



中山大學

SUN YAT-SEN UNIVERSITY

并行硬件和并行软件

胡淼

中山大学计算机学院

2023 年 3 月

课程内容

- Peter Pacheco(著), 并行程序设计导论, 机械工业出版社, 2022.
 - 第二章 并行硬件和并行软件
- 陈国良, 吴俊敏(著), 并行计算机体系结构(第2版), 高等教育出版社, 2021.
 - 第二章 性能评测

Roadmap

- 2.1 背景知识
- 2.2 对冯.诺依曼模型的改进
- 2.3 并行硬件
- 2.4 并行软件
- 2.5 输入和输出
- 2.6 性能的评价
- 2.7 并行程序设计
- 总结

2.4 并行软件

- 并行硬件提供了并行软件的基础
- 通常
 - 在共享内存系统中：
 - 启动一个单独的进程，然后派生（fork）多个线程
 - 线程执行任务
 - 在分布式内存系统中：
 - 启动多个进程
 - 进程执行任务

单程序多数据流程序SPMD

- **SPMD**程序不是在每个核上运行不同的程序
- 在执行时，通过使用条件转移语句，表现得像是在不同处理器上执行不同的程序
- 不仅仅可以数据并行，也可以任务并行

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



编写并行程序

1. 将任务在进程/线程之间分配
 - (a) 使得每个进程/线程获得大致相等的工作量
 - (b) 使得需要的通信量最小
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];
```

2.4.3 共享内存Shared Memory

- 动态线程Dynamic threads

- 主线程通常等待工作请求。当请求到达时，派生出一个工作线程来执行该请求；当线程工作完成时，就会终止执行再合并到主线程中去
- 充分利用了系统的资源
- 但是线程的创建和终止消耗了时间

- 静态线程Static threads

- 主线程在完成必要的设置之后，派生出所有的线程，并且在工作结束前所有的线程都在运行
- 更好的执行，但是浪费系统资源

非确定性（Nondeterminism）

...

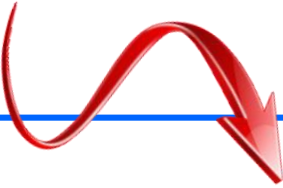
```
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x ) ;
```

...



Thread 1 > my_val = 19

Thread 0 > my_val = 7



Thread 0 > my_val = 7

Thread 1 > my_val = 19

非确定性（Nondeterminism）

```
my_val = Compute_val ( my_rank ) ;  
x += my_val ;
```

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19

非确定性（Nondeterminism）

- 竞争条件
- 临界区
- 互斥
- 互斥锁 (**mutex, or simply lock**)

```
my_val = Compute_val ( my_rank ) ;  
Lock(&add_my_val_lock ) ;  
x += my_val ;  
Unlock(&add_my_val_lock ) ;
```

忙等待-信号量

busy-waiting

```
my_val = Compute_val ( my_rank ) ;  
if ( my_rank == 1)  
    while ( ! ok_for_1 ) ; /* Busy-wait loop */  
x += my_val ; /* Critical section */  
if ( my_rank == 0)  
    ok_for_1 = true ; /* Let thread 1 update x */
```

2.4.4 分布式内存

消息传递API

