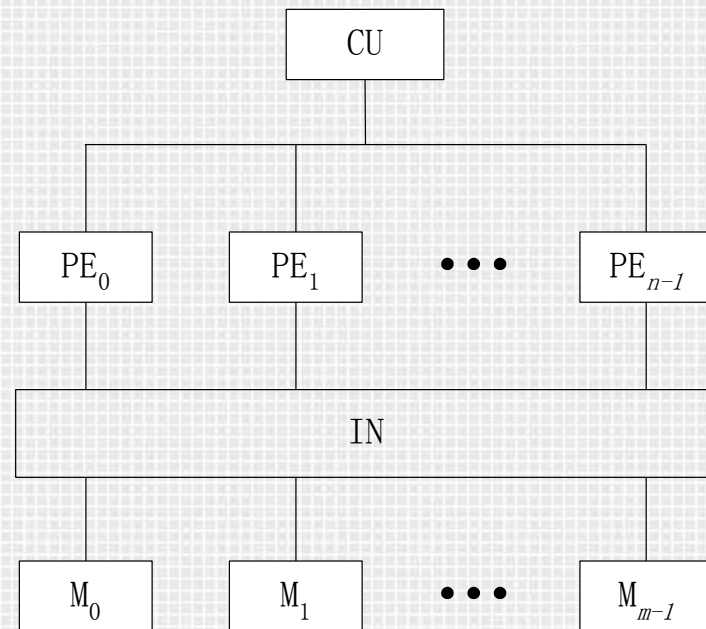
典型并行计算机系统

- 阵列处理机 { 分布存储
共享存储
- 向量处理机 { 流水线
并行向量机
- 共享存储多处理机 { 紧耦合多机系统
同构对称对机系统
- 分布存储多计算机 { **DSM/SVM**

阵列处理机的两种基本结构



(a) 分布存储阵列机



(b) 共享存储阵列机

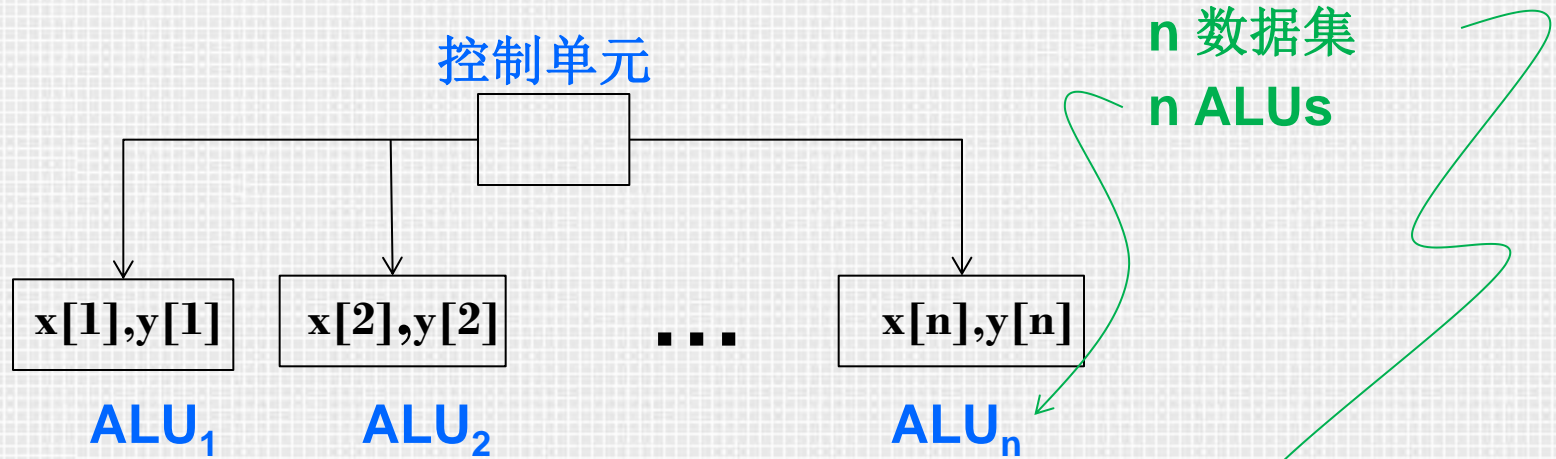


2.3.1 单指令多数据流SIMD

- 通过对多个数据**执行相同的指令**从而实现在多个数据流上的操作。
 - 一个控制单元和多个**ALU**
 - 适合对大型数组的简单循环实行并行化
- 通过**将数据分配给多个处理器**，然后让各个处理器**使用相同的指令来操作**数据子集实现并行化。这种并行也叫**数据并行**。
- **同步的。**



