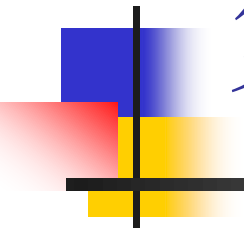
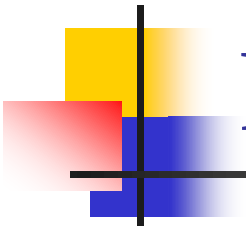


第2讲 并行硬件和并行软件

Parallel hardware And Parallel software



Roadmap

- Some background
- Modifications to the von Neumann model
- Parallel hardware
- Parallel software
- Input and output
- Performance
- Parallel program design
- Writing and running parallel programs
- Assumptions

PARALLEL SOFTWARE

——并行算法设计与分析

Flynn分类法

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

经典冯诺依曼架构

不作讨论

并行控制机制

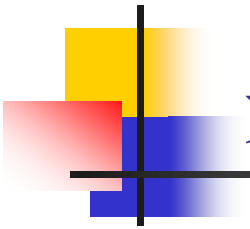
Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`

Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code:
		`if ($myid == 0) {`
		`}`

SPMD – single program multiple data

- SPMD 程序仅包含一段可执行代码，通过使用条件转移等语句，让该段代码表现的像在不同处理器上执行不同的程序。

```
if (I'm thread process i)
    do this;
else
    do that;
```



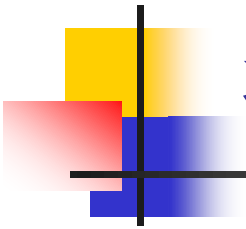
并行算法设计

- 假定已有求解问题的串行算法，我们将其改为并行版本
  - 并不一定是最好的策略——有些情况下，最优并行算法与最优串行算法完全没有关系
  - 但很有用，我们很熟悉串行算法，很多时候是切实可行的方法
- 并行算法与体系结构紧密相关！

并行算法设计

1. 进程/线程的任务分配
   - **(a) 负载均衡**，使得每个进程/线程获得大致相等的工作量
   - (b) 使得需要的**通信量尽量少**
2. 安排进程/线程之间的同步
3. 安排进程/线程之间的通信

```
double x[n], y[n];
…
for (i = 0; i < n; i++)
    x[i] += y[i];
```



提纲

- 并行算法设计
  - 任务分解:
    - 数据并行
    - 其他任务划分方法
  - 数据依赖、竞争条件
- 并行算法分析

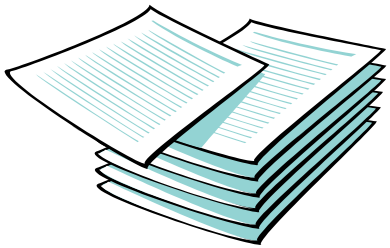
如何设计并行程序

- 任务并行
  - 将求解问题的计算分解为任务，分配给多个核心
- 数据并行
  - 将求解问题涉及的数据划分给多个核心
  - 每个核心对不同数据进行相似的计算

教授P批改试卷

15道题

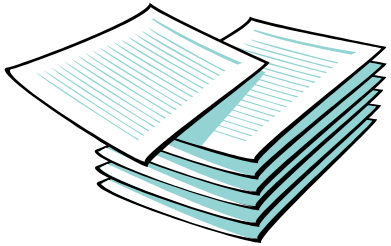
300份试卷



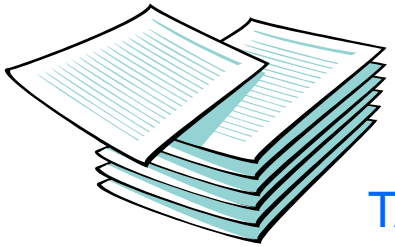
教授P有三位助教

工作分配——数据并行

TA#1

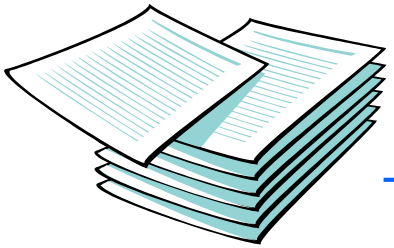


100 exams



TA#3

100 exams



TA#2

100 exams

工作分配——任务并行

TA#1

Questions 1 - 5

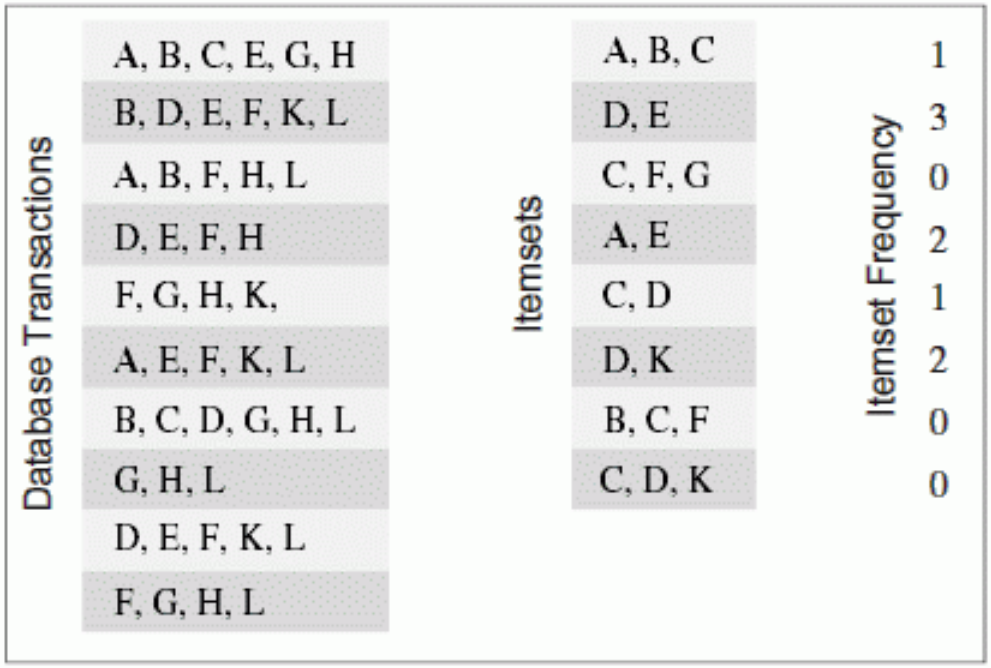
TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

商品购买频率统计

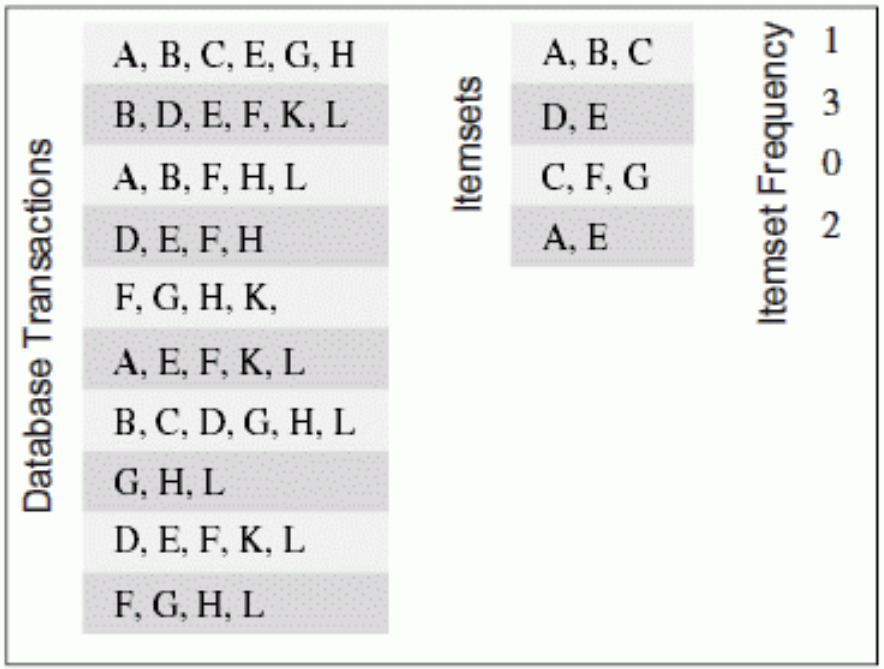


Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

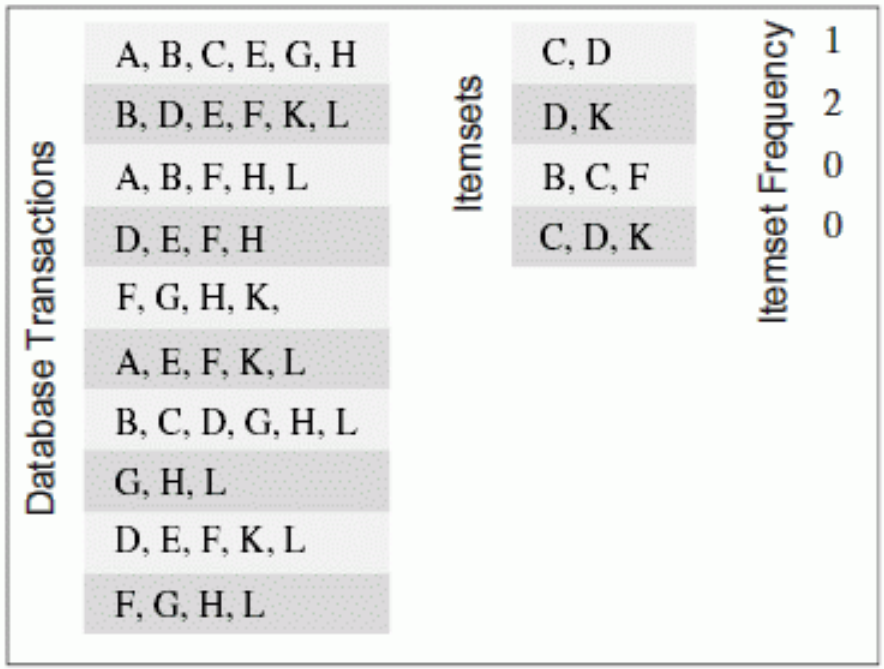
- 已知用户购物列表T，商品组合列表I➔查询商品组合的购买频率

划分输出数据(任务并行)

