中山大學
SUN YAT-SEN UNIVERSITY

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

并行程序设计与算法

并行算法设计

陶钧

taoj23@mail.sysu.edu.cn

中山大学 计算机学院

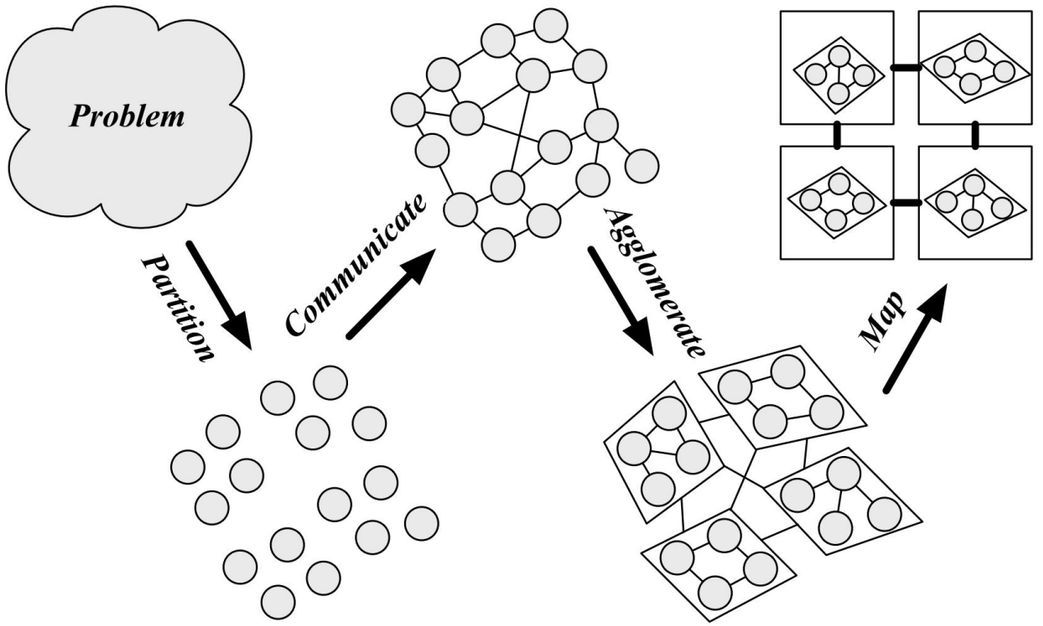
国家超级计算广州中心

课程提纲

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

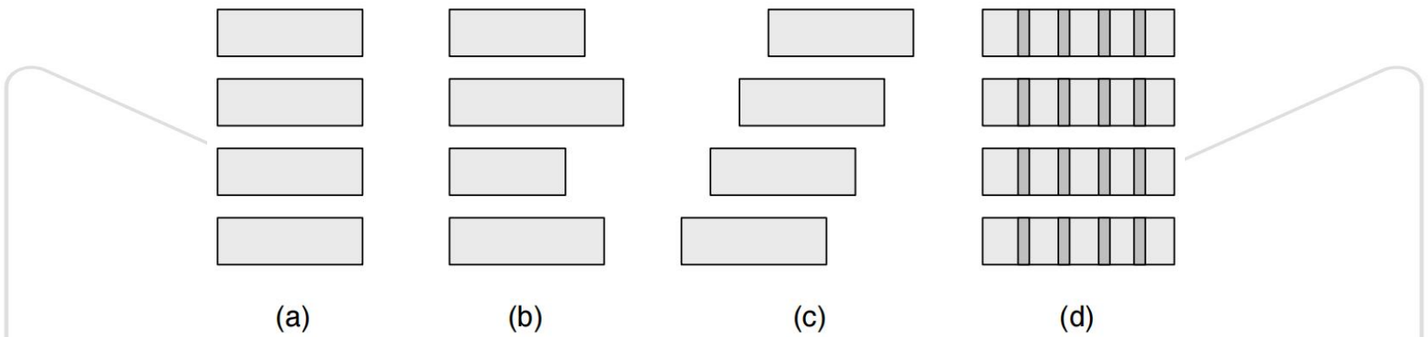
并行算法设计概述

- 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？



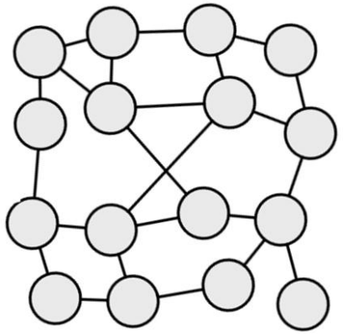
并行算法设计概述

- 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）

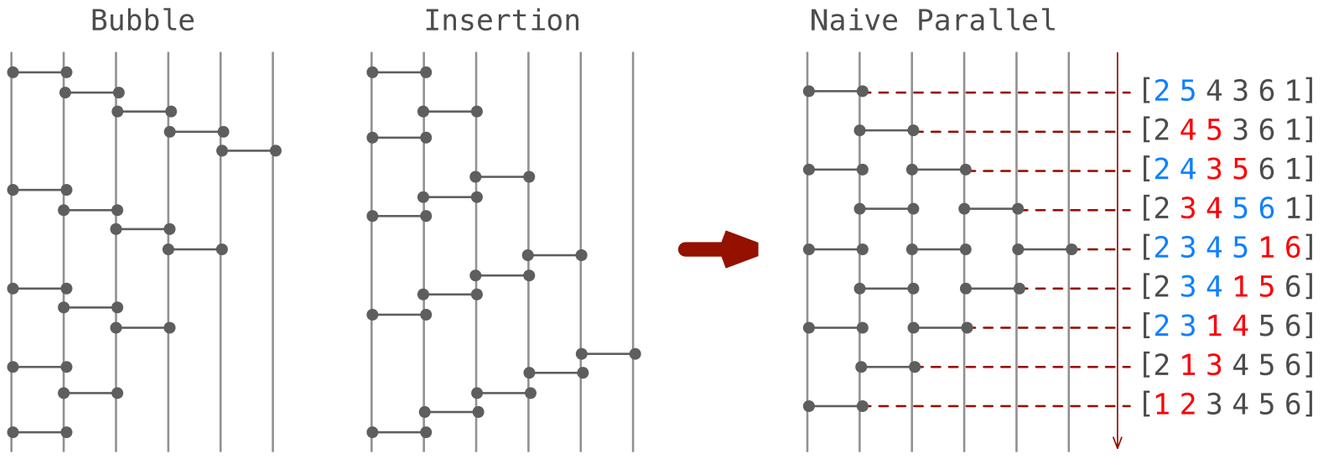


并行算法设计概述

- 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”



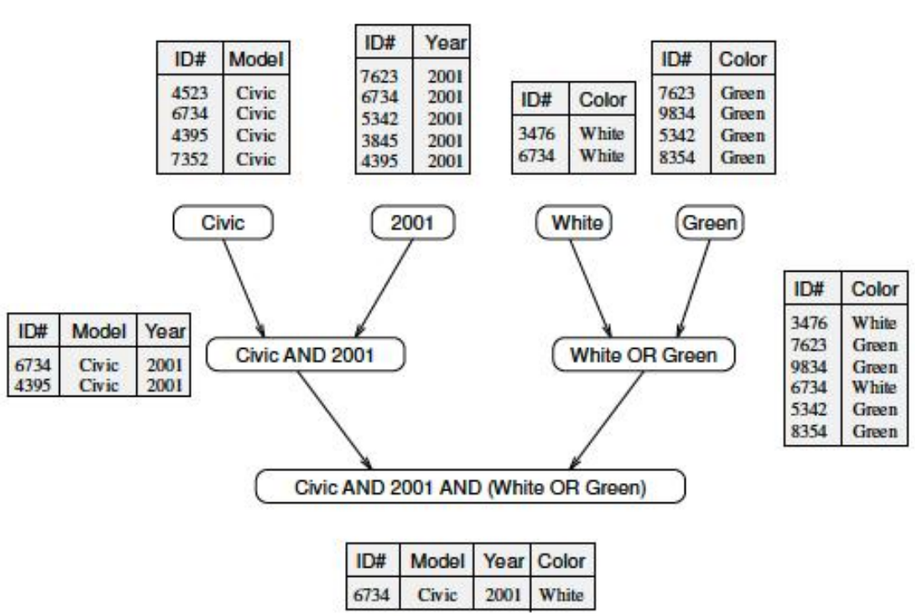
  - 通过改变计算方式消除非必须的“依赖关系”
    - $s^2 = \frac{1}{n-1}\sum(x_i - \bar{x})^2$：需串行两次规约，平方和计算依赖于均值计算
    - $s^2 = \frac{1}{n-1}(\sum x_i^2 - \frac{1}{n}(\sum x_i)^2)$：两次规约可并行（甚至可以合并成一次）

