并行计算

（Parallel Computing）

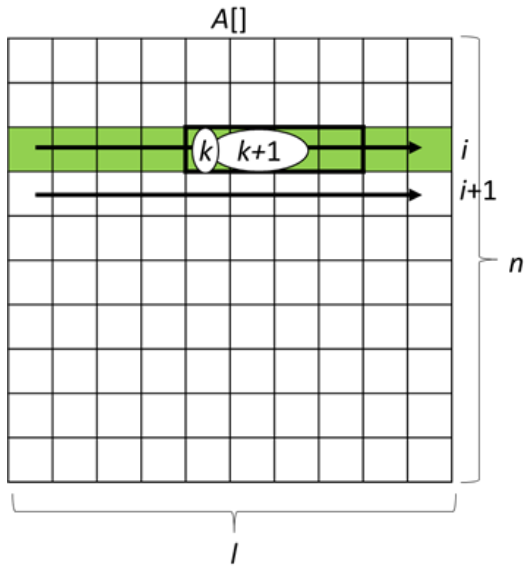
并行硬件和并行软件（二）

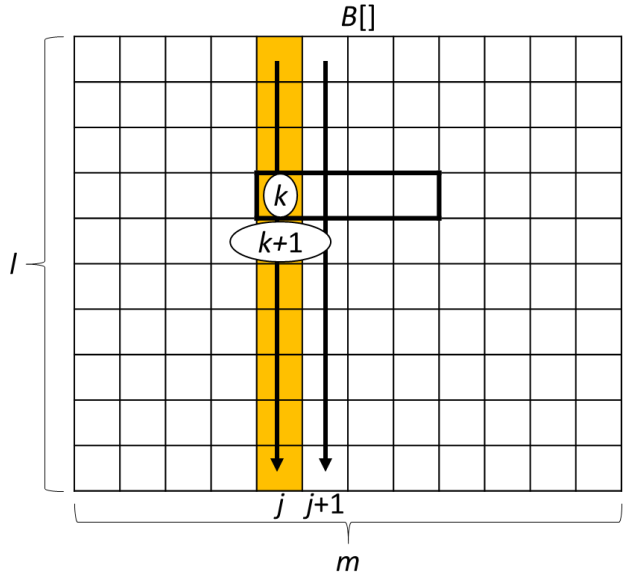
学习内容：

- 背景知识
- 对冯诺依曼架构的改进
- 并行硬件

思考：

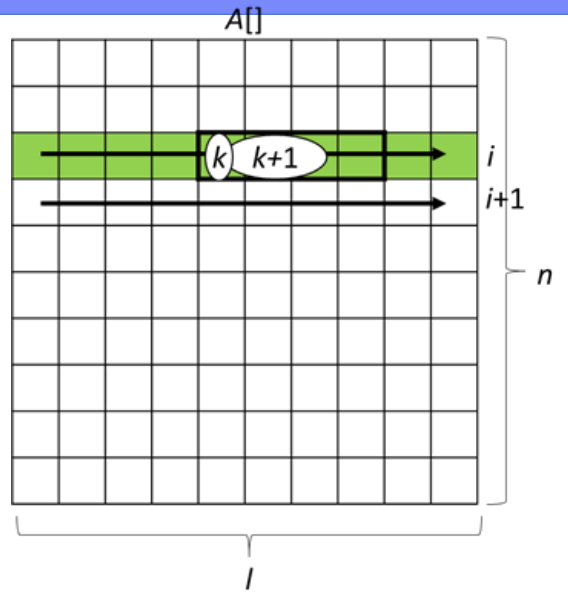
```
//Naive Matrix Multiplication
for (int i = 0; i<n; i++)
    for (int j = 0; j < m; j++) {
        float accum = 0;
        for (k = 0; k < l; k++)
            accum += A[i*l+k]*B[k*m+j];
        C[i*m+j] = accum;
    }
```

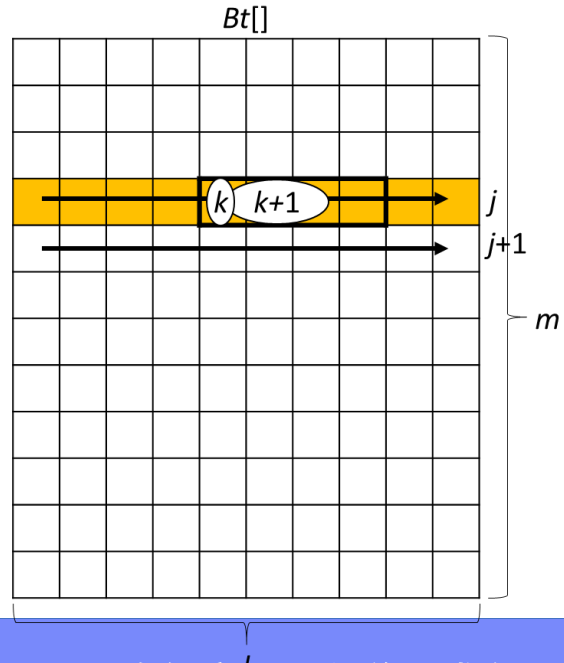




- Matrix multiplication: $A_{n\times l}\cdot B_{l\times m} = C_{n\times m}$

```
//Transpose-and-Multiply
for (k=0; k<l; k++)
    for (j = 0; j<m; j++)
        Bt[j*l+k] = B[k*m+j];
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        float accum = 0;
        for (k=0; k<l; k++)
            accum += A[i*l+k] * B[j*l+k];
        C[i*m + j] = accum;
    }
```





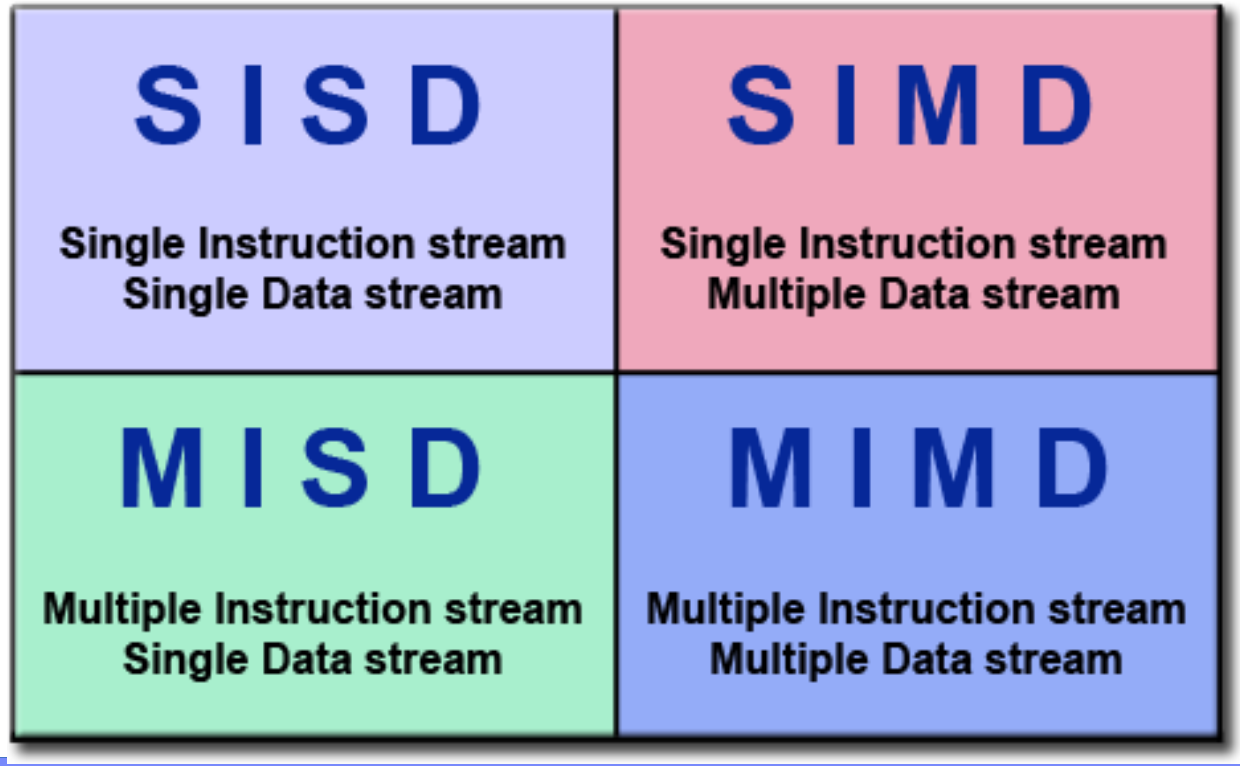
- Transpose-and-Multiply:
  1. $Bt_{m\times l} = (B_{l\times m})^T$
  2. $A_{n\times l} \cdot Bt_{m\times l} = C_{n\times m}$
- Access pattern of A contiguously: $(i,k) \Rightarrow (i,k+1)$
- Accesses pattern of Bt contiguously: $(j,k) \Rightarrow (j,k+1)$

3. 并行硬件

- 多发（Multiple issue）和 流水线（pipelining）通常对程序员不可见
- 并行硬件：对程序员可见，可通过编写代码来利用它

3. 并行硬件

- 费林分类法（Flynn's Taxonomy）：指令流的数量和同时可处理的数据流数量



classic von Neumann

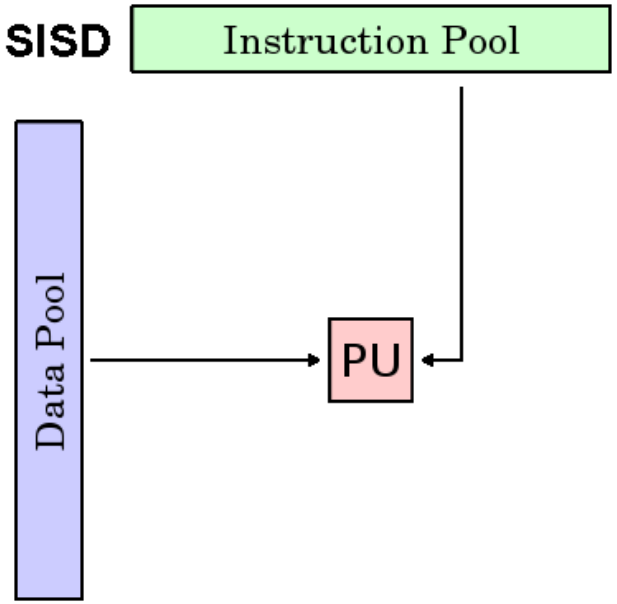
not covered

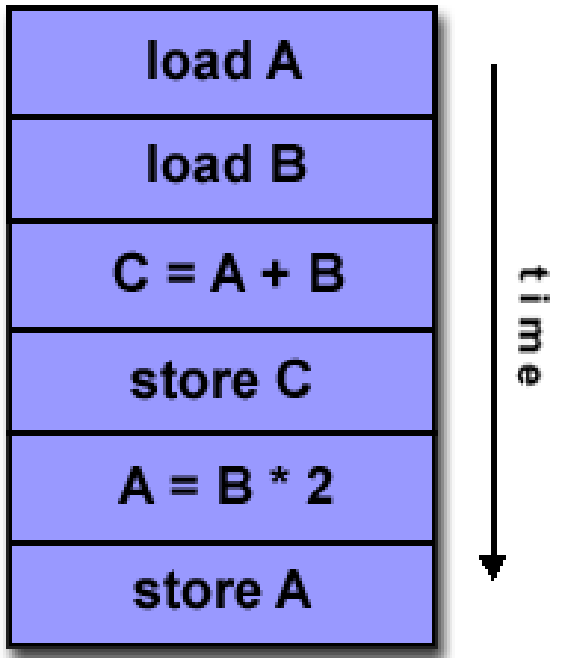
3. 并行硬件

- SISD (Single Instruction, Single Data)
  - 串行计算机
  - Single Instruction：在任何一个时钟周期内，CPU只对一个指令进行操作
  - Single Data：在任何一个时钟周期内，只有一个数据流用作输入
  - 如：老式大型机、小型机、工作站和单处理器PC

