



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

并行程序设计 with 算法

并行算法设计

陶钧

taoj23@mail.sysu.edu.cn

中山大学 计算机学院
国家超级计算广州中心

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

回顾：并行程序设计（Foster方法）及其回答的关键问题

– 如何划分任务？

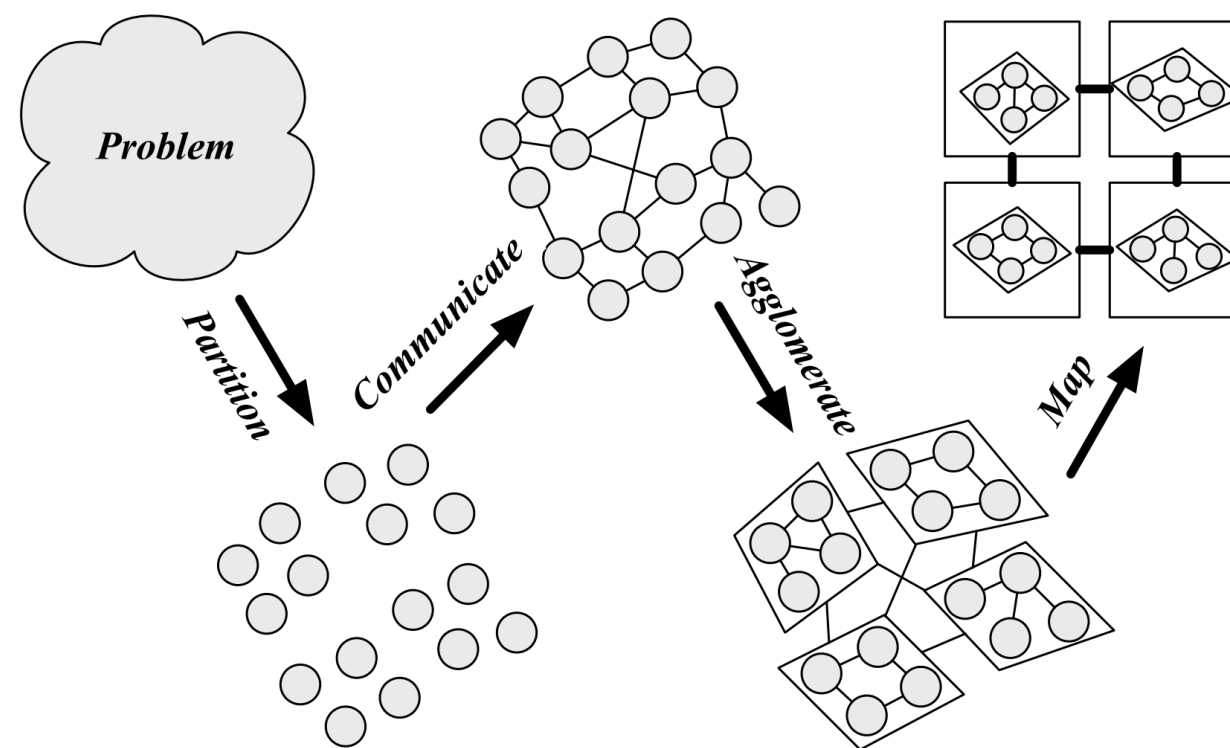
- 哪些任务可以并行？
- 如何分配每个进/线程的工作？
- 任务→子任务→进/线程

– 进/线程间如何合作？

- 子任务之间的依赖关系？
- 进/线程间如何同步？
- 进/线程间如何通信？

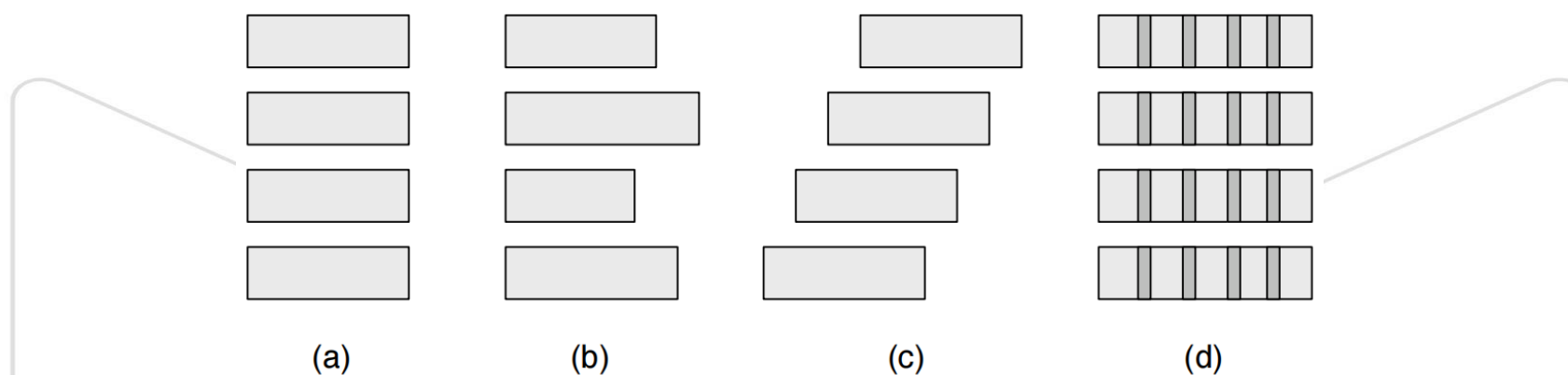
– 如何产生最终结果？

- 子任务与总体任务之间的关系？
- 进/线程的结果如何汇总？



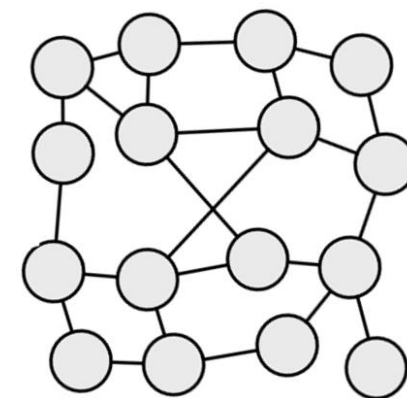
回顾：并行算法性能评估及影响效率的主要因素

- 加速比： $S_p = \frac{T_1}{T_p}$ ，其中串行算法所需时间比使用 p 个计算核心的并行算法时间
- 效率： $E_p = \frac{S_p}{p}$ ，加速比与核心数目的比值
- 影响并行算法效率的主要因素
 - 负载均衡：多个计算节点的工作量是否均衡？
 - 并发度：多个计算节点是否同时工作？
 - 并行开销：由于并行带来的额外开销（通信与同步等）

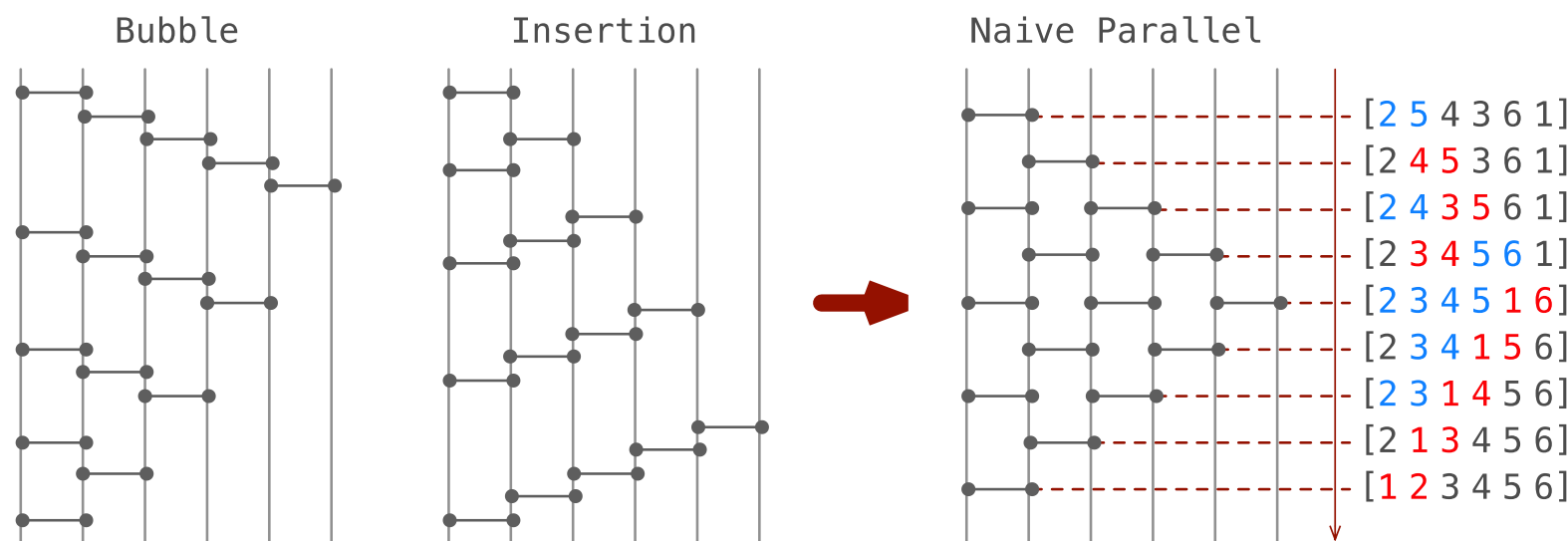


并行算法性能评估：如何从实现角度分析？

- 内存开销 M_p ：所有计算节点上消耗的内存之和
 - 若 $M_p = M_1$ ，则算法具有最高的内存效率
 - 实践中往往 $M_p > M_1$ ，可能带来额外的性能增长（甚至效率 > 100%）
- 工作量 Q_p ：所有计算节点上消耗的运算次数之和
 - 并行开销 $O_p = Q_p - Q_1$
 - 包括数值运算开销，节点内的数据访存开销，节点间的通信、同步开销
- 深度：任务图中的最常路径长度
 - 参考此前Amdahl's Law的不可并行部分
 - 实际的加速比还取决于可用的计算核心数目
- 时间：各类运算消耗的总时间
 - 与工作量相似，但也有区别
 - 例如每个节点上的内存开销及访存模式可能影响其实际执行过程中需要的时间



- 任务向另一任务提供计算所必须的数据：依赖关系
 - 通过任务剖析发现虚假的“依赖关系”



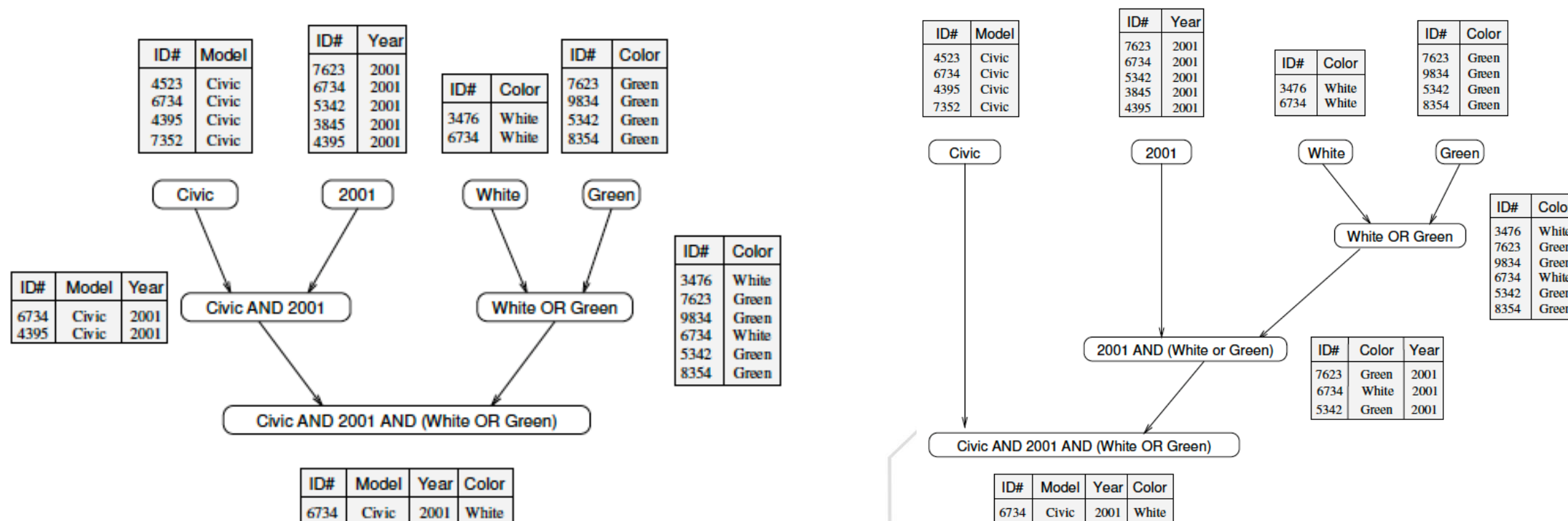
- 通过改变计算方式消除非必须的“依赖关系”

- $s^2 = \frac{1}{n-1} \sum (x_i - \bar{x})^2$: 需串行两次规约，平方和计算依赖于均值计算
- $s^2 = \frac{1}{n-1} (\sum x_i^2 - \frac{1}{n} (\sum x_i)^2)$: 两次规约可并行（甚至可以合并成一次）

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

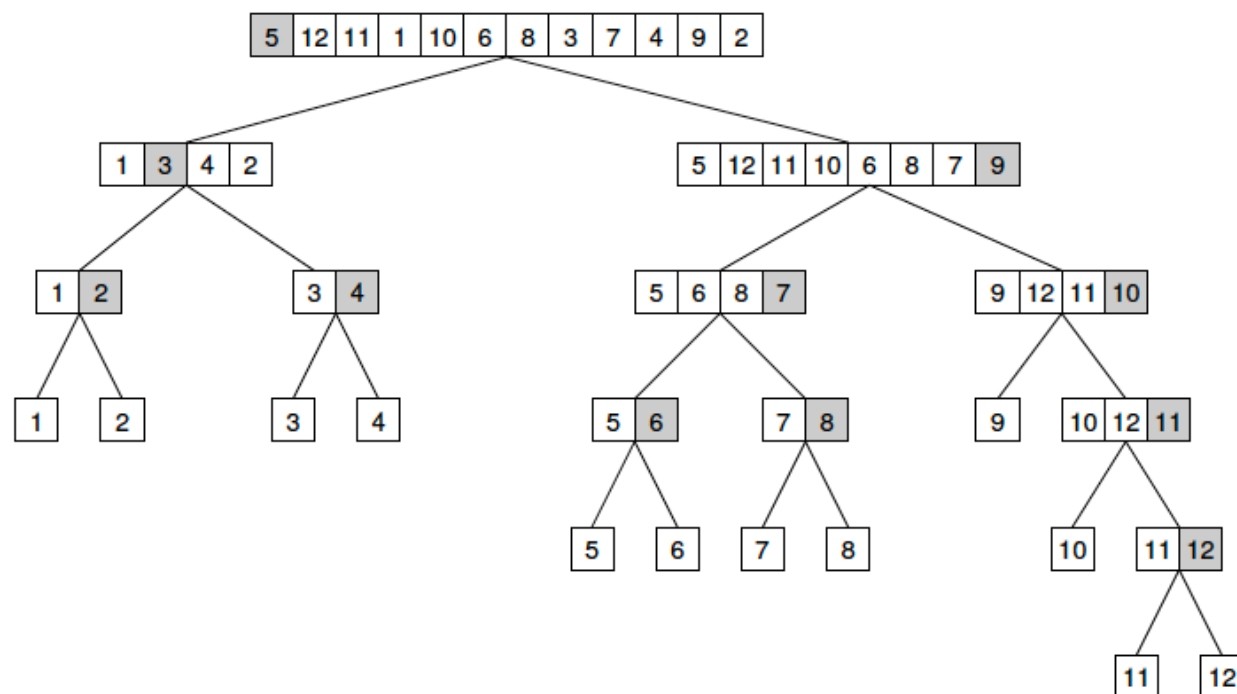
递归划分策略

- 通常适用于并行化所有分治法算法
 - 问题能被拆分为子问题（子问题的解有助于解决原始问题）
- 不断递归直到问题被划分至合适的颗粒度
- 例1：下图中所表示的数据库查询操作哪个更好？



递归划分策略

- 通常适用于并行化所有分治法算法
 - 问题能被拆分成为子问题（子问题的解有助于解决原始问题）
- 不断递归直到问题被划分至合适的颗粒度
- 例2：快速排序 vs 归并排序：哪个更适合并行？



数据划分

- 对数据进行划分，并由拥有数据的计算节点完成相关计算任务
 - Owner compute rule
- 例：数据库查询
- 基于输出数据划分
 - 只需将结果连接

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		3
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		2
	F, G, H, K,		C, D		1
	A, E, F, K, L		D, K		2
	B, C, D, G, H, L		B, C, F		0
	G, H, L		C, D, K		0
	D, E, F, K, L				
	F, G, H, L				

(b) Partitioning the frequencies (and itemsets) among the tasks

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		3
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		2
	F, G, H, K,				
	A, E, F, K, L				
	B, C, D, G, H, L				
	G, H, L				
	D, E, F, K, L				
	F, G, H, L				

task 1

Database Transactions	A, B, C, E, G, H	Itemsets	C, D	Itemset Frequency	1
	B, D, E, F, K, L		D, K		2
	A, B, F, H, L		B, C, F		0
	D, E, F, H		C, D, K		0
	F, G, H, K,				
	A, E, F, K, L				
	B, C, D, G, H, L				
	G, H, L				
	D, E, F, K, L				
	F, G, H, L				

task 2

数据划分

- 对数据进行划分，并由拥有数据的计算节点完成相关计算任务
 - Owner compute rule
- 例：数据库查询
- 基于输入数据划分：需对结果求和

Partitioning the transactions among the tasks

task 1			task 2		
Database Transactions	Itemsets	Itemset Frequency	Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1		A, B, C	0
B, D, E, F, K, L	D, E	2		D, E	1
A, B, F, H, L	C, F, G	0		C, F, G	0
D, E, F, H	A, E	1	A, E, F, K, L	A, E	1
F, G, H, K,	C, D	0	B, C, D, G, H, L	C, D	1
	D, K	1	G, H, L	D, K	1
	B, C, F	0	D, E, F, K, L	B, C, F	0
	C, D, K	0	F, G, H, L	C, D, K	0

数据划分

- 对数据进行划分，并由拥有数据的计算节点完成相关计算任务
 - Owner compute rule
- 例：数据库查询
- 输入及输出数据划分
 - 并发度更高
 - 但通信开销也更高
 - 需连接、求和

Partitioning both transactions and frequencies among the tasks

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		2
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		1
	F, G, H, K,				

task 1

Database Transactions	A, B, C, E, G, H	Itemsets		Itemset Frequency	
	B, D, E, F, K, L				
	A, B, F, H, L				
	D, E, F, H				
	F, G, H, K,		C, D		0
			D, K		1

task 2

Database Transactions		Itemsets	A, B, C	Itemset Frequency	0
			D, E		1
			C, F, G		0
			A, E		1
	A, E, F, K, L				
	B, C, D, G, H, L				

task 3

Database Transactions	A, E, F, K, L	Itemsets		Itemset Frequency	
	B, C, D, G, H, L		C, D		1
	G, H, L		D, K		1
	D, E, F, K, L		B, C, F		0
	F, G, H, L		C, D, K		0

task 4

探索式划分

- 某些问题求解过程中的计算量是不确定的，因此无法提前规划
- 常见例子为最优解的搜索，使用并行计算可同时探索多条路径
 - 但无法提前规划所有路径
- 例：滑动拼图
 - 每个计算节点负责从一个特定初始状态出发搜索到最优解的路径

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

(c)

