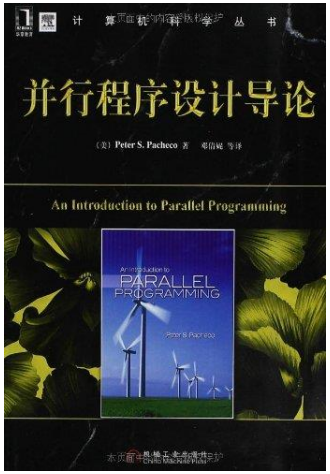
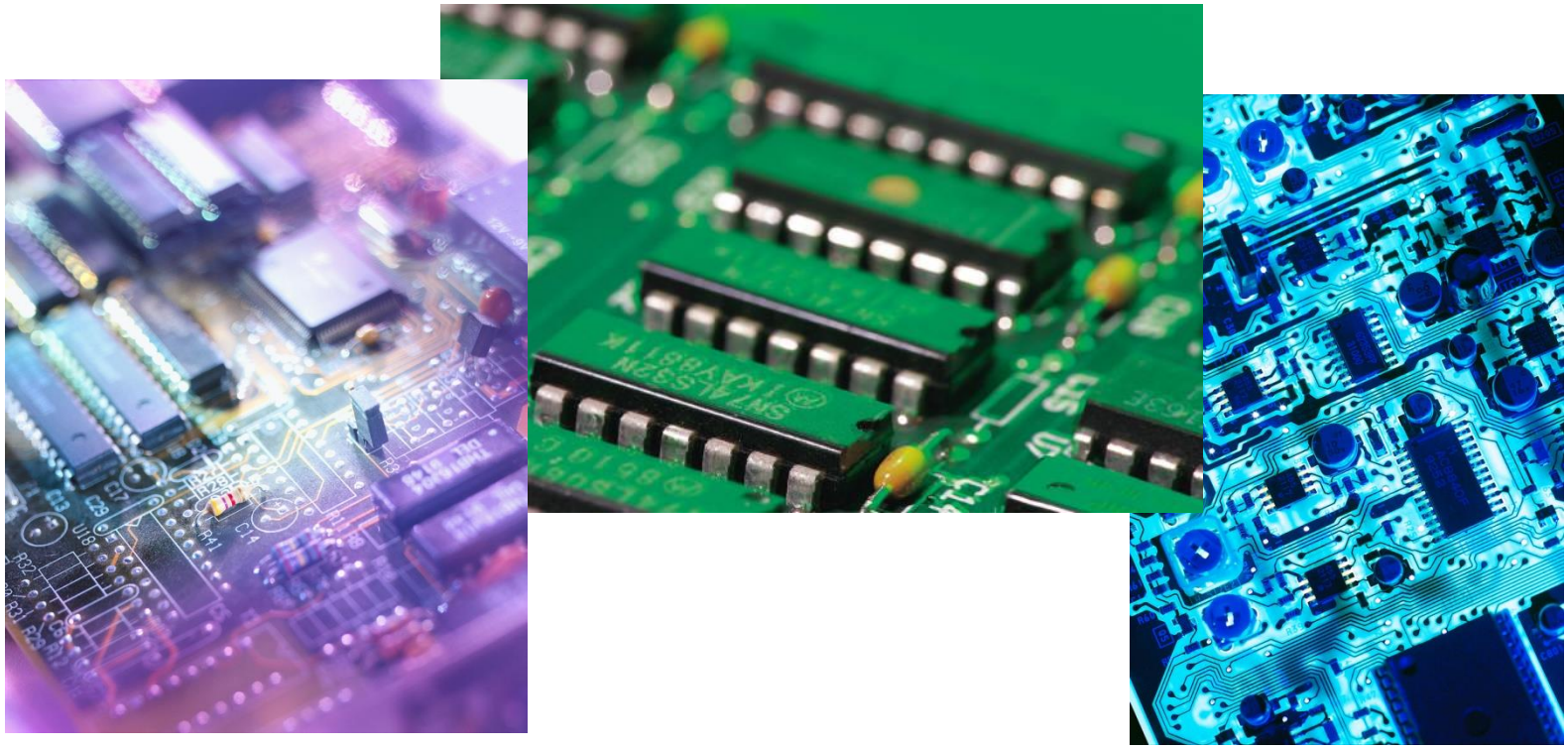


Chapter 2

并行硬件和软件

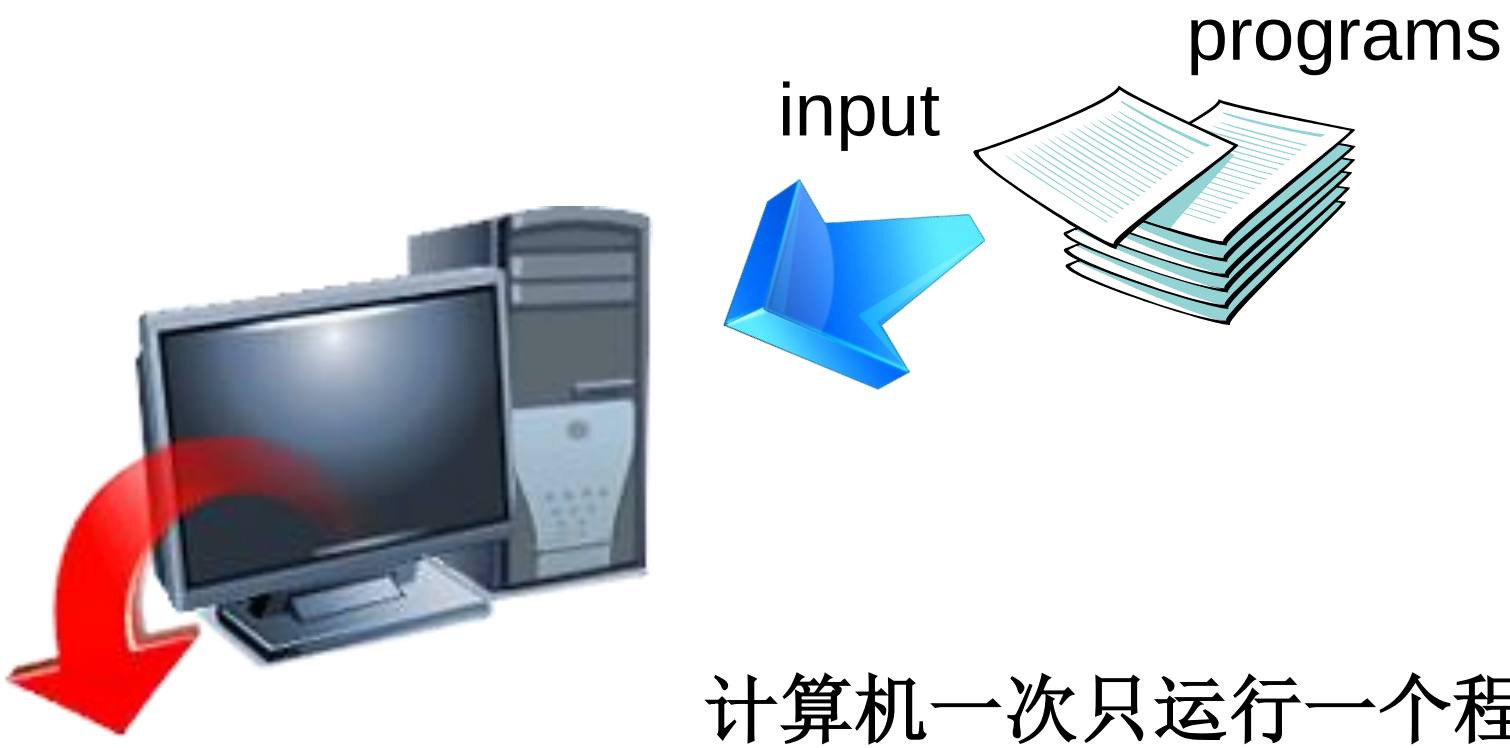
提纲

- 背景
- 冯·诺伊曼模型的改进
- 并行硬件
- 并行软件
- I/O
- 性能
- 并行程序设计
- 编写和运行并行程序
- 小结



背景

串行硬件和软件



计算机一次只运行一个程序.

冯·诺伊曼架构

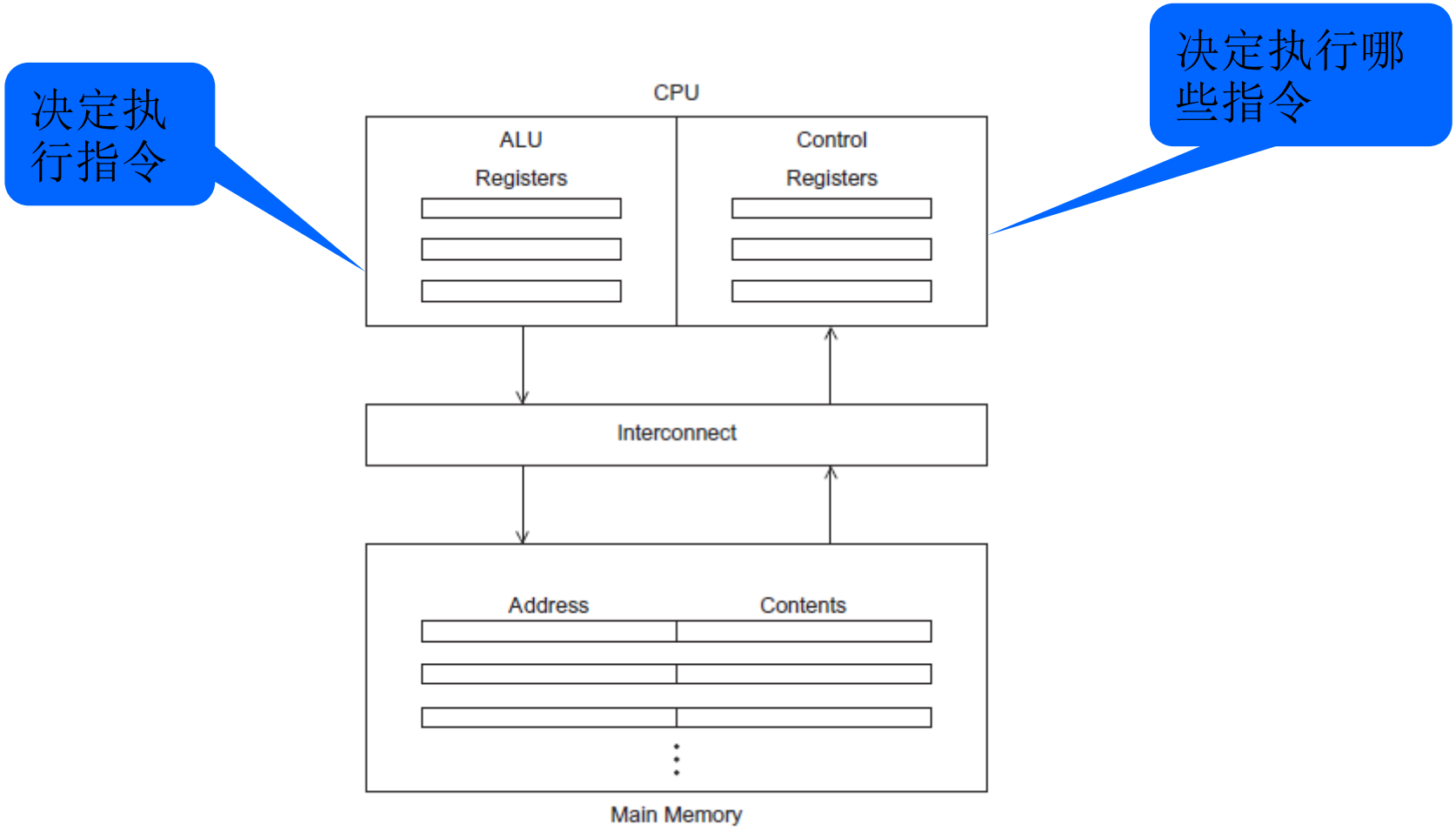
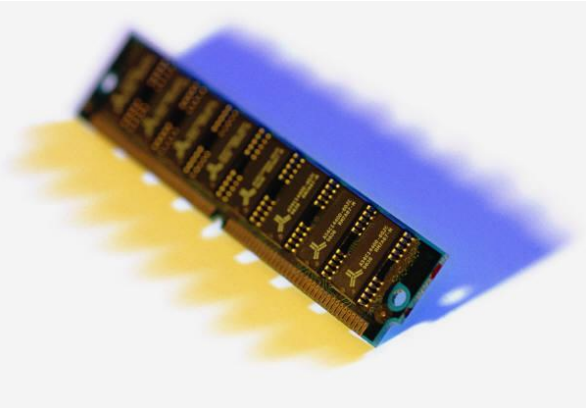


Figure 2.1 一次执行一条指令

主存

- 内存位置集合，每个位置都能够存储指令和数据
- 每个位置都包含一个地址(用于访问该位置)和该位置的内容。



中央处理器(CPU)

- 包括两部分
  - **控制单元（Control unit ）**：负责决定程序中的哪一条指令应该被执行。(老板)
  - **算术与逻辑部件（Arithmetic and logic unit**，ALU）：负责执行实际的指令。(工人)



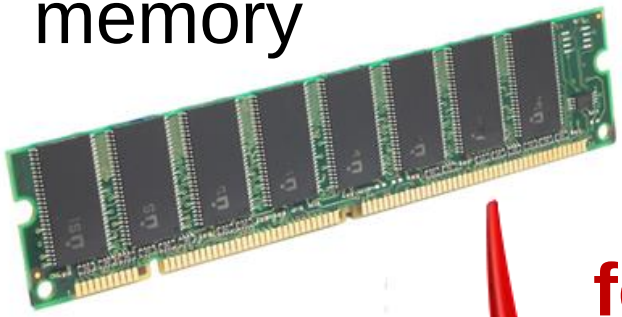
关键术语

- **寄存器（Register）**：非常快的存储，是中央处理器的一部分
- **程序计数器（Program Counter，PC）**：存储要执行的下一条指令的地址
- **总线（Bus）**：连接CPU和内存的电线和硬件

