



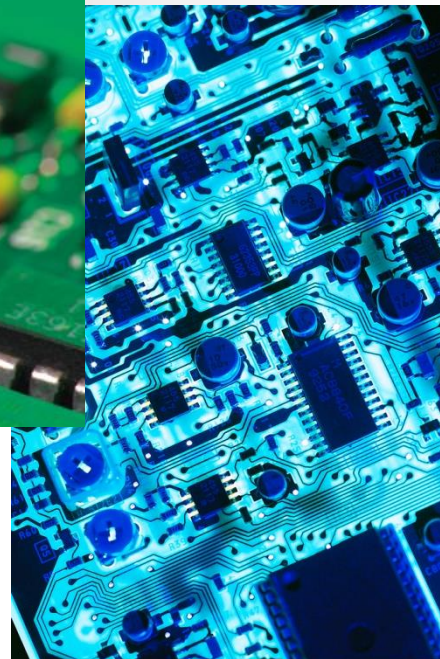
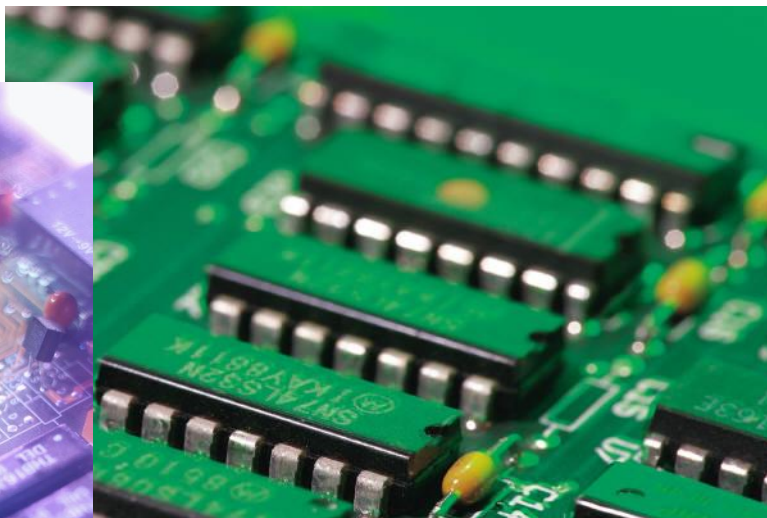
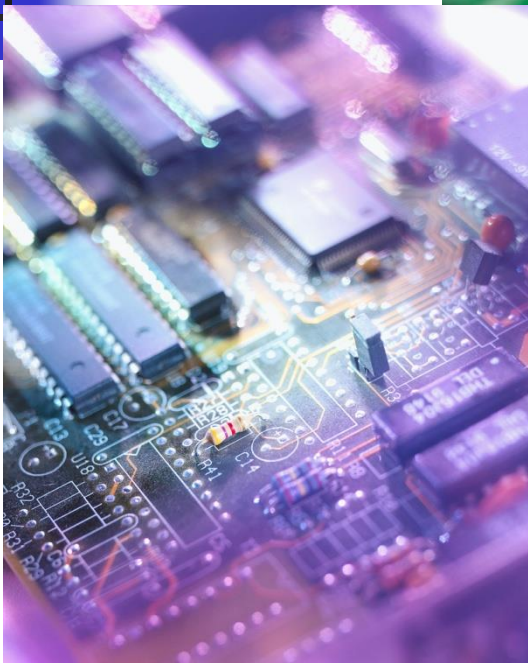
第2讲 并行硬件和并行软件

Parallel hardware And Parallel software



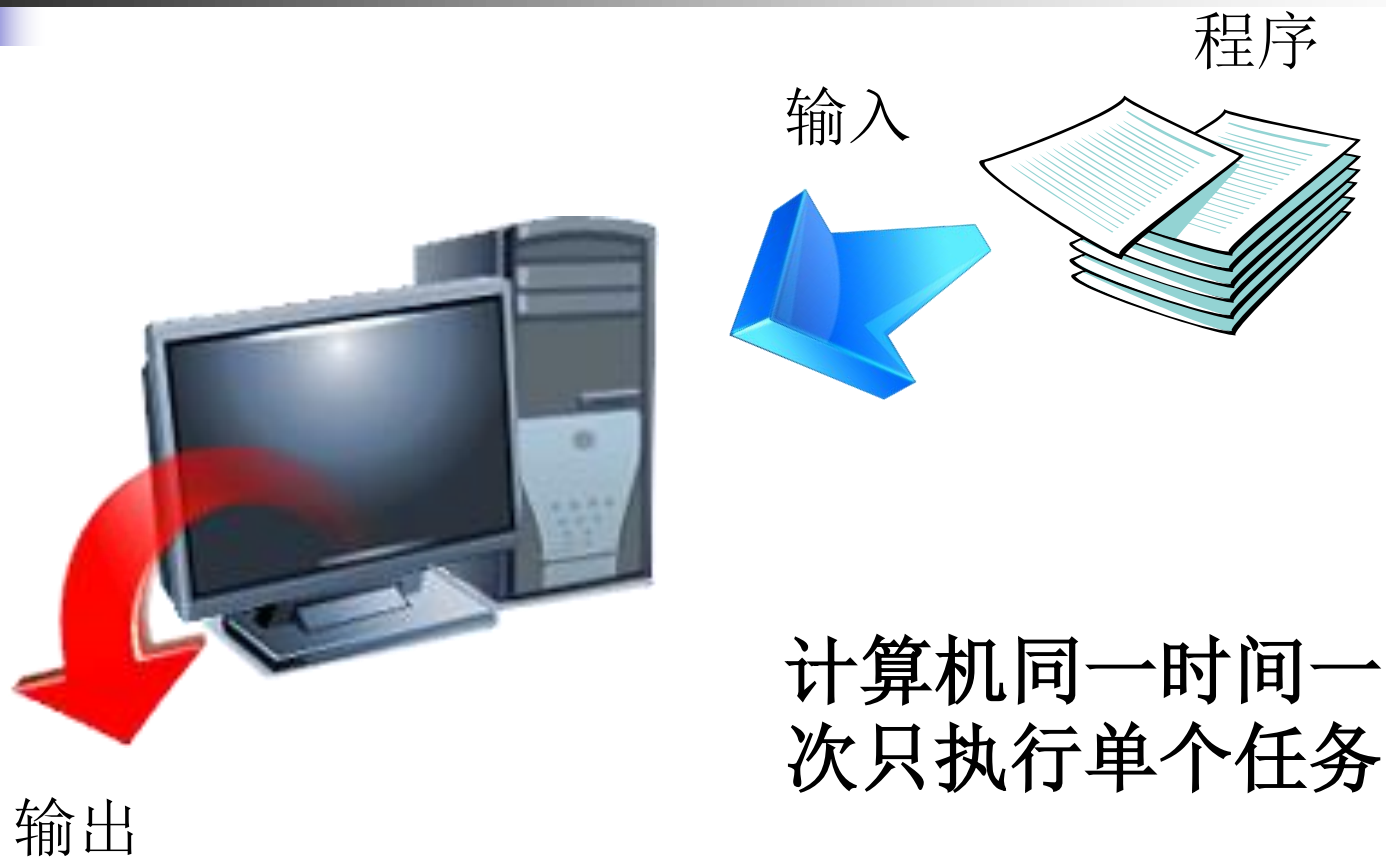
目录

- 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
- 并行程序设计
- 编写和运行并行程序
- 假设



背景知识

串行硬件和软件



冯·诺依曼结构

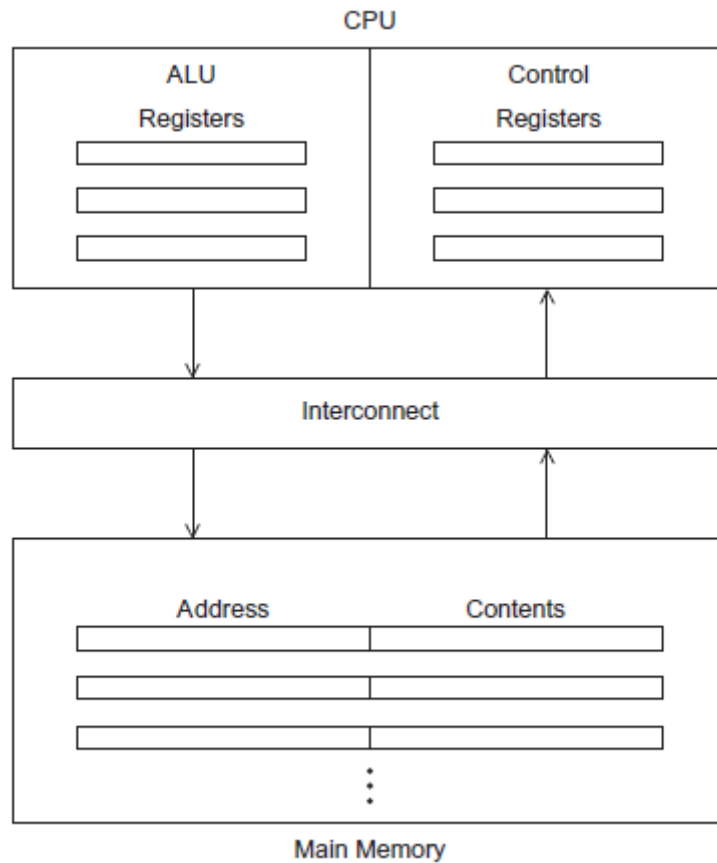
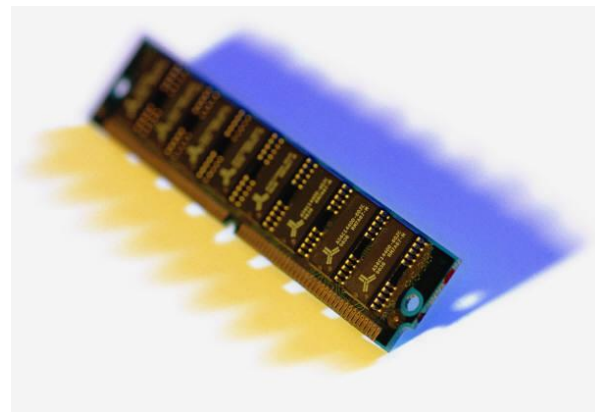


图 2.1

主存

- 主存中许多单元，每个区域都可以储存指令和数据。
- 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及区域中存储的数据和指令。



中央处理单元(CPU)

- CPU分为两个部分
- 控制单元 - 负责决定应该执行程序中那些指令(*the boss*)
- 算术逻辑单元(ALU) - 负责执行指令。 (*the worker*)



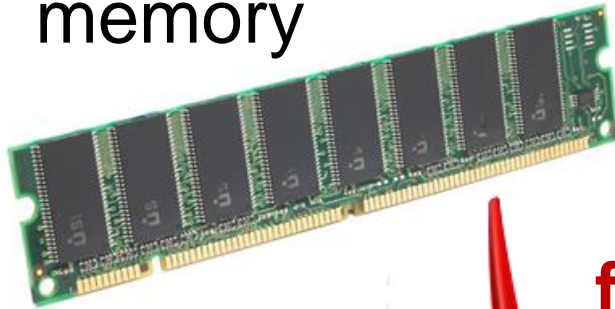
Key terms

- 寄存器 – CPU中的数据 and 程序执行时的状态信息存储在特殊的快速存储介质中
- 程序计数器 – 存储要执行的下一条指令的地址。
- 总线 – 连接 CPU 和内存的电线和硬件。