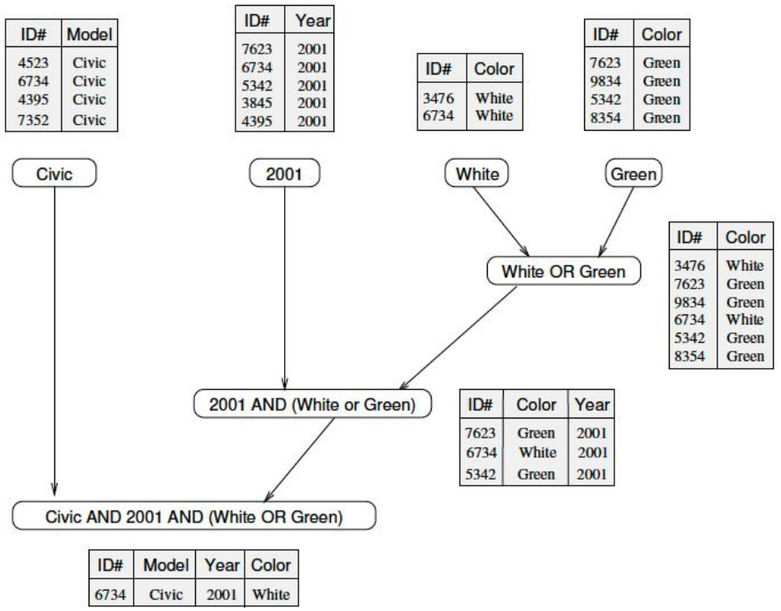
课程提纲

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

划分策略

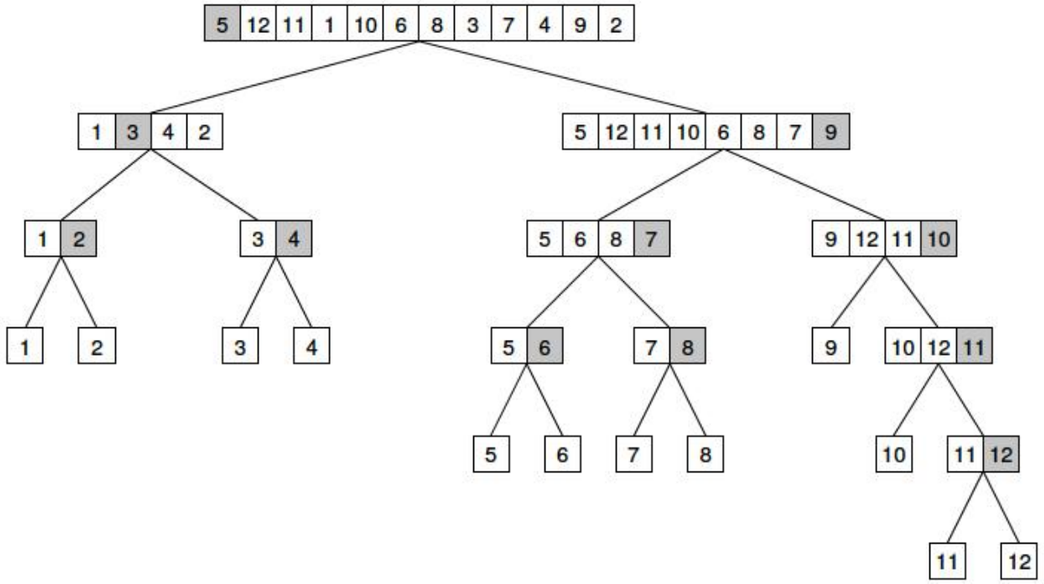
- 递归划分策略
  - 通常适用于并行化所有分治法算法
    - 问题能被拆分成为子问题（子问题的解有助于解决原始问题）
  - 不断递归直到问题被划分至合适的颗粒度
  - 例1：下图中所表示的数据库查询操作哪个更好？





划分策略

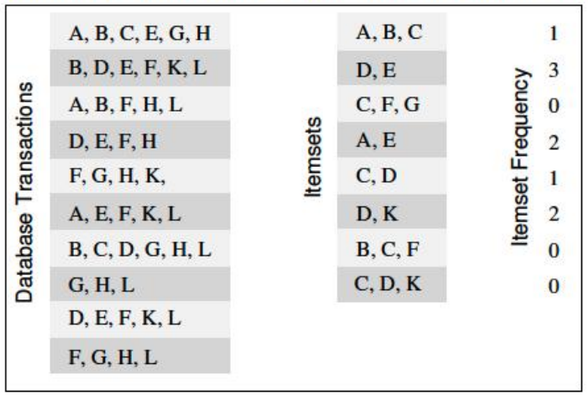
- 递归划分策略
  - 通常适用于并行化所有分治法算法
    - 问题能被拆分成为子问题（子问题的解有助于解决原始问题）
  - 不断递归直到问题被划分至合适的颗粒度
  - 例2：快速排序 vs 归并排序：哪个更适合并行？



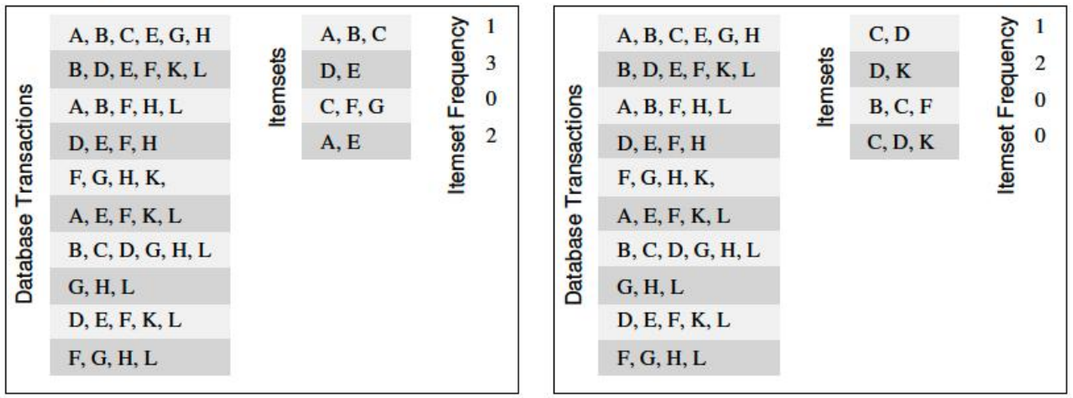
划分策略

- 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)