- 每种组合购买频率的计算是无关的
- I划分为若干部分，每部分的查询→任务

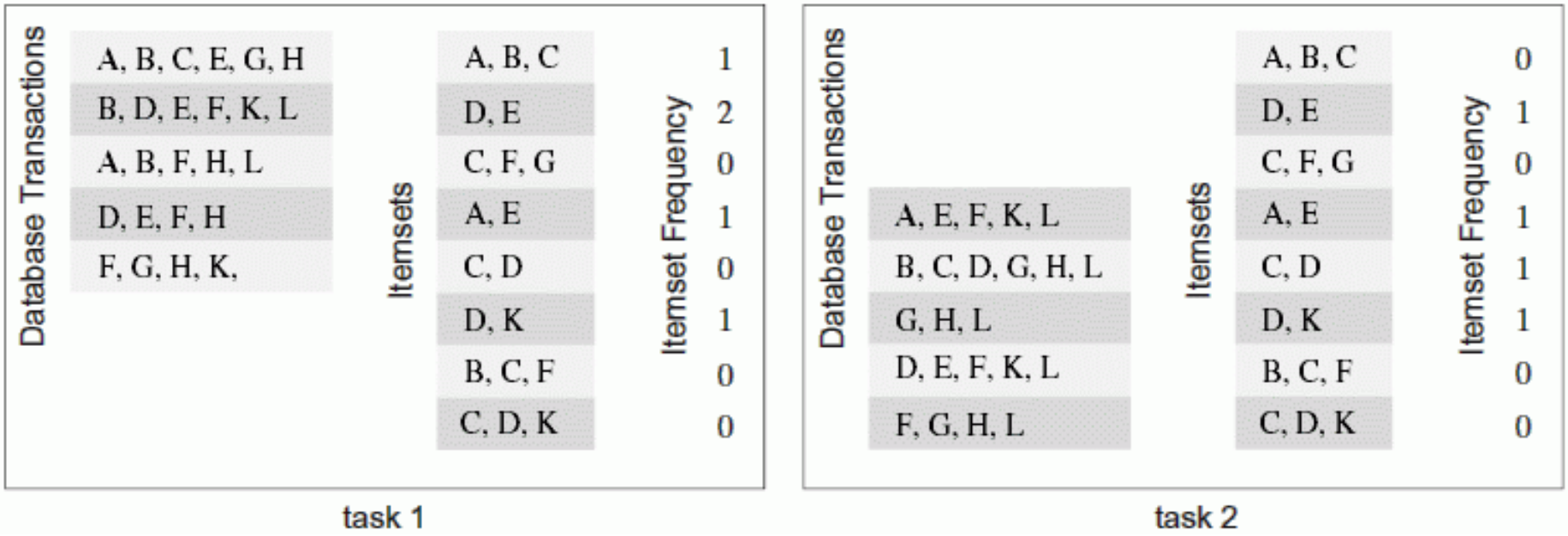


task 1



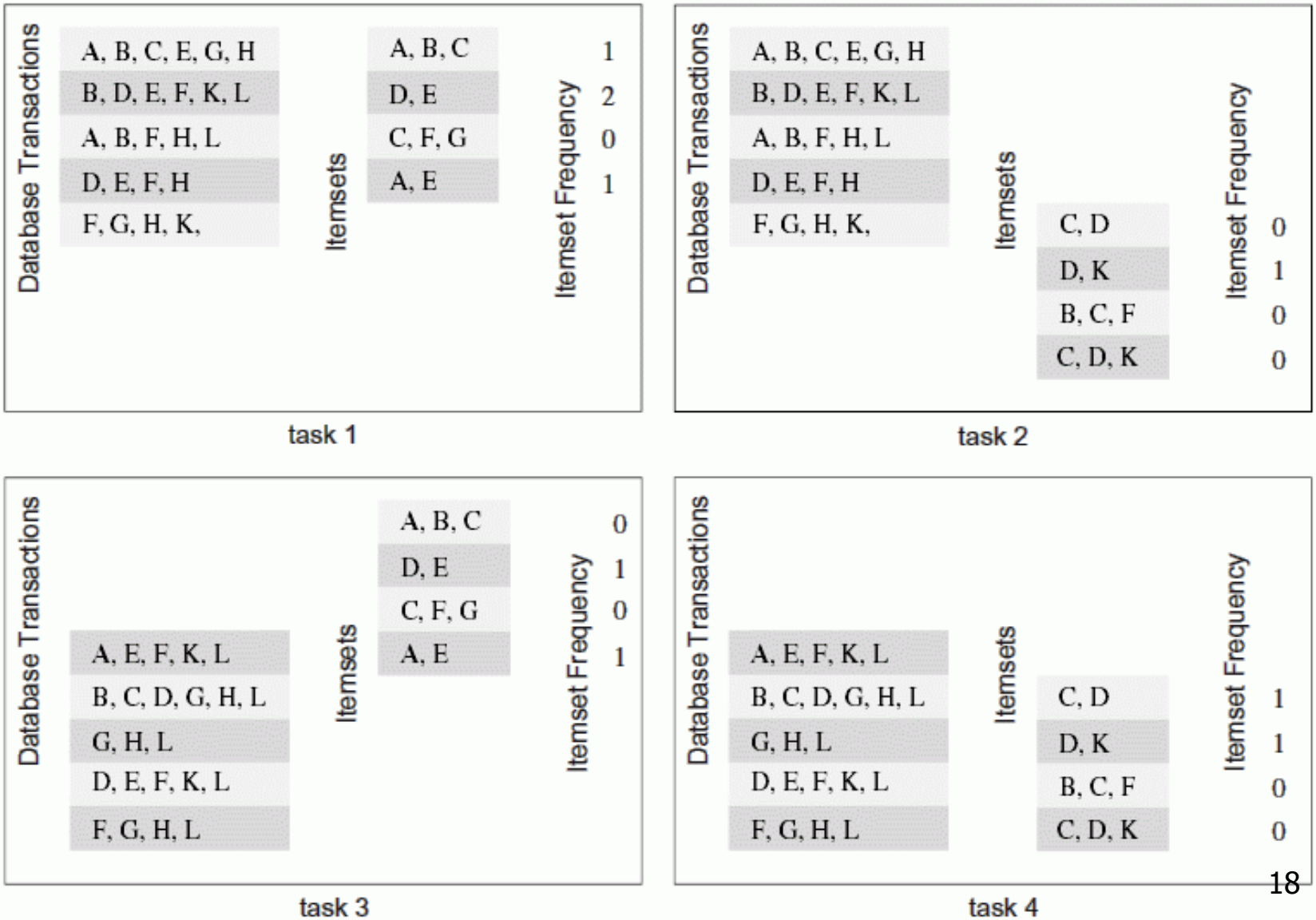
task 2

划分输入数据(数据并行)