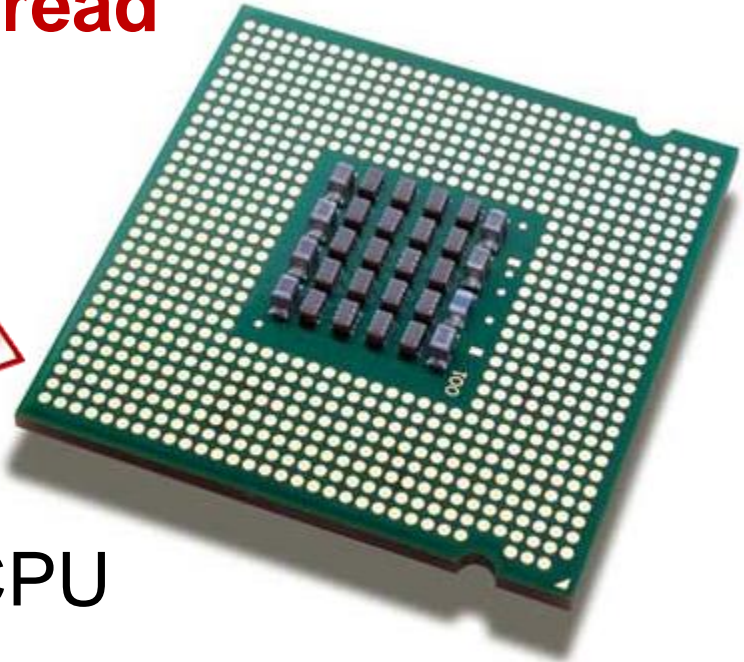
memory



fetch/read

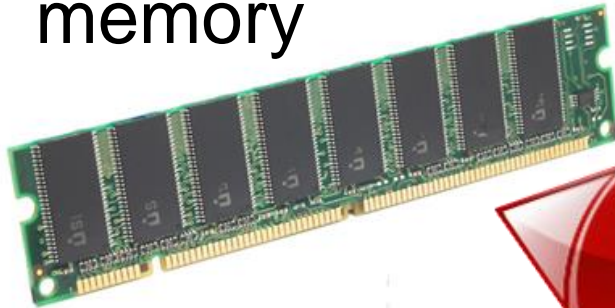


CPU





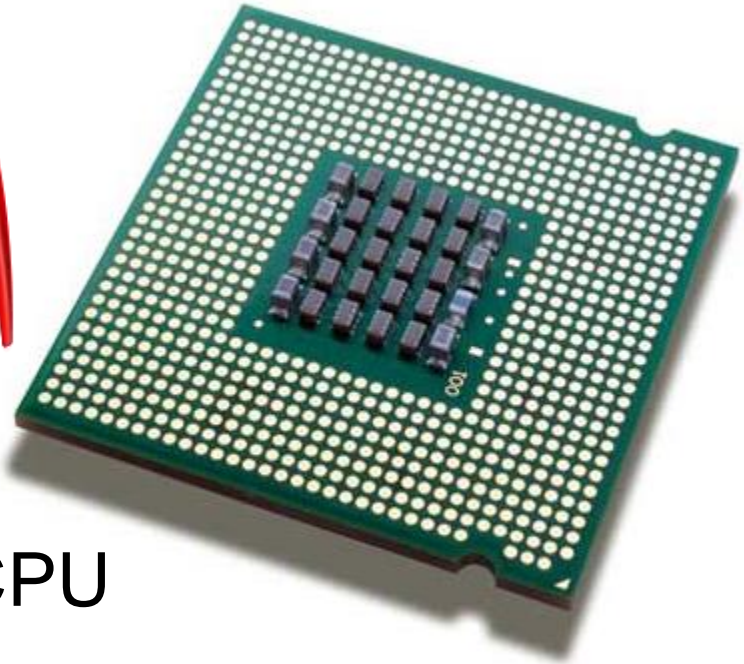
memory



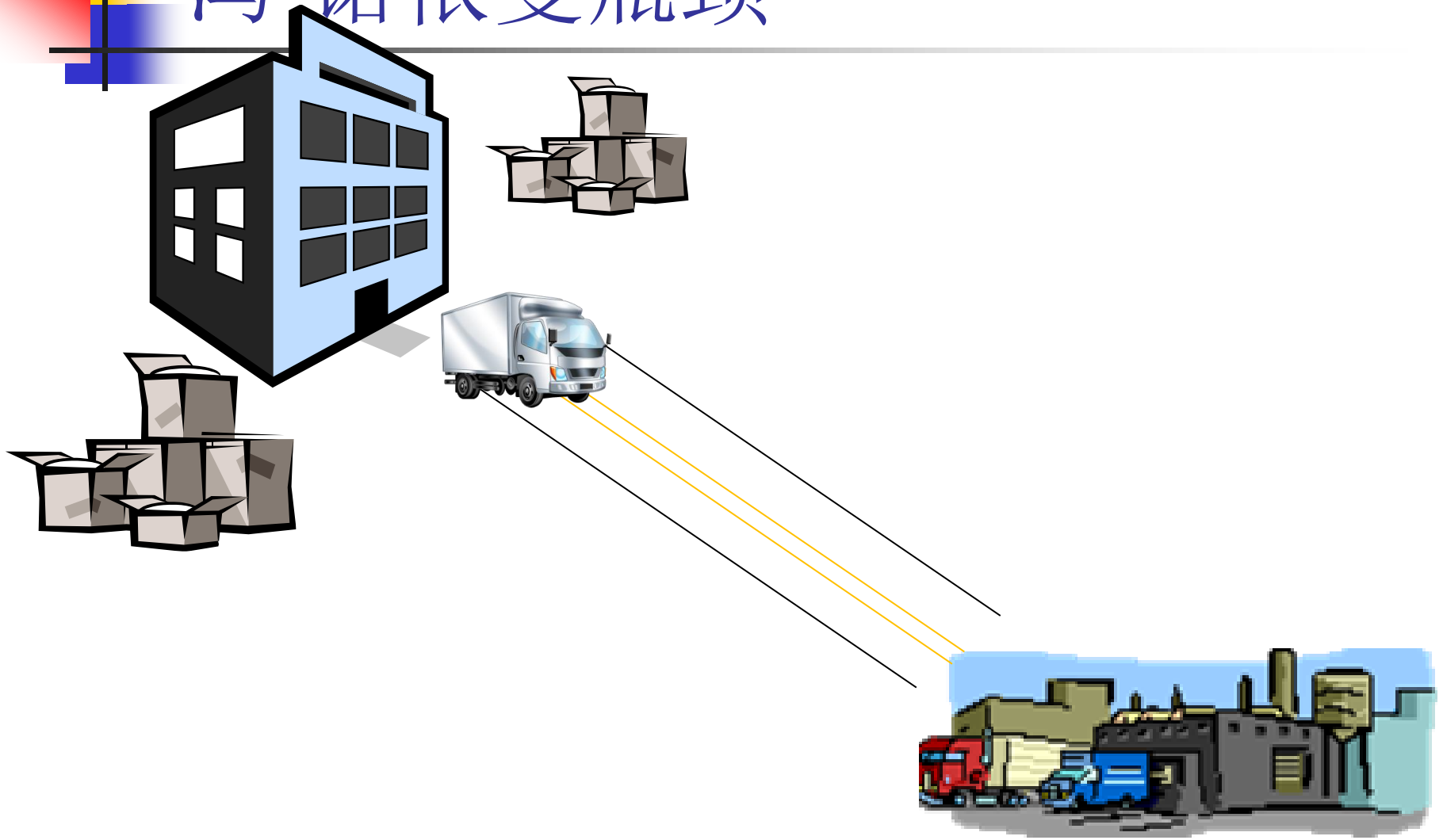
write/store



CPU



冯·诺依曼瓶颈





操作系统“进程”

- 进程是运行着的程序的一个实例。
- 一个进程包括如下实体：
 - 可执行的机器语言程序
 - 一块内存空间
 - 操作系统分配给进程的资源描述符
 - 安全信息
 - 进程状态信息



多任务操作系统

- 给人一种单一处理器系统同时运行多个程序的错觉
- 每个进程轮流运行 (**time slice**)
- 执行了一个时间片的时间后，它会等待一段时间直到它再次运行(**blocks**)



线程

- 线程包含在进程中
- 线程为程序员提供了一种机制，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。此外线程间的切换比进程间切换更快

一个进程和两个线程

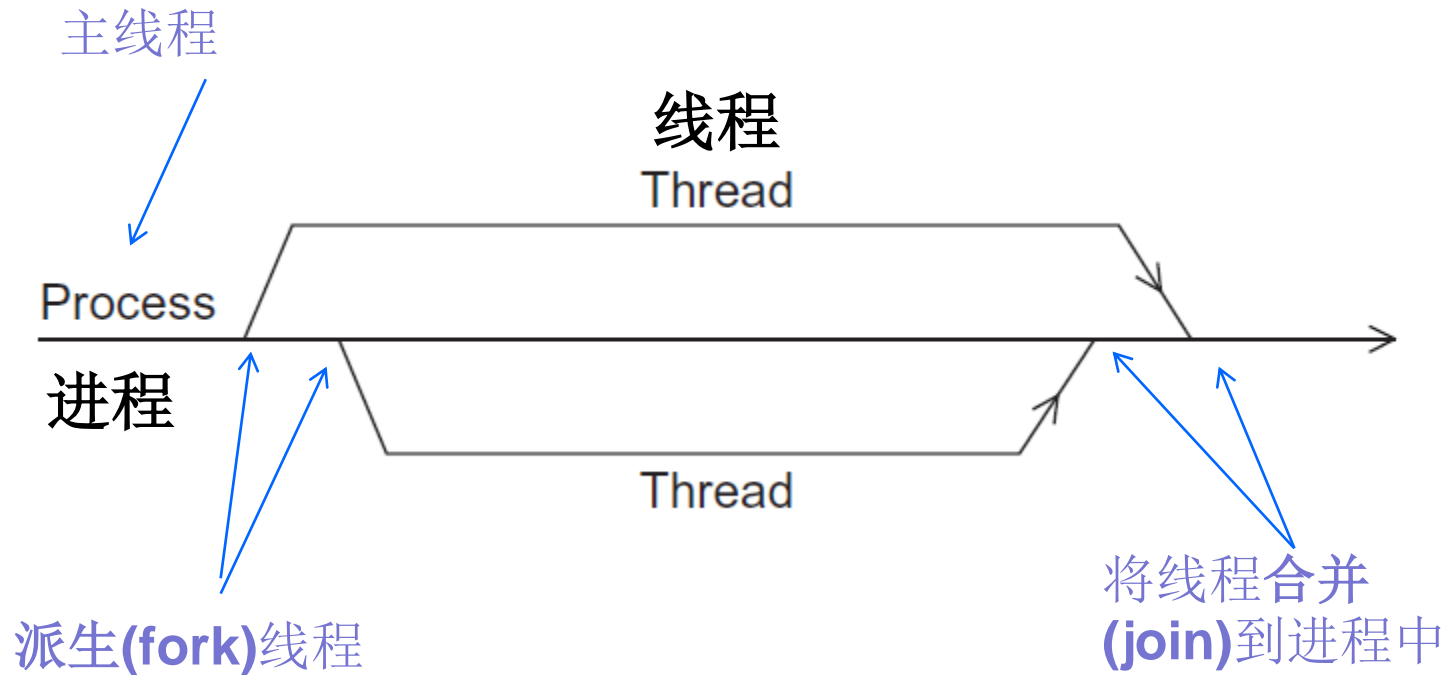
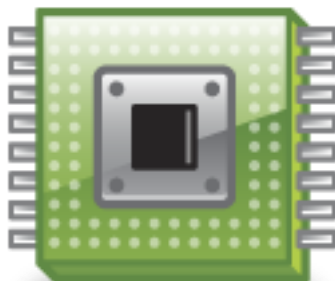


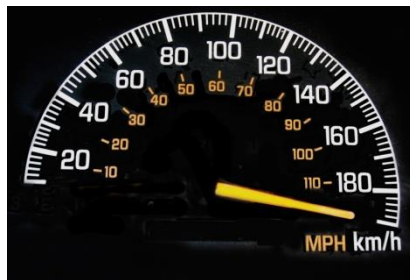
图 2.2



对冯-诺伊曼模型的改进

Cache基础知识

- 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芯片上，但比普通的内存芯片能更快地访问。





局部性原理

- 程序访问完一个存储区域往往会访问接下来的区域，这个原理称为局部性。
- 空间局部性 – 访问临近的区域
- 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];
```

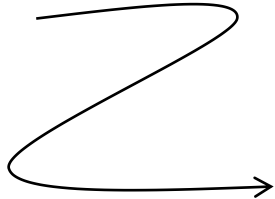


高速缓存行

- 一次内存访问能存取一整块代码和数据，而不只是单条指令或单个数据。这些块称为高速缓存块或**高速缓存行**。
 - 如一个高速缓存行可以存放16个浮点数，当执行`sum += z[0]`时，`z[0]~z[15]`会被一次性读到Cache中。
- 实际系统中，Cache通常是多层级的
 - 低层Cache（更快、更小）通常是高层Cache的Cache。也有低层不复制高层Cache的设计。

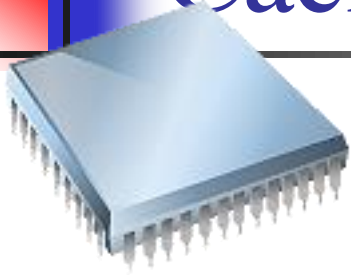
Cache 的层

smallest & fastest

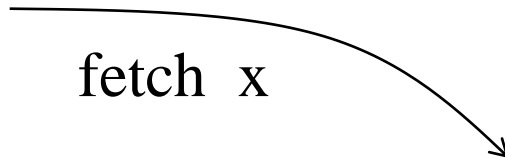


largest & slowest

Cache 命中



fetch x

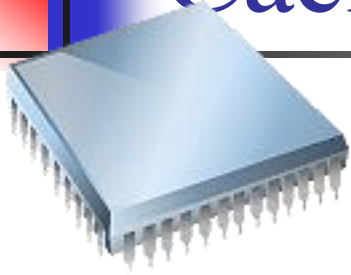


L1	x	sum
----	---	-----

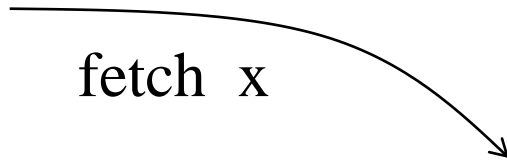
L2	y	z	total
----	---	---	-------

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

Cache 缺失



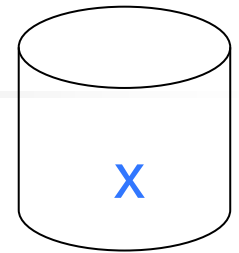
fetch x



L1 y sum

L2 r1 z total

L3 A[] radius center



main
memory



Cache的问题

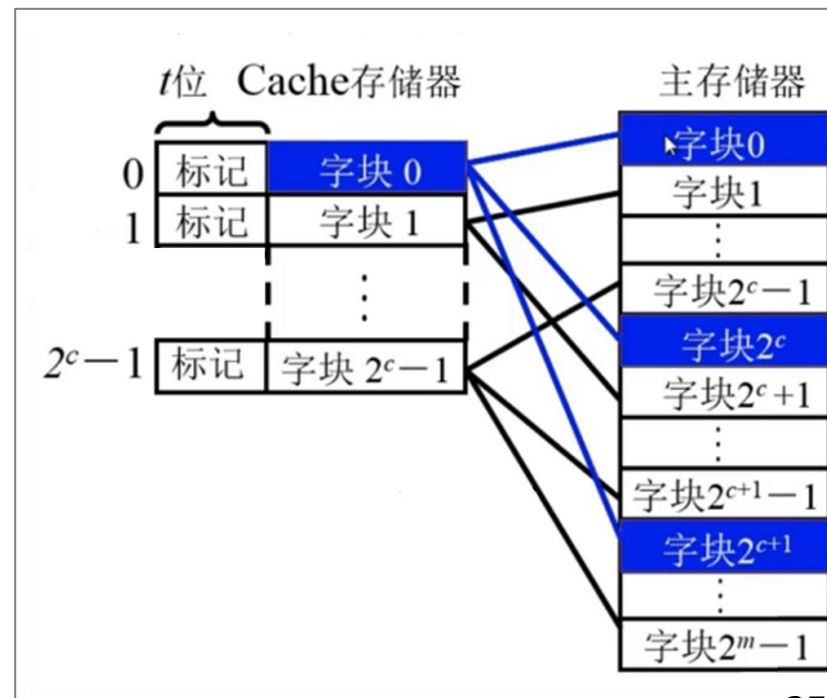
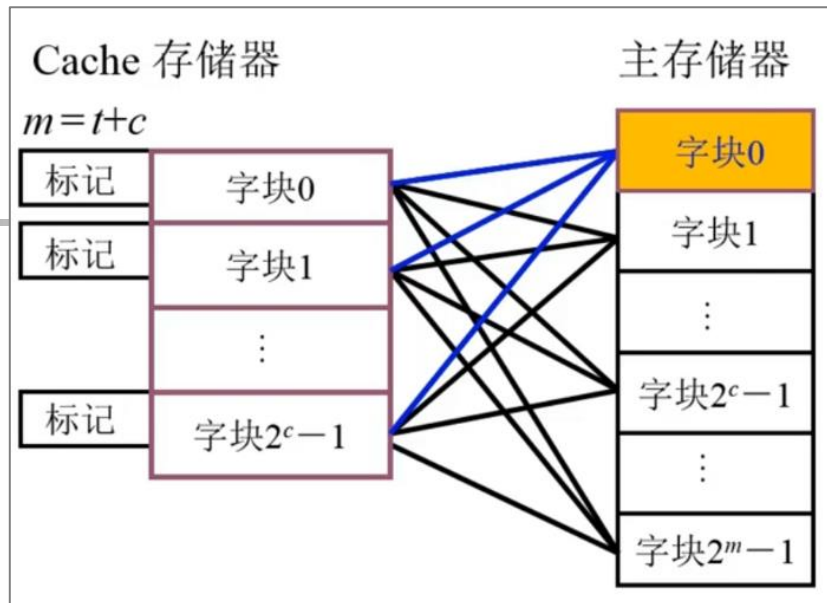
- 当CPU向Cache中写数据时，Cache中的值于主存中的值就会不同或者不一致。
- 在写直达(**write-thgrough**)Cache中，当CPU向Cache写数据时，高速缓存行会立即写入主存中。
- 在写回(**write-back**)Cache中，数据不是立即更新到主存中，而是将发生数据更新的高速缓存行标记为脏(dirty)。当发生高速缓存行替换时，标记为脏的高速缓存行被写入主存中。

Cache映射

- 全相联 – 每个高速缓存行能够放置在Cache中的任意位置

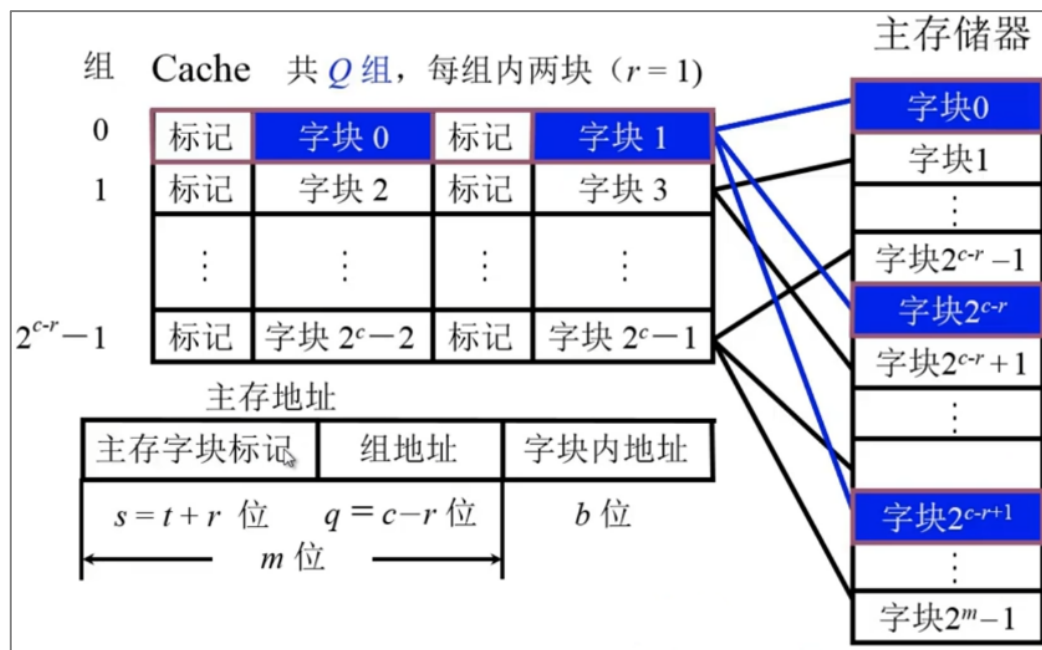
。

- 直接映射 – 每个高速缓存行在Cache中有唯一的位置。



Cache映射

- **n路组相连** – 每个高速缓存行都能放置在Cache中n个不同区域位置中的一个。



- 当内存中的行(多于一行)能被映射到Cache中的多个不同位置(全相联和n路组相连)时，需要决定替换或者驱逐 Cache 哪一行。





实例

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

表 2.1:将16行主存储器分配给4行高速缓存

Cache 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]



*虚拟存储器 (1)

- 如果运行一个大型程序或者程序需要访问大型数据集，那么所有指令或者数据在主存中放不下。
- 利用虚拟存储器(或虚拟内存)，使得主存可以作为辅存的缓存。



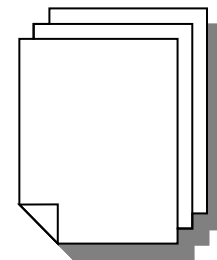
*虚拟存储器 (2)

- 它利用了空间和时间局部性。
- 它只在主存中存放当前执行程序所需要用到的部分

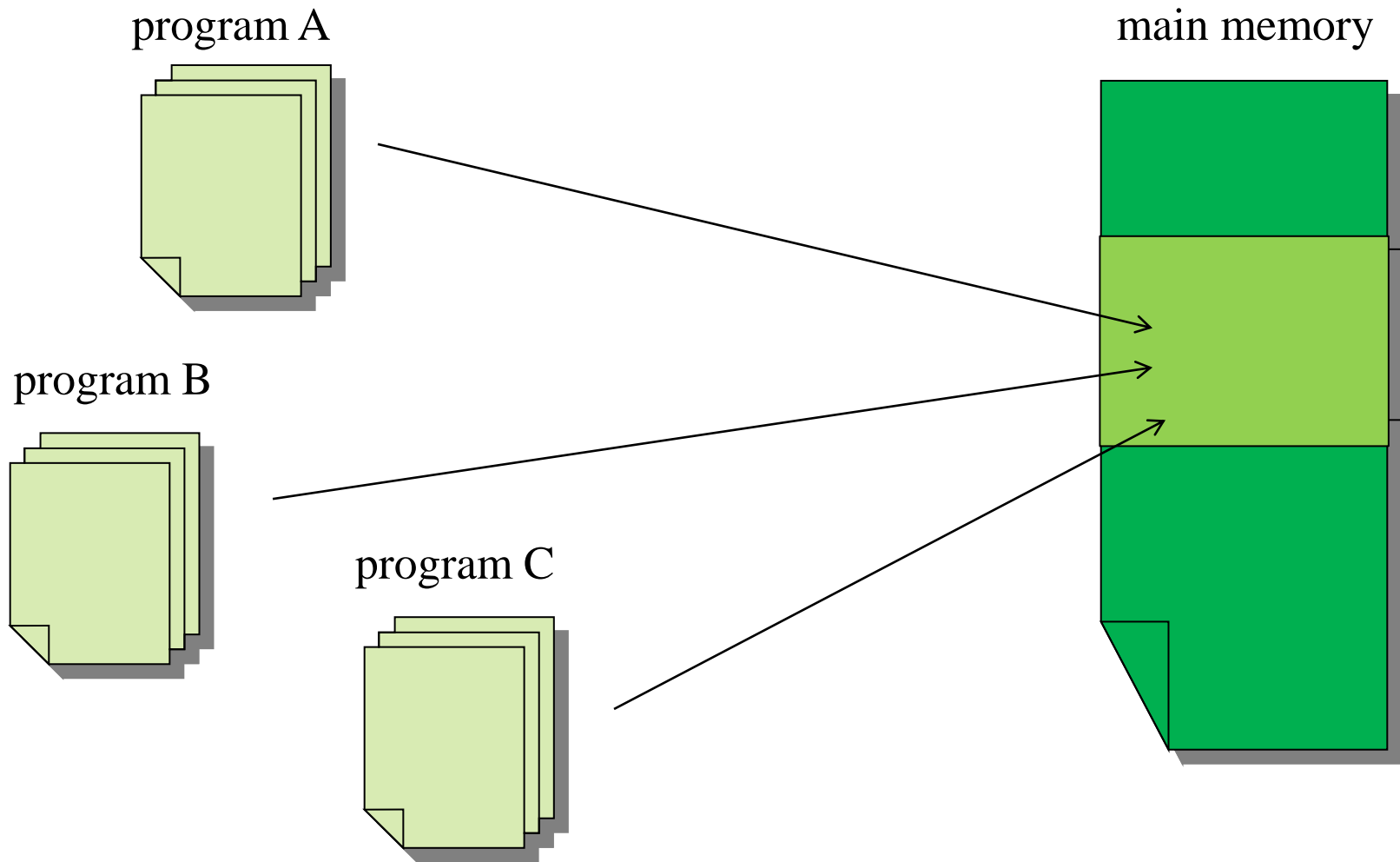


*虚拟存储器 (3)

- 交换空间 - 那些暂时用不到的部分存储在辅存的块中
- 页 – 数据和指令块
 - 页通常比较大。
 - 大多数系统采用固定大小的页，从4~16kb不等。



*虚拟存储器 (4)





*虚拟页号

- 在编译程式时，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映射成物理地址的表。
- 页表就用来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

表 2.2 虚拟地址分为两部分：虚拟页号
和页内字节偏移量



*转译后备缓冲区(TLB)

- 使用页表会增加程序总体的运行时间。
- 处理器中的一种专门用于地址转换的缓存。



*转译后备缓冲区 (2)

- TLB在快速存储介质中缓存了一些页表的条目（通常为16~512条）。
- 页面失效 – 假如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，那么这次访问成为页面失效。



指令级并行 (ILP)

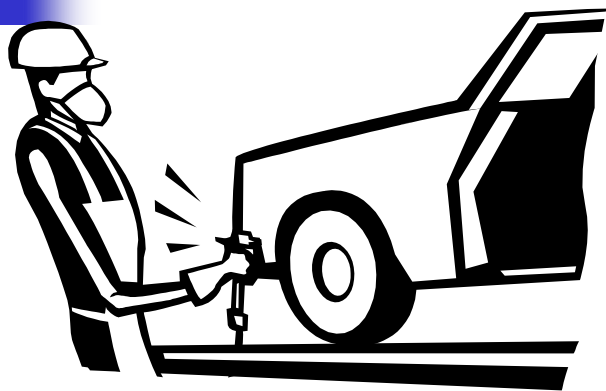
- 通过让多个处理器或者功能单元同时执行指令来提高处理器的性能。



指令级并行 (2)

- 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





流水线的例子(1)

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

Add the floating point numbers
 9.87×10^4 and 6.54×10^3



流水线的例子 (2)

