

L04-0

并行编程基础

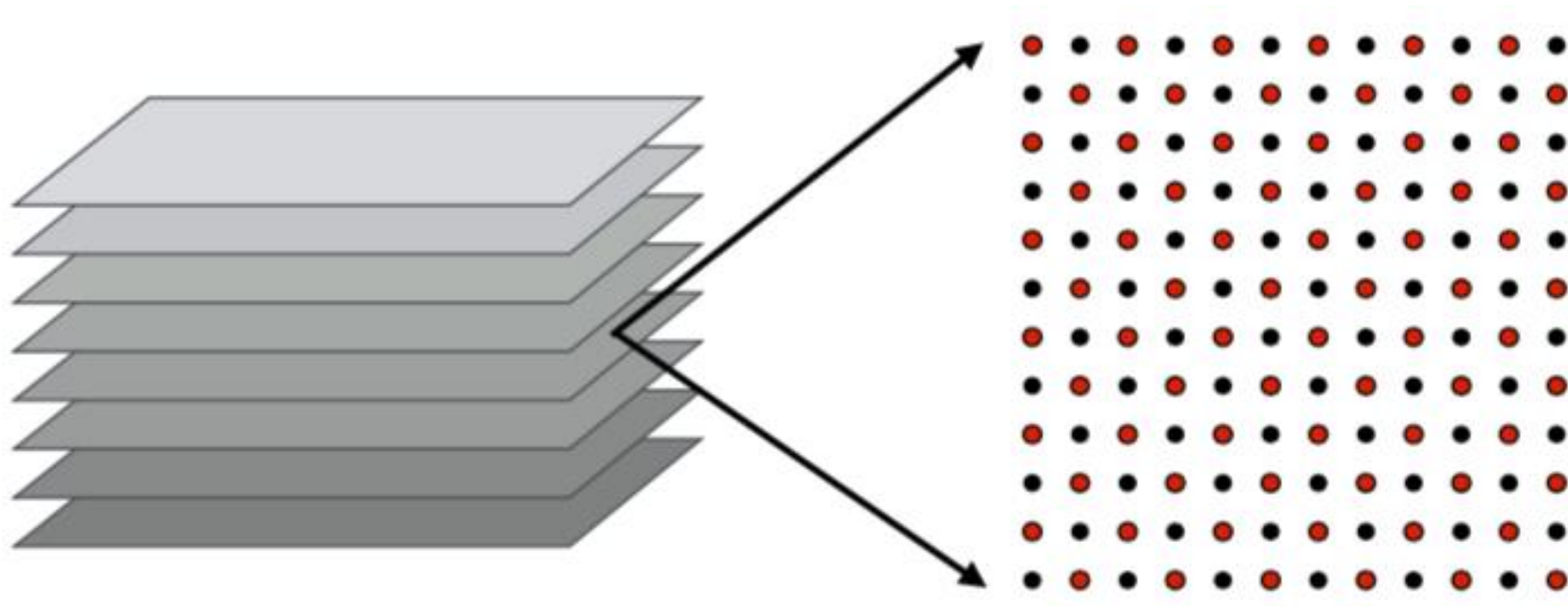
《并行处理》

邵恩
高性能计算机研究中心

Review: 三种并行编程模型

- 共享地址空间 (Shared address space)
 - 通信是非结构化的，隐含在 loads and stores 中
 - 自然的编程方式，但很容易搬起石头砸自己的脚
 - 程序可能是正确的，但性能不佳
- 消息传递 (Message passing)
 - 将所有通信结构化为消息 (messages)
 - 通常比共享地址空间更难编程
 - 结构通常有助于我获得第一个正确的、可扩展的程序
- 数据并行 (Data parallel)
 - 结构化的计算可以被视为一组计算的集合
 - 假设有一个共享空间来加载输入/存储结果，但数据并行编程模型严重限制了映射迭代之间的通信 (目标：保持迭代的独立处理)
 - 现代的编程模型解决方案中，鼓励但不强制执行这种结构

并行化的应用程序示例



- 将 3D ocean volume离散化为表示为 2D 网格的切片
- 海洋的离散演化时间（对每个点的计算时间）： Δt
- 高精度仿真需要小 Δt 和高分辨率网格（每个点数据量更大）

创建并行程序

- 思考过程
 - 确定可以并行执行的“小程序”
 - 对各个可独立执行的“小程序”分配计算资源（以及与“小程序”关联的数据的对应访存资源）
 - 管理数据访问、通信和同步

回想一下：我们的主要目标之一是提高加速比（Speedup）

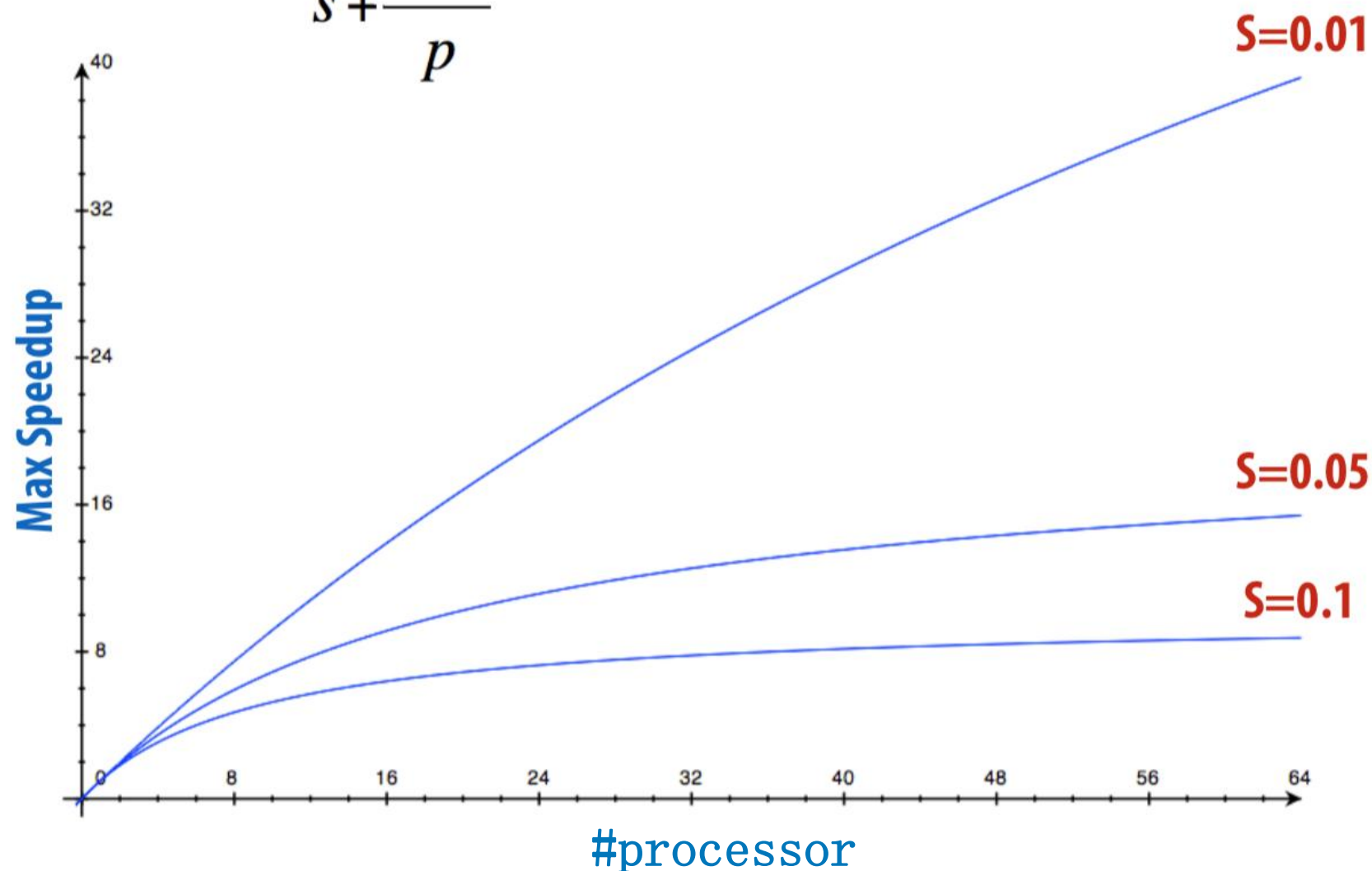
对于一个确定性的计算：

$$\text{Speedup (用P个计算核心)} = \frac{\text{Time (用1个计算核心)}}{\text{Time (用P个计算核心)}}$$

阿姆达尔定律 (Amdahl's Law)

- Let **S** = 只能串行的执行的程序在整个程序中的比例（不可并行加速的部分），S越小speedup越高
- 1-S=能够并行执行的程序**在整个程序中的比例

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



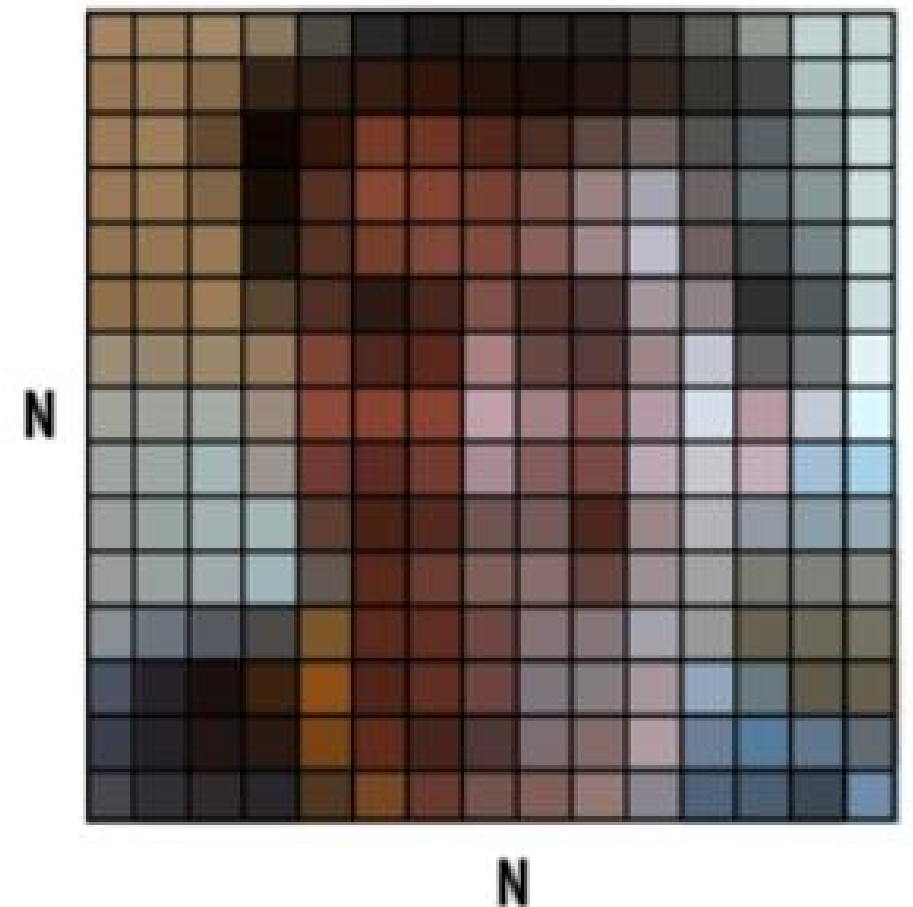
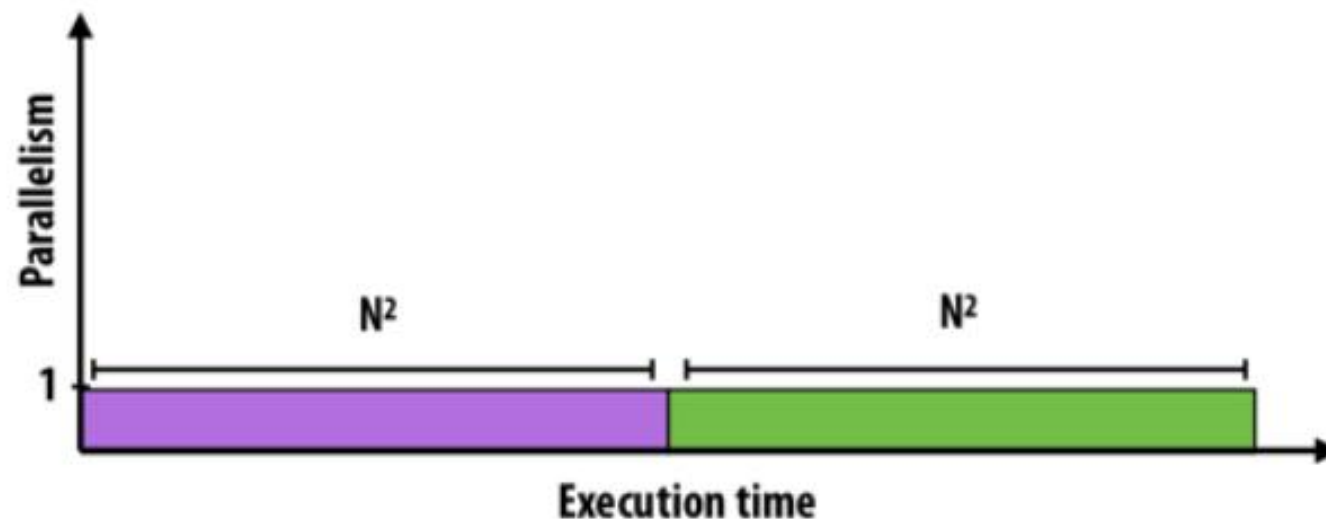
一个简单的例子