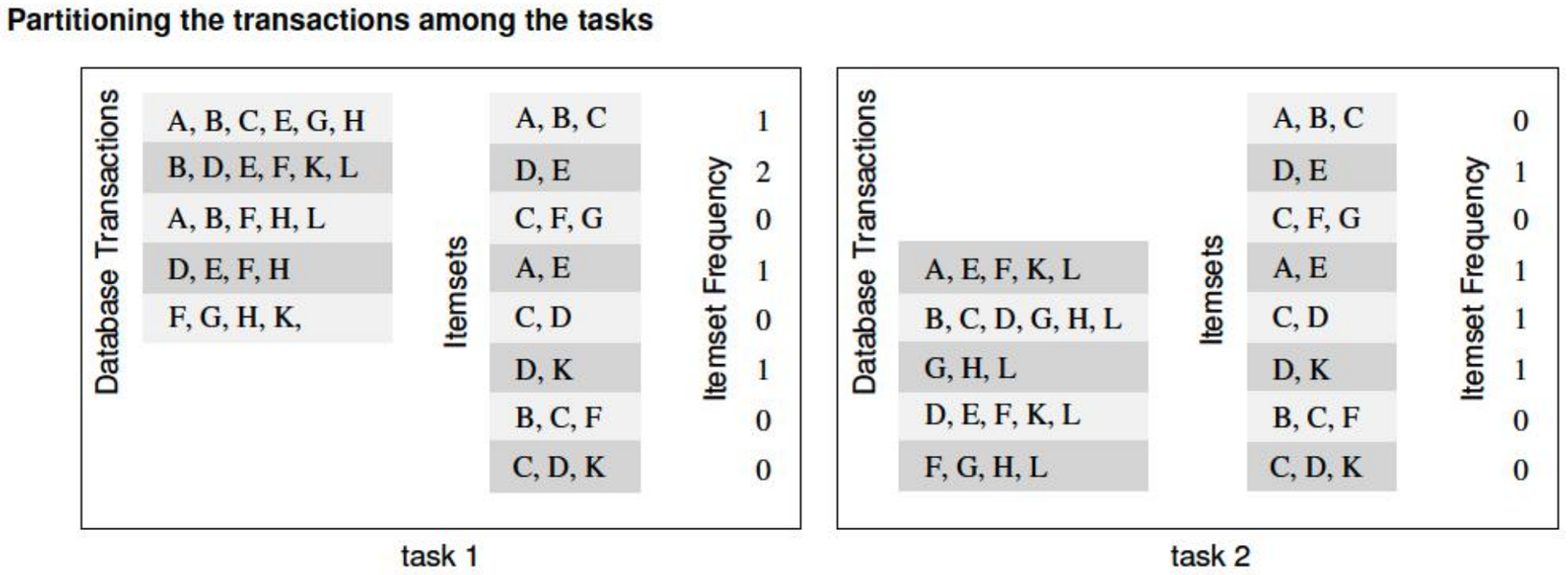


(b) Partitioning the frequencies (and itemsets) among the tasks



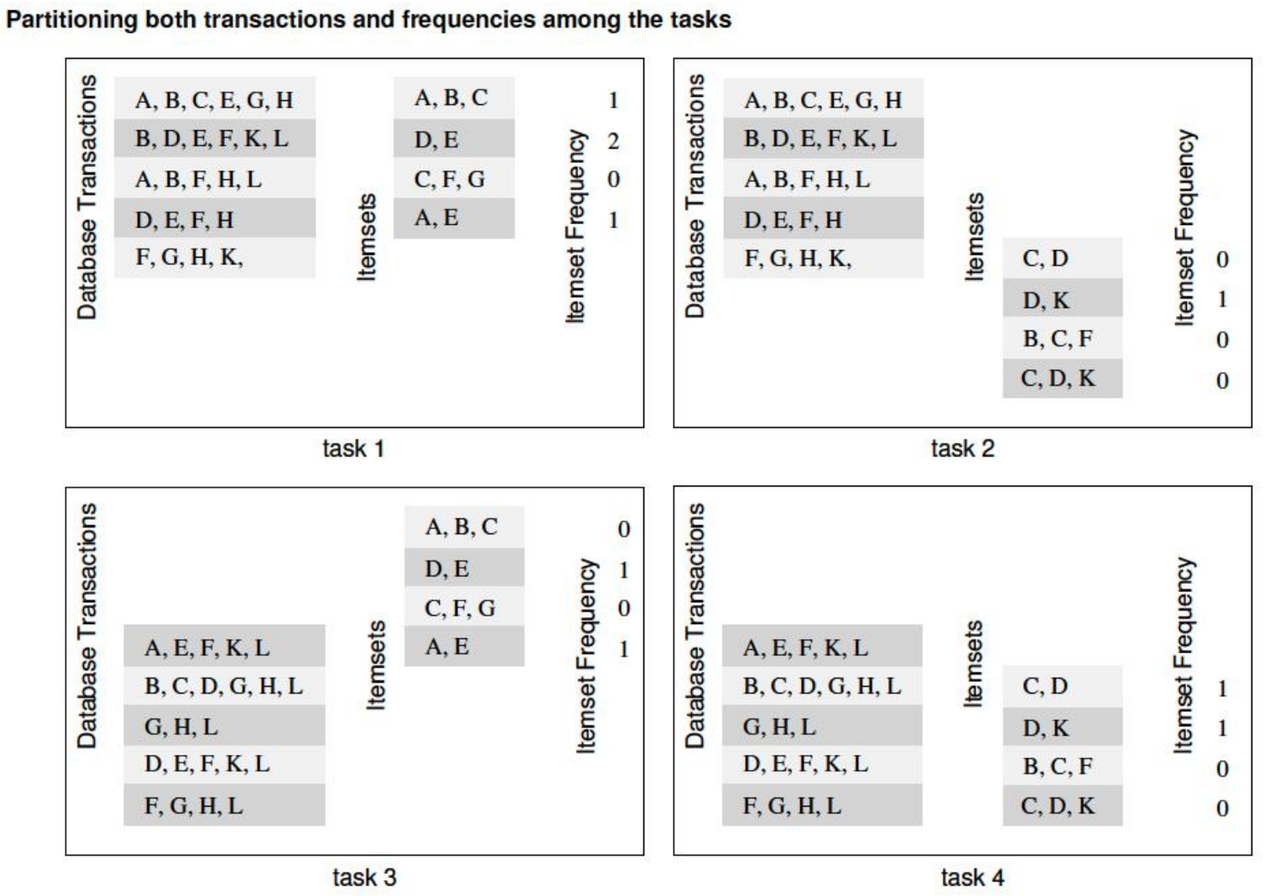
划分策略

- 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



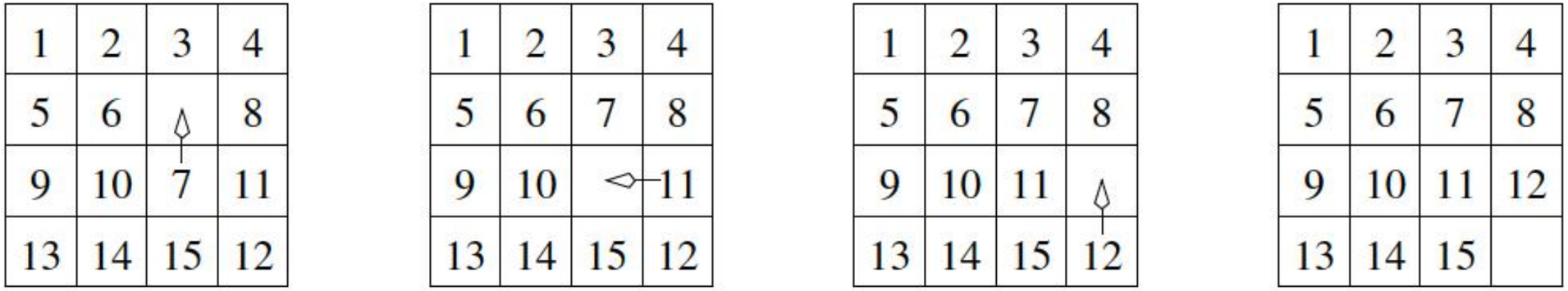
划分策略

- 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



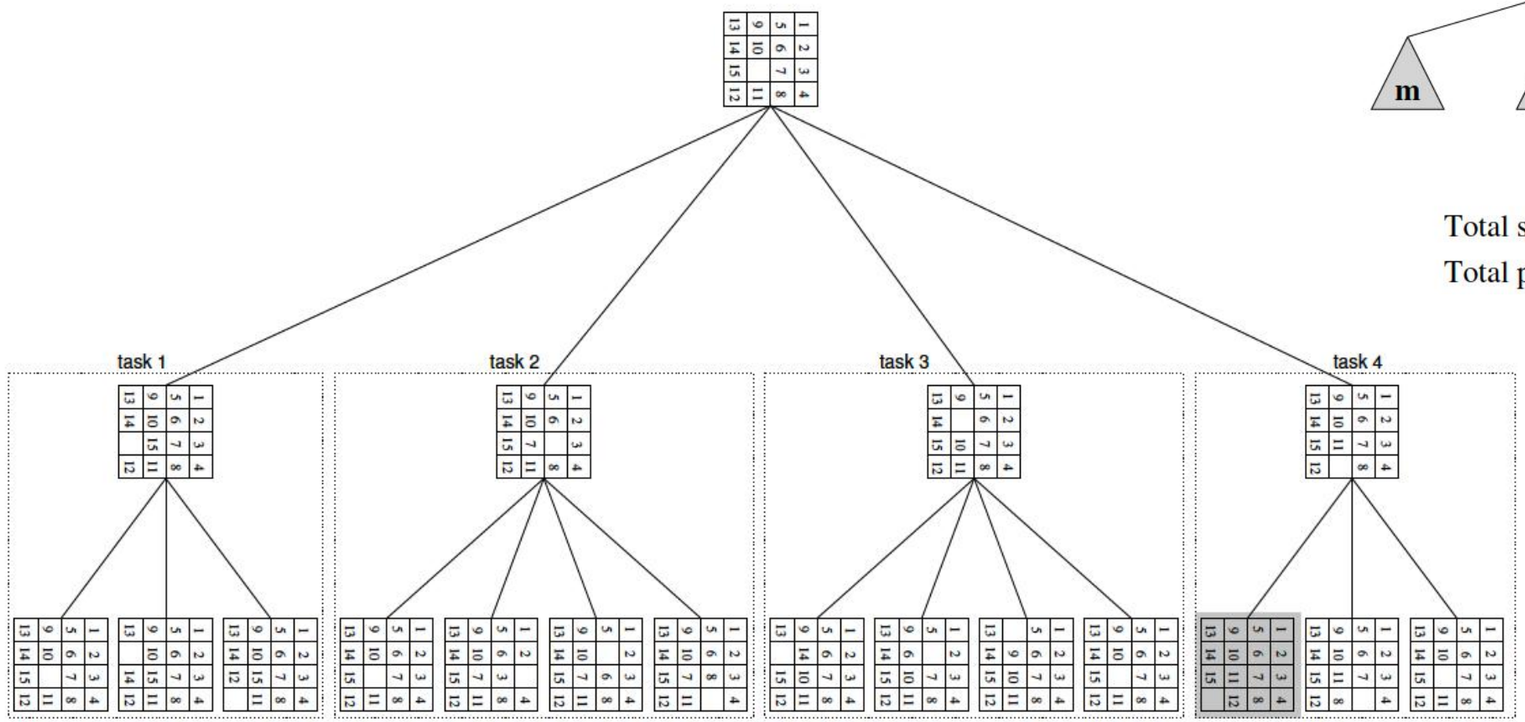
划分策略

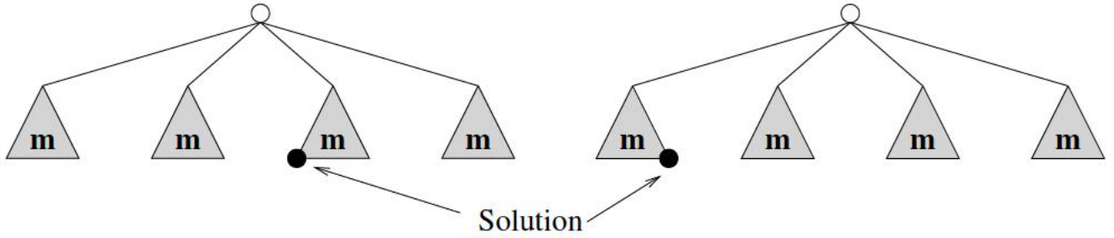
- 探索式划分
  - 某些问题求解过程中的计算量是不确定的，因此无法提前规划
  - 常见例子为最优解的搜索，使用并行计算可同时探索多条路径
    - 但无法提前规划所有路径
  - 例：滑动拼图
    - 每个计算节点负责从一个特定初始状态出发搜索到最优解的路径



