北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

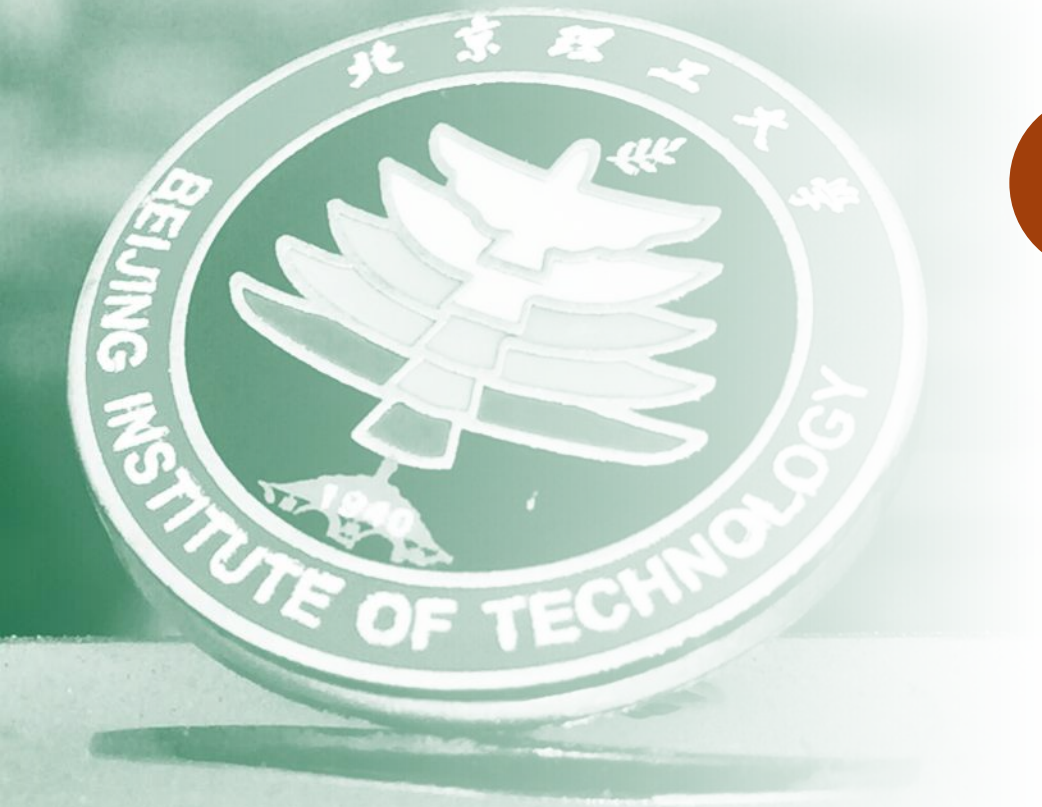
并行编程原理与实践

3. 并行编程方法

王一拙、计卫星

北京理工大学计算机学院

目录 | CONTENTS



1 并行编程的基本过程

并行编程的基本过程

串行 vs. 并行

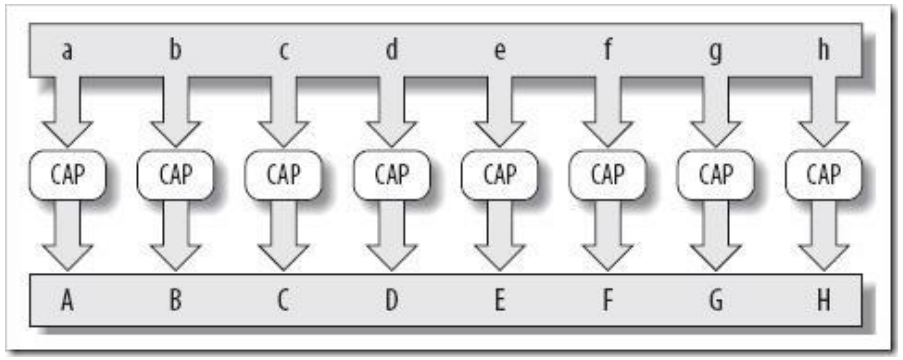
- 串行：一次只做一件事，按照顺序依次执行
- 并行：同时做多件事

并行实例

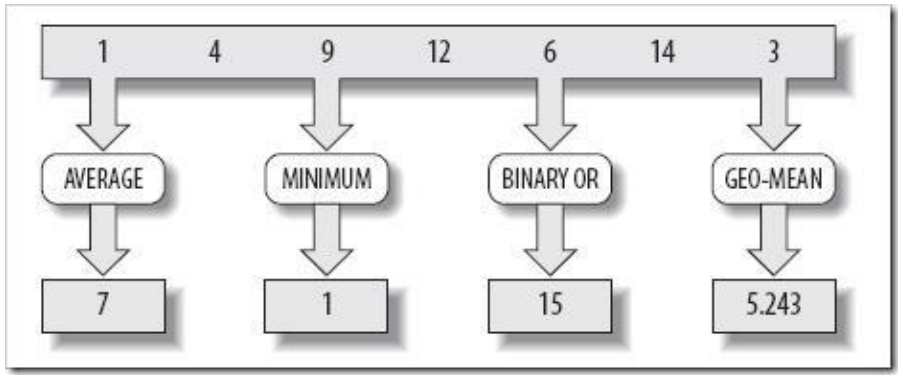
- 5名教师阅100份试卷，可以每人阅20份试卷，也可以每人阅部分题目
- 小朋友排队玩滑梯，如果有多个滑梯则可以多个人同时玩
- 装修过程中地板安装、橱柜安装、卫浴安装等可以同时进行
- 人在阅读书本时，可以边阅读，边解读字面意思，边根据其中意境加以想象，边背诵，这些“进程”之间从宏观来看，更像一种交错、并行的关系，而非绝对、死板的先后顺序

编写并行程序的两种基本方式

- 数据并行

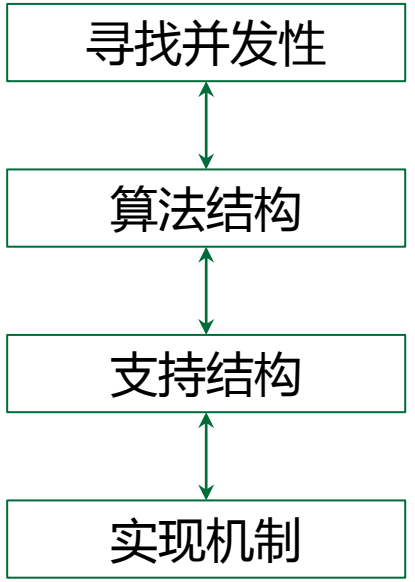


- 任务并行



并行编程过程的模式化描述

- 把并行编程的过程抽象成设计空间和模式
  - 寻找并发性设计空间
    - 任务分解
    - 数据分解
  - 算法结构设计空间
    - 分治、流水、任务并行、事件协作
  - 支持结构设计空间
    - 主从模式、SPMD
    - 共享队列、分布式队列
  - 实现机制设计空间
    - 进程/线程的管理、交互



并行程序的实现方法

- 隐式并行：由硬件和编译器自动实现串行程序的并行执行，一般在指令级或线程级实现，不需要程序员参与。如超标量、向量指令、硬件多线程等
- 显式并行：需要程序员参与来实现并行程序的划分、调度、同步与通信等细节实现，有以下三种并行程序实现方法

	实例	优点	缺点
库例程	MPI, Pthreads	容易使用；性能高；不需要扩展编译器	程序员要处理同步通信等细节
编译器注释	OpenMP, OpenACC	容易使用；代码可移植性好	受编译器限制；并行开销较大；灵活性较差
语言扩展	C++, Java	性能高；灵活性好	程序员要掌握语言高级特性，处理所有细节

并行程序的实现方法

库例程：Pthreads

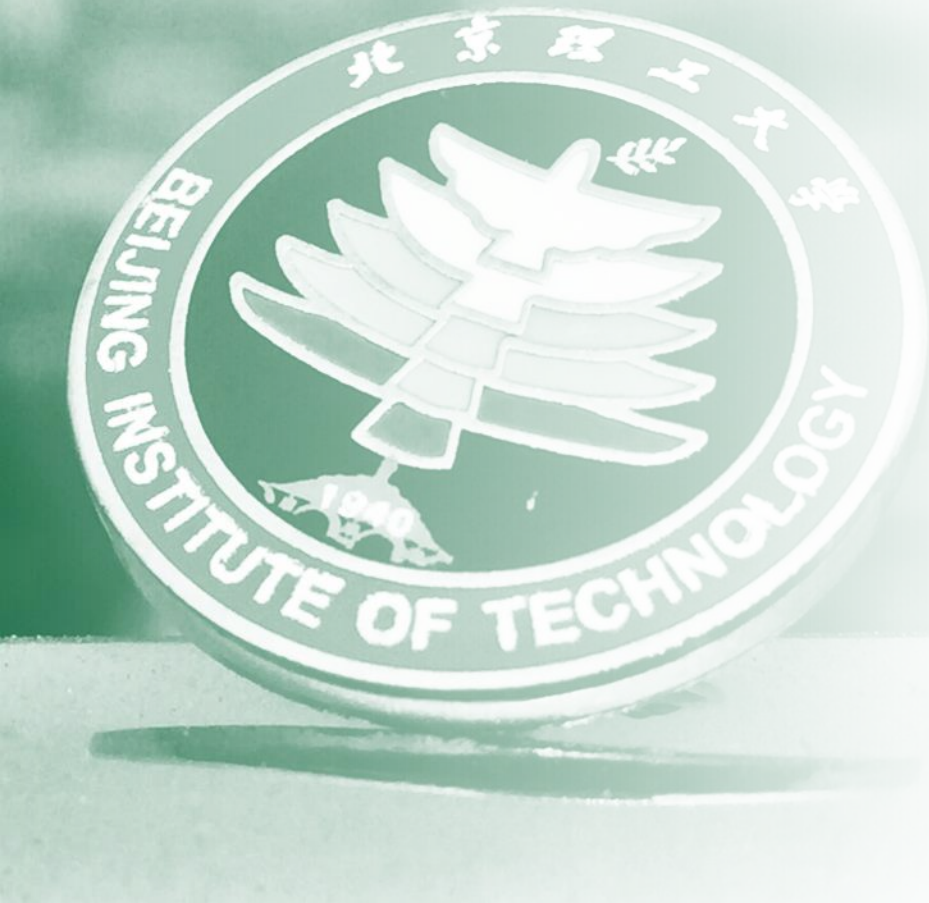
```
#include <pthread.h>
void*  function( void*  arg ){
   printf( "This is thread %d\n",
        pthread_self() );
   return( 0 );
}

int main( void ){
   pthread_attr_t attr;
   pthread_attr_init( &attr );
   pthread_create( NULL, &attr,
        &function, NULL );
   return EXIT_SUCCESS;
}
```

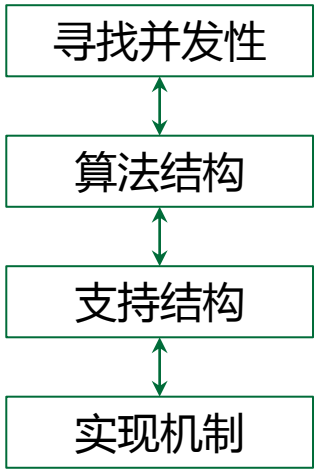
语言扩展：C++11

```
class background_task{
public:
   void operator()() const{
        do_something();
   }
};
background_task f;
std::thread my_thread(f);
```

编译器注释：OpenMP

```
#include <omp.h>
#define N 10000
void main()
{
  double A[N],B[N],C[N];
  ...
  #pragma omp parallel for
  for(int i=0; i<N; i++)
    C[i]=A[i]+B[i];
}
```


北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY
1940


寻找并发性
算法结构
支持结构
实现机制

2 寻找并发性

并行程序中的并发性与操作系统中的并发性

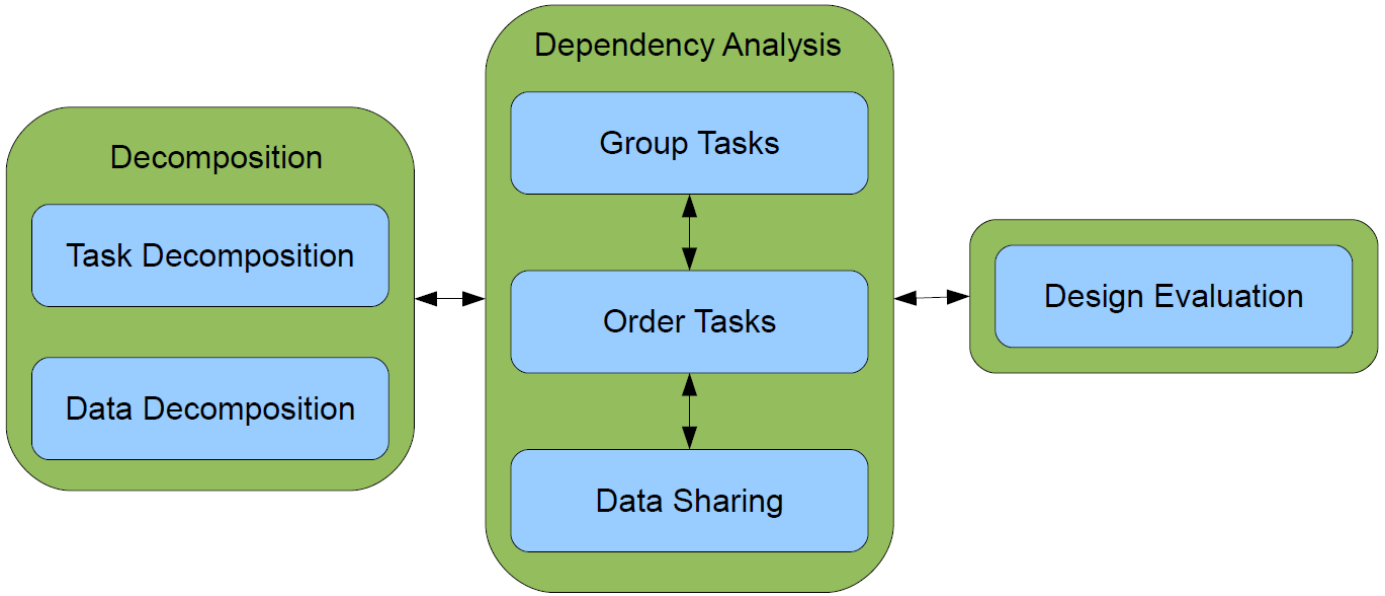
- 并发是操作系统最重要的特征，操作系统的并发性是指它应该具有处理和调度多个程序同时执行的能力。所以，对操作系统而言，问题不是寻找并发性，并发性是操作系统固有的特性。
- 操作系统的并发性主要是不同应用（进程、线程）之间的并发，粒度比较大，要考虑资源分配的公平性和系统的稳健性，因此性能目标通常与吞吐率和响应时间相关。
- 程序中的并发性是指一个应用程序内部的子任务之间的并发，可能体现为进程、线程或更小的粒度，如函数、代码段等，性能目标主要是最小化程序执行时间。