一个简单的例子考虑对 $N \times N$ 图像进行两步计算

- **第 1 步：**所有像素双倍亮度（每个网格元素独立计算）
- **第 2 步：**计算所有像素值的平均值

顺序执行程序

这两个步骤都需要 $\sim N^2$ 时间，所以总时间是 $\sim 2N^2$



并行性的首次尝试

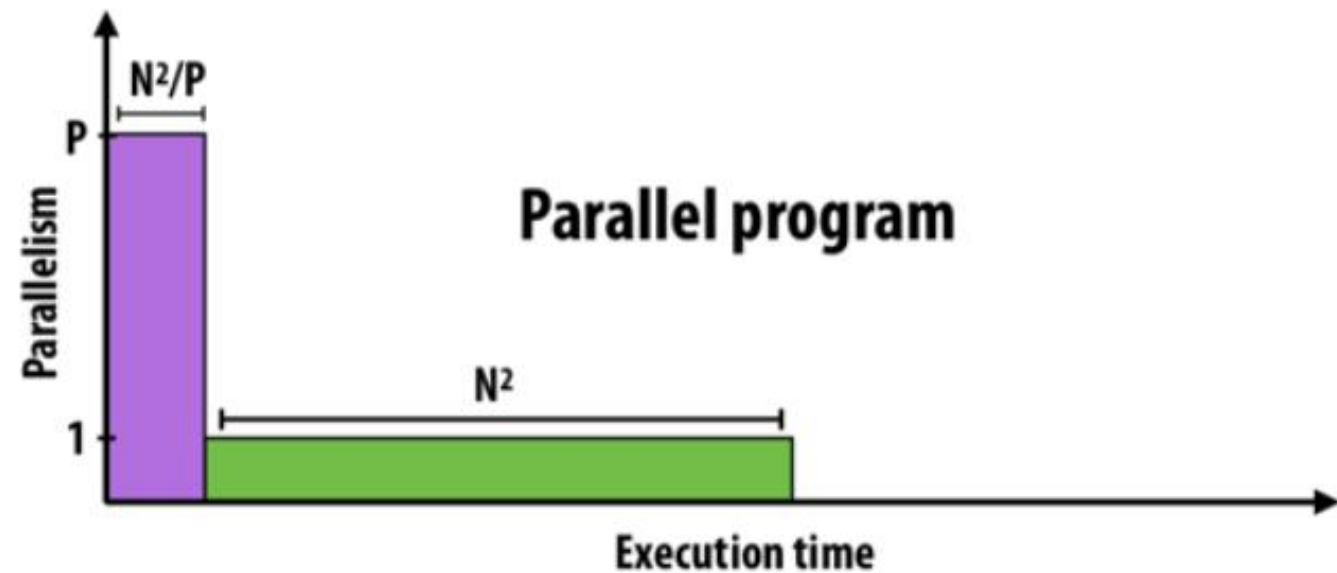
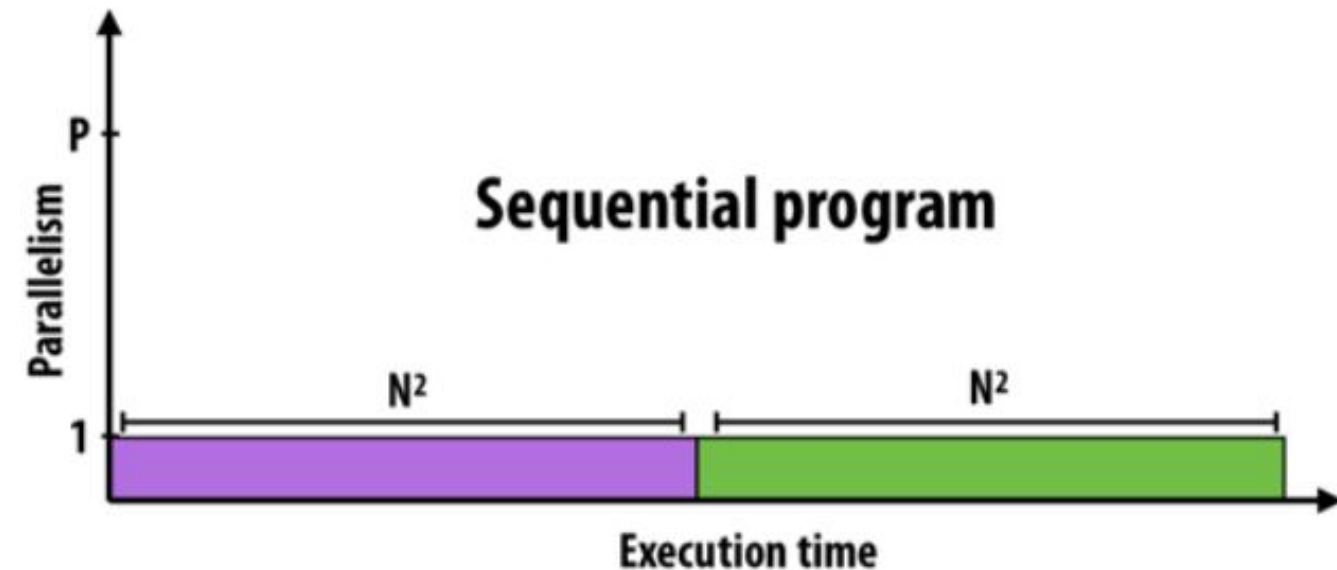
主要步骤:

第 1 步: **并行**执行阶段 1: N^2/P

第 2 步: **串行**执行第 2 阶段: N^2

$$\text{Speedup} \leq \frac{2n^2}{\frac{n^2}{p} + n^2}$$

$$\text{Speedup} \leq 2$$



再进一步进行并行优化

主要步骤:

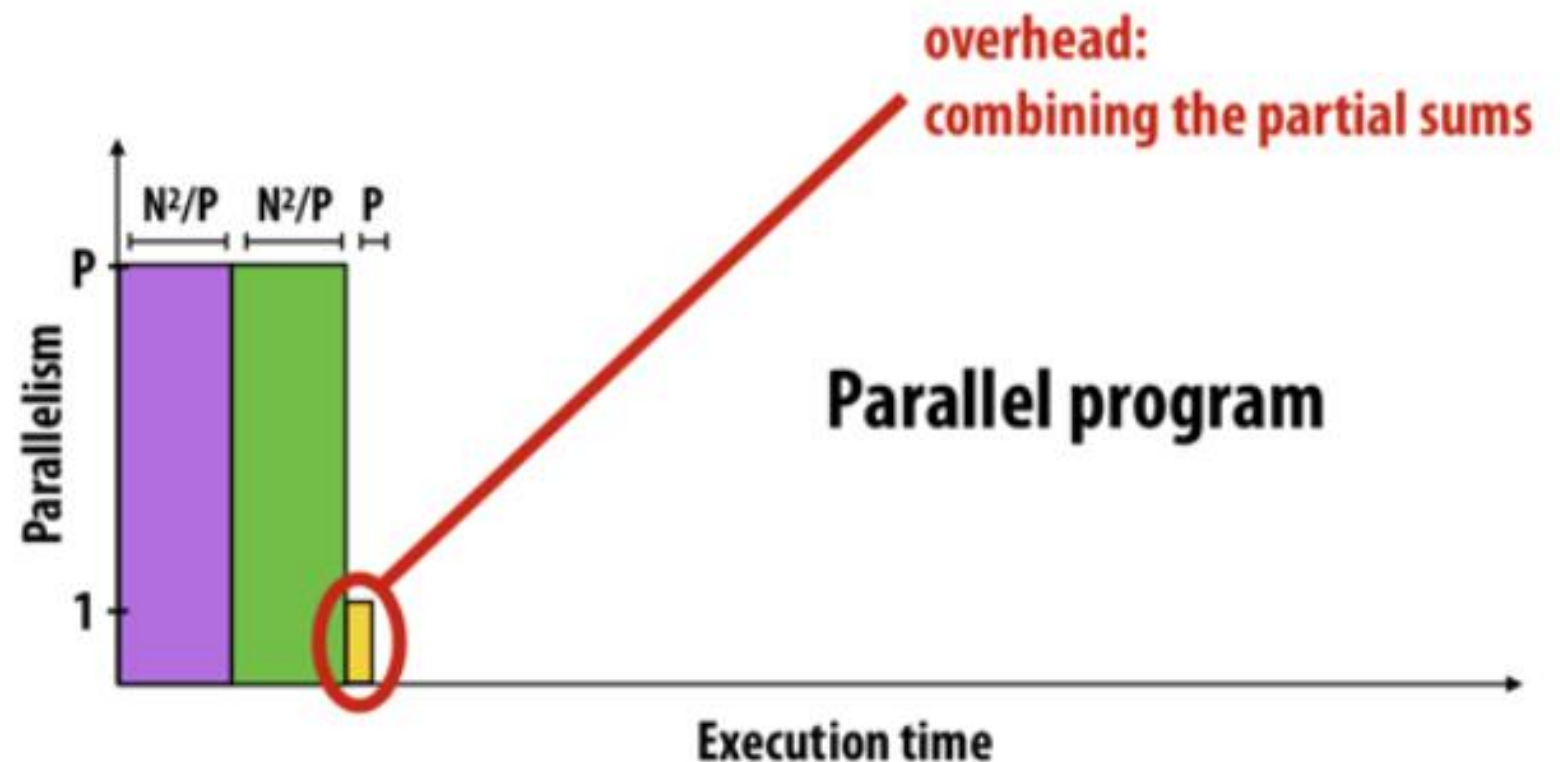
- 第 1 步: 并行执行阶段 1: N^2/P
- 第 2 步: 并行计算部分和, 串行合并结果, 阶段 2 的时间: $N^2/P + P$

Overall performance:

$$\text{Speedup} \leq \frac{2n^2}{\frac{2n^2}{p} + p}$$

Note:

speedup $\rightarrow P$ when $N \gg P$

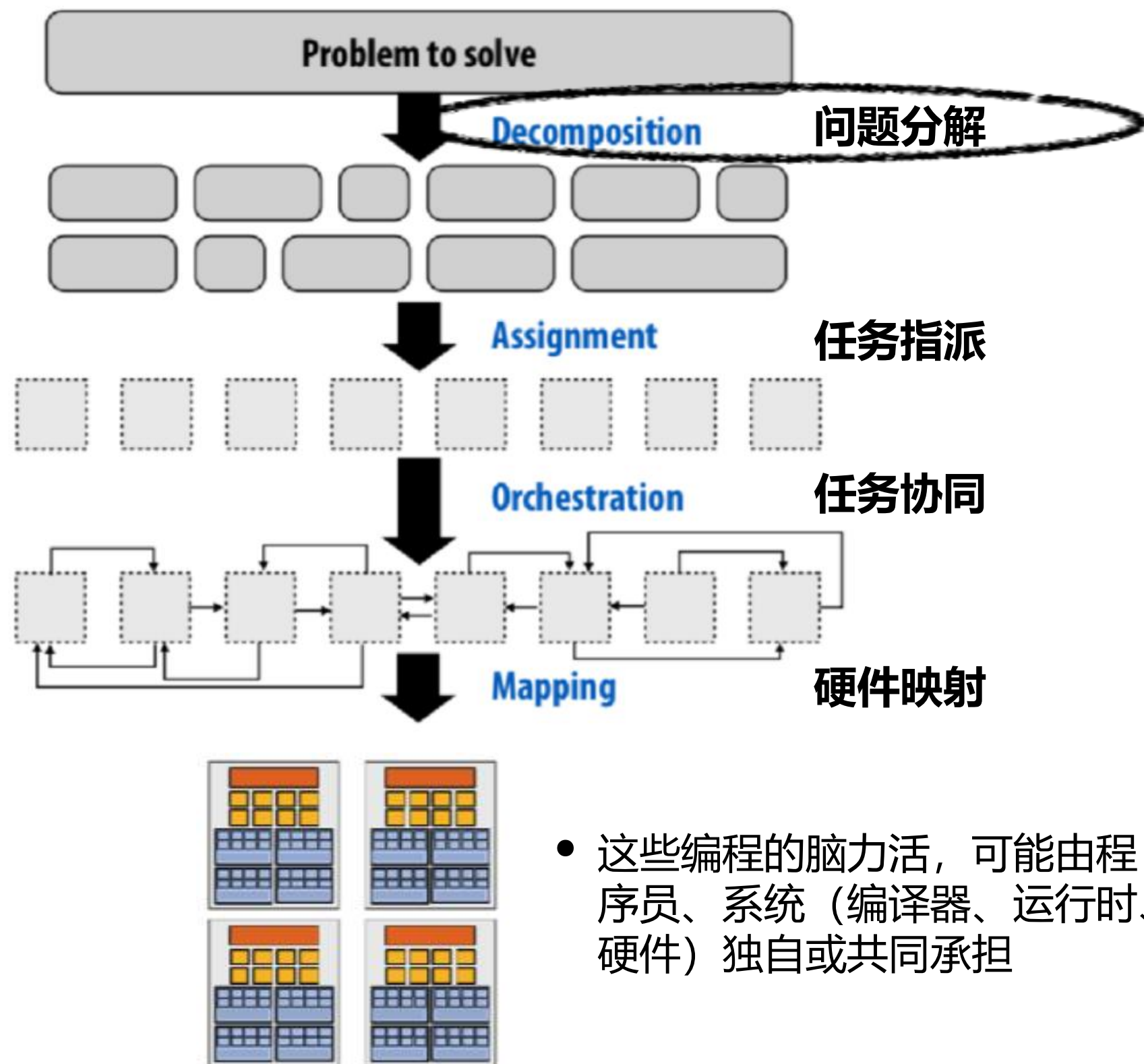


创建并行程序

子问题
(又名“任务” “需要要做的
各项独立可执行的工作”)