划分策略

- 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



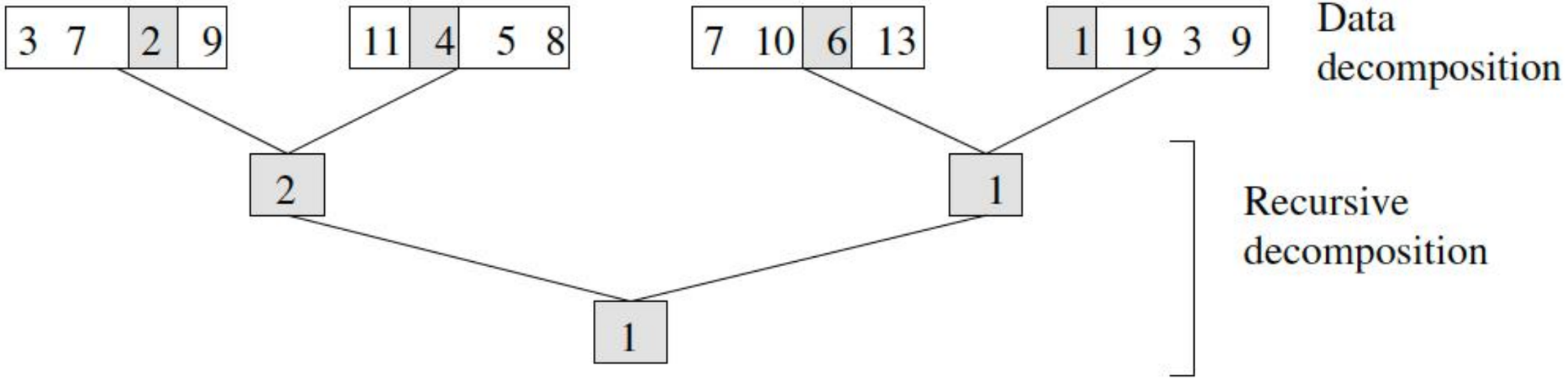


Total serial work: 2m+1
Total parallel work: 1

Total serial work: m
Total parallel work: 4m

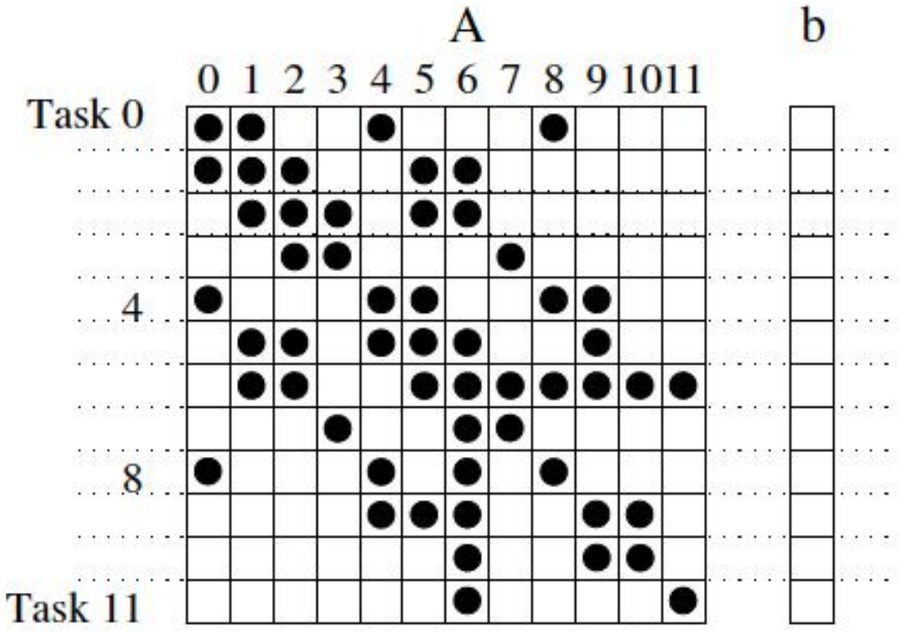
划分策略

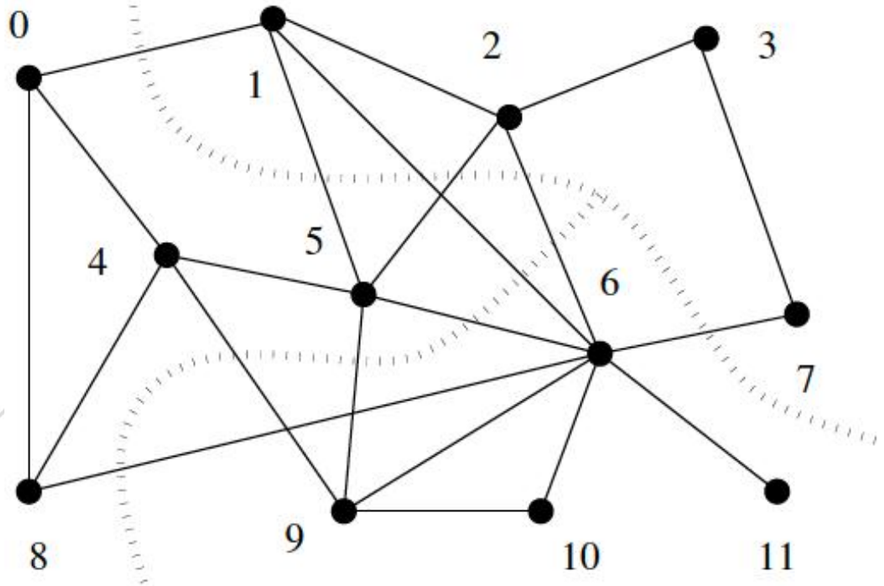
- 混合划分策略
  - 单一的划分策略往往无法取得最优效果
    - 快速排序中，递归划分并发度往往并不高
    - 并行归约中，为提高占用率，往往每个线程先分别进行串行归约
      - Brent's theorem



划分策略

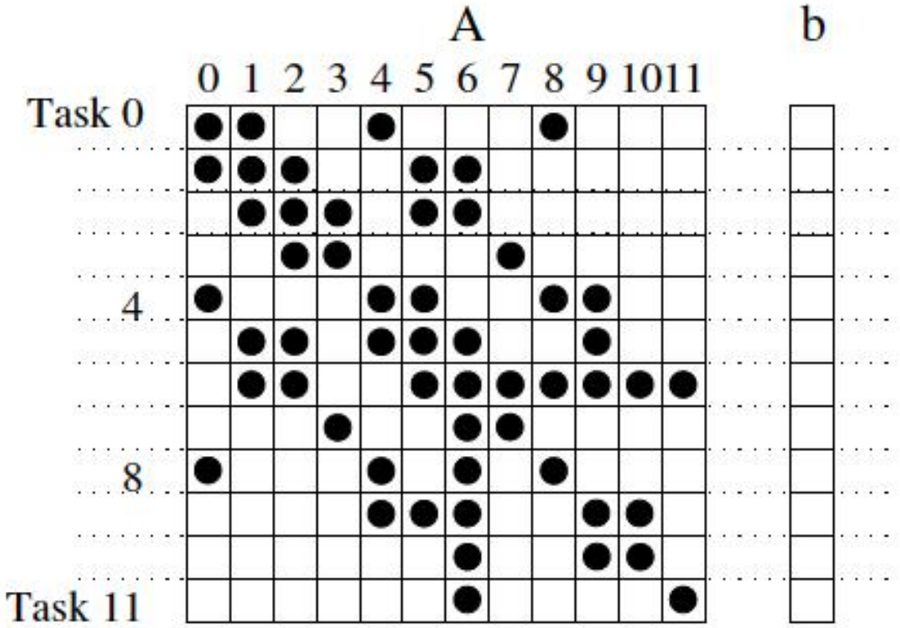
- 基于依赖关系的任务划分
  - 常在任务图关系上通过图算法或优化方法划分节点
    - 最大化数据局部性（减少数据重载入/通信）
    - 最小化空闲时间（提高并发度）
    - 最小化最大负载（负载均衡）

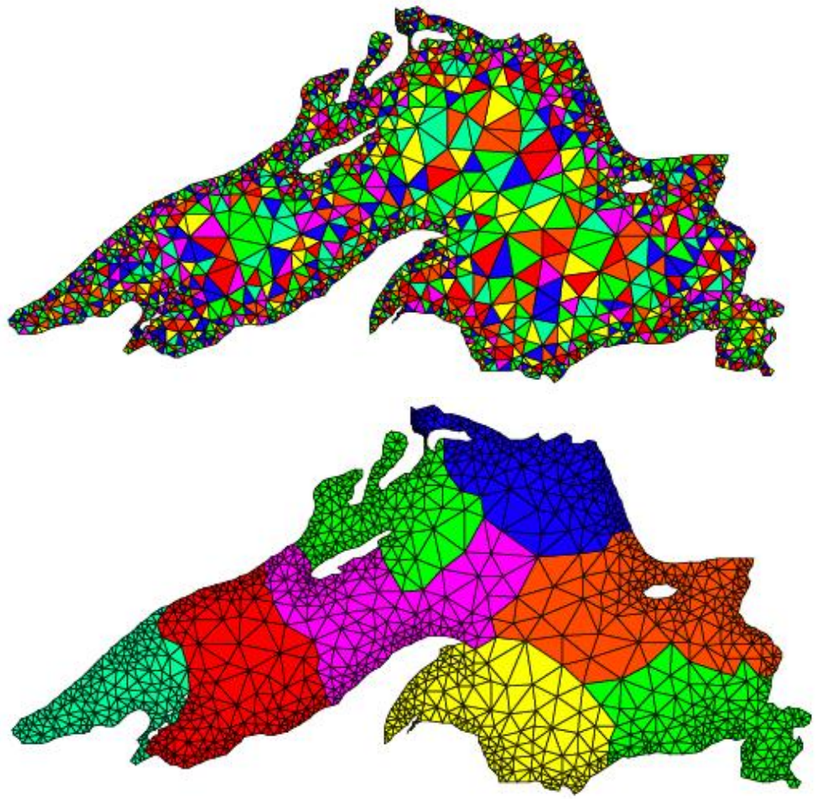




划分策略

- 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≠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





划分策略

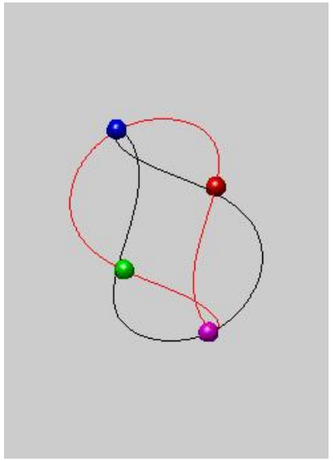
- 静态划分 vs 动态划分
  - 以上方式大多为静态划分
  - 实践中为了保持负载均衡，常在运行过程中调整计算节点任务
    - 多用于探索式划分及基于任务图的划分
- 任务划分的实现模型
  - 主从（master-slave）模型：也可用于静态
    - 一个或多个主进程产生任务，从进程执行计算
    - 任务池模型
  - 管线/生产者-消费者模型
    - 一个或一组进程负责数据处理管线中的一个环节

