北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

# 并行编程原理与实践

## 4. 并行编程的关键问题

王一拙、计卫星

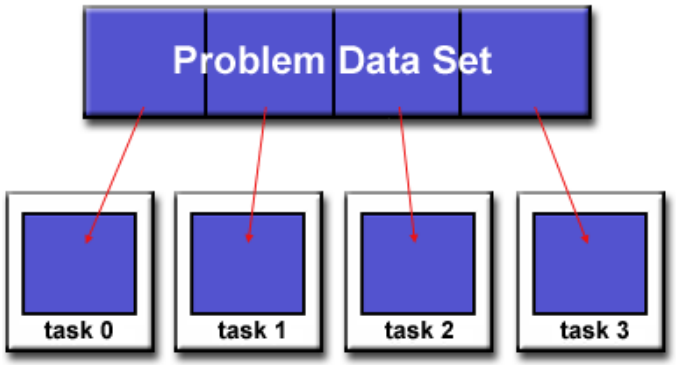
北京理工大学计算机学院

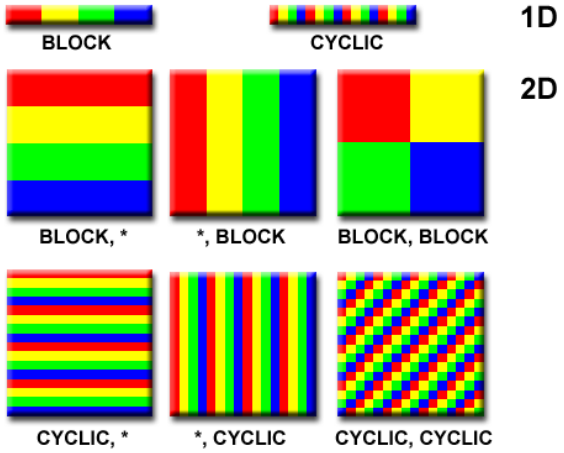
# 目录 | CONTENTS


# 1 任务划分（Partitioning）

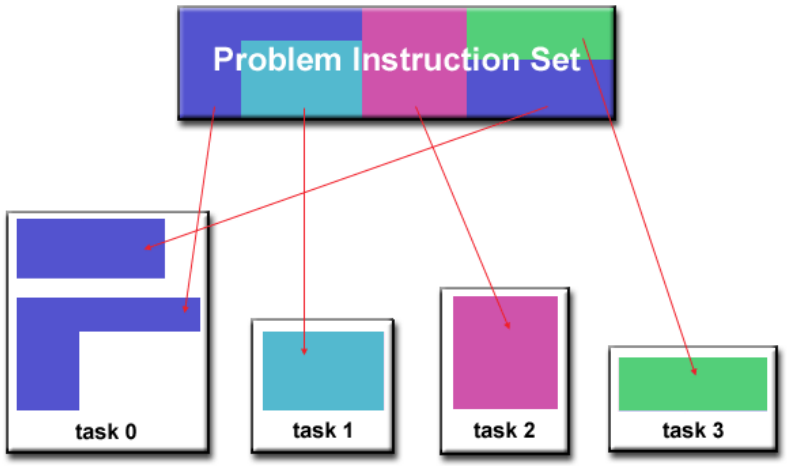
# 1 任务划分

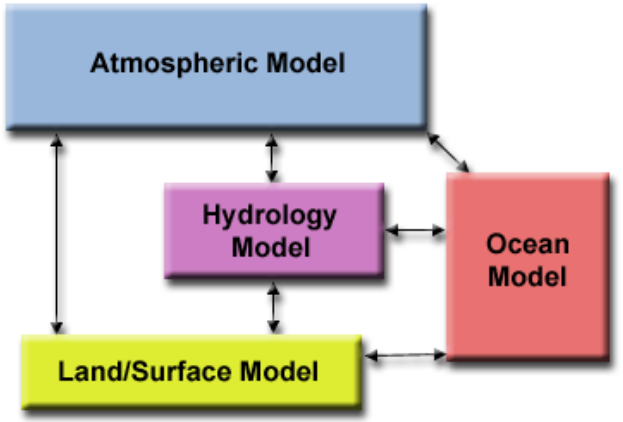
- 设计并行程序的第一步就是要将问题分解为离散的“块”，这些块可以分配给多个任务。
- 在并行任务之间划分计算工作有两种基本方法：域分解和功能分解。
- 域分解：与问题相关的数据被分解，每个并行任务都会处理部分数据。



- 功能分解：面向要执行的计算，而不是计算所操纵的数据，根据要完成的工作分解问题，每个任务执行全部工作的一部分。



## 任务划分就是要将问题分解为可以并行执行的任务

- 分解不一定是在程序运行前静态完成的，程序运行过程中可能产生新的任务
- 尽量将问题分集成足够多的任务，以保证所有运算单元都有任务可执行
- 有时需要结合域分解和功能分解
- 在并行度与任务创建和调度造成的开销间权衡
- 任务划分的关键在于确定任务间的依赖关系
  - 任务执行顺序逻辑上的依赖关系
  - 数据依赖：不同任务对同一存储单元的访问造成
    - RAW（写后读）、WAR（读后写）、WAW（写后写）

## 谁来进行任务划分？

- 大多数情况下由程序员进行
- 串行程序并行化过程中的自动任务划分是一项具有挑战性的研究问题
  - 编译器必须正确分析程序，识别其中的所有依赖
  - 有些数据依赖是在运行时才产生的，无法在编译时得知
  - 目前对循环依赖的研究较多，编译器能识别简单的循环或嵌套循环中的依赖关系

**DOALL LOOP**

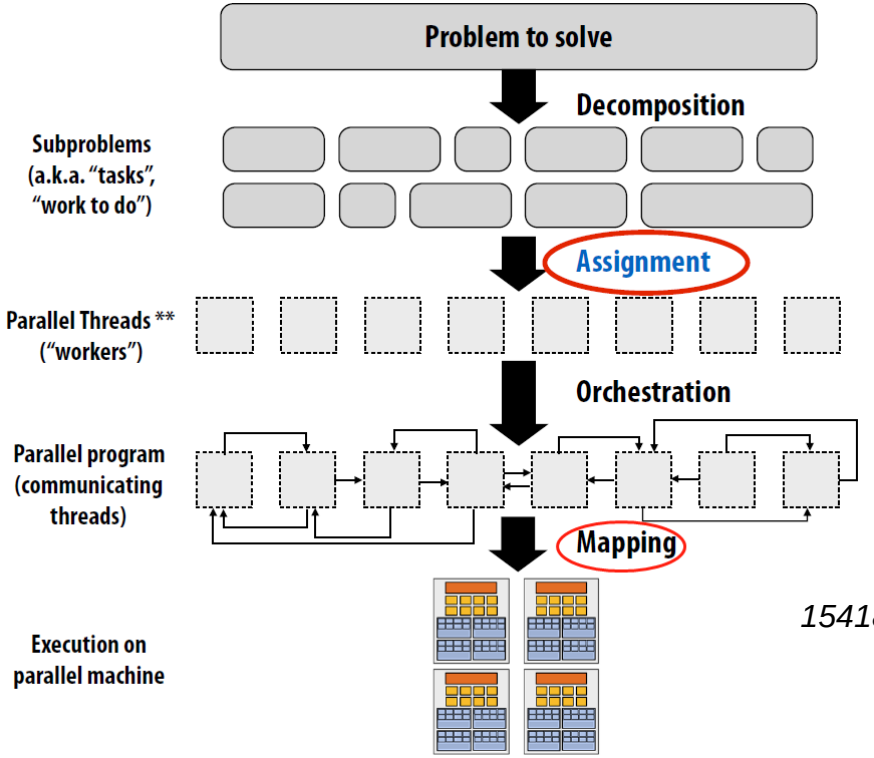
```
for (i = 0; i < n; i++){
  a[i] = b[i] + c[i];
}
```

**DOACROSS LOOP**

```
for (i = 0; i < n; i++){
   a[i] = a[i - 1] + b[i] + 1;
}
```


# 2 调度（Scheduling）

- 调度的终极目标：最小化程序执行时间，最大化资源利用率
- 调度考虑的关键问题：负载均衡、局部性
- 调度就是决定任务运行在哪个处理单元（PE）上，也就是Tasks–>PEs的映射，细分涉及：
  - 把任务分配给线程
  - 把线程分配给硬件处理单元



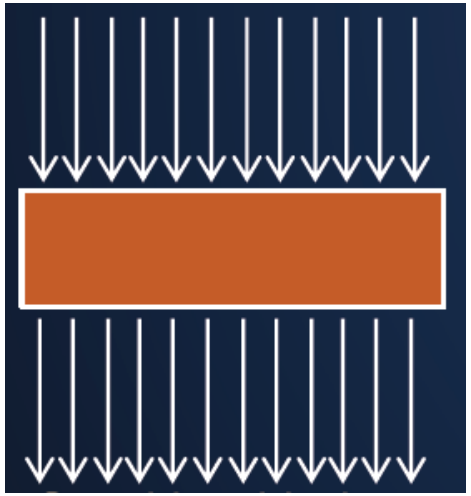

15418.courses.cs.cmu.edu

- **把任务分配给线程Tasks –> Threads (Workers)**
  - 可能在运行前静态分配，也可能在运行过程中动态分配
  - 程序员通常负责任务的划分，并行编程语言/框架的运行时系统通常负责任务的调度
  - 目标：负载均衡，减少通信，降低调度开销
- **处理线程间的同步和通信、数据的组织和分布**
  - 目标：减少同步和通信开销，保证数据访问的局部性，减少其它开销

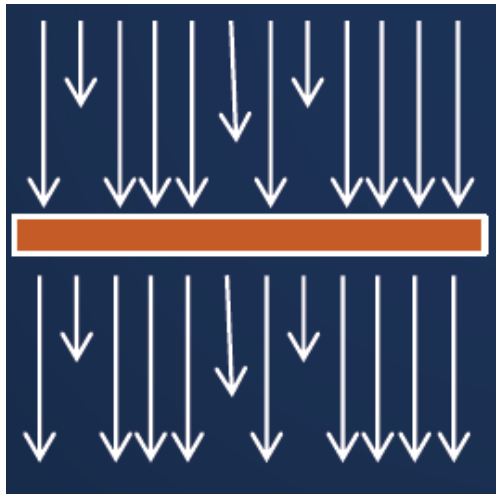
- **把线程分配（Mapping）给硬件处理单元Threads –> PEs**
  - 程序员进行线程分配 — 管理处理器亲和性(Affinity)
  - 操作系统进行线程分配
  - 编译器进行线程分配
  - 硬件进行线程分配 — CUDA线程块到GPU内核的映射
- **线程分配策略有时是相互冲突的**
  - 把相关的线程放在同一个处理器上（利用数据共享，局部性，减少通信和同步）
  - 把不相关的线程放在同一个处理器上（利于提高处理器利用率）
- **调度是一个NP完全问题（NP-Complete）**

## 负载均衡（Load balance）

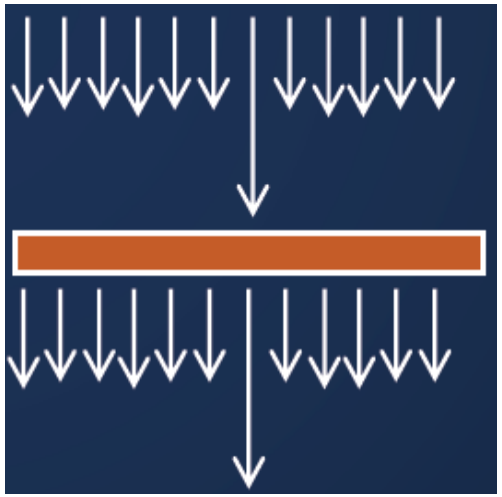
- 负载均衡是指在处理单元之间分配大致相等数量的工作，以使所有处理单元始终保持忙碌的做法。也就是让处理器空闲的时间最小化。