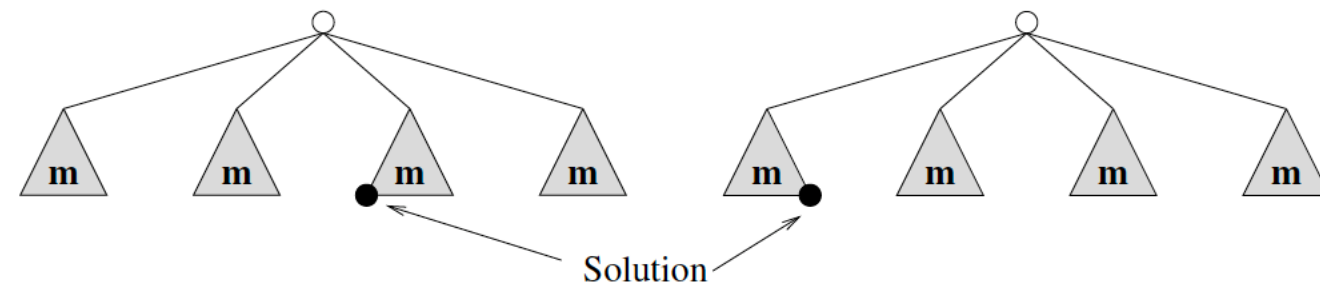
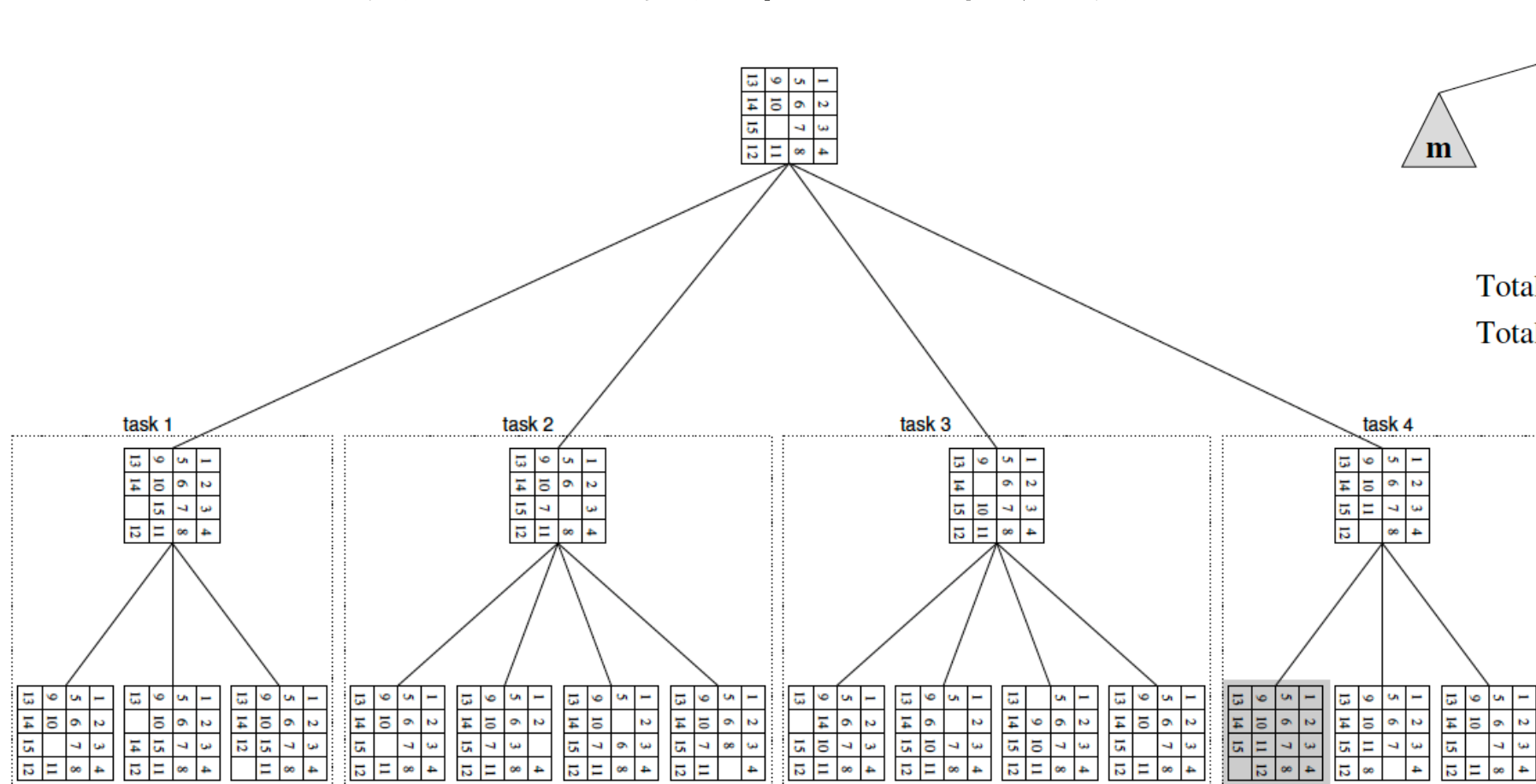
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(d)

探索式划分

— 例：滑动拼图

- 每个计算节点负责从一个特定初始状态出发搜索到最优解的路径
- 搜索开销可能取决于难以预知的路径



Total serial work: $2m+1$

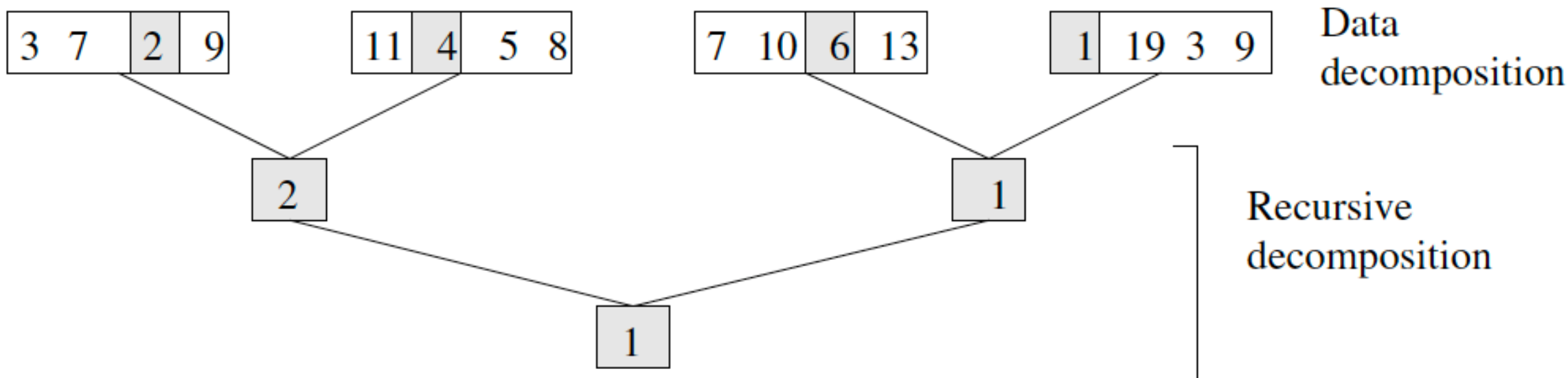
Total parallel work: 1

Total serial work: m

Total parallel work: $4m$

混合划分策略

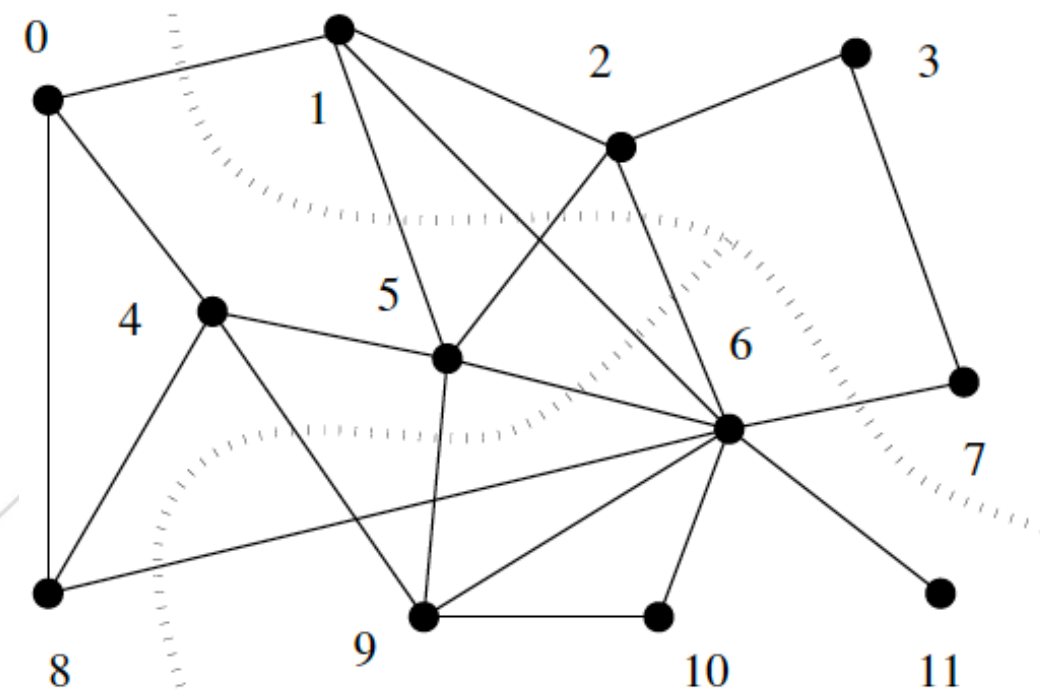
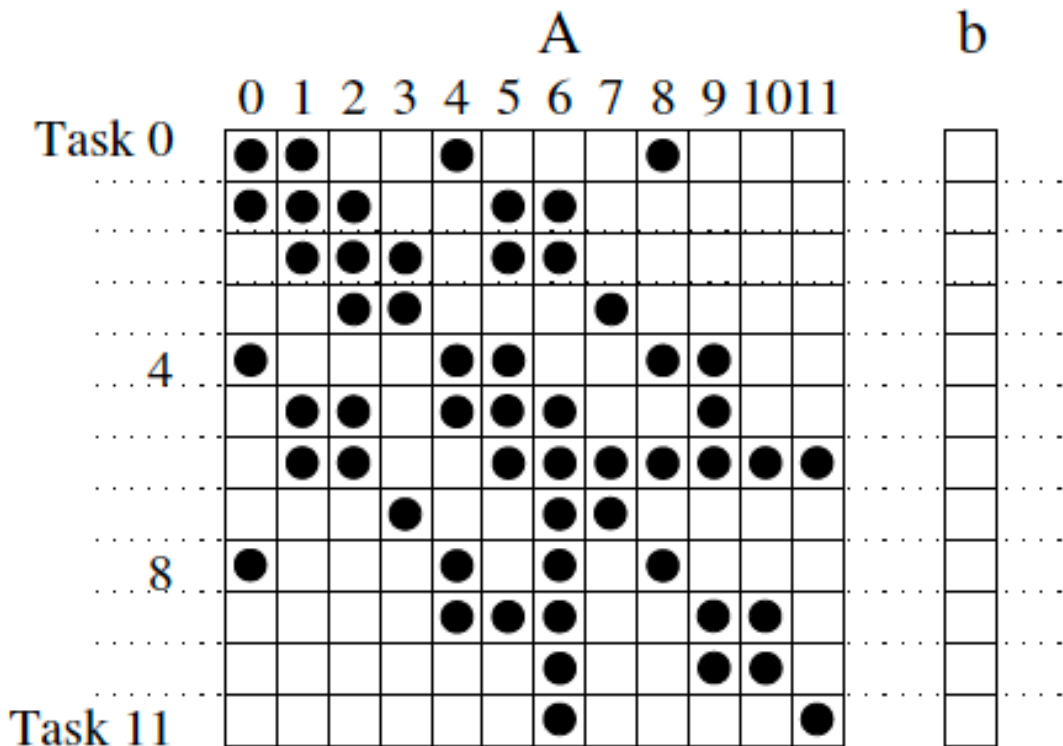
- 单一的划分策略往往无法取得最优效果
 - 快速排序中，递归划分并发度往往并不高
 - 并行归约中，为提高占用率，往往每个线程先分别进行串行归约
 - Brent's theorem



基于依赖关系的任务划分

— 常在任务图关系上通过图算法或优化方法划分节点

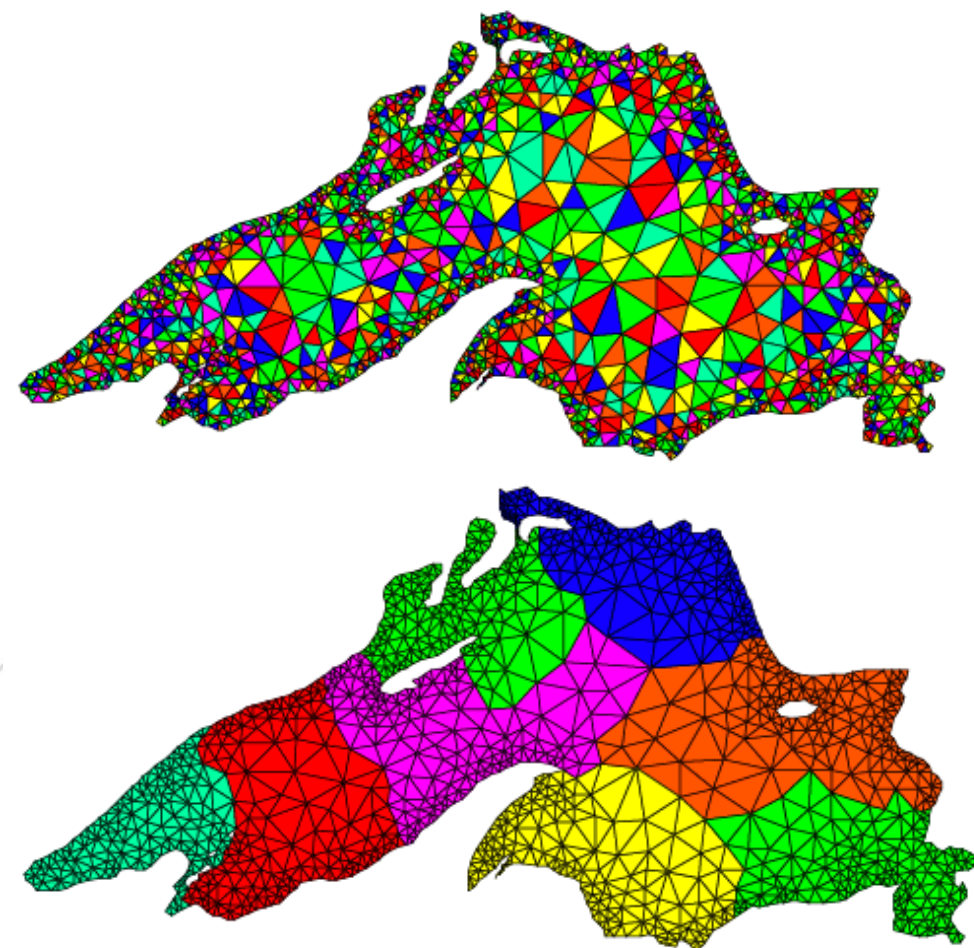
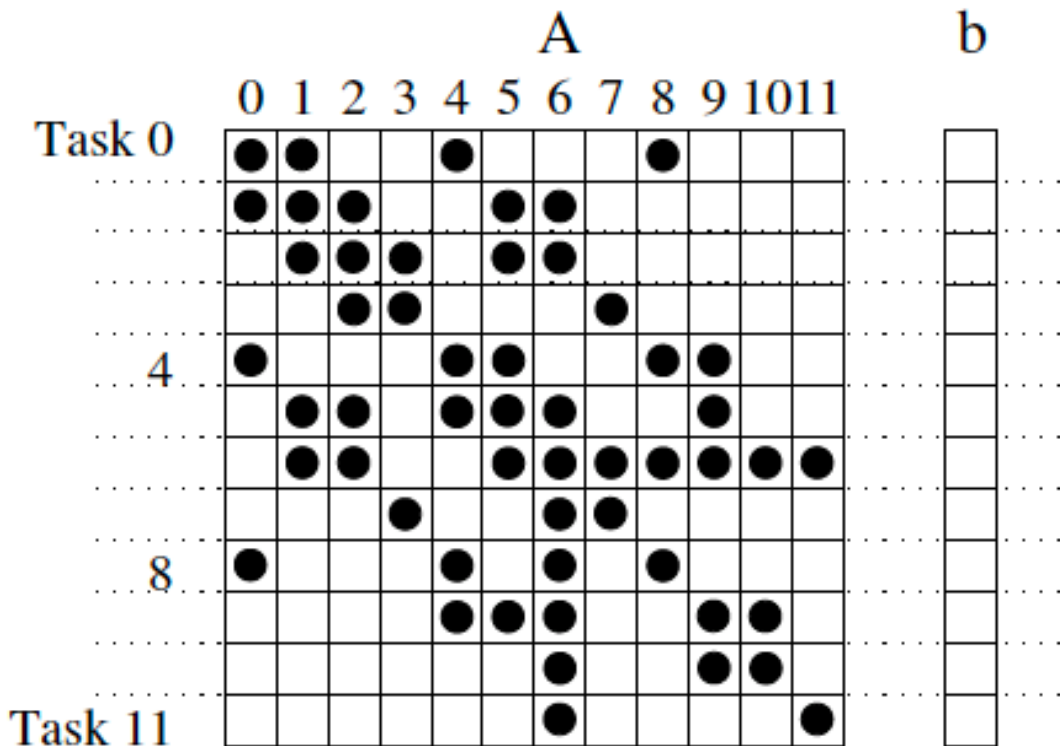
- 最大化数据局部性（减少数据重载入/通信）
- 最小化空闲时间（提高并发度）
- 最小化最大负载（负载均衡）



基于依赖关系的任务划分

— 常在任务图关系上通过图算法或优化方法划分节点

- 子任务编号连续 \neq 子任务相关度高
- 图算法：社群检测算法/最大流最小割
- 优化方法：最小化某种目标函数



◉ 静态划分 vs 动态划分

- 以上方式大多为静态划分
- 实践中为了保持负载均衡，常在运行过程中调整计算节点任务
 - 多用于探索式划分及基于任务图的划分

◉ 任务划分的实现模型

- 主从（master-slave）模型：也可用于静态
 - 一个或多个主进程产生任务，从进程执行计算
 - 任务池模型
- 管线/生产者-消费者模型
 - 一个或一组进程负责数据处理管线中的一个环节

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

• N体问题

– 定义：对于 n 个相互作用的粒子，模拟其运动过程

- 广泛应用：天文学，化学...
- 动态的过程可通过对每一个瞬态状态进行描述

– 每个粒子 q 在时刻 t 的运动状态

– 位置 $s_q(t)$ ，速度 $v_q(t)$

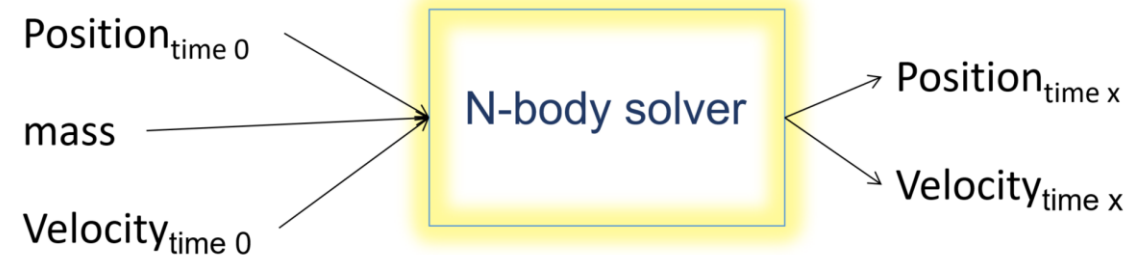
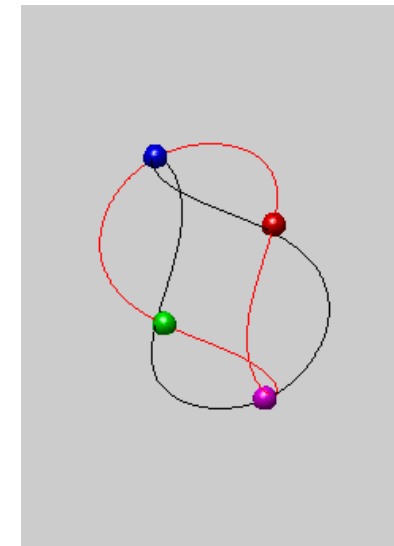
- 模拟过程

– 给定每个粒子的质量： $\{m_q\}$

– 及粒子的初始状态： $\{s_q(0), v_q(0)\}$

– 依次输出每个时刻 t 上的粒子状态

» $\{s_q(t), v_q(t)\} \rightarrow \{s_q(t + \Delta t), v_q(t + \Delta t)\}$



模拟假设

- 牛顿第二运动定律: $F = ma$
- 粒子间相互作用力满足万有引力定律: $F = G \frac{m_q m_k}{r^2}$

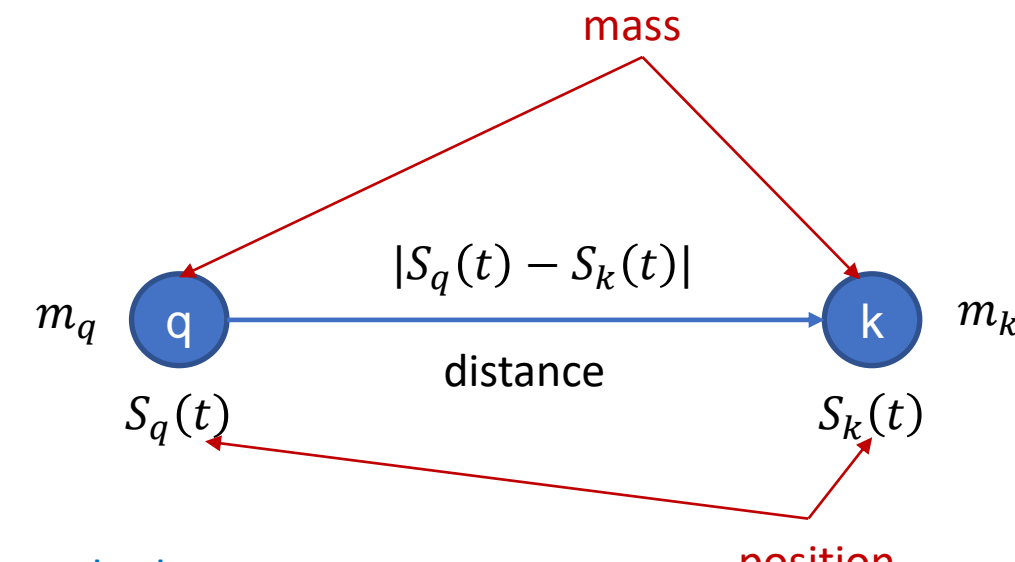


Diagram illustrating the interaction between two particles, q and k , with masses m_q and m_k respectively. Their positions are $s_q(t)$ and $s_k(t)$. The distance between them is $|s_q(t) - s_k(t)|$.

