第二讲 并行算法

- 1.并行算法的基础知识
- 2.并行计算模型
- 3.并行化方法
- 4.重点介绍：PRAM模型以及实例

第二讲 并行算法

重点介绍 1.本讲提供对Von Neumann模型和串行算法的一种（a mental break）智力突破。我们的工具是PRAM并行计算模型，PRAM模型允许并行算法设计者把处理机能当作一种无限资源，如同具有虚拟存贮器的计算机允许程序员把存贮器当作无限资源一样。PRAM模型是不真实地简单的。它不考虑处理机间的通讯复杂性。因为通讯复杂性不是一个问题，这使PRAM算法设计者能把主要精力放在开发一个计算中固有的并行性上。

第二讲 并行算法

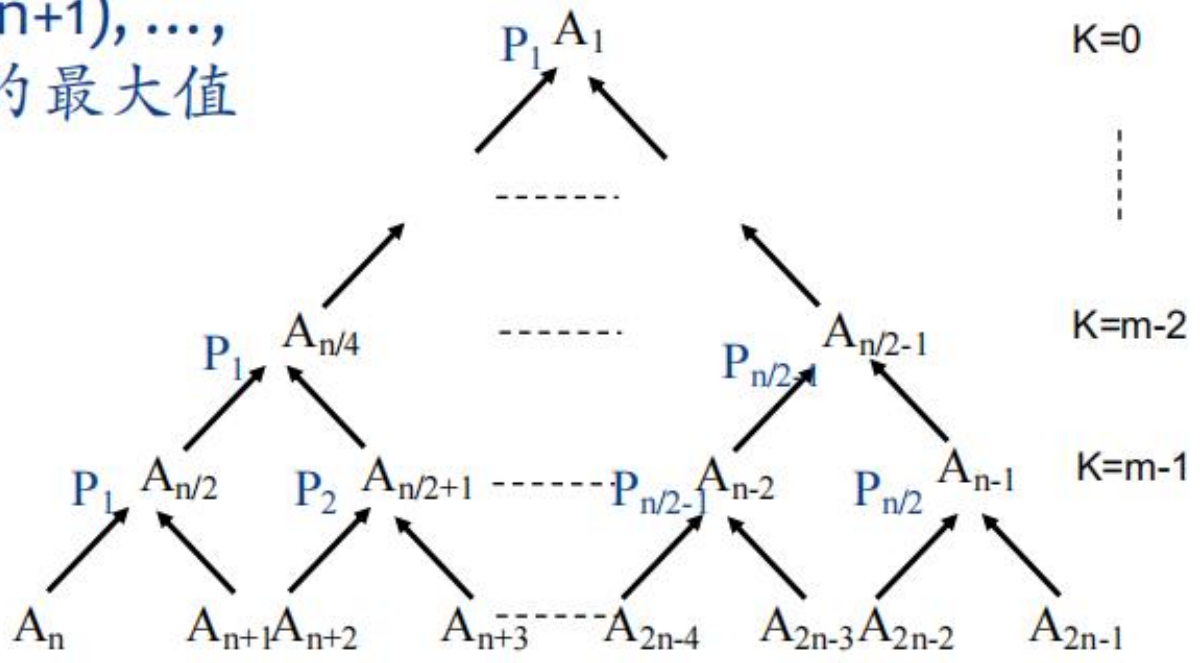
重点介绍：2. 对于某些重要的算法，诸如归约（reduction），成本最优（cost-optimal）的PRAM解答是存在的，意思是由PRAM算法执行的操作总量与最优串行算法的复杂性同阶。

并行算法的设计基础

1.并行算法的基础知识

求取最大值

* 令n=2^m，A是一个1维的数组，待求最大值的n个数开始存放在A(n), A(n+1), …, A(2n-1)，所求得的最大值置于A(1)中。



求最大值

* 算法4.1：SIMD-TC(SM)上求最大值算法

输入： $n=2^m$ 个数存放在A(n,2n-1)中；

输出： 求得的最大值置于A(1)中。

```
Begin
 for k=m-1 to 0 do
   for j=2^k to 2^(k+1)-1 par-do
     A[j]=max{A[2j], A[2j+1]}
   end for
 end for
end
```

* 时间分析

算法的时间： t(n)=m×O(1)=O(logn)；

总比较次数： O(n)；

最大的处理器数： p(n)=n/2

并行算法的设计基础

1 并行算法的定义和分类

2 并行算法的表达

3 并行算法的复杂性度量

并行算法的定义和分类

- 并行算法的定义
  - 算法
  - 并行算法
- 并行算法的分类
  - 数值计算和非数值计算
  - 同步算法和异步算法
  - 分布算法
  - 确定算法和随机算法

并行算法的定义和分类

- 算法:是解题方法的精确描述，是一组有穷的规则，它们规定了解决某一特定问题的一系列运算。
- 并行算法:是一些可同时执行的多个进程的集合，这些进程相互作用和协调工作，从而达到对给定问题的求解。
- 从不同的角度，并行算法可以分为不同的类别：数值并行算法和非数值并行算法；同步的、异步的和分布式的并行算法；共享存储的和分布存储的并行算法；确定的和随机的并行算法等。

并行算法的定义和分类

- **数值计算（Numerical Computing）**是指基于代数关系运算的一类诸如矩阵计算、多项式求值、求解线性方程组等数字计算问题。求解数值计算问题的算法称为数值算法（Numerical Algorithm）。

- **非数值计算（Non-numerical Computing）**是指基于比较关系运算的一类计算问题，比如排序、选择、搜索和匹配等符号处理问题。求解非数值计算问题的算法称为非数值算法（Non-numerical Algorithm）。

并行算法的定义和分类

- **同步算法（Synchronized Algorithm）**是指算法的各个进程的执行必须相互等待的一类算法。

- **异步算法（Asynchronized Algorithm）**是指算法的各个进程的执行不必相互等待的一类算法。

- **分布算法（Distributed Algorithm）**是指由通信链路连接的多个节点协同完成问题求解的一类算法。

并行算法的定义和分类

- **确定性算法（Deterministic Algorithm）**是指算法的每一步都能明确的指明下一步的动作的一类算法。

- **随机算法（Randomized Algorithm）**是指算法的每一步都随机的从指定范围内选取若干参数，由此确定算法的下一动作的一类算法。

并行算法的表达

描述语言

* 可以使用类Algol、类Pascal等；
* 在描述语言中引入并行语句。
* 并行语句示例
* Par-do语句

```
for i=1 to n par-do
    ……
end for
```

* for all语句

```
for all Pi, where 0≤i≤k
    ……
end for
```

并行算法的复杂性度量

***串行算法的复杂性度量**

* 最坏情况下的复杂度(Worst-CASE Complexity)

* 期望复杂度(Expected Complexity)

*** 并行算法的几个复杂性度量指标**

* 运行时间t(n):包含计算时间和通讯时间，分别用计算时间步和选路时间步作单位。n为问题实例的输入规模。

* 处理器数p(n)

* 并行算法成本c(n): c(n)=t(n)p(n)

* 如果一个求解问题的并行算法之成本，在数量级上等于最坏情况下串行求解此问题所需的执行步数，则称此并行算法是成本最优（Cost Optimal）的。

*** 总运算量W(n): 并行算法求解问题时所完成的总的操作步数。**

并行算法的复杂性度量

***Brent定理**

令W(n)是某并行算法A在运行时间T(n)内所执行的运算量，则A使用p台处理器可在t(n)=O(W(n)/p+T(n))时间内执行完毕。

* W(n)和c(n)密切相关，c(n)=t(n)*p=O(W(n)+p*T(n))

* p=O(W(n)/T(n))时，W(n)和c(n)两者是渐进一致的

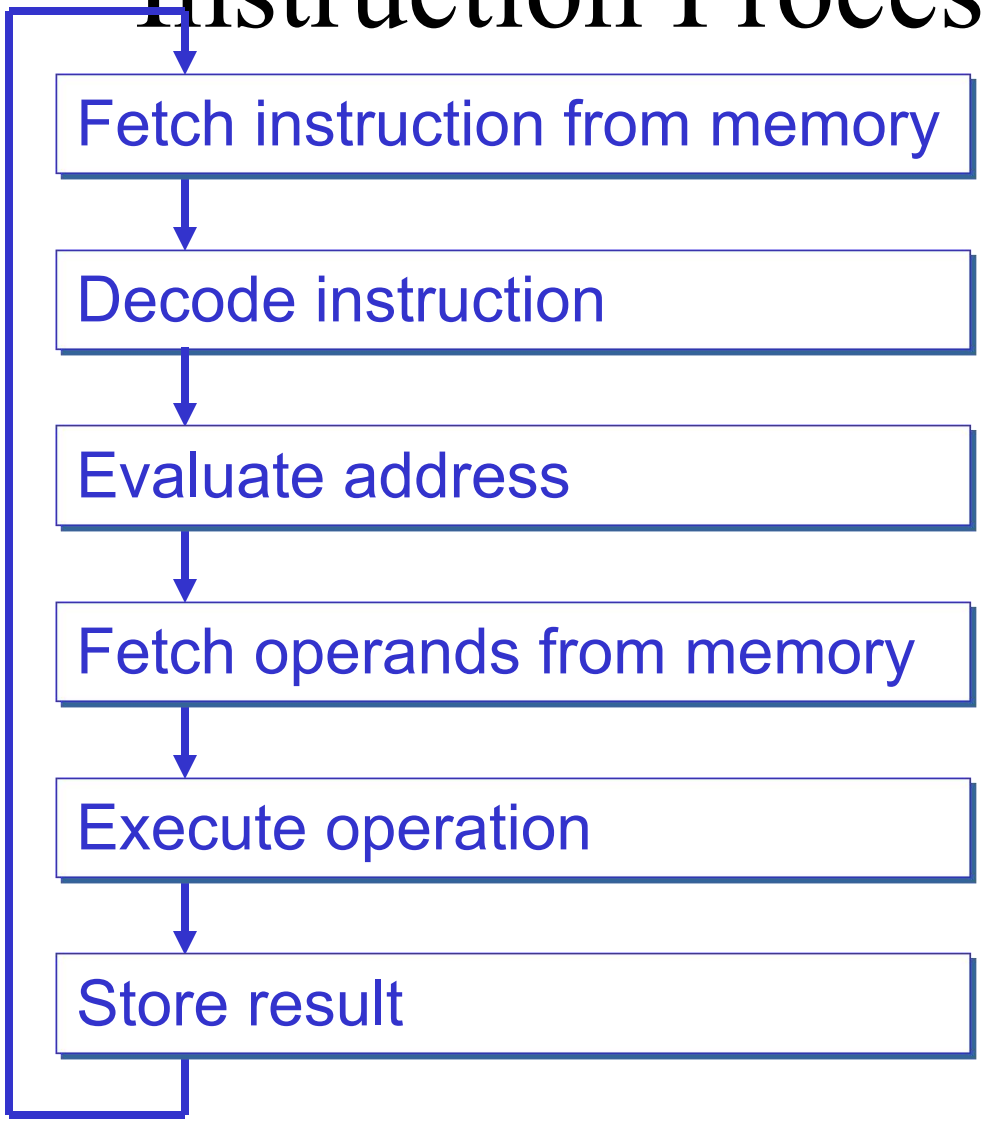
* 对于任意的p，c(n)›W(n)。这说明一个算法在运行过程中，不一定都能充分的利用有效的处理器去工作。

并行算法的设计基础

2. 并行计算模型

Von Neumann Model（冯.诺依曼模型）

Instruction Processing



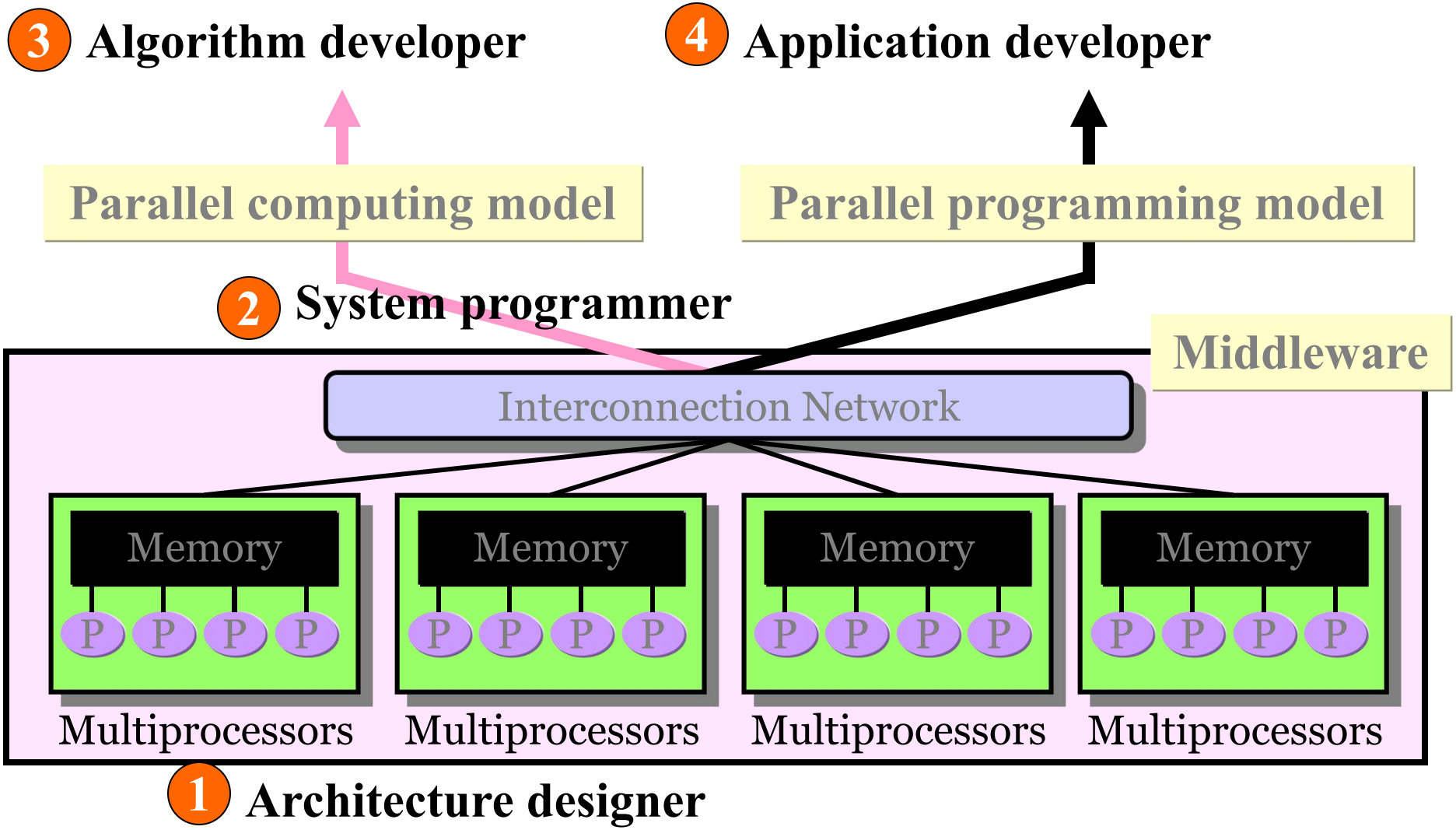
并行计算模型（Parallel Computing Model）

- Computing model
  - Bridge between SW and HW
    - general purpose HW, scalable HW
    - transportable SW
  - Abstract architecture for algorithm development
  - Ex) PRAM, BSP, LogP

并行编程模型（Parallel Programming Model）

- What programmer uses in coding applications?
  - Specifies communication and synchronization
  - Communication primitives exposed to used-level realizes the programming model
  - Ex) Uniprocessor, Multiprogramming, Data parallel, message-passing, shared-address-space

Aspects of Parallel Processing



Parallel Computing Models

- PRAM
  - Parallel Random Access Memory
  - A set of *p* processors
  - Global shared memory
    - Each processor can access any memory location in one time step
  - Globally synchronized
    - Executing same program in lockstep