```
char message [ 1 0 0 ] ;
```

```
...
```

```
my_rank = Get_rank ( ) ;
```

```
if ( my_rank == 1 ) {
```

```
    printf ( message , "Greetings from process 1" ) ;
```

```
    Send ( message , MSG_CHAR , 100 , 0 ) ;
```

```
} elseif ( my_rank == 0 ) {
```

```
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
```

```
    printf ( "Process 0 > Received: %s\n" , message ) ;
```

```
}
```

划分全局地址空间语言PGAS

Partitioned Global Address Space Languages

```
shared int n = ... ;  
shared double x [ n ] , y [ n ] ;  
private int i , my_first_element , my_last_element ;  
my_first_element = ... ;  
my_last_element = ... ;  
/ * Initialize x and y */  
...  
f o r ( i = my_first_element ; i <= my_last_element ; i++)  
    x [ i ] += y [ i ] ;
```

2.5 输入和输出

- 输入和输出的问题非常复杂
- **stdin**---标准输入文件
- **stdout**---标准输出文件
- **stderr**---标准出错文件

规则

- 在分布式内存程序中，只有进程**0**能够访问 ***stdin***。在共享内存程序中，只有主线程或者线程**0**访问 ***stdin***
- 两种系统中，所有进程/线程都可以访问 ***stdout***和***stderr***

- 因为输出到 `stdout` 的非确定性顺序，只有一个进程/线程结果输出到 `stdout`
- 调试程序时，允许多个进程/线程写 `stdout`。并且输出的记过应该包括进程/线程的序号或者进程标识符
- 只有一个进程/线程会尝试访问一个除 `stdin`, `stdout` 或者 `stderr` 外的文件

2.6 性能

2.6.1 加速比和效率

- 核的数量 = p
- 串行运行时间 = $T_{\text{串行}}$
- 并行运行时间 = $T_{\text{并行}}$

$$T_{\text{并行}} = T_{\text{串行}} / p$$

并行程序的加速比

$$S = \frac{T_{\text{串行}}}{T_{\text{并行}}}$$

