第3讲并行软件及其程序设计

张永东

内容

- 并行软件Parallel software
- 输入和输出Input and output
- 性能Performance
- 并行程序设计Parallel program design
- 编写和运行并行程序Writing and running parallel programs
- 假设Assumptions

并行程序设计

目录

- 并行程序设计
- 编辑运行
- 输入输出

并行程序设计

- 没有固定统一的并行程序设计步骤和要求。
- 可参考的设计方法：
  - 福斯特方法(Foster’s methodology)
  - 马森吉尔方法(Massingill’s methodology)
  - 卡勒方法(Culler’s methodology)

几个概念的定义

- 任务
  - 要完成的一个工作(任意内容和大小)。一个任务只能由一个处理器执行，并发性只能在任务之间
- 进程/线程
  - 一个完成任务的抽象实体。
- 处理器
  - 计算的物理资源。
- 进程/线程与处理器区别
  - 进程/线程是将多处理器抽象的一种方便的形式。通过进程/线程，而不是物理处理器来写并行性程序，然后将进程/线程映射到处理器。
  - 进程/线程数和处理器数可以不等。

Massingill's methodology

- 给出了一个并行程序创建的4个阶段：
  - 寻找并发性。
    - 识别可用的并发性，并用于算法设计中。
  - 算法结构设计。
    - 用一种高级结构组织一个并行算法。
  - 支持结构。
    - 将算法转化为源代码，考虑如何组织并行程序以及如何管理共享数据。
  - 实现机制。
    - 寻找特定的软件构造，实现并行程序。
- 4个阶段构成了4个设计空间。从“寻找并发性”开始，到达“实现机制”时，就可以得到并行程序的详细设计。

Culler's methodology

- 将程序并行化过程分为如下4个步骤：
  - 任务分解
    - 将计算分解成任务
  - 任务分配
    - 将任务分配给进程/线程
  - 进程/线程协作
    - 在进程/线程之间协调必要的数据访问、通信和同步
  - 任务映射
    - 将进程/线程映射或绑定到处理器

Foster's methodology

- Foster也提出了一个4步的并行算法设计过程：
  - 划分（Partitioning）：
    - 大任务划分为小任务，使小任务可以并行执行。
  - 通信（Communication）：
    - 确定划分得到的小任务需要的通信。
  - 集聚（Agglomeration or aggregation）：
    - 根据小任务间依赖关系和交互关系密切，就把它们合并为一个任务。
  - 映射/分配（Mapping）：
    - 把小任务分配(/映射)到不同的进程中，使得进程通信量最小且负载均衡。

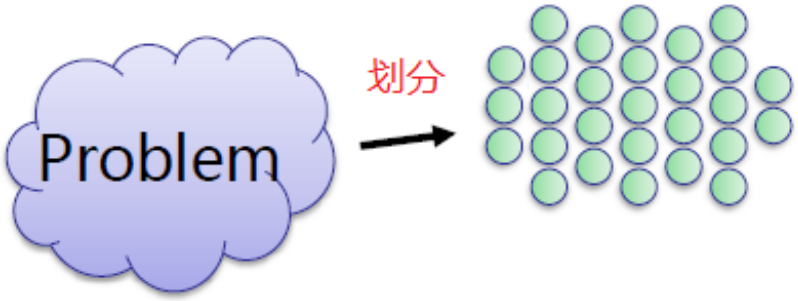
并行程序设计

- 福斯特方法Foster's methodology

通常是规模比较大的问题
有一定可并行性

并行程序设计

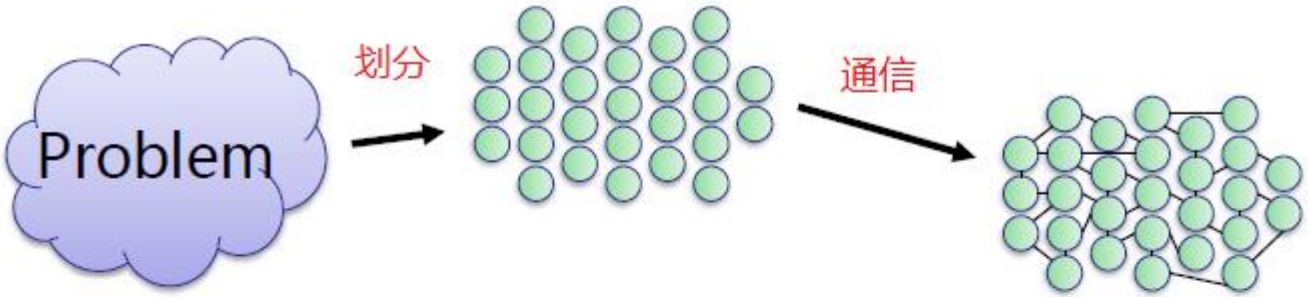
- 福斯特方法Foster's methodology



寻找并行性：
尽可能分得细
小任务间可并行

并行程序设计

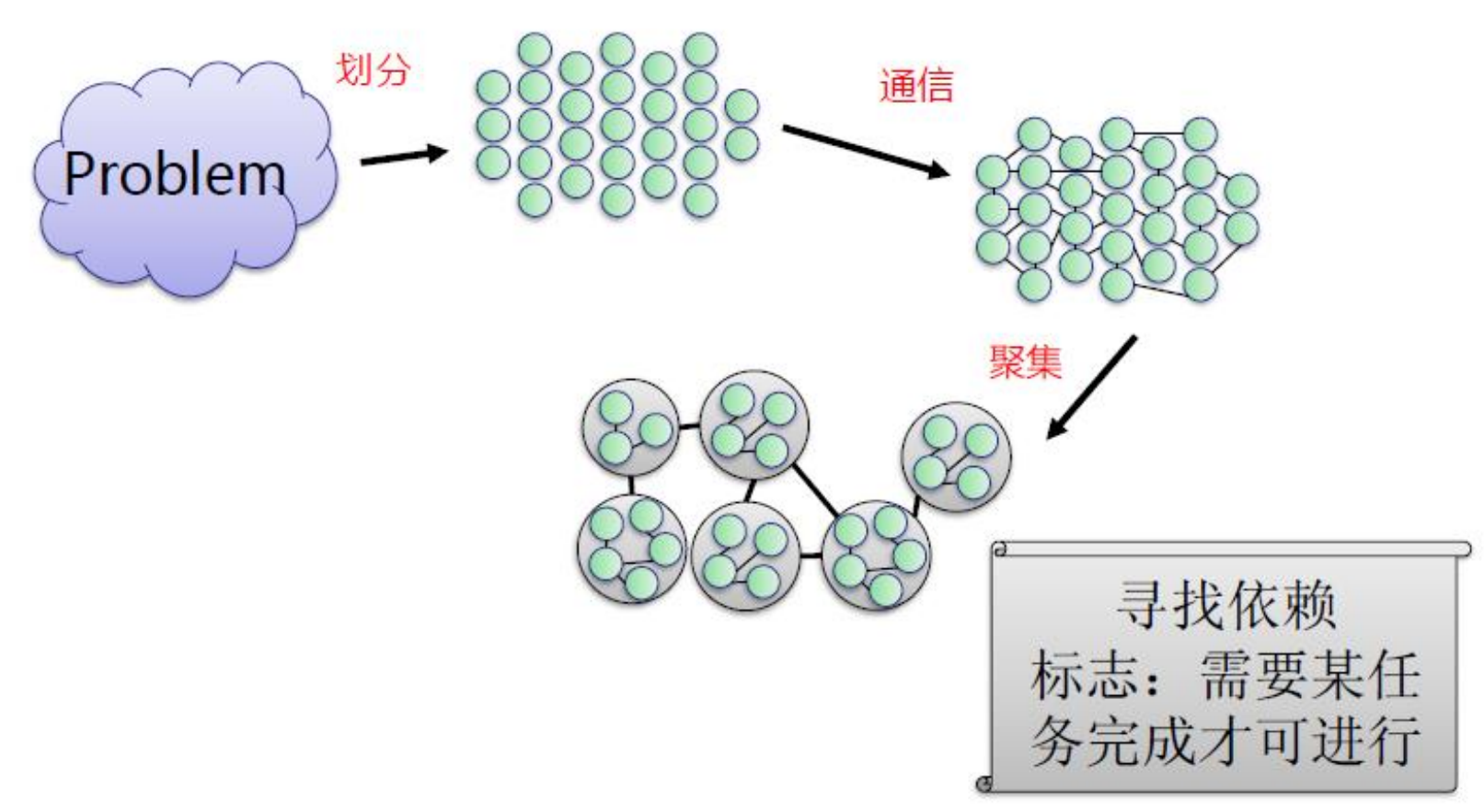
- 福斯特方法Foster's methodology



寻找要发生的通信
标志：需要某任务的数据
才可进行

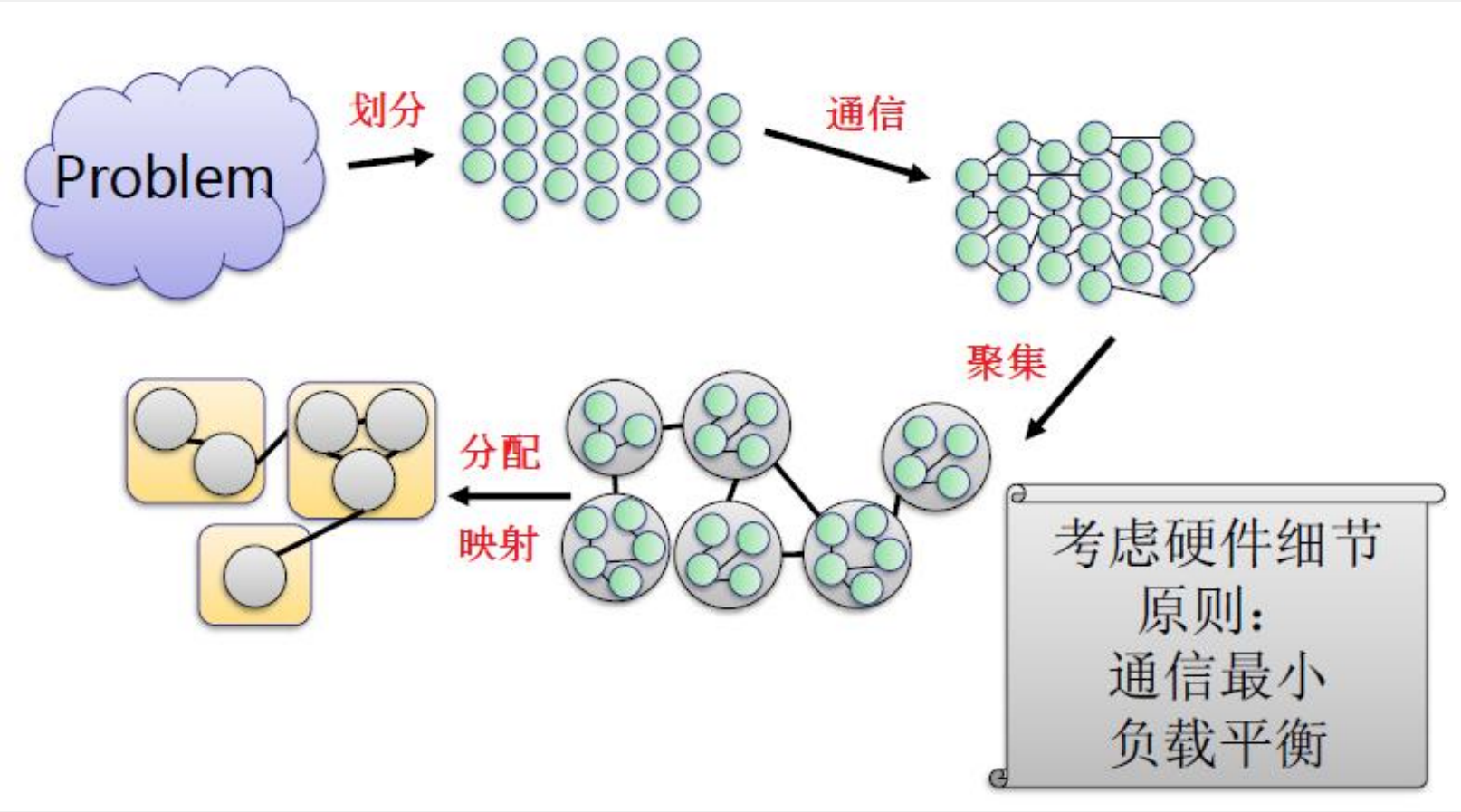
并行程序设计

- 福斯特方法Foster's methodology



并行程序设计

- 福斯特方法Foster's methodology



第一步：划分

- 划分方法
- 任务的特征

第一步：划分
——划分方法

- 如何将大任务分解为可并行执行的小任务？没有一个单一的方法能够应用于所有问题，讨论比较常用的分解方法：
  - 递归分解
  - 数据分解
  - 探索性分解
  - 推测性分解
  - 混合分解

划分方法1

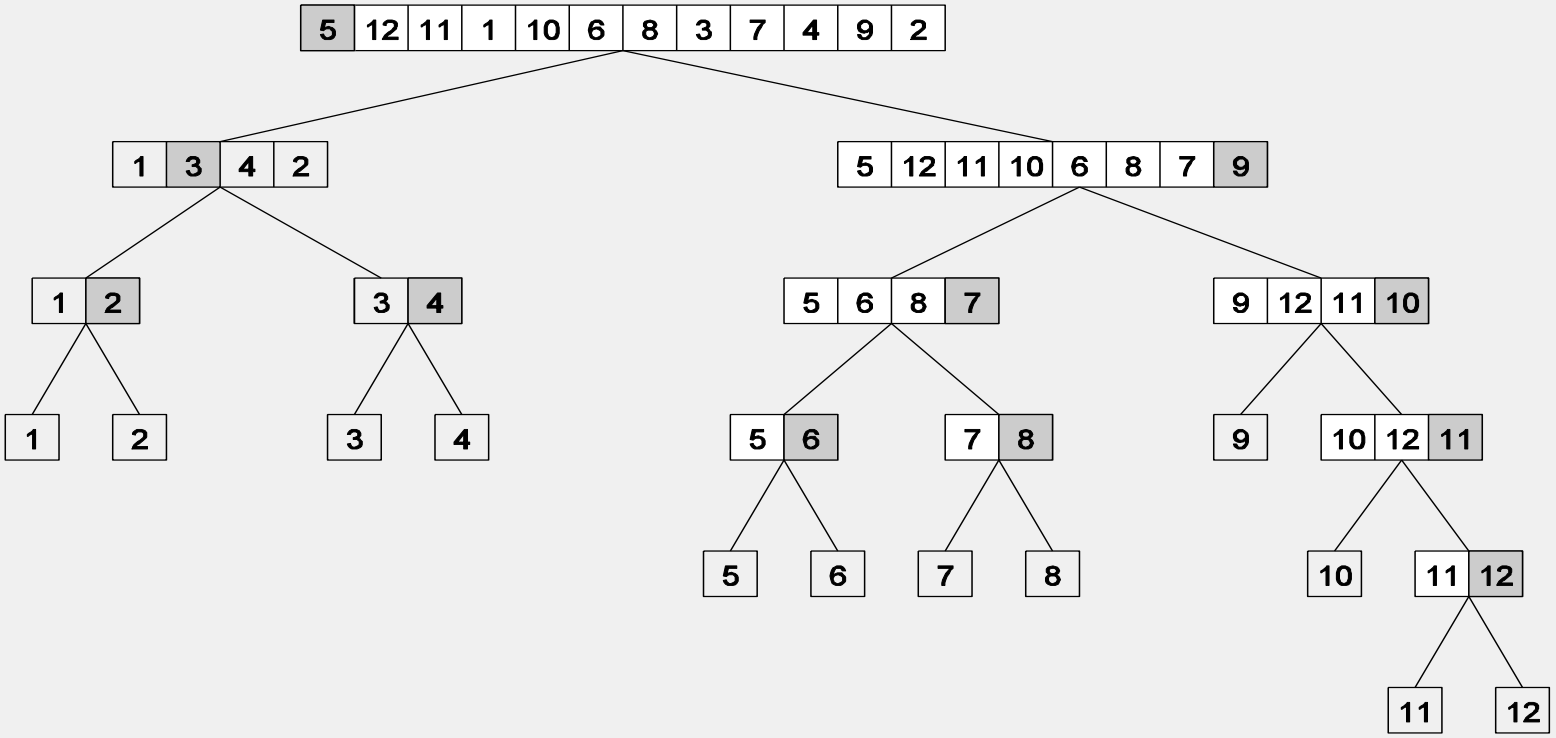
递归分解

递归分解

- 普遍适用于那些采用分治策略的问题。
- 一个给定的问题首先分解成一组子问题。
- 这些子问题又可以递归地分解成更小的子问题直到子问题的粒度满足你的要求。

递归分解：例子

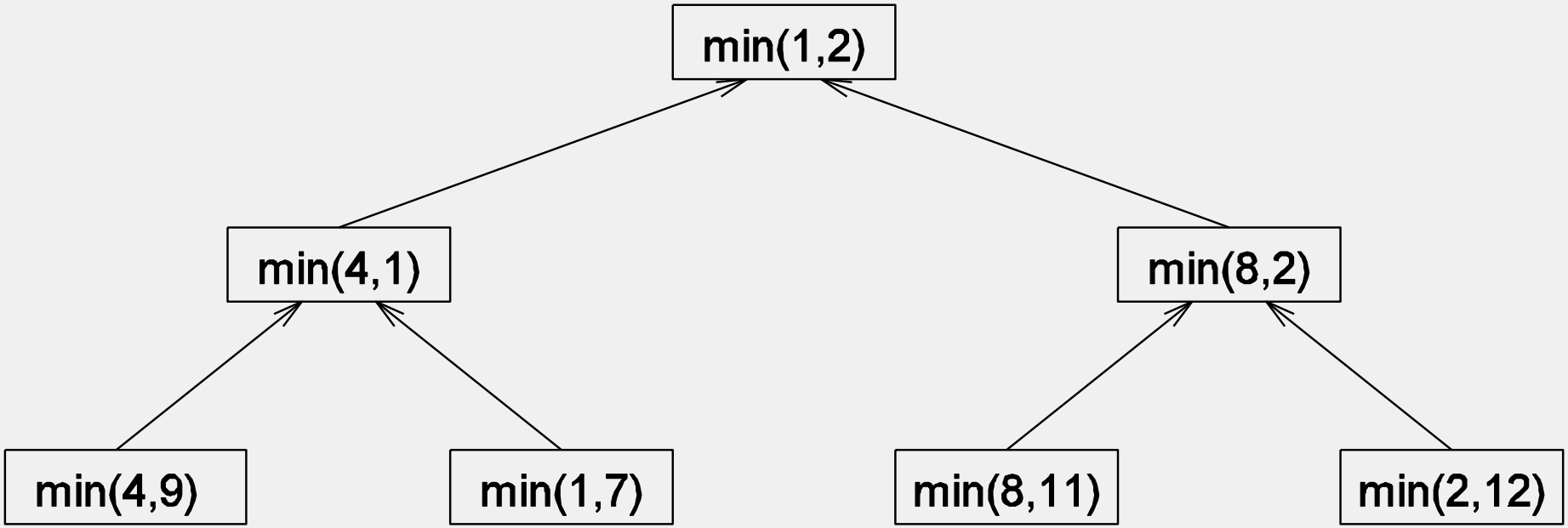
一个经典的使用分治算法的例子是快速排序，对于这个例子我们可以使用递归分解。



在这个例子当中，当这个数组通过关键点分割成子数组，每个子数组又可以并发地执行（每个子数组代表一个独立的子任务）。这步骤可以递归地重复下去。

递归分解：例子

前面那个例子说明了问题可以很自然地用递归分解来进行分解。下面我们将说明如何在集合{4,9,1,7,8,11,2,12}中寻找最小值。任务依赖图和每个任务需要的计算如下图所示：