```
float x[1000], y[1000], z[1000];
```

```
. . .
```

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

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

- 如果每次操作花费1纳秒 (10^{-9} 秒).
- 那么for循环需要花费7000纳秒



流水线 (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。
- 假设一个功能单元的输出是下面一个功能单元的输入。

流水线 (4)

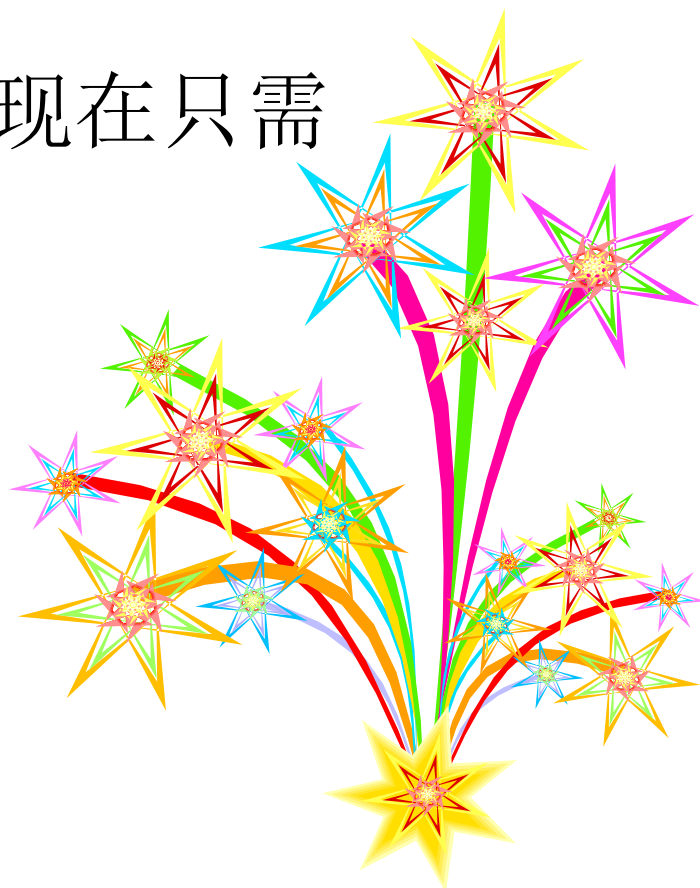
Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004				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

表 2.3: 流水线加法。

表格中的数字表示操作数/结果的下标。

流水线 (5)

- 一次浮点数加法花费7纳秒时间。
- 但是1000次浮点数加法现在只需要1006纳秒的时间了！

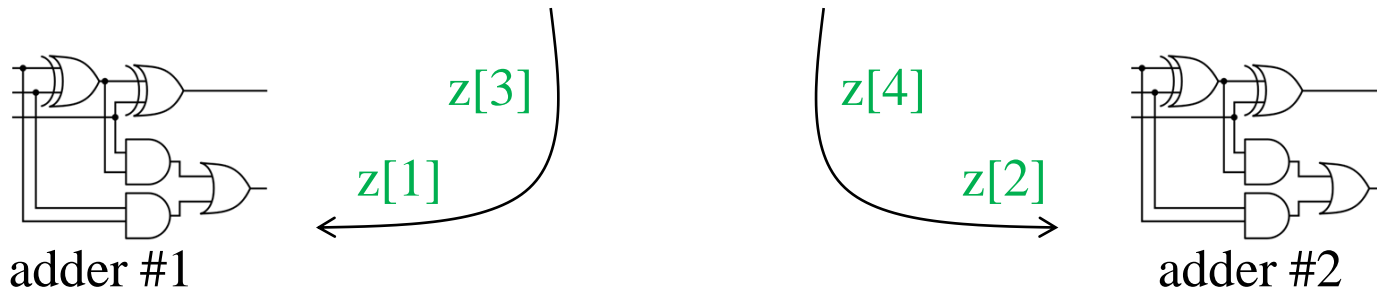


多发射 (1)

- 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];
```





多发射 (2)

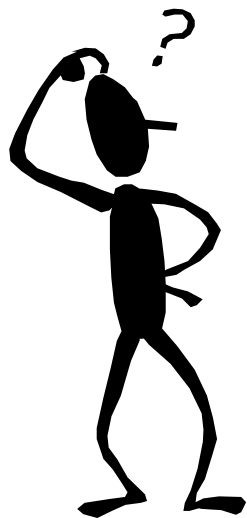
- **静态**多发射 - 功能单元是在编译时调度的。
- **动态**多发射 - 功能单元是在运行时间调度的。

超标量



超标量 (1)

- 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。

`z = x + y ;`

`a = *a_p;`

超标量 (2)

```
z = x + y ;
```

```
if ( z > 0)
```

```
    w = x ;
```

```
else
```

```
    w = y ;
```

z 可能
为正数



如果系统预测不正确，它必须返回并重新计算 $w = y$ 。



硬件多线程 (1)

- 同时执行不同线程的机会并不总是令我们满意的。
 - 例如，斐波那契数列的计算没有可同时执行的指令
- 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。
 - 例如，当前任务需要等待数据从内存中读出



硬件多线程 (2)

- **细粒度** - 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。
 - 优点: 能够避免因为阻塞而导致机器时间的浪费.
 - 缺点: 执行很长一段指令的线程在执行每条指令的时候都需要等待



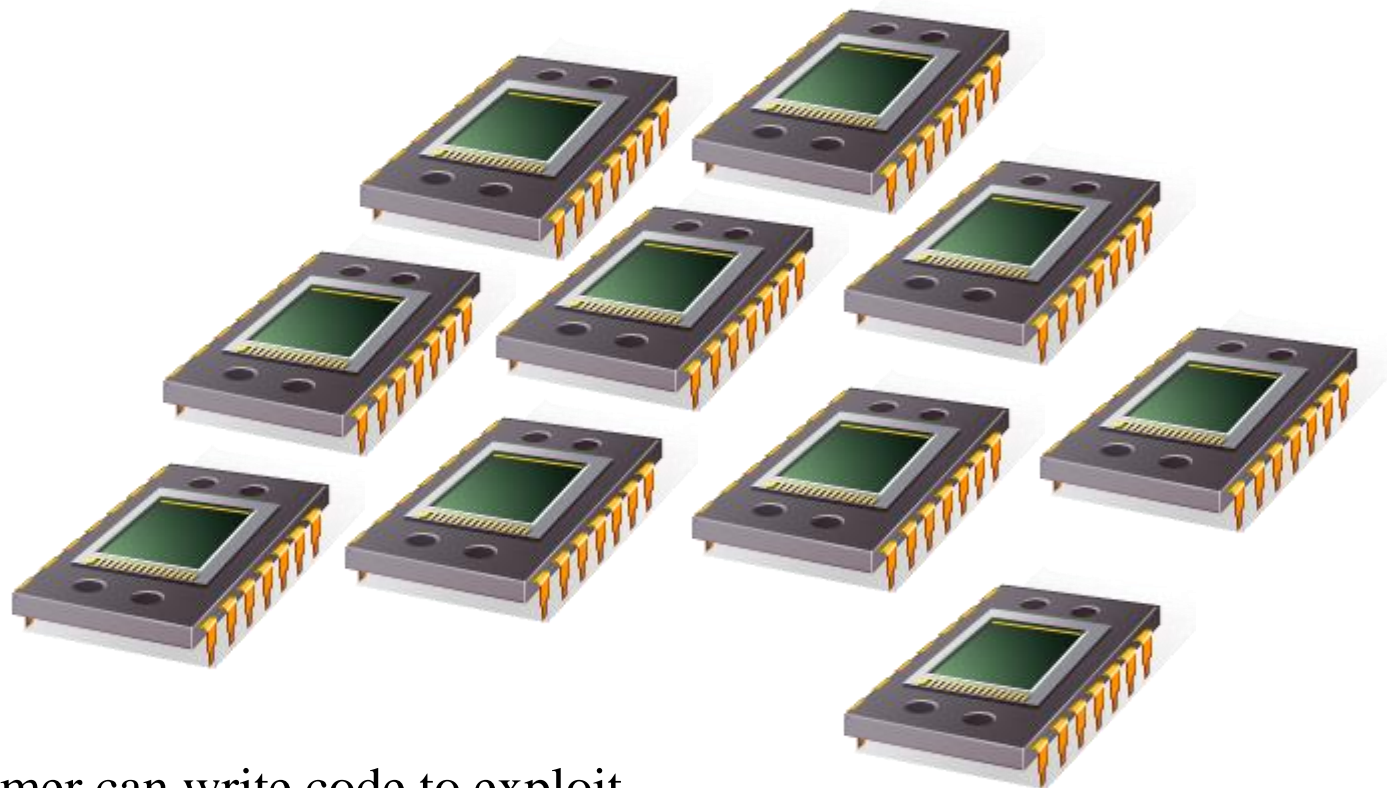
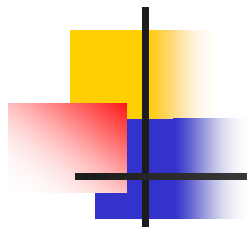
硬件多线程 (3)

- **粗粒度** - 只切换那些需要等待较长时间才能完成操作而被阻塞的线程
 - 优点: 不需要线程间的立即切换。
 - 缺点: 处理器还是可能在短阻塞时空闲，线程间的切换会导致延迟。



硬件多线程 (3)

- 同步多线程(SMT) – 是细粒度多线程的变种。
- 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.