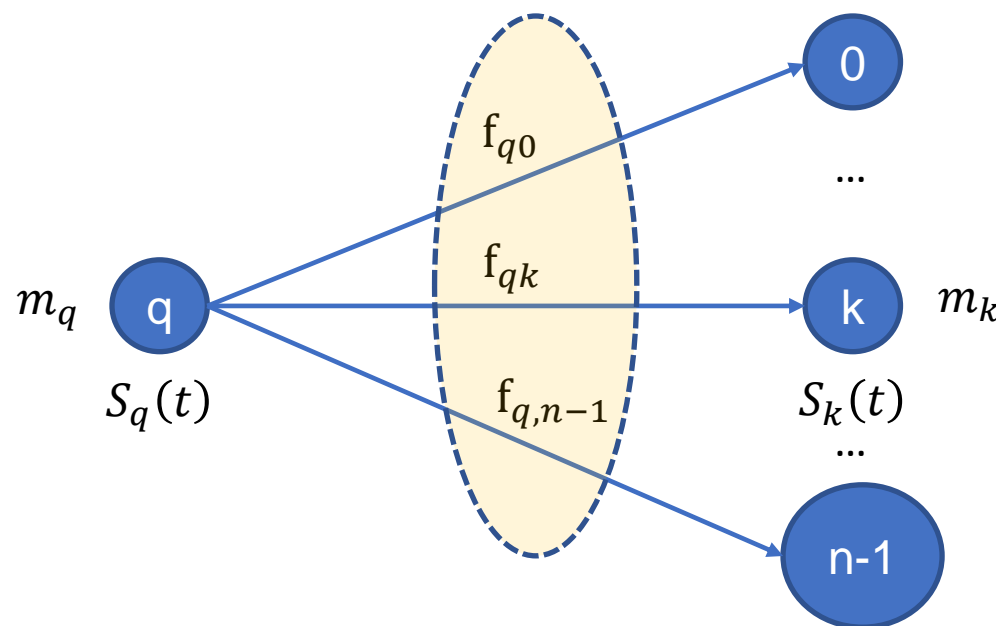
The gravitational force $\mathbf{f}_{qk}(t)$ is given by the equation (6.1):

$$\mathbf{f}_{qk}(t) = - \frac{G m_q m_k}{|s_q(t) - s_k(t)|^3} [s_q(t) - s_k(t)] \quad (6.1)$$

Where G is the gravitational constant.

模拟假设

– 粒子 q 所受合力为 q 与其他每个粒子 k 之间作用力的总和



$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3} [\mathbf{s}_q(t) - \mathbf{s}_k(t)] \quad (6.2)$$

模拟假设

– 根据牛顿第二定律可推导粒子的运动状态

$$F_q(t) = m_q a_q(t) = m_q S_q''(t)$$



$$a_q(t) = S_q''(t) = \frac{F_q(t)}{m_q}$$



$$s_q''(t) = -G \sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{|\mathbf{s}_q(t) - \mathbf{s}_j(t)|^3} [\mathbf{s}_q(t) - \mathbf{s}_j(t)] \quad (6.3)$$

$t = 0, \Delta t, 2\Delta t, \dots, T\Delta t$

• 实现（伪码）

```
Get input data;  
for each timestep {  
    if (timestep output) Print positions and velocities of particles;  
    for each particle q  
        Compute total force on q;  
        for each particle q  
            Compute position and velocity of q;  
}  
Print positions and velocities of particles;
```



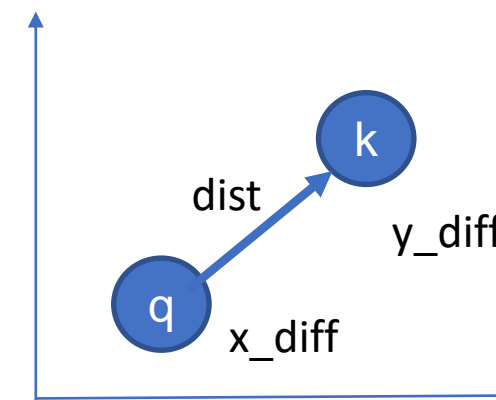
实现（计算作用力）

– 可自定义向量类型结构体简化代码

- 使用内联实现同时按分量名及分量下标访问

```
union vec2 {
    struct {
        float x, y;
    };
    float c[2];
};
```

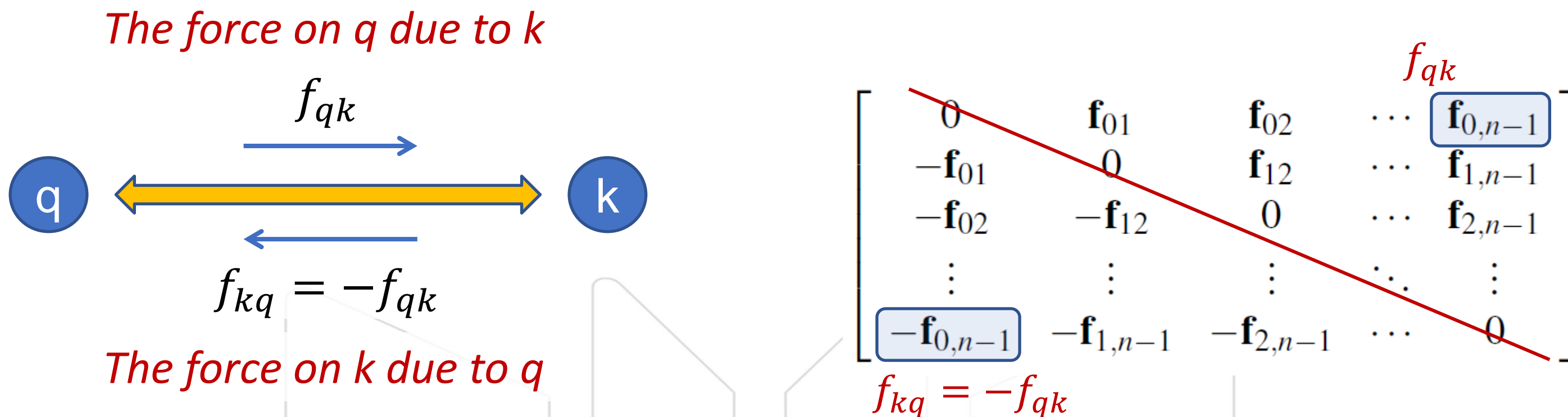
```
for each particle q {
    for each particle k != q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}
```



$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3} [\mathbf{s}_q(t) - \mathbf{s}_k(t)]$$

实现（计算作用力）

- 力的作用是相互的，因此对于一对粒子，只需计算一次作用力
 - 将粒子间的相互作用力存储于一个矩阵时，只需要计算上/下三角矩阵



实现（计算作用力）

- 力的作用是相互的，因此对于一对粒子，只需计算一次作用力
 - 计算后同时更新两个粒子所受到的合力（简化算法）

$$\begin{bmatrix}
 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\
 -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\
 -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\
 \vdots & \vdots & \vdots & \ddots & \vdots \\
 -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0
 \end{bmatrix}$$

```

for each particle q
  forces[q] = 0;
for each particle q {
  for each particle k > q {
    x_diff = pos[q][X] - pos[k][X];
    y_diff = pos[q][Y] - pos[k][Y];
    dist = sqrt(x_diff*x_diff + y_diff*y_diff);
    dist_cubed = dist*dist*dist;
    force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
    force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

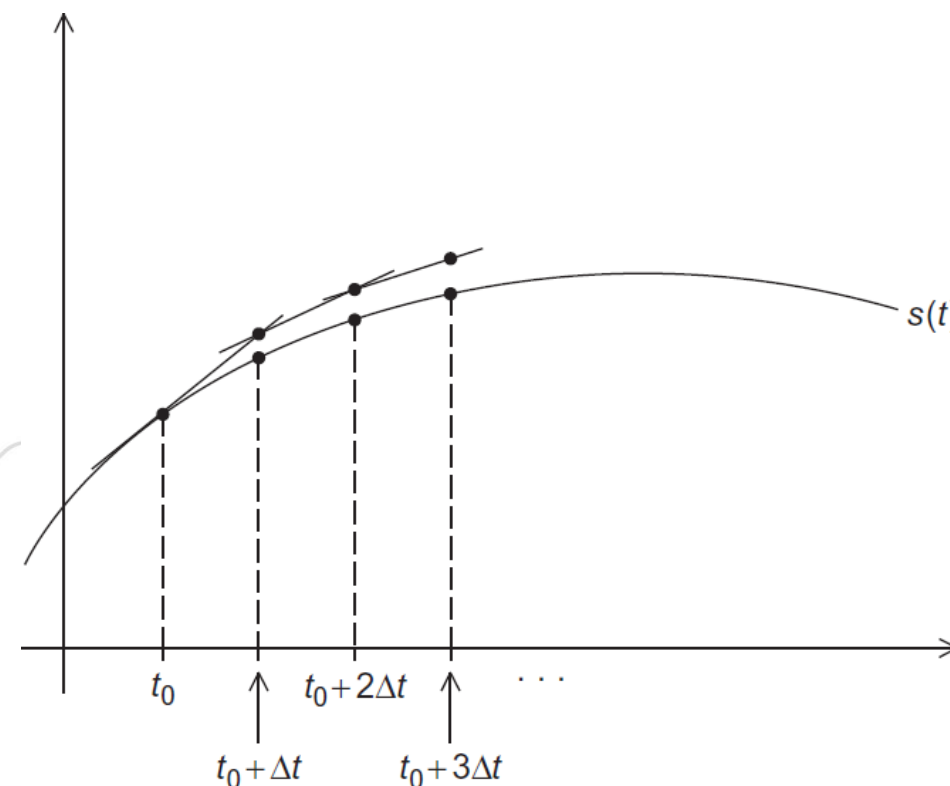
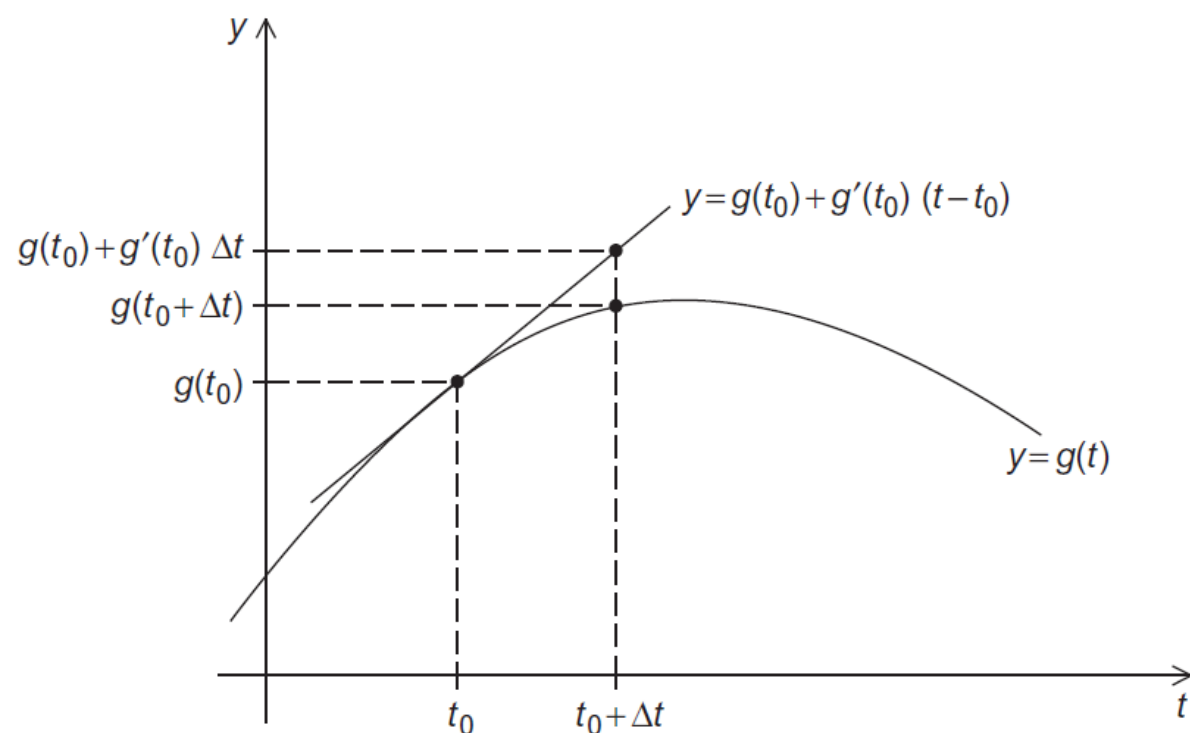
    forces[q][X] += force_qk[X];
    forces[q][Y] += force_qk[Y];
    forces[k][X] -= force_qk[X];
    forces[k][Y] -= force_qk[Y];
  }
}

```

实现（状态更新）

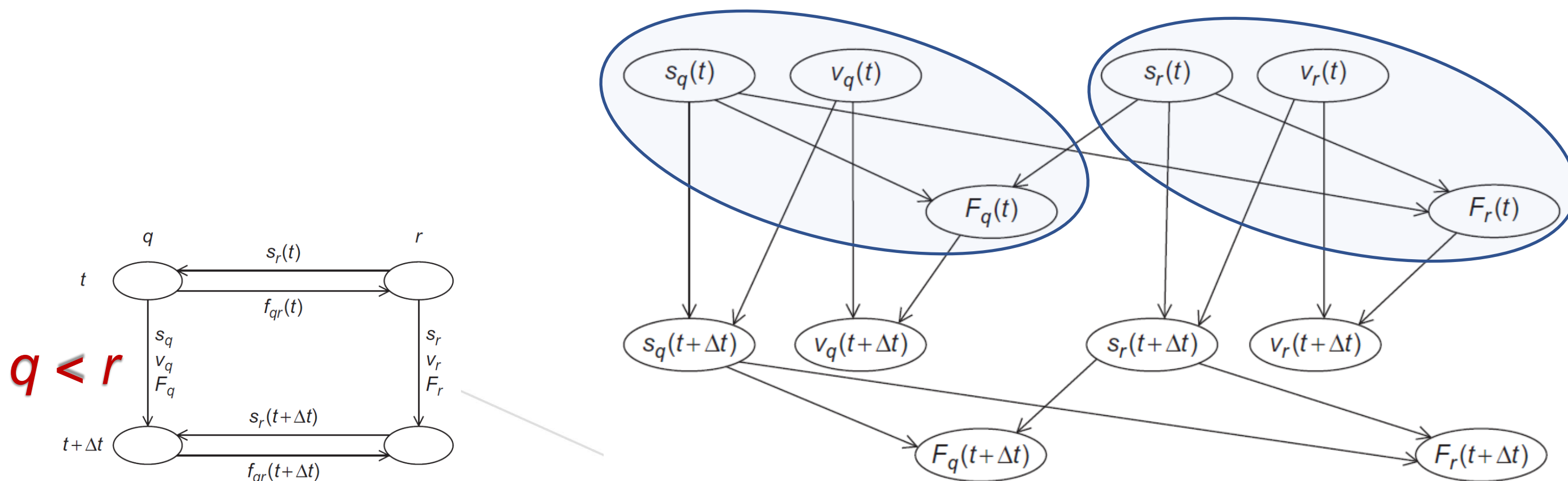
– 作用力计算给出了加速度，如何用其更新位置与速度？

- 加速度为速度的导数，而速度为位置的导数
- 近似：加速度反映了速度的变化量，速度反映了位置的变化量
- 计算（欧拉方法）：假设极小范围内，该变化量为常数



并行实现（分析）： Foster's methodology

- 任务：计算 t 时刻每个粒子 q 所受合力，并以此更新其位置及速度
- 通信：粒子间两两关联



- 并行实现（分析）： Foster's methodology

- 任务： 计算 t 时刻每个粒子 q 所受合力，并以此更新其位置及速度
- 通信： 粒子间两两关联
- 聚合&映射： 使用多个计算核心并行完成两个内层循环的计算

```
for each timestep {  
    if (timestep output) Print positions and velocities of particles  
    for each particle q  
        Compute total force on q;  
    for each particle q  
        Compute position and velocity of q;  
}
```



iterating over particles

• 并行实现（分析）： Foster's methodology

- 任务： 计算 t 时刻每个粒子 q 所受合力，并以此更新其位置及速度
- 通信： 粒子间两两关联
- 聚合&映射： 使用多个计算核心并行完成两个内层循环的计算
 - 使用OpenMP实现，只需在循环前加上`#pragma omp parallel for`

```
for each timestep {  
    if (timestep output) Print positions and velocities of particles;  
#    pragma omp parallel for  
    for each particle q  
        Compute total force on q;  
#    pragma omp parallel for  
    for each particle q  
        Compute position and velocity of q;  
}
```

• OpenMP实现（常规版本）

- 力计算：每个线程计算一个粒子所受合力
- 位置及速度更新：每个线程更新一个粒子的位置及速度
- 对于每一步而言，循环之间没有依赖关系

```
# pragma omp parallel for
for each particle q {
    forces[q][X] = forces[q][Y] = 0;
    for each particle k != q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}

# pragma omp parallel for
for each particle q {
    pos[q][X] += delta_t*vel[q][X];
    pos[q][Y] += delta_t*vel[q][Y];
    vel[q][X] += delta_t/masses[q]*forces[q][X];
    vel[q][Y] += delta_t/masses[q]*forces[q][Y];
}
```

OpenMP实现（常规版本）

- 力计算：每个线程计算一个粒子所受合力
- 位置及速度更新：每个线程更新一个粒子的位置及速度
- 对于每一步而言，循环之间没有依赖关系

```
# pragma omp parallel ← 使用omp parallel, 重用线程
  for each timestep {
    if (timestep output) Print positions and velocities of particles;
# pragma omp for
    for each particle q
      Compute total force on q;
# pragma omp for
    for each particle q
      Compute position and velocity of q;
  }
```

问题：每个线程都将输出位置及速度

OpenMP实现（常规版本）

- 力计算：每个线程计算一个粒子所受合力
- 位置及速度更新：每个线程更新一个粒子的位置及速度
- 对于每一步而言，循环之间没有依赖关系

```
# pragma omp parallel
for each timestep {
    if (timestep output) {
#         pragma omp single
            Print positions and velocities of particles;
    }

#     pragma omp for
    for each particle q
        Compute total force on q;
#     pragma omp for
    for each particle q
        Compute position and velocity of q;
}
```

使用single保证只有一个线程输出

OpenMP实现（常规版本）

- 力计算：每个线程计算一个粒子所受合力
- 位置及速度更新：每个线程更新一个粒子的位置及速度
- 对于每一步而言，循环之

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

数据形状不规则
可能影响依赖关系
及负载均衡

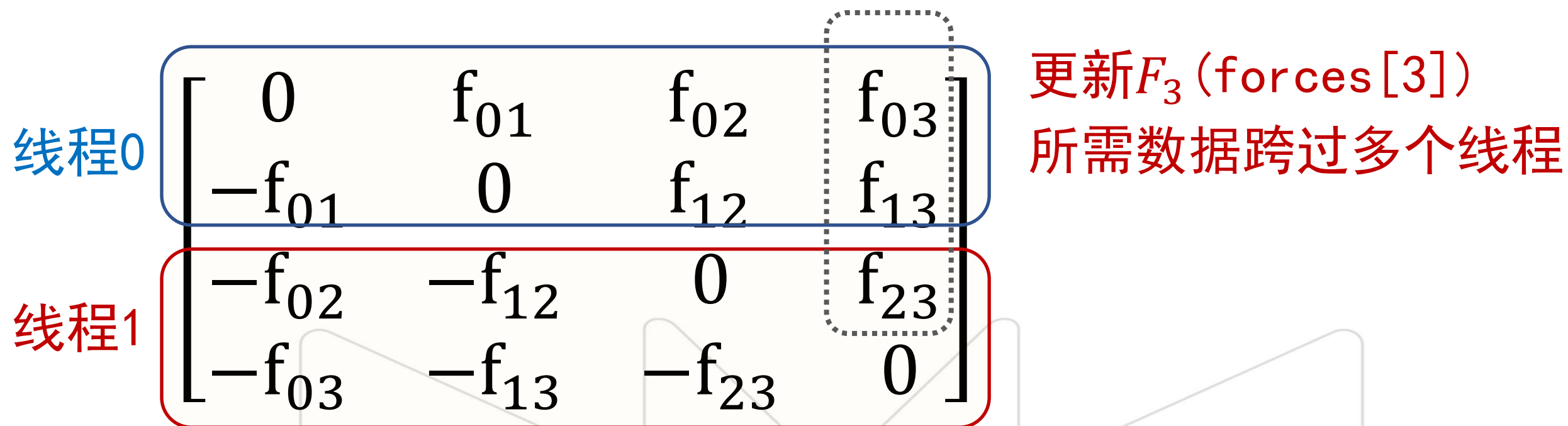
```
# pragma omp parallel
for each timestep {
    if (timestep output) {
        pragma omp single
        Print positions and velocities of particles;
    }
    pragma omp for
    for each particle q
        forces[q] = 0.0;
    pragma omp for
    for each particle q
        Compute total force on q;
    pragma omp for
    for each particle q
        Compute position and velocity of q;
}
```