SIMD example



```
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

SIMD

- 如果**ALUs** 和数据集的数目不同?
- 拆分和迭代!
- **Ex. $m = 4$ ALUs and $n = 15$ data items.**

| Round | ALU ₁ | ALU ₂ | ALU ₃ | ALU ₄ |
|-------|------------------|------------------|------------------|------------------|
| 1 | X[0],y[0] | X[1],y[1] | X[2],y[2] | X[3],y[3] |
| 2 | X[4],y[4] | X[5],y[5] | X[6],y[6] | X[7],y[7] |
| 3 | X[8],y[8] | X[9],y[9] | X[10],y[10] | X[11],y[11] |
| 4 | X[12],y[12] | X[13],y[13] | X[14],y[14] | |



SIMD 缺点

- 所有**ALUs**都得执行同样的指令；
- **ALU**必须同步操作；
- **ALU**没有指令寄存器；
- 只适合解决大型数组的简单循环问题，而不能解决其他更复杂问题。



向量处理器 **Vector processors**

- 能够对数组或者数组向量进行操作，而传统的**CPU**是对单独的数据元素或者标量进行操作。



- **系统的特征:**
- **向量寄存器**
 - 能够存储由多个操作数组成的向量，并能够同时对其内容进行操作。
- **向量化和流水化的功能单元**
 - 对向量中的每个元素做的是同样的操作。
- **向量指令**
 - 在向量上的操作的指令，而不是标量上的。



- 交叉存储器

- 内存系统可以看做由多个内存‘体’组成，每个内存体能够独立访问。
- 如果向量中的各个元素分布在不同的内存体中，那么在存入/存储连续数据时能够几乎无延迟的访问。

- 步长式存储器访问和硬件散射/聚集

- 程序能偶访问向量中固定间隔的元素
- 散射/聚集是对无规律间隔的数据进行读和写



向量处理器的优点



- 快
- 容易使用
- 向量编译器擅长识别向量化的代码
- 识别不能向量化的循环，并提供原因
- 很高的内存带宽
- 每个加载的数据都会被使用。



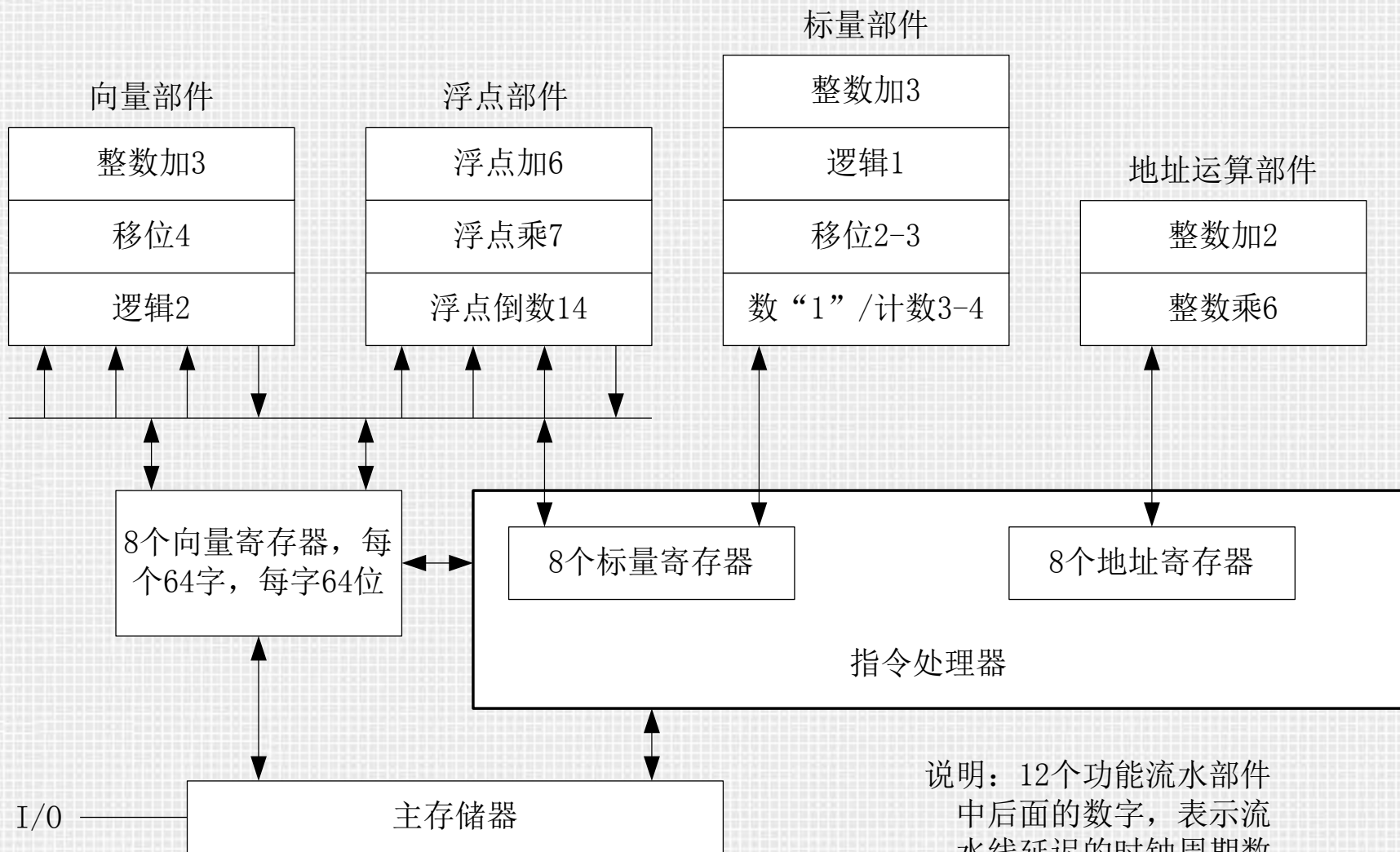
向量处理器缺点



- 它不能处理不规则的数据结构和其他的并行结构
- 处理更大的问题的能力有限，可扩展性差。



Cray-1的向量处理

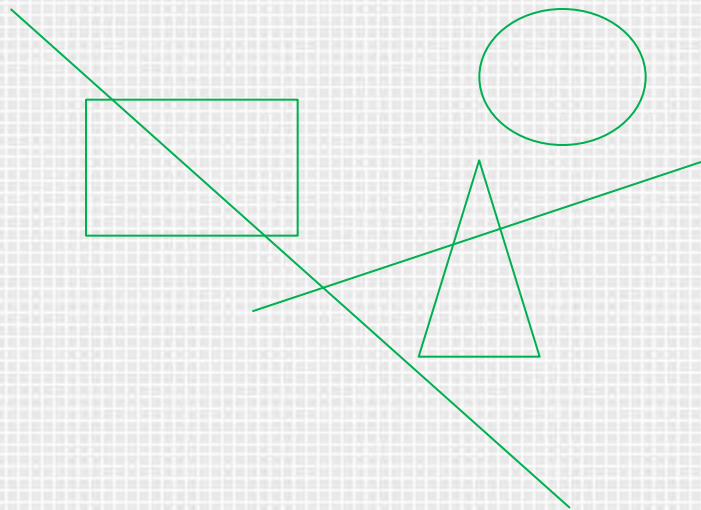




图形处理单元

Graphics Processing Units (GPU)

- 实时图形应用编程接口使用点、线、三角形来表示物体的表面。





GPUs

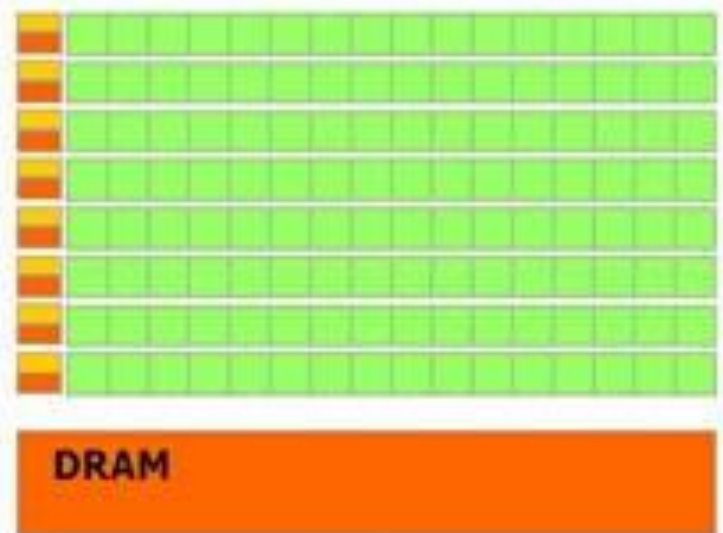
- 图形处理流水线，将物体表面转换为一个像素的数组来表示。
- 现阶段，所有的**GPU**都使用**SIMD**并行。
 - 每个**GPU**处理核中引入了大量的**ALU**。







CPU



GPU

Figure 1-2. The GPU Devotes More Transistors to Data Processing



2.3.2 多指令多数据流系统MIMD

- 支持同时多个指令在多个数据流上操作。
- **MIMD**系统通常包括一组完全独立的处理单元（核），每个处理单元都有自己的控制单元和**ALU**。
- 异步的



共享内存系统Shared Memory System

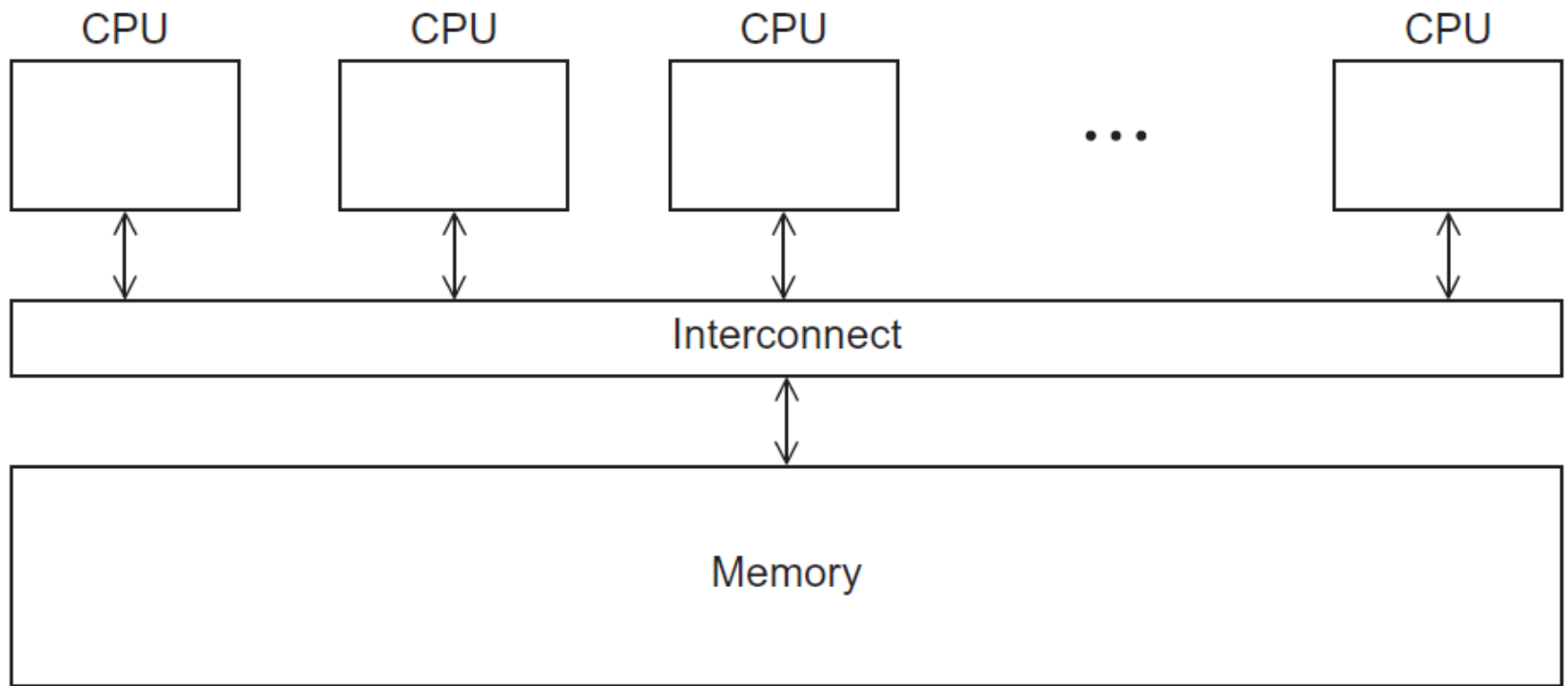
- 一组自治的处理器通过互连网络与内存系统相互连接。
- 每个处理器能够访问每个内存区域。
- 处理器之间通过访问共享的数据结构来进行隐式的通信。



共享存储的多处理机

- **MIMD—多指令多数据流机**
- 单一的共享地址空间
- 易于编程、难于扩展
- 存储访问可成为性能瓶颈
- 紧耦合与同构对称方式

共享内存系统 Shared Memory System

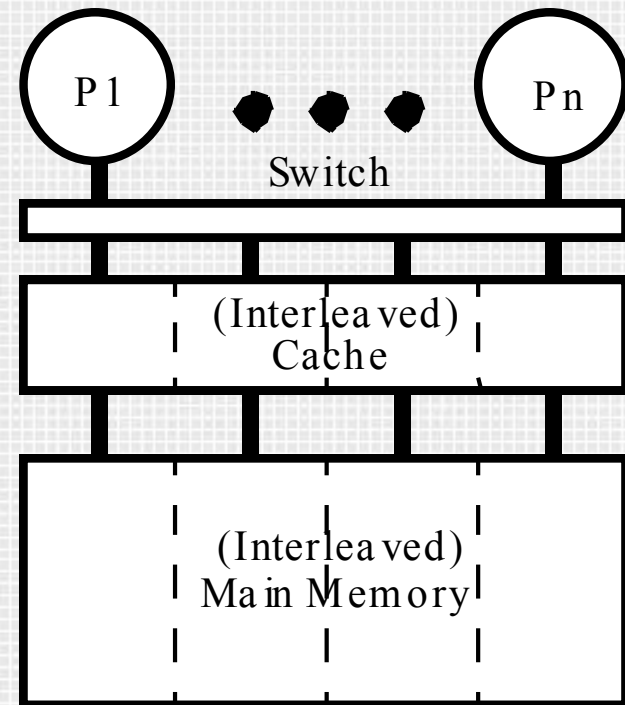


一个共享内存系统



共享高速缓存

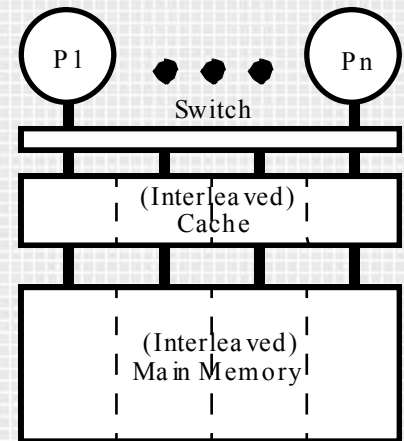
- **Alliant FX-8**
 - 产生1980's早期
 - 八个 68020s 使用带有 512 KB的交叉cache
- **Encore & Sequent**
 - 首先是32-bit 微指令 (N32032)
 - 其次是使用共享的cache





优点

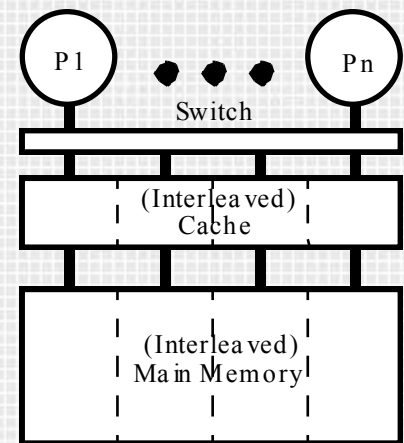
- 一个块在一个高速缓存中只缓存一个
 - 所有被高速缓存的块只有一个拷贝
- 细粒度共享
 - 通信延迟决定了存取路径适合的存储结构中的层次
 - **2-10 时钟周期**
 - **Cray Xmp 有共享的寄存器!**
- 可能的正干扰
 - 一个处理器从另一个处理器预取数据
- 缩小总存储量
 - 两个处理器只用一个代码/数据拷贝
- **Can share data within a line without “ping-pong”**
 - **long lines without false sharing**



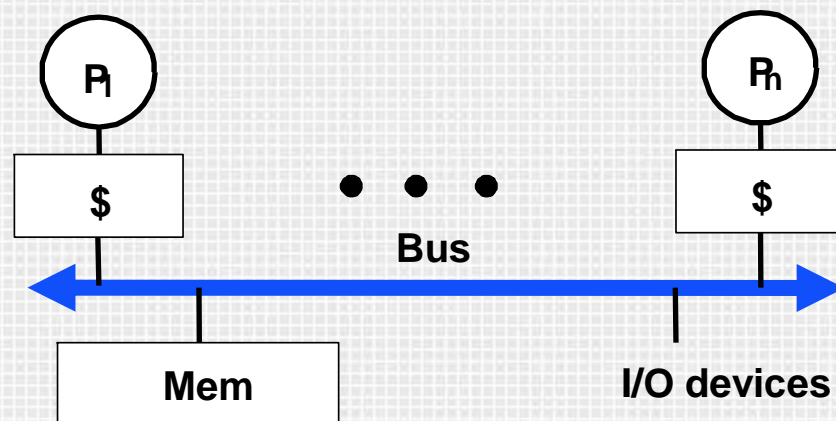


缺点

- 对高速缓存的带宽要求很高
- 增加了存取延迟
 - **X-bar**
 - 更大的**cache**
 - **L1** 命中时间决定处理器周期 !!!
- 潜在的反相关
 - 某个处理器需要另一个处理器的数据
- 今天很多**L2 caches**是共享的



基于总线的对称式共享存储

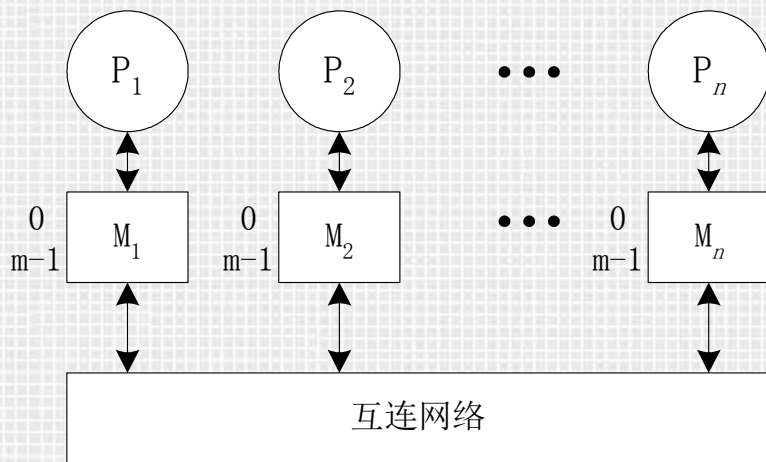


- 占领了服务器市场
 - 建立针对大型系统的平台；实用的机器，直到桌面系统
 - 对并行程序与吞吐量服务器有吸引力
 - 细粒度资源共享
 - 统一的经由**loads/stores**的存取
 - **Cache**中的自动数据移动以及相关复制
 - 有力并代价低的扩展
 - 一般 联合处理器装置存取数据
- 层次存储支持多处理器的扩展是关键