递归分解：例子

像求一个数组的最小值（或者其他像求和，与操作等满足结合律的操作）这种问题可以化为分治策略问题。下面将举例说明：

我们首先用简单的Loop操作计算给定数组的最小值：

```
1. procedure SERIAL_MIN (A, n)
2. begin
3. min = A[0];
4. for i := 1 to n − 1 do
5.                 if (A[i] < min) min := A[i];
6. endfor;
7. return min;
8. end SERIAL_MIN
```

递归分解：例子

我们可以重新把Loop操作改成如下形式：

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.     min := A [0]  ;
5. else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.     if (lmin  < rmin) then
9.                 min := lmin;
10.    else
11.                min := rmin;
12.    endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

划分方法2

数据分解

数据分解

- 识别出需要进行计算操作的数据。
- 把这些数据分割到不同的任务中。
  - 这种数据的划分将导致问题的任务划分。
- 数据可以用多种方法进行划分-这将会导致并行算法的不同的表现性能。

数据分解：输出数据分解

- 在很多计算中，每个输出元素都可以作为输入元素的函数独立计算得到。

- 输出数据的划分自然地把问题分解为多个任务。

划分输出数据：例子

考虑两个***n*** x ***n*** 矩阵相乘的问题，矩阵A乘以矩阵B得到矩阵C。输出矩阵C可以被划分为四个任务，如下所示：

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

任务 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

任务 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

任务 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

任务 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

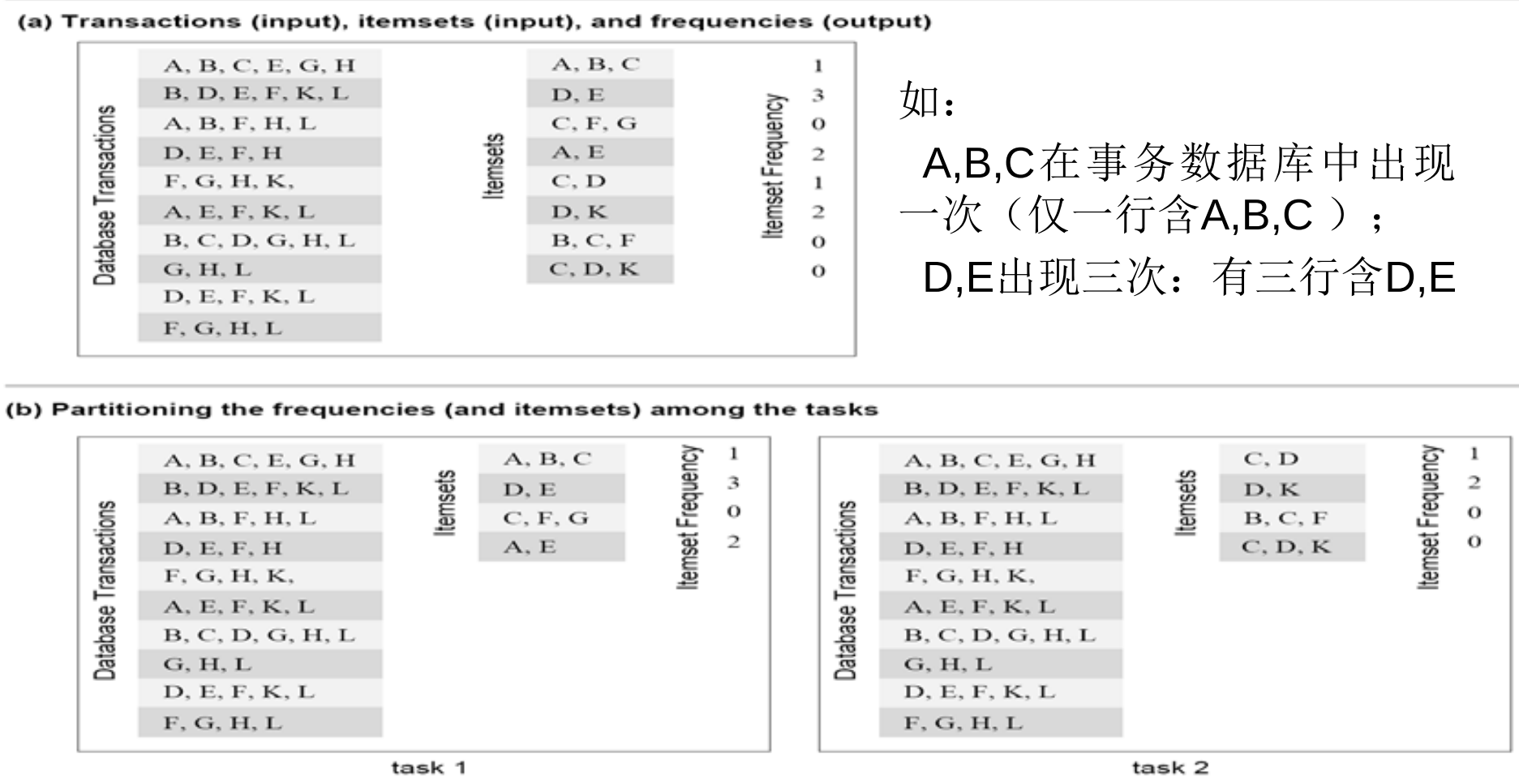
划分输出数据：例子

数据分解不仅仅只有一种任务分解。 例如，对于刚才那个例子，相同的输出数据划分，我们可以得到下面两种任务分解：

分解 I	分解 II
任务1: $C_{1,1} = A_{1,1} B_{1,1}$	任务1: $C_{1,1} = A_{1,1} B_{1,1}$
任务2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	任务2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
任务3: $C_{1,2} = A_{1,1} B_{1,2}$	任务3: $C_{1,2} = A_{1,2} B_{2,2}$
任务4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	任务4: $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
任务5: $C_{2,1} = A_{2,1} B_{1,1}$	任务5: $C_{2,1} = A_{2,2} B_{2,1}$
任务6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	任务6: $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
任务7: $C_{2,2} = A_{2,1} B_{1,2}$	任务7: $C_{2,2} = A_{2,1} B_{1,2}$
任务8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	任务8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

划分输出数据：例子

考虑计算事务数据库中的一个项目集出现频数问题。在这种情况下，输出（项目集频数）可以根据任务进行划分，即每个任务计算一部分项目集出现的频数。



Computing itemset frequencies in a transaction database.

如：

A,B,C在事务数据库中出现一次（仅一行含A,B,C ）；

D,E出现三次：有三行含D,E

划分输出数据：例子

从前面的例子可以得到如下结论：

- 如果数据库的事务在每个进程处都有拷贝，那么每个任务可以不用通过任何交互就能独立完成。

- 如果数据库的数据是分配到每个进程中（由于内存使用的关系），那么每个任务先计算部分的频数。然后这些频数再结合起来得到最后结果。

划分输入数据

- 只有当每一个输出结果都能作为输入的函数自然计算时，才能划分输出数据。

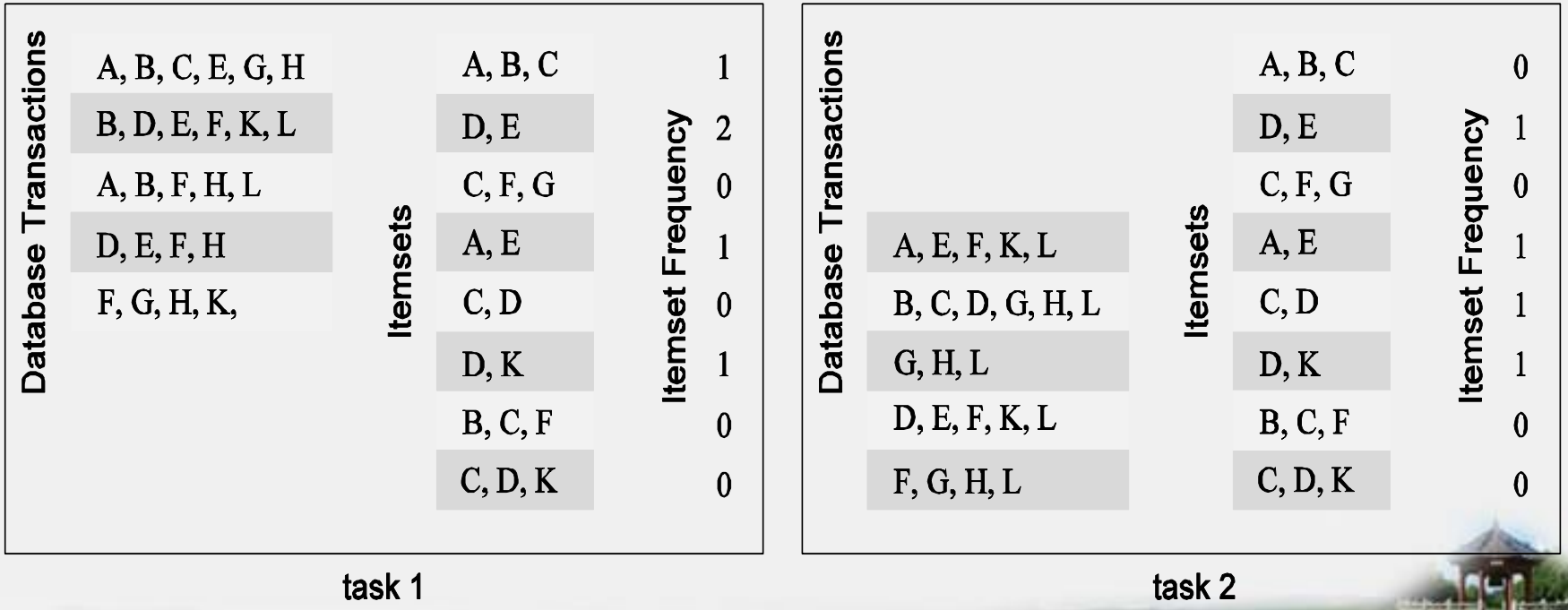
- 在很多情况中，由于输出事先是不知道，因此这是比较自然的分解方法。（例如，在寻找数组最小值的，对数组排序这些问题当中）

- 一个任务是与它的所分配的数据密切相关。这个任务的计算性能也跟所分到的那部分数据相关，后续任务将把前面的任务计算得到的部分结果结合起来。

分解输入数据：例子

像刚才那个数据库的项目集频数计算例子，输入的数据（事务集）是可以进行分解的。这样分解的一个任务就可以计算所有项目集在这部分事务数据中的频数，然后再把每个任务计算的项目集频数结合起来就可以得到最终结果。

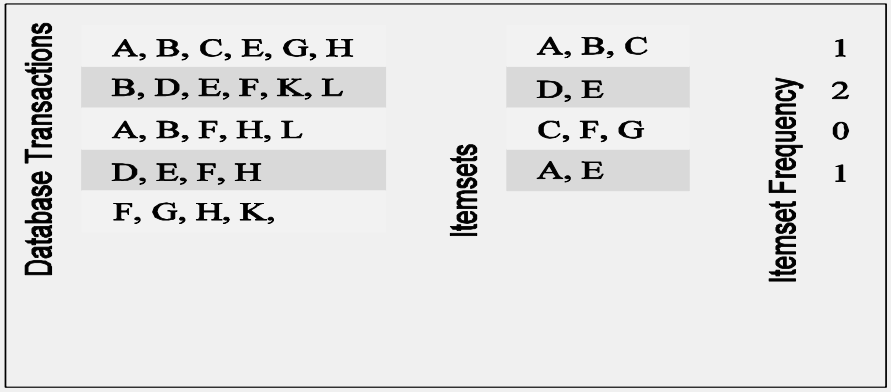
Partitioning the transactions among the tasks



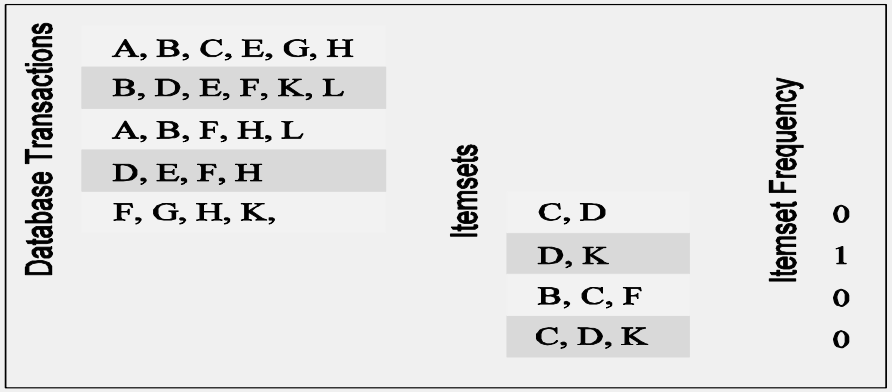
分解输入和输出数据

一般输入和输出数据可以一起进行分解，从而达到更高程度上的并发像刚才项目集频数计算的那个例子中，事务集（输入）和项目集频数（输出）都可以进行分解：

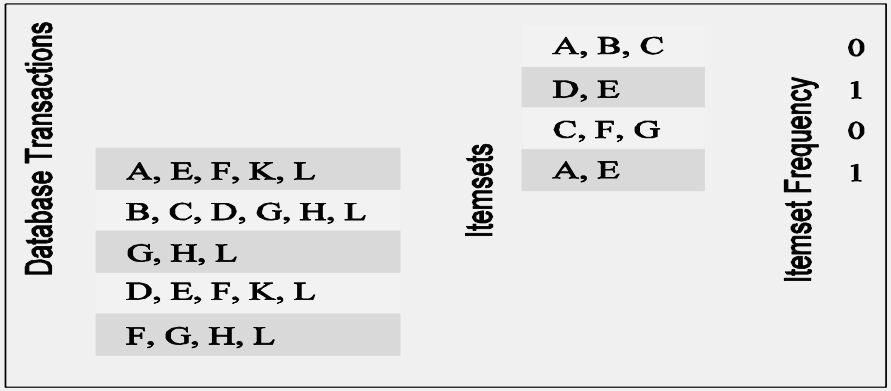
Partitioning both transactions and frequencies among the tasks