“寻找并发性”设计空间

- 过早的走出问题领域进入编程领域，可能会错过一些重要的设计选项。
- 设计者首先要考虑：问题规模是否足够大，以至于有必要用并行的方法来更快的解决问题？
- 然后需要确保问题中的关键特征和数据元素得到了很好的理解。
- 最后，要了解问题的哪些部分是计算最密集的部分，因为并行化任务的精力应聚焦在这些部分。

“寻找并发性”设计空间

- 分解模式
  - 任务分解
  - 数据分解
- 依赖性分析模式
  - 任务分组
  - 任务排序
  - 数据共享
- 设计评价模式

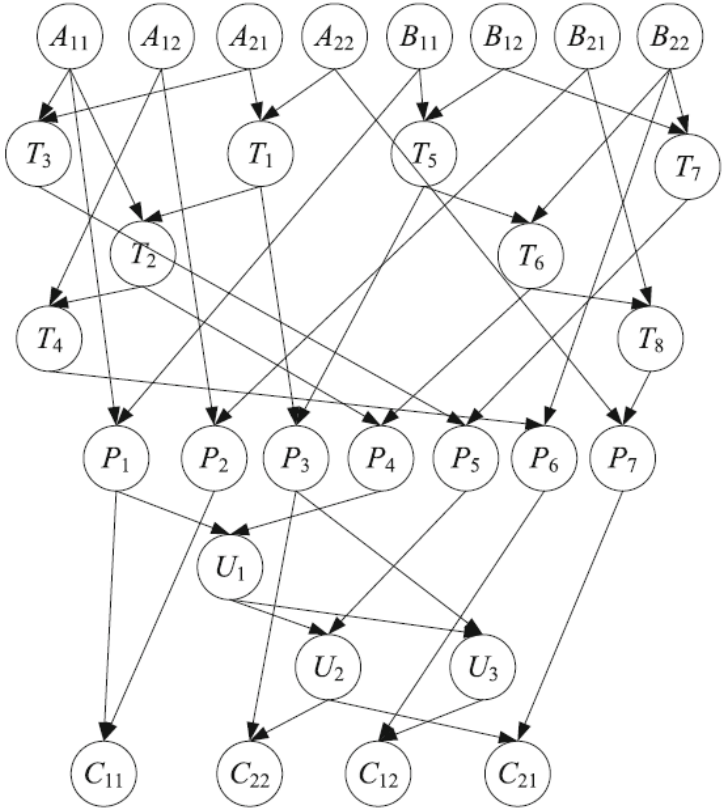


任务分解

- 示例：Winograd快速矩阵乘法

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Pre-additions	Recursive calls	Post-additions
$T_1 = A_{21} + A_{22}$	$P_1 = A_{11} * B_{11}$	$U_1 = P_1 + P_4$
$T_2 = T_1 - A_{11}$	$P_2 = A_{12} * B_{21}$	$U_2 = U_1 + P_5$
$T_3 = A_{11} - A_{21}$	$P_3 = T_1 * T_5$	$U_3 = U_1 + P_3$
$T_4 = A_{12} - T_2$	$P_4 = T_2 * T_6$	$C_{11} = P_1 + P_2$
$T_5 = B_{12} - B_{11}$	$P_5 = T_3 * T_7$	$C_{12} = U_3 + P_6$
$T_6 = B_{22} - T_5$	$P_6 = T_4 * B_{22}$	$C_{21} = U_2 + P_7$
$T_7 = B_{22} - B_{12}$	$P_7 = A_{22} * T_8$	$C_{22} = U_2 + P_3$
$T_8 = B_{21} + T_6$		



任务分解

- 将问题看成指令流，指令流中的指令能够分解成多个序列（任务），有些任务之间能并行执行。
- 如何将一个问题按功能分解成可能并发执行的任务？
  - 保证任务间足够独立，这样花费很小的代价就可以管理任务间的依赖关系；
  - 保证任务能够均匀的分布在所有的执行单元上。
- 从已有程序中发现任务的常见地方：
  - 函数调用
  - 算法循环
  - 数据分解

任务分解

- 考虑因素：
  - 灵活性：使设计能够适应拥有不同数量处理单元的并行计算机系统。
  - 效率：需要足够的任务来使得所有计算机都处于忙碌的状态；每个任务有足够量的工作来弥补创建任务、管理任务间的依赖等开销。
  - 简单：任务分解需要足够复杂以完成工作，但也需要足够简单以便于程序调试和维护。
- 建议：尽可能识别和划分很多任务，随后可以根据需要合并任务。

数据分解

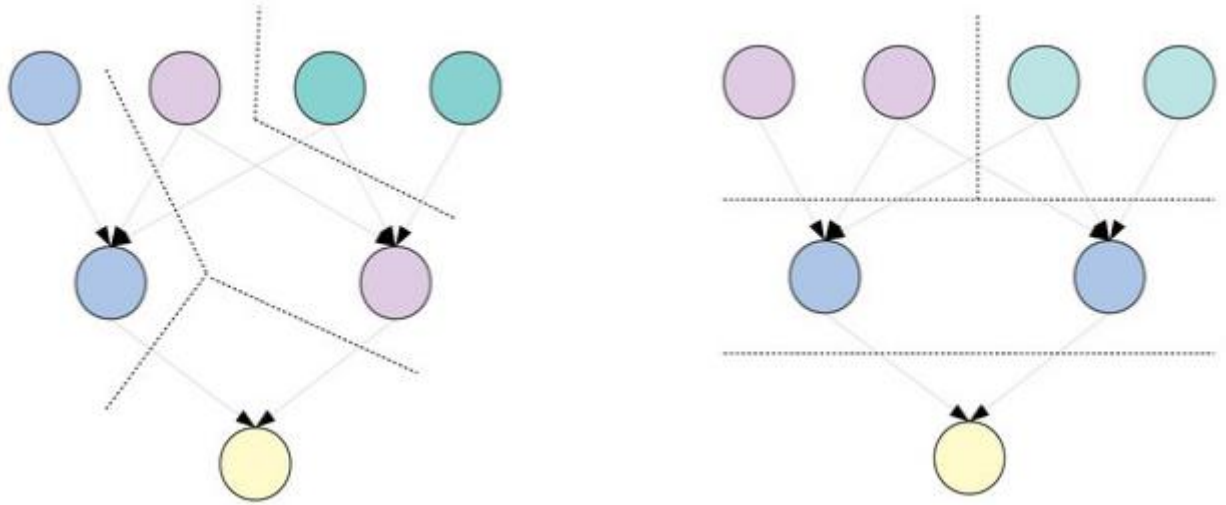
- 如何将问题的数据分解为多个能够独立操作的数据单元？
- 为了创建并行算法，关键不是要进行哪种分解，而是首先从哪种分解开始，任务分解和数据分解都要进行。
- 什么情况下该首先采用基于数据的分解方式？
  - 待解决问题中计算密集的部分是围绕着数据进行组织的；
  - 同样的操作应用到数据结构的不同部分。

数据分解

- 解决方法：
  - 如果已经基于任务进行了分解，则数据分解可由每个任务的需要所驱动。
  - 如果从数据分解开始并行算法设计，则应针对主要数据结构，考虑能否分解成多个可以并发操作的数据块。常见情况：
    - **线性数据结构**：可以采用“分段方式”对数据进行分解，例如数组；
    - **递归数据结构**：可以采用“递归方式”对数据进行分解，所谓“递归方式”就是指针对数据的一部分操作和针对整个数据的操作原理上是一样的。例如，将一个大型的树结构分解为多个能够并发更新的子树。
- 考虑因素：灵活性、效率、简单 — 数据粒度、依赖、访问竞争

任务分组和排序

- 基于任务间的约束分组任务
- 根据任务集合执行顺序上的约束来寻找并阐述任务分组间的依赖性

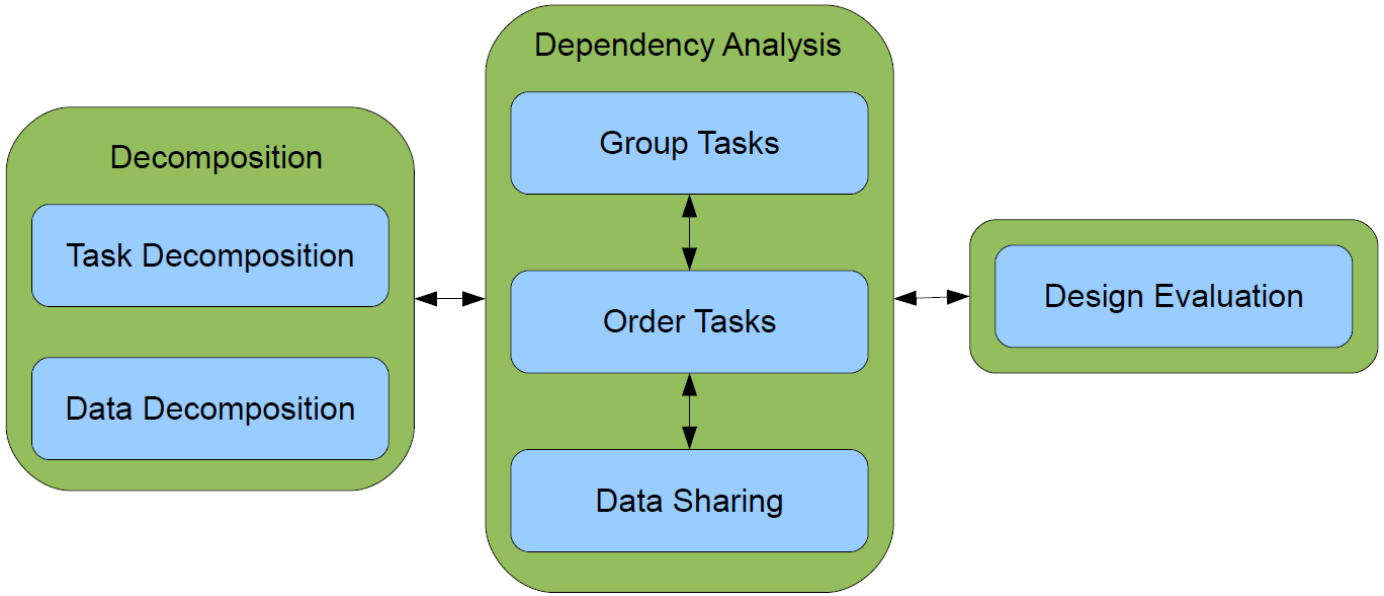


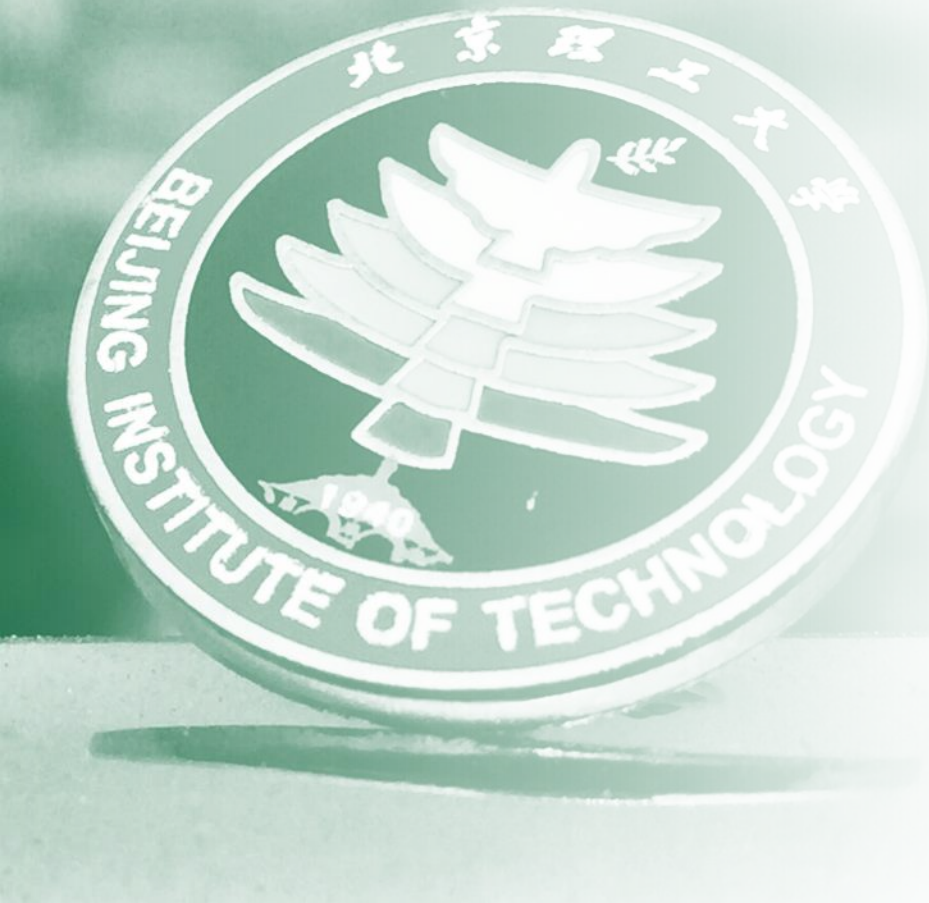
数据共享

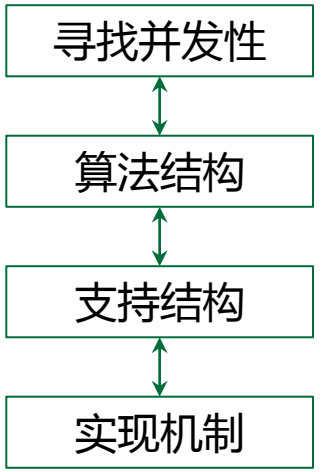
- 给定问题的数据分解和任务分解，如何在任务间共享数据？
- 考虑因素：
  - 正确性：处理不当可能导致数据竞争 — 加同步
  - 效率：为保证正确性可能导致过度的同步开销 — 复制数据；重叠通信和计算
- 首先要识别出任务间的共享数据：
  - 主要基于数据分解时，数据块的边缘可能被共享
  - 主要基于任务分解时，要确定数据如何传入传出任务，分析潜在数据共享源
- 其次，要分析如何使用这些数据：
  - 只读；可读可写；累加；多次读/单次写