3. 并行硬件

- SISD (Single Instruction, Single Data)



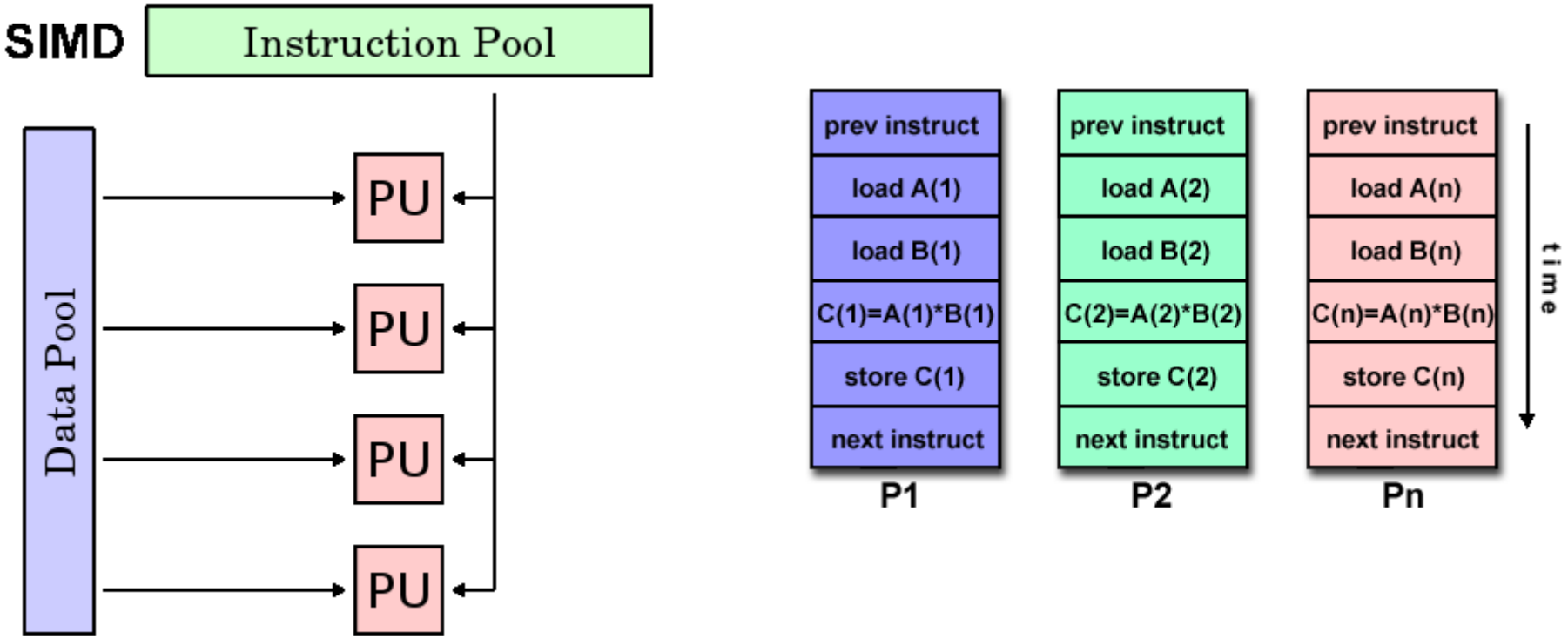


3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - 划分数据给多个处理器
  - 同一指令处理多条数据
  - 数据并行（data parallelism）

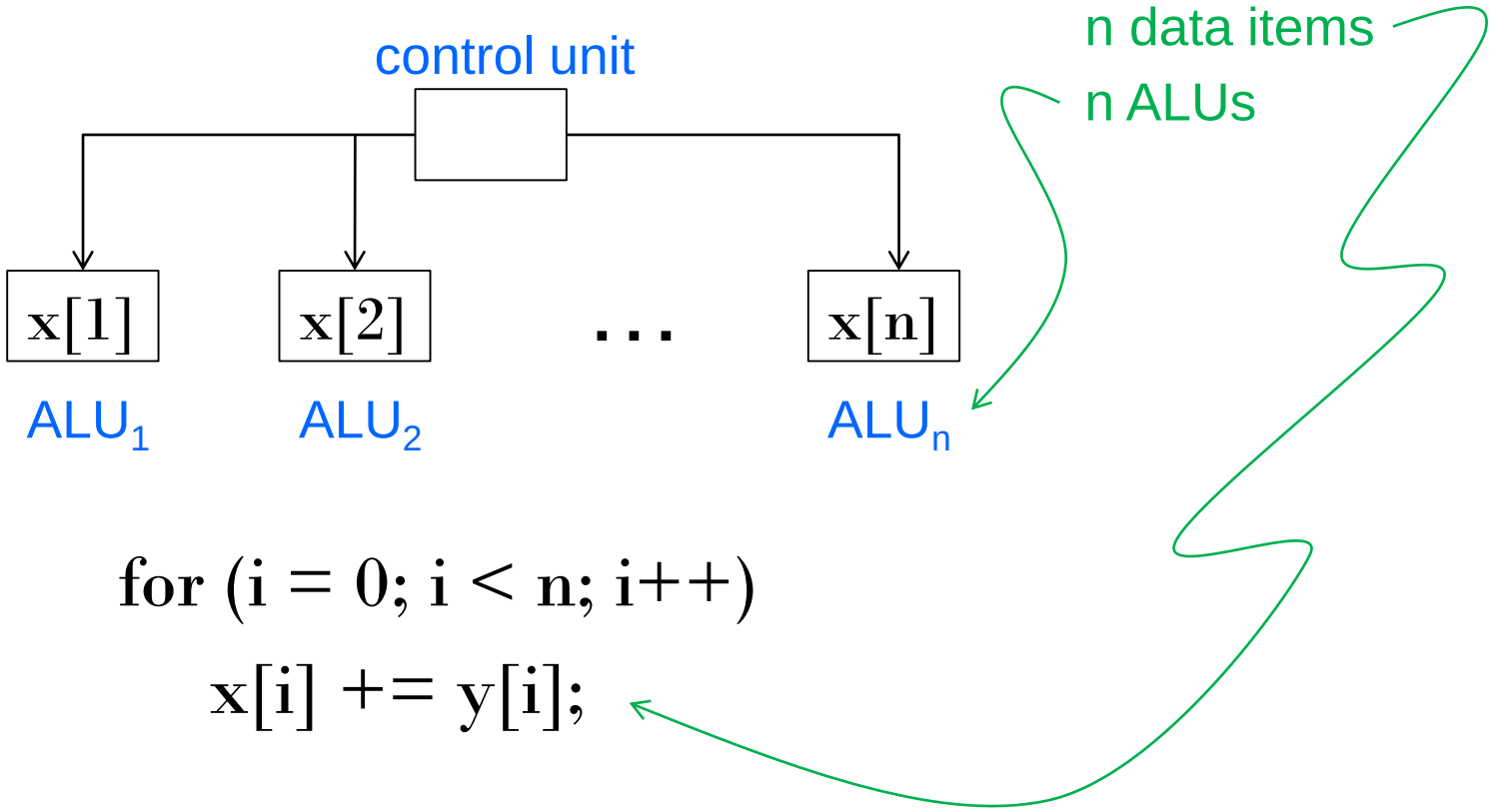
3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)



3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)



3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - *如果我们没有和数据量一样多的ALU，怎么办？*
  - 迭代的划分和处理
  - 如：m = 4 ALUs， 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]	

3. 并行硬件

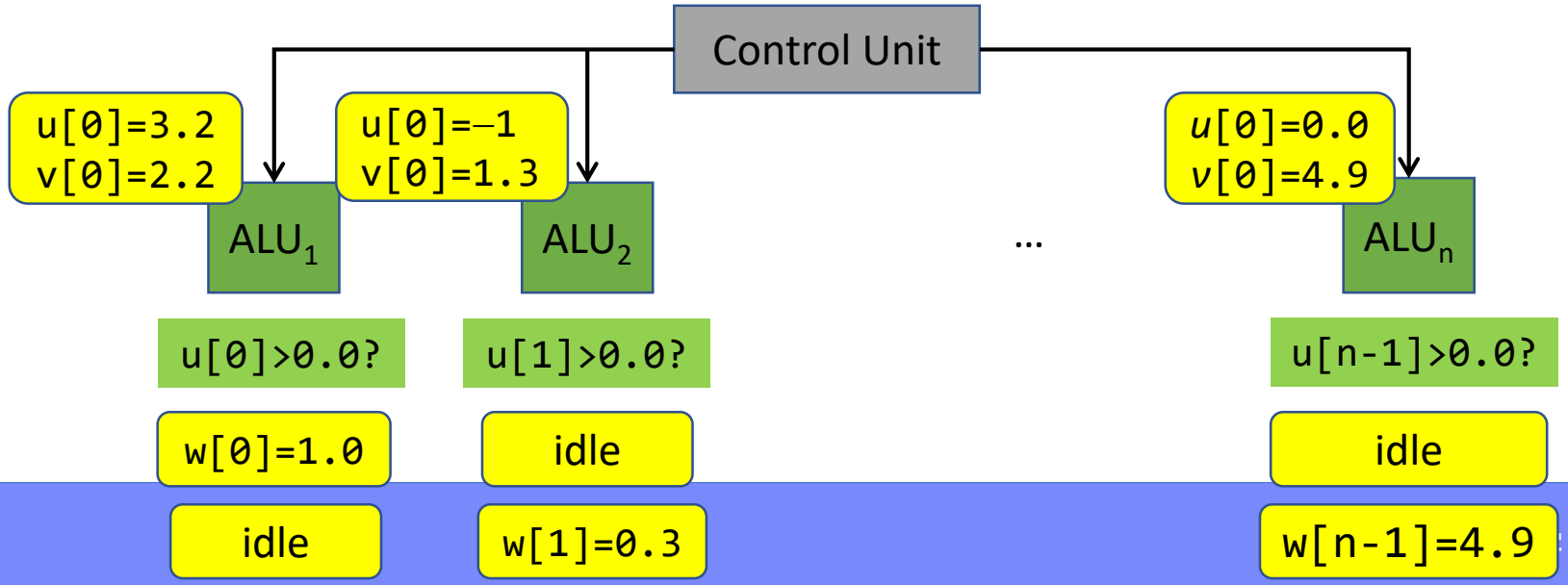
- SIMD (Single instruction stream Multiple data stream)
  - 所有 ALUs 需要执行同一指令，或者空闲
  - 在传统设计中，ALUs 必须同步执行
  - ALUs 没有指令存储
  - 对大数据并行问题有效，但对其他类型更复杂的并行问题无效

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)

```
//Mapping a Conditional Statement onto SIMD
for (i = 0; i<n; i++)
    if (u[i] > 0)
        w[i] = u[i]-v[i];
    else
        w[i] = u[i]+v[i];
```