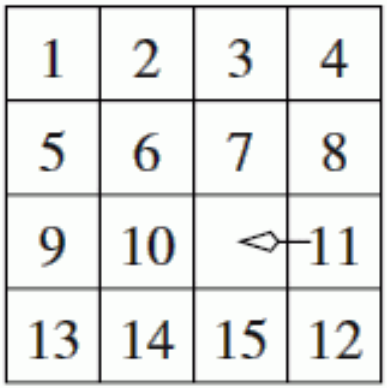
- 结果组合：两个频率列表需相加

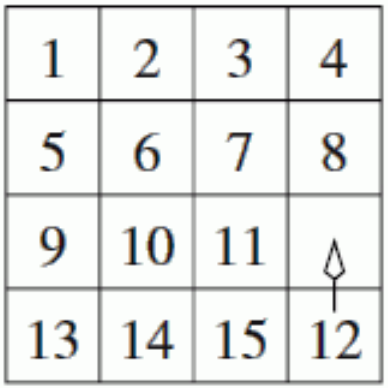
同时划分输入输出数据

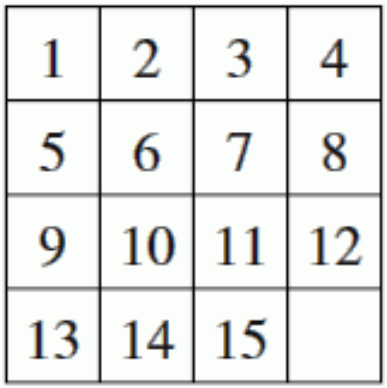

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,
Itemsets
A, B, C
D, E
C, F, G
A, E
Itemset Frequency
1
2
0
1
task 1
Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,
Itemsets
C, D
D, K
B, C, F
C, D, K
Itemset Frequency
0
1
0
0
task 2
Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L
Itemsets
A, B, C
D, E
C, F, G
A, E
Itemset Frequency
0
1
0
1
task 3
Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L
Itemsets
C, D
D, K
B, C, F
C, D, K
Itemset Frequency
1
1
0
0
task 4

搜索分解

- 15-数码问题





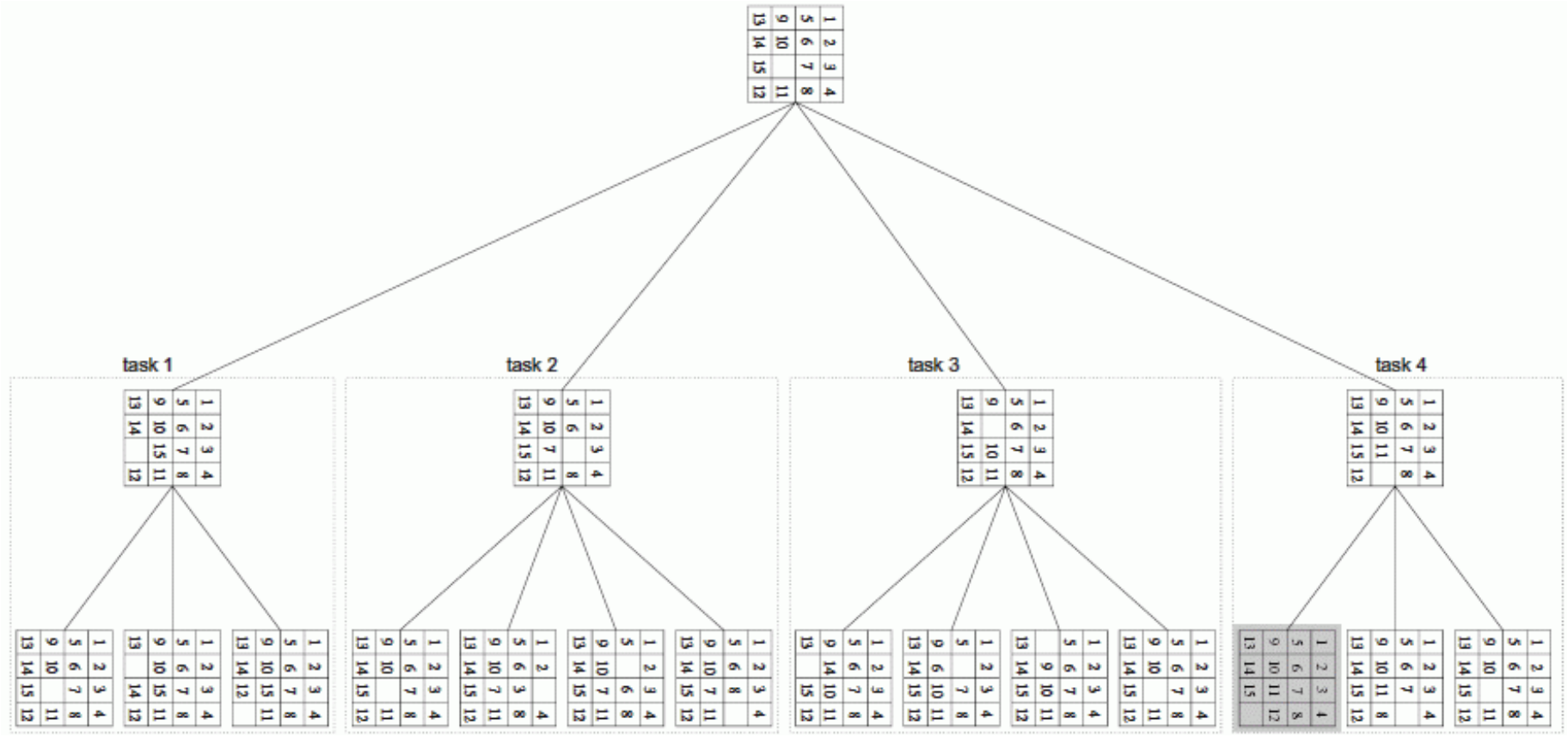


- 解法
  - 搜索空间组织为树结构
    - 根节点：初始格局
    - 节点A的孩子节点
      - 格局A的后继格局，2～4个
      - 格局A中空格与相邻位置交换的结果

15数码任务分解

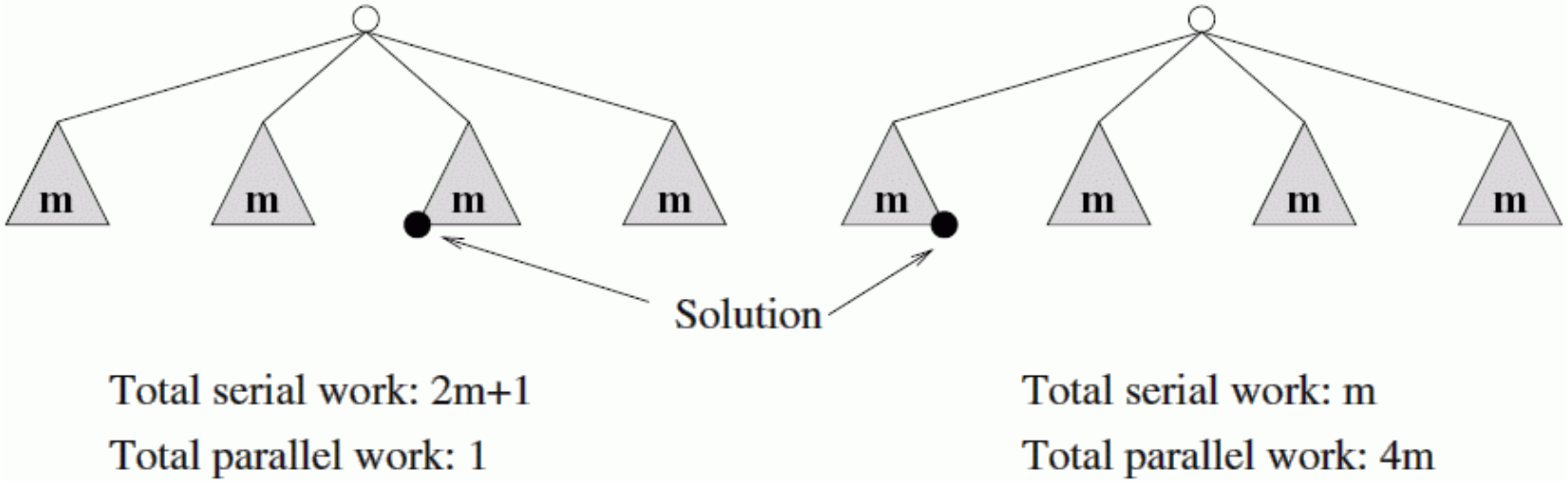
- 先串行生成一定规模的搜索树
- 任务分解——对得到的搜索树，以每个叶节点为根的子树的搜索工作作为一个任务
- 任务协调——某个任务（子树）找到解，应通知其他任务（子树）停止搜索