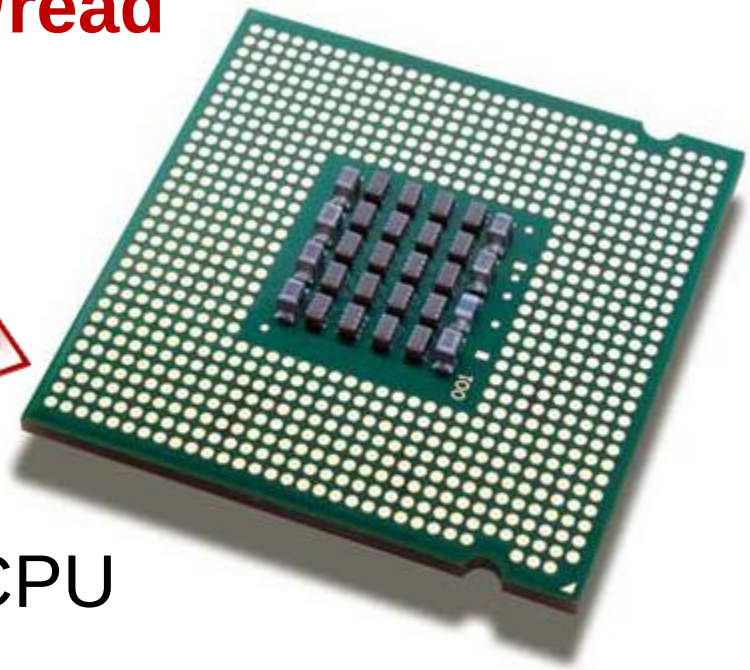


数据读取

memory



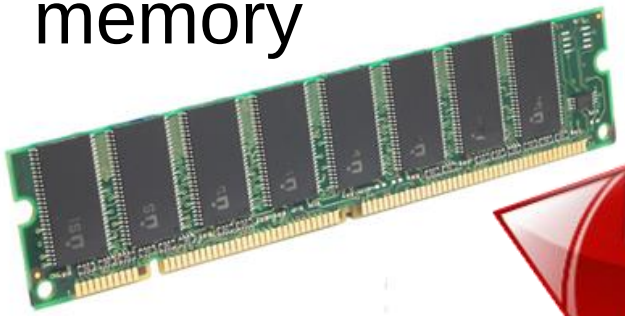
fetch/read



CPU

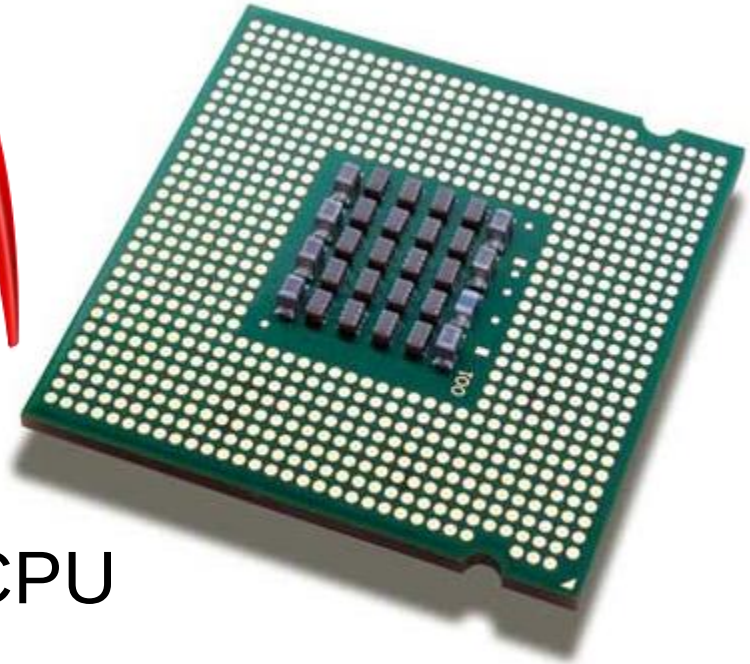
数据写入

memory



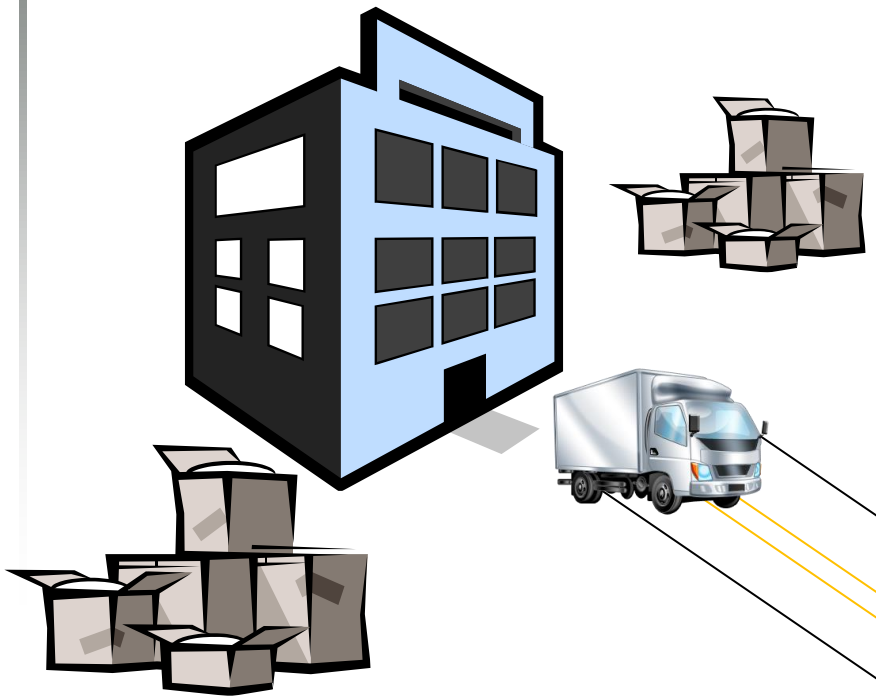


write/store



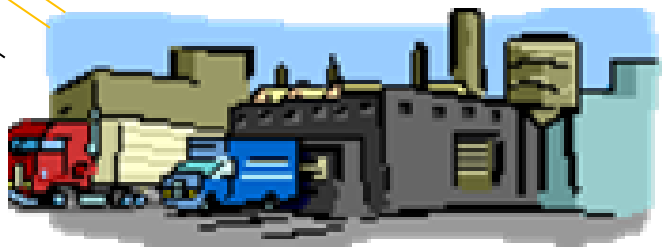
CPU

冯诺依曼架构的瓶颈



瓶颈：
主存与**CPU**之间分离
（负责生产的工厂与仓库的分离）





操作系统“进程”

- 正在执行的计算机程序的一个实例。
- 进程组成:
  - 可执行的机器语言程序
  - 内存块.
  - 操作系统分配给进程的资源描述符
  - 安全信息（软硬件资源的访问权限）
  - 进程状态信息

现代操作系统特点：多任务处理

- 使人产生一个单处理器系统同时运行多个程序的错觉.
- 每个进程轮流运行。（时间片）
- 在时间片结束后，进行等待，直到获得下一个时间片。

线程处理

- 线程包含在进程中，可以使用相同的代码，可以共享相同的内存和I/O设备

- 程序员可以将程序划分为多个独立的任务。
- 当一个线程因为等待资源而阻塞时，另一个线程可以运行.

一个进程和两个线程

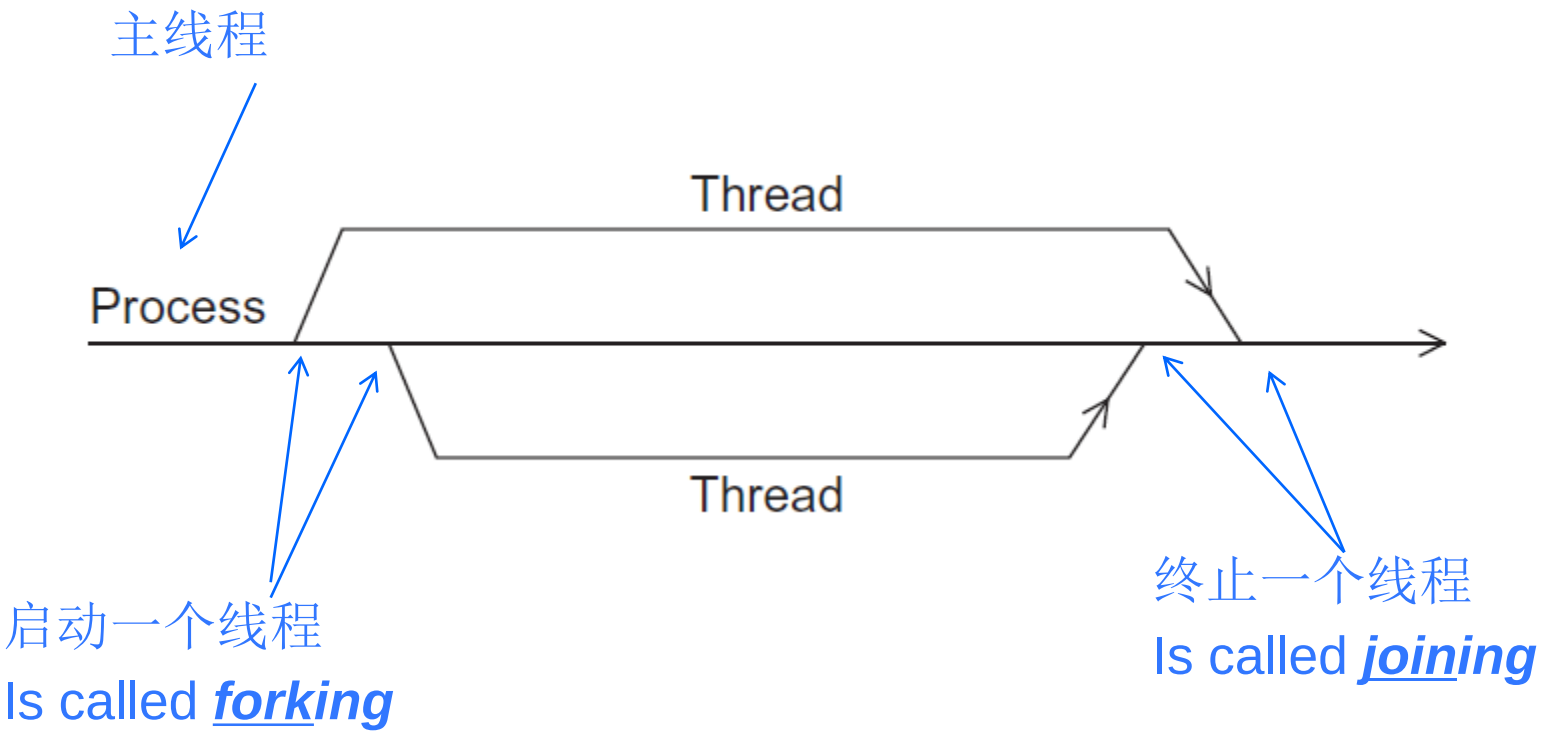
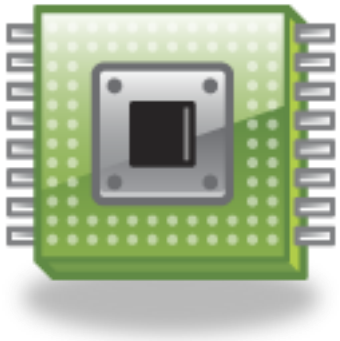
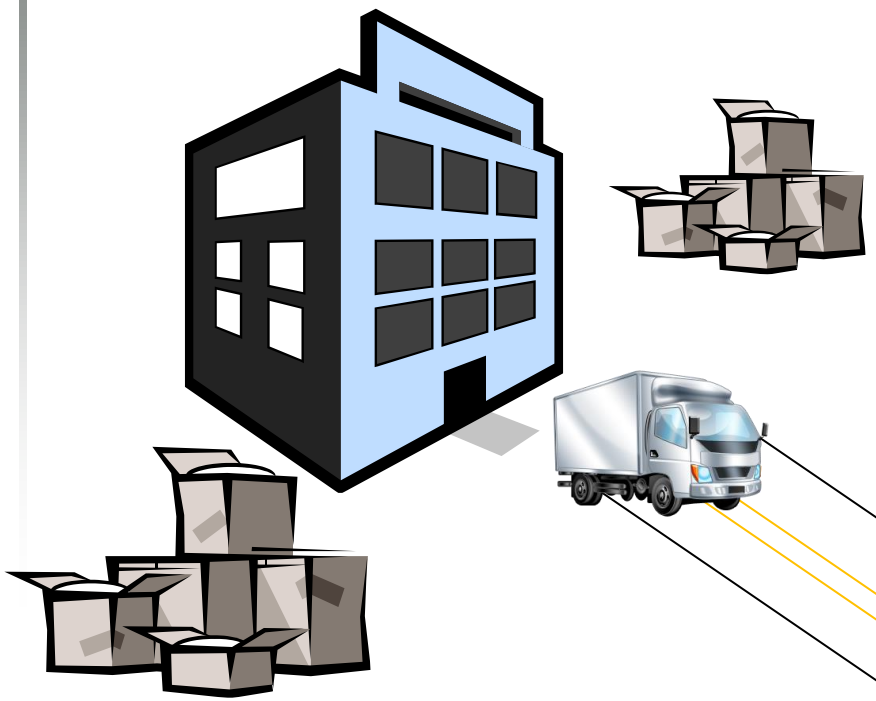


Figure 2.2



冯·诺伊曼模型的改进

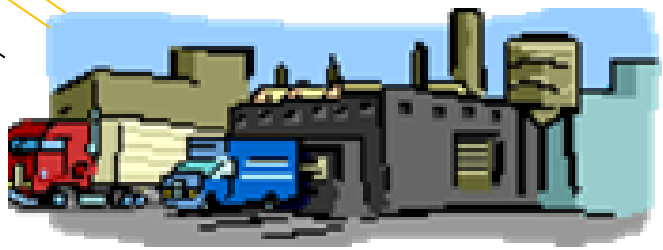
冯诺依曼架构的瓶颈



解决方案：

（1）加宽公路（多通道）

（2）迁移工厂或仓库，建立统一的场地。



CPU缓存是两种方案的结合

缓存的基础知识

- 内存位置集合，可以在比其它存储器有更短的访问开销。
- CPU缓存通常与CPU位于同一芯片上，访存开销比普通内存更快。



CPU缓存是基于访存局部性而设计

局部性原理

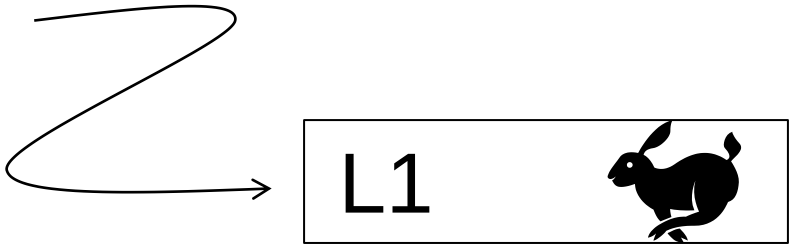
- 访问一个位置之后，紧接着是访问其附近的位置。
- **空间局部性（Spatial locality ）：**访问临近的位置。
- **时间局部性（Temporal locality ）：**最近访问的位置，在不久的将来还会访问。

局部性原理

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

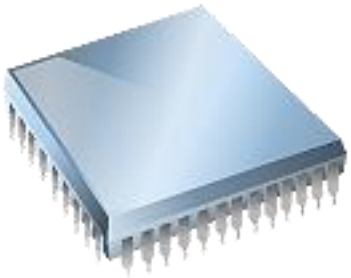
缓存级别

smallest & fastest



largest & slowest

缓存命中（Cache hit）



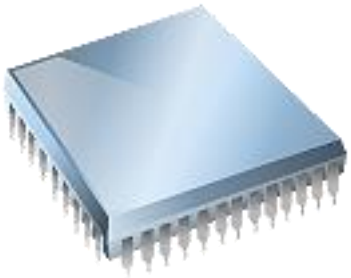
fetch x

L1 x sum

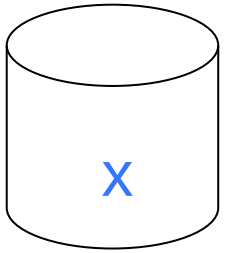
L2 y z total

L3 A[ ] radius r1 center

缓存缺失（Cache miss）



fetch x



main memory

L1 y sum

L2 r1 z total

L3 A[ ] radius center

缓存的问题

- 当CPU写数据到cache时，cache中的值可能与主存中的值不一致.
- 写直达（**Write-through** caches）通过在写入到缓存时，更新主存中的数据来解决这个问题
- 写回（**Write-back** caches）将缓存中的数据标记为脏数据。当缓存线（cache line）被内存中的新缓存线替换时，脏缓存线被写入内存.