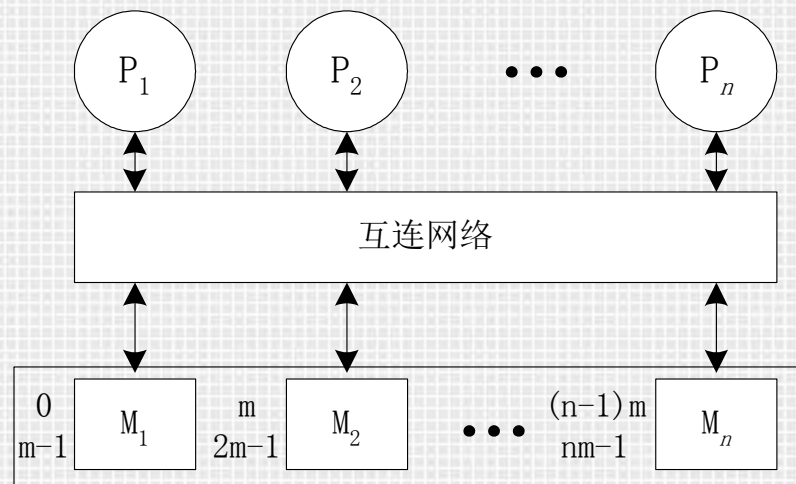


消息传递与共享存储

- 集中共享存储系统中，所有处理器共享主存储器，统一编址，处理器之间的通信通过访问共享变量来实现。编程容易，不可扩放。
- 消息传递系统中，每个处理器都有一个只有它自己才能访问的局部存储器，单独编址，处理器之间的通信必须通过显式的消息传递来进行。扩放性好，编程困难。



(a) 消息传递多计算机

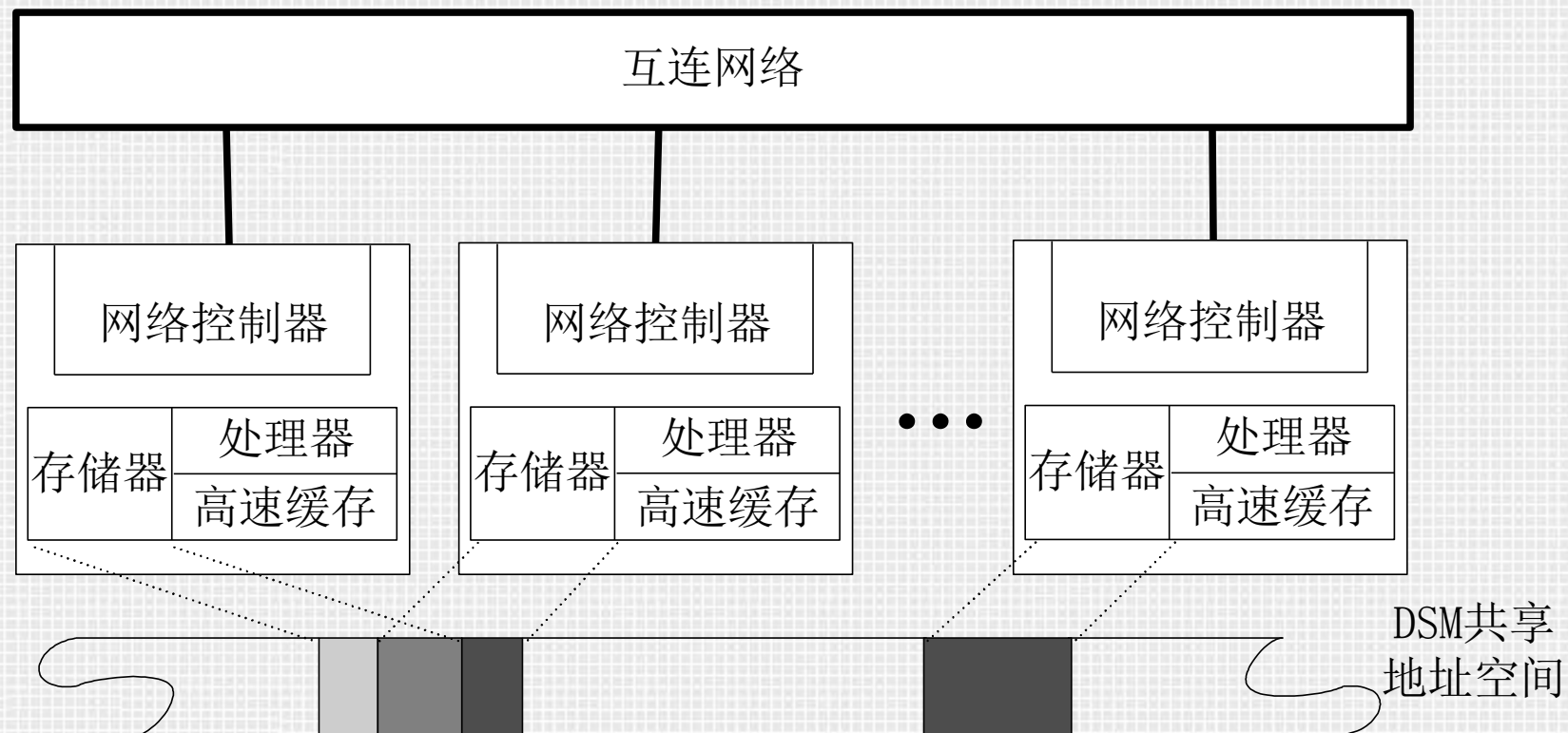


(b) 共享存储多处理机



分布式共享存储系统

- 共享存储器分布于各节点之中，节点之间通过可扩放性好的互连网络相连。
 - 在物理上分布存储的系统上逻辑地实现共享存储模型
 - 对于程序设计者隐藏了远程通信机制，保持了方便性和可移植性。
 - **DSM**系统底层分布式存储具有可扩放性和代价有效性
 - 分布式的存储器和可扩放的互连网络增加了访存带宽，但却导致了不一致的访存结构



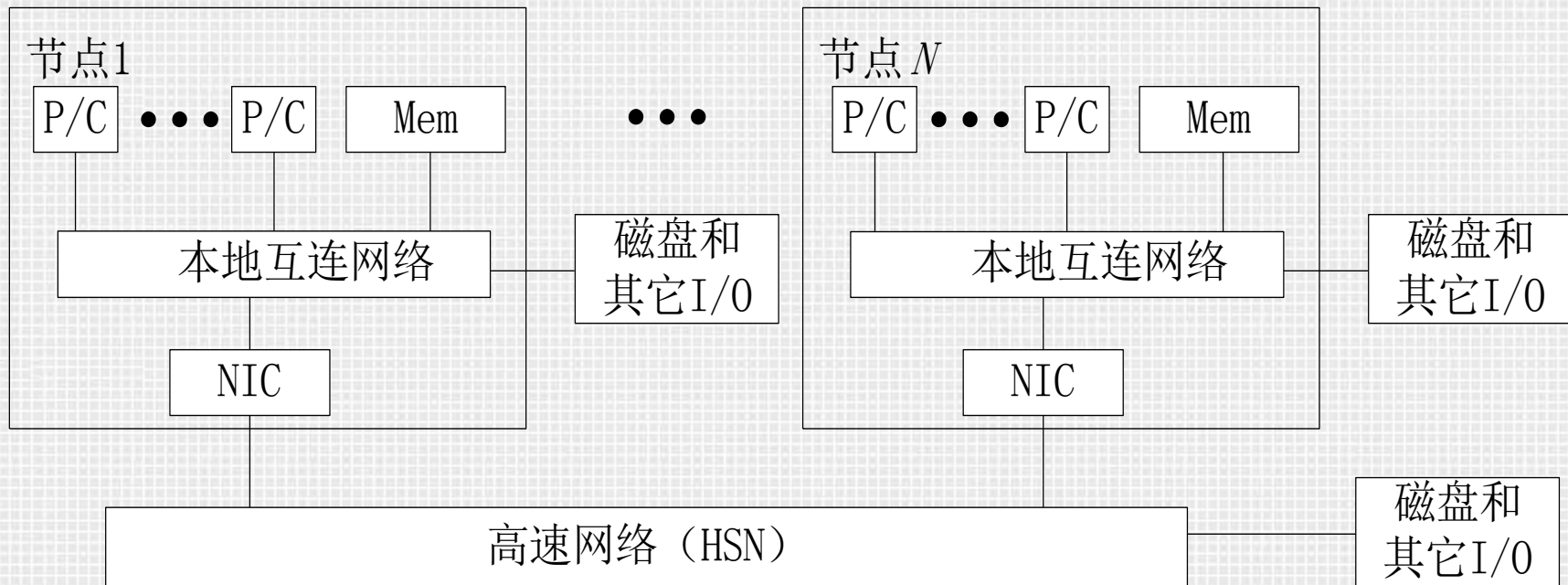


MPP

- 大规模并行处理机**MPP(Massively Parallel Processor)** 通常是指具有下列特点的大规模的计算机系统：
 - 节点中使用商品化微处理器，且每个节点有一个或多个微处理器；
 - 节点内使用物理上分布的存储器；
 - 具有高通信带宽和低延迟的互连网络，节点间紧耦合；
 - 能扩展成具有成百上千个处理器；
 - 一个异步多指令流多数据流**MIMD**机
- **Intel Paragon、IBM SP2、Intel TFLOPS和我国的曙光-1000等都是MPP**
- 两种实现途径
 - **NCC-NUMA**体系结构，**Cray T3E**
 - **NORMA**体系结构，**Intel/Sandia ASCI Option Red**
- 与机群的概念很模糊
 - 关键差别在于节点间的通信
 - 差别缩小



MPP的结构图:





MPP

- 八十年代后期及九十年代中前期迅速发展
 - **Thinking Machine**公司的**CM5**，**Intel**公司的**Paragon**，**IBM**公司的**SP2**，以及**Cray**公司的**T3D**
 - 主要被用于科学计算
- 九十年代后期，随着一些专门生产并行机的公司的倒闭或被兼并，**MPP**系统慢慢从主流的并行处理市场退出
 - 由于消息传递系统相对共享存储系统比较容易实现，它仍成为实现超大规模并行处理的重要手段，不过由于价格和应用领域的原因，基于消息传递的**MPP**系统的研制逐渐成为了政府行为
 - 新涌现的高性能计算系统绝大多数都将是由可缩放的高速互连网络连接的基于商用微处理器的对称多处理机(**SMP**)机群？
 - 体积
 - 功耗
 - 散热



ASCI平台



Option Red



Blue Mountain



ASCI Blue-Pacific

Lawrence Livermore National Laboratory





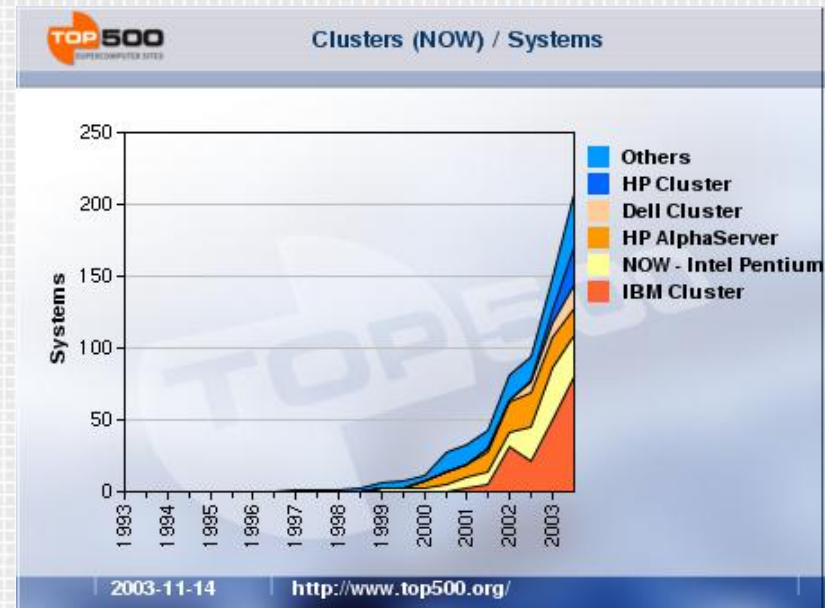
机群

- 机群是一组独立的计算机（节点）的集合体，通常有以下特征：
 - 机群的各节点都是一个完整的系统：工作站，**PC机**或**SMP**机器；
 - 互连网络通常使用商品化网络，如以太网、**FDDI**、**ATM**等；
 - 网络接口与节点的**I/O**总线松耦合相连；
 - 各节点通常有一个本地磁盘；
 - 各节点有自己的完整的操作系统。
 - 各节点除了可以作为一个单一的计算资源供交互式用户使用外，还可以协同工作并表现为一个单一的、集中的计算资源供并行计算任务使用。
- 机群与分布式系统的区别：
 - 机群继承了分布式系统的大部分知识
 - 分布式系统通常是一个计算机的动物园，具有许多不同种类的计算机
 - 机群通常是同构，耦合度较紧密，节点间互为信任关系



2006年TOP500中的机群

- 机群系统在高性能计算机中所占比例迅速增加
 - TOP10中目前有7个机群系统
 - TOP500中目前有208个机群系统（6个月前149个）
 - TOP500中最普通的并行机体系结构
 - 导致了高性能计算机的“平民化”——如排名第3的系统X是由美国弗吉尼亚工学院的一群师生采用商用部件花了4个月时间制造出来。





机群系统的迅速发展原因

- 机群价格便宜并且易于构建；**穷人的解决方案，Gordon Bell奖**
- 作为机群节点的工作站系统的处理性能越来越强大；
- 局域网上新的网络技术和新的通信协议的引入，高带宽低延迟的节点间通信；
- 机群系统比传统的并行计算机更易于融合到已有的网络系统中去；
- 机群上的开发工具更成熟，传统并行计算机上缺乏一个统一的标准；
- 机群的可扩展性良好。



SMP、MPP、机群的比较一览表

| 系统特征 | SMP | MPP | 机群 |
|----------|--------------|-----------------------|---------------|
| 节点数量(N) | $\leq O(10)$ | $O(100)-O(1000)$ | $\leq O(100)$ |
| 节点复杂度 | 中粒度或细粒度 | 细粒度或中粒度 | 中粒度或粗粒度 |
| 节点间通信 | 共享存储器 | 消息传递 或共享变量（有DSM时） | 消息传递 |
| 节点操作系统 | 1 | N(微内核) 和1个主机OS(单一) | N (希望为同构) |
| 支持单一系统映像 | 永远 | 部分 | 希望 |
| 地址空间 | 单一 | 多或单一（有DSM时） | 多个 |
| 作业调度 | 单一运行队列 | 主机上单一运行队列 | 协作多队列 |
| 网络协议 | 非标准 | 非标准 | 标准或非标准 |
| 可用性 | 通常较低 | 低到中 | 高可用或容错 |
| 性能/价格比 | 一般 | 一般 | 高 |
| 互连网络 | 总线/交叉开关 | 定制 | 商用 |

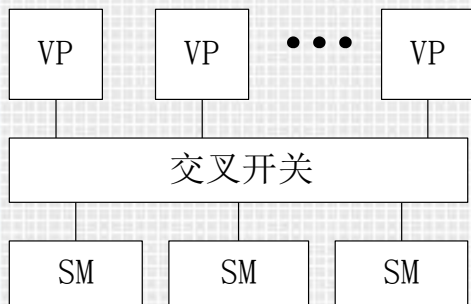


机群分类

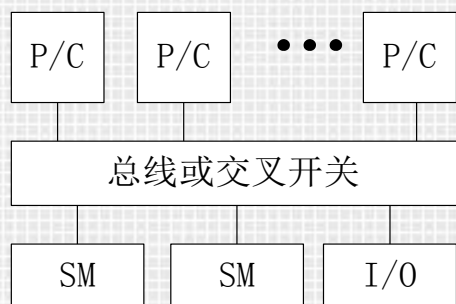
- 根据应用目标：
 - 高性能机群（**HP Cluster**）和高可用性机群（**HA Cluster**）等；
- 根据节点的拥有情况：
 - 专用机群（**Dedicated Cluster**）：所有的资源是共享的，并行应用可以在整个机群上运行。
