Chap1 绪论

五种常见 MIMD 实现方式：

属性	SIMD	PVP	SMP	MPP	DSM	COW
同构性	SIMD	MIMD	MIMD	MIMD	MIMD	MIMD
同步性	指令级同步	异步或弱同步	异步或弱同步	异步或弱同步	异步或弱同步	异步或弱同步
通信机制	数据并行	共享变量	共享变量	消息传递	共享变量	消息传递
地址空间	单空间	单空间	单空间	多空间	单空间	多空间
访存模型	UMA	UMA	UMA	NORMA	NUMA	NORMA
互连网络	定制	交叉开关	总线或交叉开关	定制	定制	商用

并行体系结构编程模型：
- shared address space：共享变量
- msg passing：如 MPI
- data parallel：如 OMP
- dataflow，systolic

Chap2 性能评测和并行编程

公式（注意单位）

渐近带宽r_∞：传送无限长的消息时的通信速率
半峰值长度$m_{1/2}$：达到一半渐近带宽所要的消息长度
特定性能π_0：表示短消息带宽

机器规模	n
时钟速率	f
工作负载	W
顺序执行时间	T_1
并行执行时间	T_n
速度	$R_n = W/T_n$
加速	$S_n = T_1/T_n$
效率	$E_n = S_n/n$
峰值速度	$R_{peak} = n R'_{peak}$
利用率	$U = R_n/R_{peak}$
通信延迟	t_0
渐近带宽	r_∞

CPI (cycle per instruction) =

Σ时钟周期数/Σ指令数

CPU 时间=指令数*CPI*时钟周期
=时钟周期数/f

MIPS (million instruction/s)
= f (MHz) / CPI

点到点通信开销 t(m) (us)

$= t_0 + m/r_\infty$

=启动时间(us)+消息长度(MB)/渐进带宽(MB/s)

$t_0 = m_{1/2}/r_\infty = 1/\pi_0$

(us)=(B)/(MB/s)=1/(MB/s)

Availability
= MTTF /（MTTF+ MTTR）

加速比性能定律

Amdahl: $S = \frac{Ws + Wp}{Ws + W_P/p} = \frac{p}{1 + f(p-1)}$

考虑 Wo $S = \frac{W_S + W_P}{W_S + \frac{W_P}{p} + W_O} = \frac{p}{1 + f(p-1) + W_O p/W}$

p→∞时，

上式极限为：

S= 1 / f

Gustafson:

S= $\frac{W_S + pWp}{W_S + p \cdot Wp / p} = f + p\,(1\text{-}f)$

问题规模 Wp 增加，串行分量比例 f 下降

Sun&Ni:

S= $\frac{f + (1-f)G(p)}{f + (1-f)G(p)/p}$

G=1 时，为 Amdahl。G=p 时，为 Gustafson。

可扩放性评价标准

等效率标准

$$S = \frac{T_e}{T_p} = \frac{T_e}{\frac{T_e + T_o}{p}} = \frac{p}{1 + \frac{T_o}{W}} \qquad E = \frac{S}{P} = \frac{1}{1 + \frac{T_o}{T_e}} = \frac{1}{1 + \frac{T_o}{W}}$$

，

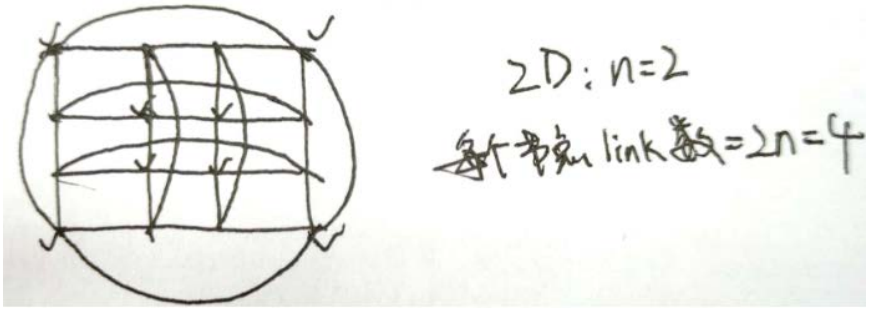
等速度度量标准

平均速度$\bar{V}$ = V /p = W / Tp, 为并行运行时间

平均延迟度量标准

平均延迟公式1：$\bar{L} =$ 启停时间 + 其他延迟$/p = \Sigma^p(T_{para} - T_i + L_i)/p$

公式2：$\bar{L} = T_o/p = (pT_{para} - T_s)/p$



Chap3 互连网络

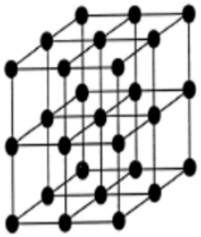
互联网络（IN）

直接互联：处理机连处理机，包需要中间处理机中转。
间接互联：包由 switch elements（如 shared bus, crossbar）中转。

直接 IN（静态互连网络）

以下四种都是严格正交的（每个节点都有到各维的边）：n 维网格、k 元 n 立方、环网、超立方

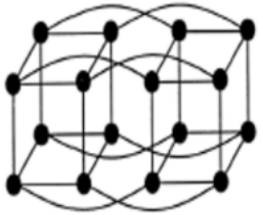
1. n 维网格
   3 元 3 维网格:



2. k 元 n 立方
   定义：
   - 严格正交：每个节点都有到各维的边。
   - k_i是维度D_i的节点数，对立方来说，所有$k_i = k$。（k元n立方是n维网格的特例）
   - 对立方来说，所有节点相邻节点数相同。

   性质：
   - **如果k=2，所有的节点都有n个相邻节点。**
   - **如果k>2，所有的节点都有2n个相邻节点。**
   - **当n=1时，k元n立方变成了具有k个节点的双向环。**

   e.g. 4 元 2 立方



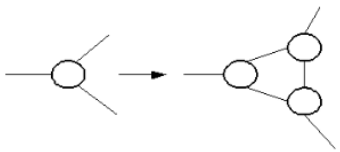
3. 超立方：e.g. 4 维超立方（2 元 4 维立方）

超立方就是2元n立方，也叫二进制n方。

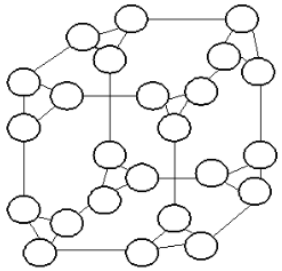
> 有立方性质，k=2元时，所有节点都有n个相邻节点，所以对超立方来讲，n即是维数，也是相邻节点数。

