并行处理
L03-02：并行编程基础
——负载平衡、局部性以及竞争

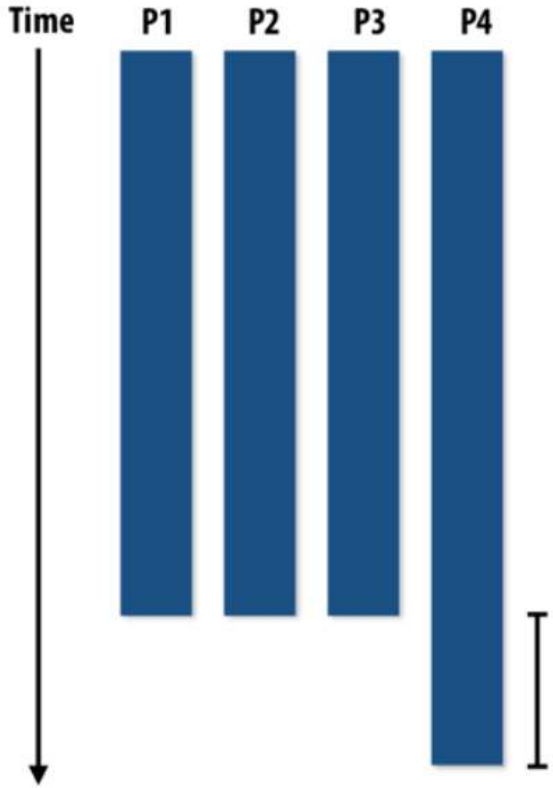
叶笑春
中国科学院计算技术研究所

面向高性能计算的并行编程

- 优化并行程序的性能是一个**反复迭代**且复杂的代码改造过程（问题分解、任务分配和资源编排）
- 并行程序优化的核心目标：**提高加速比**。包括以下关键措施（**经常是相互矛盾的**）
  - 保证计算任务在可用的计算资源上的负载平衡(Load-balance)
  - 减少通信与竞争 (避免依赖导致的计算停顿)
  - 减少为提高并行性、资源分配、减少通信与竞争等而带来的额外开销（计算、访存等）

负载平衡（Load-balance）

Ideally：所有的处理器在程序执行过程中一直在计算（它们同时计算，它们同时完成分配给自己的那部分工作）



回忆阿姆达尔定律：
即使只有少量的负载不平衡，就可以显著限制最大加速比（并行加速比）

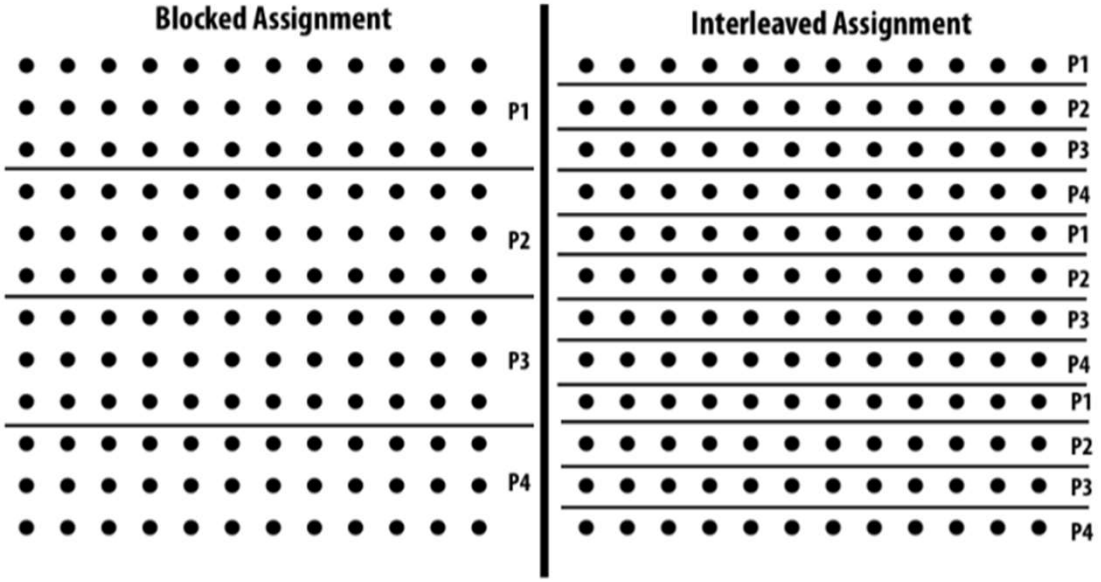
P4 多做 20% 的工作 -> P4 需要额外 20% 的时间才能完成
-> 并行程序运行时间的 20% 是串行执行

（此处无法并行的计算任务，约占整个程序工作量的5%）

静态分配(Static assignment)

- 线程的工作分配是预先确定的
  - 不一定在编译时确定（赋值算法可能取决于运行时参数，如输入数据大小、线程数等）
- 为每个线程（工作者）分配相等数量的网格单元（工作）
  - 此前，我们讨论了两种静态分配方式: 阻塞式（blocked ）和交错式（interleaved）
- 静态赋值的良好特性：简单，基本上为零运行时开销

阻塞式静态分配（blocked ）　交错式静态分配（interleaved）



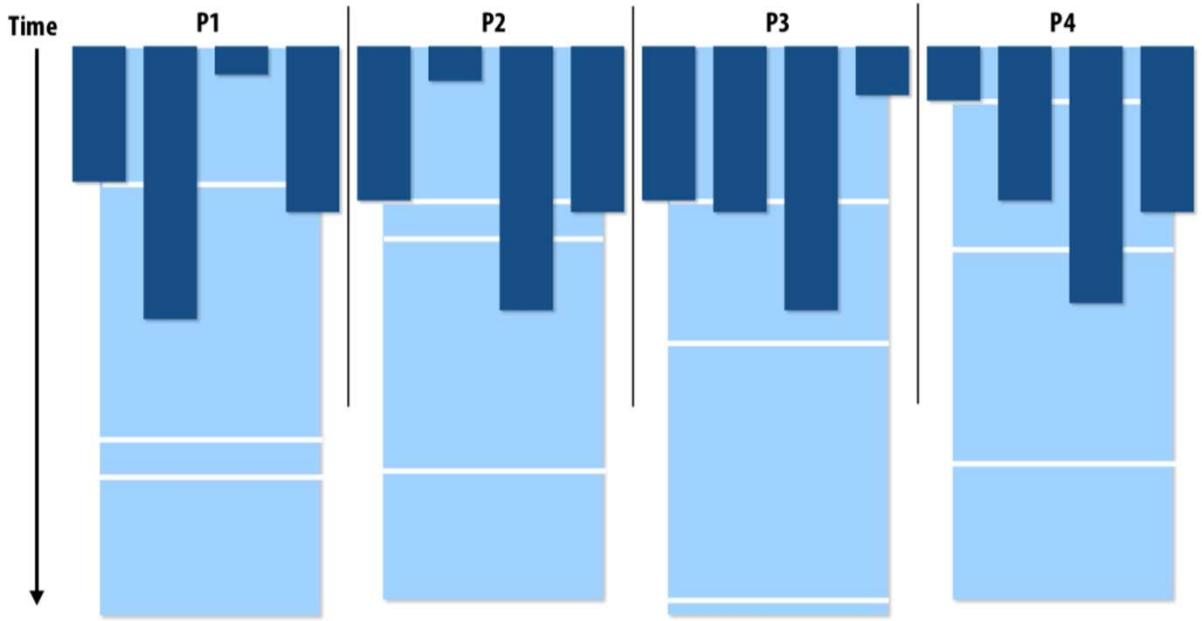
静态分配什么时候适用?

- 当工作的**成本（执行时间）和工作量**是可预测的（这样程序员就可以提前制定好分配）
- 最简单的例子：预先知道所有子计算任务的时间耗时成本相同

在上面的例子中：
有 12 个任务，已知每个计算任务的成本相同。
分配方案：静态分配三个任务给四个处理器

静态分配什么时候适用?

- 当工作是可预测的，即使并非所有工作的成本都相同时
- 当有关执行时间的统计数据已知时（例如，平均成本相同）



作业具有不平等但已知的成本：分配给处理器以确保整体良好的负载平衡
四个处理，每个处理器串行执行四个计算任务（每个处理器的总计算时间T是相似的）

“半静态”(Semi-static) 分配

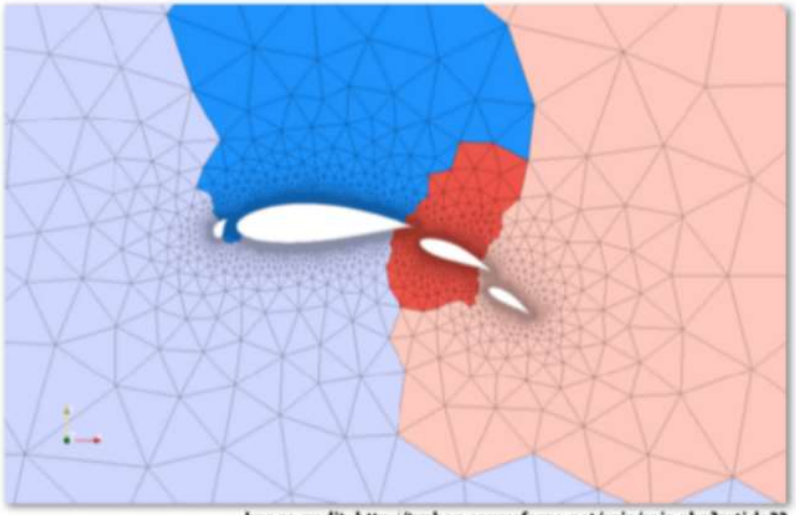
- 短期内的工作成本是可预测的
  - Idea: 用近期完成任务的执行时间，预测邻近的短期任务的执行时间
- 应用程序定期进行自我分析并重新调整分配
  - 重新调整之间的间隔分配是静态的

N 体模拟

N-Body simulation:

在模拟过程中移动时重新分配处理器（如果运动缓慢，则不需要经常发生重新分配）

自适应网格



Adaptive mesh:

网格随着对象移动或流过对象变化而变化，但变化发生得很慢（颜色表示将部分网格分配给不同的处理器）

动态分配

- 随程序的运行和执行时，动态确定分配处理器，以确保负载平衡。
  （任务的执行时间，或者说任务总数，是不可预测的）

顺序程序（独立循环迭代）

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

不可预测

并行程序
(SPMD通过多线程共享地址空间模型执行)

```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;    // shared variable

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

动态获取对应的处理项

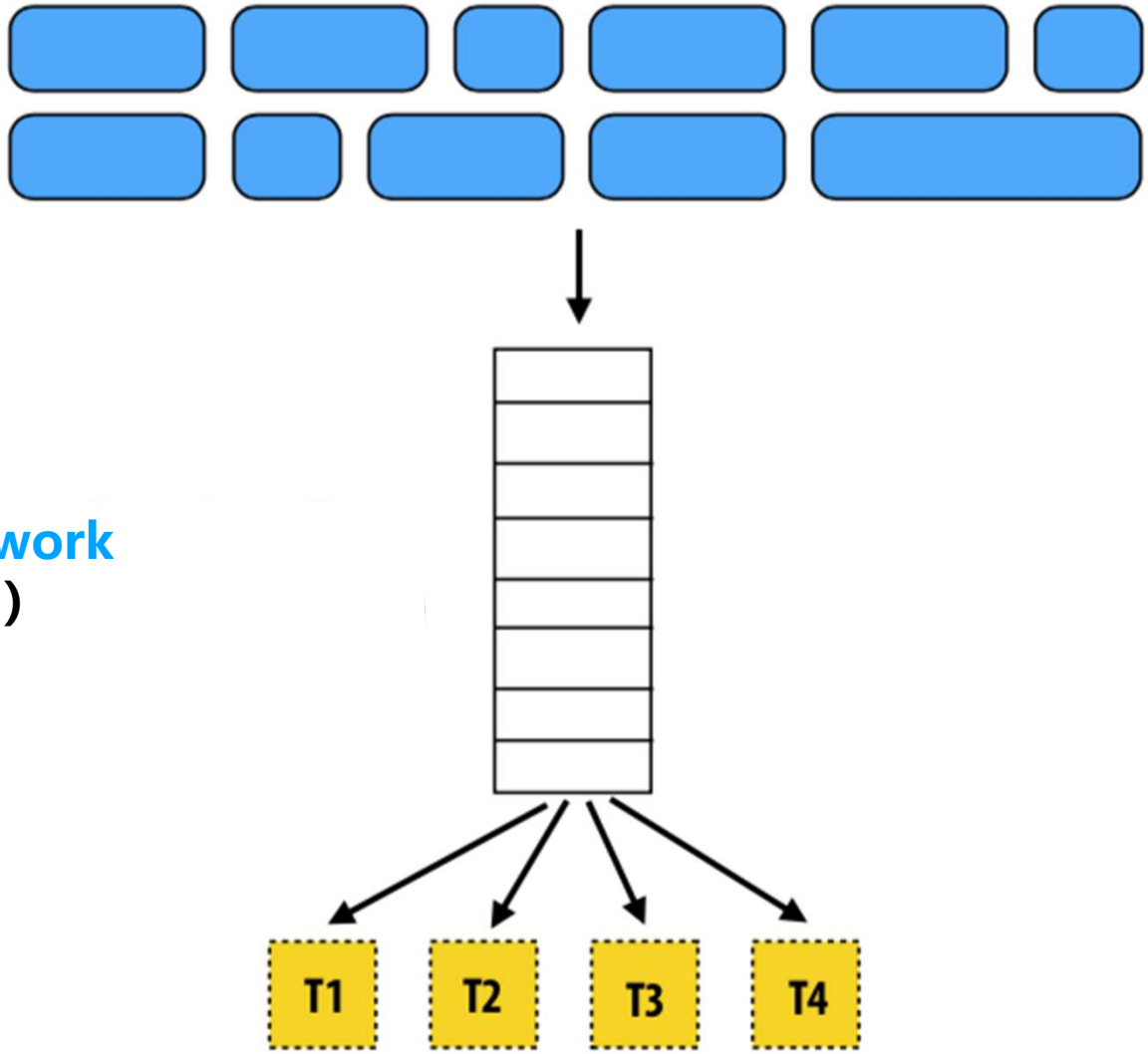
atomic_incr(counter);

使用队列(Queue)进行动态分配

问题分解（分解子任务）

Sub-problems

(a.k.a. "tasks", "work")



共享FIFO工作队列：一组要做的work

(我们假设每项work都是独立的)

Work线程：

从共享work队列中提取数据

在创建新work时将其推送到队列

每个work由什么构成?

细粒度划分：1个work对应处理1个数据元素

```
const int N = 1024;
// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

- 这是串行程序中不存在的开销
- 而且..它是串行执行（回忆阿姆达尔定律）

- **这个实现有什么潜在问题?**

可能良好的负载平衡（许多小任务，small tasks），往往带来潜在的高同步成本（在临界区要通过序列化，串行执行，降低同步开销）

提高任务粒度

```
const int N = 1024;
const int GRANULARITY = 10;
// assume allocations are only executed by 1 thread

float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     is_prime[j] = test_primality(x[j]);
}
```

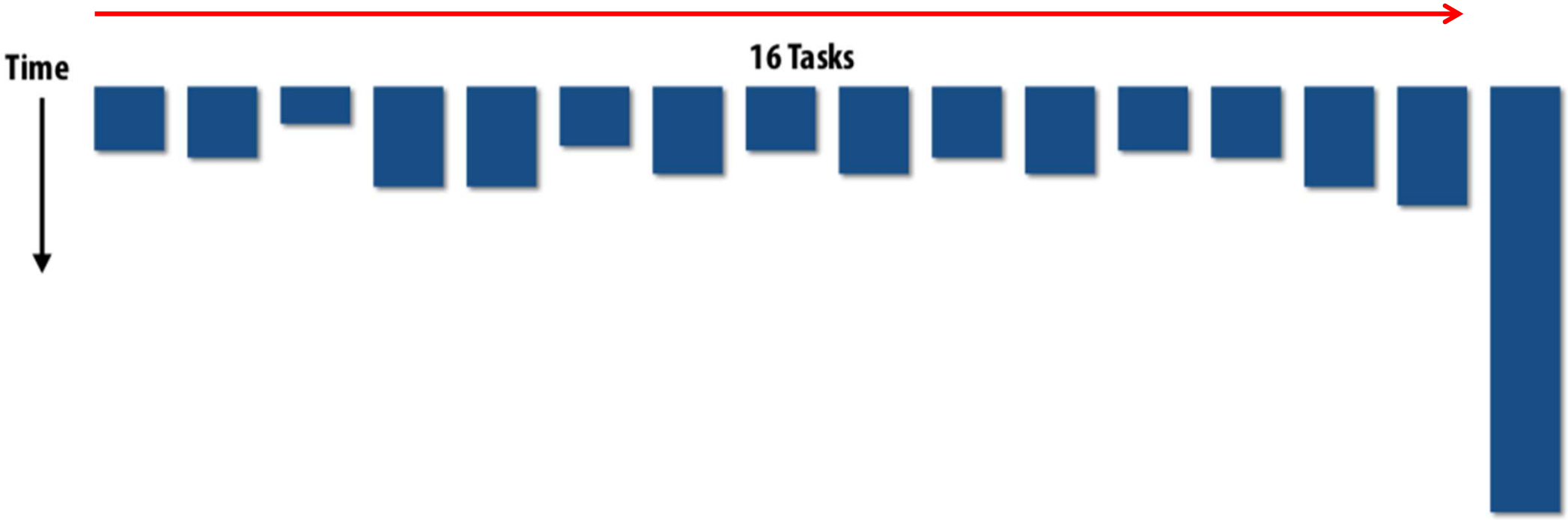
- 粗粒度划分：1个任务处理10个元素
- 降低同步成本（进入临界区的次数减少 10倍）

选取task的大小

- 一方面，我们希望有更多的任务（许多小任务通过动态分配实现良好的工作负载平衡
  - 细粒度划分子任务（每个任务处理的数据量少，任务数量更多）
- 另一方面，我们又想要减少任务数量，最小化管理分配的开销（如：同步）
  - 粗粒度划分子任务（每个任务处理的数据量多，任务数量更少）
- 因此，理想的粒度取决于许多因素（共同主题：必须了解具体的工作负载和机器架构）

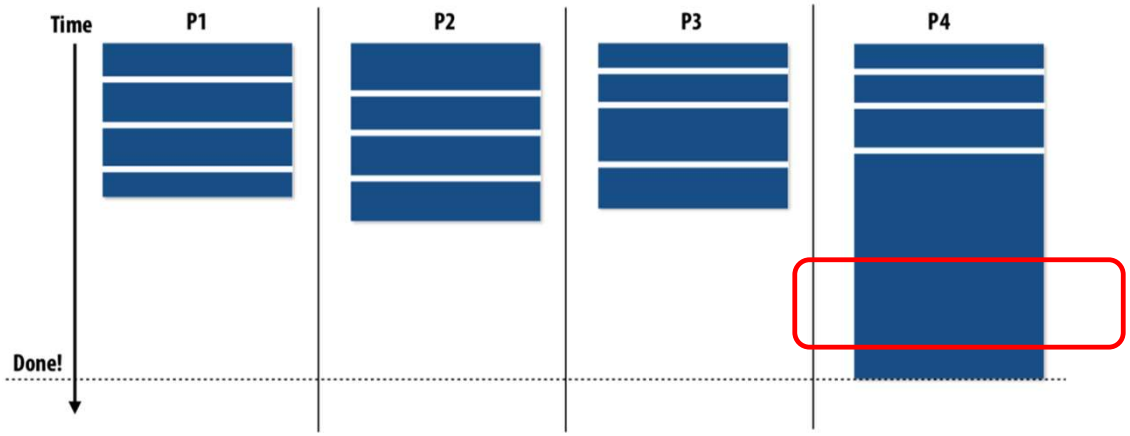
任务调度

- 通过共享工作队列进行动态调度
- 如果系统按从左到右的顺序将这些任务分配给worker，会发生什么?



更智能的任务调度

- 如果调度程序最后运行长任务会怎样?
  - 会出现潜在的负载不平衡! (load imbalance!)