15数码任务分解图示



与数据分解的区别

- 数据分解：每个任务的计算工作的结果对最终结果都是有用的，都要全部做完
- 搜索分解：一旦一个任务找到解，全部任务即可停止，工作量可能大于，也可能小于串行算法



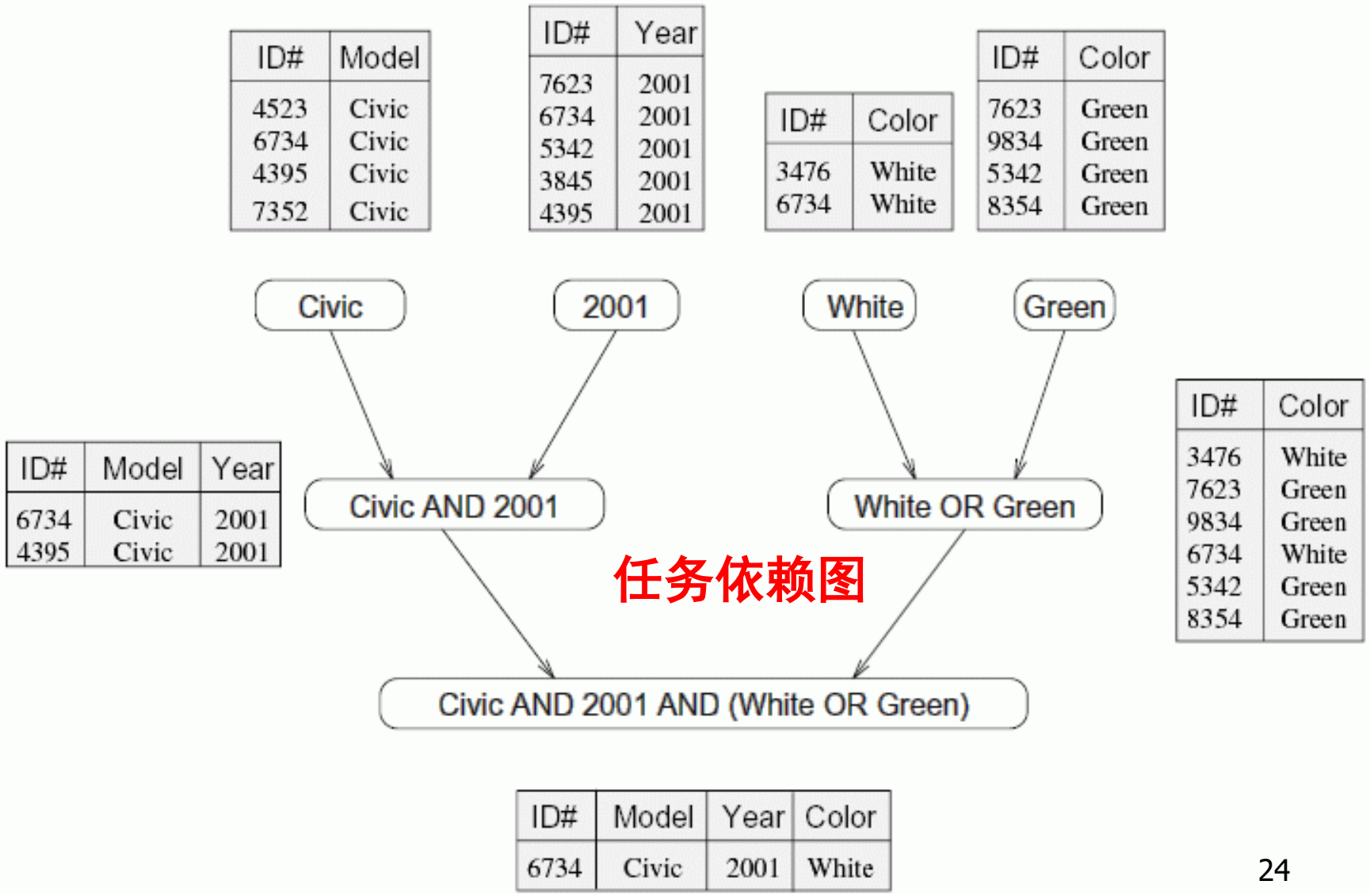
更多任务分解例：数据库查询

- 汽车数据库，进行复杂的组合查询：

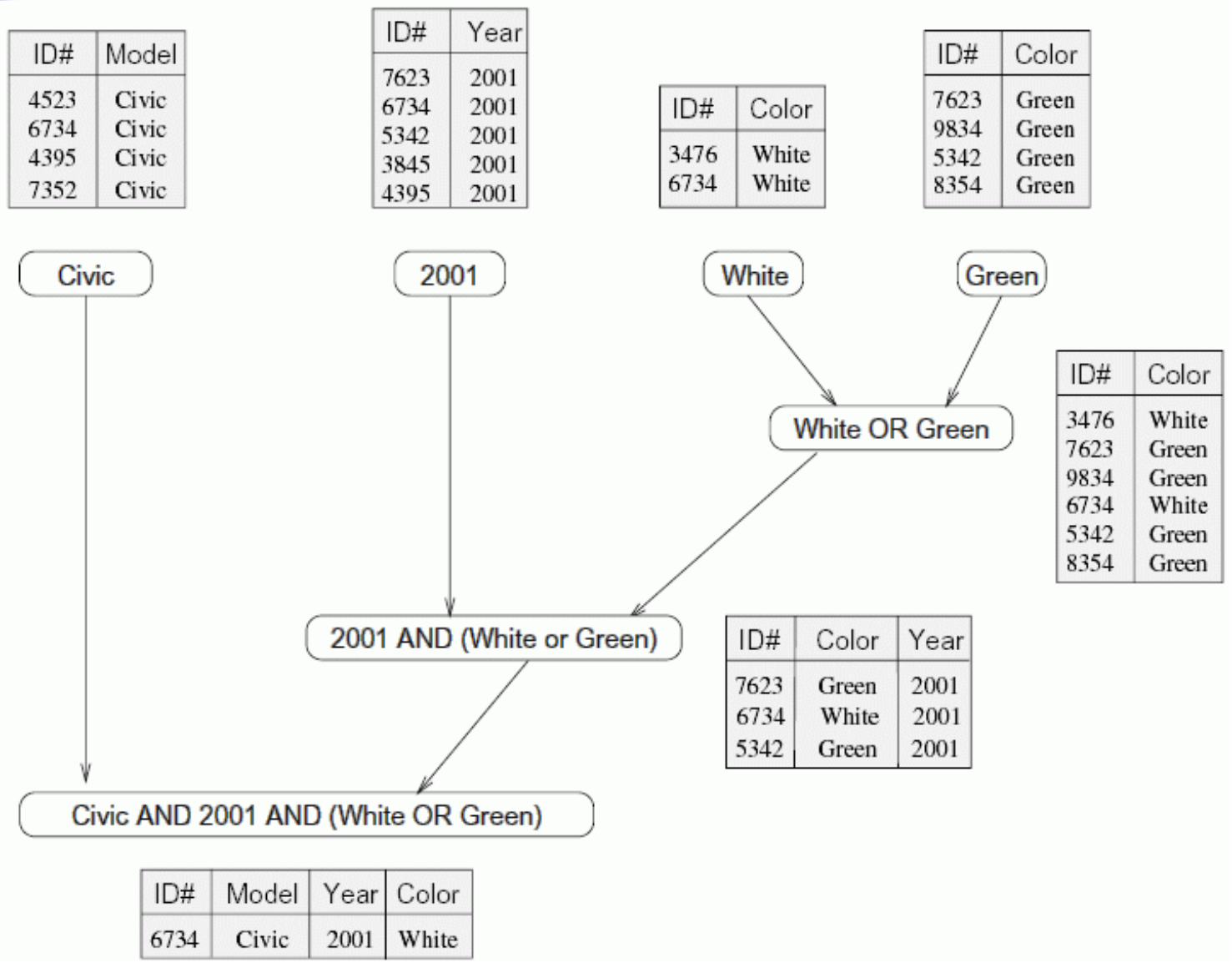
Model=“Civic” AND Year=“2001” AND
(Color=“Green” OR Color=“White”)