并行程序的效率

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{串行}}}{T_{\text{并行}}} \right)}{p} = \frac{T_{\text{串行}}}{p \cdot T_{\text{并行}}}$$

并行开销

$$T_{\text{并行}} = T_{\text{串行}} / p + T_{\text{开销}}$$

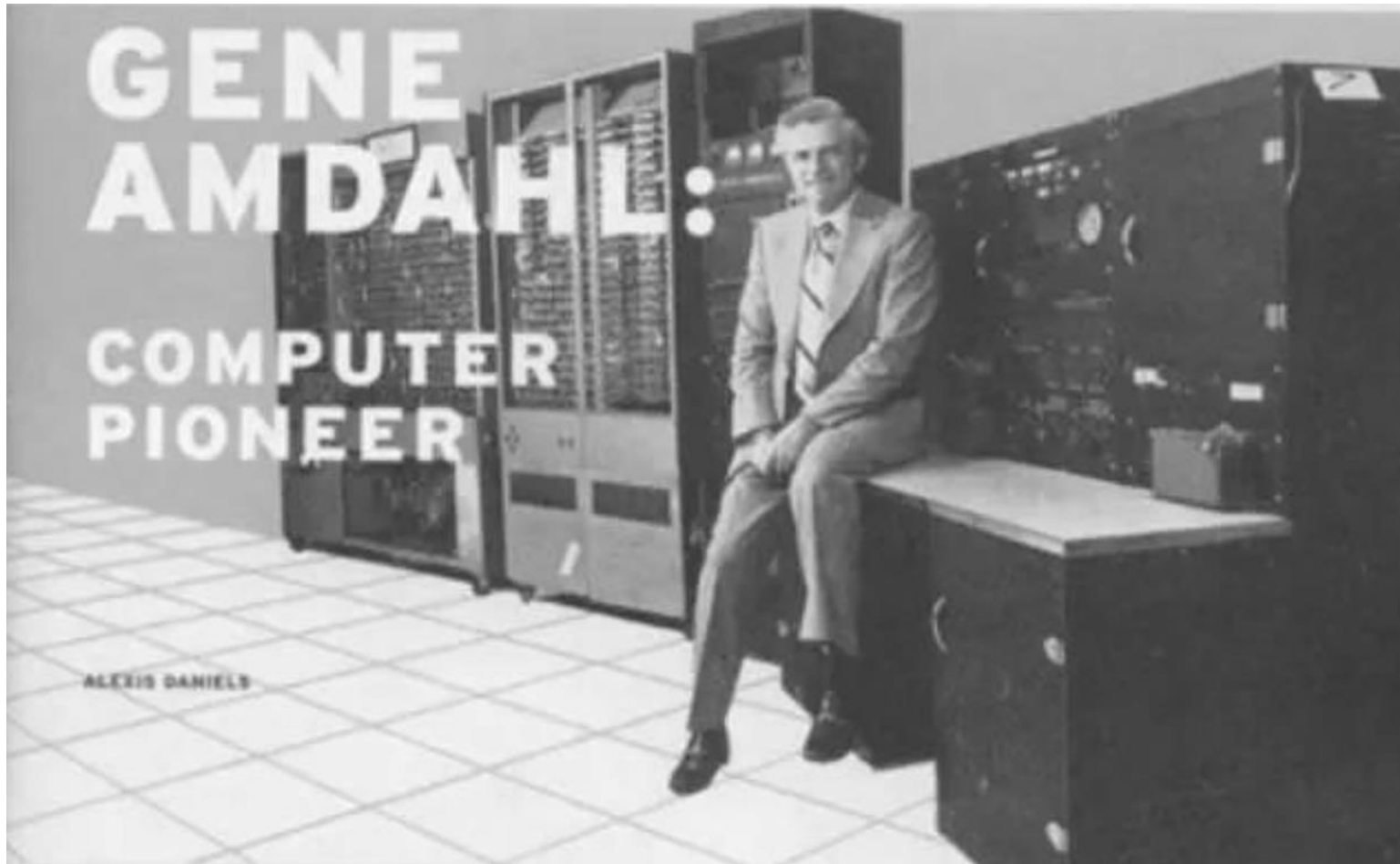
2.6.2 Amdahl定律

- 大致上，除非一个串行程序的执行几乎全部都并行化，否则，不论有多少可以利用的核，通过并行化所产生的加速比都会受限

Amdahl定律

- Amdahl 定律
 - 1967 年，作为 IBM360 系列机的主要设计者，美国计算机科学家吉恩·阿姆达尔（Gene Amdahl）提出了 Amdahl 定律
 - Amdahl 定律给出了**固定负载**条件下程序并行化效率提升的理论上限
 - 在并行计算领域，Amdahl 定律常用于预测使用多个处理器后的理论加速比上限

Amdahl定律



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)
 - W_s : 应用程序中的串行分量, f 是串行分量比例 ($f = W_s/W$)
 - W_p : 应用程序中可并行化部分, $1-f$ 为并行分量比例
 - $W_s + W_p = W$
- $T_s = T_1$: 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
 - 增加处理器加快执行速度, 从而达到了加速的目的

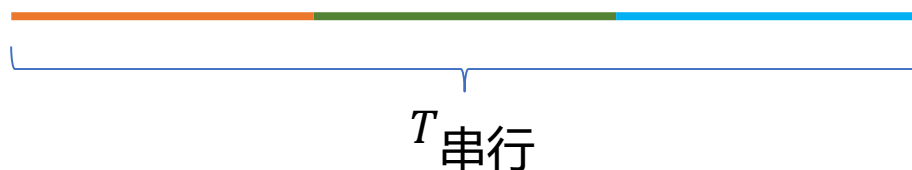
加速比

- 一个并行程序的加速比 (Speedup) :
 - T_p : 并行执行时间
 - T_s : 串行执行时间

$$S = \frac{T_s}{T_p}$$

线性加速比

- 加速前：



- 在 3 核的计算机：



- 对于 p 核的高性能计算机：

$$T_{\text{并行}} = \frac{T_{\text{串行}}}{p}$$

此时，并行程序拥有**线性加速比 (p)**

Amdahl定律

- 固定负载的加速公式: $S = \frac{W_s + W_p}{W_s + W_p / p}$
- $W_s + W_p$ 可相应地表示为 $[f + (1 - f)]W$

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

示例

- **90%的部分并行化.**
- **程序可并行化部分的加速比为p**
- **$T_{\text{串行}} = 20 \text{ seconds}$**
- **程序可并行化部分的运行时间:**
 $0.9 \times T_{\text{串行}} / p = 18 / p$