All ALUs required to execute the same instruction (synchronously) or idle



3. 并行硬件

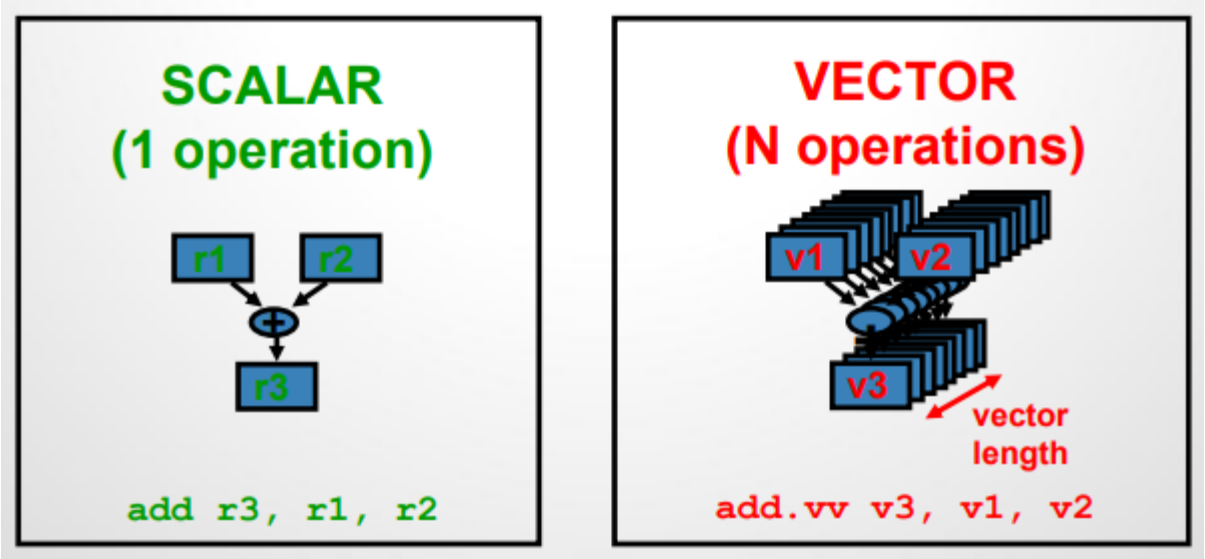
```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

- SIMD (Single instruction stream Multiple data stream)
  - 向量处理器（Vector processors）
    - 对数组或向量进行操作，而传统的 CPU 对单个数据元素或标量进行操作
    - 向量寄存器（Vector registers）：能够存储操作数向量并同时对其内容进行操作，由系统决定，4 – 128 个 64位元素
    - 向量化和流水线功能单元：相同的操作应用于向量中的每个元素（或元素对）（SIMD）
    - 向量指令（Vector instructions）：对向量操作而不是标量

3. 并行硬件

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

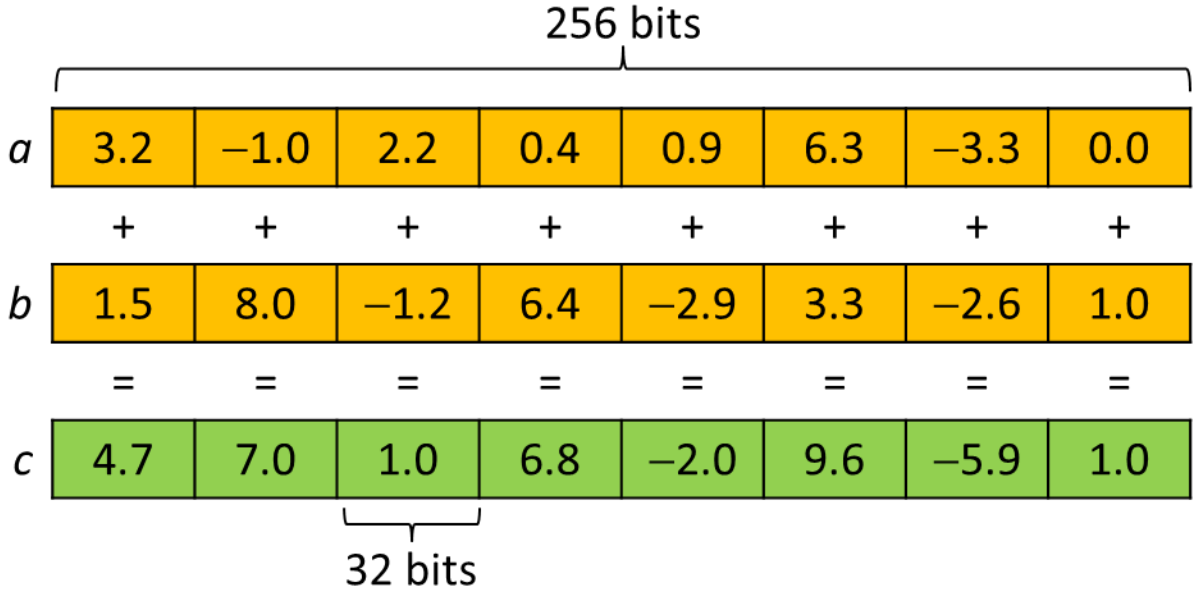
- SIMD (Single instruction stream Multiple data stream)
  - 向量处理器（Vector processors）



3. 并行硬件

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

- SIMD (Single instruction stream Multiple data stream)
  - 向量处理器（Vector processors）