并行线程**
(“工人”)

并程序
(线程间通信)



- 这些编程的脑力活，可能由程序员、系统（编译器、运行时、硬件）独自或共同承担

问题分解

- 将问题分解成可以并行执行的任务
- 问题分解可以是静态发生，即运行前分解（也可动态）
- 可以在程序执行时识别新任务（在运行时完成）

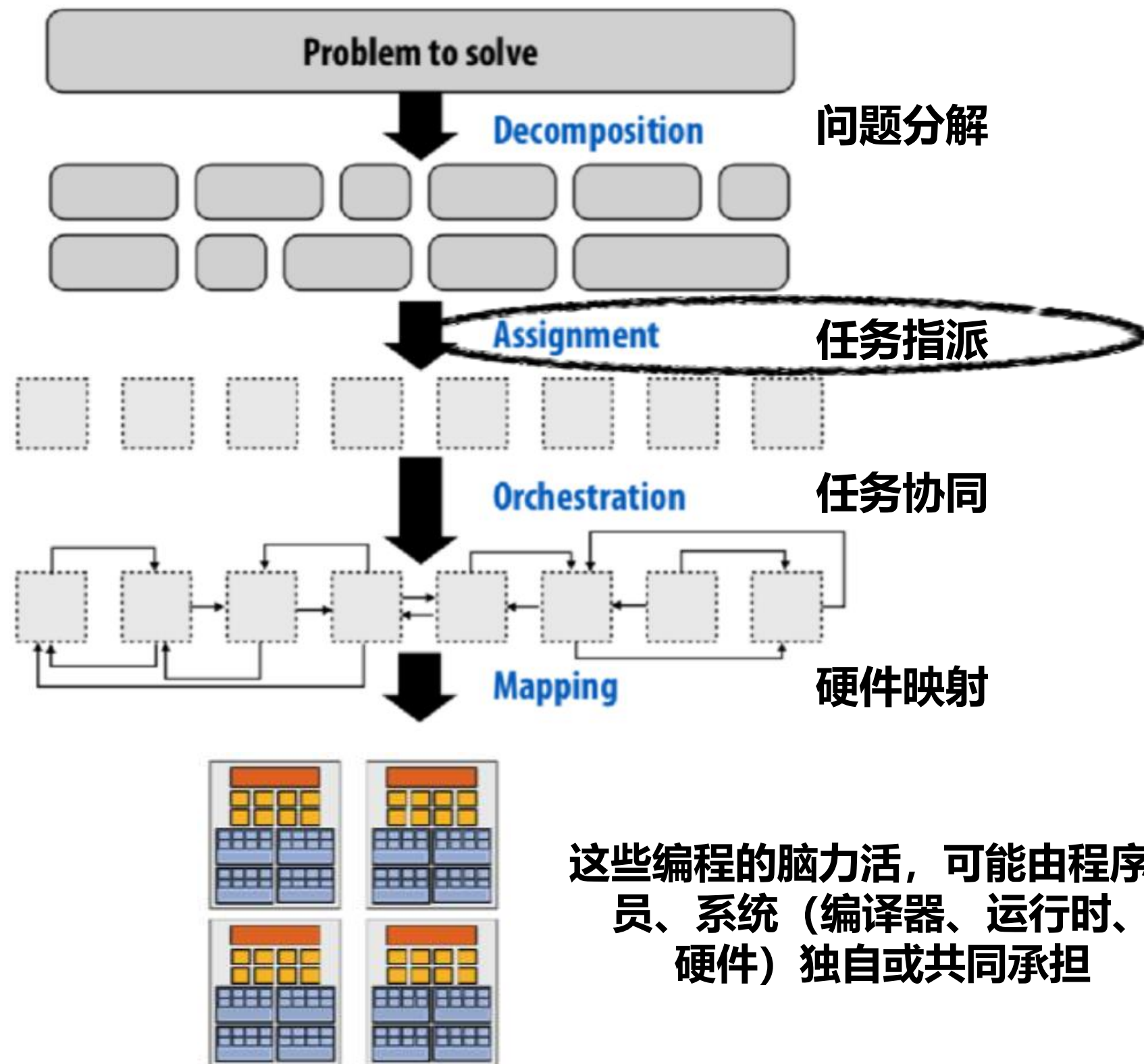
Main idea: 创建至少足够的任务，以保持机器上的所有计算单元都保持忙碌，避免资源空闲

分解的关键方面：识别任务间的依赖关系（或识别哪些任务间缺乏依赖关系）

问题分解

- 谁来做问题分解？
 - 大多数情况下：程序员
- 对顺序执行程序**自动问题分解**仍然是一个具有挑战性的研究问题
 - 编译器必须分析程序，识别依赖关系
 - 有通过简单的循环嵌套来实现问题分解的成功案例（少数）
 - 用于复杂通用代码的**“神奇并行编译器”**尚未实现

任务指派

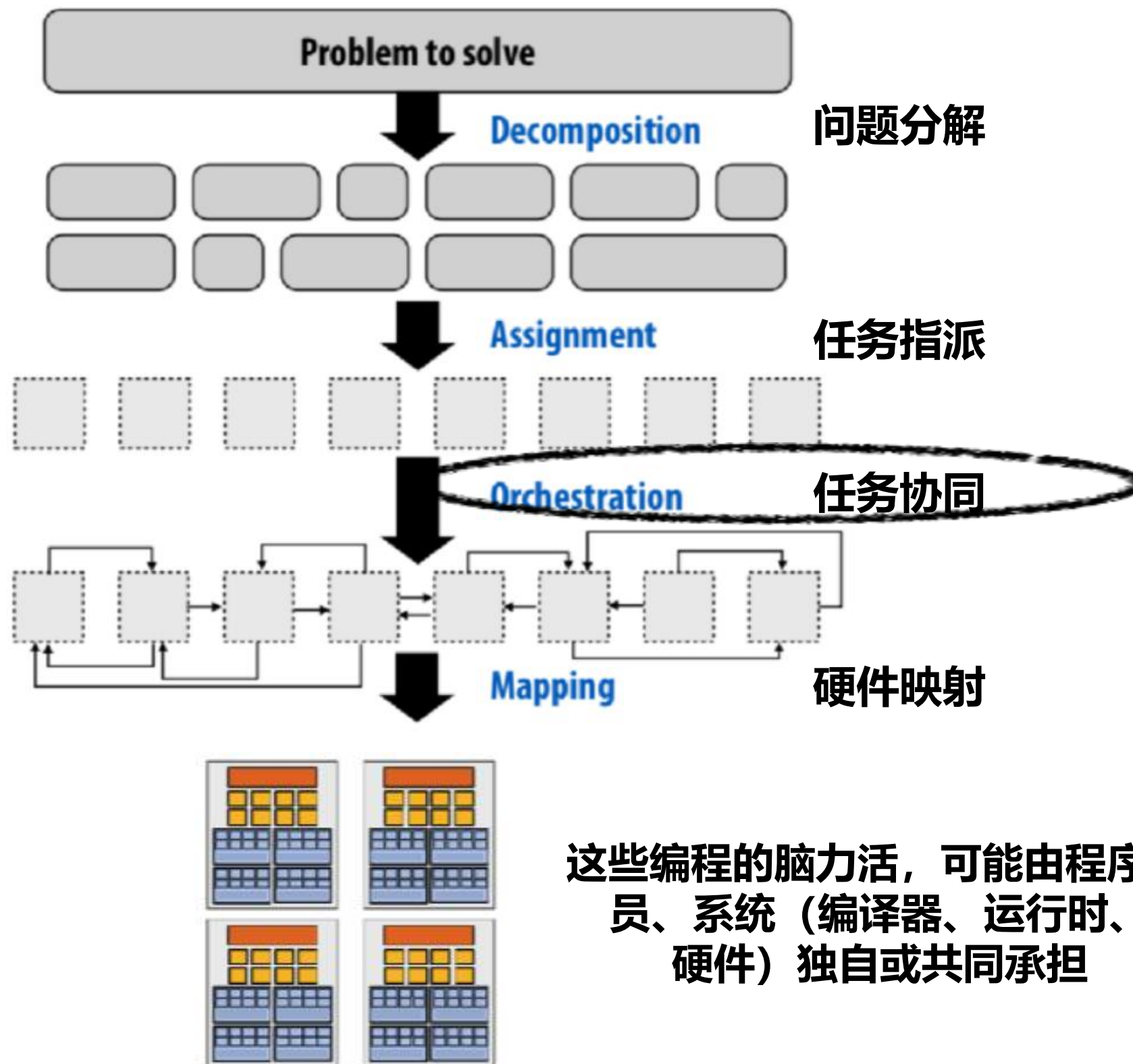


这些编程的脑力活，可能由程序员、系统（编译器、运行时、硬件）独自或共同承担

任务指派

- 将任务分配给线程
 - 将任务视为要做的事情
 - 将线程视为工人
- 目标：平衡工作量，降低沟通成本
- 可以静态执行，也可以在执行过程中动态执行
- 虽然程序员通常负责分解，但许多语言/运行时负责分配

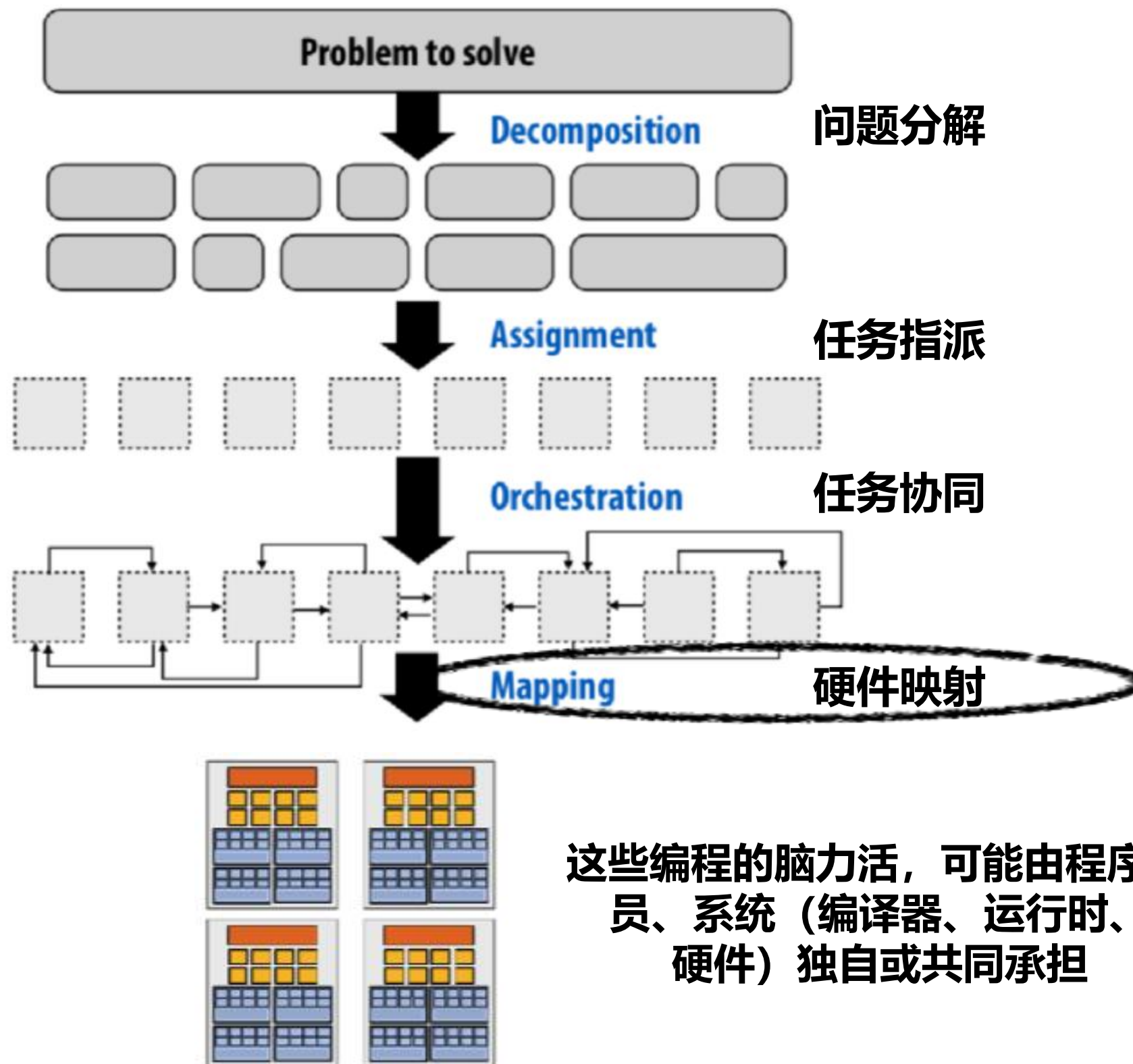
任务协同



任务协同

- 会涉及到以下细节
 - 构建进程间的消息沟通
 - 如有必要，需要额外添加同步，以保留依赖关系
 - 在内存中组织数据结构
 - 调度任务
- 目标：计算系统体系结构会影响其中的许多设计或编程决策
 - 如果同步代价高昂（通信延迟或访存延迟开销大），可能会更少地使用调用进程间的同步