没有依赖关系
可以直接并行

• OpenMP实现（简化算法）：依赖关系分析

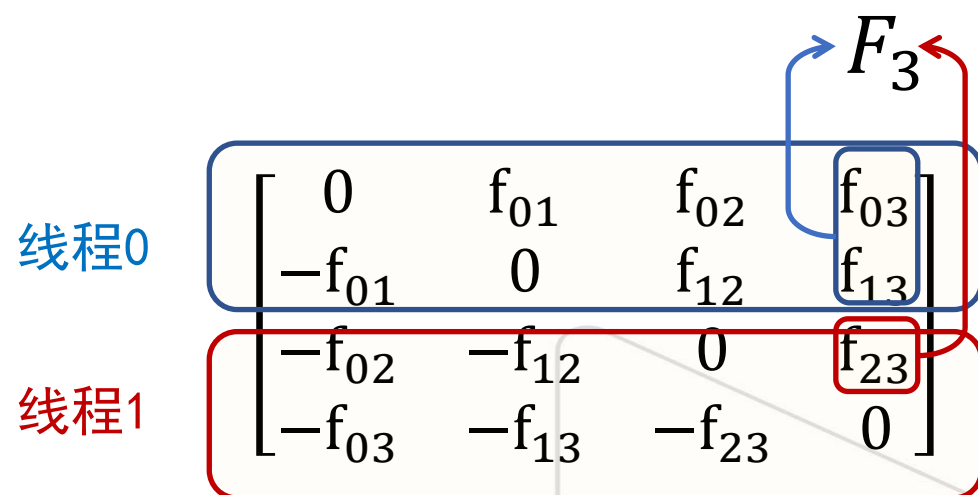
– 力计算：每个线程计算一个粒子所受合力

- 以粒子3所受合力为例： $F_3 = -f_{03} - f_{13} - f_{23}$
- 更新forces[3]时可能出现竞争条件（对于共享内存并行框架而言）



OpenMP实现（简化算法：依赖关系分析）

- 更新`forces[q]`时可能出现竞争条件
- 解决方案1：使用临界区保护对`forces[]`数组的更新
 - 计算`force_qk`后，由对应该线程线程更新`forces[q]`及`forces[k]`
 - 更新可能被高度串行化



```
# pragma omp critical
```

```
{
```

```
    forces[q][X] += force_qk[X];
```

```
    forces[q][Y] += force_qk[Y];
```

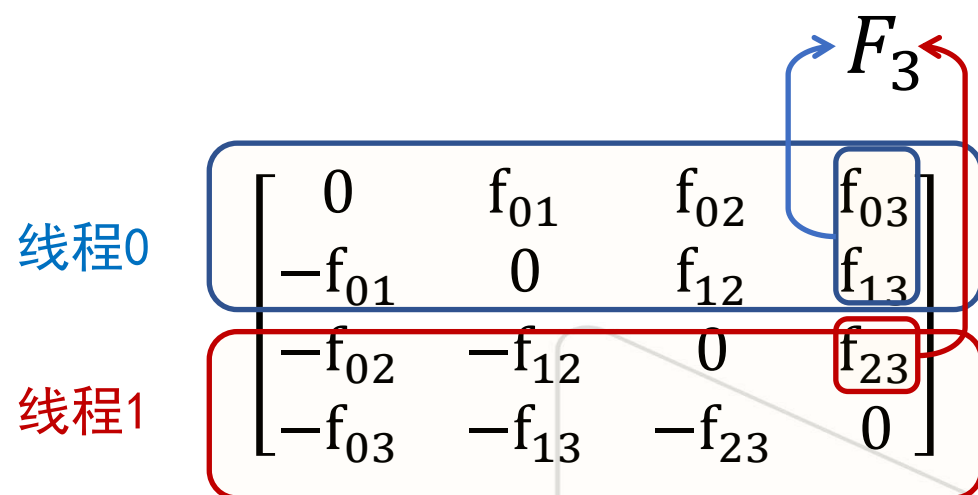
```
    forces[k][X] -= force_qk[X];
```

```
    forces[k][Y] -= force_qk[Y];
```

```
}
```

• OpenMP实现（简化算法）：依赖关系分析

- 更新`forces[q]`时可能出现竞争条件
- 解决方案2：使用锁保护单个`forces[q]`的更新
 - 每个变量对应一个锁对象
 - 但性能依然不如串行版本

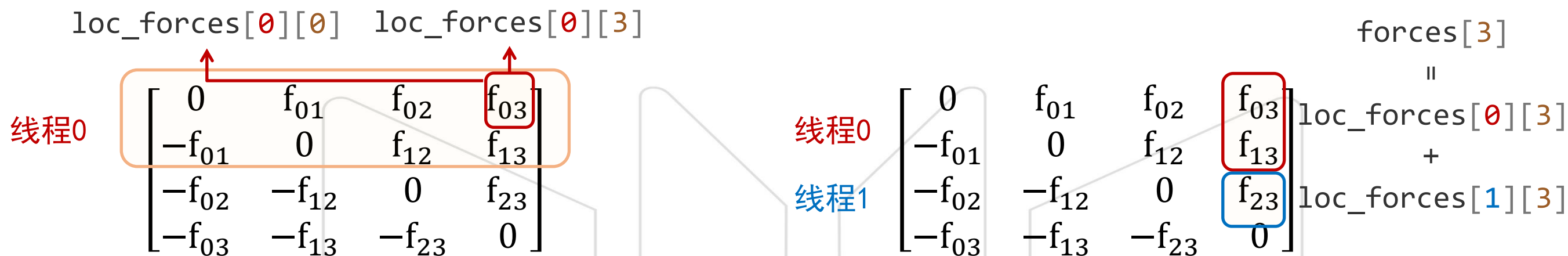


```
omp_set_lock(&locks[q]);  
forces[q][X] += force_qk[X];  
forces[q][Y] += force_qk[Y];  
omp_unset_lock(&locks[q]);
```

```
omp_set_lock(&locks[k]);  
forces[k][X] -= force_qk[X];  
forces[k][Y] -= force_qk[Y];  
omp_unset_lock(&locks[k]);
```

OpenMP实现（简化算法）：依赖关系分析

- 更新`forces[q]`时可能出现竞争条件
- 解决方案3：分阶段进行；每个线程分配部分粒子
 - 阶段1：每个线程*i*计算其分配的粒子对其他粒子`q`的局部合力
 - 存储于对应的局部数组`loc_forces[i][q]`
 - 阶段2：每个线程*i*对其分配的粒子`q`计算局部合力的和
 - `forces[q] += loc_forces[...][q]`



OpenMP实现（简化算法）：依赖关系分析

- 更新forces[q]时可能出现竞争条件
- 解决方案3：阶段1：计算局部合力

```
# pragma omp for
for each particle q {
    force_qk[X] = force_qk[Y] = 0;
    for each particle k > q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
        force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

        loc_forces[my_rank][q][X] += force_qk[X];
        loc_forces[my_rank][q][Y] += force_qk[Y];
        loc_forces[my_rank][k][X] -= force_qk[X];
        loc_forces[my_rank][k][Y] -= force_qk[Y];
    }
}
```

loc_forces[0][0] loc_forces[0][3]

线程0

0	f_{01}	f_{02}	f_{03}
$-f_{01}$	0	f_{12}	f_{13}
$-f_{02}$	$-f_{12}$	0	f_{23}
$-f_{03}$	$-f_{13}$	$-f_{23}$	0

OpenMP实现（简化算法）：依赖关系分析

- 更新forces[q]时可能出现竞争条件
- 解决方案3：阶段2：对局部合力求和

```
# pragma omp for
```

```
for (q = 0; q < n; q++) {
```

```
    forces[q][X] = forces[q][Y] = 0;
```

```
    for (thread = 0; thread < thread_count; thread++) {
```

```
        forces[q][X] += loc_forces[thread][q][X];
```

```
        forces[q][Y] += loc_forces[thread][q][Y];
```

```
    }
```

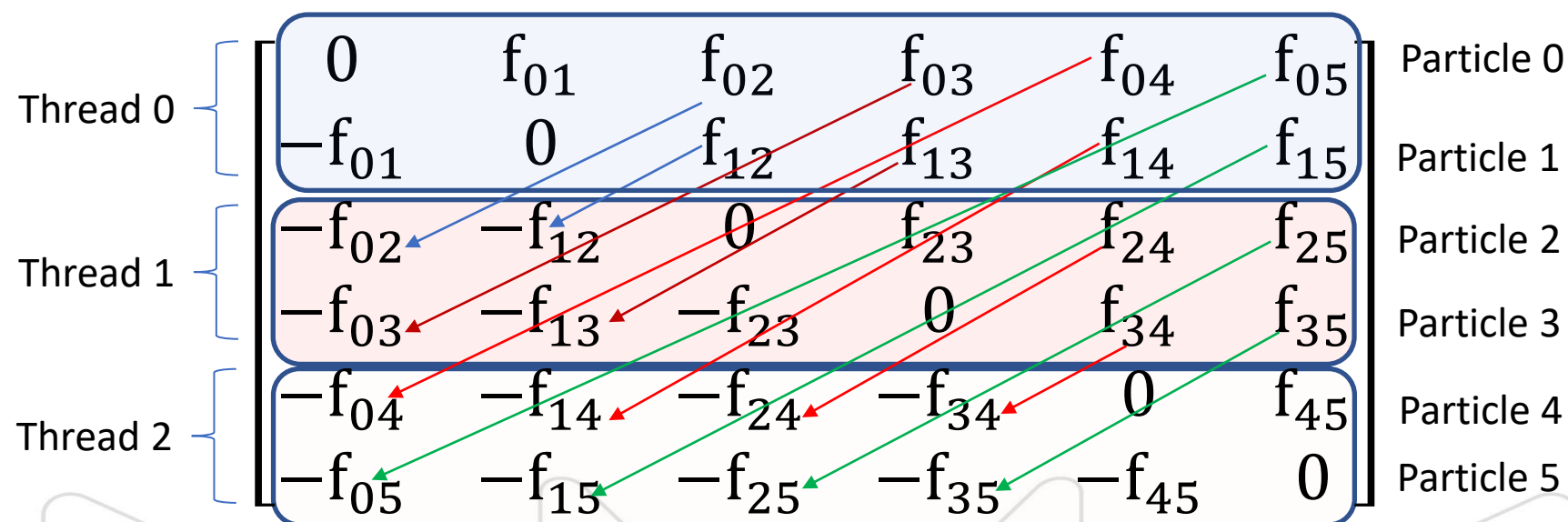
```
}
```

$$\begin{array}{l}
 \text{线程0} \\
 \text{线程1}
 \end{array}
 \begin{bmatrix}
 0 & f_{01} & f_{02} & f_{03} \\
 -f_{01} & 0 & f_{12} & f_{13} \\
 -f_{02} & -f_{12} & 0 & f_{23} \\
 -f_{03} & -f_{13} & -f_{23} & 0
 \end{bmatrix}
 \begin{array}{l}
 \text{loc_forces}[0][3] \\
 + \\
 \text{loc_forces}[1][3]
 \end{array}
 \begin{array}{l}
 \text{forces}[3] \\
 \parallel \\
 \text{loc_forces}[0][3] \\
 + \\
 \text{loc_forces}[1][3]
 \end{array}$$

OpenMP实现（简化算法）：负载均衡分析

– 如何划分数据？

- 当前对作用力矩阵的行使用了块划分
- 每个线程处理连续的数行



OpenMP实现（简化算法）：负载均衡分析

– 如何划分数据？

- 当前对作用力矩阵的行使用了块划分
- 每个线程处理连续的数行
- **负载并不均衡**

– 每个线程工作量为其处理的矩阵元素数目 $\times 2$

$$\begin{bmatrix}
 0 & f_{01} & f_{02} & f_{03} & f_{04} & f_{05} \\
 -f_{01} & 0 & f_{12} & f_{13} & f_{14} & f_{15} \\
 -f_{02} & -f_{12} & 0 & f_{23} & f_{24} & f_{25} \\
 -f_{03} & -f_{13} & -f_{23} & 0 & f_{34} & f_{35} \\
 -f_{04} & -f_{14} & -f_{24} & -f_{34} & 0 & f_{45} \\
 -f_{05} & -f_{15} & -f_{25} & -f_{35} & -f_{45} & 0
 \end{bmatrix}$$

		Thread		
Thread	Particle	0	1	2
0	0	$f_{01} + f_{02} + f_{03} + f_{04} + f_{05}$	0	0
	1	$-f_{01} + f_{12} + f_{13} + f_{14} + f_{15}$	0	0
1	2	$-f_{02} - f_{12}$	$f_{23} + f_{24} + f_{25}$	0
	3	$-f_{03} - f_{13}$	$-f_{23} + f_{34} + f_{35}$	0
2	4	$-f_{04} - f_{14}$	$-f_{24} - f_{34}$	f_{45}
	5	$-f_{05} - f_{15}$	$-f_{25} - f_{35}$	$-f_{45}$

• OpenMP实现（简化算法）：负载均衡分析

– 如何划分数据？

- 对作用力矩阵的行使用循环划分
- 每个线程按其编号依次处理一行
- 负载更为均衡（局部合力计算阶段）

– 思考：是否还有优化空间？

- 是否能按矩阵中的块进行划分？是否能使用调度算法自动分配？

Thread 0	0	f_{01}	f_{02}	f_{03}	f_{04}	f_{05}	Particle 0
Thread 1	$-f_{01}$	0	f_{12}	f_{13}	f_{14}	f_{15}	Particle 1
Thread 2	$-f_{02}$	$-f_{12}$	0	f_{23}	f_{24}	f_{25}	Particle 2
Thread 0	$-f_{03}$	$-f_{13}$	$-f_{23}$	0	f_{34}	f_{35}	Particle 3
Thread 1	$-f_{04}$	$-f_{14}$	$-f_{24}$	$-f_{34}$	0	f_{45}	Particle 4
Thread 2	$-f_{05}$	$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$	0	Particle 5

◉ Pthreads实现分析

– 数据存储

- Pthreads线程的**局部**变量为**私有**（其他线程无法访问）
- 线程**共享**变量为**全局**
 - 注意：简化算法中的loc_forces数组需要共享
- 基本**数据结构**（存储方式）与OpenMP实现**一致**

– 循环并行

- OpenMP可以使用调度算法完成任务分配
- Pthreads需要显式指明每个线程的工作内容

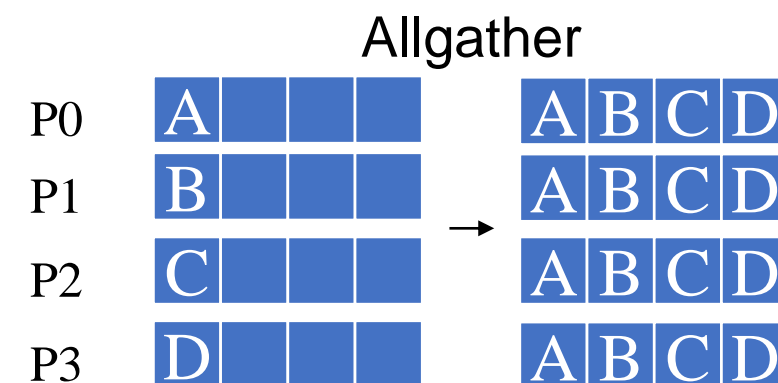
– 同步：每次迭代后需要同步，迭代中的不同阶段也需要同步

- OpenMP的parallel for并行块后有隐含的同步
- Pthreads需要显式进行同步（并非所有实现都有barrier，参考课件4）

• MPI实现分析

- MPI为分布式内存，而OpenMP/Pthreads都是共享内存
- 数据存储：依然以按作用力矩阵行划分为例
 - 所有进程都需要使用所有粒子质量（全局数组，但存储于局部；只读）
 - 每个进程只需要维护部分粒子的位置、合力、速度
 - 每个进程在更新合力时需要使用其他进程的粒子位置

```
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions into global pos array;
}
Print positions and velocities of particles;
```

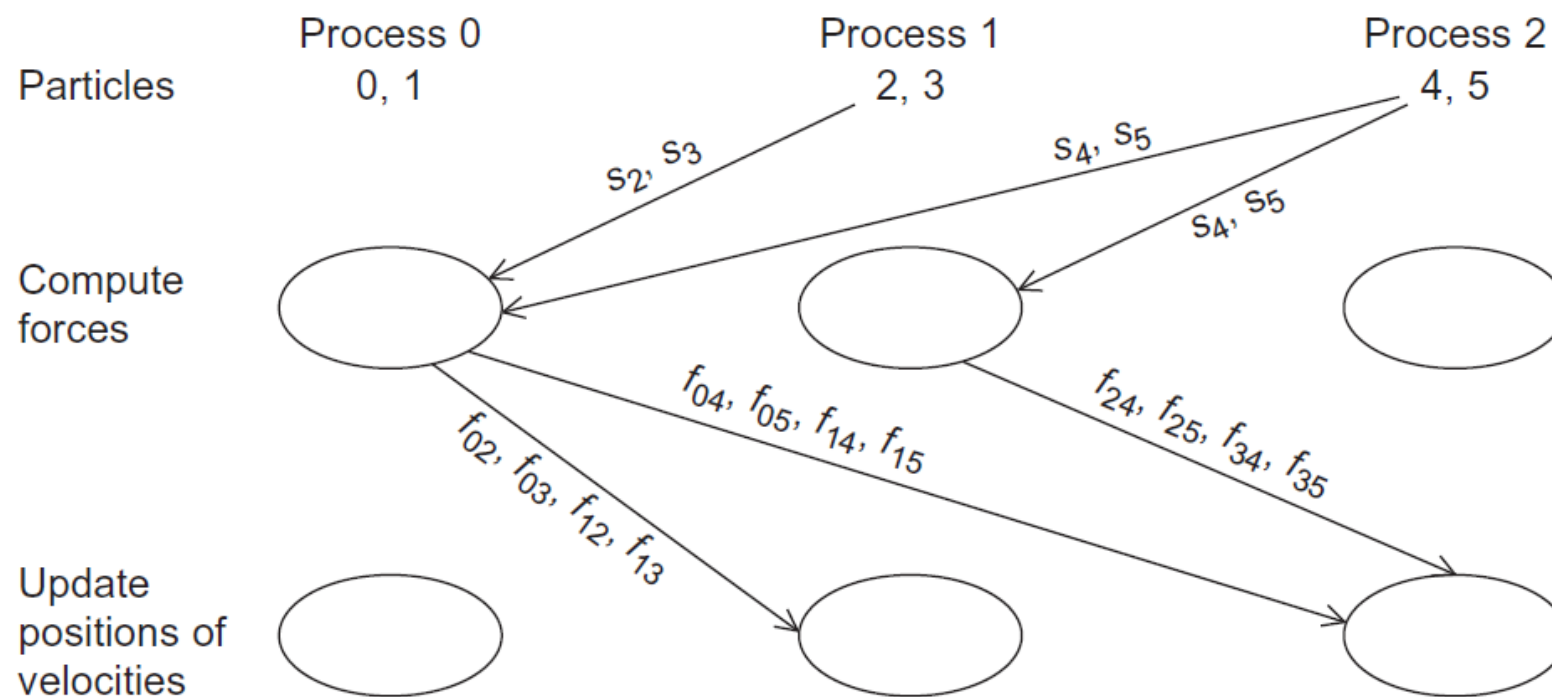


• MPI实现分析

– 数据存储：依然以按作用力矩阵行划分为例

- 每个进程只需要维护部分粒子的位置、合力、速度
- 每个进程在更新合力时需要使用其他进程的粒子位置

```
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions into global pos array;
    Print positions and velocities of particles;
}
```