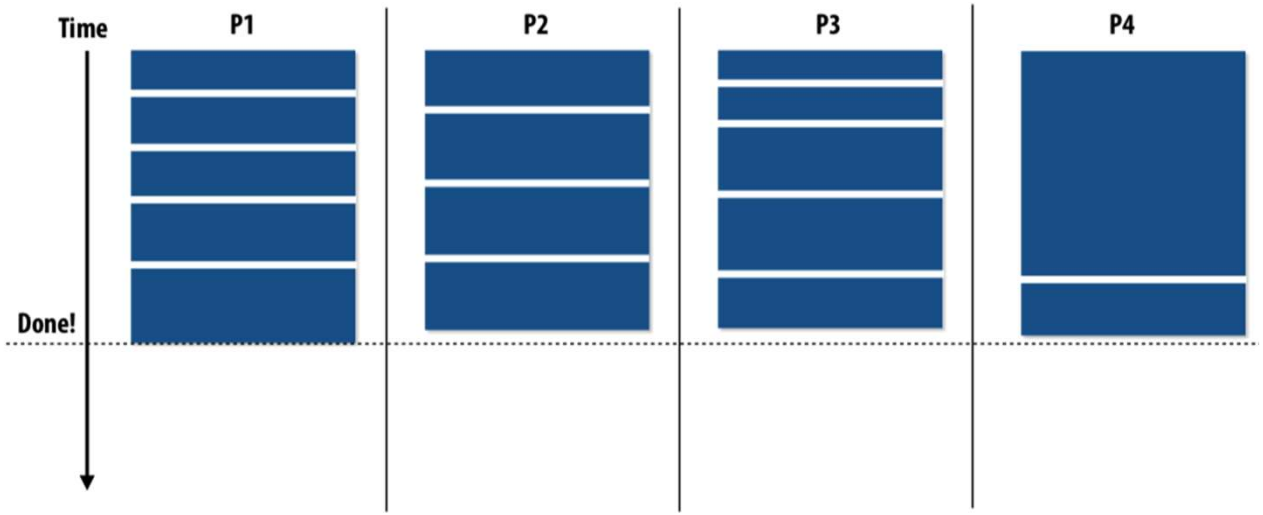


“长杆task”
(long pole task)

- 不平衡问题的一种可能解决方案：将工作分成更多的小任务
  - 相对于整体执行时间，希望“长杆task”的执行时间变短
  - 可能会增加同步开销（用不同处理器执行），
  - 也可能效果不佳！因为把常规task切小后，也很可能无法并行到不同处理器上，还是在相同的处理器上串行执行（长杆任务基本上是顺序执行的）

更智能的任务调度

- 安排长杆task优先执行，以减少计算结束时的“溢出”



- 将所有线程任务按执行时间长短排序，安排执行时间长的任务优先执行
  - 每个线程执行的task数量不同, 但对应的总的计算量大致相同
  - 需要对各个task的执行时间具有可预测性，评估各个task的执行时间

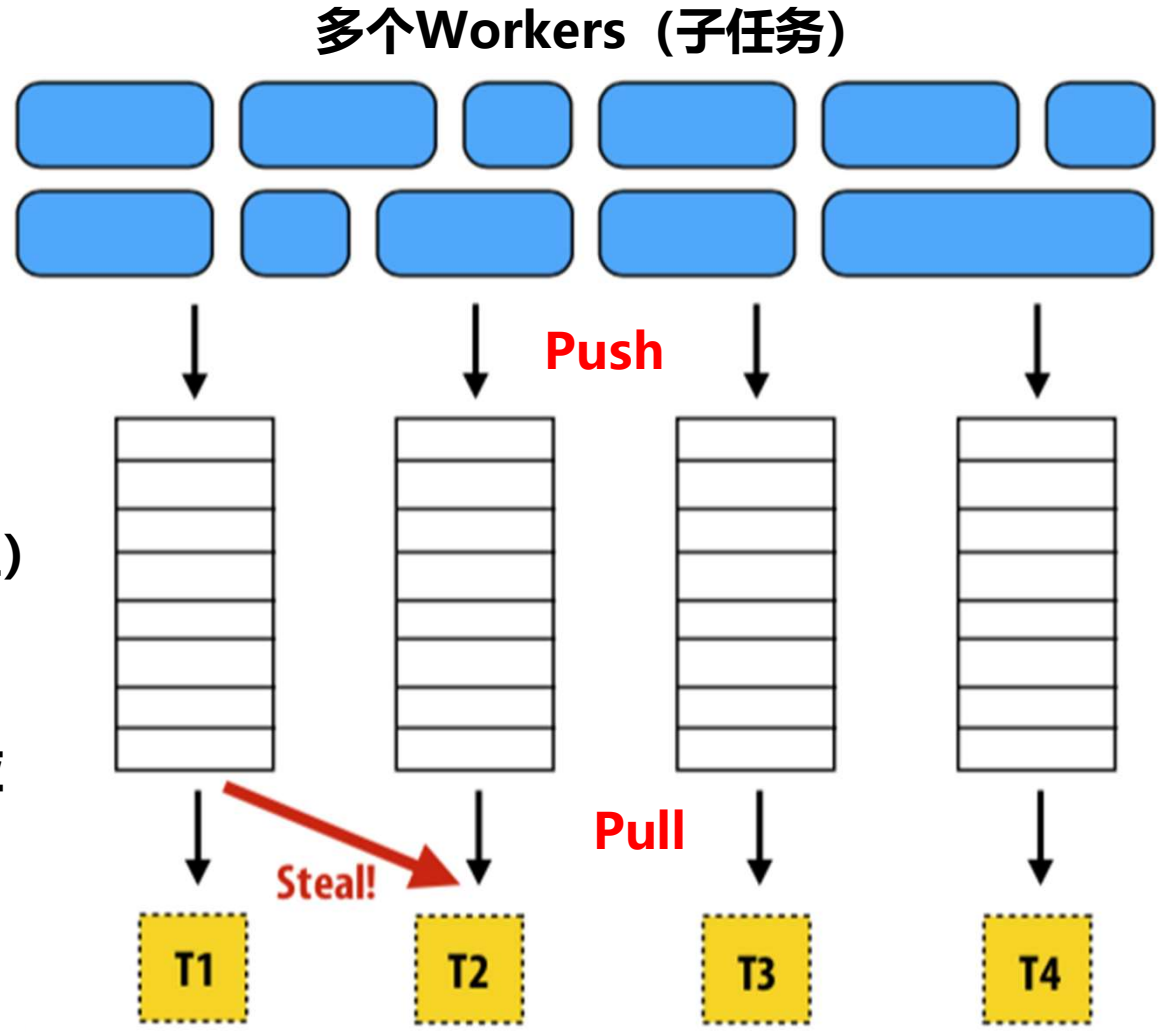
确保负载平衡时再减少同步开销

- 避免在单个工作队列上的各个worker间进行同步，所以考虑使用多个独立队列

问题分解（分解子任务）

Subproblems

(a.k.a. "tasks", "work to do")



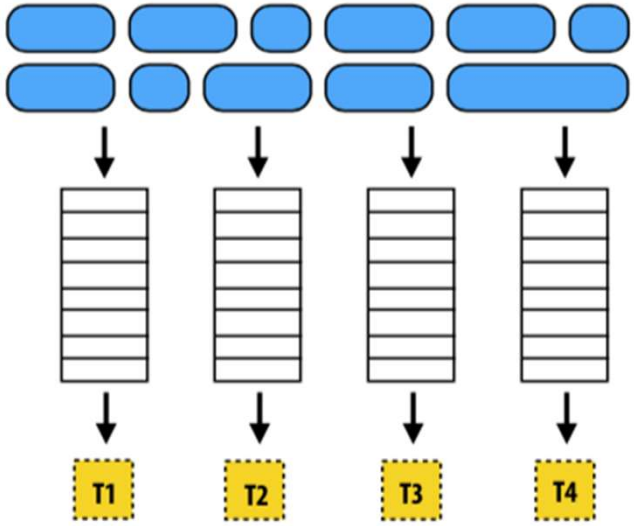
一组工作队列
(一般来说，一个队列对应一个线程)

工作线程：

- **从工作队列中拉取(Pull)数据**
- **将新工作推送(Push)到自己对应的工作队列**
- **当本地工作队列为空时...**
- **从另一个队列中进行“工作窃取”（work Stealing机制）**

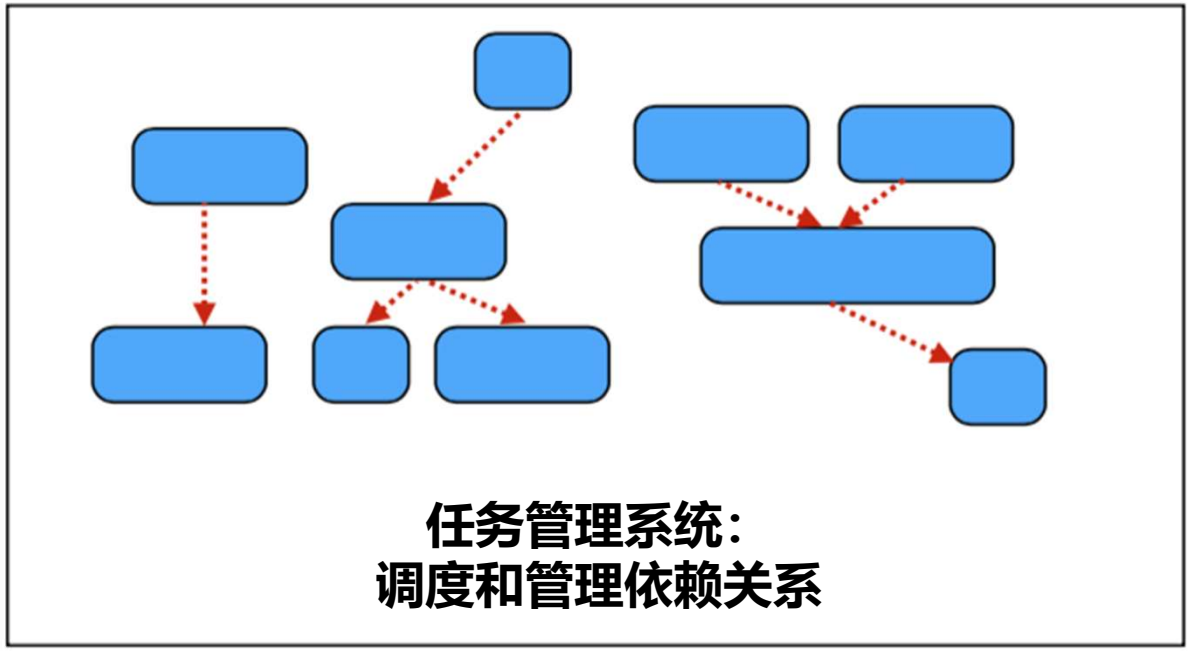
分布式工作队列

- 避免了频繁的同步
- 还可能提升数据的局部性
  - 常见情况：线程处理它们自己创建的任务（生产者-消费者局部性）
- “窃取”期间仍会有同步/通信开销
  - 窃取仅在必要时发生以确保良好的负载平衡
- 实施挑战
  - 从谁那里偷？/要偷多少？
  - 如何检测程序终止？
  - 确保本地队列访问速度快（同时保留互斥）

