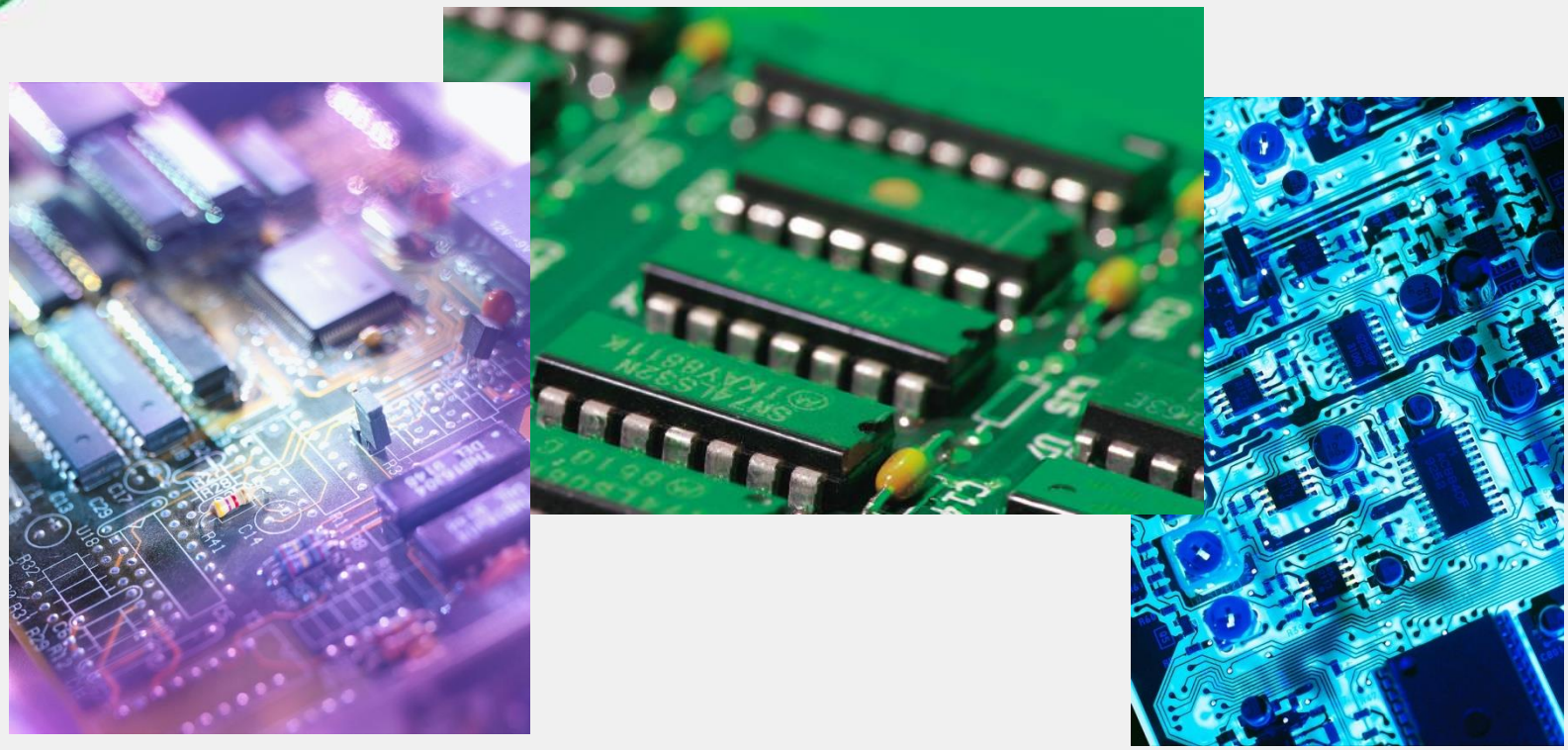
第2讲并行硬件

内容

- 背景Some background
- 冯诺依曼模型的改进Modifications to the von Neumann model
- 并行硬件Parallel hardware

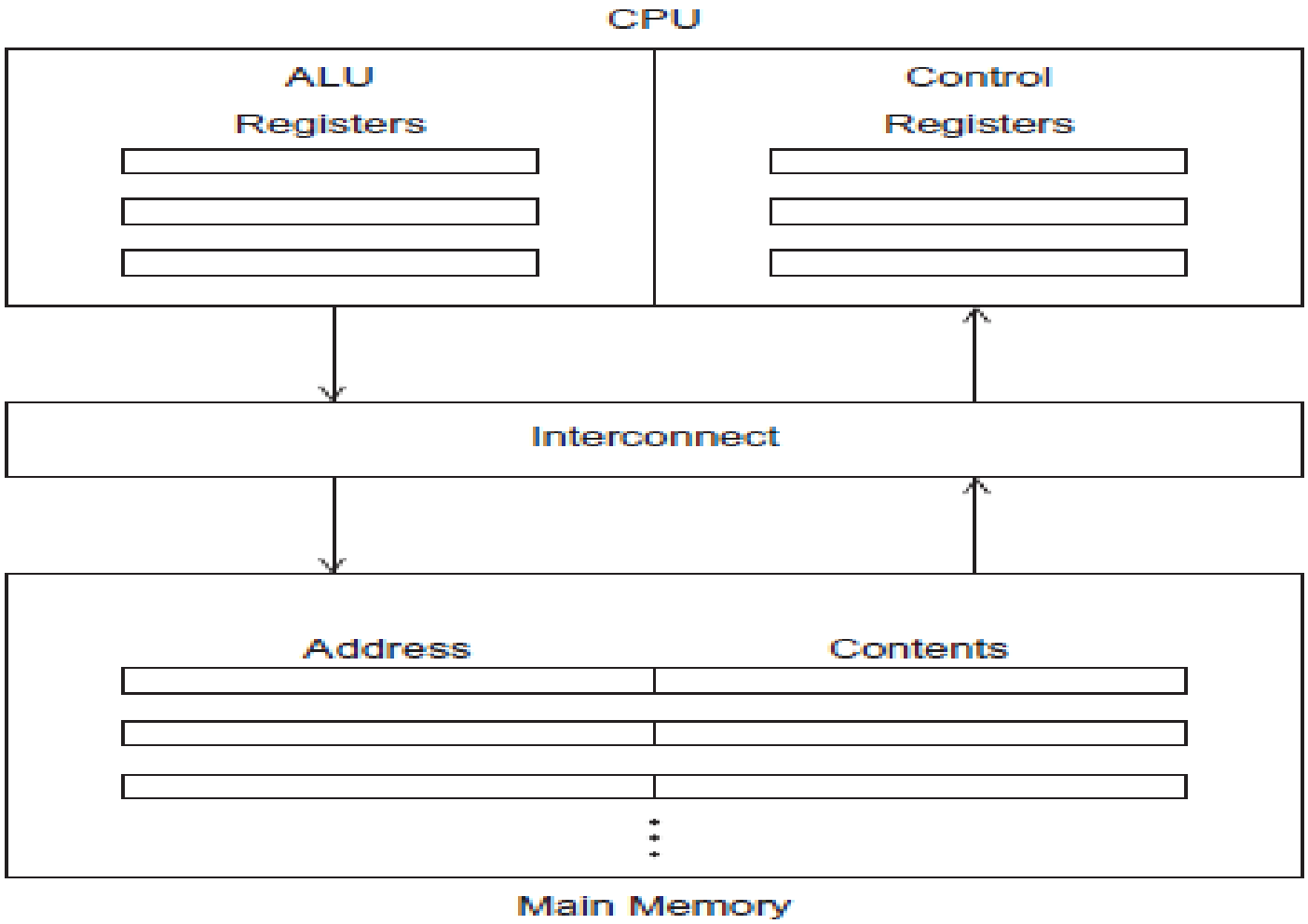


背景(SOME BACKGROUND)

串行硬件和软件

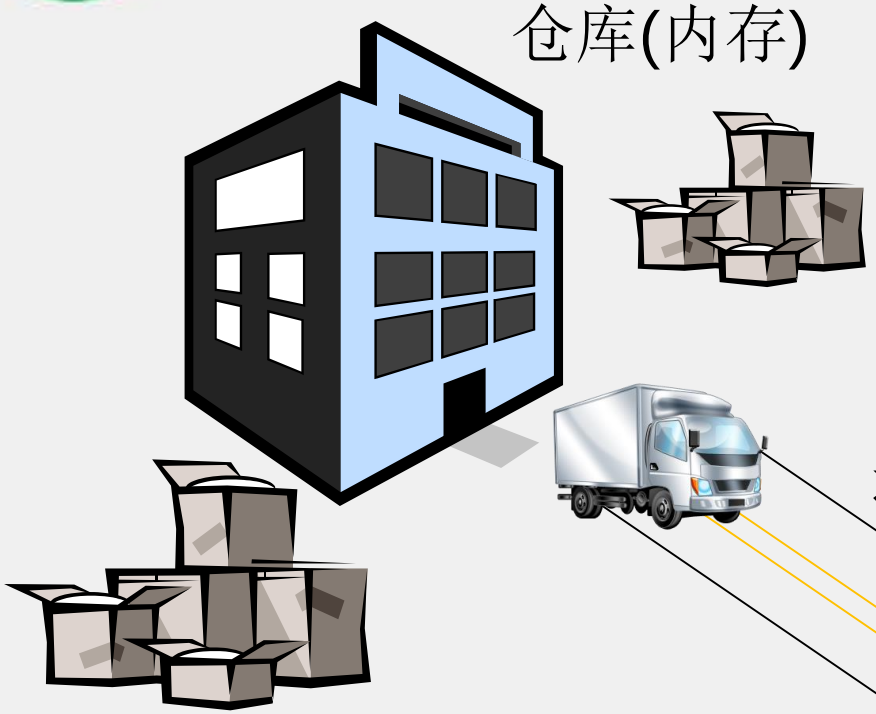
一次运行一个程序

冯诺依曼体系结构


CPU
ALU
Registers
Control
Registers
Interconnect
Address
Contents
Main Memory

冯诺依曼瓶颈von Neumann bottleneck

仓库(内存)

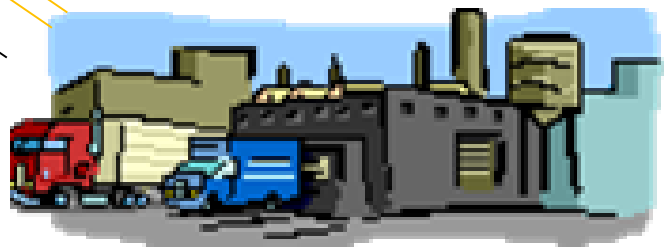


内存墙memory wall

——内存的读写慢速使CPU性能不能发挥

运输线（总线）——慢

工厂(CPU)



内存系统性能的局限——内存墙影响

- 内存系统通常是大多数应用的瓶颈，而非处理器的速度。
- 延迟和带宽是决定内存系统性能的重要参数。
- 延迟是从发送内存请求到请求数据在处理器中可用的时间。
- 带宽是数据从内存送到处理器的速率。

内存系统的性能：带宽和延迟

- 理解延迟和带宽的差别是非常重要的。
- 考虑消防水管的例子。如果打开消防龙头后2秒钟水才从消防水管的尽头流出，那么这个系统的延迟就是2秒。
- 一旦水开始流动，如果水管中水的流速为5加仑每秒，则此系统的带宽是5加仑每秒。
  - 如果你想立即得到水管中的水，减少延迟是重要的。
  - 如果你想扑灭大火，你需要更高的带宽。

内存延迟：一个例子

- 考虑某一处理器以1GHz（1纳秒时钟）运行，与之相连的DRAM有100纳秒的延迟（无高速缓存）。假设处理器有两个乘法-加法部件，并且在每个1纳秒的周期内能执行4条指令。有以下事实：
- 处理器的峰值速度是4GFLOPS。
- 由于内存延迟等于100个周期，并且块大小为一个字，每次发送内存请求时，处理器在处理数据前必需等待100个周期。

内存延迟：一个例子

- 考虑在上述平台上计算两个向量的点积。
- 计算点积对每对向量元素进行一次乘法-加法运算（浮点运算），即平均每次浮点运算需要取一次数据。
- 这种浮点运算的最大速度仅仅为每100纳秒1次（1GFLOPS/100或10MFLOPS），只是处理器峰值速度的很小部分！

内存带宽的影响

- 内存带宽是由内存总线的带宽和内存部件决定的。
- 可以通过增加内存块大小来提高内存带宽。
  - 底层系统使用l时间单位（l是系统延迟）去发送b个数据单位（b是块大小）。
  - 需要留意的是增加块大小并不会改变系统的延迟。
  - 物理上可以看作一条宽数据总线连接多个存储区。构建宽数据总线的代价是昂贵的。
- 得到第一个字后，连续的字在紧接着的总线周期里被发送到内存总线。

内存带宽的影响：例子

- 再次考虑之前点积的例子，内存块大小从1个字增加为4个字。我们重新在这个方案中考虑点积计算。
- 假设向量在内存中线性排列，那么在200个周期内进行8次浮点运算（4次乘法-加法）。
- 这是因为一次内存访问取出向量中4个连续的字。
- 因此，两次访问能取出每个向量中的4个元素。这等同于25纳秒进行一次浮点运算，或40MFLOPS的最大速度。

访存数目对算法运行时间的影响

- 假设系统有单层存储：内存+CPU，访存速度慢，CPU速度快
- 初始所有数据都在内存中，访存和计算串行
  - m = 算法需要CPU访问内存元素(字)的数目
  - t_m = 一次访存操作的时间
  - f = 算法需要算术运算的次数
  - t_f = 一次算术运算操作的时间 << t_m
  - q = f / m 平均每次访存可进行算术运算的数目：计算强度

q：计算强度：决定算法效率

- 最佳运行时间= f* t_f （这时要求所有数据都在CPU中，按CPU峰值运算的时间）
- 实际运行时间为
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$

硬件匹配度：决定机器效率

- q 越大，运行时间越接近最佳时间 $f * t_f$
  - $q \geq t_m/t_f$， 则获得至少一半的机器峰值速度
- 单位时间执行的浮点操作数

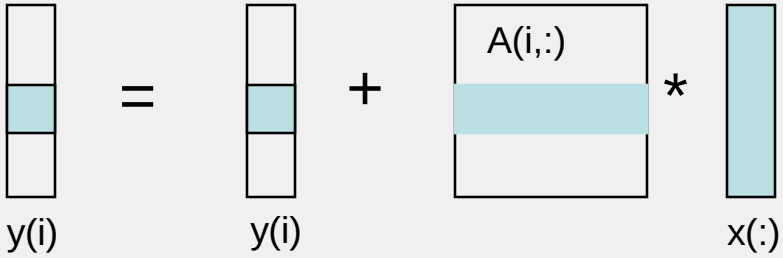
 # Flop = $f/ (f * t_f * (1 + t_m/t_f * 1/q)) = q/(q*t_f + t_m)$

- **q越大，t_m影响越小**

应用: 矩阵向量乘——计算q的例

{公式： y = y + A*x}

```
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



应用: 矩阵向量乘

```
{read x(1:n) into CPU}                    （1）
{read y(1:n) into CPU}                    （2）
for i = 1:n
        {read row i of A into CPU}        （3）
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)  （4）
{write y(1:n) back to memory}             （5）
```

- m = 内存访问数= (1)+(2)+(5)+(3) = $3n + n^2$
- f = 算术运算数 = (4)= $2n^2$
- q = f / m ≈ 2
- 单位时间的#flop = $2n^2 / (2n^2 * t_f + n^2 * t_m) = 2 / (2* t_f + t_m)$

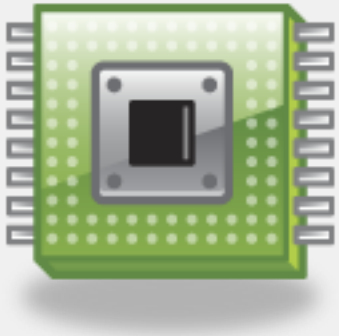
- 由于$t_f << t_m$，所以矩阵向量乘的速度受限于内存

矩阵向量乘模型

- 对nxn = 1000x1000阶矩阵
- 运行时间
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$
  - $= 2*n^2 * t_f * (1 + t_m/t_f * 1/2)$
- 对 t_f 和 t_m, 采用 R. Vuduc’s PhD (pp 351-3)的数据
  - http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf
  - t_m = 最小内存延迟/每个缓存行的字数
    (minimum-memory-latency / words-per-cache-line)

	Clock	Peak	Mem Lat (Min,Max)		Linesize	t_m/t_f
	MHz	Mflop/s	cycles		Bytes	
Ultra 2i	333	667	38	66	16	24.8
Ultra 3	900	1800	28	200	32	14.0
Pentium 3	500	500	25	60	32	6.3
Pentium3M	800	800	40	60	32	10.0
Power3	375	1500	35	139	128	8.8
Power4	1300	5200	60	10000	128	15.0
Itanium1	800	3200	36	85	32	36.0
Itanium2	900	3600	11	60	64	5.5

***硬件匹配度*(q must be at least this for ½ peak speed)**



冯·诺伊曼模型的改进

引入高速缓存

局部性原理

- 程序会在不久的将来（时间局部性）访问邻近的区域（空间局部性）。
- **空间局部性** – 访问附近位置。
- **时间局部性** – 在不久的将来访问。

局部性原理

```
float z[1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

使用高速缓存改善内存访问速度

- 利用局部性原理——程序执行和数据访问行为的局部性——在CPU和内存之间设立高速缓存，**是更小更快的内存元件。**
- 目的：为了让数据存取的速度适应CPU的处理速度
  - 高速缓存读写比RAM快，造价比RAM高

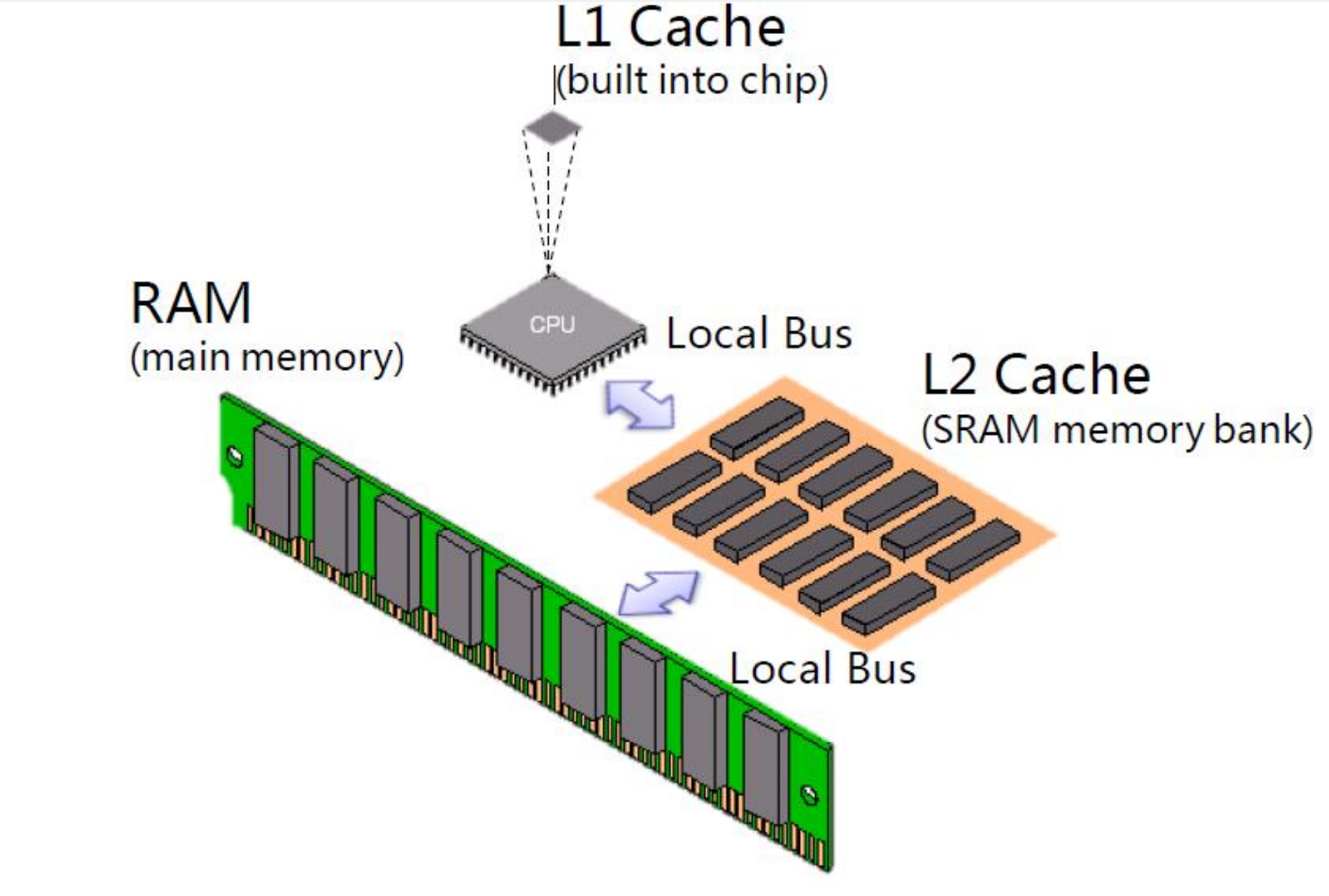
以块为单位访存

- 系统使用更宽的互连结构来访问数据和指令：
  - 一次内存访问能存取一整块代码和数据，而不只是单条指令和单条数据。即以块为单位将内存数据装入Cache，这些块称为高速缓存块或者高速缓存行。
    - 一个典型的高速缓存行能存储8 ~ 16倍单个内存区域的信息。

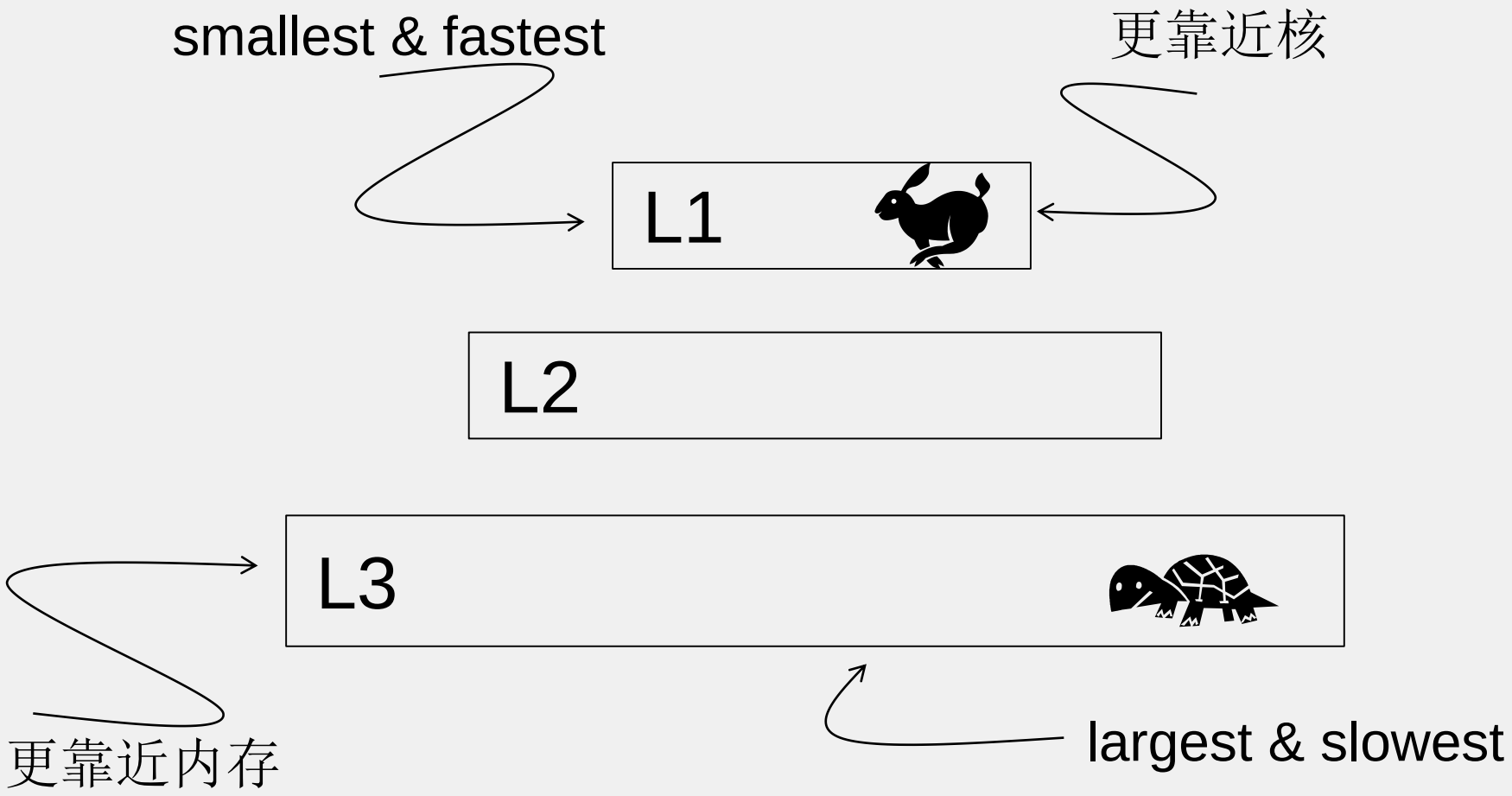
使用高速缓存改善内存访问速度

- 高速缓存是一种低延迟高带宽的存储器。
- 如果某一块数据被重复使用，高速缓存就能减少内存系统的有效延迟。
- 由高速缓存提供的数据份额称为高速缓存命中率。
- 高速缓存命中率通常决定了内存系统的性能。

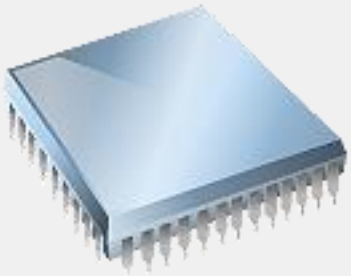
现代多级高速缓存


L1 Cache
(built into chip)
CPU
Local Bus
RAM
(main memory)
L2 Cache
(SRAM memory bank)
Local Bus

高速缓存的层次

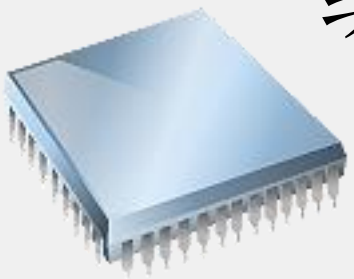


缓存命中(Cache hit)

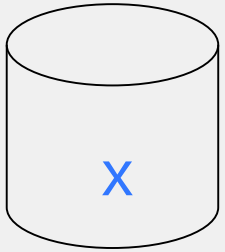


fetch x

缓存未命中(Cache miss)



fetch x



main memory

L1 y sum

L2 r1 z total

L3 A[ ] radius center

高速缓存的影响：例子

- 考虑上一例子的机器架构。在这个例子中我们引入一个大小为32KB延迟时间为1纳秒或一个时钟周期的高速缓存。用这个配置进行两个32 × 32的矩阵A和矩阵B相乘。我们选择这样一组数字使得高速缓存能存储矩阵A、B以及结构矩阵C。