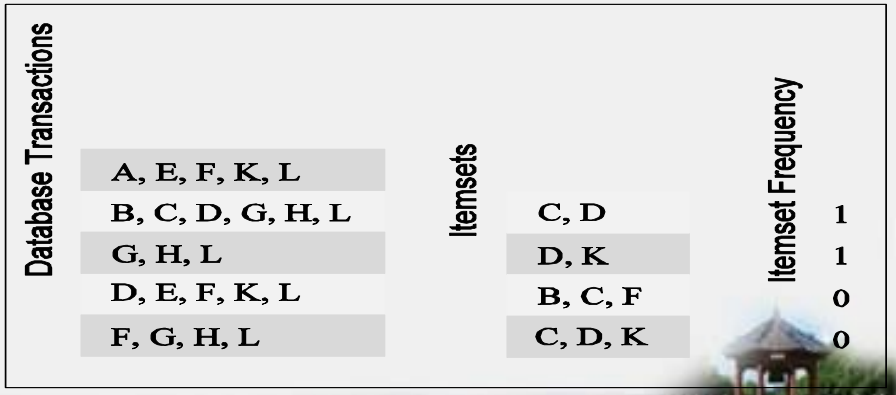
task 1



task 2



task 3



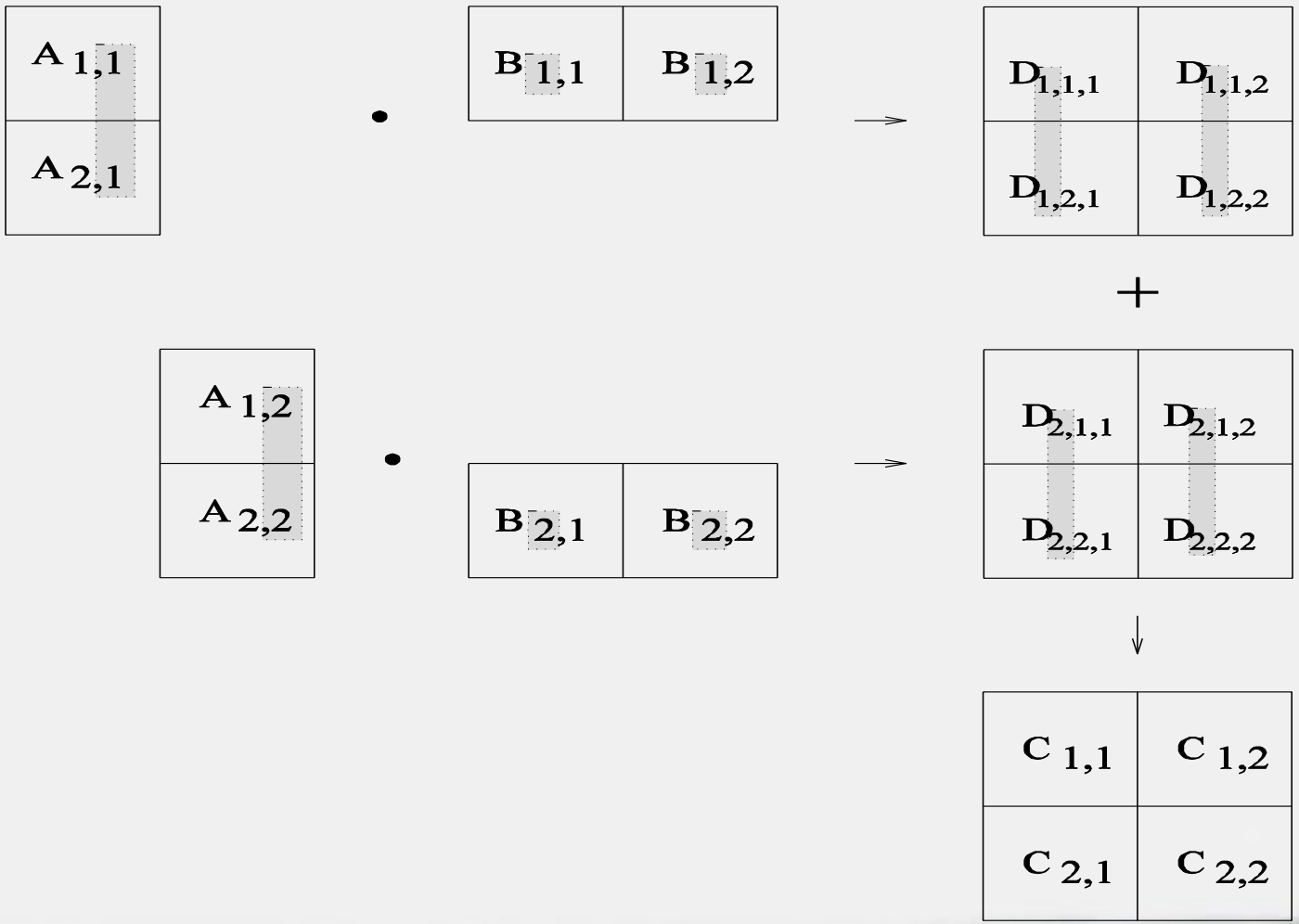
task 4

划分中间数据

- 计算一般可以看成从输入到输出的一系列转换。
- 在这些情况下，利用某一中间阶段产生的数据作为分解是比较好的。

划分中间数据：例子

让我们回顾一下之前的稠密矩阵相乘的例子，我们用图来表示如何利用中间矩阵D。



划分中间数据：例子

对中间数据的划分可以得到下面的分解，总共有8+4个任务：

阶段 I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

阶段 II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

任务01: $\boldsymbol{D}_{1,1,1} = \boldsymbol{A}_{1,1}\boldsymbol{B}_{1,1}$

任务02: $\boldsymbol{D}_{2,1,1} = \boldsymbol{A}_{1,2}\boldsymbol{B}_{2,1}$

任务03: $\boldsymbol{D}_{1,1,2} = \boldsymbol{A}_{1,1}\boldsymbol{B}_{1,2}$

任务04: $\boldsymbol{D}_{2,1,2} = \boldsymbol{A}_{1,2}\boldsymbol{B}_{2,2}$

任务05: $\boldsymbol{D}_{1,2,1} = \boldsymbol{A}_{2,1}\boldsymbol{B}_{1,1}$

任务06: $\boldsymbol{D}_{2,2,1} = \boldsymbol{A}_{2,2}\boldsymbol{B}_{2,1}$

任务07: $\boldsymbol{D}_{1,2,2} = \boldsymbol{A}_{2,1}\boldsymbol{B}_{1,2}$

任务08: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\boldsymbol{B}_{2,2}$

任务09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

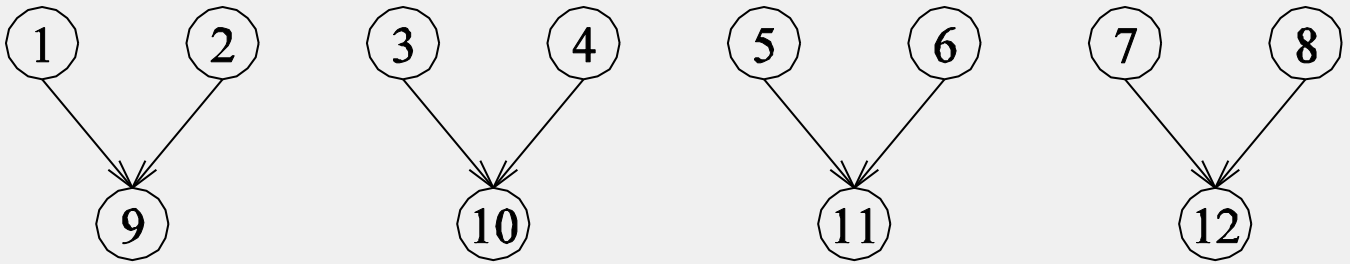
任务10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

任务11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

任务12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

划分中间数据：例子

上面一个例子的任务依赖图如下图所示：



“拥有者-计算”规则

- “拥有者-计算”规则是指分配到一部分数据的进程必须完成与这部分数据相关的所有计算。

- 在划分输入数据的情况中，拥有者-计算规则表明了所分到输入数据的进程都执行了这部分数据的所有计算。

- 在划分输出数据的情况中，所分到输出数据的进程都由所分配到这部分数据的进程执行所有的计算。

划分方法3

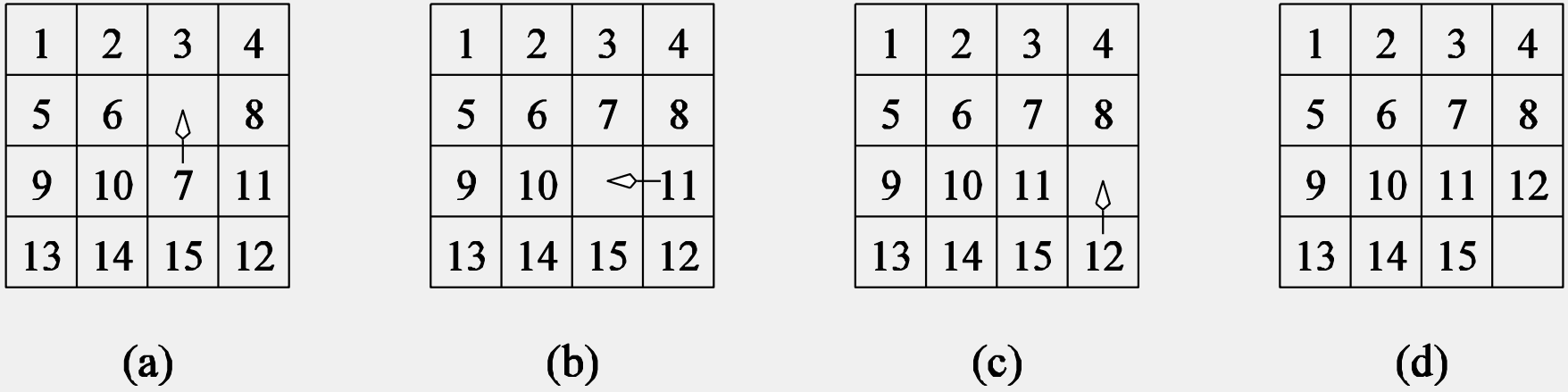
探索性分解

探索性分解

- 在很多情况中，问题的执行是并进的，因此问题也如此进行分割。
- 这些问题的解对应于解空间的一个搜索情况。
- 大量的离散优化问题都属于这种情况（0/1规划，整数规划，QAP），定理证明，博弈论。

探索性分解：例子

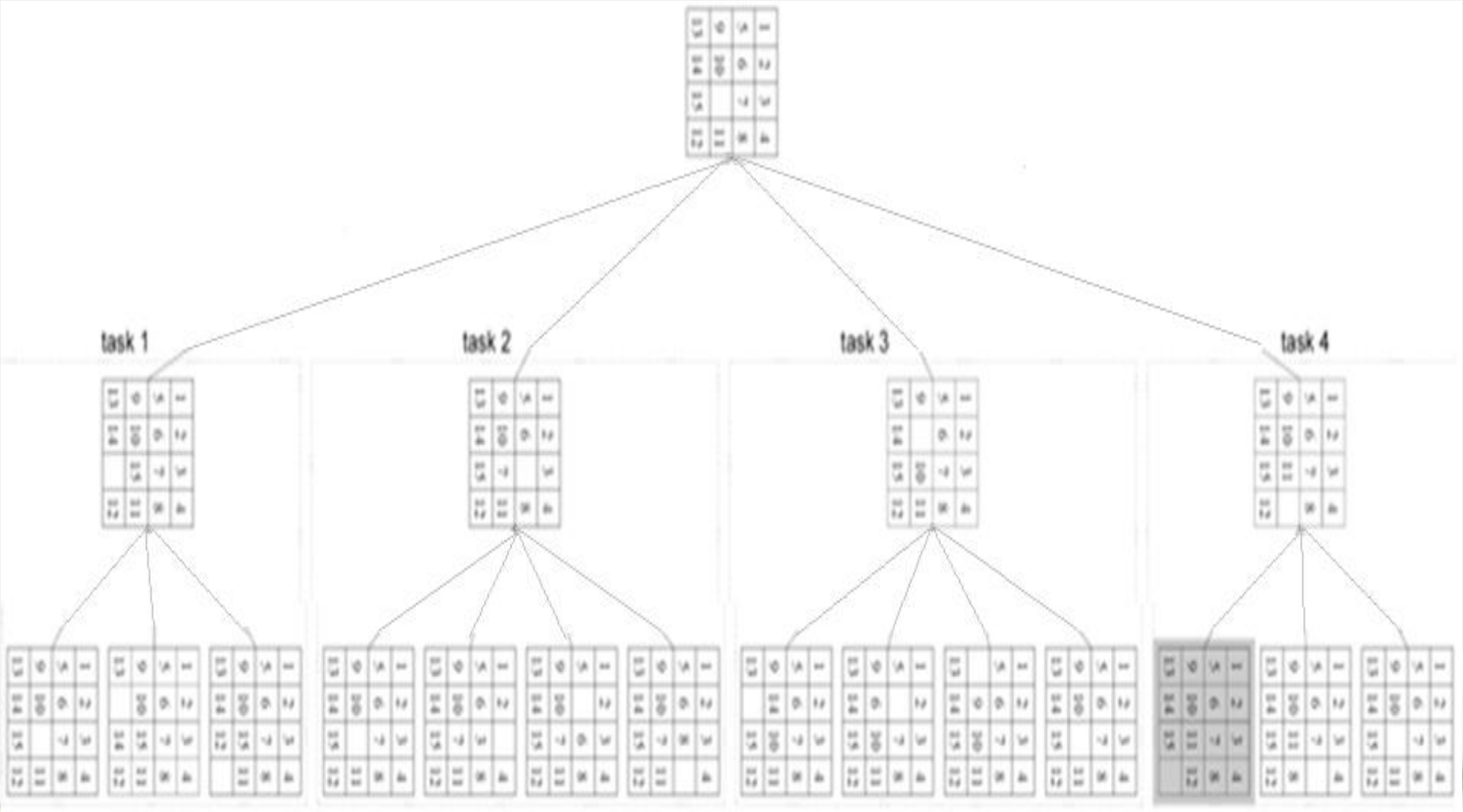
探索性分解的一个简单的应用就是拼好15方块的拼图。下图显示用三个步骤把拼图从初始状态转换成终止状态。



当然，这个问题的解一般不会这么简单求得出来。

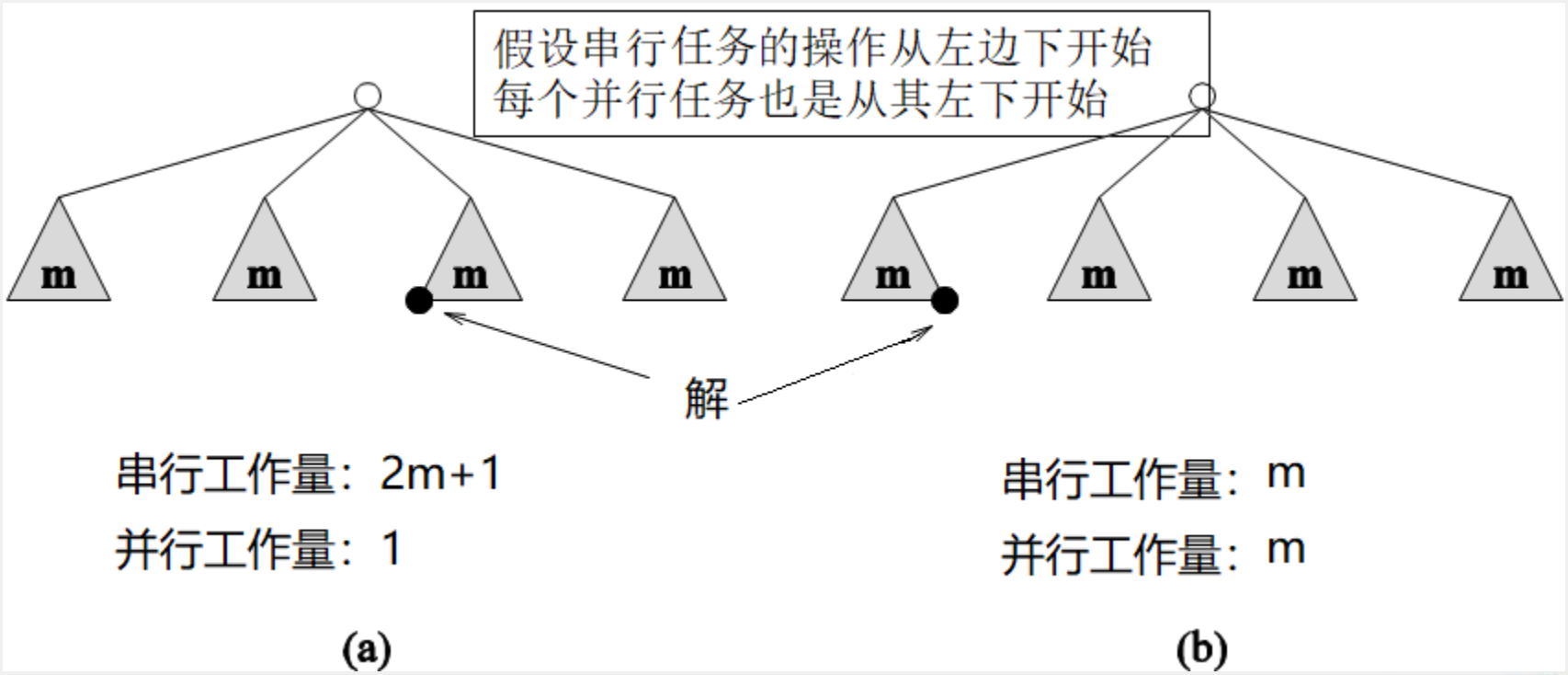
探索性分解：例子

这个状态空间的搜索可以通过产生大量后续状态来进行，并且每个任务负责一部分后续状态的产生。



探测性分解：异常计算

- 在很多情况下，探测性分解技术会改变的并行形式的计算工作量。
  - 一个任务获得结果而结束，其他未完成任务就可以终止
  - 并行计算时间可以少于(图a)或多于或等于串行时间(图b)
- 这个改变可能导致超线性加速比或者次线性加速比，也可能没有加速。



并行计算具超线性加速

并行计算没有加速

探测性分解：与数据分解区别

- 两者相似：搜索空间可以认为是被划分的空间
- 本质不同点：
  - 探测性分解：一个任务获得结果而结束，其他未完成任务就可以终止
  - 数据分解导出的任务都完全地执行，每个任务都获得最终解的一部分