“寻找并发性”设计空间

- 分解模式
  - 任务分解
  - 数据分解
- 依赖性分析模式
  - 任务分组
  - 任务排序
  - 数据共享
- 设计评价模式



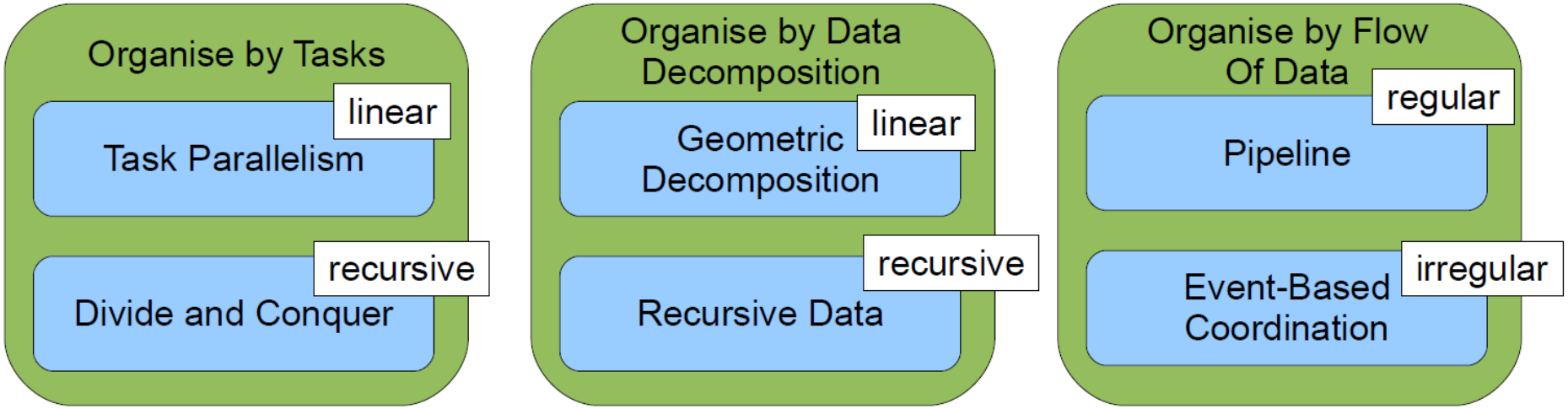




3 并行程序的算法结构

算法结构设计空间

- 为问题寻找一种或几种高效的算法结构模式
- 需要考虑：效率、简单、可移植性、可扩展性

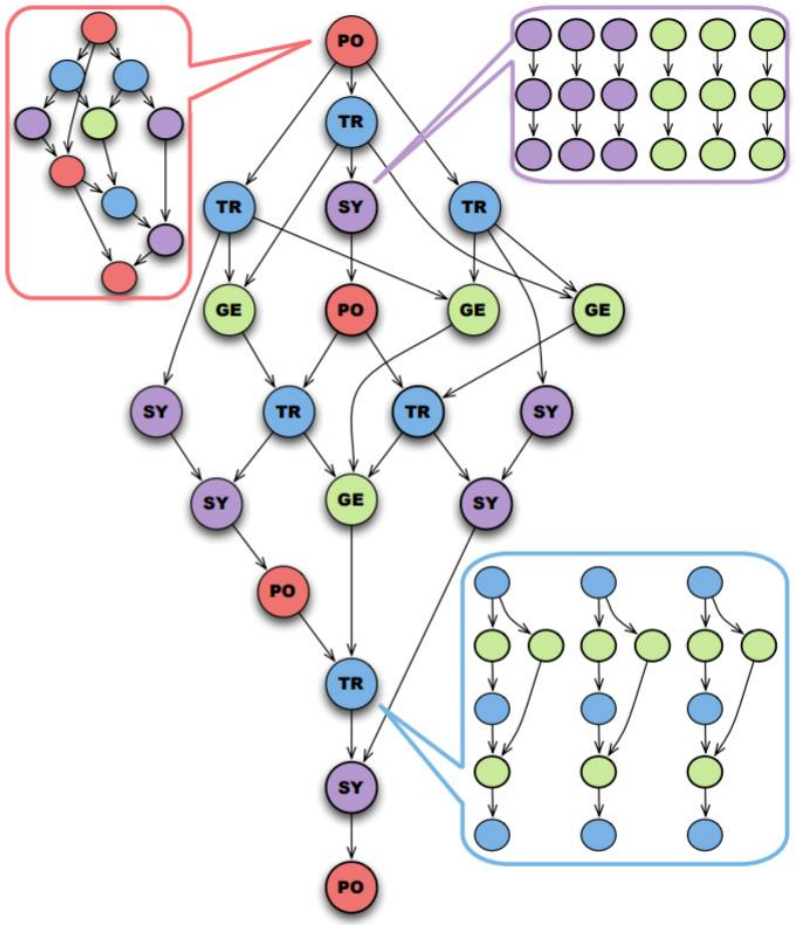


通过任务来组织

- 任务间相互独立
  - 大规模并行（massively parallel）
- 任务间有某些依赖关系（Dependencies）
  - 任务之间按照时间上的先后顺序形成一个DAG
  - 任务可以分解成子任务递归的解决，采用分治模式
- 任务创建和调度开销 vs. 任务本身的工作量
  - 设置阈值（thresholds）切换到串行程序

通过任务来组织 — 任务并行

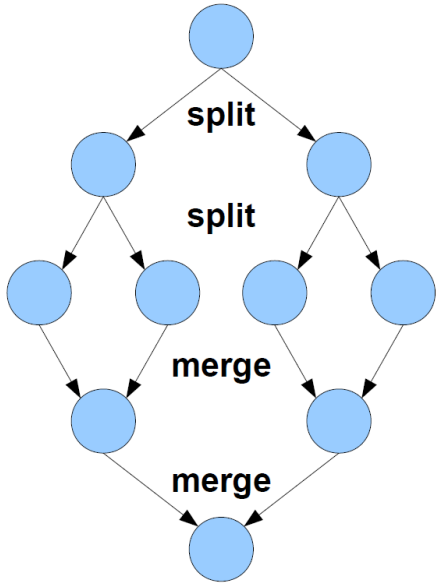
- 把任务作为并行的基本单位，主要涉及任务的划分、调度、数据分布、同步和通讯等问题
- 任务之间的关系可用有向无环图（DAG, Directed Acyclic Graph）表示

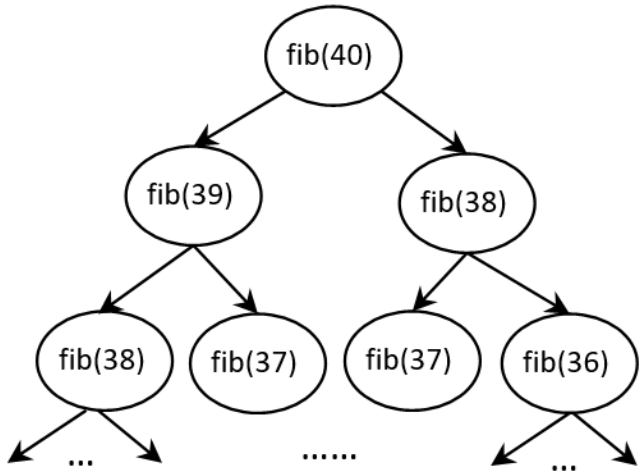


W. Wu, et al., "Hierarchical DAG Scheduling for Hybrid Distributed Systems," *2015 IPDPS*

通过任务来组织 — 分治（Divide and Conquer）

- 递归式的并行（Recursive Parallelism）
  - 递归深度
  - 深度优先或广度优先
- Cilk，TBB，MapReduce



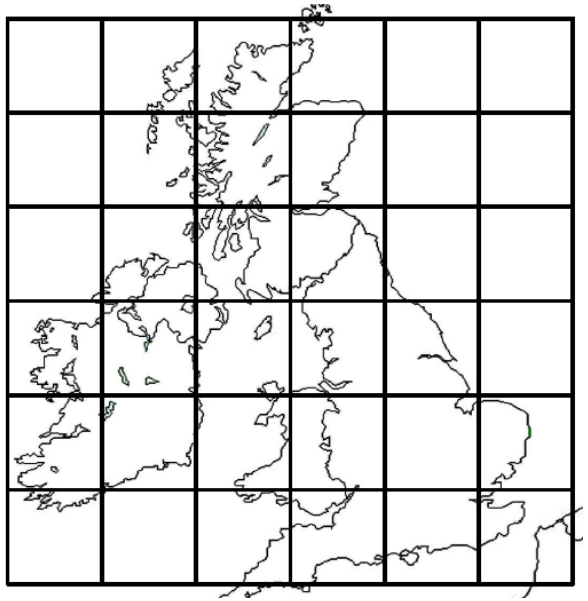


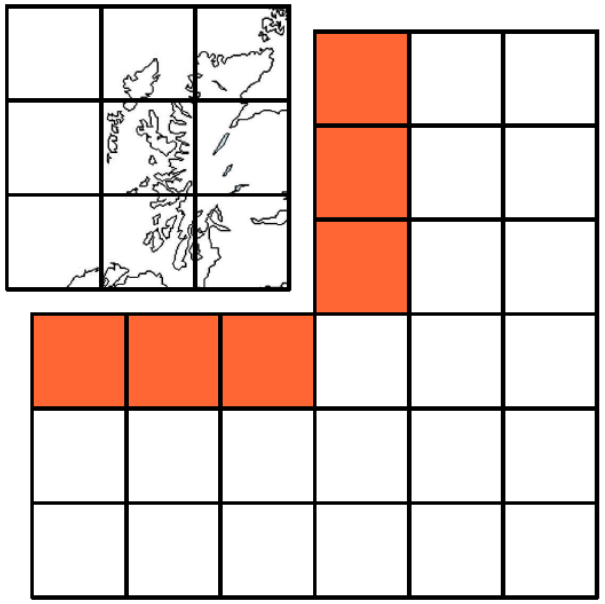
通过数据分解来组织 — 几何分解

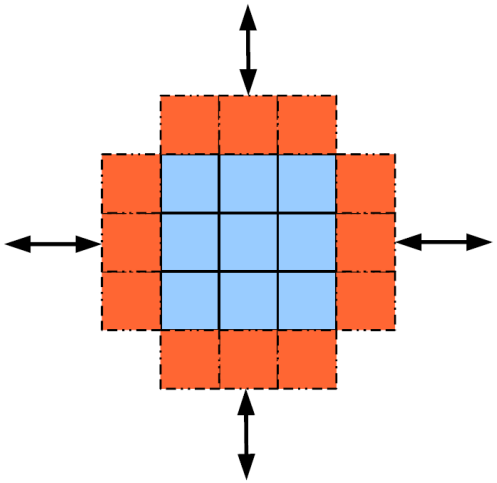
- 数据可以分解为多个可以同时更新的“块”（Block/Chunk）
- 有时数据块的更新需要其它块中的信息（通常是相邻块）
  - 例如：热扩散模拟
- 要考虑的关键问题：
  - 数据分解粒度：负载均衡~通信开销~调度开销
  - 是否通过冗余复制减少通信量
  - 数据交换和更新（通信和计算）是否能重叠
  - 数据分布和任务调度
  - 程序结构：通常是SPMD或循环并行

通过数据分解来组织 — 几何分解

- 网格计算程序
  - Halo exchange



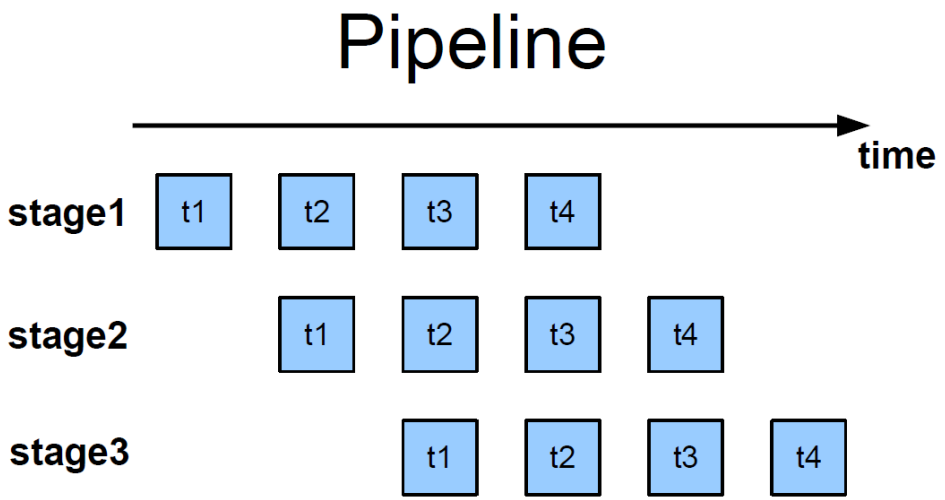




通过数据分解来组织 — 递归数据

- 问题涉及对可递归的数据结构（链表、树、图）的操作，考虑如何对这些数据结构并行的执行相关操作
  - 组合优化中，遍历树或图中所有节点
  - 计算链表元素的部分和（前缀扫描）
  - 寻找由树组成的森林中的树根
- 要考虑的关键问题：
  - 数据分解
  - 程序结构
  - 同步