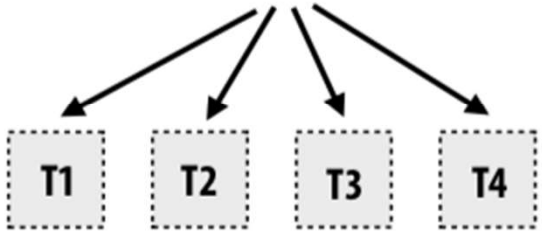


任务队列中的工作彼此并不独立（有相互依赖）

= 依赖关系



在满足所有任务依赖性之前，任务不会从队列中出队并分配给工作线程



worker可以向任务系统提交新任务（带有可选的显式依赖项）

```
foo_handle = enqueue_task(foo);                 // enqueue task foo (independent of all prior tasks)
bar_handle = enqueue_task(bar, foo_handle);     // enqueue task bar, cannot run until foo is complete
```

小结-负载平衡

- 挑战：如何实现良好的工作负载平衡
- 希望所有处理器一直工作（否则资源空闲）
- 关键是如何实现这种平衡的低开销（ low-cost ）解决方案
  - 最小化计算任务间的管理开销（例如，调度/分配逻辑）
  - 最小化同步成本
- 静态分配与动态分配
  - 这不是一个非此即彼的决定，而是需要经过一系列思考的多次选择
  - 尽可能使用有关工作负载的先验知识，以减少负载不平衡和任务管理/同步成本
    - 在极限情况下，如果系统知道一切，就使用完全静态分配

局部性：什么是局部性?

- **时间局部性**：指在相对较短的持续时间内对特定数据和/或资源的重用
- **空间局部性**：指在相对靠近的存储位置内使用数据元素

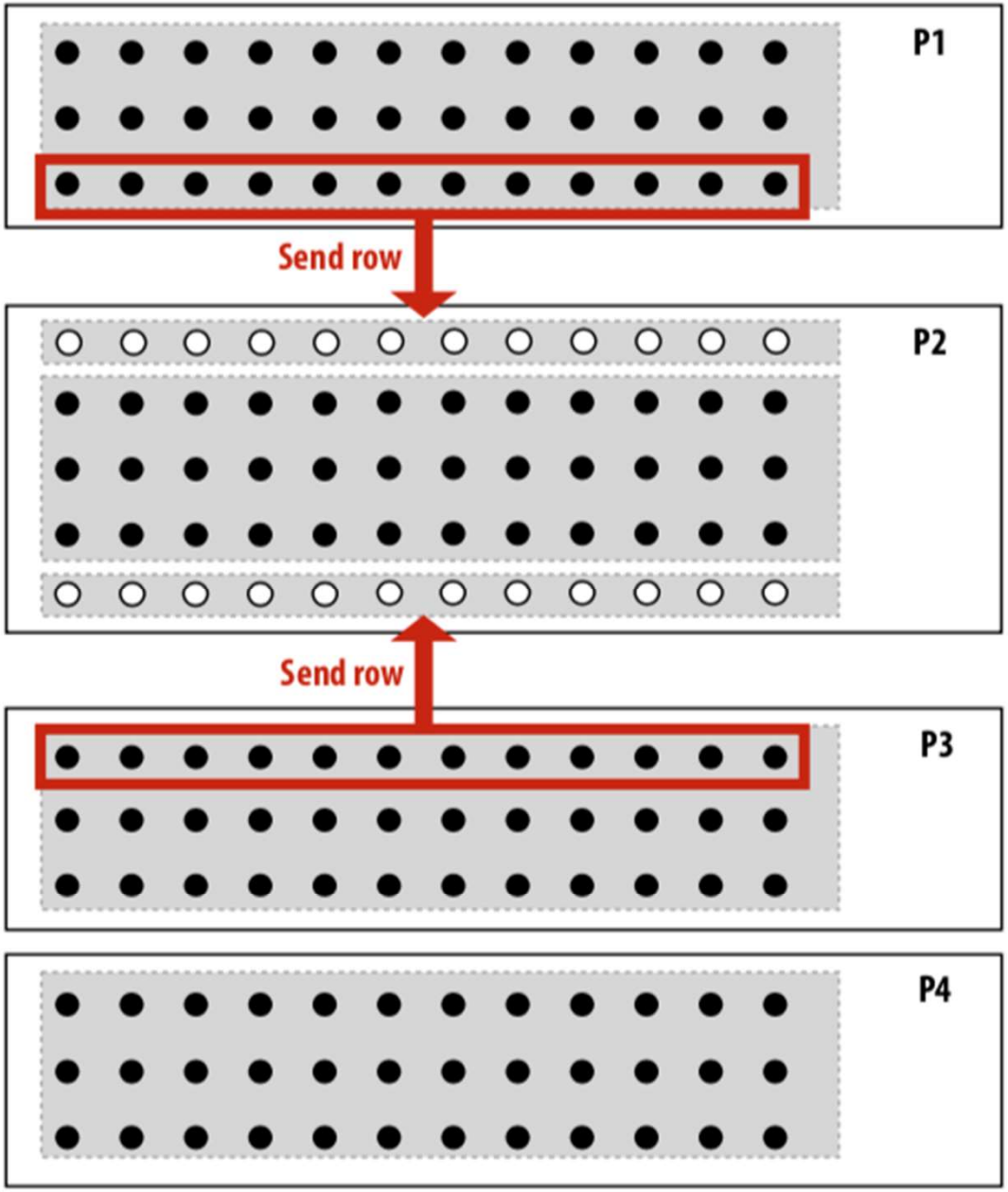
思考：cache利用了什么局部性原理?

局部性优化的目标是减少额外的数据通信开销

数据通信的两个原因

- 固有原因
- 人为原因

原因一：固有原因引发的通信



并行算法中必须进行的通信。通信是算法的基础

在我们课程开始时的消息传递示例中，发送 ghost 行是固有的通信

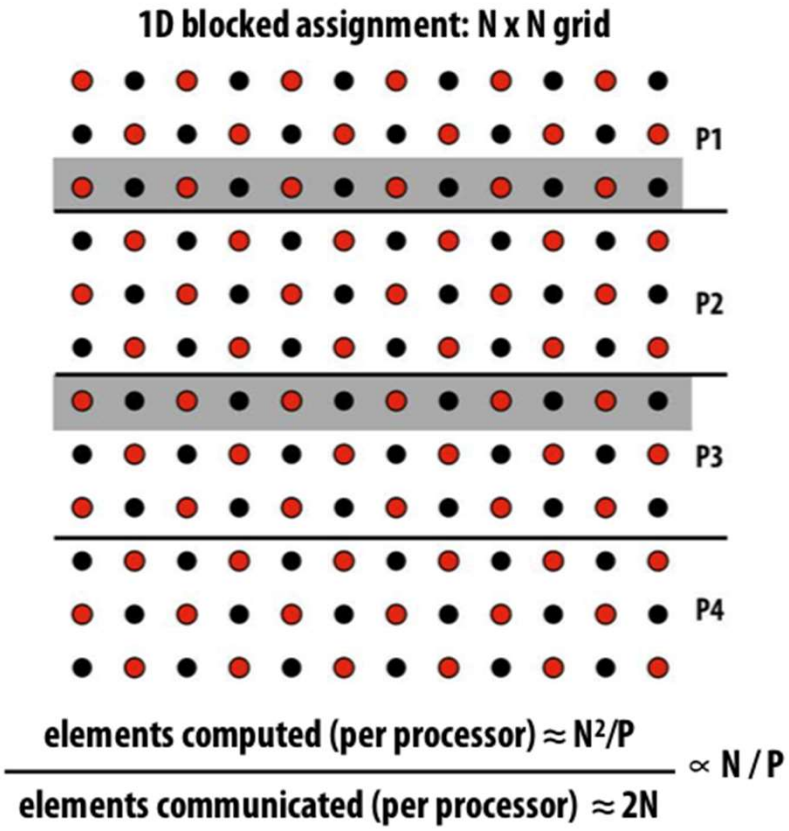
计算与通信的比率

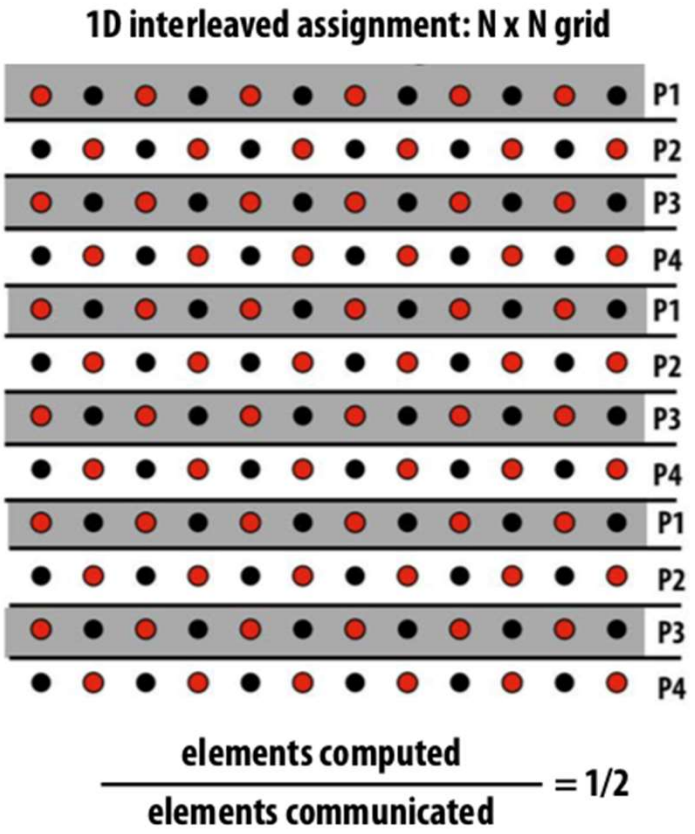
$$\frac{\text{amount of computation(e.g., instructions)}}{\text{amount of communication(e.g., bytes)}}$$