硬件映射

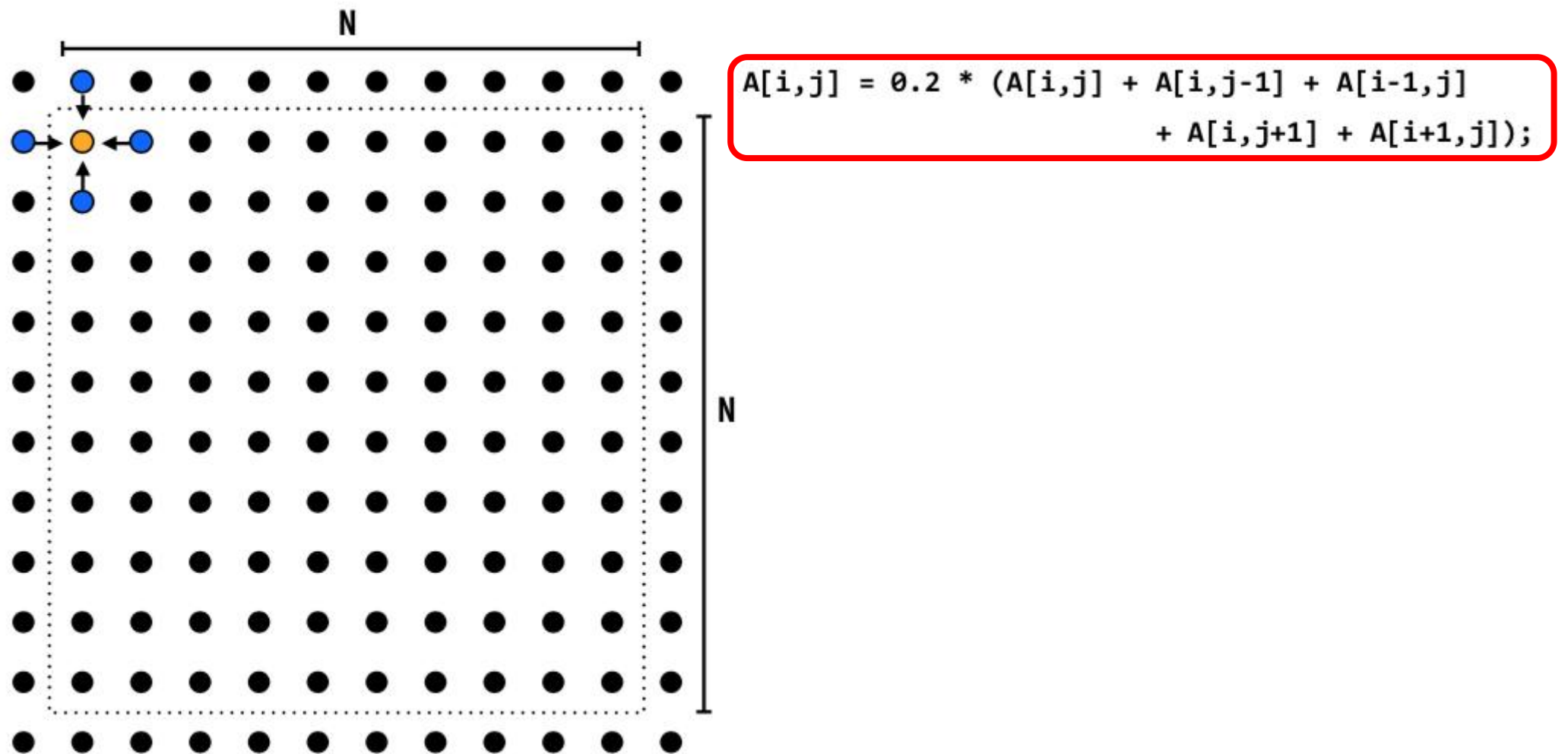


硬件映射

- 将“线程” / “工作者” 映射到硬件的计算执行单元
- 示例 1：操作系统映射
 - 将 pthread 映射到 CPU core的硬件执行上下文
- 示例 2：编译器映射
 - 将程序实例映射到向量指令通道 (vector instruction lanes)
- 示例 3：硬件映射
 - 将 CUDA 线程块映射到 GPU cores上
- 一些有趣的映射
 - 将相关性高的多个线程（如：主线程与协作线程）放在同一个处理器上（最大化局部性、数据共享、最小化通信/同步成本）
 - 将不相关的线程放在同一个处理器上以更有效地使用机器
 - 可能是访存带宽有限的，避免访存强占
 - 可能是计算资源有限的，避免上下文切换

示例：基于二维网格（2D-grid）的求解器

- 用迭代法求解 $(N+2) \times (N+2)$ 网格的偏微分方程(partial differential equation, PDE)
- 对网格执行高斯-塞德(Gauss-Seidel)迭代扫描直到收敛

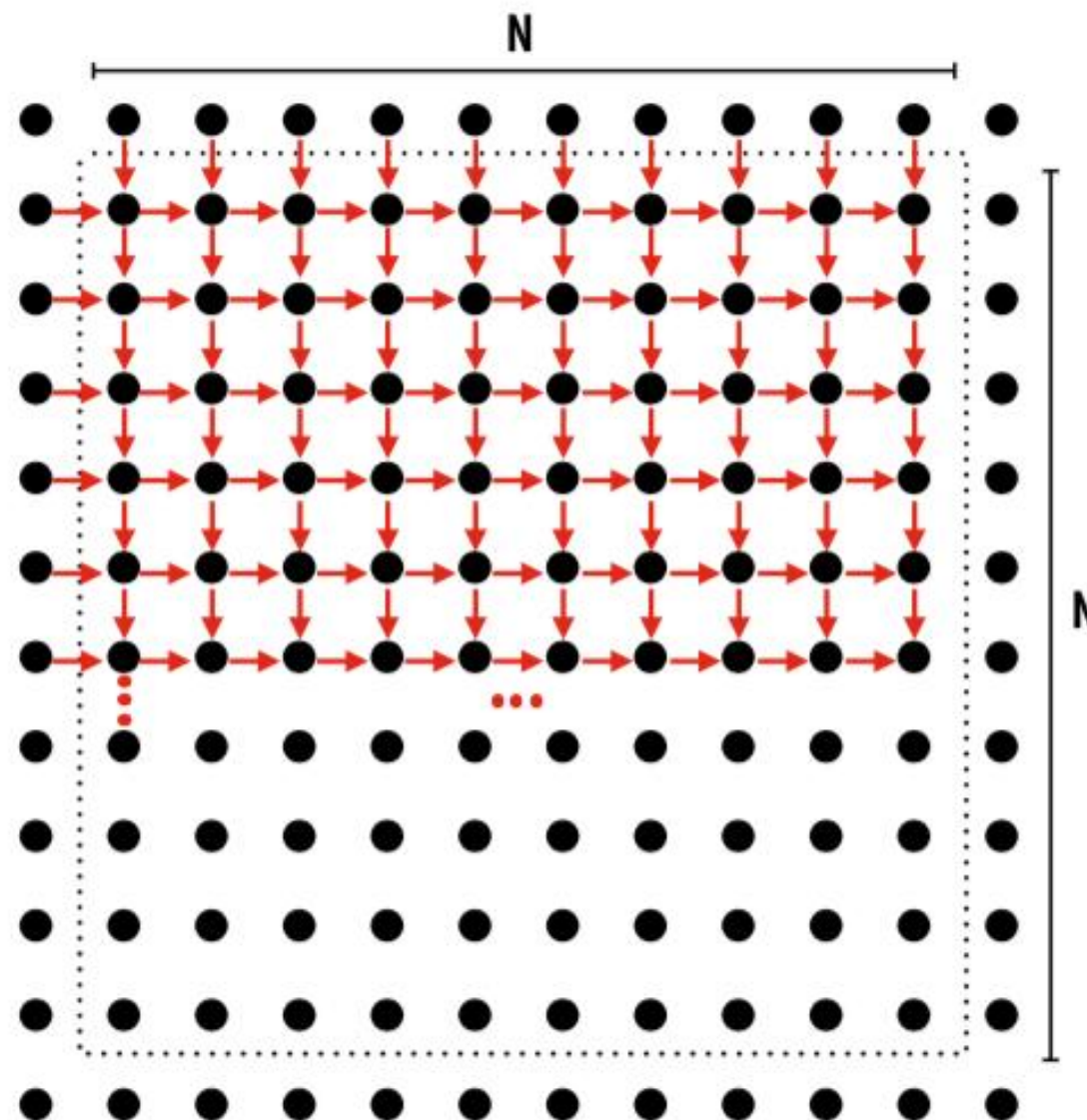


网格求解器算法

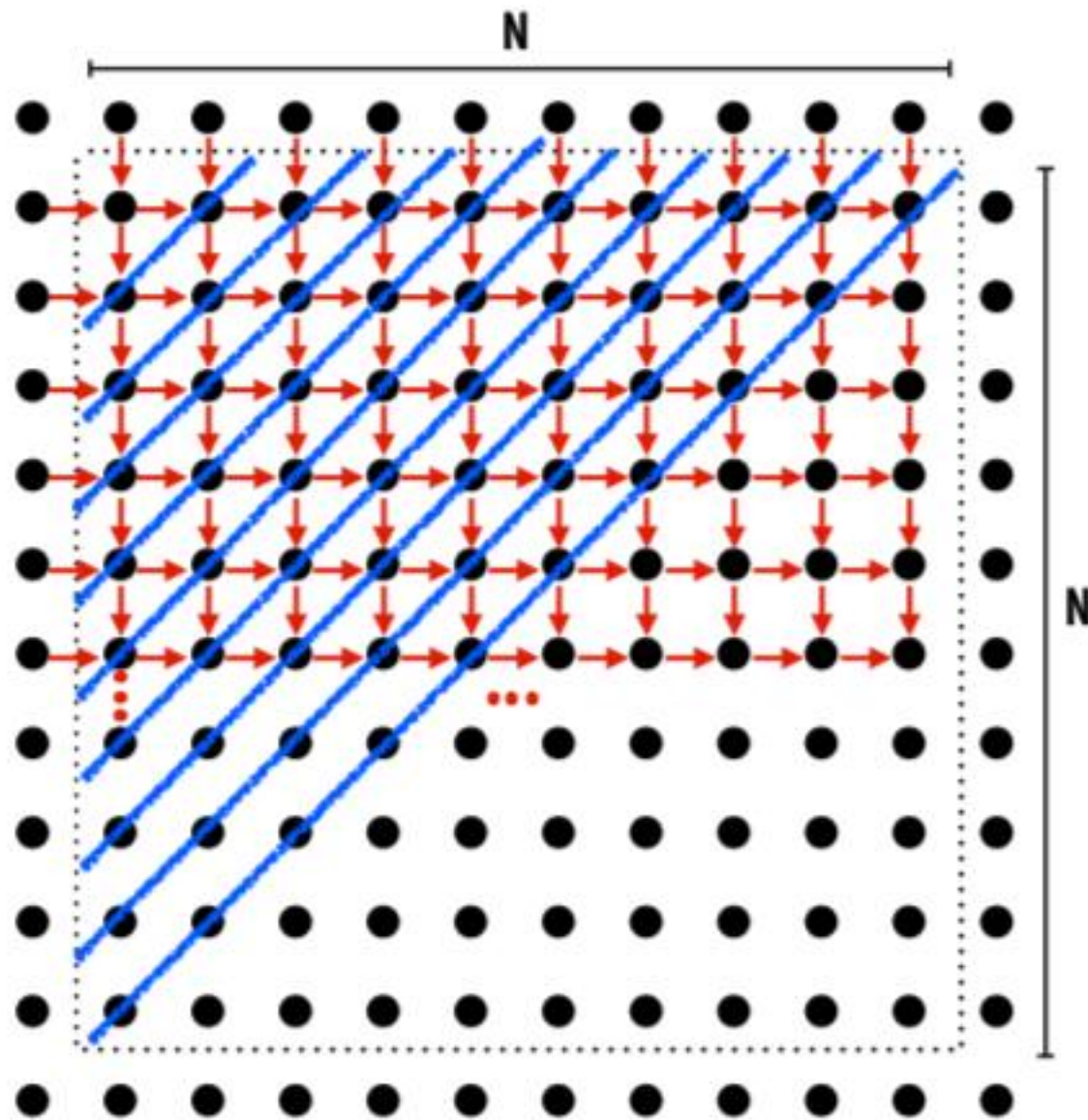
```
const int n;  
float* A;                                // assume allocated to grid of N+2 x N+2 elements  
  
void solve(float* A) {  
  
    float diff, prev;  
    bool done = false;  
  
    while (!done) {                        // outermost loop: iterations  
        diff = 0.f;  
        for (int i=1; i<n; i++) {          // iterate over non-border points of grid  
            for (int j=1; j<n; j++) {  
                prev = A[i,j];  
                A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +  
                                A[i,j+1] + A[i+1,j]);  
                diff += fabs(A[i,j] - prev); // compute amount of change  
            }  
        }  
  
        if (diff/(n*n) < TOLERANCE)        // quit if converged  
            done = true;  
    }  
}
```

