

L04-2

并行编程基础

—局部性、通信和竞争

《并行处理》

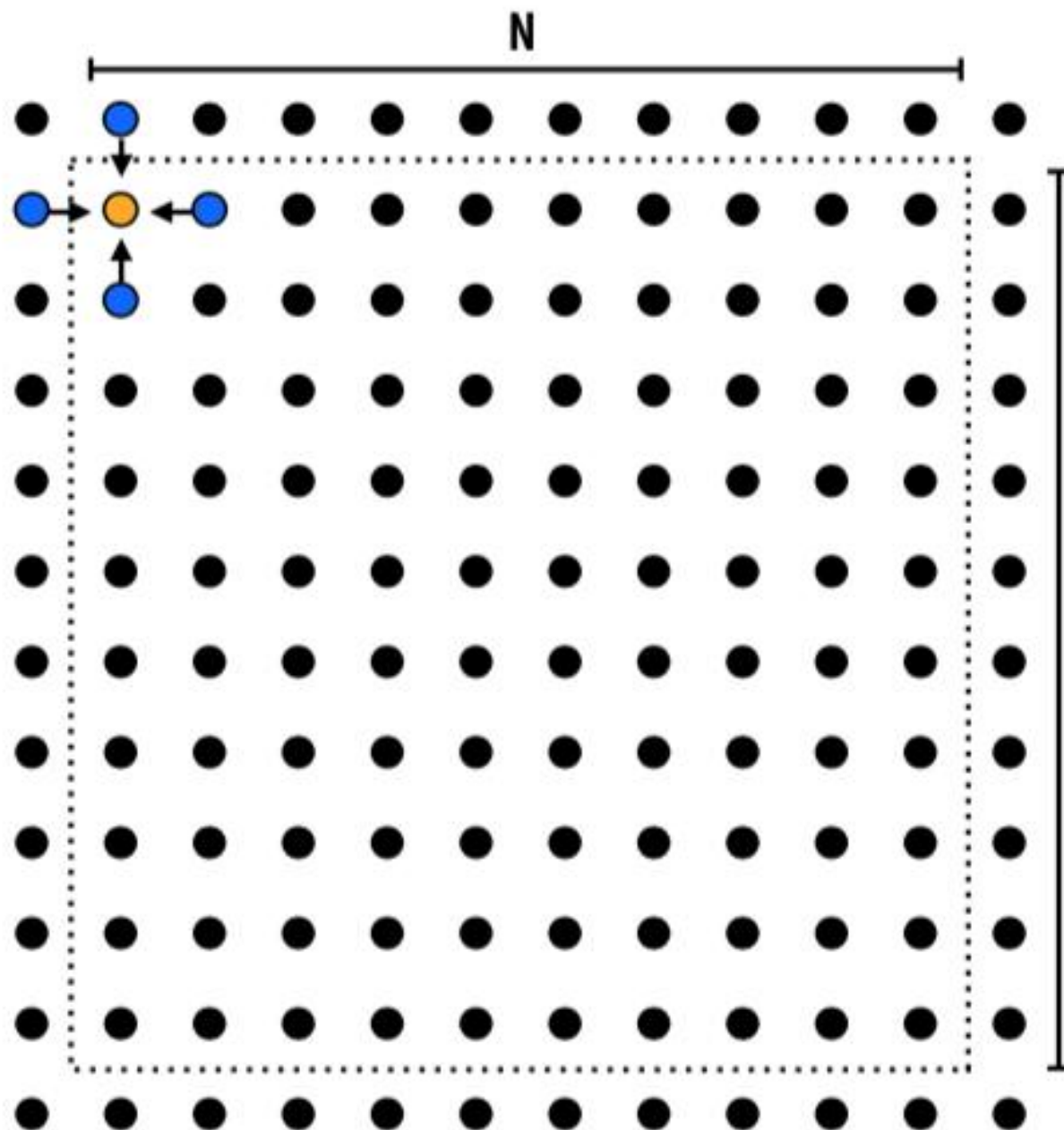
邵恩
高性能计算机研究中心

今天：越来越多的并行程序需要被优化

- 上一课：为每个工作人员（独立的计算部件：threads、processors等）分配工作的调度策略
 - 目标：实现良好的工作负载平衡，同时最大限度地减少开销
 - tradeoffs：讨论静态和动态工作分配之间的权衡
 - 提示：保持简单（实施、分析，然后根据需要调整/优化）
- 今天：最小化通信与交互开销的优化技巧（minimizing communication costs）

回顾一下：用消息传递（Message passing）实现求解器的例子

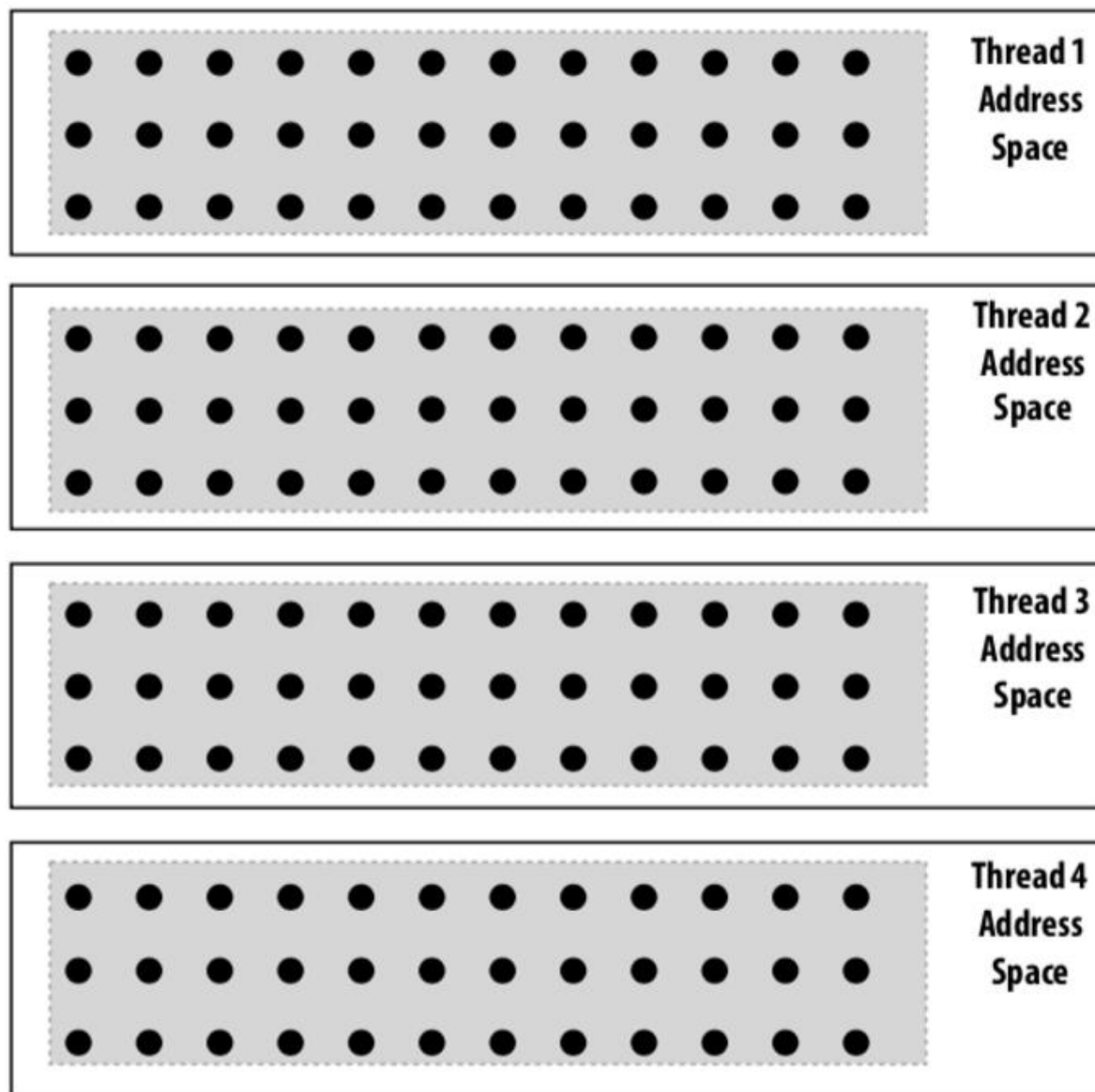
- 通信的特征变得十分明确，每个点会跟上下左右紧邻的四个点进行通信与交互



$$A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j]);$$

消息传递模型 (Message passing model)

- 每个线程都在自己的地址空间中运行和处理数据



- 图中：共有四个线程
- 网格数据被分成四个独立数据分组
- 每个分组驻留在四个唯一的线程地址空间之一（四个每线程私有数组）

引发不同地址空间之间的**数据拷贝**

例子：

红色处理单元完成计算任务后，线程1和线程3发送一行数据给线程2（线程2需要最新的红处理单元的结果信息，来更新下一阶段的黑处理单元）

“幽灵单元” (Ghost cells)：是从远程地址空间复制的网格单元。通常说幽灵单元中的信息被其他线程“拥有”。

Thread 2 logic:

```
float* local_data = allocate(N+2, rows_per_thread+2);
```

```
int tid = get_thread_id();
```

```
int bytes = sizeof(float) * (N+2);
```

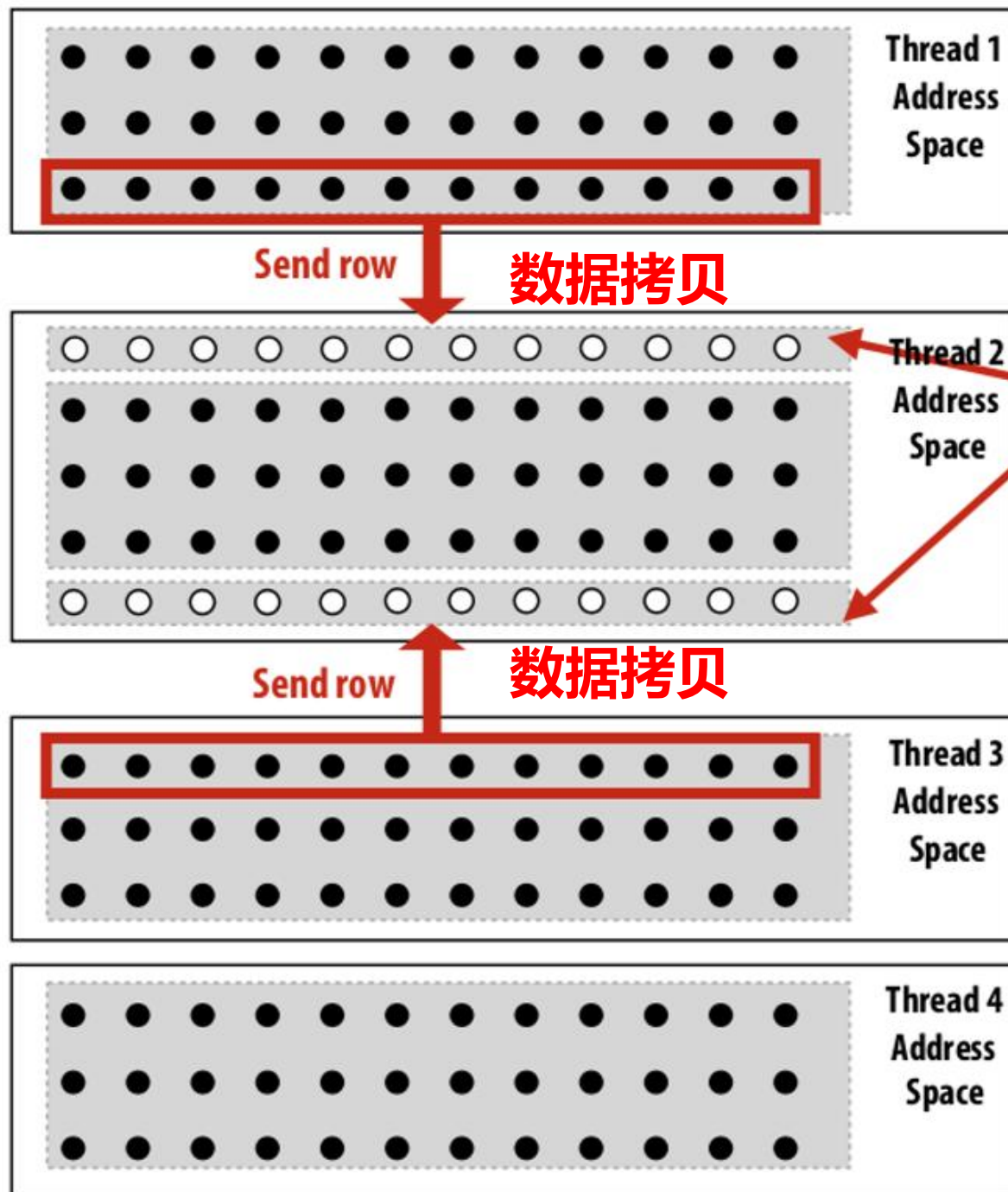
```
// receive ghost row cells (white dots)
```

```
recv(&local_data[0,0], bytes, tid-1);
```

```
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);
```

```
// Thread 2 now has data necessary to perform
```

```
// future computation
```



消息传递模型 (Message passing model)