- **不可并行化部分的运行时间:**
 $0.1 \times T_{\text{串行}} = 2$

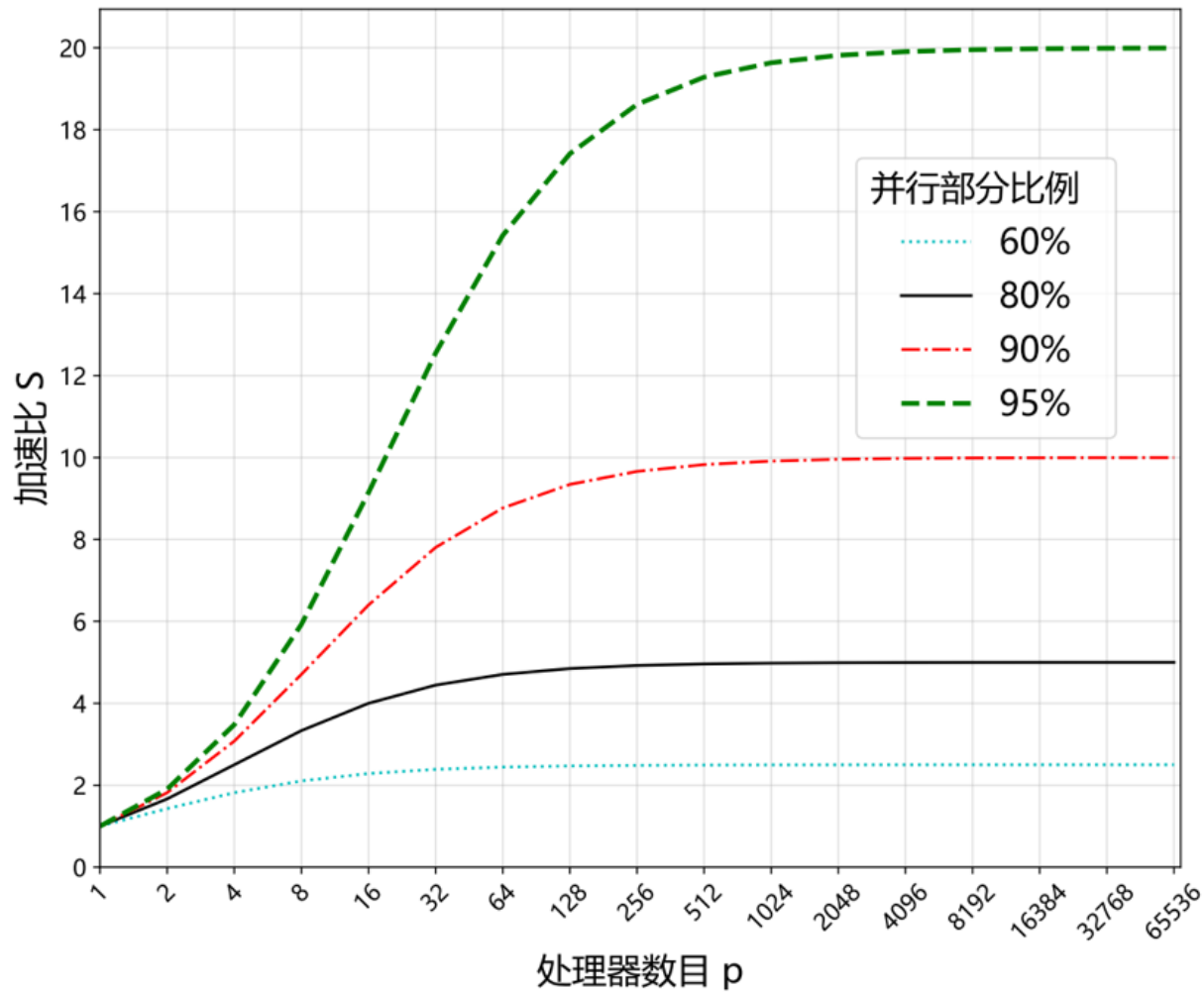
- 程序并行版本的全部运行时间为：

$$T_{\text{并行}} = 0.9 \times T_{\text{串行}} / p + 0.1 \times T_{\text{串行}} = 18 / p + 2$$

■ 加速比

$$S = \frac{T_{\text{串行}}}{0.9 \times T_{\text{串行}} / p + 0.1 \times T_{\text{串行}}} = \frac{20}{18 / p + 2}$$

Amdahl定律



Amdahl定律

- 固定负载的加速公式:

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

- $p \rightarrow \infty$ 时, 上式极限为: $S = 1 / f$

加速比是有上限的

Amdahl定律

- 除非一段串行程序的**所有部分**都可以**并行化**，否则可以达到的加速比是**非常有限的**
 - 不论使用多少核
- 并行程序的加速比与串行执行所需要的时间无关，只与**可并行部分的长度 W_p** 和**并行系数 P** 有关

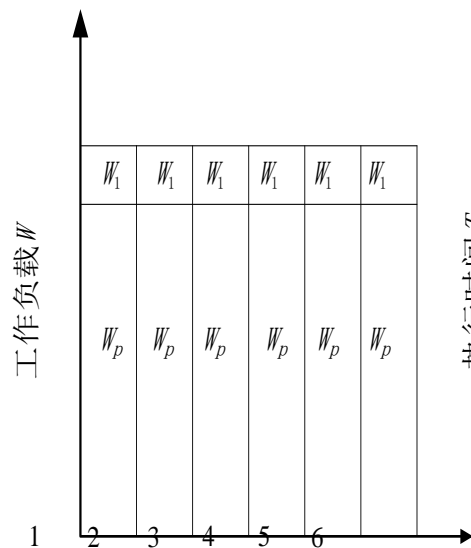
Amdahl定律

- 实际并行加速不仅受限于程序的串行分量，也受并行程序运行时额外开销的影响，令 W_o 为额外开销

$$S = \frac{W_s + W_p}{W_s + \frac{W_p}{p} + W_o} = \frac{W}{fW + \frac{W(1-f)}{p} + W_o} = \frac{p}{1 + f(p-1) + W_o p / W}$$

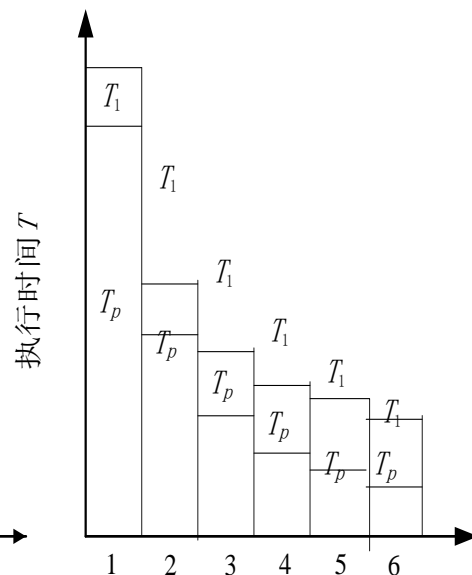
- $p \rightarrow \infty$?

Amdahl定律



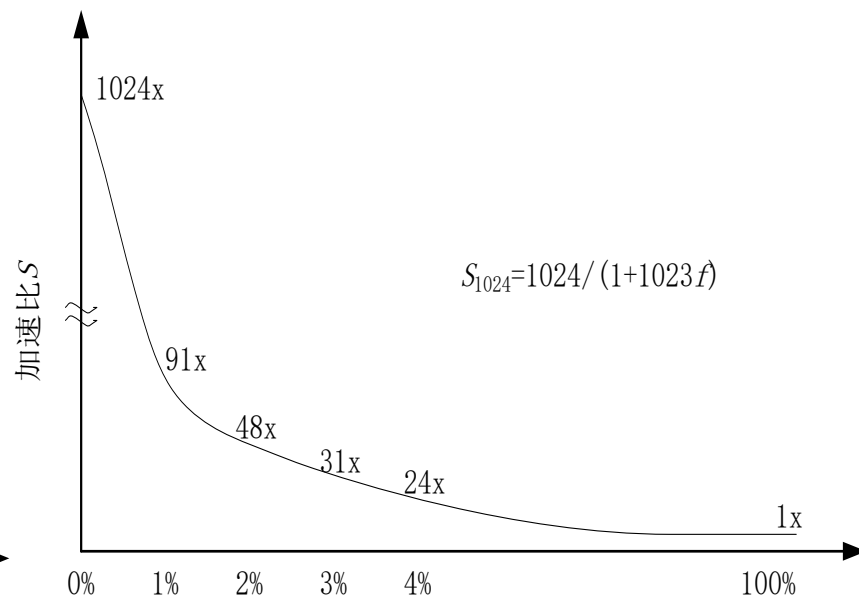
处理器数 P

(a)



处理器数 P

(b)



程序中顺序部分的百分比 f

(c)

Amdahl定律

- Amdahl 定律是适用于**固定负载**情况下的加速比模型，它的基本出发点是：
 - (1) 在科学计算领域，有很多问题由于规模较大导致无法满足实时性要求，即问题求解的时间开销是个关键因素，而问题的规模也就是负载是固定不变的
 - (2) 由于计算负载是固定不变的，因此增加参与计算的处理器数量能够提高程序的执行速度，达到加速的目的

效率

- **效率 (Efficiency) :**
 - 加速比 S 与线性加速比 p 的比值

$$E = \frac{S}{p} = \frac{\frac{T_s}{T_p}}{p} = \frac{T_s}{p \cdot T_p}$$

Ex. 一个并行程序的加速比和效率