```
//AVX2-Programming with C/C++ Intrinsics
__m256 a, b, c;           // declare AVX registers
...                       // initialize a and b
c = _mm256_add_ps(a, b); // c[0:8] = a[0:8] + b[0:8]
```

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - 向量处理器（Vector processors）

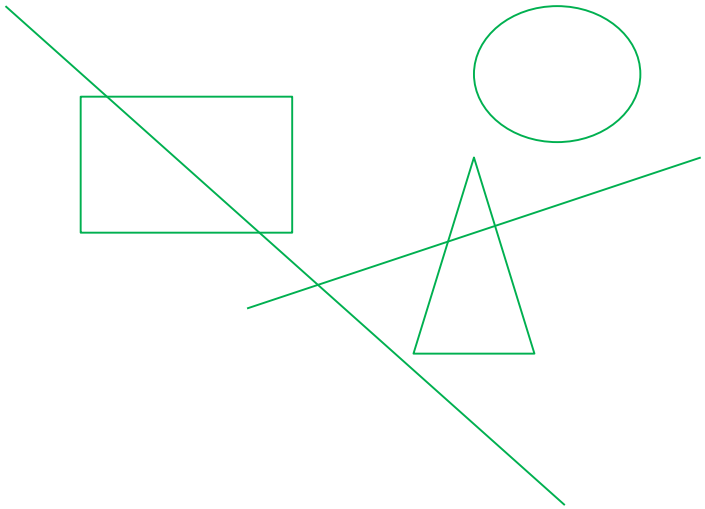
    - 快速、易于使用
    - 向量化编译器擅长识别可以被向量化的代码
    - 编译器还可以提供有关无法向量化代码的信息，帮助程序员重新评估代码
    - 高内存带宽
    - 加载的每个数据元素都被实际使用

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - 向量处理器（Vector processors） 
    - 无法像其他并行架构那样处理不规则数据结构
    - 处理更大问题的能力有限

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - GPU (Graphics Processing Units)
    - 实时图形应用程序编程接口使用点、线和三角形在内部表示物体的表面

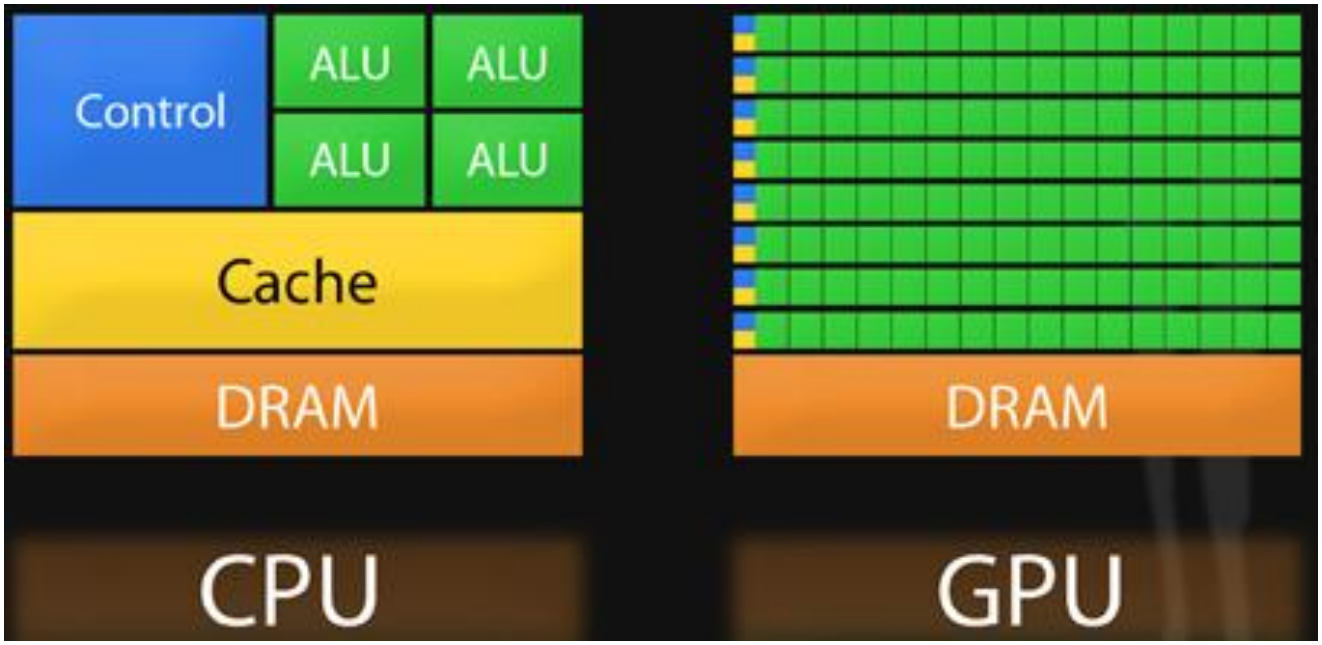


3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - GPU (Graphics Processing Units)
    - 图形处理流水线将内部表示转换为可以发送到计算机屏幕的像素数组
    - 流水线上的几个阶段是可编程的（着色器函数 – shader functions）
    - shader 函数通常很短，只有几行 C 代码
    - shader 函数可以隐式并行，可以应用于图形流中的多个元素

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - GPU (Graphics Processing Units)

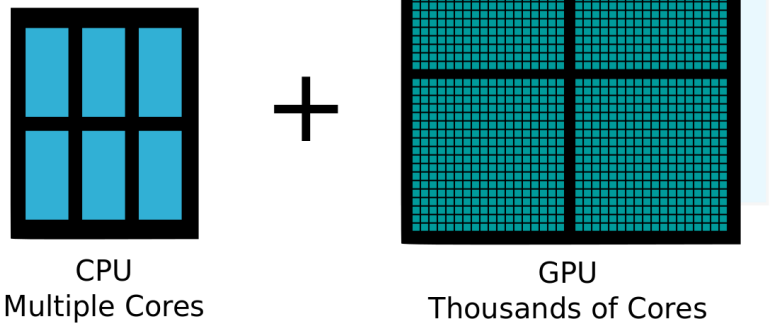


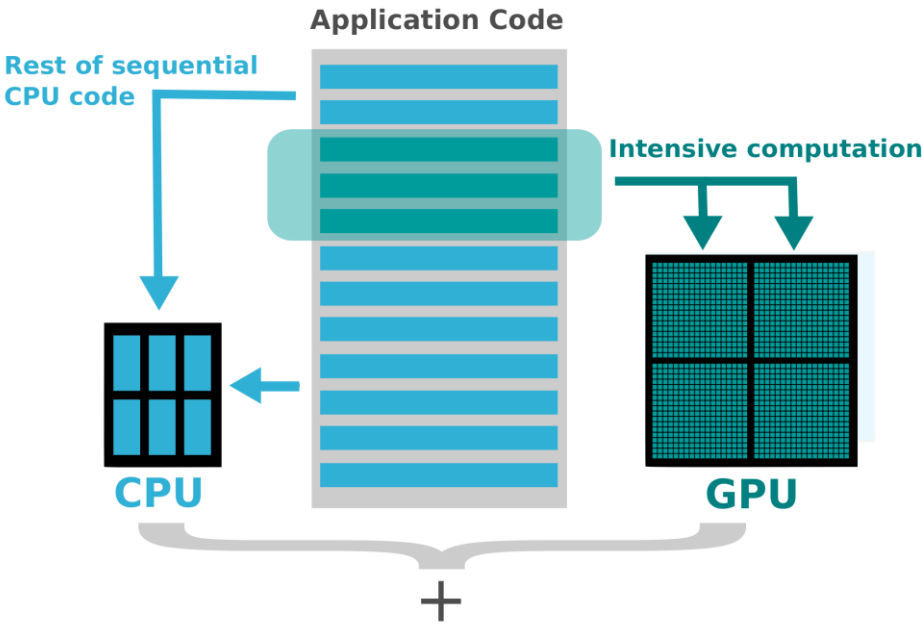
3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - GPU (Graphics Processing Units)
    - GPU通常可以通过使用 SIMD 并行来优化性能
    - 每个 GPU core 中包含大量的 ALUs
    - GPU 不是纯 SIMD 系统，虽然在给定的 core 上 ALUs 可以使用 SIMD 并行，但当前的 GPU 可以有几十个 core，它们可以执行独立的指令流

3. 并行硬件

- SIMD (Single instruction stream Multiple data stream)
  - GPU (Graphics Processing Units)



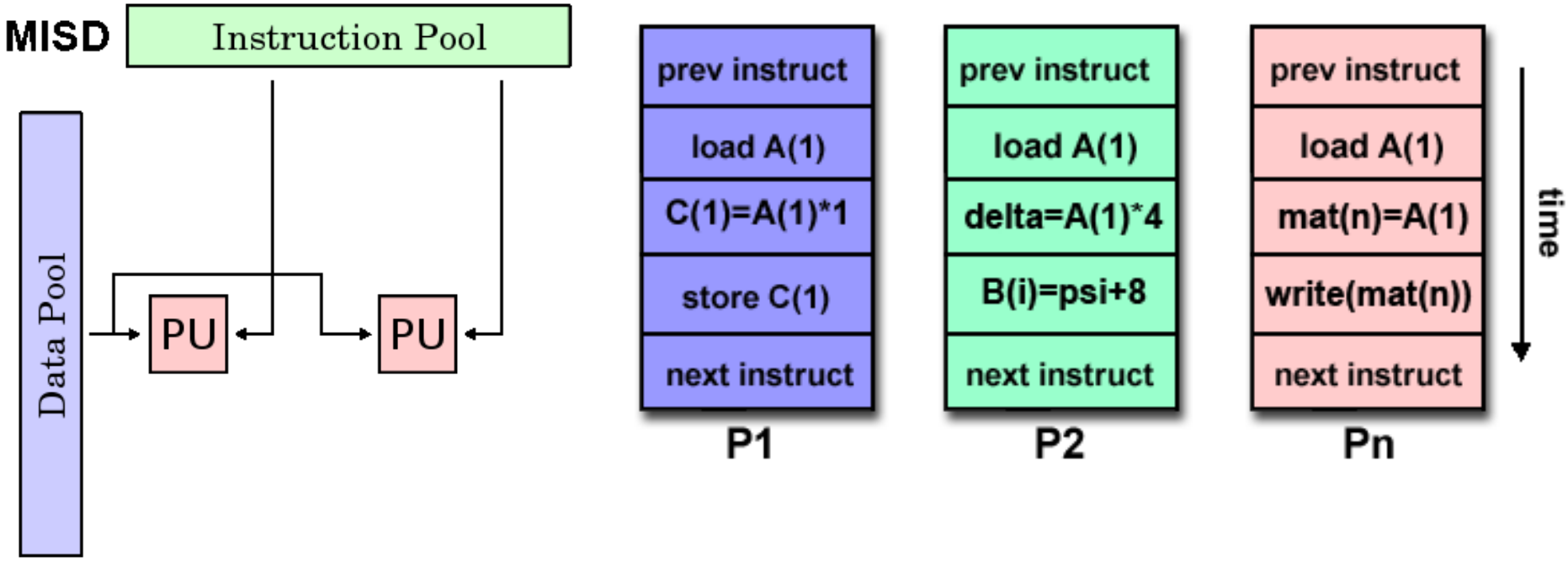


3. 并行硬件

- MISD (Multiple Instruction, Single Data)
  - Multiple Instruction ： 每个处理单元通过单独的指令独立地对数据进行操作
  - Single Data ： 单个数据流被送入多个处理单元
  - 这类并行计算机的实际例子很少，可能的用途：
    - 在单个信号流上工作的多个频率滤波器
    - 多个密码算法试图破解单个编码消息

3. 并行硬件

- MISD (Multiple Instruction, Single Data)

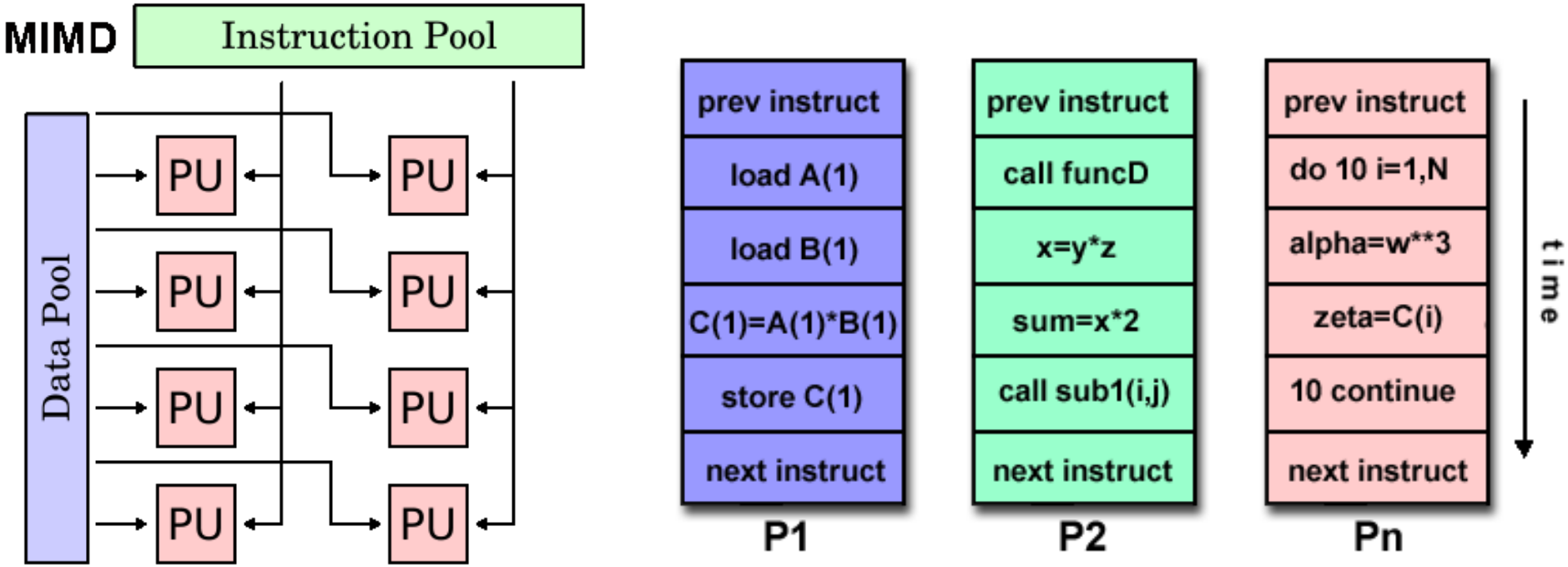