Message passing solver

类似于共享地址空间的求解器的代码结构，但现在**通信**在消息发送和接收中是在代码中**显式指定的**

向“邻居线程”发送和接收幽灵单元
(send,recv)

执行计算
(就像在共享地址空间版本的求解器中一样)

所有线程将本地my_diff发送到线程 0

线程0计算全局diff，在全部计算结束后，将结果发送回所有其他线程

```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids

////////////////////////////////////

void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid != 0)
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            send(&localA[rows_per_thread,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

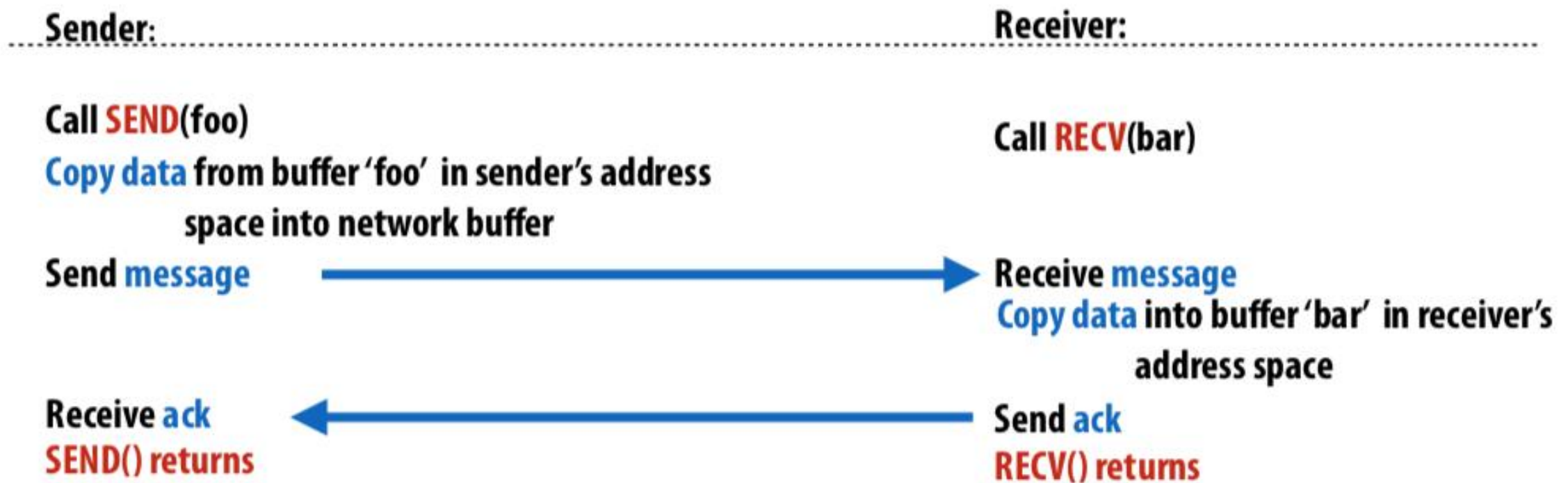
        if (tid != 0)
            recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            recv(&localA[rows_per_thread+1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        for (int i=1; i<rows_per_thread+1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                   localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            for (int i=1; i<get_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSG_ID_DONE);
        }
    }
}
```

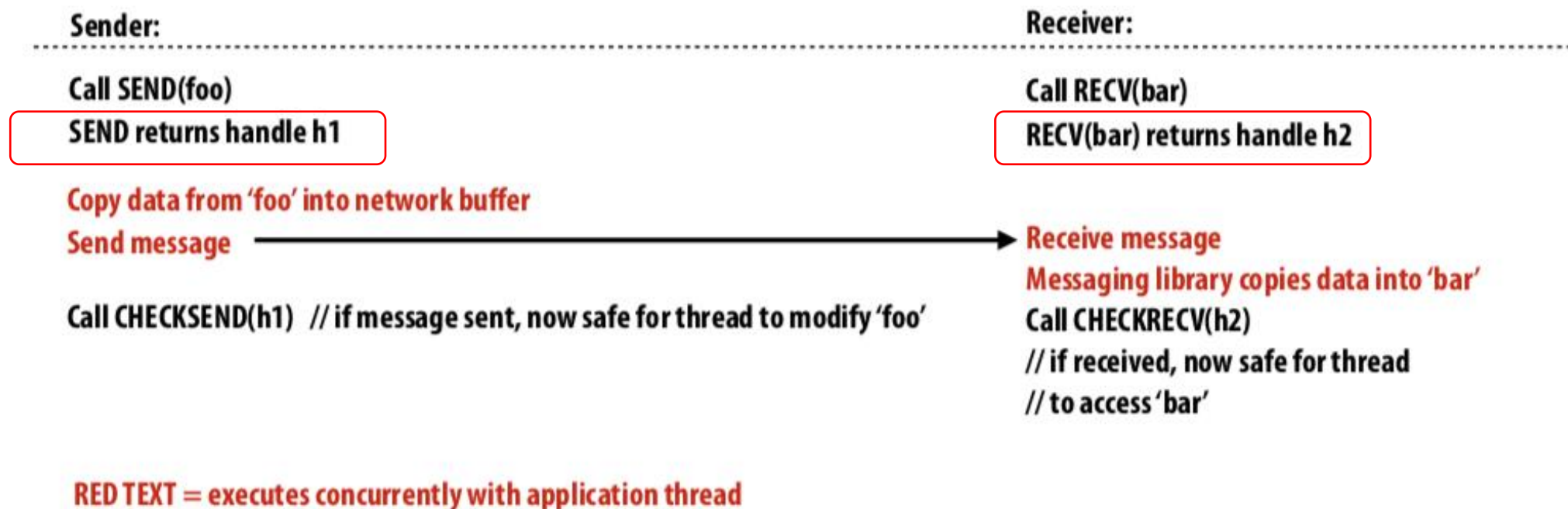
同步（阻塞）式send & receive

- `send()`: 有数据发送方发起，调用返回的条件有一个：1.发送方接收到来自接收方的ack，确认接收方的数据成功接收。
- `recv()`: 由数据接收方发起，调用返回的条件有两个：1.接收到的数据被拷贝至接收方的缓存；2.发送完成ack至发送这。

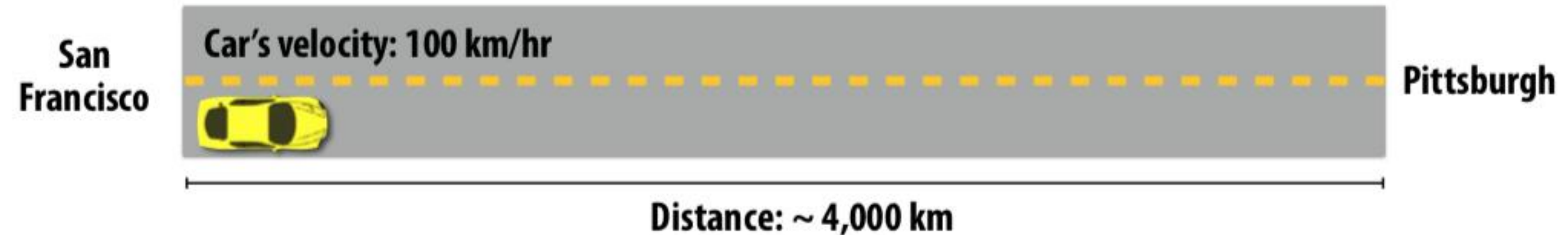


异步（非阻塞）式send & receive

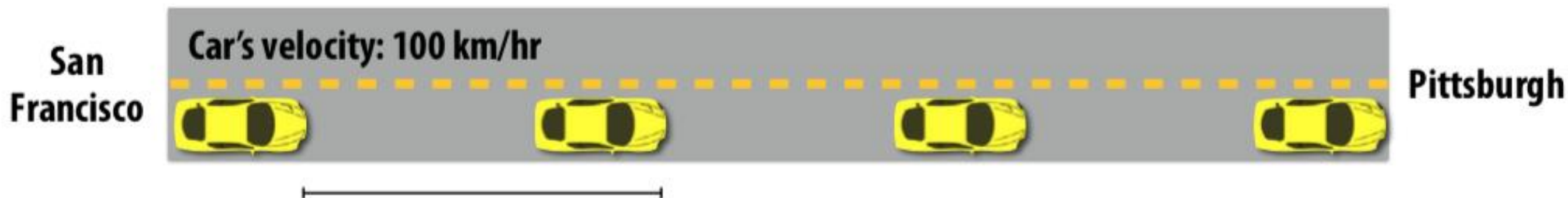
- send(): 所有的调用返回（return）立即会被执行
 - 提供给 send() 的缓冲区不能通过调用线程修改，因为消息处理与线程执行同时发生
 - 调用线程可以在等待消息发送的同时执行其他工作
- recv(): 发布未来接收的意图，立即返回
 - 使用 checksend(), checkrecv() 来确定发送/接收的实际状态
 - 调用线程可以在等待接收消息的同时执行其他工作



延迟 vs. 吞吐量



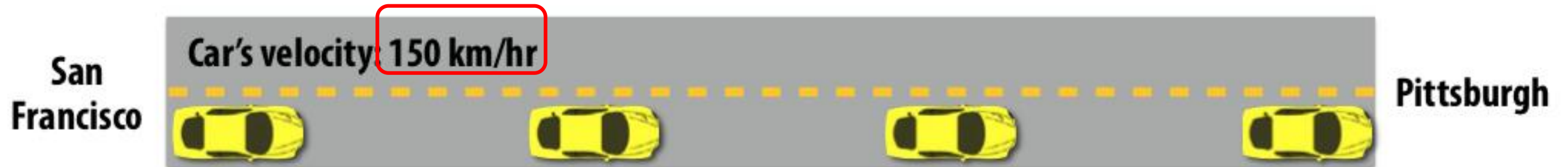
将一个人从旧金山移动到匹兹堡的延迟：40 小时
即：单独完成一件工作需要的时间



每辆汽车在高速公路上相距 1 公里

吞吐量：每小时运送**100**人次（每1/100个小时1辆车）
即：单位时间完成的**任务数量**

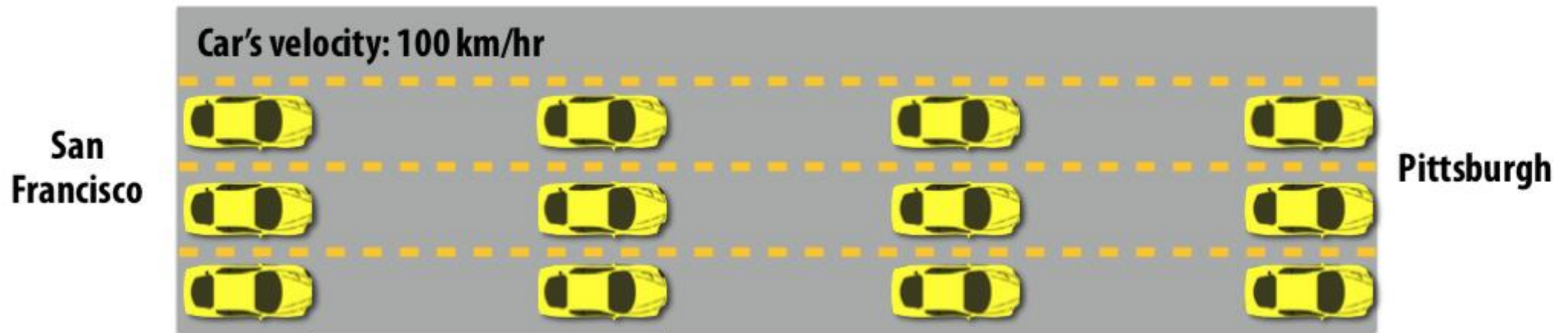
提高吞吐量



每辆汽车在高速公路上相距 1 公里

方法一：开快点！

吞吐量 **150** 人/小时（每 1/150 小时 1 辆车）



每辆汽车在高速公路上相距 1 公里

方法二：多建车道！

吞吐量：每小时**300**人（每1/100小时3辆）

Review: 延迟 vs. 吞吐量

- 延迟
 - 完成程序操作所需的时间量
 - 缓存开销：错过缓存 (misses cache) 的内存加载，需要消耗有 200 个周期的延迟
 - 通信开销：一个数据包从本地计算机发送到 Google 需要 20 毫秒
- 吞吐量
 - 执行操作的速率，即：单位时间内完成的操作数量
 - 内存可以 25GB/s 的速度向处理器提供数据
 - 一条通信链路每秒可以发送 1000 万条消息