4. 立方环

将立方每个节点替换为 **m** 元环，就得到了一个 **m-**立方环。



(c)顶点代之以环



(d)3-立方环

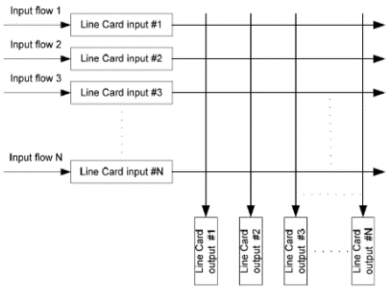
表3.1 静态互连网络特性一览表

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 个节点	2	$N-1$	1	非	$N-1$
环形	N 个节点	2	$\lfloor N/2 \rfloor$（双向）	2	是	N
2-D网孔	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$2(\sqrt{N}-1)$	$\sqrt{N}$	非	$2(N-\sqrt{N})$
Illiac网孔	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$\sqrt{N}-1$	$2\sqrt{N}$	非	$2N$
2-D环绕	$(\sqrt{N}\times\sqrt{N})$ 个节点	4	$2\lfloor\sqrt{N}/2\rfloor$	$2\sqrt{N}$	是	$2N$
二叉树	N 个节点	3	$2(\lceil\log N\rceil-1)$	1	非	$N-1$
星形	N 个节点	$N-1$	2	$\lfloor N/2 \rfloor$	非	$N-1$
超立方	$N=2^n$ 节点	n	n	$N/2$	是	$nN/2$
立方环	$N=k\cdot 2^k$ 节点	3	$2k-1+\lfloor k/2 \rfloor$	$N/(2k)$	是	$3N/2$

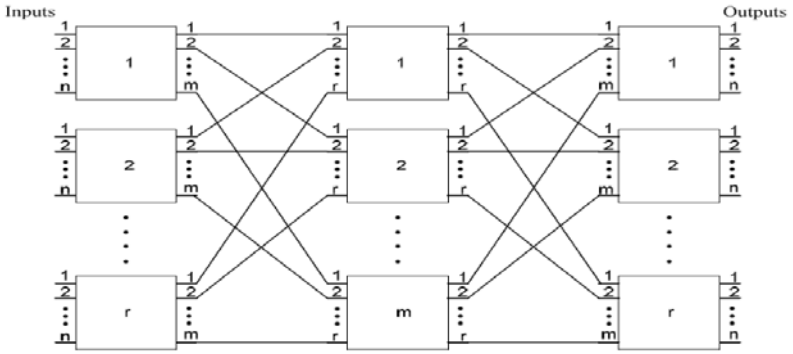
间接 IN（动态互连网络）

分为 shared bus, crossbar nets, MIN

阻塞/非阻塞：能否实现输入-输出的 n!种置换



MIN



- 非阻塞的：Clos nets
- 会阻塞的：

特点：

N个处理器，每一级***kxk***开关之间恰好有***N***条链路，网络共有log_kN级，其中每级具有***N/k***个开关。

通常 k 取 2，所以级数就是 logN，每级 N/2 个开关。

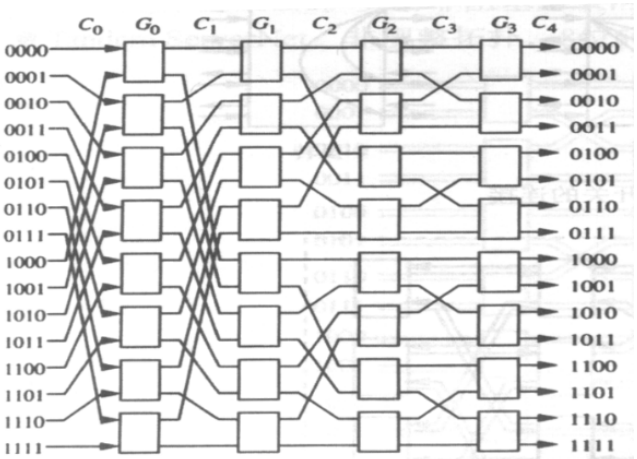
代表：Banyan, Omega, Baseline, Reverse baseline, Indirect binary n-Cube networks（二元 n 立方，就是 n 维超立方）

级间连接模式

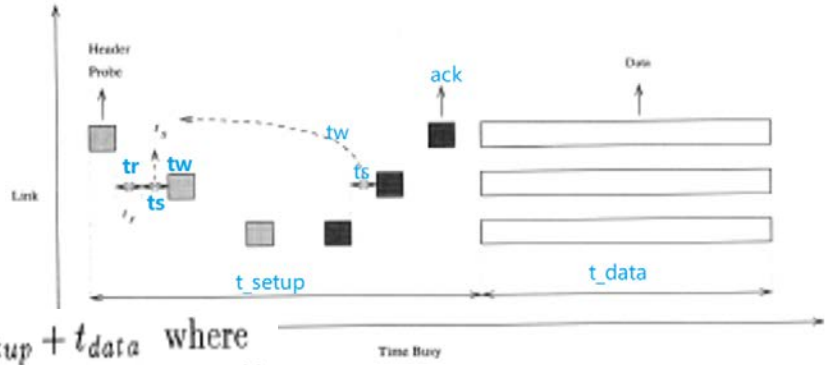
1. 完全混洗排列：就是循环左移
2. 蝶形连接：第 **i** 蝶排：交换 **i** 位和 **0** 位
3. 立方体排列：第 **i** 立方体排：**i** 位取反
4. 基准排列（**baseline**）：第 **i** 基排：**i** 位-**0** 位循环右移