PRAM算法

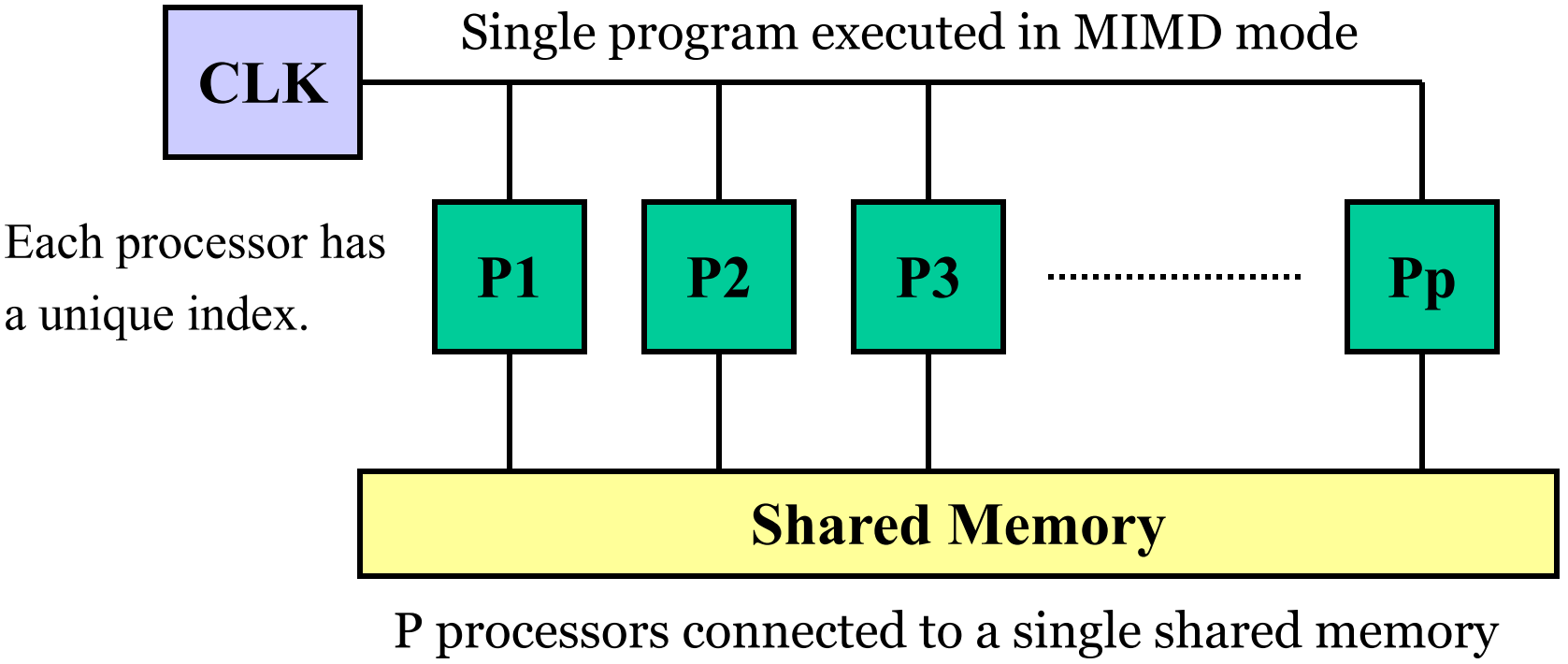
§2.1 串行计算模型

RAM（随机存取机器）

§2.2 并行计算的PRAM模型

PRAM由控制单元、全局存储器和一个无限的处理机集合组成，每个处理机有它自己的私有存储器。尽管活动的处理机执行同一指令，但每个处理机有一个唯一的索引，可以使用处理机的索引值启动或阻塞该处理机，或影响它存取的存储单元。

PRAM模型图


CLK
Single program executed in MIMD mode
Each processor has
a unique index.
P1
P2
P3
Pp
Shared Memory
P processors connected to a single shared memory

第二讲 PRAM算法

PRAM计算：

开始时，输入存储在全局存储中，一个处理单元是活动的。

在计算的每一步，一个活动的、允许中断的处理机（enabled processor）可以从私有存储器或全局存储器读取一个值，实行一个RAM操作，把结果写入局部或全局存储器单元。在一个计算步期间，一个处理机可以激活另一处理机。所有活动的、允许中断的处理机必须执行同一操作，尽管在不同的存储单元上。当最后一个处理机停机时计算终止。

§2.2 并行计算的PRAM模型

定义2.1 一个PRAM算法的成本是并行时间复杂度与所用的处理机数的乘积。

换句话说，成本等于最坏情况下求解一问题的总的执行步数。各种PRAM模型的区别主要在怎样处理读写冲突。当二个或多个处理机试图从全存储器同一单元读出，或写入时，产生读写冲突。研究文献中的大部分结果基于下述模型之一。

1．EREW（Exclusive Read Exclusive Write）　　EREW是最弱的

2．CREW（Concurrent Read Exclusive Write）

3．CRCW（Concurrent Read Concurrent Write）

成本最优：如果某算法的成本等于该问题最好的串行算法的复杂度：则说此并行算法是成本最优的。

§2.2 并行计算的PRAM模型

允许同时写是不现实的，于是又对PRAM-CRCW模型做了进一步约定，于是形成了下面几种模型：

Common：只允许所有的处理器同时写相同的数

ARBITRARY：任选一个赢者写入

Priority：只允许最优先的处理器先写

让我们考查一下这些模型的相对强度。

·EREW PRAM模型是最弱的

·PRIORITY PRAM模型是最强的

§2.2 并行计算的PRAM模型

- 因为PRIORITY PRAM模型强于EREW PRAM模型，因此在EREW PRAM上解决一个问题的算法的时间复杂度高于在PRIORITY PRAM解决同一问题的算法的时间复杂度。定理3.1量化了当从PRIORITY PRAM模型移到EREW PRAM模型时可能出现的并行时间复杂度的增长。
- **引理**3.1 P个处理器的EREW PRAM能用Θ（ logp ）时间将存储在全局存储器中的p个元素的数组分类（sort）。
- **定理**3.2 用P个处理器的EREW PRAM能模拟P个处理器的PRIORITY PRAM，但时间复杂度增加Θ（ logp ）倍，即TCROW· logp。
- **定理1**

  具有p个处理器的CRCW算法的运行速度至多比解决同一问题的最好的具有p个处理器的EREW算法快O(lg p)倍。

§2.3 PRAM算法

如果一个PRAM算法的时间复杂度比最优RAM算法的时间复杂度低，则是因为使用了并行性。由于一个PRAM算法开始时只有一个处理机是活动的，因此PRAM算法有两个阶段：

·第一阶段激活足够数量的处理机；

·第二阶段，所有已激活的处理机并行地执行计算。

给定一个活动处理机，由于执行一个单一指令活动处理机的数量能够加倍，因此仅需激活步可以使P个处理机成为活动的。在我们的PRAM算法表示中，使用元指令

spawn (<processor names>)

指示从单一活动处理机可用对数时间激活其它处理机。

为了第二阶段PRAM算法计算容易阅读，我们允许把对全局寄存器引用看作为数组引用。假定存在一个从这些数组引用到适当的全局寄存器的映射。

§2.3 PRAM算法

PRAM算法结构

for all <processor list> do {statement list} end for

表示由所指定的处理机并行执行的代码段。

除了已描述的特殊结构外，我们使用下述一些熟悉的控制结构表示PRAM算法：

if…then…else…end if

for… end for, while…end while, repeat…until等等。

符号←表示赋值。

2.3.1 并行归约（Parallel Reduction）

给定n个元素的集合和一个可结合的二元操作（associative binary opexator）。

归纳是计算的过程。

并行求和是归约操作的一个例子。

并行求和：给定n个值，求sum= $a_1 + a_2 + \cdots + a_n$

PRAM中的处理机操作的数据存储在全局寄存器中。用一个颗树来表示计算过程，树的结点（Node）是数组中的一个元素，如图2.1所示。

2.3.1 并行归约（Parallel Reduction）

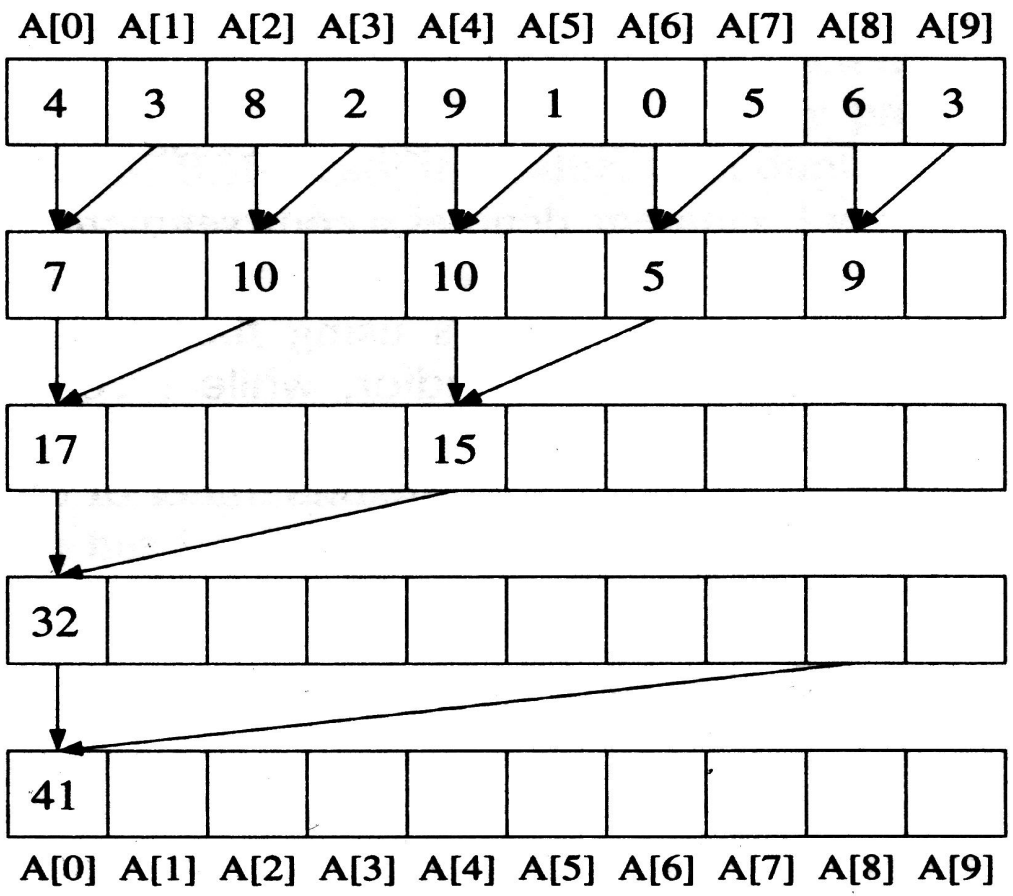


图2.1 利用PRAM算法对10个值并行归约的操作示例

2.3.1 并行归约（Parallel Reduction）

算法（Algorithm）
SUM (EREW PRAM)
Initial condition：n(≥1)个元素存储在数组 $A[0\cdots(n-1)]$中
Final condition：n个元素之和在A[0]中。
Global Variables：n,j, $A[0\cdots(n-1)]$
begin
Spawn ()
for all processors *Pi where* $0\le i\le\lfloor n/2\rfloor-1$ do
for $j\leftarrow 0$ to $\lceil\log n\rceil$-1 do
  if (i modulo 2^j)=0 and $2i+2^j$<n then
    A[2*i*]←A[2*i*]+A[2i+2^j]

2.3.1 并行归约（Parallel Reduction）

```
    end if
  end for
 end for
end
```

算法的计算复杂度或时间复杂度是Θ（?），使用的处理机数为？算法的成本H？。

2.3.1 并行归约（Parallel Reduction）

```
    end if
  end for
 end for
end
```

算法的计算复杂度或时间复杂度是$\Theta(\log^n)$，使用的处理机数为$p=\lfloor n/2 \rfloor$,算法的成本H=$(\frac{n}{2}\log n)$

- 显然，**该算法不是成本最优的**

2.3.2 前缀和（prefix sums）

给定n个元素和一个可结合的二元操作，求前缀问题是计算n个量：

$$a_1$$
$$a_1 \oplus a_2$$
$$a_1 \oplus a_2 \oplus a_3$$
$$\cdots\cdots$$
$$a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_n$$

前缀和有许多应用，例如，给定一个n个字母的数组，我们希望把大写字母抽取出来放在A的前面部分而保持它们原来的顺序。

A	A	b	C	D	e	F	g	→	A	A	C	D	F	b	e	g

2.3.2 前缀和（prefix sums）

算法

PREFIX. SUMS（CREW PRAM）

初始条件：n个元素（$n \geq 1$）存储在 $A[0 \cdots (n-1)]$ 中

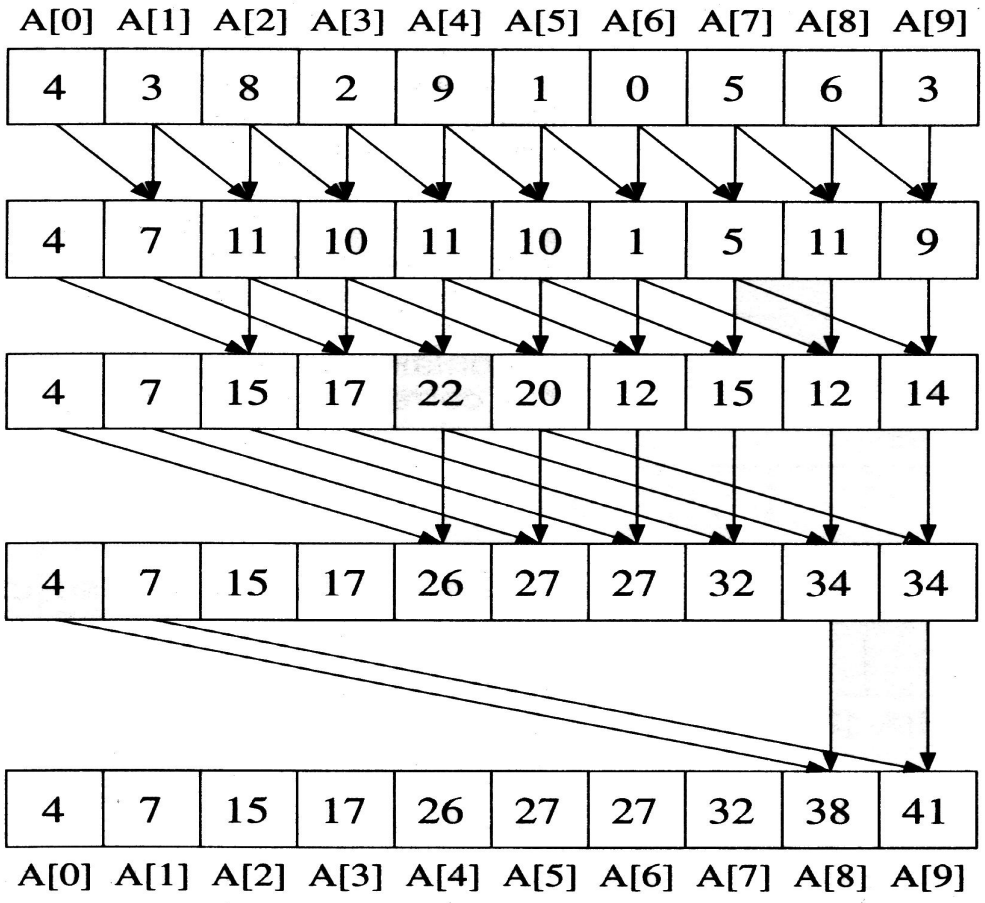
终止条件：每个 $A[i]$ 含有 $A[0] \oplus A[1] \oplus \cdots \oplus A[i]$