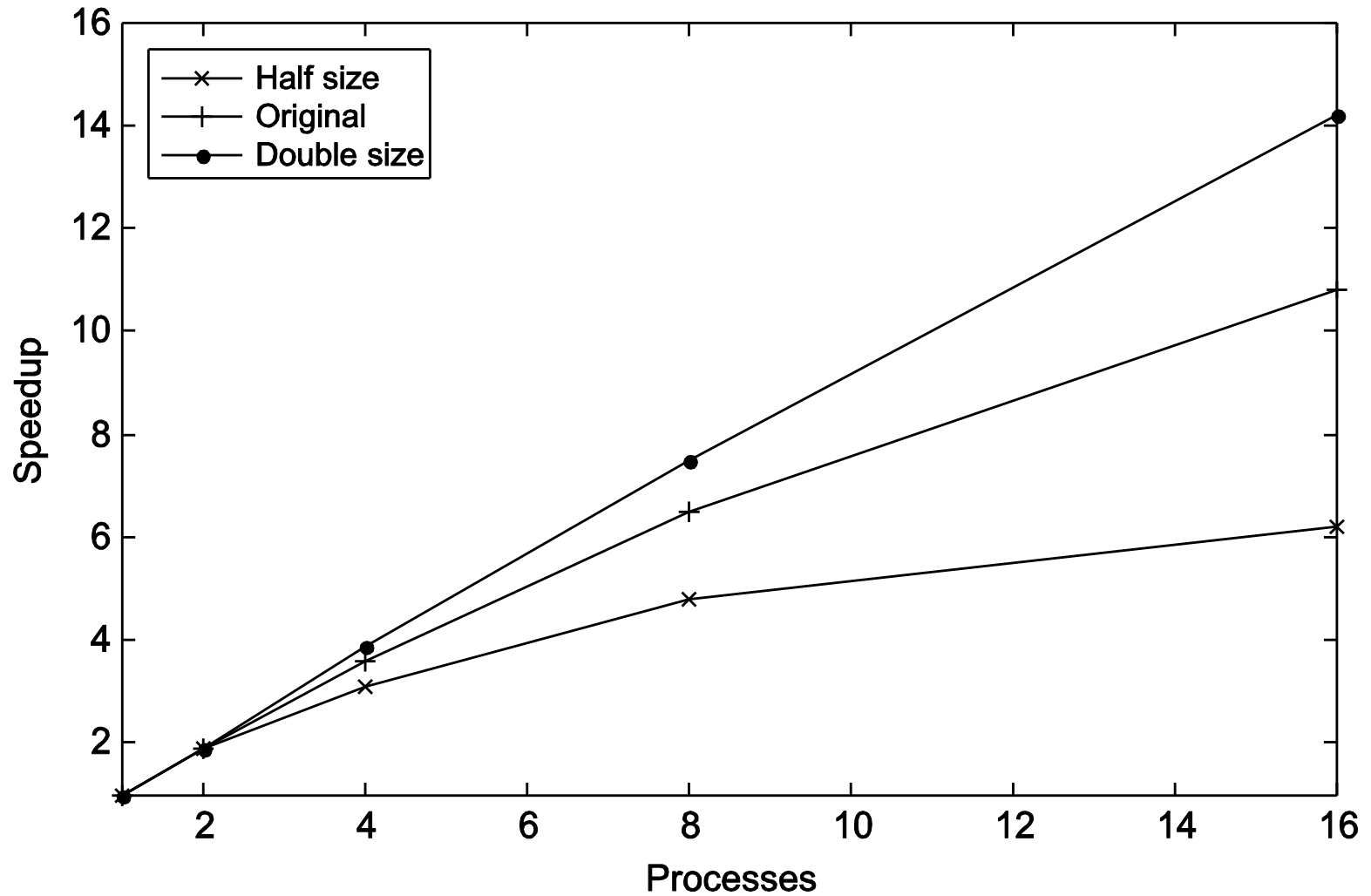
p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

E随着**p**的增大越来越小

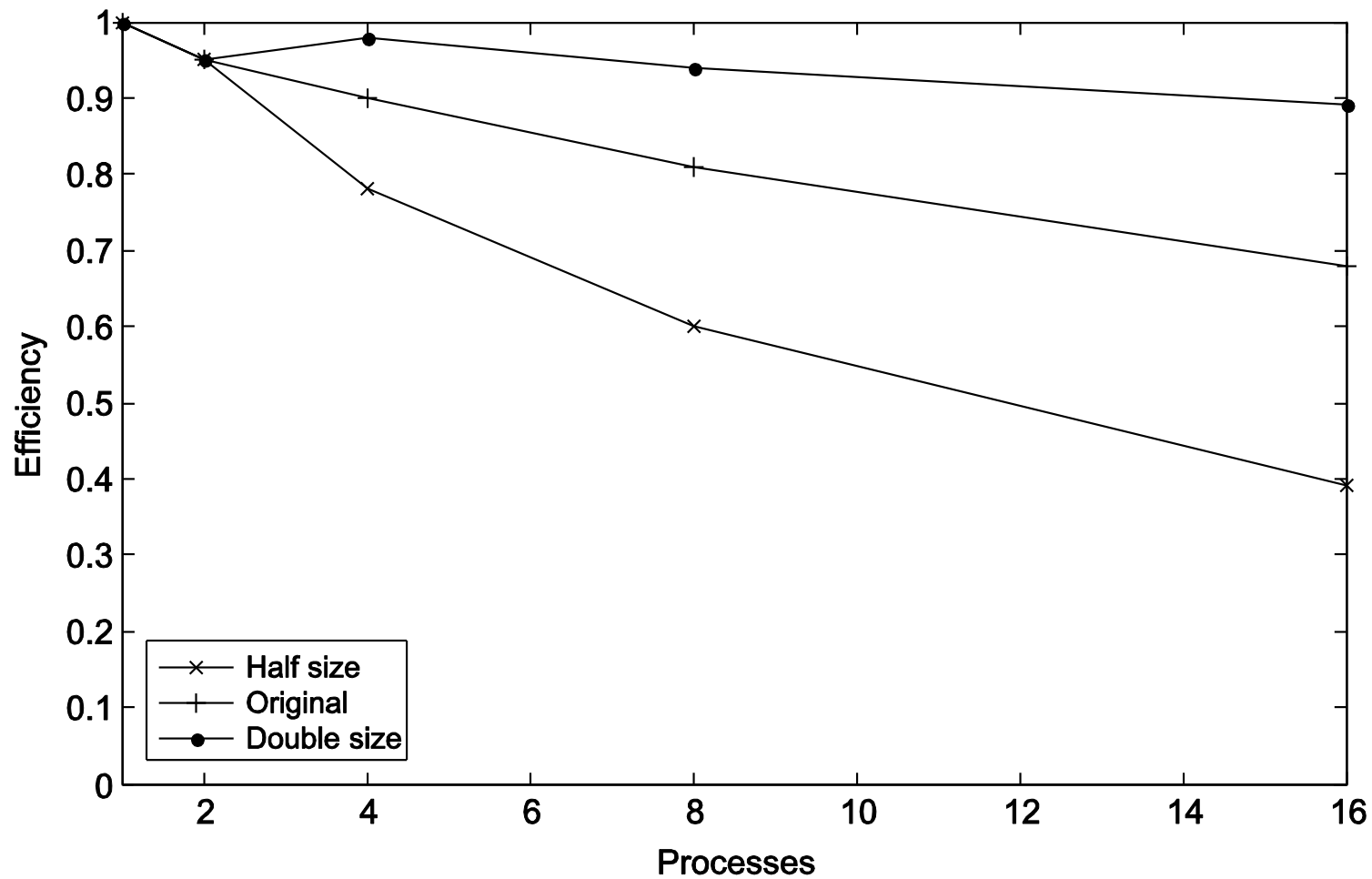
不同问题规模的并行程序的加速比核效率

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

加速比Speedup



效率Efficiency



Gustafson 定律

- Amdahl 定律推出：当处理器核数增加到无穷大时，加速比会趋近于一个上限
- 但 Gustafson（古斯塔夫森）在 1988 年进行的实际实验，在一个拥有 1024 个处理器的计算机，分别获得了 1021x/1020x/1016x 的加速比，得到了与 Amdahl 定律不一样的结论



Gustafson 定律

- 分析:

- 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

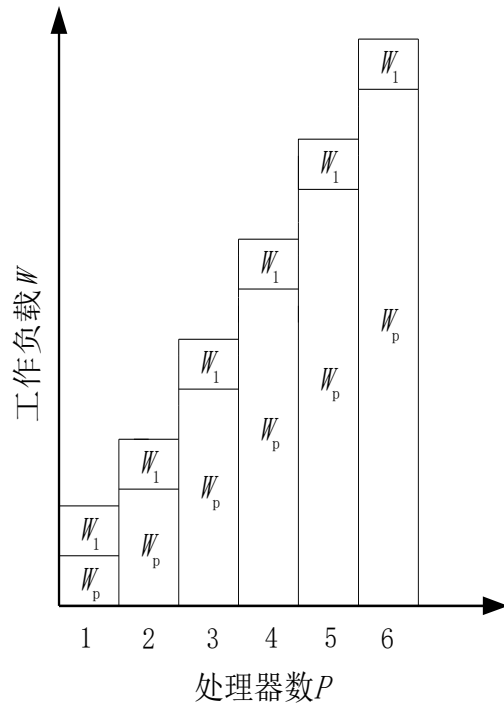
- Gustafson 加速定律: $S' = \frac{W_s + pW_p}{W_s + p \cdot W_p / p} = \frac{W_s + pW_p}{W_s + W_p}$
(放大问题规模的加速公式)

$$S' = f + p(1-f) = p + f(1-p) = p - f(p-1)$$

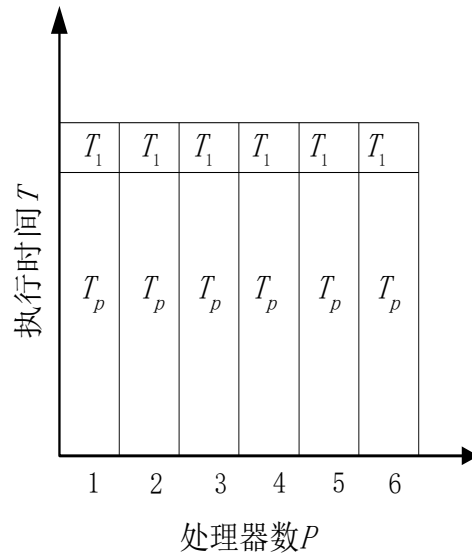
- 并行开销 W_o :

$$S' = \frac{W_s + pW_p}{W_s + W_p + W_o} = \frac{f + p(1-f)}{1 + W_o / W}$$

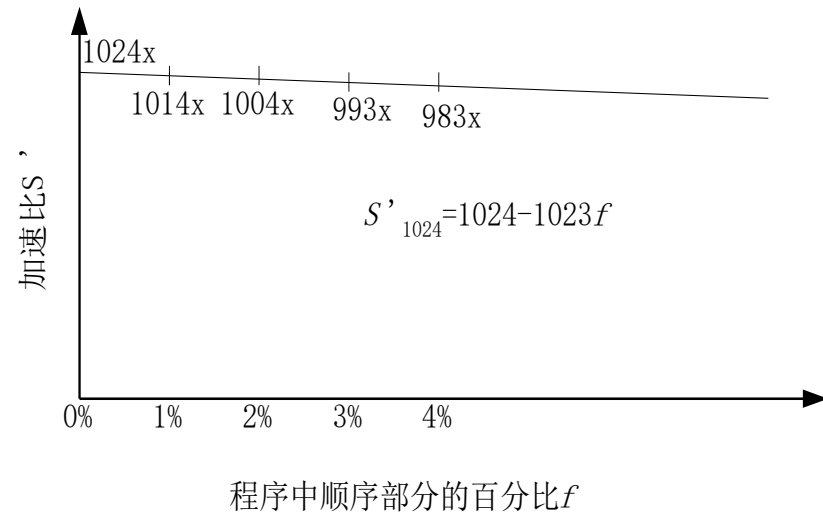
Gustafson 定律



(a)



(b)



(c)

Sun-Ni 定律

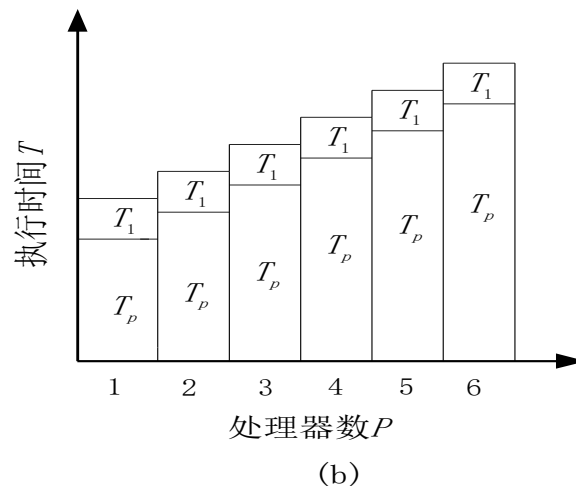
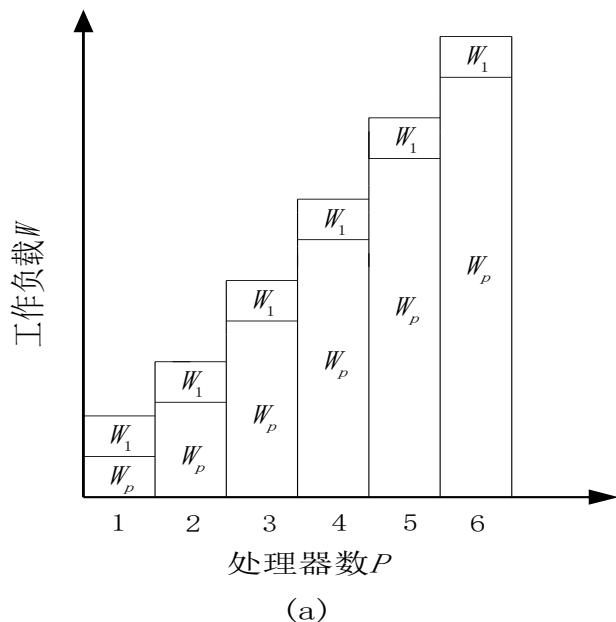
- 基本思想（**存储受限的加速定律**）：
 - 只要存储空间许可，应尽量增大问题规模以产生更好和更精确的解（此时可能使执行时间略有增加）
 - 假定在单节点上使用了全部存储容量 M 并在相应于 W 的时间内求解之，此时工作负载 $W = fW + (1-f)W$
 - 在 p 个节点的并行系统上，能够求解较大规模的问题是因为存储容量可增加到 pM 。令因子 $G(p)$ 反映存储容量增加到 p 倍时并行工作负载的增加量，所以扩大后的工作负载 $W = fW + (1-f)G(p)W$
- 存储受限的加速公式：

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