如果一次只能有一辆车在高速公路上怎么办

- 当高速公路上的汽车到达匹兹堡时，下一辆车才能离开旧金山



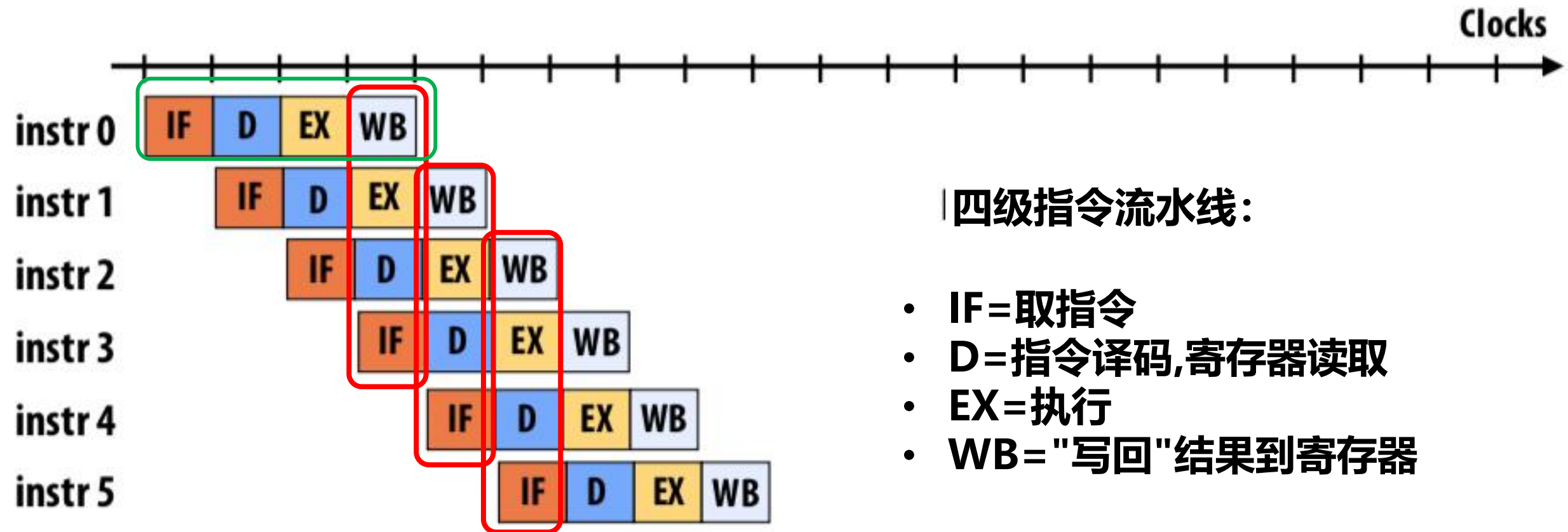
将一个人从旧金山转移到匹兹堡的延迟：40 小时

$$\begin{aligned} \text{吞吐量} &= 1/\text{延迟} \\ &= 1/40 \text{人每小时 (每40小时1辆车)} \end{aligned}$$

流水线

将每条指令的执行分解为几个较小的步骤

启用更高的时钟频率（每个时钟流水线的每个部分只完成一个简单的、短的操作）



延迟：1条指令需要4个时钟周期

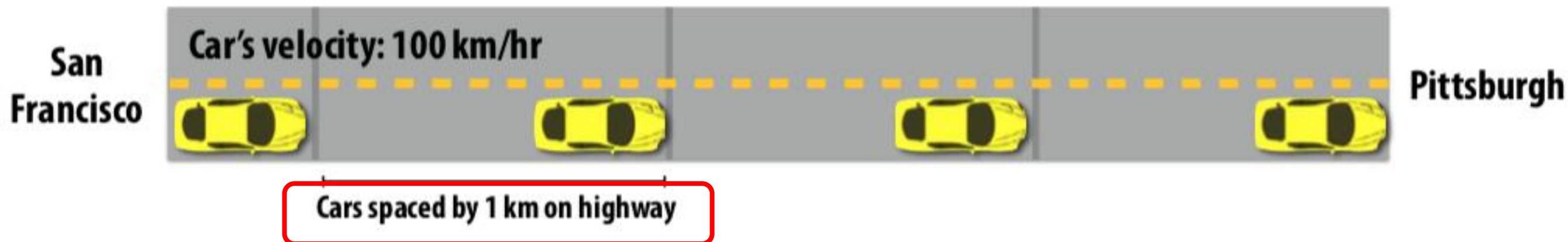
吞吐量：每个时钟周期 1 条指令，单位时间内（一个时钟周期内）完成一条指令（是的，当背靠背指令相互依赖时，必须注意确保程序的正确性。）

Intel Core i7 流水线是可变长度的（取决于指令）~15-20 阶段

类比开车去匹兹堡的例子

从旧金山开车到匹兹堡的任务被分解成不同的汽车可以并行处理的更小的子问题

(上图：两车间距 1 公里，下图：两车间距 500 米)



吞吐量=每小时**100**人 (每1/100小时1辆车)



吞吐量=每小时**200**人 (每1/100小时1辆车)

先简化一下：非流水线通信的简单模型

示例：发送 n 位消息

$$T(n) = T_o + \frac{n}{B}$$

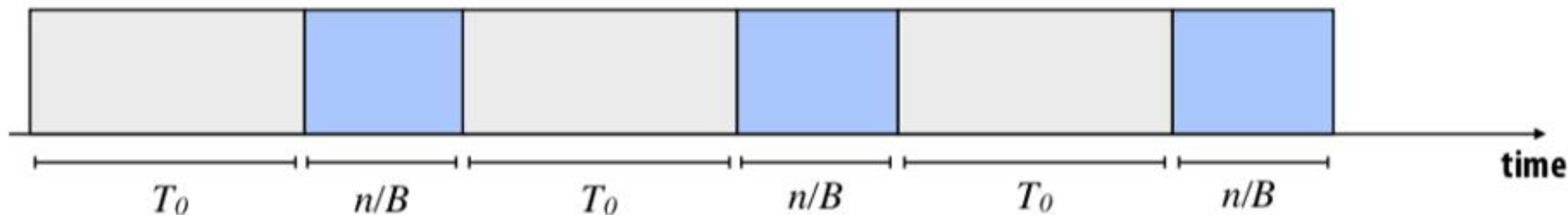
- $T(n)$ =**传输时间**（操作的整体延迟）
- T_o =**启动延迟**（例如，第一位到达目的地的时间）
- n =**操作中传输的字节数**
- B =**传输速率**（链路**带宽**）

如果处理器仅在上一条消息发送完成后才发送下一条消息.....

“有效带宽” $= n/T(n)$



有效带宽取决于**操作中传输的字节数**（一次**操作中传输大量的字节数**分摊，来分摊一次的启动延迟）



更通用的模型




示例：发送 n 位消息

总通信时间 = 开销 + 瓶颈链路占用率 + 网络延迟



通过链接 2（快速链接）发送数据： $T_o + n/B_{large}$
将消息复制到接收方节点的缓冲区：

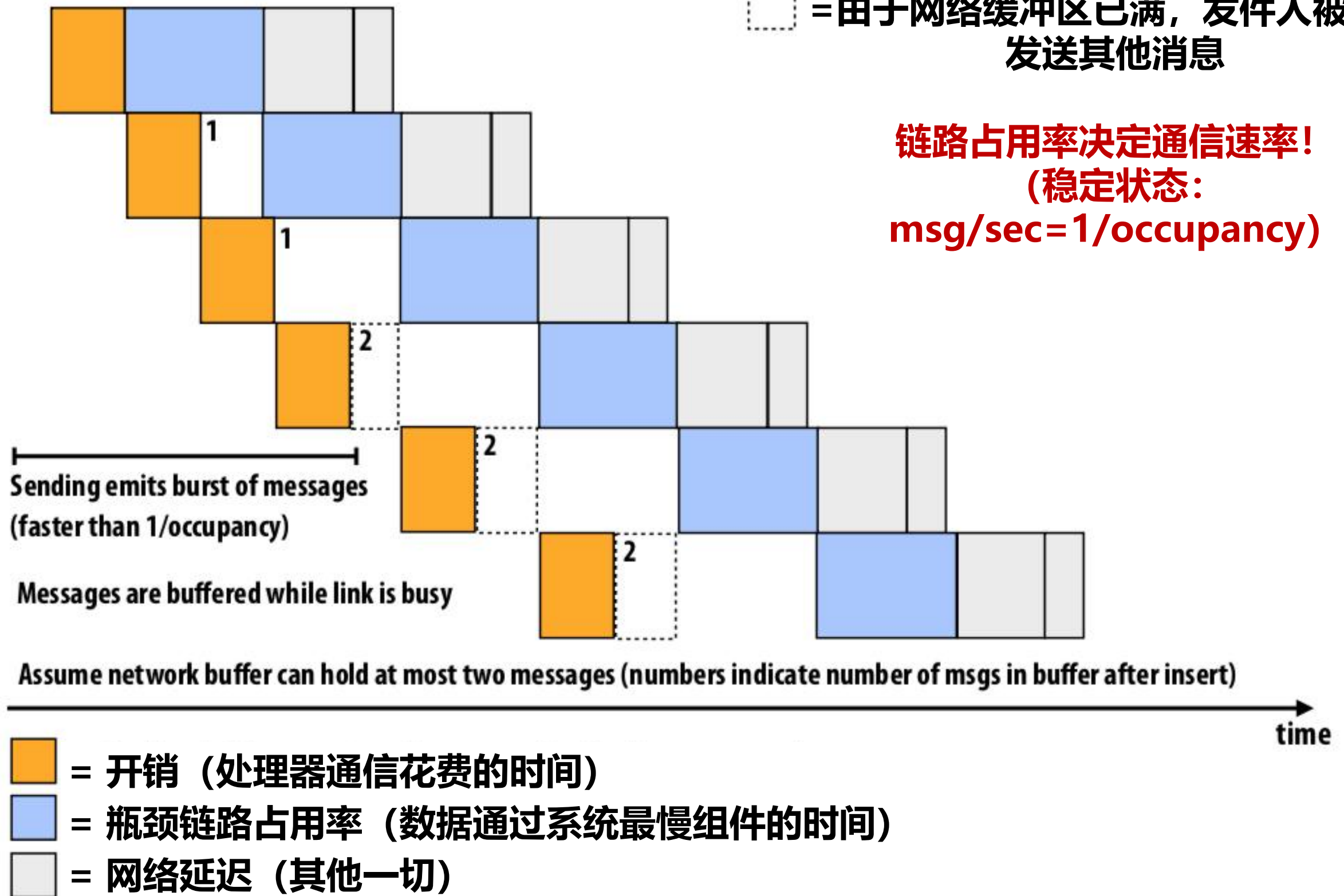


-  = 开销（处理器通信花费的时间）
-  = 瓶颈链路占用率（数据通过系统最慢组件的时间）
-  = 网络延迟（其他一切）

流水线通信

 = 由于网络缓冲区已满，发件人被阻止发送其他消息

链路占用率决定通信速率！
(稳定状态：
 $\text{msg/sec} = 1/\text{occupancy}$)



其他开销(Cost)

- 操作对程序执行时间的影响（或其他一些指标，例如，消耗的能量.....）

总通信时间 = 开销 + 瓶颈链路占用率 + 网络延迟

总通信开销 = 通信时间 - 重叠时间 (overlap)

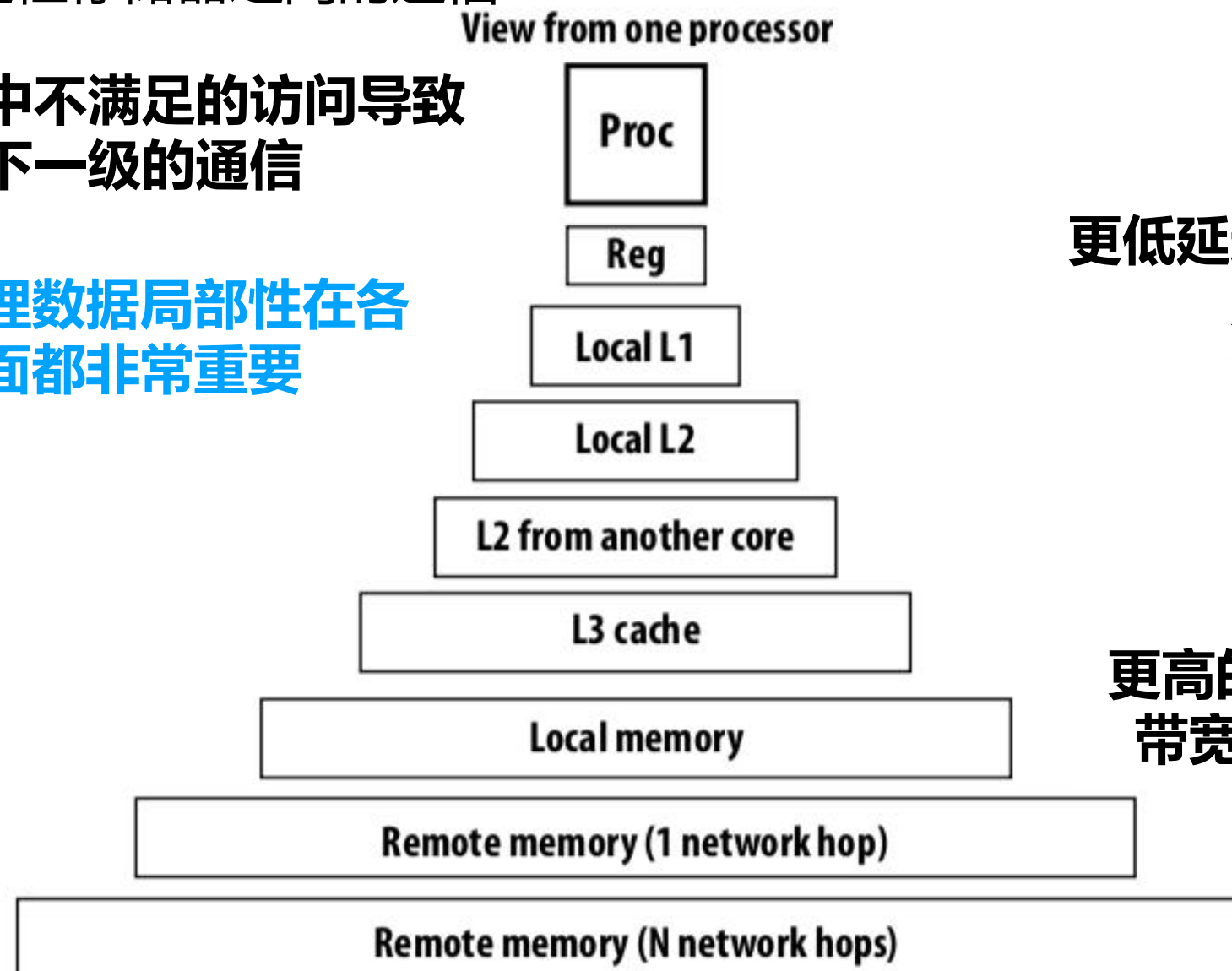
- **重叠时间 (overlap) : 与“其他工作”或操作同时执行的通信部分**
- **“其他工作”可以是执行计算的指令，也可以是其他的通信操作**