复杂的通信模式！

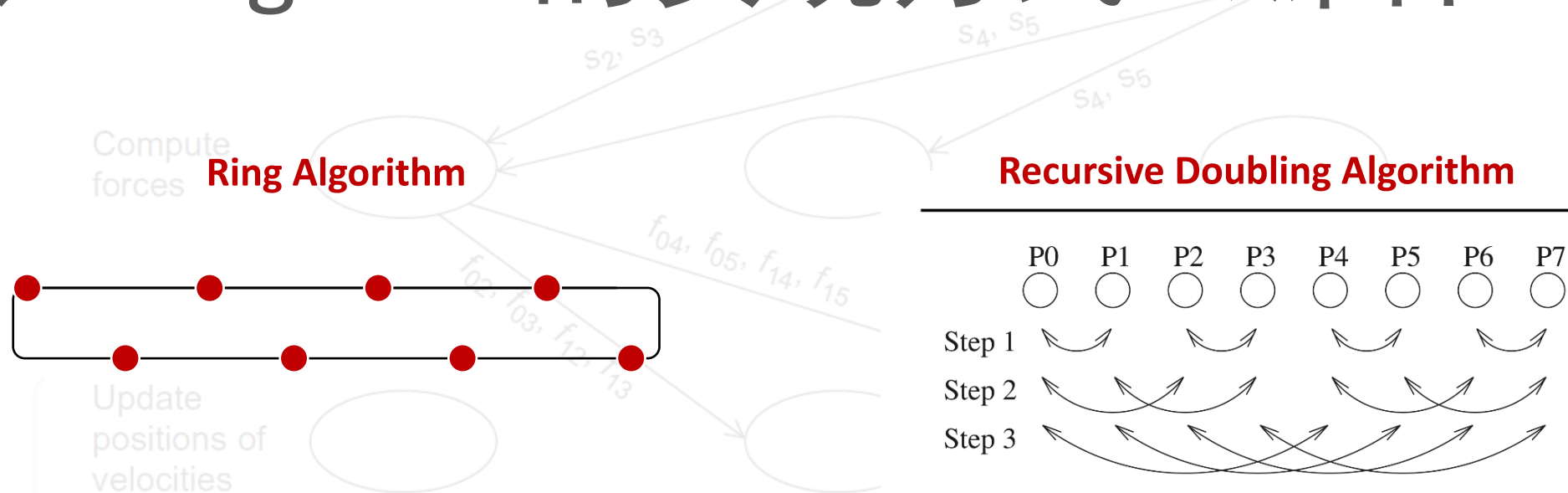
• MPI实现分析

– 数据存储：依然以按作用力矩阵行划分为例

- 每个进程只需要维护部分粒子的**位置**、**合力**、**速度**
- 每个进程在更新合力时需要使用其他进程的粒子**位置**

```
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions into global pos array;
    Print positions and velocities of particles;
```

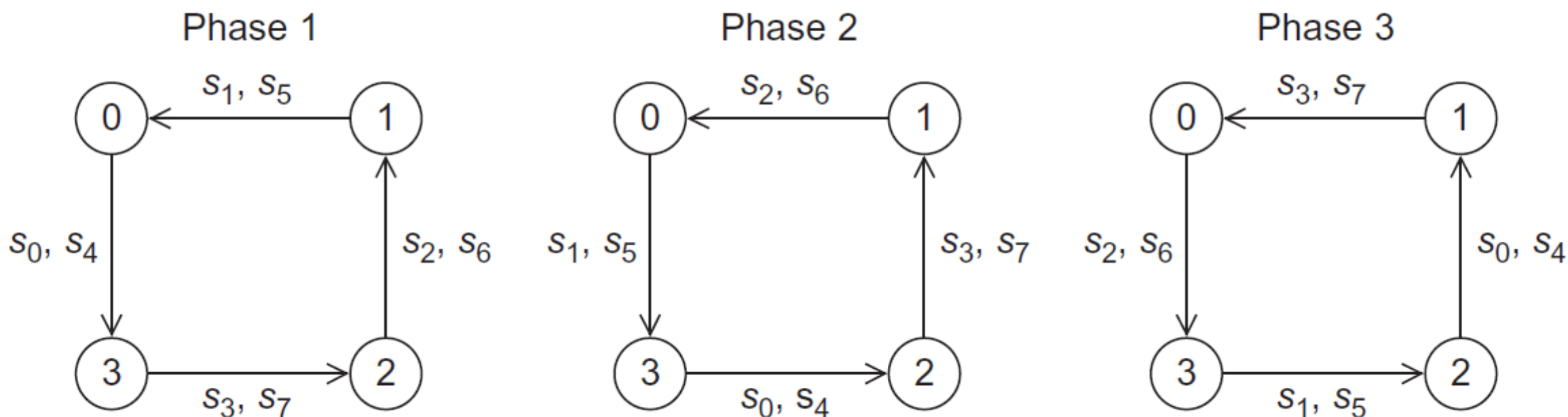
回顾：Allgather的实现方式（课件3）



是否需要**同时**拥有所有粒子位置？

● MPI实现分析：环状更新策略

- 阶段1：进程将粒子位置发送给左邻居，并接收来自右邻居的位置
- 阶段2：进程将接收的位置发送给左邻居，并接收来自右邻居的数据
 - 左邻居（进程编号比自身小1的进程）；右邻居（编号比自身大1的进程）
- 例：4个进程，8个粒子，循环划分



● MPI实现分析：环状更新策略（线程每阶段任务）

- 计算自身分配粒子和接收粒子间的相互作用力
- 更新合力
 - 自身分配粒子：将两个粒子间的作用力加到所受合力中
 - 接收粒子：将作用力从合力中减去
 - 发送位置时同时发送合力

$$\begin{array}{c}
 F_0 \\
 F_1 \\
 F_2 \\
 F_3 \\
 F_4 \\
 F_5
 \end{array}
 \begin{bmatrix}
 0 & f_{01} & f_{02} & f_{03} & f_{04} & f_{05} \\
 -f_{01} & 0 & f_{12} & f_{13} & f_{14} & f_{15} \\
 -f_{02} & -f_{12} & 0 & f_{23} & f_{24} & f_{25} \\
 -f_{03} & -f_{13} & -f_{23} & 0 & f_{34} & f_{35} \\
 -f_{04} & -f_{14} & -f_{24} & -f_{34} & 0 & f_{45} \\
 -f_{05} & -f_{15} & -f_{25} & -f_{35} & -f_{45} & 0
 \end{bmatrix}$$

• MPI实现分析：环状更新策略（实现）

- 收发粒子位置和合力
- 更新自身合力及接收合力

```
source = (my_rank + 1) % comm_sz;  
dest = (my_rank - 1 + comm_sz) % comm_sz;  
Copy loc_pos into tmp_pos;  
loc_forces = tmp_forces = 0;  
  
Compute forces due to interactions among local particles;  
for (phase = 1; phase < comm_sz; phase++) {  
    Send current tmp_pos and tmp_forces to dest;  
    Receive new tmp_pos and tmp_forces from source;  
    /* Owner of the positions and forces we're receiving */  
    owner = (my_rank + phase) % comm_sz;  
    Compute forces due to interactions among my particles  
    and owner's particles;  
}  
Send current tmp_pos and tmp_forces to dest;  
Receive new tmp_pos and tmp_forces from source;
```

• MPI实现分析：环状更新策略（实现）

Time	Variable	Process 0	Process 1
Start	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $0, 0$ s_0, s_2 $0, 0$	s_1, s_3 $0, 0$ s_1, s_3 $0, 0$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{02}, 0$ s_0, s_2 $0, -f_{02}$	s_1, s_3 $f_{13}, 0$ s_1, s_3 $0, -f_{13}$
After First Comm	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{02}, 0$ s_1, s_3 $0, -f_{13}$	s_1, s_3 $f_{13}, 0$ s_0, s_2 $0, -f_{02}$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, f_{23}$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$	s_1, s_3 $f_{12} + f_{13}, 0$ s_0, s_2 $0, -f_{02} - f_{12}$
After Second Comm	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, f_{23}$ s_0, s_2 $0, -f_{02} - f_{12}$	s_1, s_3 $f_{12} + f_{13}, 0$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, -f_{02} - f_{12} + f_{23}$ s_0, s_2 $0, -f_{02} - f_{12}$	s_1, s_3 $-f_{01} + f_{12} + f_{13}, -f_{03} - f_{13} - f_{23}$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$

最终阶段：
合并计算结果
 $\text{loc_forces} + \text{tmp_forces}$

性能比较

- 基础版本：OpenMP比MPI稍快
- 简化算法：OpenMP与MPI性能相似

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

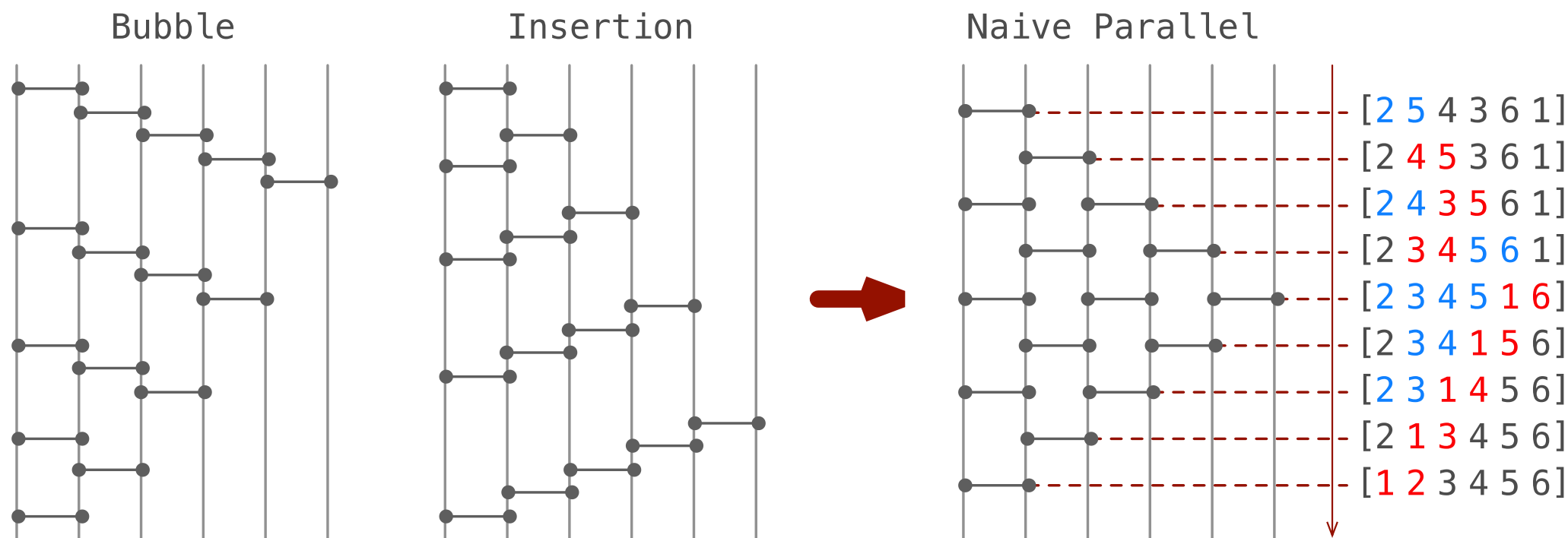
- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

- 问题定义：排序后任意一对元素都是有序的

- 稳定 vs 不稳定：具有相同值的元素，其相对顺序是否会变化？
- 数据驱动 vs 数据无关：执行步骤是否根据数据发生变化？

- 排序网络

- 一系列比较器组成的网络



排序网络与并行

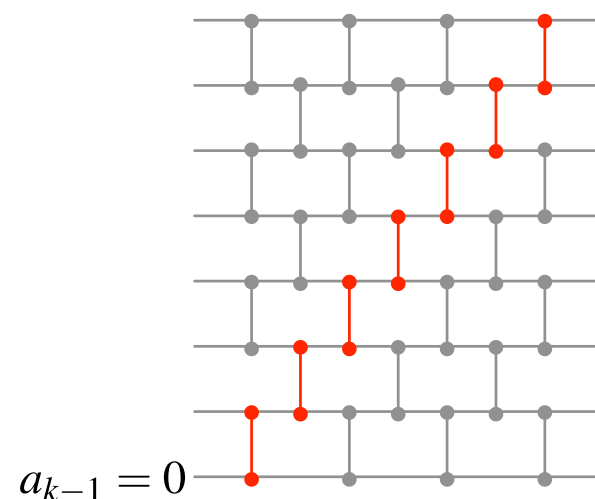
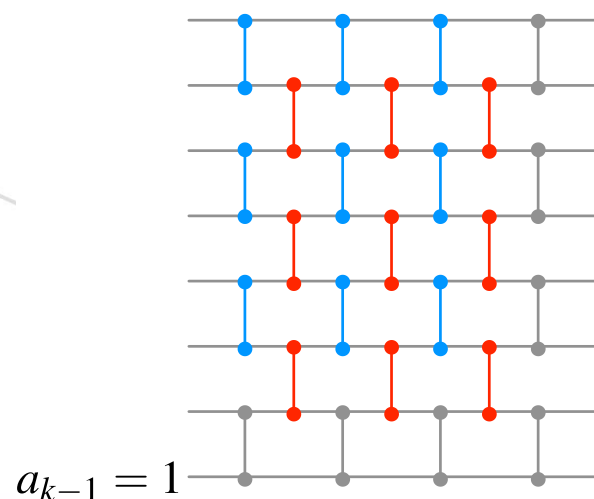
排序网络的正确性证明

0-1-principle

由Donald Ervin Knuth提出：一个比较网络如果能对长度为 2^n 的任意0-1序列进行排序，则该比较网络为排序网络（能对任意数字组成的序列进行正确的排序）。

奇偶移项排序网络（odd-even transposition sort）

- 奇偶



• 如何将层数从 $O(n)$ 进一步降低？

– 分治（divide and conquer）：此前常见的并行模式

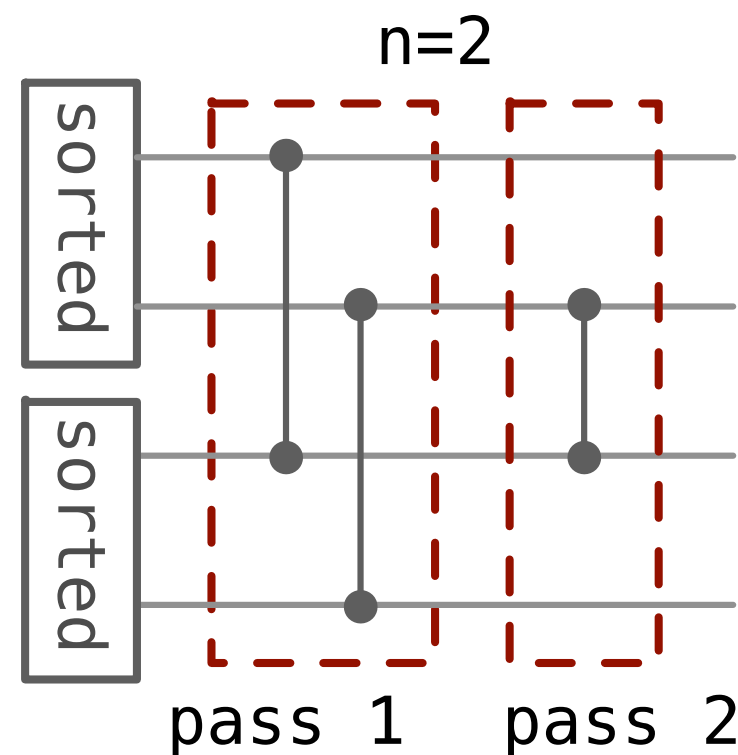
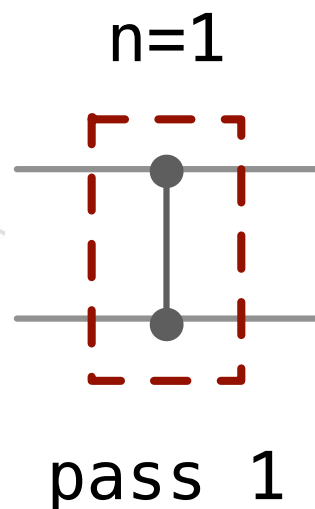
- $O(\log n)$

– 奇偶归并排序网络（odd-even merge sorting network）

- 将两个分别排序的数列 $[a_0, \dots, a_{n-1}]$ 及 $[a_n, \dots, a_{2n-1}]$ 合并

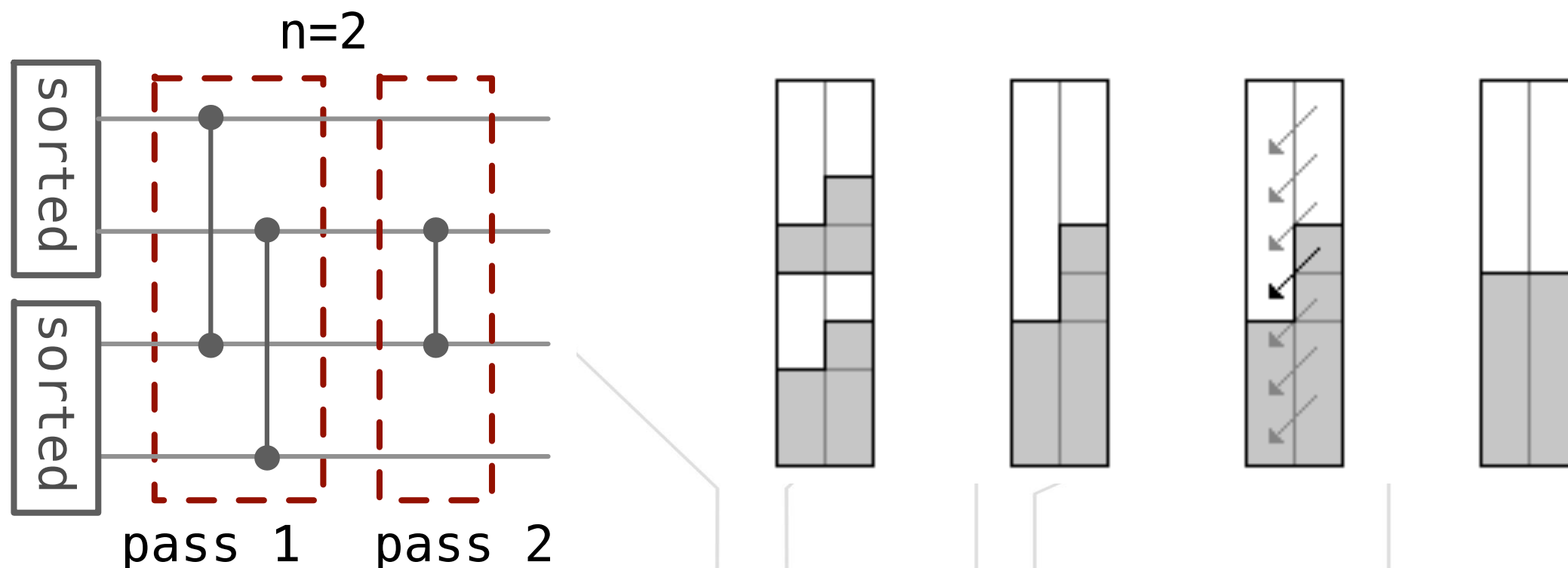
- 每次合并需要 $\log n$ 层

- 总工作量为 $O(n \log^2 n + \log n)$

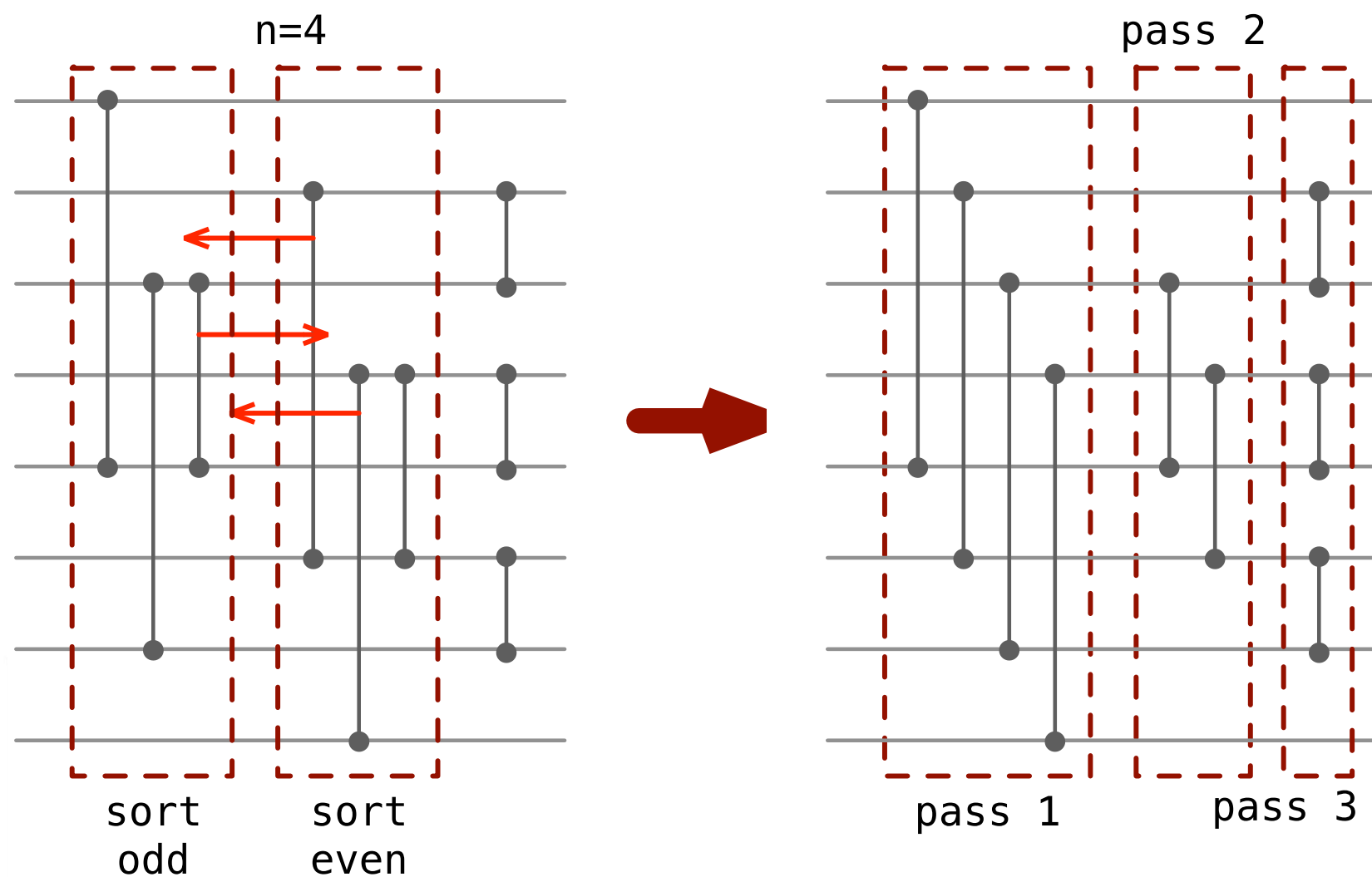


奇偶归并排序网络 (odd-even merge sorting network)