缓存映射（Cache mappings）

- 全相联：一个新cache line可以放在缓存的任何位置.
- 直接映射：每一条高速缓存线在高速缓存中有一个唯一的位置，它将被分配到那里
- n路相联：每条高速缓存线可以被放置在n个不同的位置中的一个.

n路相联

- 当内存中不止一个cache line可以被映射到缓存中的几个不同位置时，我们还需要能够决定哪一个cache line应该被替换或移除。



Example

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

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

缓存和程序

```
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]

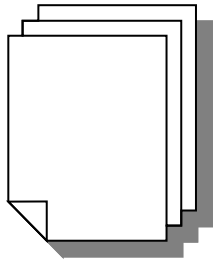
虚拟内存（Virtual memory ）

- 如果我们运行一个非常大的程序或者一个访问非常大数据集的程序，所有的指令和数据可能都不能放进主存中。
- 虚拟内存的功能是作为二级存储器的高速缓存。
- 它利用了时空局部性原理.
- 它只把正在运行的程序的活动部分保存在主存中

虚拟内存（Virtual memory ）

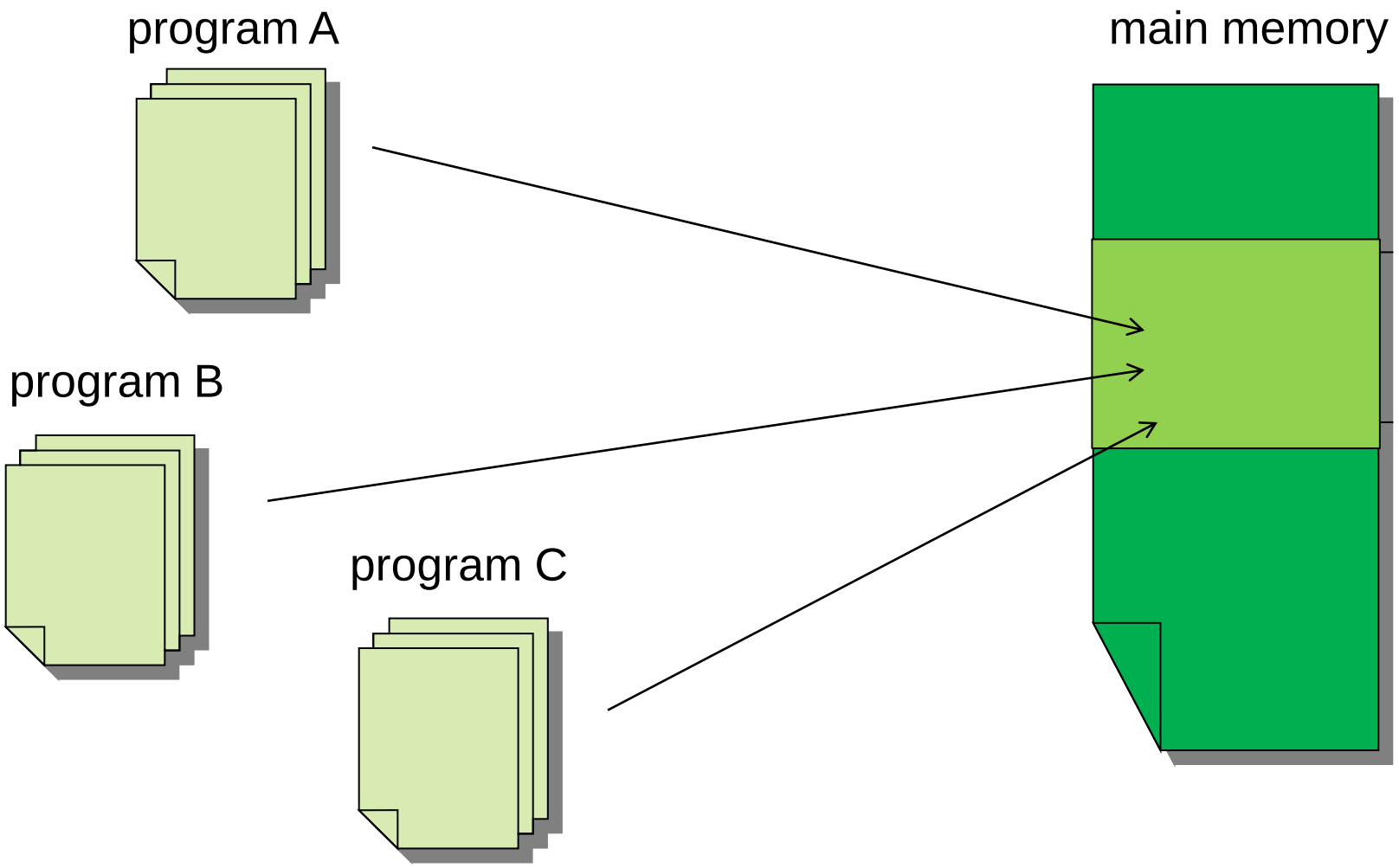
- **Swap 空间：** 那些空闲的部分被保存在一个二级存储块中。

- 分页（**Pages**）：数据块和指令块。



  - 通常分页都比较大.
  - 大多数系统的页面大小都是固定的，大小范围为4~16千字节。

虚拟内存（Virtual memory ）



虚拟页码

- 当一个程序被编译时，它的页被分配虚拟页码/逻辑页码.
- 当程序运行时，会创建一个页表，将虚拟页码映射到物理地址.
- 页表用于将虚拟地址转换为物理地址.

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

Table 2.2: Virtual Address Divided into Virtual Page Number and Byte Offset

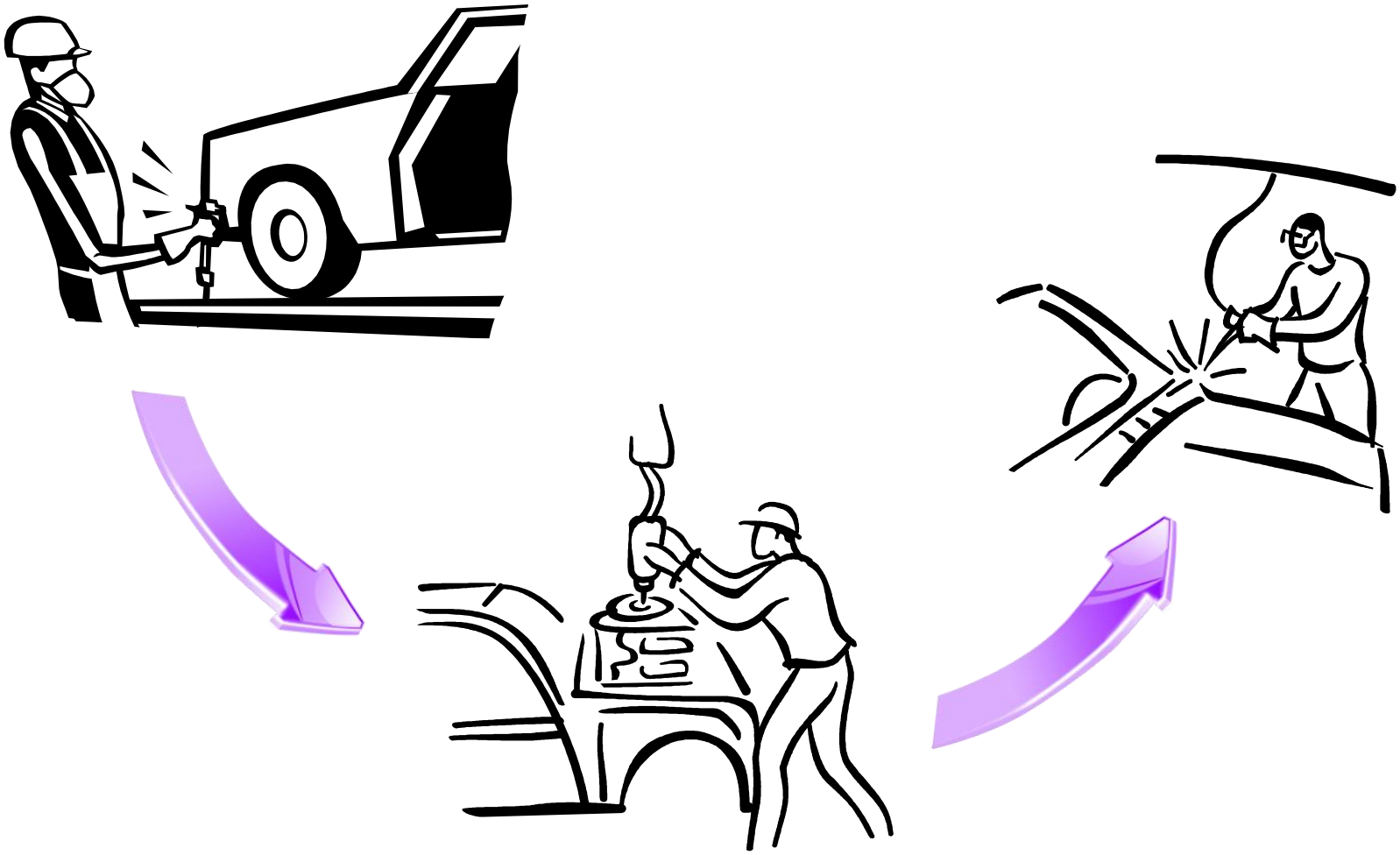
转译后备缓冲区(Translation-lookaside buffer ,TLB)

- 使用页表有可能显著增加每个程序的总体运行时间。
- 处理器中的一种特殊的地址转换缓存。
- 在非常快的内存中缓存少量页表项(通常是16-512)。
- 页面失效:试图访问页表中某个页的有效物理地址，但该页仅存储在磁盘上。

指令级并行(Instruction Level Parallelism ,ILP)

- 通过让多个处理器或功能单元同时执行指令来提高处理器性能。
- 流水线:功能单元是分阶段安排。
- 多发:多个指令同时启动执行。

流水线（Pipelining）



流水线例子

浮点数9.87×10^4与6.54×10^3的加法

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

流水线例子

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

- 假设每个操作需要1纳秒(10^{-9}秒)
- 这个for循环大约需要7000纳秒。

流水线例子

- 将浮点加法器分成7个单独的硬件或功能部件。
- 第一个单元获取两个操作数，第二个单元比较指数，等等。
- 一个功能部件的输出作为下一个功能部件的输入。

流水线例子

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

流水线加法

流水线例子

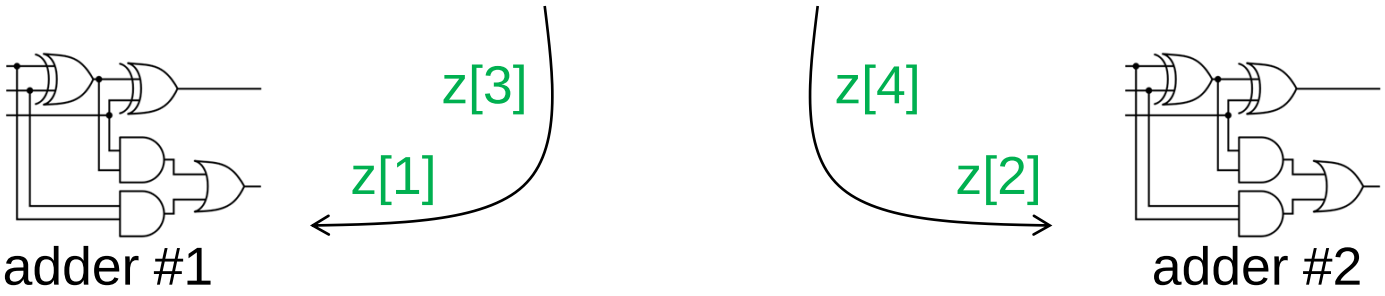
- 一个浮点数加法仍然需要7纳秒.
- 但是现在1000个浮点数加法需要1006纳秒!