高速缓存的影响：例子（续）

- 在该问题中有以下现象：
- 将两个矩阵取到高速缓存中等同于取2K字，这大约需要200微秒=2K*100ns。
- 两个n × n的矩阵相乘需要进行$2n^3$次运行，这样本例就需要64K次操作，如果每周期执行4条指令，需要16K周期（或16微秒）。
- 因此总计算时间大约是加载/存储操作时间以及计算时间之和，即200+16微秒。
- 所以最大计算速度为64KFLO/216μs或64KFLO/(216 μs/1M) =303MFLOPS。

高速缓存的影响

- 在一段时间间隔内重复引用数据项的概念称为引用的时间局部性。
- 在我们的例子中，我们有$O(n^2)$次数据访问以及$O(n^3)$次计算。这种渐近估计的差异使得在上述例子中高速缓存的作用是令人满意的。
- 数据重用对于高速缓存是至关重要的。

缓存问题

- 当CPU写数据到cache，那么cache中的值与内存不一致
  - **Write-through写通：cache**通过在数据写入cache时更新主内存中的数据来处理此问题。
  - **Write-back写回：**将cache中的数据标记为dirty数据。当cache行将被另一内存行的数据替换时，dirty行将写入被内存。

Cache 映射

确定内存行装入cache的位置，即每个内存行对应哪个cache行。

- 全相联(**Full associative**) —— 一个内存行可以装入cache的任何位置。
- 直接映射(**Direct mapped**) – 每个内存行只能装入cache的唯一固定位置. 如，若cache有m行，内存行装入位置=内存行号%m。
- n路组相联(***n*-way set associative**) –每个内存行可装入cache的n个固定位置. 如，将cache行顺序分组，n行一组，若cache有m组，内存行装入位置按内存行与cache组直接映射获得：k号内存行可装入第k组中的任何cache行。

Example

将16行的主存映射到4行的Cache上

Memory Index 内存行号	Cache 位置		
	全相联	直接映射	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

全相联——每个内存行可对应任何cache行

直接映射——内存行号 % cache行数，余数相同的对应同一cache行

2路映射——内存行号 % 2=余数：0对应0和1号，1对应2或3号 cache行；实际将cache行分组（2个一组），然后内存行号与cache组号直接映射

Caches and programs

```
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]

假设刚开始时，Cache中没有A数组的任何元素，一个高速缓存行可以存放A的4个元素。

如果Cache中有四个高速缓存行，那么cache可以存放A的4行元素。
这时上面两个循环将出现4次Cache缺失。

Caches and programs（2）

```
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]
	A[2][0]	A[2][1]	A[2][2]	A[2][3]
	A[3][0]	A[3][1]	A[3][2]	A[3][3]

假设只有两个Cache line

- 第一个循环Cache缺失还是4次。
  首先访问A[0][0]：不在Cache中 → 一次Cache缺失 → 系统将A[0][0]、A[0][1]、A[0][2]、A[0][3]的行从内存中读出并写入Cache中 → 依次访问的A[0][1]、A[0][2]、A[0][3]都在Cache中；其次访问A[1][0]：一次Cache缺失 → 系统将A[1][0]、A[1][1]、A[1][2]、A[1][3]写入Cache中 → 访问A[1][1]、A[1][2]、A[1][3]都在Cache中；
  类似的，访问A[2][0]：一次Cache缺失；访问A[3][0]：一次Cache缺失；
  共4次Cache缺失。
- 第二个循环Cache缺失？

利用Cache改进矩阵乘法性能
——更一般的讨论方法

应用：矩阵乘法

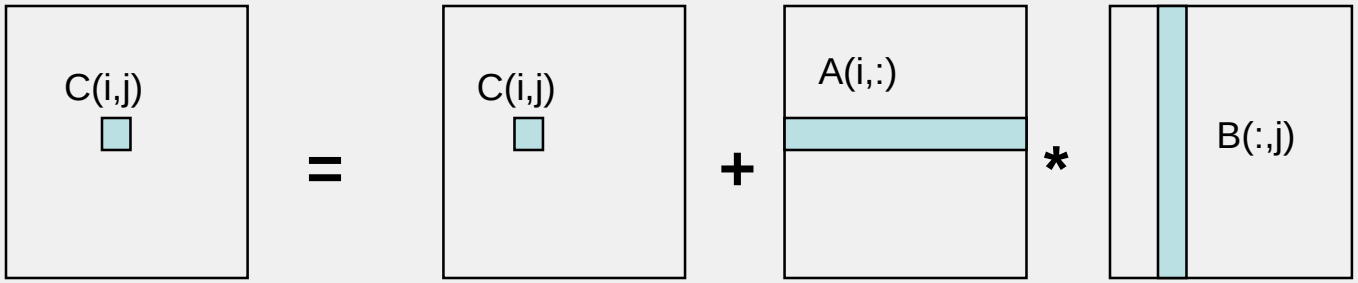
```
{实现 C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

算术运算数 = $2*n^3 = O(n^3)$

访存数 = $3*n^2$ words

■理论上 q 可达 $2*n^3 / 3*n^2 = O(n)$

■实际可以达到吗？



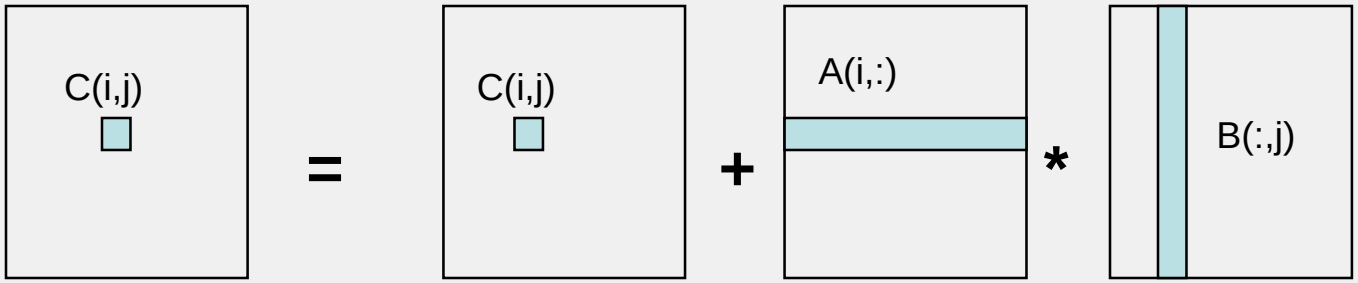
简化假设

- 为简化后面在分析，做如下假设：
  - 忽略处理器中内存和算术运算之间的并行性
    - 算术运算和访存是串行的，它们的执行时间相加
  - 假设快速内存(cache)仅可容纳三个向量和少量其他变量（这与前面例子的假设不同）
    - 对任何级别的cache，这是合理的
    - 对寄存器，这就不合理：寄存器数目<=32 words
  - 假设快速内存访问的时间成本为0
    - 对寄存器，这是合理的
    - 对cache，这不是必须的 (1-2 cycles for L1)
  - 内存访问延迟是常数

简单矩阵乘法(1)

{实现 C = C + A*B}

```
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



简单矩阵乘法(2)

非分块矩阵乘法的访存数目

$m = n^3$ to read each column of B n times

$+ n^2$ to read each row of A once

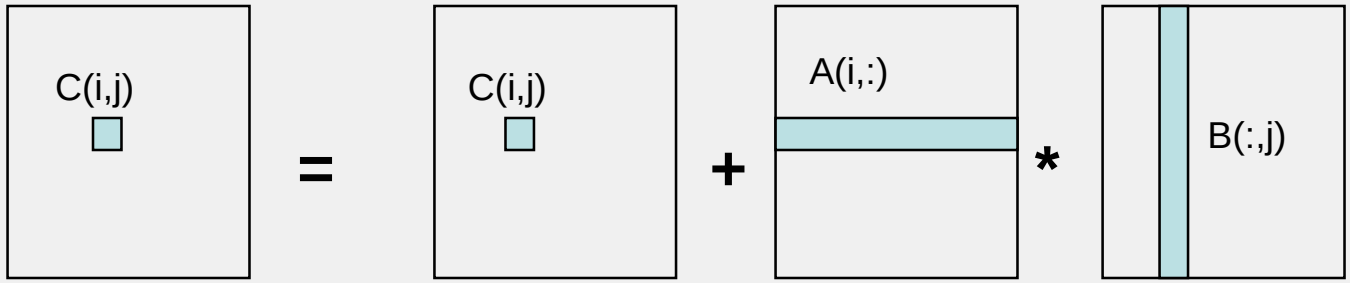
$+ 2n^2$ to read and write each element of C once

$= n^3 + 3n^2$

因此 $q = f / m = 2n^3 / (n^3 + 3n^2)$

≈ 2 （当n很大时）, 仅保持了其矩阵向量乘部分的效率，整体没有改进

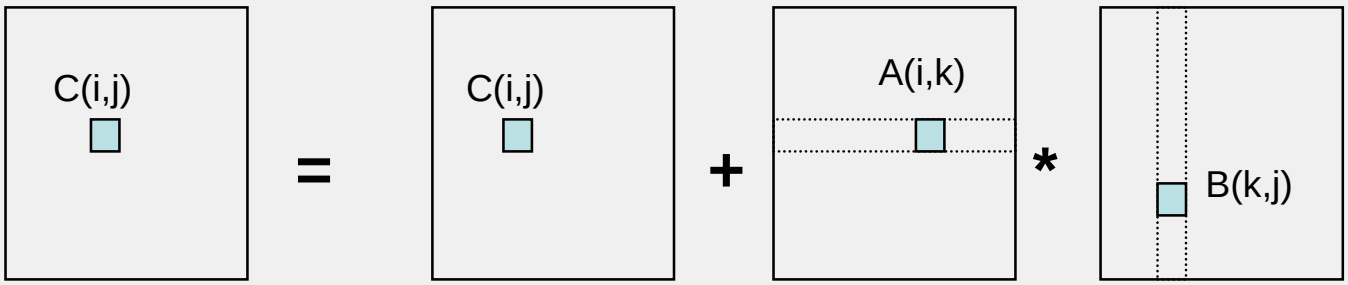
- 内部两个循环只是矩阵向量乘：A的第i行乘以B



分块矩阵乘法(1)

假设A,B,C是N*N阶分块矩阵，其子矩阵是b*b阶的，b=n / N。

```
for i = 1 to N
  for j = 1 to N
    {read block C(i,j) into fast memory}
    for k = 1 to N
        {read block A(i,k) into fast memory}
        {read block B(k,j) into fast memory}
         C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
     {write block C(i,j) back to slow memory}
```



分块矩阵乘法(2)

Recall:

m 是慢速内存与快速内存之间传输元素的数目

矩阵是 nxn 阶，并且有 NxN 个bxb的子块

f 是浮点操作的数目，对这个问题 $f = 2n^3$

q = f / m 是算法计算强度——算法效率的度量

因此:

$m = N*n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (n/N)^2 = N*n^2$)

$+ N*n^2$ read each block of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\approx n / N = b$ for large n

So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply (q=2)。

➢b<n可以任意取吗？

分析b的取值

The blocked algorithm has computational intensity $q \approx b$

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size M_{fast}

$$3b^2 \le M_{fast}, \quad \text{so} \quad q \approx b \le (M_{fast}/3)^{1/2}$$

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size

$$M_{fast} \ge 3b^2 \approx 3q^2 = 3(t_m/t_f)^2$$

- This size is reasonable for L1 cache, but not for register sets
- Note: analysis assumes it is possible to schedule the instructions perfectly

		required
	t_m/t_f	**KB**
Ultra 2i	24.8	14.8
Ultra 3	14	4.7
Pentium 3	6.25	0.9
Pentium3M	10	2.4
Power3	8.75	1.8
Power4	15	5.4
Itanium1	36	31.1
Itanium2	5.5	0.7

优化矩阵乘法的限制

- The blocked algorithm changes the order in which values are accumulated into each C[i,j] by applying commutativity and associativity
  - Get slightly different answers from naïve code, because of roundoff - OK
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \approx b \leq (M_{fast}/3)^{1/2}$$

- There is a lower bound result that says we cannot do any better than this (using only associativity, so still doing n^3 multiplications)

- **Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O((M_{fast})^{1/2})$**
  - **#words moved between fast and slow memory = $\Omega (n^3 / (M_{fast})^{1/2})$**

矩阵乘法的通信下限

- Hong/Kung theorem is a lower bound on amount of data communicated by matmul
  - Number of words moved between fast and slow memory (cache and DRAM, or DRAM and disk, or …) = $\Omega\ (n^3 / M_{fast}^{1/2})$
- Cost of moving data may also depend on the number of "messages" into which data is packed
  - Eg: number of cache lines, disk accesses, …
  - #messages = $\Omega\ (n^3 / M_{fast}^{3/2})$
- Lower bounds extend to anything "similar enough" to 3 nested loops
  - Rest of linear algebra (solving linear systems, least squares…)
  - Dense and sparse matrices
  - Sequential and parallel algorithms, …
- More recent: extends to any nested loops accessing arrays
- Need (more) new algorithms to attain these lower bounds…

内存带宽的影响

- 内存带宽是由内存总线的带宽和内存部件决定的。
- 可以通过增加内存块大小来提高内存带宽。
- 底层系统使用*l*时间单位（*l*是系统延迟）去发送b个数据单位（b是块大小）。

内存带宽的影响：例子

- 再次考虑之前点积的例子，内存块大小从1个字增加为4个字。我们重新在这个方案中考虑点积计算。
- 假设向量在内存中线性排列，那么在200个周期内进行8次浮点运算（4次乘法-加法）。
- 这是因为一次内存访问取出向量中4个连续的字。
- 因此，两次访问能取出每个向量中的4个元素。这等同于25纳秒进行一次浮点运算，或40MFLOPS的最大速度。

内存带宽的影响

- 需要留意的是增加块大小并不会改变系统的延迟。
- 物理上来讲，这种情况可以看作一条宽数据总线连接多个存储区。
- 实际上，构建宽数据总线的代价是昂贵的。
- 在更切实可行的系统中，得到第一个字后，连续的字在紧接着的总线周期里被发送到内存总线。

内存带宽的影响：例子

- 例如，若数据总线为32位，第一个字在100纳秒的延迟后被送到内存总线，其后每个总线周期送入总线一个字。这样计算方法与宽总线例子所讲的略有不同，因为整个高速缓存行在100+3 × （内存总线周期）纳秒后才可用。如果数据总线以200MHz运行，高速缓存行的访问时间就增加了15纳秒。这并没有显著地改变对执行速度的限制。

内存带宽的影响

- 上面的例子清楚地说明增加带宽对提高峰值计算速度的影响。
- 对数据的布局作这样的假设，内存中连续的数据字被连续的指令使用。（引用的空间局部性）
- 如果我们以数据布局为中心，计算次序必须重新排序，以提高数据引用的空间局部性。

内存带宽的影响：例子

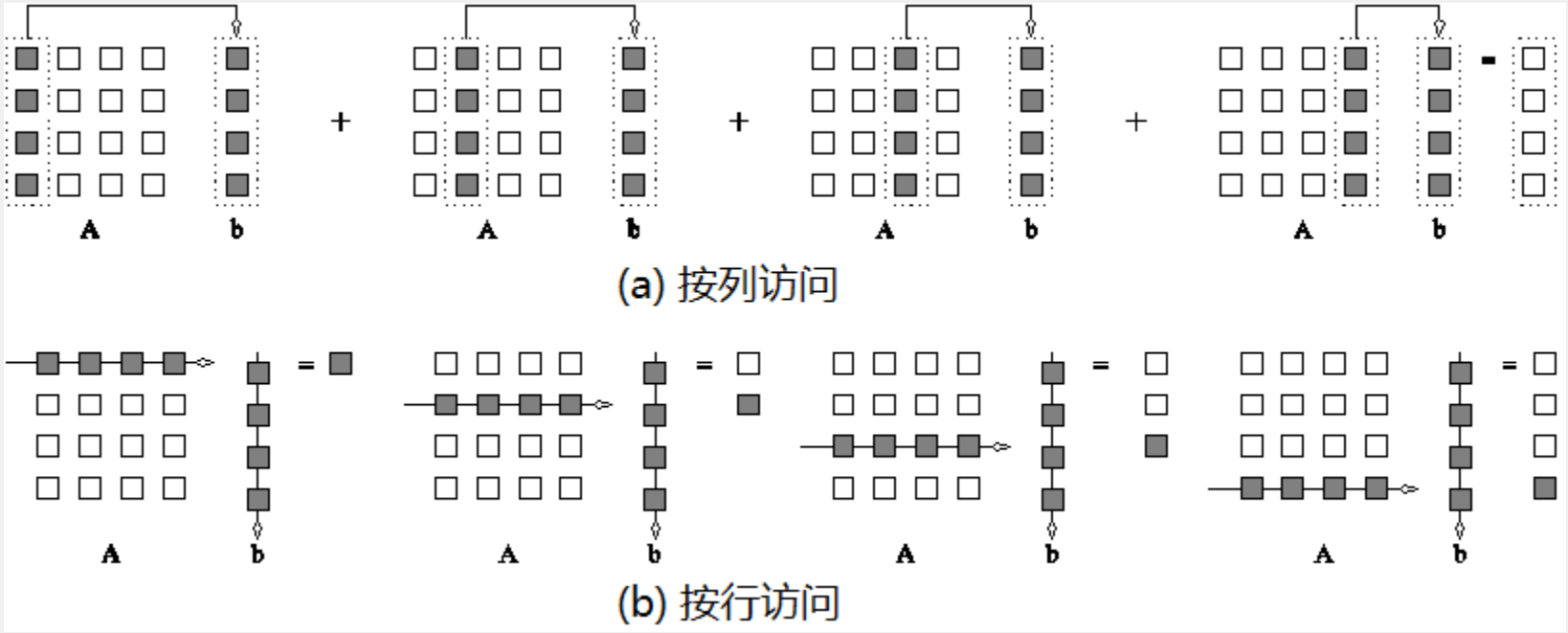
- 考虑以下代码段：

```
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
        for (j = 0; j < 1000; j++)
            column_sum[i] += b[j][i];
```

- 该段代码对矩阵b的列求和，把所得的结果送到向量column_sum中。

内存带宽的影响：例子

- 向量column_sum很小，很容易放到高速缓存中；
- 矩阵b的元素按照列顺序进行访问；
- 跨距访问会导致很差的性能。



- 向量与矩阵相乘：a）列与列相乘，保留运行中的和；b）计算矩阵每一行与向量的点积作为结果的元素。

内存带宽的影响：例子

- 我们可以如下修改上述代码：

```
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++)
        column_sum[i] += b[j][i];
```

- 在本例中，矩阵按行序遍历。性能预计会得到显著提高。

内存系统性能：总结

- 利用应用程序的空间局部性与时间局部性对于减少内存延迟以及提高有效带宽非常重要。
- 某些应用程序具有较大的时间局部性，因此更能承受较低的内存带宽。计算次数与内存访问次数的比是一个很好预测内存带宽承受程度的指标。
- 内存的布局以及合理组织计算次序能对空间局部性和时间局部性产生重大影响。

虚拟存储器

虚拟存储器

- 如果我们运行一个非常大的程序或者访问一个非常大数据集时，将所有的指令和数据装入主存储器是不合适的，所以将它们放在辅助存储器。
- 虚拟内存用作辅助存储的缓存
  - 利用局部性原理；
  - 只将当前执行程序所需要的部分保存在主内存中，暂时不需要的部分放在辅助存储器中的特定空间（称为交换空间swap space），并在需要时调入主存。
  - 与CPU Cache类似，虚拟存储器也是以块为单位进行操作。这些块通常称为页（page），大小4kB~16kB。

虚拟地址和虚页号

- 编译器编译时将程序安排在逻辑地址空间中，所有的指令和数据都有逻辑地址（也称为虚拟地址）
  - 逻辑地址分为逻辑页号和页内偏移两部分
  - 虚页号=逻辑页号.
- 程序运行时需要将逻辑地址转换为物理地址
  - 虚页号转换为物理页号→逻辑地址转为物理地址.
- 页表(**page table)—— 通过页表将**虚页号转换为物理页号.

虚拟地址									
虚页号					页内偏移				
31	30	…	13	12	11	10	…	1	0
1	0	…	1	1	0	0	…	1	1

页表(Page table)

Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program's overall run-time.
- A special address translation cache in the processor.

Translation-lookaside buffer (2)

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.

- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

并发和并行

An operating system “process”

- An instance of a computer program that is being executed.
- Components of a process:
  - The executable machine language program.
  - A block of memory.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up, it waits until it has a turn again. (**blocks**)

Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into (more or less) independent tasks.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.

A process and two threads

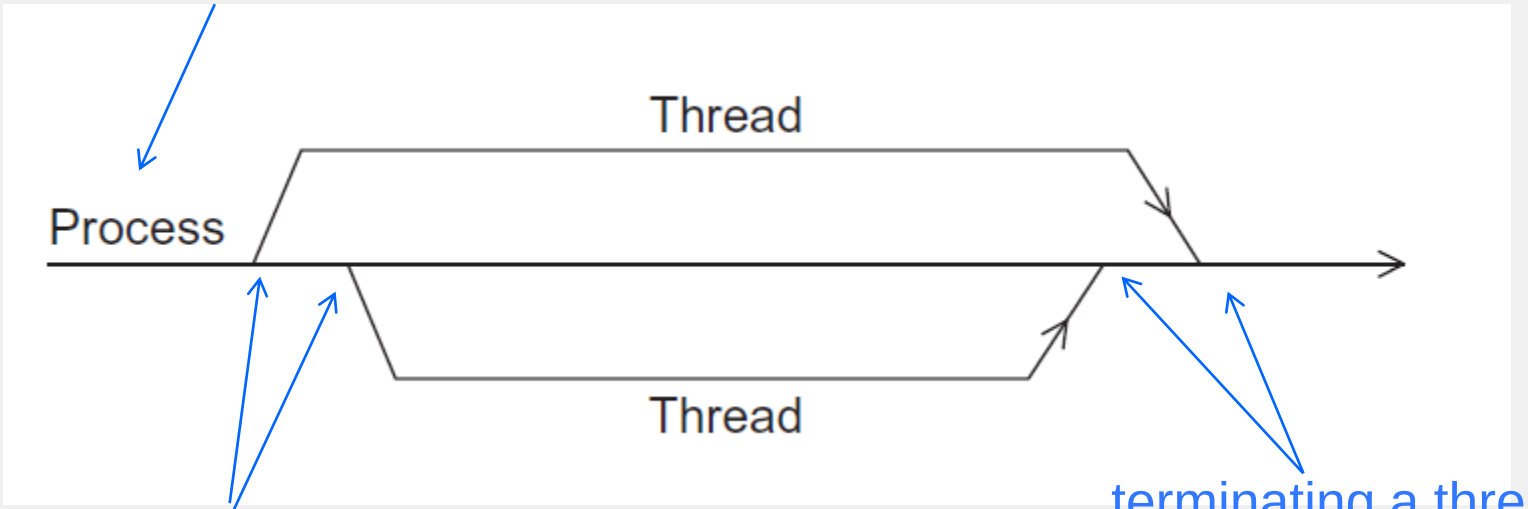


Figure 2.2

指令级并行

Instruction Level Parallelism (ILP)

指令级并行
Instruction Level Parallelism (ILP)

- 通过让处理器的多个组件或功能单元同时执行指令来提高处理器性能。

- **Pipelining流水线** - 将功能单元分阶段安排执行
- **Multiple issue多发射** - 多个指令能够同时启动，或同时可将多条指令交给CPU执行

流水线

与工厂的装配流水线类似

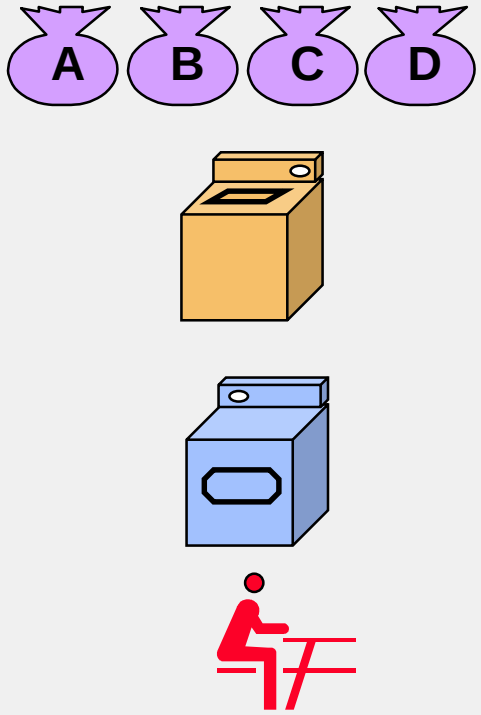
汽车装配流水线：

1. 将汽车的引擎栓到底盘上的同时；
2. 将前面已处理过的部件连接变速器、传动轴和引擎；
3. 把前面两步已完成的产品装上车架。

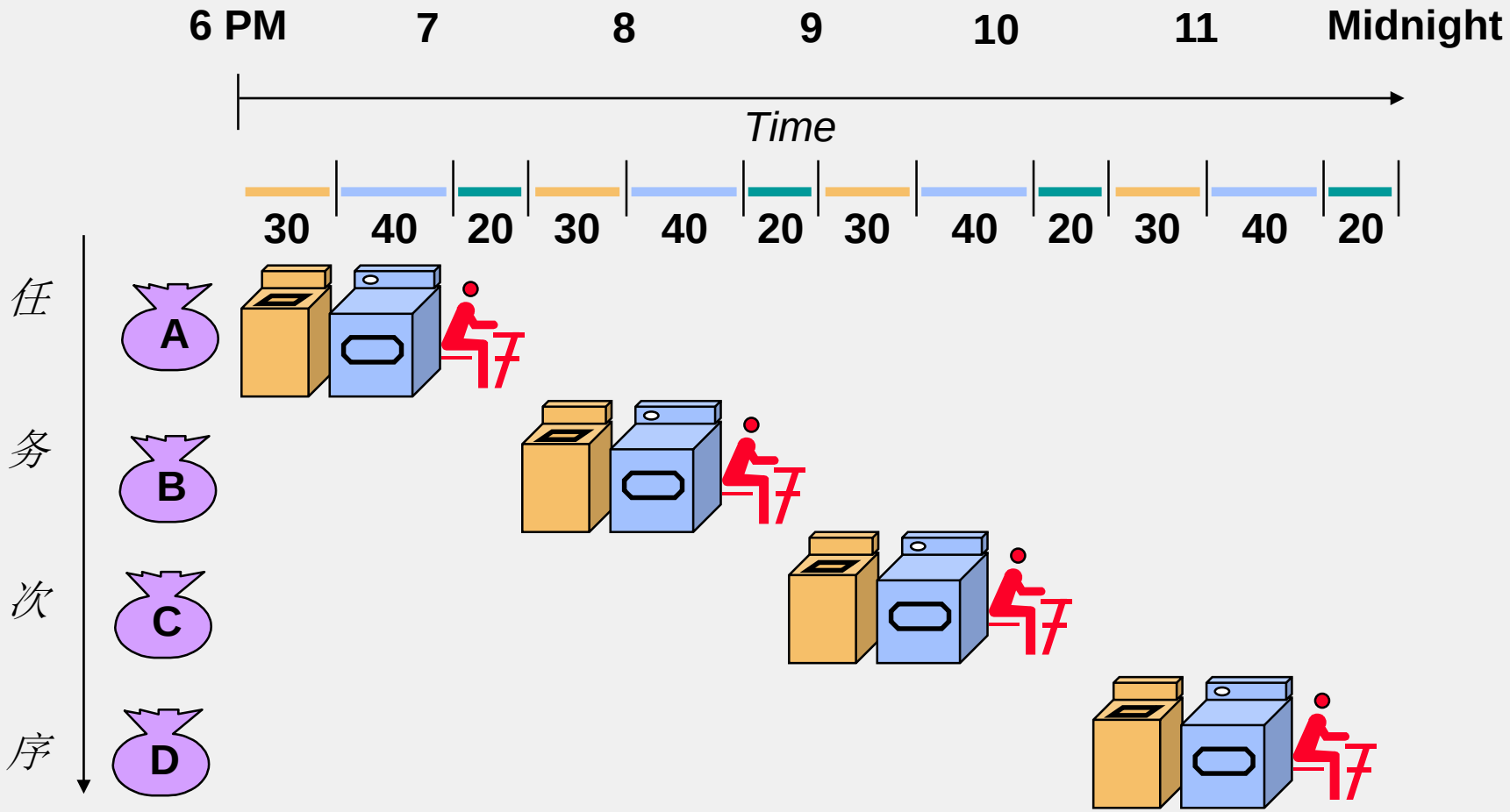
如果同时装配多辆汽车则可以采用流水线方法增加装配产量：单位时间的汽车装配数目。

流水线

- 以洗衣为例，汽车装配类似。
- 四个人：Ann, Brian, Cathy, Dave
  每人都要洗一批衣服。
  洗衣阶段：洗衣, 干衣, 折叠
- 洗衣时间：30 minutes
- 干衣时间：40 minutes
- 折叠时间： 20 minutes