Delta MIN

1. Baseline：C0 完全混洗；Ci, i=1..n，第 n-i 个基准排列
2. Buffefly：Ci, i=0,n-1 为第 i 个蝶排；Cn 恒等排列
3. Cube：C0 完全混洗；Ci, i=1..n，第 n-i 个蝶排
4. Omega：Ci, i=0..n-1 完全混洗；Cn 恒等排列



交换技术



电路交换

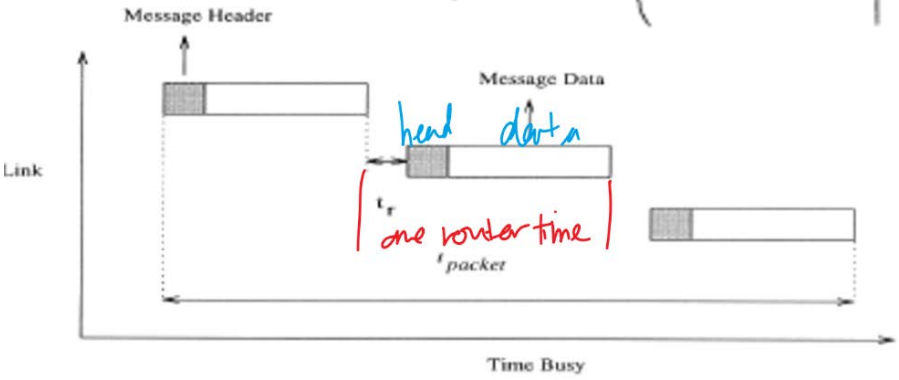
流控单位：微片

$$t_{circuit} = t_{setup} + t_{data} \quad \text{where}$$
$$t_{setup} = D * (t_r + 2 * (t_s + t_w))$$
$$t_{data} = \frac{1}{B} * \left\lceil \frac{L}{W} \right\rceil$$

路径长度

报文交换（SAF, Save and Forward

$$t_{packet} = D * \left(t_r + (t_s + t_w) * \left\lceil \frac{L+W}{W} \right\rceil \right)$$

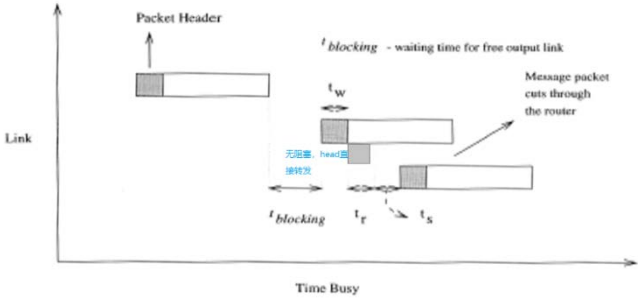


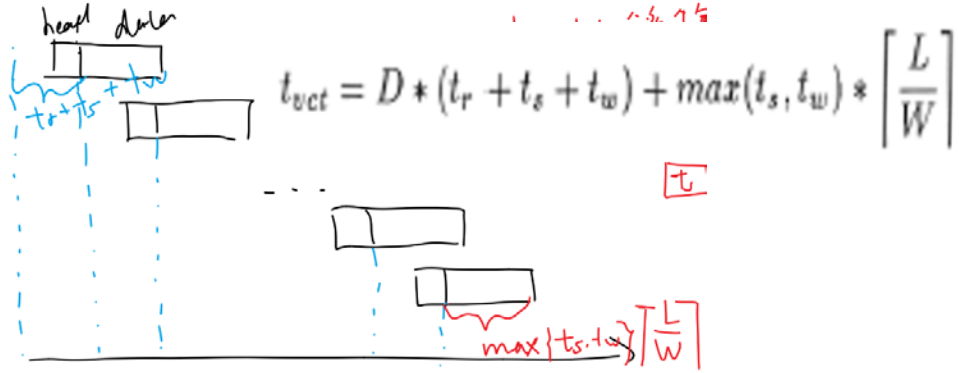
图中一段即一条 **link** 上的传输

虚跨步交换（VCT, Virtual Cut Through） 流控单位：报文

- 报文交换假定:在制定路由决策及报文转发之前,报文必须完全被缓冲。
- 实际上在接收到报文头后就可以马上进行路由决策，而且如果输出通道是空闲的，路由器可以马上转发报文头随后的数据字节。

无阻塞时间:

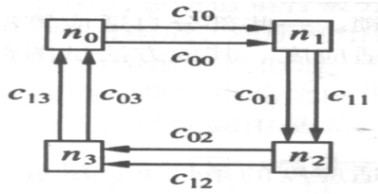




虫孔交换：流控单位变微片 版的 **VCT**，更小更紧凑

无阻塞时间和无阻塞 VCT 一样。

虚通道：将一条物理通道分成多条逻辑通道（虚通道）复用，交换机内的缓冲也相应分成多份。目的是减少死锁。



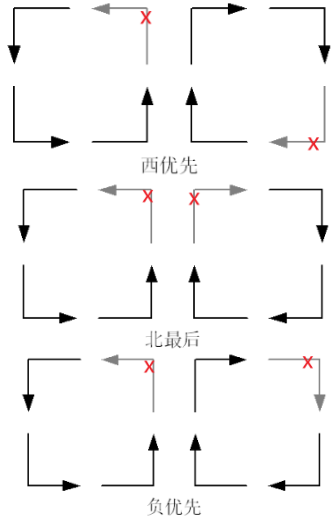
单播路由算法

虚通道解决死锁：**n** 维网格。设虚通道消除环：分高低通道，编号小->大 走高通道，编号大->小 走低通道。

转弯模型：

- 对于二维网孔，有**8**种可能的“转弯”，会形成两种简单的圈
- 在二维网孔中，有**16**种方法禁止两转弯（**Two Turn**），其中**12**种可以避免死锁，只有**3**种是独立的。

- 16-12=4 种构成抽象环



西优先算法:

首先，西优先：Xoff<0,

Channel X-

然后，Xoffset>=0, Yoffset>=< 0

共 6 种组合。

X, Y 不等于 0 时可放一起，Channel = Select (X+-, Y+-)