 - 非专用机群（**Nondedicated Cluster**）：全局应用通过窃取**CPU**时间获得运行，考虑进程迁移和负载平衡等问题。
- 根据节点的配置：
 - 同构机群：各节点有相似的体系并且使用相同的操作系统。
 - 异构机群，节点可以有不同的体系，运行的操作系统也可以不同。
- 根据节点的硬件构成：
 - 分为**PC机群CoPC**（**Cluster of PCs**）或称为**PC堆PoPC**（**Pile of PCs**），工作站机群**COW**（**Cluster of Workstations**）和**SMP机群CLUMPs**（**Cluster of SMPs**）。
- 根据节点的操作系统：
 - **Linux机群**（如**Beowulf**），**Solaris机群**（如**Berkeley NOW**），**NT机群**（如**HPVM**），**AIX机群**（如**IBM SP2**）等。



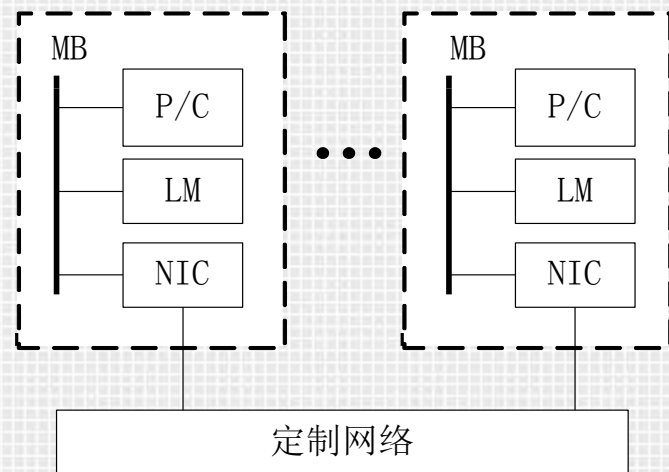
并行计算机结构模型



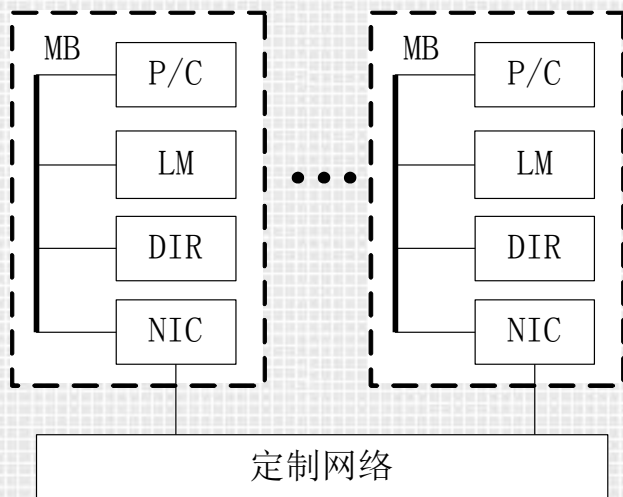
(a) PVP



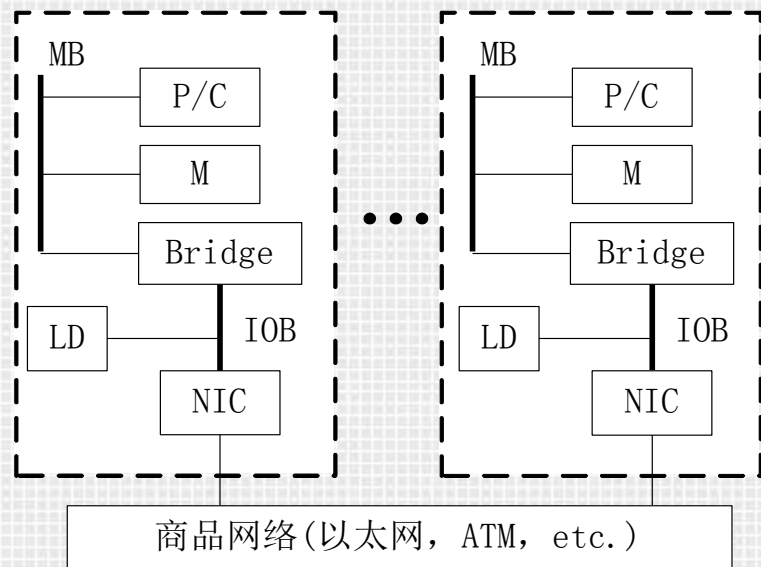
(b) SMP



(c) MPP



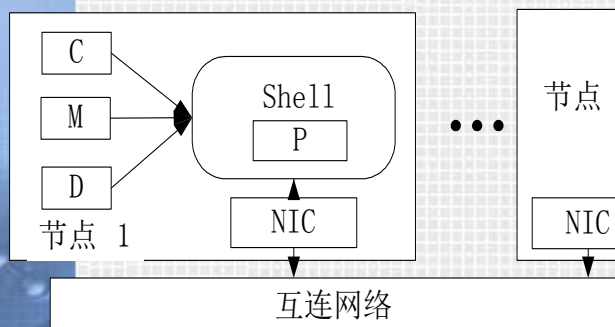
(d) DSM



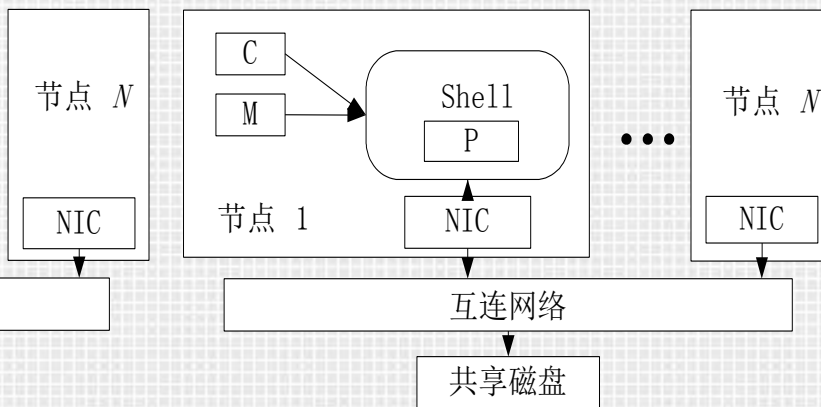
(e) COW

并行计算机体系合一结构

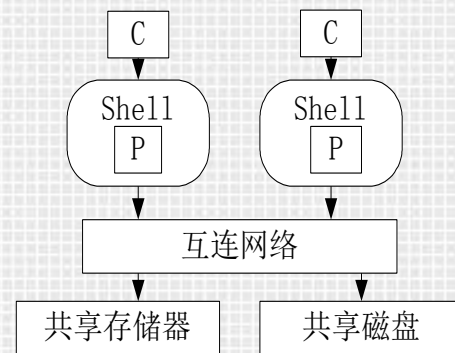
- **SMP、MPP、DSM和COW**并行结构渐趋一致。
 - 大量的节点通过高速网络互连起来
 - 节点遵循**Shell**结构：用专门定制的**Shell**电路将商用微处理器和节点的其它部分（包括板级**Cache**、局存、**NIC**和**DISK**）连接起来。优点是**CPU**升级只需要更换**Shell**。



(a) 无共享



(b) 共享磁盘



(c) 共享存储

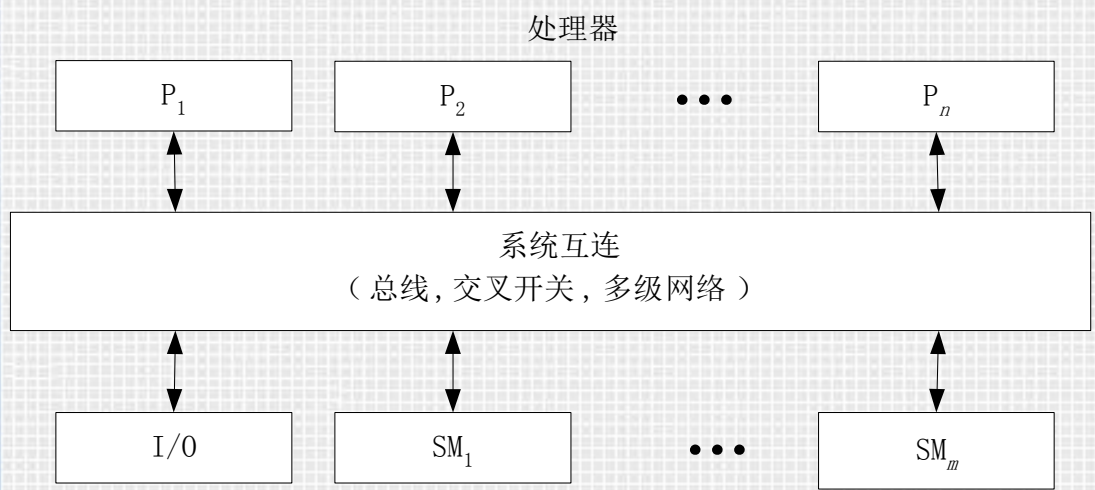


五种结构特性一览表

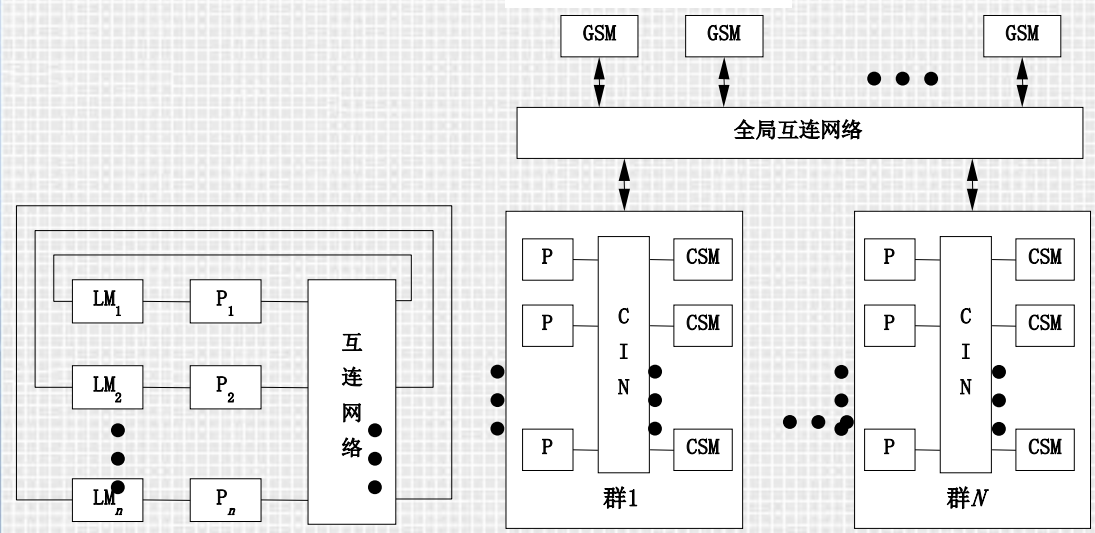
| 属性 | PVP | SMP | MPP | DSM | COW |
|-------|----------------------------------|---|---|--------------------------------|--------------------------------|
| 结构类型 | MIMD | MIMD | MIMD | MIMD | MIMD |
| 处理器类型 | 专用定制 | 商用 | 商用 | 商用 | 商用 |
| 互连网络 | 定制交叉开关 | 总线、交叉开关 | 定制网络 | 定制网络 | 商用网络 (以太ATM) |
| 通信机制 | 共享变量 | 共享变量 | 消息传递 | 共享变量 | 消息传递 |
| 地址空间 | 单地址空间 | 单地址空间 | 多地址空间 | 单地址空间 | 多地址空间 |
| 系统存储器 | 集中共享 | 集中共享 | 分布非共享 | 分布共享 | 分布非共享 |
| 访存模型 | UMA | UMA | NORMA | NUMA | NORMA |
| 代表机器 | Cray C-90, Cray T-90, 银河1号 | IBM R50, SGI Power Challenge, 曙光1号 | Intel Paragon, IBMSP2, 曙 光1000/2000 | Stanford DASH, Cray T 3D | Berkeley NOW, Alpha Farm |



存储器存取模型(UMA NUMA COMA NORMA)



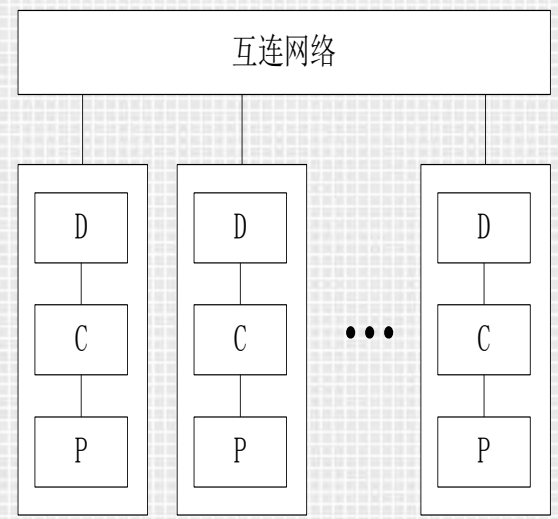
UMA



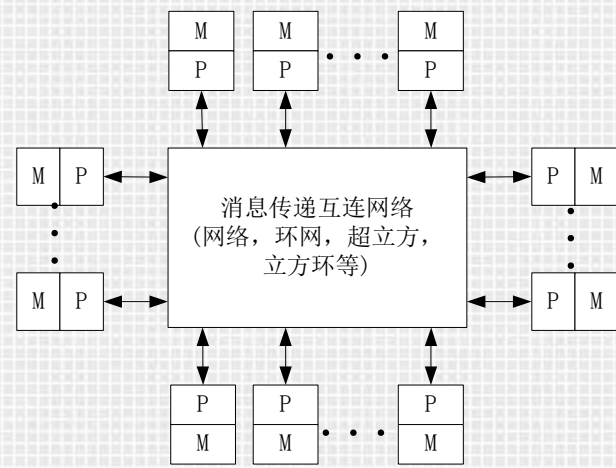
(a) 共享本地存储模型

NUMA

(b) 层次式机群模型



COMA

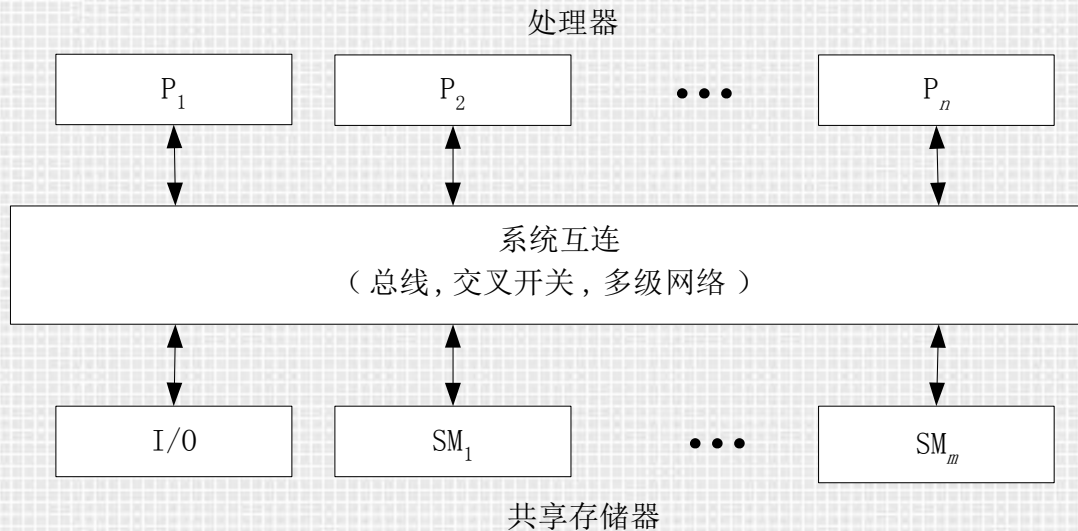


NORMA



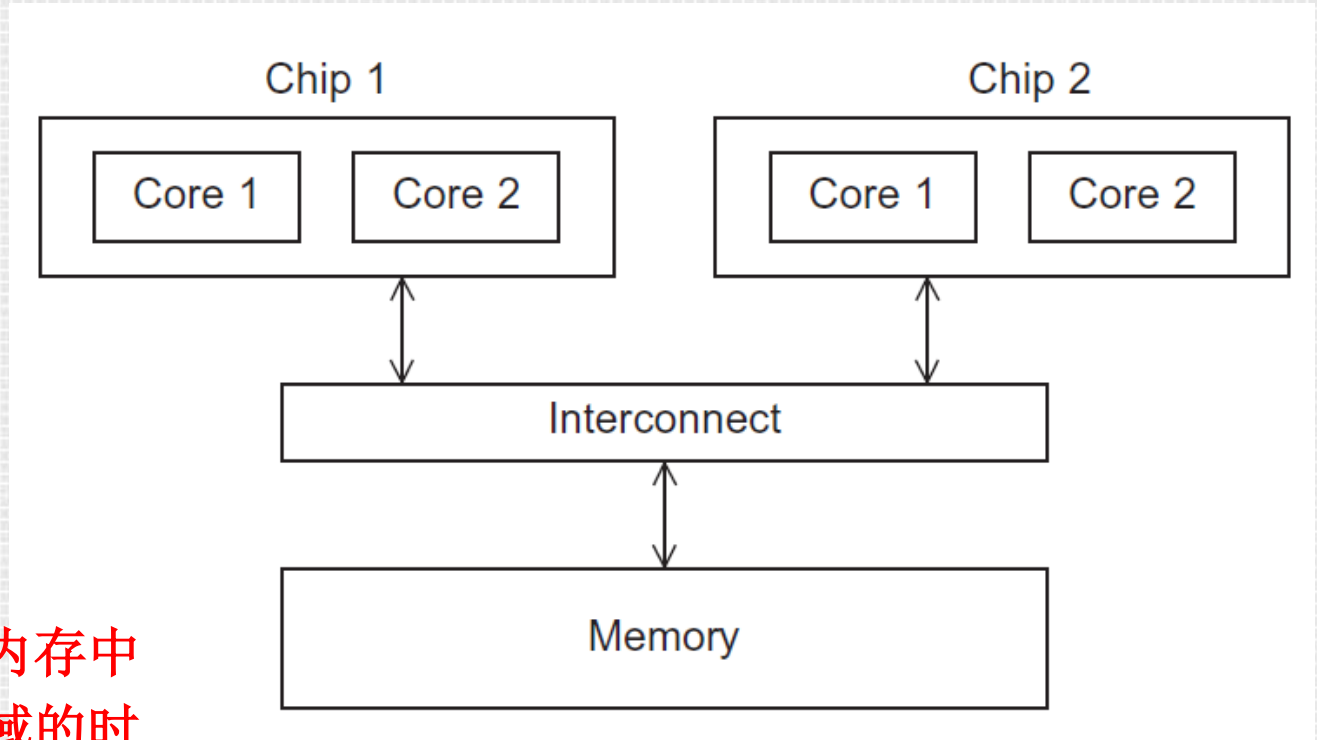
并行计算机访存模型（1）

- **UMA（Uniform Memory Access）模型**是均匀存储访问模型的简称。其特点是：
 - 物理存储器被所有处理器均匀共享；
 - 所有处理器访问任何存储字取相同的时间；
 - 每台处理器可带私有高速缓存；
 - 外围设备也可以一定形式共享。





UMA multicore system



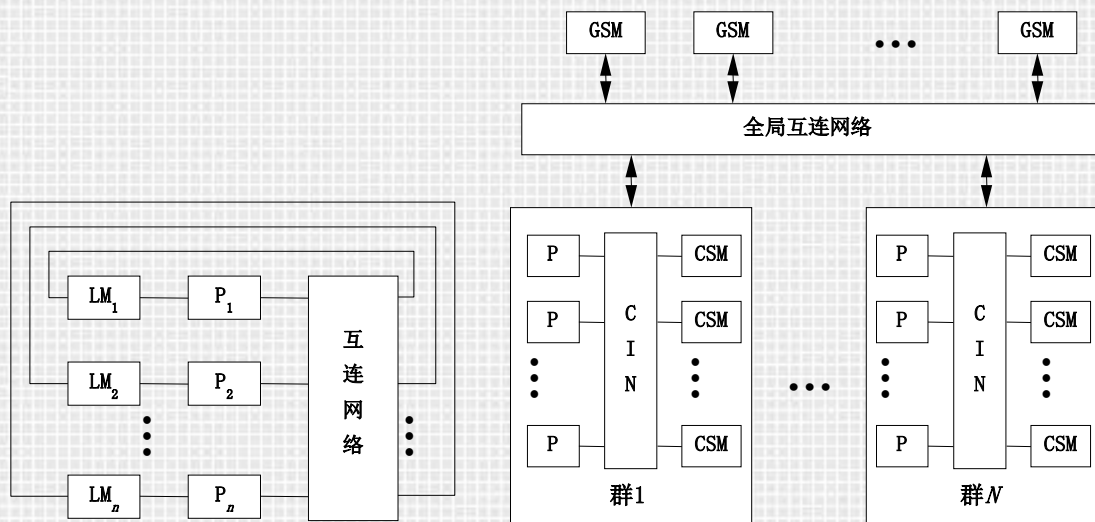
每个核访问内存中
任何一个区域的时间
都相同

一个UMA多核系统



并行计算机访存模型（2）

- NUMA(Nonuniform Memory Access)模型是非均匀存储访问模型的简称。特点是：
 - 被共享的存储器在物理上是分布在所有的处理器中的，其所有本地存储器的集合就组成了全局地址空间；
 - 处理器访问存储器的时间是不一样的；访问本地存储器LM或群内共享存储器CSM较快，而访问外地的存储器或全局共享存储器GSM较慢(此即非均匀存储访问名称的由来)；
 - 每台处理器照例可带私有高速缓存，外设也可以某种形式共享。

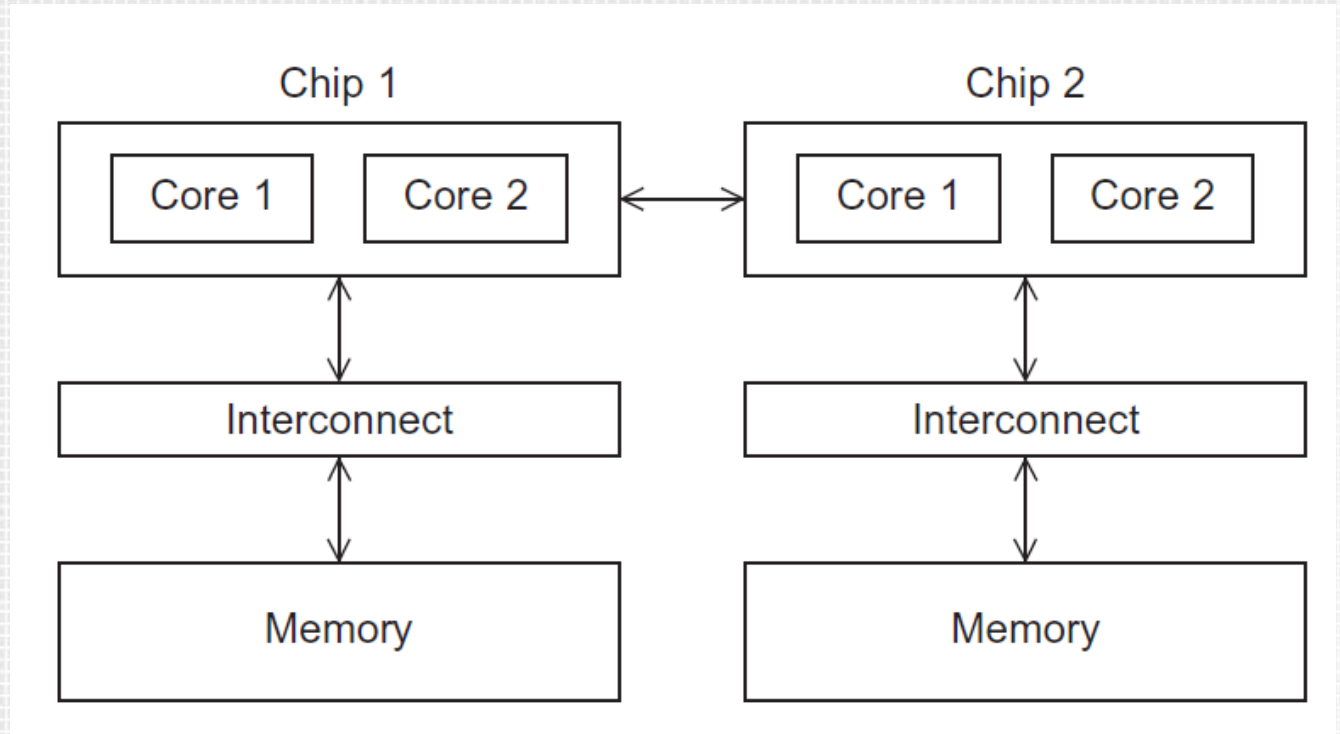


(a) 共享本地存储模型

(b) 层次式机群模型



NUMA multicore system



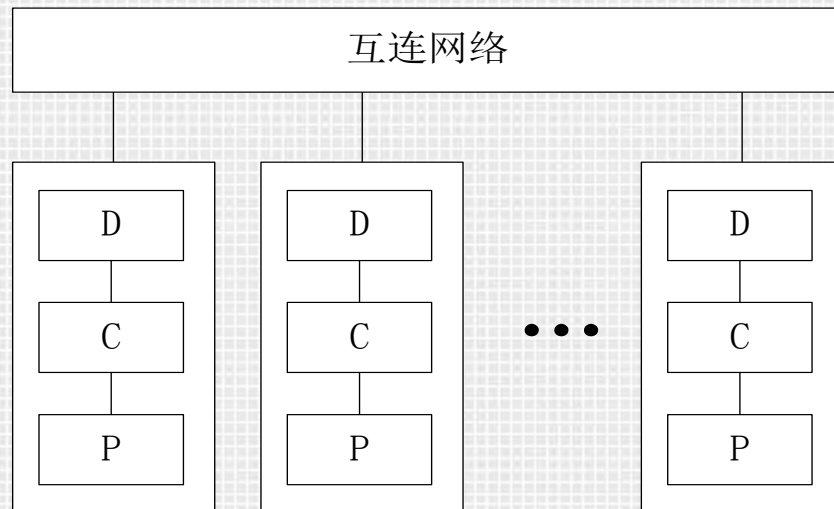
访问与核直接连接的那块内存区域比访问其他内存区域要快很多，因为访问其他内存区域需要通过另一块芯片。

一个NUMA多核系统



并行计算机访存模型（3）

- **COMA(Cache-Only Memory Access)**模型是全高速缓存存储访问的简称。其特点是：
 - 各处理器节点中没有存储层次结构，全部高速缓存组成了全局地址空间；
 - 利用分布的高速缓存目录**D**进行远程高速缓存的访问；
 - **COMA**中的高速缓存容量一般都大于2级高速缓存容量；
 - 使用**COMA**时，数据开始时可任意分配，因为在运行时它最终会被迁移到要用到它们的地方。

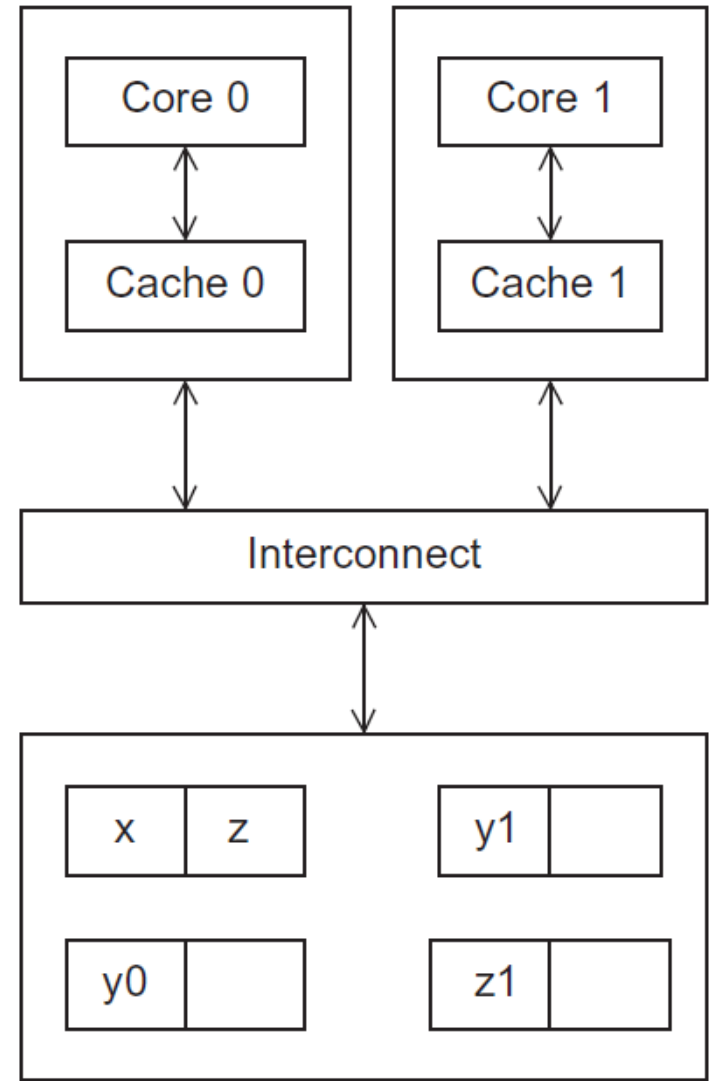