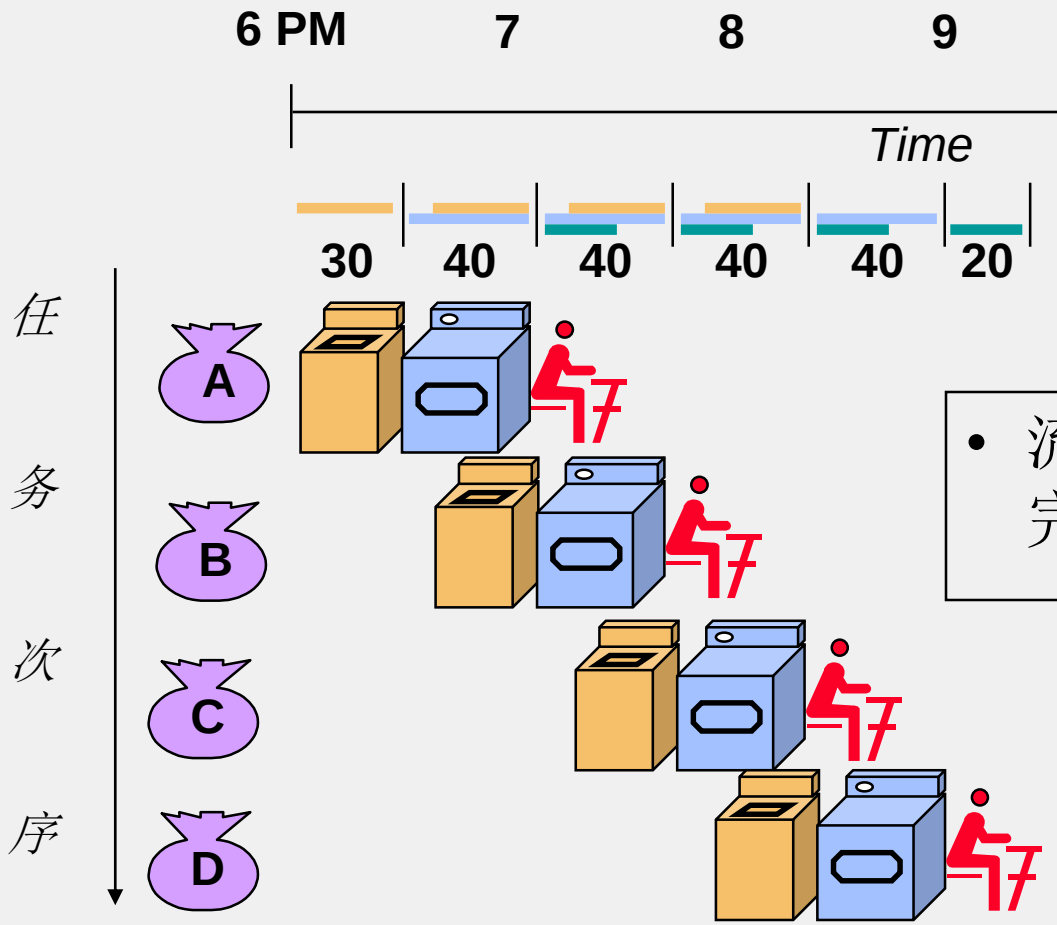
非流水线——串行洗衣



串行洗衣花费 6小时完成4批衣服。

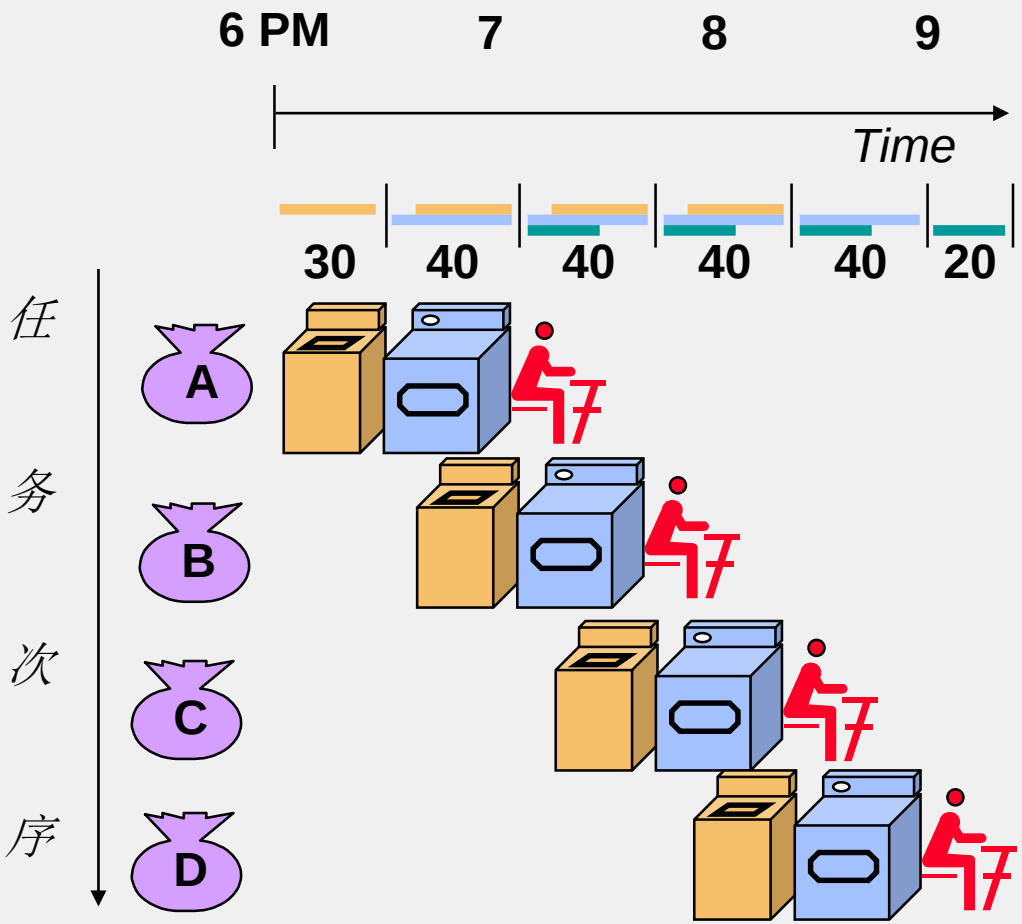
若采用流水线, 完成这4批洗衣要多少时间?

流水线—洗衣的例



- 流水线洗衣花费 3.5小时完成 4批衣服

流水线



流水线特点

- 对单个任务无改善，对整体有改善：吞吐量变大
- 改善被最慢的阶段限制
- 是多任务同时操作
- 加速潜力 = 阶段的数目
- 阶段时间失衡将降低加速

流水线——加法的例（1）

时间	操 作	操作数一	操作数二	结 果
1	取操作数	9.87×10^4	6.54×10^3	
2	比较指数	9.87×10^4	6.54×10^3	
3	移位一个操作数	9.87×10^4	0.654×10^4	
4	相加	9.87×10^4	0.654×10^4	10.524×10^4
5	规格化结果	9.87×10^4	0.654×10^4	1.0524×10^5
6	舍入结果	9.87×10^4	0.654×10^4	1.05×10^5
7	存储结果	9.87×10^4	0.654×10^4	1.05×10^5

两个浮点数相加：

$9.87 \times 10^4 + 6.54 \times 10^3$

如果每个操作花费时间 **1**纳秒**(1ns=10^{-9} s)**，那么这个加法需要**7**纳秒**.**

流水线——加法的例(2)

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

- 此循环需要 7000ns
  - 一个加法需要7ns

流水线——加法的例(3)

- 将浮点数加法器划分成7个独立的硬件或者功能单元。第一个单元取两个操作数，第二个比较指数，以此类推。
- 当执行for循环时，首先取出x[0]和y[0]，然后在比较X[0]和y[0]指数时取出x[1]和y[1]，以此类推，见下图。

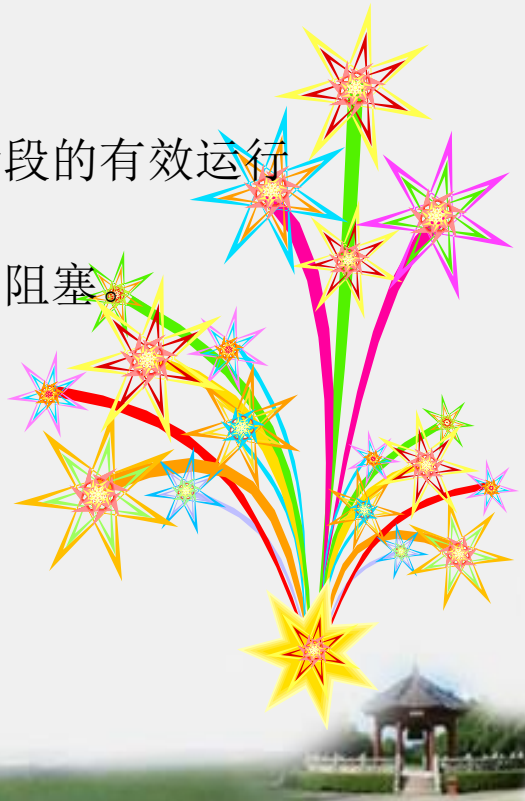
时间	取	比较	移位	加	规格化	舍入	存储
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

表中的数表示操作数或结果数组的下标.

流水线加法

流水线——加法的例(3)

- 一个浮点加法执行时间仍然是7ns。
- 但是，1000个浮点加法仅需要1006ns!
  而不是7000ns，时间几乎1/7。
- k个阶段的流水线可达到k倍的性能提高？
  - 一般来说这是不可能的。
    - 如果各种功能单元的运行时间不同，则每个阶段的有效运行时间取决于最慢的功能单元。
    - 有些延迟（例如等操作数）也会造成流水线的阻塞。



多发射 (1)
(Multiple Issue)

- 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.
  - 假设有两个完整的浮点数加法器，则计算下面循环所需要的时间减半：
    - 当第一个加法器计算Z[0]时，第二个加法器计算Z[1];
      当第一个加法器计算Z[2]时，第二个计算Z[3]; 依次类推。

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

多发射(2)

多发射分为静态和动态两种：

- 静态多发射——如果功能单元是在编译时调度的；
- 动态多发射——如果是在运行时间调度的。

- 一个支持动态多发射的处理器称为超标量（superscalar)

预测(1)
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，然后在猜测的基础上执行代码。

- 下面的代码是一个简单的例子

预测 (2)

```
z = x + y;
if (z > 0)
    w = x;
else
    w = y;
```

系统预测z = x+y的结果z可能为正数，因此执行赋值操作w=x。

如果预测错误，即z的值为负或者为零，需要回退，然后执行w=y。

- 如果预测工作由编译器来做，那么它通常在代码中嵌入测试语句来验证预测的正确性，如果预测错误，就会执行修正操作。
- 假如由硬件做预测操作，处理器一般会将预测执行的结果缓存在一个缓冲器中。
  - 如果预测正确，缓冲器中的内容会传递给寄存器或者内存；
  - 如果预测错误，则缓冲器中的内容被丢弃，指令重新执行。

超标量执行

- 若处理器具有k条流水线，能同时发射k条指令——处理器称为超流水线处理器。
- 同一周期发射多条指令的执行称为超标量执行

超标量流水线

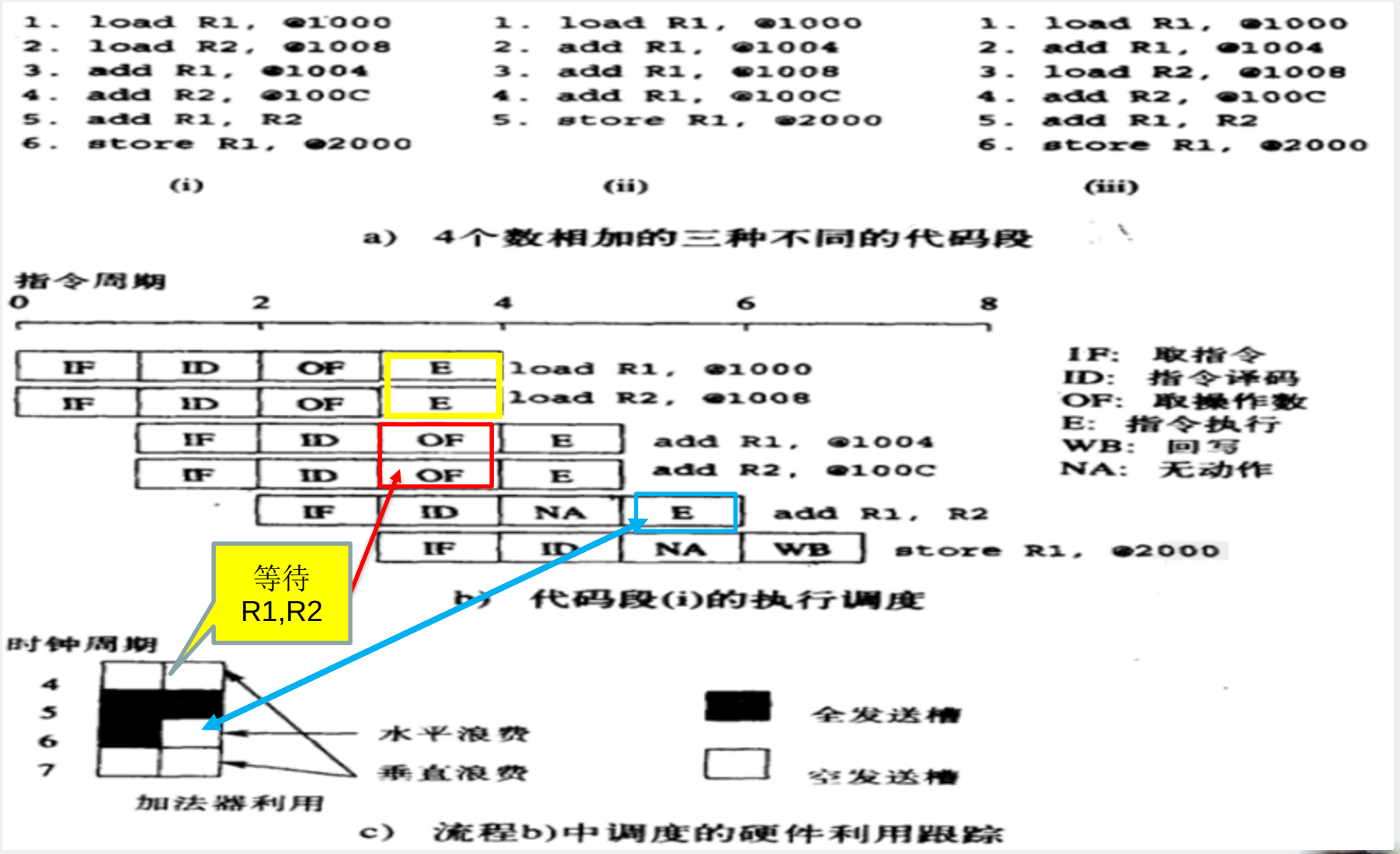
- 超标流水线可看作从标量流水线演化而来
  - 消除了标量流水线的限制。
- 有三个特点：
  - 超标量流水线属于并行流水线
    - 每个时钟周期可以启动多条指令执行。
  - 属于多配置流水线
    - 在执行阶段提供多个不同的功能部件
  - 采用动态流水线技术
    - 通过乱序执行获得较好的的程序执行时间

超标量流水线

动态并行流水线

- 并行流水线中为了最大线地减少不必要的指令停顿，后续指令必须能够绕过停顿的前导指令。而这将会导致指令的原始次序在执行时发生改变。
- 通过指令的乱序执行可克服指令流的相关限制. 从而开发潜在的并行性. 即指令所需的操作数一旦可用，指令就开始执行.
- 支持乱序执行的并行流水线称为动态并行流水线. 动态流水线通过使复杂多记录缓冲来实现乱序执行，并允许指令以可变的顺序进入和离开缓冲.
-

超标量执行：一个例子



两路超标量指令执行示例（假设访存占一个时钟周期）

超标量流水线系统

如何利用它？

- 超标量流水线系统是动态多发射系统
  - 能够乱序执行指令，但现行的系统中，指令是顺序加载的，执行的结果也是顺序提交的。即指令的结果是按程序中规定的顺序写入寄存器和内存中的。
- 已有编译器优化技术能够对指令进行重新排序。

- 编程优化建议：减少分支，即减少乱序执行时的预测错误。

硬件多线程

Hardware multithreading (1)

- There aren't always good opportunities for simultaneous execution of different threads.
- Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory.

Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.
  - Pros: potential to avoid wasted machine time due to stalls.
  - Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

Hardware multithreading (3)

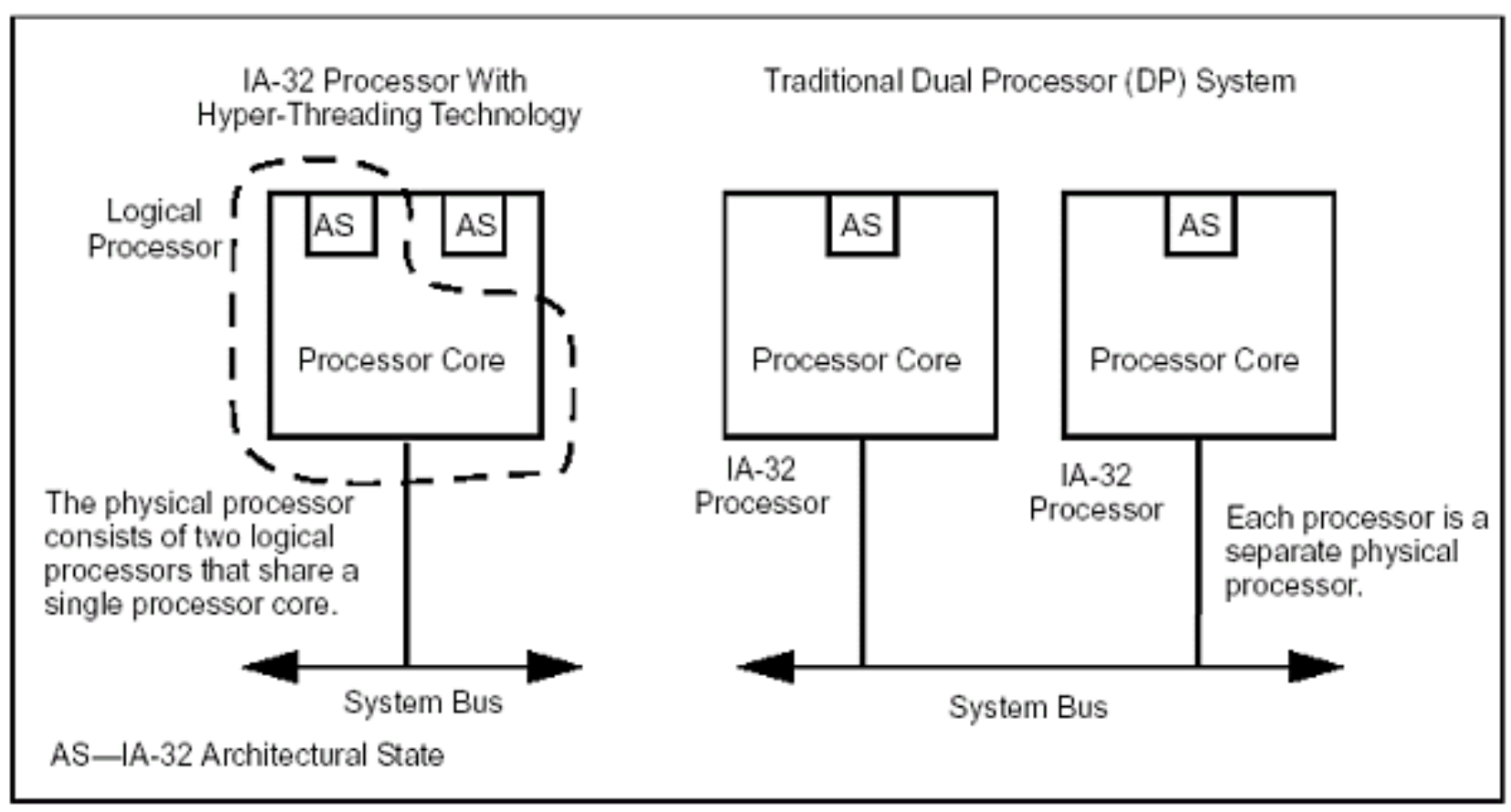
- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.
  - Pros: switching threads doesn't need to be nearly instantaneous.
  - Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

Hardware multithreading (4)

- **Simultaneous multithreading** (SMT) - a variation on fine-grained multithreading.

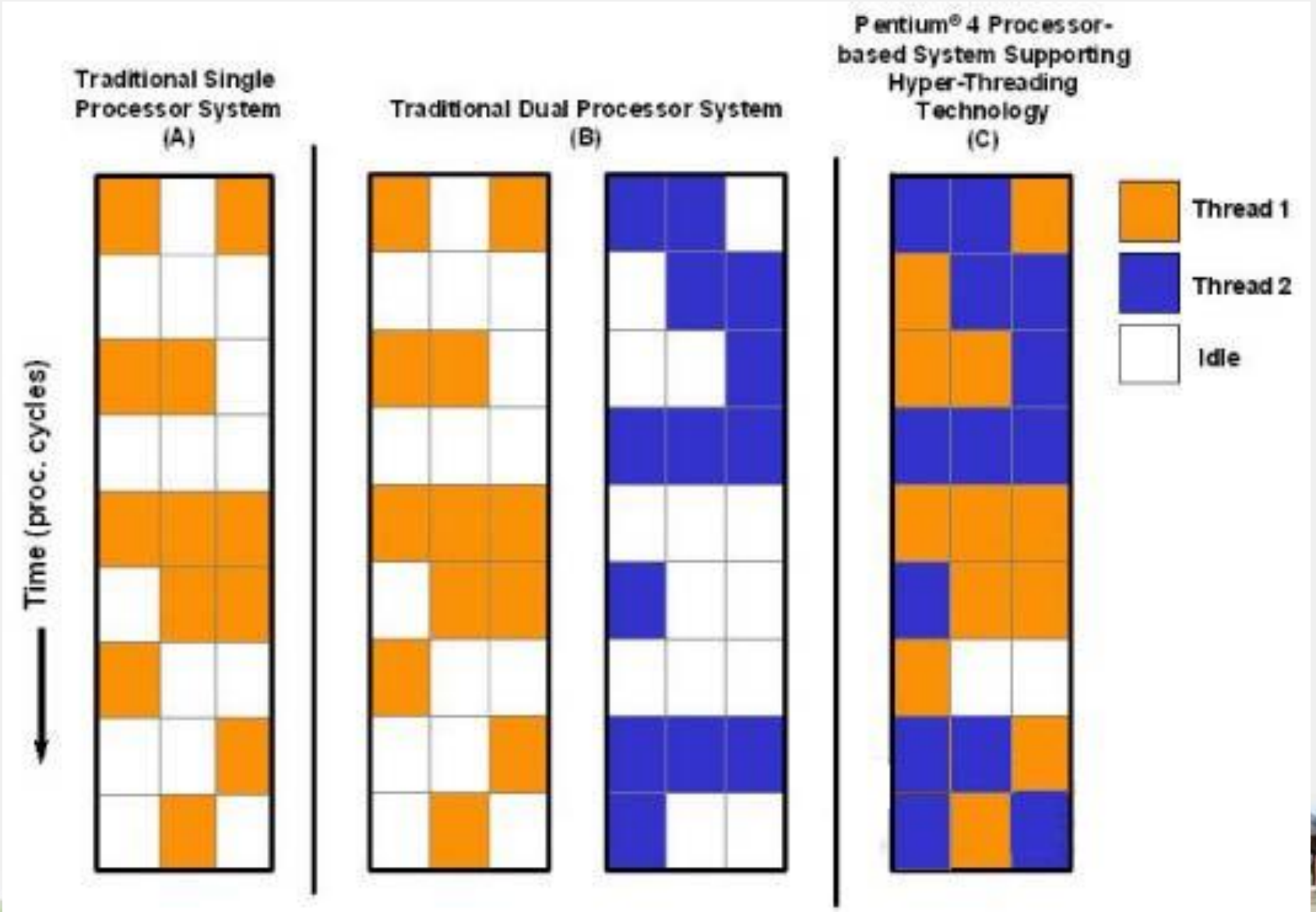
- Allows multiple threads to make use of the multiple functional units.