并行硬件



Flynn's 分类法

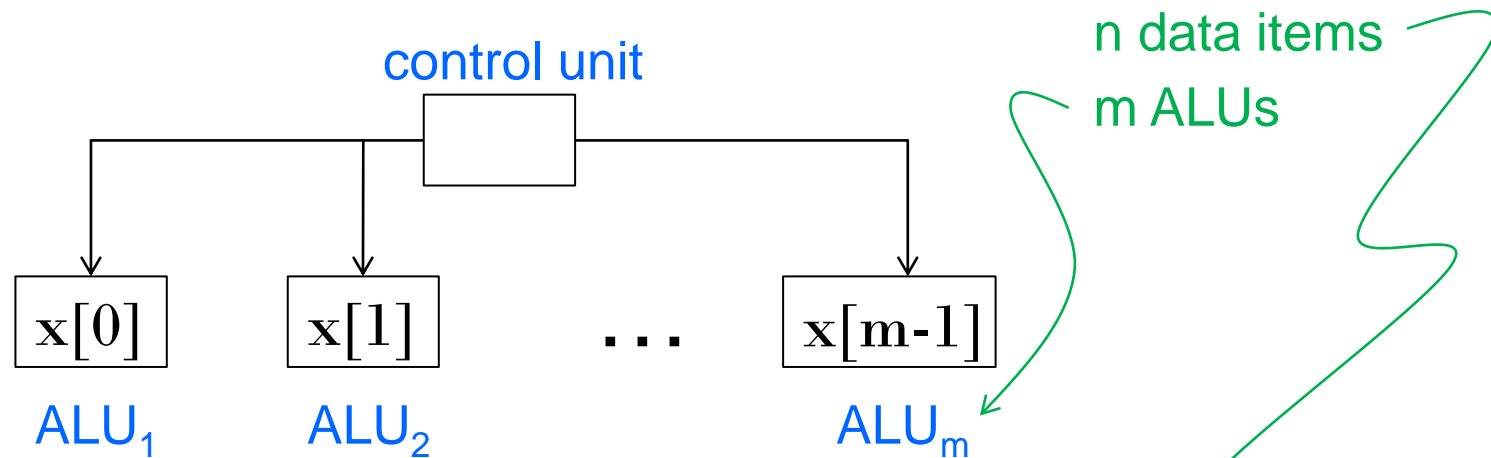
<i>classic von Neumann</i> SISD 单指令单数据流 (Single instruction stream Single data stream)	(SIMD) 单指令多数据流 (Single instruction stream Multiple data stream)
MISD 多指令单数据流 (Multiple instruction stream Single data stream) <i>not covered</i>	(MIMD) 多指令多数据流 (Multiple instruction stream Multiple data stream)



SIMD

- 通过将数据分配给多个处理器实现并行化。
- 使用相同的指令来操作数据子集。
- 这种并行称为数据并行。

SIMD 例子



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




SIMD

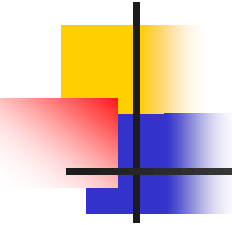
- 如果我们没有像数据项一样多的 ALU 怎么办?
- 迭代地划分任务和流程。
- 如: $m = 4$ ALUs and $n = 15$ data items.

Round3	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]	



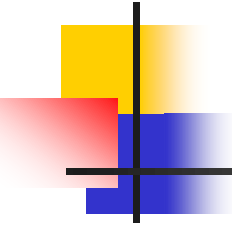
SIMD 缺点

- 所有的ALU要么执行相同的指令，要么同时处于空闲状态。
- 在“经典”的SIMD系统中，ALU必须同步操作。
- ALU没有指令存储器。
- SIMD并行性在大型数据并行问题上非常有用，但是在处理其他并行问题时并不优秀。



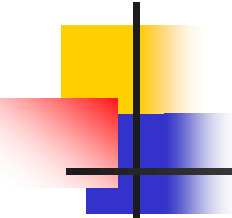
向量处理器 (1)

- 向量处理器对数组或者数据向量进行操作，而传统地CPU是对单独的数据元素或者标量进行操作。
- 向量寄存器
 - ▣ 它是能够存储由多个操作数组成的向量，并且能够同时对其内容进行操作的寄存器。



向量处理器 (2)

- 向量化和流水化的功能单元
 - 对向量中的每个元素需要做同样的操作，这些操作需要应用到两个或以上对应元素上。
- 向量指令
 - 在向量上操作而不是在标量上操作。



向量处理器 (3)

○ 交叉存储器

- 内存系统由多个内存‘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，那么在装入/存储连续数据时能够几乎无延迟访问。

○ 步长式存储器访问和硬件的散射/聚集

- 程序能够访问向量中固定间隔的元素。

向量处理器的优点



- 速度快。
- 容易使用。
- 向量编译器擅长于识别向量化的代码。
- 编译器也能提供代码为什么不能向量化的原因。
 - ▣ 帮助程序员重新评估代码。
- 很高的内存带宽。
- 每个加载的数据都会使用。

向量处理器的缺点

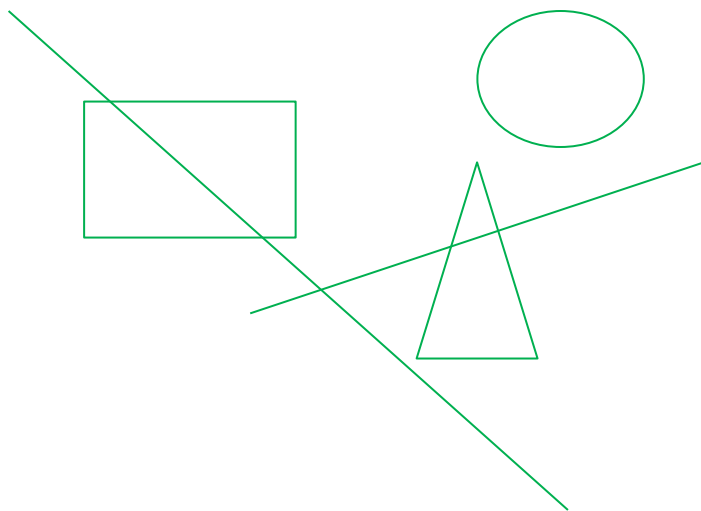
- 它不能处理不规则的数据结构和其他的并行结构。
- 它的可扩展性是个限制。可扩展性是指能够处理更大问题的能力。





*图形处理单元 (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

- GPU 使用图形处理流水线将物体表面的内部表示转化为一个像素的数组。这个像素数组能够在计算机屏幕上显示出来。
- 流水线的许多阶段是（称为着色函数）可编程的。
 - 典型的着色函数一般比较短，通常只有几行C代码。





GPUs

- 着色函数一般是隐式并行的，因为它们能够应用到图形流中的多种元素（例如顶点）上。
- GPU常使用SIMD 来优化性能
- The current generation of GPU's use SIMD parallelism.
- 现在所有的GPU都使用SIMD并行
 - ▣ 尽管它们不是纯粹的SIMD系统



MIMD

- 支持同时多个指令流在多个数据流上操作。
- 通常包括一组完全独立的处理单元或者核，每个处理单元或者核都有自己的控制单元和ALU。

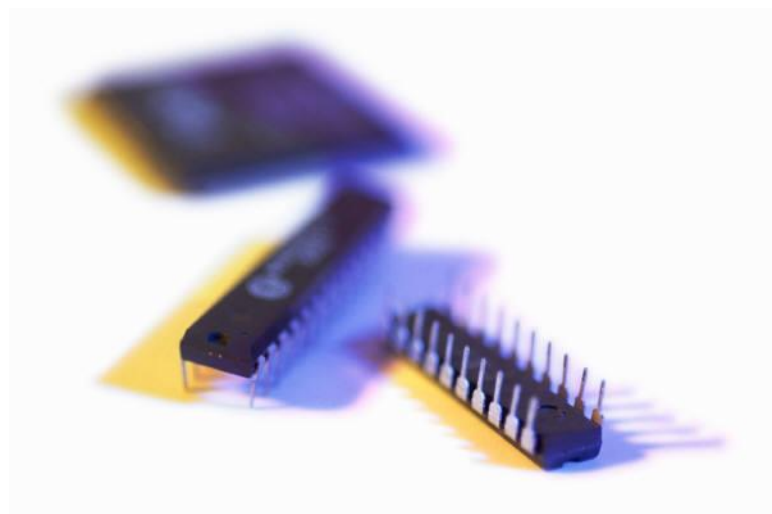
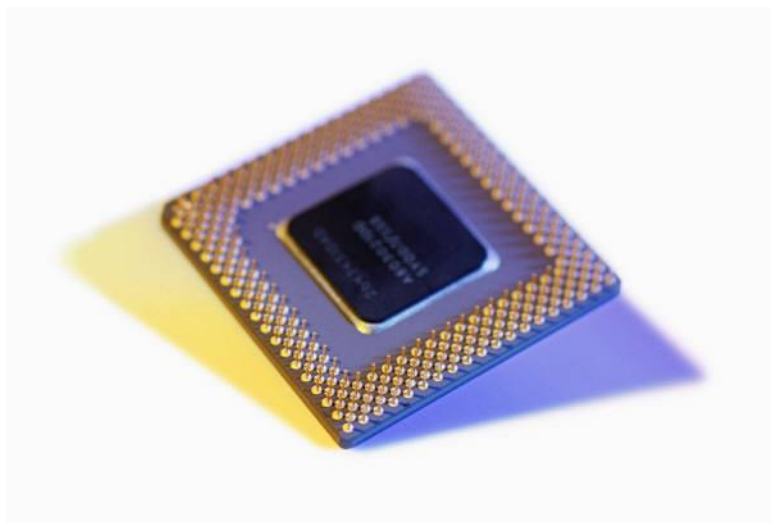


共享内存系统 (1)

- 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。
- 处理器通过访问共享的数据结构来隐式地通信。

共享内存系统 (2)

- 最广泛使用的共享内存系统使用一个或者多个多核处理器。
 - (在一块芯片上有多个CPU或者核)