P 立方路由算法：源$s=s_{n-1}s_{n-2}...s_0$目的$d=d_{n-1}d_{n-2}...d_0$

如果$s_i=0$且$d_i=1$，则$i \in E_0$，否则$i \in E_1$

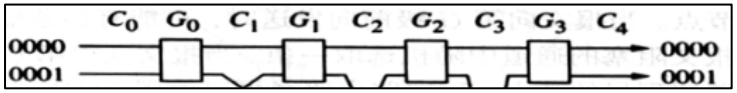
MIN 中的路由

以Omega MIN为例：

1. 先分析地址映射
2. 比较多个I/O对，每一级输出，未出现相同则无阻塞

Delta 网自路由：

第 i 级开关 Gi 的自路由标志



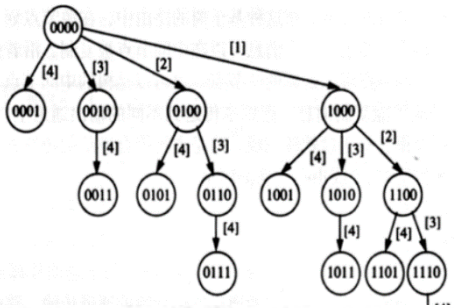
Omega：

开关级数 i
对应dst第 n-i-1 位
该位值（第i级自路由标志）

Butterfly：$t_i = d_{(i+1)\%n}$

Cube：$t_i = d_{n-1-i}$

例 5.3　图 5.11 显示了一个 4 立方中根节点为 0000 的广播树，方括号中的数字表示对应消息传输所用的时间步。



多播路由算法

1. 基于树的多播路由

（1）超立方中的广播树

输入：s 为源节点，v 为本地节点

过程：

1. 令 $d = v \oplus s$。
2. 令 $k = FirstOne(d)$。
3. 如果 $k = 0$，则退出。
4. 对于 $i = k-1$ 到 0，经第 i 维输出通道向邻居节点 $v \oplus 2^i$ 发送一个消息副本。

考虑一个n立方拓扑结构，该算法形成一棵生成二项式树，系统中的每个节点都在不超过n步的时间内恰好接收一次广播消息。令s为源节点的地址，v为接收广播消息的节点地址，FirstOne(d)表示一个n位二进制数d中最低有效位为1的位置(0到n-1)；若d=0，令Firstone(d)=n。

（2）双通道 XY 多播虫孔路由

- 该算法基于网络分割概念，一个多播操作由几个子多播操作实现，每个子多播以一个目的节点子集为目的，在不同的子网上路由。因为子网是不相交和无环的，不存在任何资源的环相关，该算法是无死锁的
- 二维网格中的每条通道都加倍，网络被分成四个子网：$N_{+x,+y}$ $N_{+x,-y}$ $N_{-x,+y}$ $N_{-x,-y}$，其中$N_{+x,+y}$包含[(i,j),(i,j+1)]和[(i,j),(i+1,j)]的单向通道，以此类推。

（3）MIN 基于树的多播 剪枝

2. 基于路径的多播路由

基于树的多播，由于分叉可能导致死锁。基于路径的多播于是将所有分叉都砍掉（极致“剪枝”），一条路走到底。

2D mesh 中，基于哈密尔顿路径的路由：$l(x,y)=y*n+x$　y为偶数

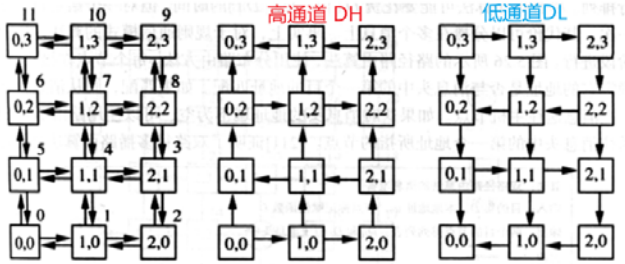
节点标记：n是mesh宽度

单播：下一跳的标记 w。u 当前节点，v 目的节点　$=y*n+n-x-1$　y为奇数

$l(w)=max\{l(z):l(z)<=l(v)，z为u的相邻节点\},若l(u)<l(v)$

$=min\{l(z):l(z)>=l(v)，z为u的相邻节点\},若l(u)>l(v)$

双路径多播：



算法：双路径路由算法的消息准备

输入：目的集 D，本地地址 u_0，节

输出：两个目的节点排列列表：D_H

过程：

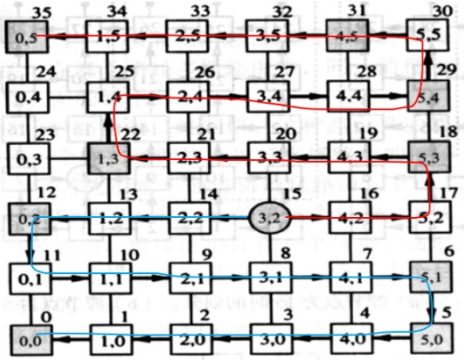
1. 把 D 分成两个集合 D_H 和 D_L，使 D_H 中包含 ℓ 值比 $\ell(u_0)$ 大的目的节点，D_L 中包含 ℓ 值比 $\ell(u_0)$ 小的目的节点。
2. 按升序方式对 D_H 中的目的节点排序；按降序方式对 D_L 中的目的节点排序。
3. 创建两个消息，一个消息头中包含 D_H，另一个消息头中包含 D_L。

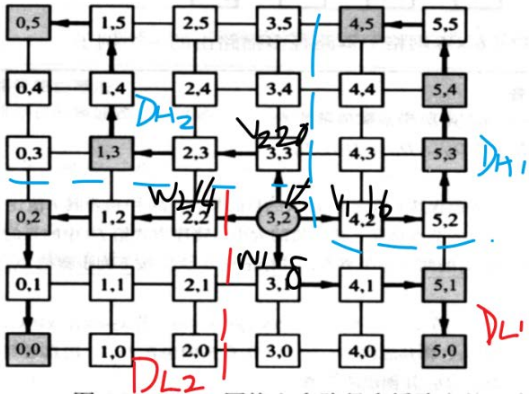
算法：路径路由算法

输入：排序目的地址表 $D_M=(d_1, \cdots, d_k)$ 的消息，本地地址 w，节点标记赋值函数 ℓ

过程：

1. 若 $w=d_1$，则 $D'_M=D_M-\{d_1\}$，消息副本送到本地节点；否则 $D'_M=D_M$。
2. 若 $D'_M=\emptyset$，则终止消息的传输。
3. 使 d 成为 D'_M 中的第一个地址，使 $w'=R(w, d)$。
4. 把消息头中带有目的地址列表 D'_M 的消息送到节点 w' 中。