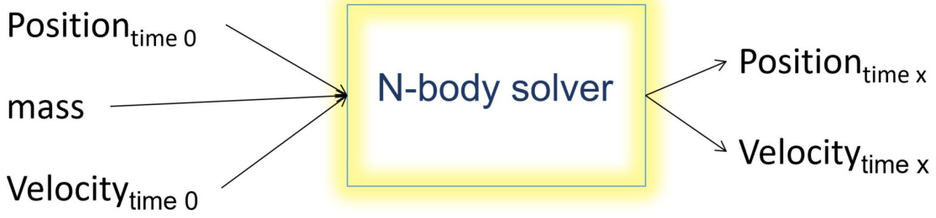
课程提纲

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯

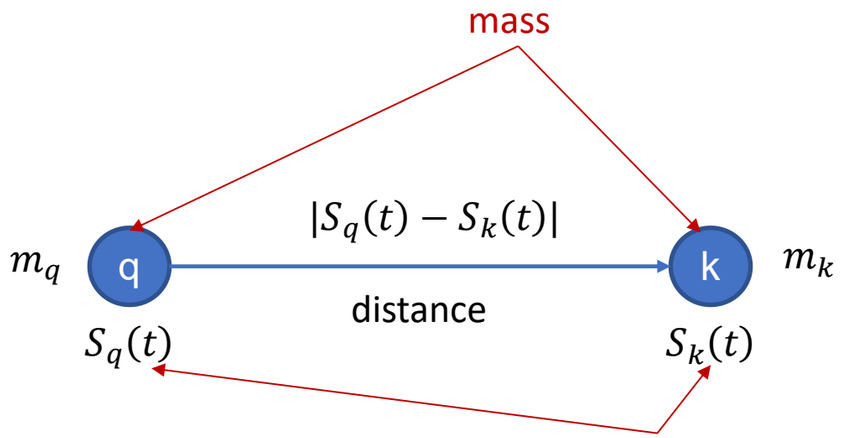
直接划分与实现

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





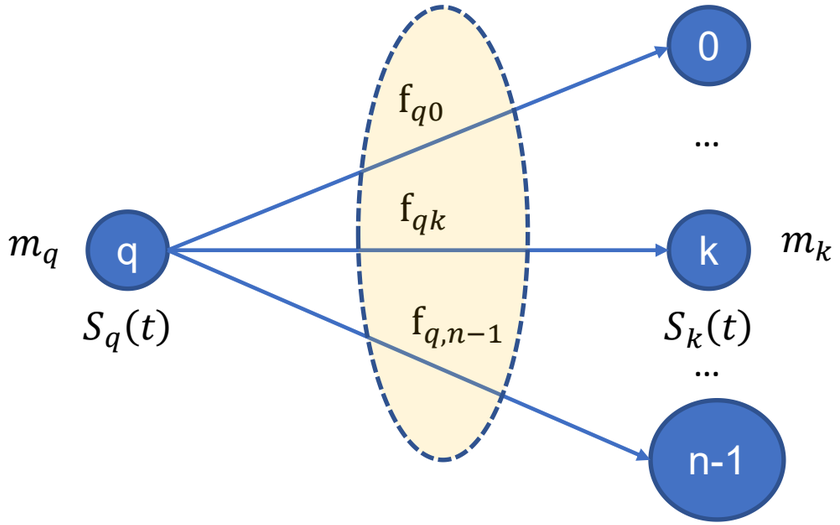
- 模拟假设
  - 牛顿第二运动定律：$F = ma$
  - 粒子间相互作用力满足万有引力定律：$F = G\frac{m_q m_k}{r^2}$



G: gravitational constant

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right] \qquad (6.1)$$

- 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



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

- 模拟假设
  - 根据牛顿第二定律可推导粒子的运动状态

$$\mathrm{F}_q(t) = m_q a_q(t) = m_q S_q''(t)$$

$$a_q(t) = S_q''(t) = \frac{\mathrm{F}_q(t)}{m_q}$$

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

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

N体问题

- 实现（伪码）

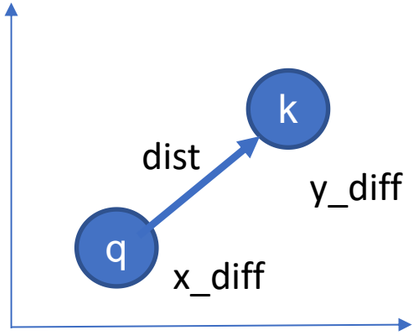
```
Get input data;
for each timestep {
    if (timestep output) Print positions and velocities of particles;
    for each particle q
        Compute total force on q;
    for each particle q
        Compute position and velocity of q;
}
Print positions and velocities of particles;
```

N体问题

- 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

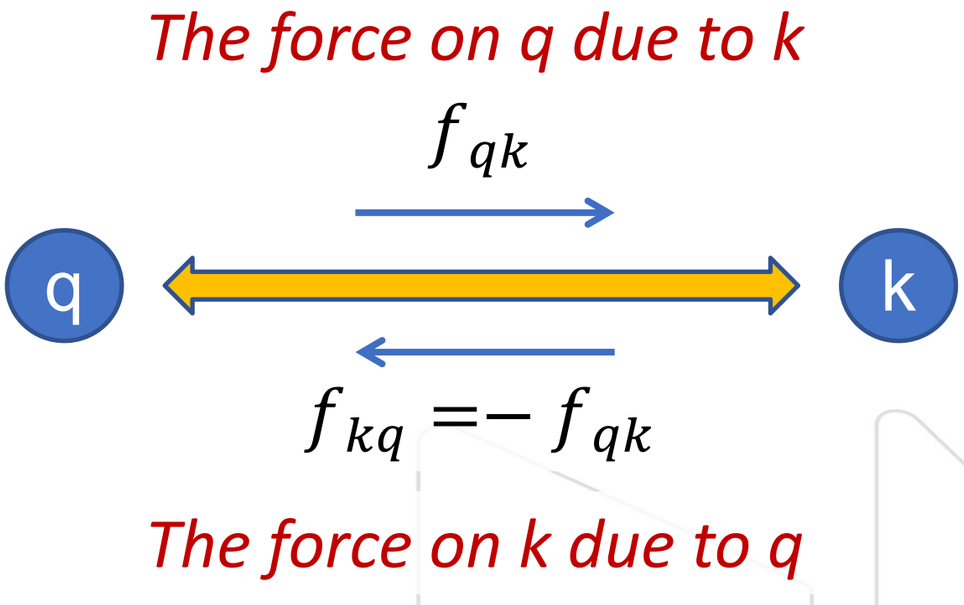
```
union vec2 {
    struct {
        float x, y;
    };
    float c[2];
};
```

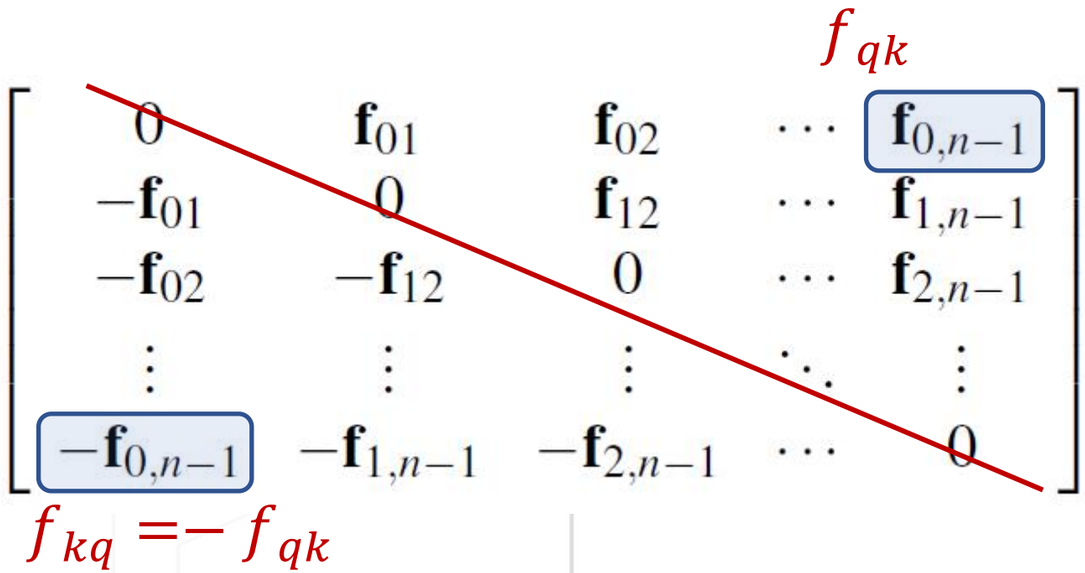
```
for each particle q {
    for each particle k != q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
       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



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

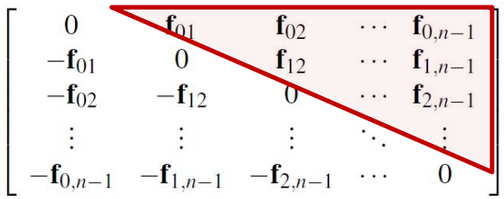
- 实现（计算作用力）
  - 力的作用是相互的，因此对于一对粒子，只需计算一次作用力
    - 将粒子间的相互作用力存储于一个矩阵时，只需要计算上/下三角矩阵





N体问题

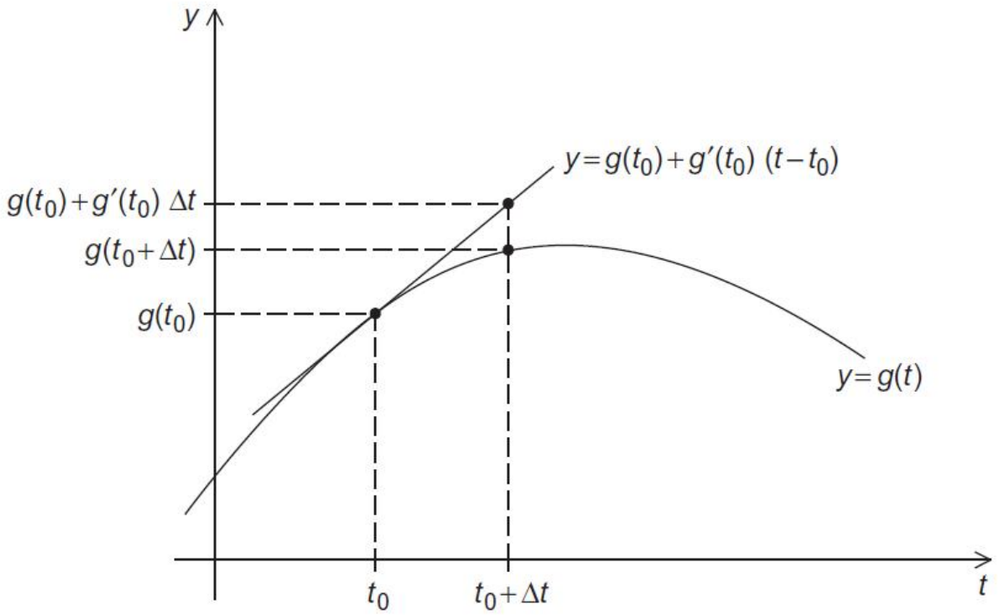
- 实现（计算作用力）
  - 力的作用是相互的，因此对于一对粒子，只需计算一次作用力
    - 计算后同时更新两个粒子所受到的合力（简化算法）

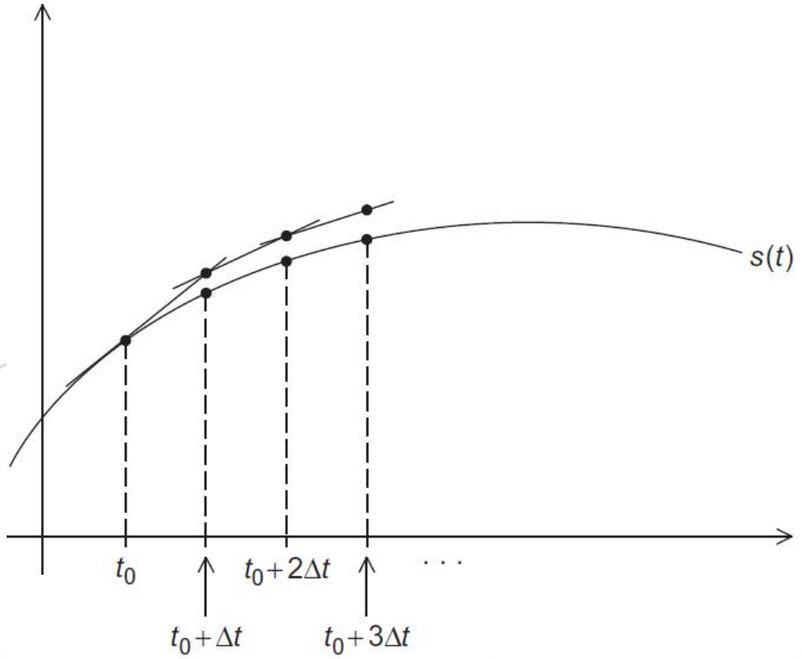


$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} &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

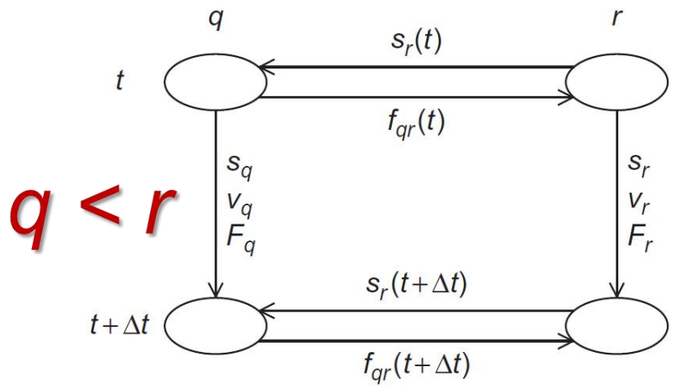
```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed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