超线程（HT）
Hyper-threading



超线程（HT）

Hyper-threading



隐藏内存延迟的方法

- 如果在网络传输高峰期浏览网页，对浏览器的响应不足，可以使用下面的三种简单方法之一来缓解：
- 预测要浏览的网页，并提前发送请求；
- 打开多个浏览器，在每个浏览器中访问不同的网页，这样当等待某个页面载入时，可以先看其他的页面；
- 一次访问多个页面——在多个访问中分摊延迟。
- 第一种方法称为预取，第二种方法称为多线程，第三种方法与访问内存时的空间局部性有关。

多线程隐藏延迟

- 线程是程序流程中的单一控制流。下面是一个解释线程的简单例子：

```
for (i = 0; i < n; i++)
        c [i] = dot_product(get_row(a, i), b);
```

- 由于每一个点积是相互独立的，可以表示为并发的执行单位。可以安全地改写代码如下：

```
for (i = 0; i < n; i++)
        c [i] = create_thread（dot_product, get_row(a, i), b);
```

多线程隐藏延迟：例子

- 在代码中，函数dot_product第一次执行的实例要访问两个向量元素并等待它们。
- 同时，函数第二次执行的实例能在下一个周期内访问另两个向量元素并等待它们，依次类推。
- 经过l时间单位后（l指内存系统的延迟），第一次执行的实例从内存中得到了所需的数据，并能执行需要的计算。
- 在下一周期，下一个执行的实例所需数据到达，依次类推。在每个时钟周期我们都能进行一次运算。

多线程隐藏延迟

- 前一例子中的执行调度是基于两个假设：内存系统能够为多个已提出的请求服务，并且处理器能够在每个周期内切换线程。
- 此外，它还要求程序能够以线程的形式显式指定并发。
- 像HEP以及Tera这样依赖多线程的计算机，它们能够在每一个周期切换执行的上下文。因此它们能有效隐藏延迟。

用预取隐藏延迟

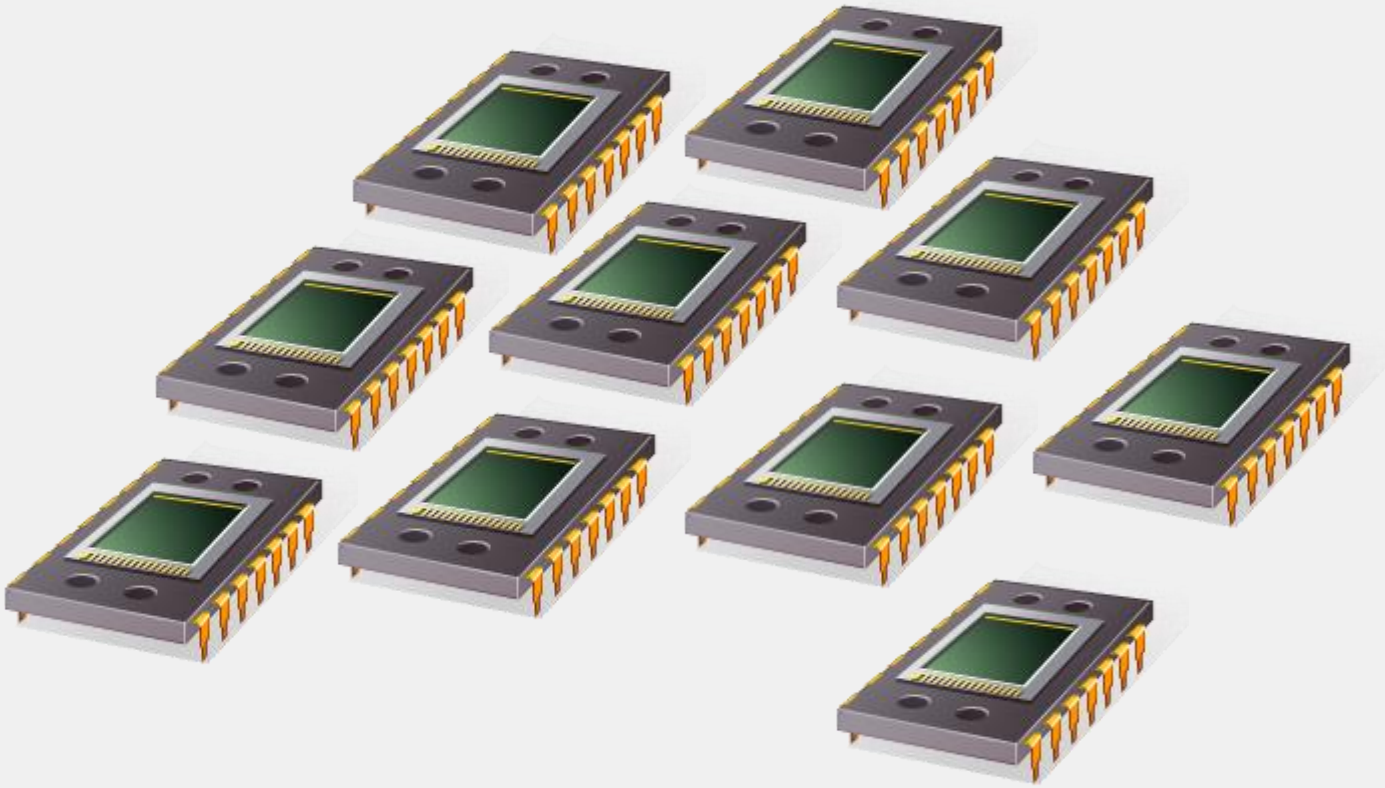
- 数据载入时高速缓存不命中会导致程序的停顿。
- 为什么不提前进行数据载入以便于数据真正需要的时候，它已经在那里！
- 这个方法的唯一缺点是你可能需要更多的空间去存储提前载入的数据。
- 而且，如果预取数据在使用前原数据就被修改了，我们必须重新发出载入命令。但是这种情况并不比没有预取的情况差。

多线程与预取的使用权衡

- 多线程和预取都会受到内存带宽的极大影响。考虑以下例子：
- 假设某一计算机，具有1GHz时钟频率的处理器，4个字的高速缓存行，访问高速缓存需要一个周期，DRAM的延迟为100纳秒。该计算机进行计算时访问1KB高速缓存的命中率为25%，访问32KB的命中率为90%。考虑两种情况：第一，单一线程执行，整个高速缓存都可用来存放串行的上下文；第二，32线程执行，每个线程有1KB的高速缓存驻留。
- 如果计算在每1纳秒的周期都有一个数据请求，你可能注意到第一种情况需要400MB/s的内存带宽，第二种情况需要3GB/s。

多线程与预取的使用权衡

- 由于每个线程获得的缓存空间更小，多线程系统的带宽需求可能会非常显著地提升。
- 多线程系统主要是带宽限制而非延迟限制。
- 多线程和预取只是用来解决延迟问题，而它们通常会使带宽问题加剧。
- 多线程和预取也需要更多的存储资源。



并行硬件PARALLEL HARDWARE

指程序员可见、可编程序来开发并行性的硬件.

- 多发射和流水线可以认为是并行硬件。但这种并行性通常对程序员是不可见的、不能直接利用的。

弗林分类法Flynn's Taxonomy

classic von Neumann

SISD

Single instruction stream Single data stream

单指令流单数据流

传统的计算机包含单个**CPU**，它从存储在内存中的程序那里获得指令，并作用于单一的数据流

(SIMD)

Single instruction stream Multiple data stream

单指令流多数据流

单个的指令流作用于多于一个的数据流上。例如有数据4、5和3、2，一个单指令执行两个独立的加法运算：4+5和3+2，就被称为单指令流多数据流。

MISD

Multiple instruction stream Single data stream

多指令流单数据流

用多个指令作用于单个数据流的情况实际上很少见。这种冗余多用于容错系统。

not covered

(MIMD)

Multiple instruction stream Multiple data stream

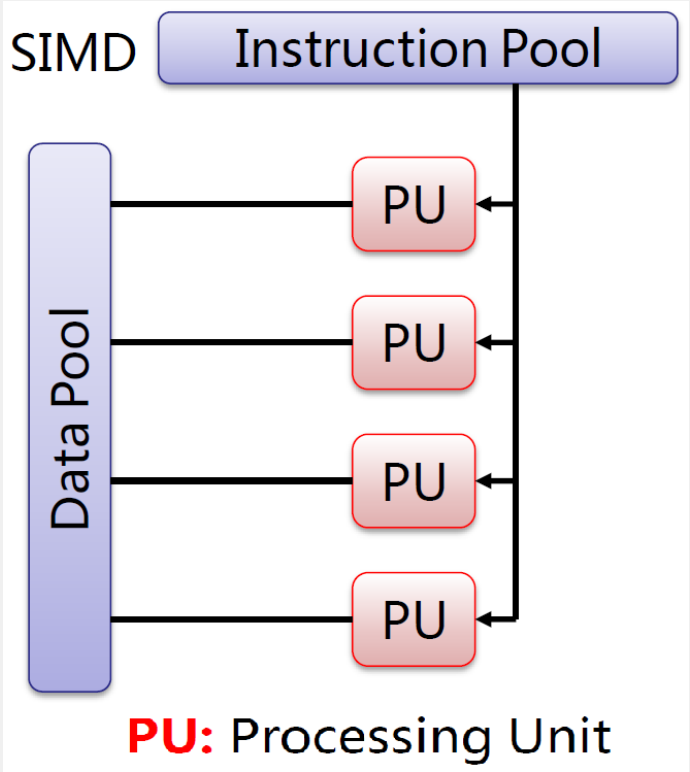
多指令流多数据流

这种系统的一个常见例子是多处理器计算机

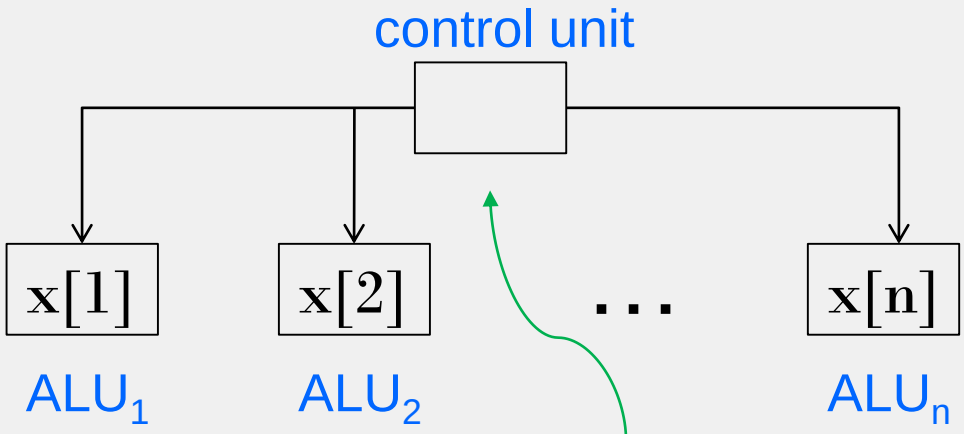
SIMD系统

SIMD系统

- 通过在处理器之间划分数据来实现并行性。
  - 对多个数据项使用相同的指令
  - 称为数据并行



SIMD的例



n 个数据项
n ALUs

SIMD一条指令完成

串行需要循环

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

SIMD系统

- 如果我们没有数据项那么多的ALU怎么办？
  – 划分工作，并循环迭代处理.
- 如：m = 4 ALUs，n = 15 个数据项.

循环变量的值	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD的缺点

- 所有的ALU都需要执行相同的指令，或者保持空闲：
  - for (i = 0;i < n; i++) if (y[i] > 0.0) x[i]+=y[i];
    - ALU都加载了y[i]，并判断y[i]>0。满足的ALU执行加法操作, 不满足的ALU处于空闲状态。
- 在经典设计中，它们在每次操作后必须同步运行。
- ALU没有指令存储器，不能通过存储指令来延迟执行指令。
- 对于大型数据并行问题有效，但对于其他类型更复杂的并行问题无效。

实现的SIMD处理器

- 向量处理器
- 图形处理单元（GPU)：但非纯粹的SIMD
- 主流处理器也有SIMD支持
  - MMX(MultiMedia eXtension)技术
  - 流 SIMD扩展技术(SSE, Streaming SIMD Extensions)
  - 高级向量扩展 (AVX， Advanced Vector Extensions)
  - 3DNow!技术

向量处理器

- 对数据数组或向量进行操作
  - 传统CPU仅对单个数据元素或标量进行操作。
- 特征
  - 向量寄存器(Vector registers).
    - 能够存储向量操作数并同时对其内容进行操作。
  - 向量化和流水化的处理器中的功能单元.
    - 对向量中的每个元素(或对应元素对)需要做同样的操作.
  - 指令是向量指令.
    - 是在向量上操作而不是在标量上操作的指令.
  - 交叉存储器
    - 内存系统由多个内存“bank”组成，每个bank能够独立访问。
    - 将一个向量的元素分布在多个bank中，从而减少或消除加载/存储连续元素的延迟。注意，访问一个bank后，再次访问它就会有时间延迟。
  - 步长式存储器访问和硬件散射/聚集
    - 程序能够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（聚集）和写（散射）

向量处理器的优点

- 速度快、易用。
- 向量编译器
  - 擅长于识别向量化的代码。
  - 能识别出不能向量化的循环而且能提供循环为什么不能向量化的原因。
    - 帮助程序员重新评估代码.
- 高内存带宽，每个加载的数据都会使用.
  - 基于Cache的系统不能完全利用cache行中的每个元素。

向量处理器的缺点

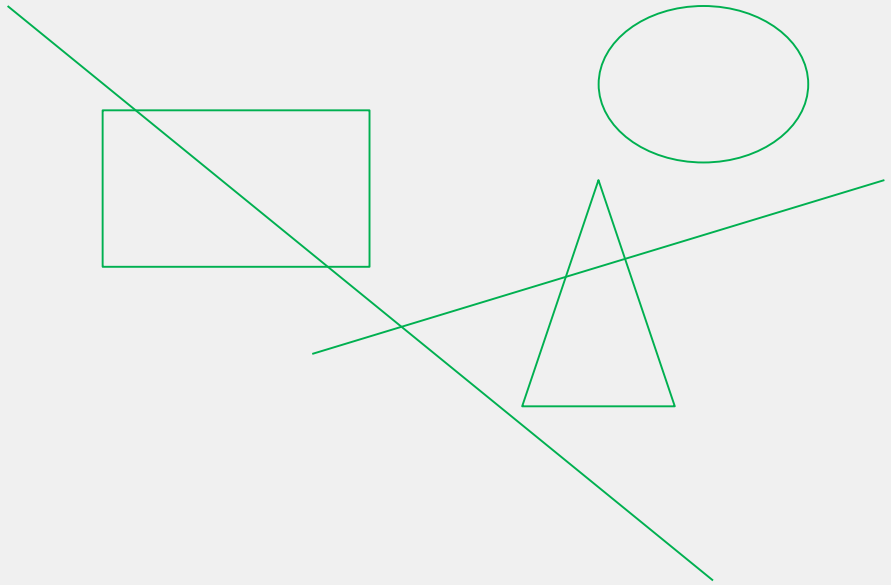
- 它们不能处理不规则的数据结构以及其他并行结构。
  - 对它的可扩展性(scalability)是个限制：可扩展性是指能够 处理更大问题的能力。
    - 造一个能处理长度不断增长的向量系统是困难的。

新一代系统通过增加向量处理器的数目而不是增加向量长度来进行扩展。

图形处理单元（GPU)

Graphics Processing Units

- 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。



GPU系统(1)

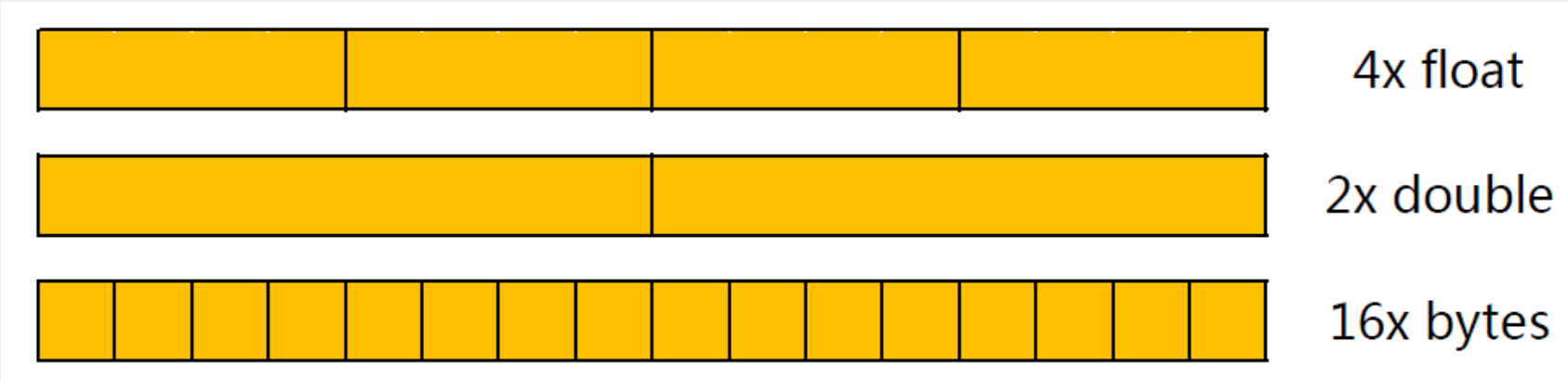
- 使用图形处理流水线(graphics processing pipeline)将物体表面的内部表示转换为像素的数组并在屏幕上显示出来。
  - 流水线的许多阶段是可编程的。
    - 可编程阶段的行为可以通过着色函数(shader function)来说明(通常只有几行C代码)。
    - 着色函数一般是隐式并行的
      - 图形流中的多种元素（例如顶点）上都要用到着色函数

GPU系统（2）

- GPU可以通过使用SIMD并行来优化性能.
- 现在所有的GPU都使用SIMD并行：
  - 通过在每个GPU处理核中引入大量的ALU (例如80个）来获取的并行性能.
    - 虽然GPU不是纯粹的SIMD系统.
- 单个图像数据大（往往数百兆）
  - GPU需要维持很高的数据访问速率
  - 为了避免内存访问带来的延迟，GPU严重依赖硬件多线程
  - 缺点是需要许多处理大量数据的线程来维持ALU的忙碌，对小问题的处理上性能相对差

Intel x86处理器的SSE

- SSE2：Intel的SIMD
  - SSE2数据类型：任务可以放到16字节中的数据。



- 加、乘等指令可以同时操作**16字节**（如向量）。
- 遇到的挑战：
  - ✓ 数据需要指明内存中的存放位置，即事先要拼在一起
  - ✓ 有些结构需要在寄存器间移动

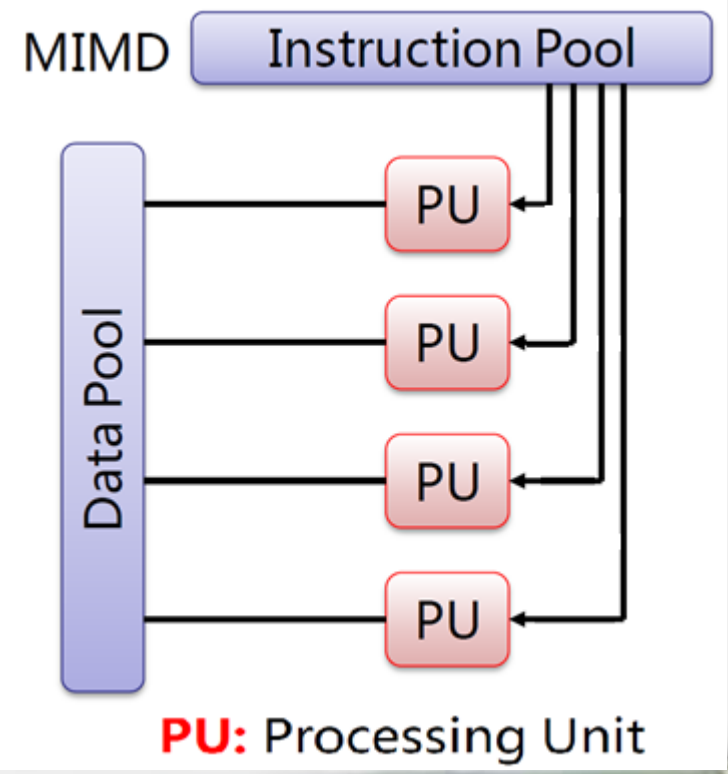
MIMD系统

MIMD系统

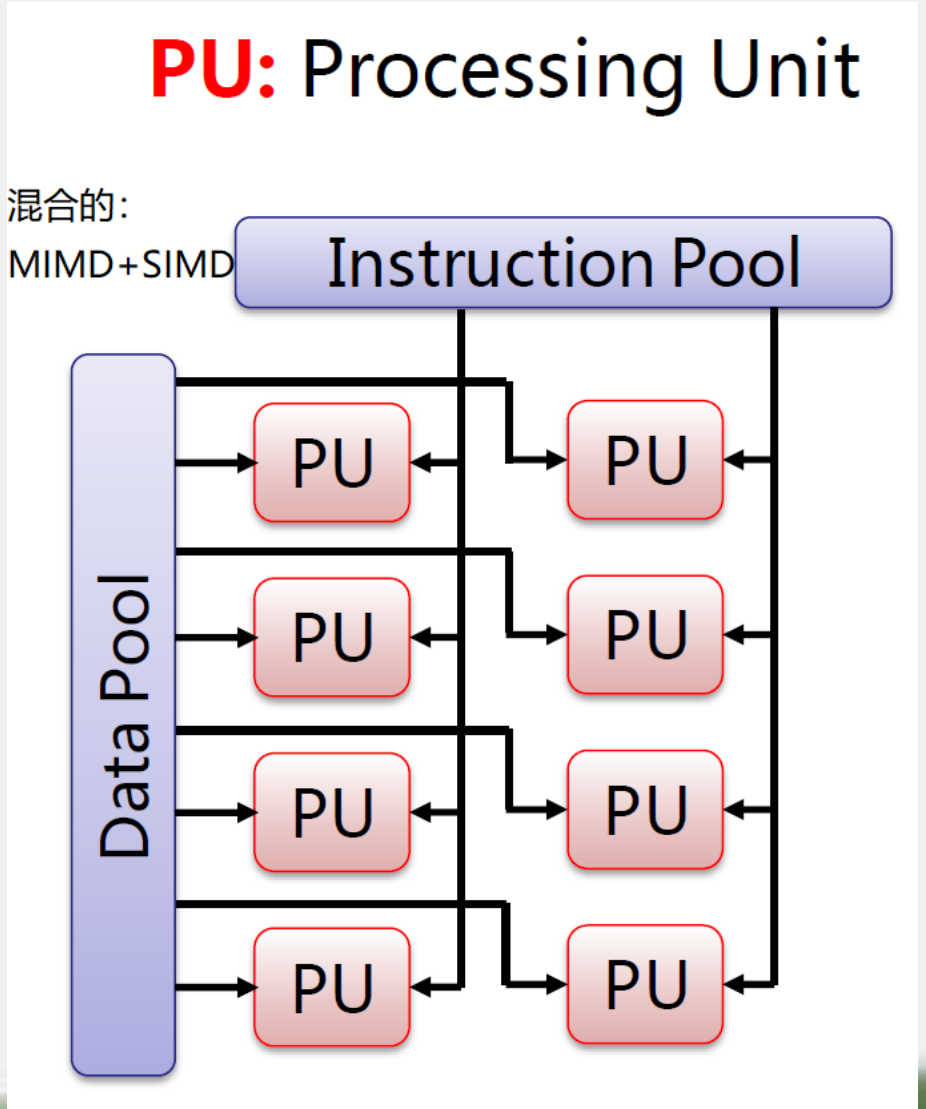
- 支持在多个数据流上同时操作多个指令流.
- 通常由完全独立的处理单元或核心组成，每个处理单元或核心都有自己的控制单元和自己的ALU.

MIMD系统有两种主要的类型：

- 共享内存系统
- 分布式内存系统

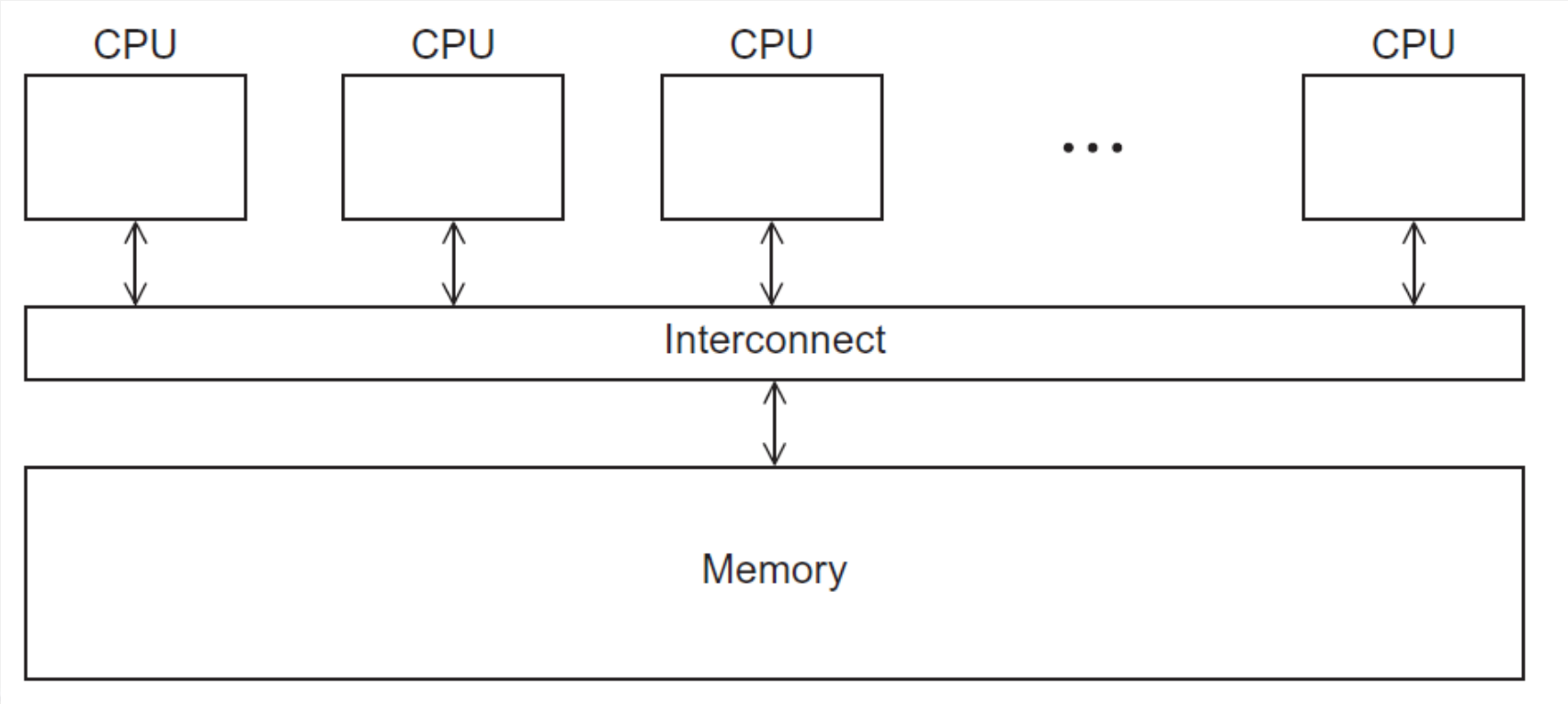


混合MIMD系统（含SIMD）



共享内存系统

- 一组处理器通过互连网络连接到存储器系统。
- 每个处理器可以访问每个内存位置。
- 处理器通常通过访问共享数据结构进行隐式通信。
- 常见的共享内存系统使用一个或多个多核处理器。（单个芯片上的多个CPU或核心）