- **运算强度**=计算与通信的比例
- 由于计算能力与可用带宽的比率通常很高，因此需要高运算强度才能有效地利用现代并行处理器
  - 提高计算比重，降低通信比重

优化思路：减少固有的通信

- 好的处理器分配决策，可以减少固有的沟通（增加运算强度）

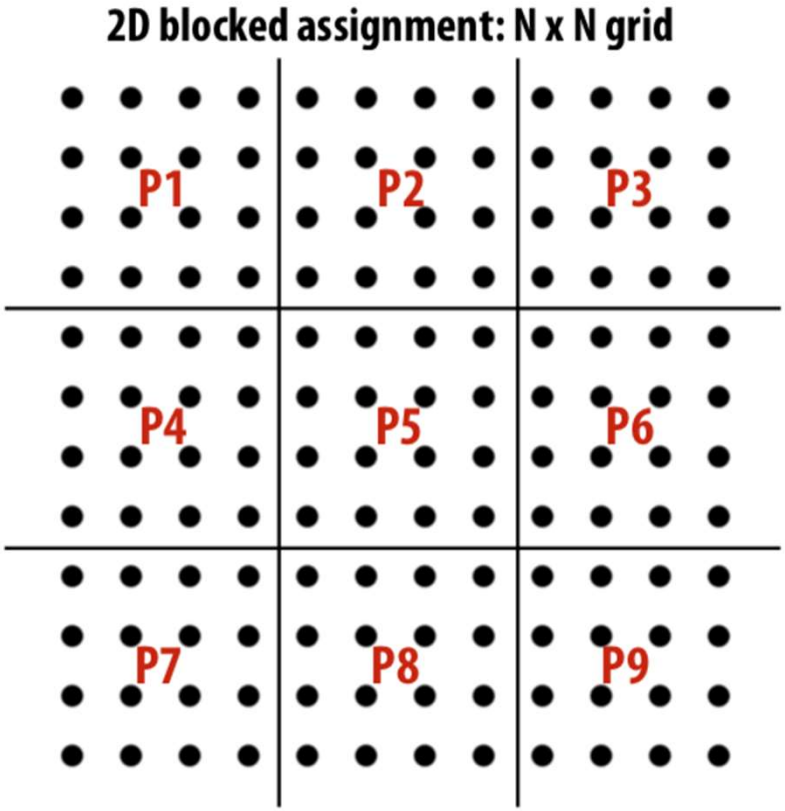




GOOD：运算强度约为N/P

BAD：运算强度 = 1/2

减少固有的通信



N^2 elements

P processors

elements computed: (per processor) $\frac{N^2}{P}$

elements communicated: (per processor) $\propto \frac{N}{\sqrt{P}}$

arithmetic intensity: $\frac{N}{\sqrt{P}}$

二维分配方式下比一维阻塞分配获得更好的运算强度

原因二：人为原因引发的通信

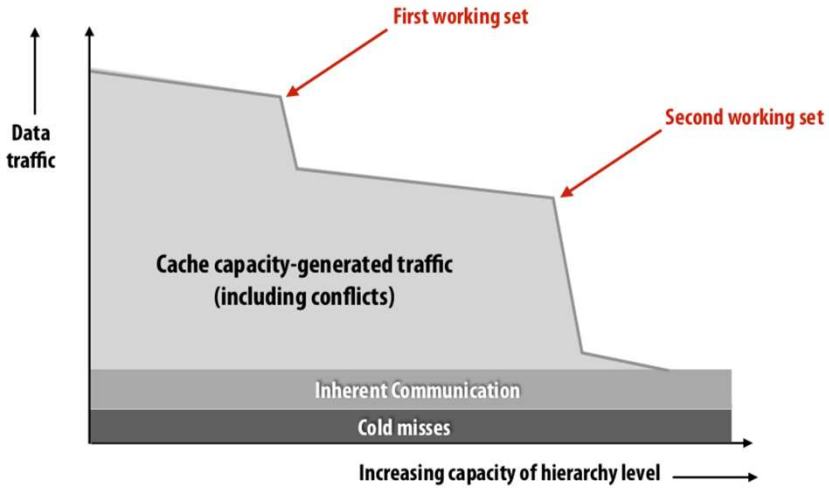
- 固有通信：在给定计算和缓存资源（假设无限容量缓存、最小粒度传输等）的情况下，从根本上**必须**在处理器之间完成的通信操作（不完成就无法执行算法）
- 人为通信：所有其他通信（人为通信源于系统实施的实际细节），也就是非必须的、业务之外的通信

人为原因引发通信的例子

- 系统存在一个数据传输的较大迁移粒度（ granularity of transfer）
  - 导致：系统必须传送比所需要的数据，更多的开销数据
  - 程序加载一个 4 字节浮点值，但必须从内存中传输整个 64 字节的cache line
- 系统可能具有导致不必要通信的操作规则（rules of operation）
  - 程序本来只是想存储16个连续的4字节值，但是整个 64 字节的cache line都要**先从内存中加载，然后存储到内存中**（cache访问机制导致的2x开销）
- 数据在分布式内存中的放置不当，导致访存路径过长（数据不在最常访问它的处理器附近，不在cache中，导致**cache miss**）
- 对数据拷贝的存储容量不足
  - 相同的数据多次传送给处理器，因为缓存太小而无法在多次访问之间保留它

Cache miss的3Cs→4Cs模型

- Cold miss：首次访问，导致缓存未命中
  - 第一次访问新数据，该miss在串行程序中无法避免
- Capacity miss：缓存容量不足，导致缓存未命中
  - 需要缓存的工作数据集大于缓存容量。 可以通过增加缓存大小来避免/减少
- Conflict miss：冲突，导致缓存未命中
  - 由缓存管理策略引起的未命中。 可以通过更改缓存关联性或应用程序中的数据访问模式来避免/减少
- Communication miss (new)：通信，导致缓存未命中
  - 由于并行系统中固有的，或人为通信引起的cache miss，比如由于cache coherence操作导致的miss

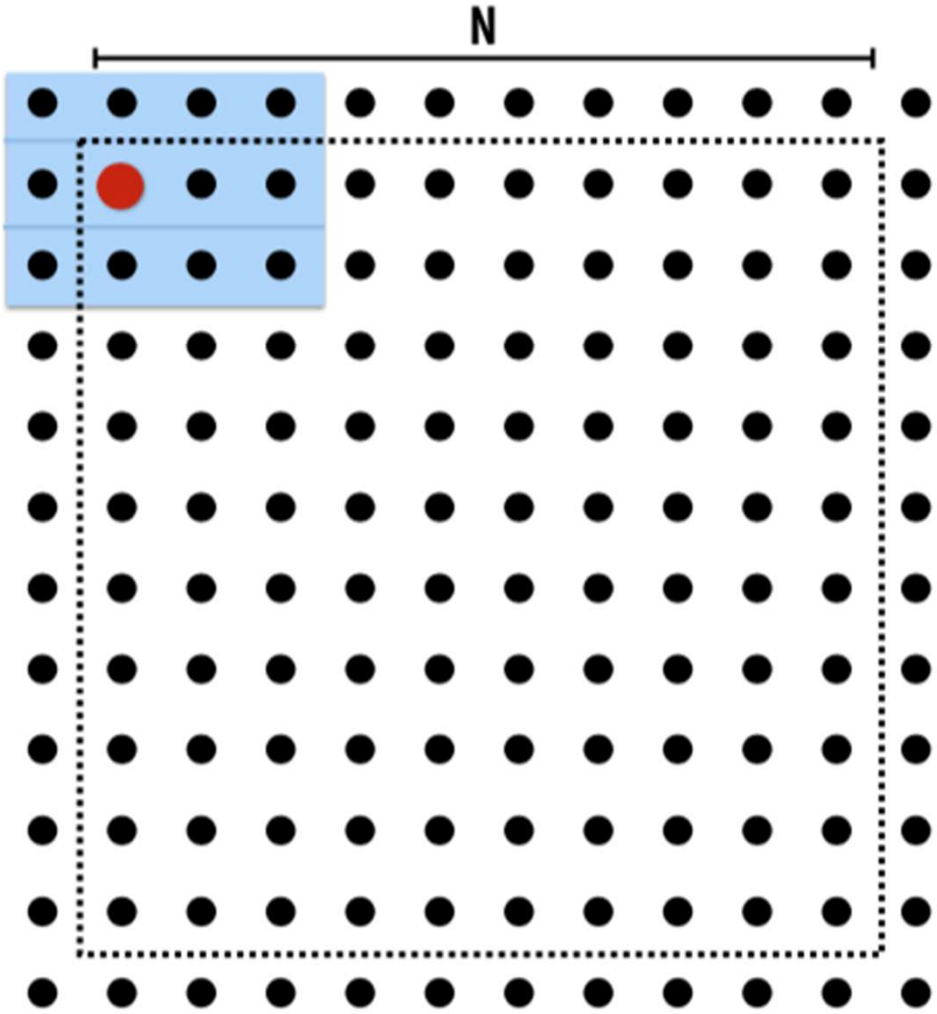


思考：为什么会有这种现象？针对不同的workload，如何设置cache容量?