3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 支持多个指令流同时在多个数据流上运行
  - 通常由完全独立的处理单元或核心组成，每个处理单元有自己的控制单元和 ALU
  - 不同于 SIMD 系统，MIMD 系统通常是异步的（asynchronous），各处理器以自己的步伐运行
  - 除非程序员强制执行同步，在任何给定的时刻，处理器可能在执行不同的语句

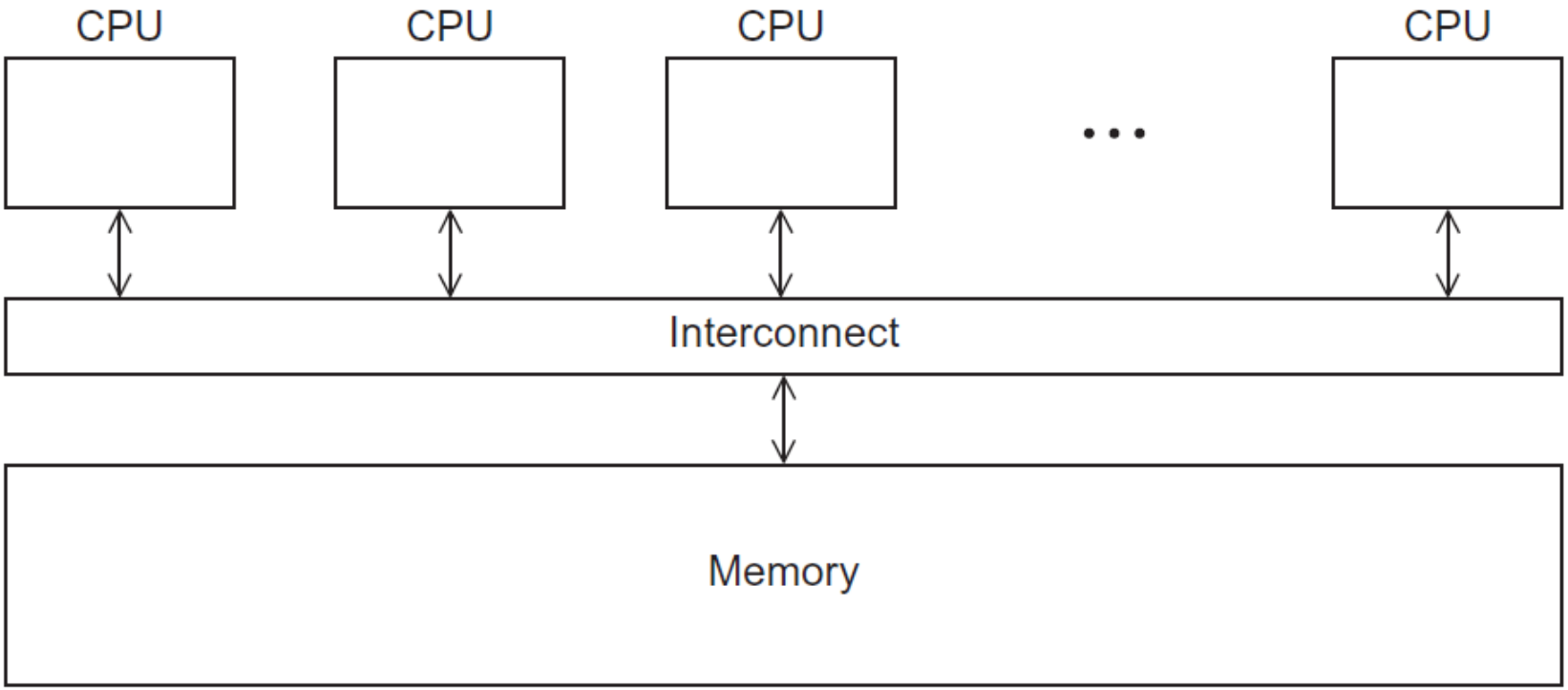
3. 并行硬件

- MIMD (Multiple instruction, multiple data)



3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）

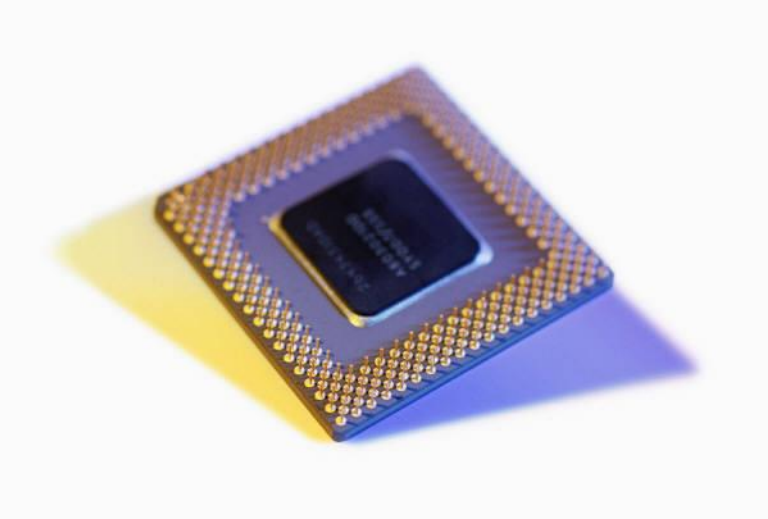


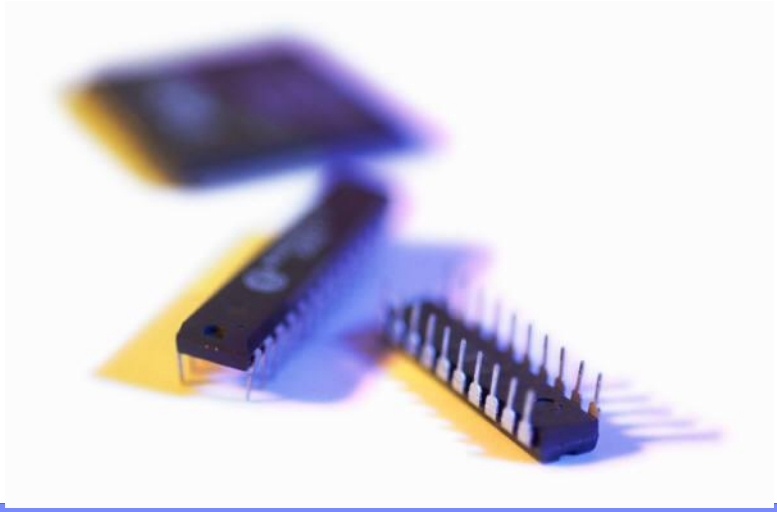
3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）
    - 处理器集合通过互连网络连接到存储器系统
    - 每个处理器都能访问共享的内存单元
    - 处理器通常通过访问共享的数据结构进行隐式通信

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）
    - 最广泛使用的共享内存系统使用一个或多个多核处理器（multicore processor）



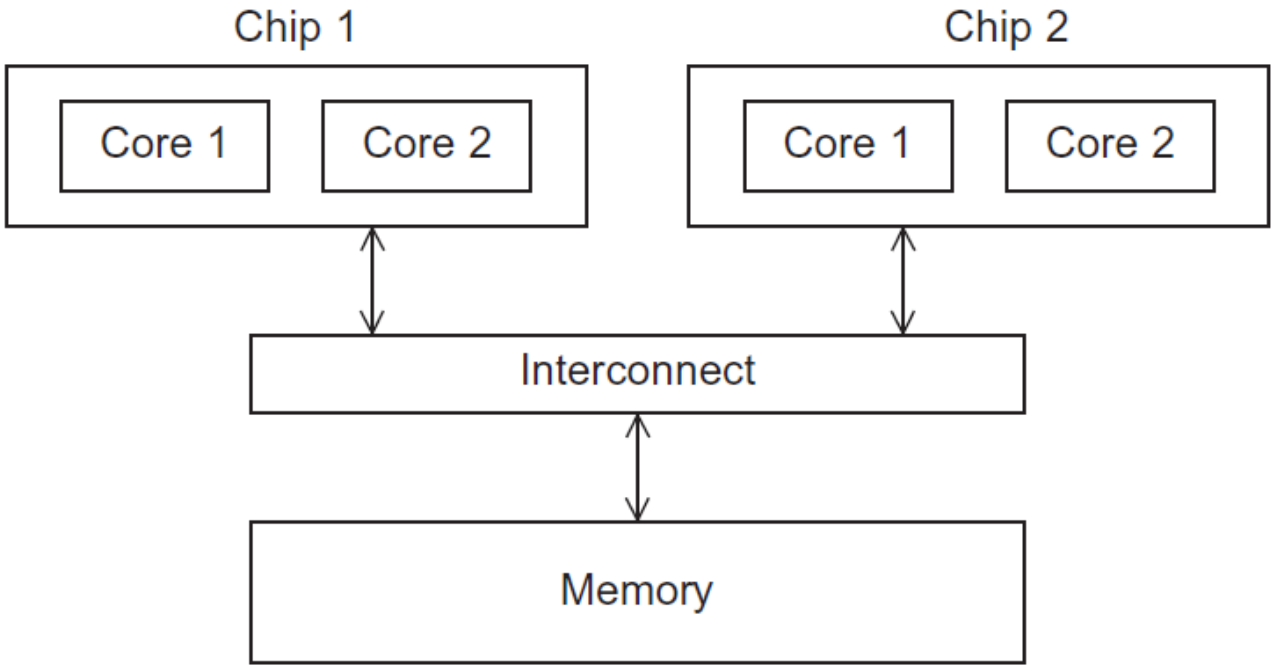


3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）
    - UMA (uniform memory access)：连接所有多核处理器到主存储器
    - NUMA (nonuniform memory access)：每个多核处理器直接连接一块内存，处理器之间可以通过特殊的硬件访问互相的内存

3. 并行硬件

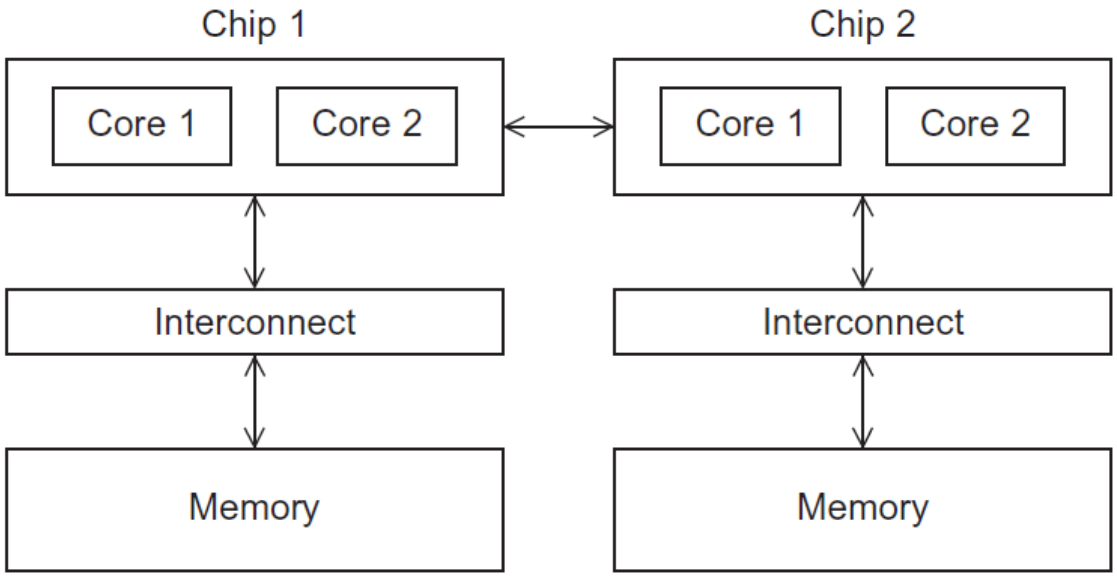
- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）
    - UMA：对于所有core，访问内存的时间相同



3. 并行硬件

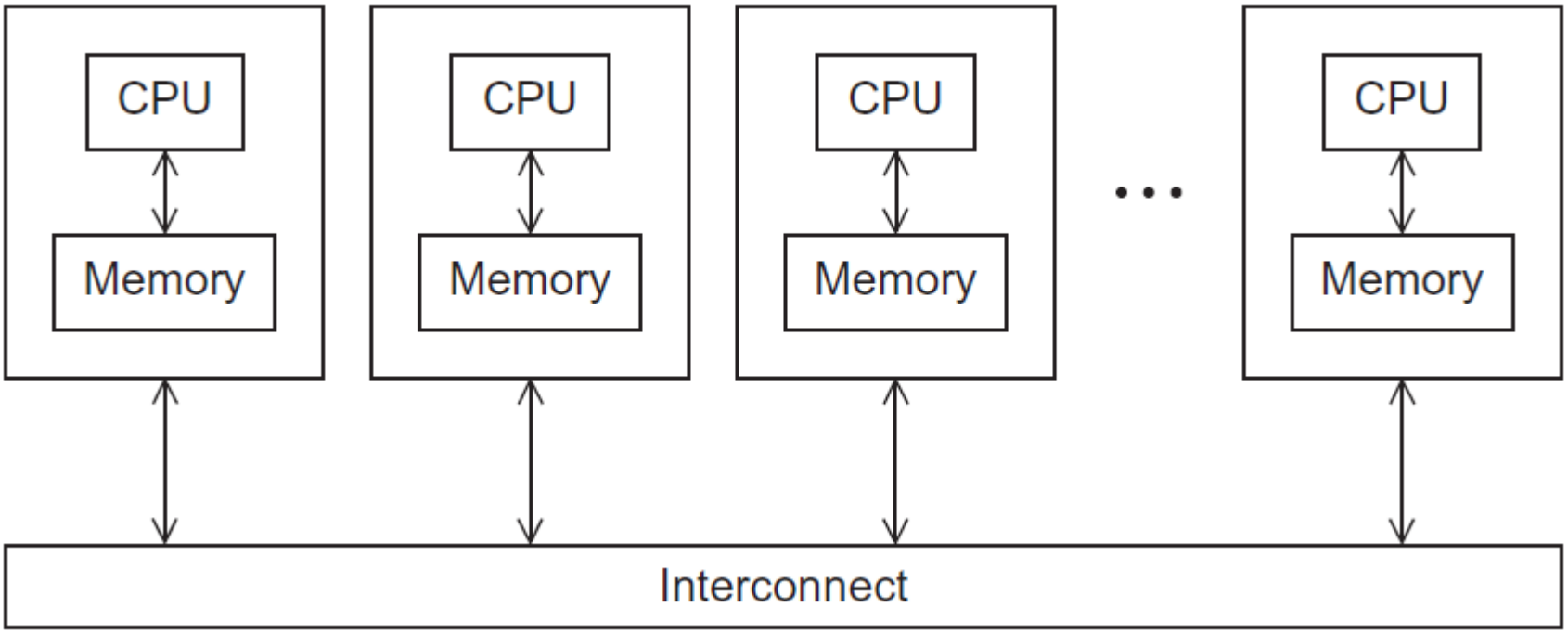
- MIMD (Multiple instruction, multiple data)
  - 共享内存系统（Shared Memory System）
    - NUMA：core访问与之直接相连的内存要快

      core通过其他芯片访问内存要慢



3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存系统（Distributed Memory System）



3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存系统（Distributed Memory System）
    - 每个处理器拥有自己的专用内存
    - 处理器之间通过通信访问其他处理器的内存（发送消息或者通过特殊的函数）

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存系统（Distributed Memory System）
    - 目前最流行的分布式内存系统：集群（Cluster）
    - 商用系统集合（如：PCs）
    - 由商用网络连接（如：Ethernet）
    - 集群的节点（Node）是由通信网络连接的单独计算单元
    - 节点通常是由一个或多个多核处理器构成的共享内存系统
    - 混合系统（hybrid systems）

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 网络互联（Interconnection networks）
    - 影响共享内存系统和分布内存系统性能
    - 共享内存互联
    - 分布内存互联

3. 并行硬件

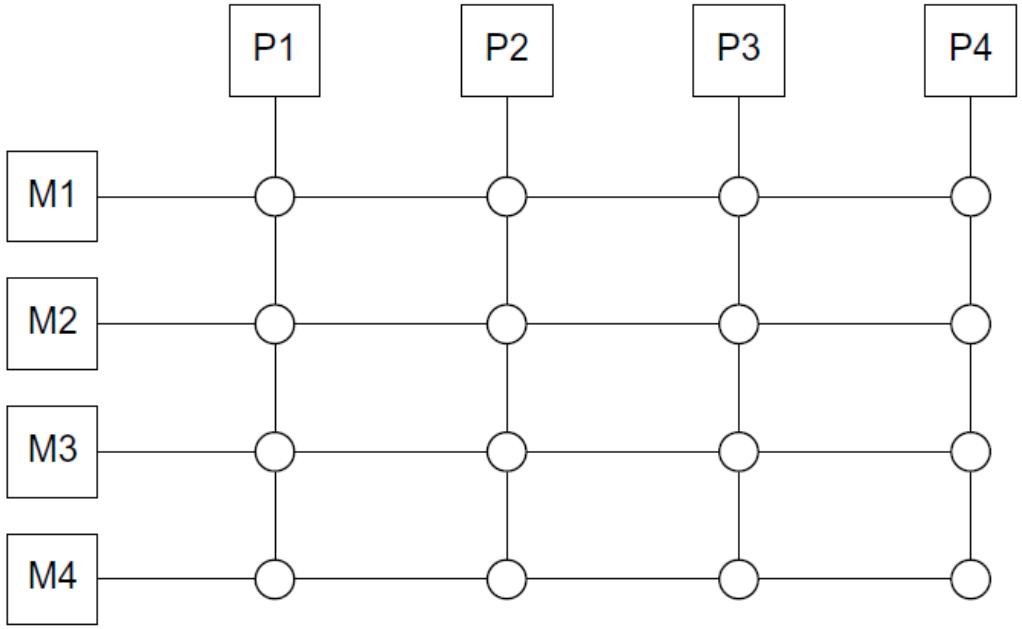