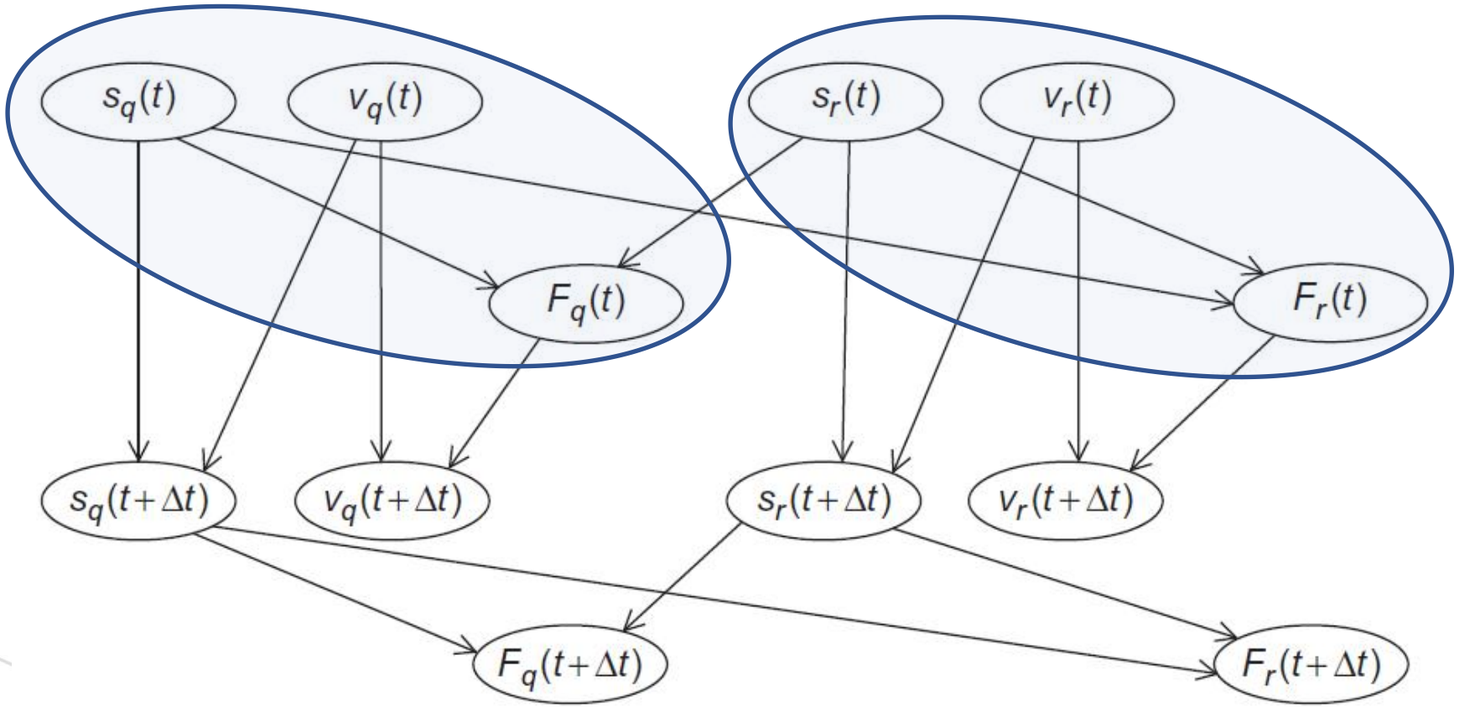
- 实现（状态更新）
  - 作用力计算给出了加速度，如何用其更新位置与速度？
    - 加速度为速度的导数，而速度为位置的导数
    - 近似：加速度反映了速度的变化量，速度反映了位置的变化量
    - 计算（欧拉方法）：假设极小范围内，该变化量为常数





- 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

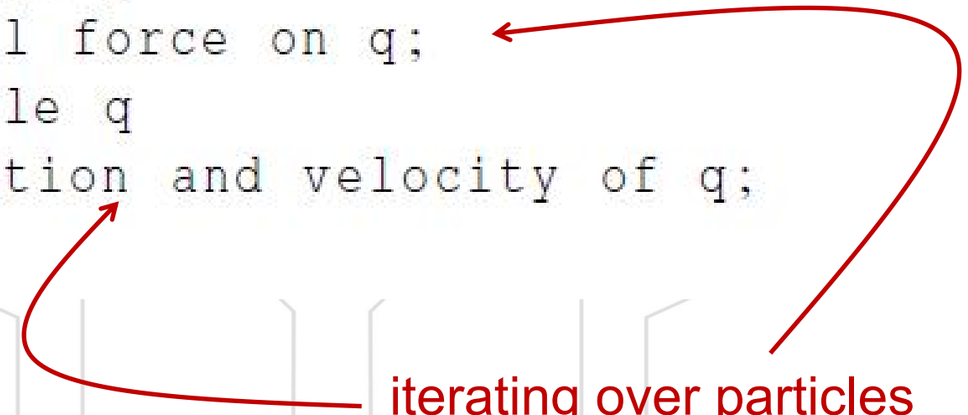




N体问题

- 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;
}
```



N体问题

- 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;
}
```