划分方法4

预测性分解

预测性分解

- 在一些应用程序中，任务之间的依赖性预先是不知道的。
  - 在这样的应用程序中，识别独立任务是不可行的。
- 对于这种问题有两种方法来解决：
  - 一个是保守的方法，即当确保这个任务是没有依赖性的才把它当作独立性任务；
  - 另外一个是乐观方法，即使有可能判断错误也安排这个任务。
- 保守方法可能会导致较低的并发性而乐观方法则需要回滚机制来处理错误的判断。

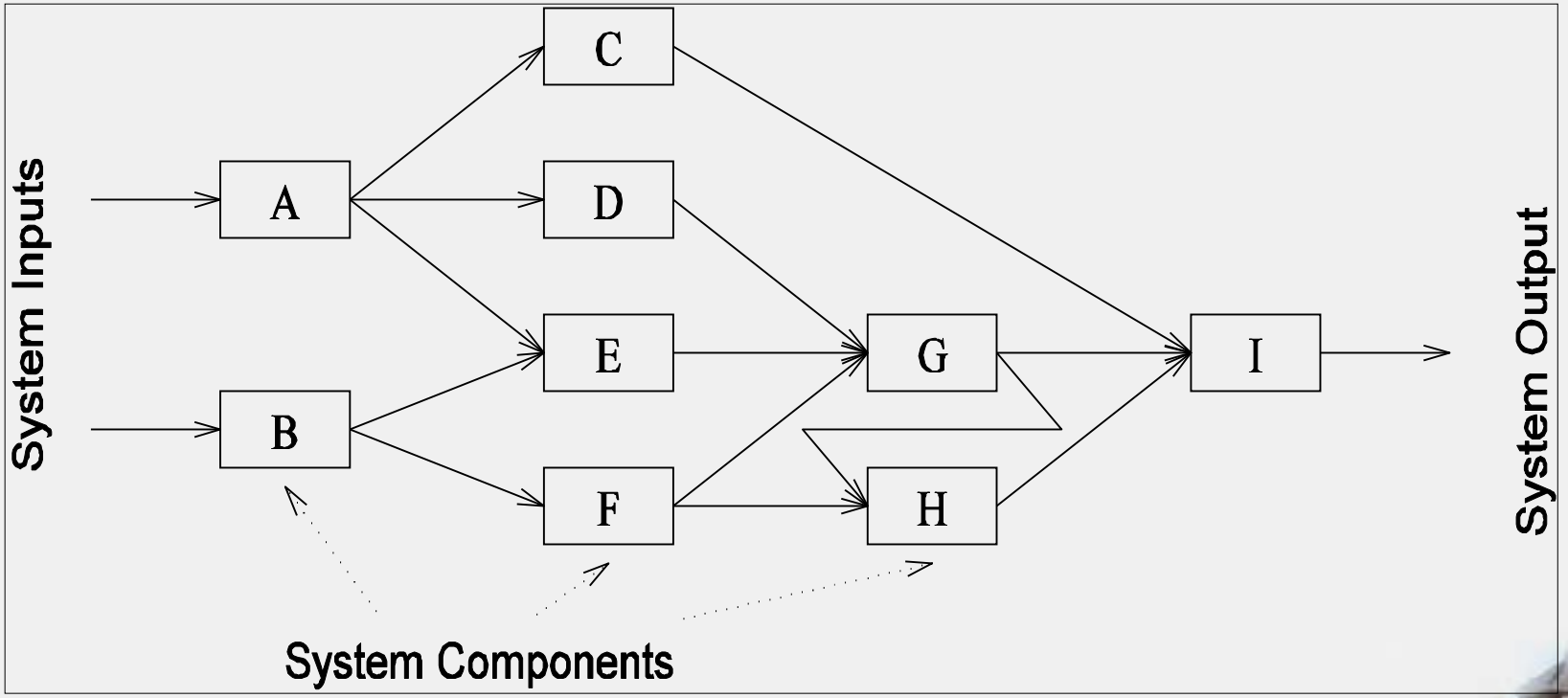
预测性分解：例子

预测性分解的一个简单的例子就是离散事件的模拟。

- 离散事件模拟的主要的数据结构是按时间顺序的事件列表。
- 事件是按时间顺序来精确提取出来的并进行加工的，如果需要的话，作为某个事件结果的事件也可以插入回事件列表中。
- 就像你一天的生活就可以看成一系列的离散事件-起床，准备出发，开车去工作，工作，吃午饭，继续工作，开车回家，吃晚饭，睡觉。
- 这些事件看起来是相互独立的，但是，在开车去工作的时候你可能遇到一些不愉快的意外而你就不想去工作了。
- 因此，用乐观的方法就可能需要使一些其他的事件进行回滚。

预测性分解：例子

另外一个例子是网络流的模拟。（例如，在一个数据包通过的流水线）在不同的输入数据和节点的延迟时间参数下，这个例子模拟了网络的行为。（注意对于不同数值跟服务速度，还有队列长度，网络是不稳定的）



划分方法5

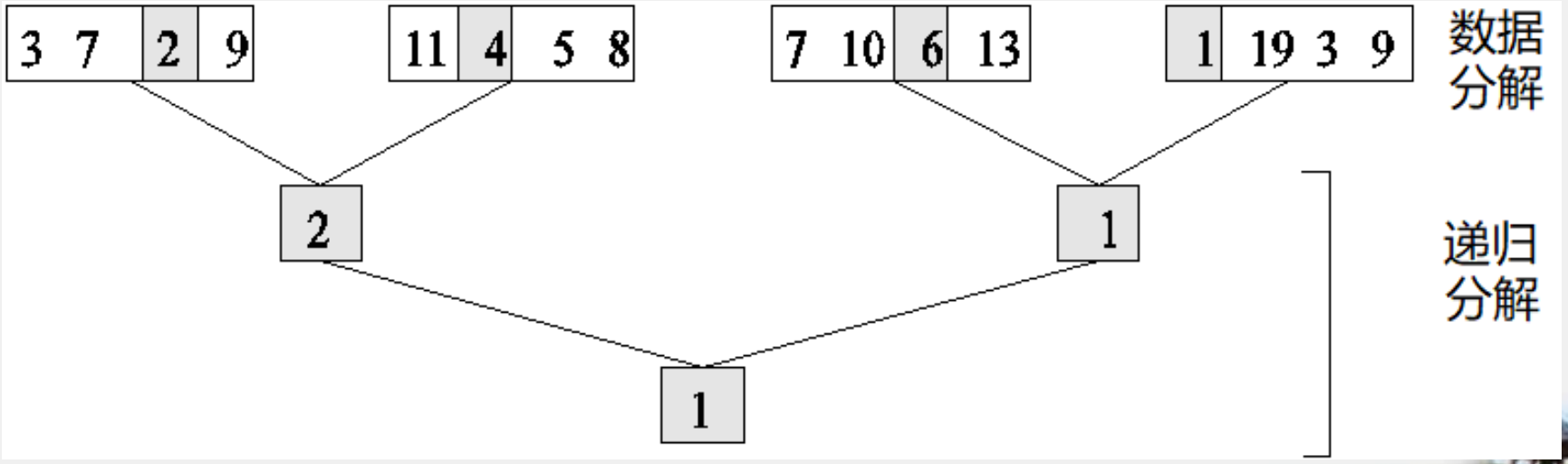
混合分解

混合分解

通常，对分解方法的混合使用对于分解一个问题是必须的。

考虑下面一些例子：

- 在快速排序中，递归分解的有效的并发性比较有限（为什么？）因此，对于数据分解跟递归分解方法的结合的效果会更好。
- 在离散事件模拟中，在每个任务处理中可能会有并发性。因此，使用预测性分解跟数据分解结合起来可能会更好。
- 像寻找最小值这种问题，数据分解跟递归分解结合起来会更好。



任务的特征

任务的特征

当问题被分解成一系列独立的任务，这些的任务的特征会影响到并行算法的选择和性能。相关的任务特征包括：

- 任务的产生.
- 任务的负载.
- 任务的粒度
- 任务相关的数据的大小.

任务产生

- 静态任务产生：任务在算法执行前是已知的。例如，典型的矩阵运算，图算法，图像处理应用，还有其他有规律的结构化问题也是。这些问题可以用数据或递归分解方法来进行处理。

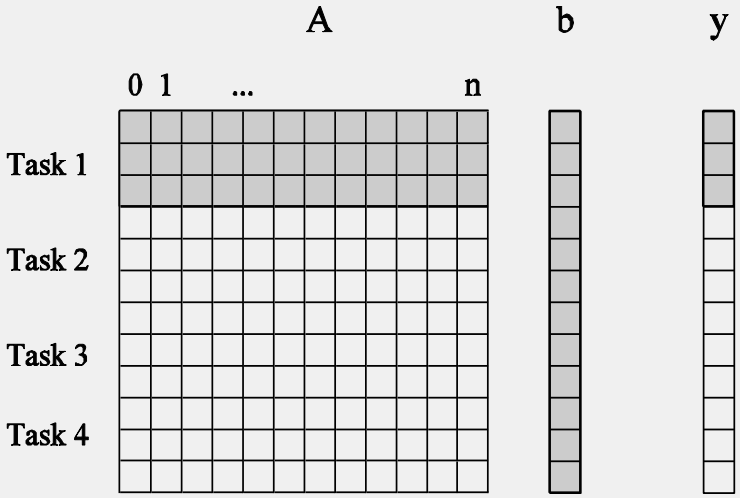
- 动态任务产生：任务是在我们计算过程中产生。像这种情况包括博弈游戏等。这些问题可以使用探测性分解和预测性分解进行处理。

任务负载

- 任务负载——任务的计算量（涉及第3步聚集）：
  - 一样（所有任务负载都一样）
  - 不一样。
  - 事先已知（或可以估计）
  - 未知或不确定（与输入有关）
    - 例如离散优化问题，就是很难估计它状态空间的大小。

任务的粒度

- 一个问题分解的任务数量决定了各任务的粒度大小。
  - 粒度与负载相关
- 如果一个问题分解的任务数量越多，那么任务的粒度就越细，负载就越小。如果分解成的任务数量比较少，那么每一个任务的粒度就比较大的，也就是较粗粒度。



稠密矩阵-向量积的一种粗粒度的任务分解，这样每一个任务都负责结果向量三个元素的计算。

任务相关的数据大小

- 负载不同的任务，相关的数据可能很小也可能很大。这些涉及后面几步的安排。
  - 一个小数据任务意味着算法可以让这个任务跟其他进程动态地进行交互（通信）。
  - 一个大数据的任务需要把任务绑定在一个进程上（分配，聚集），如果另外一个进程需那个任务的数据，算法可能会去重新构造数据而不是通过把任务的数据进行传递（例如0/1，整数规划）（通信）。

例：矩阵向量乘

- 将大任务矩阵向量乘y=Ab分解为向量乘：
  - 任务Yi：向量与矩阵行向量乘Aib
  - 任务Wij：将Aib分解为矩阵元素与向量元素乘：Aijbj——可并行执行的任务

- 也可这样分解：将矩阵向量乘y=Ab分解为A的分块矩阵向量乘：
  - 任务1：A矩阵块与b向量块乘
  - 任务2：A子块行向量与b子向量乘
  - 任务3： A子块行向量元素与b子向量元素乘： Aijbj——可并行执行的任务

第二步 通信分析

第二步 通信分析

这一步是确定划分得到的小任务需要的通信。

通信与任务间的依赖与交互有关：

- 任务的依赖关系：一些任务需要其他任务产生的数据，这样就要等到这些数据产生后再执行；
- 任务的交互关系：一些任务需要其他任务所拥有的数据，这样就要在任务之间通信。即任务数据依赖关系。

任务依赖关系和任务依赖图

任务依赖关系和任务依赖图

- 任务的依赖关系可以用有向图的形式来表示，这样的图叫做任务依赖图
  - 节点代表一个任务，边则代表任务之间的依赖关系。

例子：数据库查询处理

考虑下面查询语句的执行:

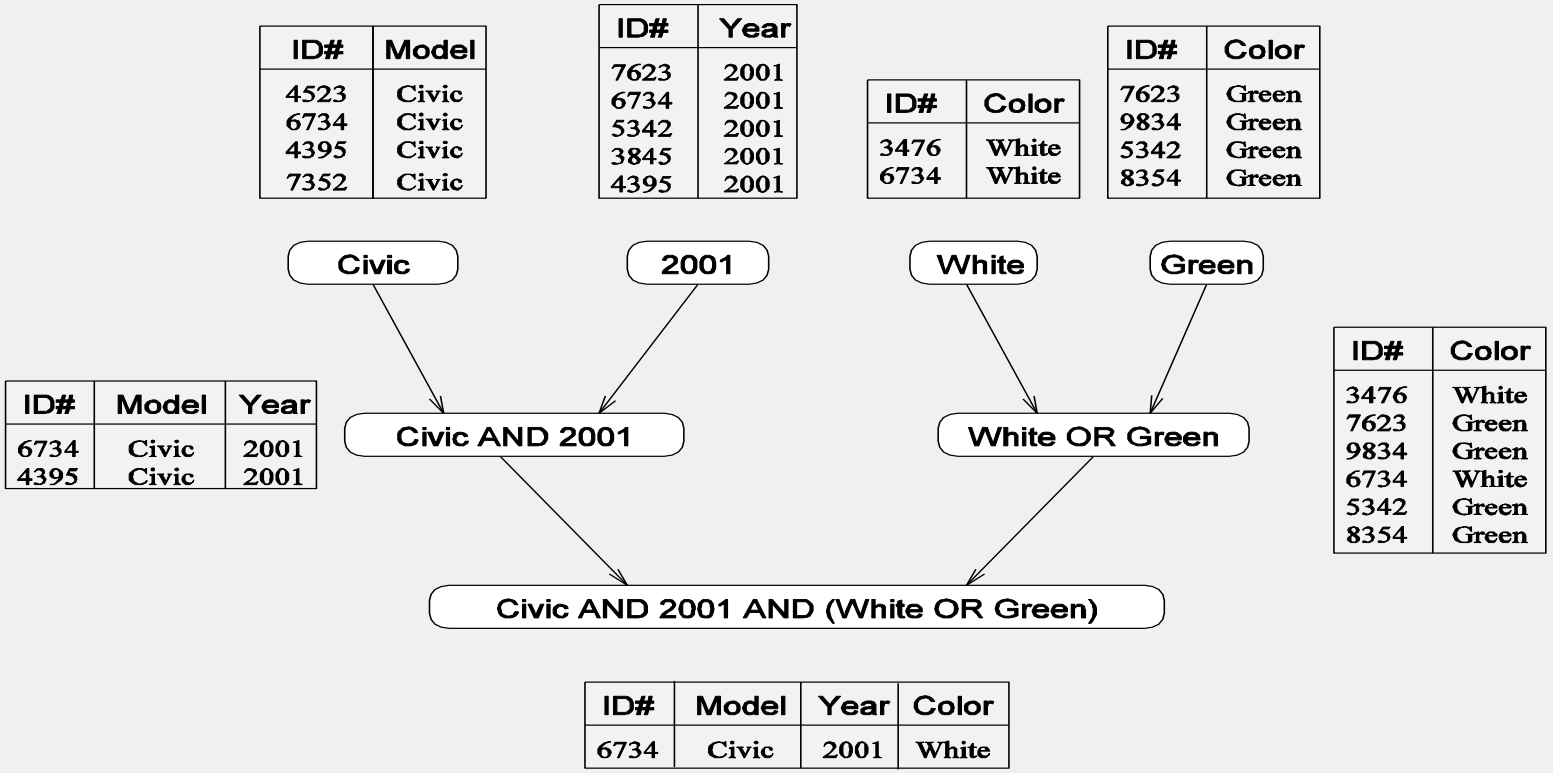
MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

在下面的数据库:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

例子：数据库查询处理

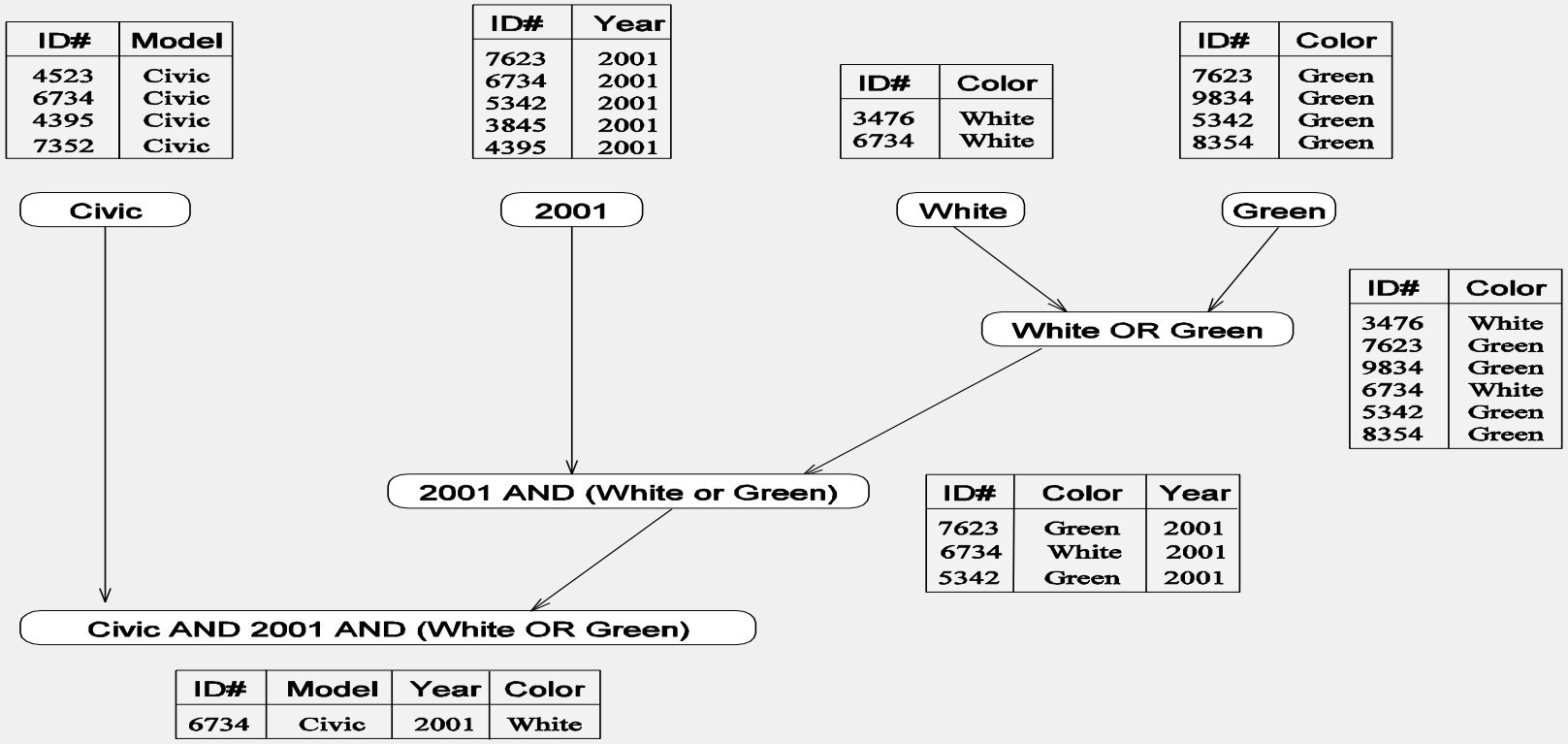
- 这个查询操作可以被分解为多个子任务
  - 分解的方法非常多。
  - 每个任务可看成生成满足一个条件的中间表，并为另一任务的输入。



把给定的查询操作分解成一些任务。
图中的边表示一个任务的输出是另外一个任务的输入。

例子：数据库查询处理

- 可以也用另一种方法进行分解.