多路径多播:

算法：多路径路由算法中的消息准备

输入：目的节点集 D，本地地址 $u_0=(x_0, y_0)$，节点赋值函数 ℓ

输出：4 条多播路径的排序目的节点列表 D_{H1}、D_{H2}、D_{L1}、D_{L2}

过程：

1. 把 D 分成两个集合 D_H 和 D_L，使 D_H 中包含其 ℓ 值比 $\ell(u_0)$ 大的目的节点，D_L 中包含其 ℓ 值比 $\ell(u_0)$ 小的目的节点。
2. 按 ℓ 值的大小以升序方式对 D_H 中的目的节点排序；按 ℓ 值的大小以降序方式给 D_L 中的目的节点排序。
3. 假设 $v_1=(x_1, y_1)$，$v_2=(x_2, y_2)$ 是 u_0 的两个邻居节点，其标记比 u_0 大 且l(v1)<l(v2) D_H 分成两个子集 D_{H1}，D_{H2}：

   $D_{H1}=\{(x, y) \mid x \leqslant x_1$，若 $x_1<x_2$；$x \geqslant x_1$，若 $x_1>x_2\}$ 和 $D_{H2}=\{(x, y) \mid x \leqslant x_2$，若 $x_2<x_1$；$x \geqslant x_2$ 若 $x_2>x_1\}$。

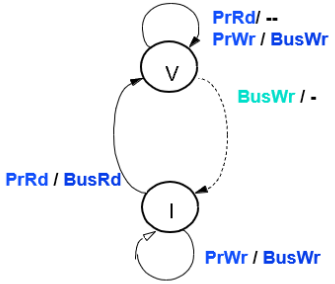
   创建两个消息，一个消息头中包含 D_{H1}，将其送到 v_1；另一个消息头中包含 D_{H2}，消息送到 v_2。
4. 以同样的方法把 D_L 分成两个集合 D_{L1}、D_{L2} 并创建两个消息。

Chap4 SMP

基于侦听的 cache 一致性协议

- **2-state** 协议：**write thru + invalid**
- **MSI**、**MESI** 协议：**write back + invalidate**
- **Dragon** 协议：**write back + update**
- **write thru + update** 会导致大量无用 **update**，所以不用。

2-state 协议（Write-thru + Invalidate）



> V：valid。I：invalid。处理机/Bus；Read/Write。

V，PrRd/-：valid，cache hit，故不会出发bus事件。

V，PrWr/BusWr：write-thru，故要直写到mem，在此之前，先通过bus invalid掉其他cache，即其他cache相应块会经历V -BusWr/- -> I。

V，BusWr/-：响应其他cache发起的BusWr，要invalid掉自己相应的cache块。

I，PrRd/BusRd：invalid，所以cache miss。BusRd，从mem中读进最新的到cache。

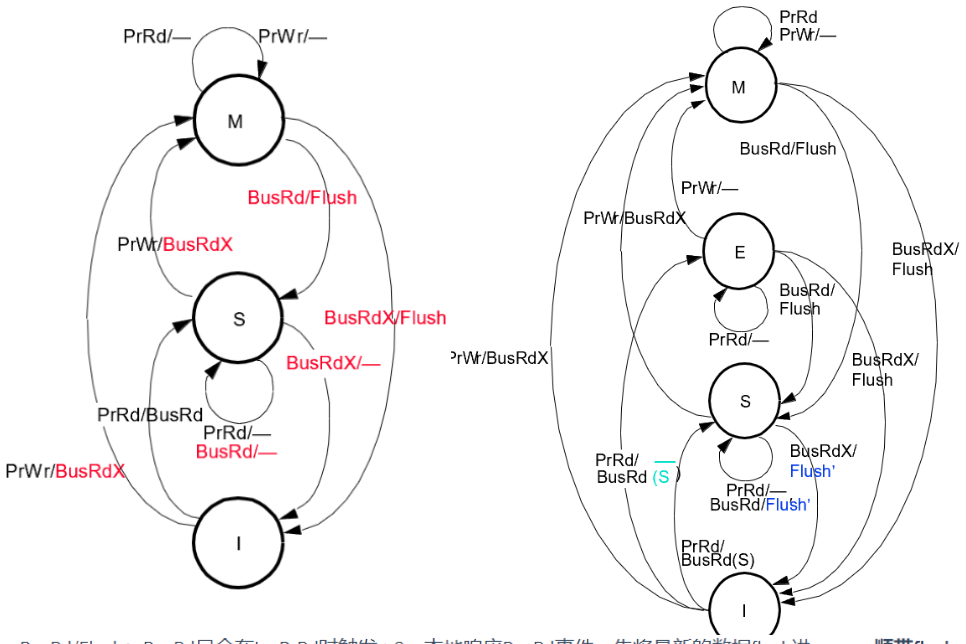
⭐I，PrWr/BusWr：要修改自己的cache时，发现被invalid了。但是，没关系，自己现在更新了，是最新的，故要通过bus invalid掉其他cache的，再直写到mem。

> 可以看出，只要PrWr，就一定会触发BusWr，invalid掉其他cache的块，再write-thru到mem。

MSI 协议（Write-back + Invalidate）

在原型基础上，作以下调整，得到最终的MSI状态转换模型：

1. 对三态重新定义：M意味着Exclusive（Own）。Valid用S（shared）代替，表示当前cache块是共享的，不止自己的cache有。
2. 将原型蓝色的BusRd修改为BusRdX新事件（read to exclusive），作用是通知其他cache块invalid掉自己相应的cache块，这个块要老子独有。
3. 加上对bus事件的相应。
4. BusWB改为Flush，除flush回mem外，如果需求还会顺带flush给其他cache。