Step 1: 识别依赖项

- 每行元素依赖于**左边**的元素
- 每列取决于**前一列**



Step 1: 识别依赖项



- 沿对角线方向上的计算任务，彼此独立
- 好处：存在并行性，可能的实施并行优化策略
 - 将对角线上的网格单元划分为独立执行的计算任务
 - 并行地进行计算，并更新值
 - 完成后，移动到下一个对角线
- 坏处：独立工作很难榨取并行的油水
 - 计算开始和结束时并行度不高
 - 频繁同步（完成每条对角线后）

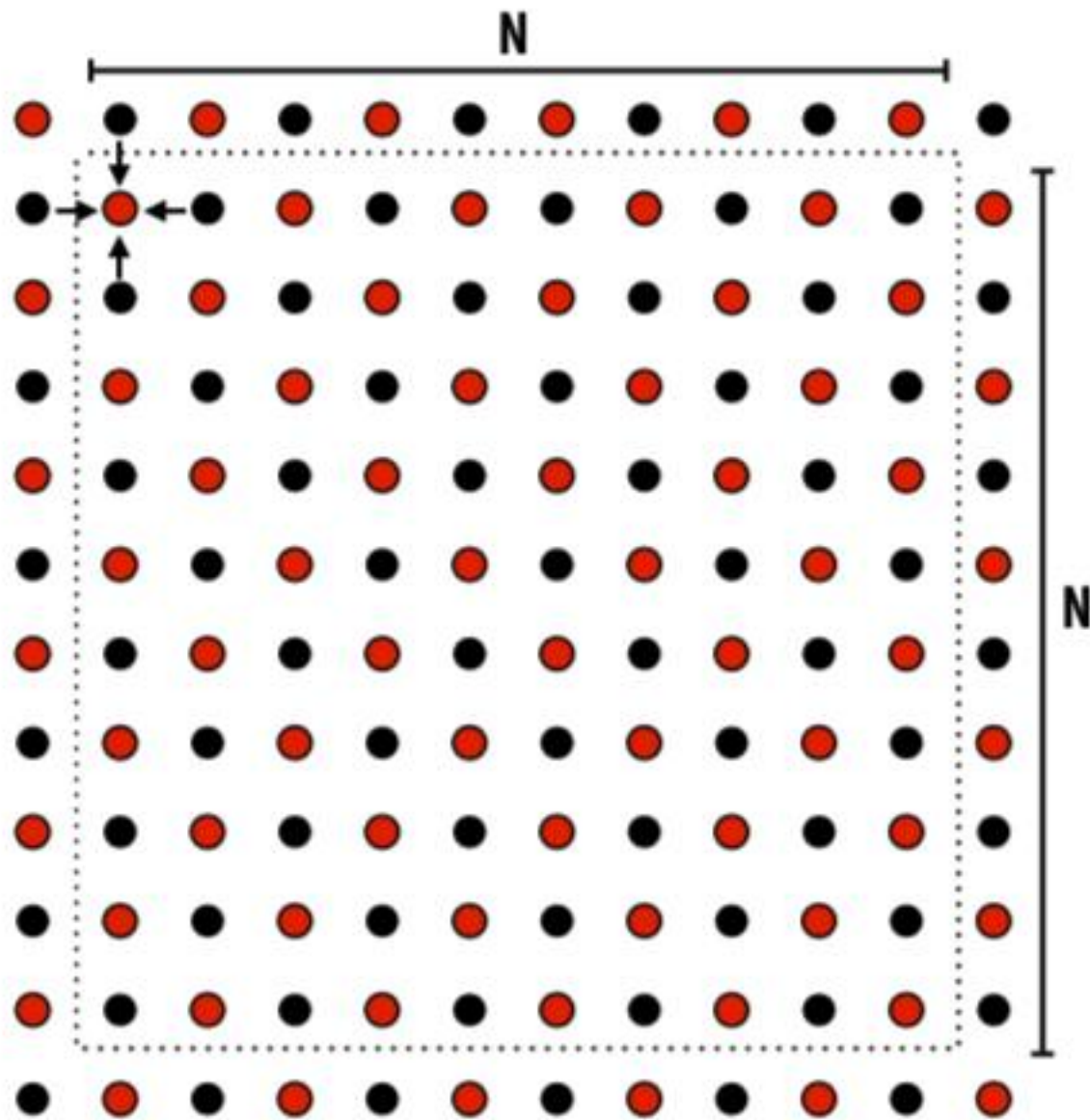
别灰心，还有办法！

想法：通过将算法更改为更适合并行性的算法来提高性能

- 更改网格单元格的**更新顺序**
- 能否在保证收敛性的条件下，**求解近似解**，来替代精确解？
 - 注意：计算的浮点值不同，但解仍然收敛到误差阈值内
- 是的，我们需要 **Gauss-Seidel 方法的领域知识**来求解线性系统，以实现应用程序允许的这种变化

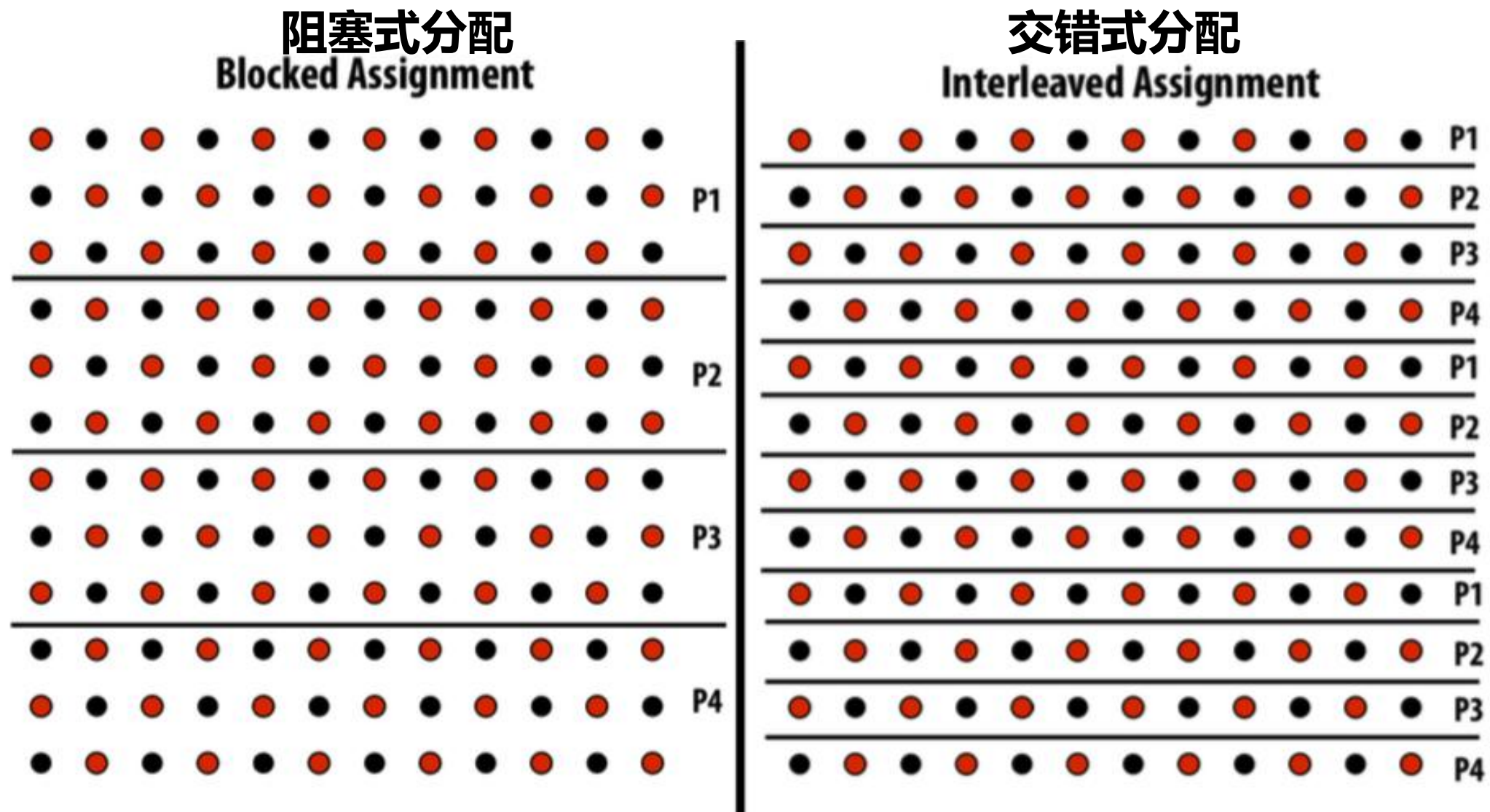
新方法

通过红黑着色，重新对网格单元更新顺序进行排序



- 并行更新所有红节点单元
- 红色不会依赖红色
- 完成更新红色单元格后，并行更新所有黑色单元格
- 尊重对红色单元格的依赖性
- 黑色不会依赖黑色
- 重复直到收敛