- 并行开销 W_o :

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

Sun-Ni 定律



- $G(p) = 1$ 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-Ni 加速均比 Amdahl 加速和 Gustafson 加速高

加速比讨论

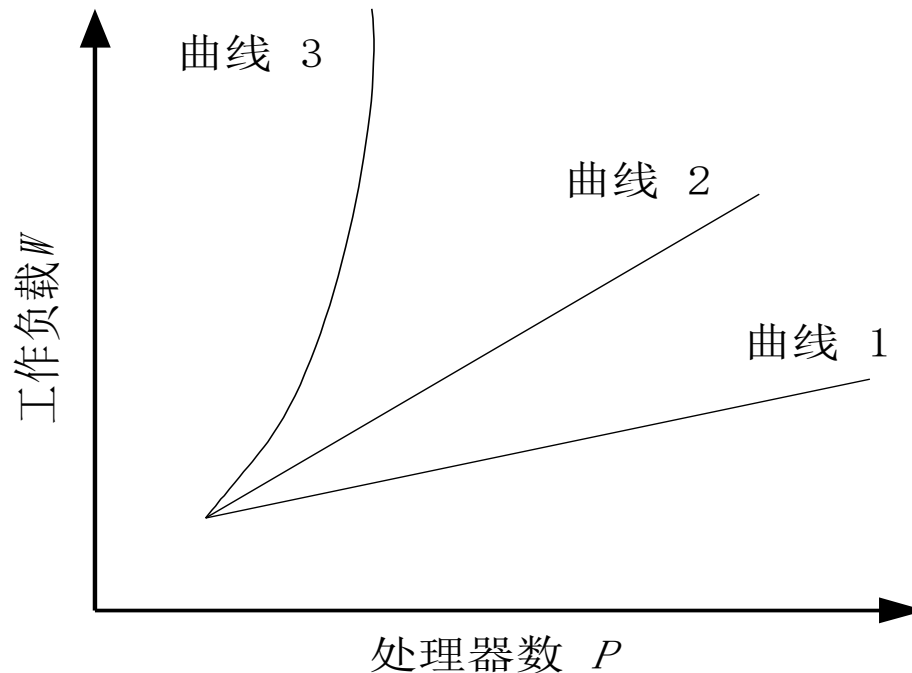
- 参考的加速经验公式： $P/\log P \leq S \leq P$
- 线性加速比：
 - 很少通信开销的矩阵相加、内积运算等
- $P/\log P$ 的加速比：
 - 分治类的应用问题
 - 类似二叉树，树的同级可并行执行，但向根逐级推进时，并行度将逐渐减少
- 通信密集类的应用问题：
 - $S = 1 / C(P)$ ，其中， $C(P)$ 是 P 个处理器的某一通信函数（线性/对数）
- 超线性加速
 - 并行搜索算法 (提前取消无谓的搜索分支)
- 绝对加速：最佳并行算法与串行算法相比
- 相对加速：同一算法在单机和并行机的运行时间之比

可扩展性 (Scalability)

- 如果一个技术可以处理规模不断增加的问题，那么它就是**可扩展的**
- 如果在增加进程/线程的个数时，可以维持固定的效率，却不增加问题的规模，那么程序称为**强可扩展的**
- 如果在增加进程/线程个数的同时，只有以相同倍率增加问题的规模才能使效率值保持不变，那么程序称为**弱可扩展的**

可扩展性

- 曲线 1 表示算法具有很好的扩展性；曲线 2 表示算法是可扩展的；曲线 3 表示算法是不可扩展的



可扩展性

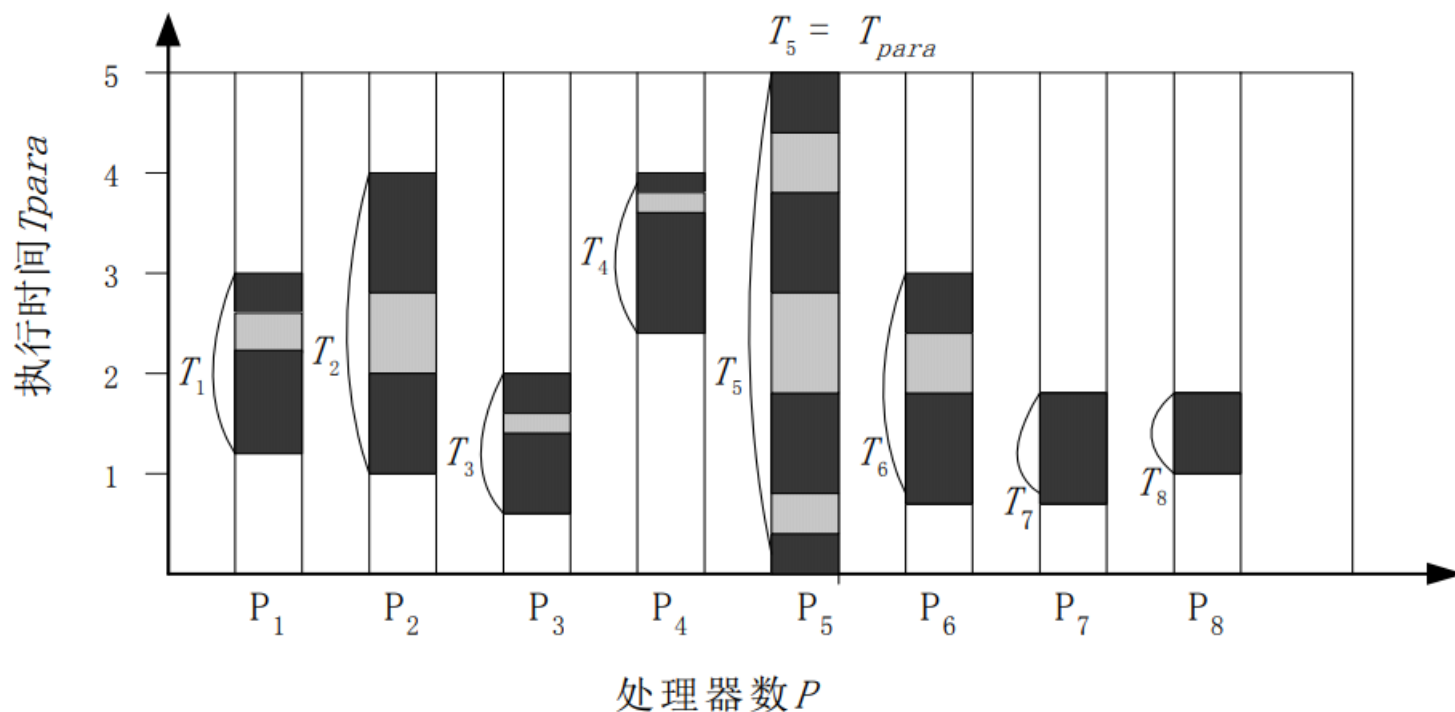
- 可扩展性研究的主要目的是：
 - (1) 探索算法和并行架构的组合
 - 确定解决某类问题使用何种并行算法和并行架构的组合，更有利于利用大量的硬件资源
 - (2) 预测算法性能
 - 根据某个算法在特定架构下的小规模处理器上的性能，预测该算法移植到较大规模的处理器上后算法的运行性能
 - (3) 计算最大加速比
 - 探索在固定的问题规模下，确定利用的处理器数量及其能获得的最大加速比
 - (4) 指导算法改进和并行架构设计
 - 根据拓展性指标，指导开发、研究人员改进并行算法或者系统架构，以提高处理器的利用效率

计时Taking Timings

- 什么样的时间？
 - 程序从开始到结束的时间
 - 感兴趣的一部分所花费的时间
 - **CPU**时间
 - ‘墙上时钟’ 时间—代码从开始执行到执行结束的总耗费时间

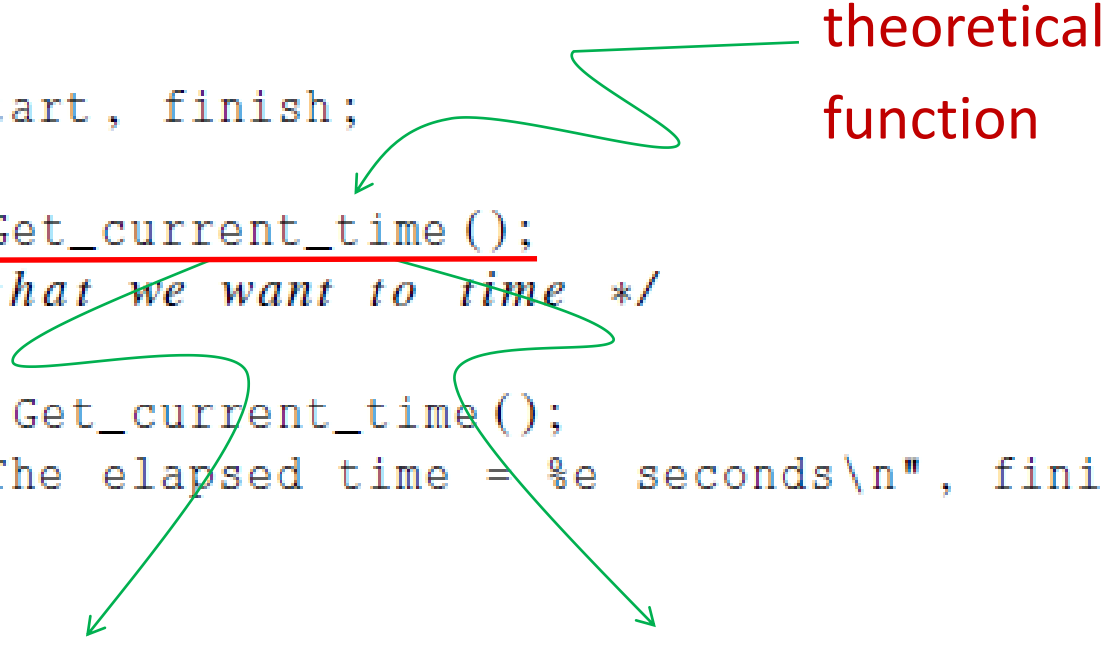
计时Taking Timings

系统平均延迟时间:
$$\bar{L}(W, p) = \sum_{i=1}^p (T_{para} - T_i + L_i) / p$$



计时Taking Timings