优化思路：减少通信的一些技巧

网格求解器中的数据访问

- 行优先遍历

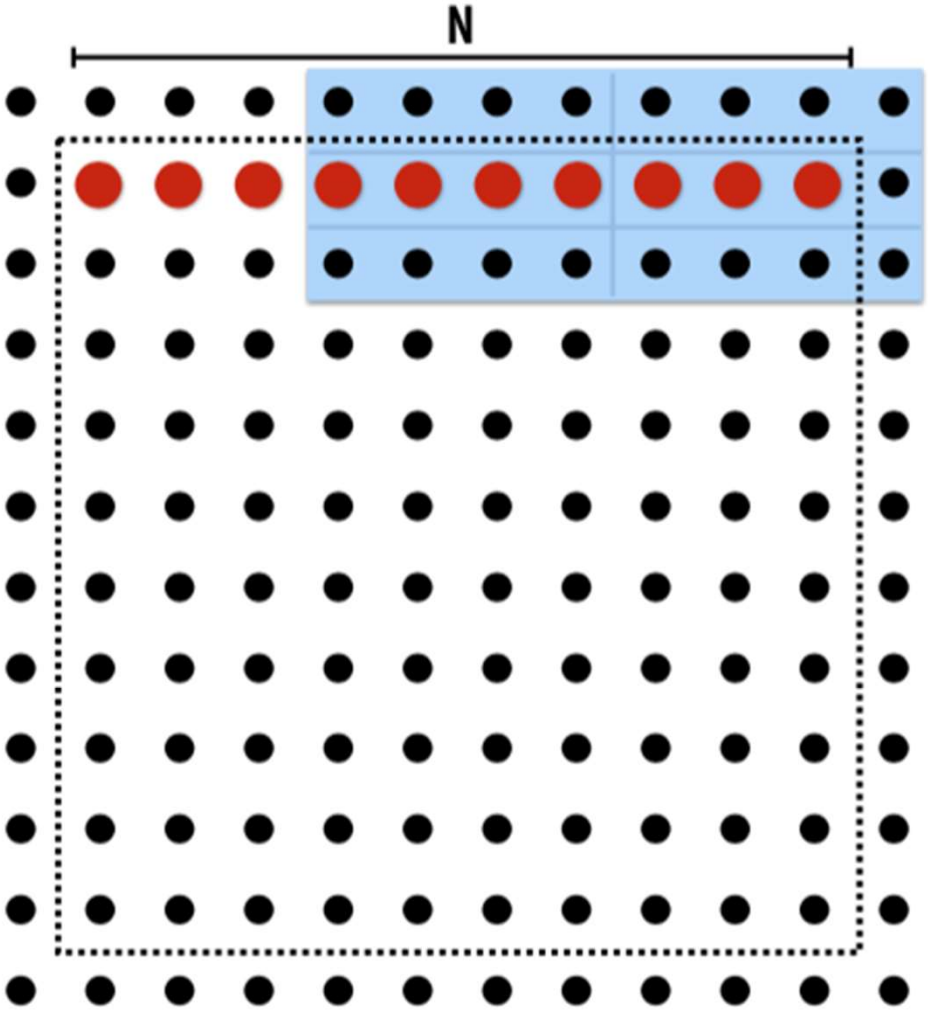


- 假设行优先网格布局。
- 假设cache line是 4 个网格元素
- 缓存容量为 24 个网格元素（6个cache line)

- 调用网格求解器应用程序
- 蓝色元素显示缓存中的数据
- 更新的网格元素为红色元素

网格求解器中的数据访问

- 行优先遍历

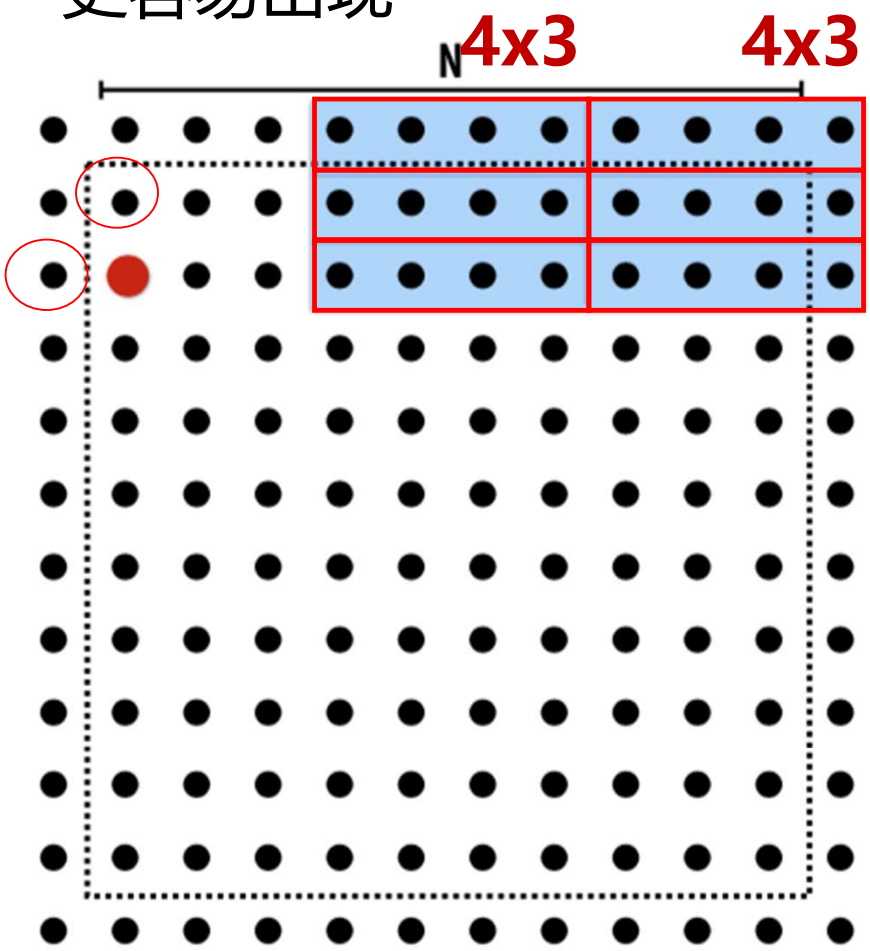


- 假设行优先网格布局。
- 假设cache line是 4 个网格元素
- 缓存容量为 24 个网格元素（6个cache line)

- 蓝色元素显示在处理第一行结束时缓存中的数据。（停在哪儿）

行优先遍历的问题

- 访问相同数据的间隔时间长，不利于数据局部性，cache miss更容易出现



- **假设行优先网格布局。**
- **假设cache line是 4 个网格元素**
- **缓存容量为 24 个网格元素（6个cache line)**

- **此时有什么问题?**

- **尽管元素 (0,2) 和 (1,1) 之前已被访问过，但在处理第 2 行开始时它们不再存在于缓存中**

改善时间局部性

- 融合循环

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

2次加载，每个数学运算1次存储
(运算强度=1/3)

2次加载，每个数学运算1次存储
(运算强度=1/3)

整体运算强度=1/3

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

4次加载，每3个数学运算1次存储
(运算强度=3/5)

顶部的代码更加模块化（例如，基于数组的数学库），但底部的代码执行的性能更好。为什么?

通过数据共享提高运算强度（arithmetic intensity）

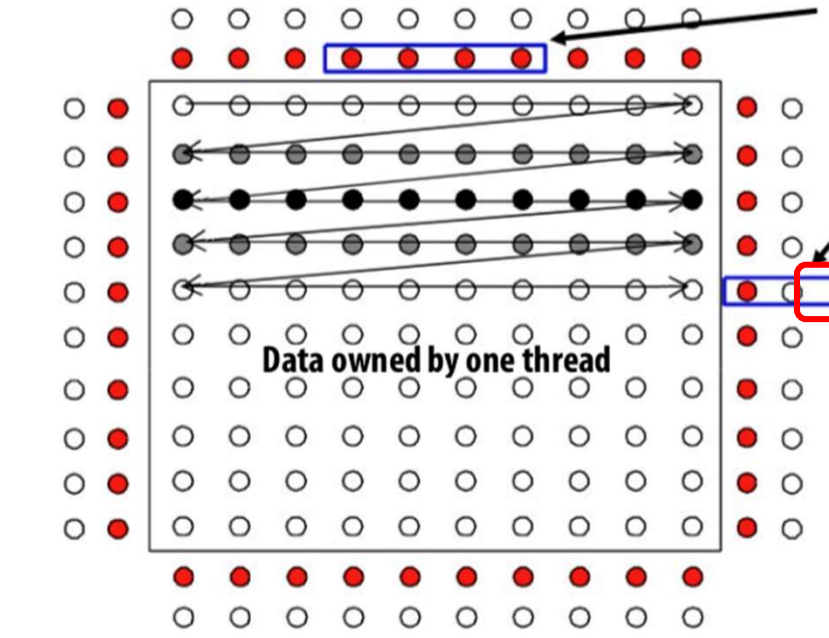
- **挖掘共享：多个本地任务共同操作相同位置的数据**
  - 安排多个线程，在同一处理器上，同时处理同一数据结构
  - 减少固有的通信
- 示例：CUDA 线程块（CUDA thread block）
  - 一种抽象：在 CUDA 程序中，以本地化的方式（不跨处理器），处理相关操作
  - 线程块中的线程，以SIMT方式进行协作操作（通过CUDA的shared memory进行数据共享）
  - GPU 在同一个 GPU 核心上，调度来自同一个线程块的多个线程