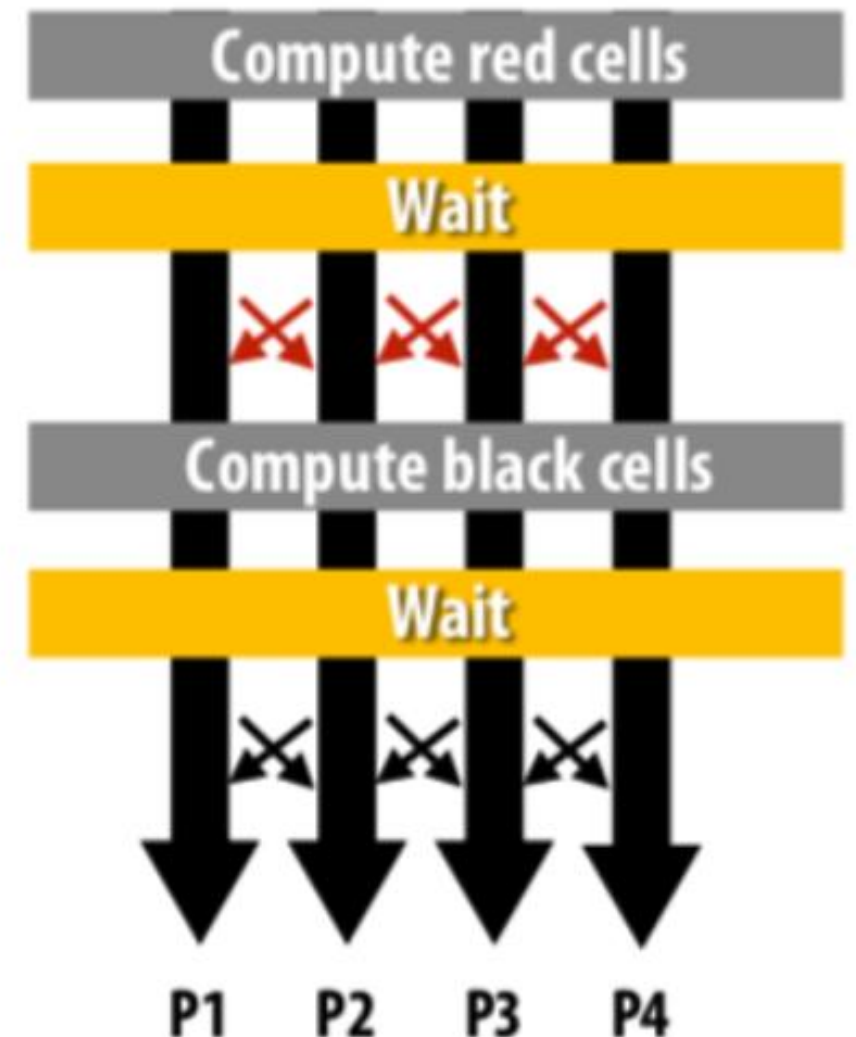
将工作分配给处理器核



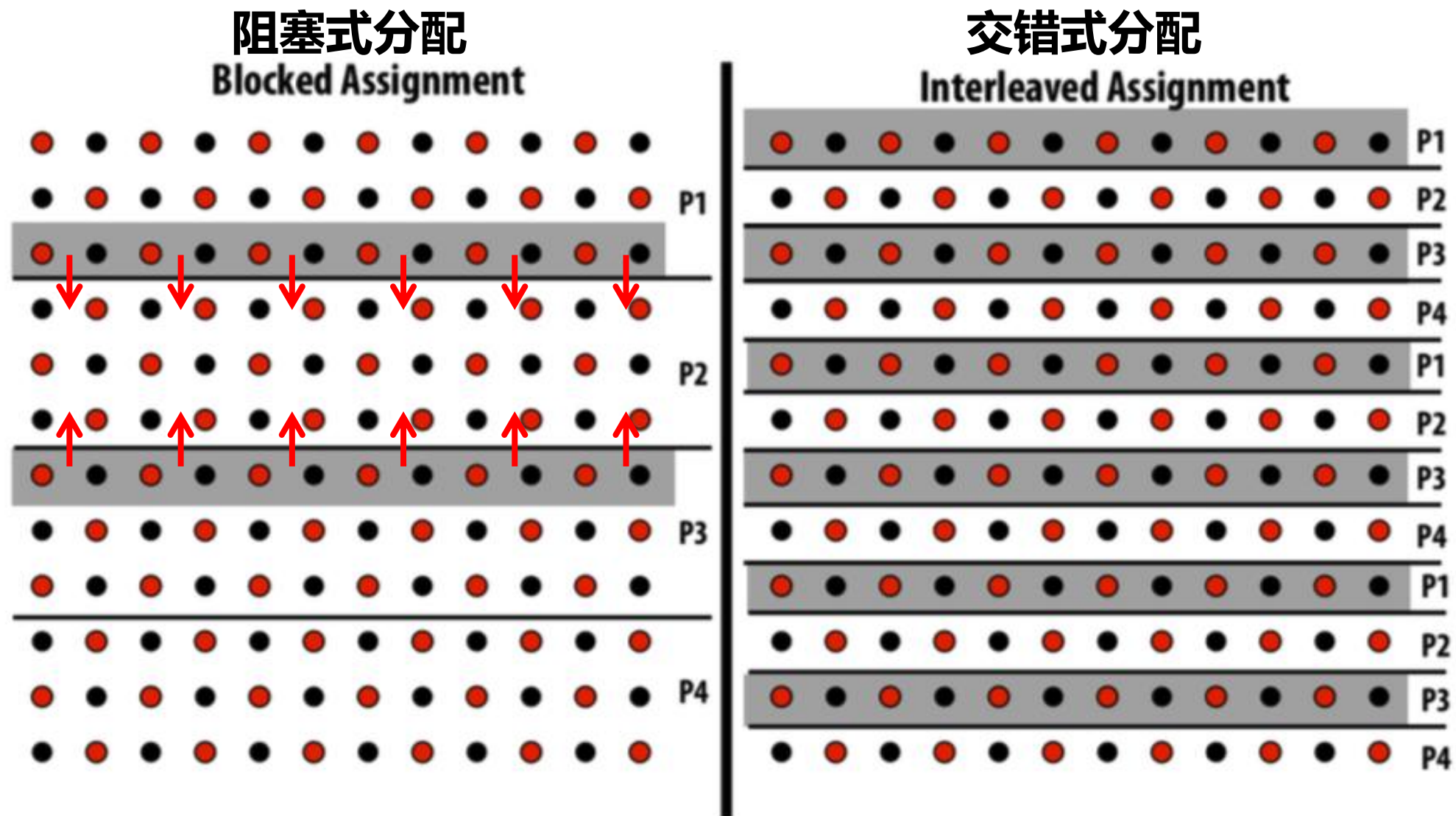
- 问题：哪个更好？有关系吗？
- 答：这取决于运行这个程序的硬件计算系统

考虑依赖关系（数据流）

- 1.并行执行红色点的计算任务，并更新结果
- 2.等待所有处理器完成更新
- 3.将更新后的红色节点的结果，传达给其他处理器
- 4.并行执行黑色点的计算任务，并更新结果
- 5.等待所有处理器完成更新
- 6.将更新的黑色节点的结果，传送给其他处理器
- 7.重复，直到收敛



由分配硬件资源，而产生的额外通信



- 阴影框：每次迭代必须发送到邻近的处理器（如：P2）的数据
- 阻塞分配需要较少的数据在处理器之间进行通信，降低处理器间数据传输总量

网格求解器的数据并行编程模型的表达

- 只显示红色点的程序代码与结果更新

```
const int n;  
  
float* A = allocate(n+2, n+2)); // allocate grid  
  
void solve(float* A) {  
  
    bool done = false;  
    float diff = 0.f;  
    while (!done) {  
        for_all (red cells (i,j)) {  
            float prev = A[i,j];  
            A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +  
                           A[i+1,j] + A[i,j+1]);  
            reduceAdd(diff, abs(A[i,j] - prev));  
        }  
  
        if (diff/(n*n) < TOLERANCE)  
            done = true;  
    }  
}
```

资源分配

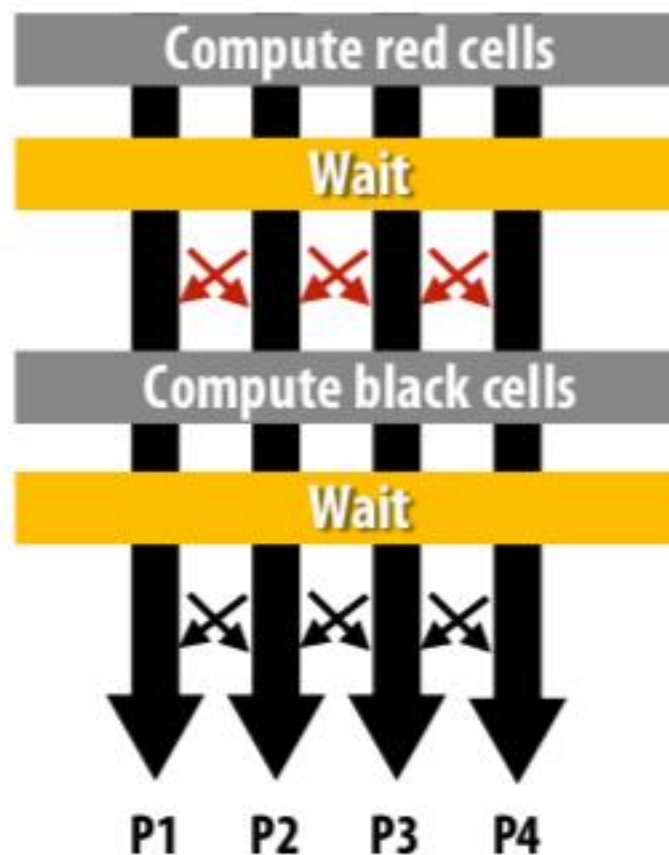
分解：
单独的网格元
素构成独立的
计算任务

编排：由系统处理
(内置通信原语：reduceAdd)

编排：由系统处理
(所有块的结束是隐式等待所有
任务完成，然后返回顺序控制)

求解器的共享地址空间表达式

- 程序员负责同步
- 通用同步原语
 - 锁（提供互斥）：一次只有一个线程在临界区
 - 阻塞操作（Barriers）：等待各个线程都结束，再执行下一项



基于共享地址空间的求解器

假设这些是**全局变量**（所有线程都可以访问）

假设 solve 函数由所有可执行的线程执行（SPMD 的工作模式）

每个 SPMD 实例的 threadId 值不同：程序员利用 threadId 值来计算要处理的网格区域

每个线程计算它负责更新的行

```
int    n;                // grid size
bool   done = false;
float  diff = 0.0;
LOCK   myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS)

    while (!done) {
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                               A[i+1,j], A[i,j+1]);

                lock(myLock)
                diff += abs(A[i,j] - prev));
                unlock(myLock);
            }
        }
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff/(n*n) < TOLERANCE)
            done = true;
        barrier(myBarrier, NUM_PROCESSORS);
    }
}
```

// check convergence, all threads get same answer

需要互斥

- 每个线程执行过程中，对寄存器的操作
 - 指令1：将 **diff** 的值存到寄存器 **r1**
 - 指令2：将寄存器 **r2** 累加 (add) 到寄存器 **r1**
 - 指令3：将寄存器 **r1** 的值存入 **diff**
- 一种有可能会出现的交错赋值情况，如下：（让 $\text{diff} = 0$ 的起始值， $r1 = 1$ ）
- 解决方案：需要这组三个指令组合要具有原子性，不可分，避免交错被不同线程调用

线程0 T0	线程1 T1
$r1 \leftarrow \text{diff}$	T0 reads value 0
$r1 \leftarrow r1 + r2$	T1 reads value 0
$\text{diff} \leftarrow r1$	T0 sets value of its r1 to 1
	T1 sets value of its r1 to 1
	T0 stores 1 to diff
	T1 stores 1 to diff

保持原子性的机制

- 通过Lock/unlock 保证关键指令或指令组的互斥

```
LOCK(mylock);  
// critical section  
UNLOCK(mylock);
```

- 一些编程语言对代码块的原子性有单独的支持

```
atomic {  
    // critical section  
}
```

- 通过硬件支持的内建专用指令，来保障原子性的“读取-修改-写入”操作

```
atomicAdd(x, 10);
```

基于共享地址空间实现的求解器

```
int      n;                // grid size
bool     done = false;
float    diff = 0.0;
LOCK     myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {

    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS)

    while (!done) {
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                               A[i+1,j], A[i,j+1]);
                lock(myLock)
                diff += abs(A[i,j] - prev);
                unlock(myLock);
            }
        }
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff/(n*n) < TOLERANCE)                // check convergence, all threads get same answer
            done = true;
        barrier(myBarrier, NUM_PROCESSORS);
    }
}
```

能看出来潜在的问题吗？
共享地址空间的优势还存在吗？
Do you see a potential performance
problem with this implementation?

基于共享地址空间实现的求解器

```
int    n;                // grid size
bool   done = false;
float  diff = 0.0;
LOCK   myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
    float myDiff;
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS)

    while (!done) {
        float myDiff = 0.f;
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                               A[i+1,j], A[i,j+1]);
                myDiff += abs(A[i,j] - prev));
            }
            lock(myLock);
            diff += myDiff;
            unlock(myLock);
        }
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff/(n*n) < TOLERANCE)
            done = true;
        barrier(myBarrier, NUM_PROCESSORS);
    }
}
```

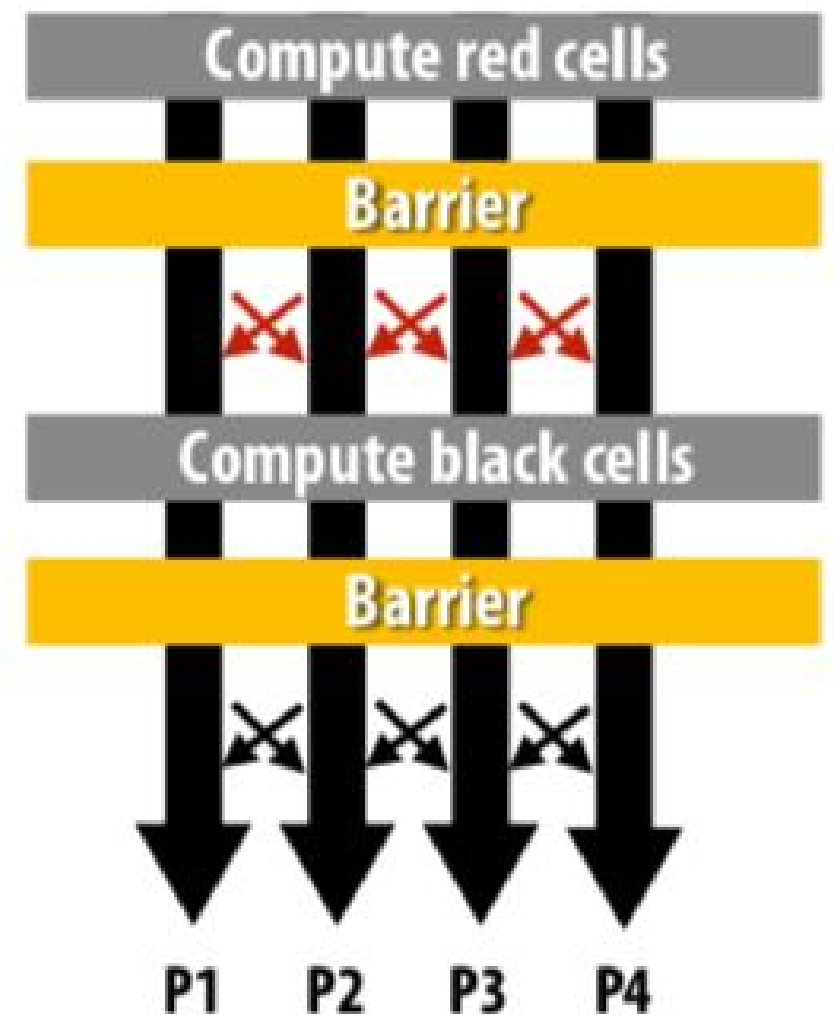
通过在局部累加为**部分和**，然后在迭代结束时全局完成归约来，以提高性能。

将每个进程的结果独立设立局部变量 myDiff，进行局部累加为**部分和**

现在每个线程只锁定一次，而不是每个 (i,j) 循环迭代一次！

Barrier 阻塞式同步原语

- Barrier 是表达依赖关系的保守方式
- Barriers 将计算分为几个阶段
 - Barrier 之前所有线程的所有计算任务归为“上一轮”，
 - Barrier 之后所有线程的所有计算任务归为“下一轮”
- 只有上一轮全部完成，下一轮才能开始