多发射技术（Multiple Issue ）

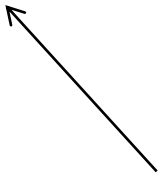
- 多发处理器复制计算单元，同时执行程序中的不同指令.

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



多发技术（Multiple Issue ）

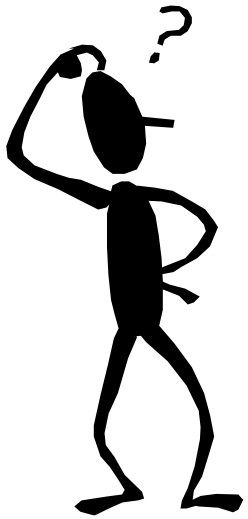
- 静态多发：计算单元在编译时被安排
- 动态多发：功能单元在运行时被安排



超标量（**superscalar**）

预测

- 为了利用多发，系统必须找到可以同时执行的指令



- 在预测过程中，编译器或处理器对一条指令进行预测，然后根据预测执行指令.

预测

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



如果系统猜测错误，它必须返回并重新计算w = y.

硬件多线程

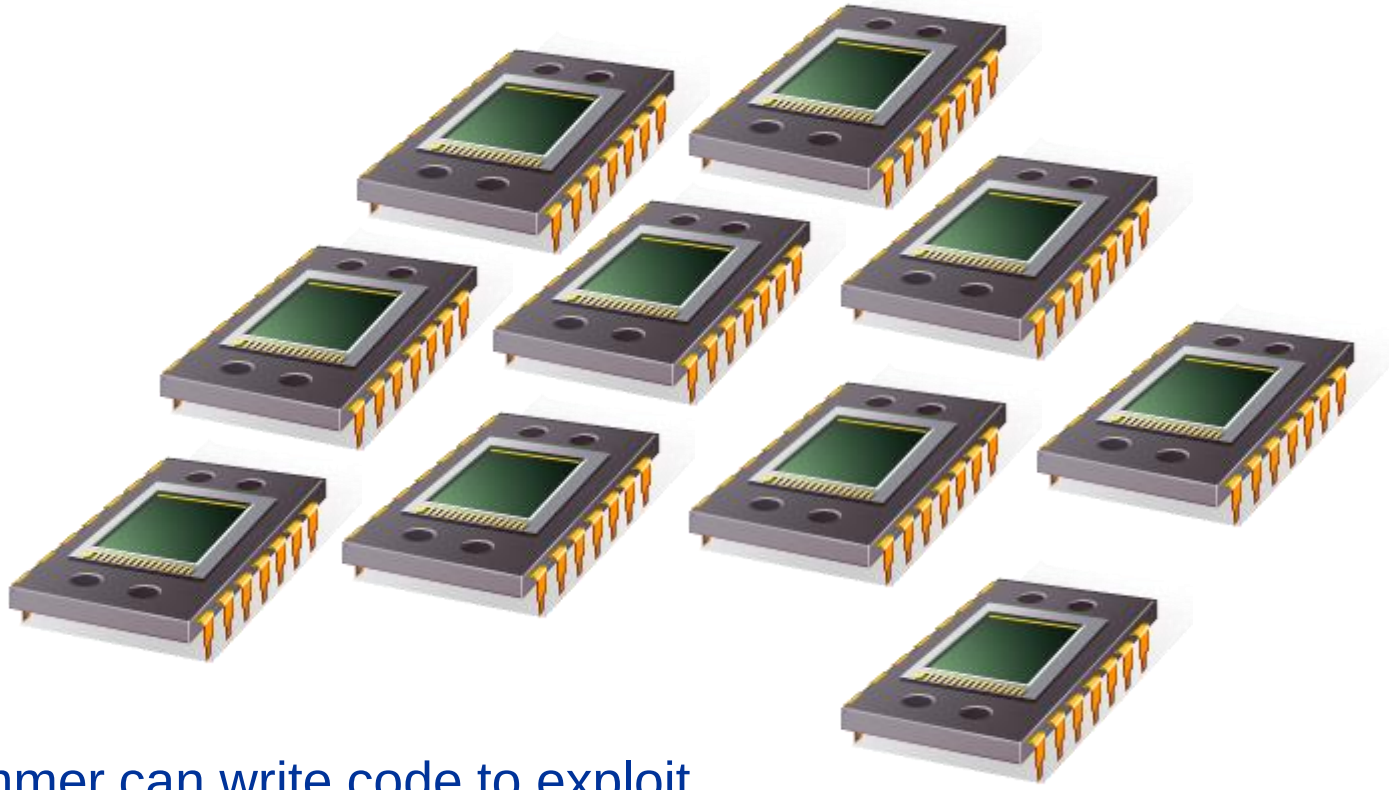
- 不同的线程并非总是能够同时执行
- 硬件多线程提供了一种方法，当前正在执行的任务已经停止时，系统可以继续做其他有用的工作。
- 例如，当前任务必须等待数据从内存中加载

硬件多线程

- 细粒度的（ **Fine-grained** ）：处理器在每条指令之后，进行线程切换，跳过停止的线程。
  - 优点:可避免因停机而浪费机器时间。
  - 缺点:准备执行一长串指令的线程可能必须等待执行每一条指令。

硬件多线程

- **粗粒度的（Coarse-grained）**：仅切换需要等待较长时间才能完成，而又被阻塞的线程。
  - 优点:切换线程不需要是瞬时的。
  - 缺点:处理器可以在较短的暂停时间内闲置，线程切换也会导致延迟。
- 同步多线程（ Simultaneous multithreading，SMT）：细粒度多线程的一种变体。允许多个线程同时使用多个功能单元，利用超标量处理器的性能。



A programmer can write code to exploit.

并行硬件

弗林的分类（Flynn's Taxonomy）

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

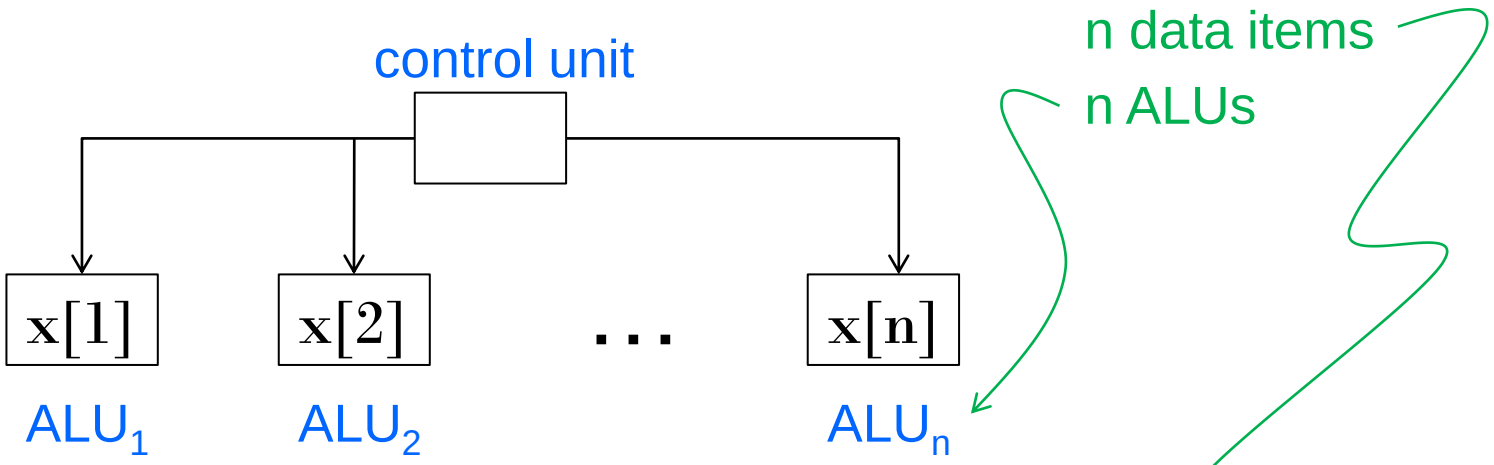
classic von Neumann

not covered

SIMD（Single instruction stream Multiple data stream）

- 通过在处理器之间划分数据而实现的并行性
- 将相同的指令应用于多个数据项
- 叫做 data parallelism.

SIMD example




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

SIMD（Single instruction stream Multiple data stream）

- 如果我们没有数据项那么多的ALU呢?
- 迭代地划分任务并处理
- Ex. 4 个ALU 和15 数据项

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ALU都需要执行相同的指令，或者保持空闲状态.
- 在经典设计中，它们还必须同步运行.
- ALU不能进行指令存储.
- 对于大型数据并行问题有效，但对于其它更复杂的并行问题无效。

向量处理器（Vector processors）

- 操作数据的数组/向量，而传统的CPU操作单个数据元素/标量.

- 向量寄存器（Vector registers）
  - 能够存储由操作数组成的向量并同时对其内容进行操作.

向量处理器（Vector processors）

- 向量化和流水线化的功能单元
  - 对向量中的每个元素（或元素对）应用相同的操作。
- 向量指令
  - 作用于向量而不是标量的指令

向量处理器（Vector processors）