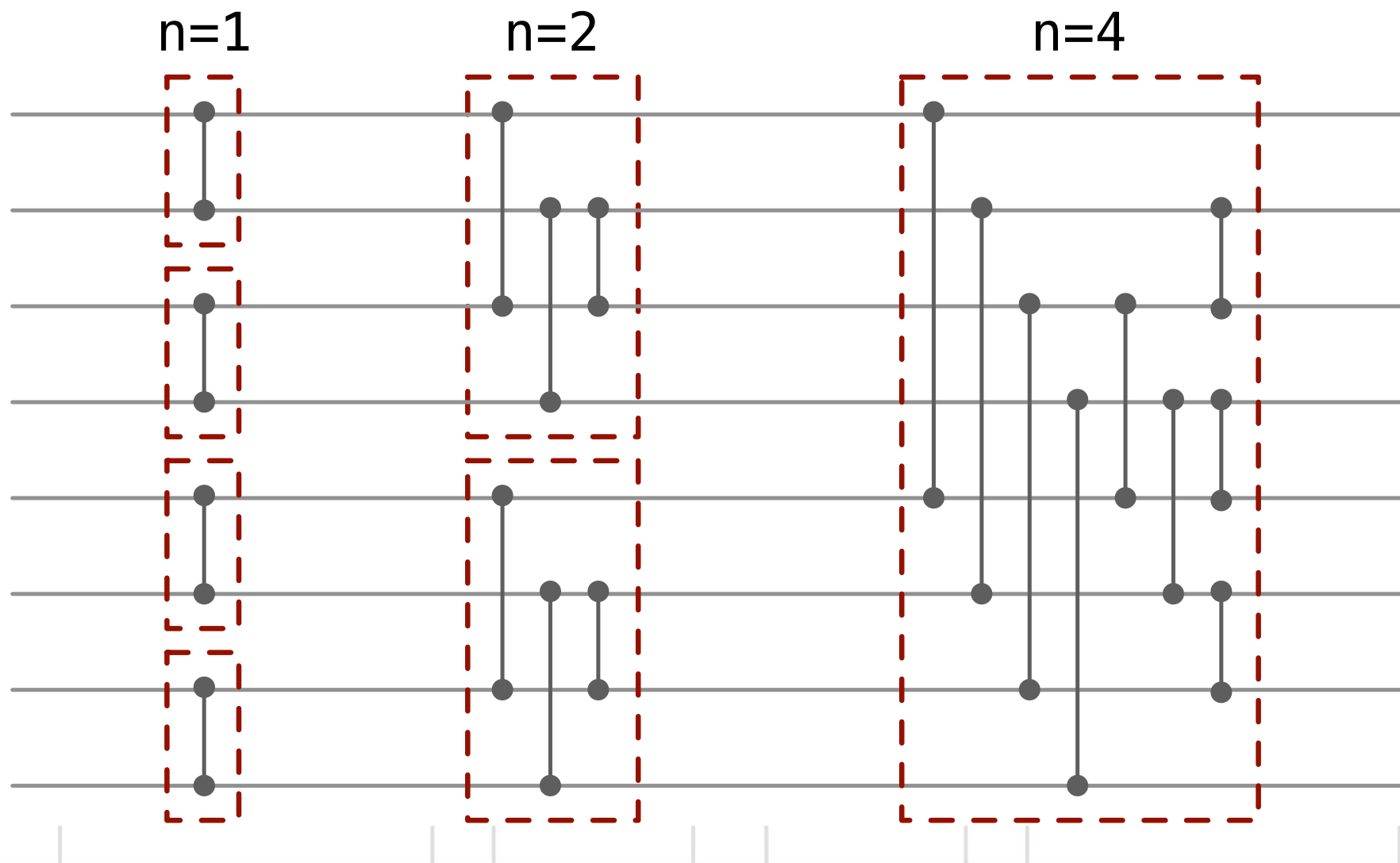
- 将两个分别排序的数列 $[a_0, \dots, a_{n-1}]$ 及 $[a_n, \dots, a_{2n-1}]$ 合并
- 对奇偶分别排序，再交换相邻数据
 - 其正确性可通过0-1 principal证明



- 奇偶归并排序网络 (odd-even merge sorting network)
 - 合并两个长度为4的序列



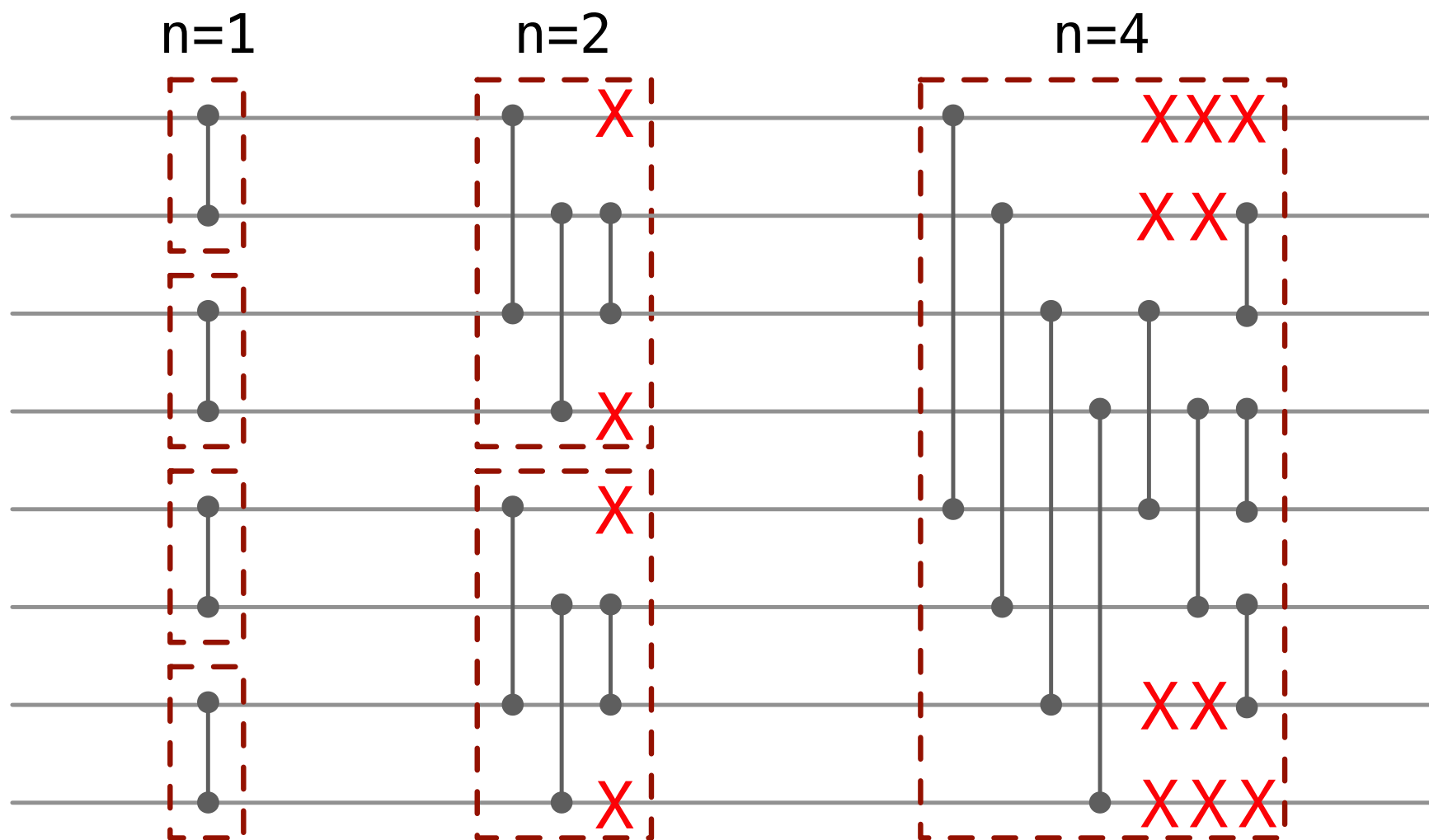
- 奇偶归并排序网络 (odd-even merge sorting network)
 - 对长度为8的数组排序全过程



存在问题？

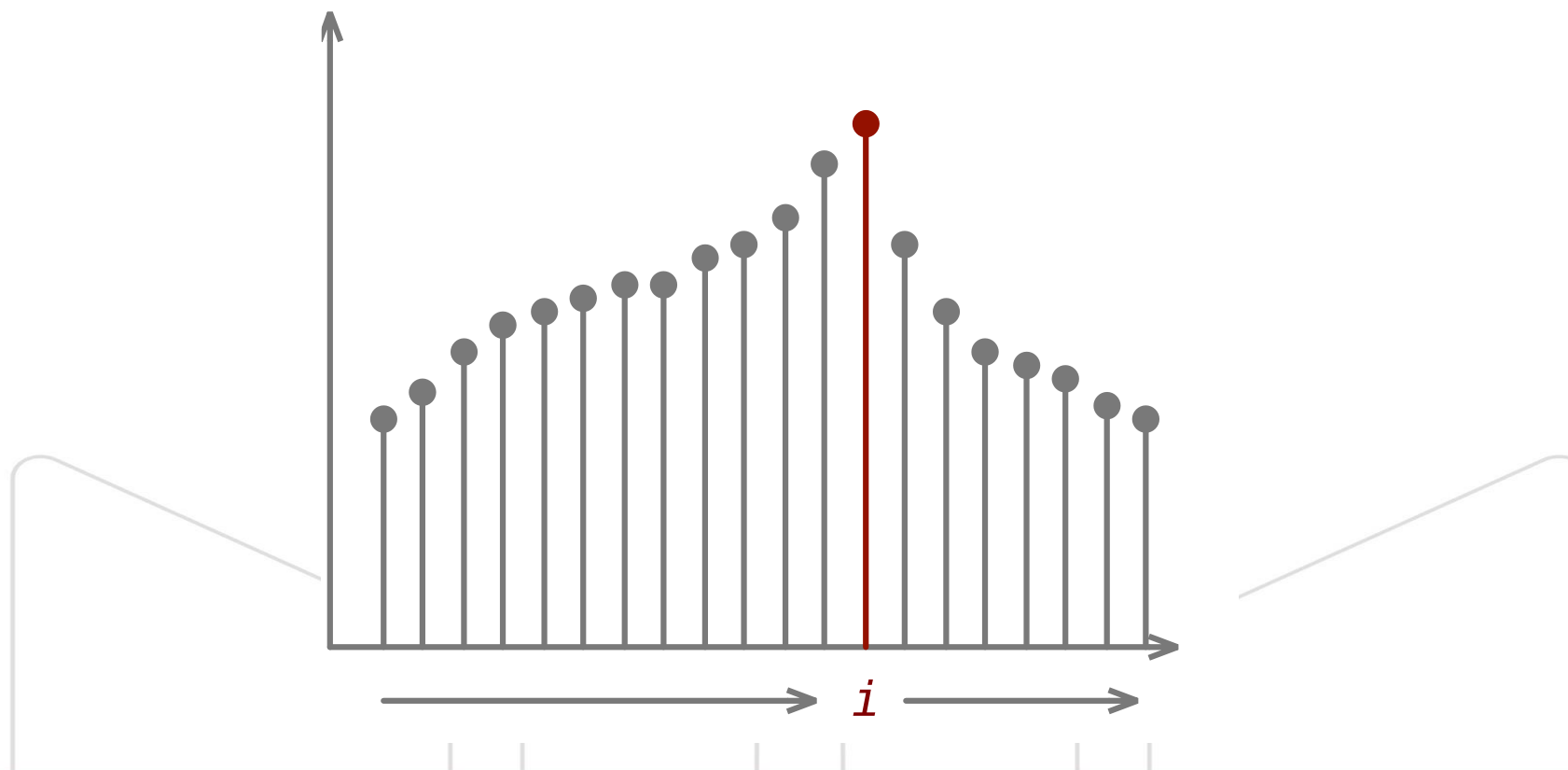
奇偶归并排序网络 (odd-even merge sorting network)

- 存在问题：不规则的数据访问模式；每一层运算量不一致（负载不均衡）；难于编程



双调排序网络 (bitonic sorting network)

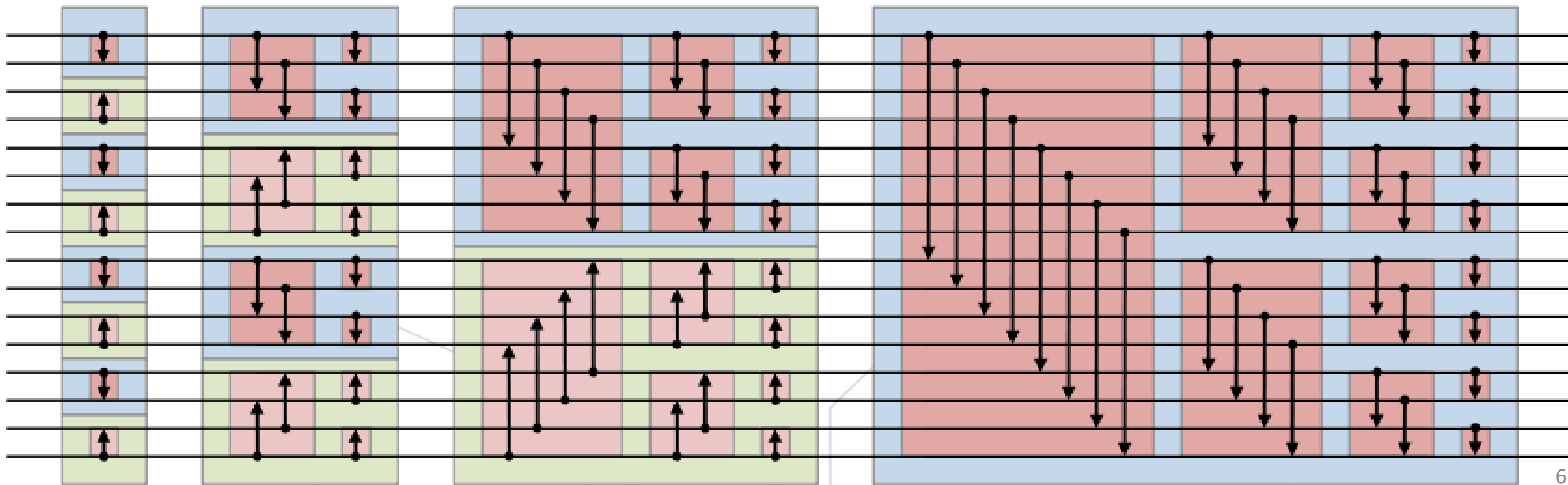
- 通过合并操作将输入合并为较大的双调序列
- 双调序列：对于序列 $[a_0, \dots, a_{n-1}]$ 存在 i 使得 $[a_0, \dots, a_i]$ 为单调递增序列，且 $[a_{i+1}, \dots, a_{n-1}]$ 为单调递减序列



双调排序网络 (bitonic sorting network)

- 第 i 步将一个长度为 2^i 的双调序列（两个长度为 2^{i-1} 的单调序列）变为一个长度为 2^i 的单调序列（相邻两个单调序列组成一个长度为 2^{i+1} 的双调序列）

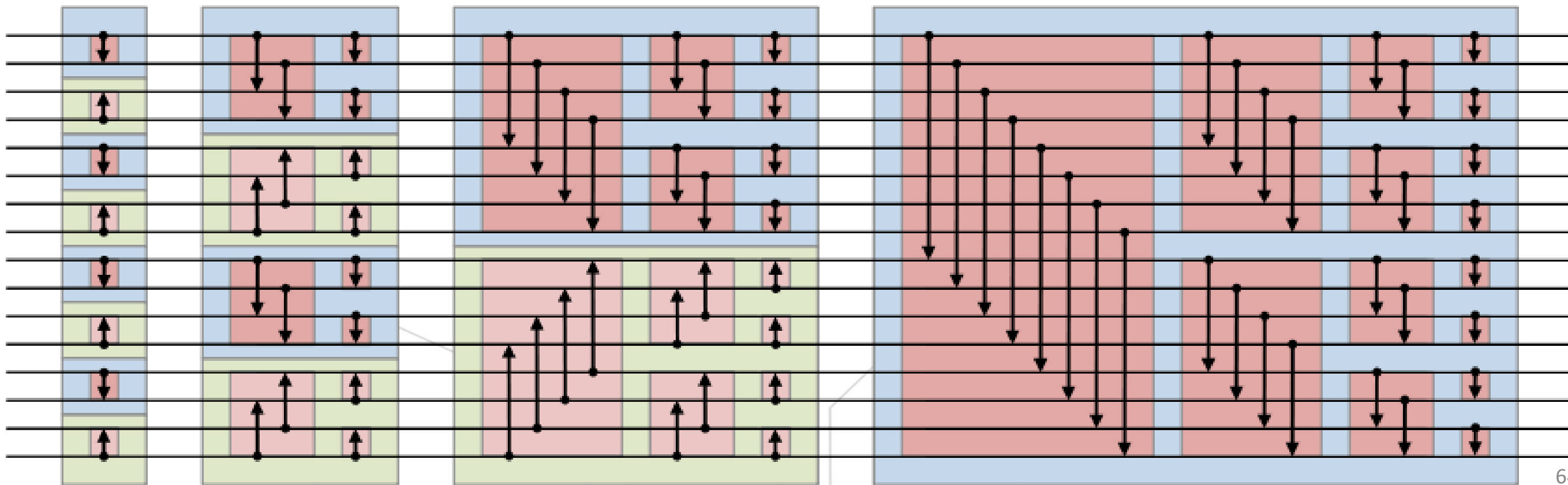
图片来自Wikipedia



双调排序网络 (bitonic sorting network)

- 第 $i = 2^k$ 步 k 步完成，每步执行 $\frac{n}{2}$ 次比较
- 便于CUDA实现：规则的访问模式；负载均衡
- 当 $2^k > block_size$ 时，需要同步

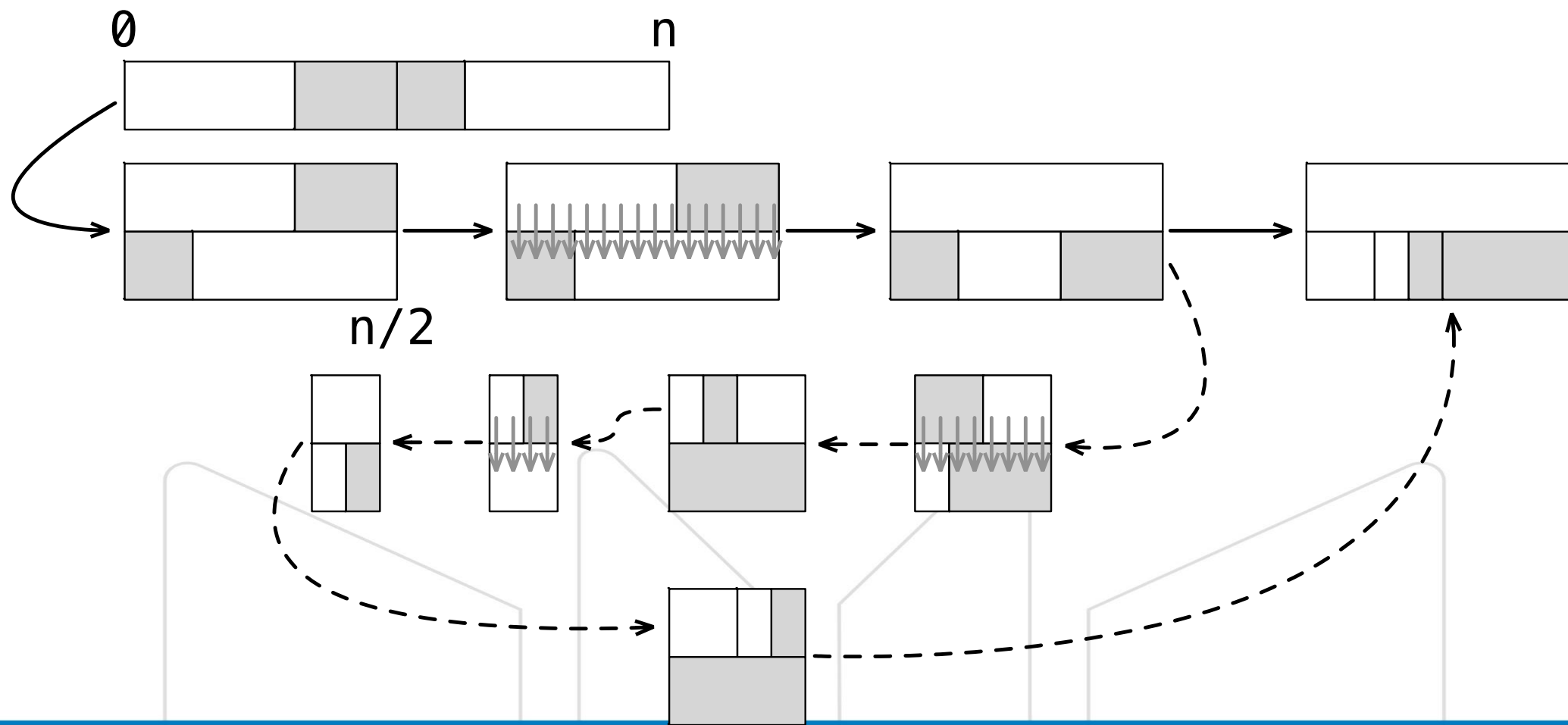
图片来自Wikipedia



双调排序网络 (bitonic sorting network)

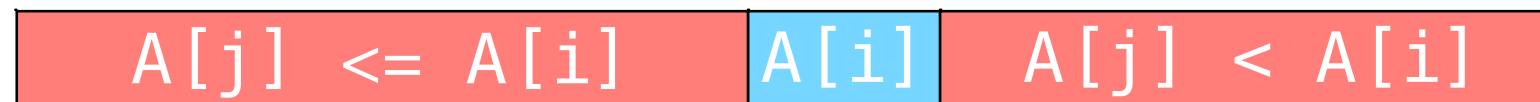
– 正确性证明: 0-1-principle

- 双调的部分每次缩减为原先范围的一半, 最终将两个双调序列合并为一个双调序列



计数排序 (counting sort)

- 第一步：计算在 $A[i]$ 左边的元素个数
- 第二步：将 $A[i]$ 置于相应位置

$$\text{rank}[i] = \text{count} (j < i \text{ where } A[j] \leq A[i]) \\ + \text{count} (j > i \text{ where } A[j] < A[i])$$


$A[i] \rightarrow A[\text{rank}[i]]$

计数排序是否是稳定的?