2.3.4 Cache 一致性

- **Cache** 由系统硬件管理，程序员不能直接对它进行操作

有两个核和两个**Cache**的内存系统





Ex.

y0 为Core 0私有

y1 和 **z1** 为Core 1 私有

x = 2; /* 共享变量*/

| Time | Core 0 | Core 1 |
|------|------------------------------|------------------------------|
| 0 | y0 = x; | y1 = 3*x; |
| 1 | x = 7; | Statement(s) not involving x |
| 2 | Statement(s) not involving x | z1 = 4*x; |

最终结果:

y0 = 2

y1 = 6

z1 = ???



- 在多核系统中，各个核的**Cache**存储相同变量的副本，当一个处理器更新**Cache**中该变量的副本时，其他处理器应该知道该变量已经更新，即其他处理器中**Cache**的副本也应该更新，这称为**Cache一致性**问题。



监听Cache一致性协议

- 基于总线
- 当多个核共享总线时，总线上传递的信号都能被连接到总线的所有核‘看’到。
 - 当核0更新它Cache中x的副本时，它也将这个更新信息在总线上广播。
 - 如果核1正在监听总线，那么它会知道x已经更新了，并将自己Cache中的x的副本标记为无效。



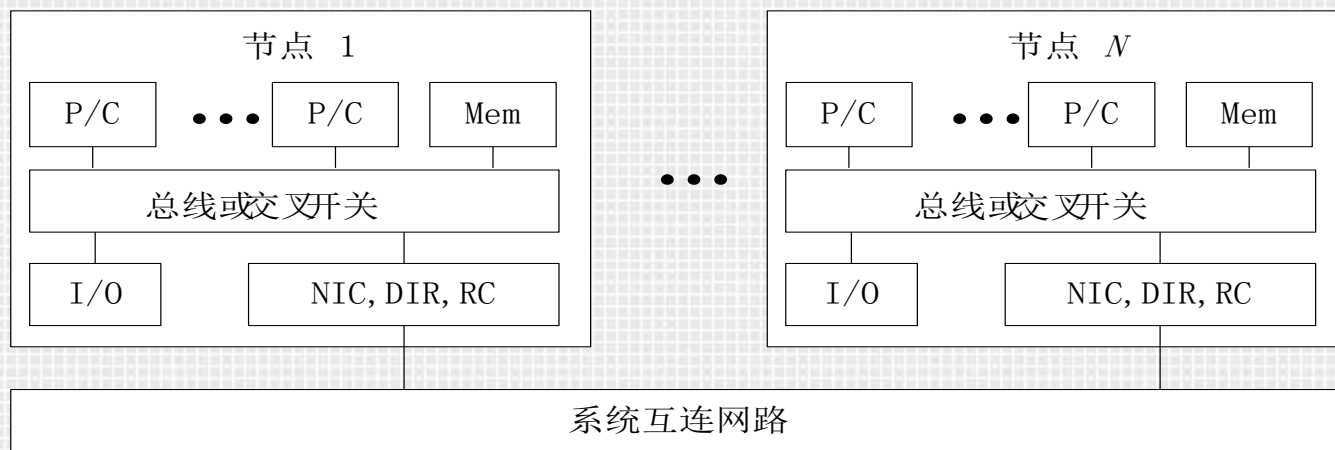
基于目录的Cache一致性协议

- 通过使用一个‘目录’的数据结构来解决监听协议的问题。
- 目录存储了每个内存行的状态。
- 当一个变量需要更新时，就会查询目录，并将所有包含该变量的高速缓存行置为无效。



并行计算机访存模型（4）

- **CC-NUMA（Coherent-Cache Nonuniform Memory Access）**模型是高速缓存一致性非均匀存储访问模型的简称。其特点是：
 - 大多数使用基于目录的高速缓存一致性协议；
 - 保留SMP结构易于编程的优点，也改善常规SMP的可扩展性；
 - **CC-NUMA**实际上是一个分布共享存储的**DSM**多处理机系统；
 - 它最显著的优点是程序员无需明确地在节点上分配数据，系统的硬件和软件开始时自动在各节点分配数据，在运行期间，高速缓存一致性硬件会自动地将数据迁移至要用到它的地方。





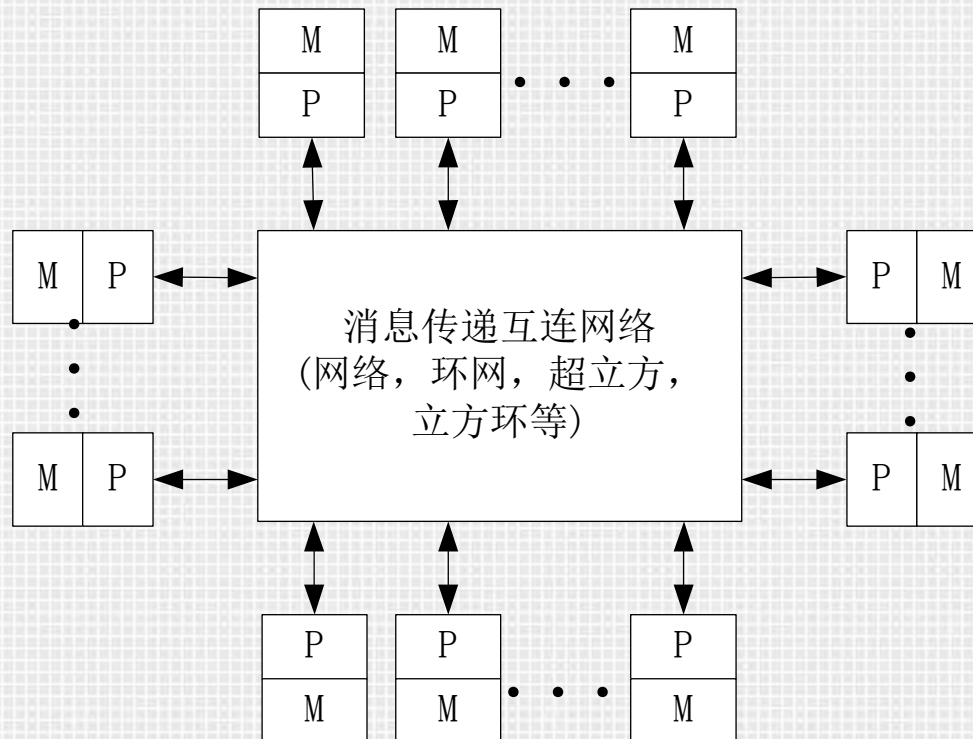
NCC-NUMA（硬件不支持高速缓存一致性）

- 为了避免一致性问题，共享数据被标识为不可高速缓存的，只有私有数据才能被高速缓存
- 好处在于仅需要很少的硬件支持就足够
- 缺点在于：
 - ①支持透明的软件高速缓存一致性的编译机制非常有限，，基于编译支持的软件高速缓存一致性是不太现实的。
 - ②如果没有高速缓存一致性，那么在与访问远地单字所需的同等开销下系统将失去获取并使用一个高速缓存行中多个字的优点。当每次访问远地主存只能获得一个单字时，共享存储所具有的空间局部性的优点就荡然无存了。
 - ③如果可以同时处理多个字（如一个高速缓存行）时，则诸如预取等延迟容忍技术效果才能更好。



并行计算机访存模型（5）

- **NORMA (No-Remote Memory Access)** 模型是非远程存储访问模型的简称。**NORMA**的特点是：
 - 所有存储器是私有的；
 - 绝大多数**NUMA**都不支持远程存储器的访问；
 - 在**DSM**中，**NORMA**就消失了。



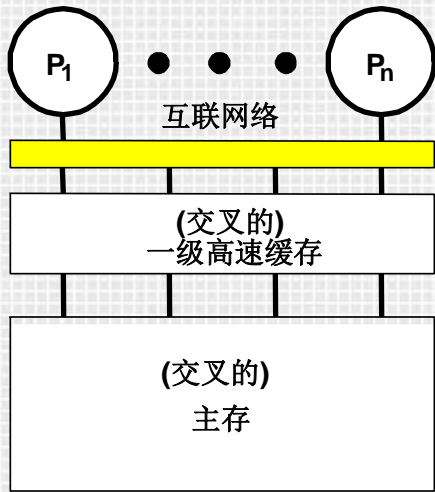


共享虚拟存储SVM结构

- **SVM(Shared Virtual Memory)系统**，又称为软件**DSM**系统，
 - **SVM**系统在基于消息传递的**MPP**或机群系统中，用软件把分布于各节点的多个独立编址的存储器组织成一个统一编址的共享存储空间。
 - 优点是在消息传递的系统上实现共享存储的编程界面，但主要问题是难以获得满意的性能
 - 与硬件共享存储系统相比，**SVM**系统中较大的通信和共享粒度(通常是存储页)会导致假共享及额外的通信；
 - 在基于机群的**SVM**系统中，通信开销很大。基于**SVM**系统的并行程序通信量通常比基于消息传递的并行程序的通信量大。
- **SVM系统的实现**
 - 在操作系统上改进，如**Ivy**、**Mermaid**、**Mirage**和**Clouds**等；
 - 由运行系统来支撑，如**CMU Midway**、**Rice Munin**、**Rice TreadMarks**、**Utah Quarks**、**DIKU CarLOS**、**Maryland CVM**和**JIAJIA**等；
 - 从语言级来实现，如**MIT CRL**、**Linda**和**Orca**等。
 - 混合实现的分布式共享存储系统，其基本思想是结合软硬件实现的分布式共享存储系统的优点。

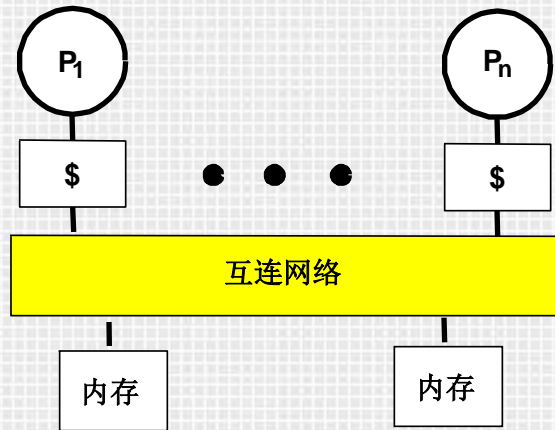


存储系统的自然扩展

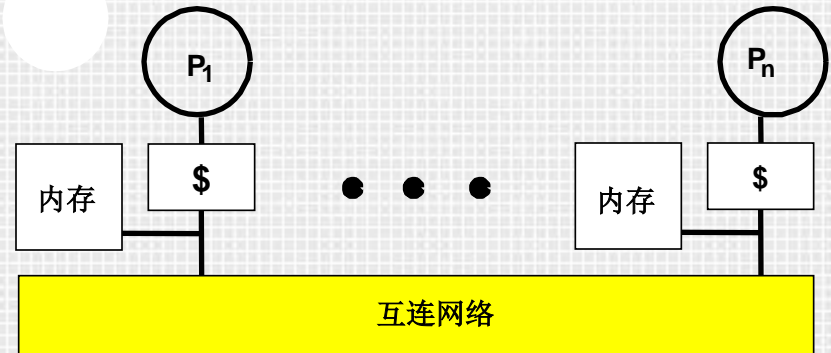


共享高速缓存

Scale →

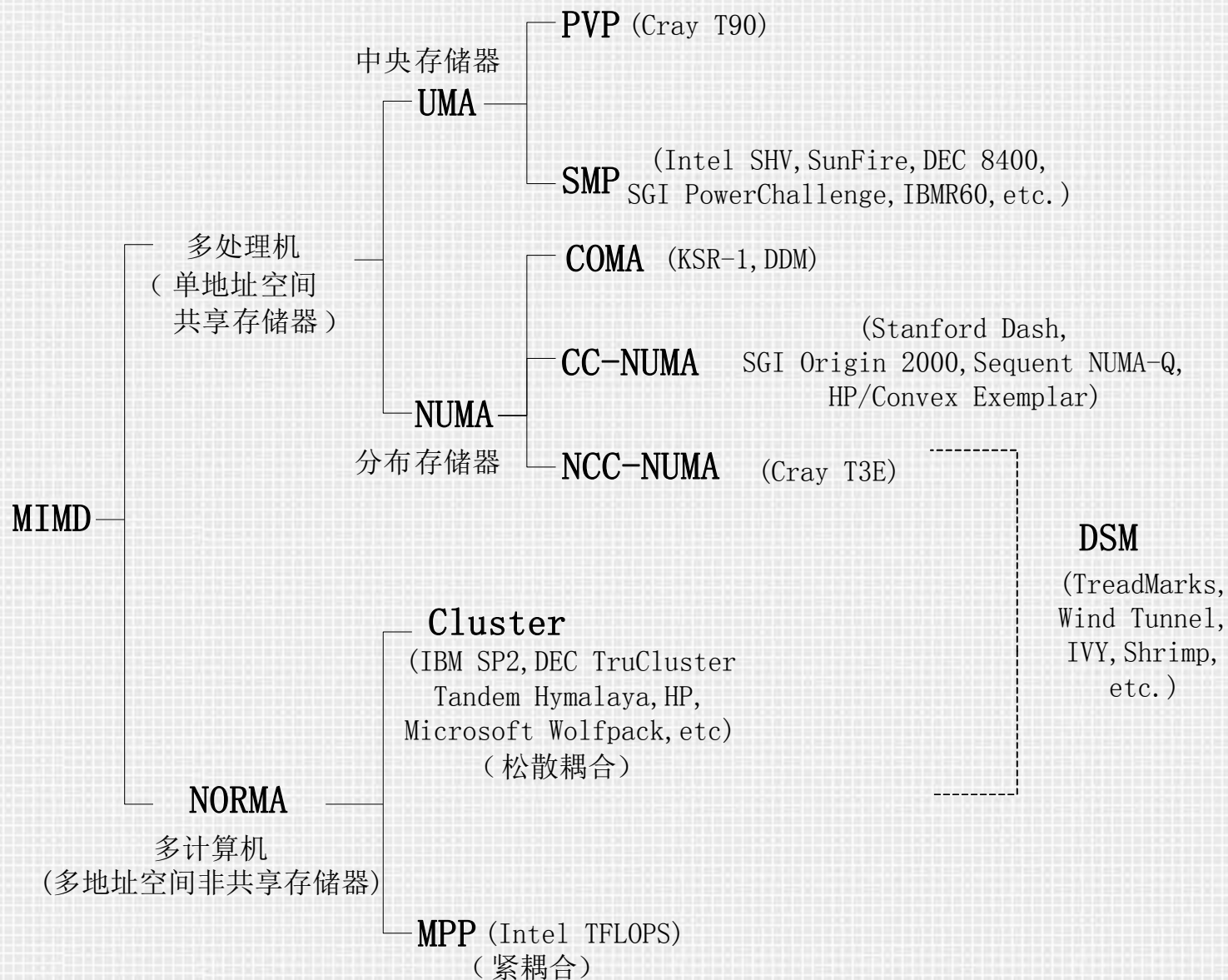


集中存储
UMA



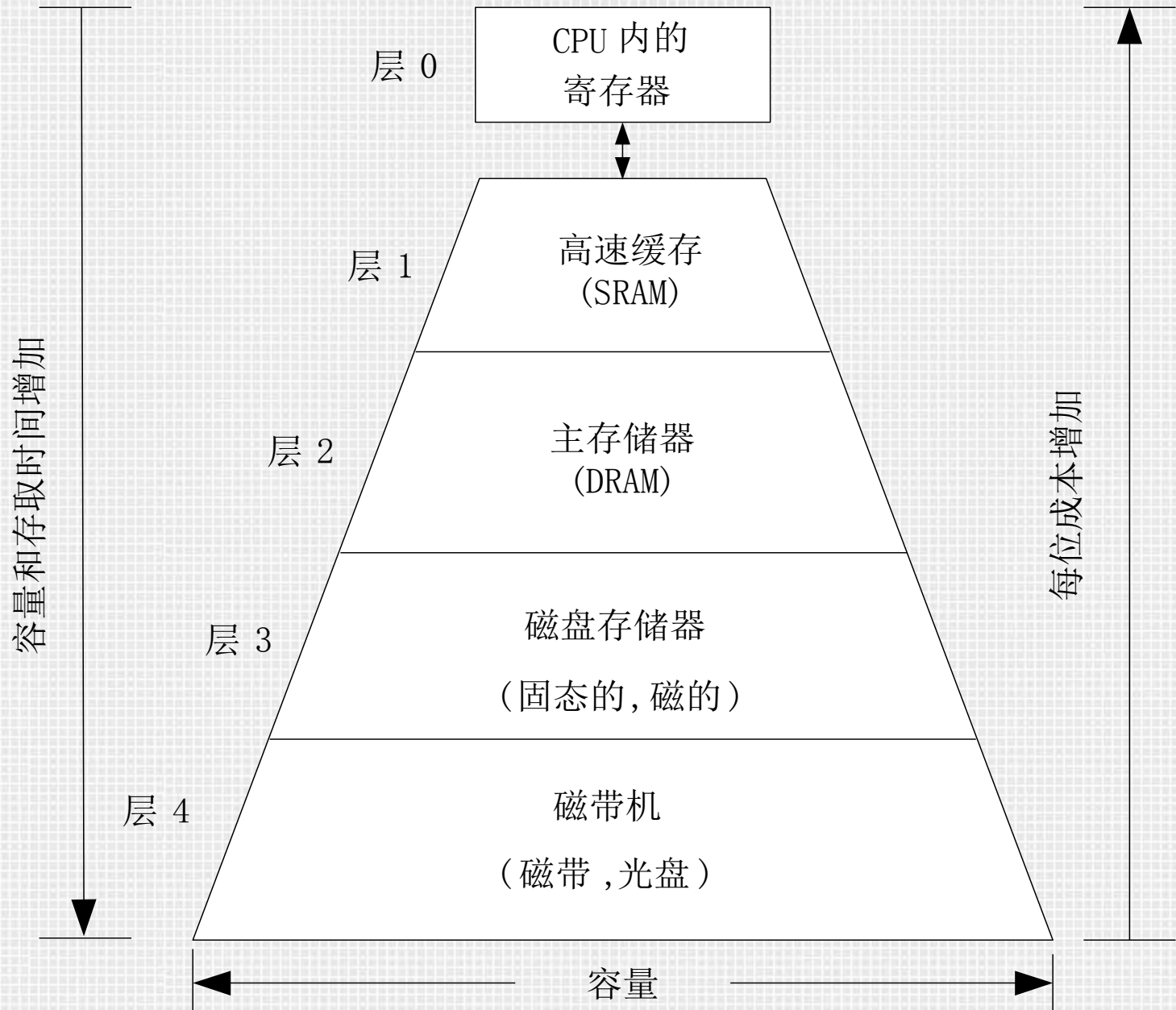
分布式内存 (NUMA)

构筑并行机系统的不同存储结构

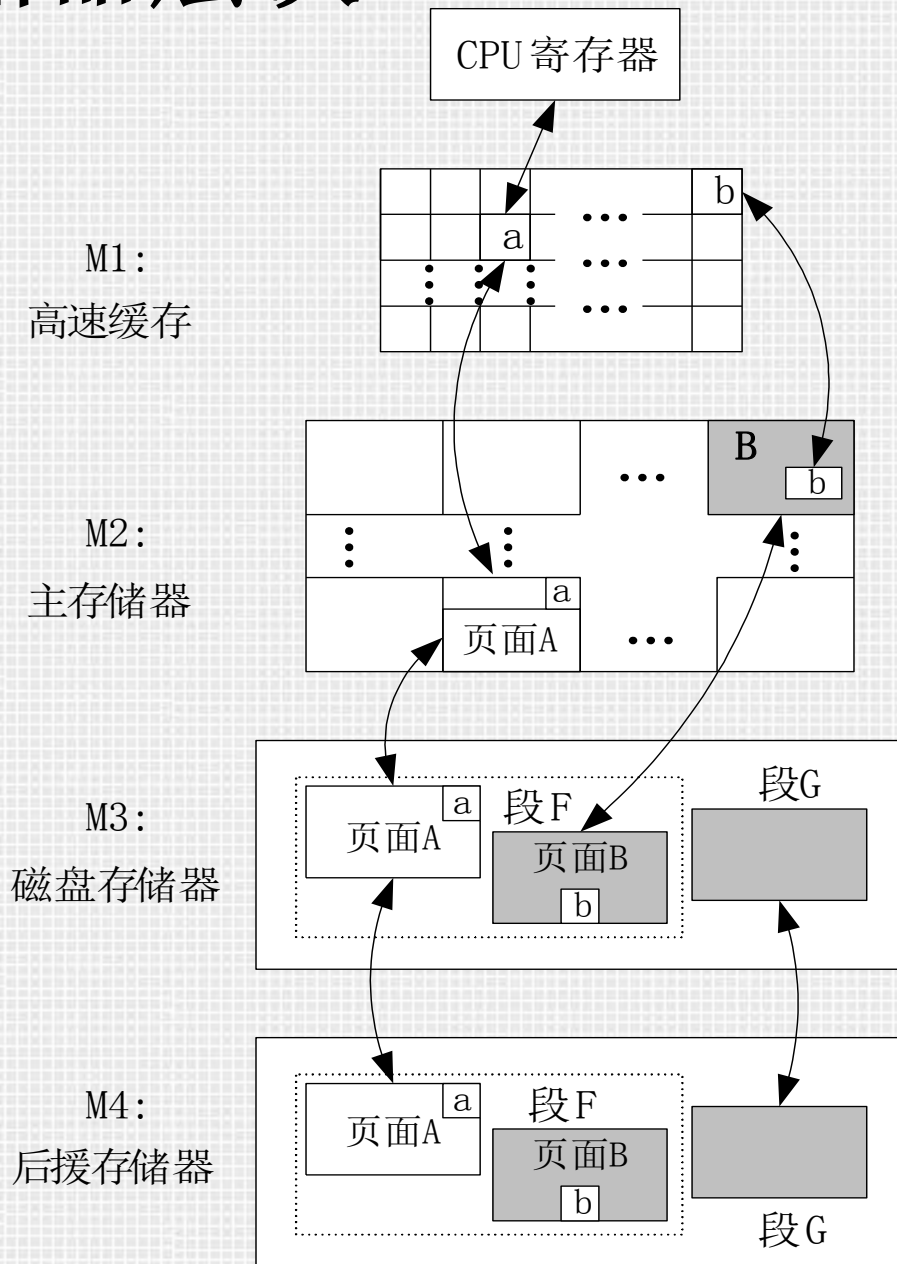




存储器层次



存储器层次



1. 从32字节高速缓存块（如块a）按字（4个字节）存取。

2. 从32块或 1KB存储页面按块（32字节）存取，如从页面B按块存取。

3. 从很多页面组成的文件按页（1KB）存取，如从段F中的页面A和页面B存取。

4. 用不同页面数目的段传送。



2.4 并行软件



- 并行硬件提供了并行软件的基础。
- 通常
 - 在共享内存系统中：
 - 启动一个单独的进程，然后派生多个线程
 - 线程执行任务
 - 在分布式内存系统中：
 - 启动多个进程
 - 进程执行任务

单程序多数据流程序SPMD

- **SPMD**程序不是在每个核上运行不同的程序
- 在执行时，通过使用条件转移语句，表现得像是在不同处理器上执行不同的程序。
- 不仅仅可以数据并行，也可以任务并行