全局变量：n, $A[0 \cdots (n-1)]$, j

```
begin
    Spawn (p_1, p_2, ..., p_{n-1})
    for all  p_i  where 1 ≤ i ≤ n-1 do
      for j ← 0  to  ⌈log n⌉-1  do
        if  i - 2^j ≥ 0 then
              A[i] ← A[i] + A[i-2^j]
        end if
      end for
  end for
end
```

2.3.2 前缀和（prefix sums）

算法的计算过程如图2.2所示。算法的时间复杂度为$\Theta(\log n)$，所用处理机数为$n-1$。



成本最优否？

图2.2 n个数的前缀和（CREW PRAM模型）

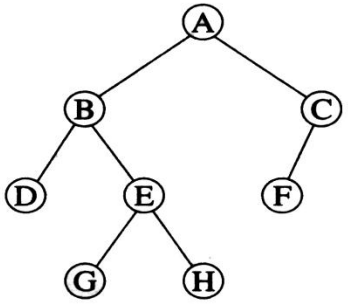
2.3.3 先序树遍历

有时试图把一个复杂的查看问题（a complicated-Looking problem）归约（reduce）为一个简单的并且已知有快速并行算法求解的问题。树的先序遍历属这类问题。

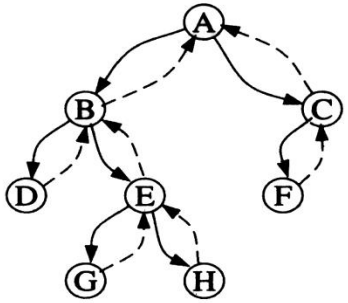
面向边的树遍历观点，产生了一个快速并行算法，算法有四个步骤。

Step1. 构造一个单向链表（singly-linked List），单向链表的每个顶点对应于遍历树时向下或向上的边，如图3.3所示。

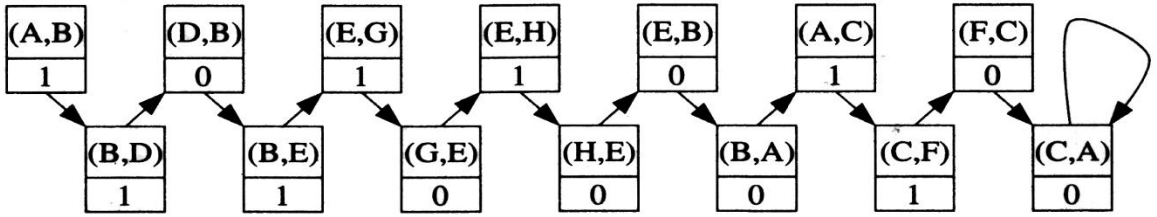
2.3.3 先序树遍历



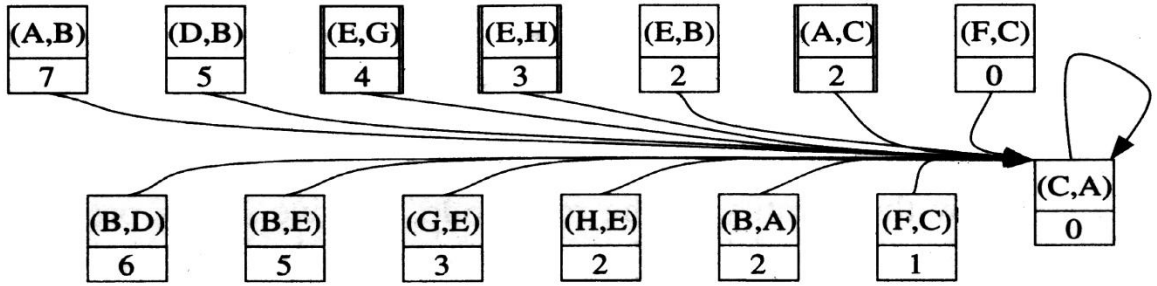
(a)



(b)



(c)



(d)

A	B	C	D	E	F	G	H
1	2	7	3	4	8	5	6

(e)

3.3.3 先序树遍历

图3.3 树的前序遍历。（a）树 （b）增加向上边 （c）对有向树的边建立联结表 （d）计算各结点到表尾的权值。（e）前序遍历结果

Step2. 给新创建的singly-linted list的每个结点赋权值。

只要向下遍历时遇到一个顶点，立刻给它作标记。根结点例外，必须使用不同的方法处理。

表中与向下的边相应的每一顶点有权值1，与向上的边相应的顶点有权值0。

（意味着穿过这个边时，该结点要计数） （表示该结点不计数）

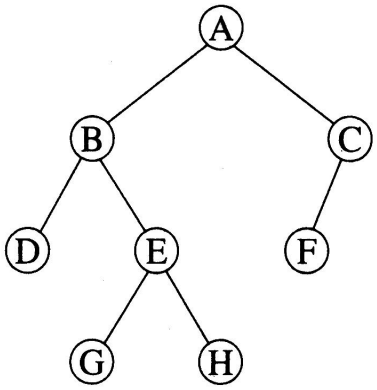
Step3. 计算。对singly-Linked List表中的每一元素，计算该元素的位序（rank）。

Step4. 与向下边相联的处理机使用已计算的位序把一个先序遍历数赋给与之相联的树的结点（在向下边的末端的结点）：

$(n+1)-$ 结点的位序数=该结点的先序遍历值。

3.3.3 先序树遍历

并行先序遍历算法的实现中，使用了如图3.4所示的数据结构，对于树的每一结点，数据结构存储：父结点，直接的右兄弟和最左的儿子结点。树的这种表示方法对于每一结点使得存储的数据量是一个常量，并且简化了树的遍历。



	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

图3.4 表示树的一种数据结构：对树的每个结点记录其双亲、直接右兄弟和最左孩子

3.3.3 先序树遍历

算法中，一个处理机处理一条边。由于分为向下边和向上边，共有2（n-1）条边，因此需要2（n-1）个处理机。

一旦所有的处理机被激活。

第一步：构造单向链表，链表的元素相应于前序遍历中的边。

$P(i,j)$处理边 (i,j)，$P(i,j)$处理机必须计算 $P(i,j)$的后继边。如果parent[i]=j，则(i,j)是向上的边。向上边有三种后继边：如果I有右兄弟k，则后继也为(j,k)；否则如果i有祖父结点r，则后继边为(j,r)；否则j是树的终止结点，则在表的末端结点放上一个园圈，并知j为树根，置它的先序数为1。

如果parent $[i] \neq j$，则 (i,j)是向下的边，有两种类型的后继：如果儿子结点j有儿子k，则后继边为(j,k)；否则j是叶结点，后继结点为 (j,i)。

3.3.3 先序树遍历

第二步：赋权值，即给position赋值，1赋给向下的边，0赋给向上的边

第三步：采用指针跳跃技术计算每个元素的位序，即position值。

最后的position值指明了表元素到表的末端之间的先序遍历结点数。

第四步：根据位序值。即position值计算每个结点的先序遍历标号（值）

preorder[j]←n+1-position[(i,j)]

算法描述：

```
PREORDER.TREE.TRAVERSAL(CREW PRAM):
Global n                    {Number of vertices in tree}
parent[1...n]               {Vertex number of parent node}
child[1...n]                {Vertex number of first child}
sibling[1...n]              {Vertex number of sibling}
succ[1...(n-1)]             {Index of successor edge}
position[1...(n-1)]         {Edge rank}
preorder[1...n]             {Preorder traversal number}
```

```
begin
  spawn (set of all p(i,j) where(i,j) is an edge)
  for all p(i,j) where (i,j) is an edge do
{Put the edges into a linked list}
if parent[i]=j then
  if sibling[i] ≠null then
    succ[(i,j)]←(j, sibling[i])
else if parent[j] ≠ null then
  succ[(i,j)]←(j, parent[j])
else
  succ[(i,j)]←(i,j)
  preorder[j]←1 {j is root of tree}
endif
```

```
 else
  if child[j] ≠null then succ[(i,j)]←(j, child[j])
   else succ[(i,j)]←(j,i)
  endif
endif
   {Number of edges of the successor list}
 if parent[i]=j then position[(i,j)] ←0
 else position[(i,j)] ←1
 endif
  {Perform suffix sum on successor list}
 for k←1 to  do
  position[(i,j)] ←position[(i,j)]+ position[succ[(i,j)]]
  succ[(i,j)] ←succ[succ[(i,j)]]
  endfor
   {Assign preorder values}
   if i=parent[j] then preorder[j] ←n+1-position[(i,j)]
endif
endfor
end
```

2. 异步APRAM（MIMD-SM）模型　P.28

- ❖ 模型特性
  - ➢ 每个处理器有其局部存储器、局部时钟、局部程序；
  - ➢ **无全局时钟，各处理器异步执行**；
  - ➢ 处理器通过SM进行通讯、交换数据；
  - ➢ 处理器之间的依赖关系，**需在并行算法（程序）中显式地加入同步路障语句来指定同步**，即**软件同步**。算法（程序）设计难度比APRAM模型稍难些。
  - ➢ 时间复杂度分析：**并行（并发）进程生成时间、并行计算时间、**并行（并发）进程**计算（处理）过程中的同步时间、**并行（并发）进程**结束的同步时间。**
- ❖ 指令类型：全局读、全局写、局部操作、同步
- ❖ 计算时间

设局部操作为单位时间；全局读/写平均时间为d，d随着处理器数目的增加而增加；同步路障时间为B=B(p)非降函数。

令 t_{ph} 为全局相内各处理器执行时间最长者，则APRAM上的计算时间为：

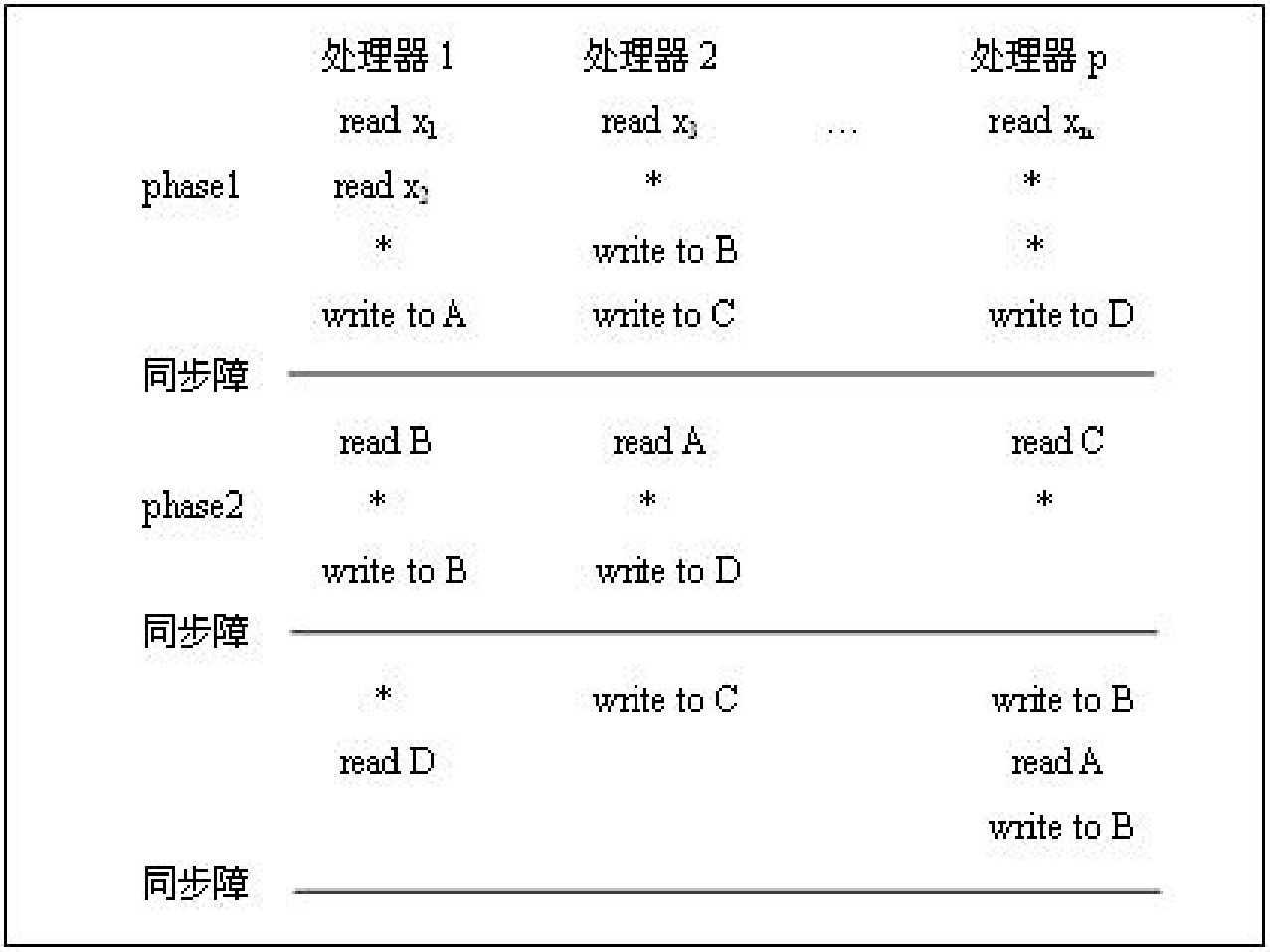
$$T = \sum t_{ph} + B \times \text{同步障次数}$$

注：多核处理结构的多核计算模型就是异步APRAM（MIMD-SM）模型。

2. 异步APRAM（MIMD-SM）模型

❖ 计算过程

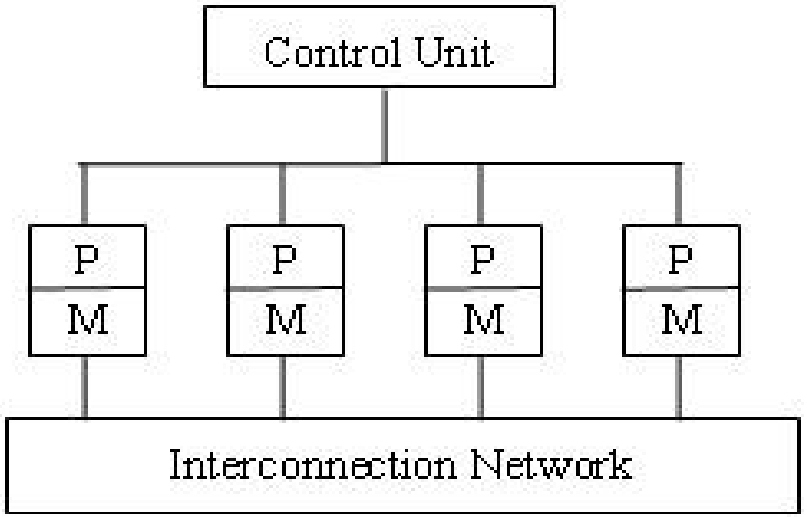
异步APRAM（MIMD-SM）模型的计算过程由同步障分开的全局相组成。



1.3.3 分布存储的并行计算模型

1. SIMD-IN（ SIMD-DM）模型——分布存储SIMD模型

- ❖ 模型特性
  - ❑ **每个处理器均自己的存储器（分布式存储）**；
  - ❑ 处理器之间通过互连网络相连，采用**传递数据方式实现通讯**；
  - ❑ 运行在各处理器的算法（进程）**同步并行执行**，处理器之间**同步由硬件实现**；
  - ❑ 算法时间复杂度分析：计算时间和**选路(路由，通信)时间**。有些应用所需的选路(路由，通信)时间比其计算时间还要多。
- ❖ 模型的结构图



1.3.3 分布存储的并行计算模型

1. SIMD-IN（ SIMD-DM）模型——分布存储SIMD模型

- **SIMD-IN（SIMD-DM）模型的常见模型**
  - SIMD-LC 一维线性连接
  - SIMD-MC 网孔连接
  - SIMD-TC 树形连接
  - SIMD-MT 树网连接
  - SIMD-HC 超立方连接
  - SIMD-CCC 立方环连接
  - SIMD-SE 洗牌交换连接
  - SIMD-BF 蝶形网络
  - SIMD-MIN 多级互连网络