负载完全均衡
可能需要较大同步开销



负载不均衡
Load imbalance

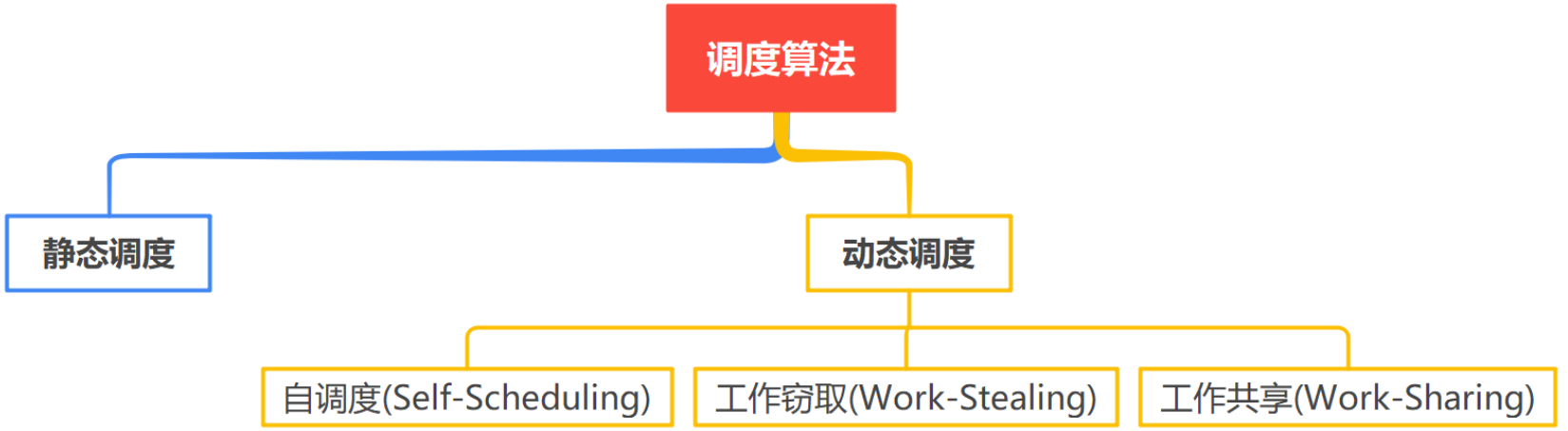


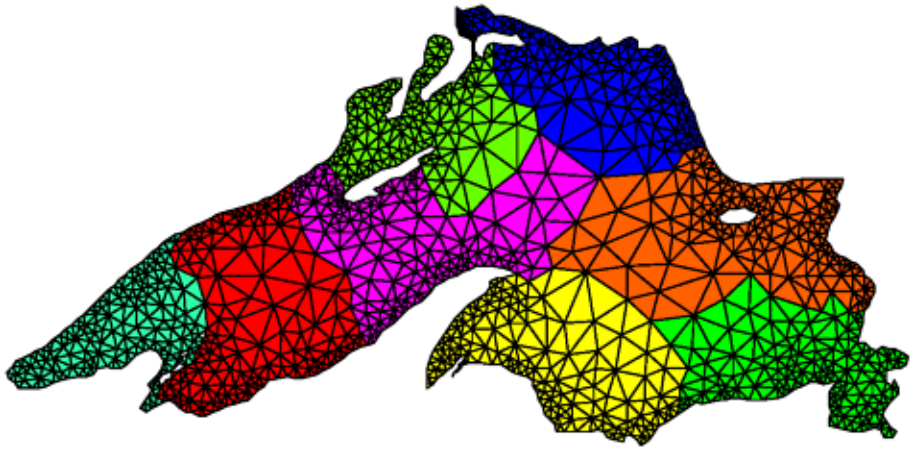
串行瓶颈造成负载不均衡
严重影响性能

## 负载不均衡的原因

- 任务本身不规则，或者任务量未知/变化
- 任务之间的同步、通信关系复杂，任务关系动态变化
- 机器环境复杂多变：其它程序的运行影响，存储层次造成的访存延迟影响等

## 调度算法（负载均衡方法）的分类



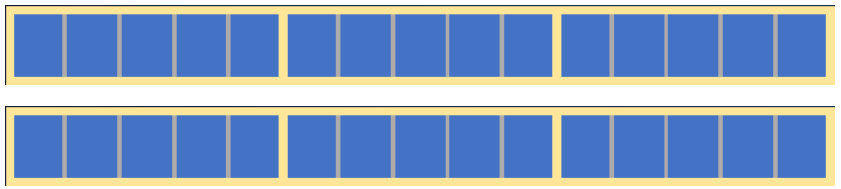


- **静态调度**
  - 为每个线程（Worker）分配相同数量的工作
  - 优点：简单，运行时没有调度开销
  - 适用场景：任务量和每个任务的执行时间是可预测的（已知的）
- “半静态”调度
  - 任务量虽未知，但近期可预测 — 可利用之前的程序运行情况来预测，如循环执行的任务
  - 应用程序定期进行Profiling，从而调整任务的分配

## 静态调度

- – E.g., loop of 30 iterations on 2 processors

- – # pragma omp for schedule(static, 5)

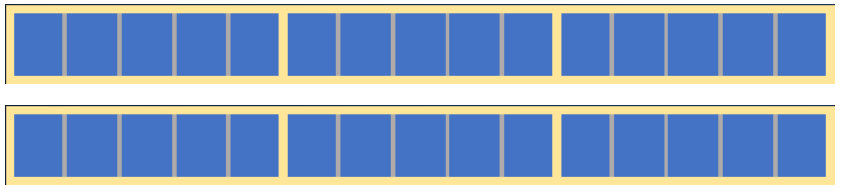


## 静态调度

- – E.g., loop of 30 iterations on 2 processors

- What if cost per iteration is not equal

- – # pragma omp for schedule(static, 5)

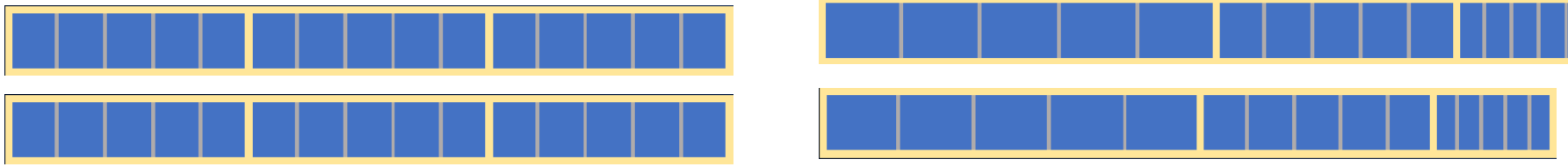


## 静态调度

- – E.g., loop of 30 iterations on 2 processors

- What if cost per iteration is not equal

- – # pragma omp for schedule(static, 5)



## 静态调度

- Divide work into chunks to reduce scheduling overhead

– E.g., loop of 30 iterations on 2 processors

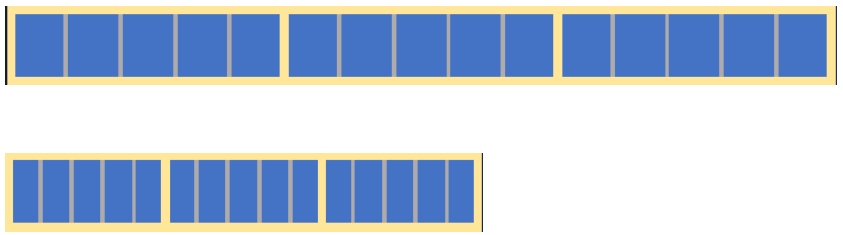
– # pragma omp for schedule(static, 5)

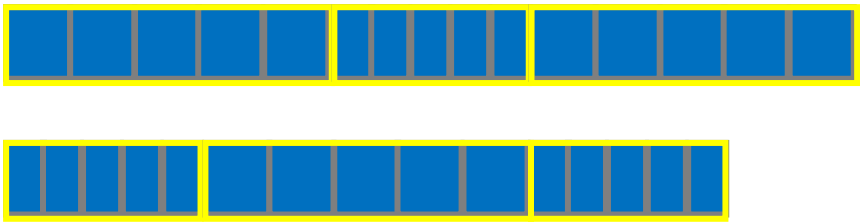
## 静态调度

- Divide work into chunks to reduce scheduling overhead

– E.g., loop of 30 iterations on 2 processors

– # pragma omp for schedule(static, 5)

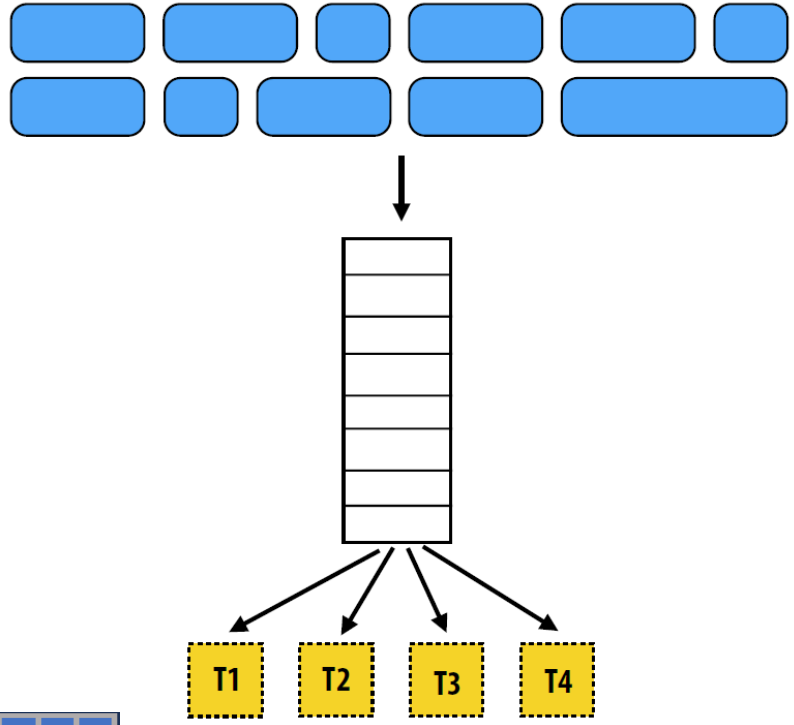


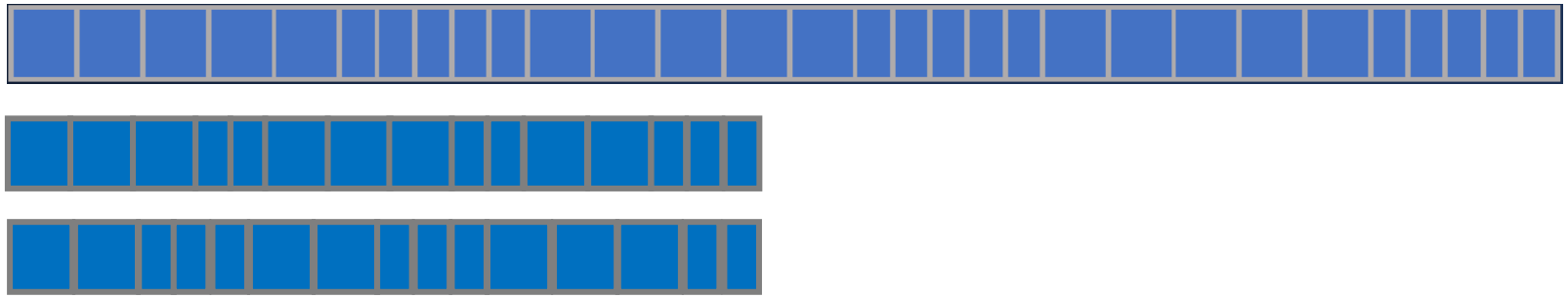


动态调度结果

## 动态调度

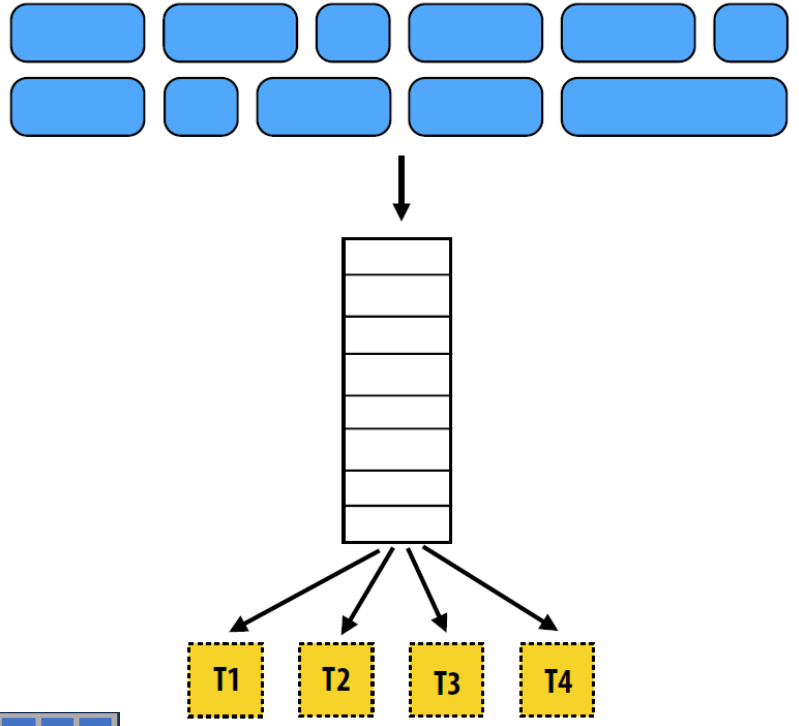
- 运行时动态分配任务以达到较好的负载均衡
- 常用于任务数量或任务执行时间未知时
- Self Scheduling [Tang and Yew ‘86]
  - 共享任务队列
  - 处理器 (workers) 每次从队列取出一个任务执行
  - 有新任务生成时加入共享队列