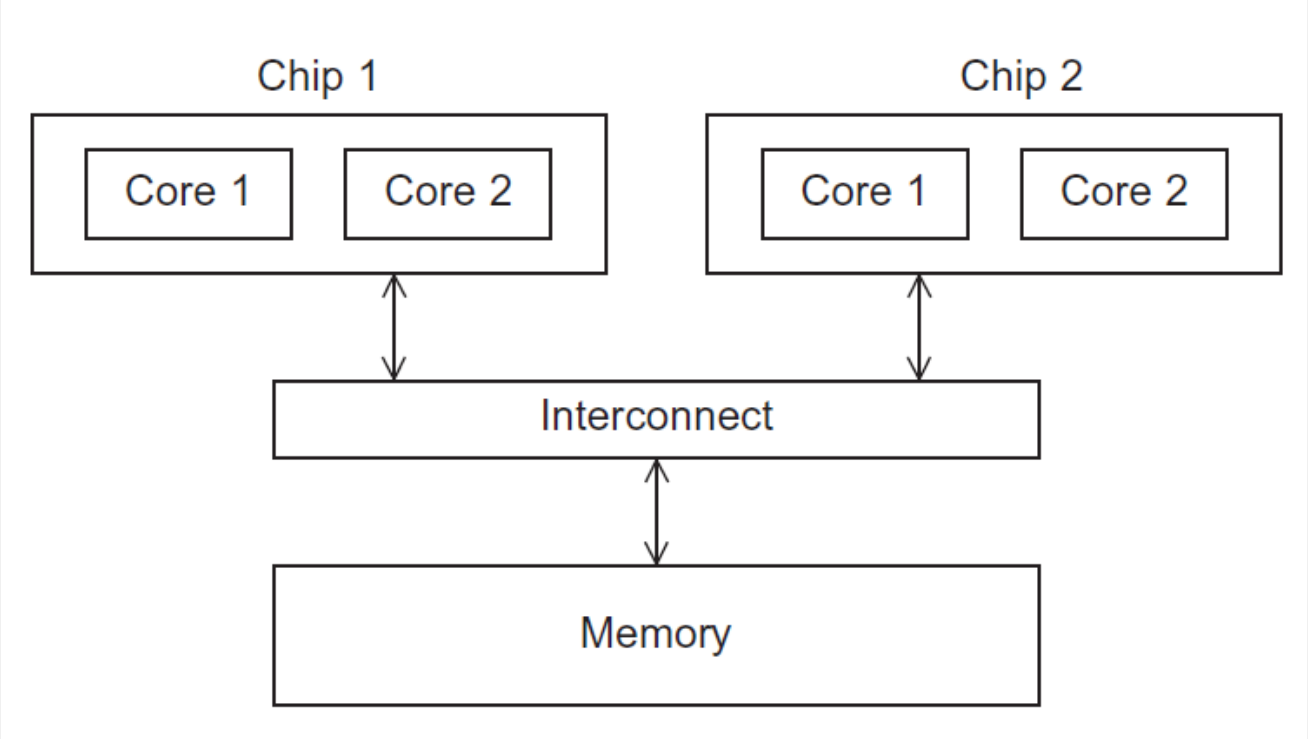


两种共享内存系统

- 一致内存访问（Uniform Memory Access, UMA）多核系统， CPU核访问任何内存区域的速度都相同。
- 非一致内存访问（Nonuniform Memory Access, NUMA)多核系统，CPU核访问不同内存区域的速度不同。

UMA多核系统

一致内存访问（Uniform MemoryAccess, UMA）
——系统中每个核访问内存任何位置的时间都相同



UMA系统中的算法性能分析

- 假设系统有单层存储：内存+CPU，访存速度慢，CPU速度快
- 初始所有数据都在内存中
  - m = 算法需要CPU访问内存元素(字)的数目
  - t_m = 一次访存操作的时间
  - f = 算法需要算术运算的次数
  - t_f = 一次算术运算操作的时间 << t_m
  - q = f / m 平均每次访存可进行算术运算的数目

q：计算强度：

决定算法效率

- 最佳运行时间= f* t_f （这时要求所有数据都在CPU中，按CPU峰值运算的时间）
- 实际运行时间为
  - $f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$

硬件匹配度：决定机器效率

- q 越大，运行时间越接近最佳时间 $f * t_f$
  - $q \geq t_m/t_f$， 则获得至少一半的机器峰值速度
- 单位时间执行的浮点操作数

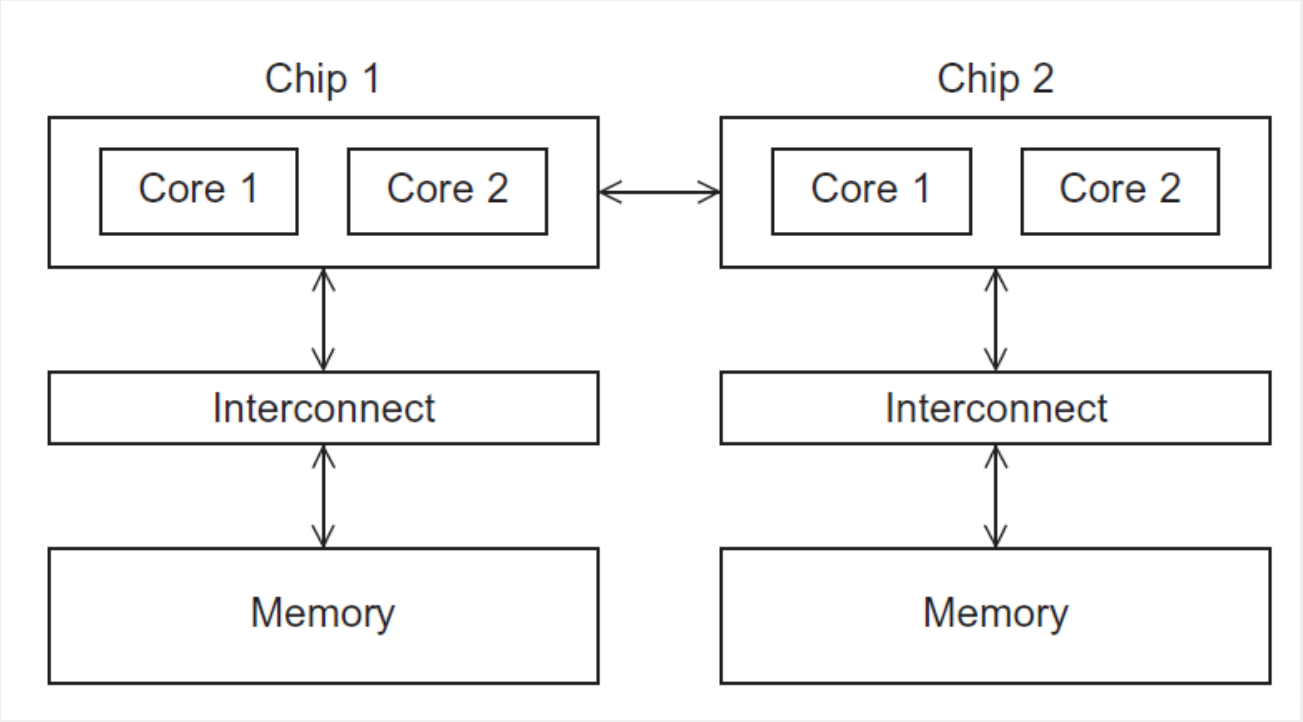
单位时间的#FLO = $f/ (f * t_f * (1 + t_m/t_f * 1/q)) = q/(q*t_f + t_m)$

- **q越大，t_m影响越小**

NUMA多核系统

非一致内存访问（Nonuniform Memory Access, NUMA)。对每个芯片，内存区域分为：

- 芯片直接和不直接连接；
  - ✓ 必须通过其他芯片访问不直接连接区；
  - ✓ 芯片访问其直接连接区域速度较快，不直接的则较慢。



NUMA多核系统的优化

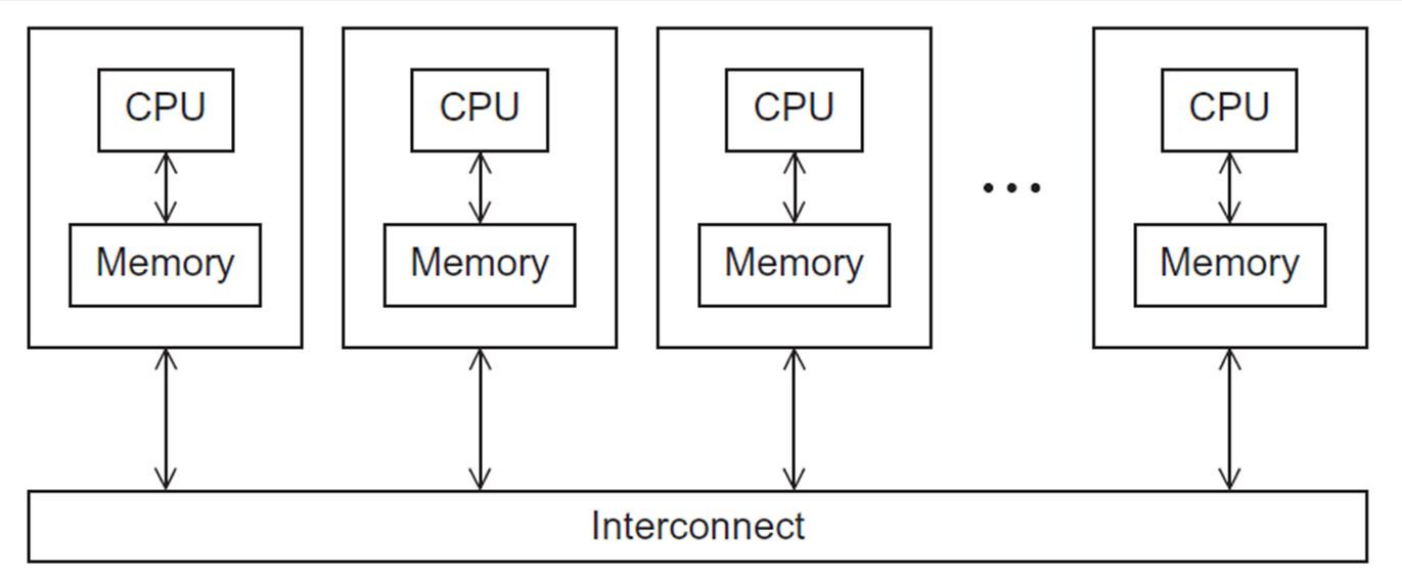
- 通常NUMA分配内存时会先分配离自己近的内存区域
  - 优化要求下面两点
    - 保证分配在离自己近的内存区域
      - 在线程内分配，如线程内使用malloc函数
    - 保证线程不会迁移到另一核心上
      - 通过绑核命令使线程不会在核心间迁移
      - 绑核还有利于L1 cache提高命中率（ L1 是核私有的）
        - » 在GCC环境下的OpenMP可通过环境变量GOMP_CPU_AFFINITY设置实现
        - » Pthread用pthread_setaffinity_np函数实现
        - » Windows用SetThreadAffinityMask函数实现

NUMA系统中的算法性能分析

- 系统有单层存储：内存+CPU，但访存速度不一致
- 初始所有数据都在内存中
  - m_1 = 算法中CPU核访问直接连接内存元素(字)的数目
  - t_{m1} = 一次访问直接连接内存操作的时间
  - m_2 = 算法中CPU核访问不直接连接内存元素(字)的数目
  - t_{m2} = 一次访问不直接连接内存操作的时间
  - f = 算法需要算术运算的次数
  - t_f = 一次算术运算操作的时间 << t_m
  - $q_1 = f / m_1$， $q_2 = f / m_2$：平均每次访存可进行算术运算的数目
- 最佳运行时间= $f^* t_f$
- 实际运行时间为
  - $f * t_f + m_1 * t_{m1} + m_2 * t_{m2} = f * t_f * (1 + t_{m1}/t_f * 1/q_1 + t_{m2}/t_f * 1/q_2)$
- q_1、q_2越大，运行时间越接近最佳时间 $f * t_f$
  - $t_{m2} \geq 2\, t_{m1}$;
  - $q_1 \geq 2t_{m1}/t_f$， $q_2 \geq 2t_{m2}/t_f$，则获得至少一半的机器峰值速度

分布式内存系统

- 集群(Clusters)——最广泛使用
  - 由一组商品化系统组成；通过商品化网络连接.
- 节点(Nodes) ——通过通信网络相互连接的独立计算单元。其独立计算单元通常都是有一个或者多个多核处理器的共享内存系统。
  - 与纯粹的分布式内存系统不同，这种系统通常称为混合系统。
  - 通常认为一个集群有多个共享内存节点。



分布式内存系统算法性能分析

- 系统由互联网络+内存+CPU组成，访存和网络传输速度不同
- 初始所有数据都在内存中
  - m = 算法中CPU核访问内存元素(字)的数目
  - t_m = 一次访问内存操作的时间
  - c = 算法中通过互联网络通信的元素(字)数目
  - t_c = 一次通信操作的时间
  - f = 算法需要算术运算的次数
  - t_f = 一次算术运算操作的时间 << t_m
  - q = f / m ，q_c = f / c：平均每次访存/通信可进行算术运算的数目
    - 有时也称q为本地 q ， q_c为远程q
- 最佳运行时间= f* t_f
- 实际运行时间为
  - $f * t_f + m * t_m + c * t_c = f * t_f * (1 + t_m/t_f * 1/q + t_c/t_f * 1/q_c)$
- q、q_c越大，运行时间越接近最佳时间 f * t_f
  - $q \geq 2t_m/t_f$ ， $q_c \geq 2t_c/t_f$ ，则获得至少一半的机器峰值速度

SPMD模型

- MIMD模型的一个变体称为单程序多数据模型（SPMD），用同一程序的多个实例在不同数据上执行。
  - 容易看出SPMD模型与MIMD模型在编程灵活性以及底层架构支持方面是密切相关的。
  - 这样的平台包括多处理器PC、集群等等。

SIMD与SPMD比较

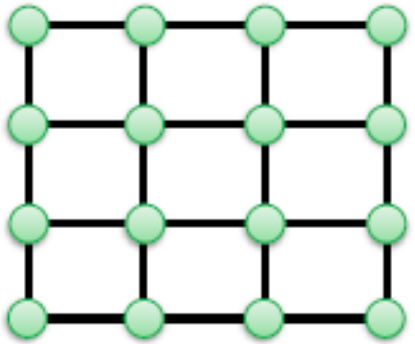
- SIMD和SPMD都是可以处理多数据的
- 不同点是：SIMD是多个数据执行相同的操作，SPMD是多个数据可以执行不同的操作也可以执行相同的操作。
  - SIMD是从指令级上看的，这意味着SIMD处理的多数据是执行相同的操作，比如都执行加法。
  - SPMD是从程序级上看的，这意味着处理的多数据不一定是执行相同的操作，因为程序里面可以有分支等，即执行路径可以是多条。

SIMD与MIMD比较

- SIMD计算机比MIMD计算机需要的硬件更少（单个控制单元）。
- SIMD处理器需要专门设计。
- 不是所有应用本质上适合使用SIMD处理器。
- 支持SPMD模式的平台却可以用现成的元件以及相对少的投入很快制造出来。

并行计算机的互连网络

互连网络评价参数



- **结点度**：与结点相连接的边数。
- **链路的长度**：链路中包含的边数。
- **距离**：两个结点之间最短的链路的长度。
- **网络直径**：
  max{网络中两个结点之间的距离}。
- **网络规模**：网络中的结点数。
- **等分宽度**：网络被切成节点数相等的两部分时(节点数为奇数，两半差1)，沿切口的最小边数。
- **对称性**：若从任何结点看，网络的拓扑结构都一样，则该网络称为对称的。

互连网络评价参数

- **链路的长度**：链路中包含的边数。
- **距离**：两个结点之间最短的链路的长度。

——与无向图中的定义一样

互连网络评价参数

- **网络直径：**

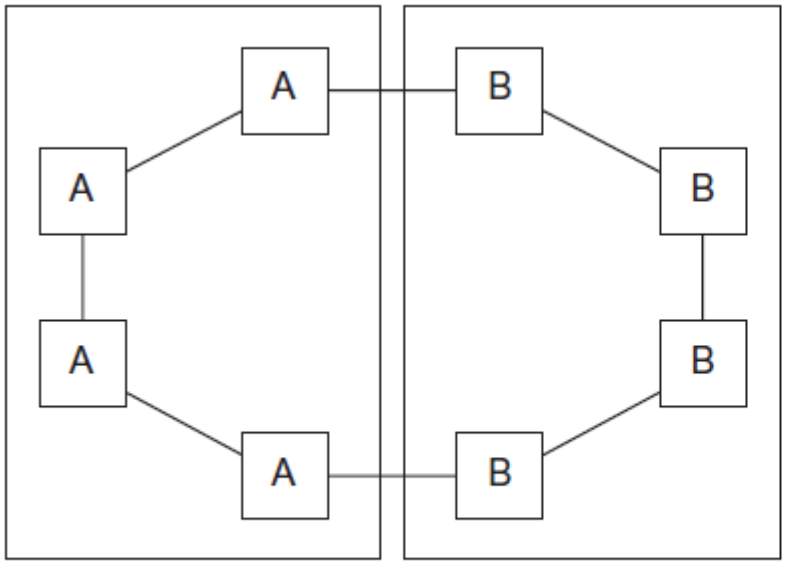
  max{网络中两个结点之间的距离}。

互连网络评价参数

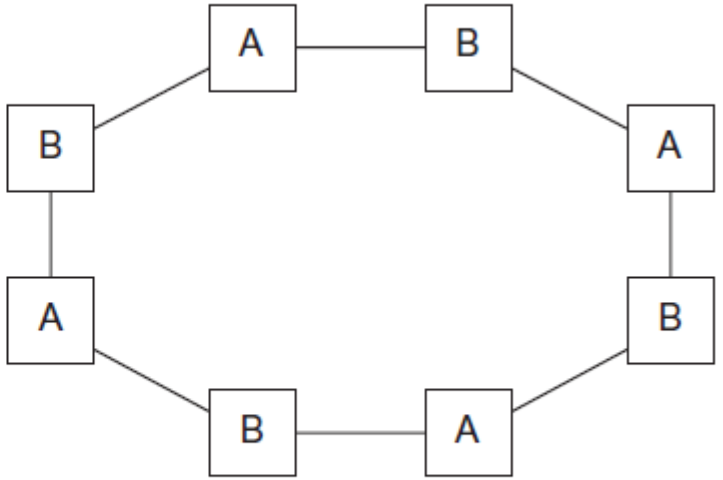
- **等分宽度**：网络被切成节点数相等的两部分时(节点数为奇数，两半差1)，沿切口的最小边数。
  - 衡量“同时通信的链路数目”或者“连接性”的一个标准：想象并行系统被分成两部分，每部分都有一半的处理器或者节点。等分宽度是在所有等分的两部份之间能同时发生的通信次数中最小的。
  - 即切断最少的链路数从而将节点分成两等份，这切断的链路数就是等分宽度。
  - 对分宽度反映网络的瓶颈(最差情况)的拥塞程度，可以作为网络的代价标准。

等分宽度

8节点的环分成两组，每组有四个节点。下面是两种分法：



(a)能够同时通信的次数为2



(b)能够同时发生4次通信

- 等分宽度是取最小的2而不是4。

互连网络

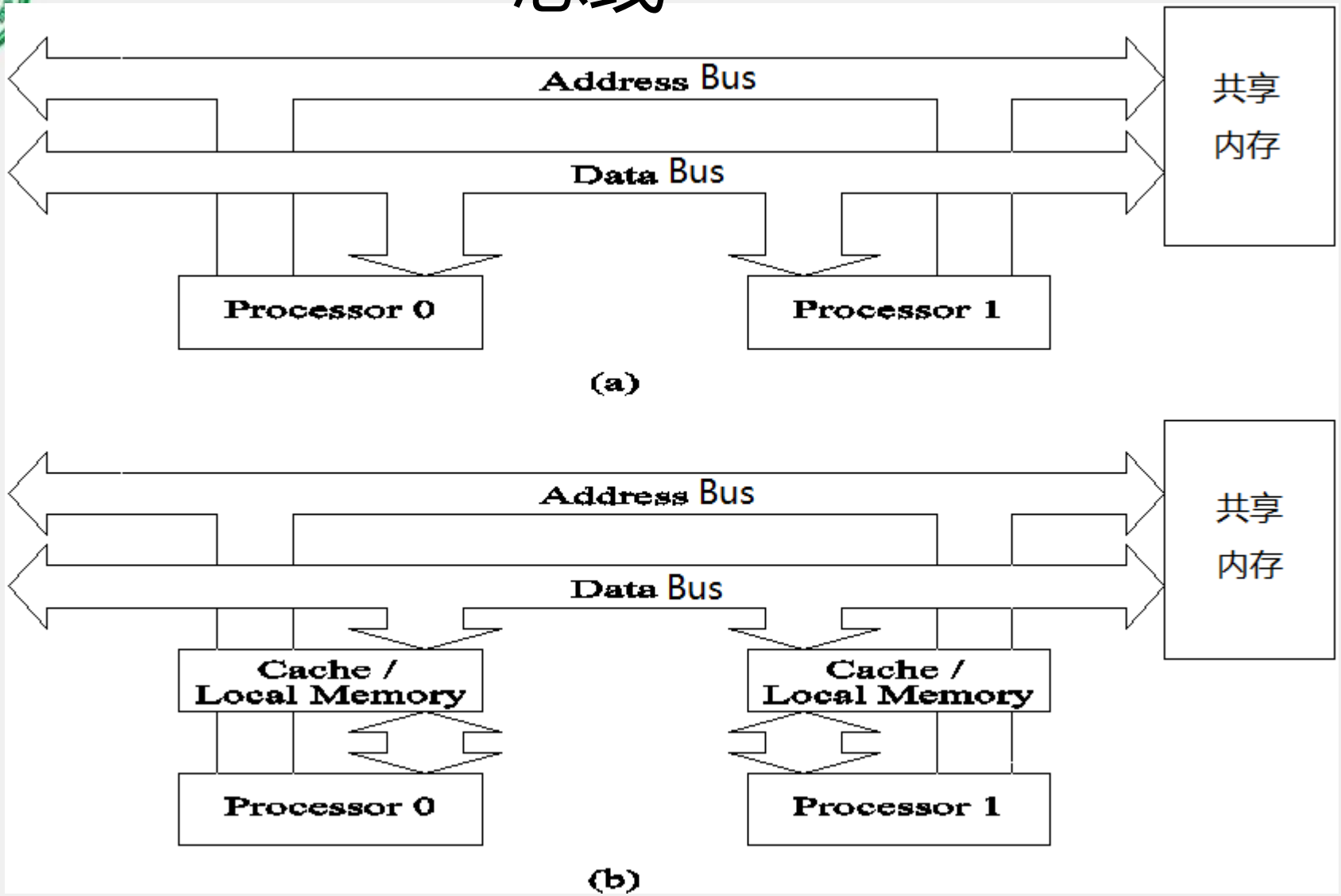
- 对系统性能的影响大。

- 分两类:
  - 共享内存的互联网络
  - 分布式内存的互联网络

共享内存的互联网络

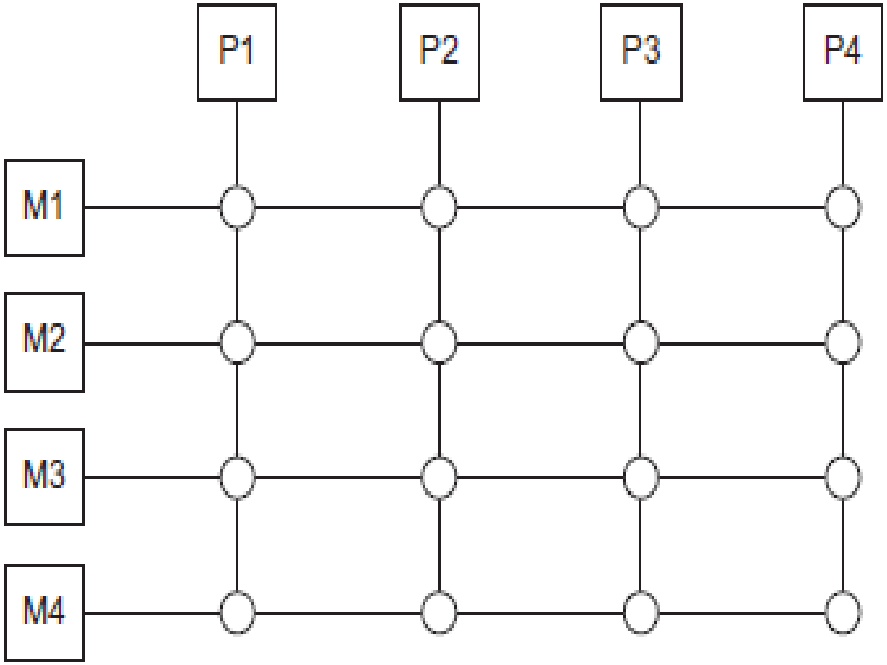
- 总线互联
  - 一组并行通信线和控制对总线访问的硬件组成的.
  - 连接到总线上的设备共享通信线
    - 总线设备的增多，会争夺总线，预期性能会下降。
- 交换互连
  - 使用交叉开关矩阵(crossbar)
  - 允许不同设备之间同时通信
  - 比总线快，但交换机和链路的成本相对较高。

总线

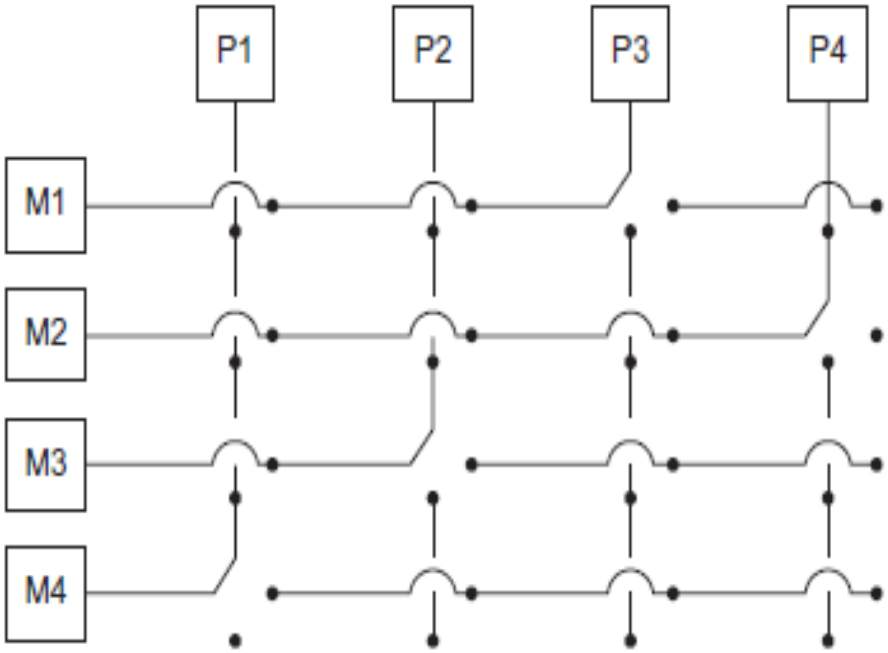


- 基于总线的互连：a）没有本地高速缓存；b）具有本地内存或高速缓存。
- 因为处理器需要访问的数据大部分具有局部性，所以本地内存可以提升基于总线的计算机的性能。