共享内存系统

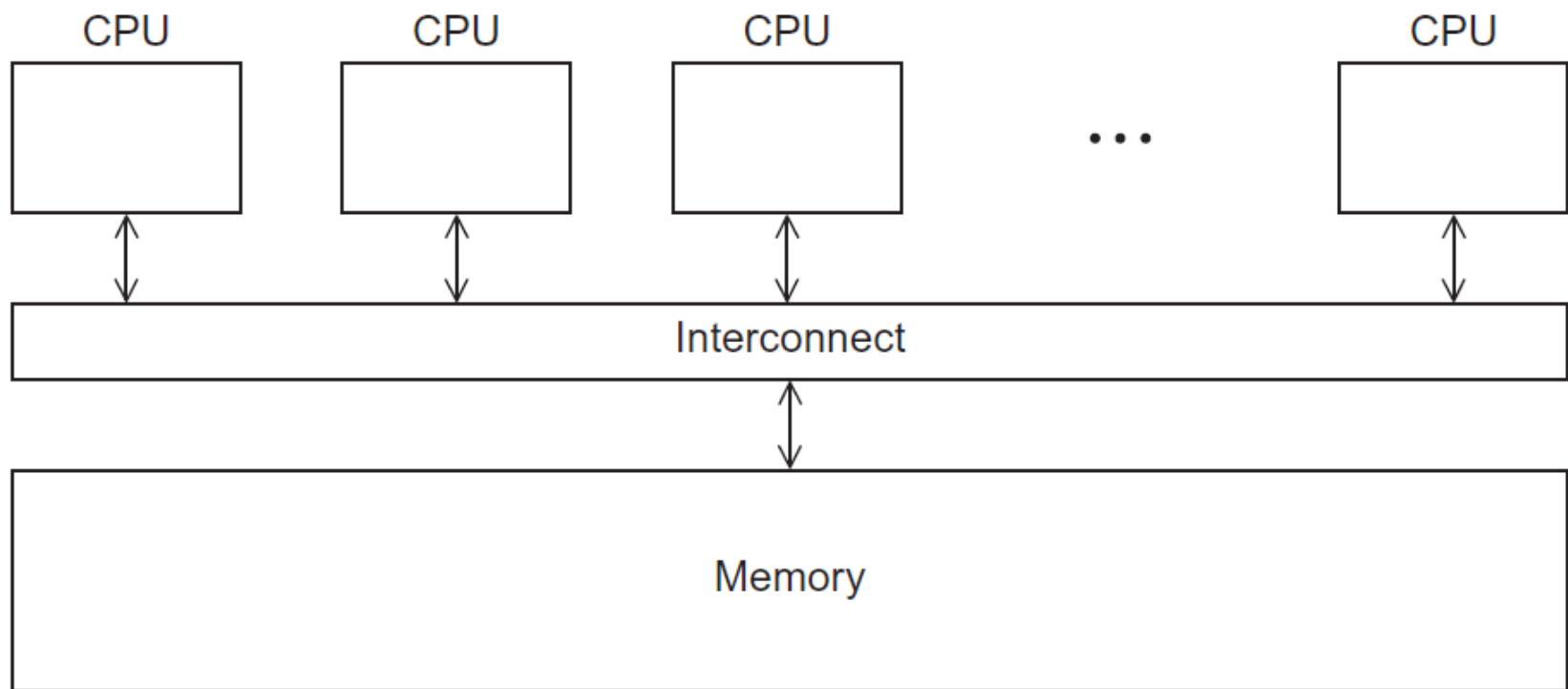


图 2.3

UMA 一致内存访问系统

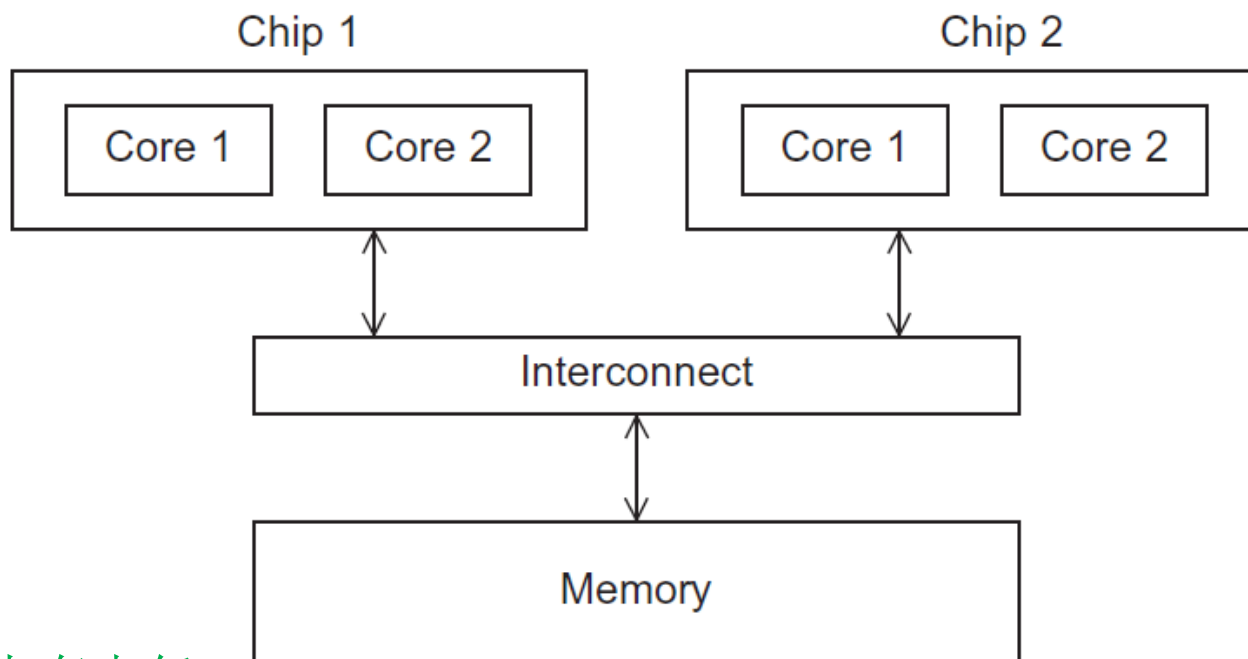
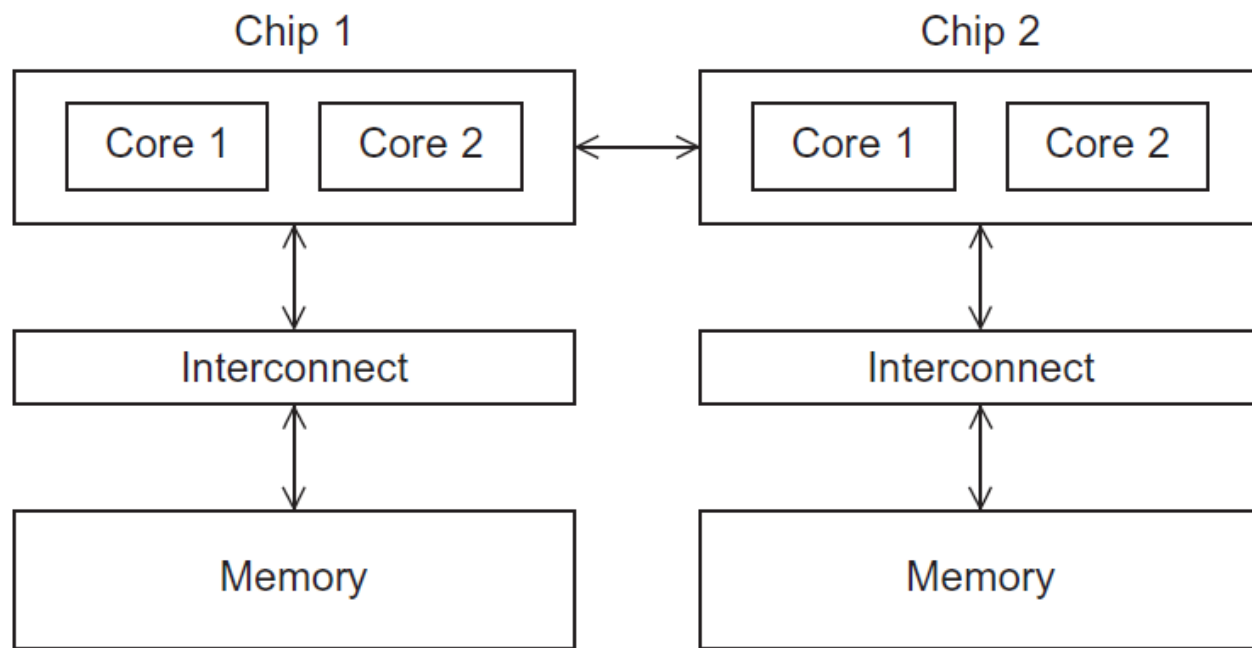


图 2.5

每个核访问内存中任何一个区域的时间都相同。

NUMA 非一致内存访问系统



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。

图 2.6



分布式内存系统

- 集群 (最广泛使用)
 - 由一组商品化系统组成。
 - 通过商品化网络连接。
- 这些系统中的节点是通过通信网络相互连接的独立计算单元。

a.k.a. hybrid systems

分布式内存系统

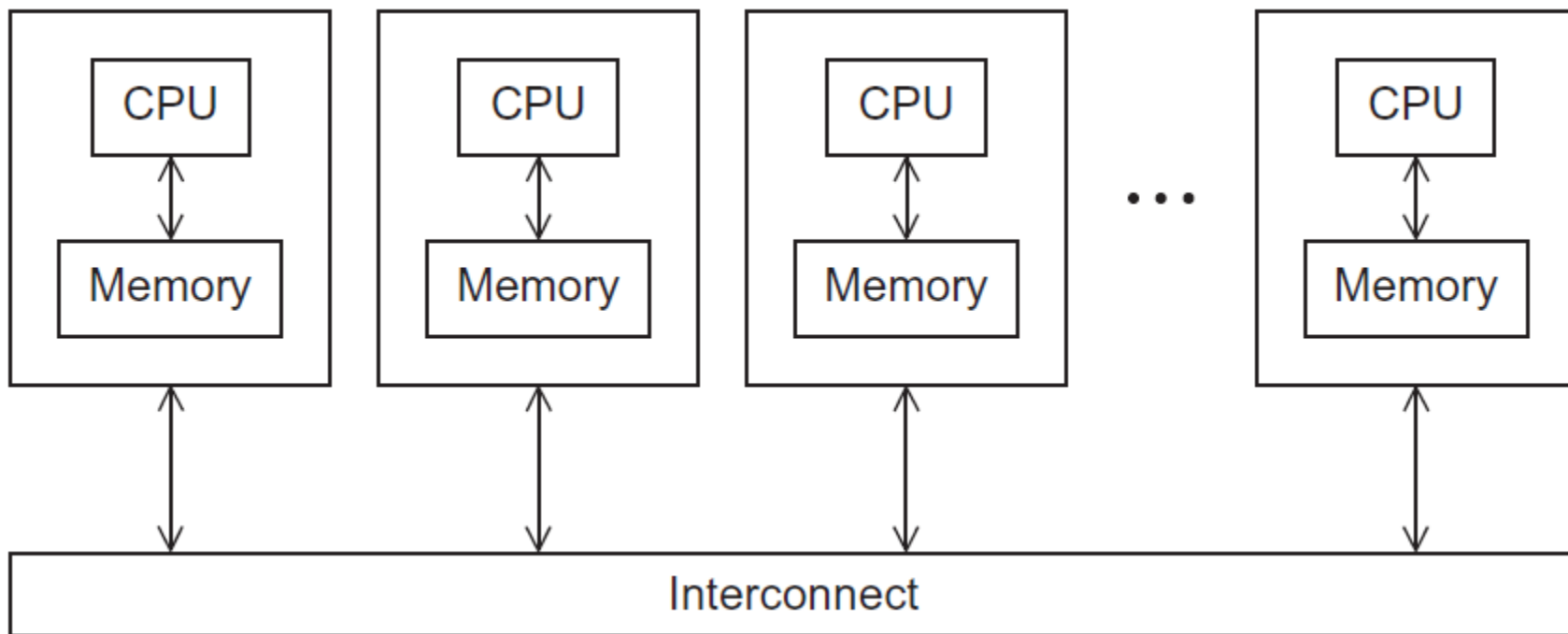


图 2.4



互连网络

- 在分布式内存系统和共享内存系统中都扮演了一个决定性的角色
- 分为两类:
 - 共享内存互连网络
 - 分布式内存互连网络



共享内存互连网络

○ 总线互连

- 总线是由一组并行通信线和控制对总线访问的硬件组成的
- 总线的核心特征是连接到它的设备共享通信线路
- 随着连接到总线的设备数量的增加，对总线的使用的竞争会增加，性能会下降。



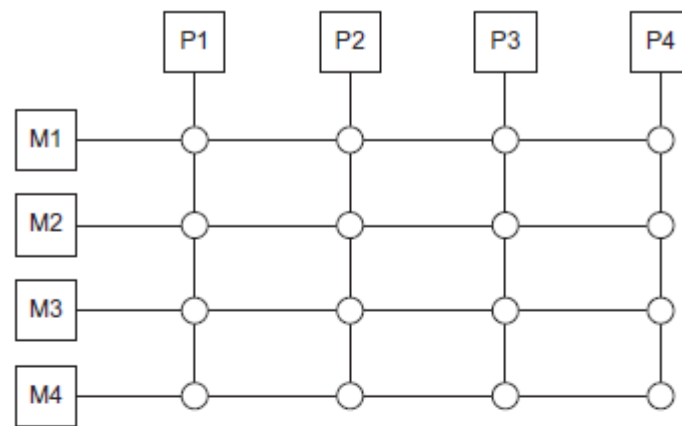
共享内存互连网络

○ 交换互连网络

- 使用交换器来控制相互连接设备间的数据传递。
- 交叉开关矩阵 (Crossbar)
 - 允许不同设备之间同时进行通信。
 - 比总线速度快。
 - 但是交换器和链路带来的开销也相对高。

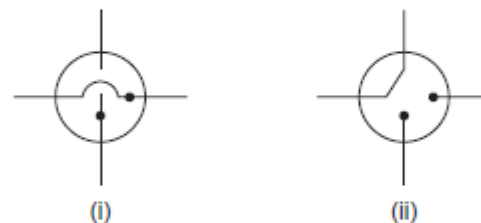
图 2.7

(a) 一个连接4个处理器(P_i)和4个内存模块(M_j)的交叉开关矩阵



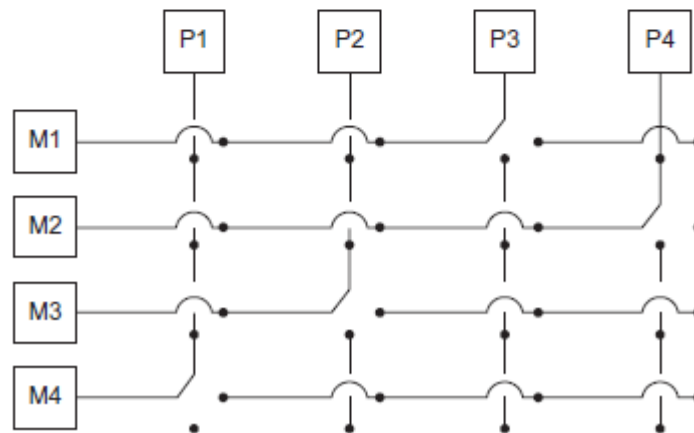
(a)

(b) 交叉开关矩阵内部的交换器



(b)

(c) 多个处理器同时访问内存



(c)



分布式内存互连网络

○ 通常分成两种

□ 直接互连

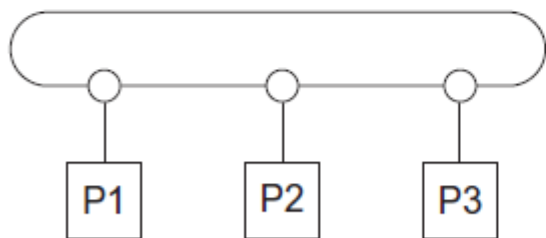
- 每个交换器与一个处理器-内存对直接相连，交换器之间也互相连接。

□ 间接互连

- 交换器不一定与处理器直接连接。

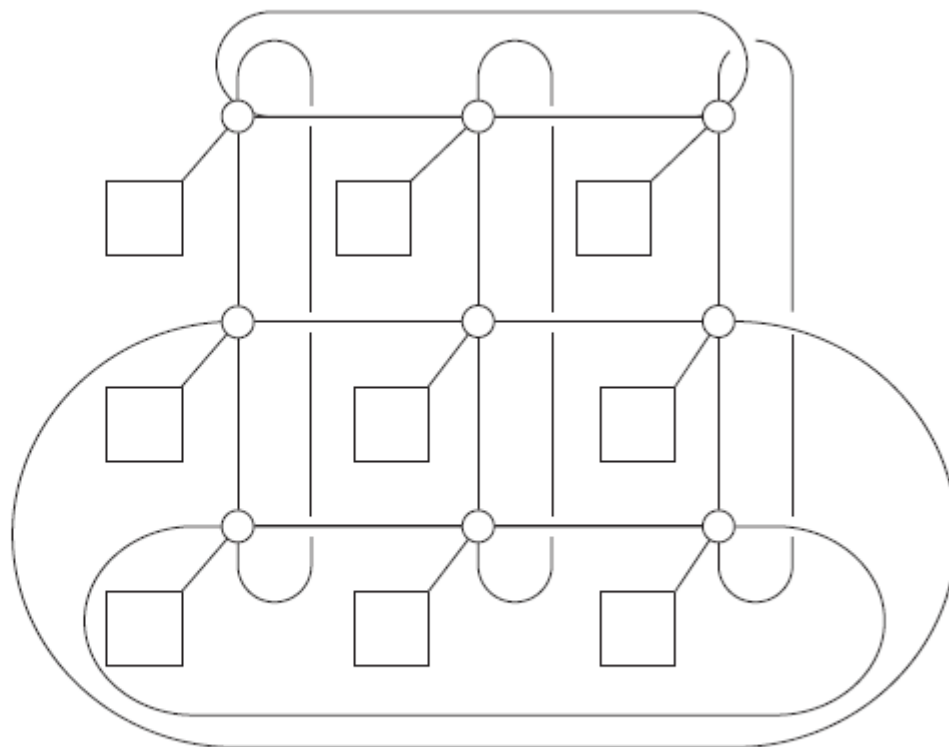
直接互连

图 2.8



(a)

一个环



(b)

一个二维环面网络

等分宽度

- 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。在这两部分之间能同时发生多少通信呢？
 - 计算等分带宽另一种方法，计数去除最少的链路数从而将节点分成两份