在立方体结构的SIMD计算机上的算法描述：

begin

for $i \leftarrow 2$ *downto* 0 *do*

$k \leftarrow 2^i$

for all p_j *where* $0 \le j < k$ *do*

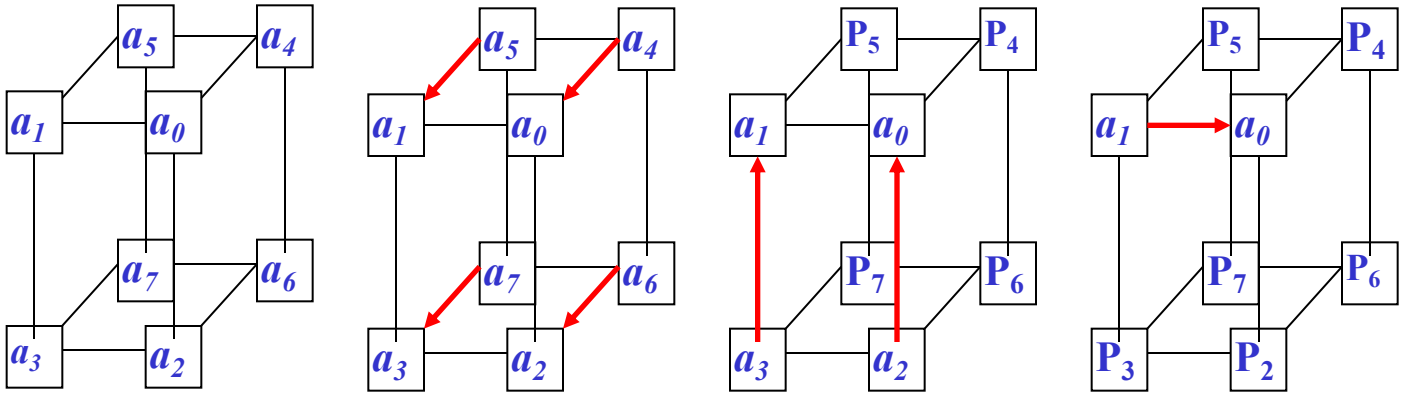
$t_j \Leftarrow a_{j+k}$

$a_j \leftarrow a_j + t_j$

endfor

endfor

end



立方体模型上的结合扇入求和算法

BSP模型

- 基本概念
  - BSP 模型（Bulk Synchronous Parallel ）由Valiant(1990)提出的，“块”同步模型，是一种异步MIMD-DM模型，支持消息传递系统，块内异步并行，块间显式同步。
  - BSP模型从PRAM的基础上发展起来，它早期的版本就叫做XPRAM。相比之下，APRAM模型是一种"轻量级"同步模型，而BSP则是"大"同步模型。

BSP模型参数

BSP模型作为计算机语言和体系结构之间的桥梁，并以下述三个参数描述的分布存储的多计算机模型：

- 处理器/存储器模块，模型中用p表示处理器/存储器模块数目

- 处理器/存储器模块之间点对点传递消息的路由器，模型中用g表示路由器吞吐率（也称为带宽因子）；

- 执行时间间隔L为周期的障碍同步的障碍同步器，其中L表示全局同步之间的时间间隔；

- 模型参数
  - *p*：处理器数(带有存储器)
  - *l*：同步障时间(Barrier synchronization time)
  - *g*：带宽因子(time steps/packet)=1/bandwidth

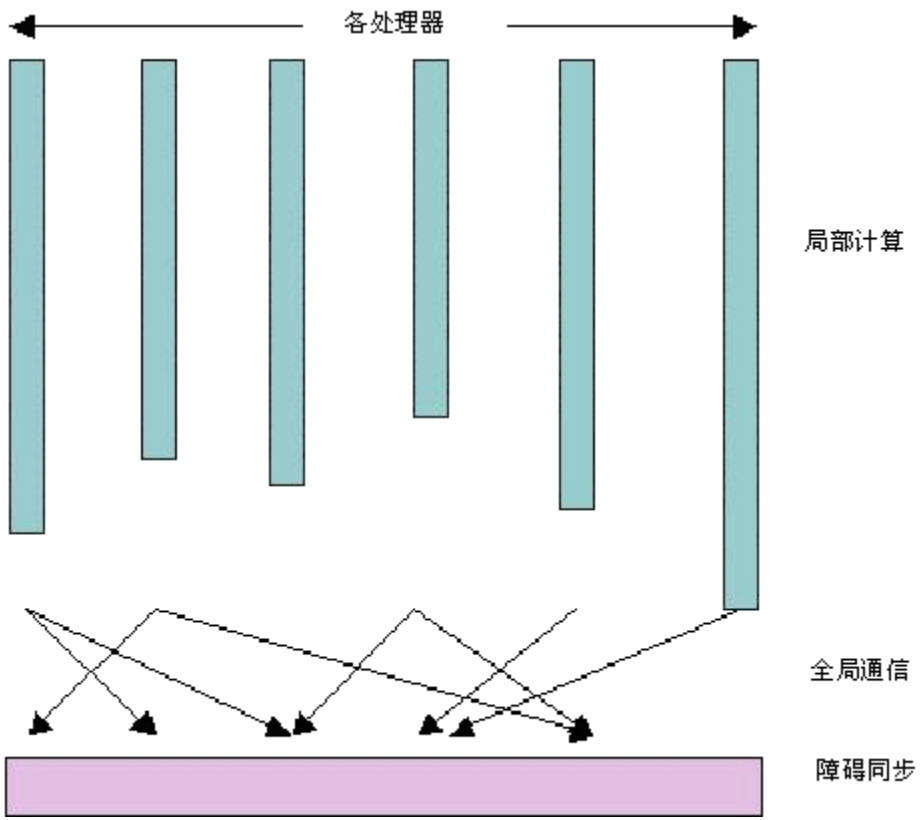
BSP模型

- 计算过程

  BSP模型的计算可以用下面的图表示。

  在BSP模型中，计算由一系列用全局同步分开的周期为L的计算组成，这些计算称为超级步。在各超级步中，每个处理器均执行局部计算，并通过选路器接受和发送消息；然后作一全局检查，以确定该超级步是否已由所有的处理器完成；若是，则进行到下一超级步，否则，下一个L周期被分配给未曾完成的超级步。

BSP模型



BSP模型

- **BSP模型中的成本分析**

在BSP的一个超级计算步中，可以抽象出BSP的成本模型如下：

一个超级计算步成本= $\max_{\text{所有进程}}\{w_i\} + \max\{h_i g\} + L$

其中，wi是进程I的局部计算时间，hi是进程I发送或接收的最大通信包数，g是带宽的倒数（时间步/通信包），L是障碍同步时间（注意，在BSP成本模型中，并没有考虑到I/O的时间）。所以，在BSP计算中，如果用了s个超级步，则总的运行时间为：

$$T_{BSP} = \sum_{i=0}^{s-1} w_i + g\sum_{i=0}^{s-1} h_i + sL$$

这个性能公式对算法和程序分析是很简单方便的。

BSP模型

- BSP模型的特点

  BSP模型是个分布存储的MIMD计算模型，其特点是：
  （1）它将处理器和路由器分开，强调了计算任务和通信任务的分开，而路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖具体的互连网络拓扑，又简化了通信协议；
  （2）采用障碍同步的方式以硬件实现的全局同步是在可控的粗粒度级，从而提供了执行紧耦合同步式并行算法的有效方式，而程序员并无过分的负担；

BSP模型

（3）在分析BSP模型的性能时，假定局部操作可以在一个时间步内完成，而在每一个超级步中，一个处理器至多发送或接收h条消息（称为h-relation）。假定s是传输建立时间，所以传送h条消息的时间为gh+s，如果 $gh \geq 2s$ 则L至少应该大于等于gh。很清楚，硬件可以将L设置尽量小（例如使用流水线或大的通信带宽使g尽量小），而软件可以设置L的上限（因为L越大，并行粒度越大）。在实际使用中，g可以定义为每秒处理器所能完成的局部计算数目与每秒路由器所能传输的数据量之比。如果能够合适的平衡计算和通信，则BSP模型在可编程性方面具有主要的优点，而直接在BSP模型上执行算法（不是自动的编译它们），这个优点将随着g的增加而更加明显；

BSP模型

- （4）为PRAM模型所设计的算法，都可以采用在每个BSP处理器上模拟一些PRAM处理器的方法来实现。理论分析证明，这种模拟在常数因子范围内是最佳的，只要并行宽松度（Parallel Slackness），即每个BSP处理器所能模拟的PRAM处理器的数目足够大。在并发情况下，多个处理器同时访问分布式的存储器会引起一些问题，但使用散列方法可以使程序均匀的访问分布式存储器。

BSP模型

- 对BSP模型的评价

  - 在研究并行计算时，Valiant试图也为软件和硬件之间架起一座类似于冯.诺伊曼机的桥梁，它论证了BSP模型可以起到这样的作用，正是因为如此，BSP模型也常叫做桥模型；

  - 一般而言，分布存储的MIMD模型的可编程性比较差，但在BSP模型中，如果计算和通信可以合适的平衡（例如g=1），则它在可编程方面呈现出主要的优点；

  - 在BSP模型上，曾直接实现了一些重要的算法（如矩阵乘、并行前序运算、FFT和排序等），他们均避免了自动存储管理的额外开销；

  - BSP模型可以有效的在超立方体网络和光交叉开关互连技术上实现，显示出，该模型与特定的技术实现无关，只要路由器有一定的通信吞吐率；

BSP模型

- - 在BSP模型中，超级步的长度必须能够充分的适应任意的h-relation，这一点是人们最不喜欢的；
  - 在BSP模型中，在超级步开始发送的消息，即使网络延迟时间比超级步的长度短，它也只能在下一个超级步才能使用；
  - BSP模型中的全局障碍同步假定是用特殊的硬件支持的，这在很多并行机中可能没有相应的硬件；
  - Valiant所提出的编程模拟环境，在算法模拟时的常数可能不是很小的，如果考虑到进程间的切换（可能不仅要设置寄存器，而且可能还有部分高速缓存），则这个常数可能很大。

BSP模型

- 优缺点

  强调了计算和通讯的分离，

  提供了一个编程环境，

  易于程序复杂性分析。

  但需要显式同步机制，限制至多h条消息的传递等。

logP模型

- 基本概念
  - 根据技术发展的趋势，20世纪90年代末和未来的并行计算机发展的主流之一是巨量并行机，即MPC（Massively Parallel Computers），它由成千个功能强大的处理器/存储器节点，通过具有有限带宽的和相当大的延迟的互连网络构成。所以我们建立并行计算模型应该充分考虑到这个情况，这样基于模型的并行算法才能在现有和将来的并行计算机上有效的运行。根据已有的编程经验，现有的共享存储、消息传递和数据并行等编程方式都很流行，但还没有一个公认的和占支配地位的编程方式，因此应该寻求一种与上面的编程方式无关的计算模型。而根据现有的理论模型，共享存储PRAM模型和互连网络的SIMD模型对开发并行算法还不够合适，因为它们既没有包含分布存储的情况，也没有考虑通信和同步等实际因素，从而也不能精确的反映运行在真实的并行计算机上的算法的行为，所以，1993年D.Culer等人在分析了分布式存储计算机特点的基础上，提出了点对点通信的多计算机模型，它充分说明了互联网络的性能特性，而不涉及到具体的网络结构，也不假定算法一定要用现实的消息传递操作进行描述。
  - 由Culler(1993)年提出的，是一种分布存储的、点到点通讯的多处理机模型，其中通讯由一组参数描述，实行隐式同步。

logP模型

- 模型参数

  LogP模型是一种分布存储的、点到点通信的多处理机模型，其中通信网络由4个主要参数来描述：

  – *L*：network latency表示源处理机与目的处理机进行消息（一个或几个字）通信所需要的等待或延迟时间的上限，表示网络中消息的延迟。

  – *o*：communication overhead表示处理机准备发送或接收每个消息的时间开销（包括操作系统核心开销和网络软件开销），在这段时间里处理不能执行其它操作。

  – *g*：gap=1/bandwidth表示一台处理机连续两次发送或接收消息时的最小时间间隔，其倒数即微处理机的通信带宽。

  – *P*：#processors处理机/存储器模块个数

  注：*L*和*g*反映了通讯网络的容量

logP模型

- **LogP模型的特点**

  (1)抓住了网络与处理机之间的性能瓶颈。g反映了通信带宽，单位时间内最多有L/g个消息能进行处理机间传送。

  (2)处理机之间异步工作，并通过处理机间的消息传送来完成同步。

  (3)对多线程技术有一定反映。每个物理处理机可以模拟多个虚拟处理机(VP)，当某个VP有访问请求时，计算不会终止，但VP的个数受限于通信带宽和上下文交换的开销。VP受限于网络容量，至多有L/g个VP。

logP模型

- (4)消息延迟不确定，但延迟不大于L。消息经历的等待时间是不可预测的，但在没有阻塞的情况下，最大不超过L。

  (5)LogP模型鼓励编程人员采用一些好的策略，如作业分配，计算与通信重叠以及平衡的通信模式等。

  (6)可以预估算法的实际运行时间。

logP模型

- **LogP模型的不足之处**

（1）对网络中的通信模式描述的不够深入。如重发消息可能占满带宽、中间路由器缓存饱和等未加描述。

（2）LogP模型主要适用于消息传递算法设计，对于共享存储模式，则简单地认为远地读操作相当于两次消息传递，未考虑流水线预取技术、Cache引起的数据不一致性以及Cache命中率对计算的影响。

（3）未考虑多线程技术的上下文开销。

（4）LogP模型假设用点对点消息路由器进行通信，这增加了编程者考虑路由器上相关通信操作的负担。

logP模型

- 优缺点

  捕捉了MPC的通讯瓶颈，隐藏了并行机的网络拓扑、路由、协议，可以应用到共享存储、消息传递、数据并行的编程模型中；但难以进行算法描述、设计和分析。

- **BSP vs. LogP**
  - BSP→LogP：BSP块同步→BSP子集同步→BSP进程对同步＝LogP
  - BSP可以常数因子模拟LogP，LogP可以对数因子模拟BSP
  - BSP＝LogP+Barriers－Overhead
  - BSP提供了更方便的程设环境，LogP更好地利用了机器资源
  - BSP似乎更简单、方便和符合结构化编程

LogP模型和BSP模型的比较

- BSP把所有的计算和通信视为一个整体行为而不是一个单独的进程和通信的个体行为，它采用各进程延迟通信的办法，将单独的消息组合成一个尽可能大的通信实体，然后进行路由和传输，这就是所谓的整体大同步。它简化了算法（程序）的设计和分析，但在同时也牺牲了运行时间，因为延迟通信意味着所有的进程都必须等待着它们中最慢的进程。一种改进的方法是采用子集同步，即将所有的进程按快慢程度分成若干个子集，于是整体的大同步就演变成子集内的同步。如果子集小到每个集合只包含消息的发送/接收者，则它就变成了异步的个体同步，这也就是LogP模型所描述的情形。也就是说，如果BSP中考虑到个体通信所造成的开销（Overhead）而去掉障碍同步就变成了LopP，这可以用下面的公式来说明：

  BSP+Overhead-Barrier=LogP

LogP模型和BSP模型的比较

- BSP模型的创始人L.G.Valiant曾从理论上论证并行计算不必优化在单一通信级（Single-Message Level），他认为整体大同步能够大大的简化并行计算（算法和程序设计）的设计、分析、验证、性能预测和具体实现，而基于成对消息传递的个体异步并行计算（例如LogP模型），在时间上的收益比起计算性能上难以分析和预测的缺点来说，并不合算。目前，对BSP模型的质疑主要集中在两点，即延迟通信到某一个特点的通信点以及频繁的障碍同步，会不会成为性能下降的主要原因，从而使成本过高。BSP模型的支持者们对这两个问题进行了研究，他们认为，延迟通信能提供更多的优化通信的机会，采用通信聚集和全局通信调度能减少拥挤和竞争；而障碍同步对采用共享存储结构的系统是并不太费时的，而对分布存储结构的系统，造成障碍同步比较昂贵的主要原因是，目前底层软件绝大多数都不支持对相应的硬件的访问，但不论怎样，障碍同步的成本可以折合到全局通信中，从而可以部分的抵消。