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

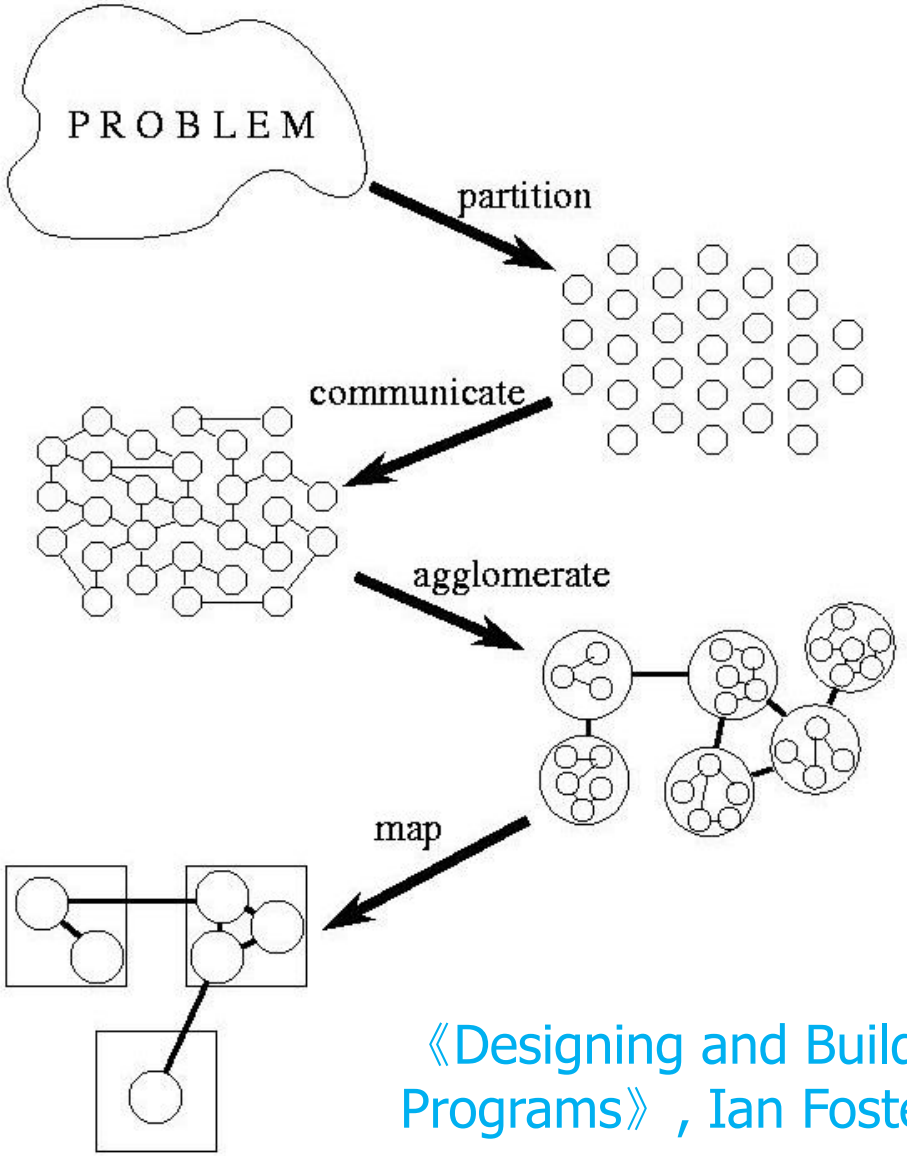
一种任务分解方法


ID# Model
4523 Civic
6734 Civic
4395 Civic
7352 Civic
ID# Year
7623 2001
6734 2001
5342 2001
3845 2001
4395 2001
ID# Color
3476 White
6734 White
ID# Color
7623 Green
9834 Green
5342 Green
8354 Green
Civic
2001
White
Green
ID# Model Year
6734 Civic 2001
4395 Civic 2001
Civic AND 2001
White OR Green
ID# Color
3476 White
7623 Green
9834 Green
6734 White
5342 Green
8354 Green
任务依赖图
Civic AND 2001 AND (White OR Green)
ID# Model Year Color
6734 Civic 2001 White

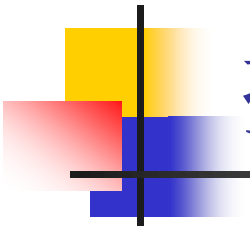
另一种任务分解


ID# Model
4523 Civic
6734 Civic
4395 Civic
7352 Civic
ID# Year
7623 2001
6734 2001
5342 2001
3845 2001
4395 2001
ID# Color
3476 White
6734 White
ID# Color
7623 Green
9834 Green
5342 Green
8354 Green
Civic
2001
White
Green
White OR Green
ID# Color
3476 White
7623 Green
9834 Green
6734 White
5342 Green
8354 Green
2001 AND (White or Green)
ID# Color Year
7623 Green 2001
6734 White 2001
5342 Green 2001
Civic AND 2001 AND (White OR Green)
ID# Model Year Color
6734 Civic 2001 White

Ian Foster的方法学



《Designing and Building Parallel Programs》, Ian Foster



提纲

- 并行算法设计
  - 任务分解:
    - 数据并行
    - 其他任务划分方法
  - 数据依赖、竞争条件
- 并行算法分析

一个简单的例子

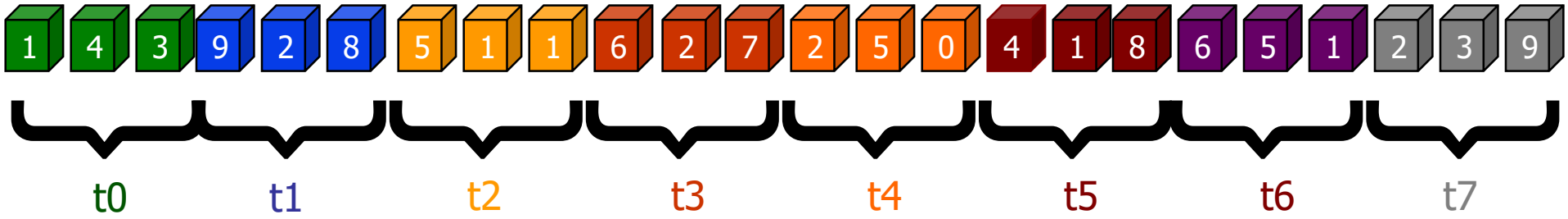
- 计算n个值并求它们的和
- 串行算法：

```
Int sum = 0;
for (i = 0; i < n; i++) {
  x = Compute next value(. . .);
  sum += x;
}
```

- 并行算法？

版本1：计算任务划分

- 假定每个核心计算连续n/t个元素的部分和（t为线程数或处理器数）
- 例子：n=24，t=8




```
int block_length_per_thread = n/t;
int start = id * block_length_per_thread;
Int sum = 0;
\*进入线程
for (i=start; i<start+block_length_per_thread; i++)  {
    x = Compute_next_value(…);
    sum += x;
}
```

发生了什么？

- 循环步之间的求和运算存在依赖→
  线程间依赖
  - 但可以重排顺序，因为加法运算满足结合律
- 取数-加法-存结果必须是原子操作，以保持结果与串行执行一致
- 定义
  - 原子性（atomicity）：一组操作要么全部执行要么全不执行，则称其是原子的。即，不会得到部分执行的结果。
  - 互斥（mutual exclusion）：任何时刻都只有一个线程在执行

竞争条件多种描述

- 执行结果依赖于两个或更多事件的**时序**，则存在**竞争条件**（race condition）
- 多个进程/线程尝试更新同一个共享资源时，结果可能是无法预测的，则存在**竞争条件**。
- 更一般地，当多个进程/线程都要访问共享变量或共享文件等共享资源时，如果至少其中一个访问是更新操作，那么这些访问就可能导致某种错误，称之存在**竞争条件**。

数据依赖与同步

- **数据依赖**（data dependence）就是两个内存操作的序，为了保证结果的正确性，必须保持这个序
- **同步**（synchronization）在时间上强制使各执行进程/线程在某一点必须互相等待，确保各进程/线程的正常顺序和对共享可写数据的正确访问

版本2：加锁

- 插入互斥（mutex），保证任何时刻只有一个线程读数-加法-存结果——原子操作

```
int block_length_per_thread = n/t;
mutex m;
int start = id * block_length_per_thread;
Int sum = 0;
for (i=start; i<start+block_length_per_thread; i++)  {
    my_x = Compute_next_value(…);
    mutex_lock(m);
    sum += my_x;
    mutex_unlock(m);
}
```

已是正确的。但够好吗？

版本3：粗粒度

- 在将局部和加到全局和时才加锁

```
int block_length_per_thread = n/t;
mutex m;
int sum=0;
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
    my_x = Compute_next_value(…);
    my_sum += my_x;
}
mutex_lock(m);
sum += my_sum;
mutex_unlock(m);
```

版本4：消除锁

- “主”线程完成部分和相加

```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
sum = 0
for (i=start; i<start+block_length_per_thread; i++)  {
    my_x = Compute_next_value(…);
    my_sum[id] += my_x;
}
if (id == 0) { // 主线程
   sum = my_sum[0];
   for (i=1; i<t; i++) sum += my_sum[i];
}
```