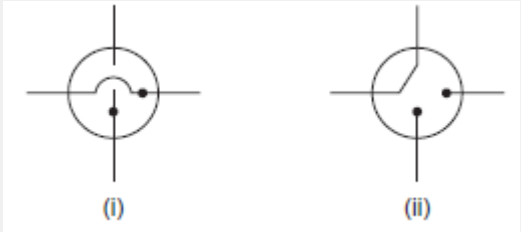
交叉开关矩阵（crossbar）



(a)连接4个处理器（Pi）和4个内存模块（Mj）的交叉开关。



(b)处理器同时访问内存



(c)交叉开关矩阵内部的交换器结构

分布式内存互连网络（1）

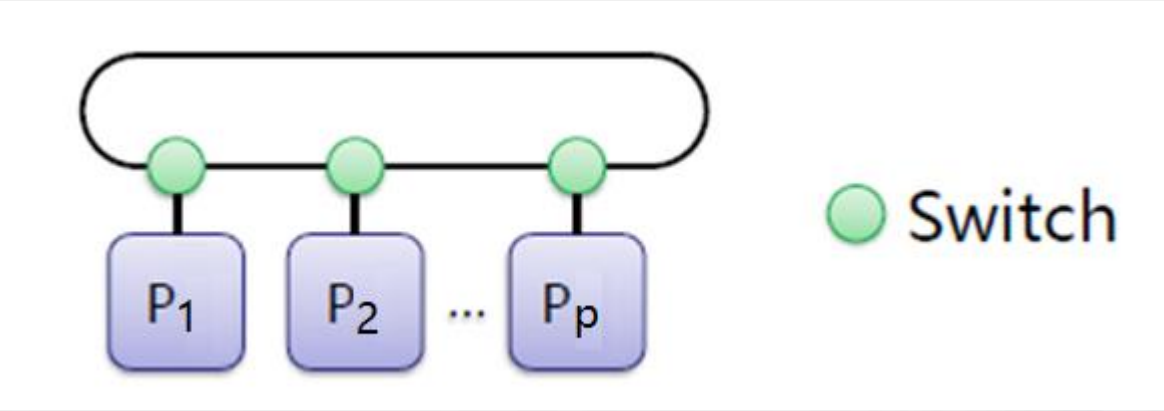
- 分两种
  - 直接互连
    - 一个开关（交换器）对应一对处理器-内存
    - 开关间互联
    - 静态网络
  - 间接互连
    - 处理器-内存统一连接到一个开关网络
    - 由开关网络负责处理器间的互联
    - 动态网络

分布式内存互联网络（2）

- 直接互联网络
  - 环（Ring）
  - 环绕网络（Toroidal Mesh）
  - 超立方（Hypercube）
  - 其他

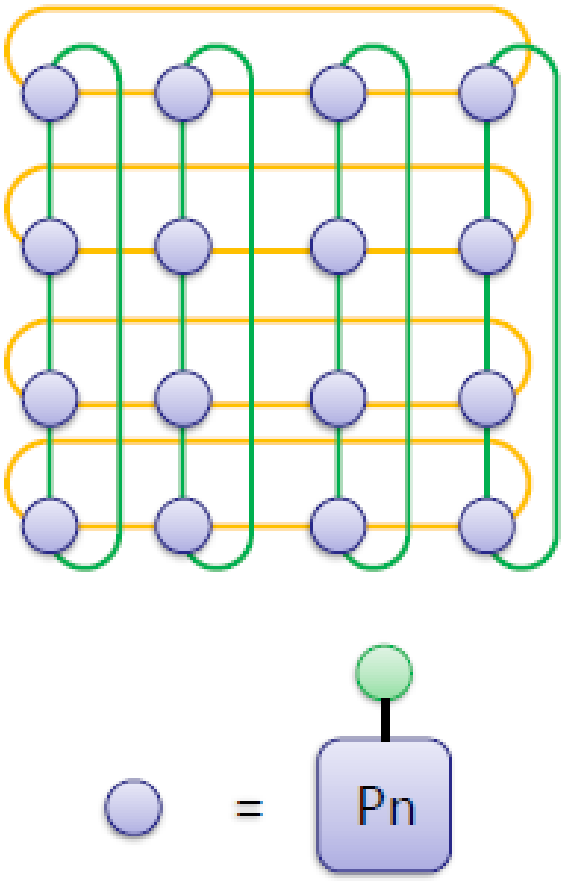
分布式内存的互联网络（3）

- 直接互联网络——环
  - 将一个线性阵列的两端相连就构成一个环。
  - 单向和双向，双向可靠性高
  - 允许有多个通信同时发生：P1和P2,P3和P4
  - 对称
  - 节点数p，单向直径（p-1），双向直径（p/2）
  - 等分宽度为2

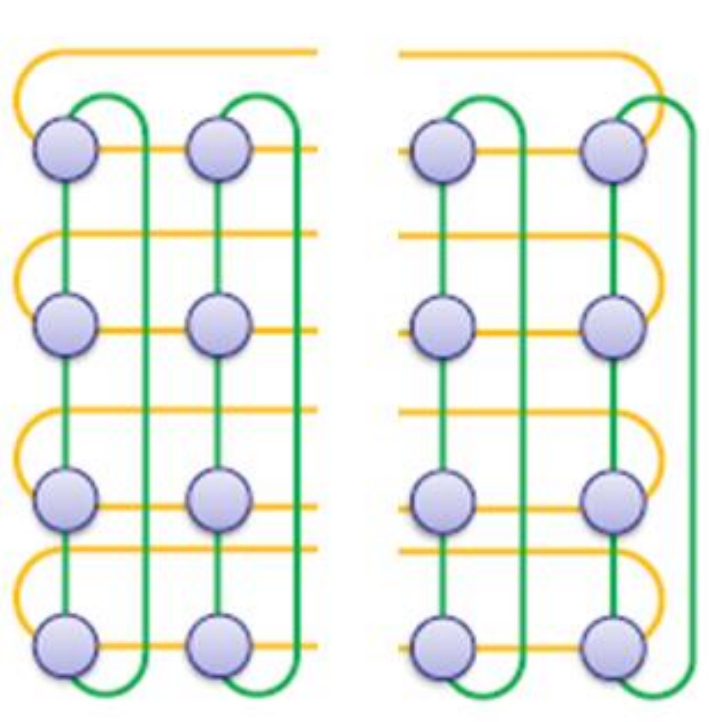


分布式内存的互联网络（4）

- 直接互联网络——环绕网络
  - 如果把一个网格在各维上都环绕连接起来，就构成环绕网格。
  - 对称
  - 节点数p，直径（$\sqrt{p}$），节点度为4
  - 等分宽度为$2\sqrt{p}$



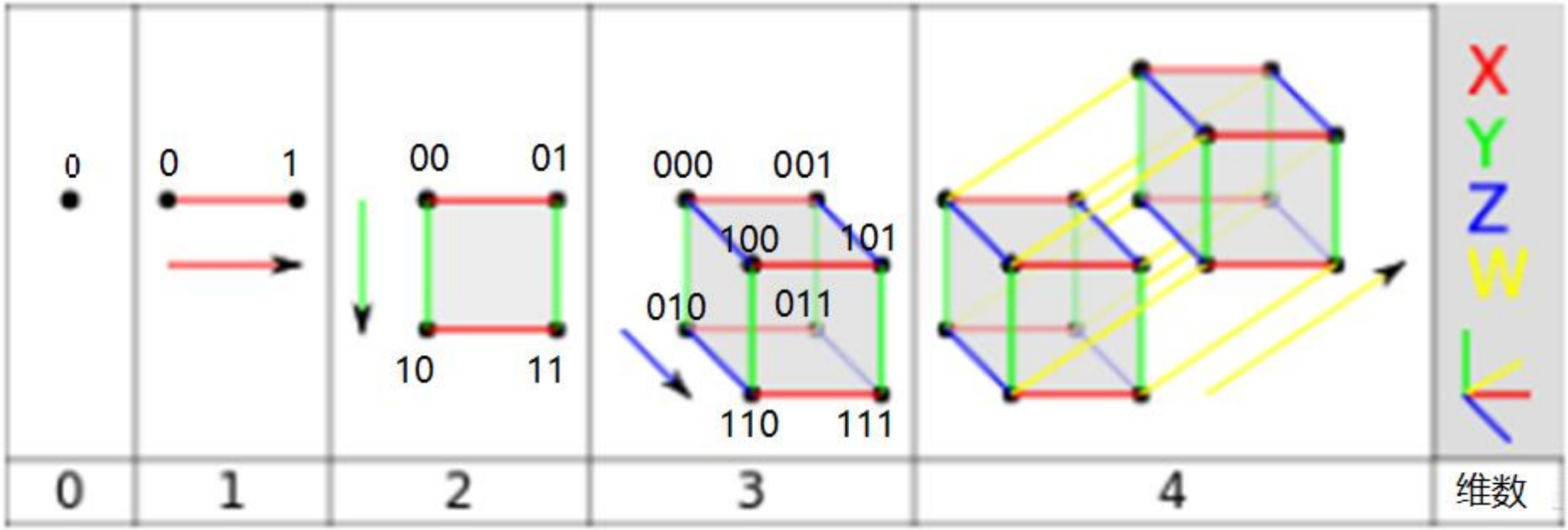
环绕网络的对分宽度



有$p = q^2$个节点（q为偶数）的正方形二维环面网格，通过切断一些“中间”的水平链路和“回绕”的水平链路，将这些节点分成两份。这意味着等分宽度最多是$2q= 2\sqrt{p}$。

分布式内存的互联网络（5）

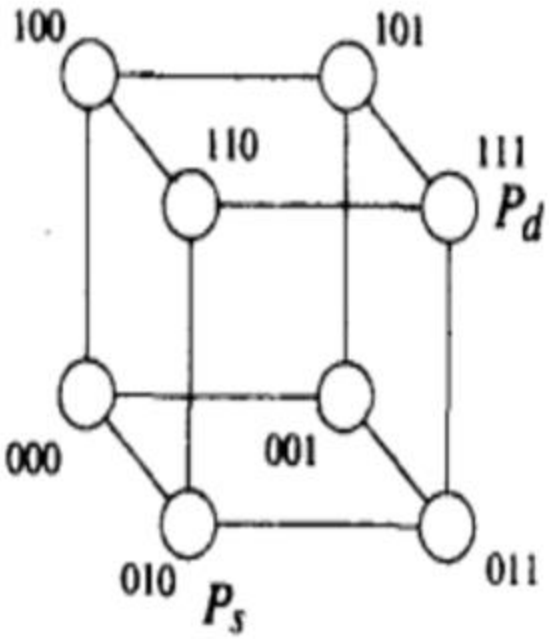
- 直接互联网络——超立方
  - 一个立方体由$p=2^n$个结点组成，它们分布在n维上，每维有两个结点；
  - n维超立方可以通过两个(n-1)维超立方互连得到；

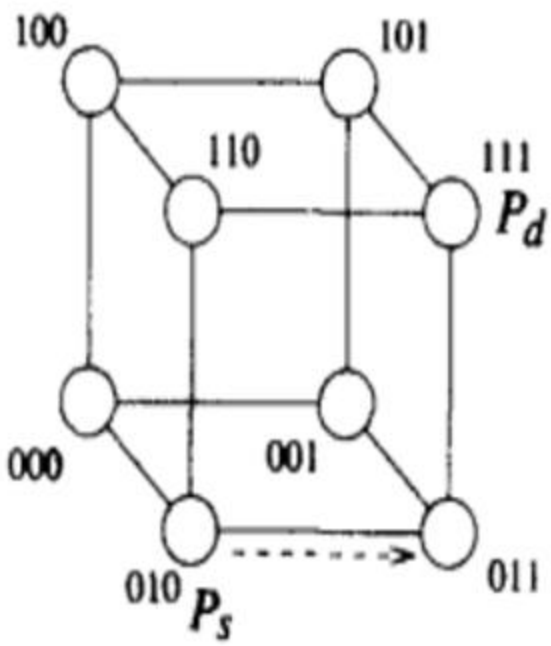


超立方的性质

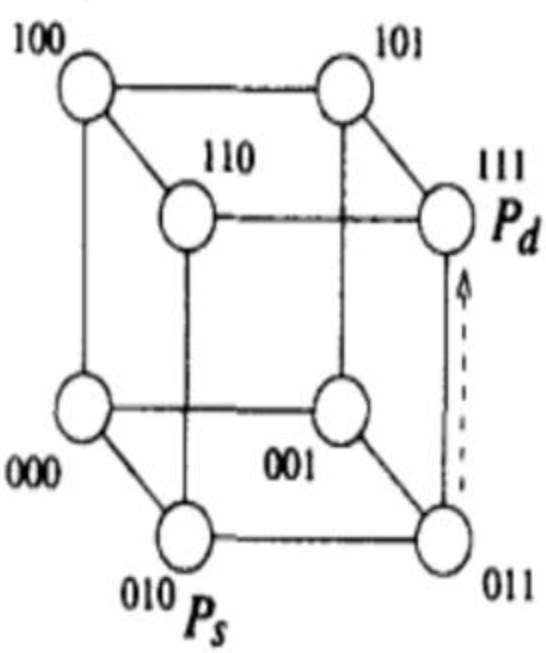
- 一个立方体由$p=2^n$个结点组成，它们分布在n维上，每维有两个结点；
- n维超立方可以通过两个(n-1)维超立方互连得到；
- 一个n维超立方的结点度为n，即每个节点有log p=n个邻居。网络直径也是n，对分宽度为p/2，对称；
- 两节点的距离是由两标号的不相同位的数目决定的：
  - 相邻节点：有1位不同
- 许多其它结构诸如二叉树，网格和许多其它低维网络都能嵌入到超立方体中去；
- 它的结点度随维数线性增加，所以超立方体不是一种可扩展的结构。

超立方的路由选择机制





步骤1 (010 → 011)



步骤2 (011 → 111)

在使用超立方路由选择的三维超立方体中从源节点P_s(010)到目标节点P_d(111)路由消息

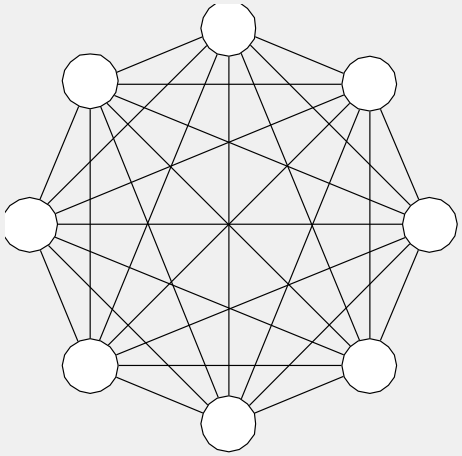
- 先选择与目标有2位相等，与源仅1位不等的点

直接互连网络：全连接网络

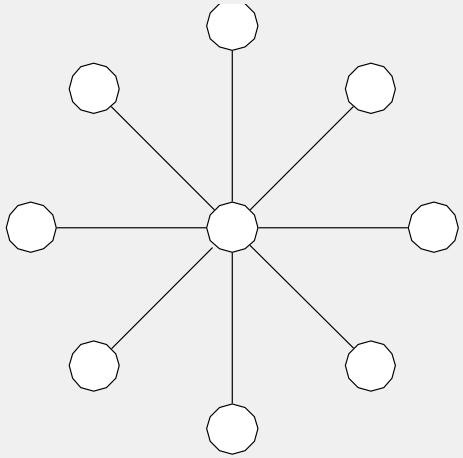
- 每个处理器都连接到其他处理器。
- 网络的链路规模为$O(p^2)$。
- 该网络的性能能够得到很好地扩展，但是对于较大的p硬件复杂度使得该网络不可实现。

直接互连网络：

全连接网络和星形连接网络



(a)

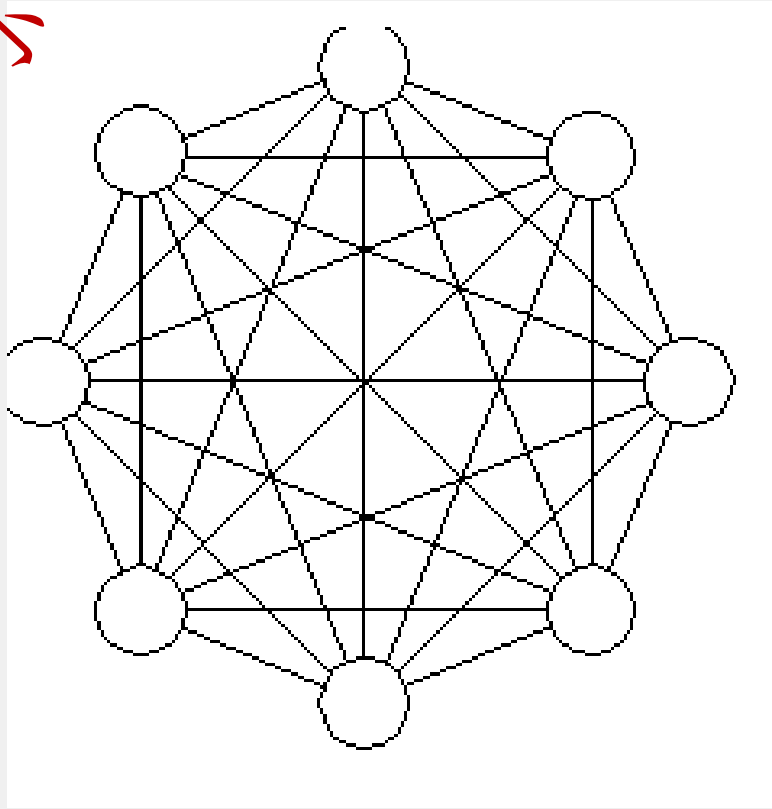


(b)

a）8节点的全连接网络；b）9节点的星形连接网络

全连接网络

不切实际



- 网络中的开关两两互相连接.
- 对分宽度 = $p^2/4$，节点度=p-1，直径为1

星形连接网络

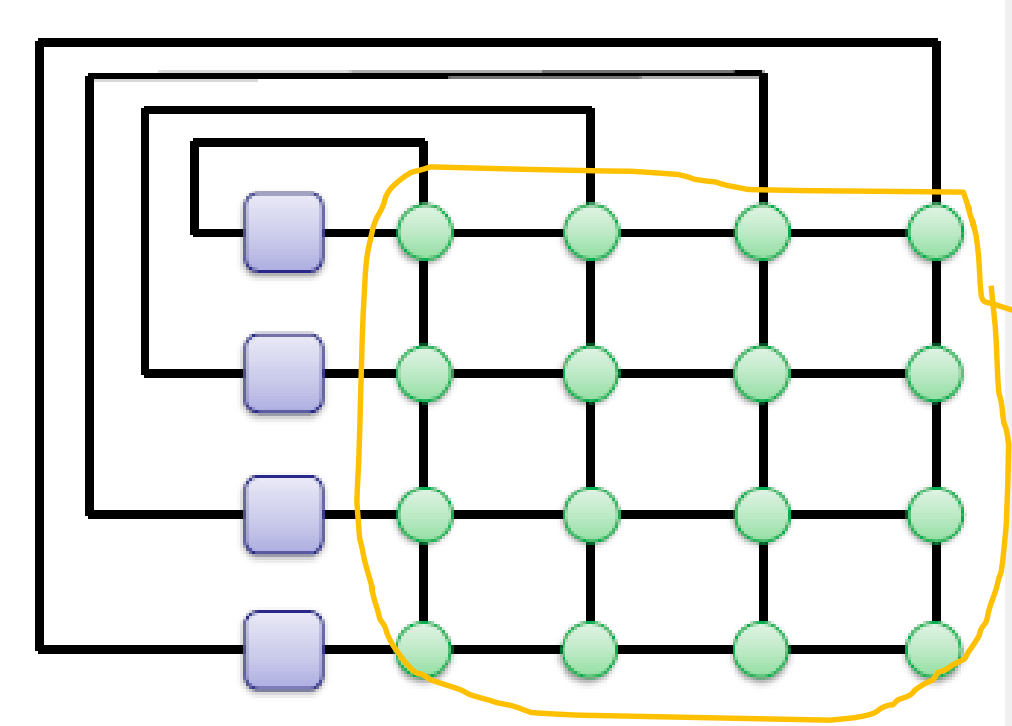
- 每个节点只连接到作为中心的公共节点。
- 任意一对节点的距离为*O(1)*。然而，中心节点称为了性能瓶颈。
- 星形连接网络与总线网络类似。

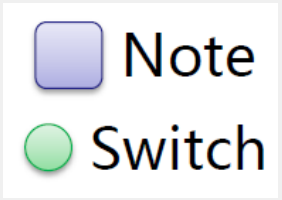
分布式内存的互联网络（6）

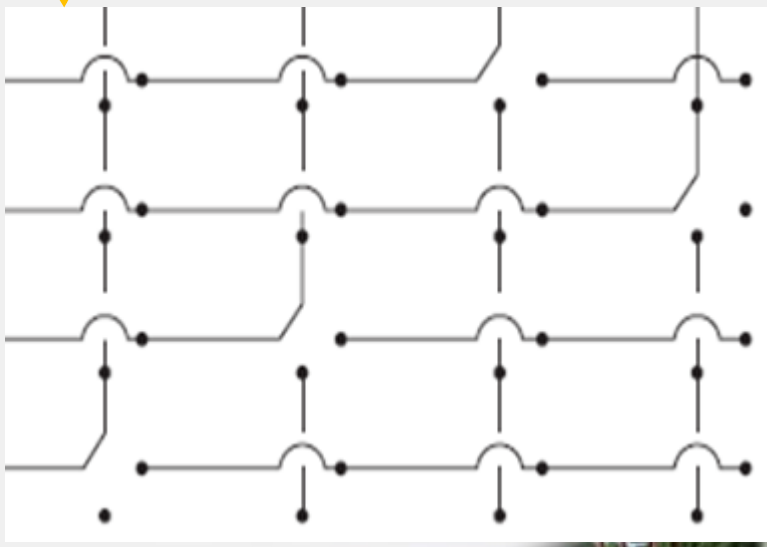
- 间接互联网络
  - 交叉开关矩阵（Crossbars）
  - 多级网络
    - Ω 网络（Omega Network）

分布式内存的互联网络（7）

- 间接互联网络——交叉开关矩阵



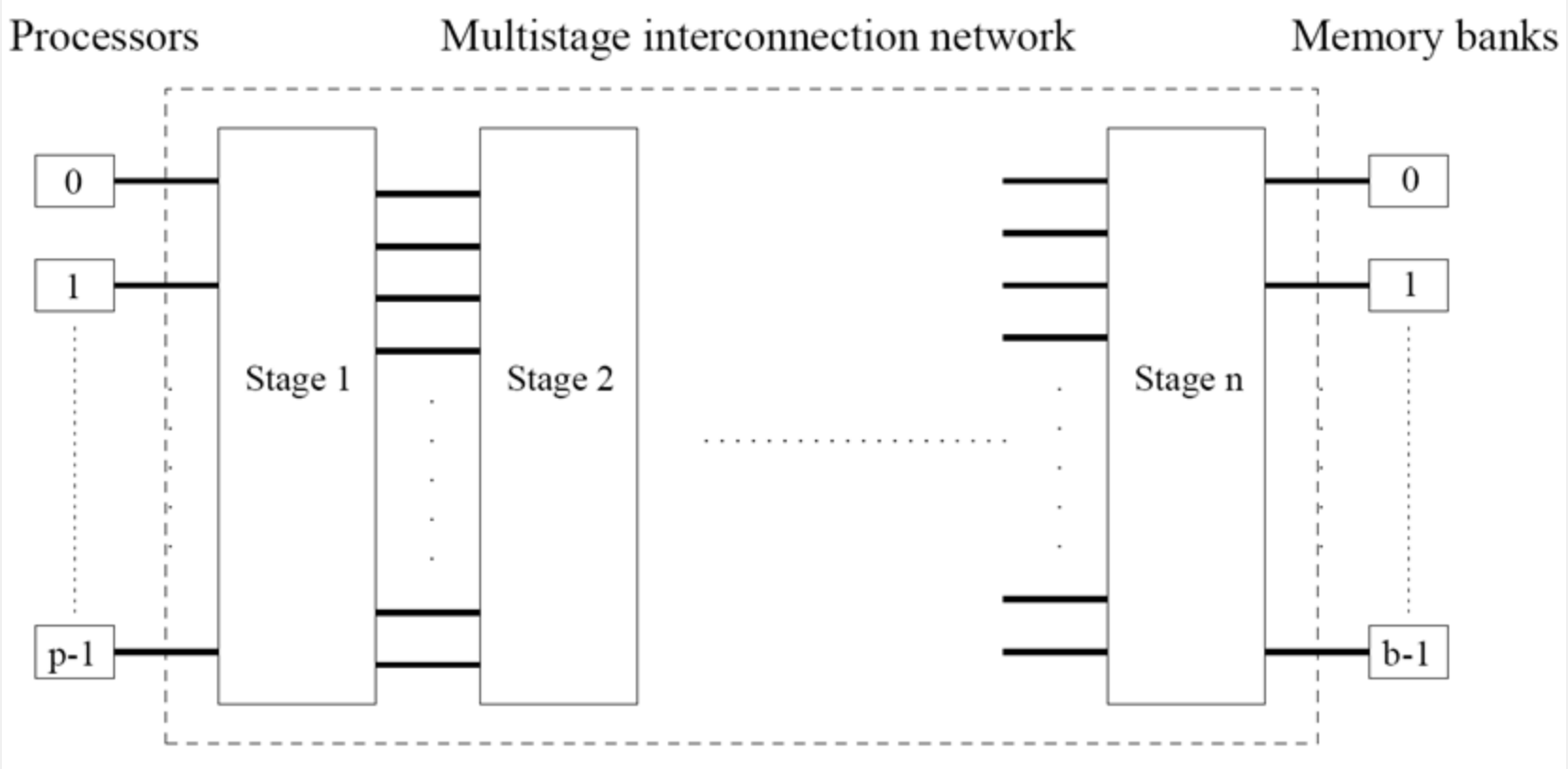




分布式内存的互联网络（8）

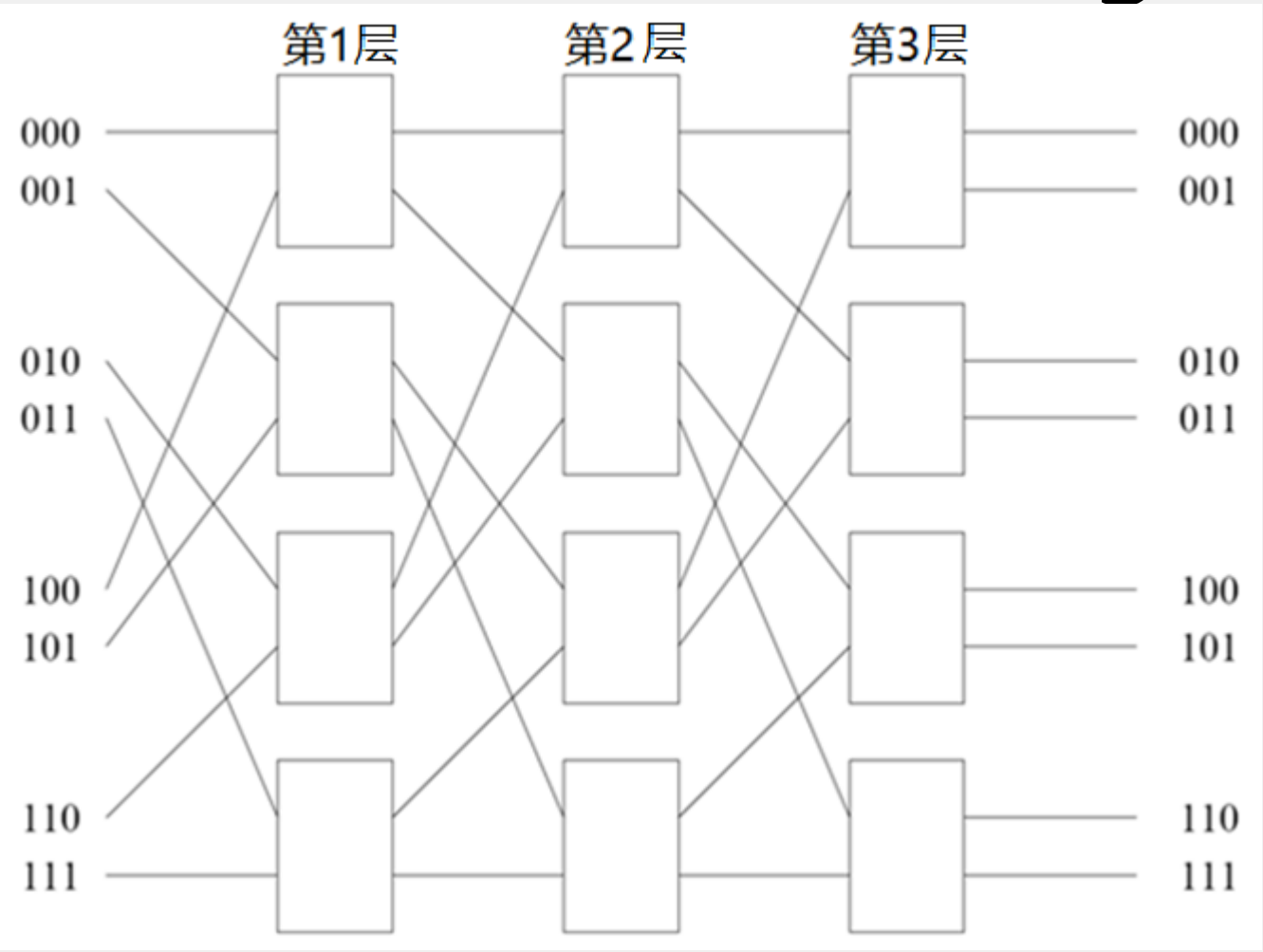
- 间接互联网络——多级网络
  - 交叉开关网络从性能角度上考虑有不错的可扩展性，但是从成本上考虑其可扩展性不好。
    - 交叉开关网络的静态对应网络是全连接网络。
  - 总线网络从成本上来说有不错的可扩展性，但是从性能上考虑其可扩展性不好。
  - 多级网络是两者的折中。