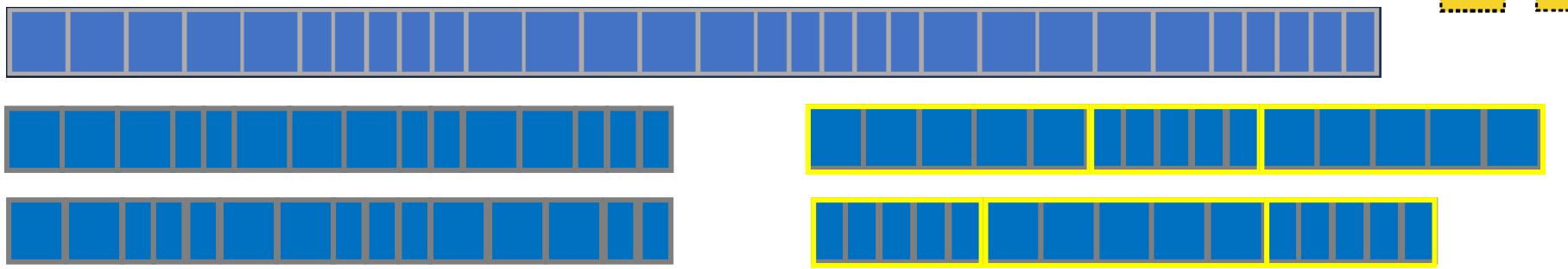


## 动态调度

- 运行时动态分配任务以达到较好的负载均衡
- 常用于任务数量或任务执行时间未知时
- Self Scheduling [Tang and Yew ‘86]
  - 共享任务队列
  - 处理器 (workers) 每次从队列取出一个任务执行
  - 有新任务生成时加入共享队列



## 动态调度

- 基于块的自调度算法[Kruskal and Weiss ‘85]
  - 为每个空闲处理器分配固定大小的“块”（即一批相邻的循环迭代）
  - 减少共享任务队列的访问冲突 (locking)
- 开始分配较大的块（降低调度开销），接下来逐渐分配较小的块来平衡负载。
- Guided Self Scheduling [Kuck and Polychronopolous ‘87]
  - 第i次分配的任务数量（块大小）

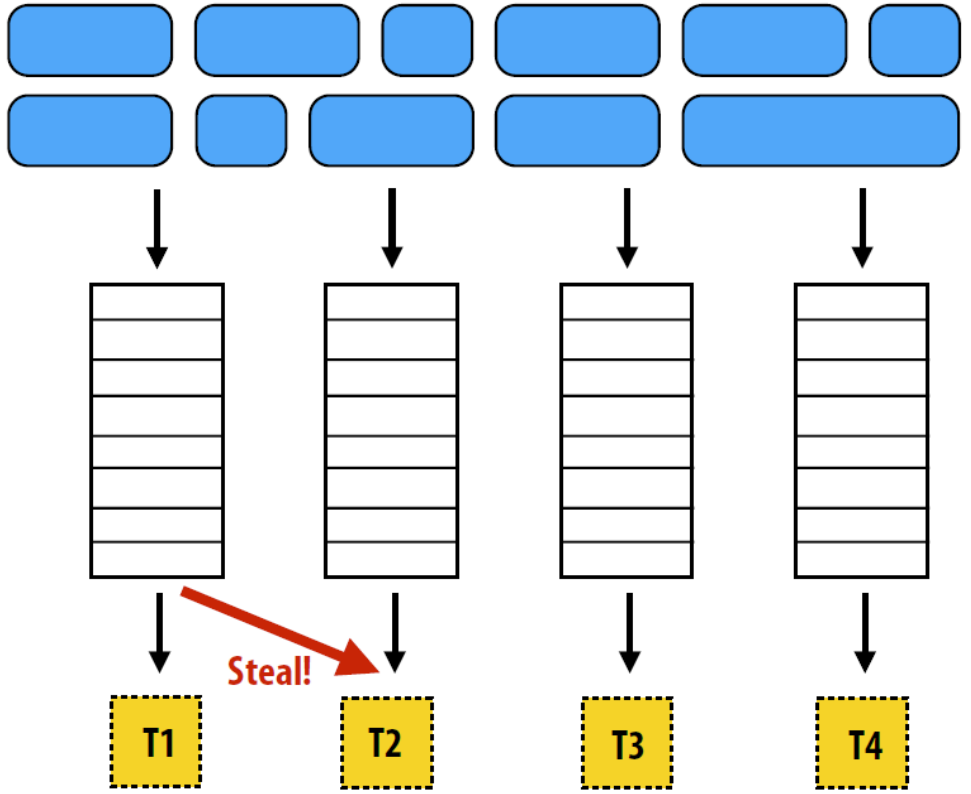
    Ki = ceiling(Ri/p)
    - Ri剩余的任务数量
    - p是处理单元个数

TABLE I. SAMPLE CHUNK SIZES FOR $N = 1000$ AND $P = 4$

| Scheme | Chunk sizes |
|---|---|
| GSS | 250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1 |
| FSS | 125 125 125 125 62 62 62 62 32 32 32 32 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1 |
| TSS | 125 117 109 101 93 85 77 69 61 53 45 37 29 21 13 5 |

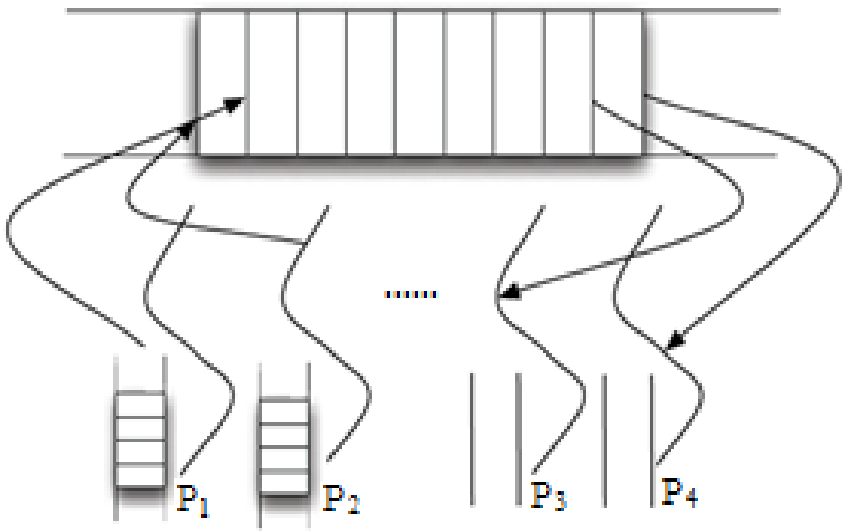
## 动态调度

- 工作窃取（Work-stealing）
  - 分布式任务队列
  - 空闲时窃取任务
- 关键问题：
  - 从哪里窃取？
  - 每次窃取多少？
  - 如何判断程序结束？

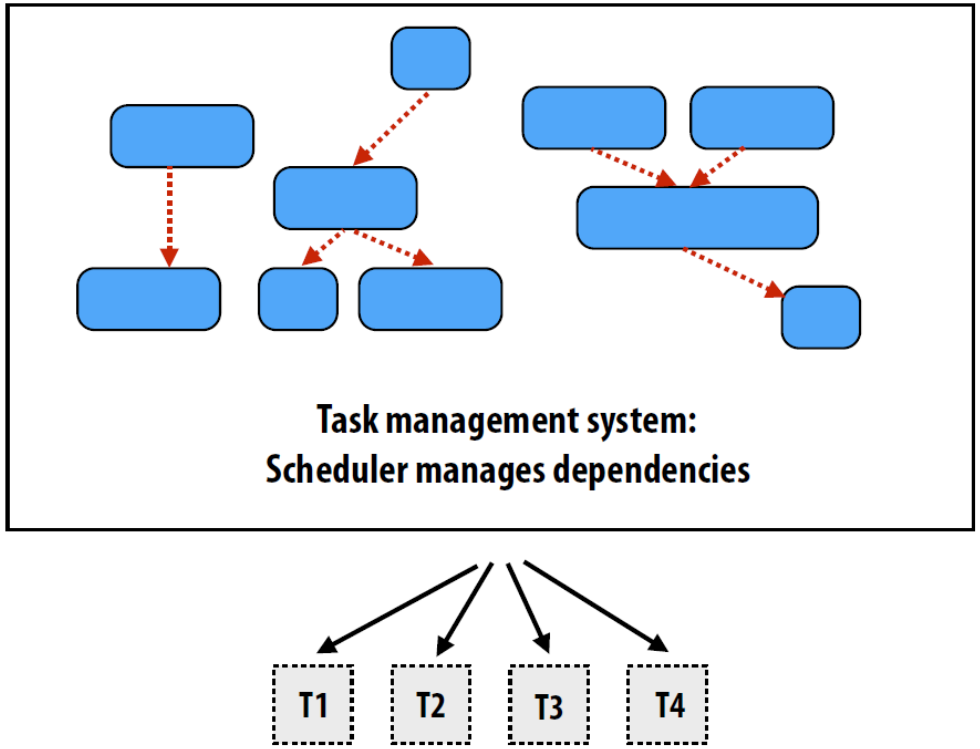


## 动态调度

- 工作共享（Work-sharing）
  - 集中式任务队列
  - 繁忙时推送任务
- 调度时机：
  - 通常在新任务生成时进行调度
  - 设置阈值控制本地任务量



- **任务之间存在依赖关系时：**
  - 只有当任务所依赖的其它任务都执行完后，调度器才会将该任务分配给工作线程
  - 工作线程提交新任务到系统中时，要明确任务的依赖关系




*15418.courses.cs.cmu.edu*

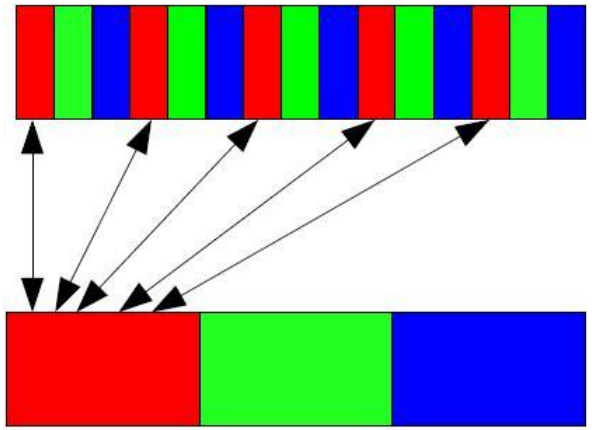

# 3 数据分布

- **数据存放在哪里？**
  - 数据的初始划分 — 数据和计算任务的关系
  - 数据在整个系统中的分布 — 数据和存储系统（层次）的关系
    - 集中存储 / 分散存储
  - 运行时的数据传输（通信）— 重叠；复制
- **数据以什么样的格式（Layout）存储最合适？**
  - 数据重组（Reorganization）

## 数据重组

- 为什么要进行数据的重新组织？
  - 提高后继程序的局部性
  - 向量化的需要
- 聚合（Gather）
- 散发（Scatter）
- 打包（Pack）
- 几何分解和分区（Geometry Decomposition, Partitioning）
- 结构的数组（AoS）vs. 数组的结构（SoA）

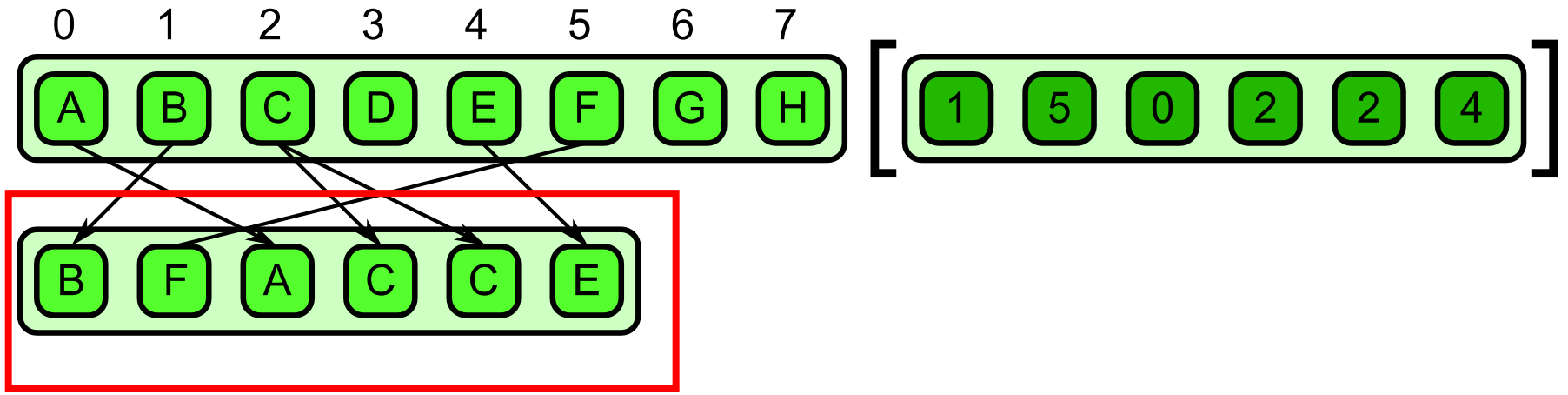
```
struct Pixel {
    float r, g, b;
};
Pixel image_AoS[480][640];
```



```
struct Image {
    float R[480][640];
    float G[480][640];
    float B[480][640];
};
Image image_SoA;
```