一个环的两种等分

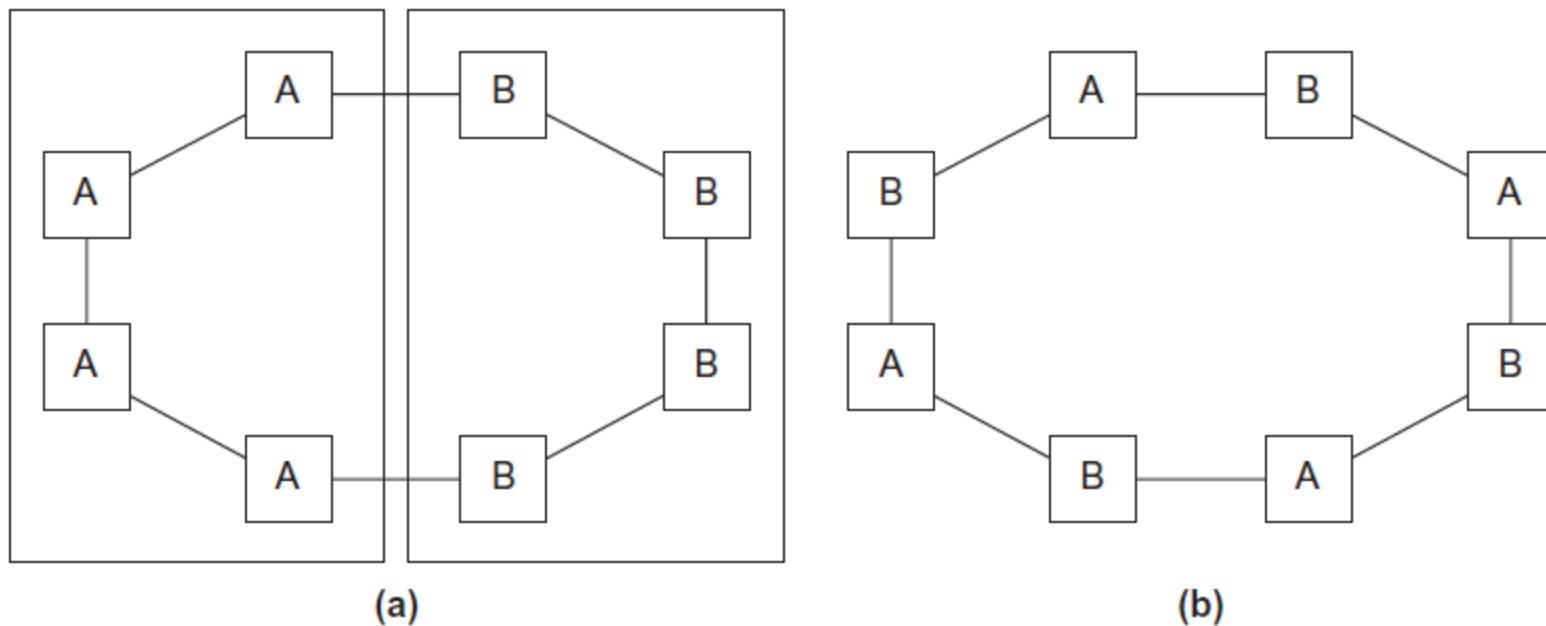


图 2.9

等分宽度是基于最坏情况来估计的。

一个二维环面网格的等分

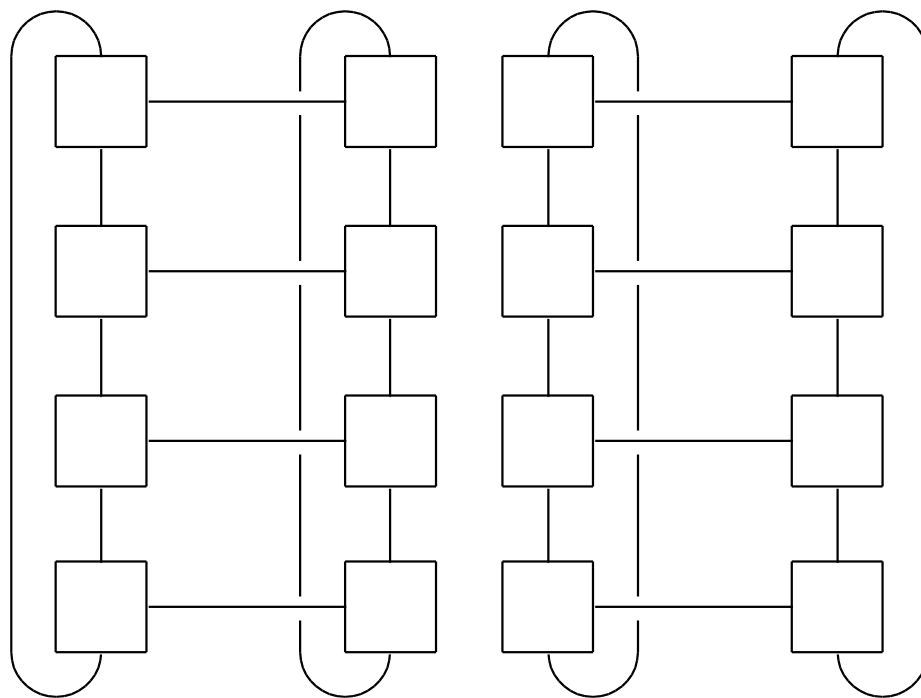


图 2.10



定义

○ 带宽

- 指它传输数据的速度。
- 通常用兆位每秒或者兆字节每秒来表示。

○ 等分带宽

- 用来衡量网络的质量。
- 不是计算连接两个等分之间的链路数，而是计算链路的带宽。
 - 如果在环中，链路的带宽是10亿位每秒，等分带宽就是20亿位每秒

全相连网络

- 每个交换器与每一个其他的交换器直接连接。

不切实际的

等分宽度 = $p^2/4$

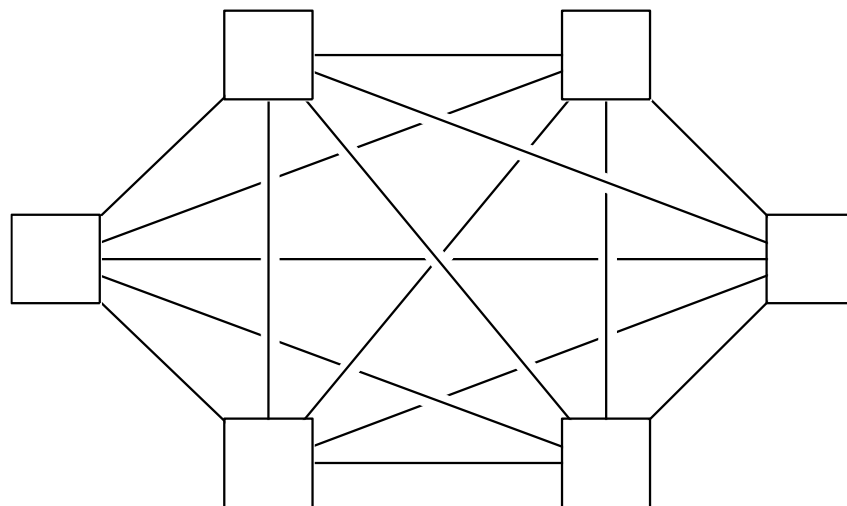


图 2.11

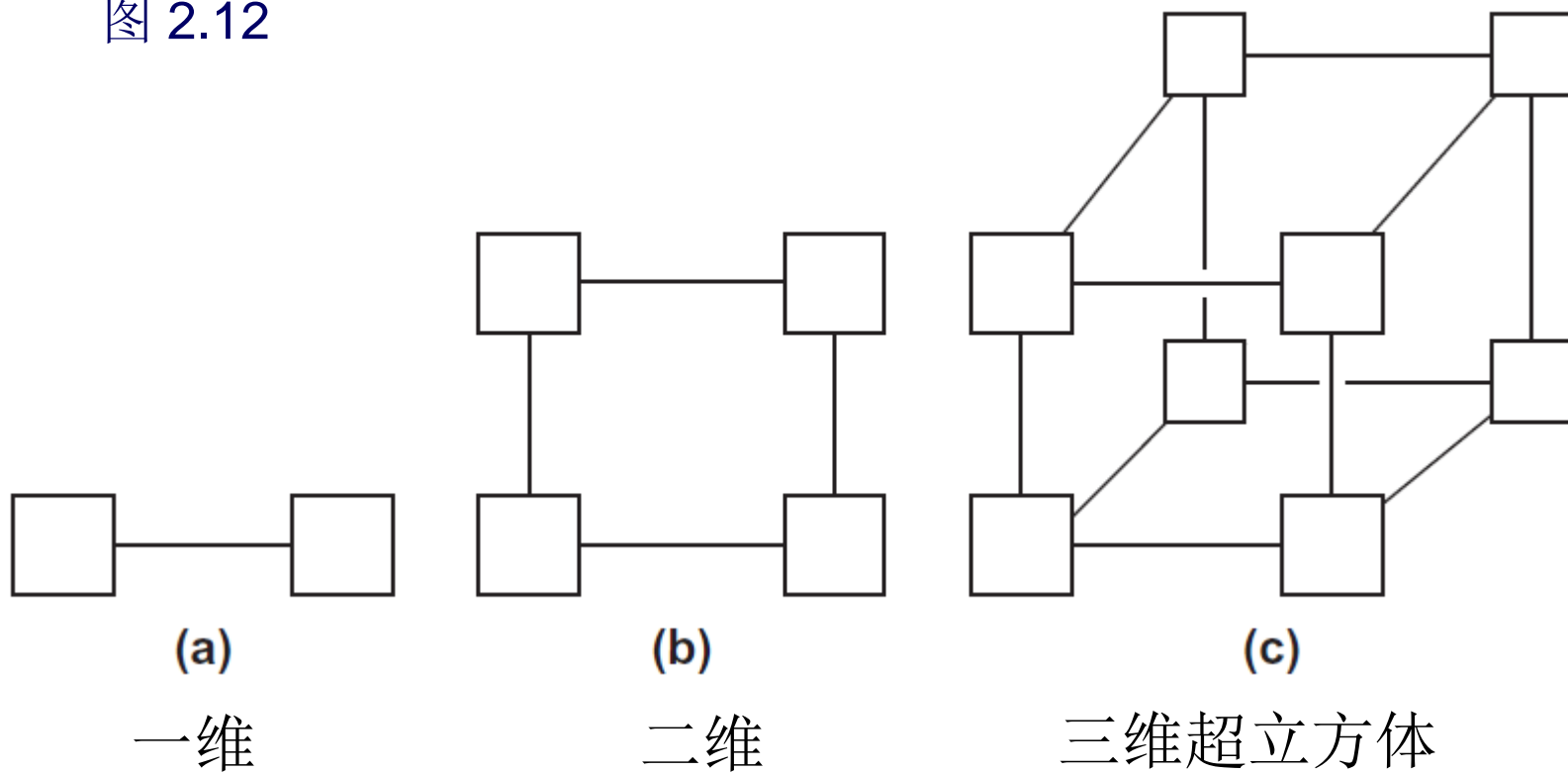


超立方体

- 高度互连的直接互连网络。
- 递归构造的：
 - 一维超立方体是有两个处理器的全互连系统。
 - 二维超立方体是由两个一维超立方体组成，并通过“相应”的交换器互连。
 - 类似地，一个三维超立方体是由两个二维超立方体组成的。

超立方体

图 2.12





*间接互连

- 间接网络中相对简单的例子:
 - 交叉开关矩阵
 - Omega 网络
- 通常由一些单向连接和一组处理器组成，每个处理器有一个输入链路和一个输出链路，这些链路通过一个交换网络连接。