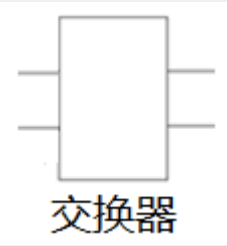
间接互联网络：多级网络



- 典型多级互连网络示意图

间接互联：多级Omega网络





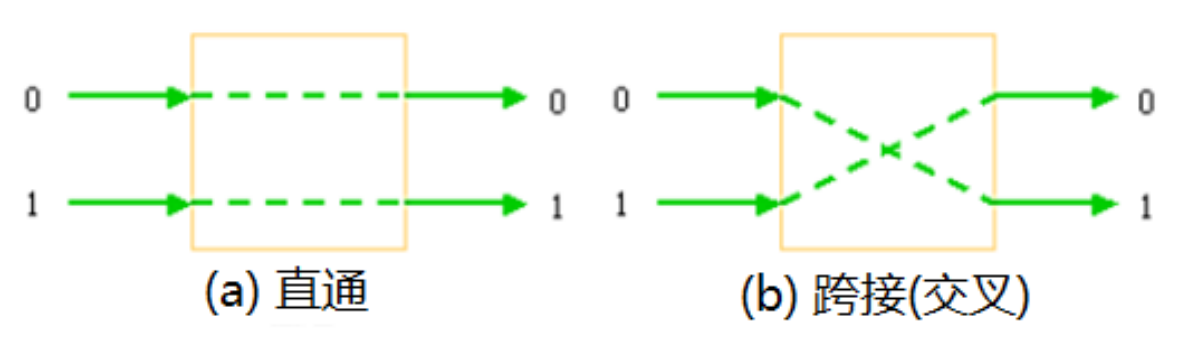
问题：1）每一层如何路由可由左边任意入口到达右边任意出口；

2）开关（交换器）连接方式？

间接互联网络：多级Omega网络

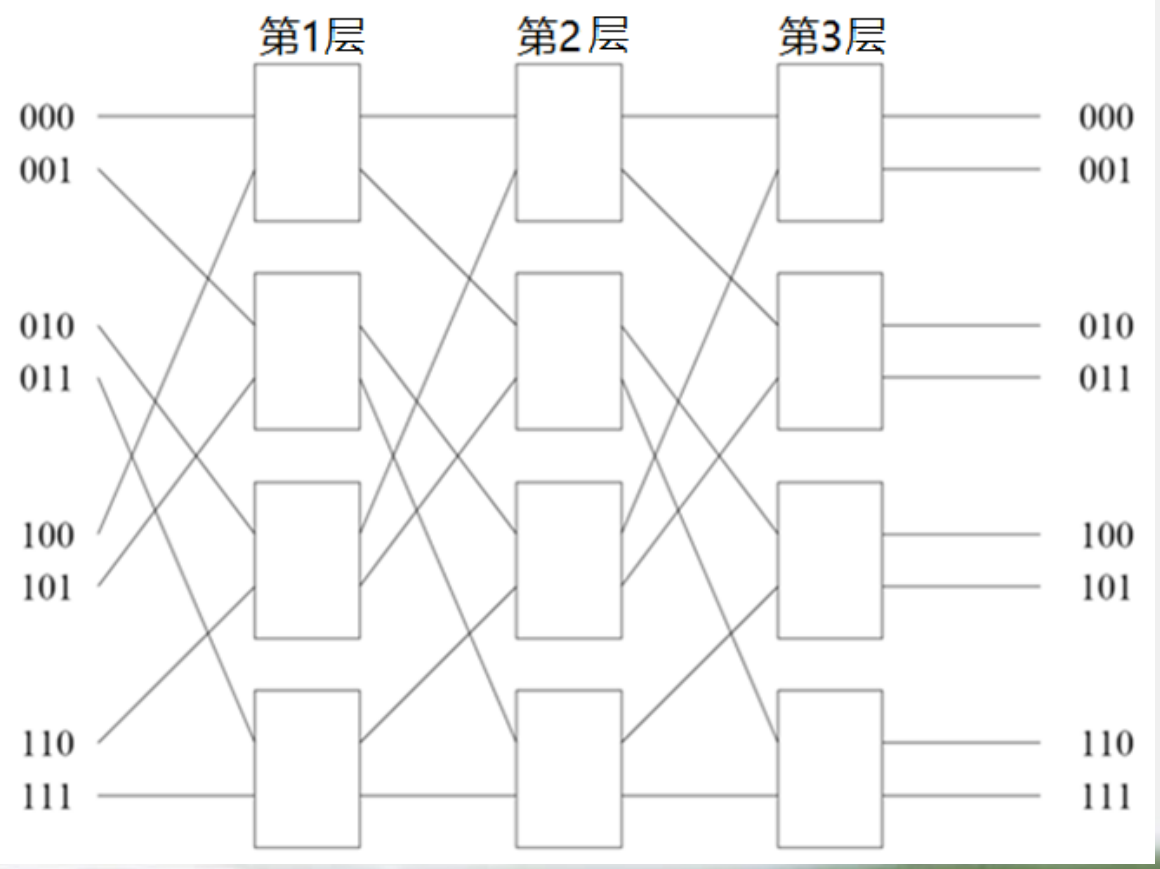
- 完全混洗互连模式使用2×2开关。
- 开关的连接方式有两种:

  (a)直通式连接; (b)跨接连接。



间接互联网络：多级Omega网络

- 使每一个输入i都可以连到任何一个输出j：
  - 采用树型结构: 一个i经过第一层（一个开关）可以连接两个开关，经过第二层可以连接四个开关，第三层四个开关可选联任何j。



间接互联网络：多级Omega网络

- 大多数经常使用的多级互连网络是Omega网络。
- 在每一层与开关的连线按下面条件连接：连线的i 端(输入)连接到 j 端(输出)，如果它们满足以下条件：

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i + 1 - p, & p/2 \le i \le p - 1 \end{cases}$$

- 目标：使网络由*log p*层组成，p是输入/输出端的个数。

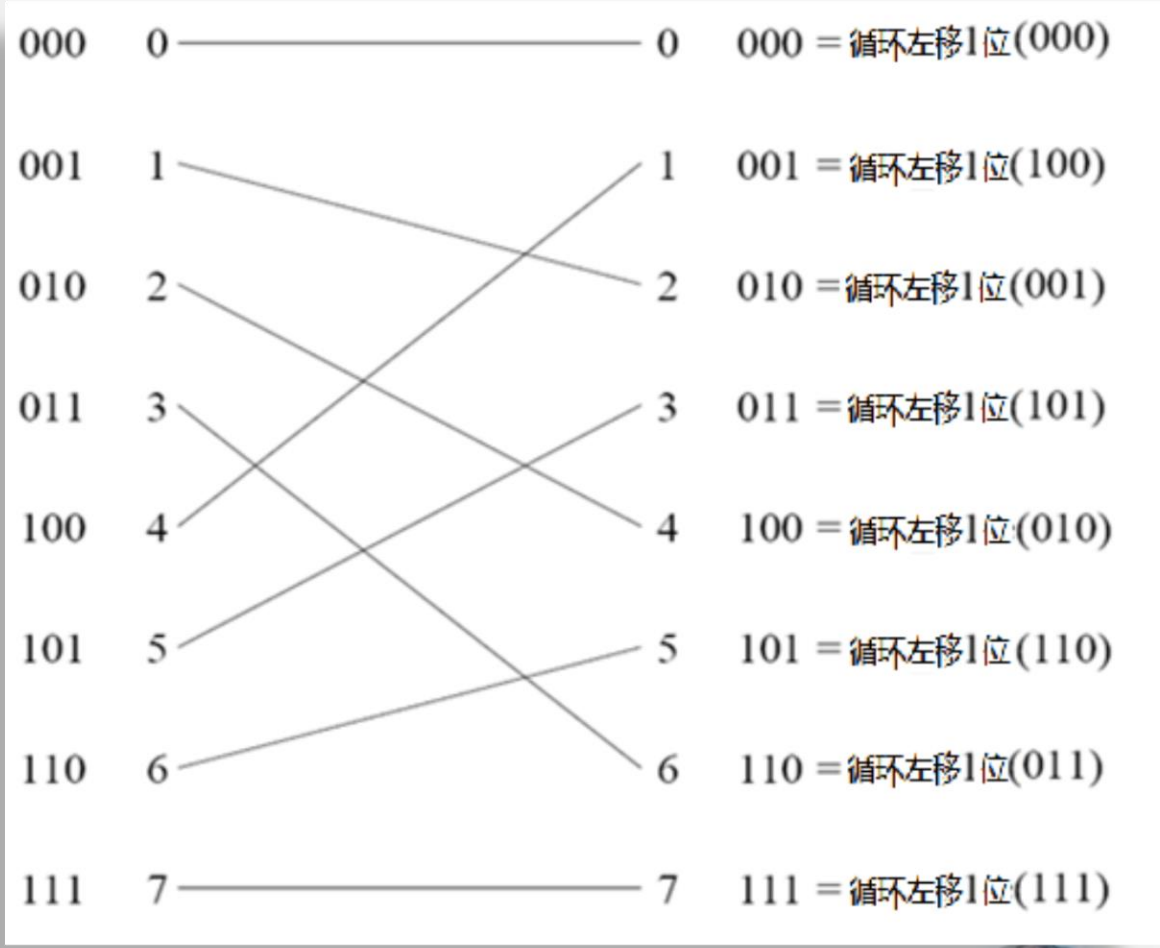
间接互联网络：多级Omega网络

- Omega网络的每一层使用完全混洗互连模式，如下：

- 连接条件：

$$j=\begin{cases} 2i, & 0\le i\le p/2-1 \\ 2i+1-p, & p/2\le i\le p-1 \end{cases}$$

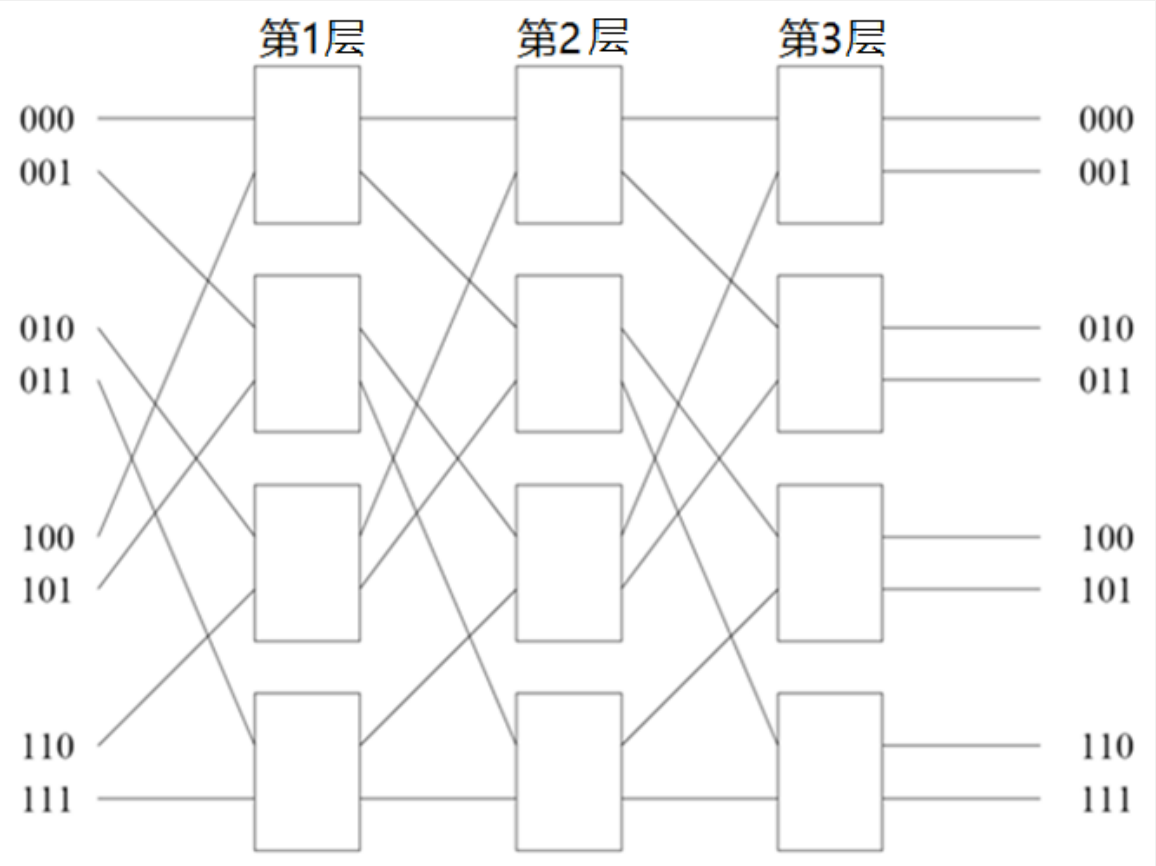
相当于将输入
i循环左移一位 → 输出j



含8个输入端和输出端的完全混洗互连

间接互联网络：多级Omega网络

- 一个使用完全混洗连接和开关的完全Omega网络现描述如下：（连接条件：左端点循环左移一位 = 右端点）



最左端的111 → 最右端的000路由全过程如下：

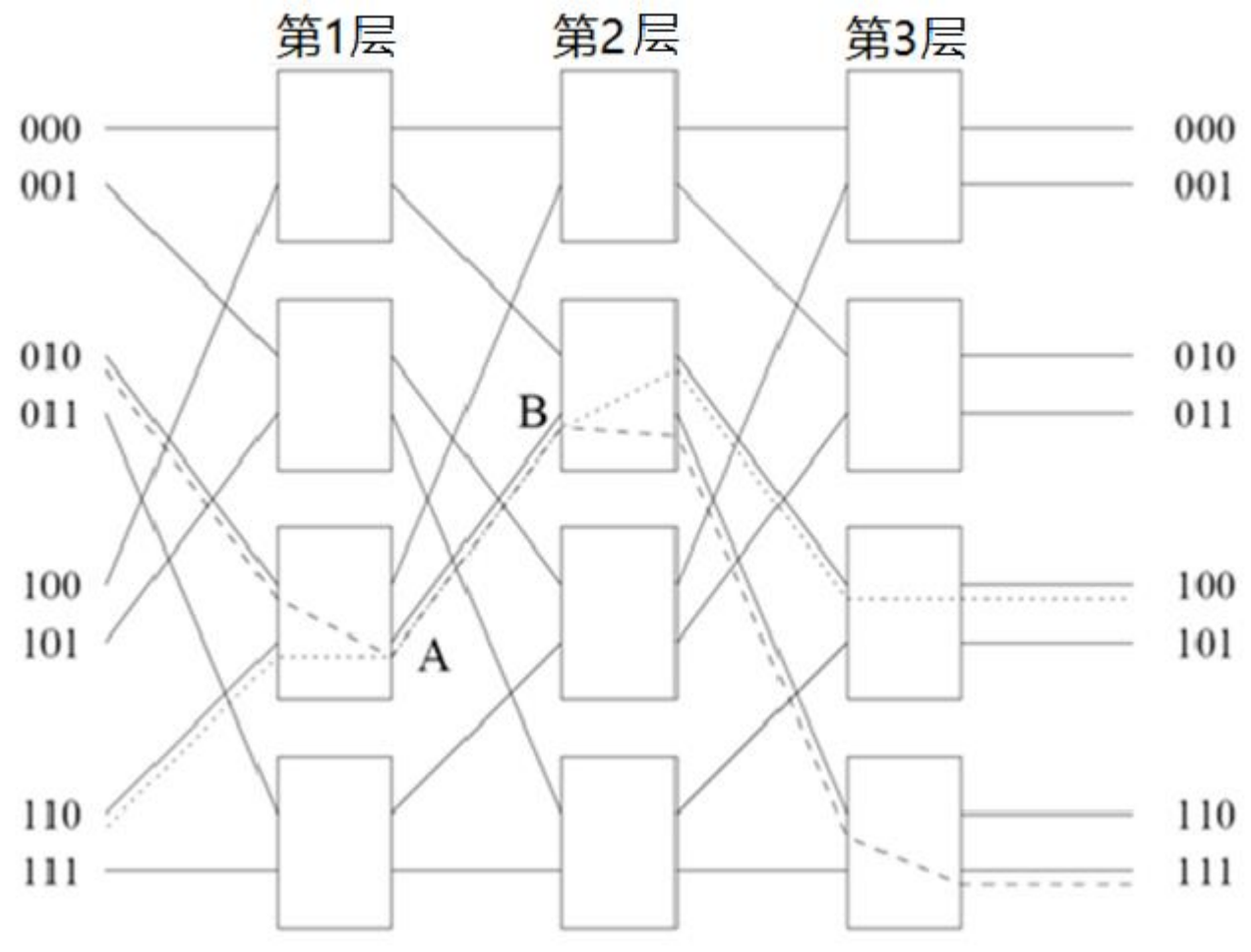
- 沿连线从**111**到第**1**层开关**111**端（端点按**111**左移**→111**）
- 第**1**层**: 111→110**(跨接)
- 沿连线从**110**到第**2**层开关**101**端（端点按**110**左移**→101**）
- 第**2**层**: 101→100** (跨接)
- 沿连线从**100**到第**3**层开关**001**端（端点按**100**左移**→001**）
- 第**3**层**001→000** (跨接)

- 连接8个输入和输出的完全Omega网络
- Omega网络含***p/2*** × ***log p***个开关节点，网络成本随***O (p log p)***增加。

间接互联：多级Omega网络-路由

- 令s表示源存储区的二进制形式，d表示目的处理器的二进制形式。
- 数据通过链路来到第一个开关节点。如果s和d的最高位相同，则开关切换为直通模式，否则切换为跨接模式。
- 这样的处理在*log p*级开关中重复。
- 这是一个阻塞网络：见下页

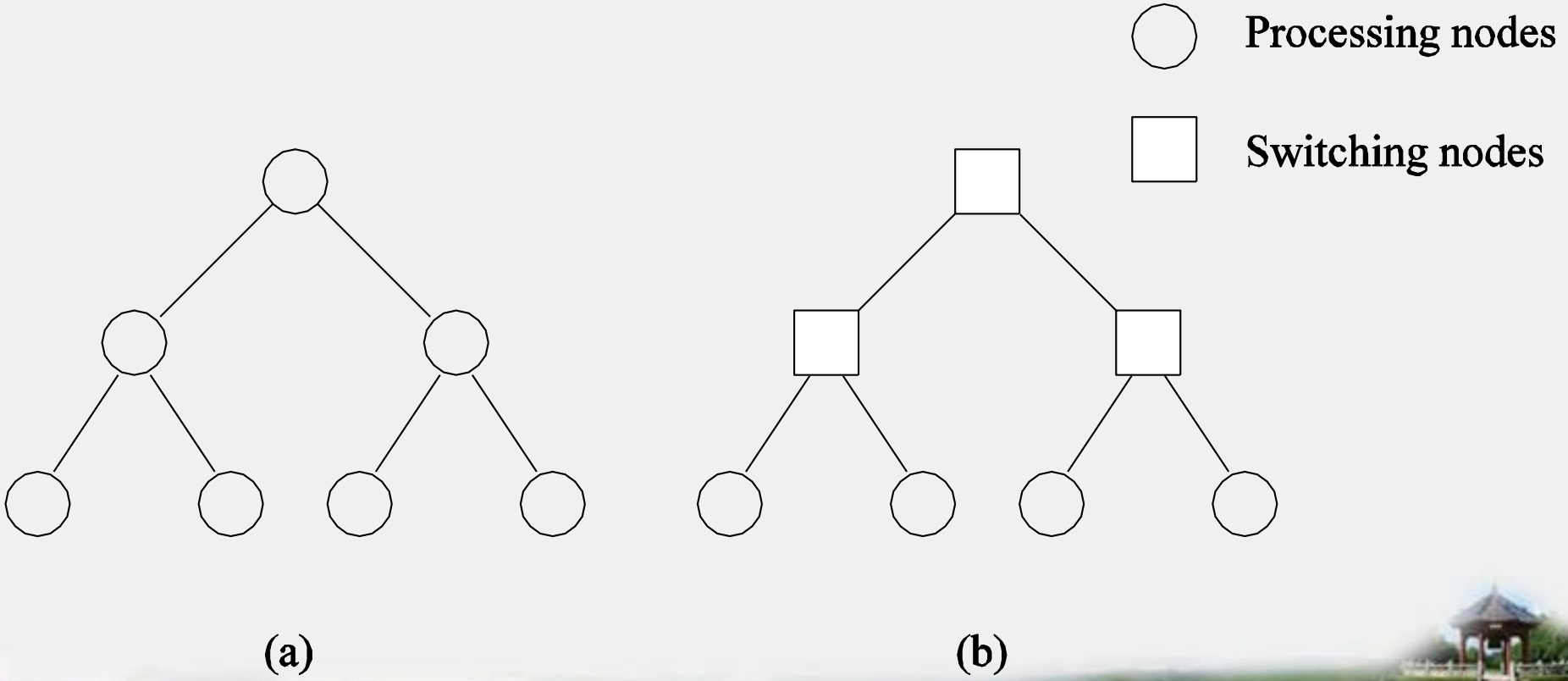
间接互联：多级Omega网络-路由



- Omega网络中的阻塞实例：一个消息（010到111或110到100）在链路AB处阻塞。

其他网络：基于树的网络

- 完全二叉树网络：
  - a）静态树网络——直接连接；
  - b）动态树网络——间接连接。

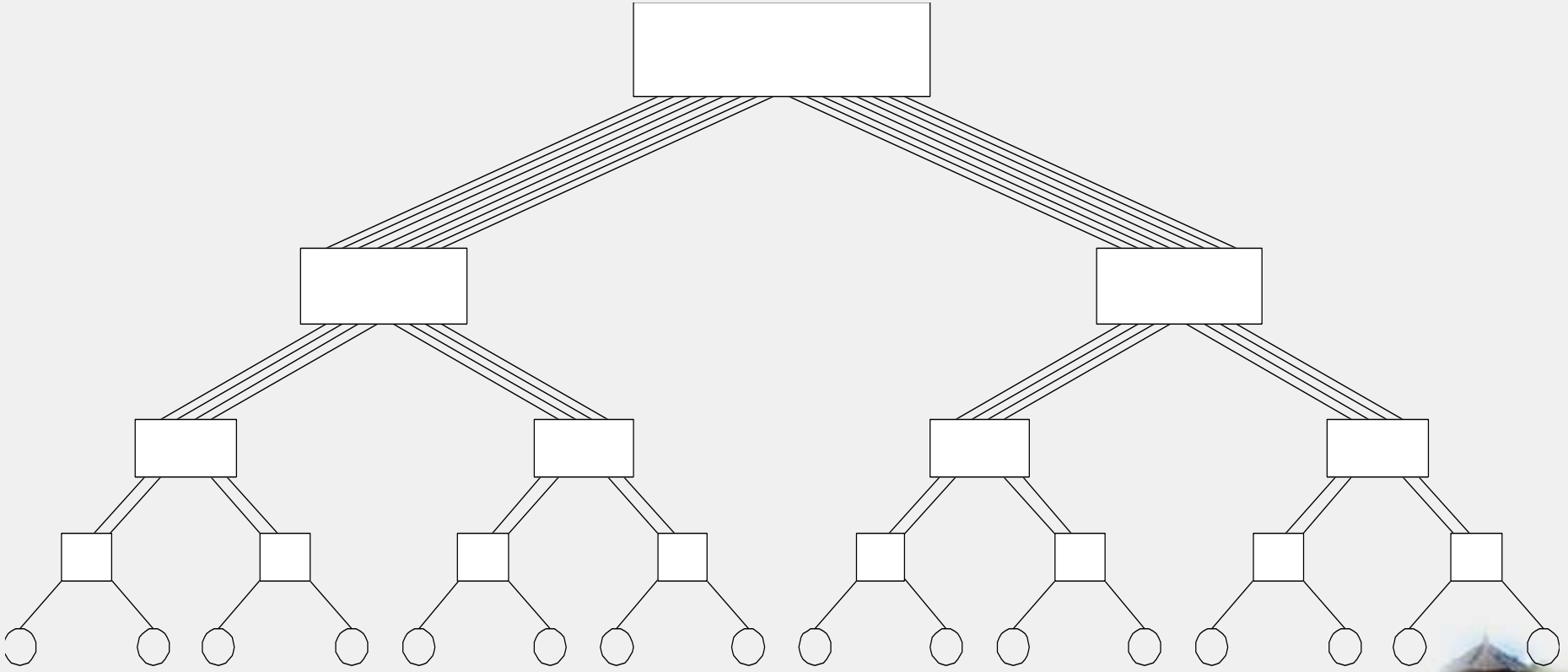


其他网络：树的性质

- 任意两个节点的距离不超过$2\log p$ 。
- 树的高层链路比底层链路承受更大的流量。
- 由于这个原因，胖树作为一种树结构的变体，它的链路数随着层次的增高而增多。
- 树可以在二维网络中设计出来而不需要线路交叉。这是树结构引人注目的性质。

其他网络：胖树

- 包含16节点的胖树结构。



缓存一致性

缓存一致性问题

- 指在含有多个Cache的并行系统中，数据的多个副本（因为没有同步更新）而造成的不一致问题。
- 原因
  - 共享一个可写变量
  - 进程迁移：一个进程从当前处理器移励到指定的处理器上
  - 某些I/O操作
  - 等等