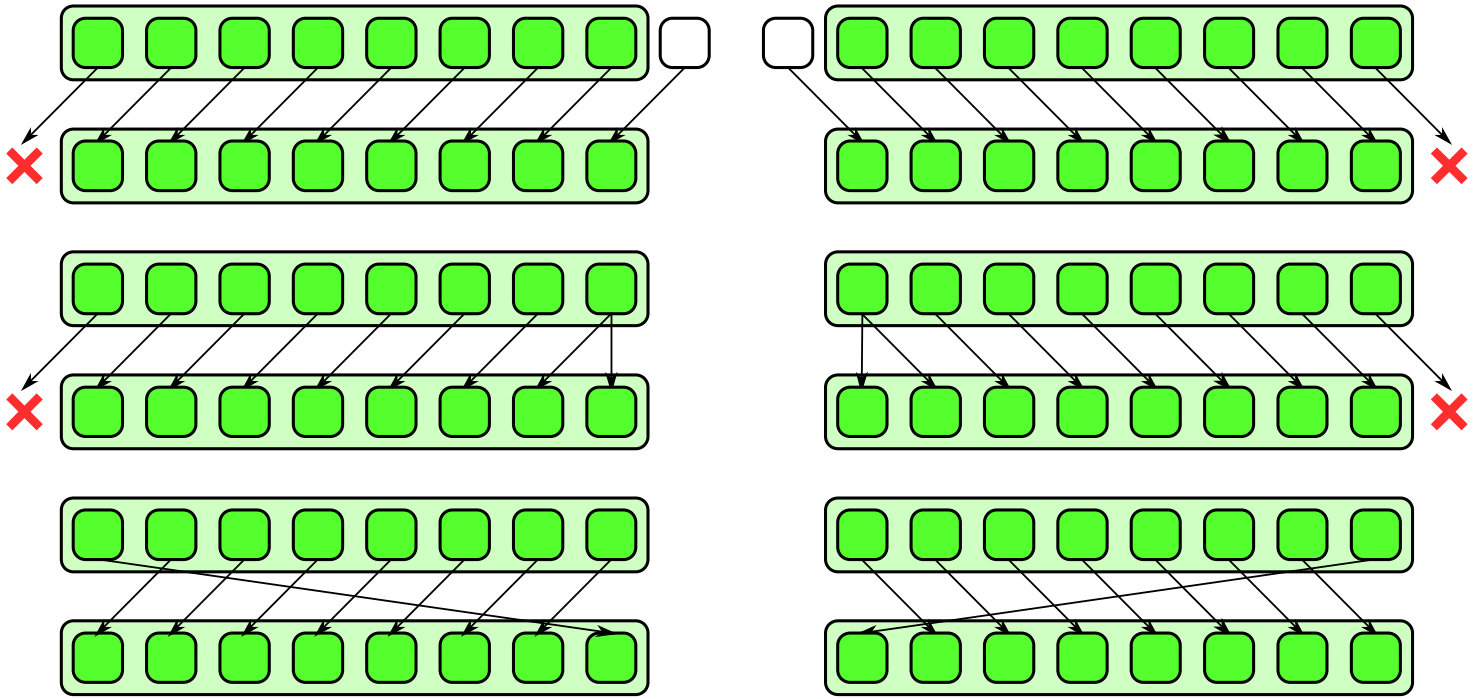
## 聚合（Gather）

- 给定一个源数组和一组位置集合（地址和数组索引），收集源数组给定位置的数据并存入一个输出集合中
- 输出数据类型和输入数据一样，输出集合大小和索引集合一样

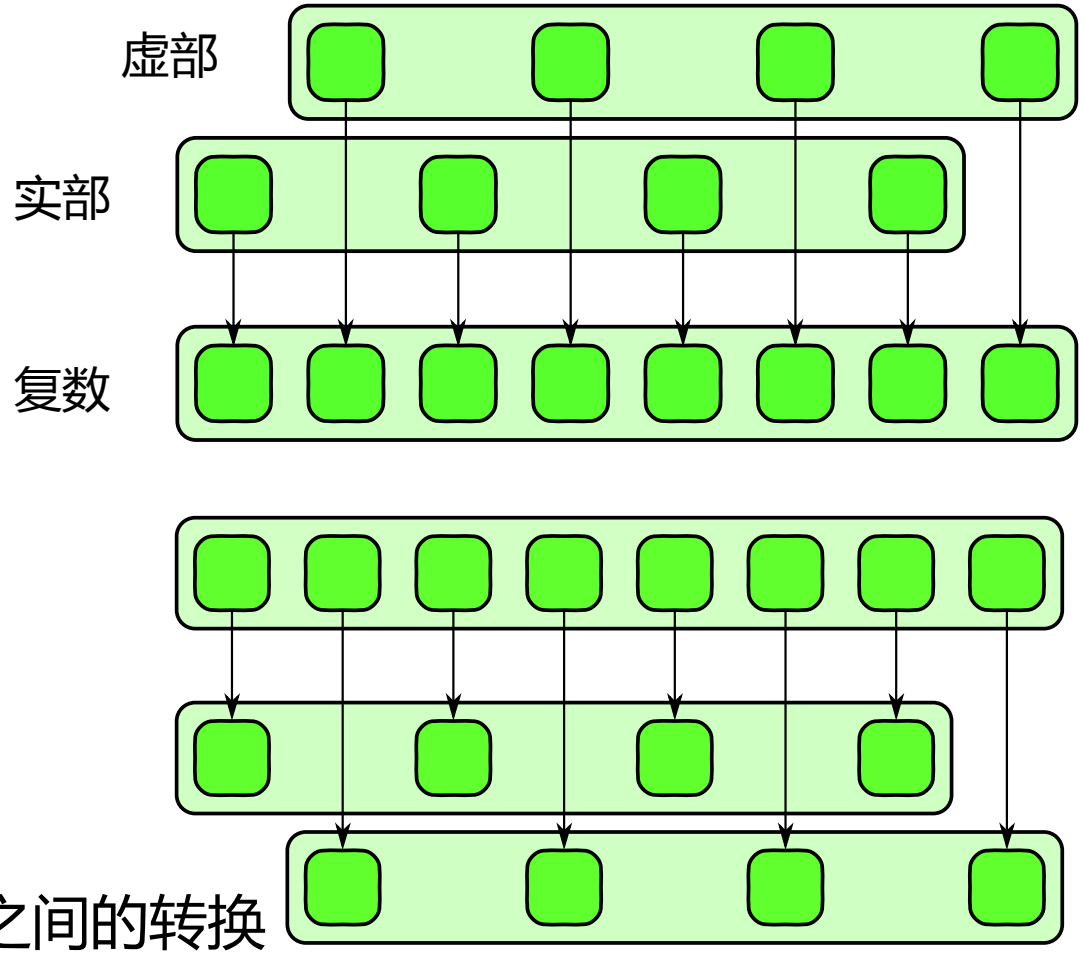


## 聚合（Gather）的特殊情况

- 移位（Shifts）
  - 左移、右移、循环移位
  - 数据访问偏移固定距离
  - 注意边界处理
  - 可用向量指令实现
  - 良好的局部性

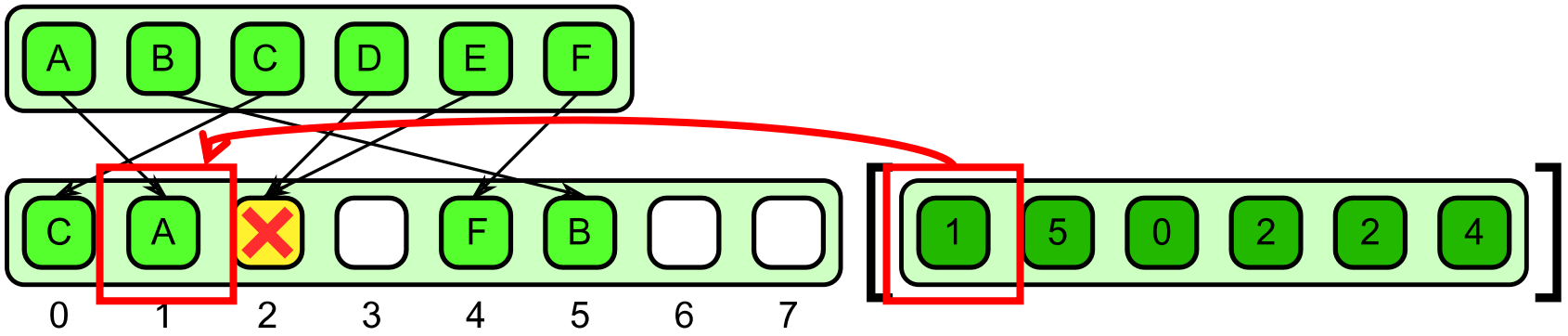


- **聚合（Gather）的特殊情况**
  - 拉合（Zip）
    - 对数据进行交错排列
    - 可推广到多组数据的拉合
    - 可拉合不同类型的数据
  - 拉拆（Unzip）
    - 拉合的逆操作
    - 提取特定步幅和偏移量的子数组
  - 拉合与拉拆可用于AoS和SoA数据布局之间的转换