扩展的内存层次结构

- 让我们更加笼统地思考 “通信”
 - 处理器与其高速缓存之间的通信
 - 处理器和本地内存之间的通信
 - 处理器和远程存储器之间的通信

本地内存中不满足的访问导致
与下一级的通信

因此，管理数据局部性在各
个层面都非常重要



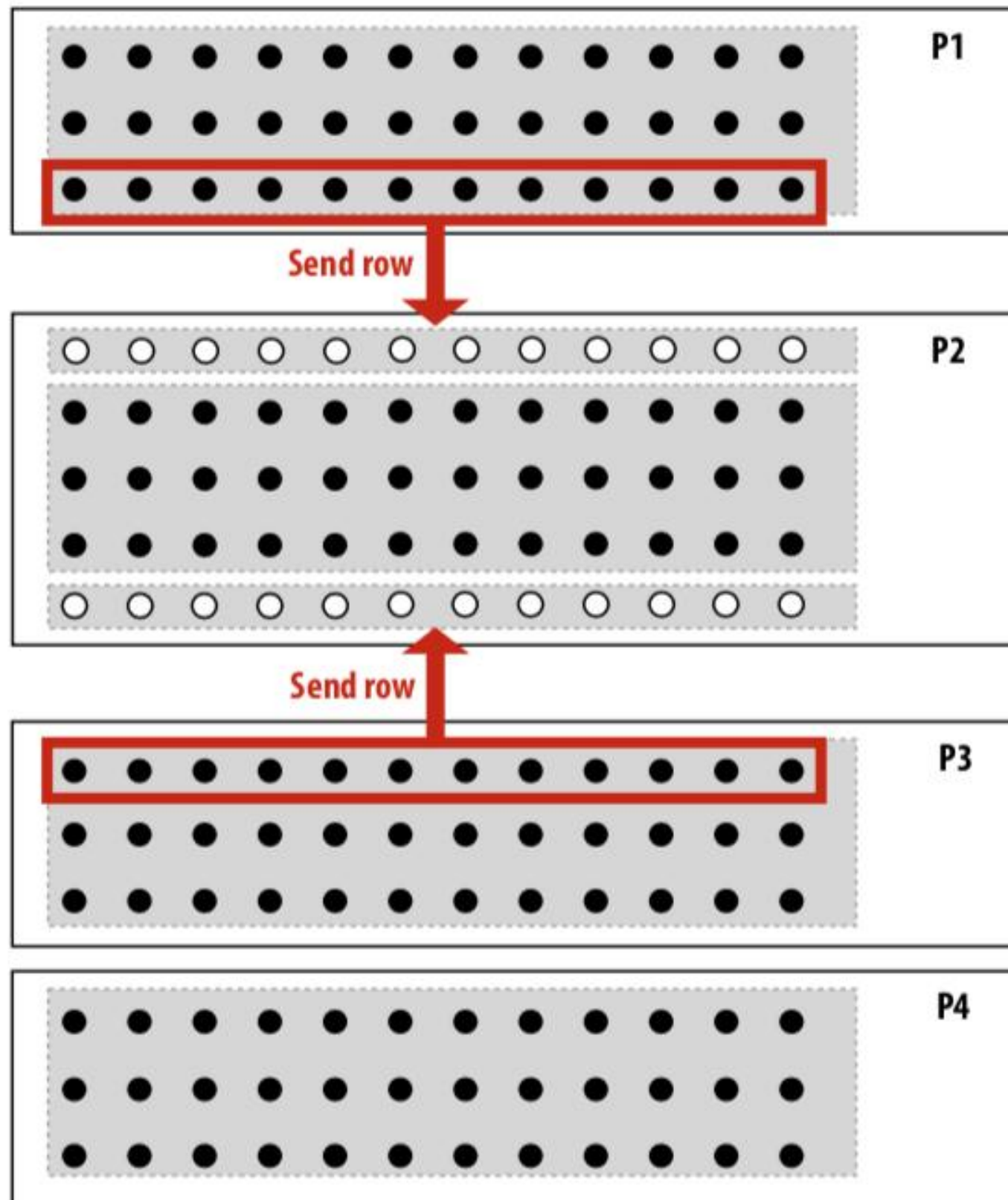
更低延迟、更高带宽、
更小容量

更高的延迟，更低的
带宽，更大的容量

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

通信与计算的比率

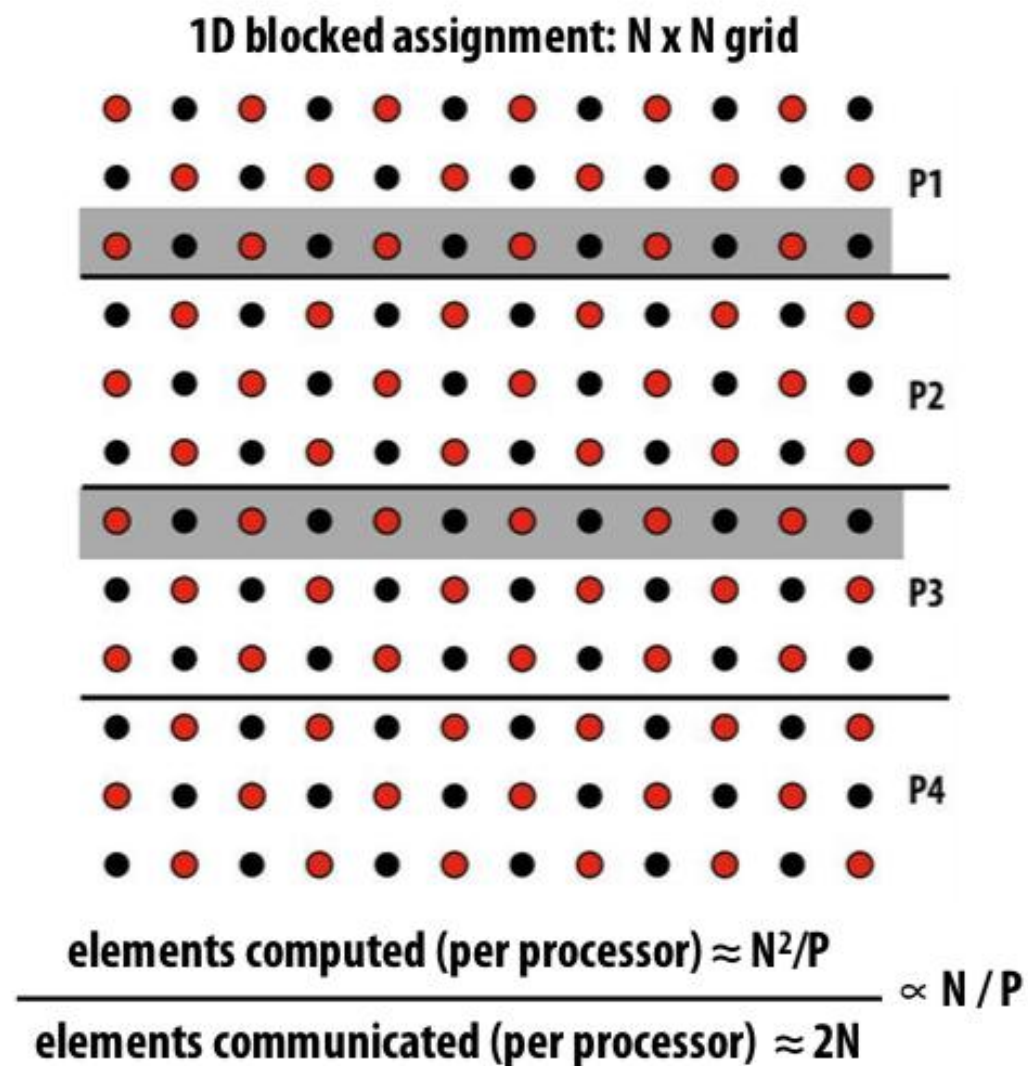
amount of communication (e.g., bytes)

amount of computation (e.g., instructions)

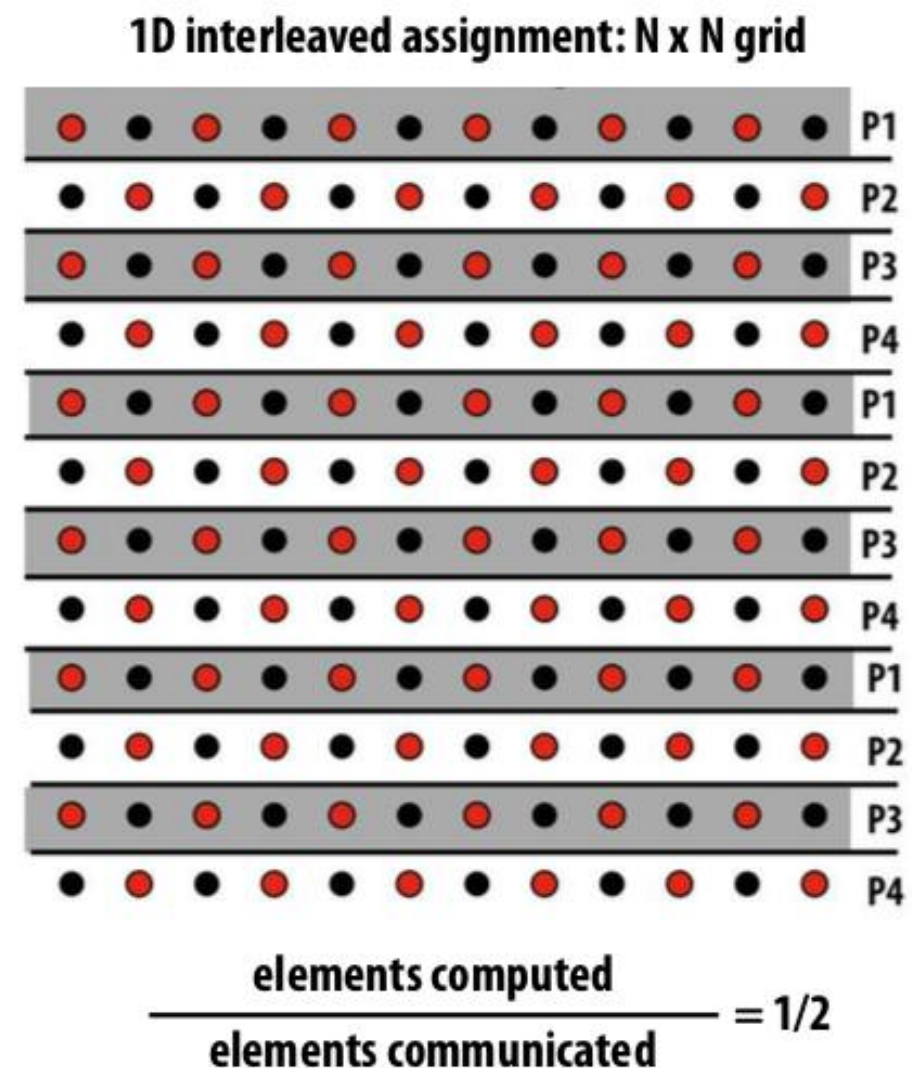
- 如果分母是计算的执行时间，通信与计算的比率给出了代码的平均（访存、通信）带宽需求
- 算术强度 = $1 / \text{通信与计算的比率}$
- 由于计算能力与可用带宽的比率很高，因此需要高算术强度才能有效地利用现代并行处理器
- 提高计算比重，降低通信比重