M，BusRd/Flush：BusRd只会在I，PrRd时触发->S。本地响应BusRd事件，先将最新的数据flush进mem，**顺带flush给发起BusRd的cache**，再转为S态（现在不止我自己有了）。

S，PrWr/BusRdX：我将修改我的数据，这会使我的cache块最新，别人都invalidate（BusRdX），我要独有。

M，BusRdX/Flush：别人要修改数据，它的cache块会变成最新的，我应该先把我的相应块先flush回mem，再转为I态。

> 可以不flush吧，毕竟自己都不是最新了。Flush应该是保险做法，防止拿着最新cache块的机器突然挂掉，至少mem里存的还是次新的。

S，BusRdX/-：这个BusRdX应该是其他共享的cache，要修改它的cache块时出发的（S，PrWr/BusRdX），所以我可以不flush。

> 按M，BusRdX/Flush的道理说，还是flush一下保险。

MESI 协议

四态：

- **invalid**
- **exclusive or *exclusive-clean* (only this cache has copy, but not modified)**
- **shared (two or more caches may have copies)**
- **modified (dirty)**

> 从原先的M中分出E：就是为做出下面这些优化，如S状态下的flush变为flush'，一个cache flush即可。

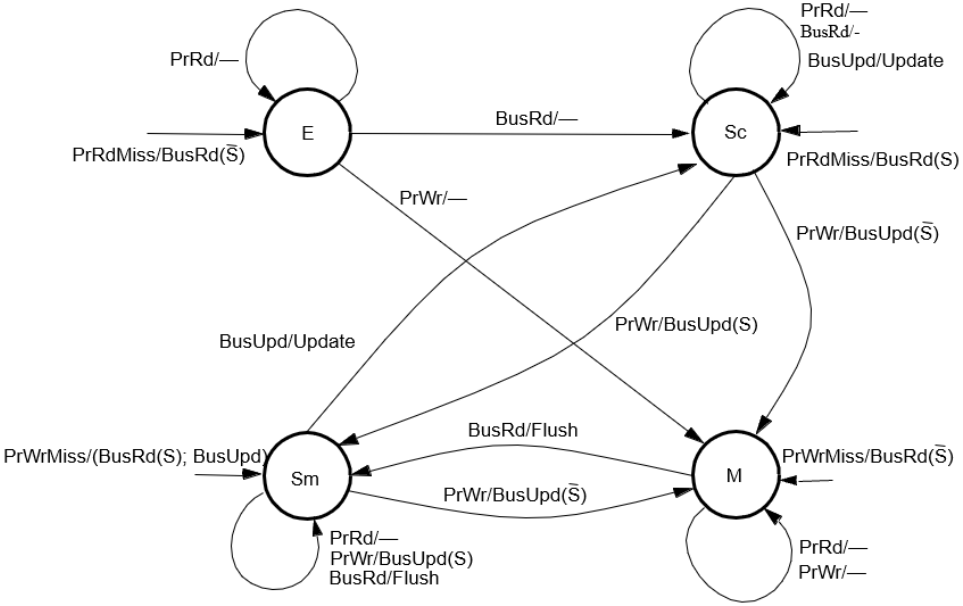
Cache块状态转换图：

- BusRd(S)：应该是BusRd(Shared)，I，PrRd->S时触发，直接从其他cache要，而不是从mem。**别的cache按BusRd响应。**

  BusRd($\bar{S}$)：I，PrRd->E时发起，直接从其他cache要，而不是从mem，并通知其他cache invalidate。**别的cache按BusRdX响应。**
- Flush：只会由M、E发起，flush到mem，如果需求，顺带flush给其他cache（->S时）。只有一个cache flush。

  Flush'：多个cache share相同块，MSI中每个cache都要flush，现在只需**一个cache flush'**即可（to mem, to other caches）。

Dragon 协议（Write-back + Update）



PrRdMiss/BusRd($\bar{S}$)->E：只有当其它cache无状态时，才能成功发起。若其他cache有，则是PrRdMiss/BusRd(S)->Sc。这里的BusRd($\bar{S}$)跟MESI中不同：MESI中是从其他cache要，并将它们都invalid；Dragon只能从mem读，所以，我觉得不加($\bar{S}$)，直接BusRd也OK，大概是象征着独占，所以都加了bar。BusRd(S)和MESI中相同，就是从其他cache要。

Sc, BusUpd/Update ->Sc：比如 其他cache Sm下PrWrMiss触发BusUpd。因为其他cache改了数据，所以本地要Update。

Sc, PrWr/BusUpd(S)->Sm：自己要改数据，升为Sm，并通知其他share的cache update。其他cache按BusUpd响应，这个(S)象征着升为Sm，有种独占的意味（假·独占）。

Sm, BusUpd/Update->Sc：其他cache改数据触发BusUpd，本地要Update，并将为Sc。

Sc, PrWr/BusUpd($\bar{S}$)：通知其他cache我要独占，所以是($\bar{S}$)，应该会导致其他share的cache变为无状态。

M, BusRd/Flush->Sm：其他cache要share，我这最新，flush到mem和要的cache。

Memory Consistency Model

Cache coherence vs. Mem consistency

- Cache coherence：目的是使cache透明化。有了cache后，就多了一份数据X的copy，多个cache和mem中X如何同步。
- Mem consistency：目的是确定多线程背景下的mem order，只看mem，不考虑cache。一个线程内的语句可reorder，多个线程的语句可交错，这就像多副牌混洗，全局牌序（语句序）即mem order。Mem order不同，执行结果可能不同（不一致，因为共享变量）。

Sequential Consistency (SC)

- **A program defines a sequence of loads and stores (this is the "program order" of the loads and stores)**
- **Four types of memory operation orderings**
  - **W→R: write to X must commit before subsequent read from Y ***
  - **R→R: read from X must commit before subsequent read from Y**
  - **R→W: read to X must commit before subsequent write to Y**
  - **W→W: write to X must commit before subsequent write to Y**