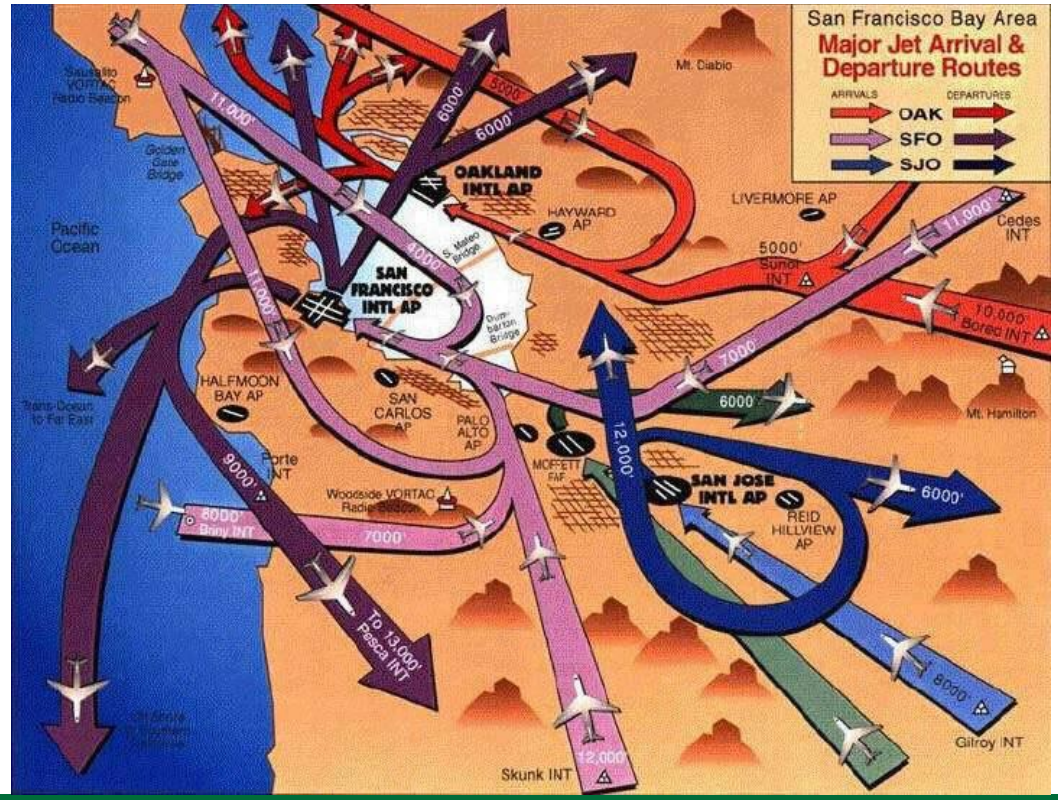
- 通过数据流来组织 — 流水线

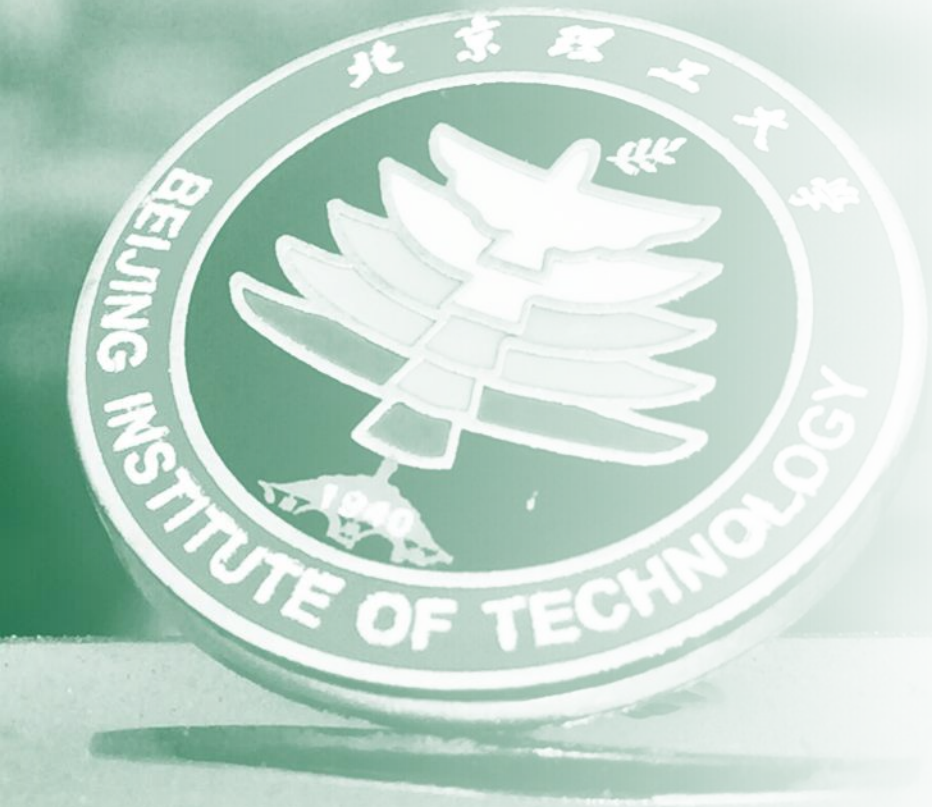


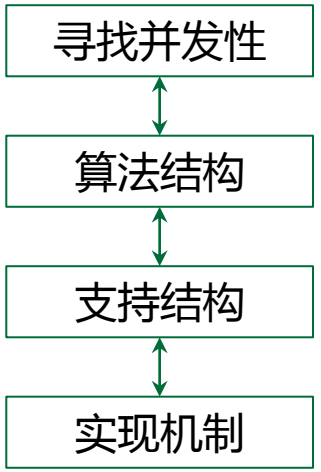
通过数据流来组织 — 基于事件的协作（Event-Based Coordination）

- 任务以非规范的模式交互，任务之间的数据流交互隐含了任务间的顺序约束
- 典型应用：离散事件仿真

并行离散事件仿真
(PDES, parallel discrete event simulation)

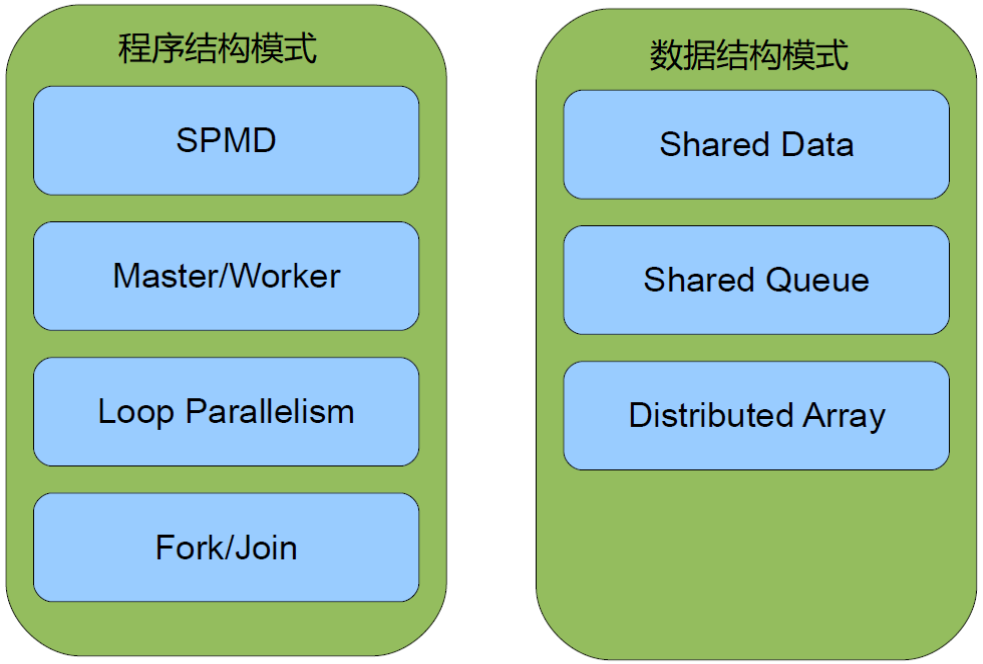






4 并行编程模式

- “支持结构”设计空间
  - 程序构造模式：描述源代码构造方法
  - 数据结构模式：管理数据依赖性



	任务并行	分治	几何分解	递归数据	流水线	基于事件的协作
SPMD	★★★★	★★★	★★★★	★★	★★★	★★
循环并行	★★★★	★★	★★★			
主/从模式	★★★★	★★	★	★	★	★
派生/聚合	★★	★★★★	★★		★★★★	★★★★

结构化串行编程模式

- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion
- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

结构化并行编程模式

- Superscalar sequence
- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Pack/expand
- Fork/join
- Pipeline
- Partition
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Priority scatter
- *Permutation scatter
- !Atomic scatter

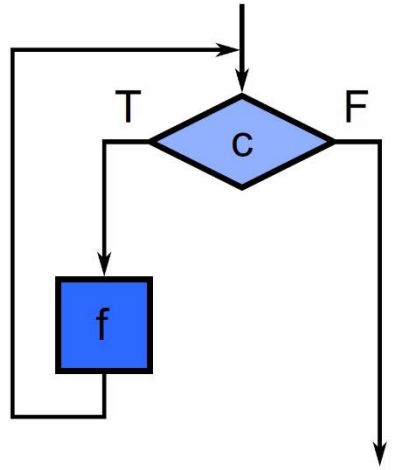
结构化编程 — 串行控制模式

- 顺序、选择、循环迭代、递归

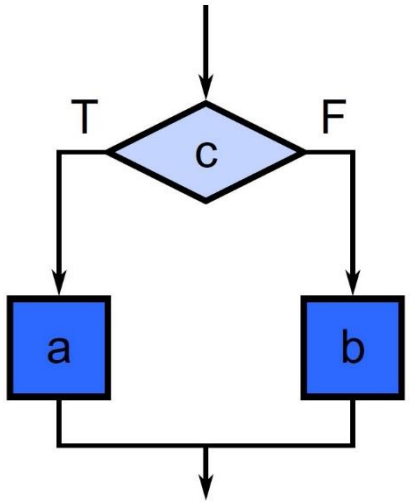
```
1  T = f(A);
2  S = g(T);
3  B = h(S);
```


```
1  for (i = 0; i < n;
2    a;
3  }
```

```
1  while (c) {
2    a;
3  }
```

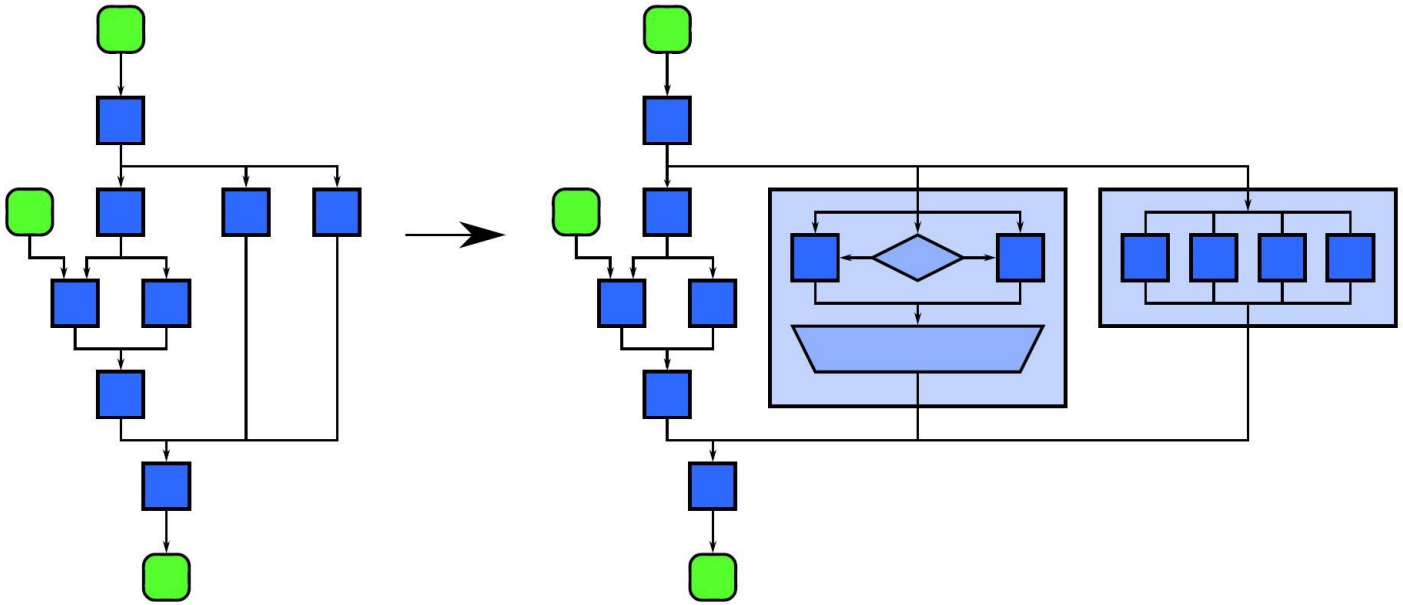



```
1  if (c) {
2    a;
3  } else {
4    b;
5  }
```



嵌套模式（Nesting Pattern）

- 嵌套是分层组合模式的能力
- 嵌套可应用于串行和并行模式中

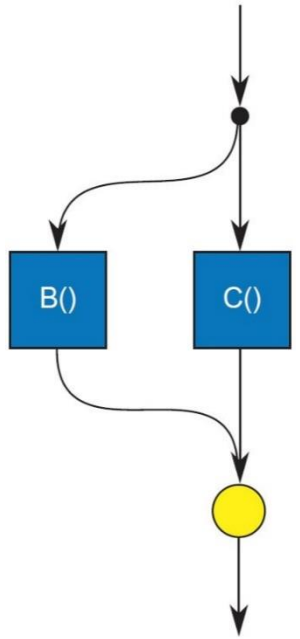


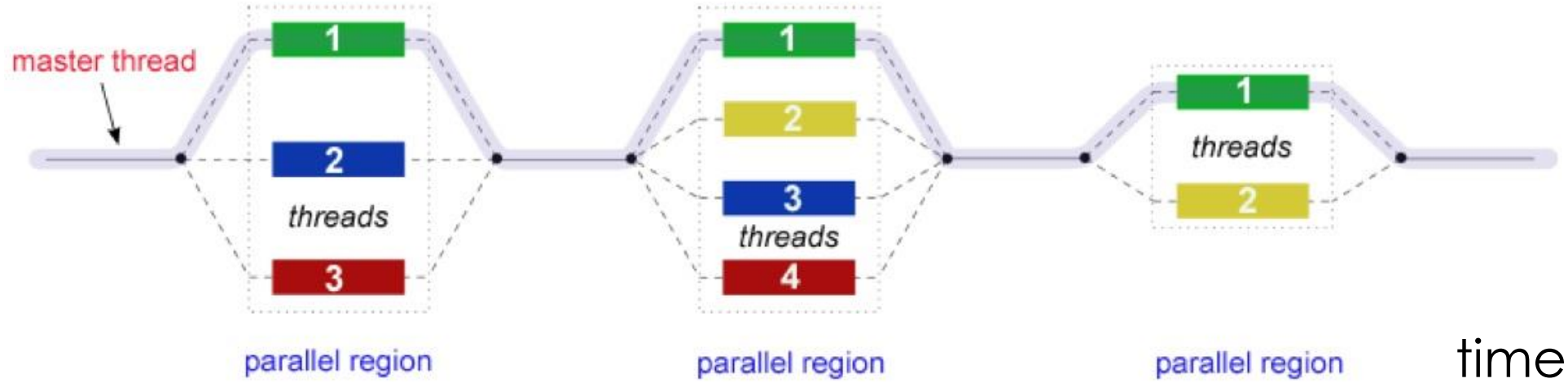
结构化编程 — 并行控制模式

- 并行控制模式扩展了串行控制模式
- 每个并行控制模式都与至少一个串行控制模式有关，但是放宽了串行控制模式的假设
- 并行控制模式：fork-join，map，reduction，stencil，pipeline，scan，recurrence