LogP模型和BSP模型的比较

- BSP模型和LogP模型在本质上是等效的，它们可以相互模拟：用BSP去模拟LogP所进行的计算时，通常会慢常数倍，而用LogP去模拟BSP所进行的计算时，通常会慢对数倍。直观上讲，BSP为算法（和程序）设计和分析提供了很多的方便，而LogP模型却提供了更强大的机器资源的控制能力。BSP在精确度方面带来的损失比起它所能提供的更结构化的编程风格的优点来说，是可以接受的。总之，BSP模型由于它的简明性、性能的可预测性、可移植性和编程的结构性，赢得了更多的青睐。

C3模型

- 1994年S.E.Hambrush等人在分析高性能可扩展的网络计算机系统特点的基础上提供了C3(Computation, Communication, Congestion)模型，它是一种与体系结构无关的粗粒度的并行计算模型，他以时下逐渐流行的并行集群系统为目标，旨在反映计算复杂度、通信模式和资源（信道）冲突对粗粒度网络并行算法的影响。与H-PRAM相似，它认为可将算法分成多个“超步”（相当于H-PRAM中的“计算段”和其后的通信操作）。
- C3模型假定处理机不能同时发送和接收消息，它对超步的性能分析分为两部分：计算单元CU，依赖于本地计算量；通信单元COU，依赖与处理机发送和接收数据的多少、消息的延迟及通信引起的拥挤量。该模型考虑了两种路由（存储转发路由和虫蚀寻径路由）和两种发送/接收原语（阻塞和无阻塞）对COU的影响。

C3模型

- **C3模型的5个参数**
  - (1) p 处理器个数
  - (2) h 网络延迟
  - (3) b 网络的对分宽度
  - (4) S 启动时间，及建立一个消息时的开销
  - (5) L 消息包的长度，即消息包所含字节数

C3模型

- **C3 模型的开销**

C3模型用2个量C和La来描述网络的拥挤，其中，C表示参与通信的处理机对的数目， La表示处理机间路由消息包的平均数目，则有

(1) 连接上的拥挤量Cl=La*C/b

(2) 处理机上的拥挤量Cp=La*C/b*h

在一个超步中，若Si与Ri分别表示第i 个处理机总的发送和接收时间，则有

$$COU=\max(Si+Ri)+Cl+Cp,\ 0<i<p-1$$

C3模型

- **C3 模型的特点**

(1)用Cl和Cp来度量网络的拥挤对算法性能的影响；

(2)考虑了不同路由和不同发送或接收原语对通信的影响；

(3)不需要用户指定调度细节，就可以评估超步的时间复杂性；

(4)类似于H-PRAM模型的层次结构，C3模型给编程者提供了K级路由算法的思路，即系统被分为K级子系统，各级子系统的操作相互独立，用超步代替了H-PRAM中的Sub PRAM进行分割。

C3模型

- **C3 模型的不足之处**

(1)Cl度量的前题假设为同一通信对中的2个处理机要分别位于网络对分后的不同子网络内；

(2)模型假设了网络带宽等于处理机带宽，这影响了正确描述可扩展系统；

(3)在K级算法中，处理机间顺序可以由多种排列，但C3模型不能区分不同排列的难易程度。

BDM模型

- **BDM模型的参数**

1996年J.F.JaJa等人提出了一种块分布存储模型(BDM, Block Distributed Model)。它是共享存储编程模式与基于消息传递的分布存储系统之间的桥梁模型。主要的4个参数为：

(1) P 处理器个数；

(2)τ 处理机从发出访问请求到得到远程数据的最大延迟时间(包括准备请求时间、请求包在网络中路由的时间、目的处理机接收请求的时间以及将包中M个连续字返回给原处理机的时间)；

(3)M 局部存储器中连续的M个字；

(4)σ 处理机发送数据到网络或从网络接收数据的时间。

BDM模型

- **BDM模型的特点**

(1)用M反映出空间局部性特点，提供了一种评价共享主存算法的性能方法，度量了因远程访问引起的处理间的通信；

(2)BDM认可流水线技术。某个处理机的K次预取所需的时间为$\tau+KM\sigma$ (否则为$K(\tau+M\sigma)$)

(3)可编程型好；

(4)考虑了共享主存中的存储竞争问题；

(5)可以用来分析网络路由情况。

BDM模型

- **BDM模型的不足**

(1)认为初始数据置于局存中，对于共享主存程序的编程者来说，需要额外增加数据移动操作；

(2)未考虑网络中影响延迟的因素(如处理机的本地性、网络重拥挤等)；

(3)未考虑系统开销。

并行程序模型 小结

- 并行计算模型是设计和分析并行算法的基础。PRAM模型由于过于抽象而不能很好地反映并行算法的实际运行性能，所以研究考虑通信、同步等因素的影响，从而建立能更真实的反映并行算法运行性能的更实际的计算模型是当今并行算法研究的主要方向之一。本节后面讨论的模型都属于这种模型。但过去几年来，学术界主要是从理论分析的角度，来研究它们之上的一些典型的并行算法的设计与分析；而近期的研究热点逐渐从这些模型上的算法研究转向了利用这些模型进行程序设计的研究，即从理论研究转向了实际应用。因为任何一种并行算法的应用都最终落实到具体的编程上，所以这种转变是顺应应用要求的。例如，一些研究者就为BSP模型构造了一些函数库，这些BSP库就是一些为程序员编写BSP应用程序（这些应用程序按照BSP的超级计算步的风格进行编写）所提供的一组简单有力的编程界面函数，这些函数可以改善程序的可移植性而不依赖于具体的并行计算机结构。尽管PVM和MPI等也是目前可供使用的开发可移植并行代码的方法，但它们的功能过于复杂而难于掌握，而且它们都没有为编程者提供能设计高效代码的成本函数，而像BSP模型却提供了简单和可定量分析程序运行时间的成本函数。因此研究基于这些实用计算模型的并行程序设计方法是非常有意义的

并行算法复杂性的度量

2. 并行算法复杂性的度量

- 加速比$S_p(n)$：$S_p(n)=t_s(n)/t_p(n)$，$1\le S_p(n)\le p(n)$

  其中$t_s(n)$为求解问题的最快的串行算法在最坏情形下所需的运行时间，$t_p(n)$为求解同一问题的并行算法在最坏情形下的运行时间。

  - **加速比$S_p(n)$反映算法的并行性对运行时间的改进程度**。**工程实践中达到对数加速**以上认为该并行算法可行。
  - 若$S_p(n)=p(n)$，则达到线性加速；若$S_p(n)>p(n)$，则为超线性加速(一般出现在某些特殊的应用中，如人工智能中的并行搜索等)

- 并行效率$E_p(n)$：$E_p(n)=S_p(n)/p(n)$，$0<E_p(n)\le 1$

  $E_p(n)$反映了并行系统中**处理器的利用程度**。

- 工作量(或运算量) $W(n)$：并行算法所执行的总的**基本操作（运算）**步次数。(与处理器的数目无关)

  例如：矩阵乘积算法的基本操作（运算）是“**乘法**”；数据排序算法的**基本操作（运算）**是“**比较**”。

并行算法的复杂性度量

并行度：算法的并行度是指该算法中能用一个计算步(并行)完成的运算或操作的个数。在并行机上即为可同时运算的处理机的数目，而在向量机上则为向量操作的长度。

平均并行度：算法的平均并行度为该算法总的操作数除以计算步数。在向量机上则是平均向量长度。

算法的相容性：如果当问题的规模 $n \to \infty$ 时，并行（向量）算法和串行算法运算总数之比 $P(n)/S(n)$ 有界，则称同一问题的一个 向量算法或并行算法与串行算法是相容的；否则，称为不相容的，这时 $P(n)/S(n) \to \infty$ 。

可扩展性：可扩展性是指在确定的应用背景下，计算机系统性能随处理器个数的增加而线性的提高。对于算法而言，就是对其适应并行计算时多处理器个数增加的能力的度量。当可使用的处理器个数增加时，如果算法的效率曲线保持不变或下降不大，则认为该算法在所用的并行机上可扩展性好。反之，则称其可扩展性差。

并行计算性能评测

2.2.1 并行加速比和效率

- 加速比性能定律
  - 并行系统的加速比是指对于一个给定的应用，并行算法（或并行程序）的执行速度相对于串行算法（或串行程序）的执行速度加快了多少倍。
  - Amdahl 定律
  - Gustafson定律
  - Sun Ni定律

2.2.2 Amdahl 定律

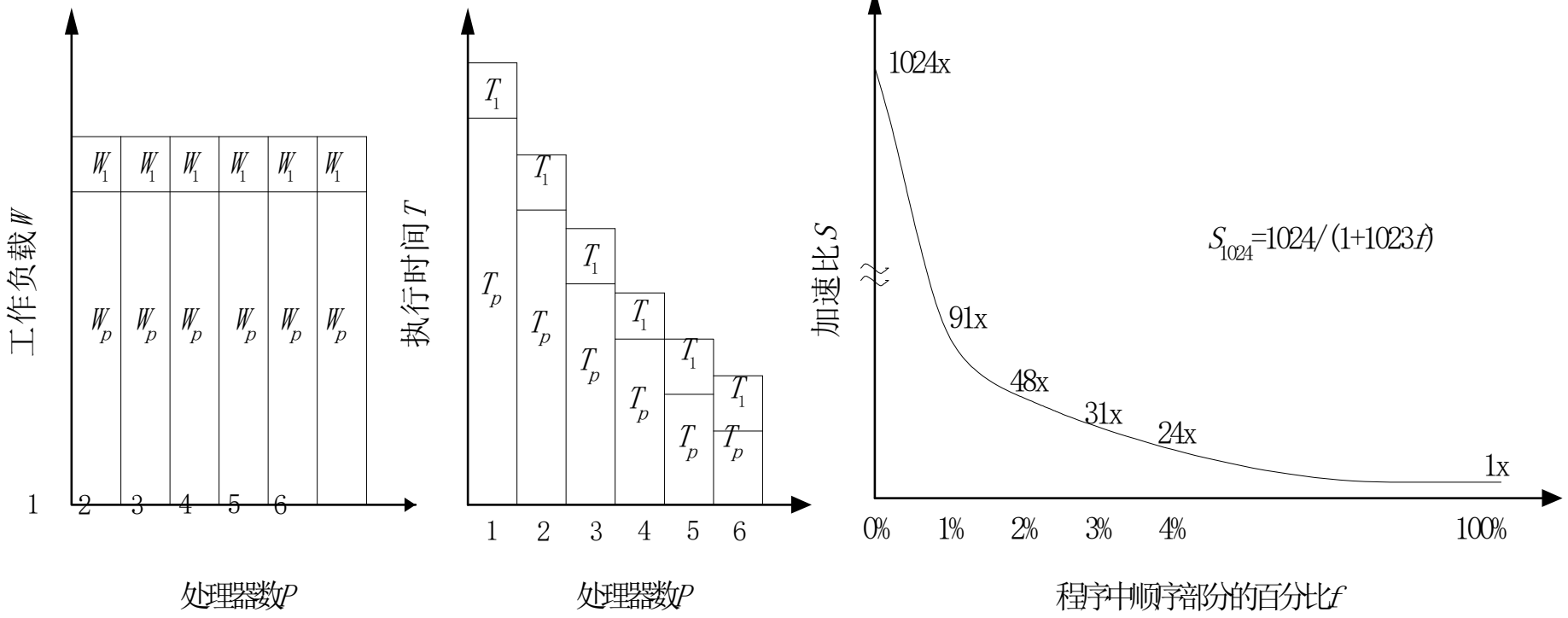
- P：处理器数；
- W：问题规模（计算负载、工作负载，给定问题的总计算量）；
  - Ws：应用程序中的串行分量，f是串行分量比例（f = Ws/W，Ws=W1）；
  - WP：应用程序中可并行化部分，1-f为并行分量比例；
  - Ws +W p =W；
- Ts=T1 ：串行执行时间，T p ：并行执行时间；
- S：加速比，E：效率；
- 出发点：
  - 固定不变的计算负载；
  - 固定的计算负载分布在多个处理器上的，
  - 增加处理器加快执行速度，从而达到了加速的目的。

- 固定负载的加速公式： $S = \dfrac{Ws + Wp}{Ws + W_P / p}$
- **$W_s + W_p$**可相应地表示为**f+（1-f）**

$$S = \frac{f + (1 - f)}{f + \dfrac{1 - f}{p}} = \frac{p}{1 + f(p - 1)}$$

- **p→∞**时，上式极限为： **S= 1 / f**
- **W_o**为额外开销

$$S = \frac{W_S + W_P}{W_S + \dfrac{W_P}{p} + W_O} = \frac{W}{fW + \dfrac{W(1-f)}{p} + W_O} = \frac{p}{1 + f(p-1) + W_O p / W}$$


工作负载W
W_1
W_p
1
2
3
4
5
6
处理器数P
(a)
执行时间T
T_1
T_p
处理器数P
(b)
加速比S
1024x
91x
48x
31x
24x
1x
S_{1024}=1024/(1+1023f)
0%
1%
2%
3%
4%
100%
程序中顺序部分的百分比f
(c)

2.2.3 Gustafson定律

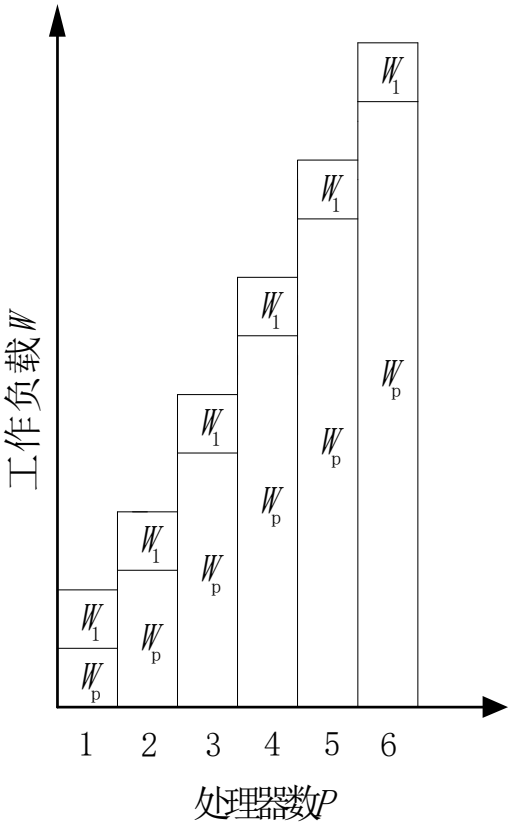
- 出发点：
  - 对于很多大型计算，精度要求很高，即在此类应用中精度是个关键因素，而计算时间是固定不变的。此时为了提高精度，必须加大计算量，相应地亦必须增多处理器数才能维持时间不变；
  - 除非学术研究，在实际应用中没有必要固定工作负载而计算程序运行在不同数目的处理器上，增多处理器必须相应地增大问题规模才有实际意义。
- Gustafson加速定律：

$$S' = \frac{W_S + pWp}{W_S + p \cdot Wp / p} = \frac{W_S + pWp}{W_S + W_P}$$

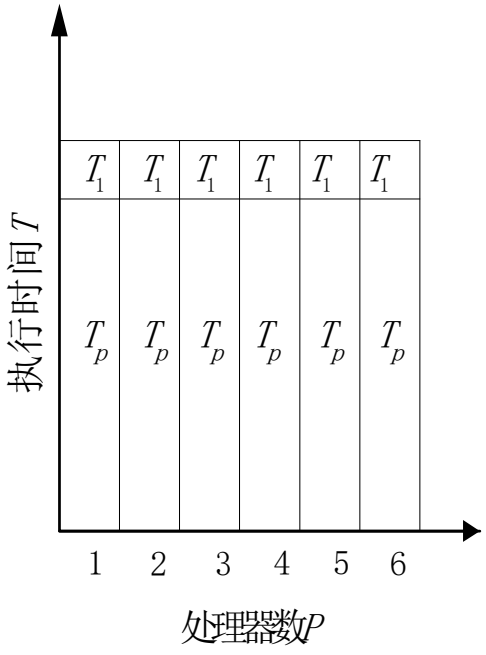
- 并行开销W o：

$$S' = f + p(1\text{-}f) = p + f(1\text{-}p) = p\text{-}f(p\text{-}1)$$

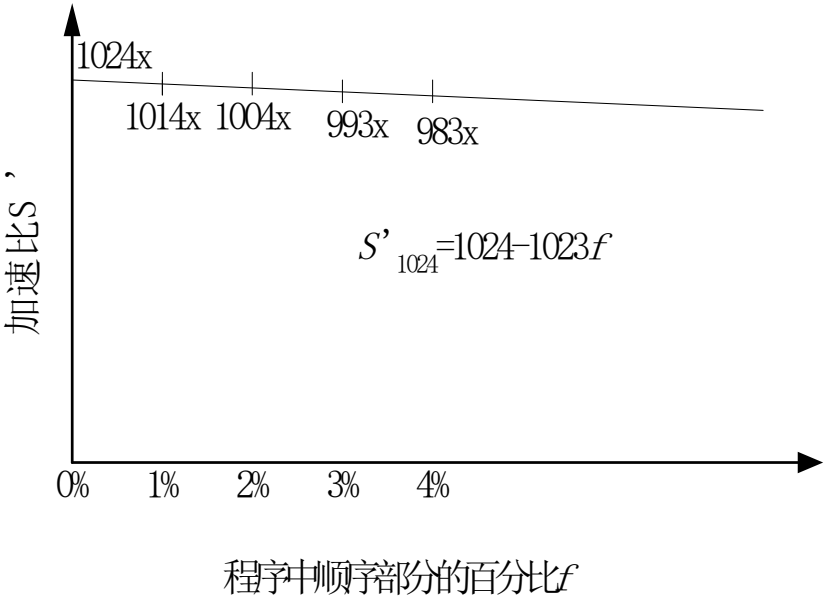
$$S' = \frac{W_S + pW_P}{W_S + W_P + W_O} = \frac{f + p(1-f)}{1 + W_O / W}$$