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（增加算术强度）

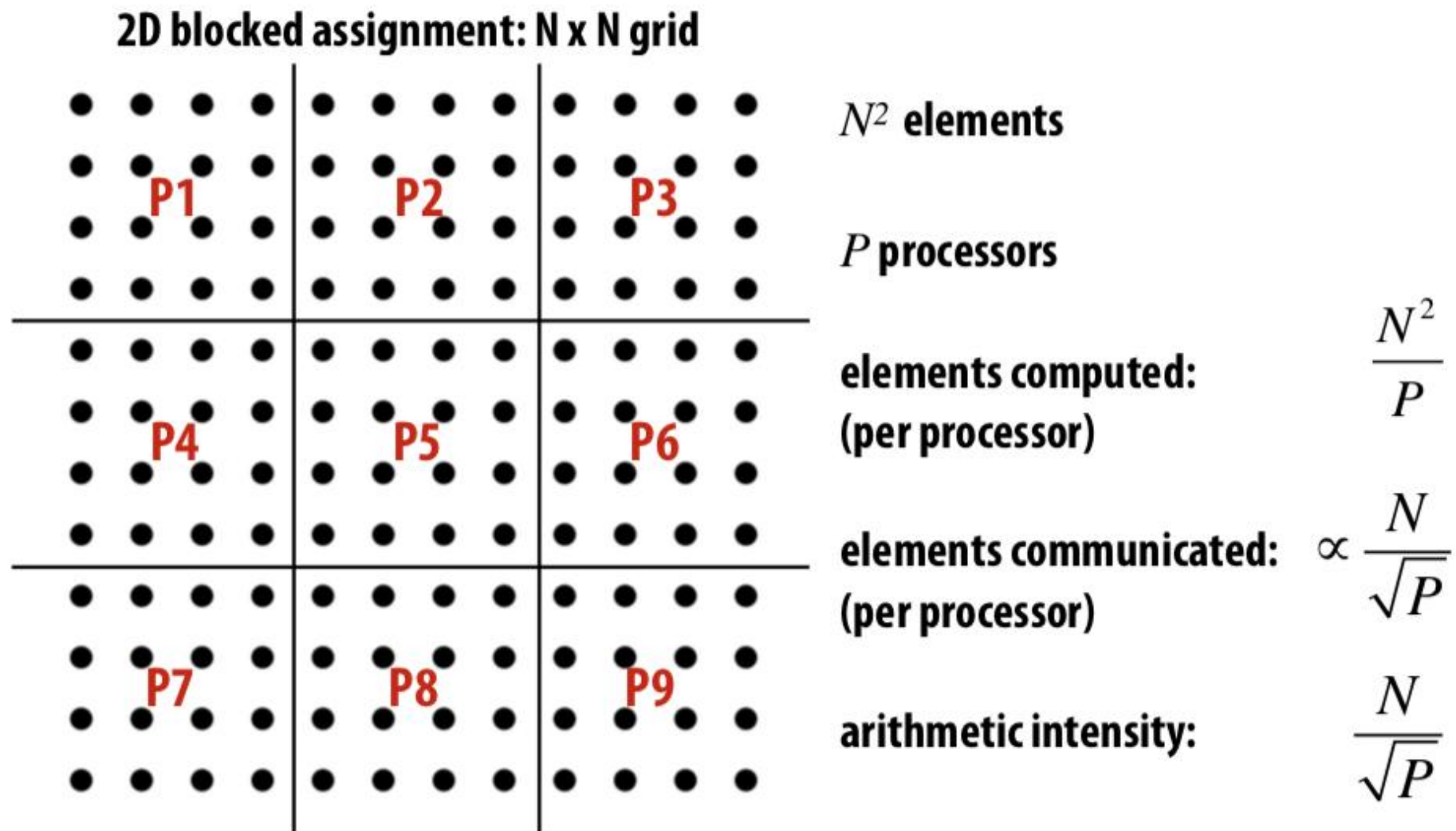


GOOD: 算术强度约为 N/P



BAD: 算术强度 $= 1/2$

减少固有的通信



- 看似更好：比一维阻塞分配渐近更好的通信缩放
- 但是：二维处理器划分策略的通信成本，随 P 呈次线性增长

原因二：人为原因引发的通信

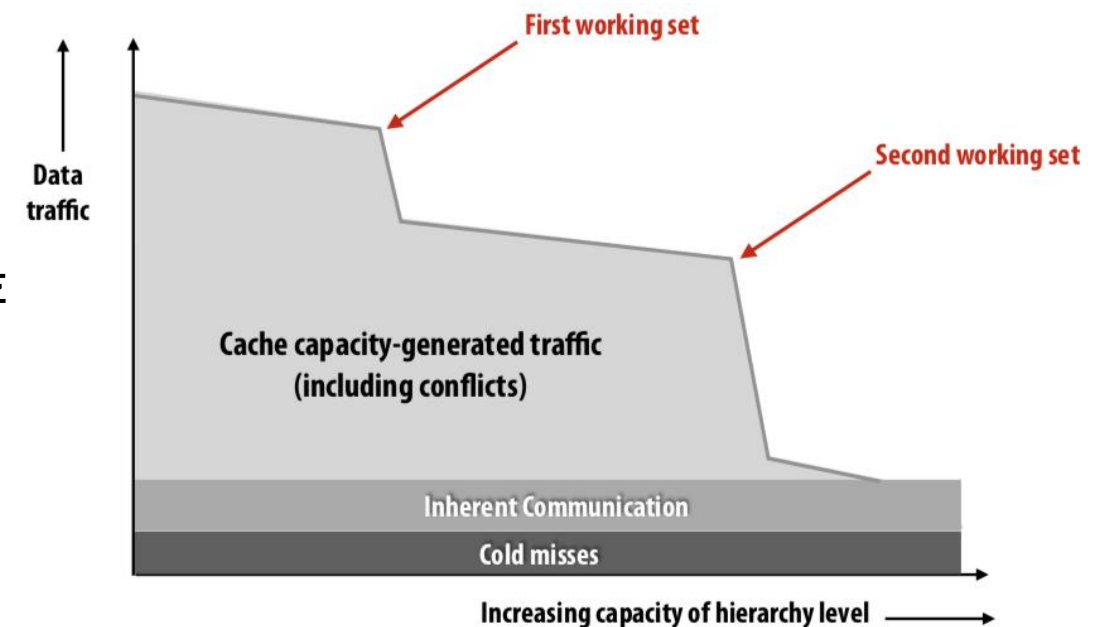
- 固有通信：在给定计算和缓存资源（假设无限容量缓存、最小粒度传输等）的情况下，从根本上**必须**在处理器之间完成的通信操作（不完成就无法执行算法）
- 人为通信：所有其他通信（人为通信源于系统实施的实际细节），也就是**非必须的、业务之外的**通信

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**)
 - 程序本来只需要存储 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**
- 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**）
- **数据复制拷贝能力**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访问之间保留它

四个Cs (Cache miss) 模型

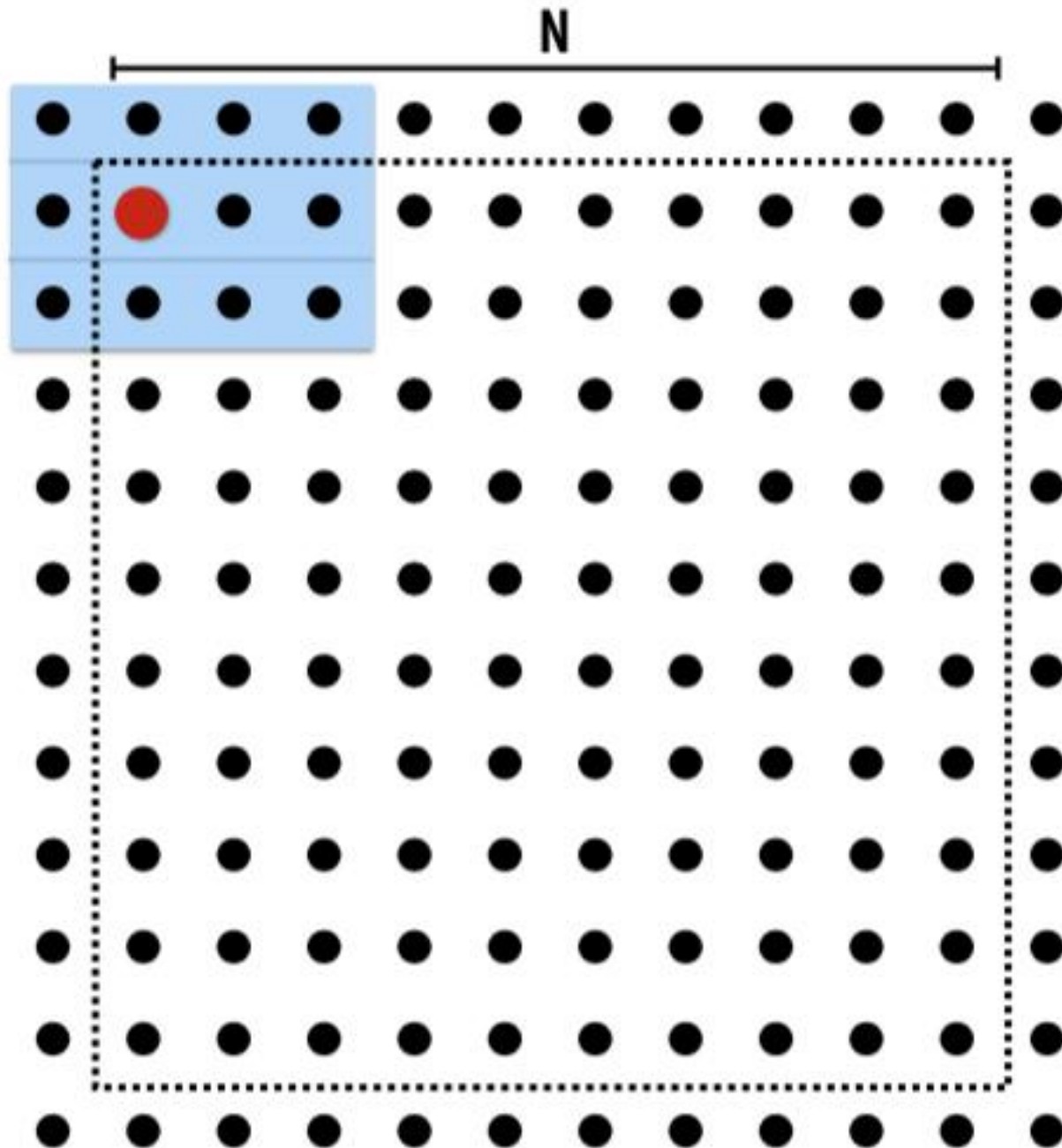
- Cold miss: 第一次访问新数据, 导致缓存未命中
 - 第一次接触新数据, 导致程序的串行执行
- Capacity miss: 缓存容量不足, 导致缓存未命中
 - 需要缓存的工作数据集大于缓存容量。可以通过增加缓存大小来避免/减少
- Conflict miss: 冲突, 导致缓存未命中
 - 由缓存管理策略引起的未命中。可以通过更改缓存关联性或应用程序中的数据访问模式来避免/减少
- Communication miss (new): 通信, 导致导致缓存未命中
 - 由于并行系统中固有的, 或人为的通信