结构化并行编程模式 — fork-join

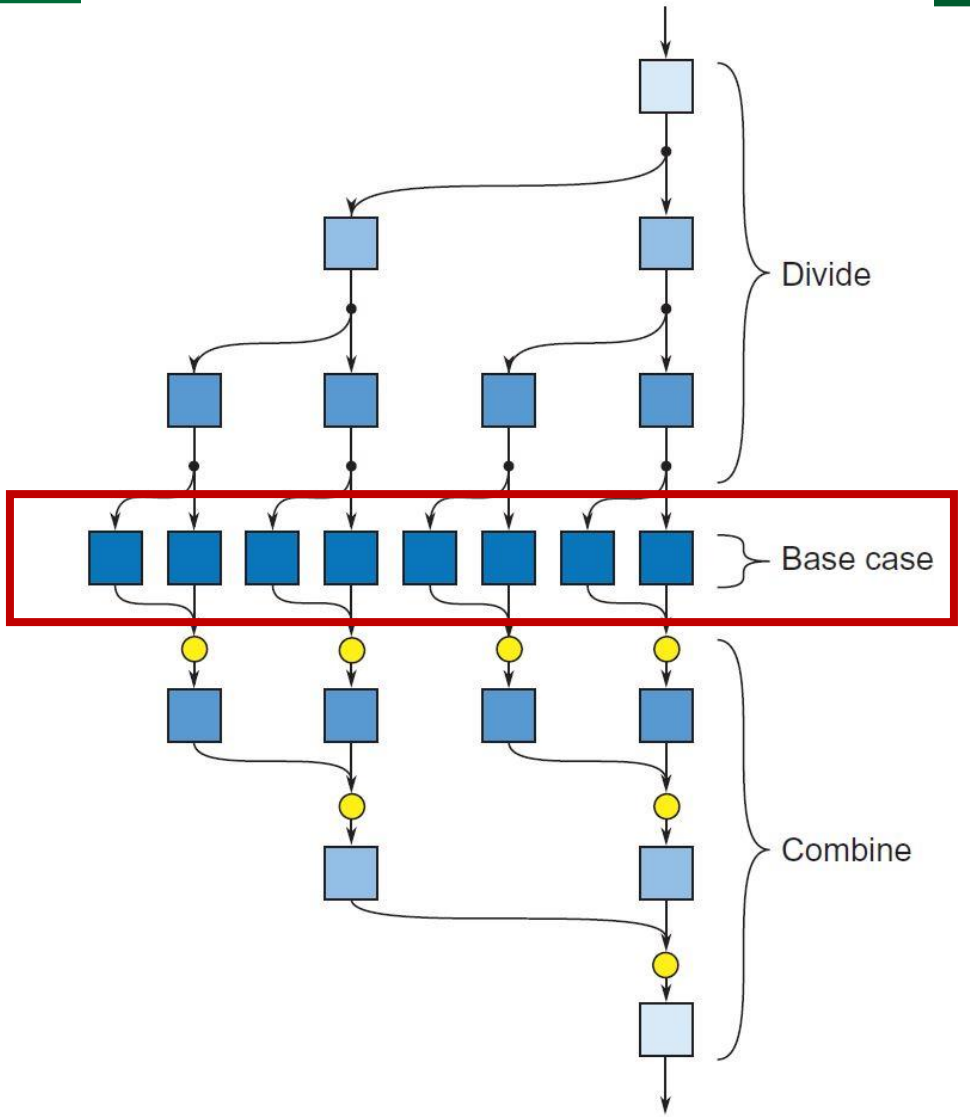
- 控制流分成（fork）多个并行的指令流，之后再合到一起（join）
- 在某些编程模型中也称为：spawn-sync





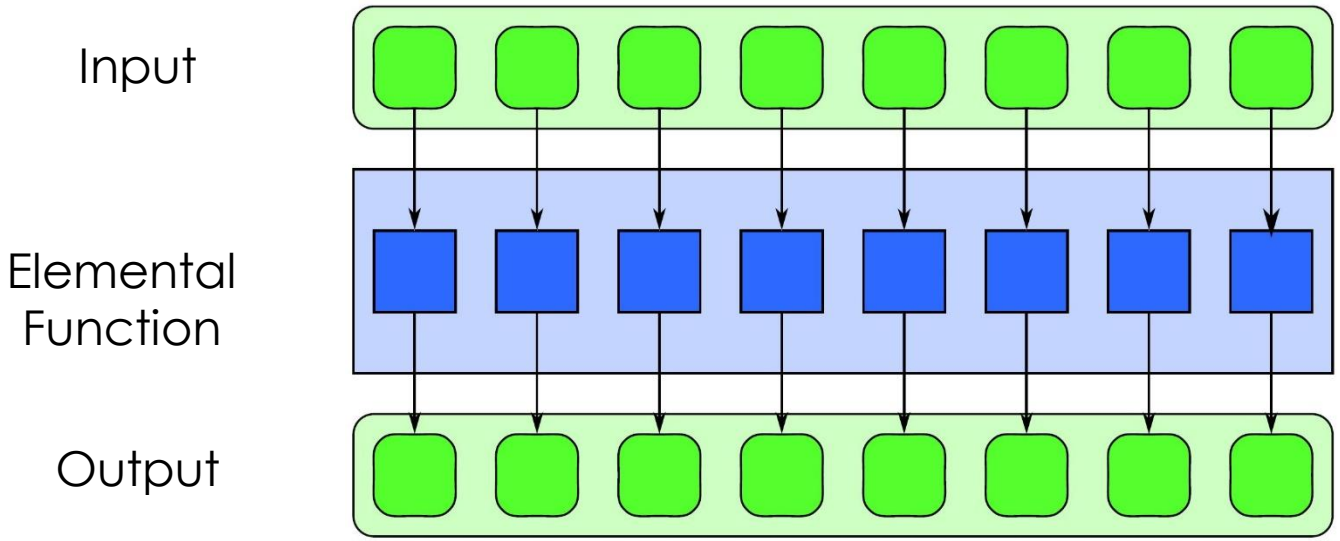
结构化并行编程模式 — fork-join

- Fork-join 模式也常用于分治算法（Divide-Conquer）
- 需要考虑哪些问题？



结构化并行编程模式 — map

- Map：对集合中的每个元素执行相同的操作

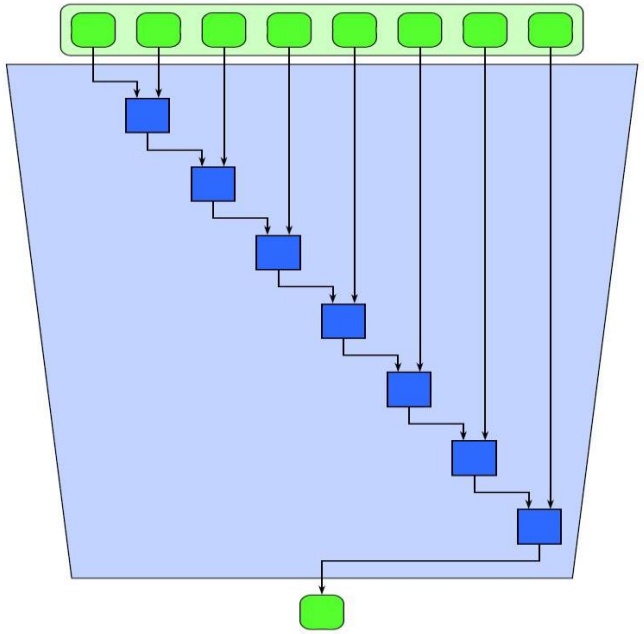



```
for (i=0; i<n; ++i) {
    f(A[i]);
}
```

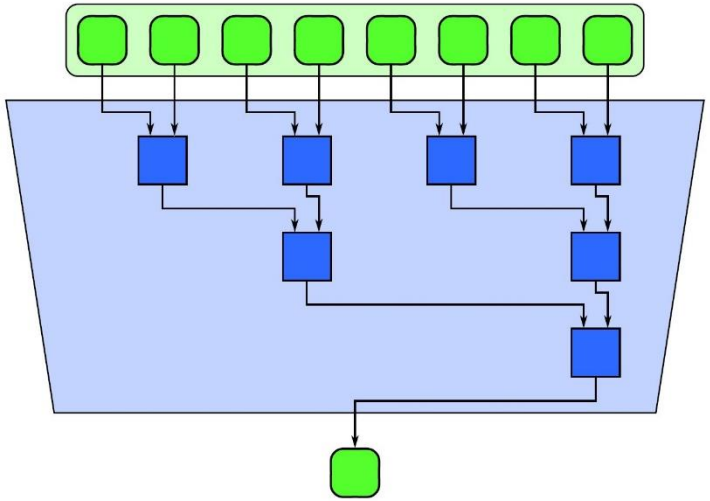
结构化并行编程模式 — reduction

- Reduction：使用某种“规约函数”合并集合中的元素
- 规约函数示例：加法，乘法，最大值，最小值, 布尔值AND，OR, XOR

Serial Reduction



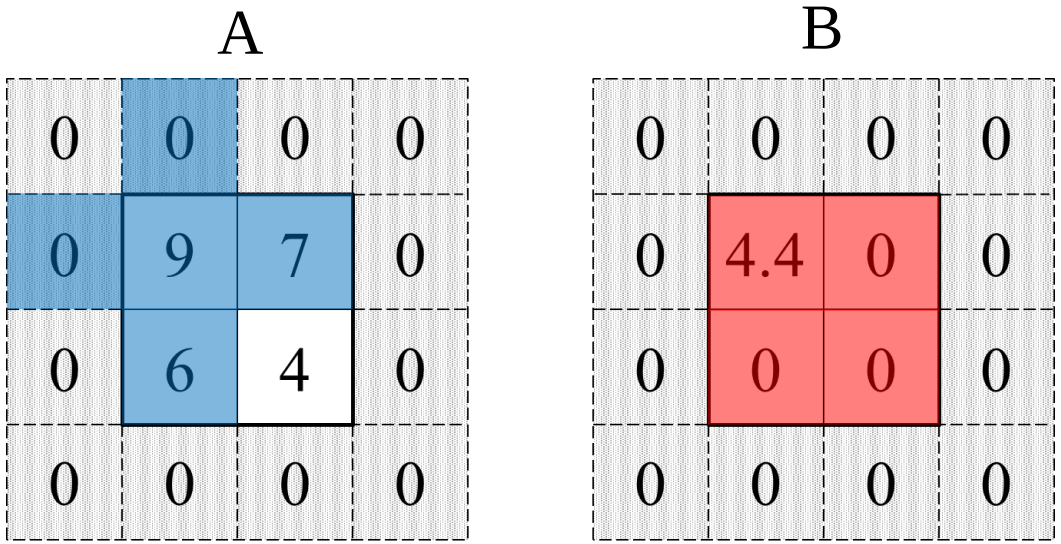
Parallel Reduction



```
b = 0;
for(i=0;i<n;++i){
    b += f(B[i]);
}
```

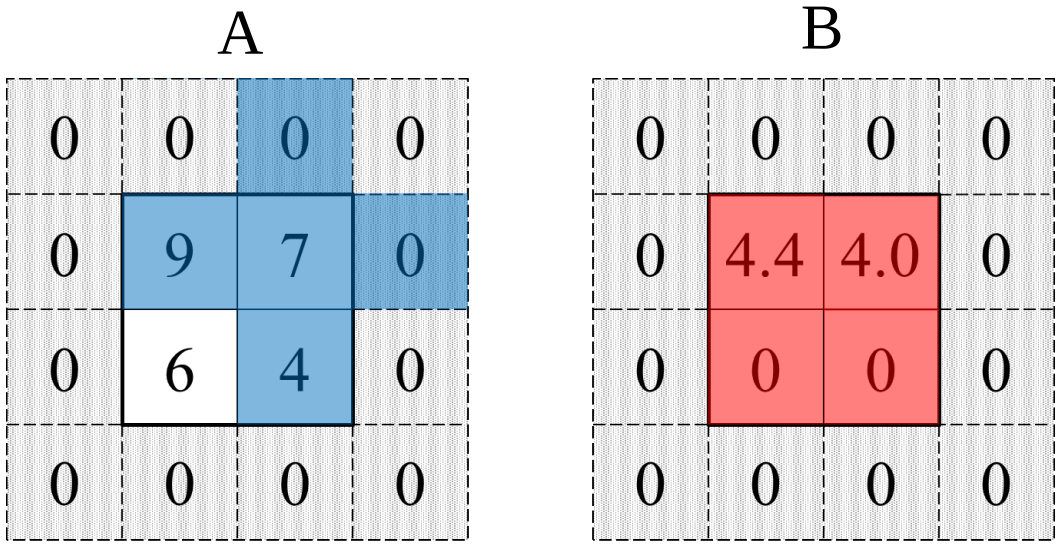
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map
- 用偏移窗口实现蒙版操作
- 需要考虑边界条件，但大部分操作是在内部进行的