工作负载W
W_1
W_p
1 2 3 4 5 6
处理器数P

(a)


执行时间T
T_1
T_p
1 2 3 4 5 6
处理器数P

(b)


加速比S'
1024x
1014x 1004x 993x 983x
$S'_{1024}=1024-1023f$
0% 1% 2% 3% 4%
程序中顺序部分的百分比f

(c)

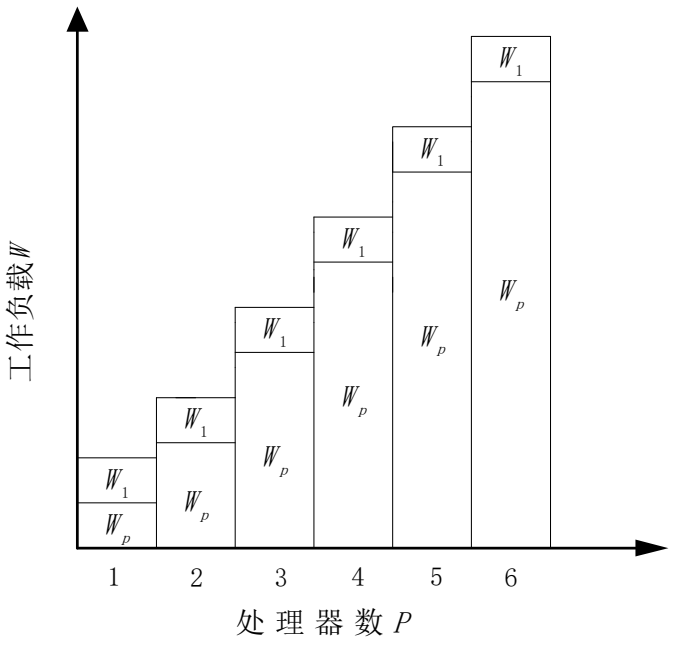
2.2.4 Sun 和 Ni定律

- 基本思想：
  - 只要存储空间许可，应尽量增大问题规模以产生更好和更精确的解（此时可能使执行时间略有增加）。
  - 假定在单节点上使用了全部存储容量M并在相应于W的时间内求解之，此时工作负载W= fW +（1-f）W。
  - 在p 个节点的并行系统上，能够求解较大规模的问题是因为存储容量可增加到pM。令因子G（p）反应存储容量增加到p倍时并行工作负载的增加量，所以扩大后的工作负载W = fW +（1-f）G（p）W。
- 存储受限的加速公式：

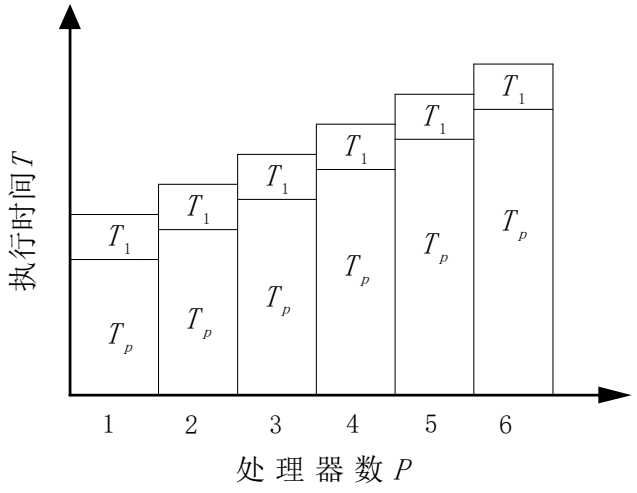
$$S'' = \frac{fW + (1-f)G(p)W}{fW + (1-f)G(p)W/p} = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p}$$

- 并行开销W o:

$$S' = \frac{fW + (1-f)WG(p)}{fW + (1-f)G(p)W/p + W_O} = \frac{f + (1-f)G(p)}{f + (1-f)G(p)/p + W_O/W}$$



(a)



(b)

- **G（p）=1时就是Amdahl加速定律；**
- **G（p）=p 变为 f + p（1-f），就是Gustafson加速定律**
- **G（p）>p时，相应于计算机负载比存储要求增加得快，此时 Sun和 N i 加速均比 Amdahl 加速和 Gustafson 加速为高。**

加速比讨论

- 参考的加速经验公式： $p/\log p \le S \le P$
- 线性加速比：很少通信开销的矩阵相加、内积运算等
- $p/\log p$的加速 比：分治类的应用问题
- 通信密集类的应用问题 ：$S = 1 / C （ p ）$
- 超线性加速
- 绝对加速：最佳并行算法与串行算法
- 相对加速：同一算法在单机和并行机的运行时间

并行算法的设计基础

3. 并行化方法

并行化方法

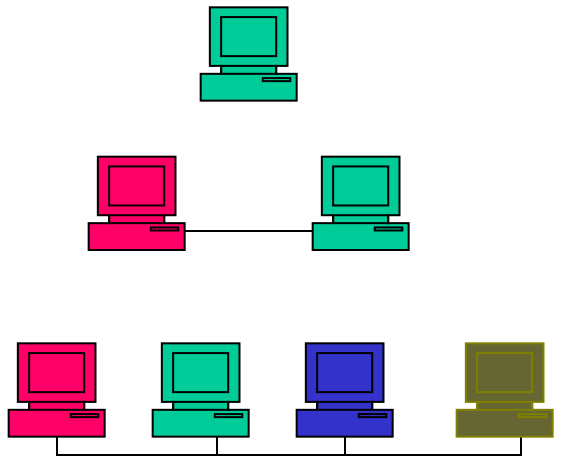
- 分而治之
  - 模型并行、算法并行、程序并行
  - 自然并行：数据并行、功能分割
  - 任务主从型、SPMD型、管道型（流水线）、二叉树型（分解与递归）

分而治之

$Y=(A+B(C+DEF))+G$ 6

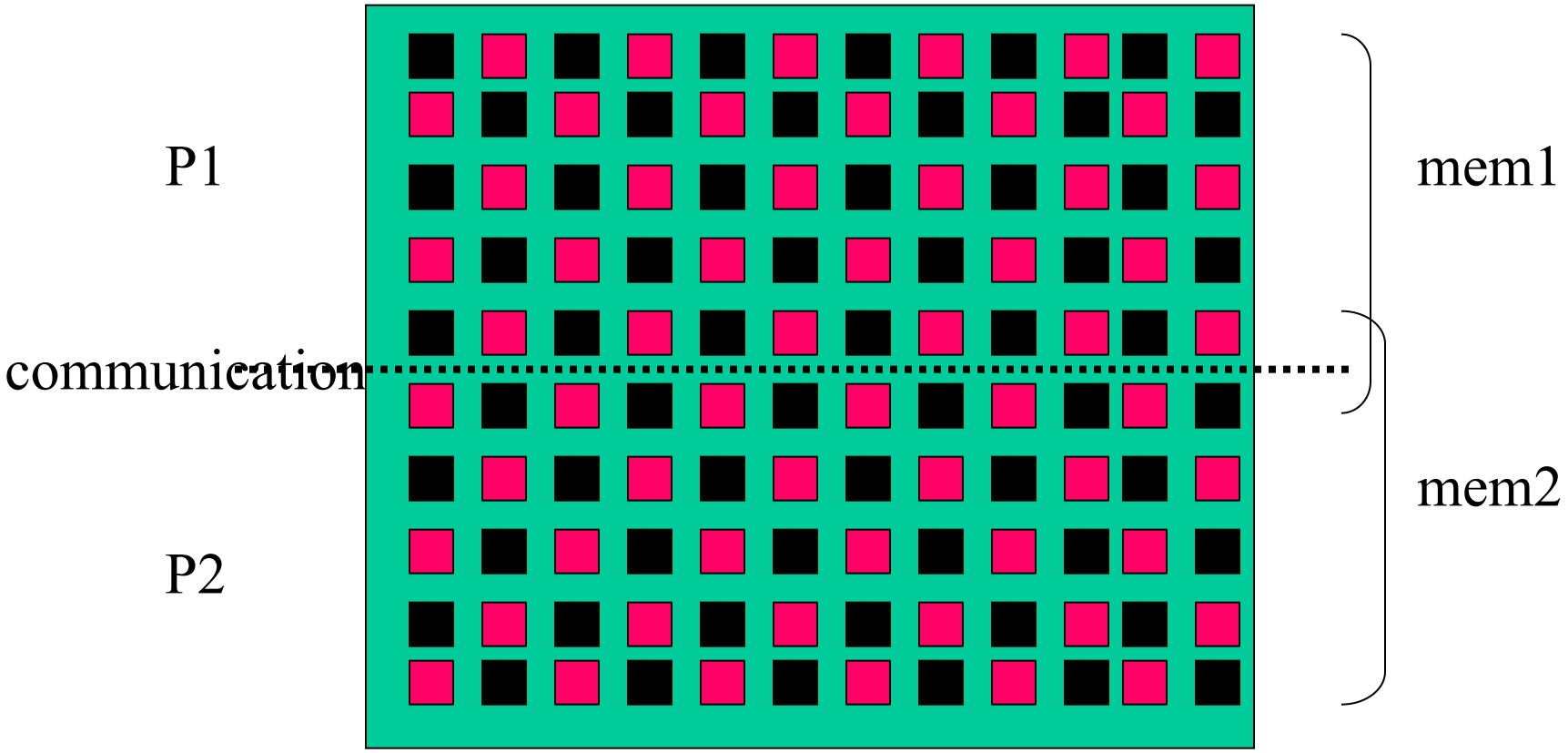
$Y=(A+G)+B(C+DEF)$ 5

$Y=(A+G+BC)+BD*EF$ 3



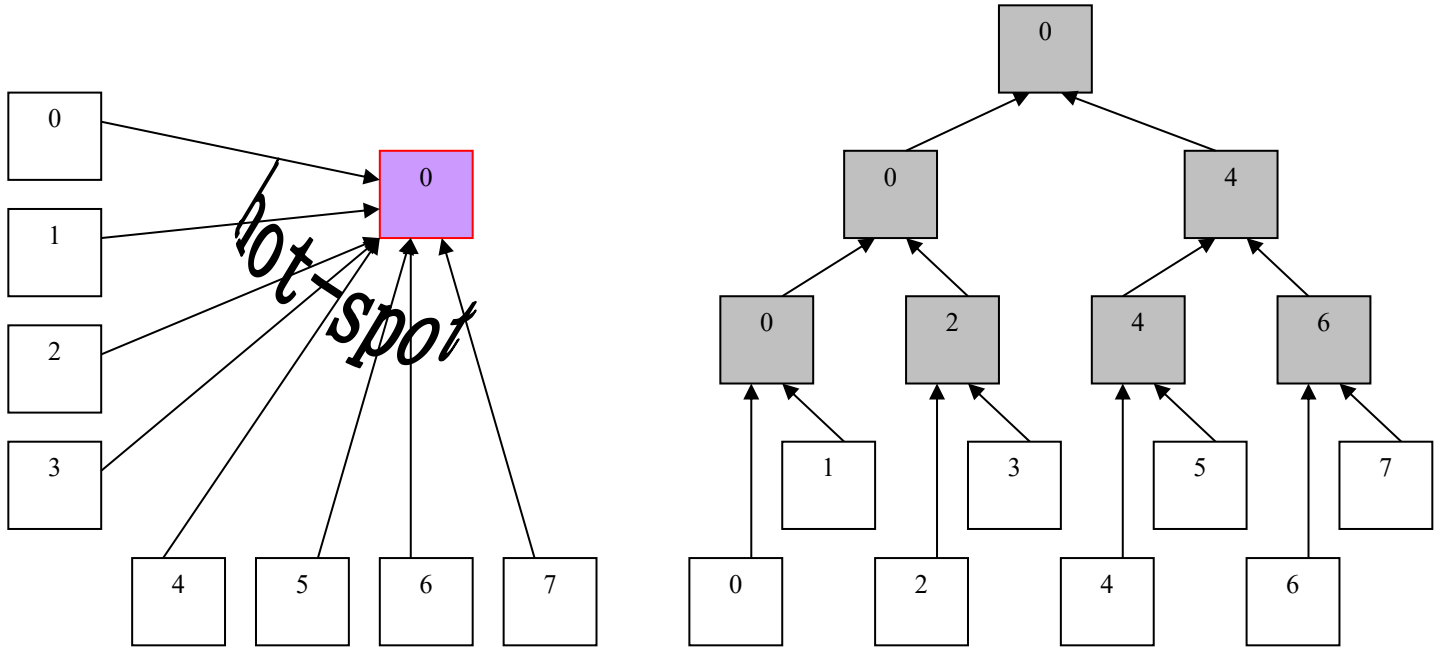
并行化方法

红黑格法



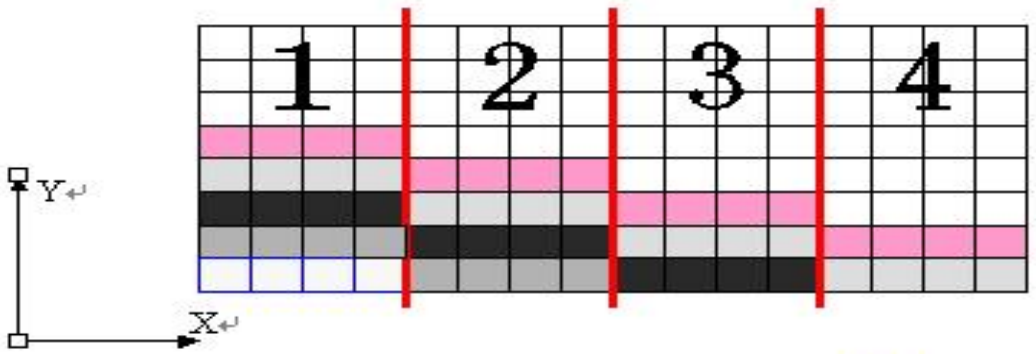
并行化方法

- 并行二叉树技术：解决通信瓶颈问题，通信复杂度*O(P)*下降到*O(log P)*



并行化方法

- **并行流水线技术：** 解决串行算法中不可避免的数据相关性，例如Gauss-Seidel迭代、双曲型方程中的上下游流场数据依赖问题等



1. P1 计算线（I=1-4，J=1），P2-P4 空闲；
2. P1 计算线（I=1-4，J=2），P2 计算线（I=1-4，J=1），P3-P4 空闲；
3. P1 计算线（I=1-4，J=3），P2 计算线（I=1-4，J=2），P3 计算线（I=1-4，J=1），P4 空闲；
4. P1 计算线（I=1-4，J=4），P2 计算线（I=1-4，J=3），P3 计算线（I=1-4，J=2），P4 计算线（I=1-4，J=1）；
5. P1 计算线（I=1-4，J=5），P2 计算线（I=1-4，J=4），P3 计算线（I=1-4，J=3），P4 计算线（I=1-4，J=2）；
6. 依次类推，可并行计算，只要 Y 方向网格点数足够多；