优化思路：减少通信的一些技巧

网格求解器中的数据访问

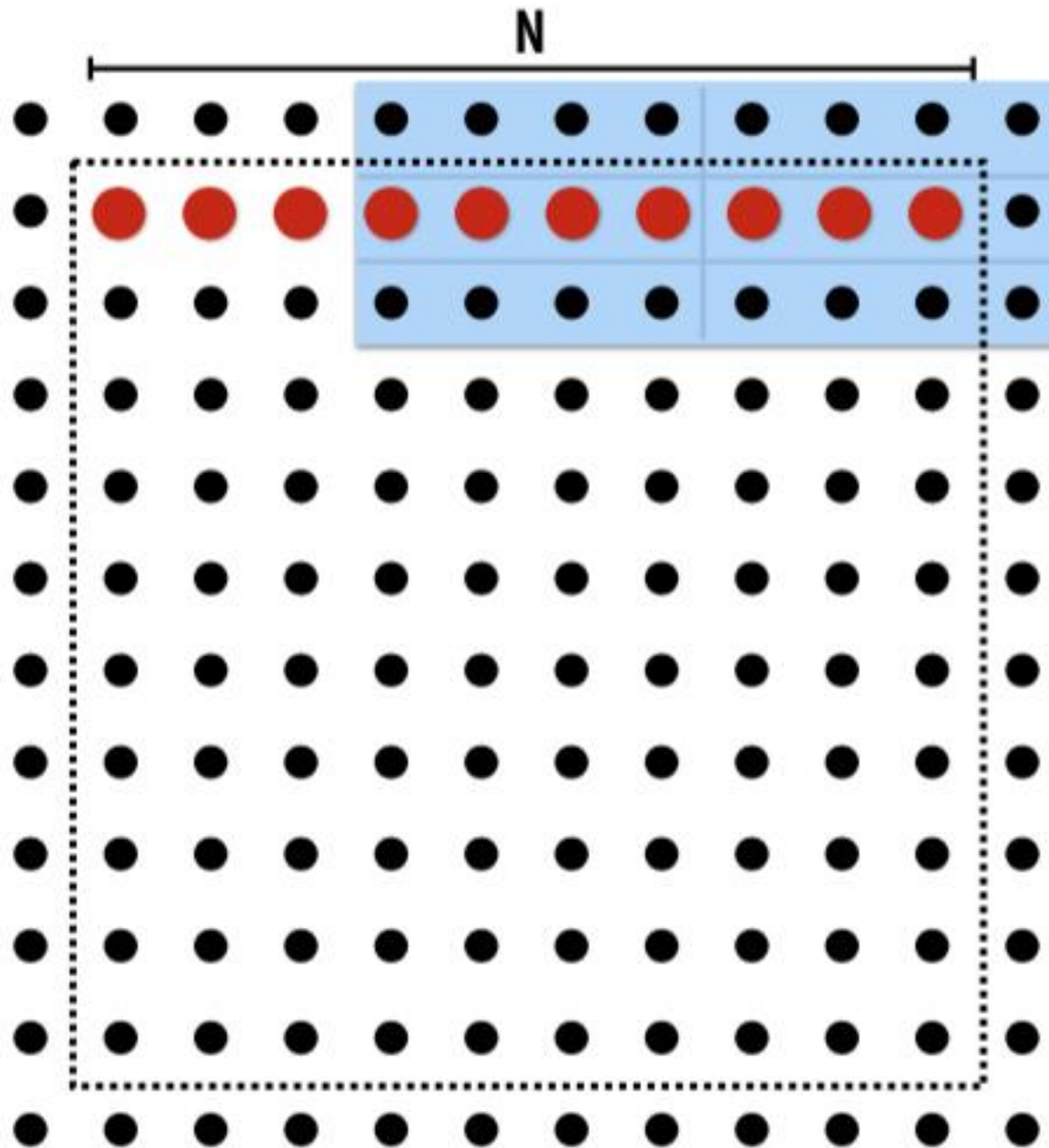
- 行优先遍历



- 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

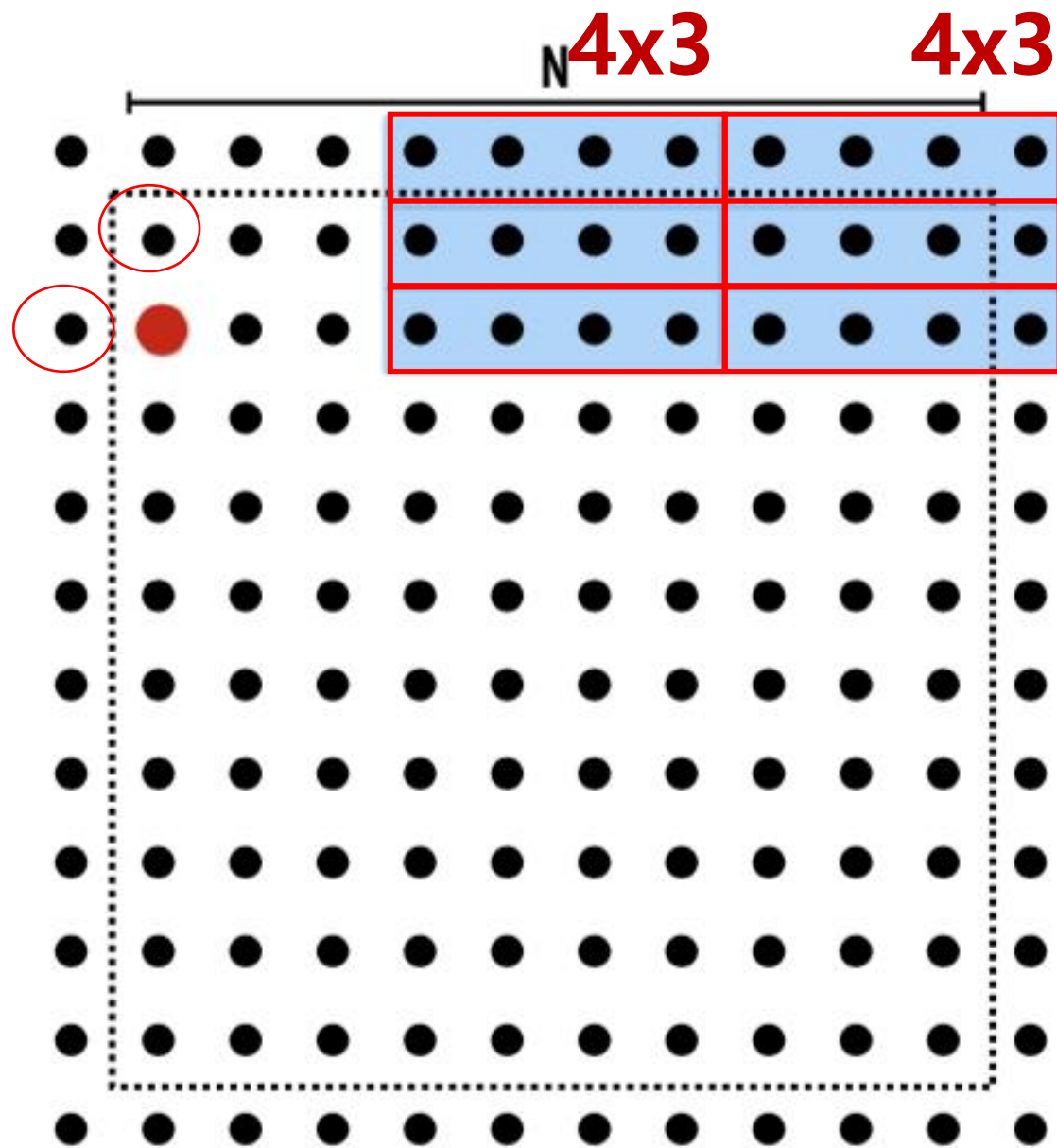
- 行优先遍历



- 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)
- 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];  
}
```

2次加载，每个数学运算1次存储
(运算强度=1/3)

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

2次加载，每个数学运算1次存储
(运算强度=1/3)