```
if (I'm thread process i)
    do this;
else
    do that;
```





编写并行程序

1. 将任务在进程/线程之间分配
 - (a) 使得每个进程/线程获得大致相等的工作量
 - (b) 使得需要的通信量最小
2. 安排进程/线程之间的同步.
3. 安排进程/线程之间的通信.

```
double x[n], y[n];  
...  
for (i = 0; i < n; i++)  
    x[i] += y[i];
```




2.4.3 共享内存Shared Memory

- 动态线程**Dynamic threads**

- 主线程通常等待工作请求。当请求到达时，派生出一个工作线程来执行该请求；当线程工作完成时，就会终止执行再合并到主线程中去。
- 充分利用了系统的资源
- 但是线程的创建和终止消耗了时间

- 静态线程**Static threads**

- 主线程在完成必要的设置之后，派生出所有的线程，并且在工作结束前所有的线程都在运行。
- 更好的执行，但是浪费系统资源。

非确定性

...

```
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x );
```

...

Thread 1 > my_val = 19

Thread 0 > my_val = 7


Thread 0 > my_val = 7

Thread 1 > my_val = 19



```
my_val = Compute_val ( my_rank ) ;  
x += my_val ;
```

| Time | Core 0 | Core 1 |
|------|-------------------------------|--------------------------------|
| 0 | Finish assignment to my_val | In call to Compute_val |
| 1 | Load x = 0 into register | Finish assignment to my_val |
| 2 | Load my_val = 7 into register | Load x = 0 into register |
| 3 | Add my_val = 7 to x | Load my_val = 19 into register |
| 4 | Store x = 7 | Add my_val to x |
| 5 | Start other work | Store x = 19 |

- 
- 竞争条件
 - 临界区
 - 互斥
 - 互斥锁 (mutex, or simply lock)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```




忙等待-信号量