正确吗？为什么？

同步方法：障碍

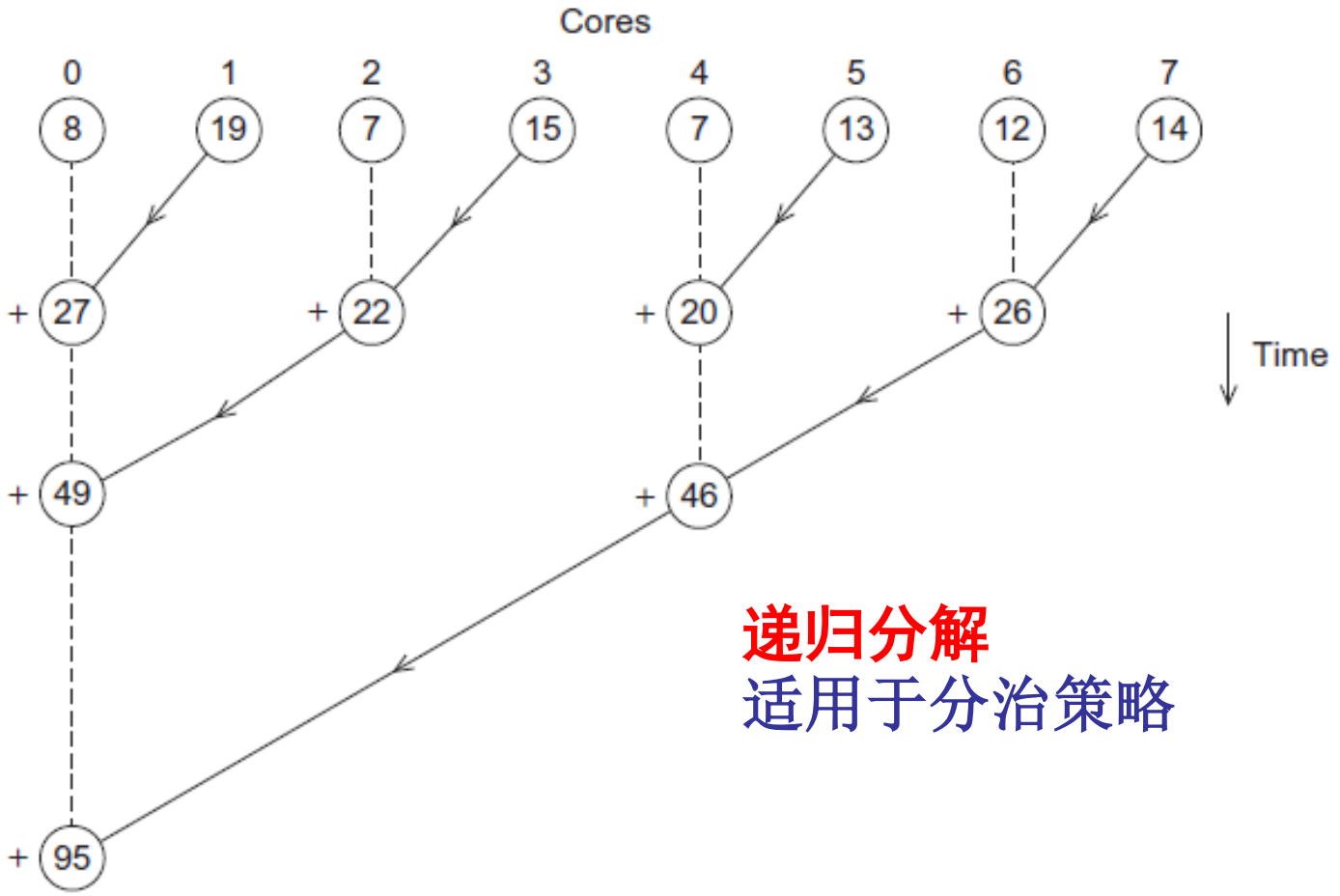
- 如果主线程开始计算全局和的时候其他线程还未完成计算，就会得到不正确的结果
- 如何强制主线程等待其他线程完成之后再进行全局和计算呢？
- 定义
  - **障碍**（barrier）阻塞线程继续执行，在此程序点等待，直到所有参与线程都到达障碍点才继续执行
  - 障碍如何实现？

版本5：消除锁，但增加障碍

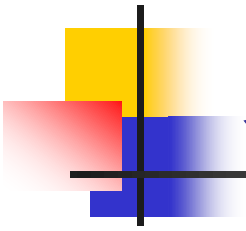
```
int block_length_per_thread = n/t;
mutex m;
shared my_sum[t];
int start = id * block_length_per_thread;
for (i=start; i<start+block_length_per_thread; i++)  {
    my_x = Compute_next_value(…);
    my_sum[t] += x;
}
Synchronize_cores(); // 所有参与线程都设置障碍
if (id == 0) { // 主线程
  sum = my_sum[0];
  for (i=1; i<t; i++) sum += my_sum[t];
}
```

现在正确了！

版本6：多核并行求全局和

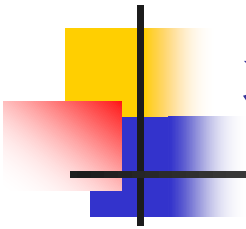


递归分解
适用于分治策略



求和例子的总结

- 求和计算有竞争条件和数据依赖
- 使用mutex和障碍进行同步保证正确结果
- 更多地进行本地运算，以提高线程间并行计算的粒度
- 在这个例子中看到了哪些额外开销？
  - 分配计算任务的额外代码
  - 锁开销：加锁/解锁操作本身开销和线程间竞争导致的空闲等待
  - 负载不均

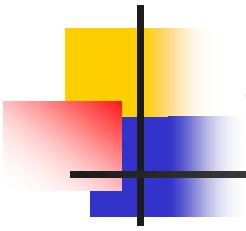


提纲

- 并行算法设计
  - 任务分解:
    - 数据并行
    - 其他任务划分方法
  - 数据依赖、竞争条件
- 并行算法分析
  - 基本指标
  - 可扩展性

并行算法分析

- 串行算法评价：算法时间复杂度表示为输入规模的函数
- 并行算法评价：除了输入规模之外，还应考虑处理器数目、处理器相对运算速度、通信速度
- 评价标准
  1. 运行时间
  2. 加速比：并行算法比串行算法快多少？——问题：很多串行算法，选哪一个？



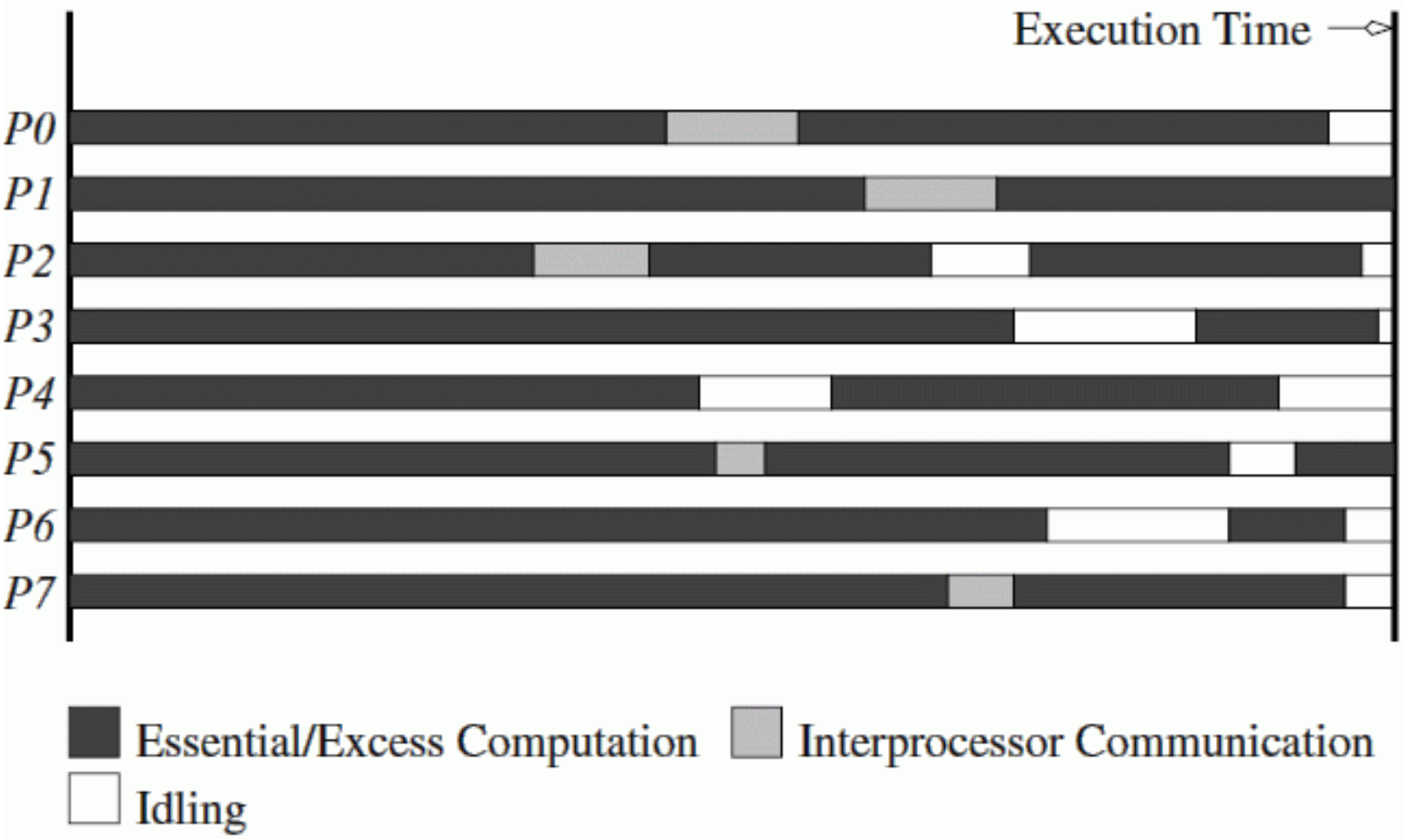
并行程序设计的复杂性

- 足够的并发度（Amdahl定律）
- 并发粒度
  - 独立的计算任务的大小
- 局部性
  - 对临近的数据进行计算
- 负载均衡
  - 处理器的工作量相近
- 协调和同步
  - 谁负责？处理频率？

并行算法额外开销

- 除了串行算法要做的之外的工作
  - 进程间通信：最大开销，大部分并行算法都需要
  - 进程空闲：负载不均、同步操作、不能并行化的部分
  - 额外计算
    - 最优串行算法难以并行化，将很差的串行算法并行化，并行算法计算量>最优串行算法
    - 最优串行算法并行化也会产生额外计算：并行快速傅立叶变换，旋转因子的重复计算

并行算法额外开销（续）


Execution Time
P0
P1
P2
P3
P4
P5
P6
P7
Essential/Excess Computation
Interprocessor Communication
Idling