- MIMD (Multiple instruction, multiple data)
  - 共享内存互联
    - 总线（Bus）
      - 一组并行通信线路和一些控制总线访问的硬件
      - 通信线由连接到它的设备共享
      - 随着连接到总线的设备数量增加，总线的竞争增加，导致性能降低

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存互联
    - 交换互联（Switched interconnect）
      - 使用交换机控制连接设备之间的数据路由
      - 交叉开关（Crossbar）
        - 允许不同设备之间同时通信
        - 比总线块
        - 但交换机和链路的成本相对较高

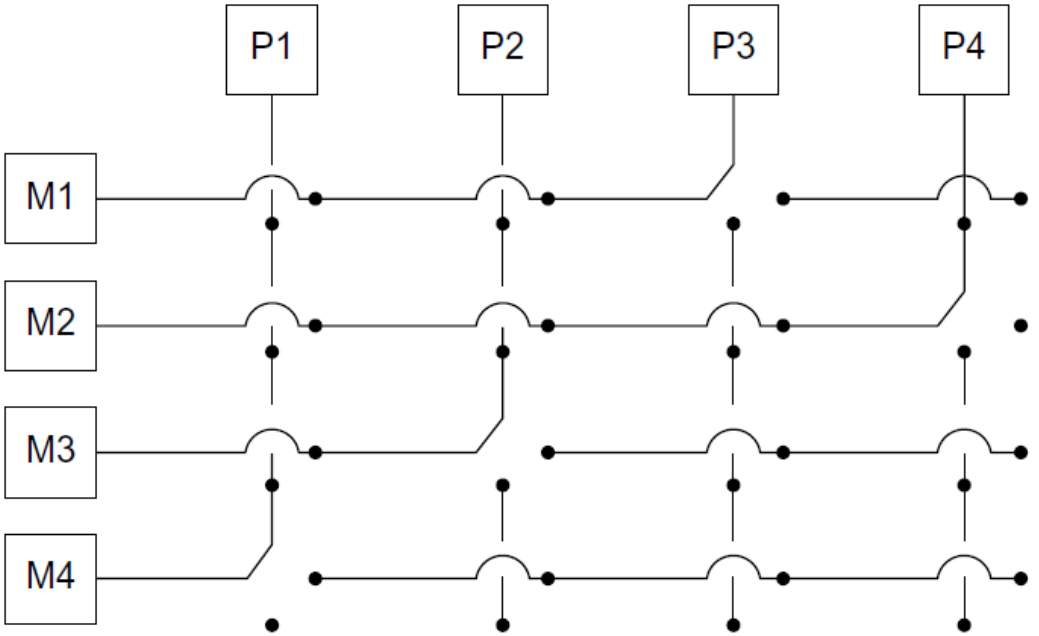
3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存互联
    - 交换互联（Switched interconnect）
      - 交叉开关（Crossbar）



3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 共享内存互联
    - 交换互联（Switched interconnect）
      - 交叉开关（Crossbar）

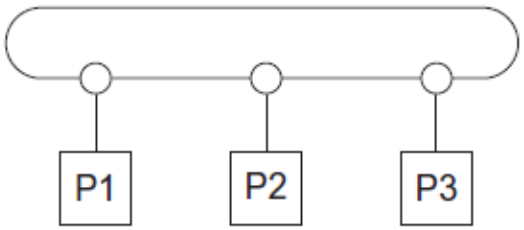


3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 直接互联：每个交换机都直接连接到一个处理器的内存，并且交换机彼此连接
    - 间接互联：交换机可以不直接连接到处理器

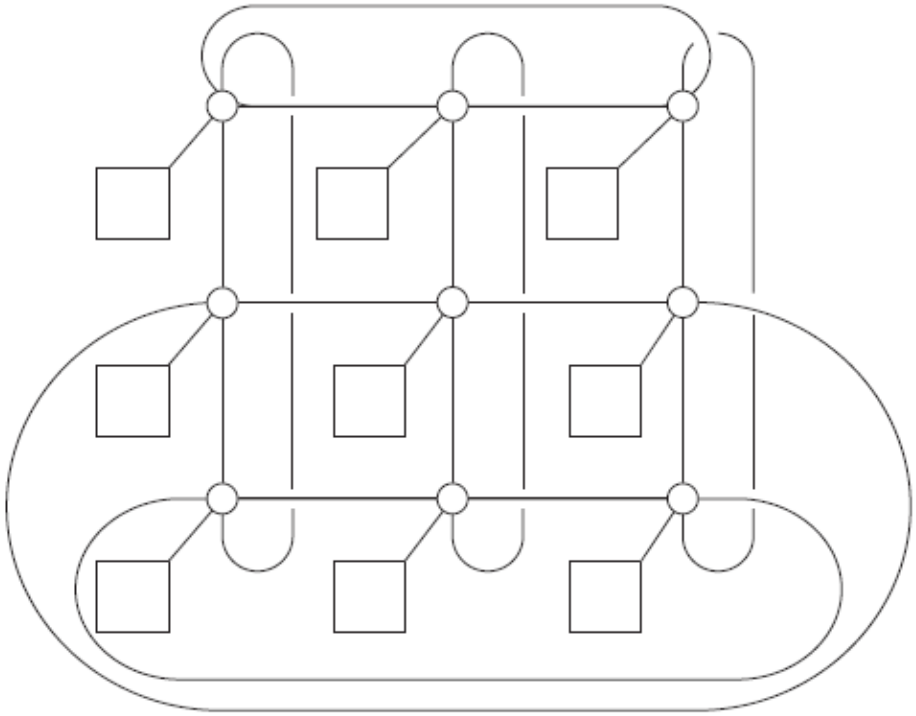
3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 直接互联



(a)

环（ring）



(b)

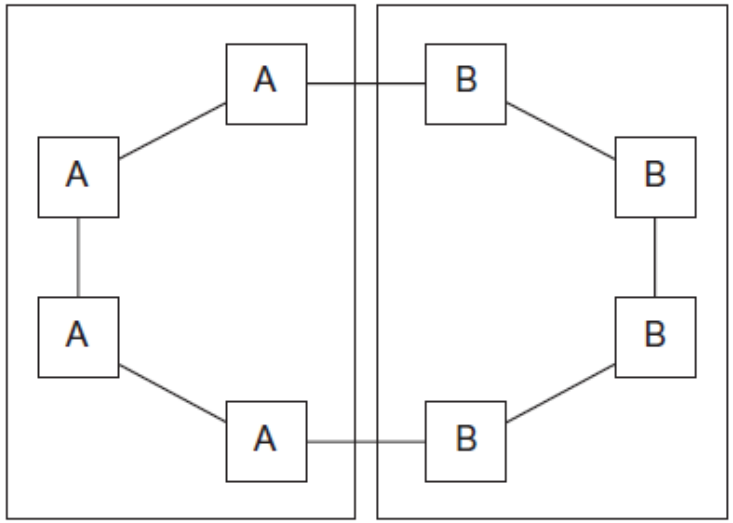
环形网格（ toroidal mesh ）

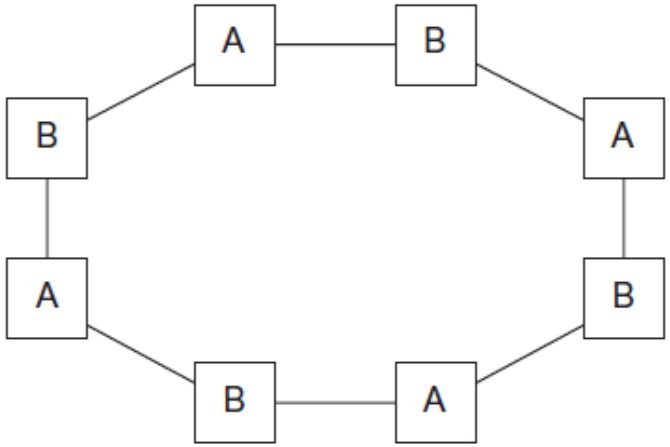
3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 直接互联
      - 等分宽度（Bisection width）：测量“同时通信数量”或“连接性”的一种方法
      - 将网络分成两部分，两部分之间可同时通信的数量

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 直接互联
      - 等分宽度

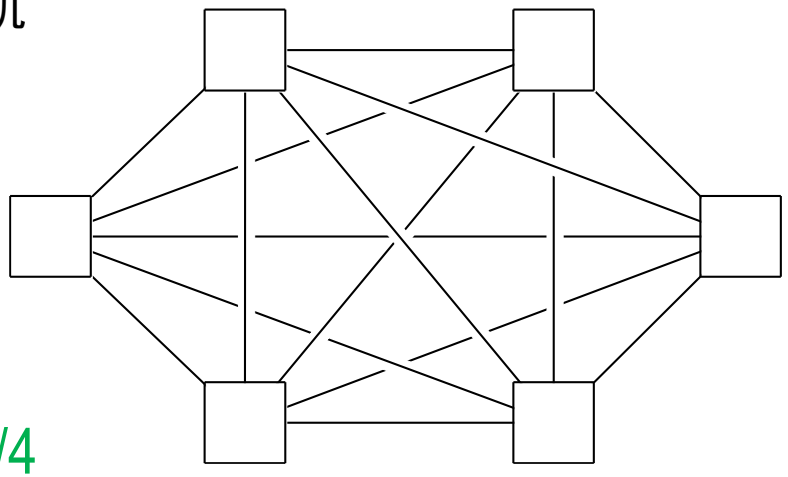




3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 直接互联
      - 最理想的网络是全连接网络（Fully connected network）
      - 每个交换机都直接连接到其他交换机

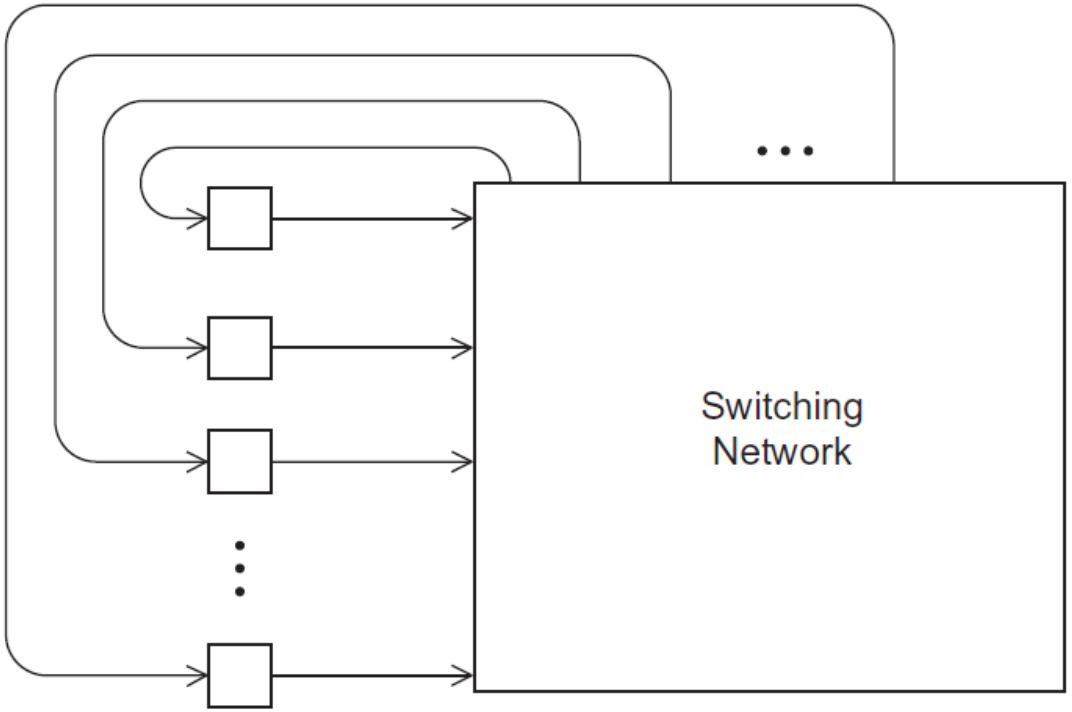
impractical



bisection width = $p^2/4$

3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 分布式内存互联
    - 间接互联：交换机不直接连接处理器



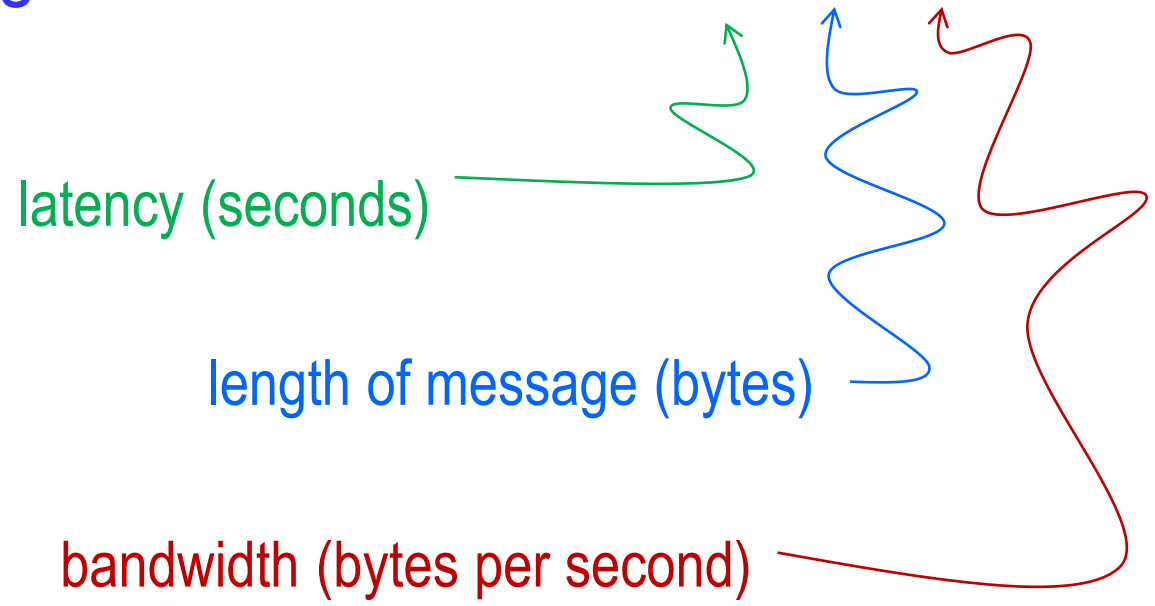
3. 并行硬件

- MIMD (Multiple instruction, multiple data)
  - 延迟和带宽（Latency and bandwidth）
    - 延迟：从源开始传输数据到目标开始接收第一个字节之间经过的时间
    - 带宽：目标在开始接收第一个字节后接收数据的速率，通常以兆位或每秒兆字节为单位

3. 并行硬件

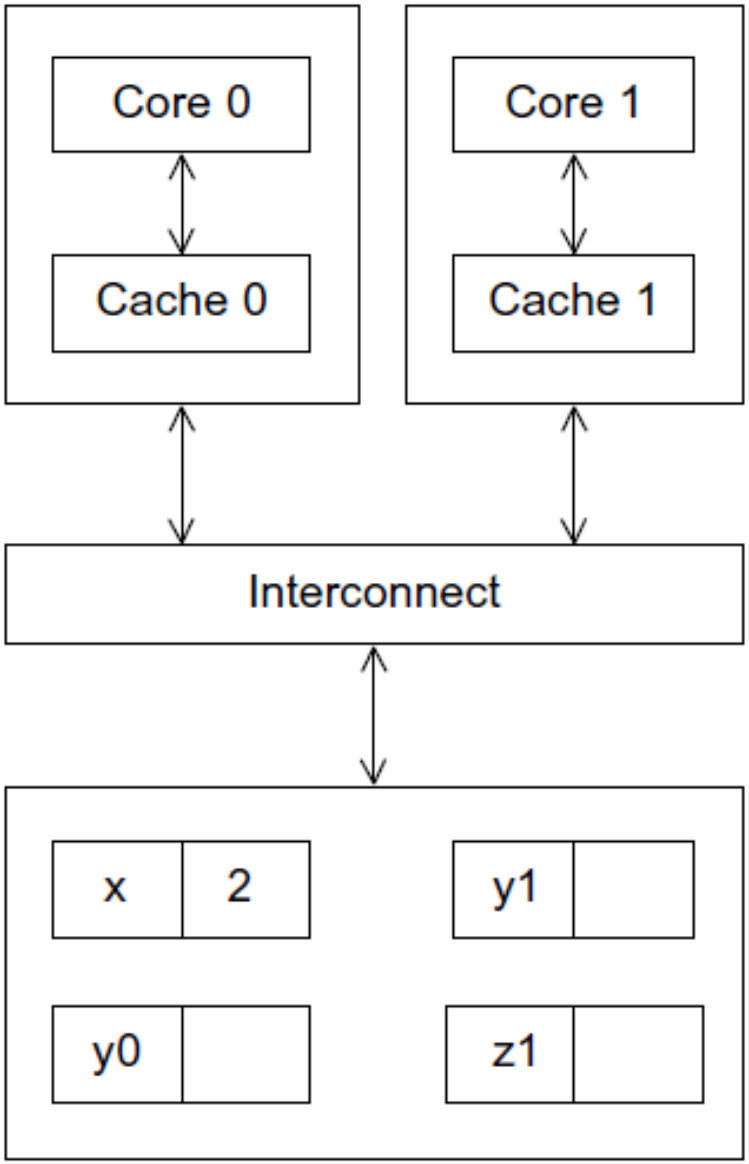