```
double start, finish;  
...  
start = Get_current_time();  
/* Code that we want to time */  
...  
finish = Get_current_time();  
printf("The elapsed time = %e seconds\n", finish-start);
```



theoretical
function

MPI_Wtime

omp_get_wtime

计时Taking Timings

```
private double start, finish;  
...  
start = Get_current_time();  
/* Code that we want to time */  
...  
finish = Get_current_time();  
printf("The elapsed time = %e seconds\n", finish-start);
```

计时Taking Timings

- 计时函数的分辨率 (resolution) 问题

计时Taking Timings

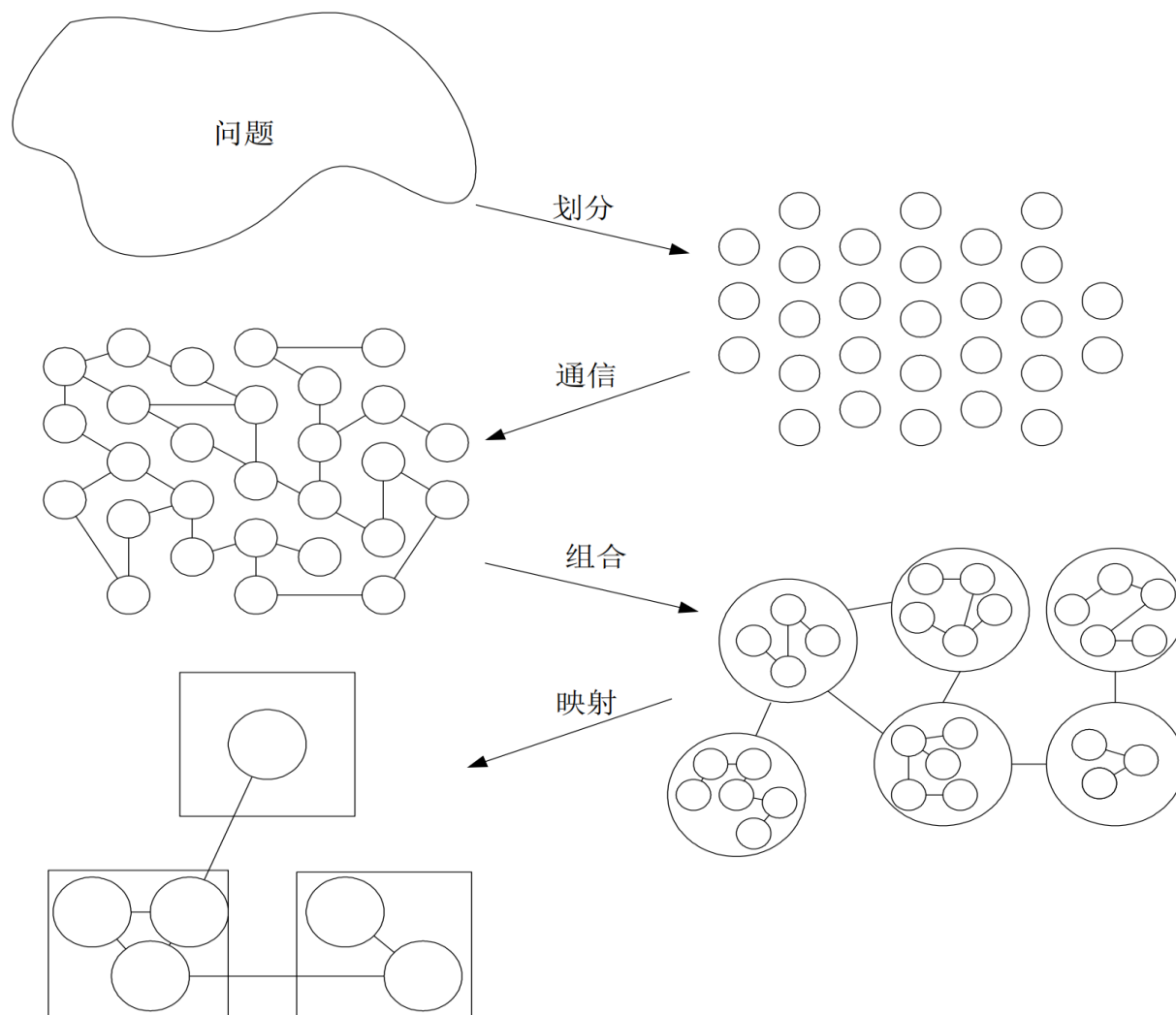
```
shared double global_elapsed;  
private double my_start, my_finish, my_elapsed;  
.  
.  
.  
/* Synchronize all processes/threads */  
Barrier();  
my_start = Get_current_time();  
  