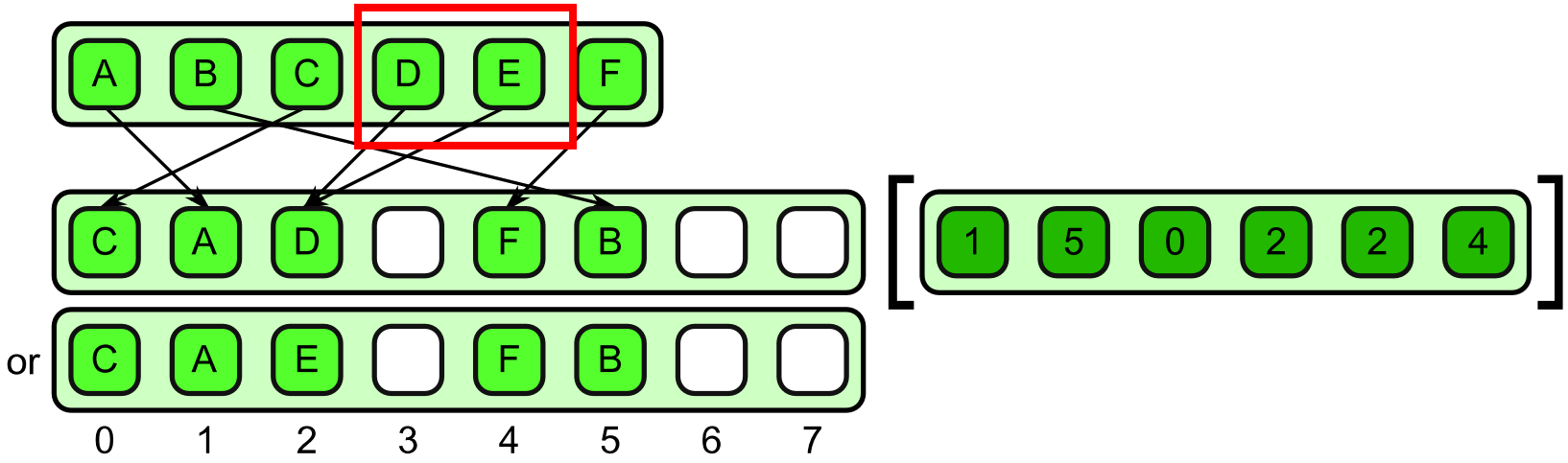
## 散发（Scatter）

- 和聚合类型，不同之处在于给定的是写入位置（索引）而不是读取位置，然后将输入数据并行的写入指定位置
- 有可能写入同一个位置，引发冲突（collisions）



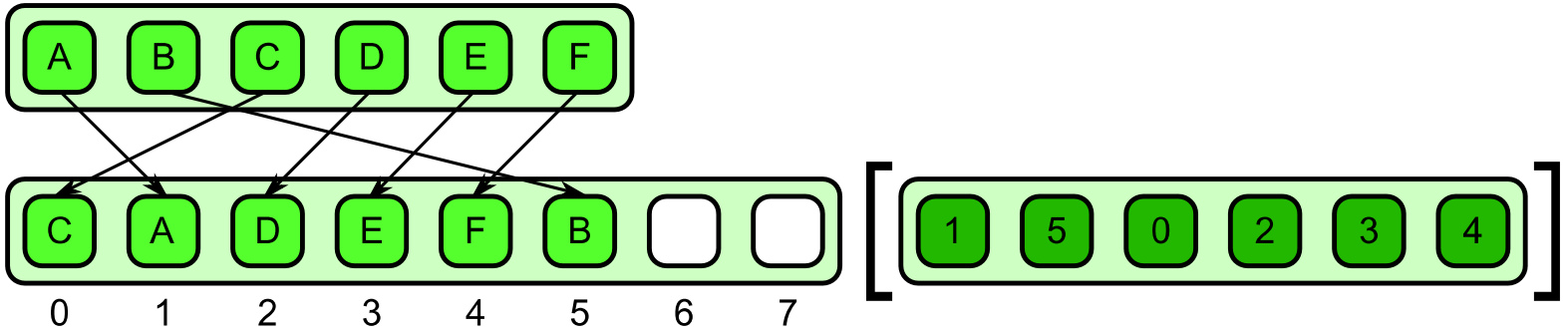
## 散发（Scatter）

- 按解决冲突的方式分为：
  - 原子散发（Atomic Scatter）



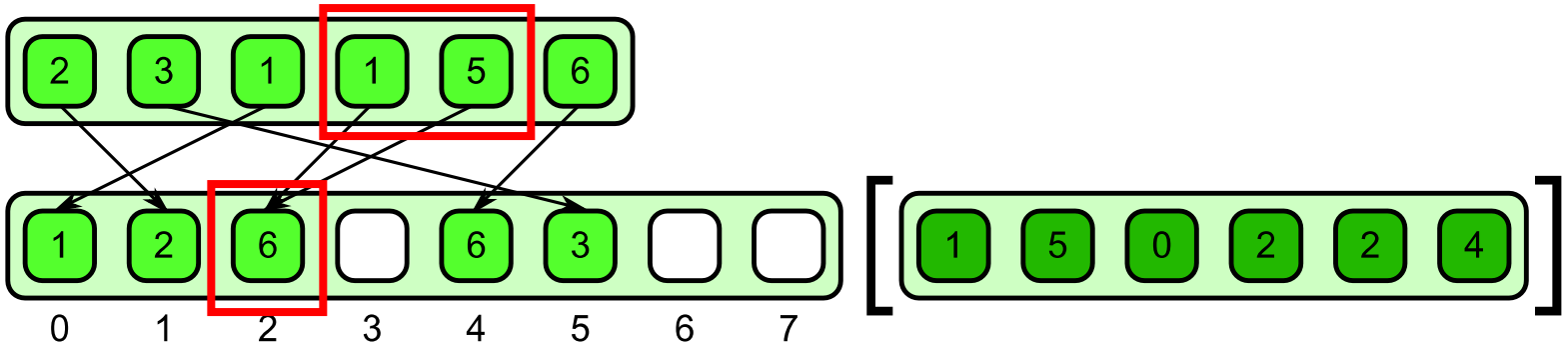
## 散发（Scatter）

- 按解决冲突的方式分为：
  - 排列散发（Permutation Scatter）



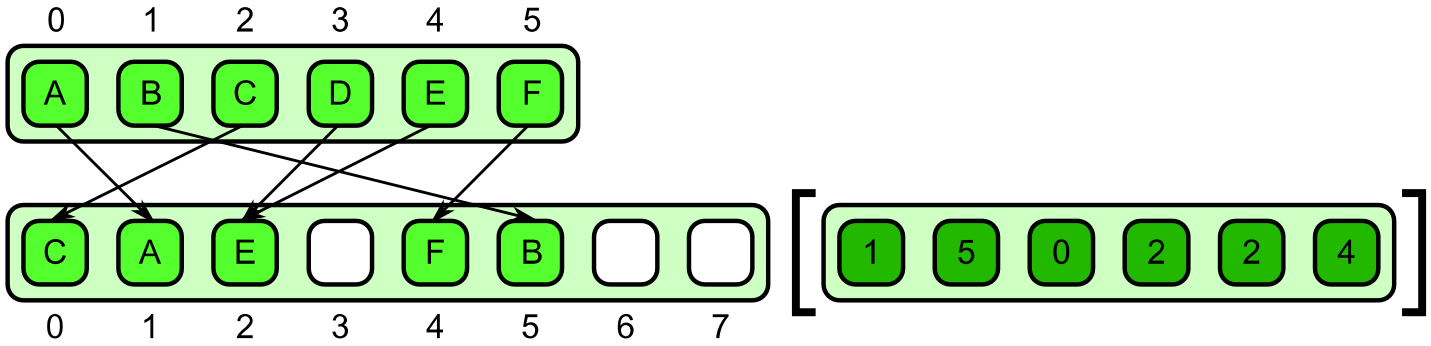
## 散发（Scatter）

- 按解决冲突的方式分为：
  - 归并散发（Merge Scatter）



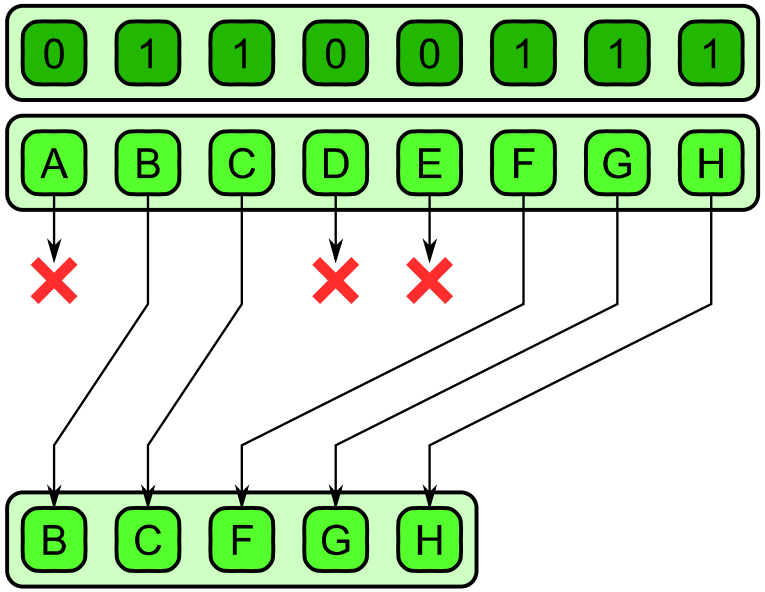
## 散发（Scatter）

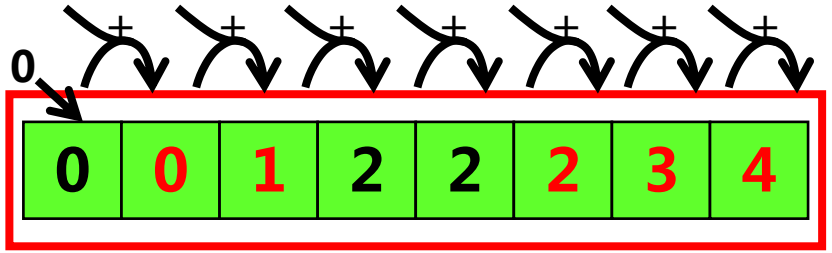
- 按解决冲突的方式分为：
  - 优先散发（Priority Scatter）

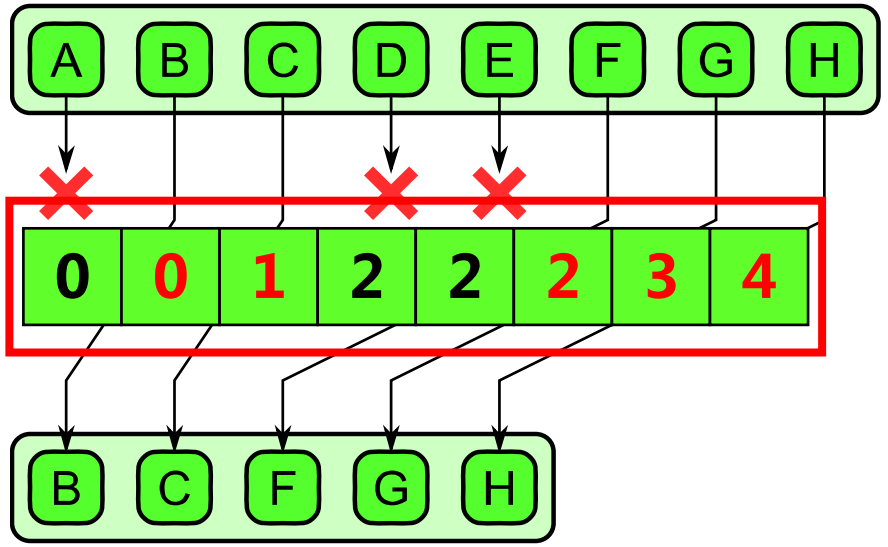


## 打包（Pack）

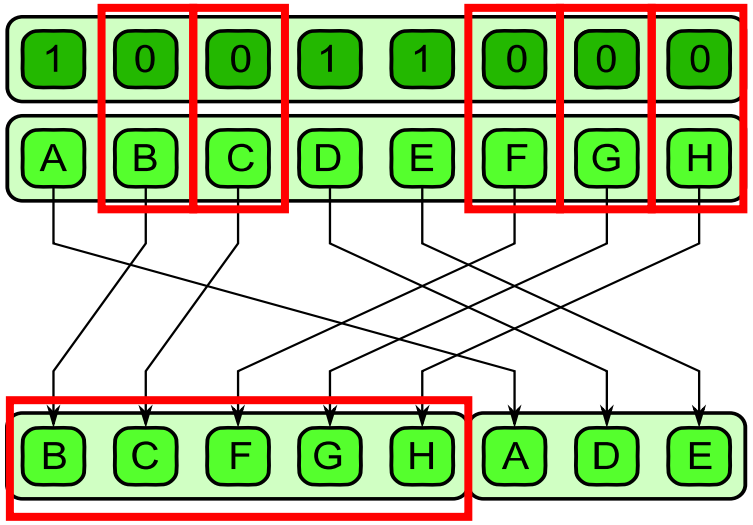
- 用于消除集合中不被使用的元素
- 重组剩余的元素使其在内存中保持连续



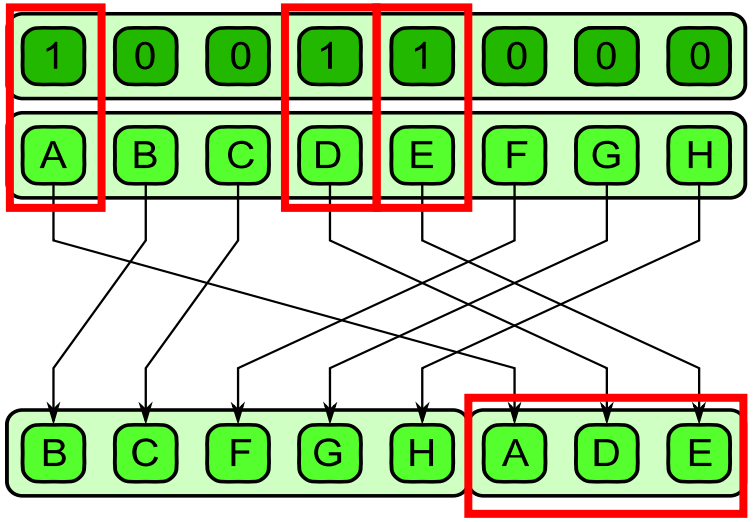



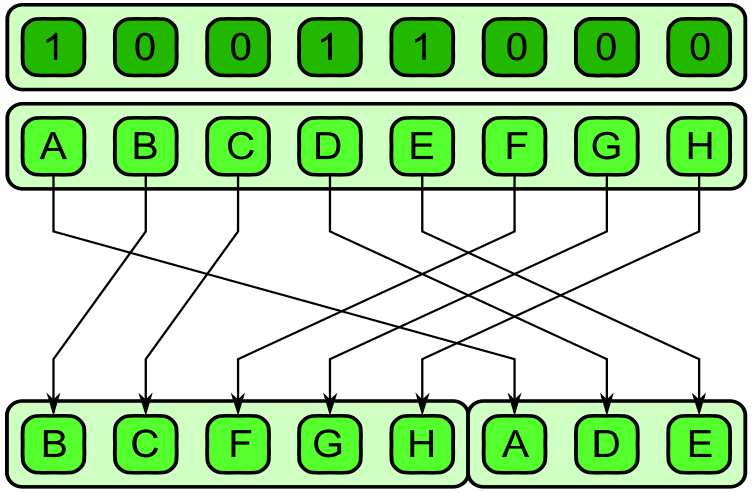
- **打包（Pack）的推广**
  - 拆分（Split）
  - 整合（Unsplit）
  - 拆分和整合常用向量指令实现