```
my_val = Compute_val ( my_rank ) ;
```

```
if ( my_rank == 1)
```

```
    while ( ! ok_for_1 ) ; /* Busy-wait loop */
```

```
x += my_val ; /* Critical section */
```

```
if ( my_rank == 0)
```

```
    ok_for_1 = true ; /* Let thread 1 update x */
```



2.4.4 分布式内存

消息传递API

```
char message [ 1 0 0 ] ;
```

```
...
```

```
my_rank = Get_rank ( ) ;
```

```
i f ( my_rank == 1) {
```

```
    sprintf ( message , "Greetings from process 1" ) ;
```

```
    Send ( message , MSG_CHAR , 100 , 0 ) ;
```

```
} e l s e i f ( my_rank == 0) {
```

```
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
```

```
    printf ( "Process 0 > Received: %s\n" , message ) ;
```

```
}
```



划分全局地址空间语言PGAS

```
shared i n t n = ... ;  
shared double x [ n ] , y [ n ] ;  
private int i , my_first_element , my_last_element ;  
my_first_element = ... ;  
my_last_element = ... ;  
/* Initialize x and y */  
...  
for ( i = my_first_element ; i <= my_last_element ; i++)  
    x [ i ] += y [ i ] ;
```




2.5 输入和输出

- 输入和输出的问题非常复杂，不详细赘述。
- **stdin**---标准输入文件
- **stdout**---标准输出文件
- **stderr**---标准出错文件



规则

- 在分布式内存程序中，只有进程**0**能够访问 ***stdin***。在共享内存程序中，只有主线程或者线程**0**访问 ***stdin***。
- 两种系统中，所有进程/线程都可以访问 ***stdout***和***stderr***。



- 因为输出到 **stdout** 的非确定性顺序，只有一个进程/线程结果输出到 **stdout**。
- 调试程序时，允许多个进程/线程写 **stdout**。并且输出的记过应该包括进程/线程的序号或者进程标识符。
- 只有一个进程/线程会尝试访问一个除 **stdin**, **stdout** 或者 **stderr** 外的文件。



2.6 性能



2.6.1 加速比和效率



- 核的数量 = p
- 串行运行时间 = $T_{\text{串行}}$
- 并行运行时间 = $T_{\text{并行}}$

线性加速比

$$T_{\text{并行}} = T_{\text{串行}} / p$$



并行程序的加速比

$$S = \frac{T_{\text{串行}}}{T_{\text{并行}}}$$



并行程序的效率

$$E = \frac{S}{p} = \frac{\left[\frac{T_{\text{串行}}}{T_{\text{并行}}} \right]}{p} = \frac{T_{\text{串行}}}{p \cdot T_{\text{并行}}}$$



Ex. 一个并行程序的加速比和效率

| p | 1 | 2 | 4 | 8 | 16 |
|-----------|-----|------|------|------|------|
| S | 1.0 | 1.9 | 3.6 | 6.5 | 10.8 |
| $E = S/p$ | 1.0 | 0.95 | 0.90 | 0.81 | 0.68 |

E随着**p**的增大越来越小

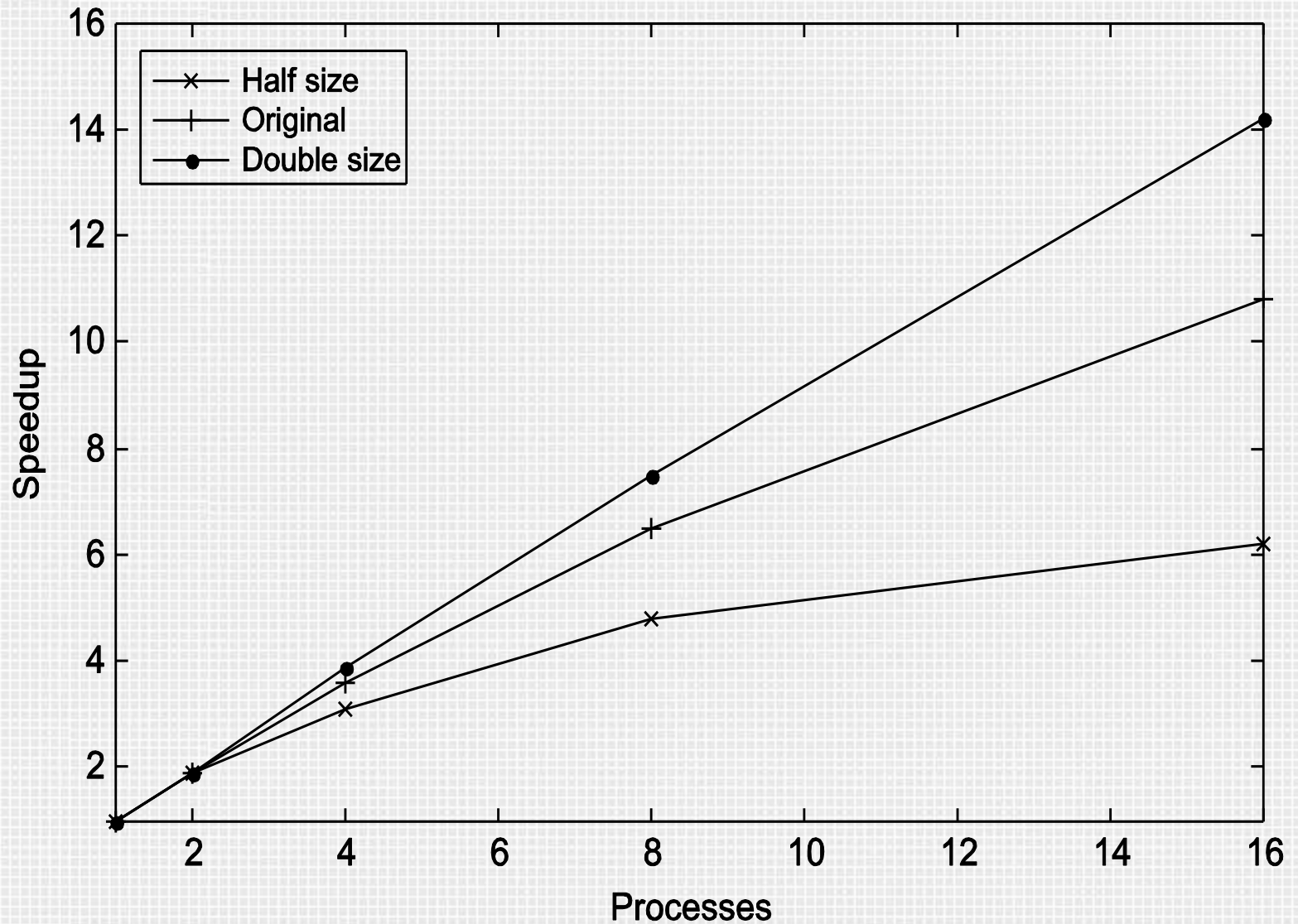


不同问题规模的并程序序的加速 比核效率

| | p | 1 | 2 | 4 | 8 | 16 |
|----------|-----|-----|------|------|------|------|
| Half | S | 1.0 | 1.9 | 3.1 | 4.8 | 6.2 |
| | E | 1.0 | 0.95 | 0.78 | 0.60 | 0.39 |
| Original | S | 1.0 | 1.9 | 3.6 | 6.5 | 10.8 |
| | E | 1.0 | 0.95 | 0.90 | 0.81 | 0.68 |
| Double | S | 1.0 | 1.9 | 3.9 | 7.5 | 14.2 |
| | E | 1.0 | 0.95 | 0.98 | 0.94 | 0.89 |

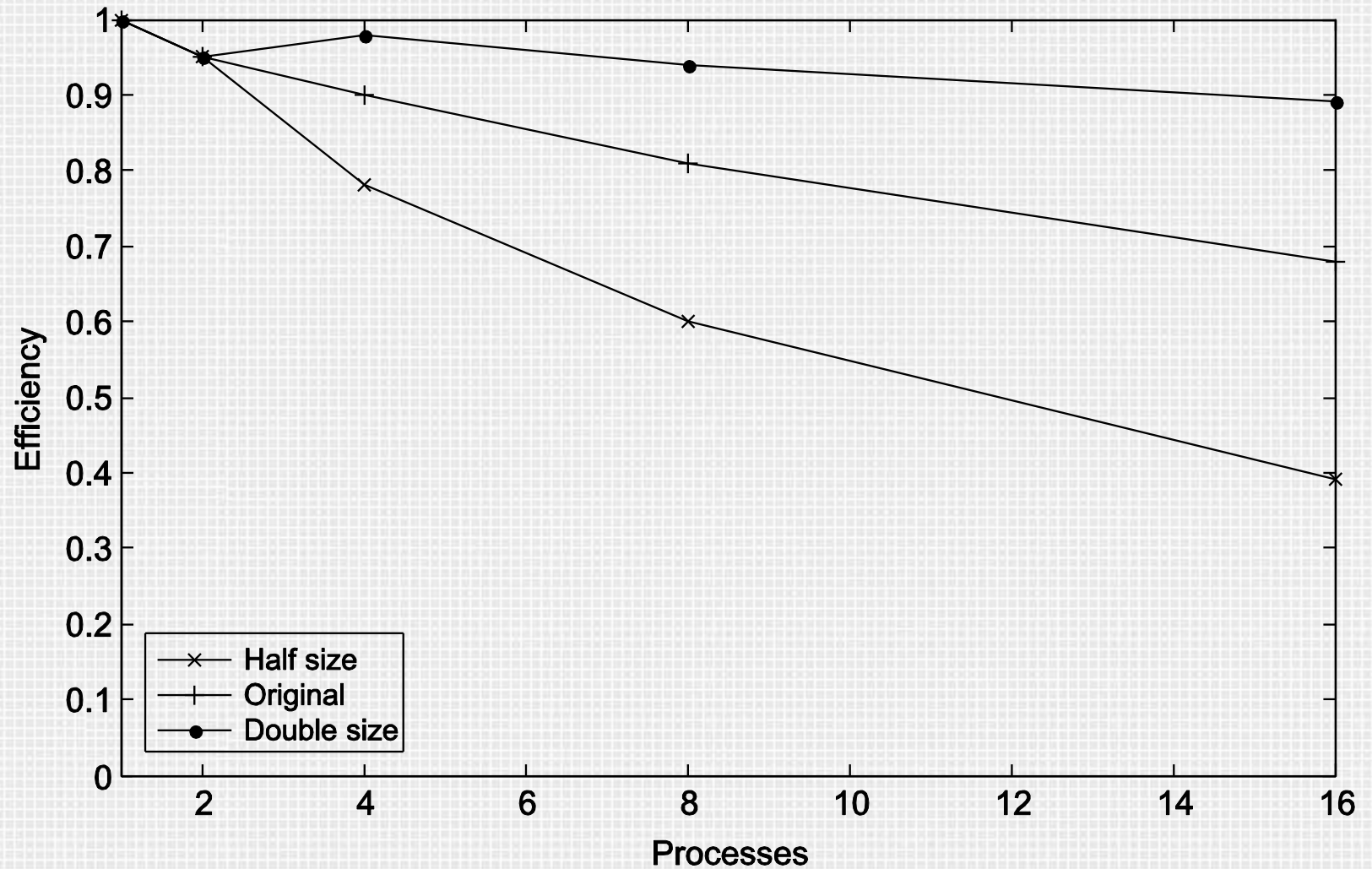


加速比Speedup





效率Efficiency





并行开销

$$T_{\text{并行}} = T_{\text{串行}} / p + T_{\text{开销}}$$



2.6.2 阿姆达尔定律


- 大致上，除非一个串行程序的执行几乎全部都并行化，否则，不论有多少可以利用的核，通过并行化所产生的加速比都会受限。






示例

- 90%的部分并行化.
- 程序可并行化部分的加速比为p.
- $T_{\text{串行}} = 20 \text{ seconds}$
- 程序可并行化部分的运行时间:
$$0.9 \times T_{\text{串行}} / p = 18 / p$$
- 不可并行化部分的运行时间:
$$0.1 \times T_{\text{串行}} = 2$$

- 
- 程序并行版本的全部运行时间为:

$$T_{\text{并行}} = 0.9 \times T_{\text{串行}} / p + 0.1 \times T_{\text{串行}} = 18 / p + 2$$

■ 加速比

$$S = \frac{T_{\text{串行}}}{0.9 \times T_{\text{串行}} / p + 0.1 \times T_{\text{串行}}} = \frac{20}{18 / p + 2}$$




可扩展性

- 如果一个技术可以处理规模不断增加的问题，那么它就是**可扩展的**。
- 如果在增加进程/线程的个数时，可以维持固定的效率，却不增加问题的规模，那么程序称为**强可扩展的**。
- 如果在增加进程/线程个数的同时，只有以相同倍率增加问题的规模才能使效率值保持不变，那么程序称为**弱可扩展的**。



计时Taking Timings

- 什么样的时间？
 - 程序从开始到结束的时间
 - 感兴趣的一部分所花费的时间
 - **CPU**时间
 - ‘墙上时钟’ 时间—代码从开始执行到执行结束的总耗费时间





**theoretical
function**