- 交叉存储器（Interleaved memory）
  - 存储器有多个bank，每个bank可以被独立访问.
  - 将向量的元素分布到多个bank，从而减少或消除加载/存储连续元素的延迟.
- 跨步存储器访问、硬件发/收（scatter / gather）
  - 程序访问位于固定间隔位置的向量元素

向量处理器：优点



- 速度快
- 使用方便
- 向量化编译器擅长识别代码并利用它
- 编译器还可以提供不能被向量化的代码信息
  - 帮助程序员重新评估代码.
- 高存储器带宽
- 使用Cache中的所有数据

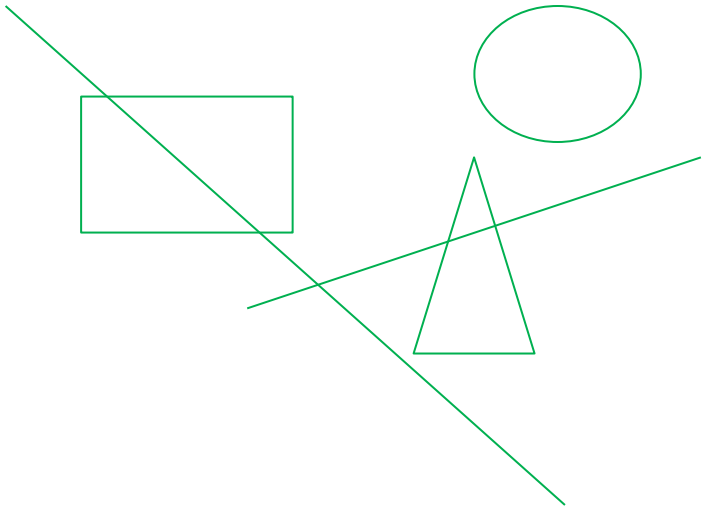
向量处理器：缺点

- 它们不能处理不规则的数据结构，不能通用于其它并行架构。
- 他们处理更大问题的能力有限，可扩展性弱



GPU(Graphics Processing Units)

- 实时图形应用程序编程接口（API），使用点、线和三角形表示目标的表面



GPUs

- 图形处理流水线将内部数据表示转换成可以发送到计算机屏幕的像素数组。

- 这个流水线的几个阶段，也称作着色器（Shader）功能，是可编程的.
  - 通常只有几行C代码.



GPUs

- Shader也是隐式并行的，因为它们可以应用于图形流中的多个元素。
- 早期GPU通常通过使用SIMD的并行性来优化性能.
- 目前的GPU使用SIMD并行.
  - 尽管它们不是纯粹的SIMD系统，而是SPMD.

MIMD（Multiple instruction stream Multiple data stream）

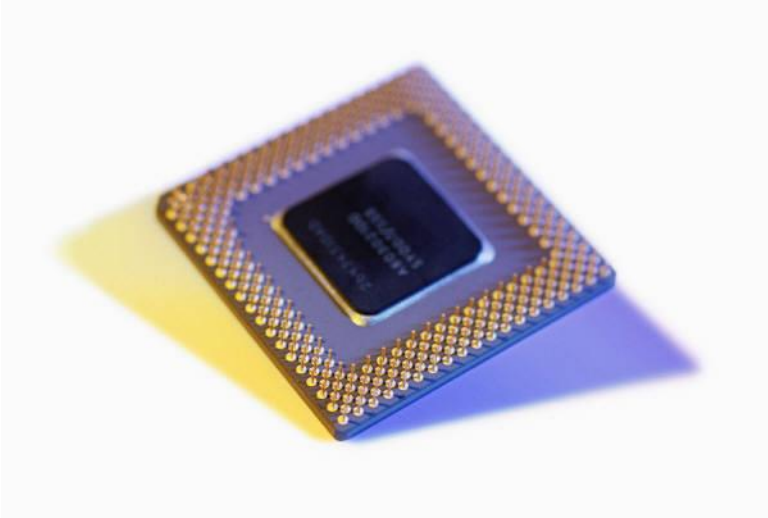
- 支持多个数据流上同时运行多个指令流.
- 通常由一组完全独立的处理单元或核心组成，每一个核心都有自己的控制单元（CU）和计算单元（ALU）。

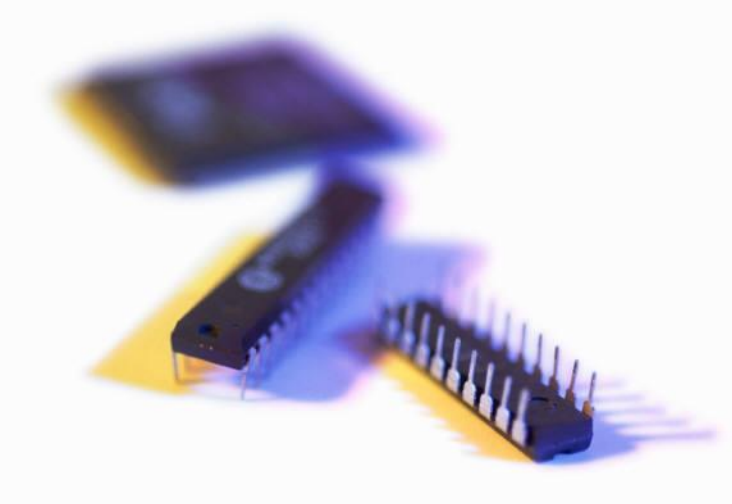
共享存储器系统（Shared Memory System）

- 一组自主处理器通过互连网络连接到存储器系统。
- 每个处理器都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.

共享存储器系统（Shared Memory System）

- 大多数可用的共享内存系统使用一个或多个多核处理器.
  - (在一个芯片上的多个CPU或核心)





共享存储器系统（Shared Memory System）

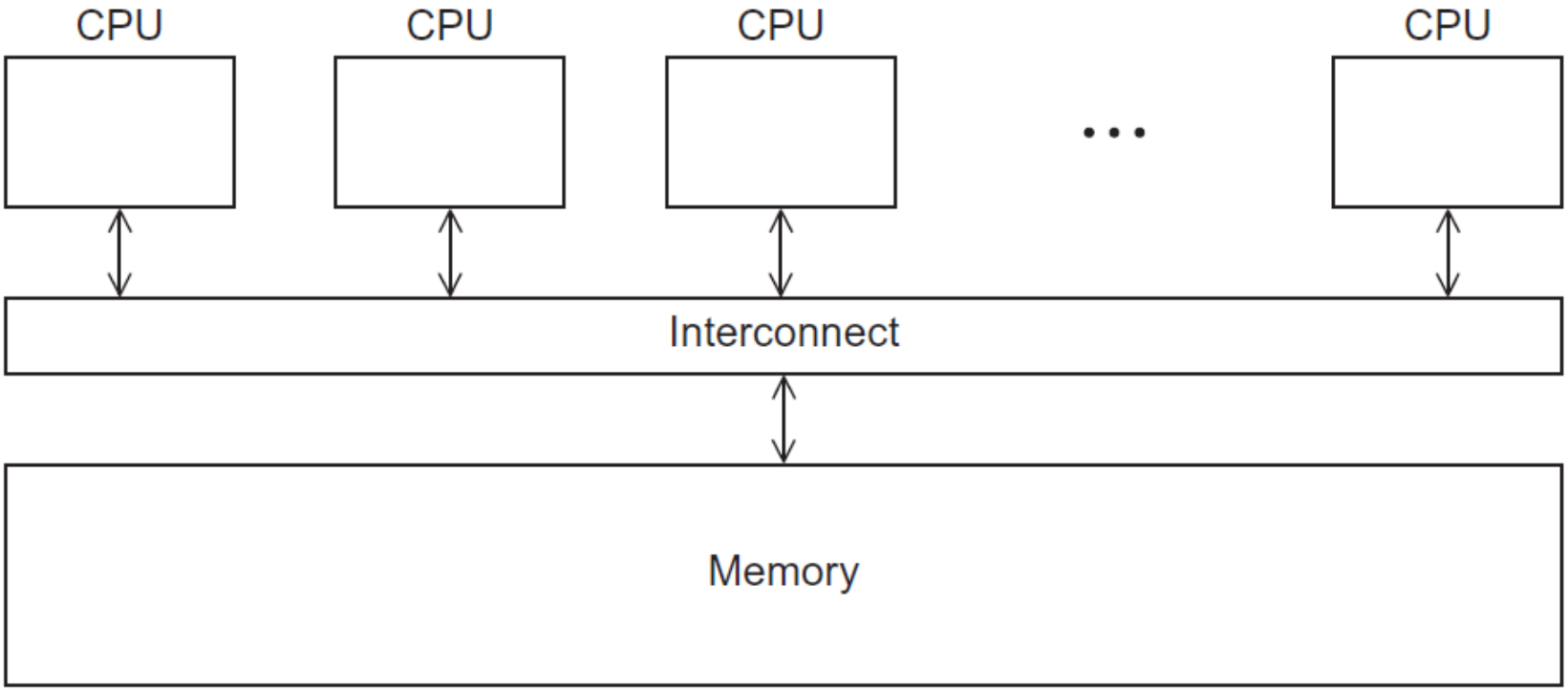
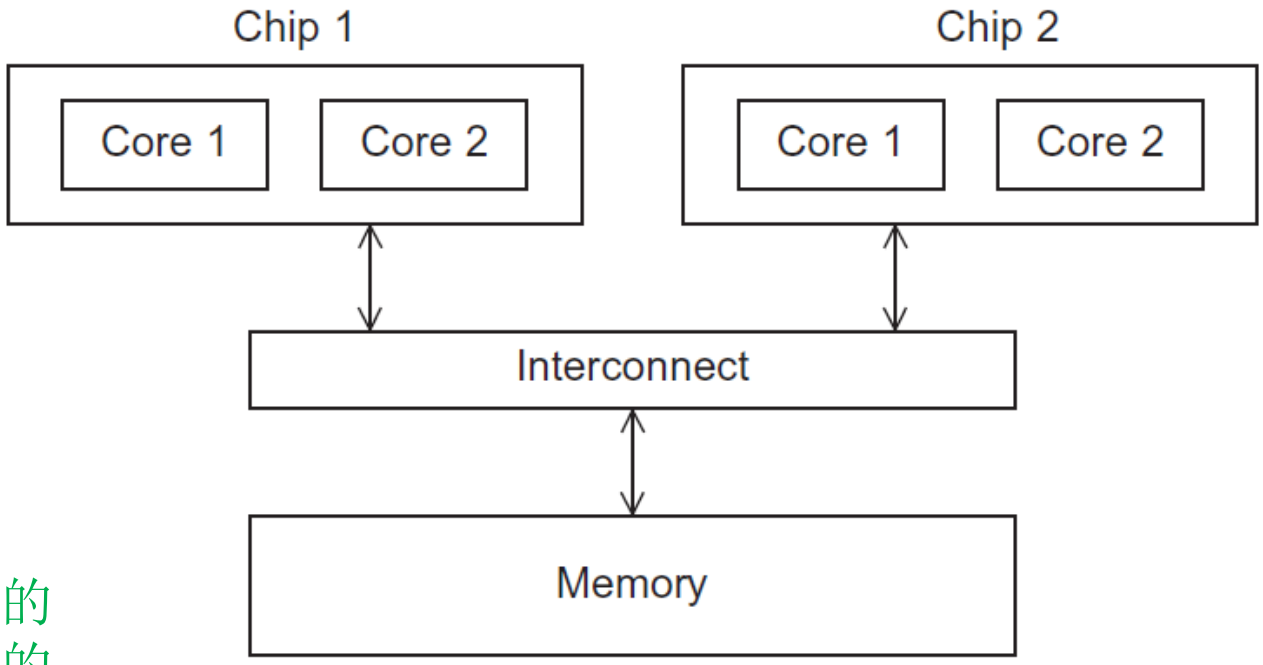


Figure 2.3

UMA multicore system



访问所有内存的时间对于所有的核心都是相同的.

Figure 2.5

UMA: Uniform Memory Access

NUMA multicore system

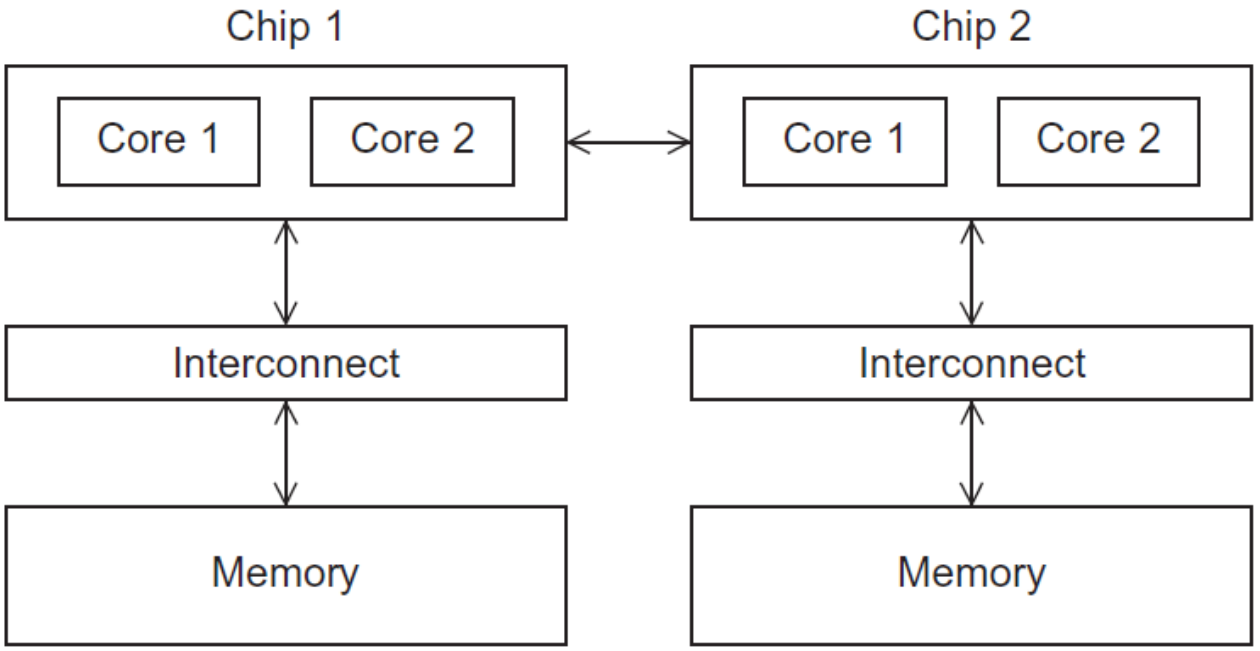


Figure 2.6

与核心直接相连的存储单元访问速度比必须通过另一个芯片访问的存储单元更快.

UMA: Non-uniform Memory Access

分布式系统（Distributed Memory System）

- 集群（Clusters） ※最流行
  - 商用系统的集合.
  - 商用互连网络连接.

- 集群的节点（ Nodes ）是单个计算单元，这些计算单元通过通信网络连接起来构成集群。

也称作：混合系统(hybrid systems)

分布式系统（Distributed Memory System）

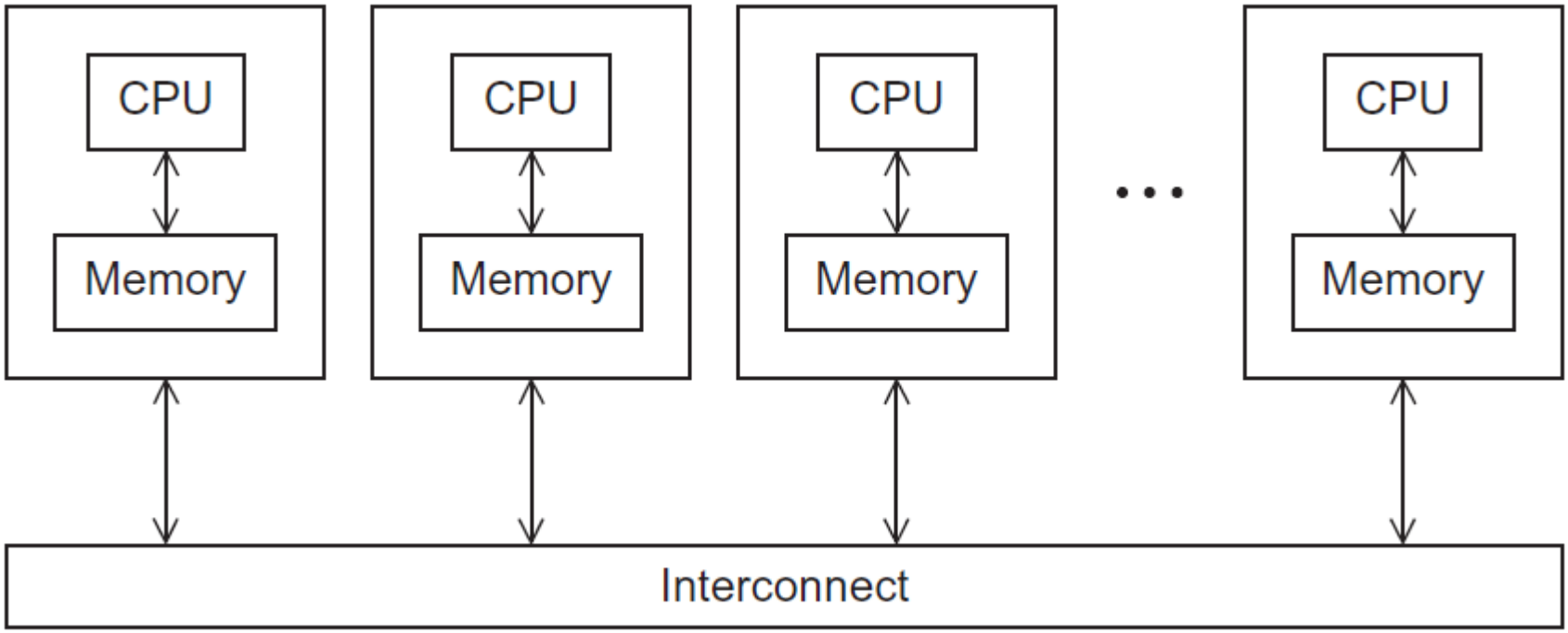


Figure 2.4

互连网络

- 影响分布式和共享内存系统的性能因素之一
- 分为两类:
  - 共享内存互联（Shared memory interconnects）