基数排序 (radix sort)

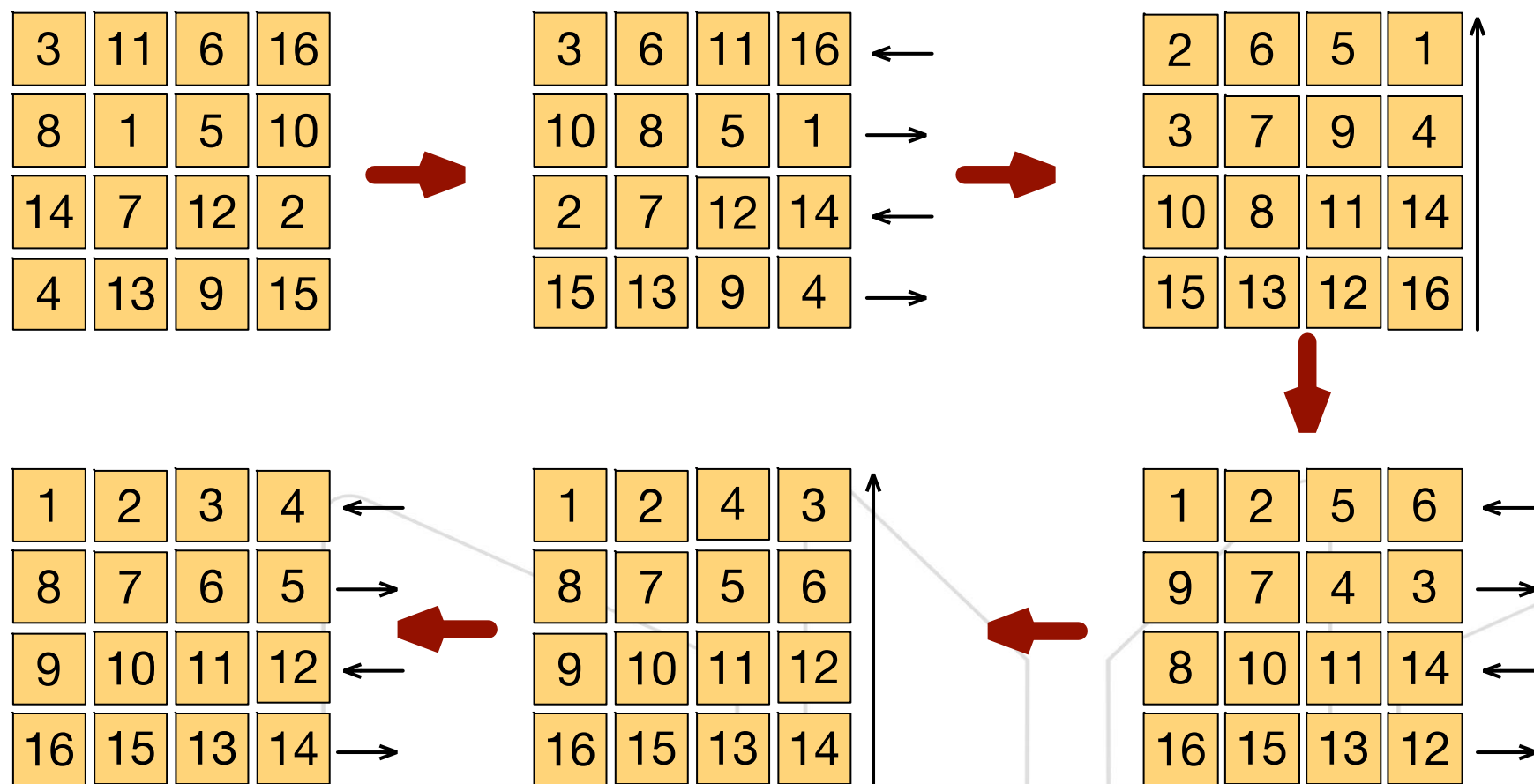
- 按位排序 (可从最高位或最低位开始)
- 对每一位排序时需要稳定的排序
- 可使用binary counting sort对二进制的每一位分别排序实现

326	690	704	326
453	751	608	435
608	453	326	453
835	704	835	704
751	835	435	608
435	435	751	690
704	326	453	751
690	608	690	835

并行扫描!

Shear sort

- 将数组沿“蛇形”放置在矩阵中
- 交替在行、列两个方向上进行排序
- 分治/排序网络？



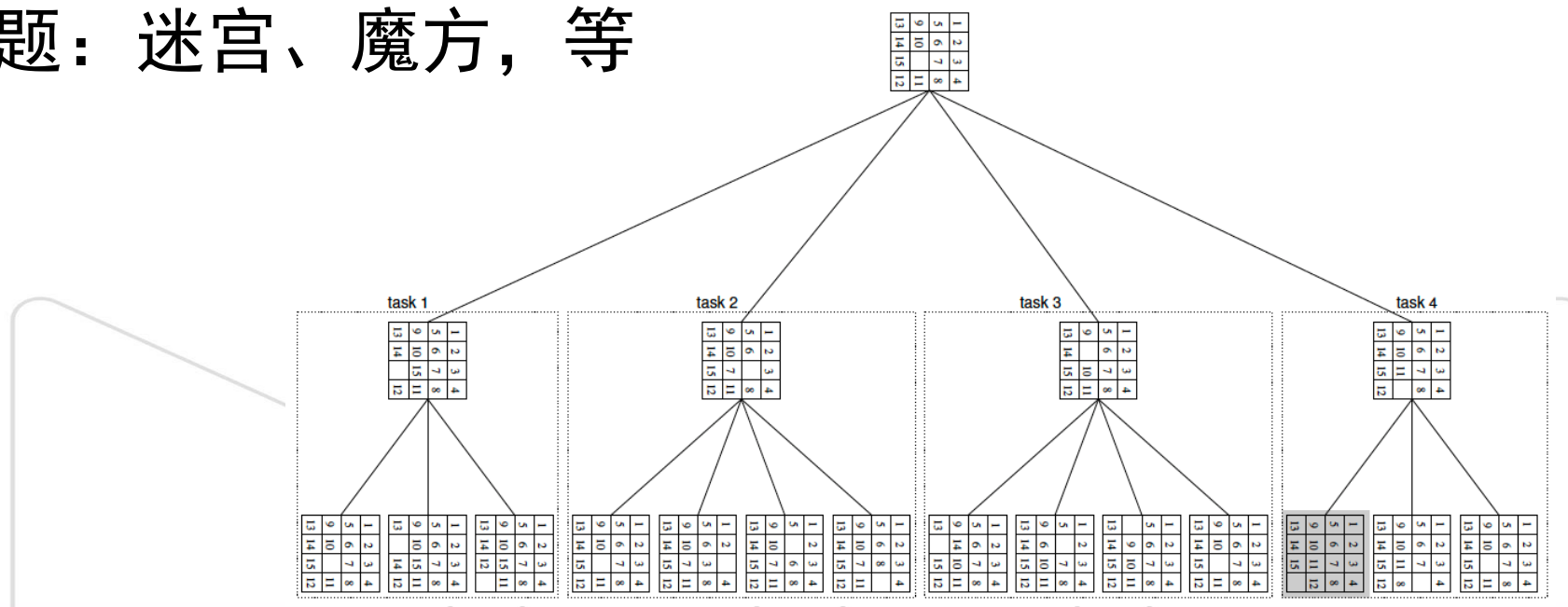
如何证明正确性？
需要重复多少次？

- Sorting network and the 0-1-principle
 - <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/sortieren.htm>
 - <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm>
 - <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oetsen.htm>
 - https://en.wikipedia.org/wiki/Sorting_network#Zero-one_principle
- Sorting
 - https://en.wikipedia.org/wiki/Sorting_algorithm
- Parallel Sorting
 - https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm
- A great tutorial considering shell sort as a variant of bubble sort (odd-even)
 - <http://parallelcomp.uw.hu/ch09lev1sec3.html>
- Enumeration, odd-even, parallel merge, hyper quick sort
 - https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

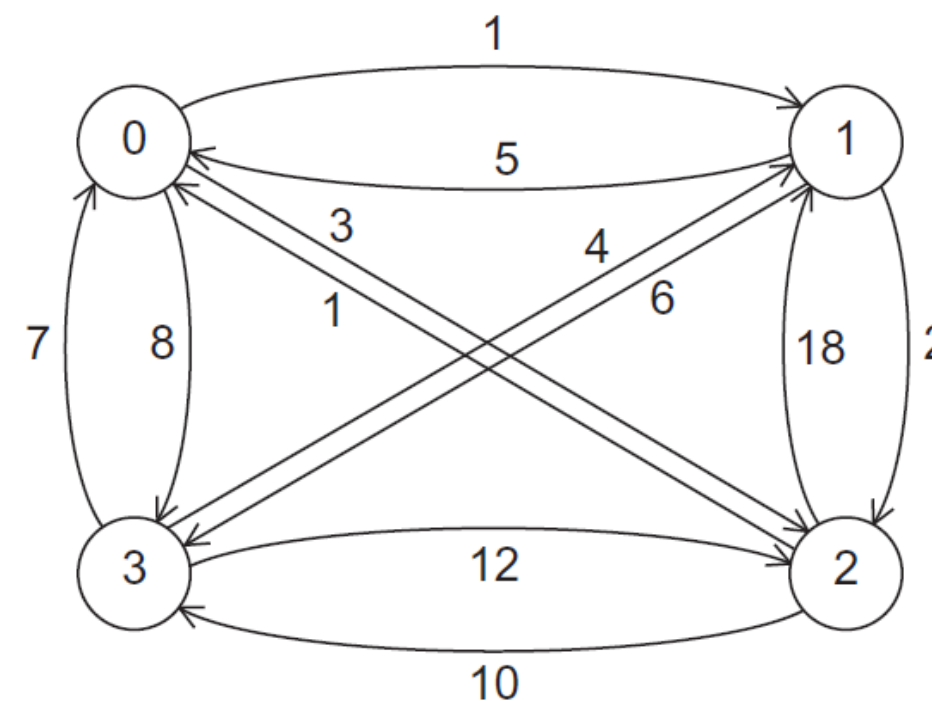
并行回溯与树形搜索

- 树形搜索：按一定策略遍历树中节点，直到找到目标解
- 回溯算法：构建问题的解，不满足则回溯（深度遍历）
- 适用问题
 - 组合优化问题：旅行商问题（TSP），背包问题，图着色问题
 - 约束满足问题：数独，N皇后问题
 - 搜索问题：迷宫、魔方，等



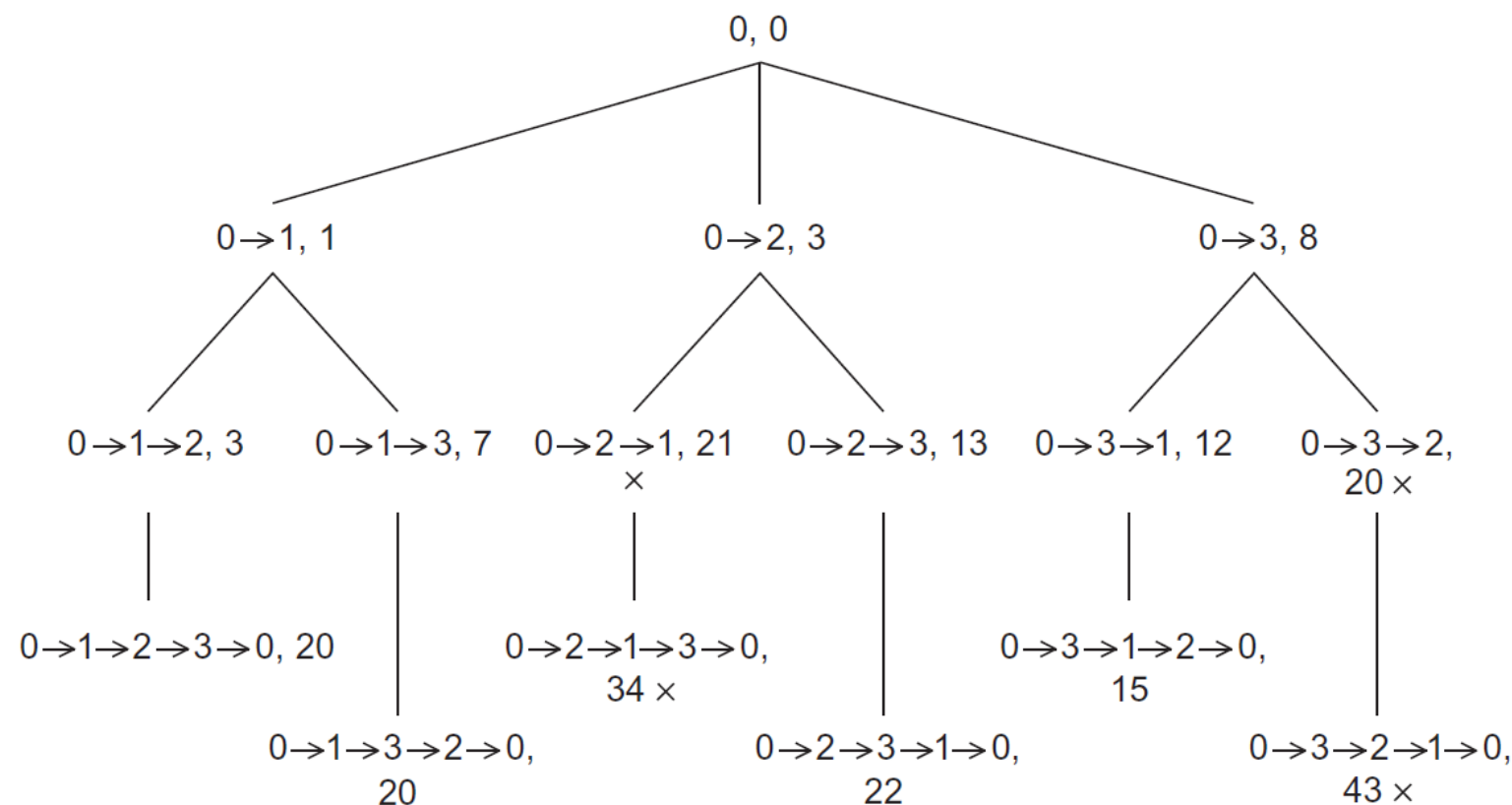
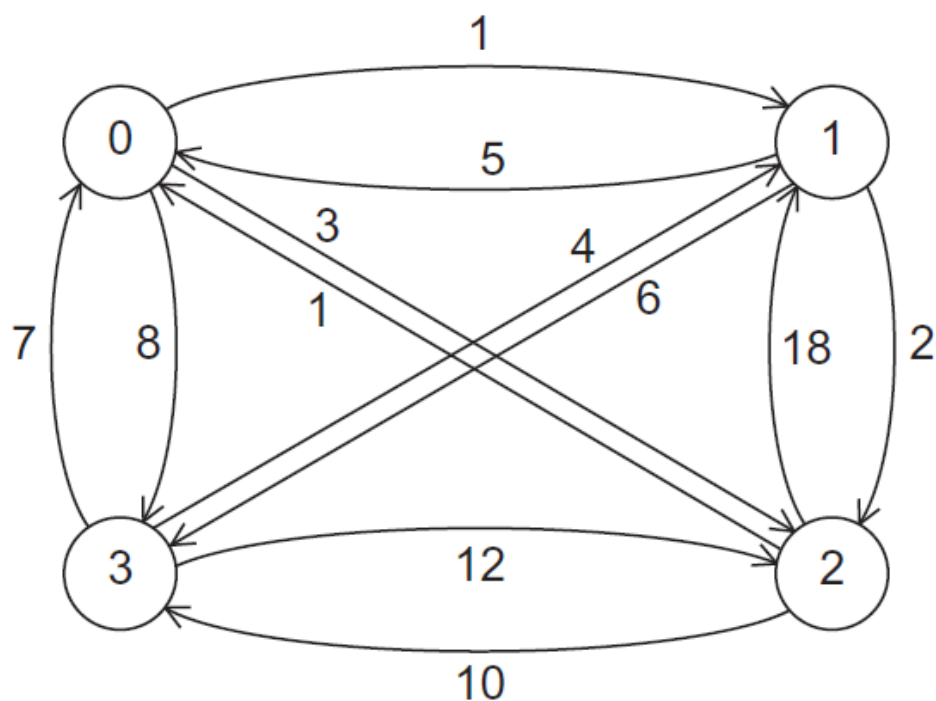
旅行商问题

- 最短路径，访问途中每个节点一次（且仅一次）
- NP-完全（NP-Complete）问题
 - 非确定机在多项式时间内可以解决的问题
 - NP: non-deterministic polynomial
 - 确定机在多项式时间内可以验证的问题
 - 一个可解（多项式时间内），全体可解
- 没有有效解法
 - 在任意情况下都优于穷举
 - 有近似解



问题定义及求解

- 最短路径，访问途中每个节点一次（且仅一次）
- 使用树形搜索穷举：每次列出下一个访问节点



串行递归实现

```
void Depth_first_search(tour_t tour) {  
    city_t city;  
  
    if (City_count(tour) == n) {  
        if (Best_tour(tour))  
            Update_best_tour(tour);  
    } else {  
        for each neighboring city  
            if (Feasible(tour, city)) {  
                Add_city(tour, city);  
                Depth_first_search(tour);  
                Remove_last_city(tour);  
            }  
    }  
}  
/* Depth_first_search */
```

是否已经访问
n个城市

是否是最佳路径

替换当前最佳路径

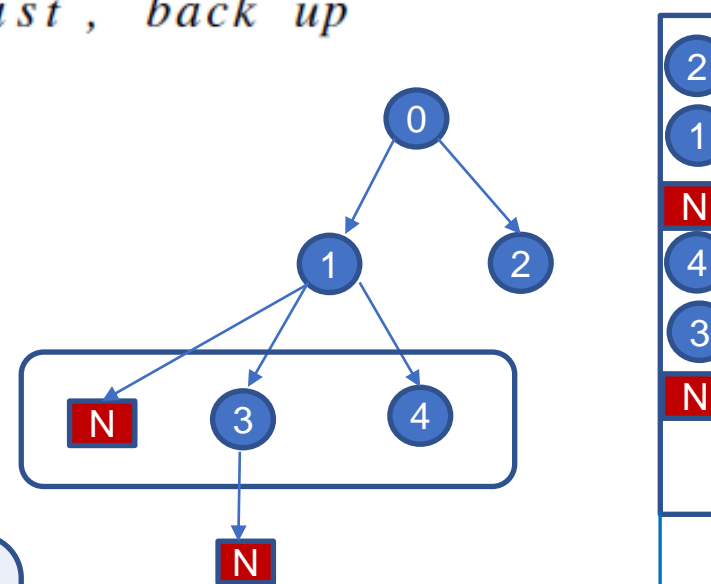
如果city未访问
选择city作为下一个节点

将刚加入当前路径的city
从列表中去掉（标记）

串行迭代实现（使用栈存储城市）

```
for (city = n-1; city >= 1; city--)  
    Push(stack, city);  
while (!Empty(stack)) {  
    city = Pop(stack);  
    if (city == NO_CITY) // End of child list, back up  
        Remove_last_city(curr_tour);  
    else {  
        Add_city(curr_tour, city);  
        if (City_count(curr_tour) == n) {  
            if (Best_tour(curr_tour))  
                Update_best_tour(curr_tour);  
            Remove_last_city(curr_tour);  
        } else {  
            Push(stack, NO_CITY);  
            for (nbr = n-1; nbr >= 1; nbr--)  
                if (Feasible(curr_tour, nbr))  
                    Push(stack, nbr);  
        }  
    }  
} /* if Feasible */  
} /* while !Empty */
```