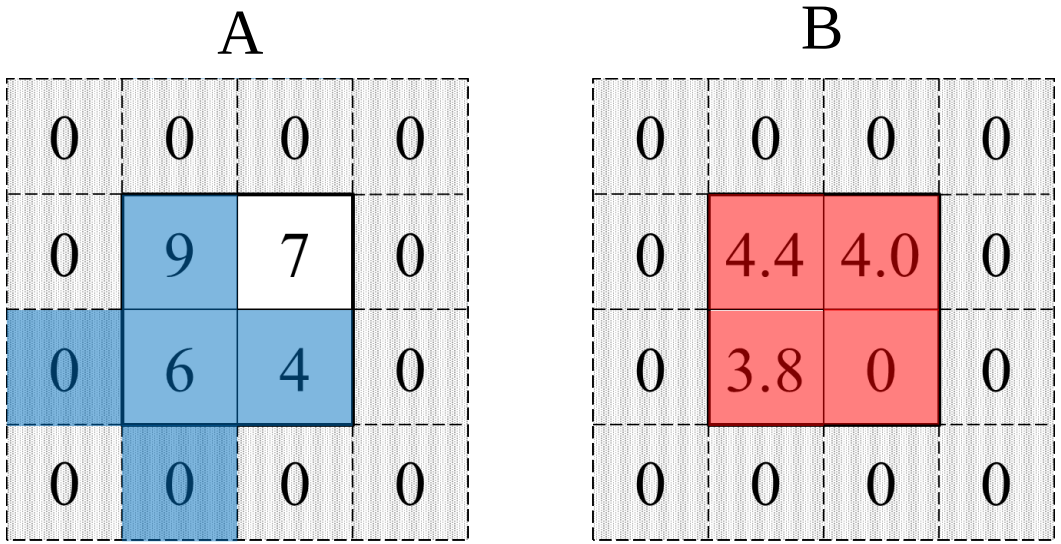
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map
- 用偏移窗口实现蒙版操作
- 需要考虑边界条件，但大部分操作是在内部进行的



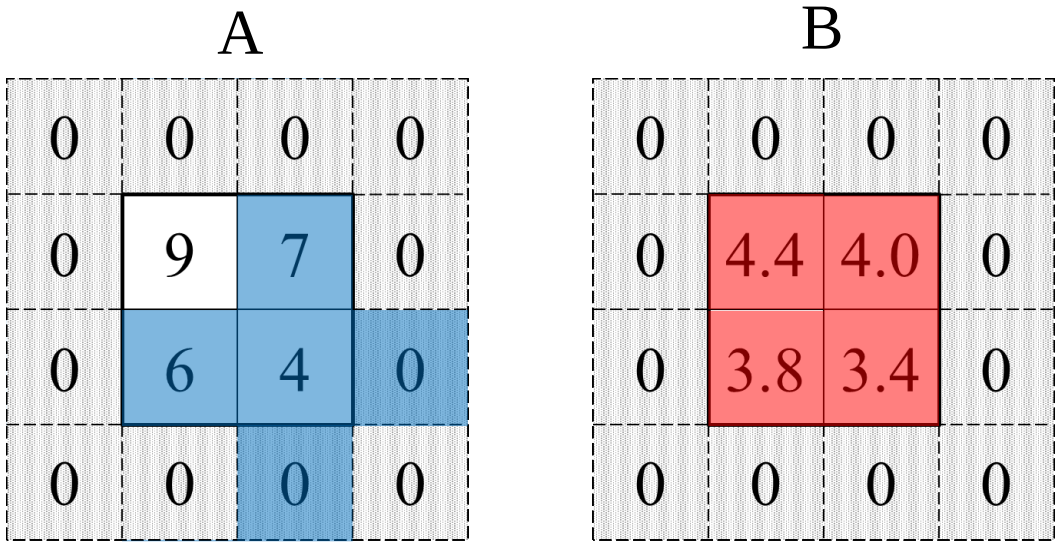
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map
- 用偏移窗口实现蒙版操作
- 需要考虑边界条件，但大部分操作是在内部进行的



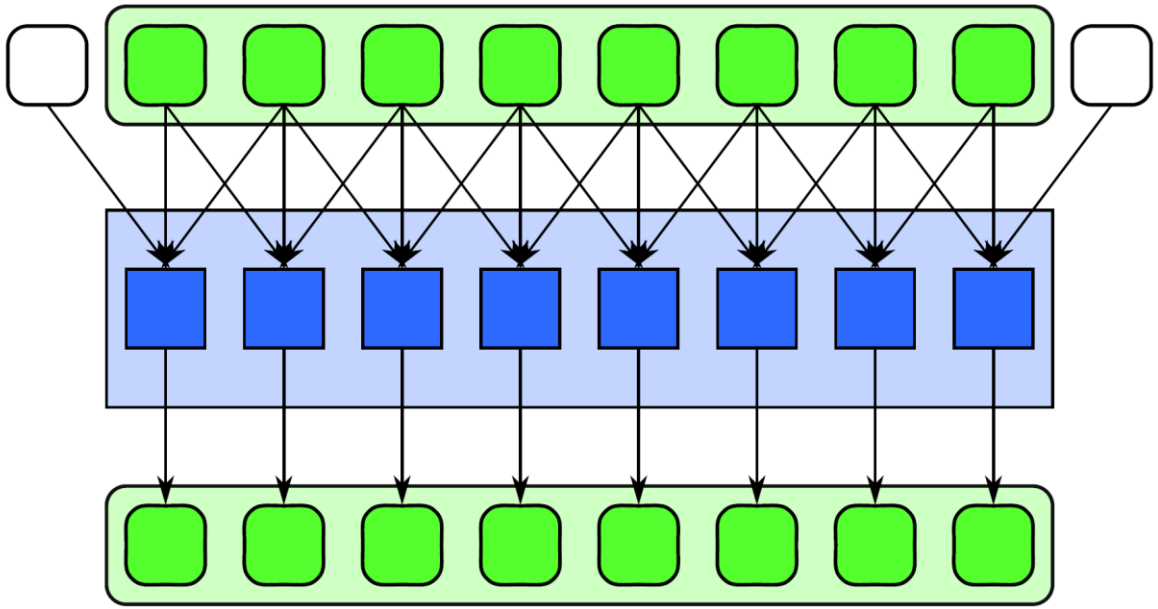
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map
- 用偏移窗口实现蒙版操作
- 需要考虑边界条件，但大部分操作是在内部进行的



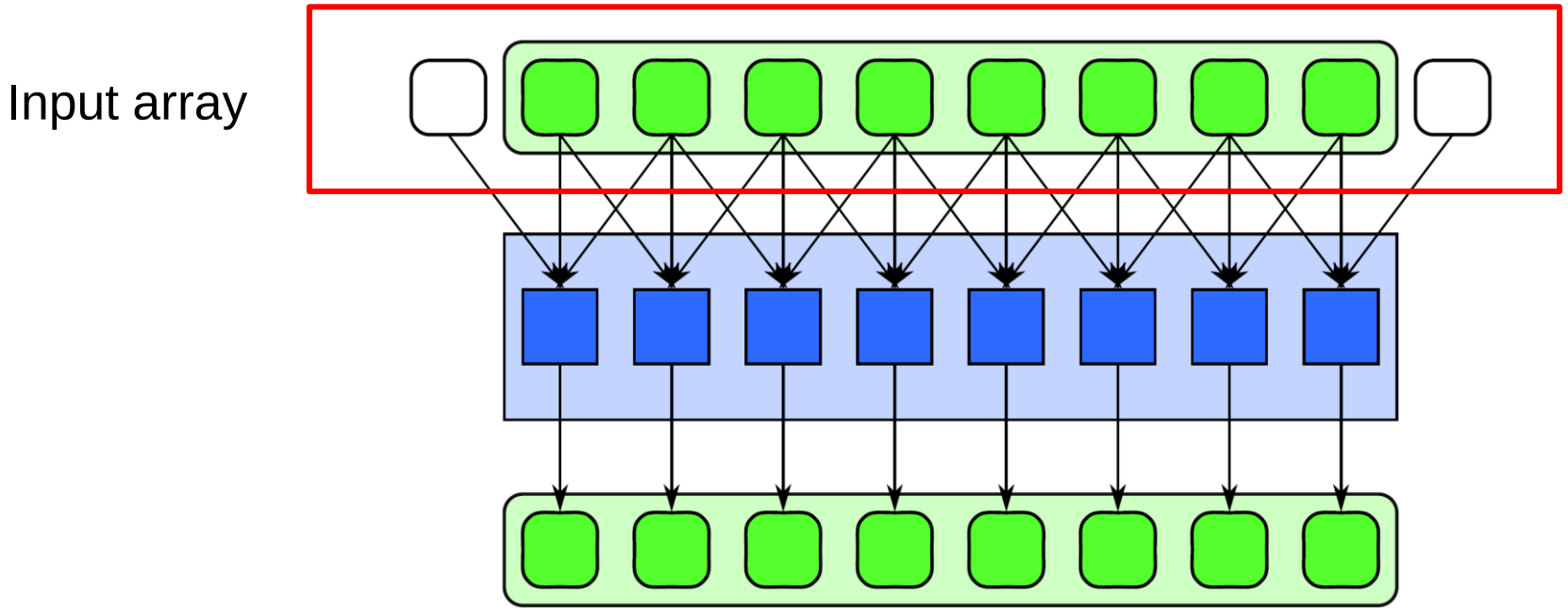
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map



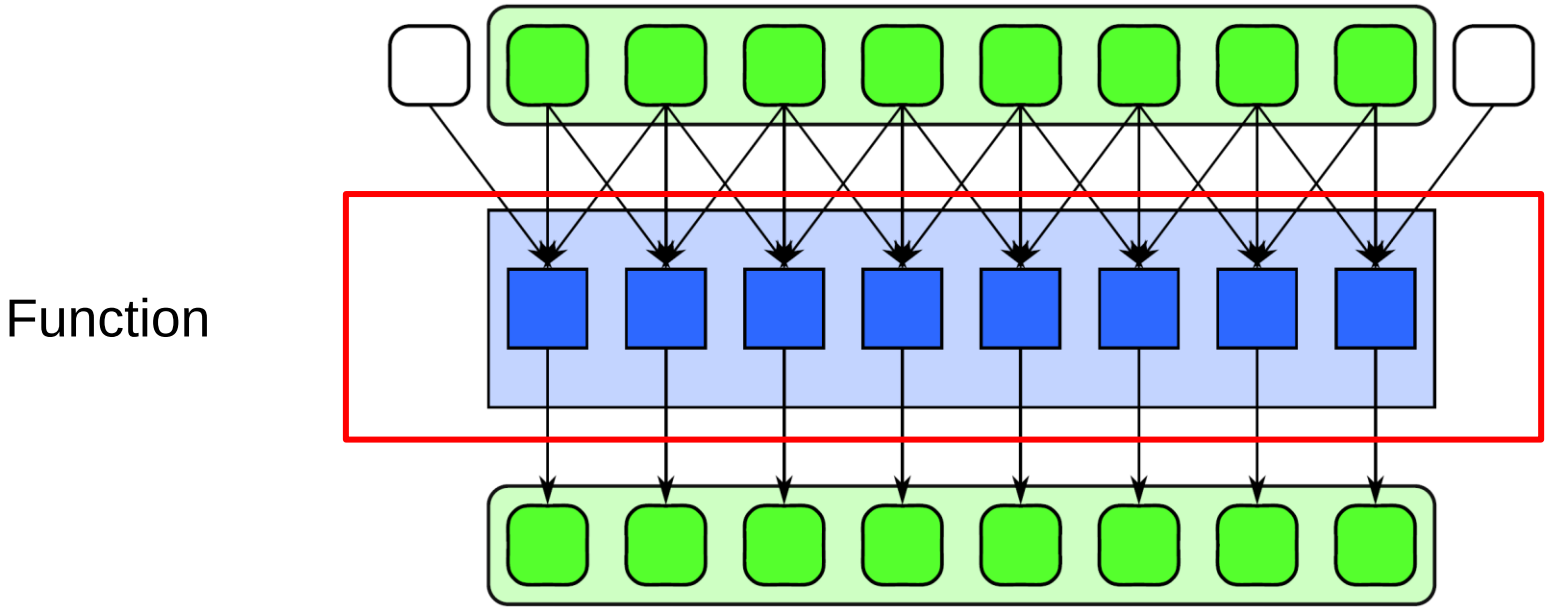
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map



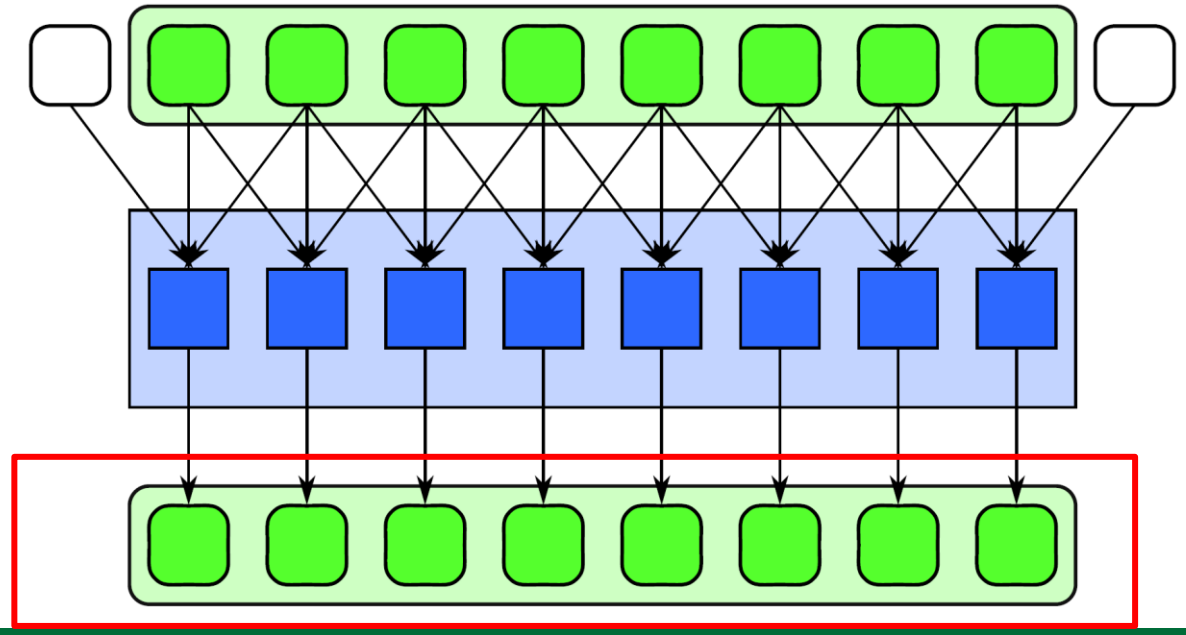
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map