  - 分布式内存互联（Distributed memory interconnects）

共享内存互联

- 总线互连
  - 一组并联的通信导线以及对总线访问权限进行控制的硬件集合
  - 通信线路由连接到它的设备共享
  - 随着连接到总线的设备数量的增加，对总线的使用的争用会增加，性能会下降.

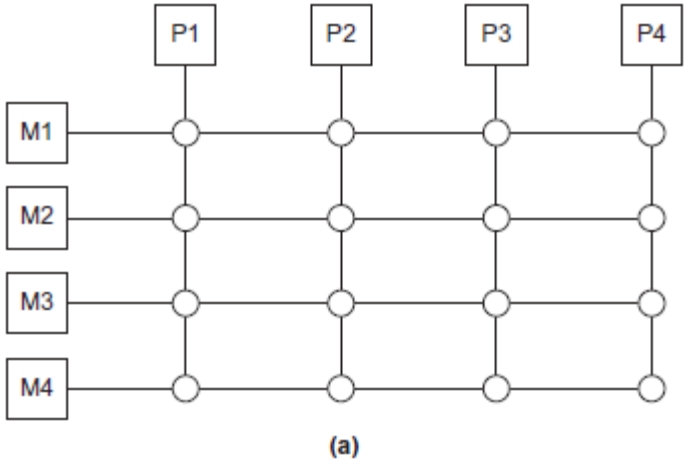
共享内存互联

- 交换互联
  - 使用交换机来控制连接设备之间的数据路由.
  - 交叉开关矩阵 –
    - 允许不同设备之间同时通信.
    - 比总线快.
    - 但是交换器和链路的成本相对较高.

Figure 2.7

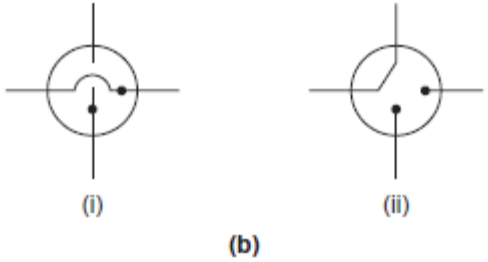
(a)

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

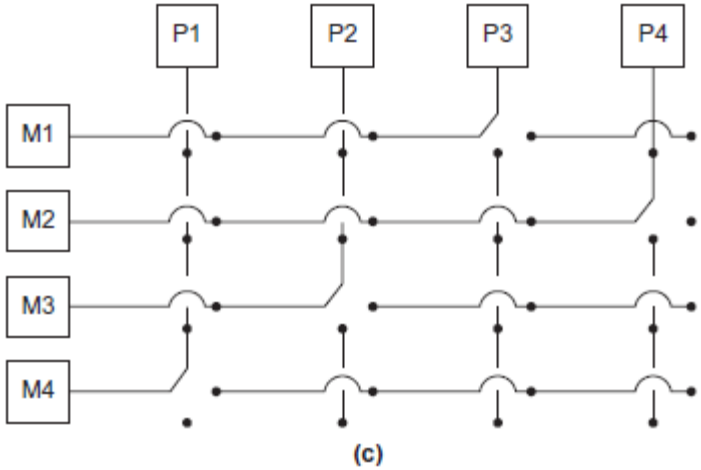


(b)

交叉开关矩阵内部的交换器



(c) 多个处理器同时访问内存

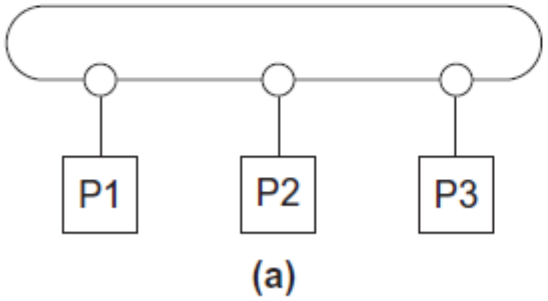


分布式内存互联

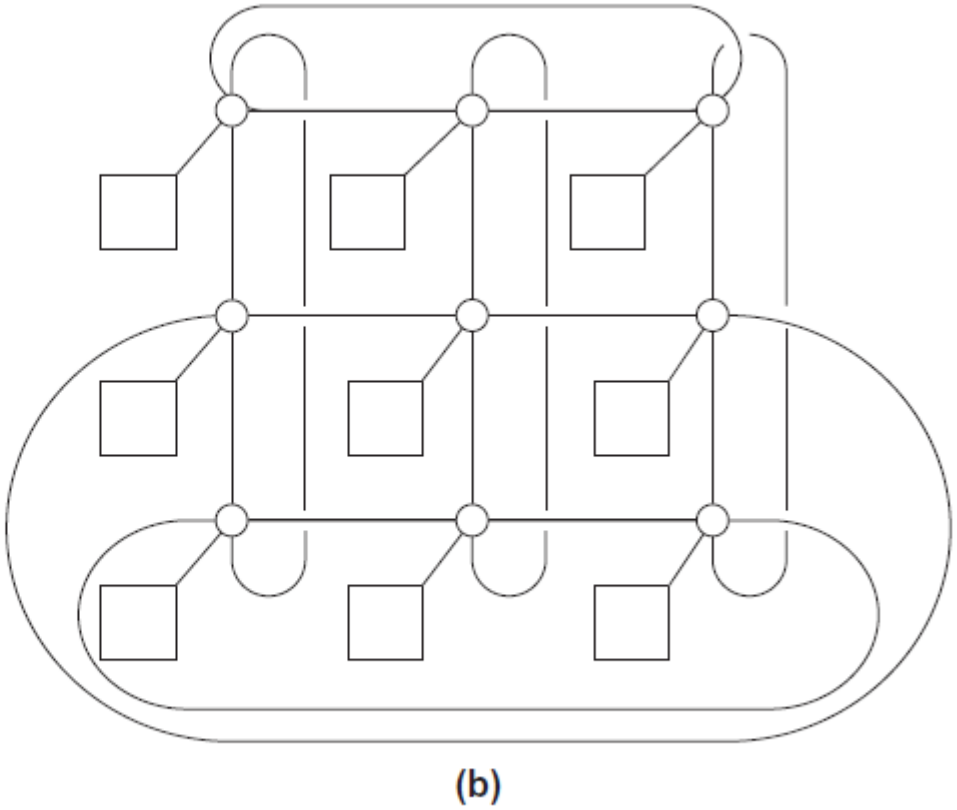
- 分两组
  - 直接互连（Direct interconnect ）
    - 每个交换器与一个处理器-内存对直接相连，交换器之间也相互连接.
  - 间接的互连（Indirect interconnect）
    - 交换器不一定与处理器直接连接.

直接互连

Figure 2.8



环



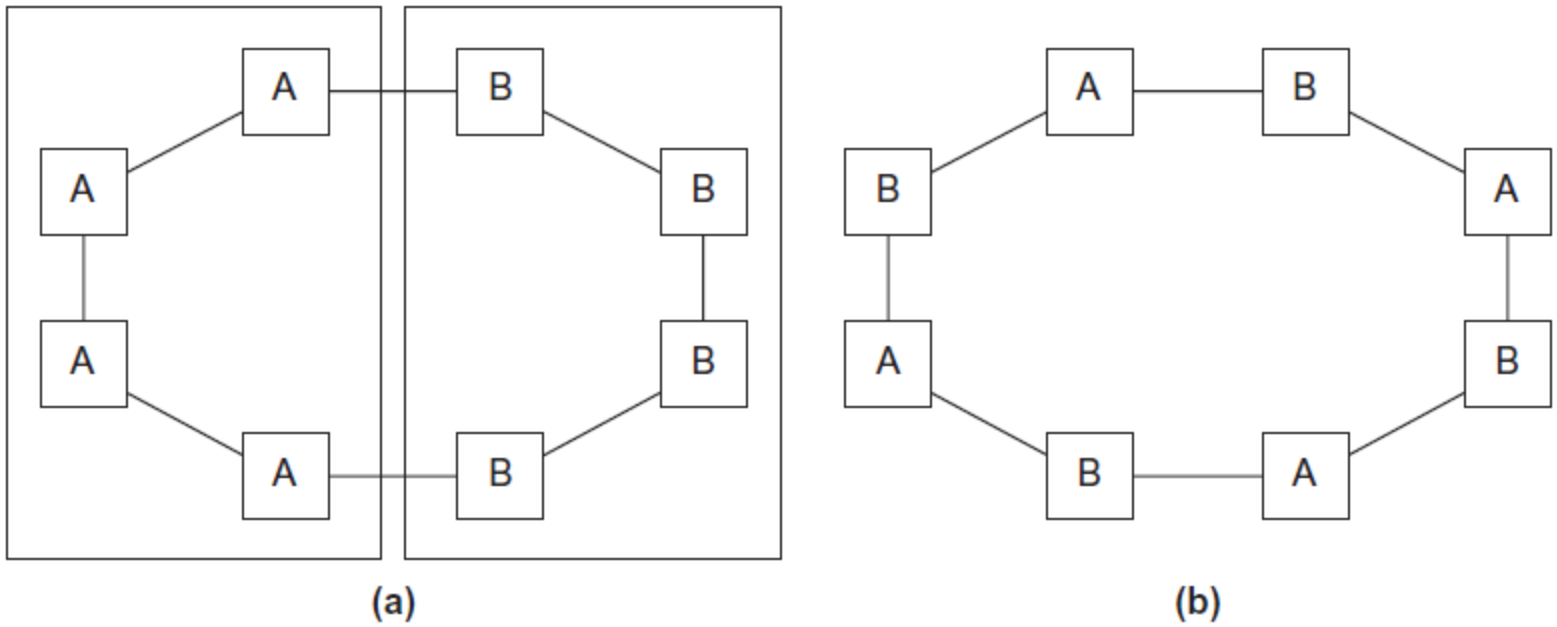
二维环面网格

等分宽度（Bisection width ）

- 衡量“同时通信的链路数量”或“连接性”的标准。
- 两部分之间能同时发生多少通信?



环的两种等分



通信链路数为2

通信链路数为4

Figure 2.9

二维环面网格的等分

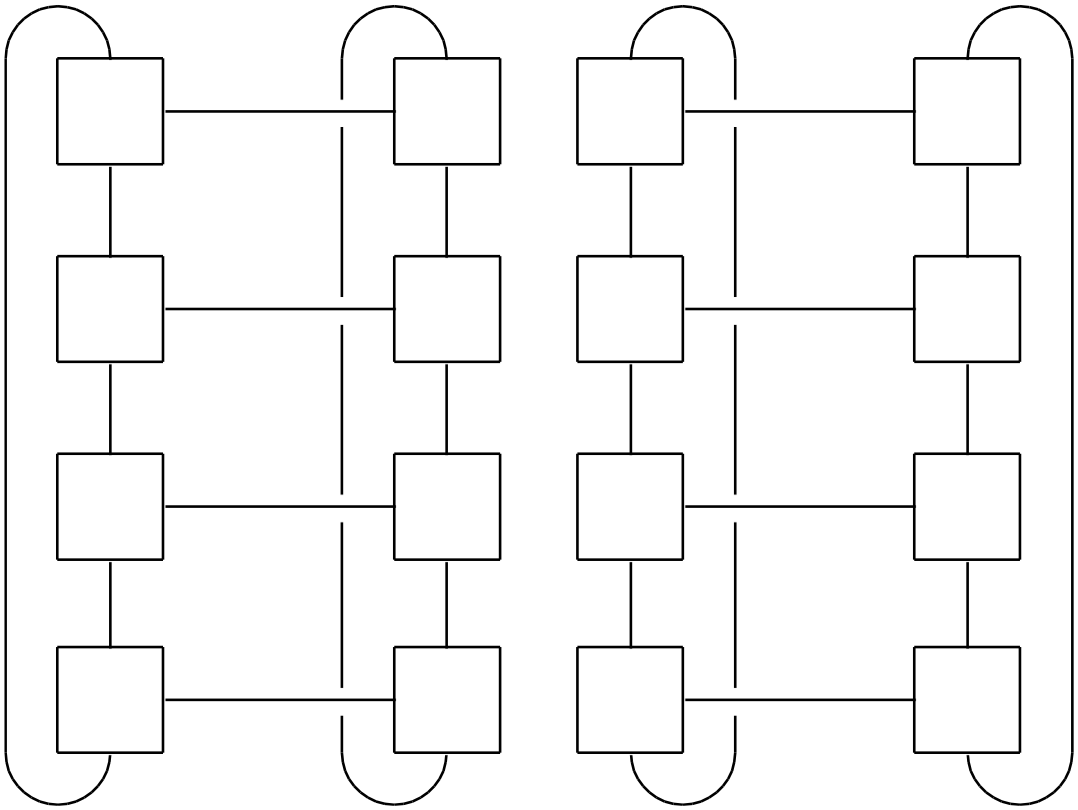


Figure 2.10

定义

- 带宽
  - 链路传输数据的速率
  - 通常用兆比特或兆比特每秒来表示
- 等分带宽
  - 网络质量的衡量标准.
  - 它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.

全相连网络

- 每一个交换器与其它交换器直接相连.

不切实际

等分宽度= $p^2/4$

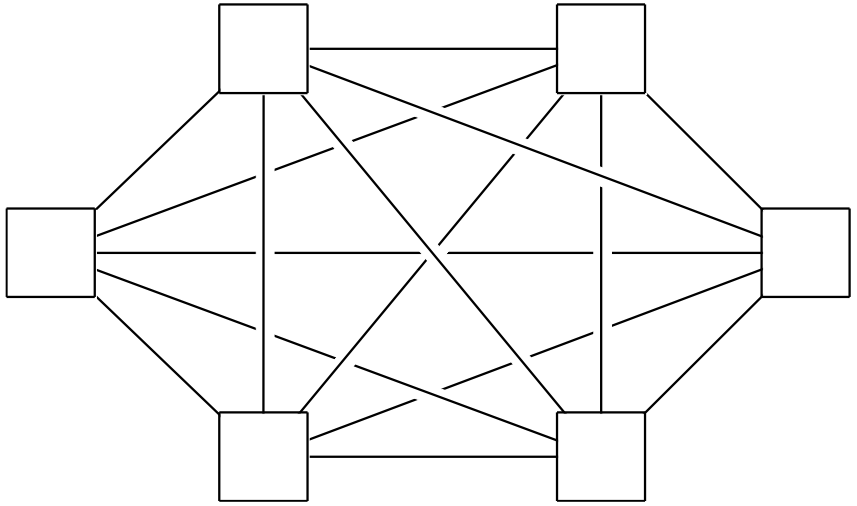


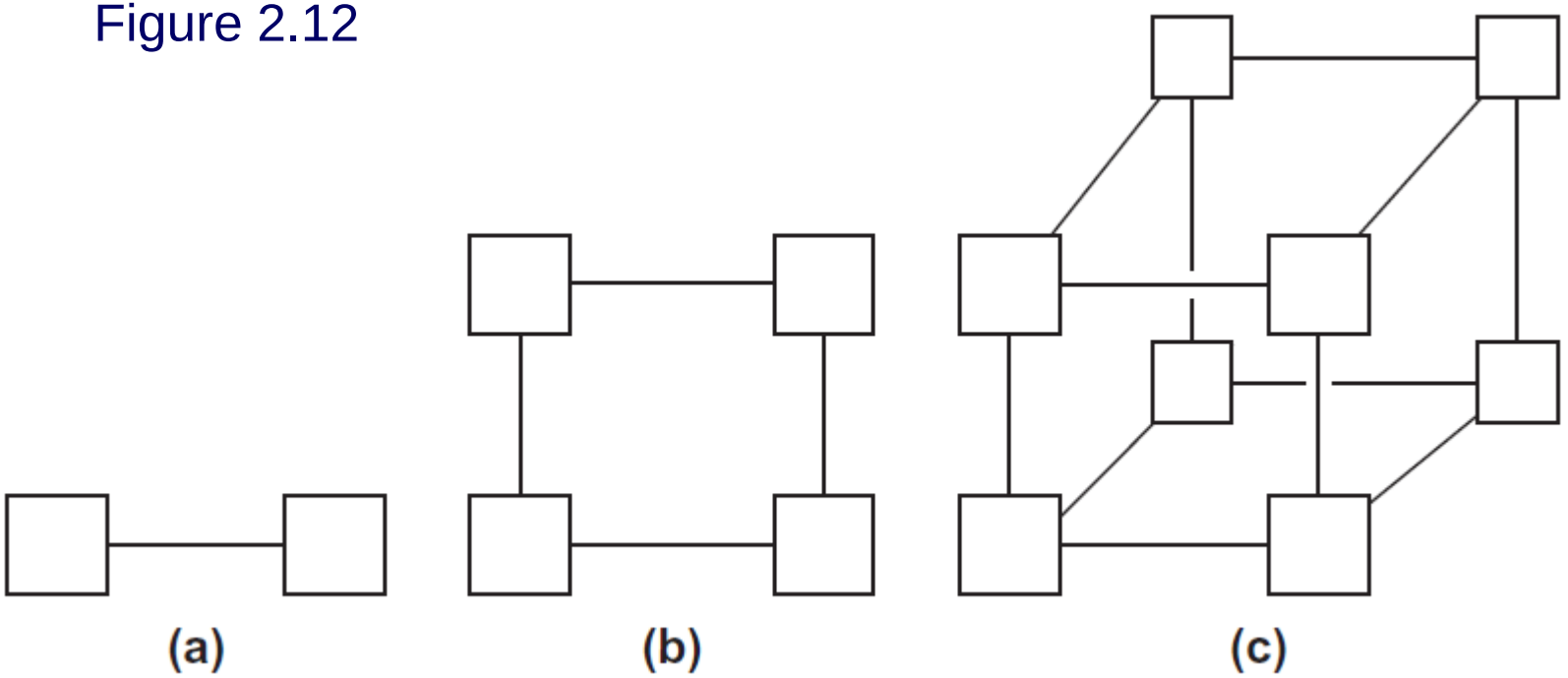
Figure 2.11

超立方体（Hypercube）

- 高度互联的直接互连网络
- 递归构造:
  - 一维超立方体是有两个处理器的全互联系统
  - 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，三维超立方体是由两个二维超立方体构成.

超立方体（Hypercube）

Figure 2.12



one-　　two-　　three-dimensional

间接互连

- 间接网络的简单例子:
  - 交叉开关矩阵（Crossbar）
  - 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

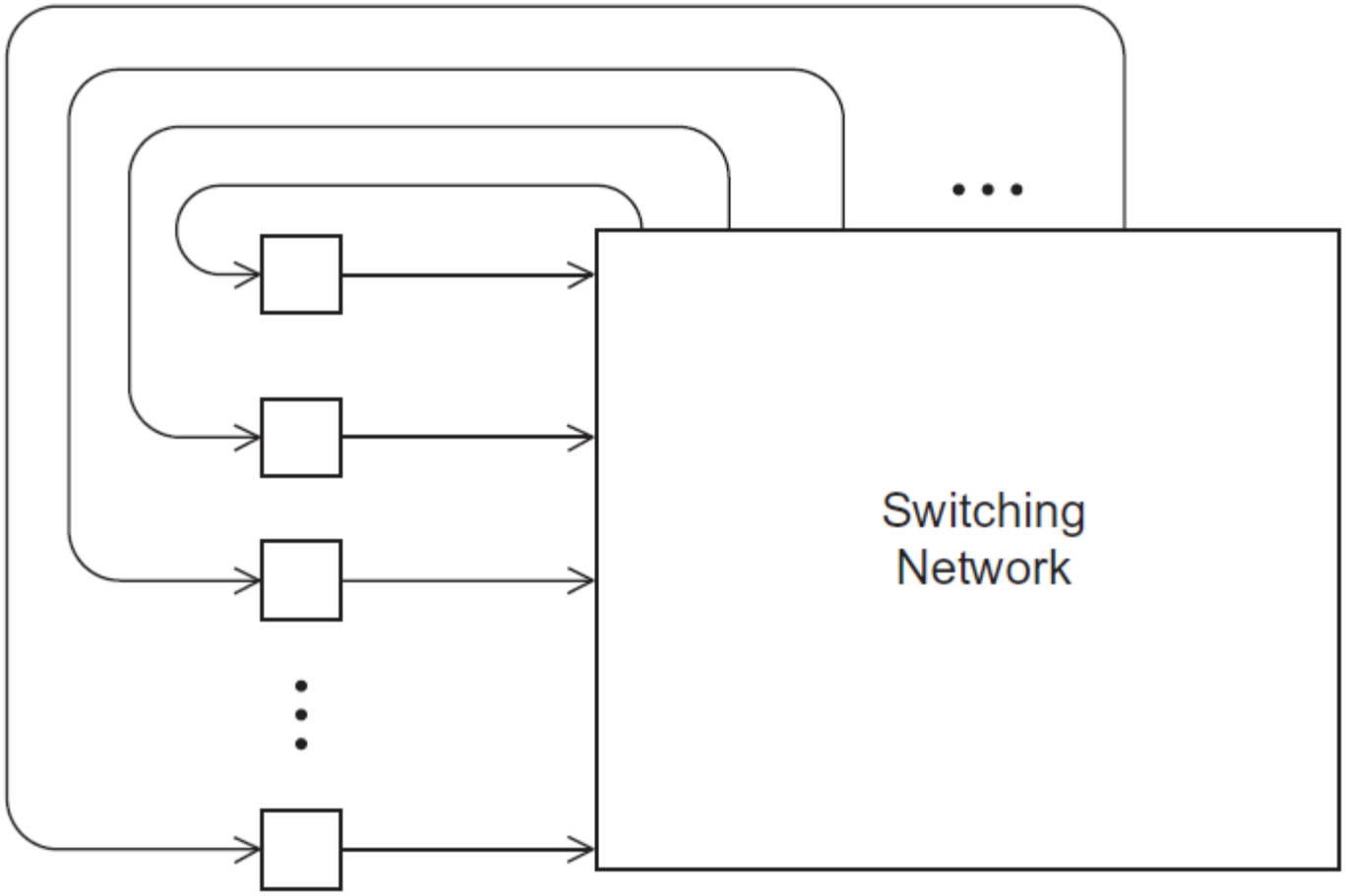


Figure 2.13

分布式内存交叉开关矩阵互联网络

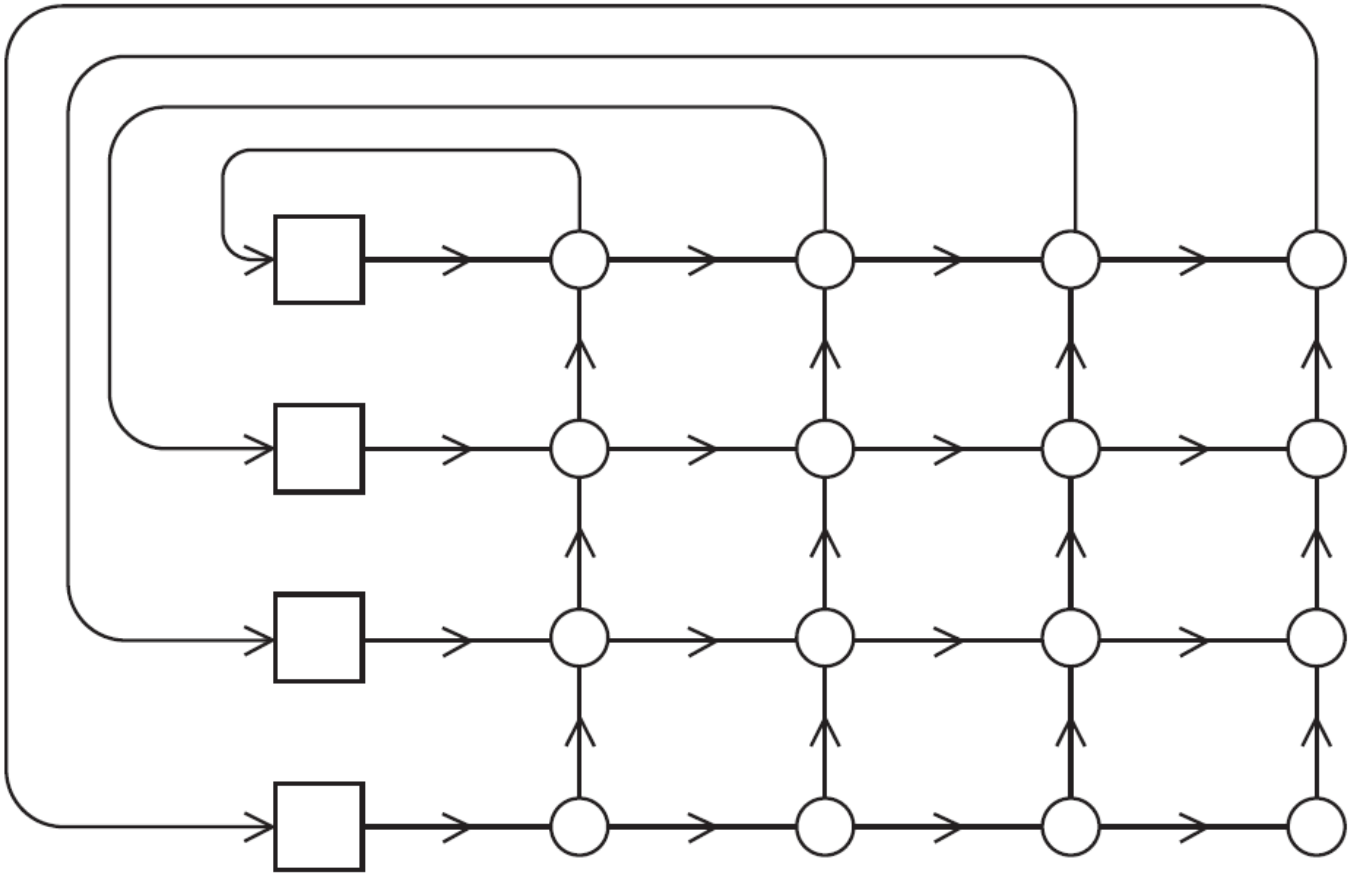


Figure 2.14

omega 网络

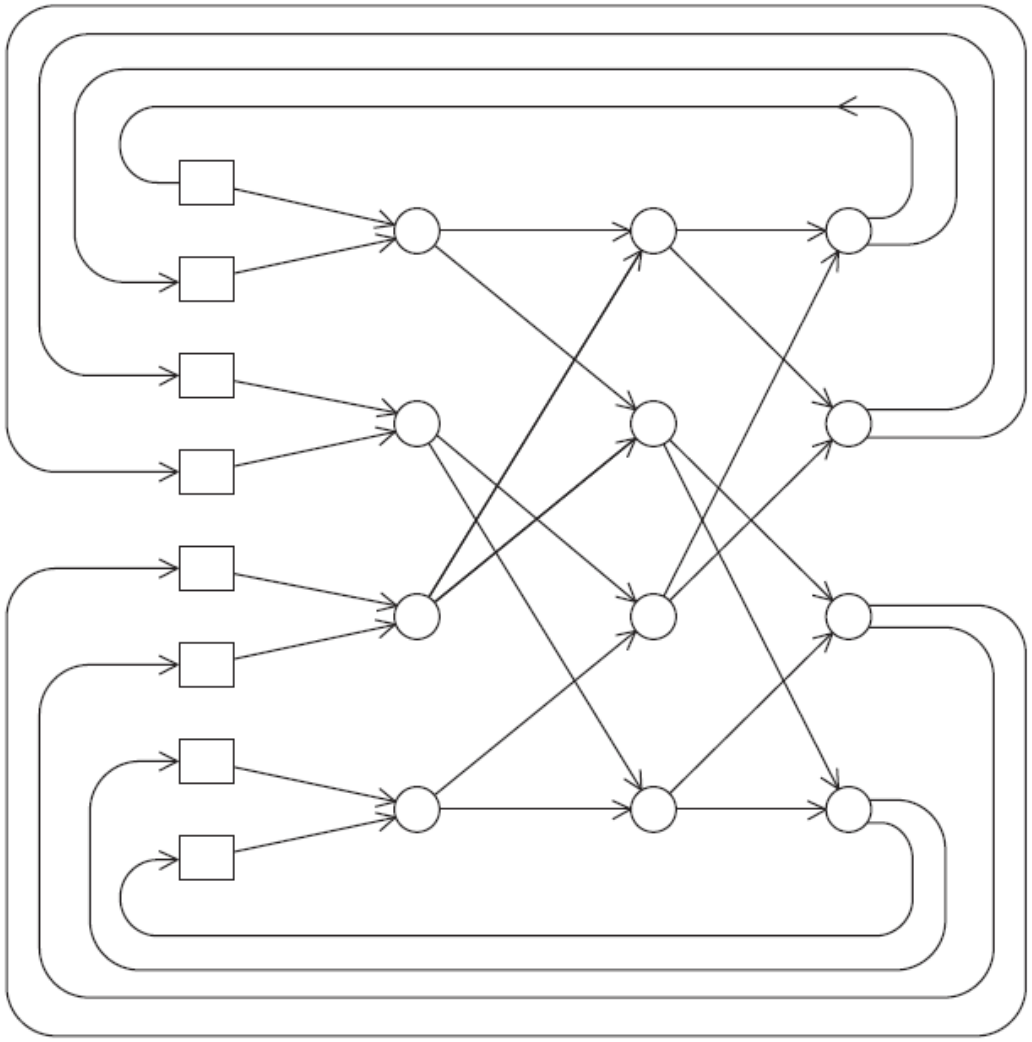


Figure 2.15

omega网络中的一个交换器

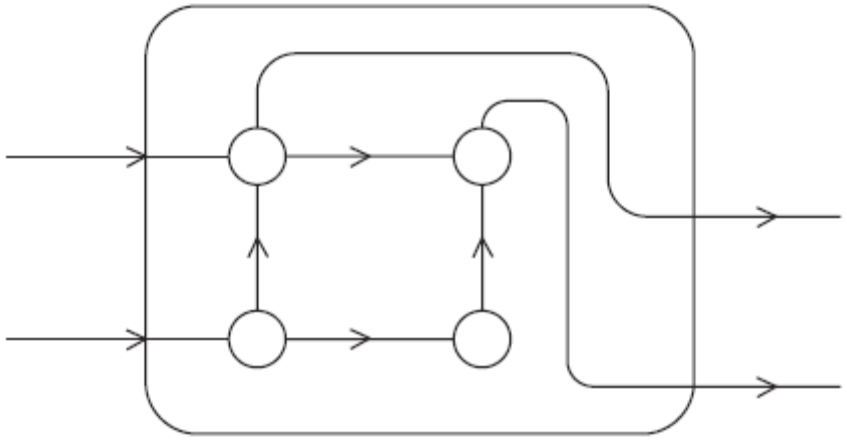