N体问题

- 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];
   }
```

N体问题

- 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) 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omp parallel，重用线程

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

N体问题

如何实现简化算法？

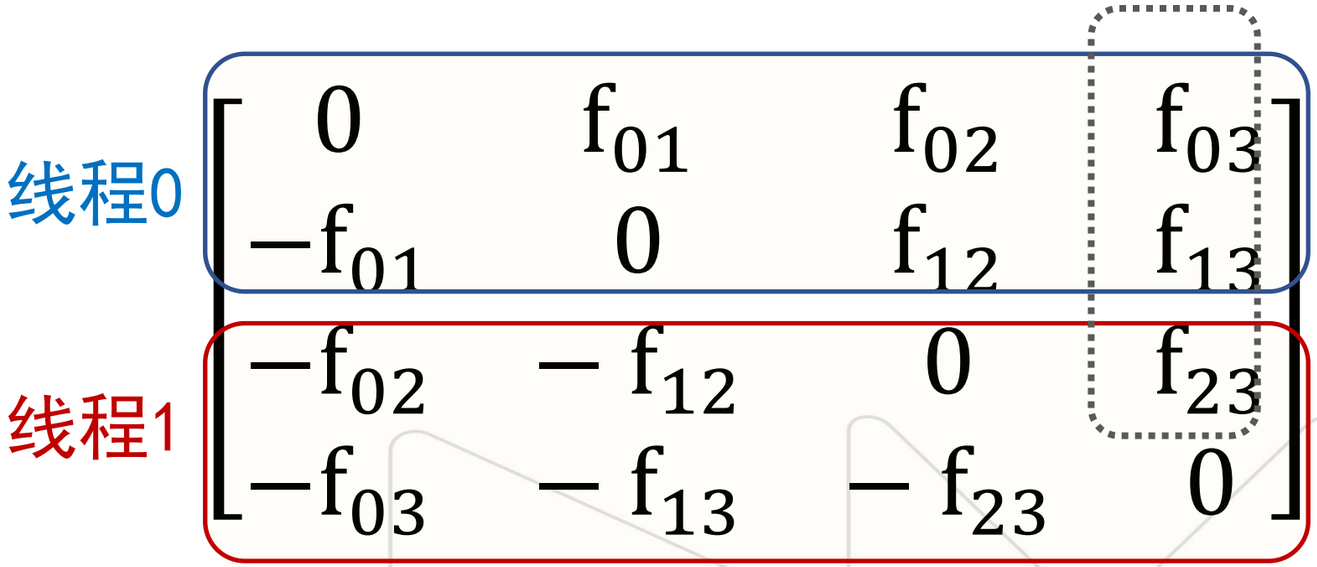
$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} &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

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

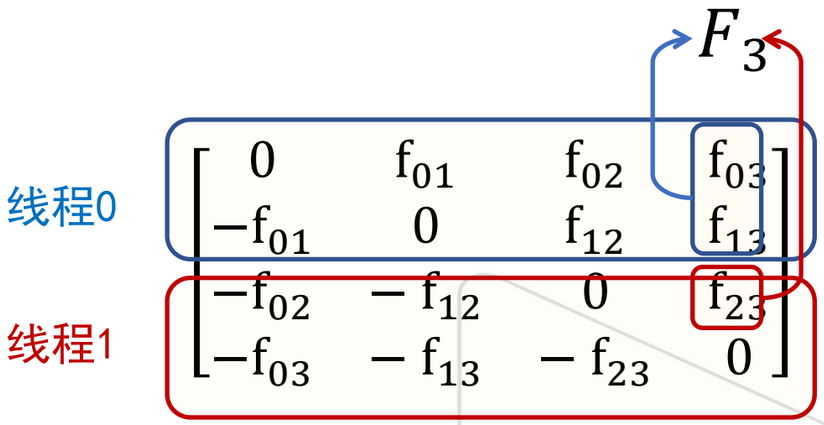
- OpenMP实现（简化算法）：依赖关系分析
  - 力计算：每个线程计算一个粒子所受合力
    - 以粒子3所受合力为例：$F_3 = -f_{03} - f_{13} - f_{23}$
    - 更新*forces[3]*时可能出现竞争条件（对于共享内存并行框架而言）



更新F_3(forces[3])
所需数据跨过多个线程

N体问题

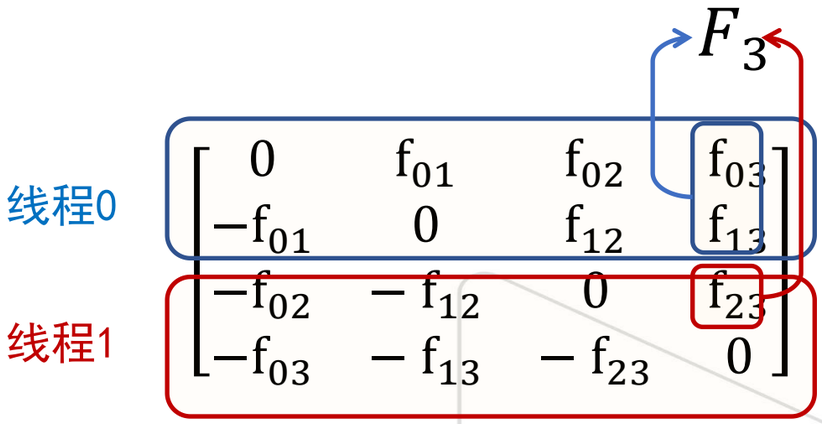
- OpenMP实现（简化算法：依赖关系分析
  - 更新forces[q]时可能出现竞争条件
  - 解决方案1：使用临界区保护对forces[]数组的更新
    - 计算force_qk后，由对应该线程线程更新forces[q]及forces[k]
    - 更新可能被高度串行化




```
#  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];
   }
```

N体问题

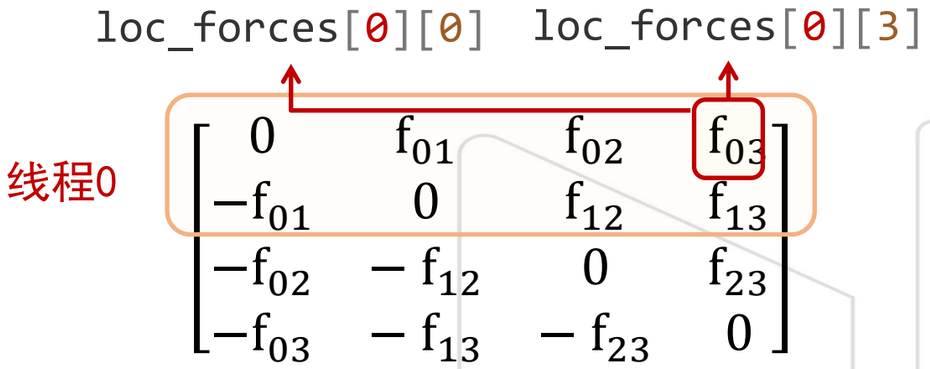
- OpenMP实现（简化算法）：依赖关系分析
  - 更新forces[q]时可能出现竞争条件
  - 解决方案2：使用锁保护单个forces[q]的更新
    - 每个变量对应一个锁对象
    - 但性能依然不如串行版本



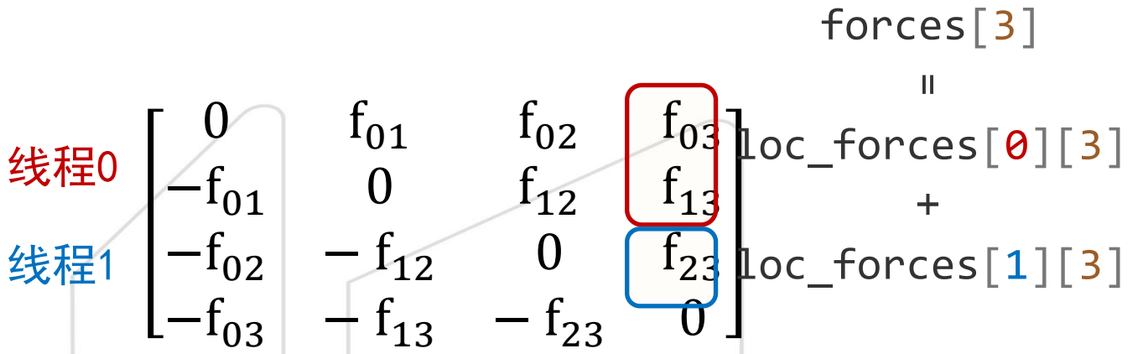

```
omp_set_lock(locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(locks[q]);

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

N体问题

- 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]



$$\begin{bmatrix} 0 & f_{01} & f_{02} & f_{03} \\ -f_{01} & 0 & f_{12} & f_{13} \\ -f_{02} & -f_{12} & 0 & f_{23} \\ -f_{03} & -f_{13} & -f_{23} & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & f_{01} & f_{02} & f_{03} \\ -f_{01} & 0 & f_{12} & f_{13} \\ -f_{02} & -f_{12} & 0 & f_{23} \\ -f_{03} & -f_{13} & -f_{23} & 0 \end{bmatrix}$$

N体问题

- 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

$$\begin{bmatrix} 0 & f_{01} & f_{02} & f_{03} \\ -f_{01} & 0 & f_{12} & f_{13} \\ -f_{02} & -f_{12} & 0 & f_{23} \\ -f_{03} & -f_{13} & -f_{23} & 0 \end{bmatrix}$$

N体问题

- 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

forces[3]

=

loc_forces[0][3]

+

loc_forces[1][3]

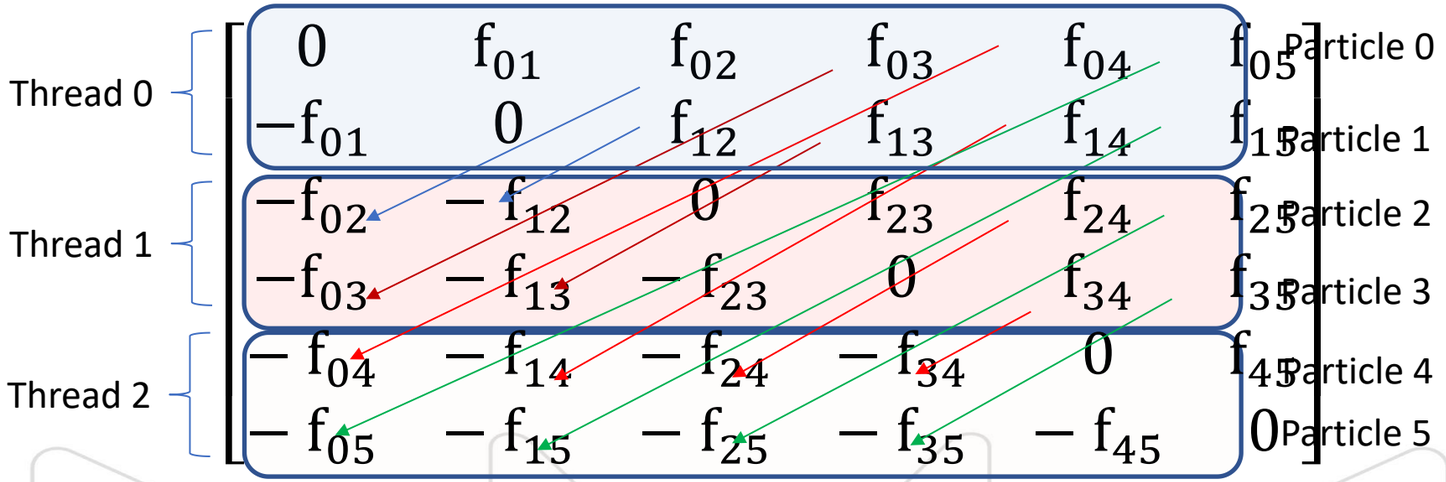
线程0

线程1

$$\begin{bmatrix} 0 & f_{01} & f_{02} & f_{03} \\ -f_{01} & 0 & f_{12} & f_{13} \\ -f_{02} & -f_{12} & 0 & f_{23} \\ -f_{03} & -f_{13} & -f_{23} & 0 \end{bmatrix}$$

```
#  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];
      }
   }
```