```
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);
```

整体算术强度=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
(算术强度=3/5)

顶部的代码更加模块化（例如，基于数组的数学库，如 Python 中的 numpy）
底部的代码执行的性能更好。（**算术强度**）

通过共享数据提高**算术强度** (arithmetic intensity)

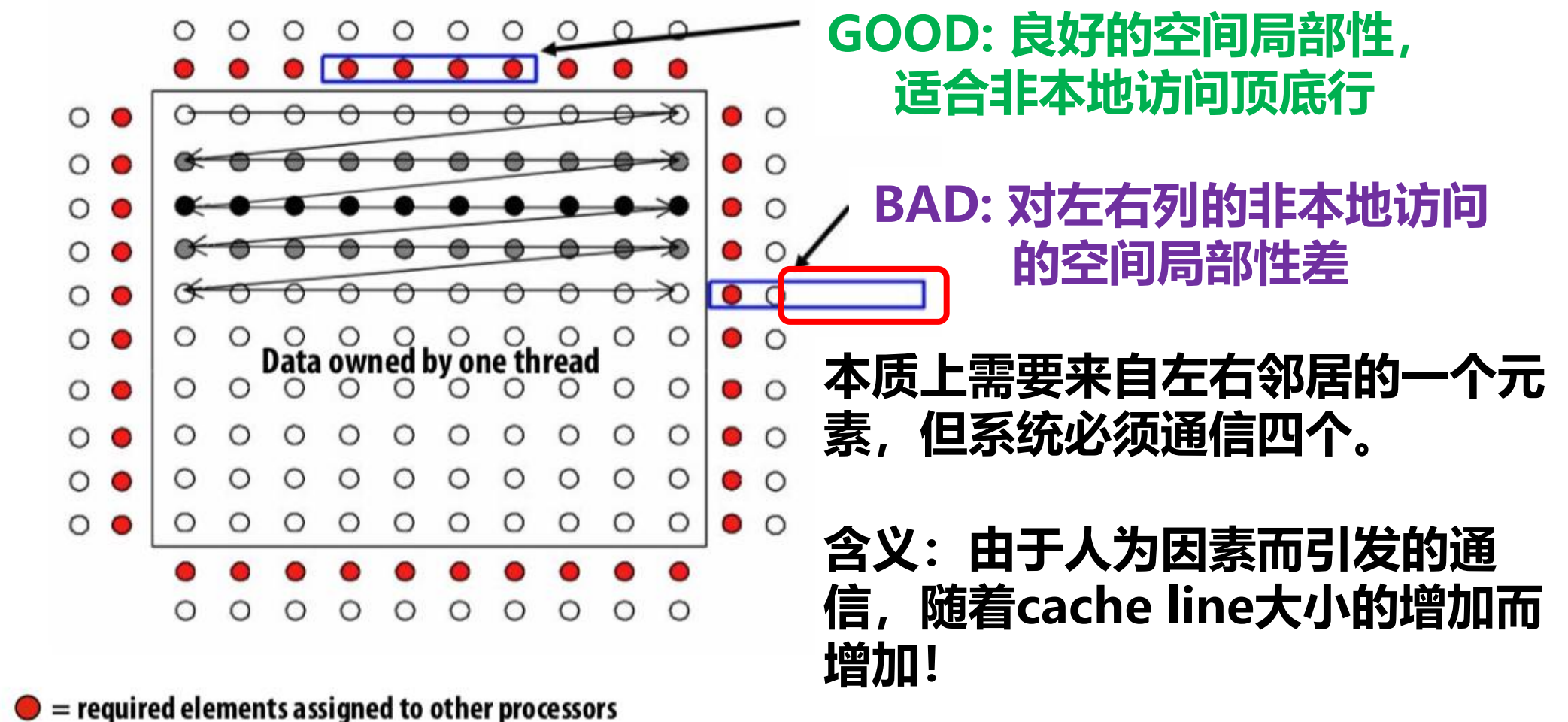
- **利用共享：共同对相同位置的数据，进行操作的多个本地任务 (co-locate tasks)**
 - 安排多个线程，在同一处理器上，同时处理同一数据结构
 - 减少固有的沟通
- 示例：CUDA 线程块 (CUDA thread block)
 - 一种抽象：在 CUDA 程序中，以本地化的方式（不跨处理器），处理相关操作
 - 线程块中的线程，经常协作执行同一种计算操作 (SIMT, 单指令多线程)
 - 所以 GPU 实现总是在同一个 GPU 核心上，调度来自同一个线程块的多个线程

利用空间局部性

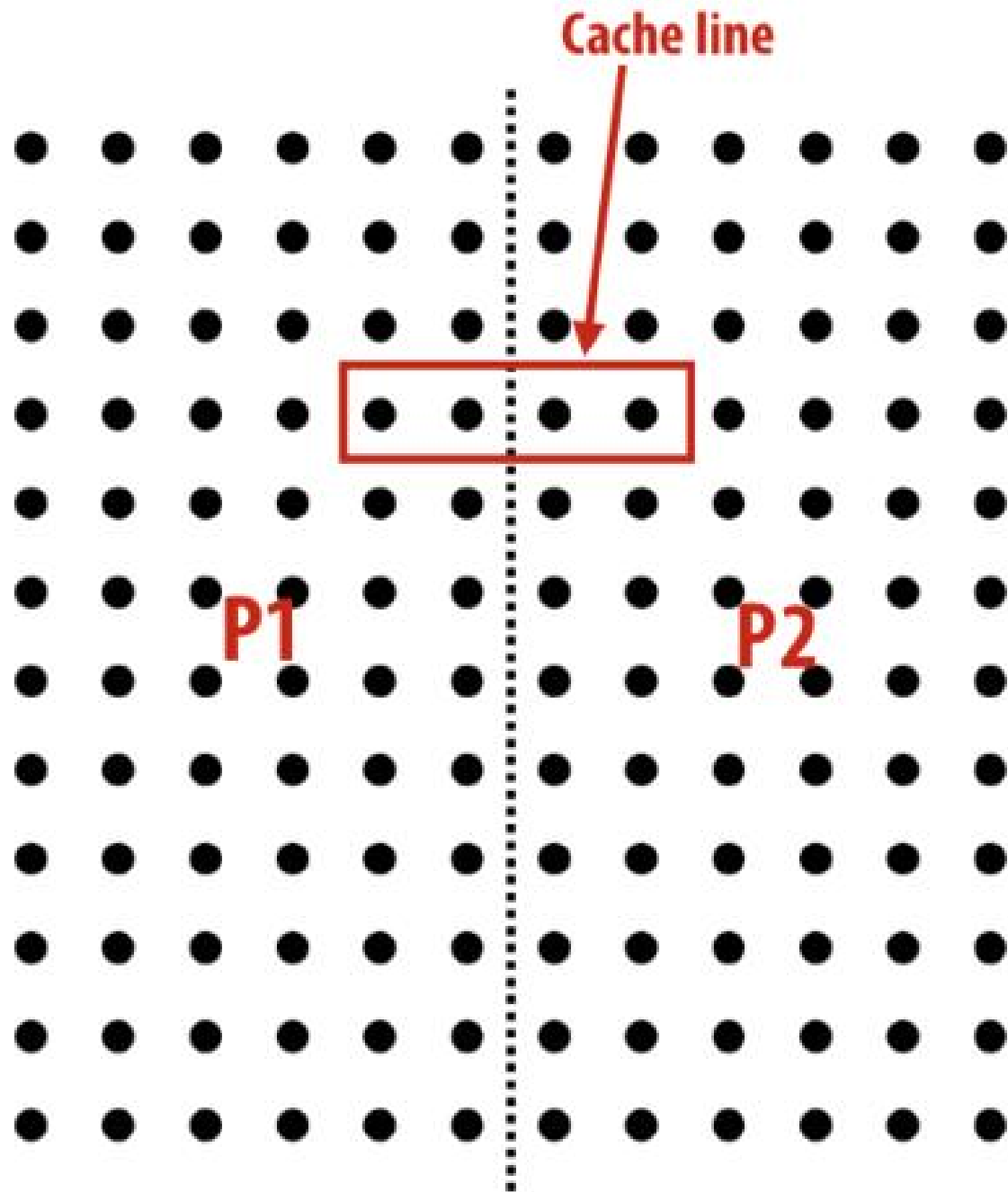
- 通信的粒度可能很重要，因为它可能会引入人为的通信
 - 通信/数据传输的粒度
 - 缓存一致性 (cache coherence) 的粒度

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



以Cache line作为通信粒度

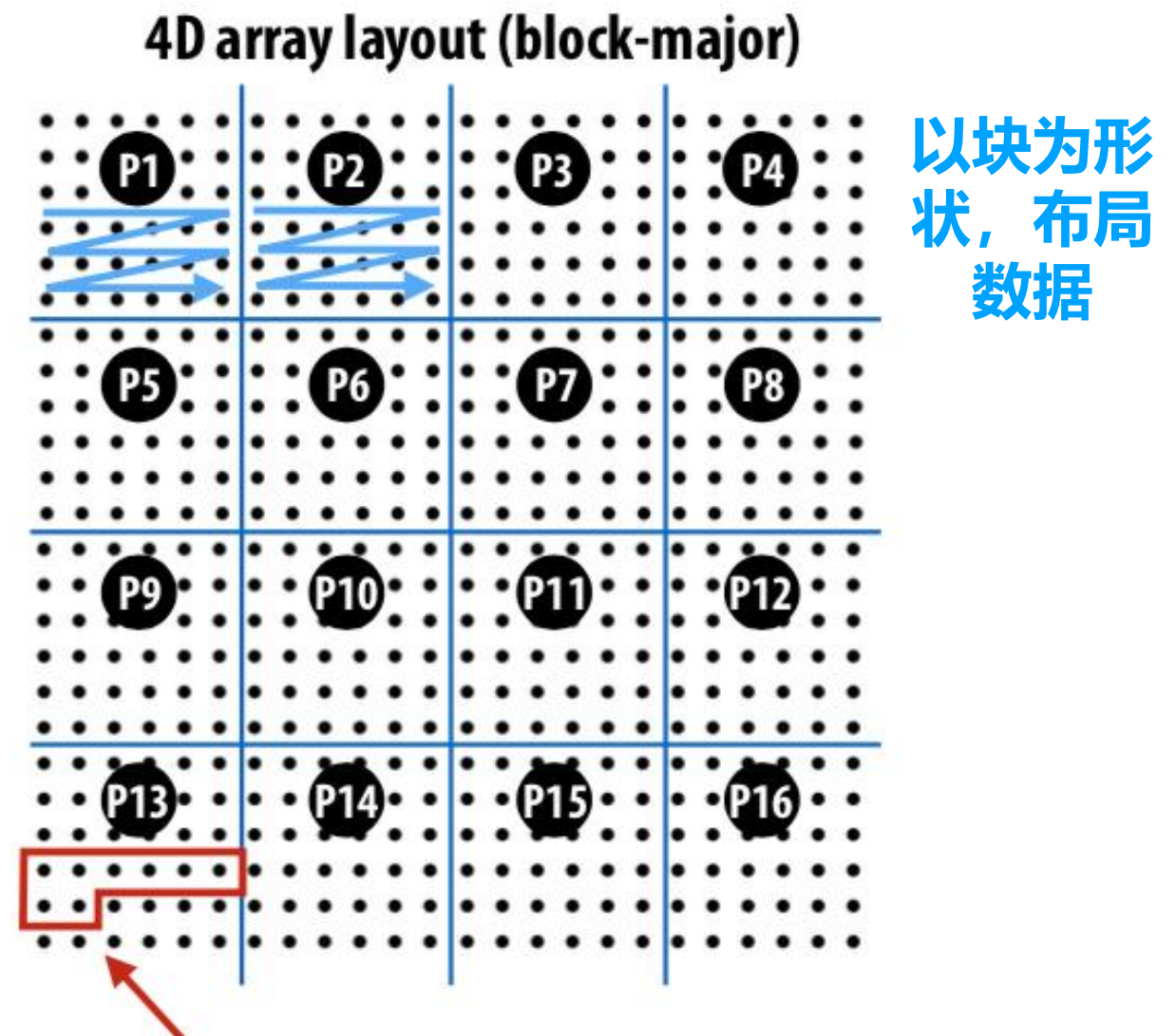
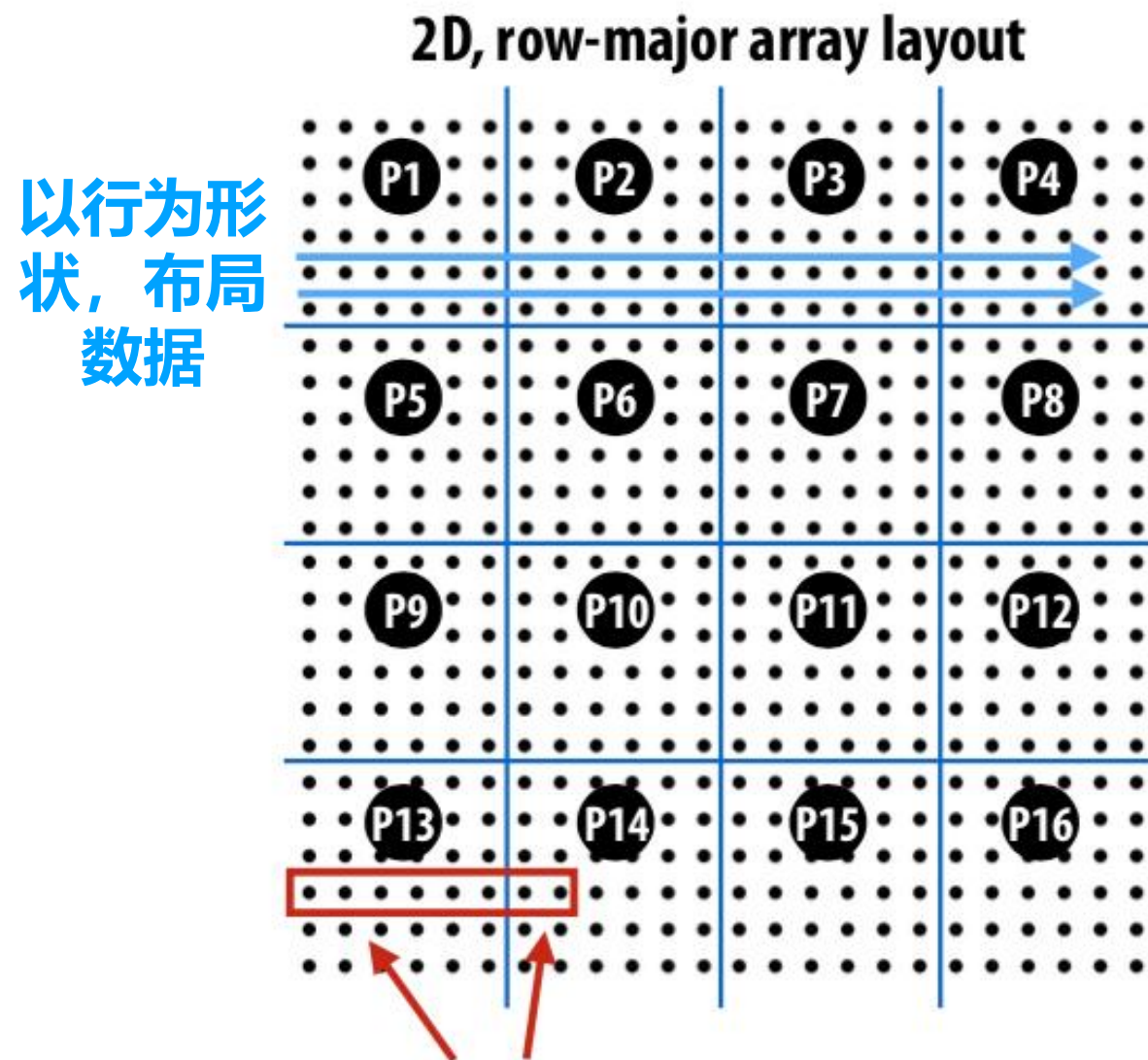


- 数据按列分成两半。分区分配给在处理器1 (P1) 和处理器2(P2)上运行的线程
- 线程访问它们被分配的元素（不存在固有通信），不跨处理器
- 但是，由于同一个cache line被两个处理器同时写入数据，所以在真实系统上的数据访问会触发（人为的）通信*，来解决缓存一致性的问题
 - P1和P2需要对同一个cache line写入数据

* 即将到来的缓存一致性讲座中的更多详细信息

减少人为通信

- 分块数据布局 (blocked data layout)



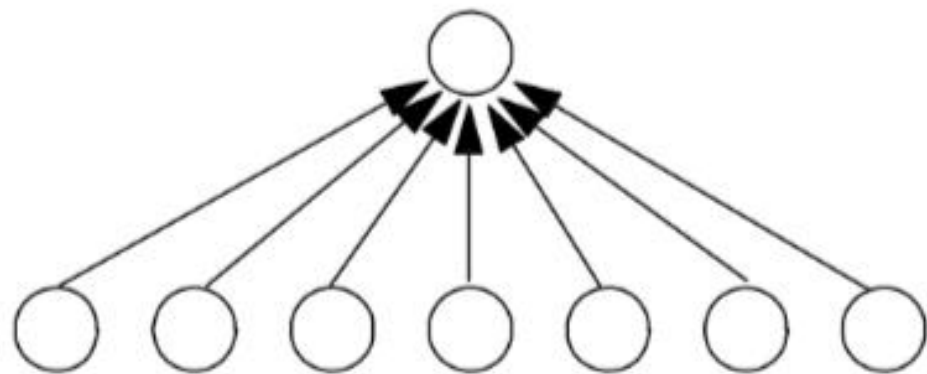
- 注意：不要混淆将工作分配给线程与地址空间中的分块数据布局
- 在上述两种情况处理器线程分配（处理分配）情况相同，同一分区由同一处理器线程处理
- 仅右侧是以块为形状做数据数据布局，引发通信

竞争 (Contention)

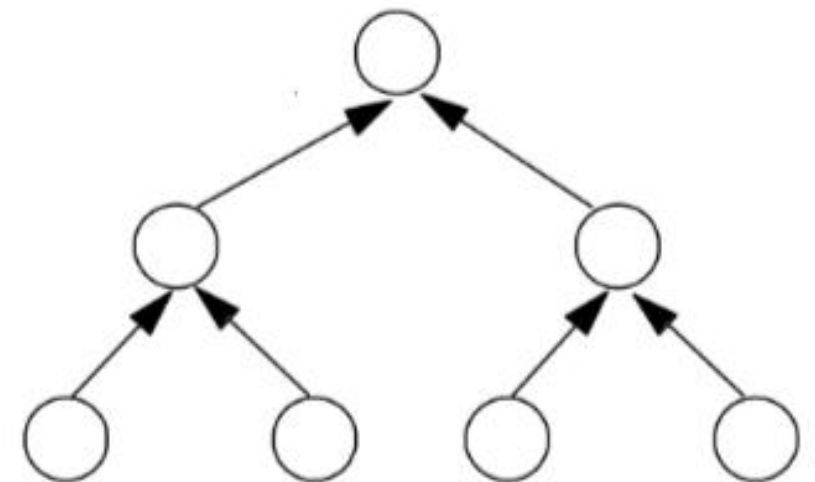
竞争 (Contention)

- 单类资源，可以表示在给定的吞吐量（每单位时间内完成的处理数量）条件下，能够下执行的操作数量
 - 如：内存、通信链路、服务器等...，都可以视为一种资源
- 当在一个小的时间窗口期内，对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)

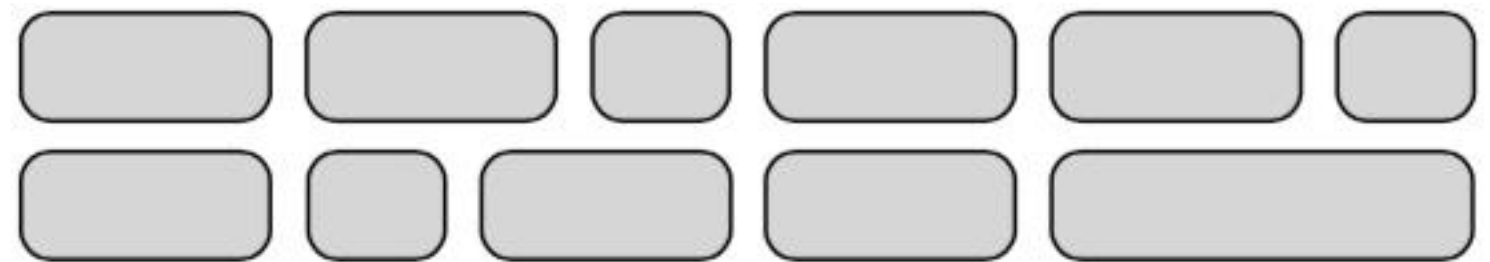
减少竞争

- 分布式作业队列

问题分解（分解子任务）

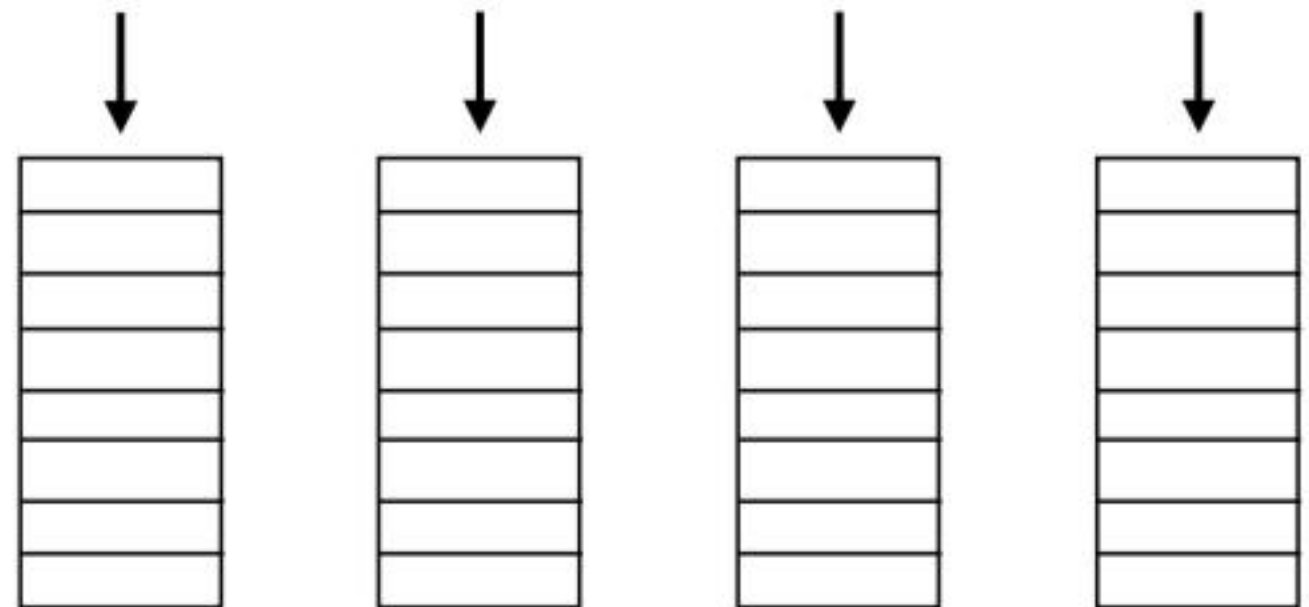
Subproblems

(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)

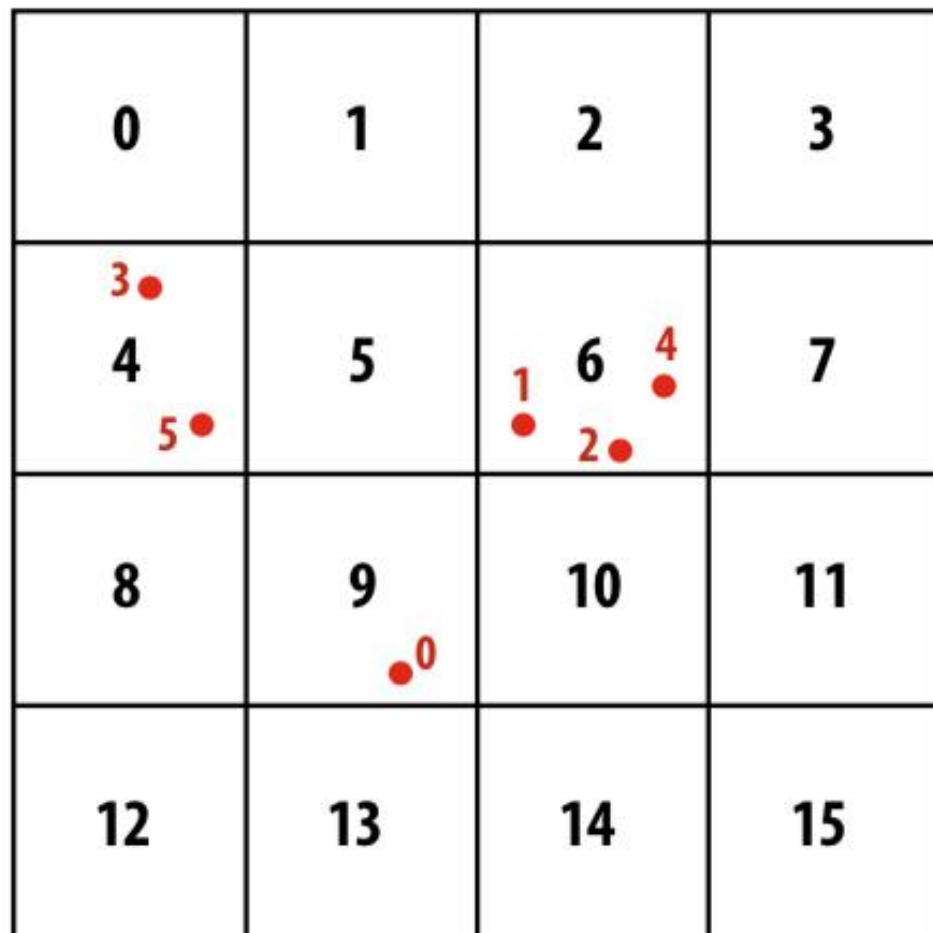


工作线程：

- 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

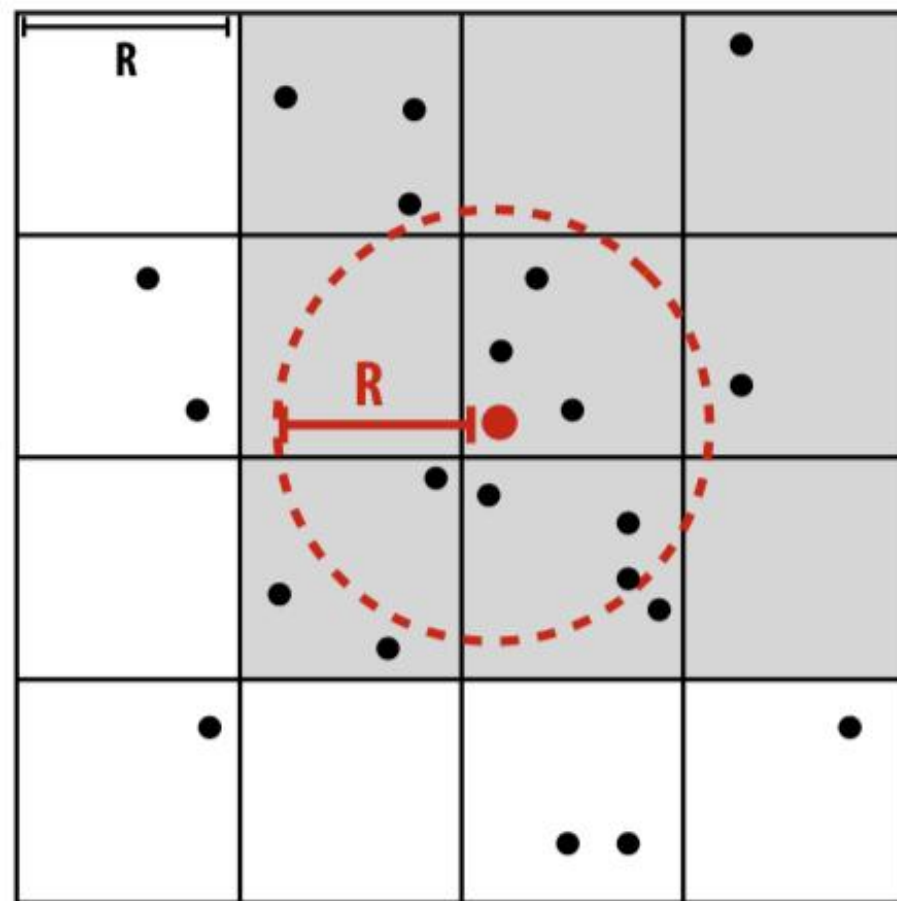
- 在大型并行机上创建粒子数据结构网格
- 将 1M 个点粒子放置在基于 2D 位置的 16 单元格均匀网格中
- 在 GPU 上构建二维列表数组



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
 - 工作效率低下：执行的粒子单元的一个计算（**针对一个点，要计算跟周围的16个网格的受力**），比顺序执行整个算法多 16 倍