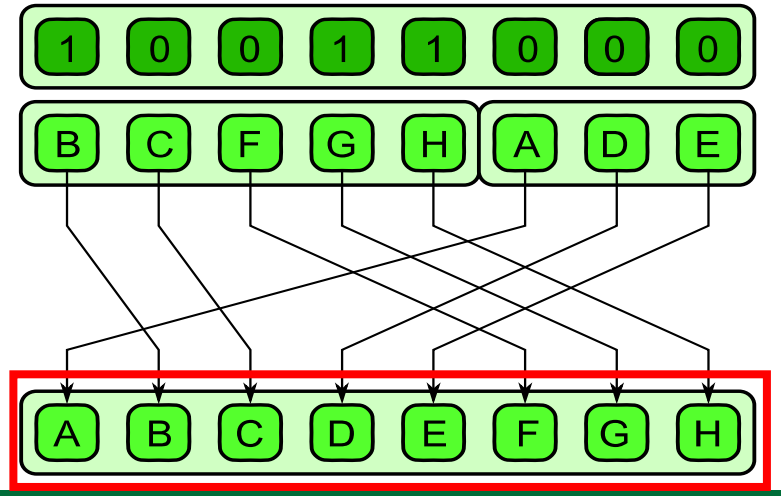
- **打包（Pack）的推广**
  - 拆分（Split）
  - 整合（Unsplit）
  - 拆分和整合常用向量指令实现



- **打包（Pack）的推广**
  - 拆分（Split）



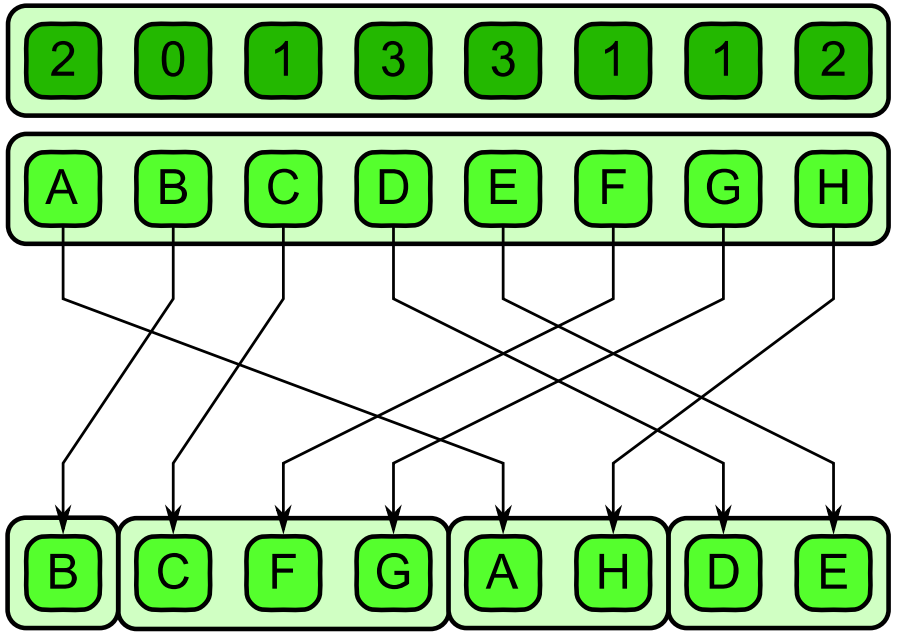

  - 整合（Unsplit）




  - 拆分和整合常用向量指令实现

## 打包（Pack）的推广

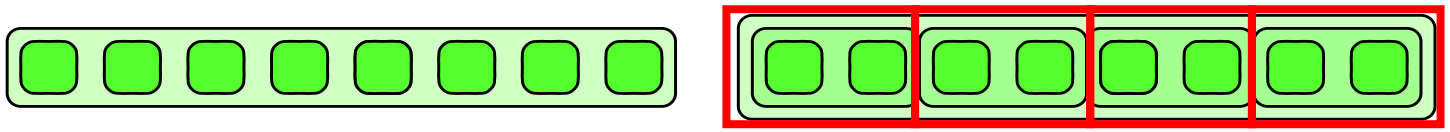
- 分箱（Bin）

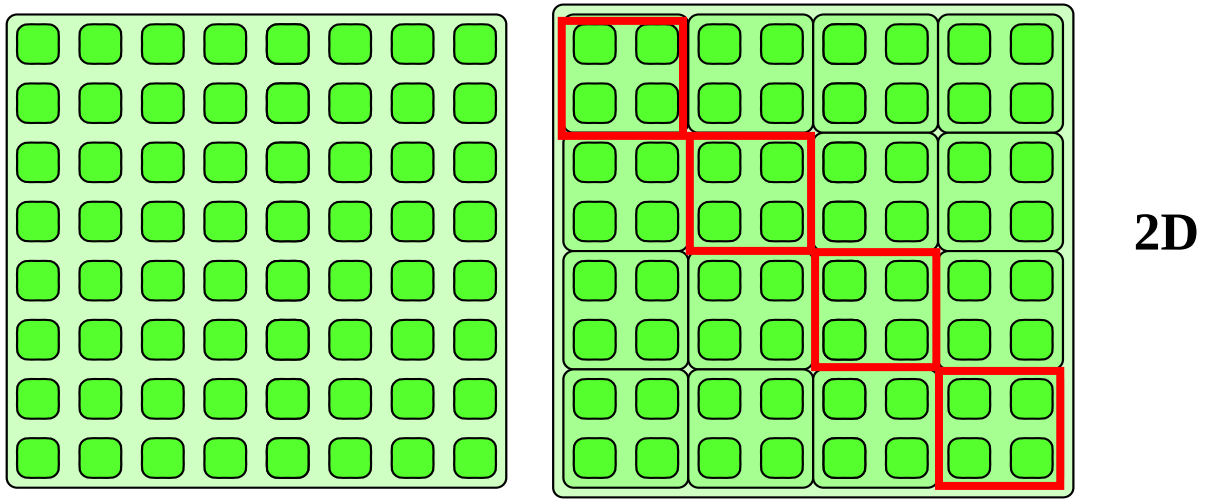



**数据分成四个分类**

## 几何分解和分区（Geometry Decomposition, Partitioning）

- 并行算法经常将数据划分成多个部分，分别处理，然后合并，也就是分治
- 分治算法对数据的划分称为几何分解，如果数据划分后不重叠，称为分区



## 结构的数组（AoS）vs. 数组的结构（SoA）

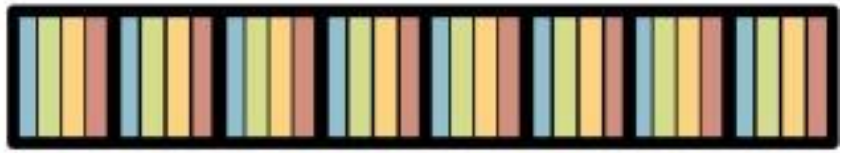
- AoS：如果随机访问数据，可能会提高缓存利用率
- SoA：利于向量化

Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```

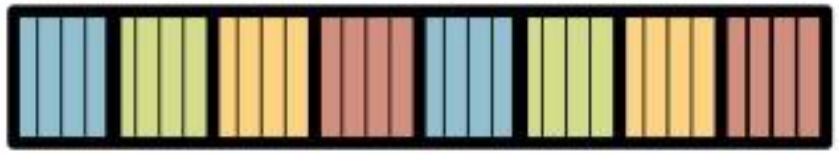
Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```



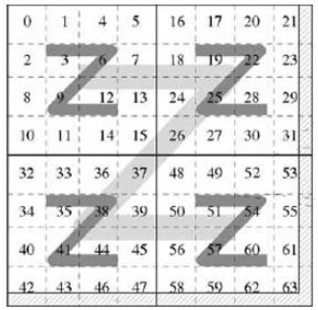
Array of Structure of Tiled Array (ASTA)

```
struct foo{
 float a[4];
 float b[4];
 float c[4];
 int d[4];
} A[2];
```

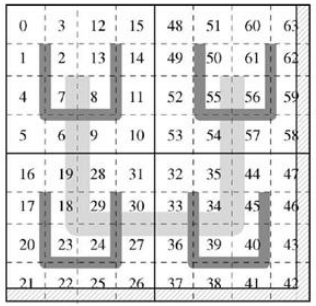


## 矩阵数据的不同布局方式示例

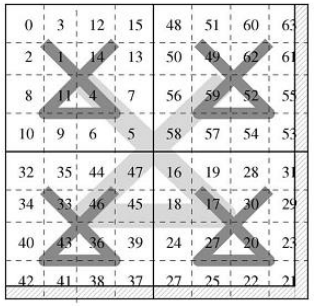
- 通过分片（Tiling）和分块（Blocking）矩阵数据来适应L1、L2 cache大小



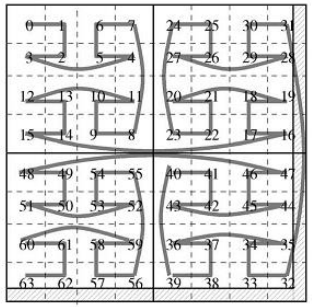
(a) Z-Morton layout



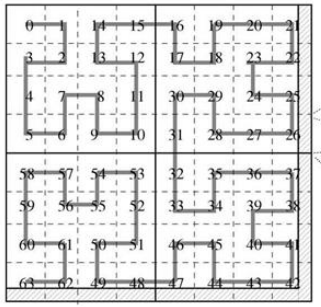
(b) U-Morton layout



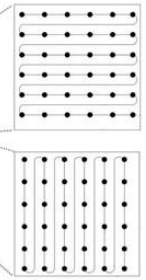
(c) X-Morton layout



(d) G-Morton layout



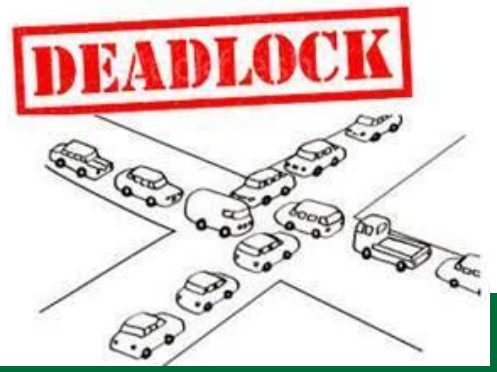
(e) Hilbert layout

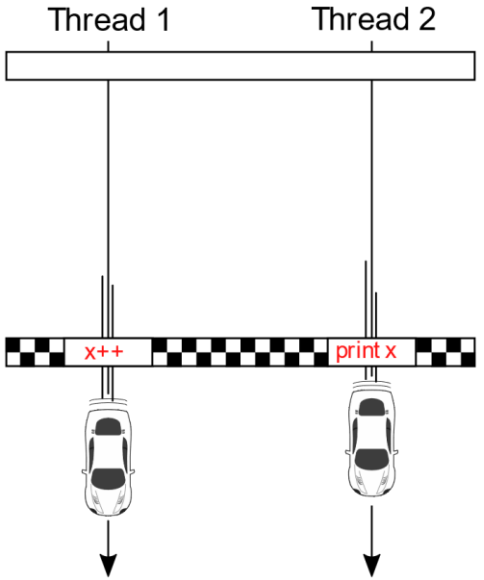


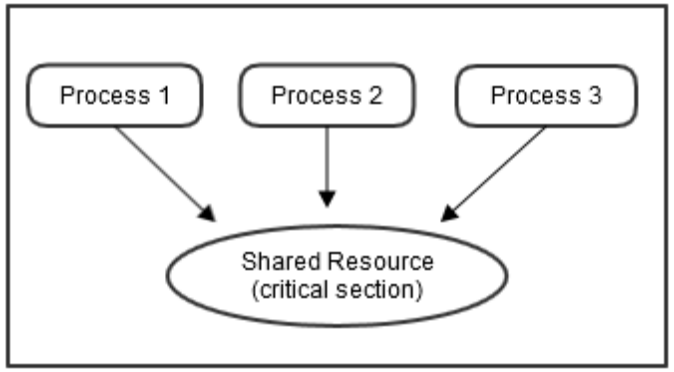
(f) Row-major / Column-major


# 4 同步（Synchronization）

- **为什么要进行同步操作（Synchronization）？**
  - 对大多数并行程序，管理工作（任务）执行顺序是关键的设计因素。
  - 在基于共享资源的并行软件构造中，对共享资源访问的同步控制是其中的关键之一。
- **缺少同步或同步不当会造成什么？**
  - 程序执行顺序逻辑不正确
  - 数据竞争
  - 死锁