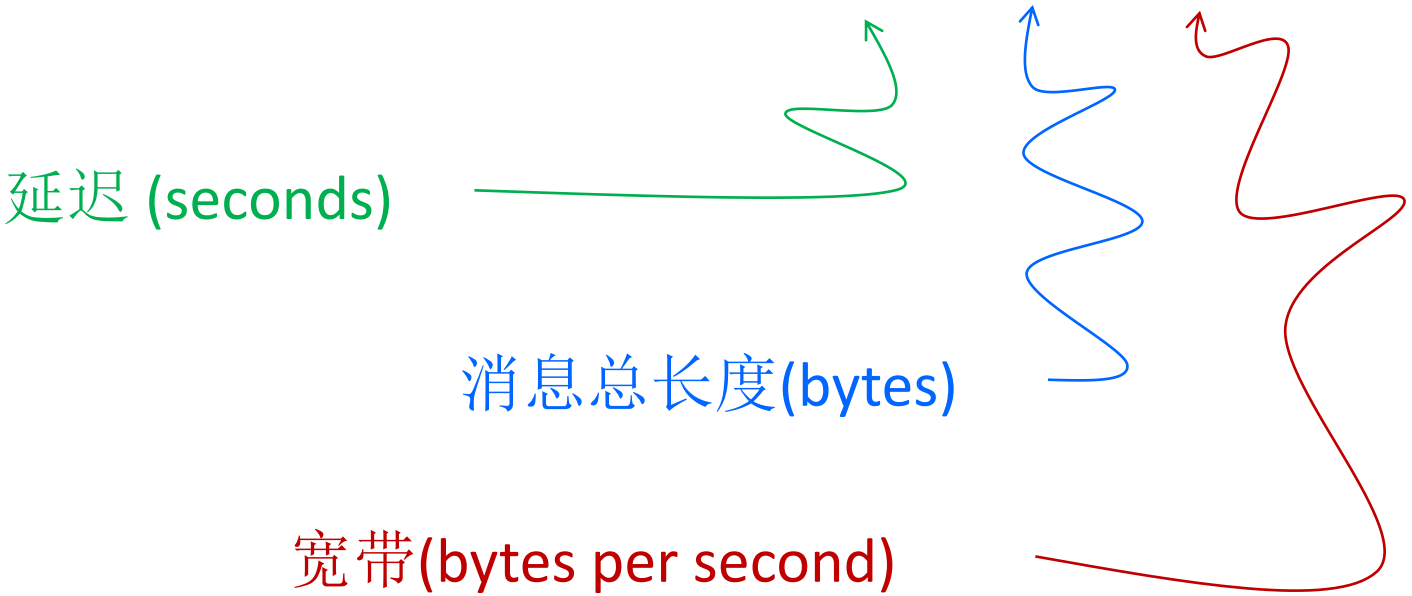


Figure 2.16

其他定义

- 传输数据时，我们感兴趣的是数据到达目的地需要多长时间.
- 延迟（Latency）
  - 从发送源开始发送数据到目的地开始接收第一个字节所经过的时间.
- 带宽（Bandwidth）
  - 目的地在开始接收第一个字节后接收数据的速率

消息传输时间 = l + n / b



Cache一致性

- 程序员无法控制缓存和何时更新.

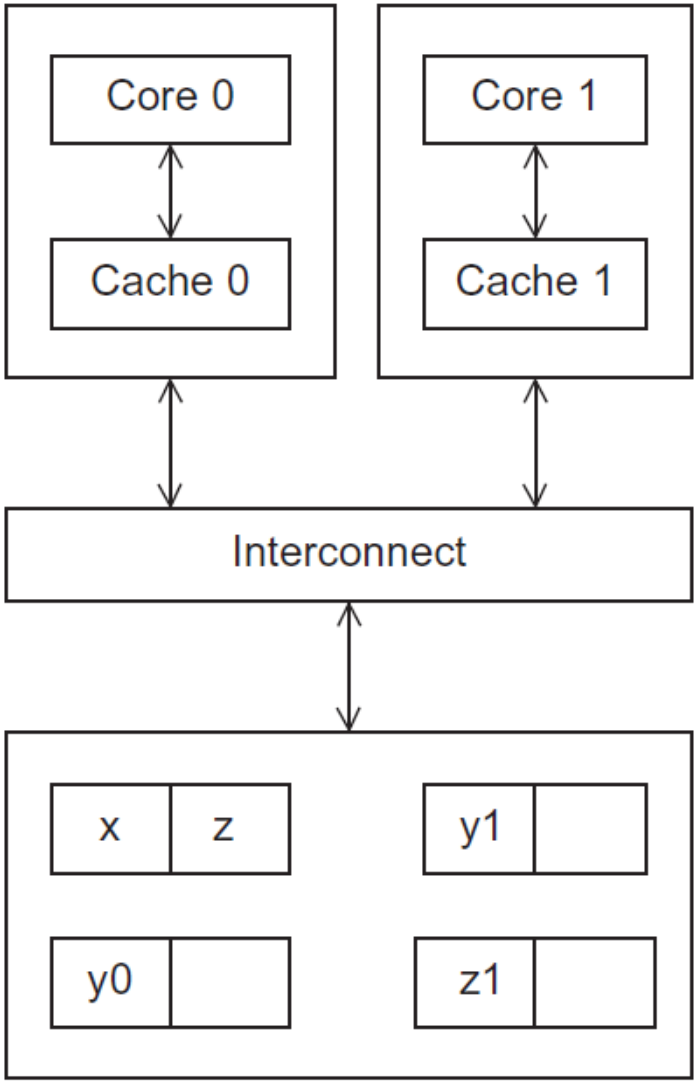


Figure 2.17

有两个核和两个Cache的共享内存系统

Cache一致性

y0 是 Core 0 的私有变量

y1 和 z1 Core 1 的私有变量

x = 2; /*共享变量*/

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

y0 最终结果是 2

y1 最终结果是 6

z1 = ???

监听 Cache 一致性

- 多个核共享总线.
- 在总线上传输的任何信号都可以被连接到总线的所有核“看”到.
- 当core 0更新存储在其Cache中的x的副本时，它也会通过总线广播该信息.
- 如果core1正在“监听”总线，它将知道x已被更新，它将自己Cache中的x副本标记为非法的.

基于目录的 Cache 一致性

- 使用一种称为目录的数据结构，该目录存储每个Cache line的状态.
- 当一个变量被更新时，就会查询目录，并将所有包含该变量的cache line置为非法



并行软件

系统开发成本主要在于软件

- 硬件和编译器可以跟上需求速度
- 目前为止...
  - 在共享内存程序中:
    - 启动单个进程并fork多个线程.
    - 多个线程执行任务.
  - 在分布式存储程序中:
    - 启动多个进程.
    - 进程执行任务.

SPMD – single program multiple data

- SPMD程序由单个可执行程序组成，通过使用条件分支，该程序的行为就像多个不同的程序一样.

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



并行程序的编写

1. 在进程/线程之间分配工作
   (a) 每个进程/线程得到的工作量大致相同（负载均衡）
   (b) 通信最小化.
2. 安排进程/线程同步.
3. 安排进程/线程之间的通信.

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

共享存储器

- 动态线程
  - 主线程等待任务，fork新线程，当新线程完成时任务后，结束新线程
  - 资源的有效使用，但是线程的创建和终止非常耗时.
- 静态线程
  - 创建线程池并分配任务，但线程不被终止直到被清理.
  - 性能更好，但可能会浪费系统资源.

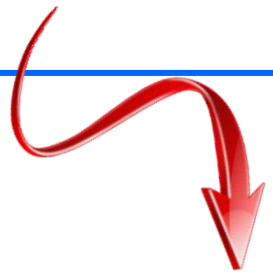
非确定性

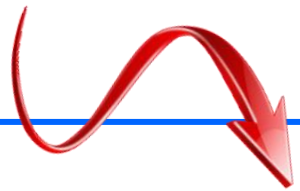
```
my_val = Compute_val ( my_rank ) ;
x += my_val ;
```

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

非确定性

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





Thread 0 > my_val = 7

Thread 1 > my_val = 19

Thread 1 > my_val = 19

Thread 0 > my_val = 7

非确定性

- 竞争条件
- 临界区
- 互斥
- 互斥锁 (互斥量/锁)

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

忙等待