另一种分解的任务之间的依赖关系。

不同的分解方法可能会使它们最终并行性能产生差异。

并发度

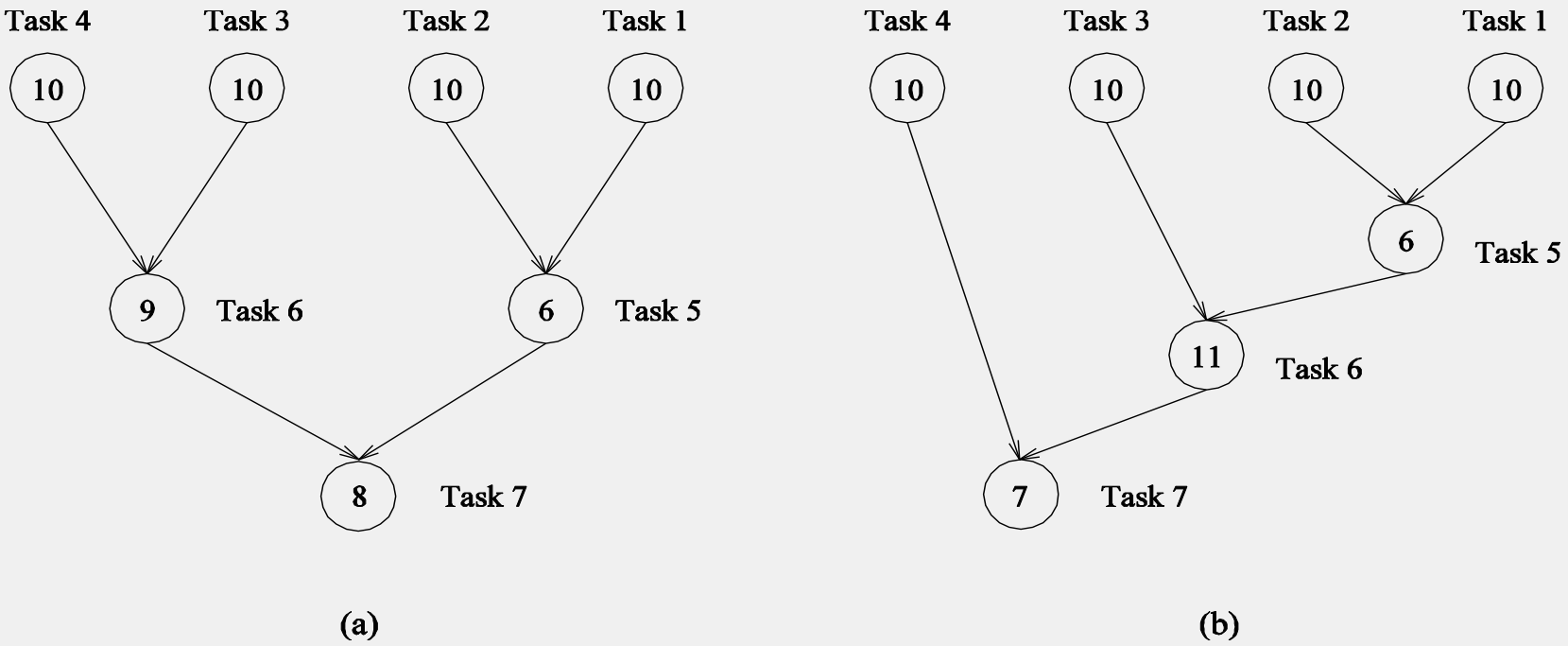
- 并行执行的任务数量就是分解的并发度。
- 并行执行的任务数量会影响程序的执行情况，任意时刻可执行的最大任务数称为最大并发度。
  - 数据库查询这个例子的最大并发度是多少呢？
  - 通常，树形任务依赖图的最大并发度等于叶子数(假定每个任务的计算量为1)。
- 平均并发度是程序在执行过程中能并行运行的任务平均数量。假设数据库这个例子的每一个任务都需要相同的运行时间，那么每一个种分解平均并发度是多少呢？
  - 通常，树形任务依赖图的平均并发度等于任务总数/树高(假定每个任务的计算量为1)。
  - 平均并发度=所有任务总计算量/关键路径的计算量。
- 当分解的任务粒度更细时，程序的并发度会增加，反之亦然。

关键路径长度

- 依赖图中的一条有向路径代表一串要顺序执行的任务。
- 这种路径越短，代表并行后的程序的执行时间越短。
- 任务依赖图中的最长路径的长度叫做关键路径长度。

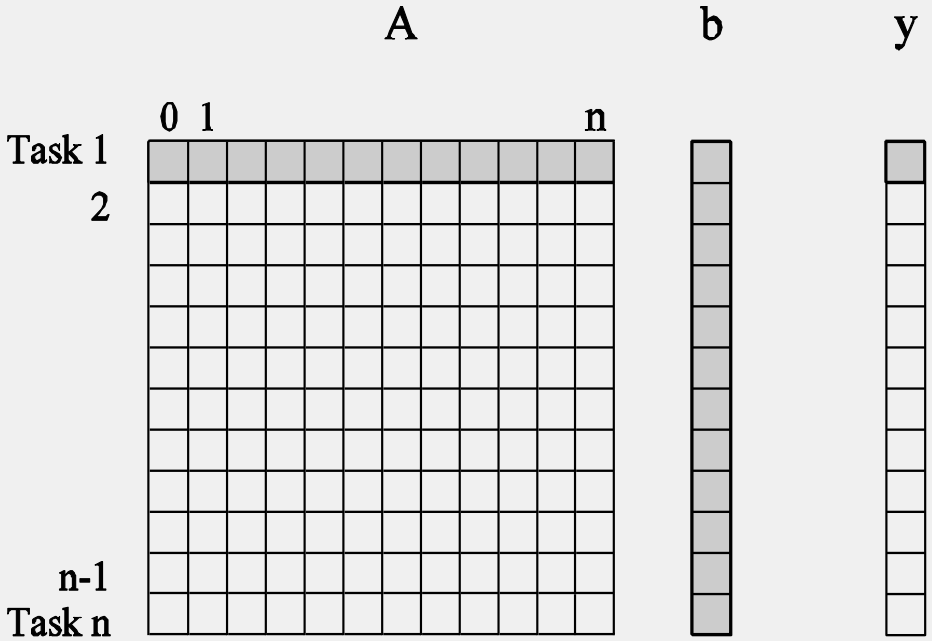
关键路径长度

下面为两种数据库查询分解方法的任务依赖图:



这两个任务依赖图的关键路径长度分别是多少？如果每个任务需要10个时间单位，每个分解的最短并行执行的时间是多少？为了达到最短的并行运行时间每种情况需要多少个处理器？它们的最大并发度是多少？

例子：稠密矩阵与向量相乘



输出向量y的每一个元素的计算都是独立于其他元素的。因此，稠密矩阵-向量积可以分解成n个任务。图中强调了任务1使用矩阵和向量的一部分。

观察结果：当任务共享数据（例如，向量b）,它们之间没有任何依赖。也就是说，没有任务需要等待其他任务的完成。所有任务的所做操作都是一样的。这是不是这个问题可分解得到的最大任务数量？

并行性能的限制

- 当分解粒度越来越小时并行算法的运行时间可以变得很短。
- 每个问题的任务分解粒度都有一个固有的限度。例如，在稠密矩阵-向量积这个例子中，并行任务的数量不可能超过n^2。
- 并发的任务也有可能需要与其他任务进行数据交互，这样会引起通信开销。因此，对分解粒度以及关联开销的权衡是决定并行算法性能的一个重要因素。

任务交互关系和任务交互图

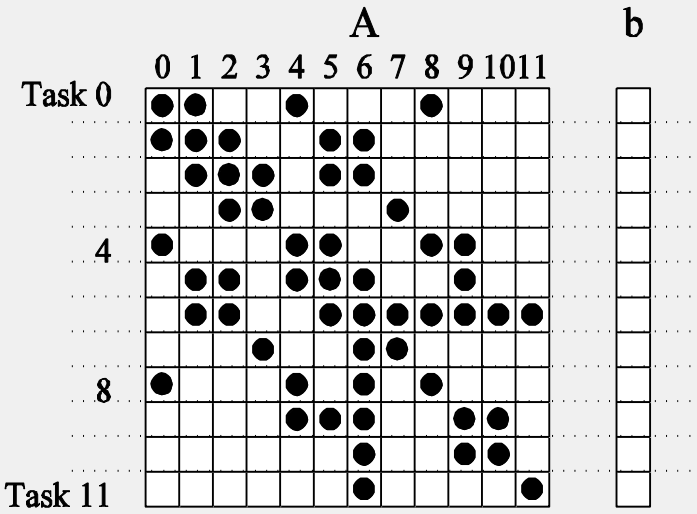
任务交互图

- 在一个分解里，子任务通常会跟其他任务交换数据，任务之间具有任务交互关系，其对应的图称为任务交互图：
  - 图中任务（即节点）和它们的数据交互关系（即边）构成了任务交互图。
  - 任务交互图表示数据依赖关系，而任务依赖图则表示控制的依赖关系。
  - 即使是在分解任务之间没有依赖关系，但可能会有交互关系。
    - 例如，在稠密矩阵-向量积这个例子中，任务之间没有依赖关系，但如果向量b没有被复制给每个任务所在的处理器，他们还是需要交换向量b的数据。

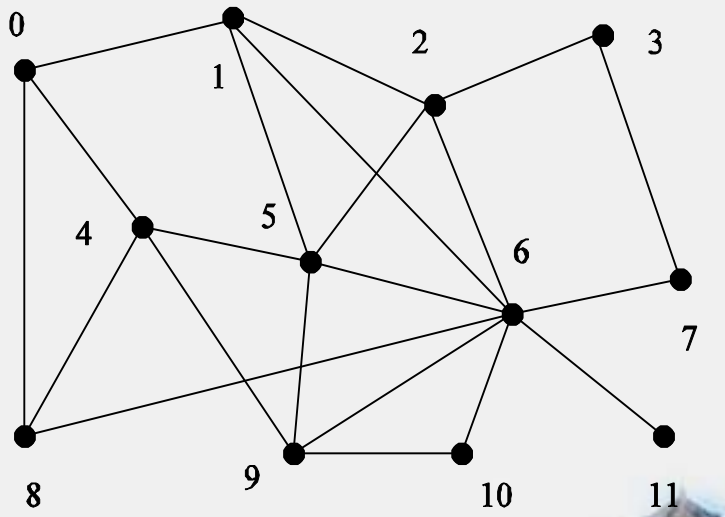
任务交互图--数据依赖图：一个例子

考虑下面一个问题：稀疏矩阵A与向量b相乘(y=Ab)。下面可以得到如下结论：

- 像先前一样，结果向量的每个元素的计算可以看成一个独立的计算。
- 但跟稠密矩阵-向量积不一样的是，只有矩阵A中的非0元素才参与计算。
- 如果考虑内存的最佳利用率，我们把b的数据分到各个任务中，那么每个任务需要交互的任务如图（a）所示。（其中图（a）表示的是交互的邻接关系）
  - 任务i：y[i]的计算者，A[i,*]和b[i]的拥有者
    - 消息传递模式中，任务i需要传递数据到其他任务



(a)



(b)

任务交互图、粒度和通信

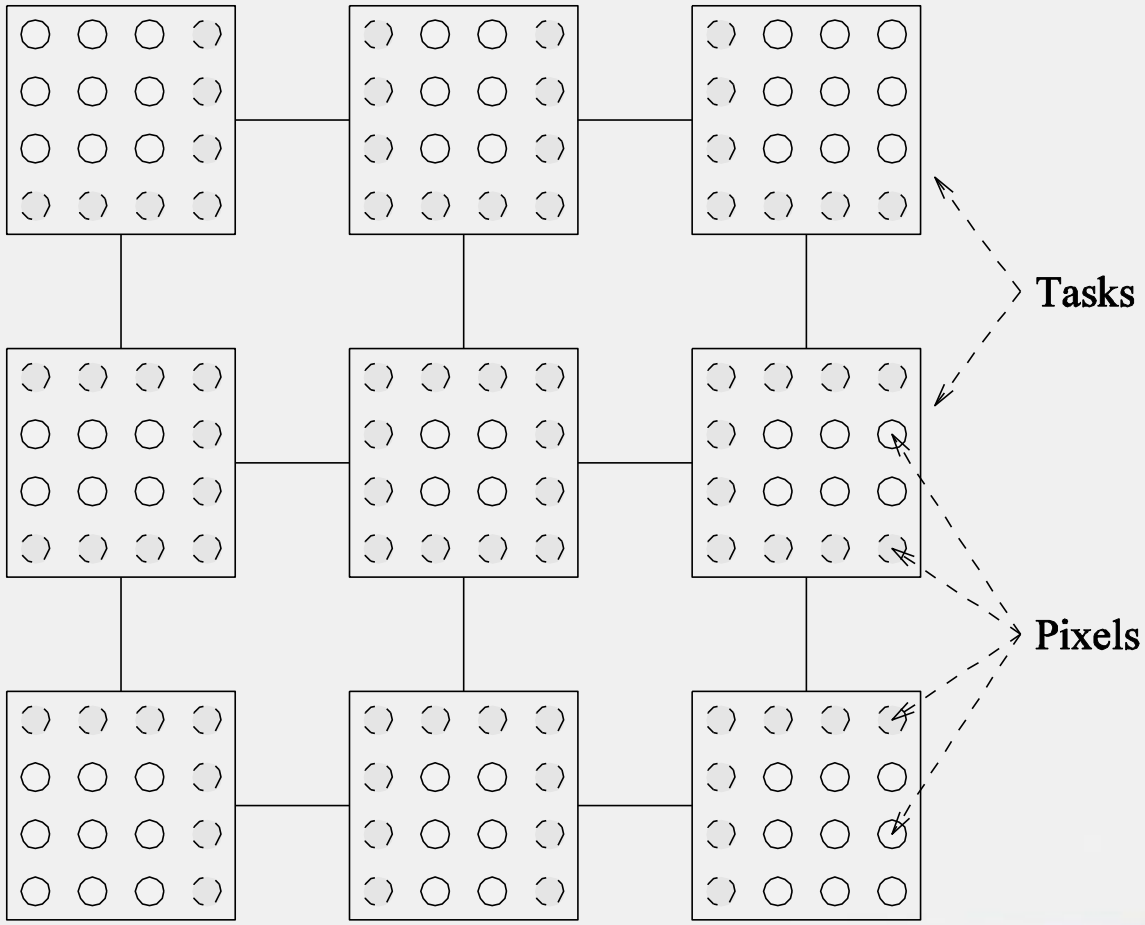
- 一般来说，如果分解的粒度变细，通信的花销则会越大。
  - 例子：考虑之前的那个稀疏矩阵-向量积的例子。假设每个节点用一个单位的时间来进行计算，并且每一个交互（边）需要一个单位时间的开销。
    - 把节点0看成一个独立的任务，其中包括一个单位的计算时间和3个单位通信时间。
    - 现在，如果我们把节点0,4和5当成一个任务，那么这个任务的计算时间为三个时间单位，而通信时间则变成了4个时间单位（4条边）。明显地，这样的计算通信时间比比刚才的更好。