如何做并行应用？

- 串行算法分析：模块化
- 串行软件测试：找消耗90％CPU的hot
- 选择并行方法：分而治之——从物理模型开始
- 并行软件框架设计：流水线/主从/平等
- 并行程序设计：MPI/PVM/OPEN MP等
- 性能测试与分析：正确性、效率
- 进一步优化：应用

并行算法设计的基本原则

- 与体系结构相结合
- 具有可扩展性
- 粗粒度
- 减少通信
- 优化性能

并行算法的一般设计方法

- 串行算法的直接并行化
- 从问题描述开始设计并行算法
- 借用已有算法解新问题

并行算法的一般设计方法

- 串行算法的直接并行化

检测和开拓串行算法中固有的并行性而直接将其并行化。

- 由串行算法直接并行化是并行算法设计最常用的方法之一
- 对于一类具有内在顺序性的串行算法, 难以直接并行
- 并非任何优秀的串行算法都可以产生好的并行算法，相反一个不好的串行算法可能产生很优秀的并行算法
- 许多数值串行并行算法可以并行化为有效的数值并行算法

并行算法的一般设计方法

- 从问题描述开始设计并行算法

  从问题的描述出发，寻求可能的新途径，设计出一个新的并行算法。不是完全排除串行算法设计的基本思想，而是更着重从并行化的具体实现上开辟新的设计方法

--设计全新的并行算法是一个挑战性和创新性的工作

并行算法的一般设计方法

- 借用已有算法解新问题：

借用已知的某类问题的求解算法求解另一问题。

例：利用矩阵乘法求所有点对间最短路径。

n个顶点加权有向图G(V,E)，矩阵Wnxn,构造矩阵D,di,j从v_i到v_j 的最短路径长度。

d^k_{ij}表示v_i到v_i 的经过至多-1个点的最短路径长度

$D^k_{ij}=W_{i,j}$

$d^k_{ij}=\min\{d^{k/2}_{il}, d^{k/2}_{lj}\}$。因此D可以经由$D^1$逐次计算$D^2$，$D^4$，… D^{n-1}，然后由$D= D^{n-1}$即可。由$D^{k/2}$求D^k，可以使用标准矩阵乘法，只是将原改为‘+’；求和运算改为“min”操作。

并行算法的基本设计技术

- 区域分解
- 功能分解
- 流水线技术
- 同步并行算法
- 异步并行算法

并行算法的基本设计技术

- 区域分解法

  按区域进行分解的一种方法,早期应用于求解椭圆型偏微 分方程

  例:求解矩形区域上的Lapalace问题

$$\begin{cases} -\Delta u(x,y) = f(x,y) \ \ (x,y) \in \Omega = (0,a) \times (0,b) \\ u(x,y)\big|_{\partial\Omega} = g(x,y) \end{cases}$$

其中, f(x, y) 和 g(x, y) 为已知函数分别定义在区域的内部和边界

- 非重叠区域的分解

(a) 一维条分解

(b) 二维块分解

- 将离散化后的方程化成一些独立的小规模问题和一个与每个小问题关联的全局问题

并行算法的基本设计技术

- 功能分解方法

将不同功能组成的问题, 按照其功能进行分解的一种手段, 其目的是逐一解决不同功能的问题, 从而获得整个问题的解

假设F(x)是$D \subset R^n$到D的一个映射,要求解x^*,使得x^*是方程F(x)=0的一个解.记F(x)的Jacobi矩阵为G(x)=F'(x),对给定的初始值$x^{(0)}$,则Newton迭代法如下:

$$x^{(k+1)} = x^{(k)} - G^{-1}(x^{(k)})F(x^{(k)}), \quad k = 0, \ldots\ldots$$

并行算法的基本设计技术

- 流水线技术

  流水线（pipelining)技术是一项重要的并行技术，基本思想：将一个任务t分成一系列子任务$t_1, t_2, \cdots, t_m$, 使得一旦t_1完成，后继的子任务就立即开始，并以同样的速率进行计算

以求解一簇递推问题为例:

$$a_{0,j}\text{给定}, \ a_{i,j} = F(a_{i-1,j}), \ i = 1,.....,n, \quad j = 1,.....,m$$

	P_0	P_1	P_2
第 1 拍	计算 $a_{i,1} := F(a_{i-1,1})$, $i = 1, \ldots, n_1 - 1$, 发送 $a_{n_1-1,1}$ 给 P_1	等待接收 $a_{n_1-1,1}$	等待接收 $a_{n_2-1,1}$
第 2 拍	计算 $a_{i,2} := F(a_{i-1,2})$, $i = 1, \ldots, n_1 - 1$, 发送 $a_{n_1-1,2}$ 给 P_1	计算 $a_{i,1} := F(a_{i-1,1})$, $i = n_1, \ldots, n_2 - 1$, 发送 $a_{n_2-1,1}$ 给 P_2	等待接收 $a_{n_2-1,1}$
第 3 拍	计算 $a_{i,3} := F(a_{i-1,3})$, $i = 1, \ldots, n_1 - 1$, 发送 $a_{n_1-1,3}$ 给 P_1	计算 $a_{i,2} := F(a_{i-1,2})$, $i = n_1, \ldots, n_2 - 1$, 发送 $a_{n_2-1,2}$ 给 P_2	计算 $a_{i,1} := F(a_{i-1,1})$, $i = n_2, \ldots, n$
⋮	⋮	⋮	⋮
第 m 拍	计算 $a_{i,m} := F(a_{i-1,m})$, $i = 1, \ldots, n_1 - 1$, 发送 $a_{n_1-1,m}$ 给 P_1	计算 $a_{i,m-1} := F(a_{i-1,m-1})$, $i = n_1, \ldots, n_2 - 1$, 发送 $a_{n_2-1,m-1}$ 给 P_2	计算 $a_{i,m-2} := F(a_{i-1,m-2})$, $i = n_2, \ldots, n$
第 m+1 拍	空闲	计算 $a_{i,m} := F(a_{i-1,m})$, $i = n_1, \ldots, n_2 - 1$, 发送 $a_{n_2-1,m}$ 给 P_2	计算 $a_{i,m-1} := F(a_{i-1,m-1})$, $i = n_2, \ldots, n$
第 m+2 拍	空闲	空闲	计算 $a_{i,m} := F(a_{i-1,m})$, $i = n_2, \ldots, n$

并行算法的基本设计技术

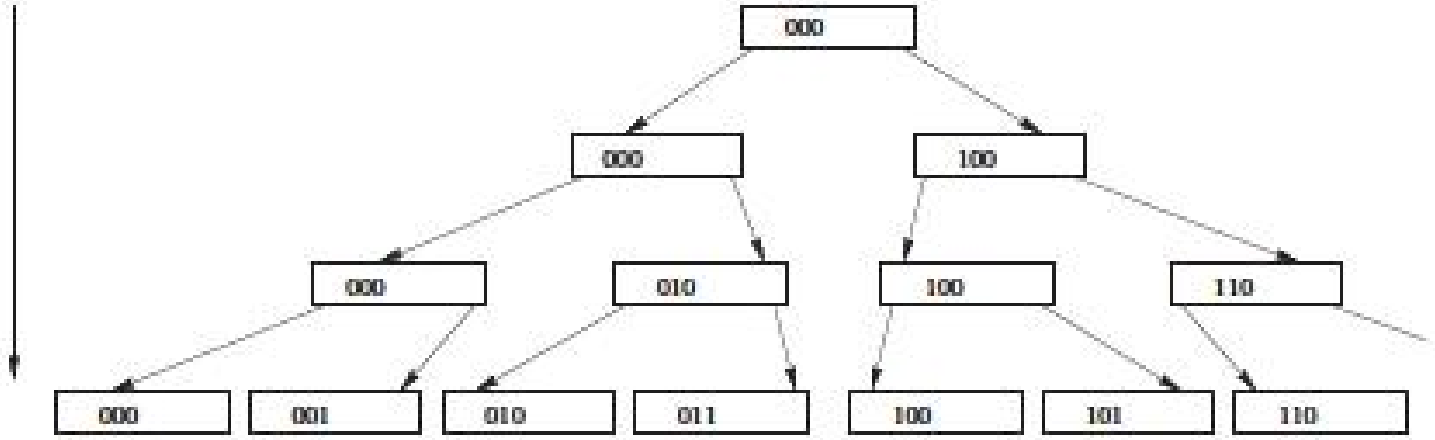
- 分而治之方法

  将一个大而复杂的问题分解成若干特征相同的子问题分而治之

例：假设在 $q=2^3$ 个处理机上计算：$s=\sum_{i}^{n-1} a_i$

$$
\begin{cases}
s_{000}(0:n) = s_{000}(0:n/2) + s_{100}(n/2:n/2) \\
s_{000}(0:n/2) = s_{000}(0:n/4) + s_{010}(n/4:n/4) \\
s_{100}(n/2:n/2) = s_{100}(n/2:n/4) + s_{110}(3n/4:n/4) \\
s_{000}(0:n/4) = s_{000}(0:n/8) + s_{001}(n/8:n/8) \\
s_{010}(n/4:n/4) = s_{010}(n/4:n/8) + s_{011}(3n/8:n/8) \\
s_{100}(n/2:n/4) = s_{100}(n/2:n/8) + s_{101}(5n/8:n/8) \\
s_{110}(3n/4:n/4) = s_{110}(3n/4:n/8) + s_{111}(7n/8:n/8)
\end{cases}
$$

并行算法的基本设计技术



- 从上至下是分解过程, 从下至上是求部分和的过程, 是一个简单的有分有治的过程

并行算法的基本设计技术

- 同步和异步并行算法

  同步并行算法需要在某一时刻需要与其它的处理机进行数据交换, 然后才能继续进行. 异步并行算法进行数据交换不需要严格确定在某一时刻, 每个处理机按照预定的计算任务持续执行, 但通常需要在一定的时候必须进行一次数据交换, 以保证算法的正确性

例:

$$x^{(k+1)} = b + Ax^{(k)}$$

$x^{(0)}$已知; A是$n \times n$矩阵; $b, x^{(k)}$是n维向量;

矩阵在处理机中是按照行分块存储

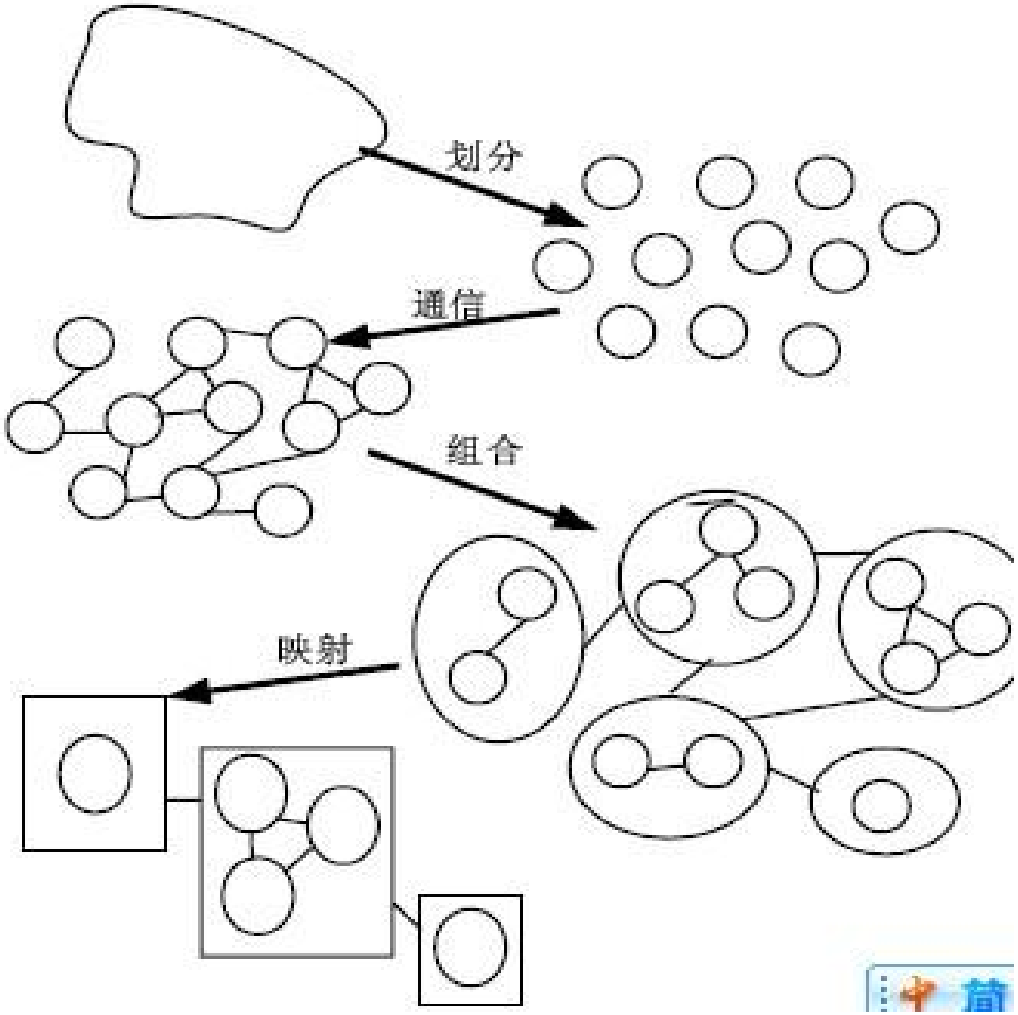
并行算法的一般设计过程

- PCAM设计方法学

1) 首先尽量开拓算法的并发性和满足算法的可扩放性(与算法相关的特性)
2) 然后着重优化算法的通信成本和全局执行时间(与机器相关的特性)
3) 同时通过必要的整个过程的反复回溯,以期望达到一个满意的设计选择

PCAM方法设计

- PCAM分为四步
  1）任务的划分
  2）通信
  3）任务组合
  4）处理器映射



并行算法的一般设计过程:划分

- 使用域分解或者功能分解将整个计算分解成一些小的任务，以便充分利用其潜在的并行性和可扩放性。
- 先集中数据的分解（域分解），然后是计算功能的分解（功能分解），两者互为补充。
- 要点：计算集、数据集互补相交，以避免数据和计算的复制

并行算法的一般设计过程:划分

- 划分标准
  - 任务数，是否至少高于目标机上处理器数的一个量级。（灵活性）
  - 是否避免了冗于的计算和存储要求。（扩放）
  - 划分的任务是否尺寸大致相当。（均衡）
  - 任务数是否与问题尺寸成比例。
  - 是否采用了几种不同的划分法，多考虑几种选择可提高灵活性，同时既考虑域分解，又要考虑功能分解。

并行算法的一般设计过程:通信

四种模式

- 局部通信vs.全局通信
- 结构化通信vs.非结构化通信
- 静态通信vs.动态通信
- 同步通信vs.异步通信

并行算法的一般设计过程:通信

- 通信标准
  - 所有任务是否执行大致同样多的通信。(可扩放性)
  - 每个任务是否只与少许近邻通信
  - 诸通信操作是否能并行执行
  - 不同任务的计算能否并行执行