利用空间局部性

- 通信的粒度可能很重要，因为它可能会引入人为的通信
  - 通信/数据传输的粒度
  - 缓存一致性（cache coherence）的粒度

通信粒度

- 如前所述，二维划分数据块的方式，将数据分配给处理器
- 假设：**通信粒度**是一个cache line，一个cache line包含四个元素



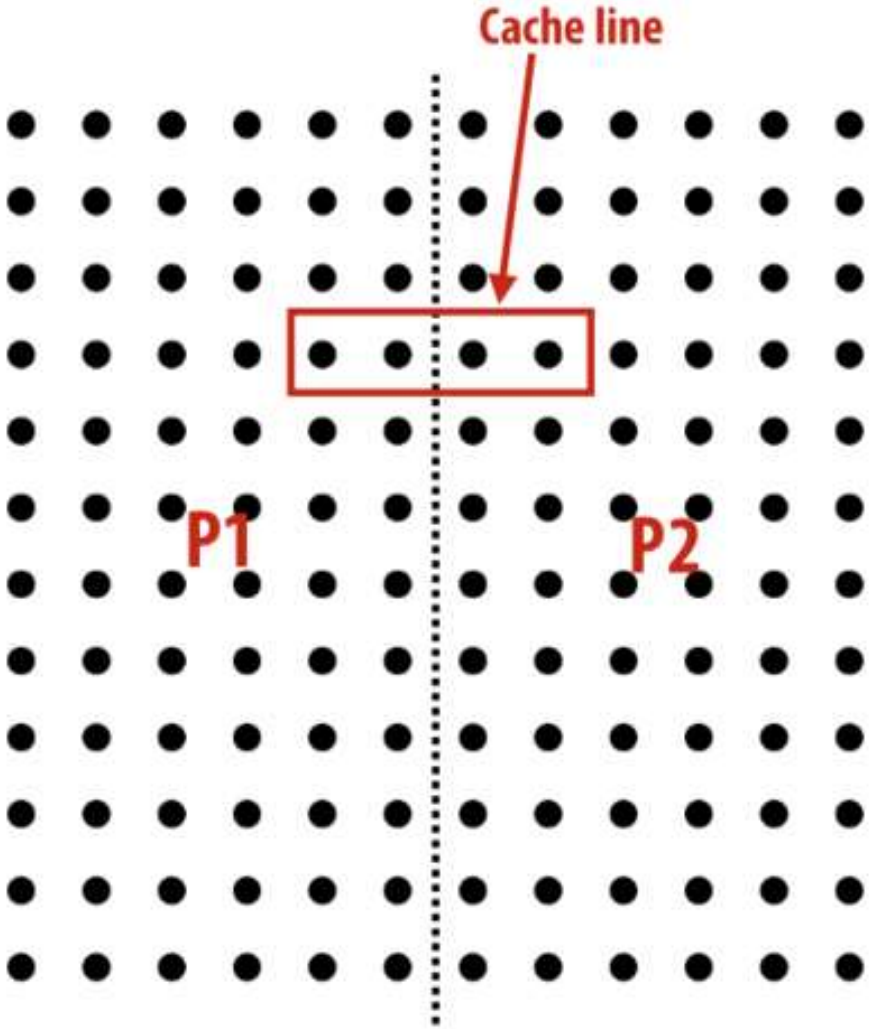
GOOD: 顶部和底部行的非本地访问具有良好的空间局部性

BAD: 对左右列的非本地访问的空间局部性差

本质上需要来自左右邻居的一个元素，但系统必须通信四个。

这意味着，由于人为因素而引发的通信，随着cache line大小的增加而增加！

以Cache line作为通信粒度

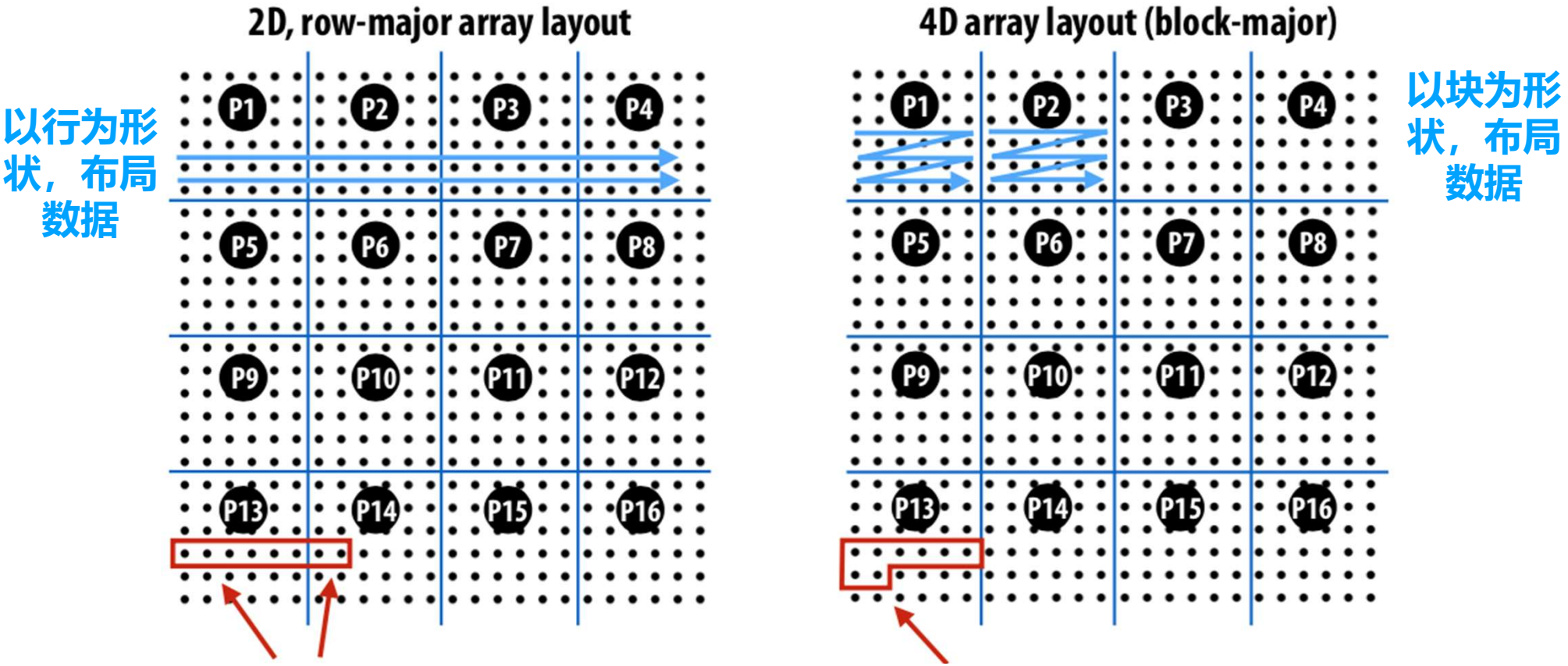


- 数据按列分成两半。分区分配给在处理器1（P1）和处理器2(P2)上运行的线程
- 线程访问分配给他们各自的元素（不存在固有通信）
- 但是，由于同一个cache line被两个处理器同时写入数据，所以在真实系统上的数据访问会触发（人为的）通信*，来解决缓存一致性的问题
  - P1和P2需要对同一个cache line写入数据

* 后面的缓存一致性课程中会有更多详细信息

减少人为通信

- 分块数据布局（blocked data layout ）



注意：不要混淆数据的分块处理与数据的分块布局

小结-局部性

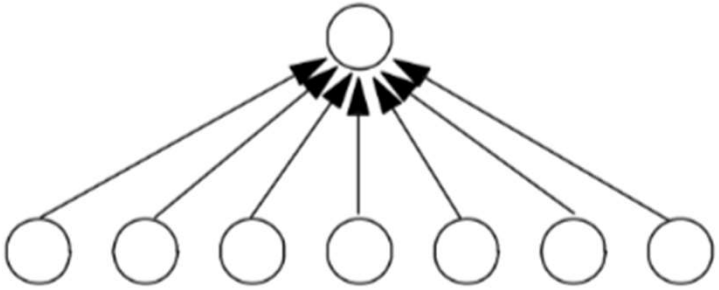
- 考虑局部性的核心目标是要减少通信，提高运算强度
- 通信包括固有与人为所引发的
  - 考虑到问题是如何分解的以及工作是如何分配的，固有的通信是完成整个程序的基础操作，难以避免
  - 人为所引发的通信取决于机器实现细节
- 识别和利用局部性：减少通信（增加运算强度）
  - 减少开销（降低通信次数、增加单次通信的信息量）
  - 结合之前讲的预取等机制来最大化通信和计算的重叠（隐藏延迟以免产生时间开销）

竞争
(Contention)

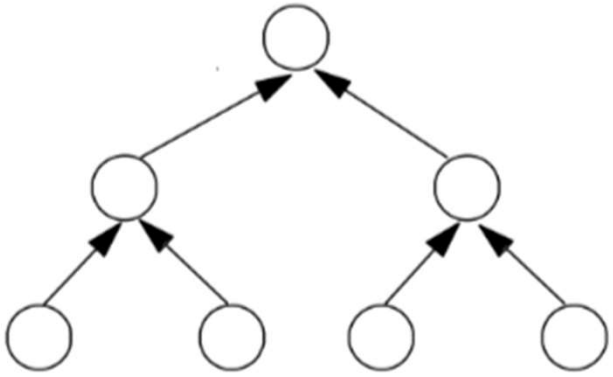
竞争（Contention）

- 每类资源，都有其最大的吞吐量：每单位时间内能够完成的处理数量
  - 如：内存、通信链路、服务器等...，都可以视为一种共享资源
- **当在一个小的时间窗口期内，对共享资源发出许多请求时，就会发生争用（资源成为热点和瓶颈）**