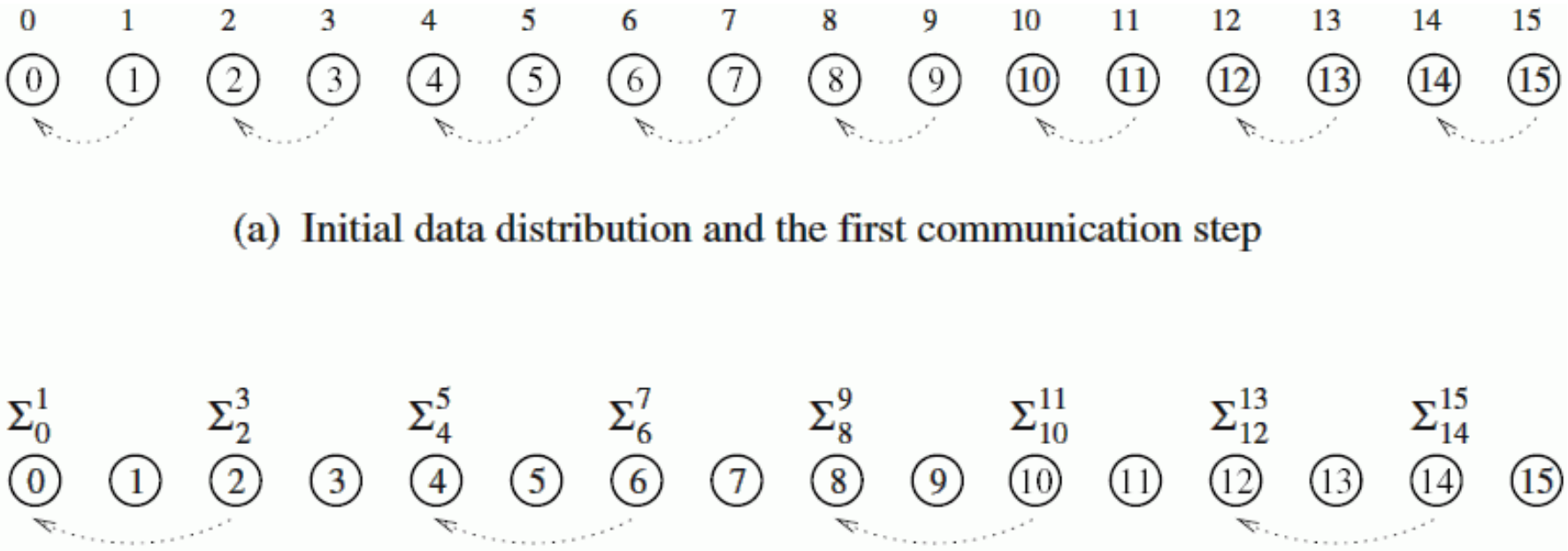
性能评价标准

- 运行时间
  - 串行算法：T_S，算法开始到结束的时间流逝
  - 并行算法：T_P，并行算法开始到最后一个进程结束所经历时间
- 并行算法总额外开销
  - $T_o = pT_P - T_S$
- 加速比
  - $S = T_S / T_P$

加速比

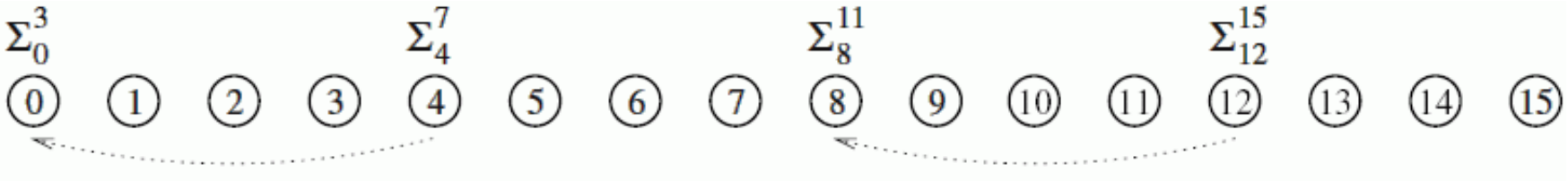
- 对比哪个串行算法？同一问题，可能存在多个串行算法，时间复杂度和并行程度可能都不一样
- 应选择“最优”串行算法
- 理论最优算法若不存在或难实现——选择已有（且可行）的算法中最优者
- S=最优串行算法时间/并行算法时间
  ——并行算法运行于p个处理器的并行平台，每个处理器与运行串行算法的处理器完全相同

例1：n个数相加，n个进程

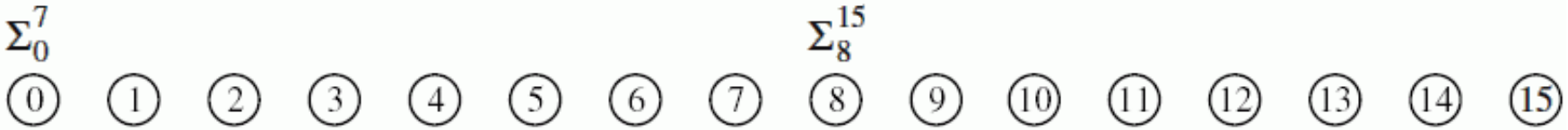


(a) Initial data distribution and the first communication step

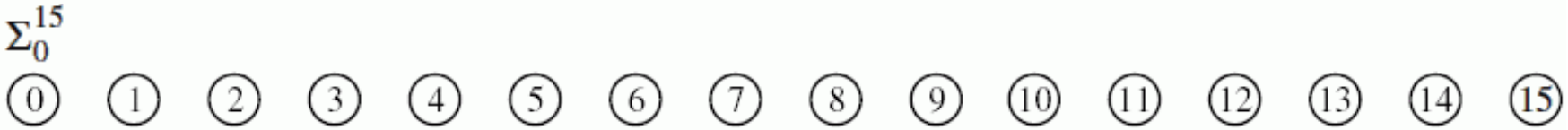
例1：（续）



(c) Third communication step



(d) Fourth communication step



例1：（续）

- 初始，每个进程保存1个数；
  最终由1个进程保存累加和
- 树形结构，logn个步骤
  每个步骤进行一次加法：t_c
  和一个机器字的传输：t_s+t_w
  ➔$T_P=\Theta(\log n)$
  ➔$S=\Theta(n/\log n)$

例2：加速比的计算

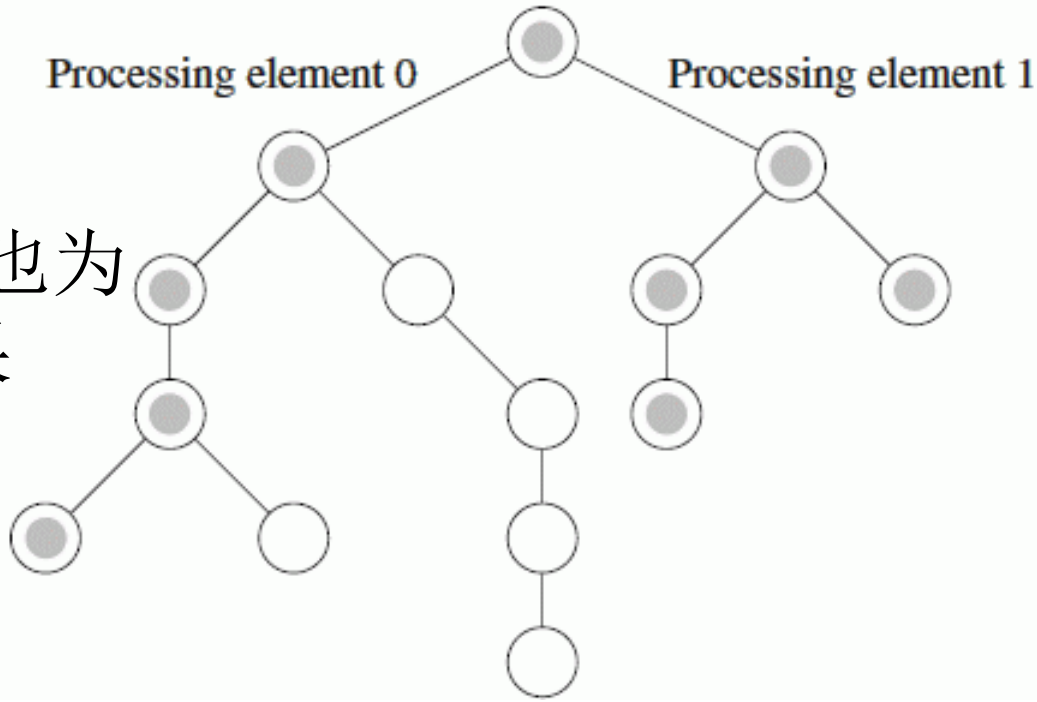
- 串行起泡排序算法时间150s，串行快速排序算法30s，并行起泡排序算法40s
- S=30/40=0.75，而不是150/40=3.75！
- 一般S≤p
- S=p, 则称该并行算法具有线性加速比
- S>p (超线性加速比)在实践中是可能出现的
  - 串行算法计算量>并行算法
  - 硬件问题不利于串行算法
    - 数据量较大，无法全部放入cache，命中率低
    - 并行算法进行数据划分，每个处理器数据量变小，可全部放入cache，命中率提高→性能提高

cache引起的超线性加速

- 2个处理器的并行系统，问题规模W
- 每个处理器由cache 64KB，命中率80%，cache延迟2ns，DRAM 100ns，平均访问时间2*0.8+100*0.2=21.6ns
- 若计算瓶颈在内存，1个内存访问可产生1个FLO，运算速度为46.3MFLOPS
- 将任务平均分配给两个进程，规模W/2，命中率90%，剩余8%为本地DRAM访问，2%为远端DRAM访问（400ns），平均访问时间2*0.9+100*0.08+400*0.02=17.8ns➔56.18MFLOPS➔2个处理器112.36MFLOPS➔S=2.43