一个通用的间接网络

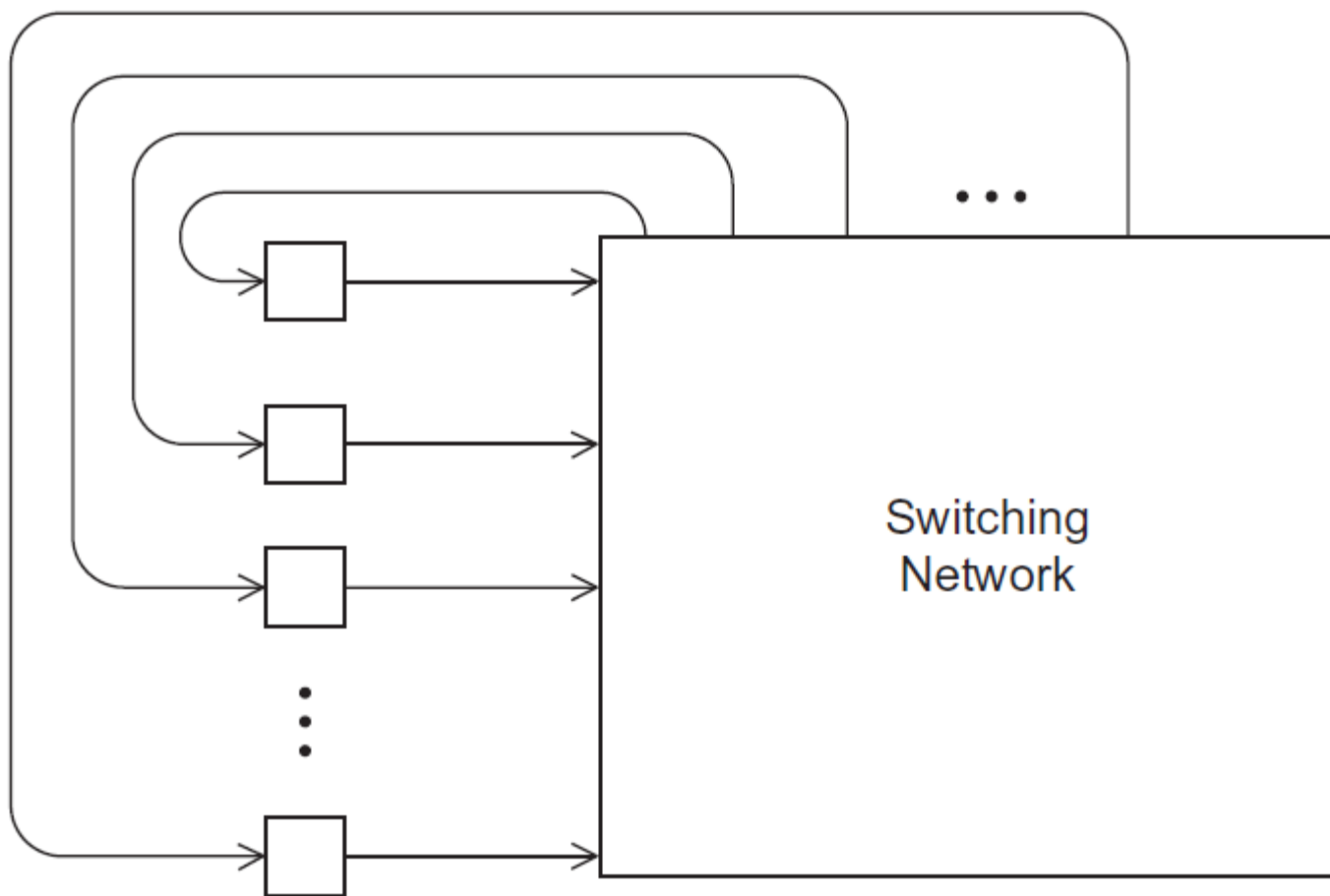


图 2.13

一个用于分布式内存的交叉开关矩阵互连网络

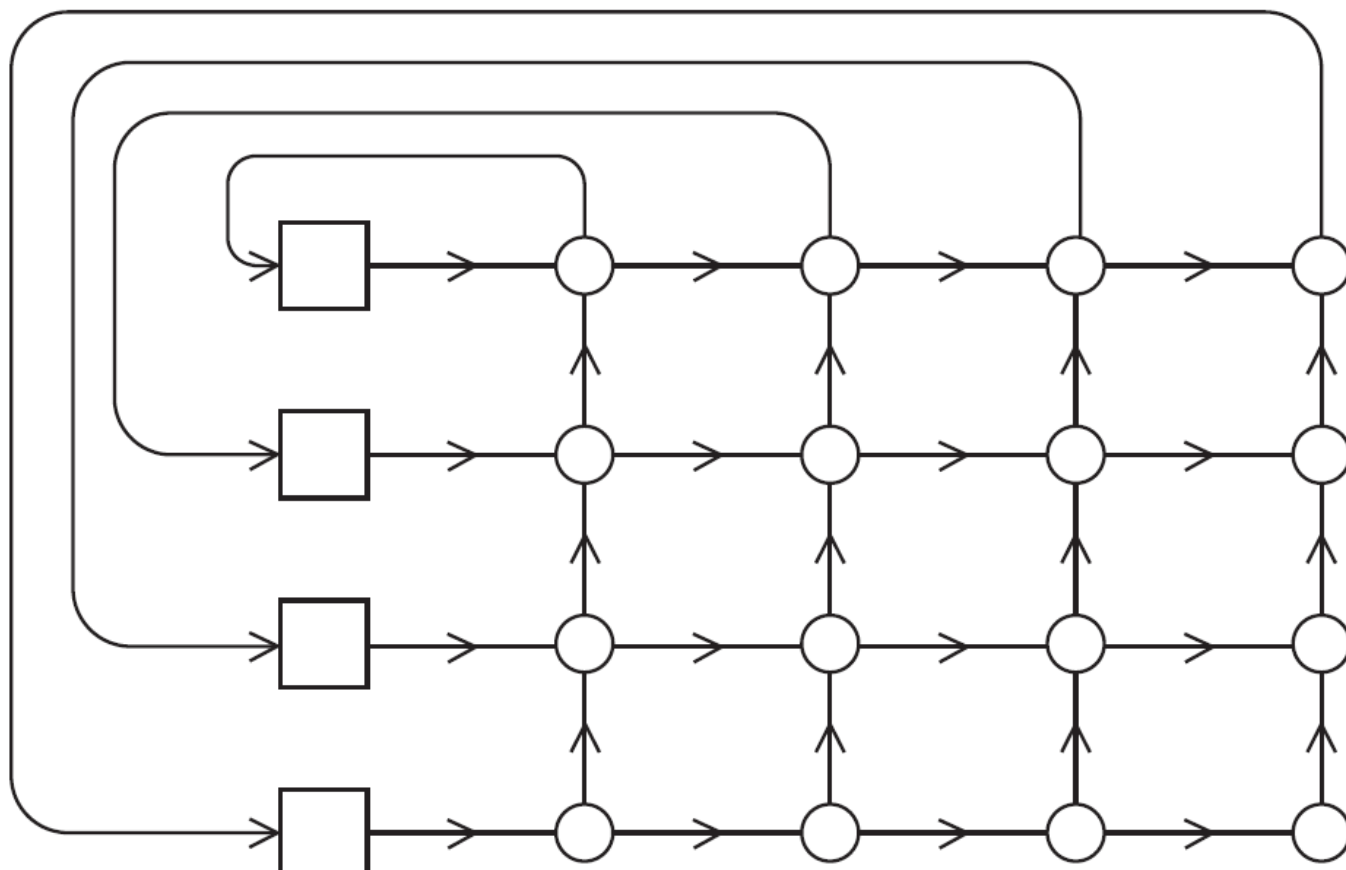


图 2.14

一个 omega 网络

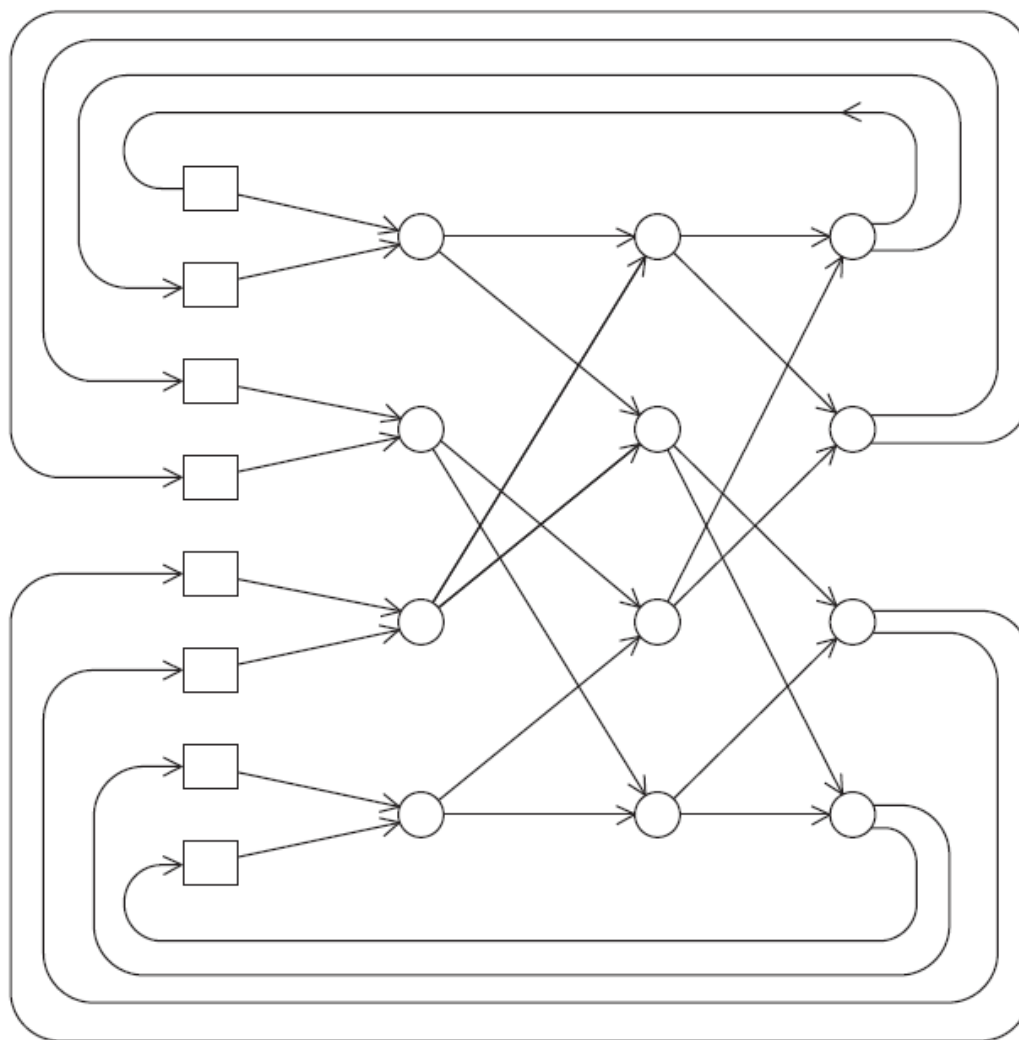


图 2.15

omega 网络中的一个交换器

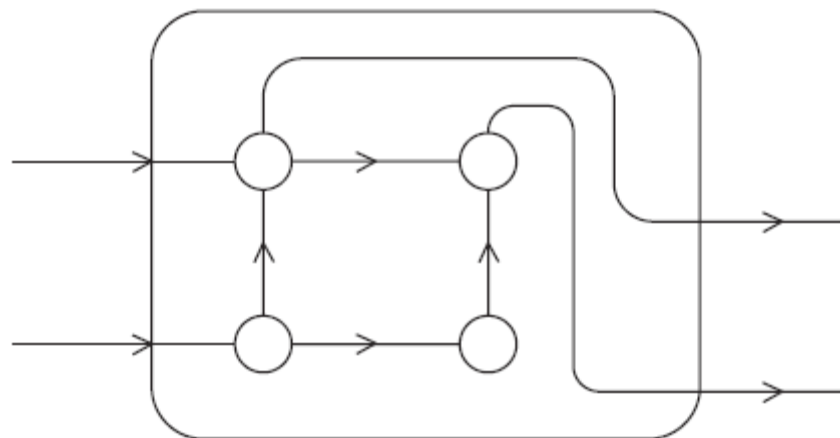


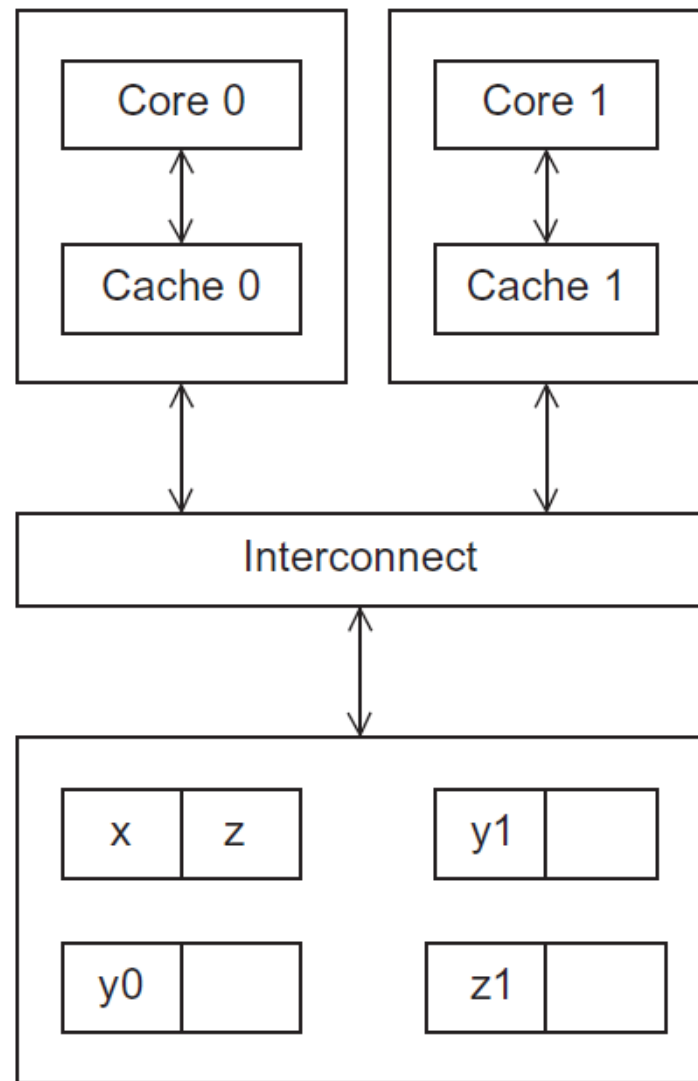
图 2.16

Cache 一致性

- 程序员无法直接控制缓存及其更新时间。

图 2.17

有两个核和两个Cache的共享内存系统





Cache 一致性

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

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;

y0 eventually ends up = 2

y1 eventually ends up = 6

z1 = ???



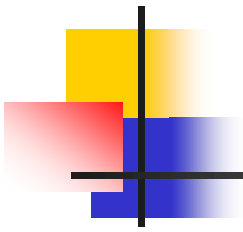
监听Cache一致性协议

- 多个核共享总线。
- 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。
- 假如核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的副本标记为非法的。



基于目录的Cache一致性协议

- 使用一个叫做目录的数据结构，目录存储每个内存行的状态。
- 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高速缓存行置为非法。



PARALLEL SOFTWARE

——并行软件



共享内存编程fork派生 join合并

○ 动态线程

- 主线程等待计算工作，fork新线程分配工作，工作线程完成任务后结束
- 资源高效利用，但线程创建/结束非常耗时

○ 静态线程

- 创建线程池，并向其中线程分配任务，但线程不结束，直至整个程序结束
- 性能更优，但可能浪费系统资源



线程安全/同步

- 共享内存并行函数或库的线程安全问题
 - 对于一个函数或库，若多线程并行调用能“正确”执行，则称它是**线程安全的**
 - 由于多线程通过共享内存通信、协调，因此线程安全的代码可使用恰当的同步操作修改共享内存状态
 - 某些串行代码的特性可能不是线程安全的？



并行程序设计的复杂性

- 足够的并发度（Amdahl定律）
- 并发粒度
 - 独立的计算任务的大小
- 局部性
 - 对临近的数据进行计算
- 负载均衡
 - 处理器的工作量相近
- 协调和同步
 - 谁负责？ 处理频率？



并行算法分析

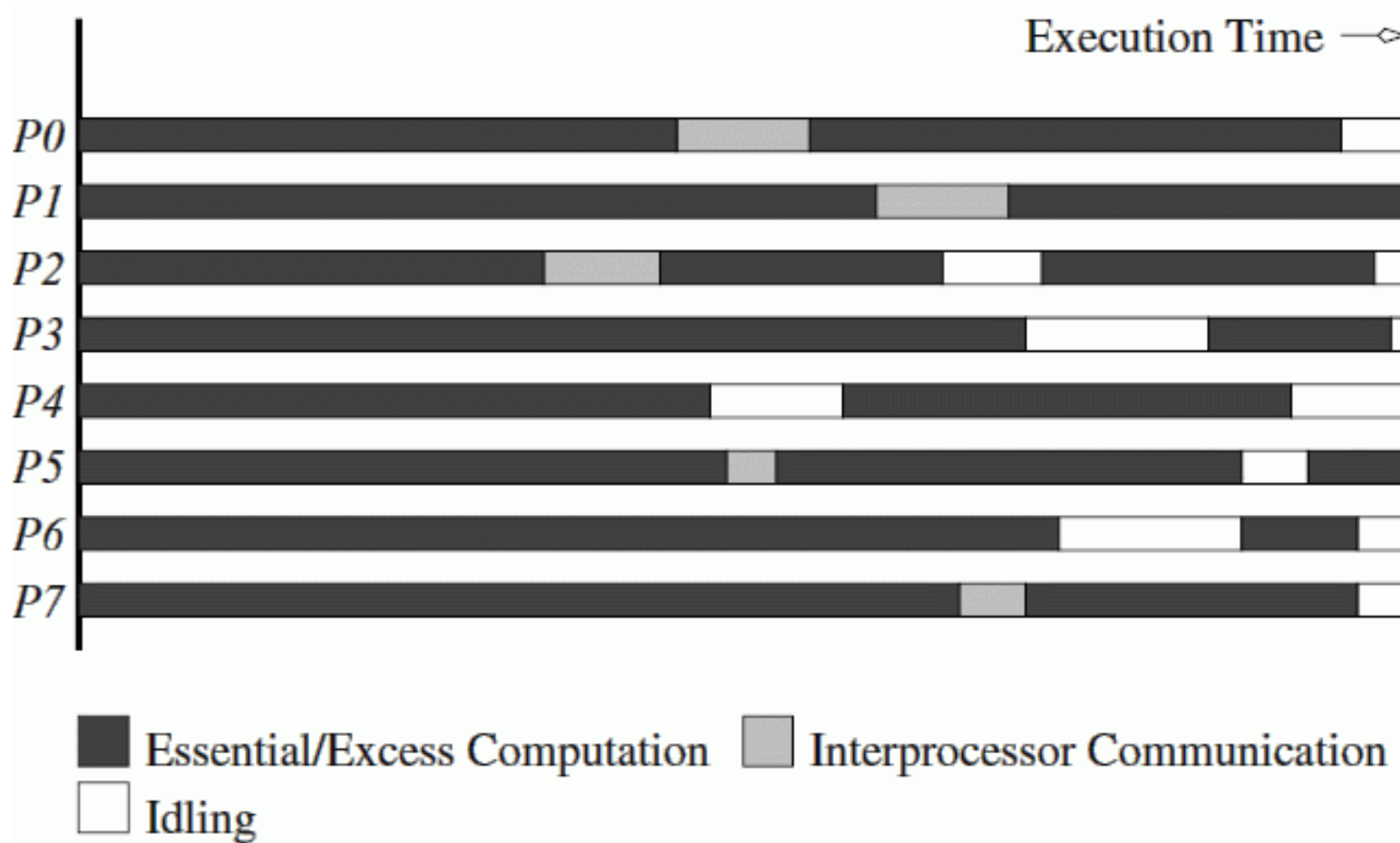
- 串行算法评价：算法时间复杂度表示为输入规模的函数
- 并行算法评价：除了输入规模之外，还应考虑处理器数目、处理器相对运算速度、通信速度



并行算法额外开销

- 除了串行算法要做的之外的工作
 - 线/进程创建、协调、合并
 - 线/进程间通信：最大开销，大部分并行算法都需要
 - 线/进程空闲：负载不均、同步操作、不能并行化的部分
 - 额外计算
 - 最优串行算法难以并行化，将很差的串行算法并行化，并行算法计算量>最优串行算法
 - 最优串行算法并行化也会产生额外计算：并行快速傅立叶变换，旋转因子的重复计算

并行算法额外开销（续）





并行算法性能评价标准

○ 运行时间

- 串行算法: T_s , 算法开始到结束的时间流逝
- 并行算法: T_p , 并行算法开始到最后一个进程结束所经历时间

○ 并行算法总额外开销

- $T_o = pT_p - T_s$

○ 加速比

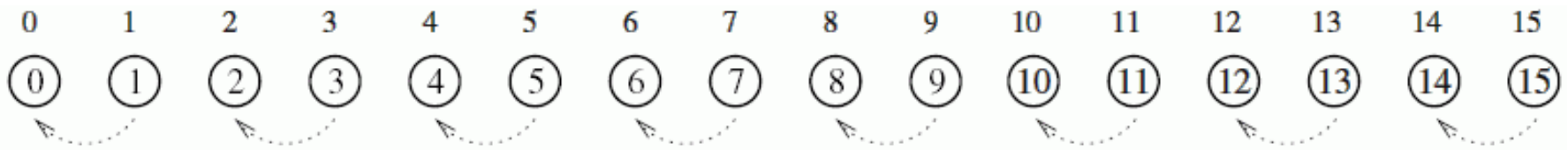
- $S = T_s / T_p$



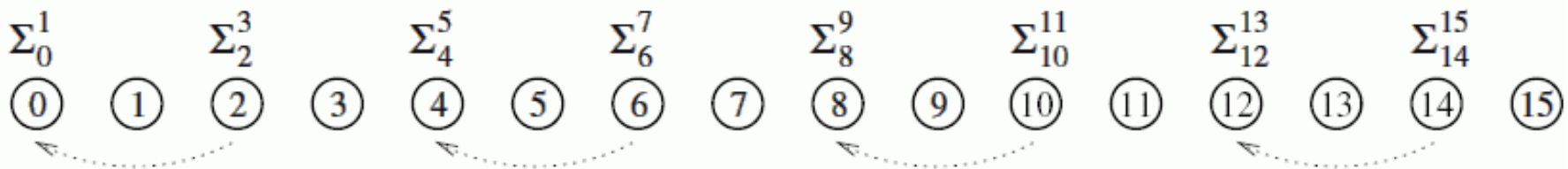
加速比

- 对比哪个串行算法？同一问题，可能存在多个串行算法，时间复杂度和并行程度可能都不一样
- 应选择“最优”串行算法
- 理论最优算法若不存在或难实现——选择已有（且可行）的算法中最优者
- $S = \text{最优串行算法时间} / \text{并行算法时间}$
——并行算法运行于 p 个处理器的并行平台，每个处理器与运行串行算法的处理器完全相同