```
double start, finish;  
...  
start = Get_current_time();  
/* Code that we want to time */  
...  
finish = Get_current_time();  
printf("The elapsed time = %e seconds\n", finish-start);
```

MPI_Wtime

omp_get_wtime



2.7 并行程序设计



Foster并行化方法

1. **划分Partitioning**: 将要执行的指令和数据按照计算部分拆分成多个小任务，关键在于识别出可以并行执行的任务。
2. **通信**: 确定上一步所识别出来的任务之间需要执行哪些通信。





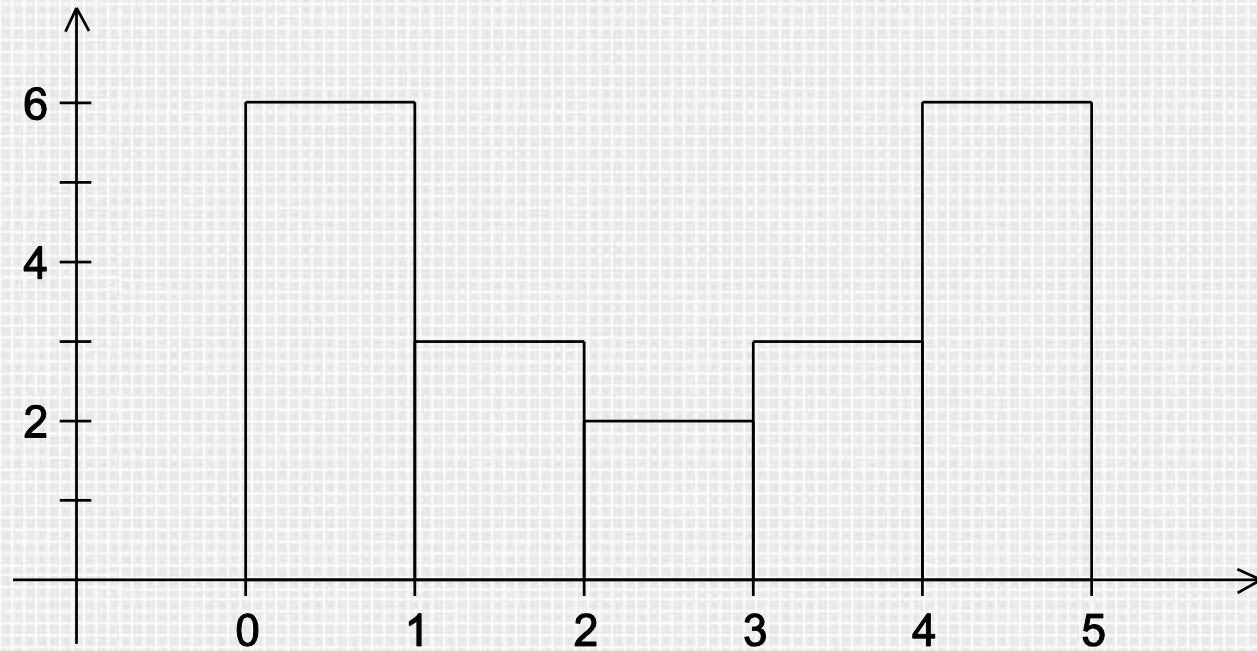
Foster并行化方法

3. **凝聚或者聚合**:将第一步和第二步所确定的任务与通信结合成更大的任务。
4. **分配**: 将上一步聚合的任务分配到进程/线程中。



示例 – 直方图histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9





Serial program - input

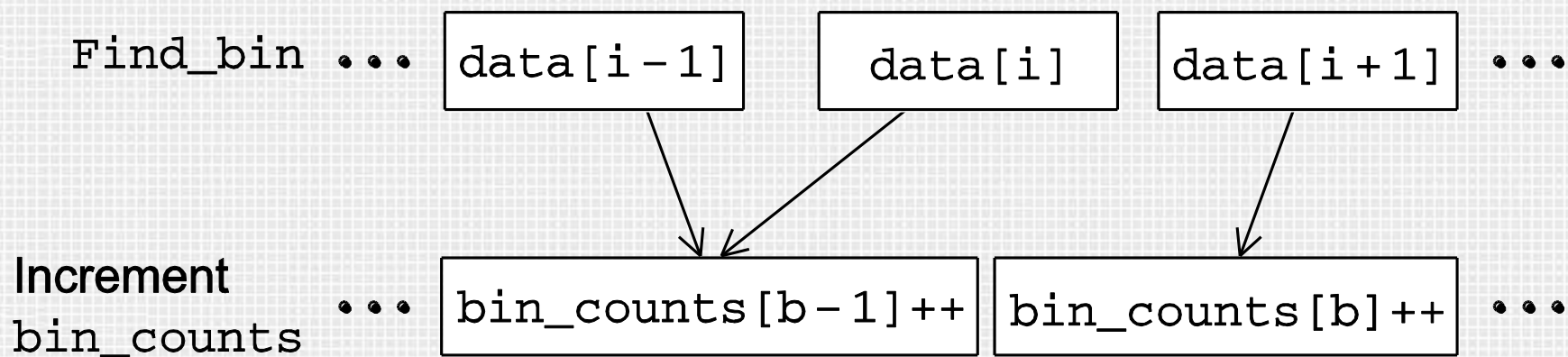
1. 数据的个数: **data_count**
2. 一个大小为**data_count**的浮点数数组:
data
3. 包含最小值的桶中的最小值: **min_meas**
4. 包含最大值的桶中的最大值: **max_meas**
5. 桶的个数: **bin_count**



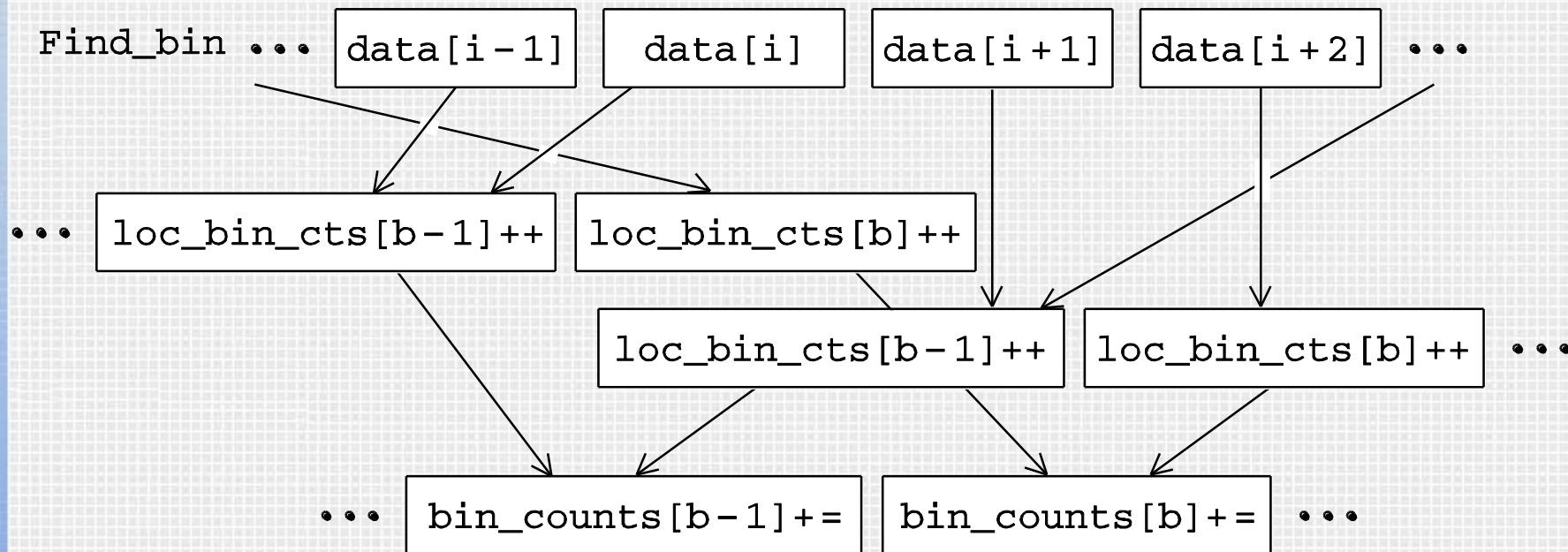
Serial program - output

1. **bin_maxes** : 一个大小为**bin_count**的浮点数数组，存储的是每个桶的上界
2. **bin_counts** : 一个大小为**bin_count**的整数数组，存储的是落在每个桶里的数据的个数。

Foster方法最开始的两个阶段

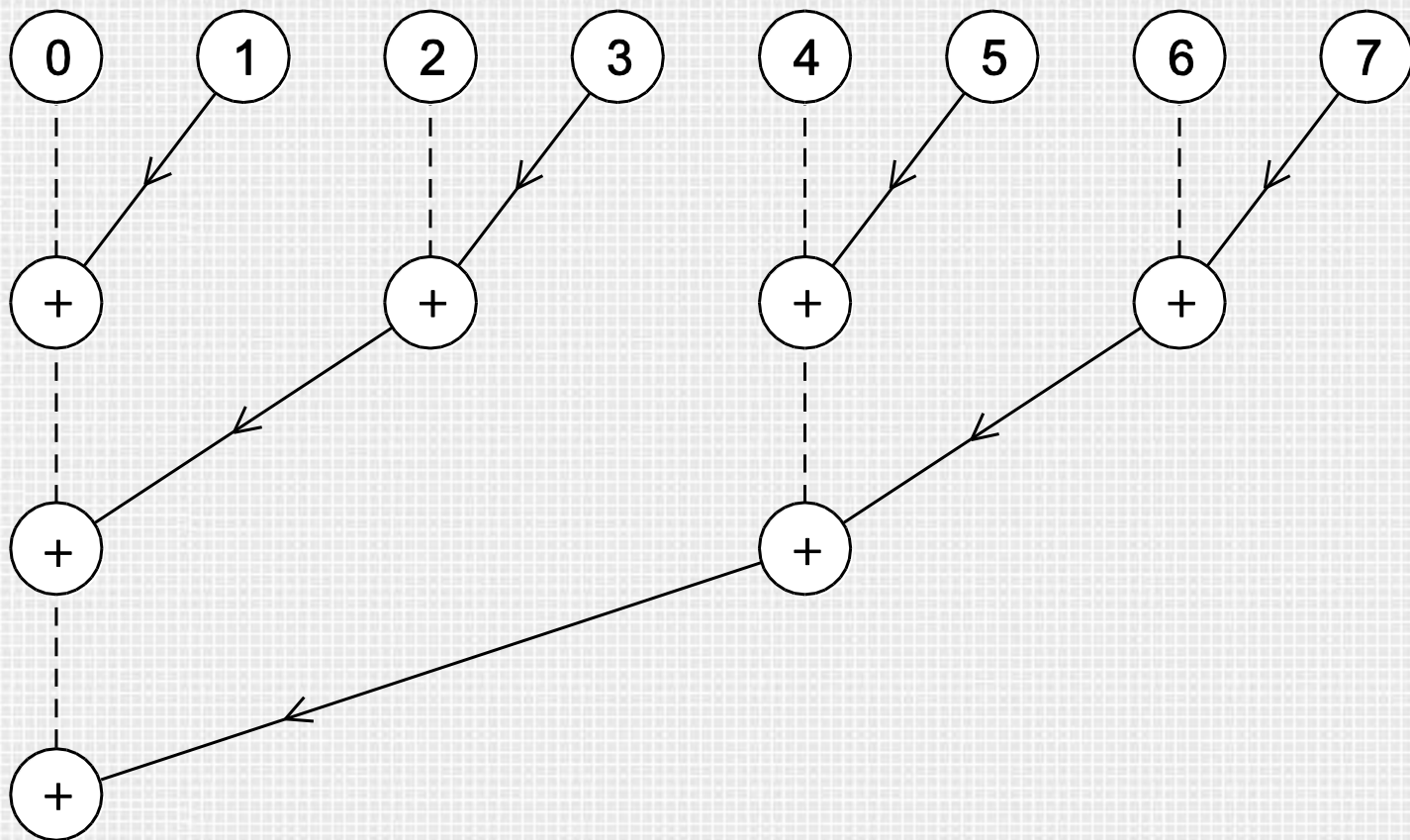


任务与通信的另一种定义方式





本地数组相加





总结

- 串行系统 **Serial systems**
 - 计算机的标准硬件模型是冯.诺依曼结构。
- 并行硬件 **Parallel hardware**
 - **Flynn**分类法.
- 并行软件 **Parallel software**
 - **MIMD**系统的软件开发。此类系统中，大部分程序是单个程序，并通过分支语句实现并行。
 - 单程序多数据流程序 **SPMD**.



总结

- 输入和输出 **Input and Output**
 - 我们学习的编程，只关注其中可以有一个进程/线程访问标准输入 **stdin**，二所有进程可以访问标准输出 **stdout** 和标准错误输出 **stderr**。
 - 但是在调试输出的时候，只让一个进程/线程访问标准输出 **stdout**。