- MIMD (Multiple instruction, multiple data)
  - 延迟和带宽（Latency and bandwidth）

Message transmission time = l + n / b



3. 并行硬件

- 缓存一致性 (Cache coherence)
  - CPU 的缓存由系统硬件管理，程序员无法对其直接控制



y0 为 Core 0 的私有变量
y1 和 z1 为 Core 1 的私有变量

x = 2; /* 共享变量 */

3. 并行硬件

- 缓存一致性 (Cache coherence)

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 = 2

y1 = 6

z1 = ???

y0 为 Core 0 的私有变量

y1 和 z1 为 Core 1 的私有变量

x = 2; /* 共享变量 */

3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 窥探缓存一致性（snooping cache coherence）
  - 基于目录的缓存一致性（Directory-based cache coherence）

3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 窥探缓存一致性（snooping cache coherence）
    - 所有的 core 共享总线
    - 总线上传输的任何信号都可以被连接到总线的 core“看到”
    - 当 core 0 更新存储在其缓存中的 x 的副本时，向总线广播此信息
    - 如果 core 1“窥探”总线，将看到 x 已被更新，可以将其 x 的副本标记为无效
    - 实际的广播只通知其他 core 包含 x 的 cache line 已经更新

3. 并行硬件

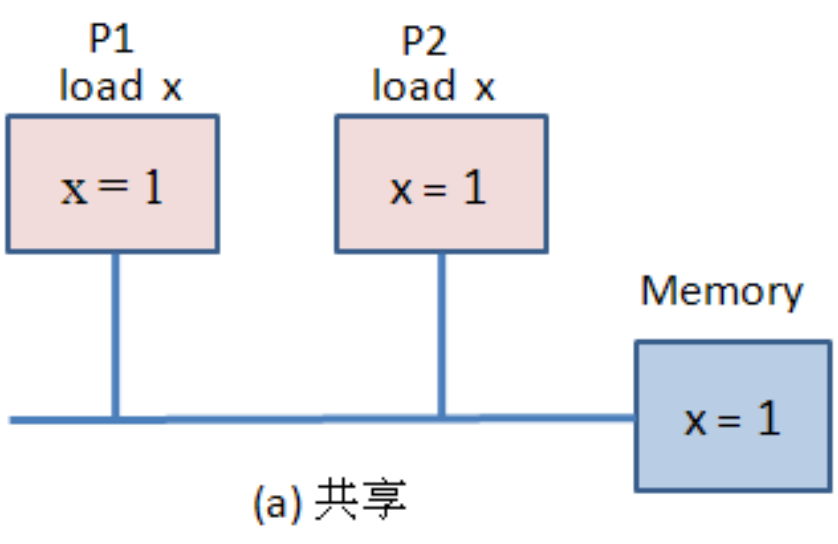
- 缓存一致性 (Cache coherence)
  - 基于目录的缓存一致性（Directory-based cache coherence）
    - 在大型网络中，广播操作是非常昂贵的
    - 窥探缓存一致性在每次有变量更新时，都会产生广播操作，会降低大型系统的性能

3. 并行硬件

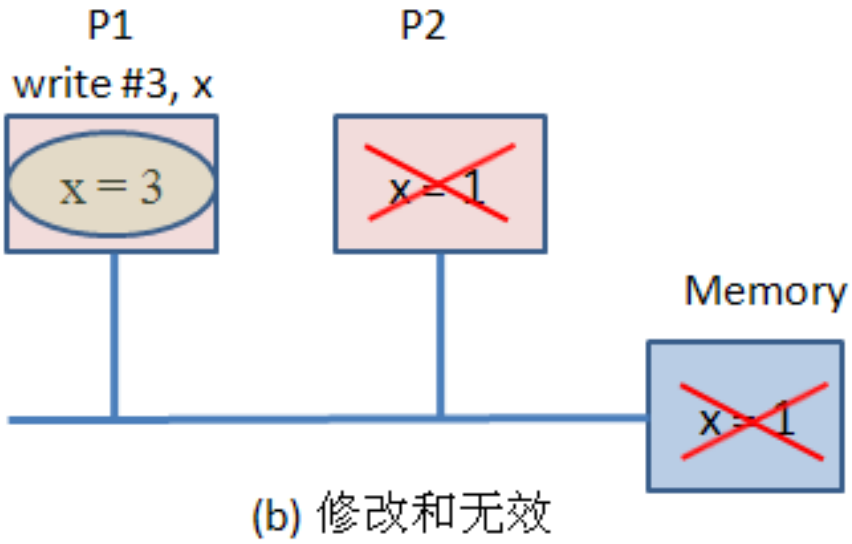
- 缓存一致性 (Cache coherence)
  - 基于目录的缓存一致性（Directory-based cache coherence）
    - 使用称为“目录”的数据结构来存储每条 cache line 的状态
    - 当一个变量被更新时，会查询目录，并且在其缓存中包含该变量的 cache line 将失效
    - 目录需要额外存储空间，但是更新缓存变量时，只需要与存储该变量的 core 联络

3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 伪共享（False sharing）
    - CPU 缓存通过硬件实现，对 cache line 操作，而不是单个变量



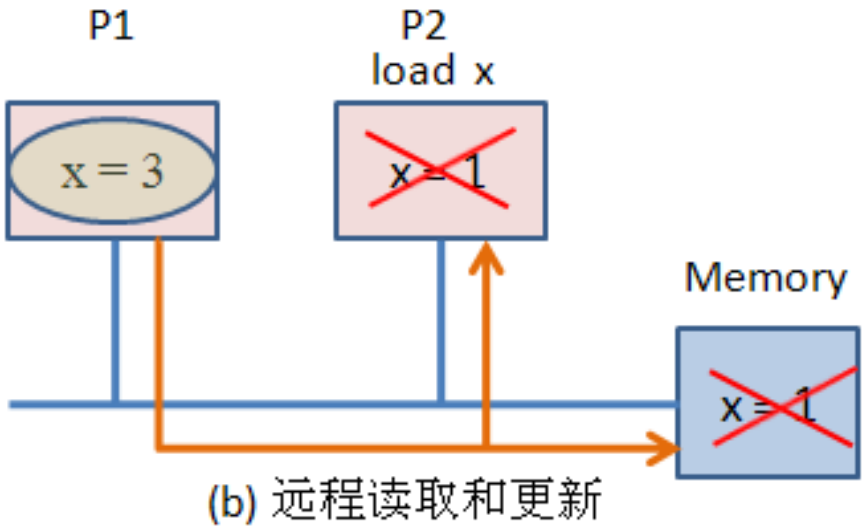
(a) 共享



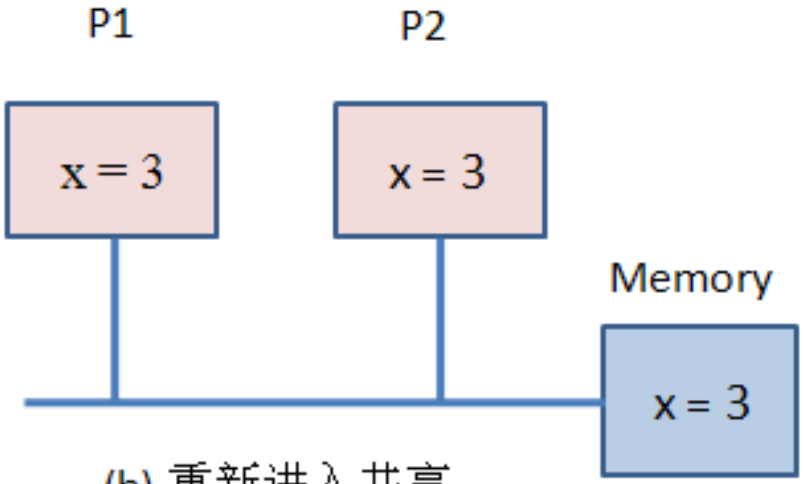
(b) 修改和无效

3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 伪共享（False sharing）
    - CPU 缓存通过硬件实现，对 cache line 操作，而不是单个变量



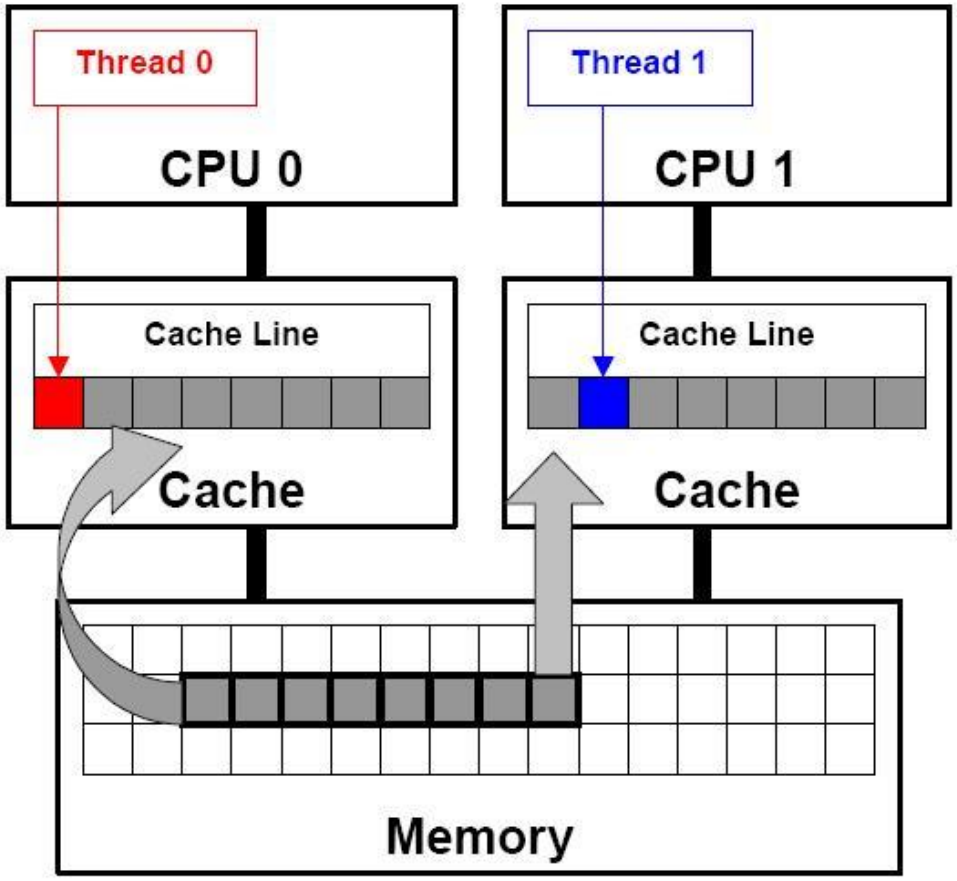
(b) 远程读取和更新



(b) 重新进入共享

3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 伪共享（False sharing）



3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 伪共享（False sharing）