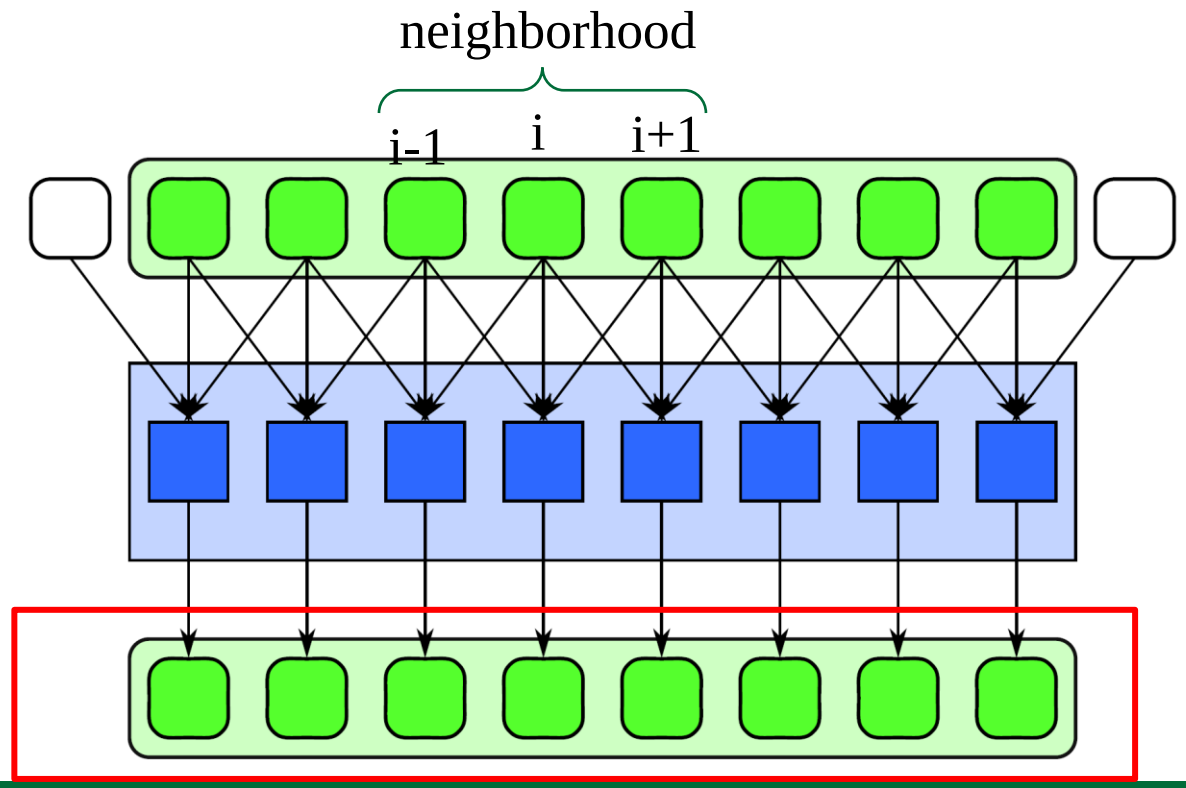
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map



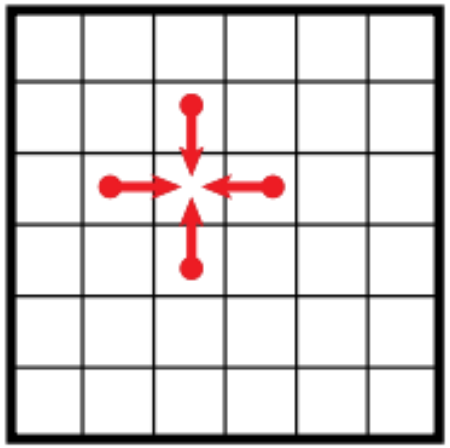
结构化并行编程模式 — Stencil

- Stencil：对相邻的一些数据执行某个特定的函数，可以看做一种Map



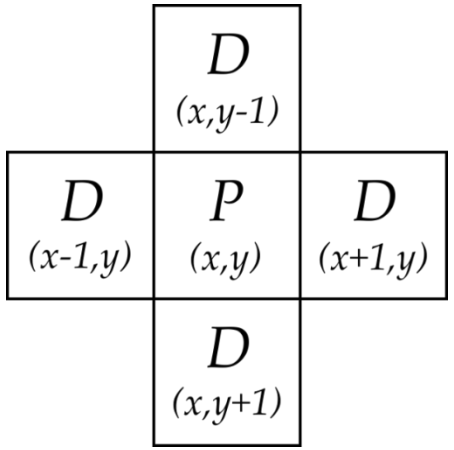
结构化并行编程模式 — Stencil

- 2-Dimensional Stencils



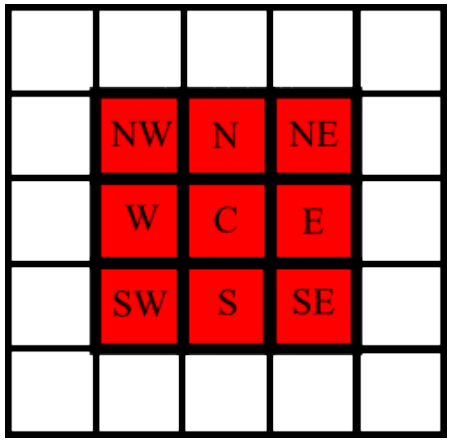
4-point stencil

Center cell (P)
is not used



5-point stencil

Center cell (P)
is used as well

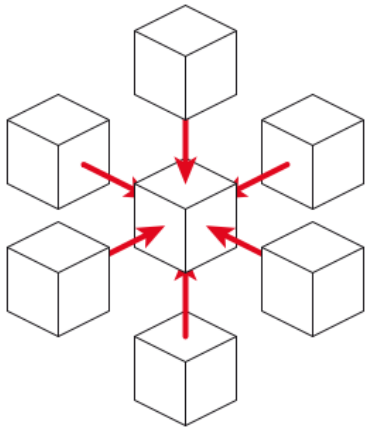


9-point stencil

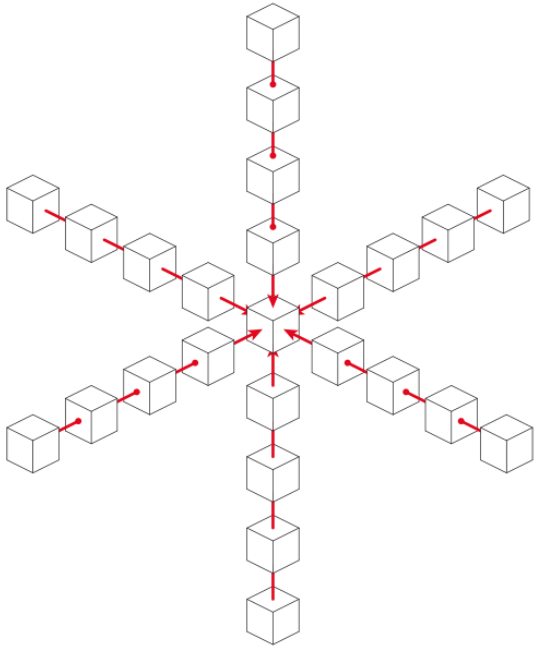
Center cell (C)
is used as well

结构化并行编程模式 — Stencil

- 3-Dimensional Stencils



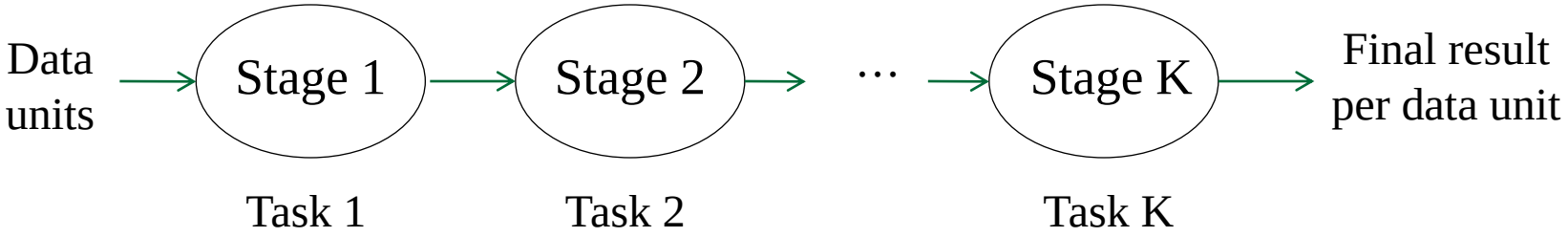
6-point stencil
(7-point stencil)

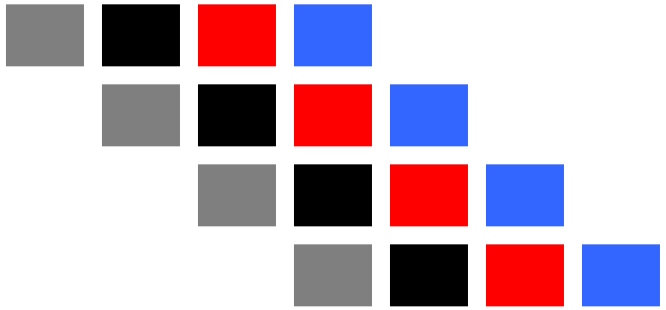


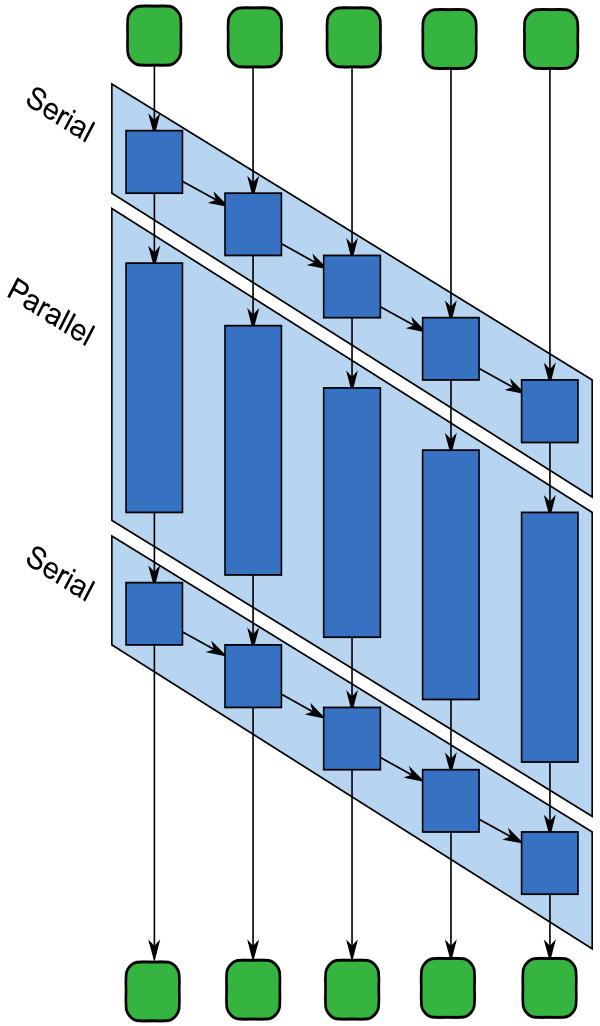
24-point stencil
(25-point stencil)