搜索分解导致超线性

- 解在树的最右节点，串行算法需搜索所有节点：$14t_c$
- 并行算法：P_0搜索左子树，P_1搜索右子树，若速度相等，只搜索9个节点，由于并行，时间为$4t_c$，加上根节点处理时间，共$5t_c$，并行算法工作量少
- $S=14t_c/5t_c=2.8$
- 若串行算法交替处理两个子树节点，工作量也为9个节点，无超线性效果
- 但解的位置完全取决于问题实例，无普适的“最优串行算法”



阿姆达尔定律(Amdahl's law)

- 除非一个串行程序的执行几乎全部都并行化，否则不论多少可以利用的核，通过并行化所产生的加速比都会是受限的。

- $S = 1 / (1 - a + a / p)$
  - a为串行程序中可被(完美)并行化的比例
  - $T_S = 1$ ，
  - $T_P = T_{不可并行} + T_{可并行} = 1-a + a/p$

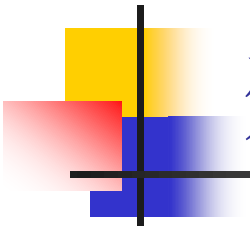
例3：

- 某串行程序运行时间为20s，可并行化比例为90%，若使用p个核将其并行化，加速比为多少？
  - $T_S = 20\ s$
  - $T_P = T_{不可并行} + T_{可并行}$

    $= 0.1 \times T_S + 0.9 \times T_S / p$

    $= 0.1 \times 20s + 0.9 \times 20s / p$

    $= 2s + 18s / p$
  - $S = T_S / T_P = 20 / (2 + 18 / p) = 10 / (1 + 9 / p)$
  - 或用公式：$S = 1 / (1 - a + a / p) = 1 / (0.1 + 0.9 / p)$



效率

- 效率（Efficiency）：度量有效计算时间
- $E = S / p = T_S / (p*T_P)$
- 理想情况=1，正常0～1。
  - 因为理想情况 S = p

效率

- 例1，n个核对n个数求和例子
  - $T_S = \Theta(n)$, $T_P = \Theta(\log n)$, $S = \Theta(n/\log n)$
  - $E = S/p = \Theta(n/\log n) / n = \Theta(1/\log n)$
- 例2，串行起泡排序算法时间150s，串行快速排序算法30s，使用3个核的并行起泡排序算法40s
  - $S = T_S / T_P = 30s/40s = 0.75$
  - $E = S/p = 0.75/3 = 0.25$

可扩展性（scalability）

- 若某并行程序核数(线程数/进程数)固定，并且输入规模也是固定的，其效率值为E。现增加程序核数(线程数/进程数)，如果在输入规模也以相应增长率增加的情况下，该程序的效率一直是E(不降)，则称该程序是**可扩展的**。
  - 我们希望，保持问题规模不变时，效率不随着线程数的增大而降低，则称程序是可扩展的（称为**强可扩展的**）。但这往往是难达到的。
  - 退求其次：问题规模以一定速率增大，效率不随着线程数的增大而降低，则认为程序是可扩展的（称为**弱可扩展的**）。

例4：并行矩阵-向量乘法

表 3-6　并行矩阵－向量乘法的加速比

comm_sz	矩阵的秩				
	1024	2048	4096	8192	16 384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

表 3-7　并行矩阵－向量乘法的效率

comm_sz	矩阵的秩				
	1024	2048	4096	8192	16 384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

- 该程序是弱可扩展的

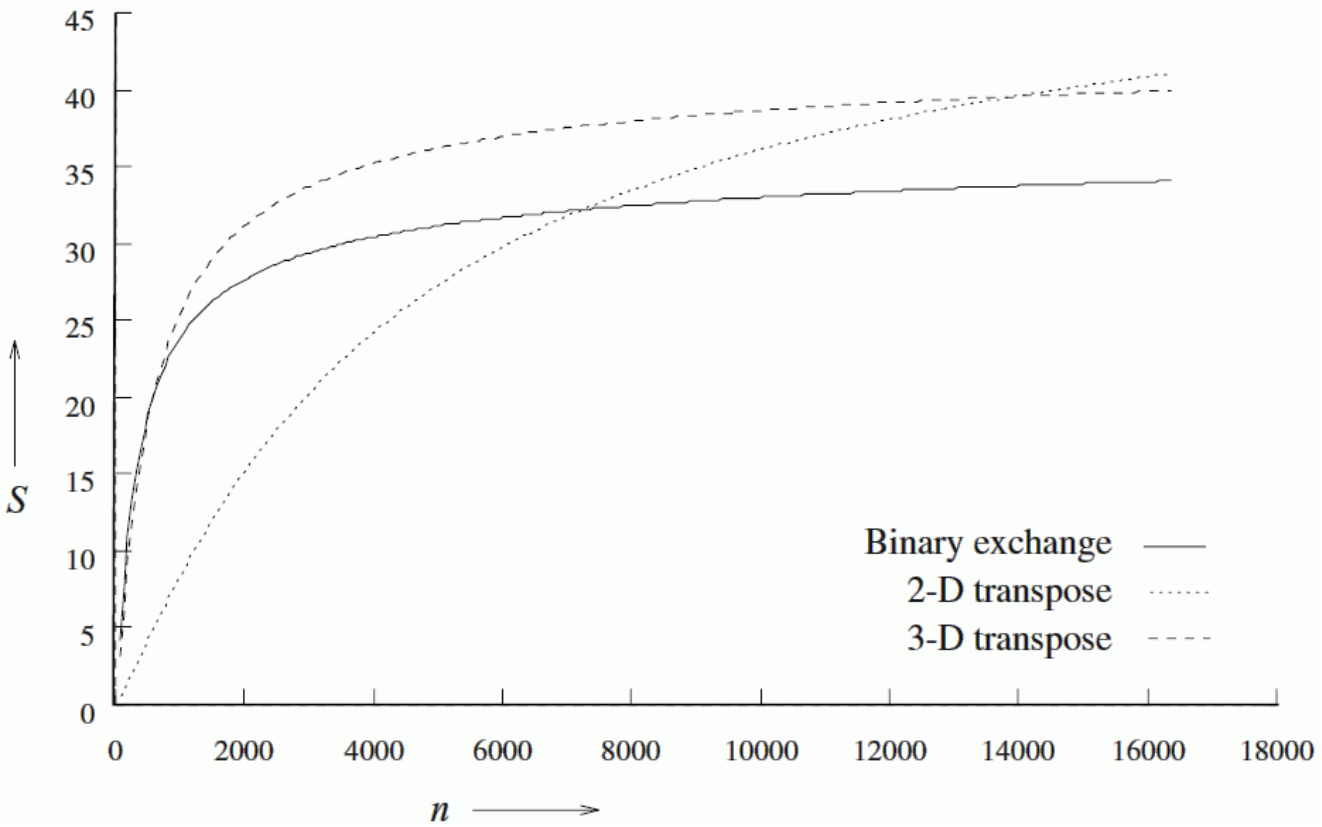
例5：

- 某程序串行版本运行时间为T_S = n秒, 这里n也为问题规模, 该程序某一并行版本运行时间为$T_P = n / p + T_O$。该并行程序是否可扩展？（假设T_O为常数，不随p变化而变化）
  - $E = T_S / p*T_P = \frac{n}{p*(n/p + TO)} = \frac{1}{1+\frac{p}{n}*\frac{T_o}{1}}$
  - 要保持E不变，当 p’ = kp 时，只需问题规模等比例增大即可，即 n’ = kn。
  - 因此，该并行程序时可扩展的。

可扩展性

- 可扩展性是高性能并行机和并行算法追求的主要目标，其主要作用：
  - 度量并行系统性能的方法之一
  - 度量并行体系结构在不同系统规模下的并行处理能力
  - 度量并行算法内在的并行性
  - 利用系统规模和问题规模已知的并行系统性能来预测规模增大后的性能：scale down, 适合开发、调试,不适合性能预测

例5：快速傅里叶变换



- 直接测量时间，小规模和大规模时，不同算法性能对比结果不一致
- 需要进行可扩展性的定量分析