```
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）

- 另一个答案：为每个 CUDA 线程分配一个粒子。线程计算包含粒子的单元格，自动更新粒子的参数列表
- 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)
```


方案三：使用更细粒度的锁

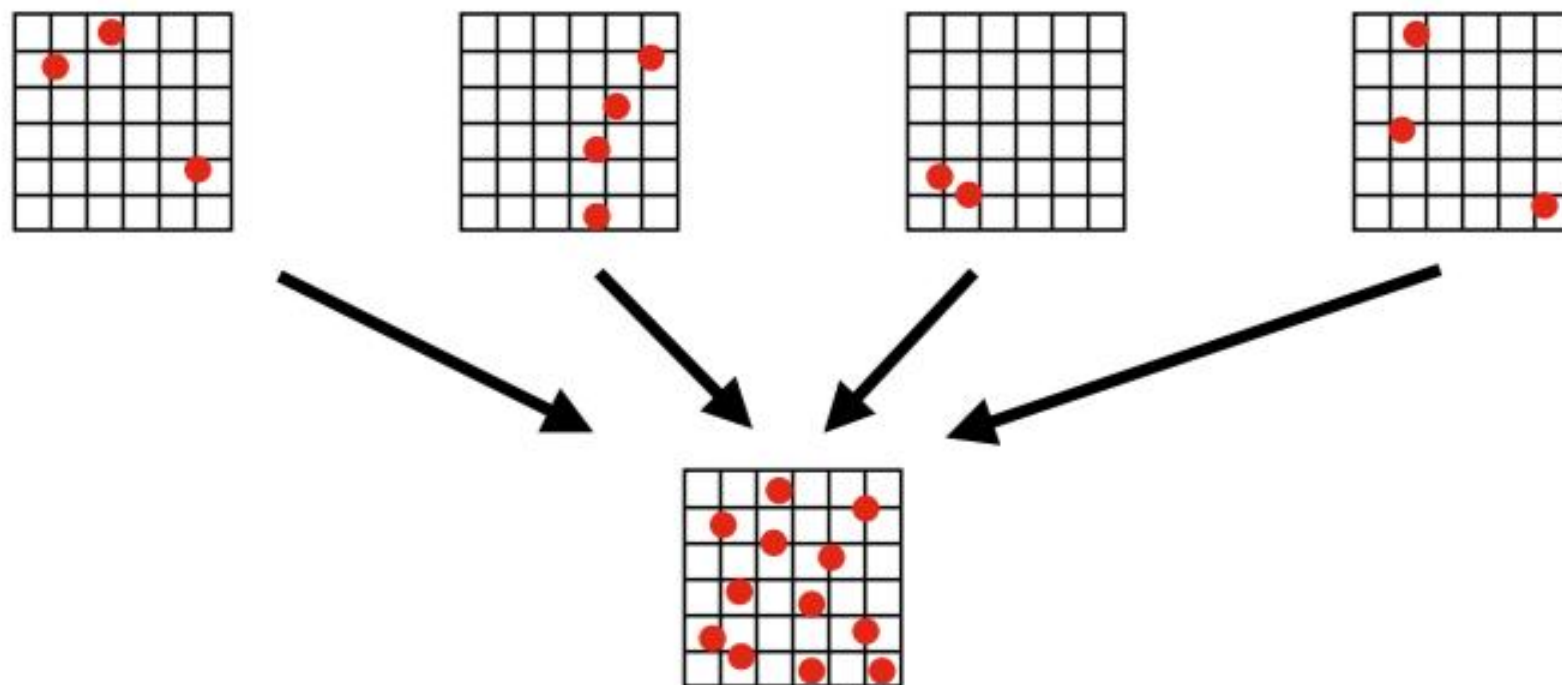
- 通过使用每个单元锁 (per-cell locks) 来缓解对单个全局锁 (single global lock) 的争用
 - 只锁一部分 (一个单元格内的节点结果)
- 假设粒子在 2D 空间中均匀分布.....比解决方案 2 少 16 倍的竞争

```
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（至少与 SMP 内核一样多的线程块）
 - 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



降低通信开销

- 减少与发送方/接收方的通信开销
 - 发送更少次数的消息，使每次发送的消息承载的信息量更大（分摊开销）
 - 将许多小消息合并成大消息
- 减少延迟
 - 应用程序编写者视角：重构代码以利用局部性
 - 硬件实现者视角：改进通信体系结构
- 减少争用
 - 复制竞争资源（例如，本地副本、细粒度锁）
 - 错开对争用资源的访问
- 增加通信/计算重叠
 - 应用编写者：使用异步通信
 - 硬件实现者：流水线、多线程、预取、乱序
 - 在应用程序中需要额外的并发性（比执行单元数更多的并发性）

总结

- 固有与人为所引发的通信
 - 考虑到问题是如何分解的以及工作是如何分配的，固有的沟通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（增加算术强度）
 - 减少开销（降低通信次数、增加单次通信的信息量）
 - 减少竞争
 - 最大化通信和计算的重叠（隐藏延迟以免产生时间开销）