并行算法的一般设计过程:组合

- 目的：合并小尺寸的任务以减少任务数，理想情况每个处理器一个任务，得到SPMD程序。
- 增加粒度：
- 在划分阶段,致力于尽可能多的任务以增大并行执行的机会.但定义大量的细粒度任务不一定产生一个有效的算法,因为这有可能增加通信的代价和任务创建的代价
- 表面-容积效应：通信量比例于子域的表面积，计算比例于容积，通信/计算之比随任务的尺寸的增加而减少->增加粒度
- 重复计算（Replication Computation）,也叫冗余计算,有时可用冗余计算来减少通信。
- 同时也要保持灵活性和减少软件成本,降低软件工程代价

并行算法的一般设计过程:组合

- 组合标准
  - 组合造成的重复计算，是否平衡了其收益？
  - 造成重复数据，是否已证实不会因限制问题尺寸和处理机数目而影响可扩放性？
  - 组合产生的任务是否具有类似的计算、通信代价？
  - 任务数目是否仍与问题尺寸成比例？

并行算法的一般设计过程:映射

- 指定每个任务到哪里执行，目的，减少算法的总的执行时间
- 策略

  1)可并发执行的任务放在不同的处理器上，增强并行度

  2)需要频繁通信的任务置于同一处理器上以提高局部性。

  3)采用域分解技术，当分解算法复杂，工作量不一样，通信也许是非结构化的，此时需要负载平衡算法。

  4)基于功能分解，会产生一些由短暂任务组成的计算，它们在开始与结束时需与别的任务协调，此时可采用任务调度算法。

并行算法的一般设计过程:映射

- 标准
  - 采用集中式负载平衡方案，管理者是否会成为瓶颈？
  - 动态平衡方案，是否衡量过不同策略的成本？
  - 采用概率或者循环法，是否有足够多的任务保证负载平衡？
  - 设计SPMD程序，是否考虑过基于动态任务创建和消除的算法？后者可以得到简单的算法，但性能可能有问题。

并行程序设计方法

应用领域	科学和工程计算，数据处理，商务应用等	
	串　行　程　序　设　计	并行程序设计
算法范例	分而治之，分支界限 动态规划，回，贪心	计算交互、工作池、异步迭代 流水线、主-从，细胞子动机
编程模型	冯诺依曼	隐式并行、数据并行 共享变量、消息传递
编程语言	Fortran,C,Cobol,4GL	Fortran90,HPF,X3H5、PVM\MPI
体系结构	不同类型的单处理机	共享内存（PVP、SMP、DSM） 数据并行(SIMD)、 消息传递（MPP、Clusters）

并行程序设计方法

- 隐式并行程序设计

  1)常用传统的语言编程成顺序源码,把”并行”交给编译器实现自动并行

  2)程序的自动并行化是一个理想目标,存在难以克服的困难

  3)语言容易,编译器难

- 显示并行程序设计

  1)在用户程序中出现”并行”的调度语句

  2)显示的并行程序开发是解决并行程序开发困难的切实可行的方法

  3)语言难,编译器容易

并行程序设计模型

- 隐式并行
- 数据并行
- 共享变量
- 消息传递

隐式并行

- 概况
  1) 程序员用熟悉的串行语言编程(未作明确的指定并行性)
  2) 编译器和运行支持系统自动转化为并行代码

- 特点
  1) 语义简单
  2) 可移植性好
  3) 单线程,易于调试和验证正确性
  4) 细粒度并行
  5) 效率很低

数据并行

- 概况
  1) SIMD的自然模型
  2) 局部计算和数据选路操作
- 特点
  1) 单线程
  2) 并行操作于聚合数据结构(数组)
  3) 松散的同步
  4) 单一地址空间
  5) 隐式交互作用
  6) 显示数据分布
- 优点:编程相对简单,串并行程序一致
- 缺点:程序的性能依赖于所用的编译系统及用户对编译系统的了解,并行粒度局限于数据级并行,粒度较小

共享变量

- 概况

  PVP,SMP,DSM的自然模型
- 特点

  1) 多线程:SPMD,MPMD

  2) 异步

  3) 单一地址空间

  4) 显示同步

  5) 隐式数据分布

  6) 隐式通信
- 典型代表

  OpenMP

消息传递

- 概况

  MPP的自然模型
- 特点

  1) 多线程

  2) 异步

  3) 多地址空间

  4) 显示同步

  5) 显式数据映射和负载分配

  6) 显示通信
- 典型代表

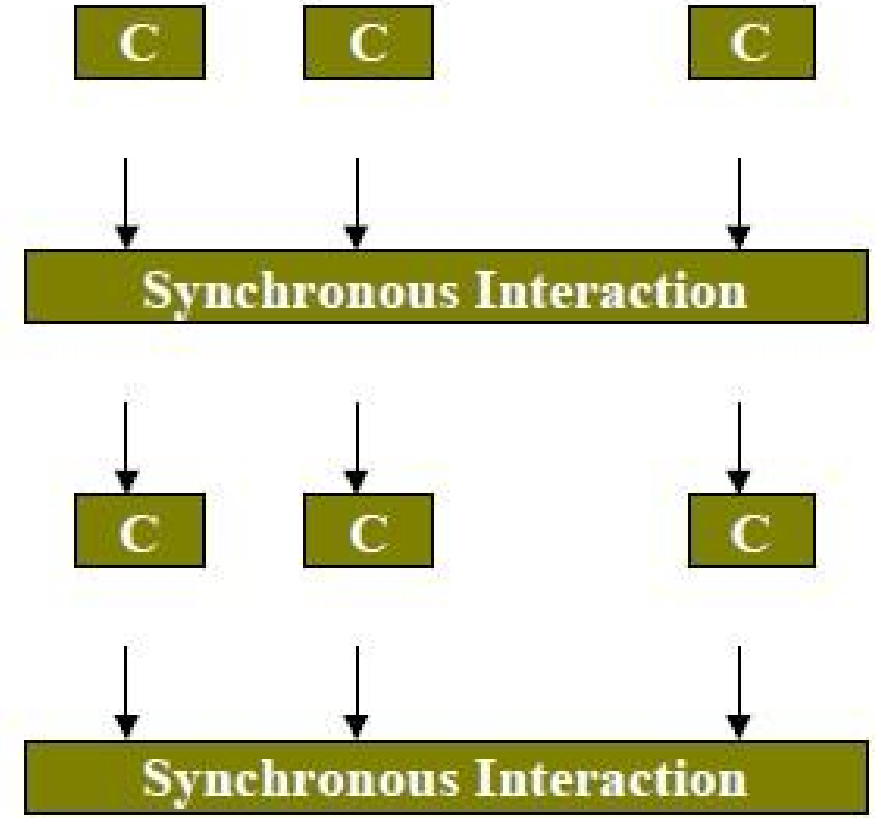
  MPI,PVM

并行编程风范

- 并行编程风范(Parallel Programming Paradigms):构造运行在并行机上的并行算法的方法
  - 相并行（Phase Parallel），也叫松散并行
  - 分治并行(Divide and Conquer Parallel)
  - 流水线并行(Pipeline Parallel)
  - 主-从并行(Master-Slave Parallel)
  - 工作池并行(Work Pool Parallel)

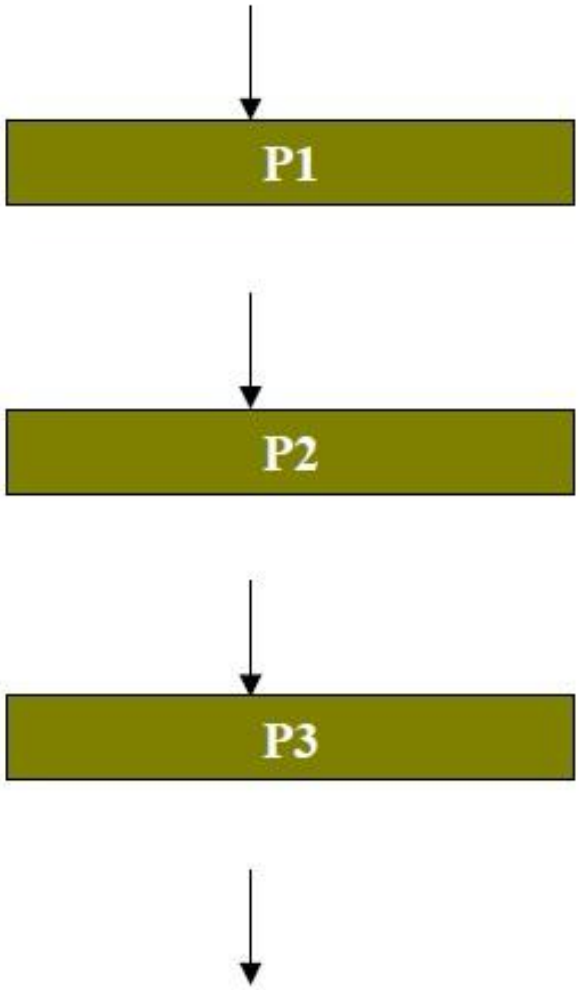
相平行

- 一组超级步(相)
- 步内各自计算
- 步间通信同步
- 方便差错和性能分析
- 计算和通信不能重叠



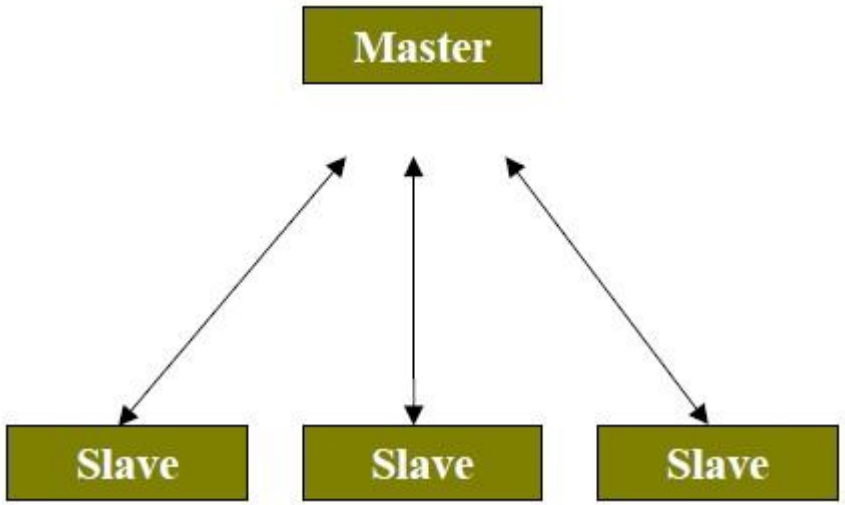
流水线并行

- 将一个任务t分成一系列子任务t1, t2, …, tm, 使得一旦t1完成, 后继的子任务就立即开始, 并以同样的速率进行计算
- 一组进程
- 流水线作业
- 流水线设计技术



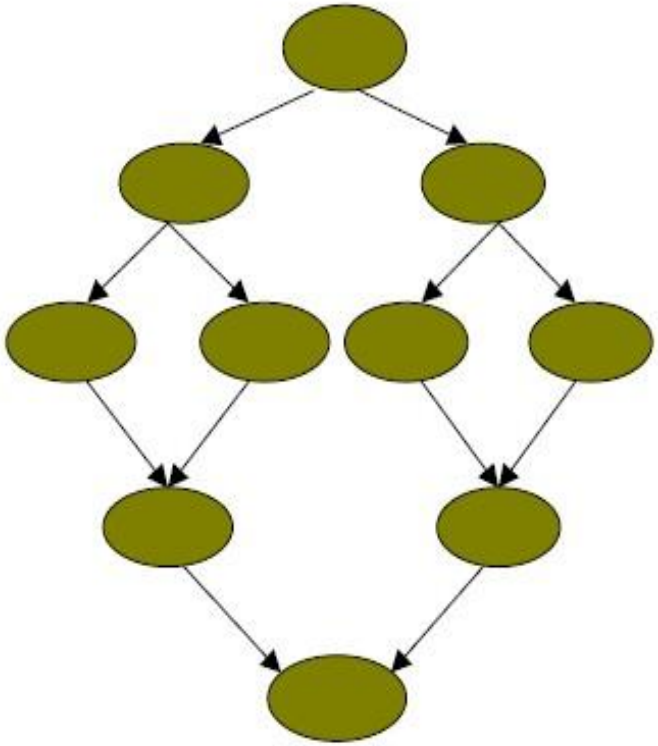
主-从并行

- 主进程:串行, 协调任务
- 子进程:计算子任务
- 划分设计技术
- 与相并行结合
- 主进程易成为瓶颈



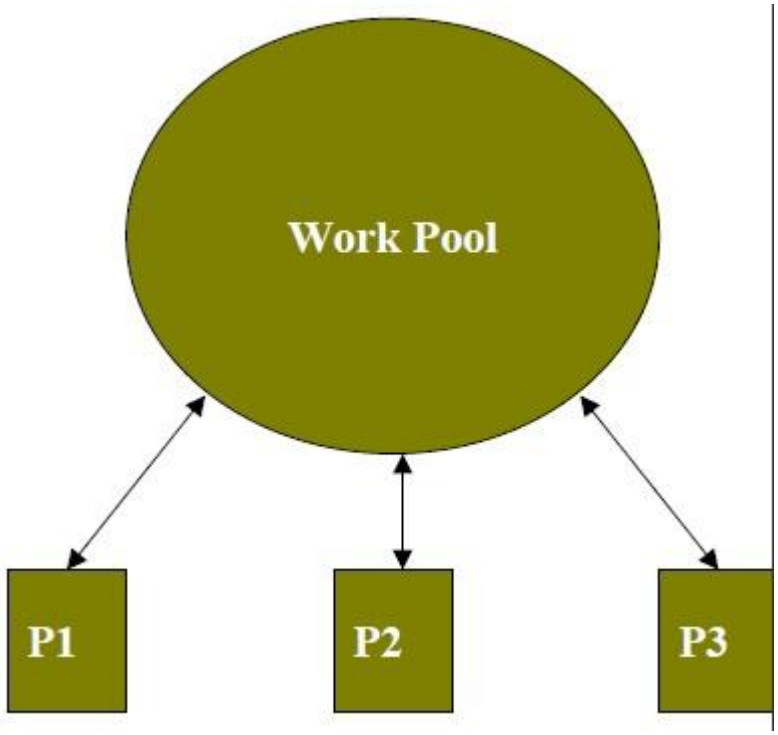
分治并行

- 将一个大而复杂的问题分解成若干特征相同的子问题分而治之
- 父进程把负载分割并指派给子进程
- 重点在于归并
- 分治设计技术
- 难以负载平衡



工作池并行

- 初始状态:一件工作
- 进程从池中取任务执行
- 可产生新任务放回池中
- 直至任务池为空
- 易于负载平衡



参考文献

- 黄铠,徐志伟著,陆鑫达等译. *可扩展并行计算技术,结构与编程*. 北京:机械工业出版社, P.33~56,P.227~237, 2000.
- 陈国良著.*并行计算—结构、算法、编程*. 北京:高等教育出版社,1999.
- Barry Wilkinson and Michael Allen. *Parallel Programming(Techniques and Applications using Networked Workstations and Parallel Computers)*. Prentice Hall, 1999.
- 李晓梅,莫则尧等著. *可扩展并行算法的设计与分析*. 北京:国防工业出版社,2000.
- 张宝琳,谷同祥等著. *数值并行计算原理与方法*. 北京:国防工业出版社,1999.
- 都志辉著. *高性能计算并行编程技术—MPI并行程序设计*. 北京:清华大学出版社, 2001.

相关网址

- MPI: http://ww.mpi-forum.org，
  http://www.mcs.anl.gov/mpi
- Pthreads: http://www.oreilly.com
- PVM: http://www.epm.ornl.gov/pvm/
- OpemMP: http://www.openmp.org
- 网上搜索:www.google.com

谢谢！