基于共享地址空间实现的求解器


为什么要设立三次Barrier?

```
int    n;                // grid size
bool   done = false;
float  diff = 0.0;
LOCK   myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
    float myDiff;
    int threadId = getThreadId();
    int myMin = 1 + (threadId * n / NUM_PROCESSORS);
    int myMax = myMin + (n / NUM_PROCESSORS)

    while (!done) {
        float myDiff = 0.f;
        diff = 0.f;
        barrier(myBarrier, NUM_PROCESSORS);
        for (j=myMin to myMax) {
            for (i = red cells in this row) {
                float prev = A[i,j];
                A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                               A[i+1,j], A[i,j+1]);
                myDiff += abs(A[i,j] - prev));
            }
            lock(myLock);
            diff += myDiff;
            unlock(myLock);
            barrier(myBarrier, NUM_PROCESSORS);
            if (diff/(n*n) < TOLERANCE)                // check convergence, all threads get same answer
                done = true;
            barrier(myBarrier, NUM_PROCESSORS);
        }
    }
}
```



基于共享地址空间实现的求解器:只有一次barrier

Idea:

通过在连续循环迭代中使用不同的 diff 变量来删除依赖项

Trade off footprint :消除依赖性! (一种常见的并行编程技术)

```
int    n;           // grid size
bool   done = false;
LOCK   myLock;
BARRIER myBarrier;
float diff[3]; // global diff, but now 3 copies

float *A = allocate(n+2, n+2);

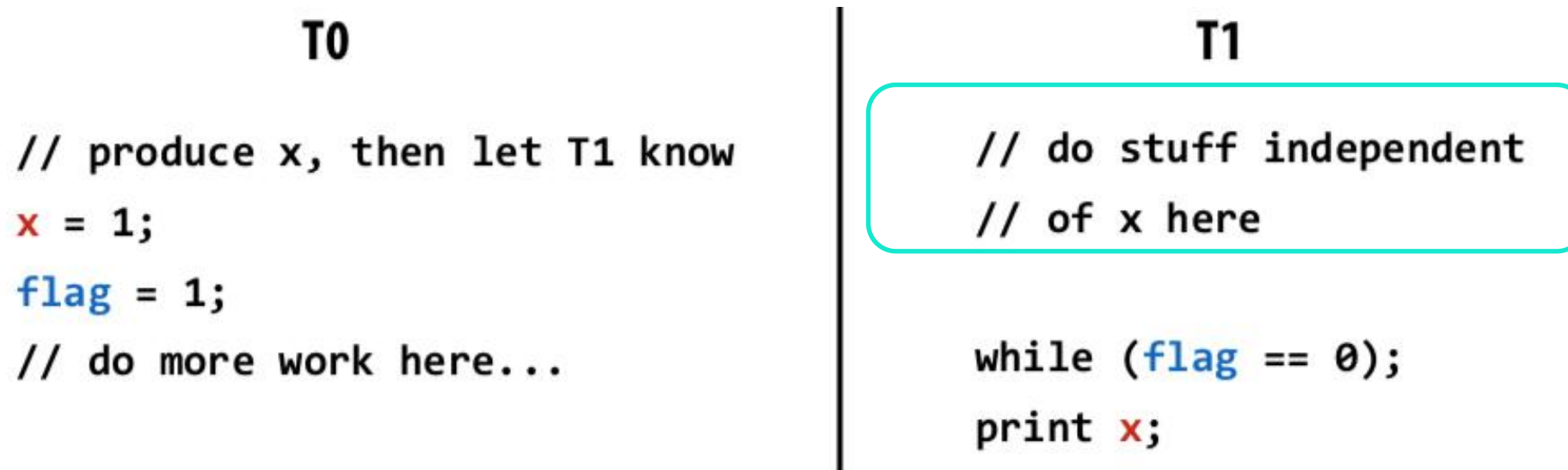
void solve(float* A) {
    float myDiff; // thread local variable
    int index = 0; // thread local variable

    diff[0] = 0.0f;
    barrier(myBarrier, NUM_PROCESSORS); // one-time only: just for init

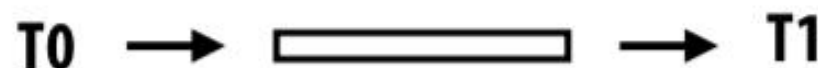
    while (!done) {
        myDiff = 0.0f;
        //
        // perform computation (accumulate locally into myDiff)
        //
        lock(myLock);
        diff[index] += myDiff; // atomically update global diff
        unlock(myLock);
        diff[(index+1) % 3] = 0.0f;
        barrier(myBarrier, NUM_PROCESSORS);
        if (diff[index]/(n*n) < TOLERANCE)
            break;
        index = (index + 1) % 3;
    }
}
```


更多关于指定依赖关系

- Barriers: 简单但保守 (粗粒度依赖)
 - 程序中的所有工作都必须在任何线程开始下一阶段之前完成, 不然下阶段无法开始
- 指定特定的依赖关系可以提高性能 (揭示更多的并行性)
 - 示例: 两个线程, 线程0产生结果 (X), 线程1使用结果X的值



A message queue of length 1



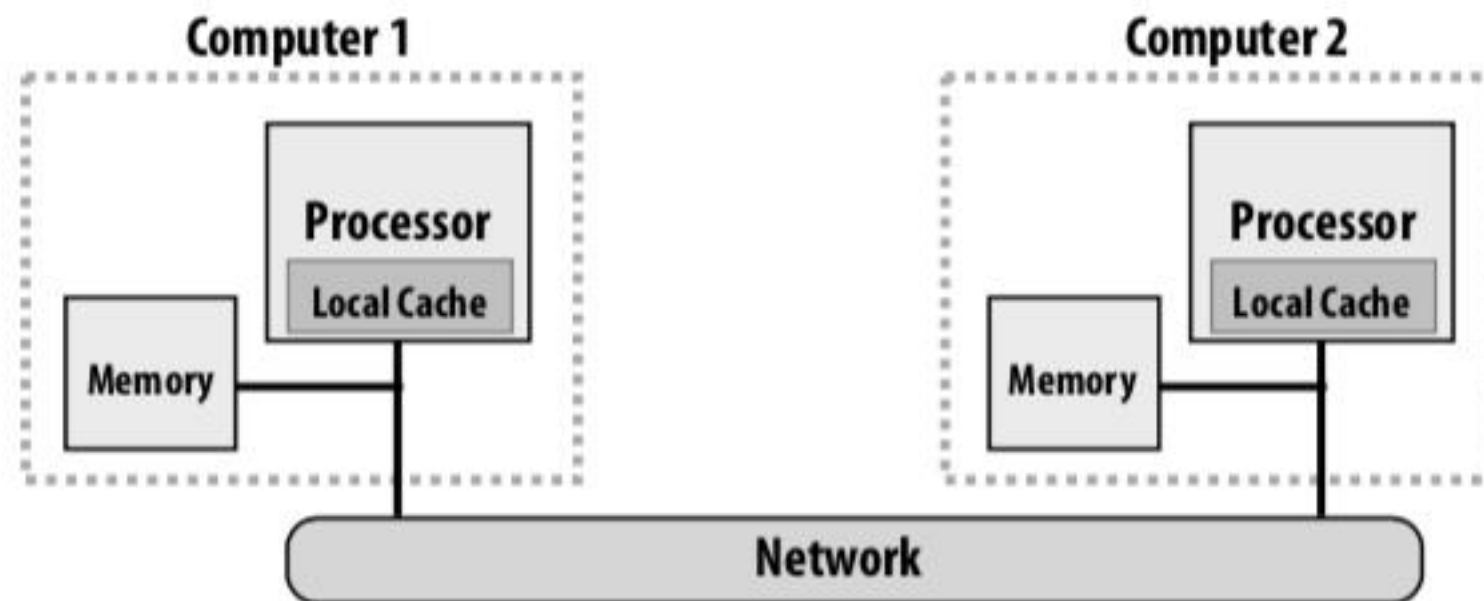
两种编程模型

- 数据并行编程模型
 - 同步
 - 单个逻辑控制线程，但系统可以并行执行 forall 循环的迭代（forall 循环体末尾的隐式barrier）
 - 通信
 - 隐式loads and stores（如共享地址空间）
 - 提供更复杂通信模式的特殊内建编程原语：例如，reduce
- 共享地址空间
 - 同步
 - 共享变量需要互斥（例如，锁）
 - Barriers 用于体现各个计算阶段之间的依赖关系
 - 通信
 - 隐式的 loads/stores，实现对共享变量的通信