任务交互的特征

- 任务交互关系产生时间分为两种：
  - 静态交互：任务和它们之间的交互在程序执行之前是知道的，这样设计程序相对来说比较简单。
  - 动态交互：任务相互作用的时间在先前并不知道。因此，这些交互比较难以实现，例如，当我们用传递消息API的时候。
- 每种形式又分为
  - 有规则的交互：对于交互有有限的模式（在图像浓淡处理中）。这些模式有利于被实现。
  - 没规则的交互：这些交互缺少规则的模式。

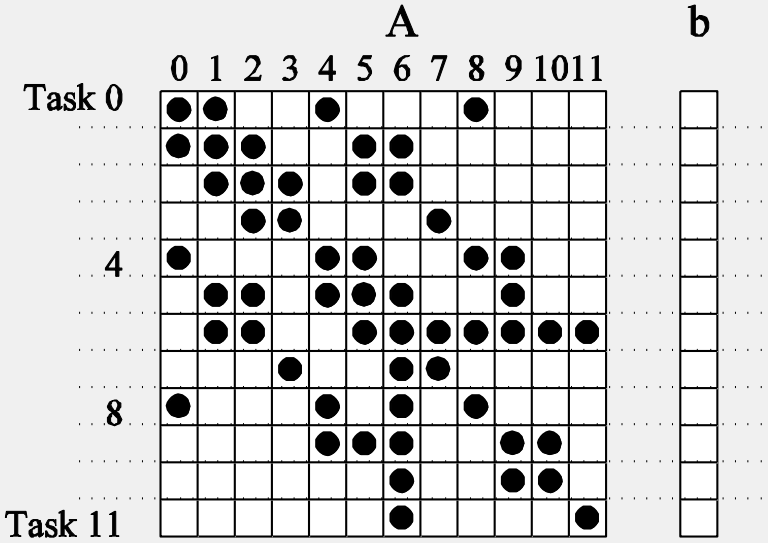
任务交互的特征：例子

- 一个简单的有规则的静态交互模式的例子是处理图像抖动。交互模式如下图所示：

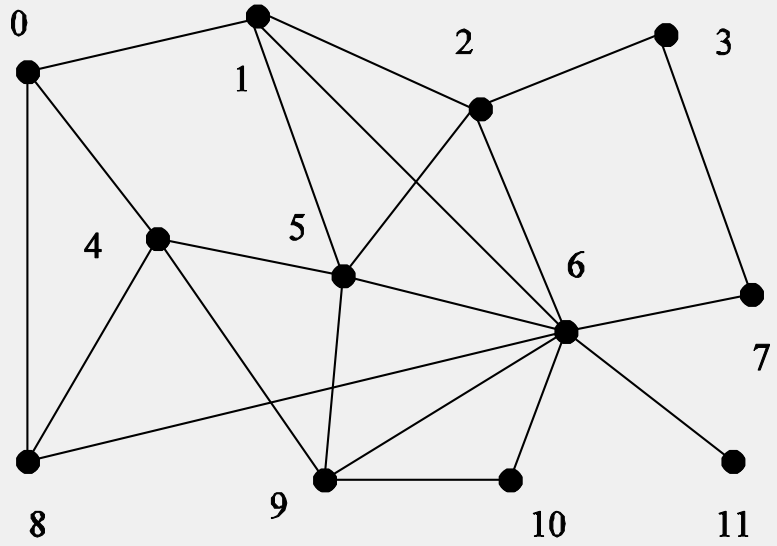


任务交互特征：例子

稀疏矩阵-向量积是一个静态无规则交互模式的一个很好的例子。下面就是稀疏矩阵跟它所关联的交互模式：



(a)



(b)

任务交互的特征

- 根据任务间共享数据访问方法不同，交互分为：
  - 只读交互：任务只需对许多并发任务之间共享的数据进行读访问。
  - 读写交互：任务需要对其他任务的数据进行读并且需要修改。
    - 读写交互的代码相对比较难写，并且需要另外的同步原语的支持。

任务交互特征

- 根据任务交互的方向，交互分为：
  - 单向交互：可以由相互交互的两个任务中的一个发起和结束。
  - 双向交互：需要参与交互的两个任务同时参与。
- 在使用消息传递API时，单向交互在某种程度上更难编写代码。

例：矩阵向量乘

- 分析各层任务之间的依赖关系，获得任务依赖图——注意根据各大任务的不同分解，依赖关系也不同；
- 分析最小任务之间的交互关系，获得任务交互图

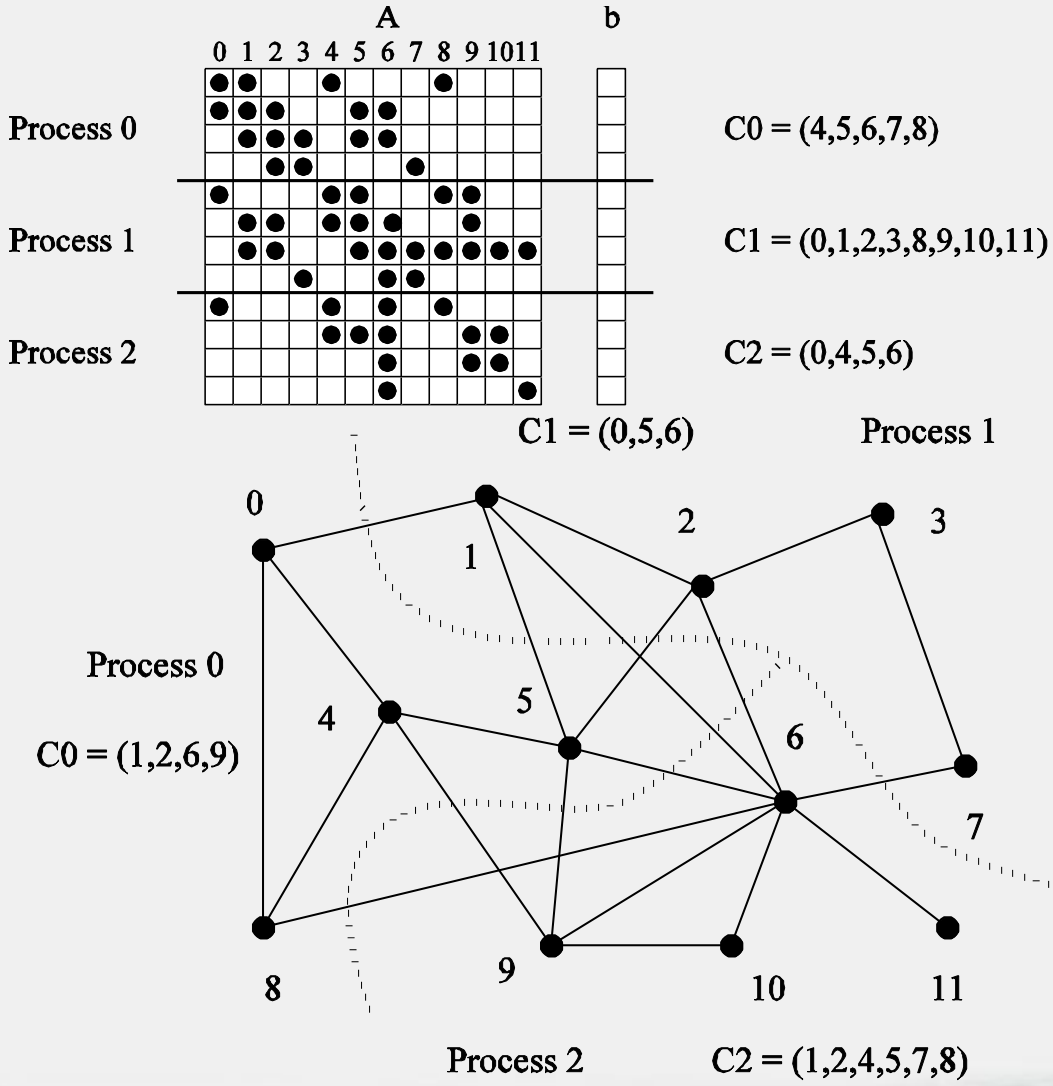
第三步 聚集

第三步 聚集

- 主要根据任务依赖关系并兼顾任务交互关系进行聚集：将一些小任务合并为大任务
  - 以矩阵向量乘为例。将y=Ab的任务分解为最小任务W_{ij}：计算$A_{ij}b_j$
    - 聚集方案1：将i相同W_{ij}合在一起；(按行分解方案)
    - 聚集方案2：将i,j满足$i_k<=i<i_k+p,j_k<=j<j_k+p$的$W_{ij}$合在一起，k=0，1，…，p-1. (分块方案)
- 聚集完成后需要重新分析聚集后的任务交互关系和任务依赖关系

任务划分：根据稀疏矩阵图进行聚合

根据稀疏矩阵图来进行稀疏矩阵-向量积的计算和聚合。Ci表示通讯关系。



第四步 分配/映射

第四步 分配/映射

- 本步骤是将聚集好的任务分配到进程/线程中，使得
  - 进程/线程负载均衡
  - 进程/线程间的交互开销最小
- 这是理想

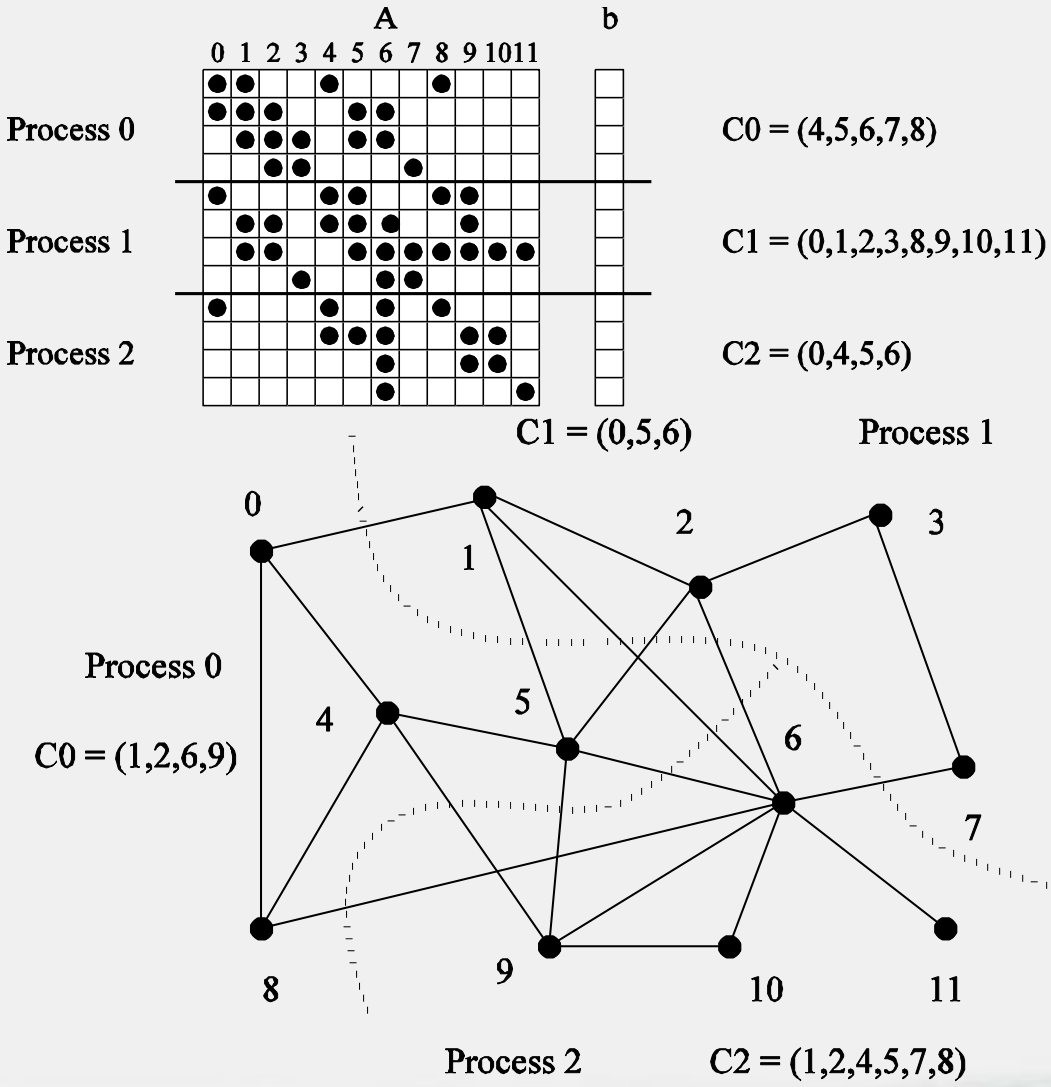
任务分配/映射到进程

- 一般来说，分解的任务的数量会超过进程的数量（此处进程代表一个抽象的实体，表示执行任务的处理代理和计算代理）。
  - 由于这个原因，一个并行算法必须提供一个任务到进程的分配/映射方法。

注意：我们把映射称作为从任务到进程，而不是到处理器。这是因为在标准的程序编程端口处，不允许简单地把任务绑定到物理的处理器上。也就是说，我们把任务聚合起来放到进程里面并且通过系统把进程映射到处理器上。我们说的进程，不是像UNIX的那种进程，而是一系列任务和相关数据的集合。

任务划分：根据稀疏矩阵图进行映射

根据稀疏矩阵图来进行稀疏矩阵-向量积的计算和映射。Ci表示通讯关系。



任务分配/映射到进程

- 适当的从任务到进程的映射对算法的并行性能来说是很重要的。
- 映射是由任务依赖图和任务交互图来共同决定的。
- 任务依赖图是用来确保工作在任何时刻能够比较平等地分配到进程里（最小化空转和最优负载平衡）。
- 任务交互图是用来确保进程之间的交互最小化（最小通信量）。

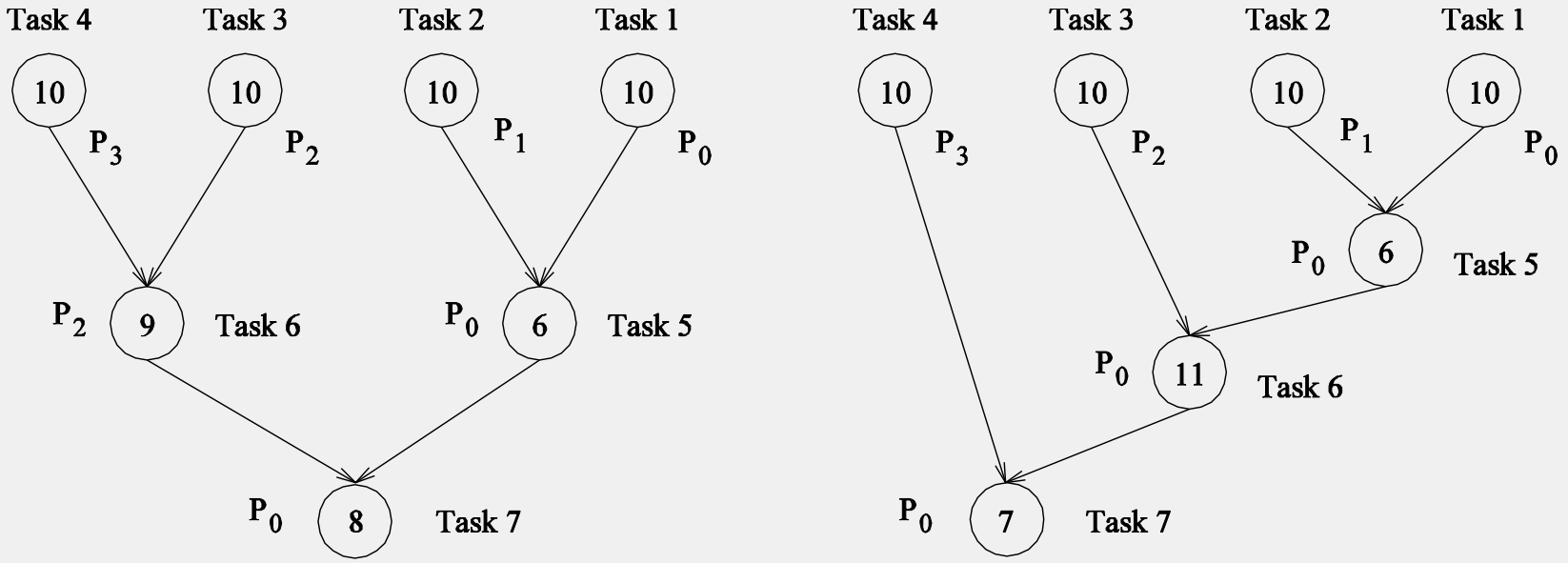
任务分配/映射到进程

一个适当的映射必须通过如下方法来减少并行程序的执行时间：

- 把相互独立的任务映射到不同的进程。
- 尽可能快地把关键路径上的任务分配到空闲的进程。
- 通过映射，尽量减少进程间的交互，而把密集的交互放在同一进程中。

注意：这些标准通常会互相冲突。例如，把问题分解成一个任务（或者根本就不分解）可以减少通信量，但是不会使程序获得加速。你还能想出其他类似这样的冲突的例子吗？

映射与进程：例子



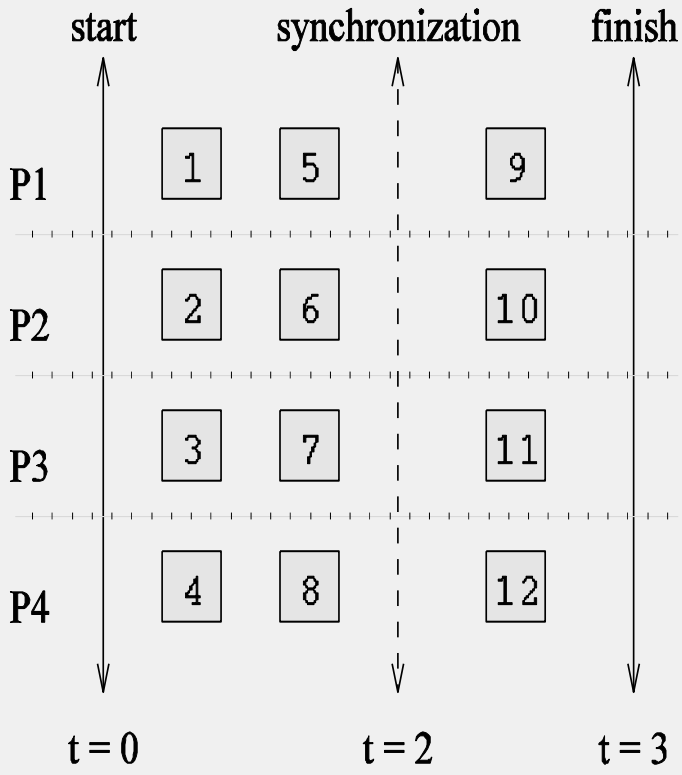
把数据库查询问题分解成的任务映射到进程中。通过任务依赖图的层次我们可以得到我们想要的映射(同一层次的节点都没有依赖关系)，即同一层次的任务分配到不同的进程当中去。