缓存一致性

- 程序员无法直接控制CPU Cache，对共享内存系统带来很多重大的影响。

例

y0 是 Core 0 的私有变量；

y1 和 z1 是 Core 1 的私有变量；

x = 2; /* 两个核的共享变量 */

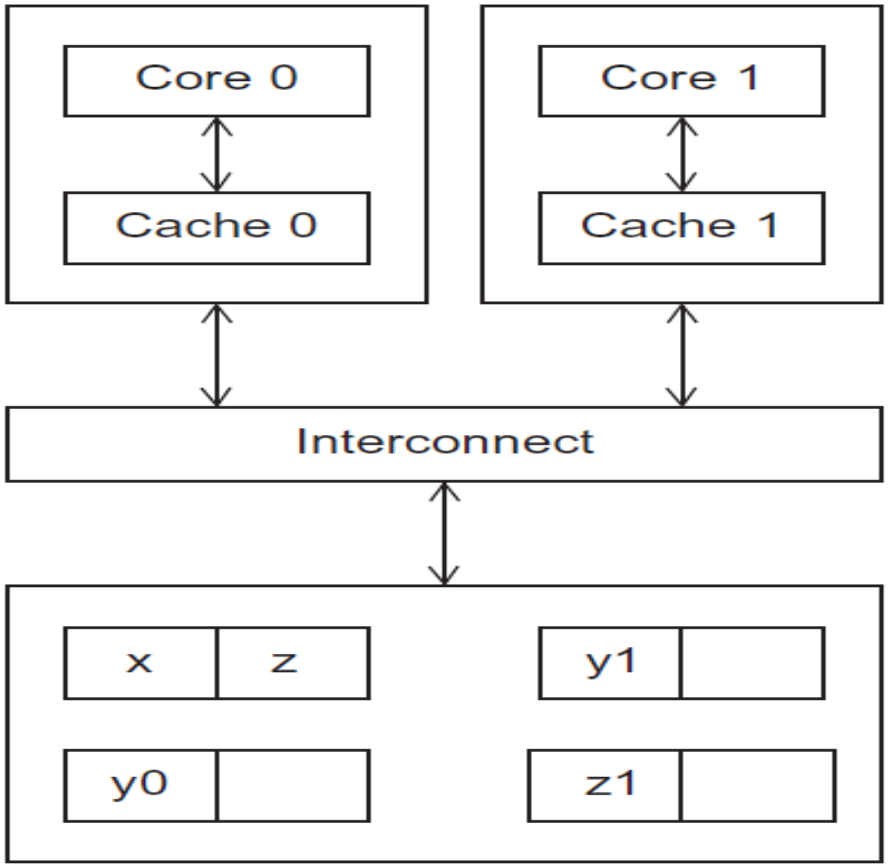
Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	不含x的语句
2	不含x的语句	z1 = 4*x;

最终结束时

y0 = 2

y1 = 6

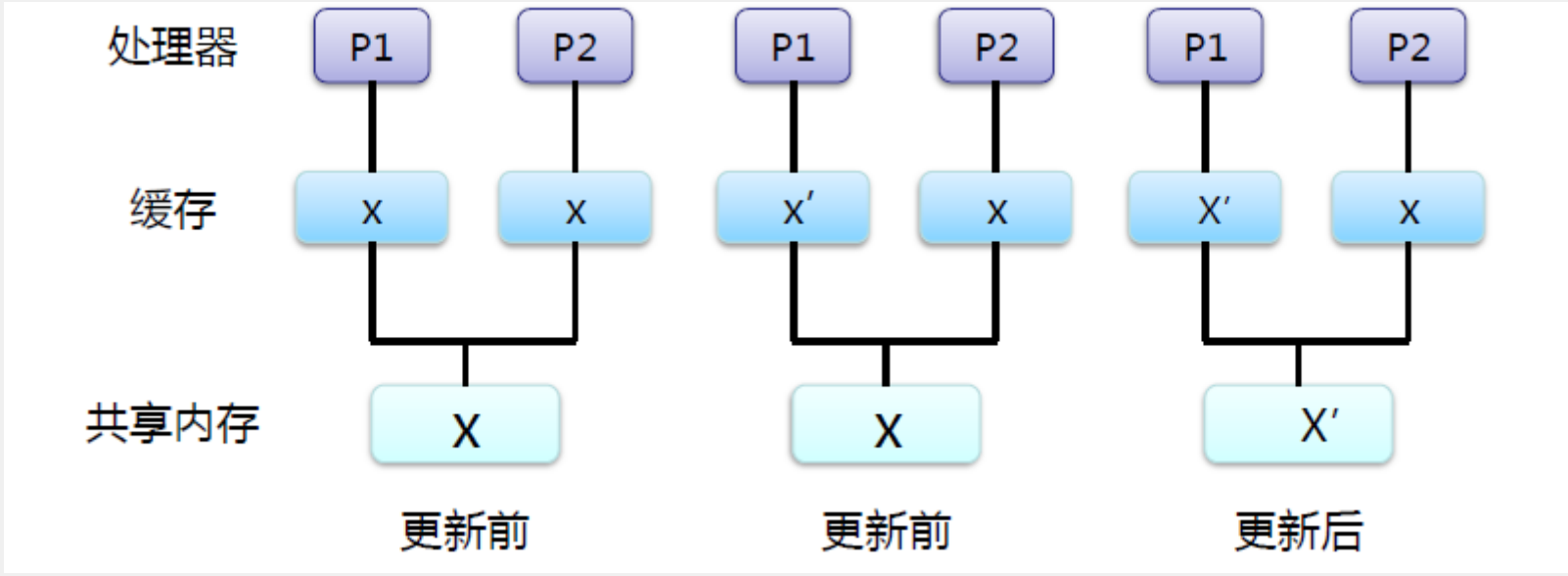
此时 z1 = ???



具有双核和双cache的共享内存系统

缓存一致性问题

- 共享可写变量造成的缓存一致性问题
  - 写直达：Cache与主存同时发生写修改。
  - 写回：只修改Cache的内容而不立即写入主存；只有当此行被换出时才写回主存。



前例，此时core 1 Cache中的x未变（=2），z1 =4*x=8;

缓存一致性策略

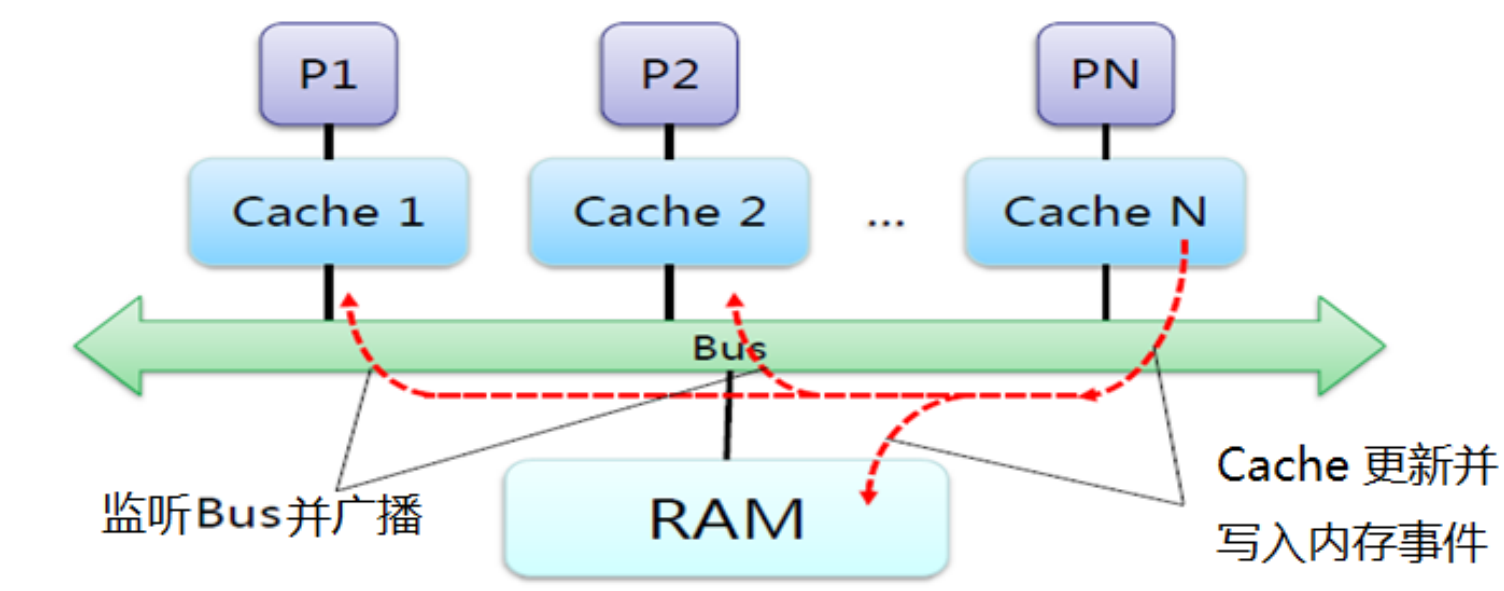
- 缓存一致性策略
  - 写无效策略
    - 指当某个处理器更新其私有Cache中的某个数据时，它通知所有其它Cache这一数据在它们中的副本从此均无效。
  - 写更新策略
    - 指当某个处理器更新其私有Cache中的某个数据时，它把所更新的数据发送给所有的其它Cache，以更新这一数据在其它Cache中的所有副本。

缓存一致性方法

- 缓存一致性协议
  - 监听总线协议——核共享总线，且能看到修改
    - 互联网络可以实现广播功能时——实际通过总线广播
  - 基于目录的协议——一个分布式数据结构，各存一部分目录
    - 互联网络不能实现广播功能时

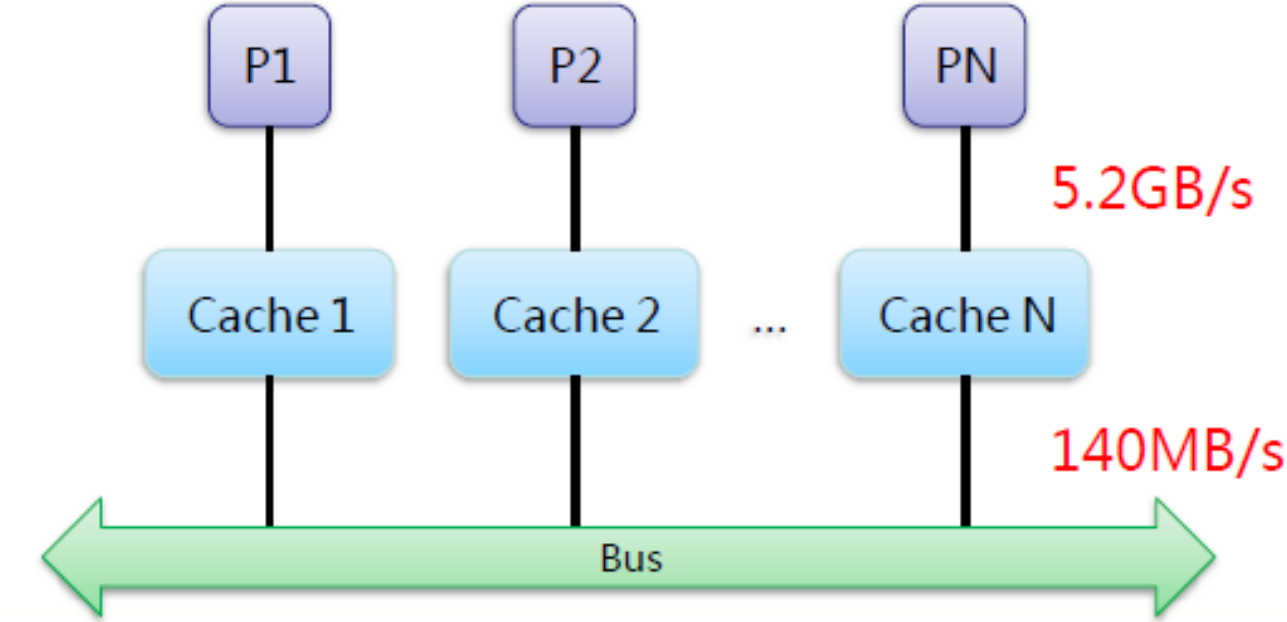
缓存一致性协议
——监听总线协议

- 核共享总线，且能看到修改
  - 指所有处理器都监听总线，当某个处理器修改了私有Cache中的数据后，它在总线上广播无效信息或更新后的数据，以使其它副本无效或更新其它副本。
  - 写一次协议、Berkeley协议，Synapse协议，Firefly协议等



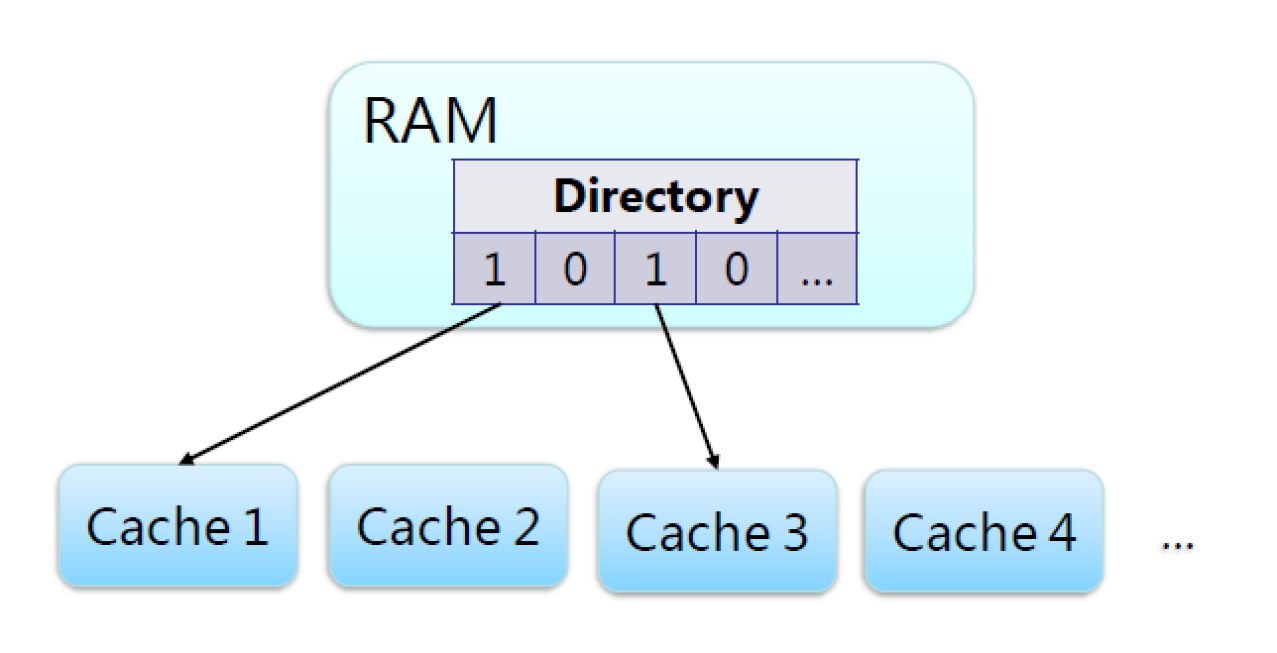
缓存一致性协议
——监听总线协议

- 局限性
  - 当某一处理器修改了私有Cache中的数据后，并非每个处理器的Cache中都有该数据的副本；
  - 总线与高速缓存的带宽差异影响了效率。



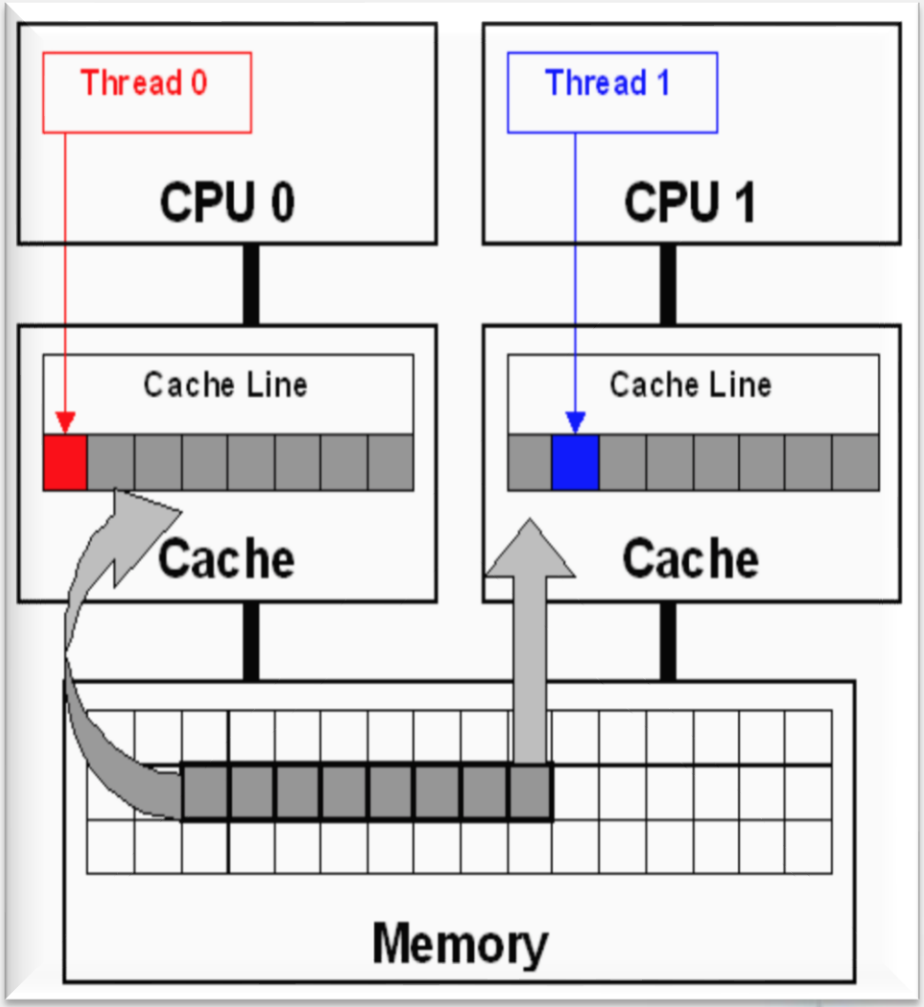
缓存一致性协议
——基于目录的协议

- 共享存储器维护一个目录，称为高速缓存目录，该目录中记载了申请了某一数据的所有处理器。这样，当数据被更新时，就根据目录的记载，向所有其Cache中包含该数据的处理器“点对点”地发送无效信息或更新后数据。



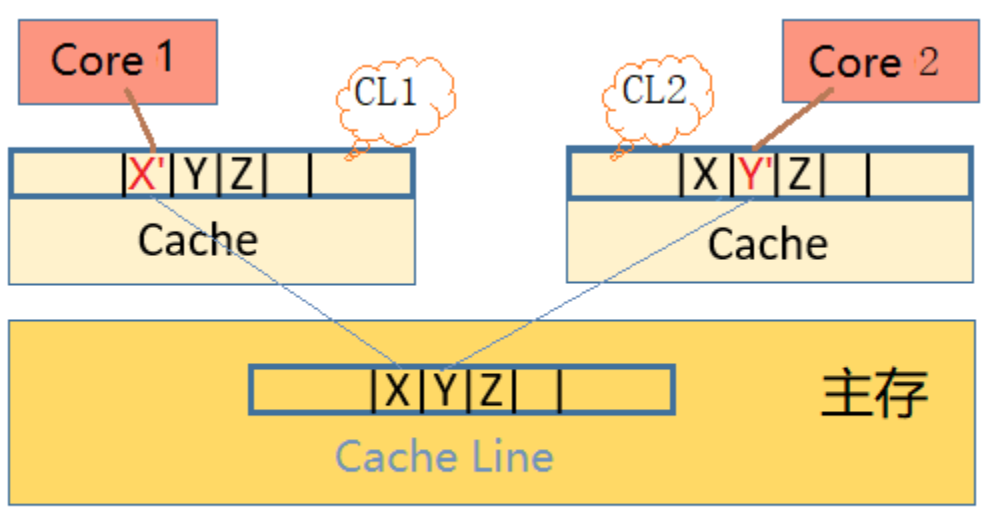
缓存一致性
——伪共享问题

- 伪共享问题
  - CPU读数据到Cache时是以Cache Line为单位读入的。如果两个CPU中的线程分别有不同的Cache Line对应同一内存块，那么当两个线程同时在各自的Cache进行写操作时，将会造成两个线程所共享的一内存块所对应的Cache内不一致：为保持一致而使效率将成百倍的下降。



伪共享引起性能问题

- 共享数据X,Y,Z已装入Cache
  - 线程A在Core1修改X为X’，线程B在Core2上修改Y为Y’。
  - 根据缓存一致性协议的更新过程如下：



共享数据X, Y, Z在一个Cache Line上并被装入Cache

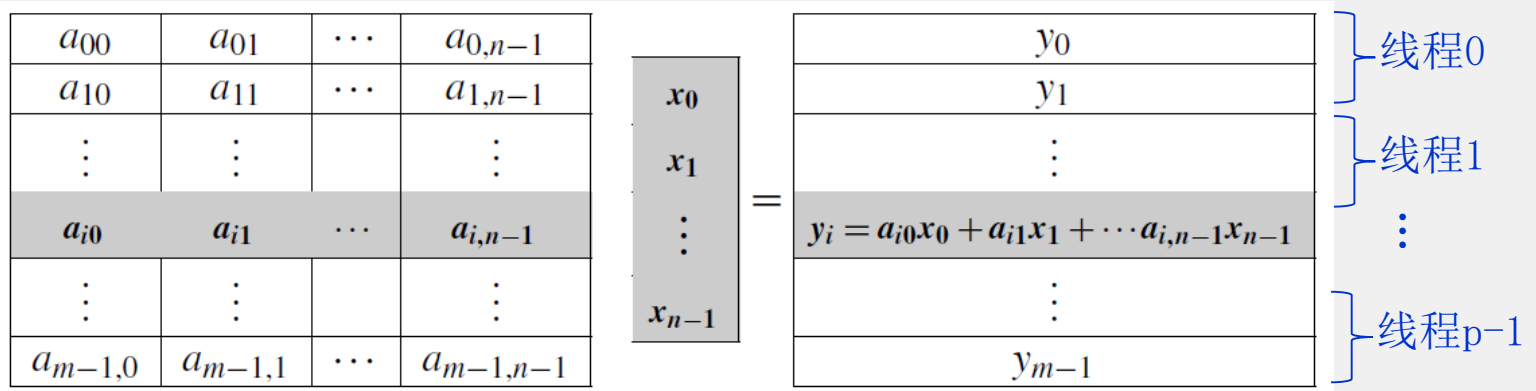
注：CL1、CL2分别为Core1、Core2的Cache Line

- 设Core1首先发起操作修改X，这时Core1上CL1由共享态变成修改态，并告知Core2 CL2无效；当Core2发起写操作时，首先导致Core1将CL1(含X’)写回主存，CL1由修改态变为无效态，而后Core2从主存重新读取该内容，其CL2由无效态变成独占态，然后修改Y，其CL2从独占态变成修改态。
- 这个过程是串行的，并且增多Cache缺失。若多个线程操作同一Cache Line发生伪共享，则变成了串行程序，降低了并发性。

伪共享问题的解决方法

- 处理伪共享的两种方式：
  - 分配的内存，可以采取一定的内存分配算法使每块内存不会在不同的Cache行里。
    - 增大数组元素的间隔（填充）使得不同线程存取的元素位于不同的cache line上。典型的空间换时间。
  - 每个线程都操作指定区间（包含若干个完整的Cache行）的数据
- 发现伪共享的Linux命令: perf c2c
  - 先用 perf c2c record 通过采样，收集性能数据
  - 再用 perf c2c report 基于采样数据，生成报告
    - c2c意思是“缓存到缓存” (cache-to-cache)。

矩阵向量乘中的伪共享影响

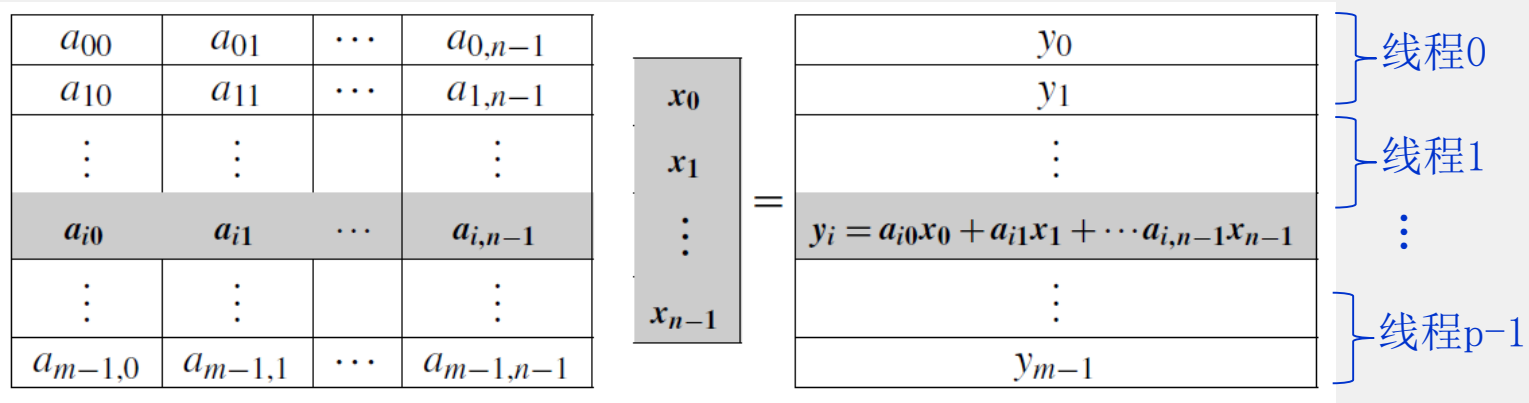


p个线程并行计算：每个线程计算向量**y**的**m/p**个元素。

Threads	Matrix Dimension 时间单位：秒		
	8,000,000 × 8	8000 × 8000	8 × 8,000,000
	Time	Time	Time
1	0.322	0.264	0.333
2	0.219	0.189	0.300
4	0.141	0.119	0.303

- 注意，当m=8，n=8,000,000时，线程增加运行时间几乎不变，为什么？
  - 当m较小时，向量y被放在一个Cache Line中，多个线程修改y，发生伪共享——几乎串行执行。

矩阵向量乘中的伪共享影响



- 当m=8，n=8,000,000时，如何避免伪共享？
  - 采用填充法。
  - 设cache line大小为size，且m/p<=size/4，这里4是float的字节数；
    将向量y声明为2维数组：
    float y[p][size/4];
    - 线程i计算m/p个数组y的元素依次存入y[i][0]~y[i][m/p -1]
    - 用空间换时间：多用了p* size/4 – m 个数组元素。

C++对象伪共享的例子

```
class Data{
    long Time;     //线程0修改
    long x;
    boolean flag; //线程1修改
    char key;      //线程2修改
    int y;
    int value;     //线程3修改
    long z;
};
```

- 将对象属性分组，一起变化的放在一组，不变的属性放到一组，其他属性放到一组。

通过填充变量，使不相关的变量分开。

```
class Data {
    long a1,a2,a3,a4,a5,a6,a7,a8;
    //防止与前一个对象产生伪共享
    long Time;
    long b1,b2,b3,b4,b5,b6,b7,b8;
    boolean flag;
    long c1,c2,c3,c4,c5,c6,c7,c8;
    char key;
    long d1,d2,d3,d4,d5,d6,d7,d8;
    int value;
    long e1,e2,e3,e4,e5,e6,e7,e8;
    long x;
    int y;
    long z;
    long f1,f2,f3,f4,f5,f6,f7,f8;
    //防止与下一个对象产生伪共享
}
```