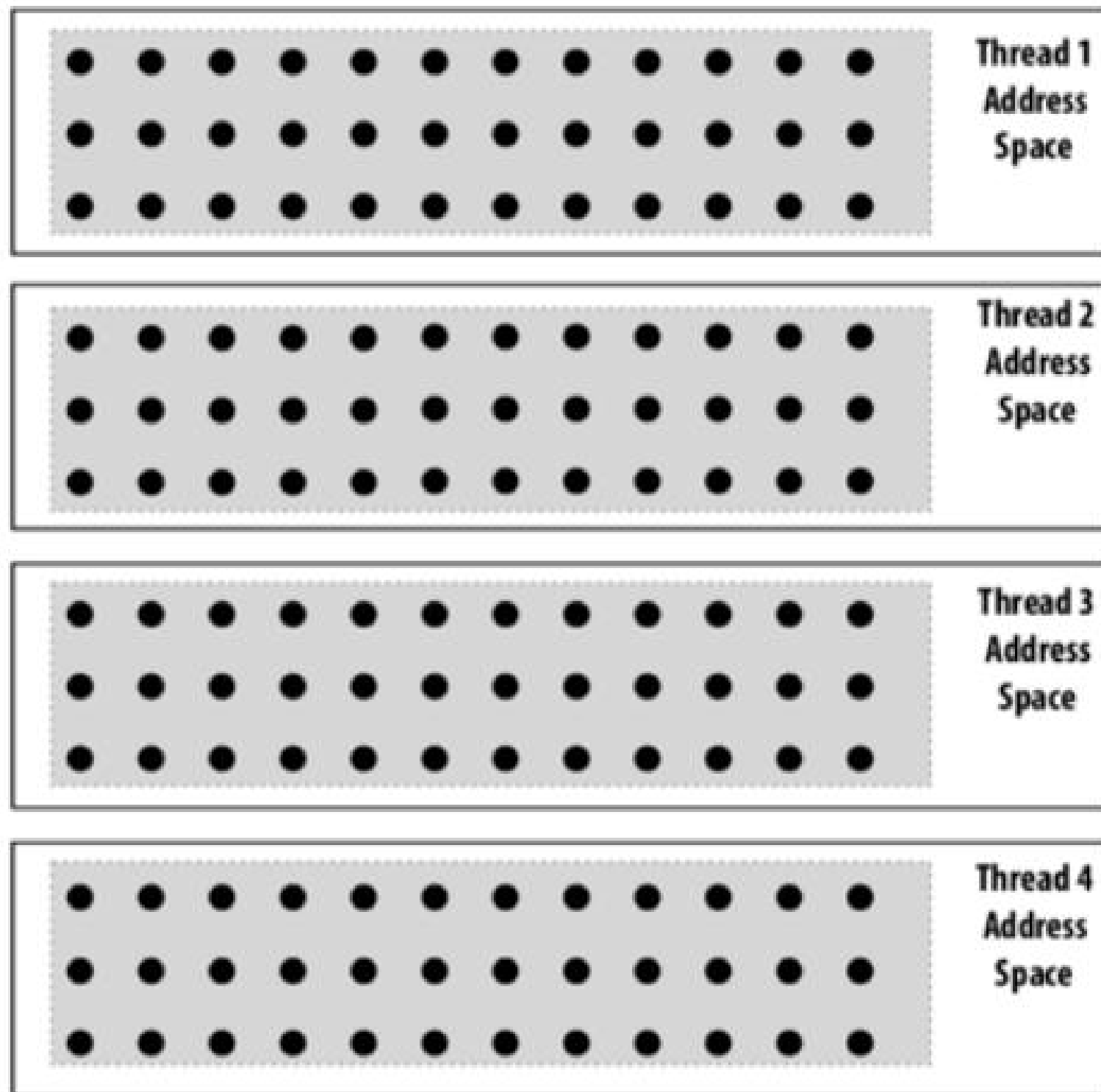
基于消息传递（Message-passing）的求解器

- 每个线程都有自己的地址空间
- 线程通过发送和接收消息进行通信和同步

不同计算节点间的消息传递: 适合于有多个计算节点组成的机群（cluster）



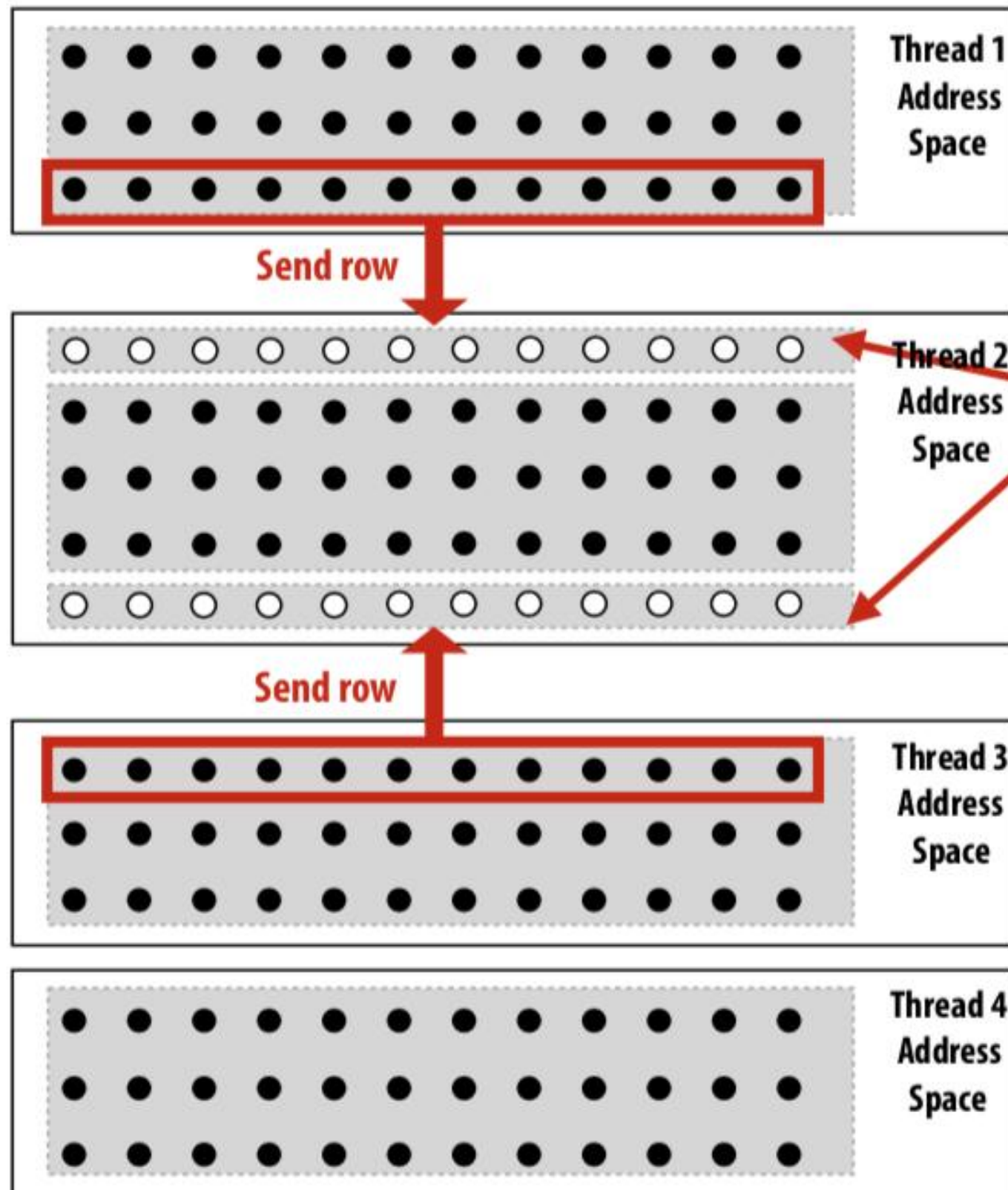
消息传递的编程模型



图中包含四个线程

网格数据被划分为四个独立的数据划分 (allocations), 每个分配驻留在四个不同的独立地址空间中, 分别从属于不同的线程
(每线程的私有数组)

需要数据的复制来实现线程间的通信



Example:

红色任务处理完成后，线程1和线程3发送一行数据给线程2（线程2需要最新的红色节点信息，来更新下一阶段的黑色节点）

“幽灵单元” 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
```

Message passing solver

类似于共享地址空间求解器的结构，但现在在消息发送和接收中通信是显式的

向“邻居线程”发送和接收幽灵行

执行计算（就像求解器的共享地址空间版本一样）

所有线程将本地 my_diff 发送到线程 0

线程 0 计算全局diff，评估是否满足终止条件，并将结果发送回其他线程

```
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);
            }
        }

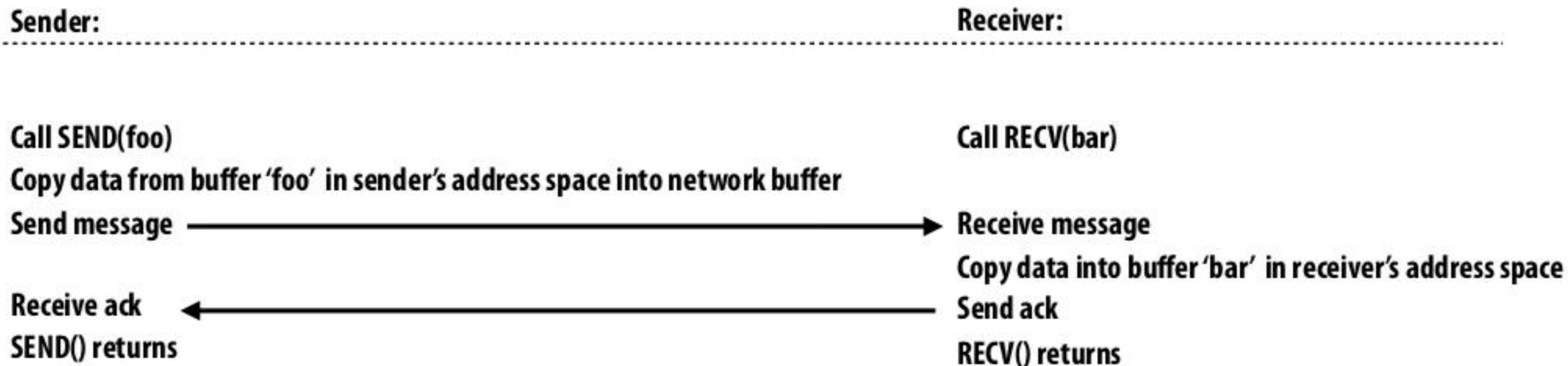
        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
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

消息传递示例的总结

- 计算
 - 数组索引是相对于本地节点的地址空间（不是全局网格坐标）
- 通信
 - 通过查看和接收消息实现节点间通信
 - 批量传输：一次传输整行（不是单个元素）
- 同步
 - 通过发送和接收消息实现进程间的同步
 - 想想如何使用消息实现互斥（mutual exclusion）、屏障（barriers）、标志（flags）
- 为了方便起见，消息传递库通常包含更高级别的原语（通过发送和接收实现）

同步synchronous（阻塞式）发送和接收的编程原语

- send()发送原语: 当发送方收到消息数据驻留在接收方地址空间后, 发送方收到确认信息 (ack), 调用返回
- recv()接收原语: 当接收到的消息中的数据被复制到接收方的地址空间后, 并将确认发送回发送方 (向发送方发送ack动作后), 调用返回



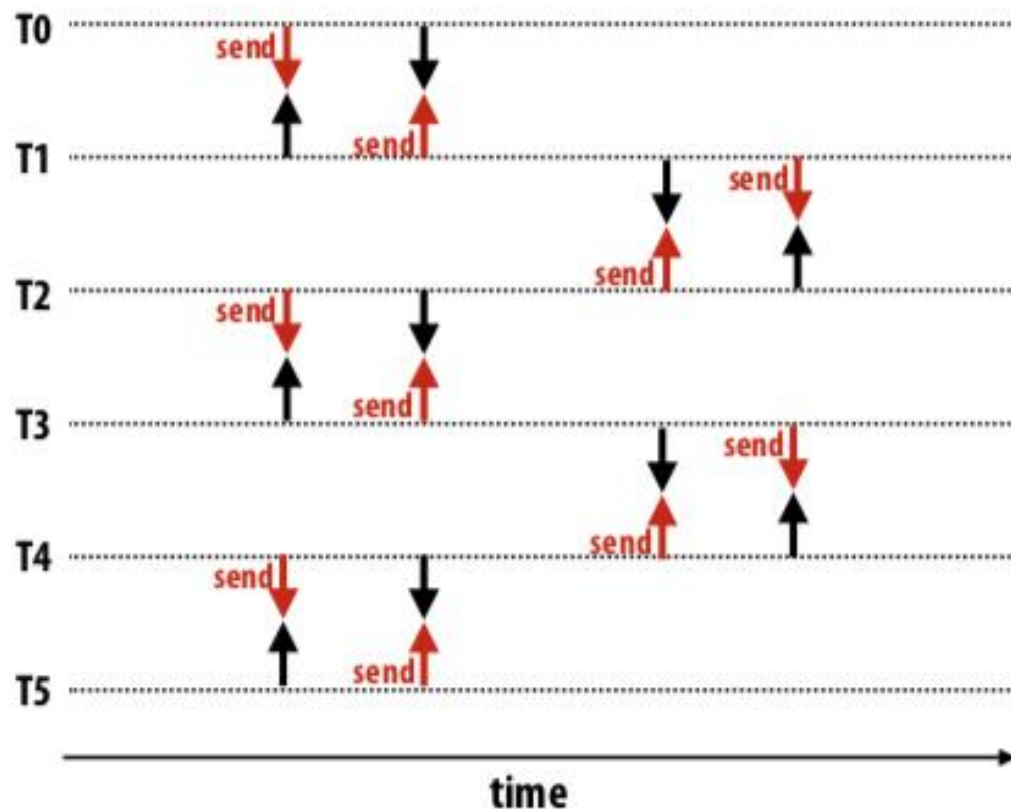
正如在之前的幻灯片中谈到的实现的情况，如果我们的消息传递求解器使用同步发送/接收，则存在一个大问题！

Why

如何解决？

Message passing solver (fixed to avoid deadlock)

向“邻居线程”发送和接收幽灵行
偶数线程send；然后奇数线程
recv，然后再send。
线程超过3个后，会引发死锁



```
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;

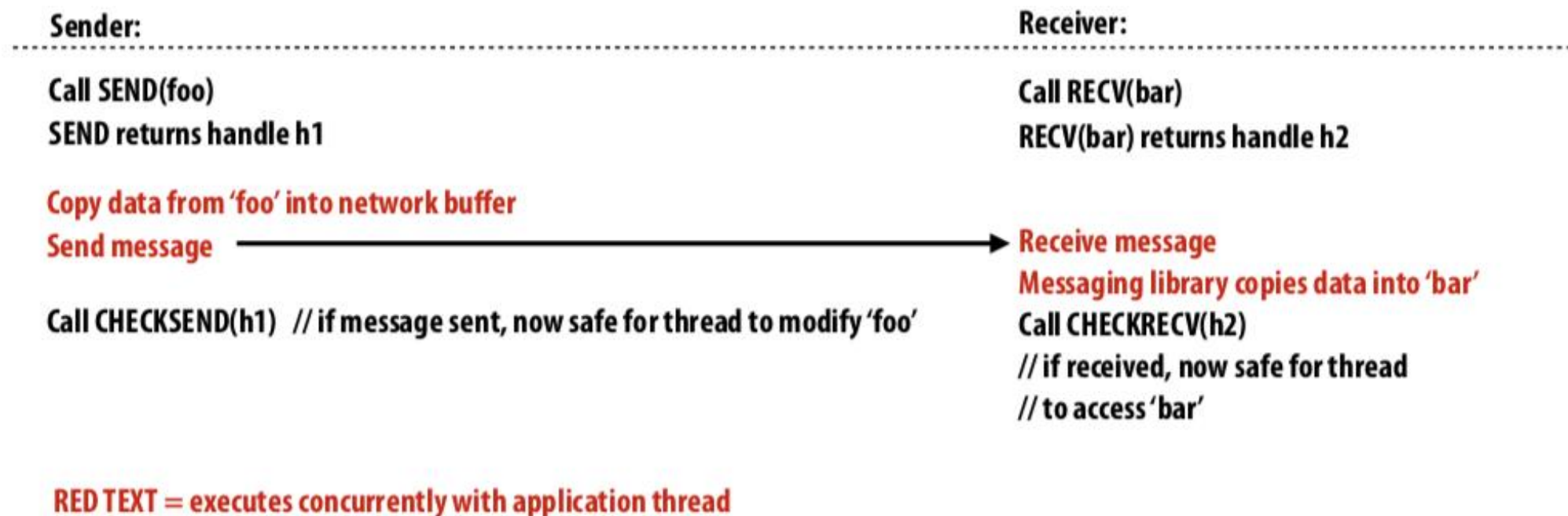
        if (tid % 2 == 0) {
            sendDown(); recvDown();
            sendUp();   recvUp();
        } else {
            recvUp();   sendUp();
            recvDown(); sendDown();
        }

        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;
            if (int i=1; i<get_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSG_ID_DONE);
        }
    }
}
```

异步Asynchronous (非阻塞 Non-blocking)

- send(): 调用立即返回
 - 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(): 发布将来接收的意图, 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



总结

- 创建并行程序的各个方面
 - 通过问题分解创建独立任务，将任务分配给worker，通过任务指派（协调线程处理任务），映射任务到硬件
 - 在接下来的讲座中，我们将深入探讨如何在每个阶段做出更好的程序设计
- 今日重点：识别依赖关系
- 知识点：结合局部性原理，减少同步次数和同步开销