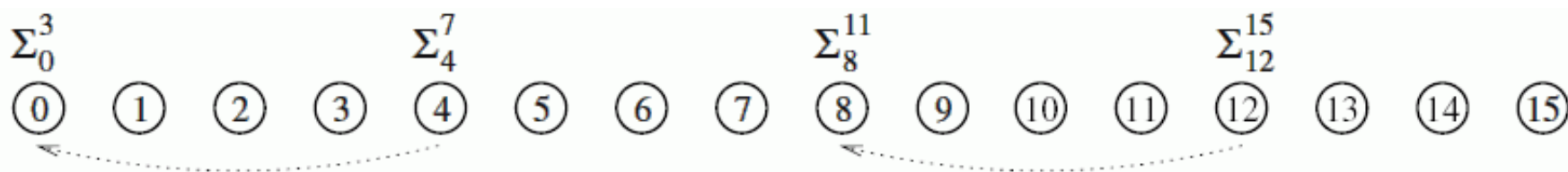
例1: n个数相加, n个进程



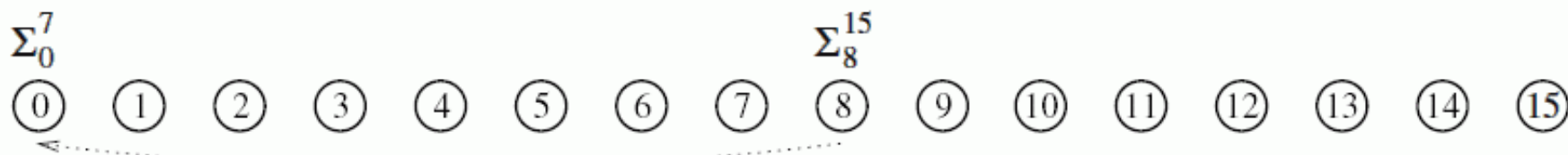
(a) Initial data distribution and the first communication step



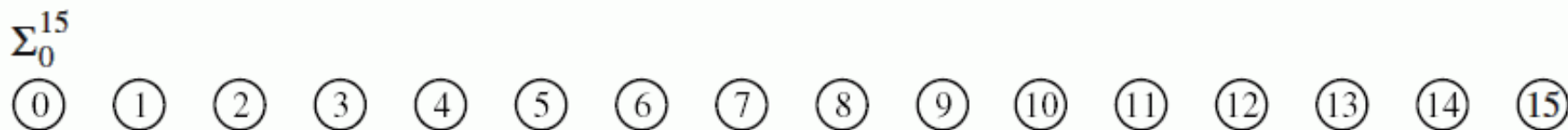
例1: (续)

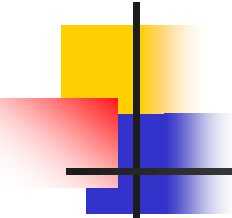


(c) Third communication step



(d) Fourth communication step





例1：（续）

- 初始，每个进程保存1个数，最终由1个进程保存累加和
- 树形结构， $\log n$ 个步骤

$$\rightarrow T_S = \Theta(n), T_P = \Theta(\log n)$$

$$\rightarrow S = T_S / T_P = \Theta(n / \log n)$$

例2：加速比的计算

- 串行起泡排序算法时间150s，串行快速排序算法30s，并行起泡排序算法40s
- $S=30/40=0.75$ ，而不是 $150/40=3.75$ ！
- 一般 $S \leq p$
- $S=p$ ，则称该并行算法具有线性加速比
- * $S > p$ (超线性加速比)在实践中是可能出现的
 - 串行算法计算量 > 并行算法
 - 硬件问题不利于串行算法
 - 数据量较大，无法全部放入Cache，命中率低
 - 并行算法进行数据划分，每个处理器数据量变小，可全部放入Cache，命中率提高 → 性能提高

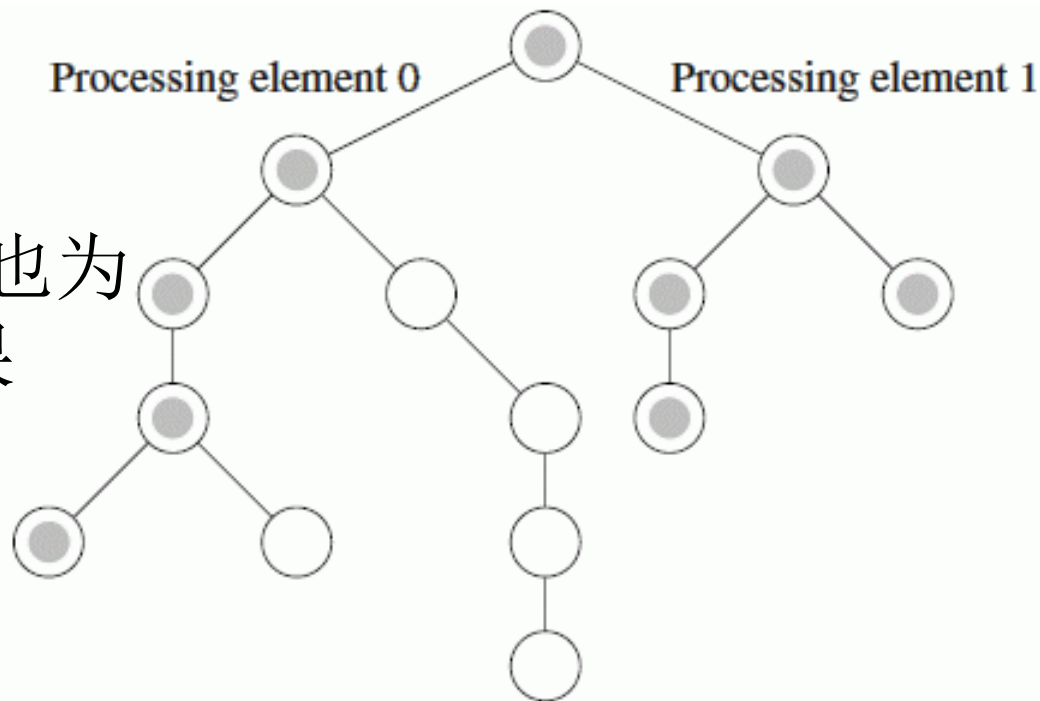


Cache引起的超线性加速

- 2个处理器的并行系统，问题规模 W
- 每个处理器由Cache 64KB，命中率80%，Cache延迟2ns，DRAM 100ns，平均访问时间 $2*0.8+100*0.2=21.6\text{ns}$
- 若计算瓶颈在内存，1个内存访问可产生1个FLO，运算速度为46.3MFLOPS
- 将任务平均分配给两个进程，规模 $W/2$ ，命中率90%，剩余8%为本地DRAM访问，2%为远端DRAM访问（400ns），平均访问时间 $2*0.9+100*0.08+400*0.02=17.8\text{ns}$ →56.18MFLOPS→2个处理器112.36MFLOPS→ $S=2.43$

搜索分解导致超线性

- 解在树的最右节点，串行算法需搜索所有节点： $14t_c$
- 并行算法： P_0 搜索左子树， P_1 搜索右子树，若速度相等，只搜索9个节点，由于并行，时间为 $4t_c$ ，加上根节点处理时间，共 $5t_c$ ，并行算法工作量少
- $S=14t_c/5t_c=2.8$
- 若串行算法交替处理两个子树节点，工作量也为9个节点，无超线性效果
- 但解的位置完全取决于问题实例，无普适的“最优串行算法”



阿姆达尔定律(Amdahl's law)

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

- $S = 1 / (1 - a + a / p)$

- a 为串行程序中可被(完美)并行化的比例

- $T_S = 1$, $T_P = T_{\text{不可并行}} + T_{\text{可并行}} = 1 - a + a/p$

- 加速比的极限是： $S = 1 / (1 - a)$





例3:

- 某串行程序运行时间为20s，可并行化比例为90%，若使用p个核将其并行化，加速比为多少？

- $T_S = 20 \text{ s}$

- $T_P = T_{\text{不可并行}} + T_{\text{可并行}}$
 $= 0.1 \times T_S + 0.9 \times T_S / p$
 $= 0.1 \times 20\text{s} + 0.9 \times 20\text{s} / p$
 $= 2\text{s} + 18\text{s} / p$

- $S = T_S / T_P = 20 / (2 + 18 / p) = 10 / (1 + 9 / p)$

- 或用公式: $S = 1 / (1 - a + a / p) = 1 / (0.1 + 0.9 / p)$



效率

- 效率（**Efficiency**）：度量有效计算时间
- $E = S / p = T_S / (p * T_P)$
- 理想情况=1，正常0~1。
 - 因为理想情况 $S = p$



效率

○ 例1, n 个核对 n 个数求和例子

□ $T_S = \Theta(n), T_P = \Theta(\log n), S = \Theta(n/\log n)$

□ $E = S/p = \Theta(n/\log n) / n = \Theta(1/\log n)$

○ 例2, 串行起泡排序算法时间150s, 串行快速排序算法30s, 使用3个核的并行起泡排序算法40s

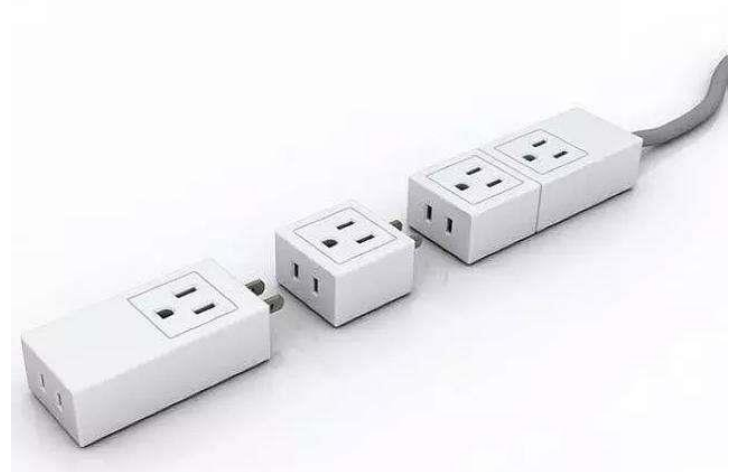
□ $S = T_S / T_P = 30s/40s=0.75$

□ $E = S/p = 0.75/3=0.25$

可扩展性 (scalability)

- 若某并行程序核数(线程数/进程数)固定，并且输入规模也是固定的，其效率值为 E 。现增加程序核数(线程数/进程数)，如果在输入规模也以相应增长率增加的情况下，该程序的效率一直是 E (不降)，则称该程序是**可扩展的**。

- 我们希望，保持问题规模不变时，效率不随着线程数的增大而降低，则称程序是可扩展的（称为**强可扩展的**）。但这往往是难达到的。
- 退求其次：问题规模以一定速率增大，效率不随着线程数的增大而降低，则认为程序是可扩展的（称为**弱可扩展的**）。



例4：并行矩阵-向量乘法

表 3-6 并行矩阵 - 向量乘法的加速比

comm_sz	矩阵的秩				
	1024	2048	4096	8192	16 384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

表 3-7 并行矩阵 - 向量乘法的效率

comm_sz	矩阵的秩				
	1024	2048	4096	8192	16 384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

➤ 该程序是弱可扩展的



例5:

- 某程序串行版本运行时间为 $T_s = n$ 秒, 这里 n 也为问题规模, 该程序某一并行版本运行时间为 $T_p = n / p + T_0$ 。该并行程序是否可扩展? (假设 T_0 为常数, 不随 p 变化而变化)

- $E = T_s / (p * T_p) = \frac{n}{p * (n / p + T_0)} = \frac{1}{1 + \frac{p}{n} * \frac{T_0}{1}}$

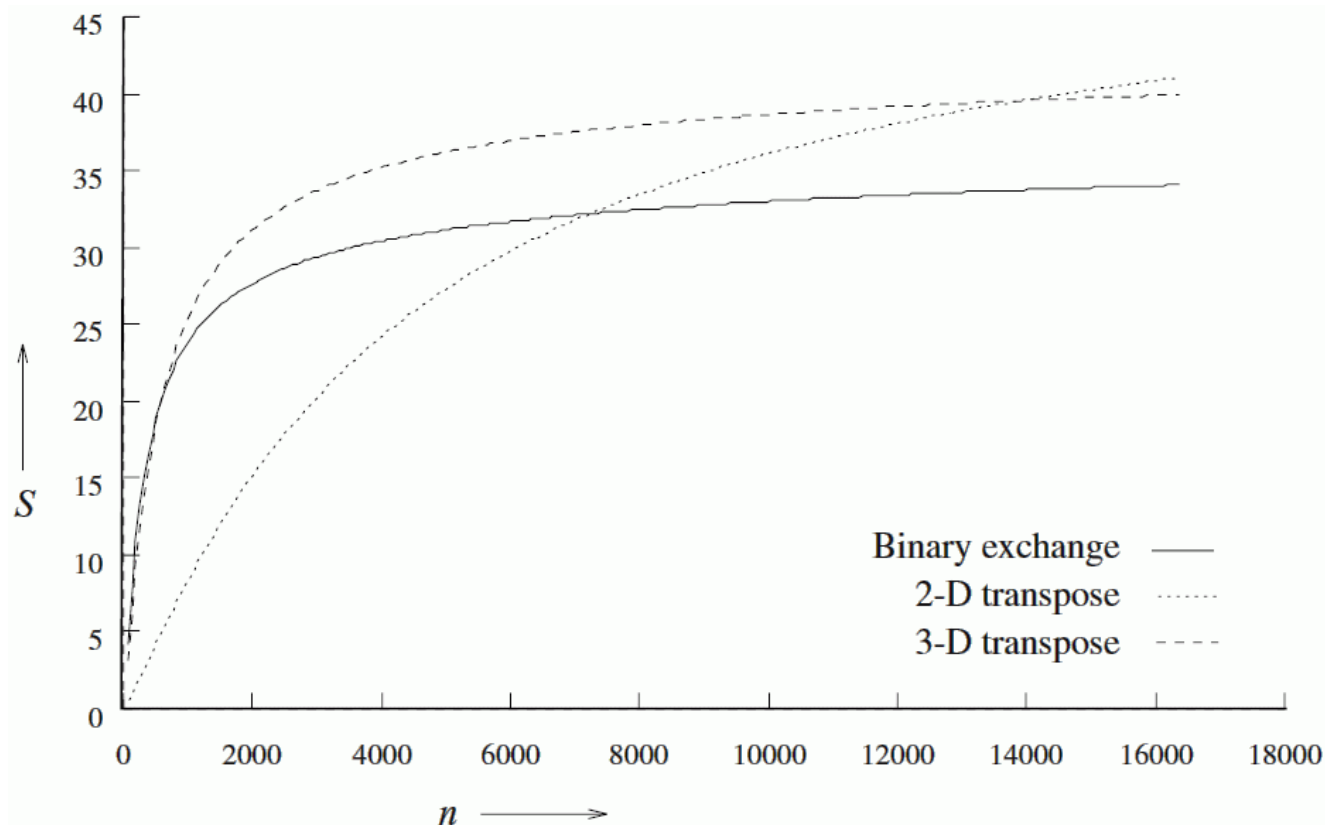
- 要保持 E 不变, 当 $p' = kp$ 时, 只需问题规模等比例增大即可, 即 $n' = kn$ 。
 - 因此, 该并行程序是可扩展的。



可扩展性

- 可扩展性是高性能并行机和并行算法追求的主要目标，其主要作用：
 - 度量并行系统性能的方法之一
 - 度量并行体系结构在不同系统规模下的并行处理能力
 - 度量并行算法内在的并行性
 - 利用系统规模和问题规模已知的并行系统性能来预测规模增大后的性能：scale down, 适合开发、调试，不适合性能预测

例5：快速傅里叶变换



- ❑ 直接测量时间，小规模和大规模时，不同算法性能对比结果不一致
- ❑ 需要进行可扩展性的定量分析