示例：更新共享变量



扁平式通信：可能发生高概率竞争（但如果没有竞争则延迟低）



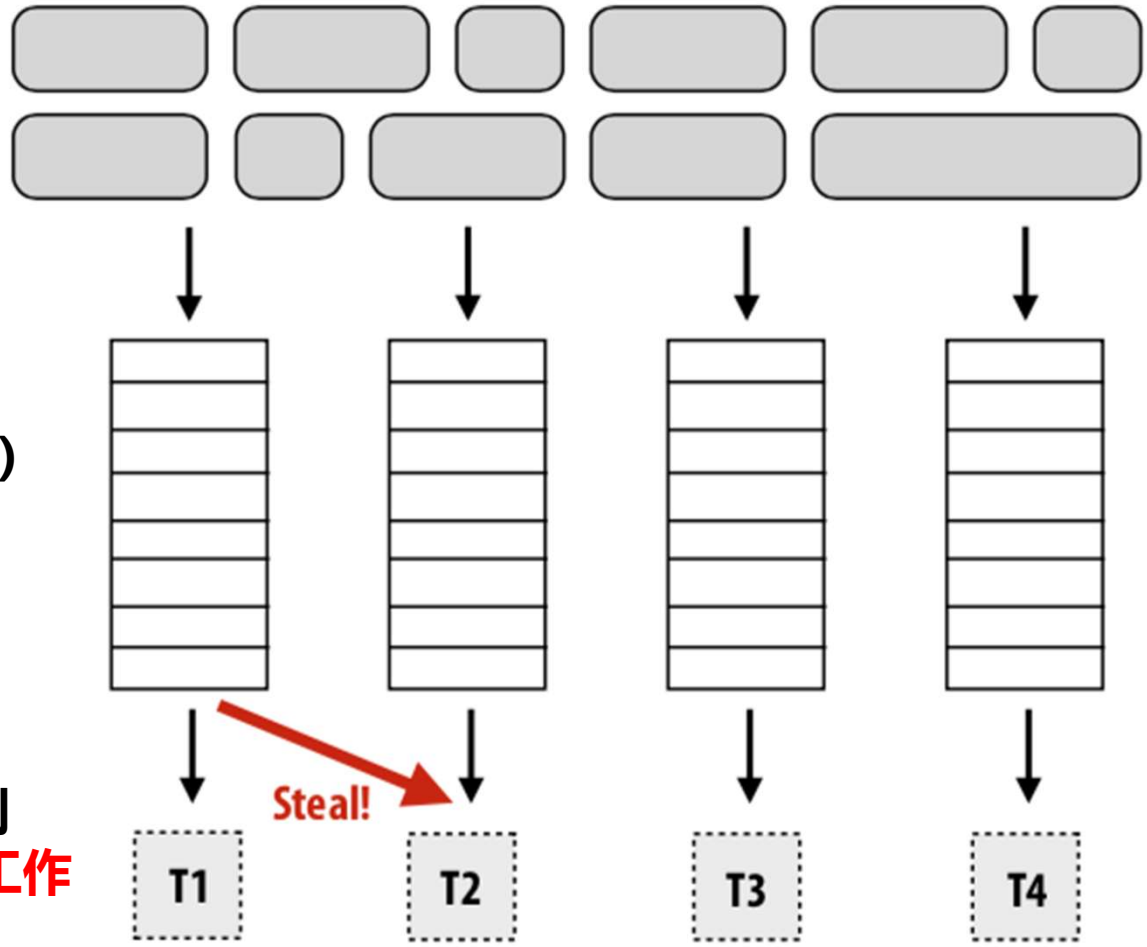
树结构式通信：减少竞争（但，即使在无竞争时，延迟比扁平式通信要高）

减少竞争

- 分布式作业队列

问题分解（分解子任务）
Subproblems
(a.k.a. "tasks", "work to do")

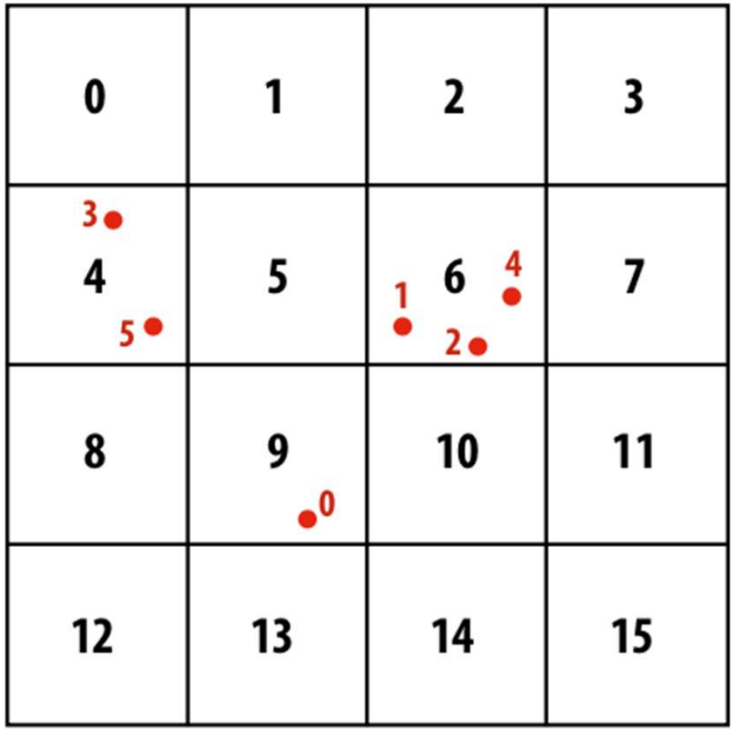
一组工作队列
(一般来说，一个队列对应一个线程)



工作线程:
- **从 T1 工作队列中拉取(Pull)数据**
- **将新工作推送(Push)到 T2工作队列**
- **当本地工作队列为空时，从另一个工作队列中窃取（Steal）工作**

示例

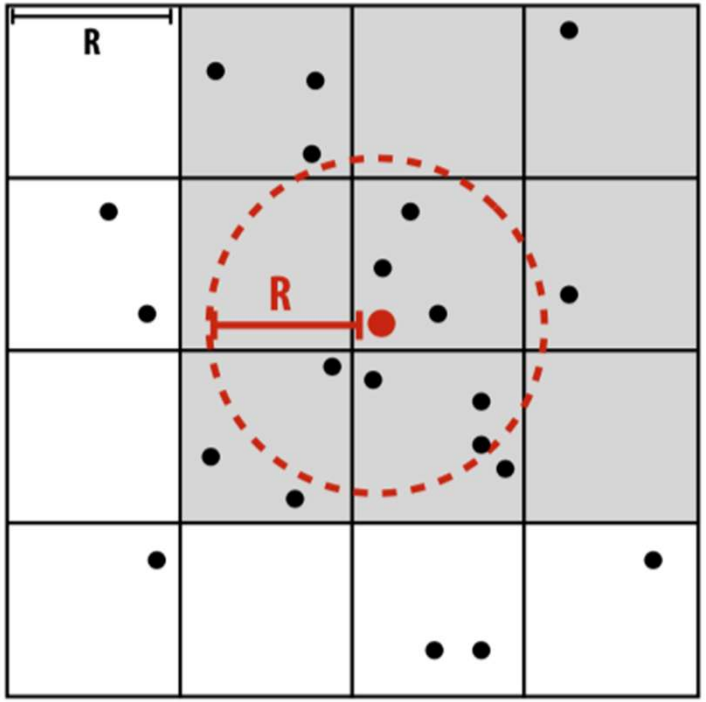
- 在大型并行机上创建粒子数据结构网格
- 将 1M 个点粒子放置在基于 2D 位置的 16 单元格均匀网格中
  - 在 GPU 上构建二维列表数组



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

这种结构的常见用法

- 一个常见的操作是计算与相邻粒子的相互作用力
- 示例：给定粒子，找到半径 R 内的所有粒子
  - 使用大小为 R 的单元格创建网格
  - 只需要检查周围网格单元中的粒子



解决方案 1：并行化单元格（并行度有限）

- 一种可能的答案是按单元分解工作：对于每个单元，独立计算其中的粒子（消除争用，因为不需要同步）
  - 并行性不足：**只有 16 个**并行任务，但需要数千个独立任务才能有效利用 GPU

```
list cell_lists[16];        // 2D array of lists

for each cell c             // in parallel
   for each particle p      // sequentially
      if (p is within c)
         append p to cell_lists[c]
```

解决方案 2：并行化粒子（竞争抢占频繁）

- 为每个 CUDA 线程分配一个粒子。线程计算完粒子后，更新到所在单元格的参数列表
  - 大规模争用：数千个线程竞争更新单个共享数据结构的访问权限
  - 需要加锁，避免一致性问题

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock;

for each particle p              // in parallel
   c = compute cell containing p
   lock(cell_list_lock)
   append p to cell_list[c]
   unlock(cell_list_lock)
```

解决方案3：使用更细粒度的锁

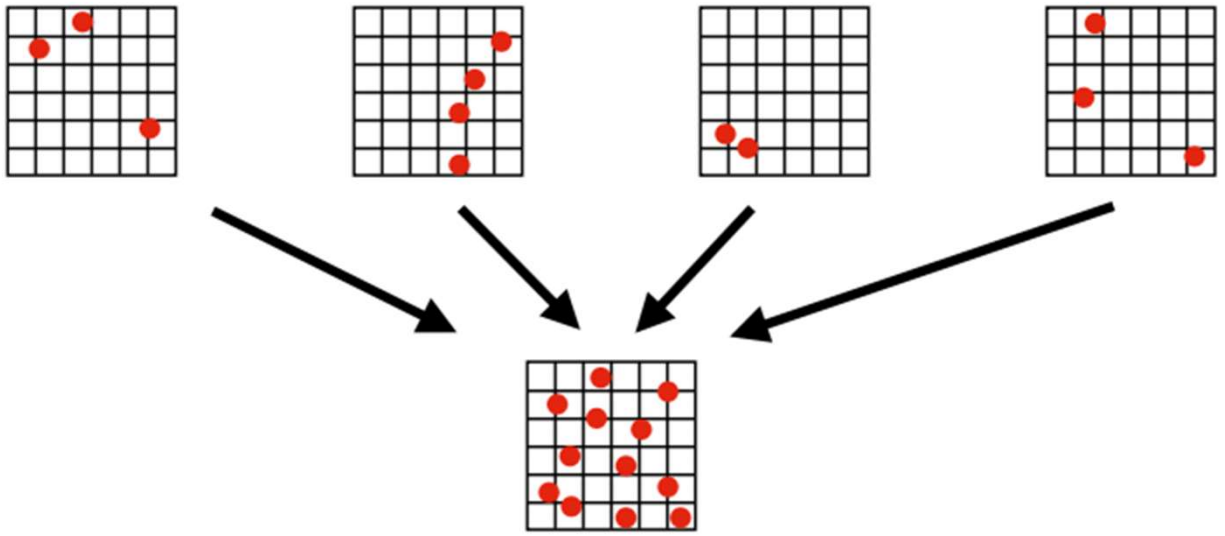
- 通过使用每个单元锁（per-cell locks）来缓解对单个全局锁（single global lock）的争用
  - 只锁一部分（一个单元格内的节点结果）
- 假设粒子在 2D 空间中均匀分布……比解决方案 2 少 16 倍的竞争

```
list cell_list[16];     // 2D array of lists
lock cell_list_lock[16];

for each particle p              // in parallel
   c = compute cell containing p
   lock(cell_list_lock[c])
   append p to cell_list[c]
   unlock(cell_list_lock[c])
```

解决方案4：计算部分结果+合并

- 另一个答案：并行生成 N 个“部分”网格，然后合并
  - 示例：创建 N 个线程块（可以参照核数来设置）
  - 线程块中的所有线程更新同一个网格
    - GOOD：实现更快的同步：竞争减少了 N 倍，同步成本也更低，因为它是在块局部变量上执行的（在 CUDA 共享内存中）
  - BAD：需要额外的工作：在计算结束时合并 N 个网格
  - BAD：需要额外的内存占用：存储 N 个列表网格，而不是 1 个



小结-竞争

- 共享资源的存在就可能引发竞争
- 对负载平衡等的优化可能引入新的竞争
- 局部性优化的目标之一也是要减少竞争
- 减少竞争的方法
  - 减少对竞争资源的访问量：复制竞争资源（例如，本地副本、细粒度锁）
  - 错开对争用资源的集中访问