最小化运行开销技术简介

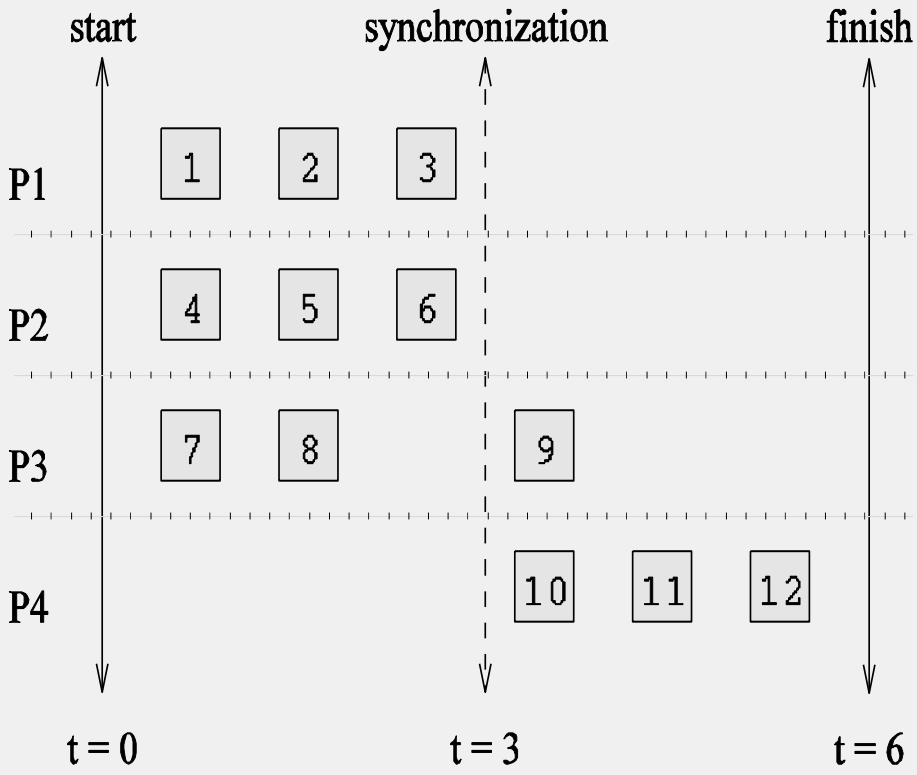
- 当一个问题被分解成并发的多个任务，这些任务就被映射到不同的进程中（可以在并行平台执行）。
- 映射必须最小化运行开销。
  - 主要的开销是进程交互开销和进程空闲开销。
  - 最小化这些开销一般是互相冲突的。
  - 把所有的工作都分配到一个进程可以减少交互开销但是就最大化了空闲开销。

最小化空闲开销的映射方法

映射必须保证同时减少空闲进程跟确保进程间的负载平衡。仅仅保证负载平衡并不会减少空闲开销。



(a)



(b)

最小空闲开销的映射方法

映射方法包括静态的和动态的。

- 静态映射：在程序执行前就把任务分配给进程。为了使程序更好的工作，我们事先需要估计每个任务的大小。即使任务大小已知，对于非均匀的任务，获得最佳映射方法通常也是NP难问题。
- 动态映射：任务是在程序执行的时候才分配到进程中的。这是因为任务可能是在程序运行时才产生的，或者它们的大小事先是不知道的。

其它影响映射方法的因素包括任务分配的数据大小还有底层域的性质。

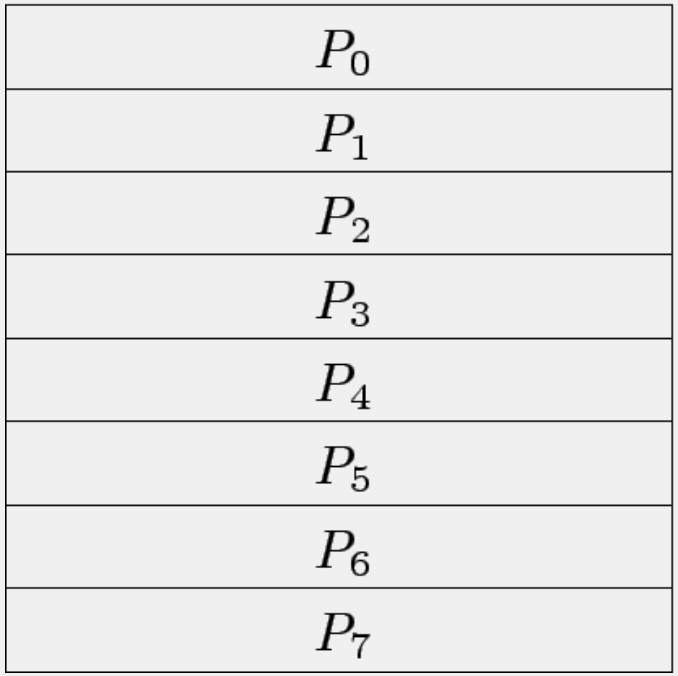
静态映射方案

- 以数据划分为基础的映射。
- 以任务图划分为基础的映射。
- 混合映射。

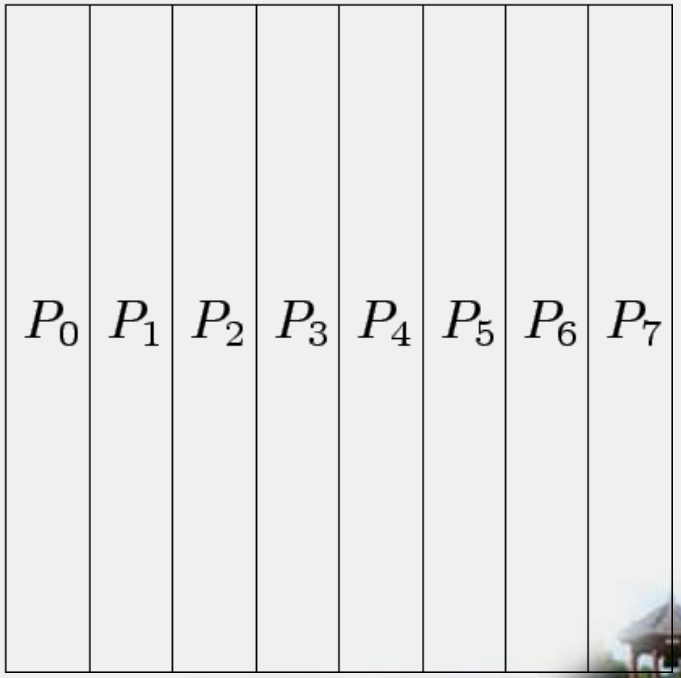
基于数据划分的映射

我们根据数据划分的“拥有者-计算”规则把计算划分为多个小任务。对于稠密矩阵的最简单的数据划分方案是把它们划分为一维的块，如下图所示：

row-wise distribution



column-wise distribution



块划分方案

块划分方案可以推广到高维的情况。

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

(a)

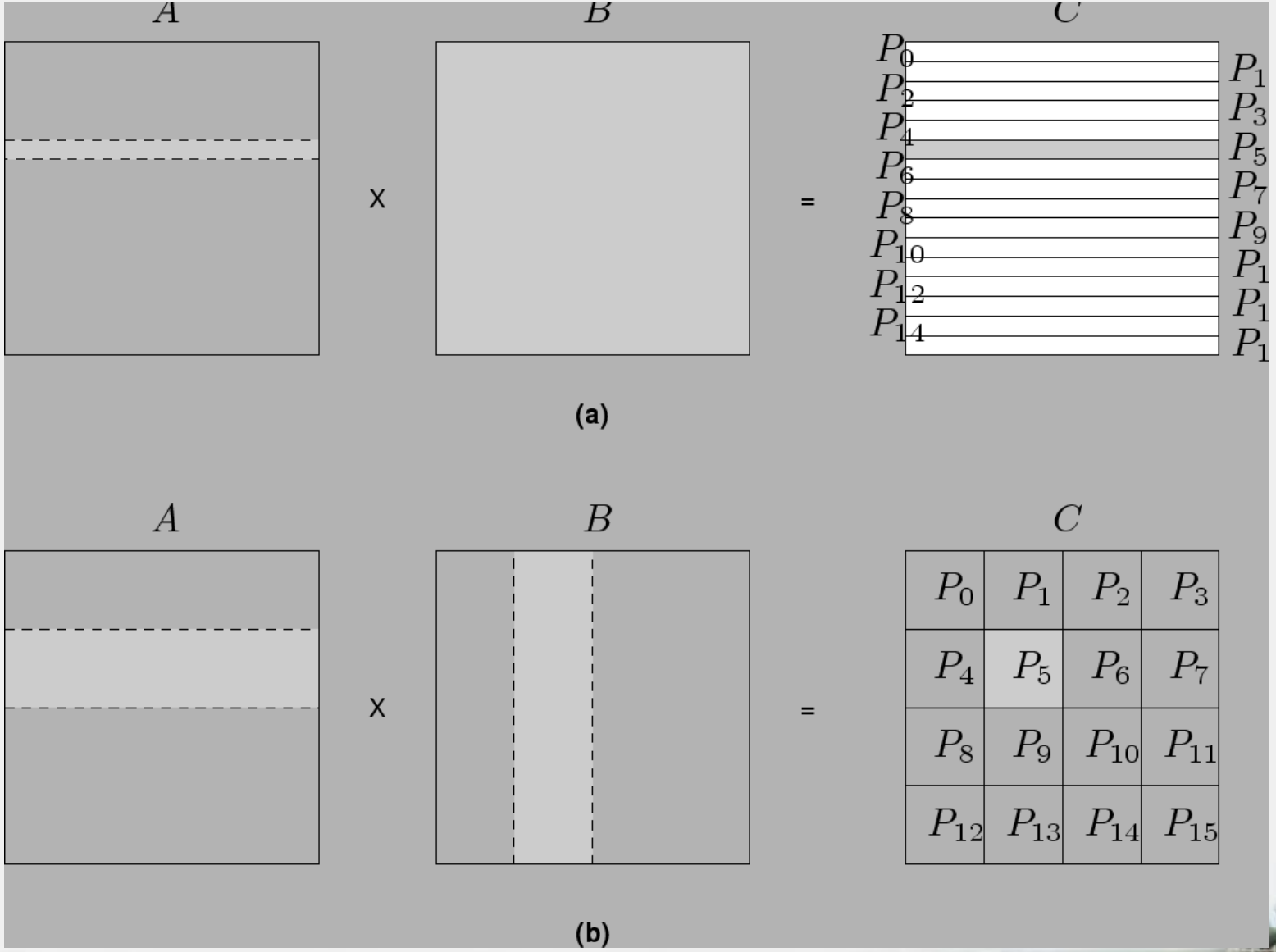
P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

块划分方案：例子

- 对于两个稠密矩阵A和B相乘的例子，我们可以用块划分方案把输出矩阵C进行划分。
- 为了负载平衡，我们分配给每一个任务C的同样数目的元素。（注意C的每个元素与一个点积运算相关）
- 一个精确的划分是由相关的通信开销来决定的。
- 总的来说，高维的分解可以允许使用更多数量的进程。

稠密矩阵相乘的数据共享



循环分配和快循环分配

- 如果一个矩阵不同元素的计算量不同，那么块分配将会引起负载不平衡。
- 一个简单是例子就是稠密矩阵的LU分解（或者高斯消去）。

稠密矩阵的LU分解

一个分解把LU分解这个问题分成了14个任务 - 注意其中任务出现的负载不平衡。

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} . \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$

2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$

3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$

4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$

5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$

6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$

7: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$

8: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$

9: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$

10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$

11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$

12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$

13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$

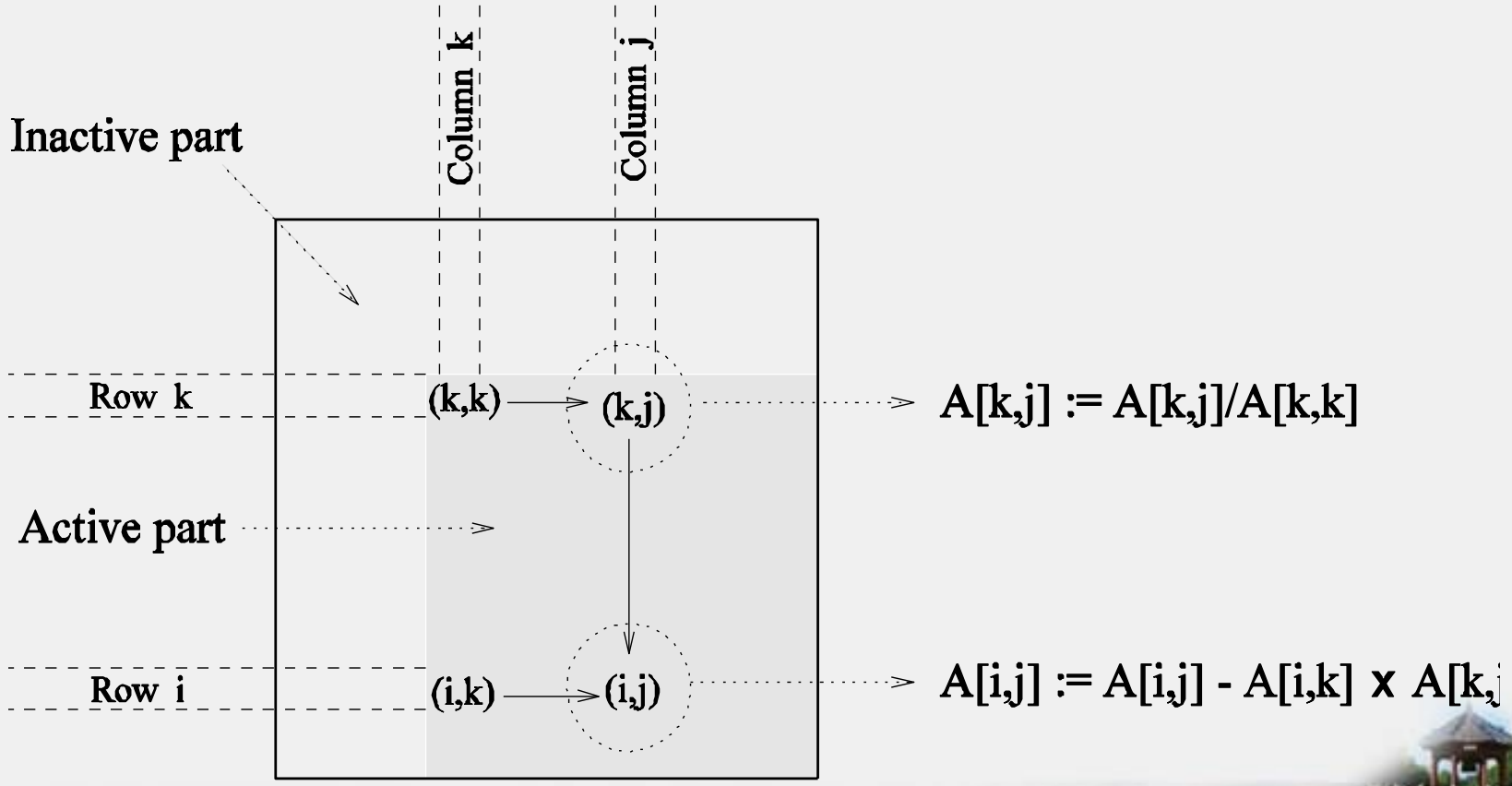
14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$

块循环分解

- 块分解方案的一个变化可以有效地减轻其中的负载不平衡和空闲问题。
- 分解一个阵列的块数要多于进程的个数。
- 块是用循环的方式来分配给进程，因此每个进程得到的块都是不相邻的。