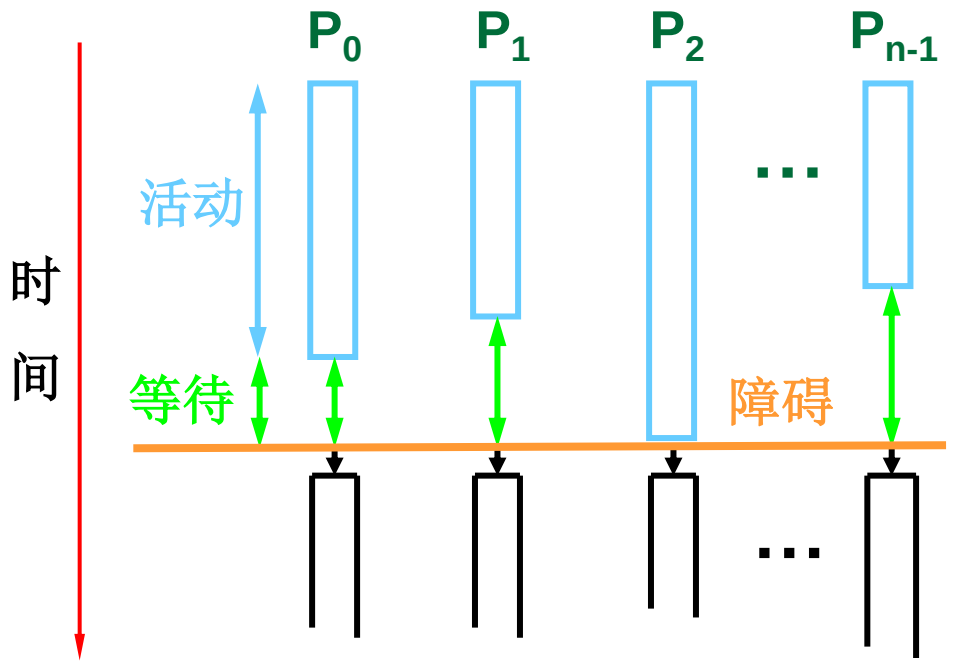



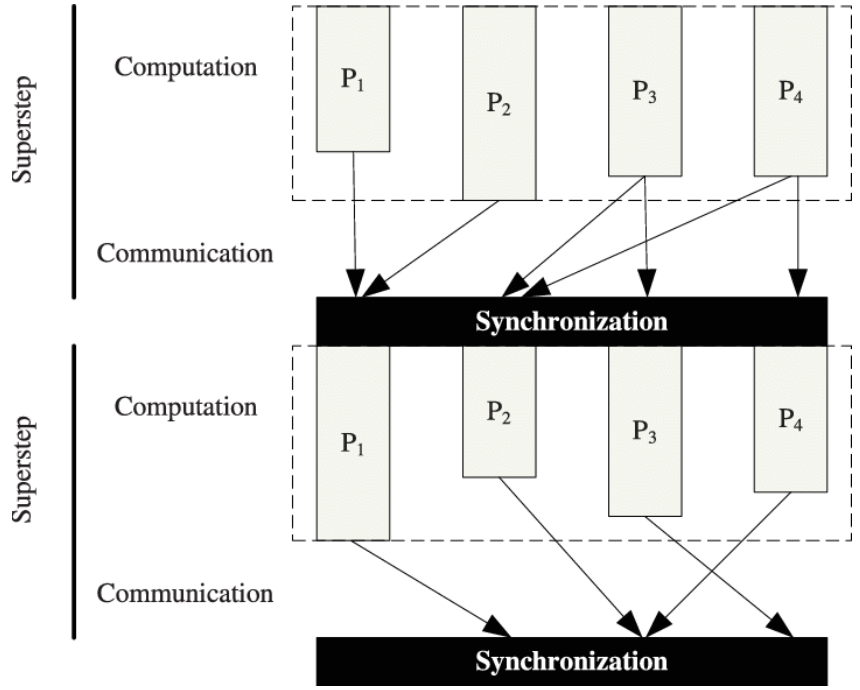
## 同步操作的种类

- 栅栏/障碍/路障（Barrier）
- 锁（Lock）
- 信号量（Semaphore）
- 同步通信操作

## 栅栏/障碍/路障（Barrier）

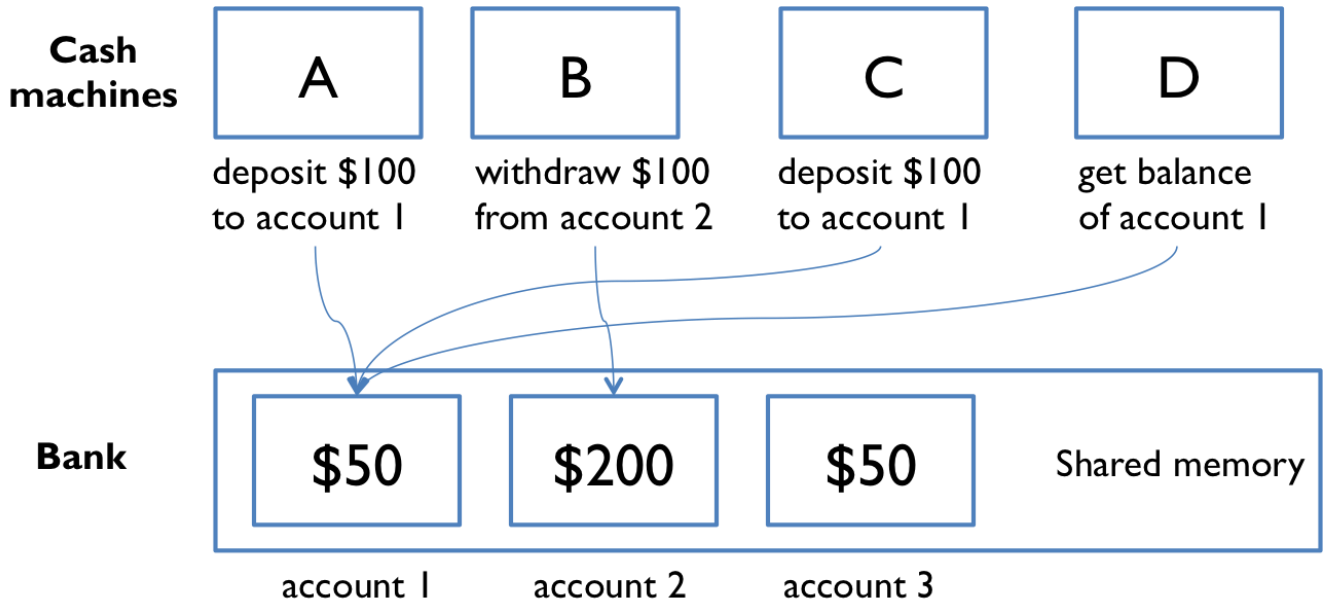
- 在参与障碍同步的每个进程中彼此必须等待的位置设置一个障碍点，当某进程执行到障碍点时暂停，等待所有进程都执行到这个障碍点上，它们才能继续运行。



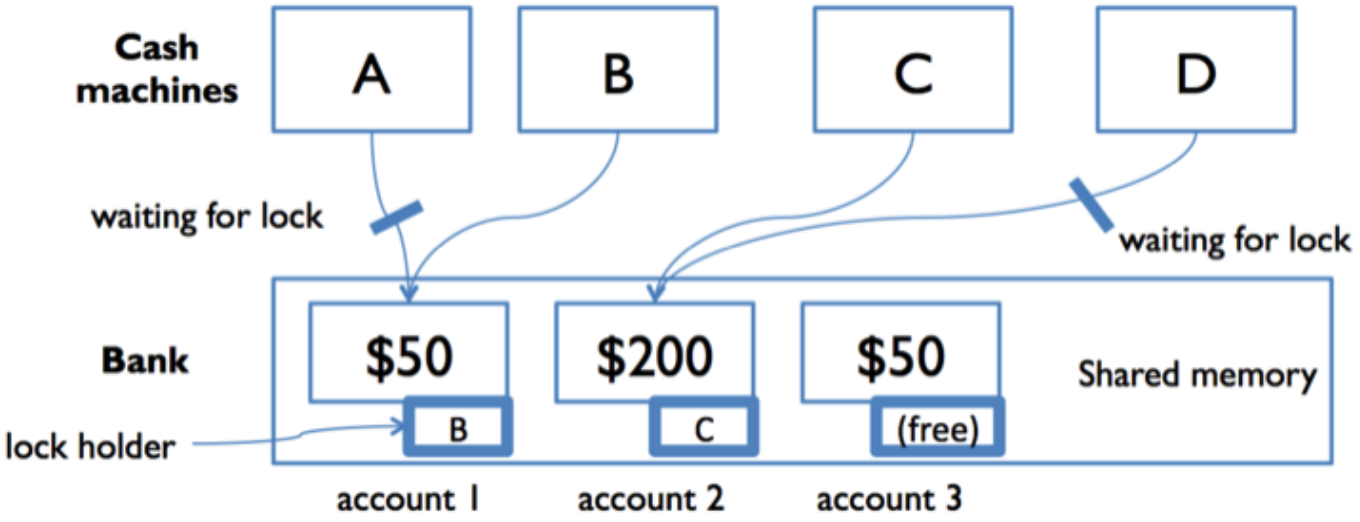
## 锁（Lock）

- 通过加锁(lock)和解锁(unlock)两个原子过程提供对共享数据的互斥访问(Mutual exclusion，Mutex)



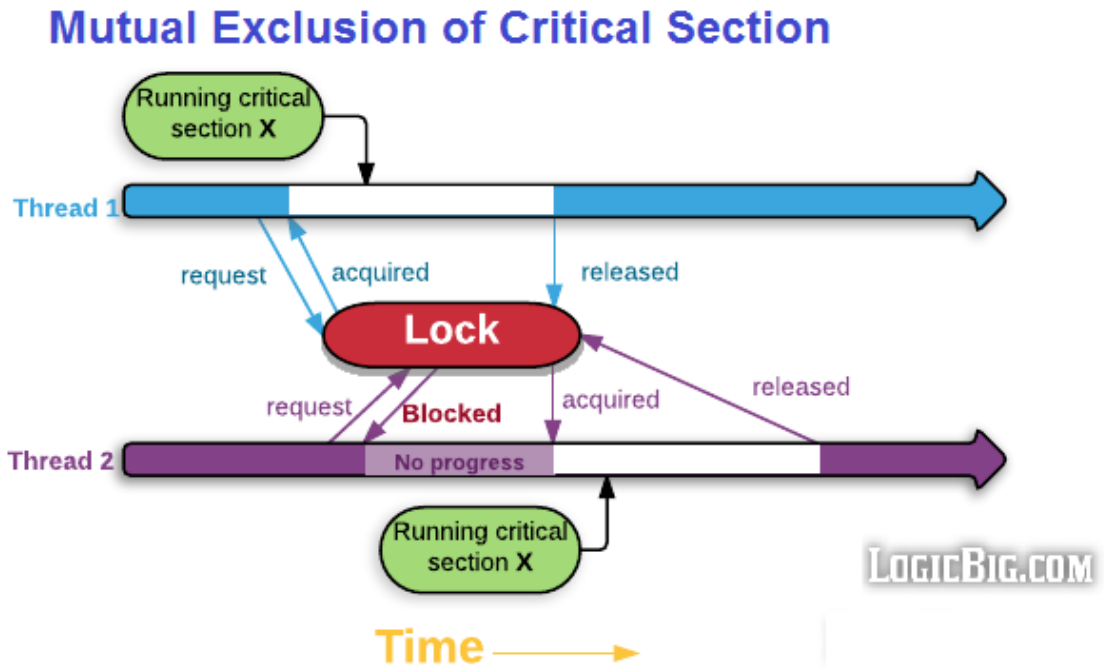
## 锁（Lock）

- 通过加锁(lock)和解锁(unlock)两个原子过程提供对共享数据的互斥访问(Mutual exclusion，Mutex)



## 锁（Lock）

- 通过加锁(lock)和解锁(unlock)两个原子过程提供对共享数据的互斥访问(Mutual exclusion，Mutex)



## 锁（Lock）的实现

- 通常需要硬件支持
- 基于原子操作（atomic operations）实现：
  - 测试并设置锁 Test and Set Lock (TSL)
  - 获取并增加 Fetch-and-Increment
  - 交换 Swap
  - 比较并交换 Compare-and-Swap (CAS)
  - 加载链接/条件存储 Load-linked / Store-Conditional LL/SC