```
int i, j, m, n;
double y[m];

/ Assign y = 0 /
. . .
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i,j);
```

3. 并行硬件

- core_count = 2, m = 8
- double 为 8 字节
- cache line 为 64 字节
- y[0] 存储在 cache line 的开始位置

```
/ Private variables /
int i, j, iter_count;

/ Shared variables initialized by one core /
int m, n, core_count
double y[m];

iter_count = m / core_count

/ Core 0 does this /
for (i = 0; i < iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);

/ Core 1 does this /
for (i = iter_count+1; i < 2iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);
. . .
```

3. 并行硬件

- 所有的 y 都在 cache line 上
- 每次 core 执行 y[i] += f(i,j)，cache line都会失效
- 下一次其他core执行该语句时，将从内存中读取更新过的 cache line

```
/ Private variables /
int i, j, iter_count;

/ Shared variables initialized by one core /
int m, n, core_count
double y[m];

iter_count = m / core_count

/ Core 0 does this /
for (i = 0; i < iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);

/ Core 1 does this /
for (i = iter_count+1; i < 2iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);
. . .
```

3. 并行硬件

- 虽然 core0 和 core1 从未访问各自的数据，但是却频繁访问内存

```
/ Private variables /
int i, j, iter_count;

/ Shared variables initialized by one core /
int m, n, core_count
double y[m];

iter_count = m / core_count

/ Core 0 does this /
for (i = 0; i < iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);

/ Core 1 does this /
for (i = iter_count+1; i < 2iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i,j);
. . .
```

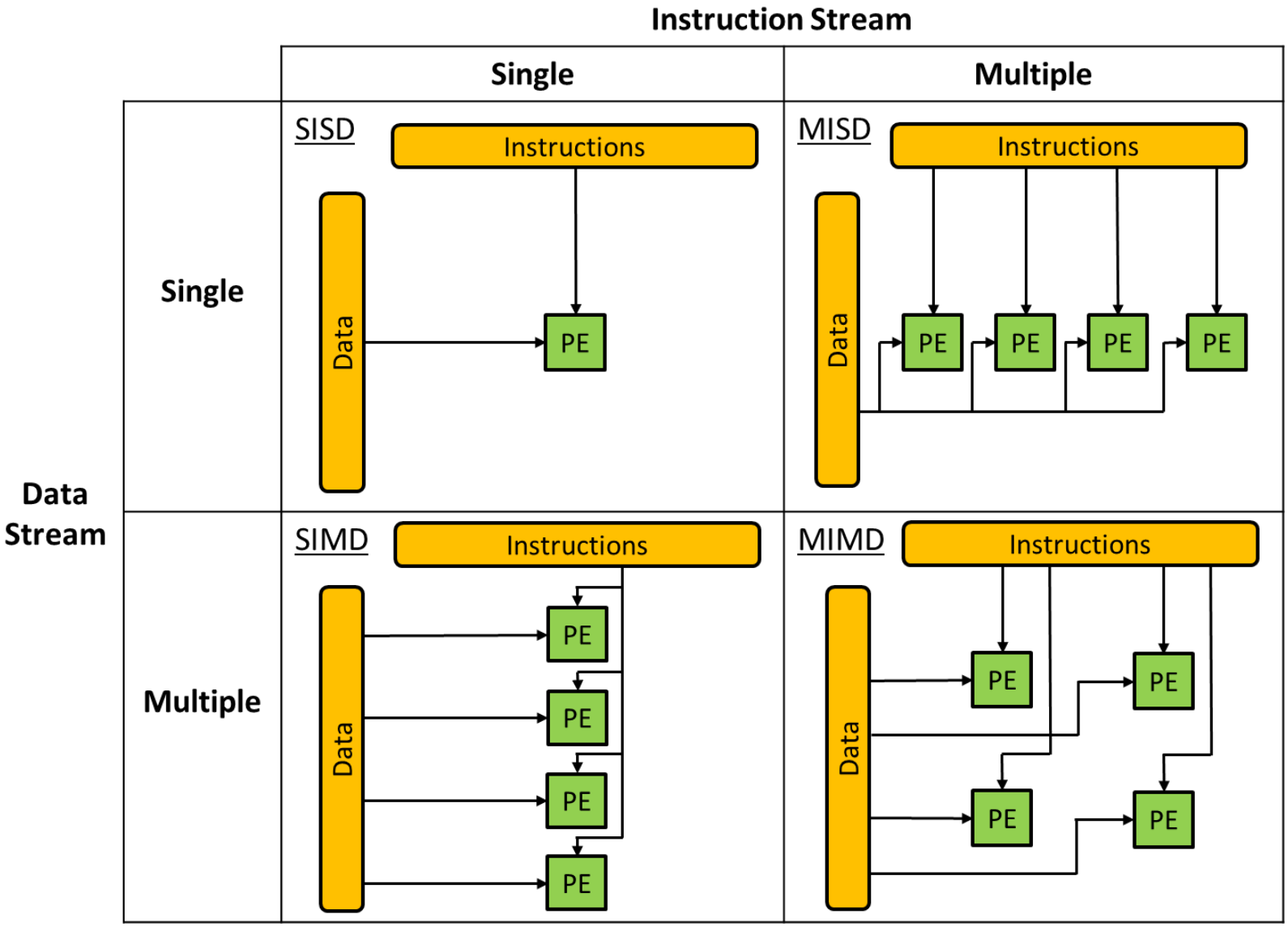
3. 并行硬件

- 缓存一致性 (Cache coherence)
  - 伪共享（False sharing）的解决方法
    - 每个线程使用局部线程数据
    - 每个线程访问的全局数据尽可能分隔开至少超过一个Cache Line

3. 并行硬件

- 共享内存 vs. 分布式内存
  - 当我们将处理器添加到总线时，在访问总线时发生冲突的可能性会急剧增加，因此总线适用于只有少数处理器的系统
  - 大型交叉开关（crossbar）非常昂贵，很少使用大型交叉开关互联系统
  - 分布式内存中超立方体和环形网络的互联方式相对便宜
  - 分布式内存系统通常更适合于大量数据或计算的问题

小结


Instruction Stream
Single
Multiple
Data Stream
Single
Multiple
SISD
Instructions
Data
PE
MISD
Instructions
Data
PE
PE
PE
PE
SIMD
Instructions
Data
PE
PE
PE
PE
MIMD
Instructions
Data
PE
PE
PE
PE