SMP 中的同步

- Simple Test&Set Lock

  将test，set封装成一个原子操作，解决了前面simple lock的问题。

  lock可用时=0；已被人拿去时=1。

  bnz lock：lock!=0时，调用lock()，try again。

```
lock:   t&s   register, location
        bnz   lock          /* if not 0, try again */
        ret                 /* return control to caller */
unlock: st    location, #0  /* write 0 to location */
        ret                 /* return control to caller */
```

Test, test&set

考虑到test&set是个原子操作，忙等时只test即可

- `while(lock != 0);`：lock!=0时，表示已经被占用，死循环不断test，但没有set。
- `t&s(lock) == 0` 时返回：意思是t&s成功时返回0吗？

```
void Lock(int* lock) {
  while (1) {

    while (*lock != 0);           // while another processor has the lock…
                                  // (assume *lock is NOT register allocated)


    if (test_and_set(*lock) == 0) // when lock is released, try to acquire it
      return;
  }
}

void Unlock(int* lock) {
   *lock = 0;
}
```

LL-SC

Load-Locked* (or -linked), *Store-Conditional

t&s的放宽版：只要read-write间没有对read所读var的修改，即成功。

但也比t&s更易失败，因为t&s是一个原子操作，现在被拆成了LL-SC的三个原子操作。

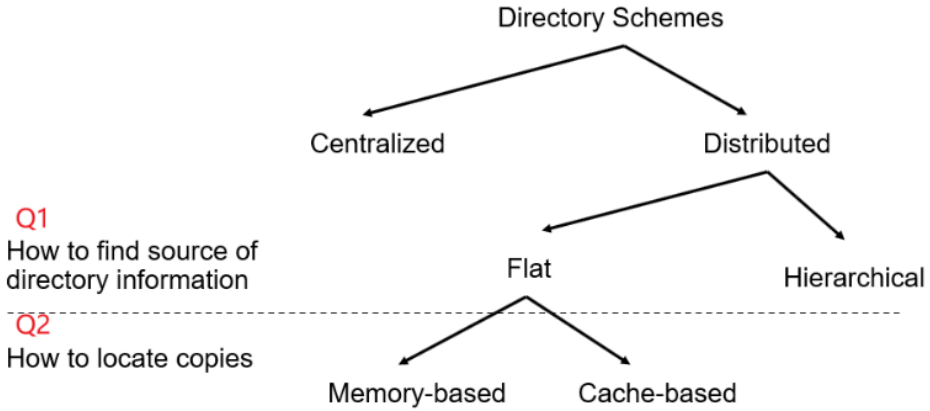
e.g. Simple lock with LL-SC

```
lock:   ll      reg1, location      /* LL location to reg1 */
        bnz     reg1,lock
        //其他操作
        sc      location, reg2      /* SC reg2 into location*/
        beqz    lock                /* if failed, start again */
        ret
unlock: st      location, #0        /* write 0 to location */
        ret
```

> 这个location应该是要互斥访问的资源。
>
> ll reg1, location：先把location copy到reg1，用于LL-SC失败时恢复。
>
> bnz reg1, lock：check 这个锁是否是独占。
>
> 其他操作：处理好得到reg2。
>
> sc location, reg2：将reg2写进location。
>
> beqz lock：再次check是否独占锁，若否，则LL-SC失败，撤销对location的修改。

Chap5 DSM

基于目录的 cache coherence 协议



Centralized：只有centralized mem + centralized dir。not scalable。

Distributed：

- Flat：directory distributed with memory。
- Hierarchical：parent nodes负责管理children。延迟高。

Q1 answers：

- Flat：数据所在的mem，连着的就是要查的dir。
- Hierarchical：发search msg给parents。

Q2 answers：只看Flat的：

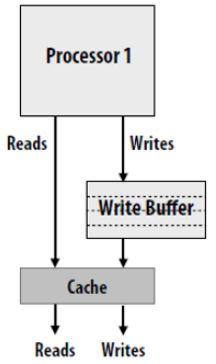
- Mem-based：info about copies存在mem，每个dir都有一份。
- Cache-based：info随copies分布在各cache，each copy points to next（用链表串起来）。

放松的 mem consistency 模型

放松W->R

有两种模型：TSO（total store ordering），PC（processor consist）

都是在proc和cache间加一个write buffer实现其他线程看到write新值滞后：



区别：TSO是reads要没看到write的新值，都没看到。PC则允许有的没看到，有的看到。PC放的更松点。

放松W->R，W->W

PSO模型：

Partial Store Ordering (PSO)

- **Execution may not match sequential consistency on program 1 (P2 may observe change to `flag` before change to A)**

e.g.

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`while (flag == 0);`
`flag = 1;`	`print A;`

Seq consit：P2 print A=1.

PSO：P2先观察到flag更新，还没看到A，故print A=0.

全放松

- **Weak ordering (WO)**
- **Release Consistency (RC)**
  - **Processors support special synchronization operations**
  - **Memory accesses before memory `fence` instruction must complete before the `fence` issues**
  - **Memory accesses after fence cannot begin until fence instruction is complete**

Chap6 MPP, Cluster

- 刀片服务器是将传统的机架式服务器的所有功能部件集成在一块电路板中，然后再插入到机箱背板上。
- 从根本上来说，刀片服务器就是一个卡上的服务器：一个单独的主板上包含一个完整的计算机系统，包括处理器、内存、网络连接和相关的电子器件。

系统特征	SMP	MPP	机群
节点数量(N)	≤O(10)	O(1000)-O(10000)	O(100)-O(1000)
节点复杂度	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

故障恢复

丢失的信息是别人记的迟了，孤儿信息是自己记迟了。
一致检查点：在该检查点集合中包括一系列没有孤儿消息的局部检查点。
严格一致检查点：没有孤儿和丢失的消息的局部检查点集合。

独立检查点：系统中的每个进程单独决定何时保存自己的状态。
多米诺效应：回滚时不断产生孤儿信息或丢失信息，造成不断回滚。

协同式检查点：进程协调它们的保存动作，形成一个全局上一致的状态。

单一系统映像（SSI）

SSI：**Single System Image**。**Cluster** 多个节点系统组织成单一系统使用：

- 单一系统：用户把整个机群视为一个单一的系统来使用；
- 单一控制：系统管理员可从一个单一的控制点配置机群的所有软硬件组件；
- 对称性：用户可以从任一个节点上获得机群服务；
- 位置透明：用户不用了解真正执行服务的物理设备的位置。