- 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



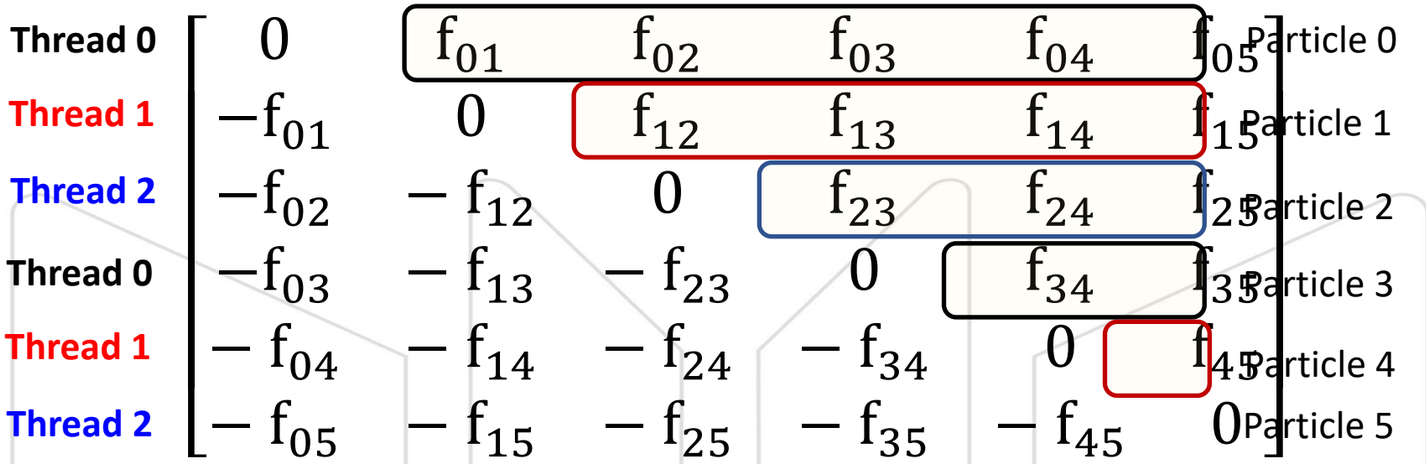
- OpenMP实现（简化算法）：负载均衡分析
  - 如何划分数据？
    - 当前对作用力矩阵的行使用了块划分
    - 每个线程处理连续的数行
    - 负载并不均衡
      - 每个线程工作量为其处理的矩阵元素数目×2

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

		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	$-f_{05}-f_{15}$	$-f_{25}-f_{35}$	$-f_{45}$

N体问题

- 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？



N体问题

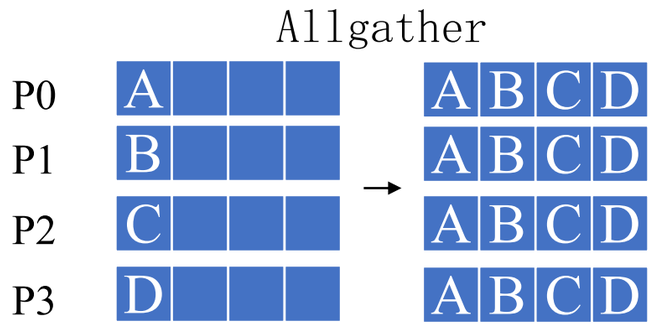
Pthreads实现分析

- 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）

N体问题

- 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

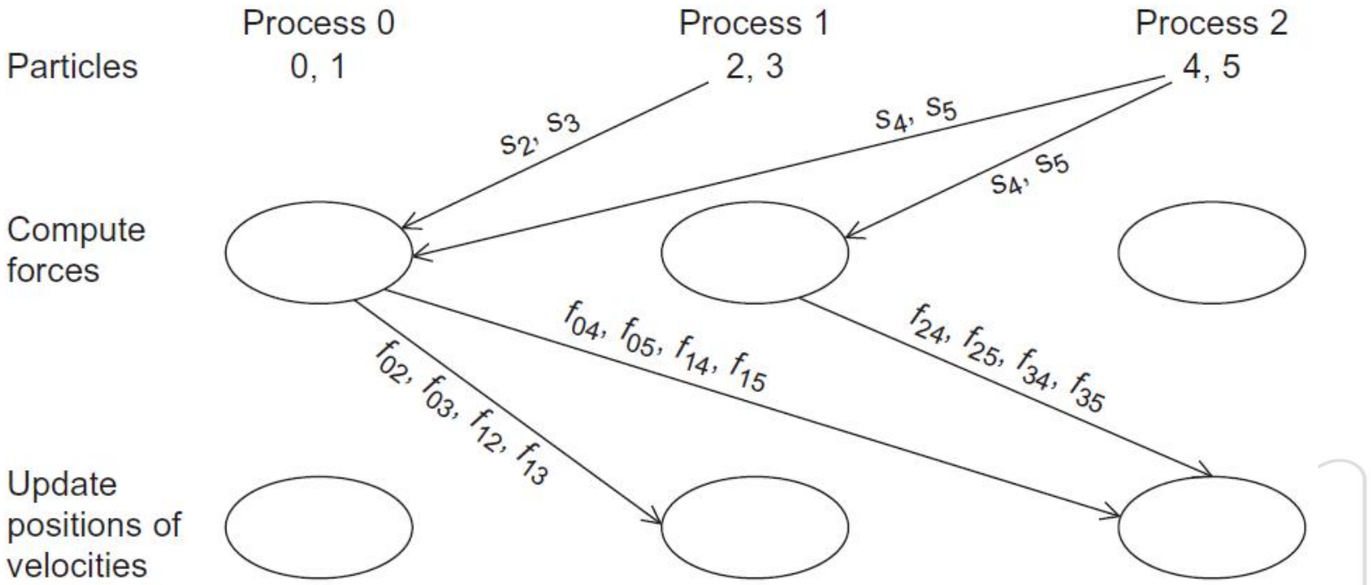
```
Get input data;
for each timestep {
   if (timestep output)
      Print positions and velocities of particles;
   for each local particle loc_q
      Compute total force on loc_q;
   for each local particle loc_q
      Compute position and velocity of loc_q;
   Allgather local positions into global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



N体问题

- 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！

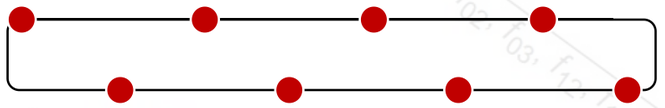
N体问题

- 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

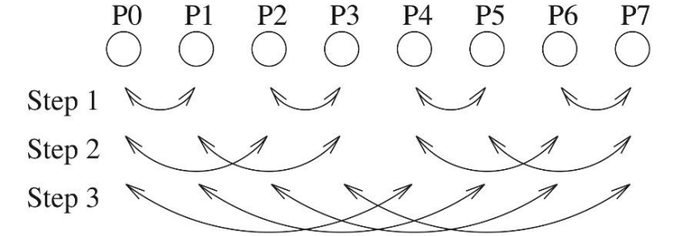
```
Get input data;
for each timestep {
   if (timestep output)
      Print positions and velocities of particles;
   for each local particle loc_q
      Compute total force on loc_q;
   for each local particle loc_q
      Compute position and velocity of loc_q;
   Allgather local positions into global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

回顾：*Allgather*的实现方式（课件3）

Ring Algorithm



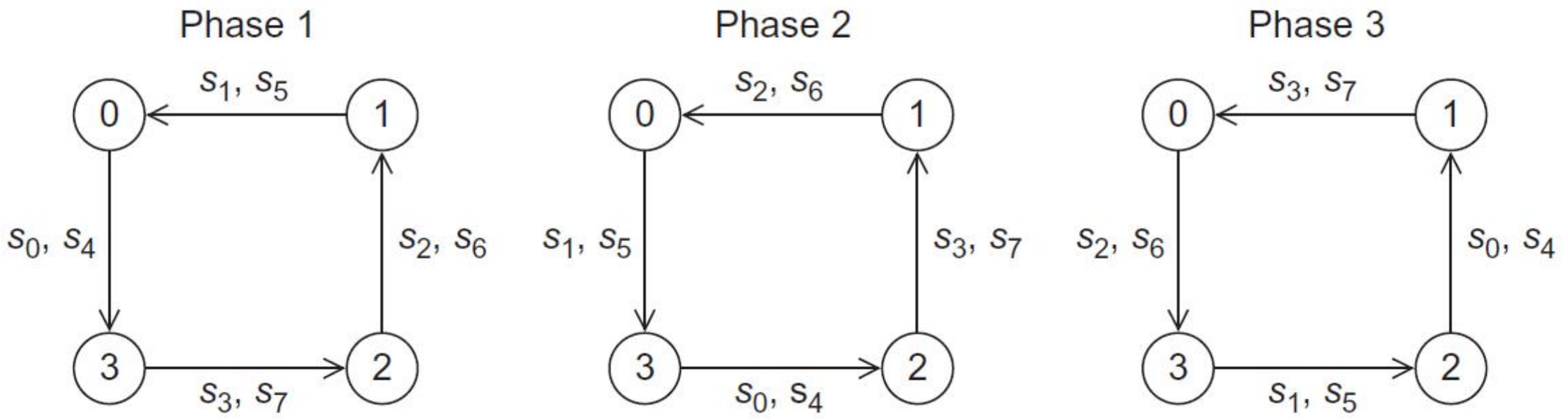
Recursive Doubling Algorithm



是否需要**同时**拥有所有粒子位置？

N体问题

- 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



N体问题

- 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

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

N体问题

- 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	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$0, 0$	$0, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, 0$	$0, 0$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}$	$0, -\mathbf{f}_{13}$
After First Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$0, -\mathbf{f}_{13}$	$0, -\mathbf{f}_{02}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12}+\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$
After Second Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12}+\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, -\mathbf{f}_{02}-\mathbf{f}_{12}+\mathbf{f}_{23}$	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}13-\mathbf{f}_{23}$

最终阶段：
合并计算结果
loc_forces+tmp_forces

N体问题

- 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

课程提纲

- 设计概述
- 划分策略
- 并行实现
- 并行分治
- 并行回溯