高斯消去法的块循环分配

高斯消去法中矩阵参与计算的部分大小发生变化，朝右下角进行收缩。通过块循环的方法来分配矩阵块，每个进程获得了矩阵的不同部分的块。



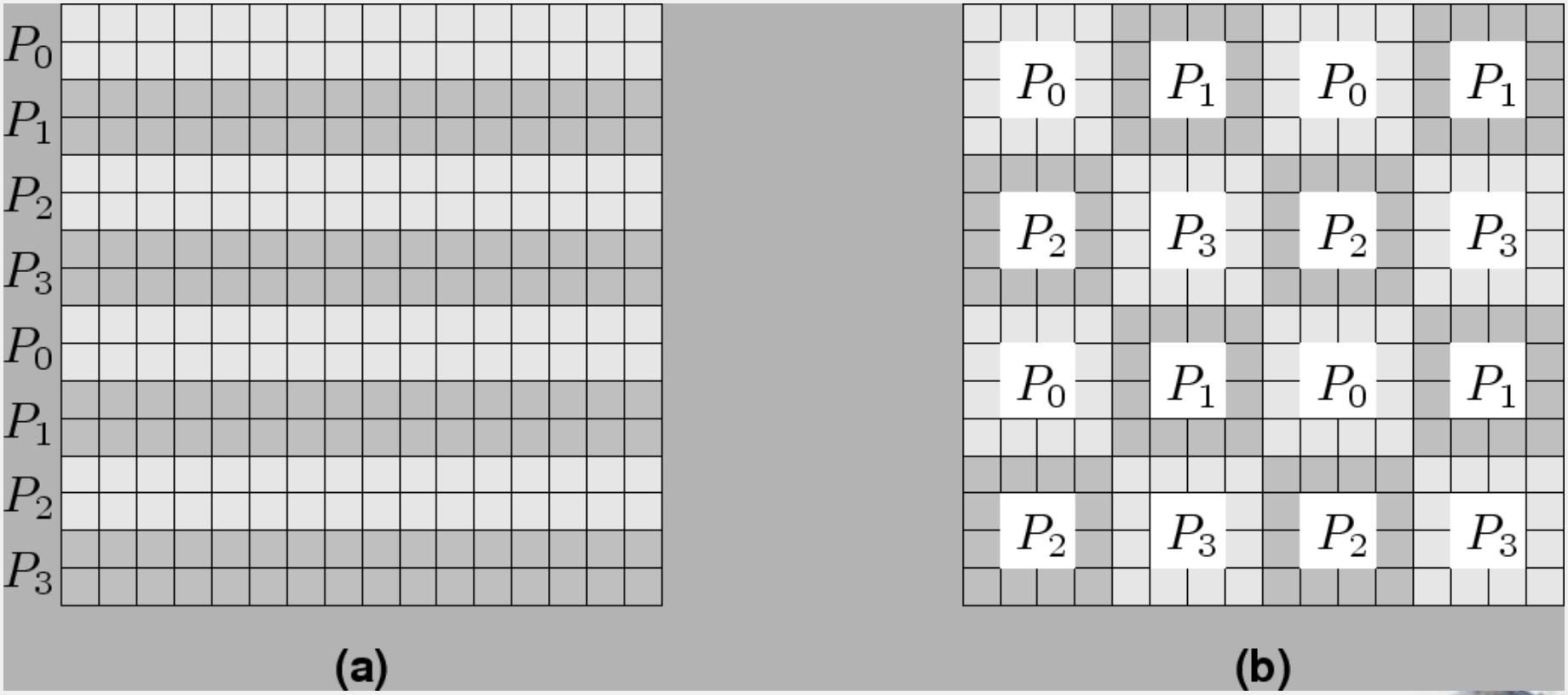
块循环分解：例子

- 一个二维块分配的LU分解任务到进程的自然映射

P_0 T_1	P_3 T_4	P_6 T_5
P_1 T_2	P_4 T_6 T_{10}	P_7 T_8 T_{12}
P_2 T_3	P_5 T_7 T_{11}	P_8 T_9 T_{13} T_{14}

块循环分解

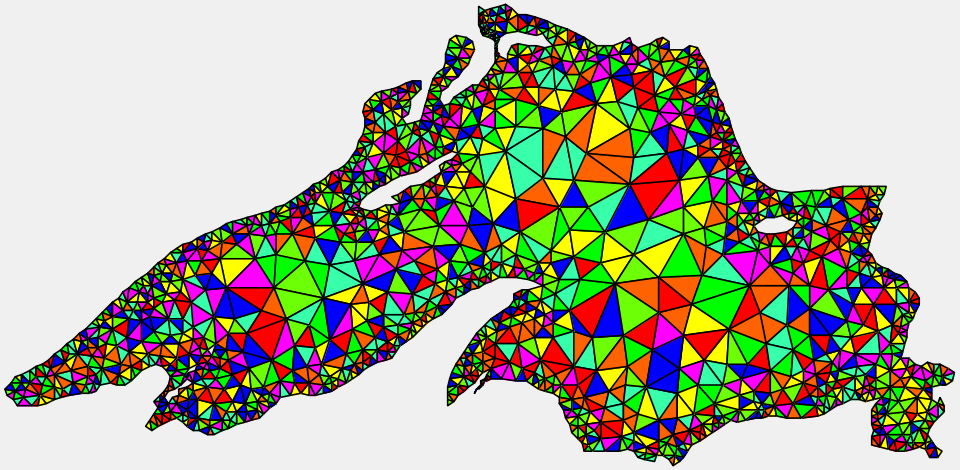
- 一种循环分解的特殊例子是每一块的大小为1个单位。
- 另外 一种循环分解的特殊例子是每一块的大小为n/p，其中n是矩阵的维数，p是进程的个数。



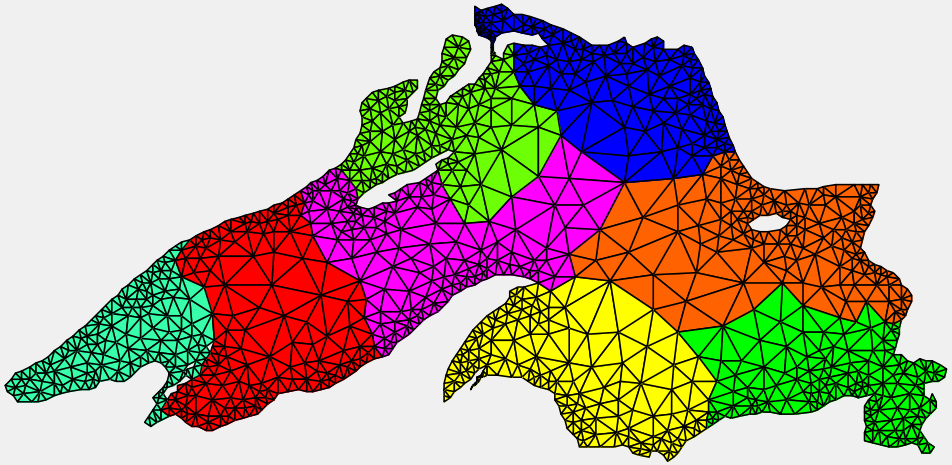
基于数据分解的图划分

- 在稀疏矩阵这个例子中，块分解的情况更加复杂。
- 考虑稀疏矩阵-向量乘的问题。
- 矩阵图是一个关于计算量（节点的个数）和通信量（节点的度）的一个有用的指示器。
- 在这种情况中，我们尽量给每一个进程分配同样个数的节点，并且减少不同进程之间的节点的边数。

苏必利尔湖的图划分



随机划分



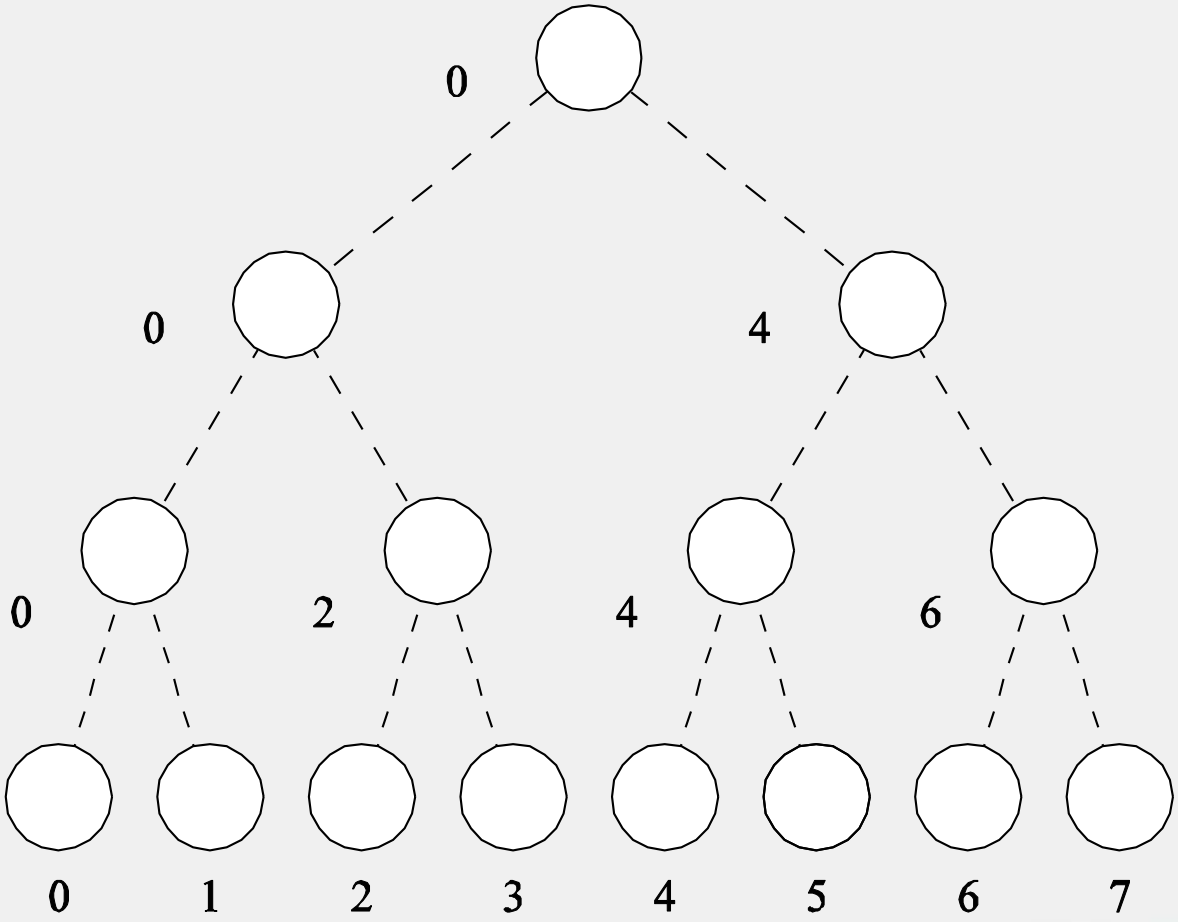
最小边切割划分

基于任务分解的划分

- 根据任务依赖图来进行任务划分。
- 根据任务依赖图来决定最优的映射是一个NP难问题。
- 对于一个有结构的图可以用启发式算法来计算。

任务分解：划分 一个二叉树依赖图

下图说明如何根据快速排序的任务依赖图来把任务划分到进程当中去。

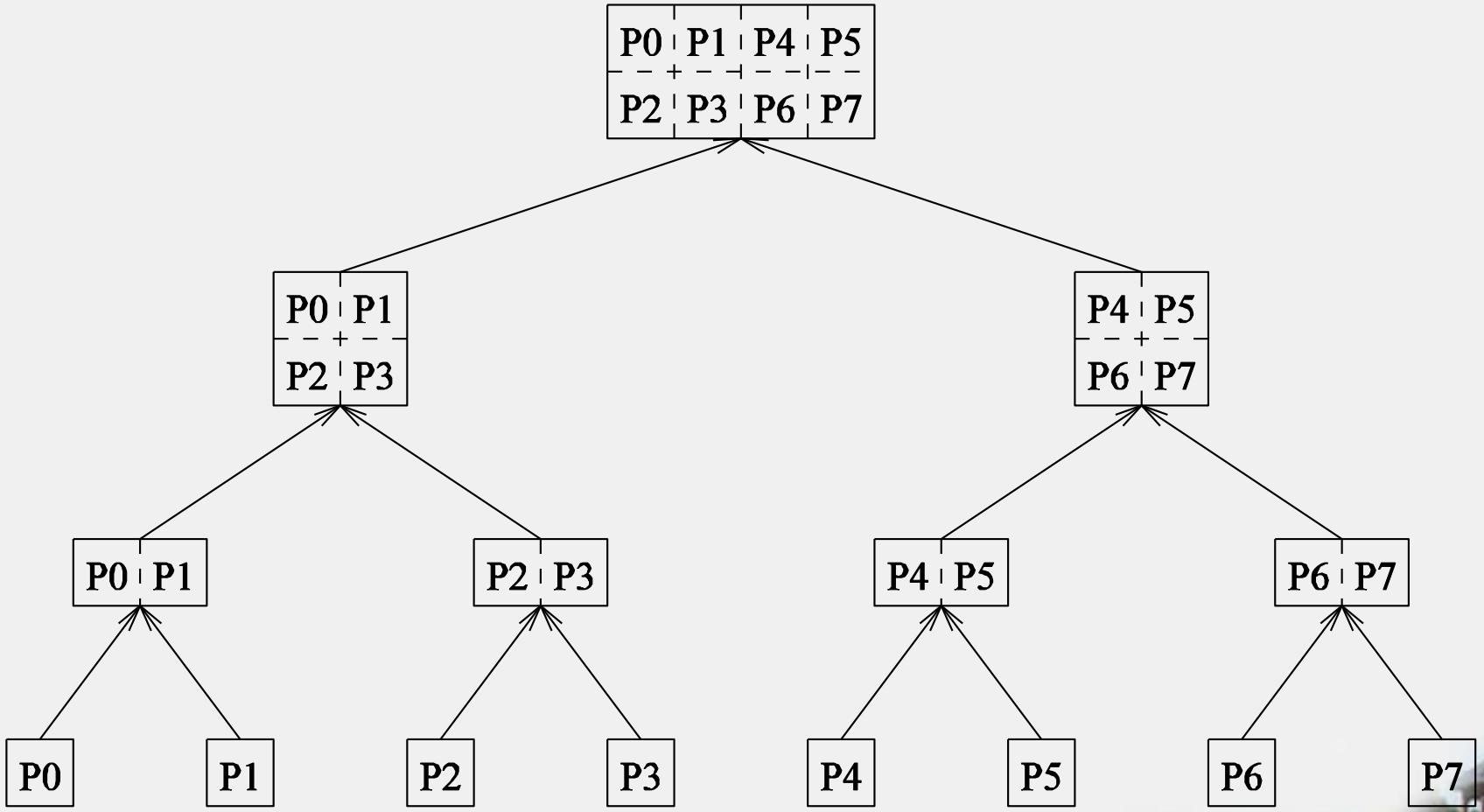


分层映射

- 有时候单一的映射方法效果并不好。
- 例如，一个二叉树（快排）的任务映射不可以使用大量的处理单元。
- 由于这个原因，在高层次上可以应用任务分解而在下一层次可以应用数据分解。

分层映射

一个在高层使用任务分解而在低层使用数据分解的例子。



动态映射方案

- 动态映射有时也被称为动态负载平衡，这是因为动态映射的主要目的是实现负载平衡。
- 动态映射方案有集中式和分布式方案。

集中式动态映射

- 进程被分为主进程和从进程。
- 当一个进程做完了当前工作，它可以向主进程请求另外的工作。
- 当进程的数量增多，主进程处则会出现拥堵。
- 为了缓和这种问题，一个进程一次得到多个任务。这叫做块调度。
- 当块太大时也会导致负载的不平衡。
- 一种方案是随着计算的进行块的大小逐渐减少。

分布式动态映射

- 每个进程可以发送任务给其它进程或者从其它进程接收任务。
- 这样可以减少集中式方案的传输瓶颈。
- 分布式方案有四个重要的问题：如何成对的发送和接收任务？由发送者还是接收者启动任务的传递？每次交换的任务量是多大？何时开始传递任务？
- 这些问题根据不同的情况有不同的答案。我们在后面将讨论一些解决方法。

- 减少交互开销的方法
- 并行算法设计模型

最小化交互开销

- 最大化数据本地性：对于可能使用的中间数据或者重复使用的数据，尽量放在需要使用的任务所在的进程里，尽量减少进程间的数据访问频率。
- 最小化数据交换量：每一次进行交互的数据量都会有开销，因此我们要尽量地减少交互的数据大小。
- 最小化交互的频率：每一次交互都需要启动时间，因此，尽量把不同的交互合起来变成一个。
- 减少争用和热点：使用分散数据的方法，并且有需要的时候可以拷贝多份数据。

最小化交互开销（续）

- 计算和交互重叠：使用无阻塞通信，多线程还有预取操作来达到目标。
- 复制数据或者计算。
- 用组通信来代替点对点通信。
- 让交互和其它交互堆叠。

并行算法模型

算法模型是指选择恰当的分解方法和映射方法并应用适合的策略来最小化交互，从而构建并行算法。

- 数据并行模型：任务分配到每个处理器上并且每个任务对不同的数据进行同样的操作。
- 任务图模型：从任务依赖图开始，任务之间的关系用来提高数据的本地化程度和减少交互开销。

并行算法模型（续）

- 主-从模型：一个或多个进程产生任务并把任务分配给其它工作的进程。这个分配可以是静态的也可以动态的。

- 流水线/生产者-消费者模型：一串数据流传过一系列的进程，这些进程将对这串数据流进行处理。

- 混合模型：混合的模型用在问题的不同层次中或者用在并行算法的不同阶段中。