结构化并行编程模式 — pipeline

- 和指令流水线类似，只不过这里的每个流水阶段执行一个任务



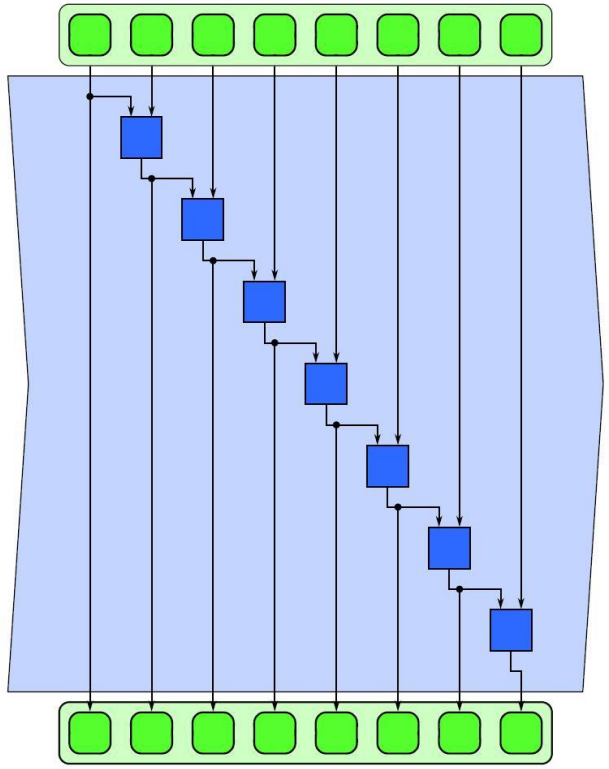




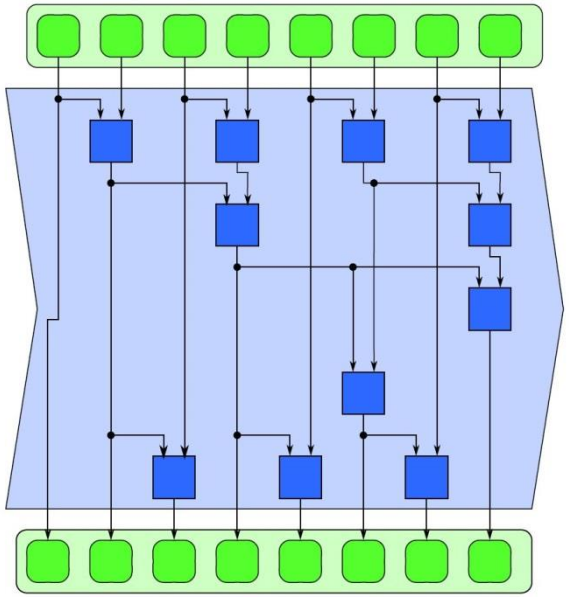
结构化并行编程模式 — scan

- Scan：对输入序列计算所有局部规约，产生新的输出序列

Serial Scan

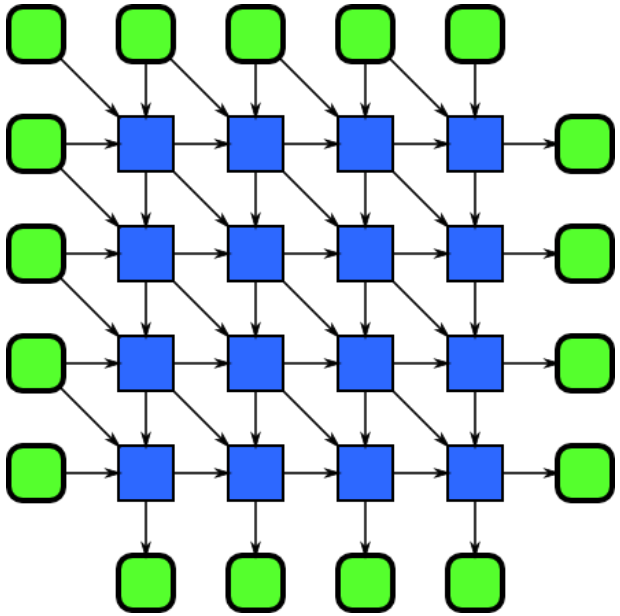


Parallel Scan



结构化并行编程模式 — recurrence

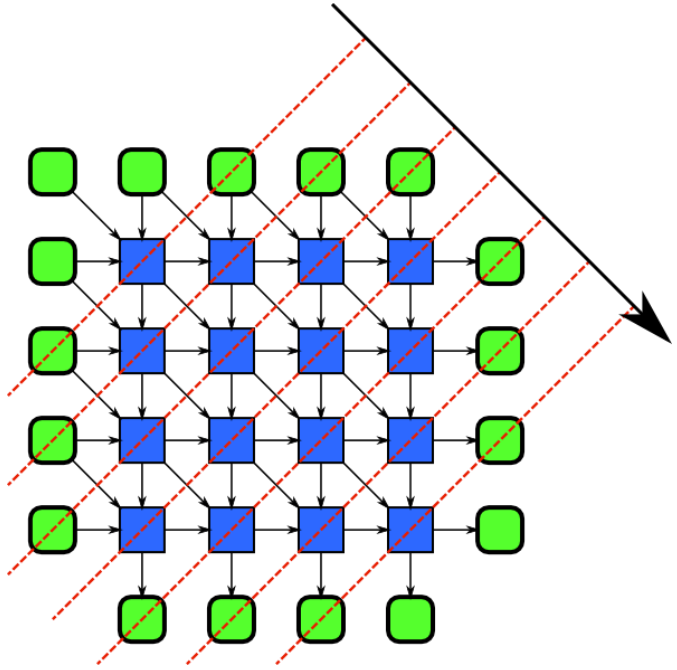
- Recurrence：循环嵌套，有数据依赖，可递推并行



```
for (int i = 1; i < N; i++) {
  for (int j = 1; j < M; j++) {
    A[i][j] = f(
      A[i-1][j],
      A[i][j-1],
      A[i-1][j-1],
      B[i][j]);
  }
}
```

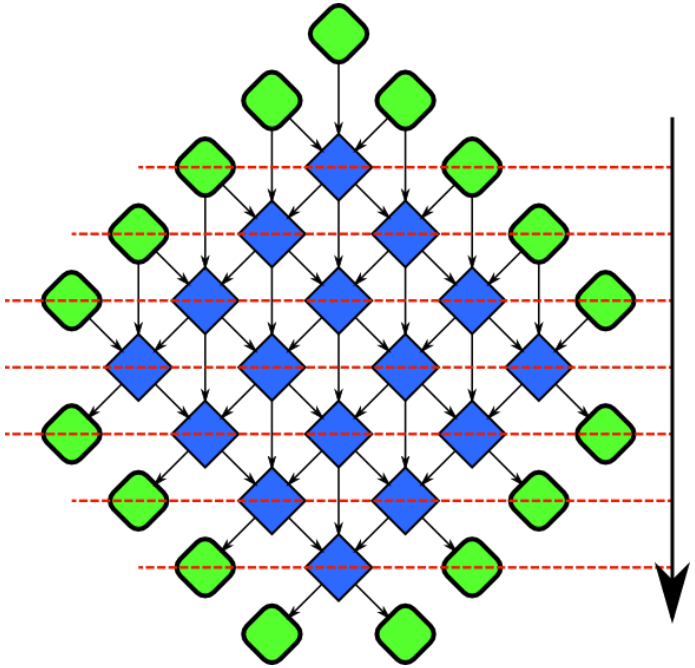
结构化并行编程模式 — recurrence

- Recurrence：循环嵌套，有数据依赖，可递推并行



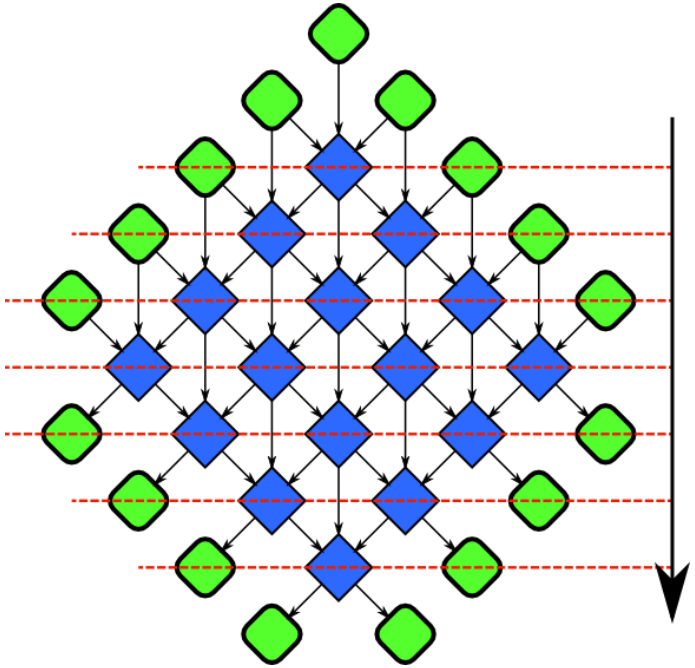
结构化并行编程模式 — recurrence

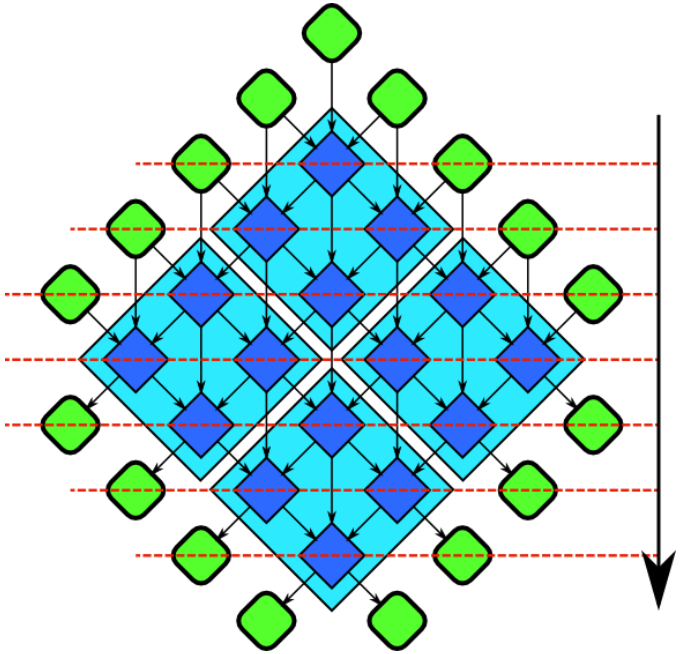
- Recurrence：循环嵌套，有数据依赖，可递推并行



结构化并行编程模式 — recurrence

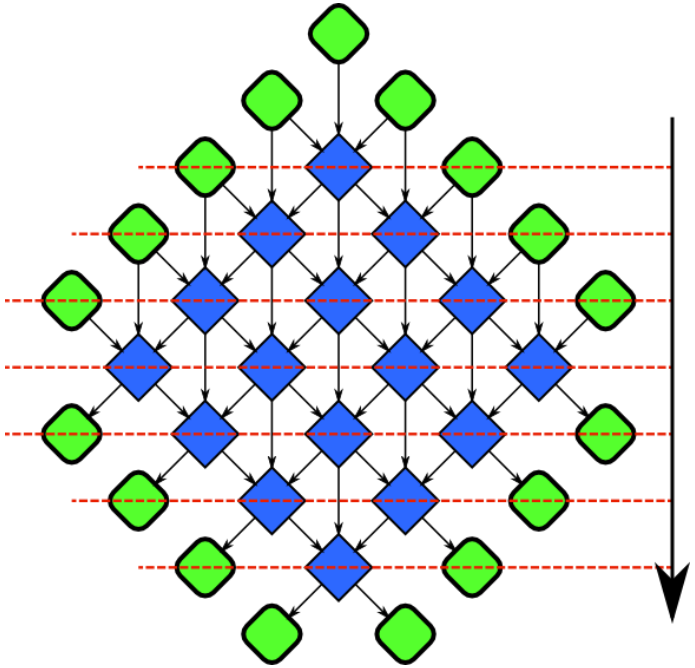
- Recurrence：循环嵌套，有数据依赖，可递推并行

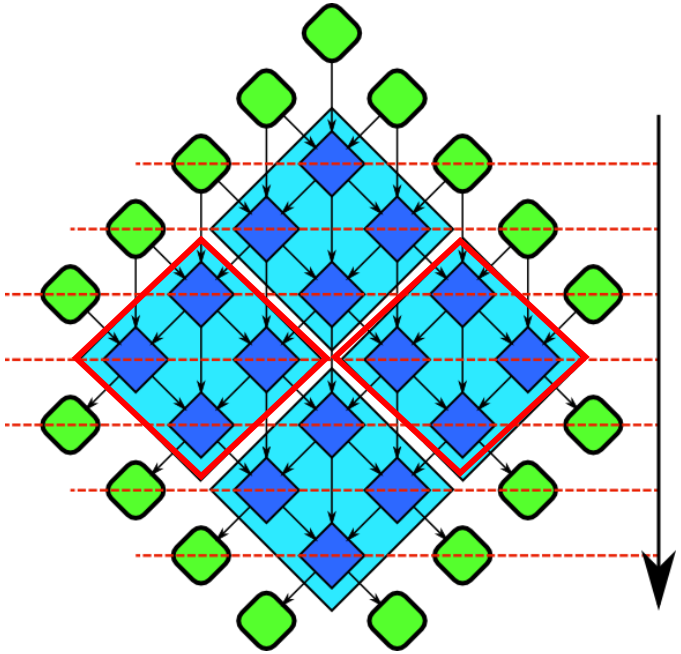




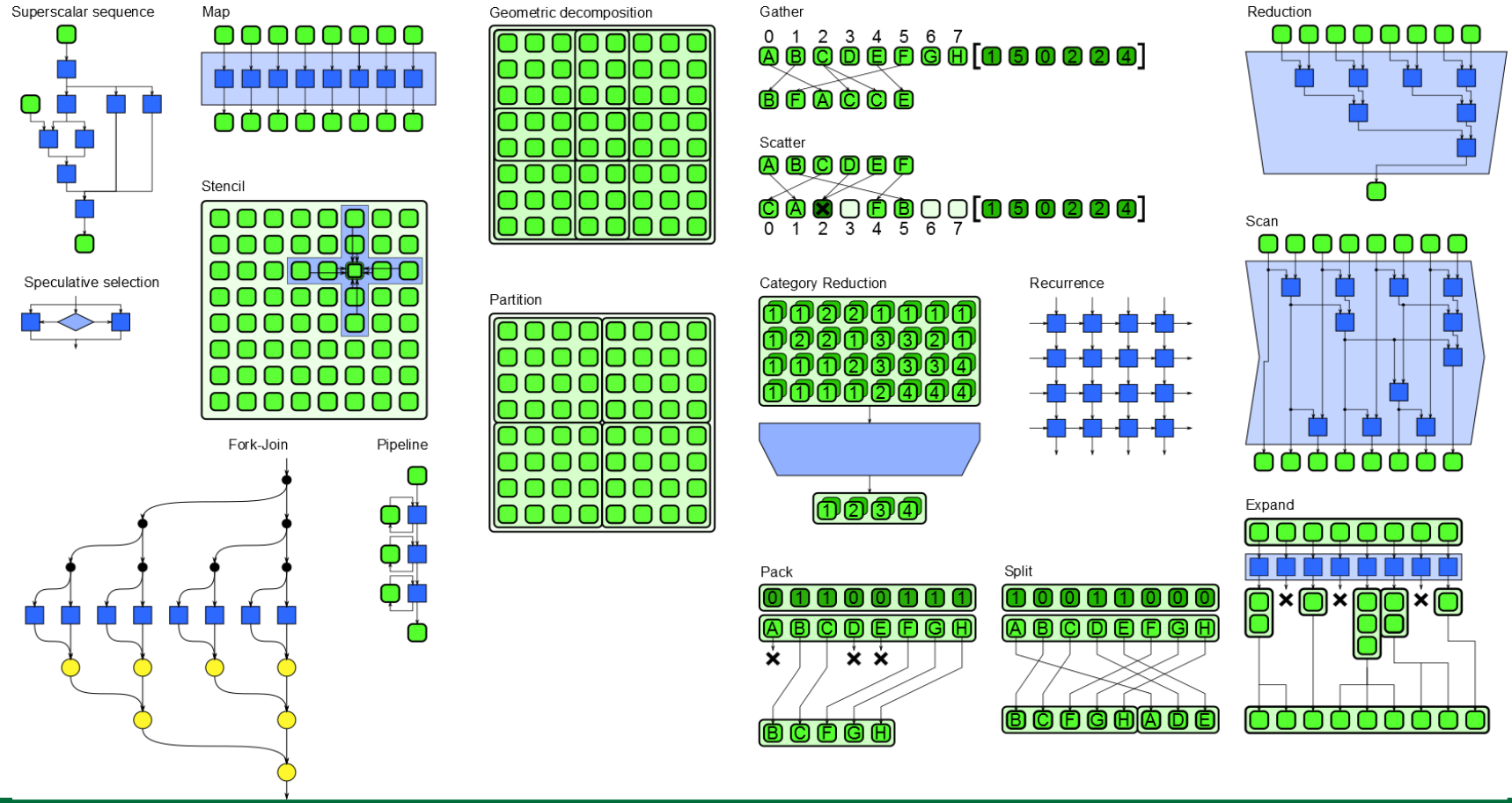
结构化并行编程模式 — recurrence

- Recurrence：循环嵌套，有数据依赖，可递推并行





结构化并行编程模式 — Overview




北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工