使用NO_CITY检查
所有子节点访问完毕



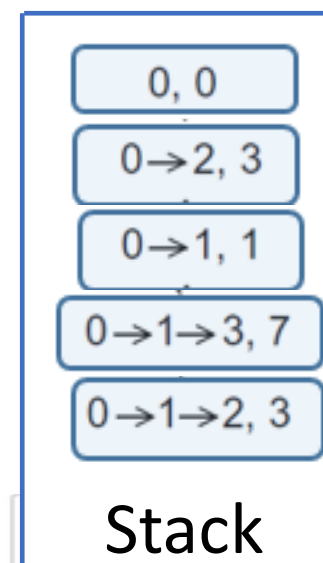
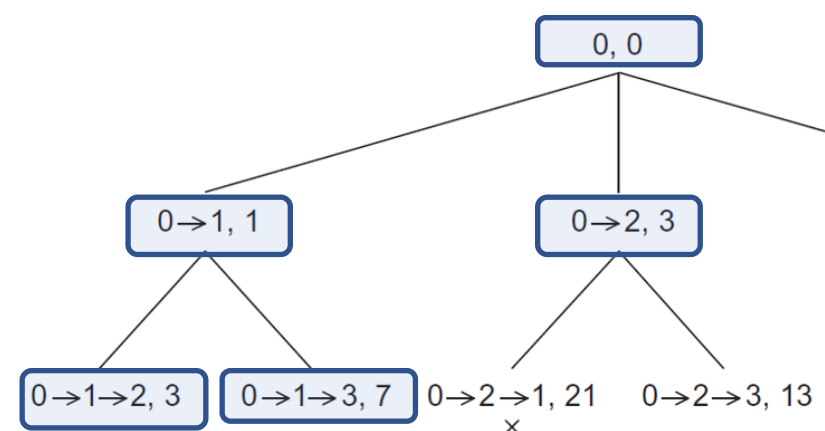
NO_CITY最早入栈
(最晚出栈)

串行迭代实现（使用栈存储路径）

```

Push_copy(stack, tour); // Tour that visits only the hometown
while (!Empty(stack)) {
    curr_tour = Pop(stack);
    if (City_count(curr_tour) == n) {
        if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
    } else {
        for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr)) {
                Add_city(curr_tour, nbr);
                Push_copy(stack, curr_tour);
                Remove_last_city(curr_tour);
            }
    }
    Free_tour(curr_tour);
}

```



串行实现的性能比较

递归	迭代（城市栈）	迭代（路径栈）
30.5	29.2	32.9

- 使用栈存储**城市**的迭代较快：每次修改路径中的一个节点
 - 但子节点只在父节点路径上增加一个城市
 - 兄弟节点对应的路径上替换一个城市
 - 只进行了**必要操作**
- 使用栈存储**路径**的迭代较慢：每次对路径进行入栈出栈
 - 增加了**不必要的存储开销**（时间/空间）
 - 但**更适合并行**：进程/线程维护各自的路径

并行分析：如何通信（数据交换）？

– 所有进程/线程都需要检查一个**共同的当前最优路径**

- **Best_tour**只需要读取当前最小开销

 - 多个进程可以同时读取

- **Update_best_tour**执行更新

 - 路径开销比当前最优路径更小时

- 如何实现？读写锁？

 - 没有线程写入时，多个线程可以同时读

 - 当有线程写入时，所有线程都需要等待

```
while (!Empty(stack)) {  
    curr_tour = Pop(stack);  
    if (City_count(curr_tour) == n) {  
        if (Best_tour(curr_tour))  
            Update_best_tour(curr_tour);  
    } else {  
        for (nbr = n-1; nbr >= 1; nbr--)  
            if (Feasible(curr_tour, nbr)) {  
                Add_city(curr_tour, nbr);  
                Push_copy(stack, curr_tour);  
                Remove_last_city(curr_tour);  
            }  
    }  
    Free_tour(curr_tour);  
}
```

• 并行分析：如何通信（数据交换）？

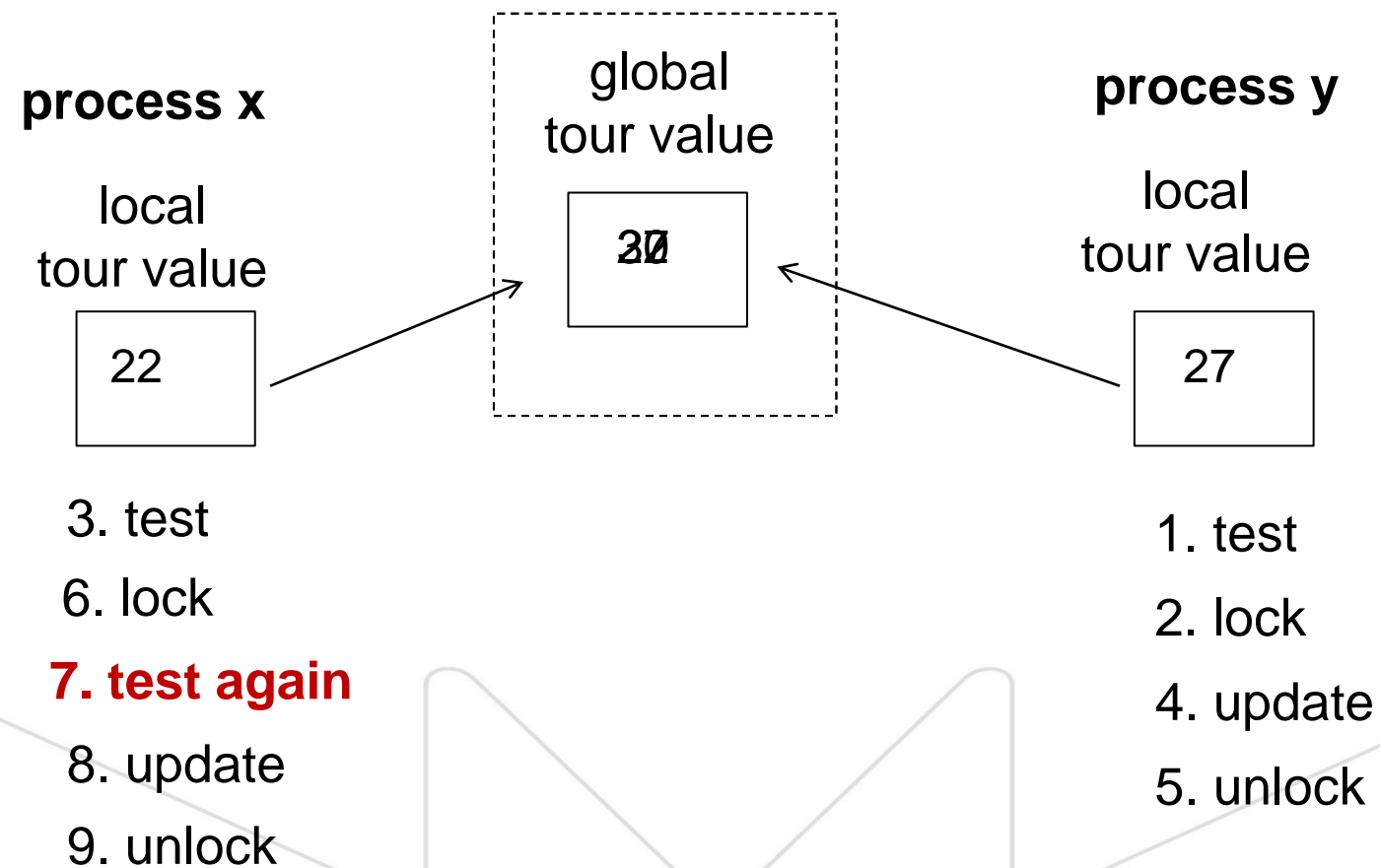
– Best_tour 读取最小开销（可能被修改）

- 读取的是修改后的值
 - 无冲突
- 读取的是修改前的值
 - 大于当前最优：不执行更新（无冲突）
 - 小于当前最优：执行更新（可能冲突）
 - » 获取锁后再检查

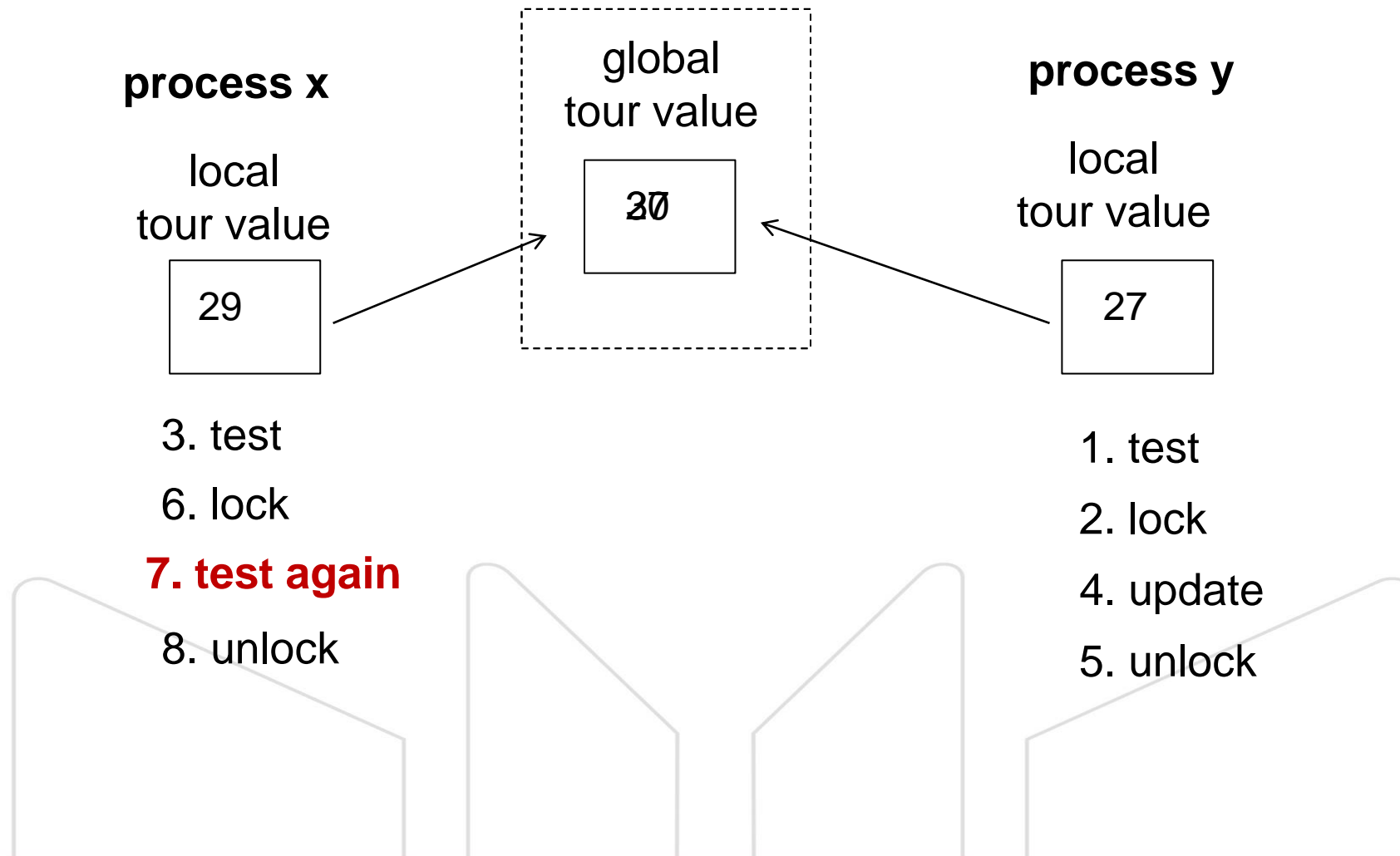
– 总结：只有可能更新时，尝试加锁

```
while (!Empty(stack)) {  
    curr_tour = Pop(stack);  
    if (City_count(curr_tour) == n) {  
        if (Best_tour(curr_tour))  
            Update_best_tour(curr_tour);  
    } else {  
        for (nbr = n-1; nbr >= 1; nbr--)  
            if (Feasible(curr_tour, nbr)) {  
                Add_city(curr_tour, nbr);  
                Push_copy(stack, curr_tour);  
                Remove_last_city(curr_tour);  
            }  
    }  
    Free_tour(curr_tour);  
}
```

- 并行分析：如何通信（数据交换）？
 - 并行更新过程

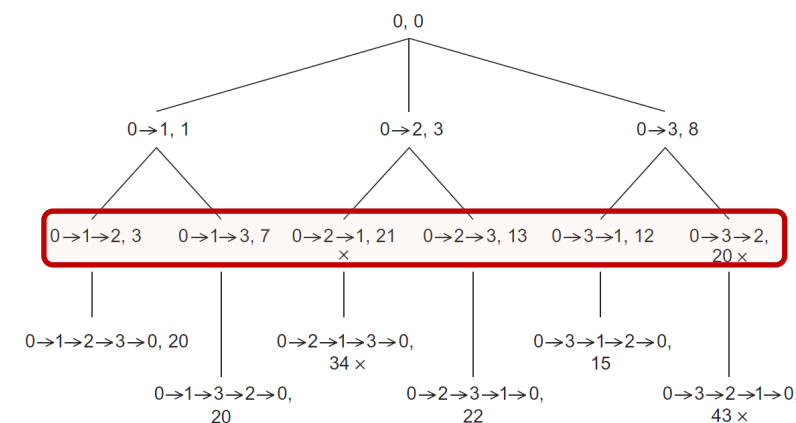


- 并行分析：如何通信（数据交换）？
 - 并行更新过程



• Pthreads并行实现（静态任务分配）

- 基本思路：由一个线程产生所有子任务
 - 广度优先搜索，产生 p 个不完整路径
 - 每个线程接着其中一个路径，完成该子树的搜索



– 并行需要处理的问题

- 需要使用局部的my_stack替换stack
 - 初始化stack
- 实现并行的Best_tour
- 实现并行的Update_best_tour

– 问题：负载可能不均衡

- 探索式划分无法预估开销

```
Partition_tree(my_rank, my_stack);
```

```
while (!Empty(my_stack)) {
    curr_tour = Pop(my_stack);
    if (City_count(curr_tour) == n) {
        if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
    } else {
        for (city = n-1; city >= 1; city--)
            if (Feasible(curr_tour, city)) {
                Add_city(curr_tour, city);
                Push_copy(my_stack, curr_tour);
                Remove_last_city(curr_tour)
            }
        }
    Free_tour(curr_tour);
}
```

• Pthreads并行实现（动态任务分配）

– 基本思路：在线程间交换任务

- 线程**完成**任务（栈为空）：**等待**其他线程**分配**任务
- 线程有多个任务（栈元素 >2 ）：**分割**栈，将部分路径分配给其他线程

– 实现

- 线程**完成栈内**任务时
 - 检查是否**所有线程**都已经完成（而非检查栈）
- 线程**依然有**任务时
 - 检查栈中是否有**多于2个任务**
 - 检查是否有**线程等待**
 - 检查**new_stack**变量是否已经创建
 - » **new_stack**：在线程间交换任务

- Pthreads并行实现（动态任务分配）
 - 使用条件变量（condition variables, 课件4）
 - 总是与锁同时使用

有额外任务分配的线程

```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
    new_stack == NULL) {
    lock term_mutex;
    if (threads_in_cond_wait > 0 && new_stack == NULL) {
        Split my_stack creating new_stack;
        pthread_cond_signal(&term_cond_var);
    }
    unlock term_mutex;
    return 0; /* Terminated = False; don't quit */
} else if (!Empty(my_stack)) { /* Stack not empty, keep working */
    return 0; /* Terminated = false; don't quit */
} else { /* My stack is empty */
    lock term_mutex;
    if (threads_in_cond_wait == thread_count - 1) { /* Last thread */
                                                    /* running */
        threads_in_cond_wait++;
        pthread_cond_broadcast(&term_cond_var);
        unlock term_mutex;
        return 1; /* Terminated = true; quit */
    }
```

完成自身任务的线程

```
} else { /* Other threads still working, wait for work */
    threads_in_cond_wait++;
    while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
    /* We've been awakened */
    if (threads_in_cond_wait < thread_count) { /* We got work */
        my_stack = new_stack;
        new_stack = NULL;
        threads_in_cond_wait--;
        unlock term_mutex;
        return 0; /* Terminated = false */
    } else { /* All threads done */
        unlock term_mutex;
        return 1; /* Terminated = true; quit */
    }
} /* else wait for work */
} /* else my_stack is empty */
```

• Pthreads并行实现性能比较（15 cities）

- 无论是静态分配还是动态分配，性能都与问题相关
- 通常动态的可扩展性更大

Threads	First Problem			Second Problem		
	Serial	Static	Dynamic	Serial	Static	Dynamic
1	32.9	32.7	34.7 (0)	26.0	25.8	27.5 (0)
2		27.9	28.9 (7)		25.8	19.2 (6)
4		25.7	25.9 (47)		25.8	9.3 (49)
8		23.8	22.4 (180)		24.0	5.7 (256)

• OpenMP实现类似，但缺少了条件变量，可能需要忙等待

- 但总体性能依然相似

Th	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	OMP	Pth	OMP	Pth	OMP	Pth	OMP	Pth
1	32.5	32.7	33.7 (0)	34.7 (0)	25.6	25.8	26.6 (0)	27.5 (0)
2	27.7	27.9	28.0 (6)	28.9 (7)	25.6	25.8	18.8 (9)	19.2 (6)
4	25.4	25.7	33.1 (75)	25.9 (47)	25.6	25.8	9.8 (52)	9.3 (49)
8	28.0	23.8	19.2 (134)	22.4 (180)	23.8	24.0	6.3 (163)	5.7 (256)

◉ MPI并行：静态任务分配

- 分布式内存系统没有共享的全局变量
 - 使用**进程0**管理全局信息
- 进程0的任务
 - 将图数据**广播**给所有进程
 - **产生最初**的任务（广度优先搜索产生相应分支）
 - 实现中可以使用MPI_Scatterv代替MPI_Scatter，处理任务无法均分的情况
 - 同样，可以使用MPI_Gatherv代替MPI_Gather
 - 搜索**结束**后，确保进程0有**最优路径**的数据
 - 使用MPI_Allreduce

◉ MPI并行：静态任务分配

– 进程0的任务

- 将图数据广播给所有进程；产生最初任务；汇总结果

– 其他进程的任务

- 搜索子树，并更新最优路径（需要广播给所有进程）
 - 然而，MPI_Bcast会阻塞进程，
 - 使用MPI_Send发送（用特殊tag标记，如NEW_COST_TAG）
 - » MPI_Send：标准版本，行为依赖于实现
 - » MPI_Ssend：同步（阻塞）版本
 - » MPI_Rsend：必须先收到匹配的Recv（否则错误）
 - » MPI_Bsend：发送时先拷贝到用户提供的缓存（buffer）
 - 使用MPI_Iprobe定期检查是否有相关标记的数据更新
 - » 有的话调用MPI_Recv接收；直接调用MPI_Recv可能导致挂起

◉ MPI并行：动态任务分配

- 思路与此前Pthreads的动态分配相似
 - 完成自身任务：发送请求获取工作
 - 有多余任务且有进程等待工作：切分任务，发送给等待进程
- 更新最优路径可使用静态任务分配的相同方式
- 如何检查结束条件？：维护一个**能量计数**
 - 初始状态下，所有进程能量为1
 - 进程完成自身任务后，将自身能量发送给进程0
 - 进程切分任务时，将能量也做对应切分，并一起发送给相应进程
 - **进程0**收集到**comm_size**的能量时，所有任务完成

◉ MPI并行：性能对比

- 线程/进程数目较少时，Pthreads性能更好
- 对于大规模问题，MPI可扩展性更强，性能更好

Th/Pr	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	Pth	MPI	Pth	MPI	Pth	MPI	Pth	MPI
1	35.8	40.9	41.9 (0)	56.5 (0)	27.4	31.5	32.3 (0)	43.8 (0)
2	29.9	34.9	34.3 (9)	55.6 (5)	27.4	31.5	22.0 (8)	37.4 (9)
4	27.2	31.7	30.2 (55)	52.6 (85)	27.4	31.5	10.7 (44)	21.8 (76)
8		35.7		45.5 (165)		35.7		16.5 (161)
16		20.1		10.5 (441)		17.8		0.1 (173)

设计思路

- Forster's methodology: 划分→通信→聚合→映射
- 注意区分哪些是内在的依赖关系, 哪些是实现带来的依赖关系

数据/任务划分

- 数据划分: 块/循环/循环块...
- 递归划分: 适用于并行分治
- 探索式划分: 适用于并行回溯

实现

- 共享内存系统: 哪些变量需要共享? 如何保护共享变量?
- 分布式系统: 哪些消息需要传递? 如何传递? 何处需要阻塞?

Questions?