关键服务：单一：入口点、文件层次、IO、管理和控制点、网络、存储空间、作业管理系统、用户界面、进程空间。

作业管理

调度作业使负载均衡。
机群作业管理的特点：对异构环境的支持；批作业支持；并行支持；交互支持；检查点和进程迁移；负载平衡。

作业管理系统（JMS）

JMS：Job Manage Sys

三部分：

- 用户服务器：用户提交作业到队列，管理队列。
- 任务调度器：执行任务的调度和排列
- 资源管理器：分配和监测资源，施行调度策略。

作业类型：

- 串行作业：在单节点上运行；
- 并行作业：同时在多个节点上运行；
- 交互作业：需要与用户进行交互，要求快速的周转时间，其输入/输出直接指向终端设备，这些工作一般不需要大量资源，用户可以期望它们迅速得到执行而不必放入队列中；
- 批处理作业：通常需要较多的资源，如大量的内存和较长的CPU时间，但不需要迅速的反应。
- 外部作业（Foreign Job）：不提交给JMS的作业。相对于机群作业而言，外部作业需要快速的反应时间。

作业调度：

问题	方案	主要问题
作业优先级	不可抢占的	高优先级作业的延迟
	可抢占的	开销，实现
资源请求	静态	负载不平衡
	动态	开销，实现
资源共享	独占式	利用率低
	空间共享	分块，大作业
	时间共享	基于进程的作业控制，现场切换的开销
调度	独立调度	严重减慢
	成组调度	难以实现
与外来作业的竞争	停留	执行外来作业速度减慢
	迁移	迁移阀值，开销

机群节点共享方式：

- 独占模式：任一时候只有一个作业在机群上运行。
- 空间共享模式：多个作业可以同时在不重叠的节点区域（节点组）上运行。
  - 分块
  - 大作业
- 时间共享模式：在专用模式和空间共享模式下，只有一个用户进程分配给一个节点。而在时间共享模式下，多个用户进程可以分配到同一个节点上。
  - 独立调度：各节点OS进行自己的调度，但这会显著损坏并行作业的性能，因为并行作业的进程间需要交互；
  - 组调度：将并行作业的所有进程一起调度。成组调度偏差（Gang-Scheduling Skew）；
  - 与外来（本地）作业竞争
    - 停留：速度变慢
    - 迁移
      - 节点可用性，迁移开销，阀值

机群文件系统

名字解析（Name Resolution）：如何利用名字来定位文件或目录。

- 集中式方法，由一个节点负责维护映射表。缺点是会出现单点错误和性能瓶颈。
- 分布式方法又可分为两种：
  - 独立名字空间：每个系统都拥有自己独立的名字空间；缺点是：它不是位置独立的（Location-independent），如果一个磁盘移到另一个节点上，要重新安装目录树。
  - 全局名字空间：所有节点都使用统一的全局名字空间。在这种情况下，目录树被划分成域（Domain），每个域有一个名字服务器负责名字解析。

数据条块化：

数据划分成数据块分布存放在磁盘阵列中，称作条块单元（Striping Unit）。连续的条块单元位于不同的磁盘上，盘阵上一组连续的条块单元称作一个条块。

划分粒度：

- 细粒度：所有的I/O操作都访问所有磁盘，高数据带宽。但任何时刻只能为一个逻辑请求服务，
- 粗粒度：小I/O请求只访问少量的磁盘，大I/O请求可以访问所有磁盘

RAID：

实现冗余的方式和数据划分粒度区分5个基本级别

RAID:

- **RAID 1**：位间隔，镜像关系；同时写入，空间浪费较大
- **RAID 2**：位间隔，Hamming码；
- **RAID 3**：位间隔，校验磁盘；细粒度
- **RAID 4**：块间隔，校验磁盘；粗粒度，读操作并发，而写操作必须串行使用校验盘。
- **RAID 5**：块间隔，循环校验。具有最好的小块读、大块读和大块写性能，但小块写性能比其他冗余策略差。

软件RAID：

- 软件（逻辑）**RAID**：
  - 将RAID的思想用在机群中，将数据分布在机群系统的多个磁盘中。
  - 软件RAID表现就象RAID 5，并且与RAID具有相同的优缺点
  - 与RAID的区别，文件系统需要负责分布数据和维护容错级别。
- 条块组（**Stripe Group**）：
  - 将机群系统所有的磁盘组成一个逻辑RAID
    - 向所有磁盘写的大的写操作非常少，导致很多小写操作。但在RAID 5，小的写操作效率差。因此，系统就不能充分利用所有磁盘的写带宽。
    - 节点的网络连接的带宽有限，不能够同时读/写所有磁盘，只能利用部分磁盘性能。
    - 发生故障的可能性大。奇偶校验机制不够，可能同时多个磁盘故障。
  - 解决方法是将数据条块化分布到磁盘的一个子集上（条块组）。
    - 系统需要执行的小的写操作数目大量减少。
    - 网络连接的带宽与条块组中磁盘的集合带宽相匹配，充分利用资源。
    - 系统中允许多个磁盘失效，只不过不能是属于同一条块组的多个磁盘。
  - 代价：减少了磁盘存储容量和有效带宽，因为每个条块组都必须有一个存放奇偶校验块磁盘，而在原来的方法中整个系统只要一个存放奇偶校验块的磁盘。

日志结构的文件系统：读性能换写性能

日志结构文件系统将整个文件系统作为一个日志来实现。日志结构的文件系统在每次块被写到一个文件时都将数据块加到日志的末尾，同时将以前写的块置为无效。这种方法允许不管写的块顺序，每个文件被顺序写入，因此提供了更快的写速度。同时简化了状态的恢复。

解决小写问题

日志结构文件系统 + 逻辑RAID

基本思想是：每个客户节点保存它的应用程序作出的所有修改的日志。这些日志被保存在内存中，当日志足够大需要使用所有磁盘时，才计算奇偶校验块。然后客户节点日志和奇偶校验块才被发送到系统中的磁盘上.