```
my_val = Compute_val ( my_rank ) ;
i f ( my_rank == 1)
    whi l e ( ! ok_for_1 ) ; /*忙等待循环*/
x += my_val ; /*临界区*/
i f ( my_rank == 0)
    ok_for_1 = true ; /*让线程1更新x*/
```

消息传递

```
char message [ 1 0 0 ] ;
. . .
my_rank = Get_rank ( ) ;
i f ( my_rank == 1) {
    sprintf ( message , "Greetings from process 1" ) ;
    Send ( message , MSG_CHAR , 100 , 0 ) ;
} e l s e i f ( my_rank == 0) {
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
    printf ( "Process 0 > Received: %s\n" , message ) ;
}
```

划分全局地址空间的语言

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

输入/输出遵循的规则

- 在分布式内存程序中，只有进程0将访问*stdin*。在共享内存程序中，只有主线程或线程0将访问*stdin*。
- 在分布式内存和共享内存程序中，所有进程/线程都可以访问*stdout*和*stderr*。

输入/输出遵循的规则

- 然而，由于输出到标准输出*stdout*的顺序的不确定性，在大多数情况下，除了调试输出外，只有一个进程/线程将用于所有到标准输出*stdout*的输出。
- 调试输出应该始终包含生成输出的进程/线程的rank或ID

输入/输出遵循的规则

- 只有单个进程/线程会尝试访问除*stdin*、*stdout*或*stderr*之外的任何单个文件。例如，每个进程/线程可以打开自己的私有文件进行读写，但是没有两个进程/线程会打开相同的文件



性能

加速比



- 核数 = p
- 串行程序运行时间= T_{serial}
- 并行程序运行时间= $T_{parallel}$

线性加速比

$$T_{parallel} = T_{serial} / p$$

并行程序加速比

$$S = \frac{T_{serial}}{T_{parallel}}$$

并行程序效率

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

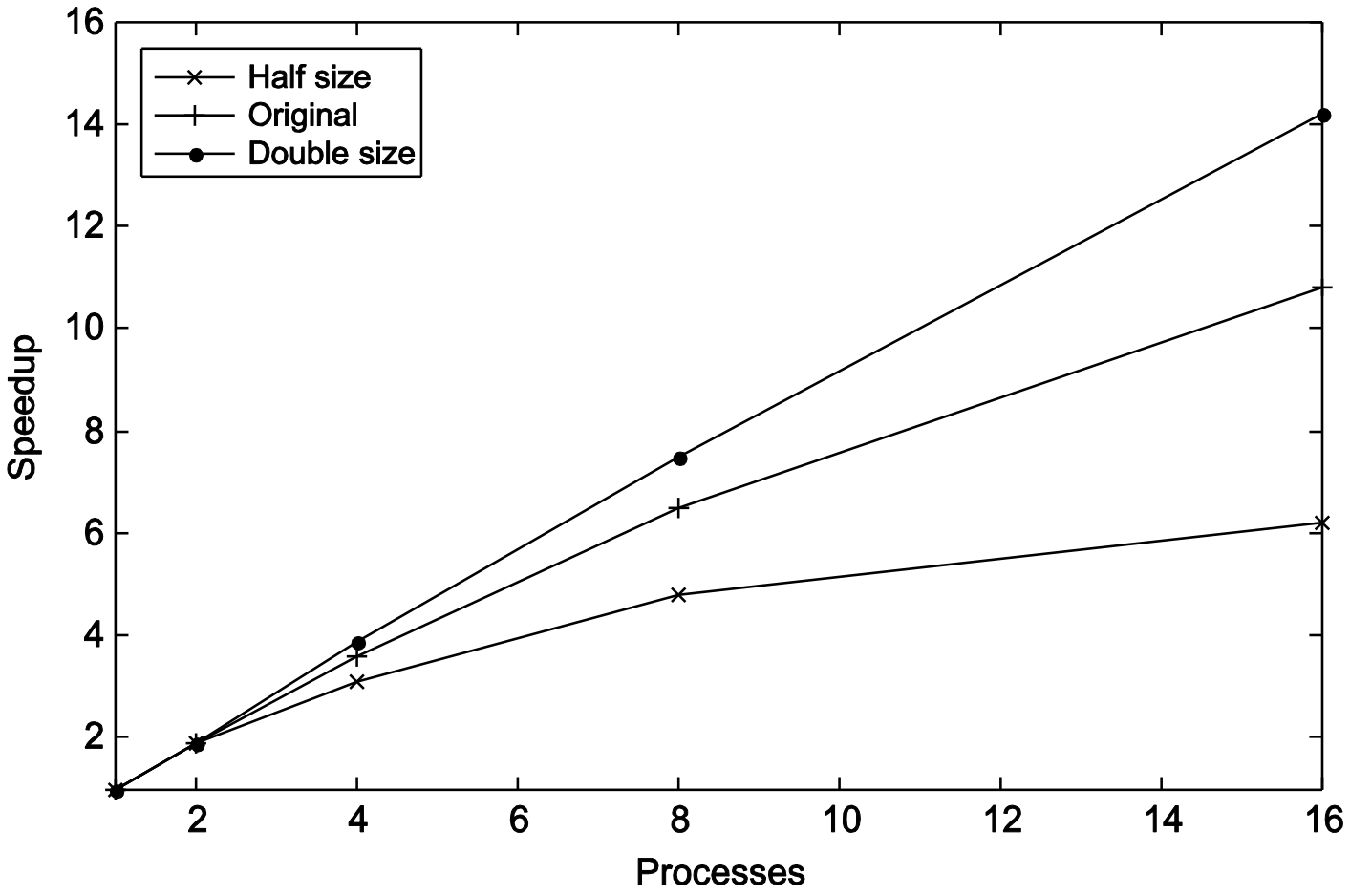
并行程序的加速比和效率

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

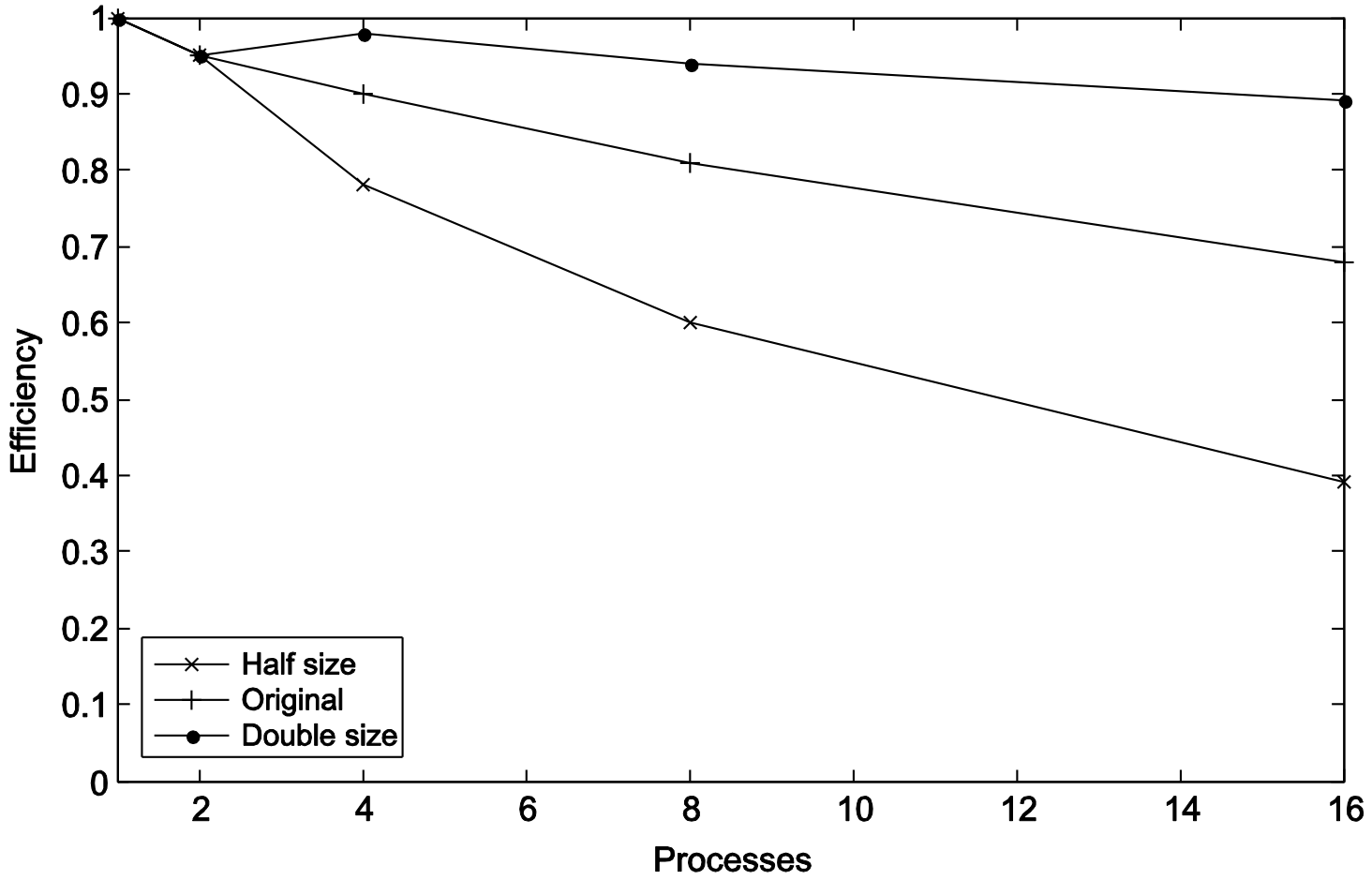
不同规模的问题上并行程序加速比和效率

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

加速比


Half size
Original
Double size
Speedup
Processes
16
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16

效率


Efficiency
Processes
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
Half size
Original
Double size

并行开销的影响

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

阿姆达尔定律（Amdahl's law）

- 除非所有的串行程序都能够并行，否则无论可用的核的数量再多，加速将非常有限



Example

- 90%的串行程序可以并行化.
- 无论使用多少核p，并行化都是“完美的”
- $T_{serial} = 20$ 秒
- 可并行部分运行时间为

$$0.9 \times T_{serial} / p = 18 / p$$

Example (cont.)

- 不可并行化”部分的运行时间为：

$$0.1 \times T_{serial} = 2$$

- 总的运行时间为：

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Example (cont.)

- 加速比

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

可扩展性

- 粗略地讲，如果一个技术可以处理规模不断增加的问题，那么它就是可扩展的
- 如果在增加进程/线程的数量时，可以维持固定的效率，缺不增加问题规模，那么程序称为强可扩展（*strongly scalable*）.
- 如果在增加进程/线程数量的同时，只有以相同倍率增加问题规模才能使效率值保持不变，那么程序就称为弱可扩展的（*weakly scalable*）

计时

- What is time?
- 开始到结束时间?
- 对感兴趣的部分程序计时?
- CPU 时间?
- 墙上时钟?



计时

虚构的函数

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

MPI_Wtime

omp_get_wtime

计时

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

计时

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```



并行程序设计

Foster的方法

1. 划分（Partitioning）:将要执行的指令和数据按照计算拆分为多个小任务。

   这一步的关键在于识别出可以并行执行的任务

Foster的方法

2. 通信（Communication）:确定前一步所识别出来的任务之间需要执行哪些通信。



Foster的方法

3. 聚集或聚合（Agglomeration or aggregation）:将第一步中确定的任务和通信合并成更大的任务。

例如，如果任务A必须在任务B执行之前执行，那么将它们聚合为单个复合任务可能更为明智。

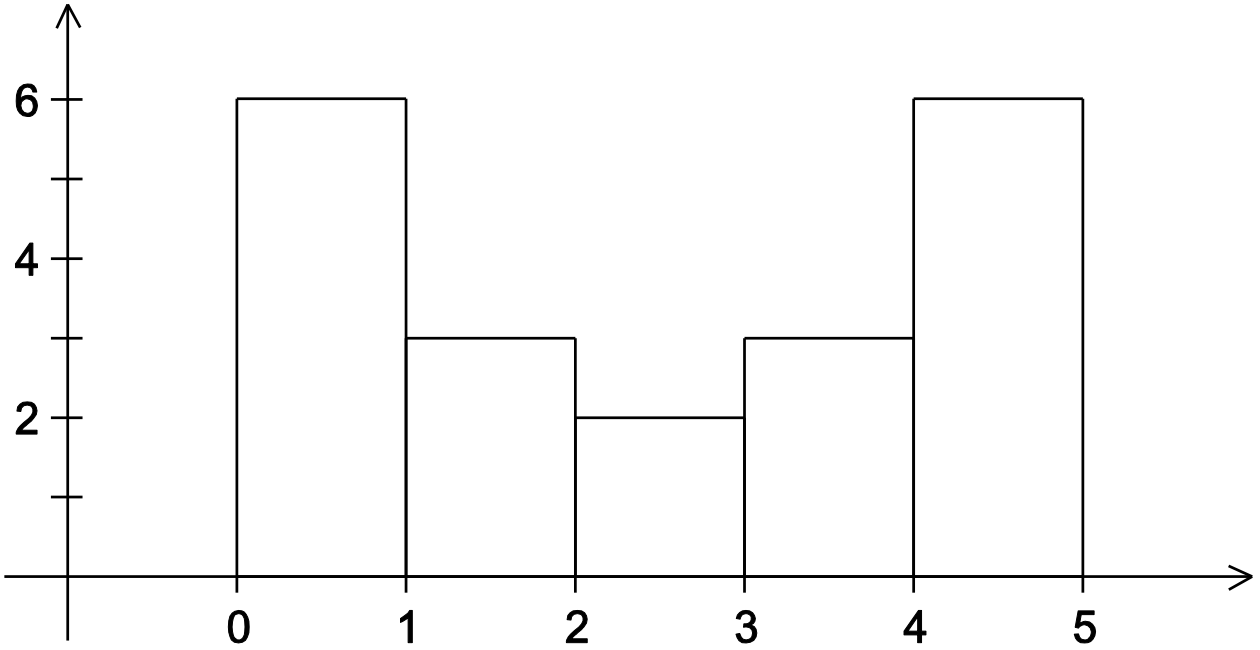
Foster的方法

4. 分配（Mapping）:将上一步聚合好的任务分配给进程/线程.

   这一步还要使通信最小化，使得每个进程/线程得到的工作量大致均衡（负载均衡）.

Example – 直方图（histogram）

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



串行程序--输入部分

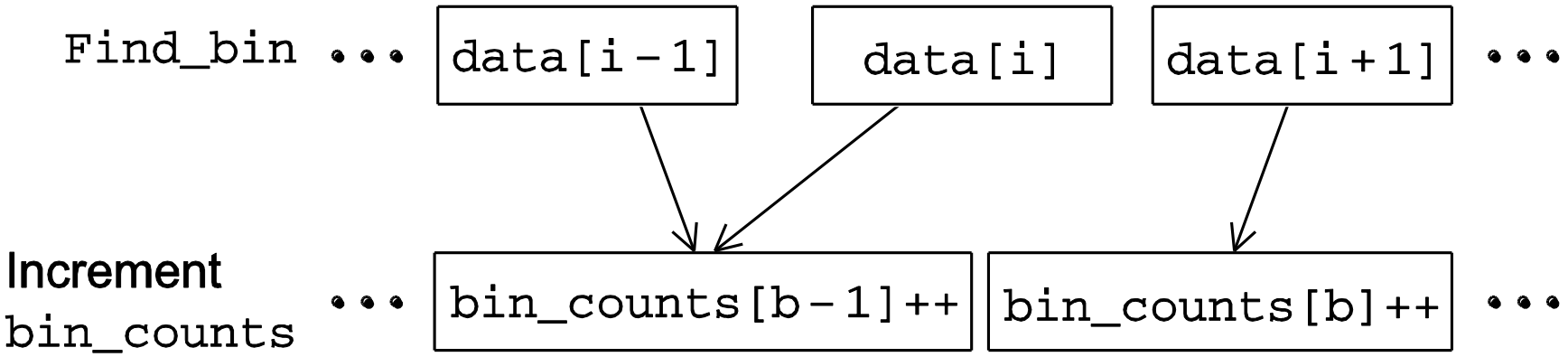
1. 数据的个数: data_count
2. 一个大小为 data_count的浮点数数组data
3. 最小值: min_meas
4. 最大值: max_meas
5. 桶的个数: bin_count

串行程序--输出部分

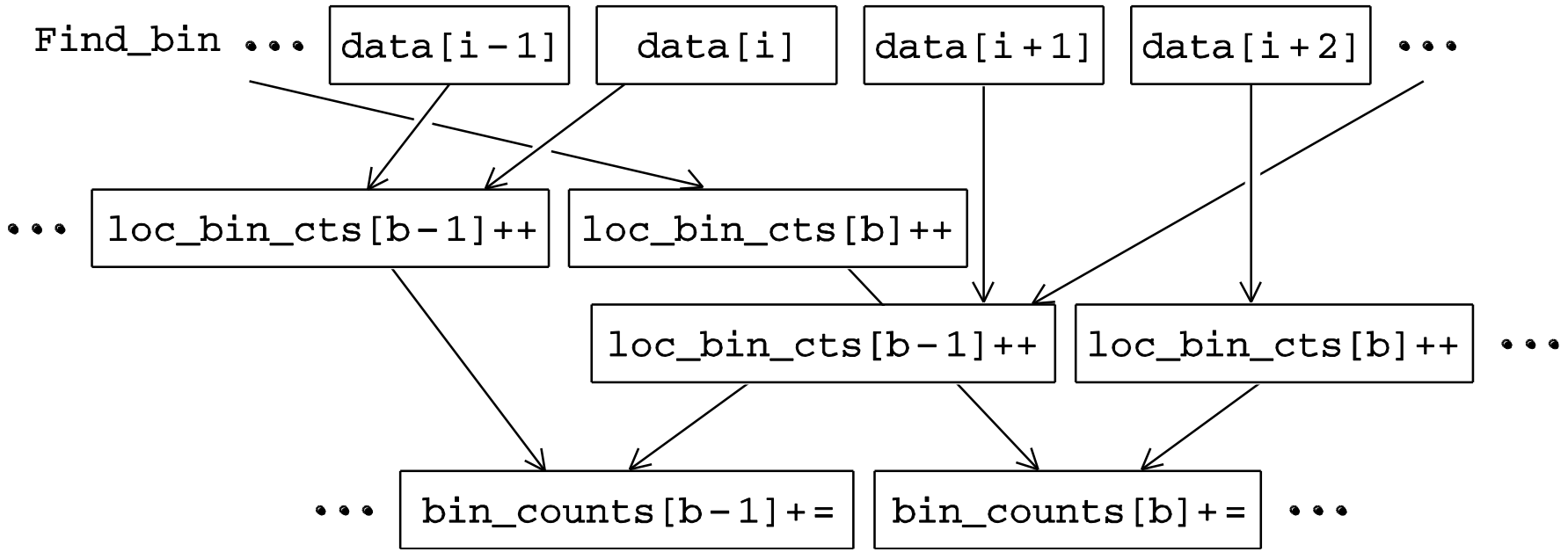
1. bin_maxes : 一个大小为bin_count的浮点数数组

2. bin_counts : 一个大小为bin_count整数数组

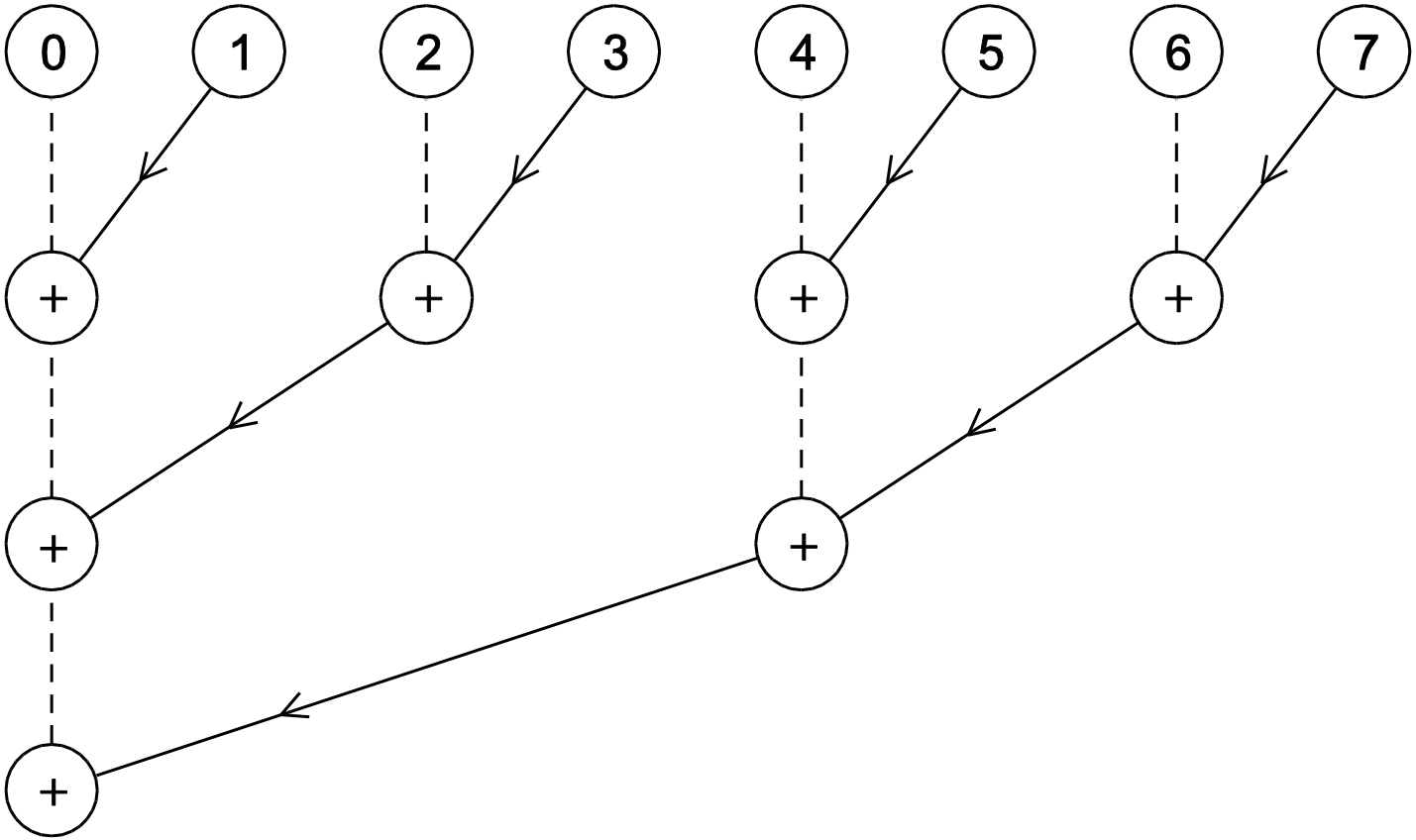
Foster方法的前两步



任务和通信的另一种定义方式



本地数组相加



小结（1）

- 串行系统
  - 计算机硬件的标准模型是冯·诺伊曼架构.
- 并行硬件
  - Flynn分类法.
- 并行软件
  - 我们关注的是同构MIMD系统的软件开发，此类系统的大部分程序是它由一个通过分支获得并行性的单一程序组成.
  - SPMD （单程序多数据流）程序.

小结（2）

- 输入和输出
  - 我们将编写程序，其中一个进程或线程可以访问stdin，所有进程可以访问stdout和stderr.
  - 然而，由于不确定性，除了调试输出外，我们通常只会让一个进程/线程访问stdout.

小结（3）

- 性能
  - 加速比
  - 效率
  - 阿姆达尔定律（Amdahl’s law）
  - 可扩展性（Scalability）
- 并行程序设计
  - Foster方法