## 锁（Lock）的分类

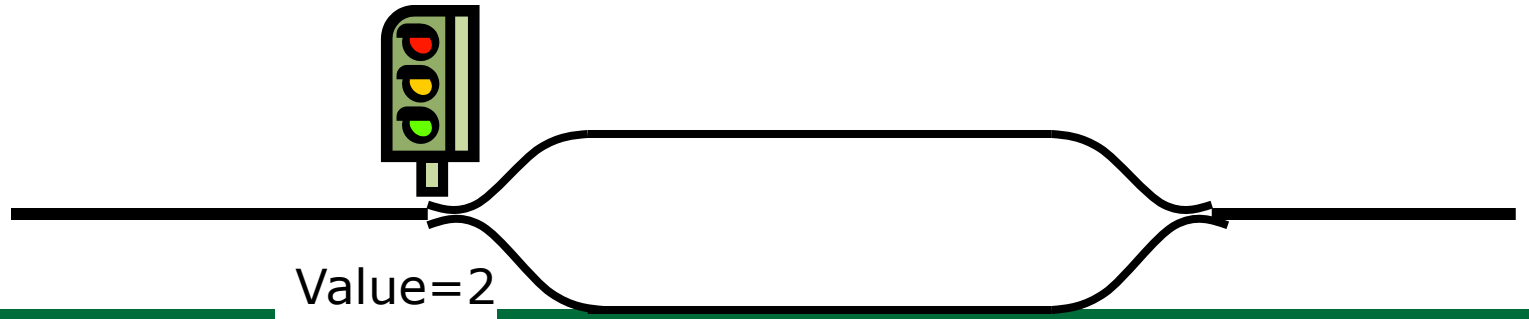
- 根据是否阻塞划分：阻塞锁、自旋锁（非阻塞锁）
- 根据是否可重入划分：可重入锁（递归锁）、不可重入锁
- 根据是否是读写锁划分：读写锁、互斥锁
- 根据是否公平划分：公平锁、非公平锁

## 信号量（Semaphore）

- 是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。
- 计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。
- 适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。
- 二进制信号量（binary semaphore）又称互斥锁（Mutex）。
- 信号量操作即操作系统课程中的“PV操作”，P（wait()）与V（signal()）

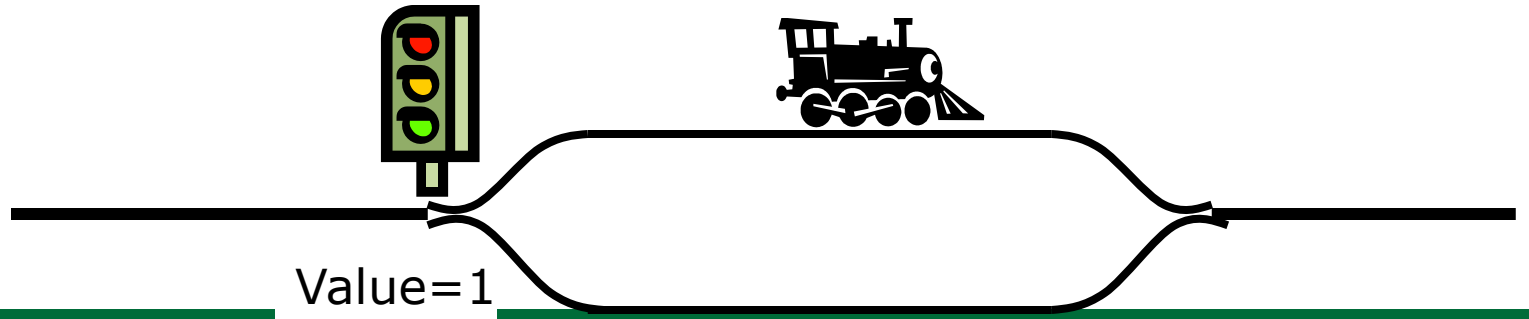
## 信号量（Semaphore）

- 是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。
- 计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。
- 适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。
- 二进制信号量（binary semaphore）又称互斥锁（Mutex）。
- 信号量操作即操作系统课程中的“PV操作”，P（wait()）与V（signal()）



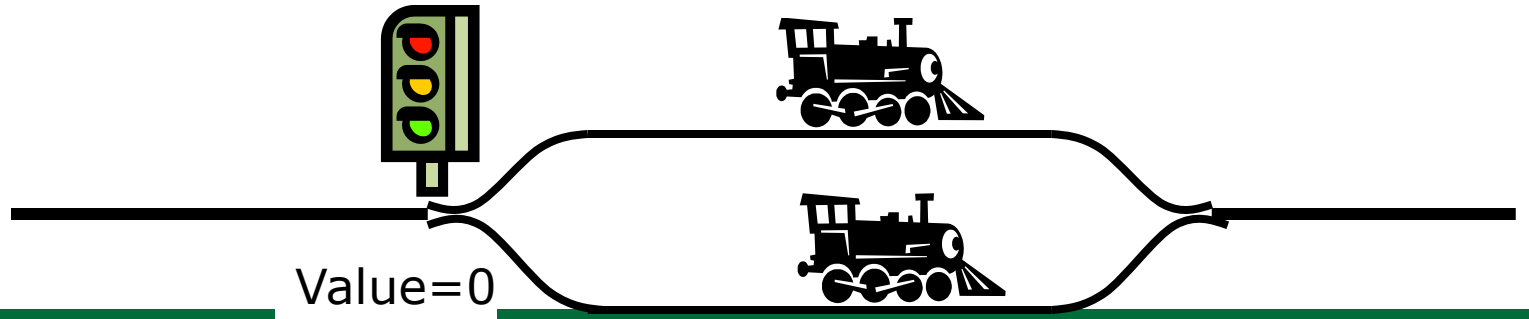
## 信号量（Semaphore）

- 是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。
- 计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。
- 适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。
- 二进制信号量（binary semaphore）又称互斥锁（Mutex）。
- 信号量操作即操作系统课程中的“PV操作”，P（wait()）与V（signal()）



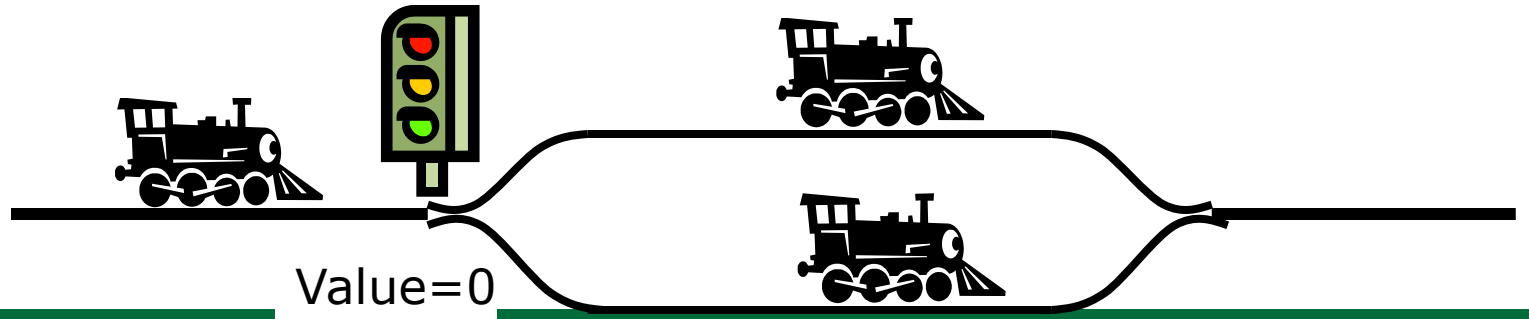
## 信号量（Semaphore）

- 是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。
- 计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。
- 适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。
- 二进制信号量（binary semaphore）又称互斥锁（Mutex）。
- 信号量操作即操作系统课程中的“PV操作”，P（wait()）与V（signal()）



## 信号量（Semaphore）

- 是一个同步对象，用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待（wait）时，该计数值减一；当线程完成一次对semaphore对象的释放（release）时，计数值加一。
- 计数值大于0，为signaled状态；计数值等于0，为nonsignaled状态。
- 适用于控制一个仅支持有限个用户的共享资源，是一种不需要使用忙碌等待（busy waiting）的方法。
- 二进制信号量（binary semaphore）又称互斥锁（Mutex）。
- 信号量操作即操作系统课程中的“PV操作”，P（wait()）与V（signal()）



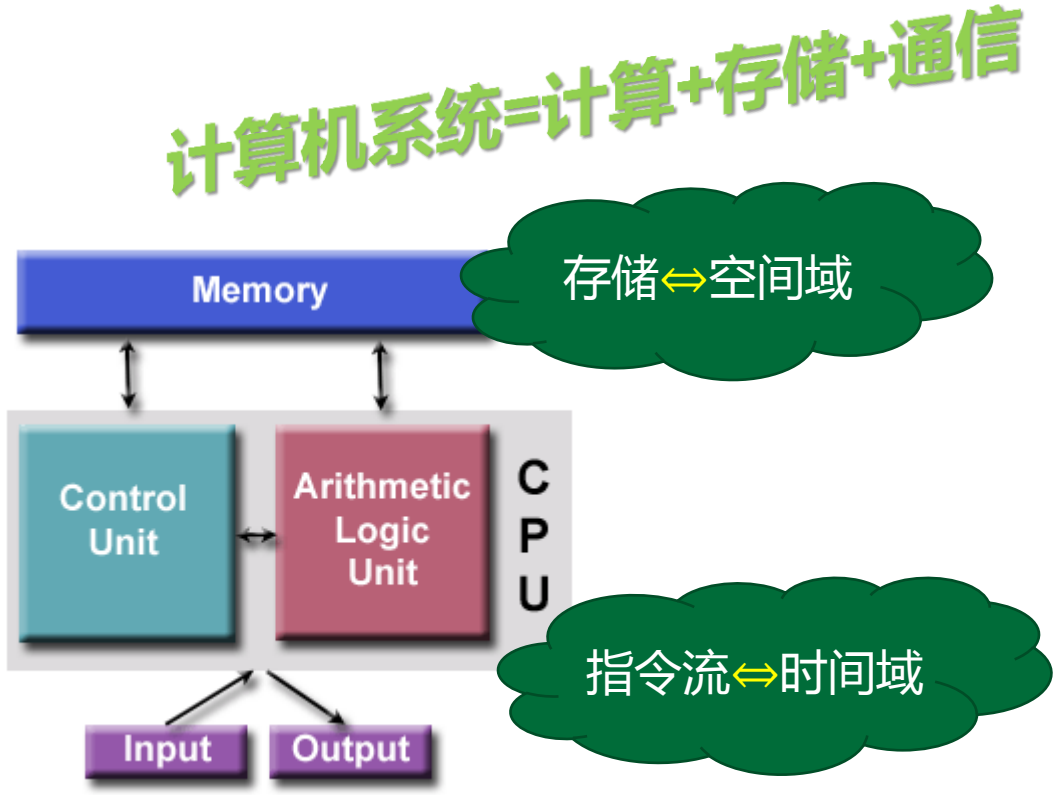
## 同步通信操作

- 仅涉及执行通信操作的那些任务。
- 当任务执行通信操作时，需要与参与通信的其他任务进行某种形式的协调。例如，在任务可以执行发送操作之前，它必须首先从接收任务接收到可以发送的确认。

| | 同步通信 | 异步通信 |
|---|---|---|
| 传输格式 | 面向比特的传输，每个信息帧中包含若干个字符 | 面向字符的传输，每个字符帧只包含一个字符 |
| 时钟 | 要求接受时钟和发送时钟同频同相，通过特定的时钟线路协调时序 | 不要求接受时钟和发送时钟完全同步，对时序要求较低 |
| 数据流 | 发送端发送连续的比特流 | 发送端发送完一个字节后，可经过任意长的时间间隔再发送下一个字节 |
| 控制开销 | 控制字符开销较小，传输效率高 | 字符帧中，假设只有起始位、8个数据位和停止位，整个字符帧中的控制位的开销就达到了20%，传输效率较低 |
| 同步方式 | 从数据中抽取同步信息 | 通过字符起止的开始位和停止位抓住再同步的机会 |
| 通信结点 | 点对多点 | 点对单点 |


# 5 通信（Communications）

- 通信在计算机系统中的作用越来越重要



- **传统的通信模式分类：**
  - 同步和异步
  - 串行和并行
  - 单工（单向）和双工（双向）
  - 单播、组播/多播和广播
- **任务间的通信模式：**
  - 点对点（一对一）
  - 一对多
  - 多对一

## 通信的优化

- 减少通信量
  - 优化算法
  - 保证负载平衡
  - 数据冗余（复制）
- 减少通信开销
  - 高效的网络接口
  - 快速的路由方法
  - 优化通信协议
  - 打包数据
- 通信和计算重叠（Overlap）


## 影响程序可扩展性（Scalability）的因素

- 阿姆达尔定律
  - 串行代码，临界区代码等
- 并行性是否充足
  - 如数据量N<处理器个数P时，扩展性差
- 并行开销
  - 线程创建，同步，通信，调度
- 局部性
- 负载均衡


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工