/* Code that we want to time */  
.  
.  
.  
  
my_finish = Get_current_time();  
my_elapsed = my_finish - my_start;  
  
/* Find the max across all processes/threads */  
global_elapsed = Global_max(my_elapsed);  
if (my_rank == 0)  
    printf("The elapsed time = %e seconds\n", global_elapsed);
```

2.7 并行程序设计

Foster并行化方法

- 任务划分（**Partitioning**）
 - 将整个计算分解为一些小的任务，其目的是尽量开拓并行执行的机会
- 通信（**Communication**）分析
 - 分析确定诸任务执行中所需交换的数据和协调诸任务的执行，由此可检测 上述任务划分的合理性
- 任务组合/凝聚/聚合（**Agglomeration/Aggregation**）
 - 按性能要求和实现的代价来考察前两阶段的结果，必要时可将一些小的任务组合成更大的任务以提高性能或减少通信开销
- 映射/分配（**Mapping**）
 - 将每个任务分配到一个处理器(进程/线程)上，其目的是最小化全局执行时间和通信成本以及最大化处理器的利用率

并行算法（程序）的设计步骤



任务划分

- 域分解（Domain Decomposition）也叫数据划分
 - 划分对象是数据
 - 算法（或程序）的输入数据、计算的输出数据、或者算法所产生的中间结果
 - 步骤：
 - 首先，分解与问题相关的数据，使这些小的数据片尽可能大致相等
 - 其次，将每个计算关联到它所操作的数据上
 - 由此将产生一系列的任务，每个任务包括一些数据及其上的操作
 - 当一个操作可能需要别的任务中的数据时，就会产生通信要求

任务划分

- 功能分解（**Functional Decomposition**）也称计算划分
 - 关注于被执行的计算上，而不是计算所需的数据上
 - 如果所做的计算划分是成功的，再继续研究计算所需的数据，如果这些数据基本上不相交，就意味着划分的成功
 - 尽管大多数并行算法采用域分解，但功能分解有时能揭示问题的内在结构
 - 搜索树没有明显的可分解的数据结构，但易于进行细粒度的功能分解：开始时根生成一个任务，对其评价后，如果它不是一个解，就生成若干叶结点，这些叶结点可以分到各个处理器上并行地继续搜索

任务划分

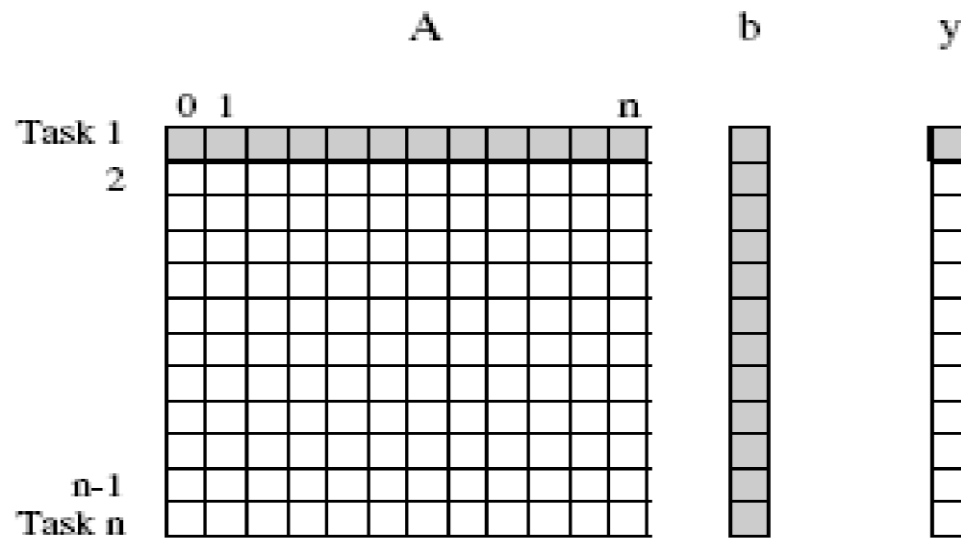
- 一个并行程序通常同时存在数据和功能并行的机会
 - 功能并行的并行度通常比较有限，并且不会随着问题规模的扩大而增加；不同的函数所涉及的数据集的大小可能差异很大，因此也难于实现负载均衡
 - 数据并行则一般具有较好的可扩展性，也易于实现负载均衡
 - 现有的绝大多数大规模的并行程序属于数据并行应用，但功能分解有时能提示问题的内在结构展示出优化的机遇

任务划分

- 合理性检查
 - (1) 所划分的任务数是否高于目标机上处理器数目1-2个量级?
 - 若不是，在后面的设计步骤中将缺少灵活性
 - (2) 划分是否避免了冗余的计算和存储要求?
 - 若不是，则产生的算法对大型问题可能不是可扩展的
 - (3) 各任务的尺寸是否大致相当?
 - 若不是，则分配处理器时很难做到负载均衡
 - (4) 划分的任务数是否与问题尺寸成比例?
 - 理想情况下，问题尺寸的增加应引起任务数的增加而不是任务尺寸的增加
 - 若不是这样，算法可能不能求解更大的问题，尽管有更多的处理器
 - (5) 是否采用了几种不同的划分法?
 - 多考虑几种选择可以提高灵活性。同时既要考虑域分解又要考虑功能分解

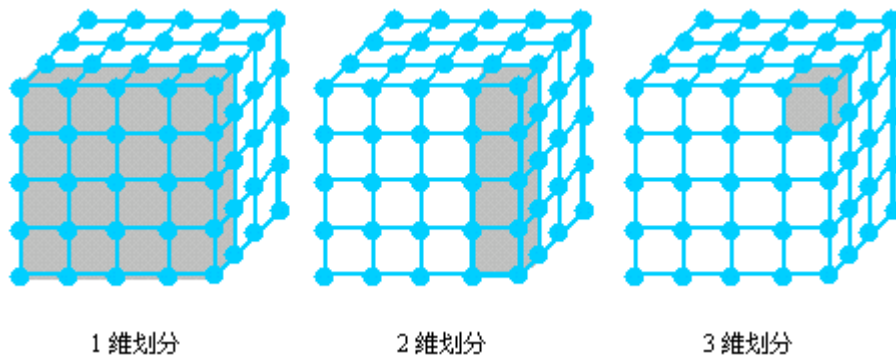
任务划分

- 示例1



任务划分

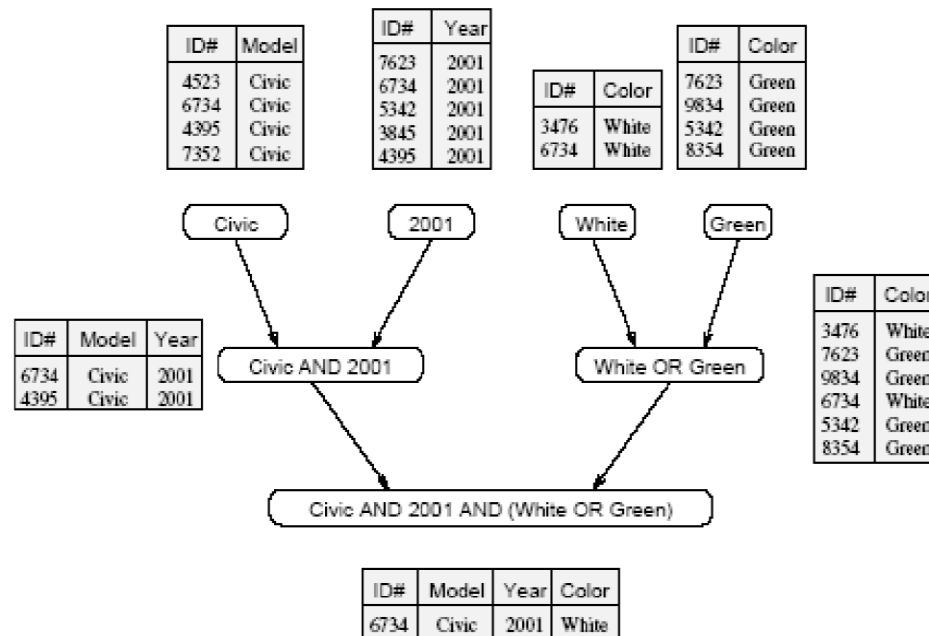
- 示例2



三维网格的域分解方法

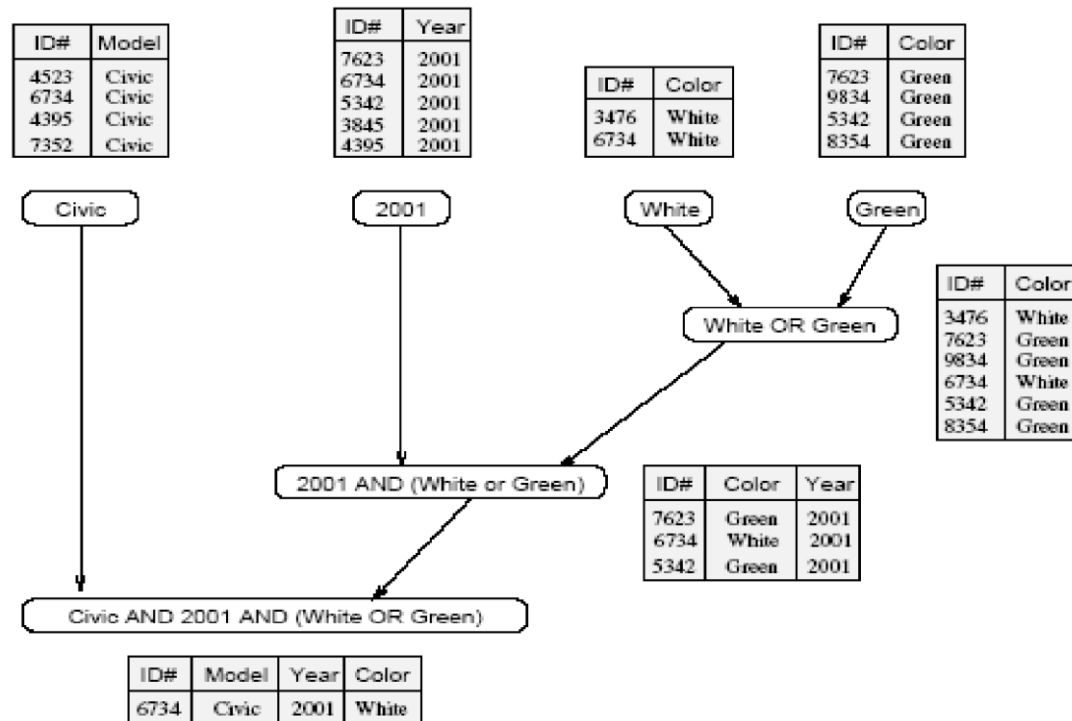
任务划分

- 示例3
 - 数据库并行搜索
 - Model="civic" AND Year="2001" AND (Color="Green" OR Color="White")
 - 任务依赖图(Task dependency graph)



任务划分

- 示例3
 - 数据库并行搜索
 - Model="civic" AND Year="2001" AND (Color="Green" OR Color="White")
 - 任务依赖图(Task dependency graph)



通信分析

- 局部/全局通信：较少的几个近邻或与很多别的任务通信
 - 局部通信：每个任务只和较少的几个近邻通信
 - 全局通信：每个任务与很多其他任务通信
- 结构化/非结构化通信：通信图 规整结构（如树、网格等）或任意图
 - 结构化：规整结构，如树、网格等
 - 非结构化：通信网络可能是任意图，如稀疏矩阵-向量乘（消息传递编程困难）
- 静态/动态通信：不随时间改变或可变的且由运行时所计算的数据决定
 - 静态：通信伙伴身份不随时间改变，如矩阵相乘
 - 动态：通信伙伴的身份可能由运行时数据决定且是可变的，如15-puzzle问题，消息传递编程较困难
- 同步/异步通信：接收方和发送方协同操作或无需协同
- One-way/two-way:
 - Two-way: 通信存在生产者-消费者关系，如读写
 - One-way: 通信只需要一方发起并完成，如只读

任务组合/凝聚/聚合

- 增加粒度
 - 大量细粒度任务有可能增加通信代价和任务创建代价
 - 表-容效应 (Surface-Volume Effect)
 - 一个任务通信需求比例于它所操作的子域的**表面积**，而计算需求却比例于子域的**容积**
- 保持灵活性
 - 可移植性
 - 可扩展性



映射/分配

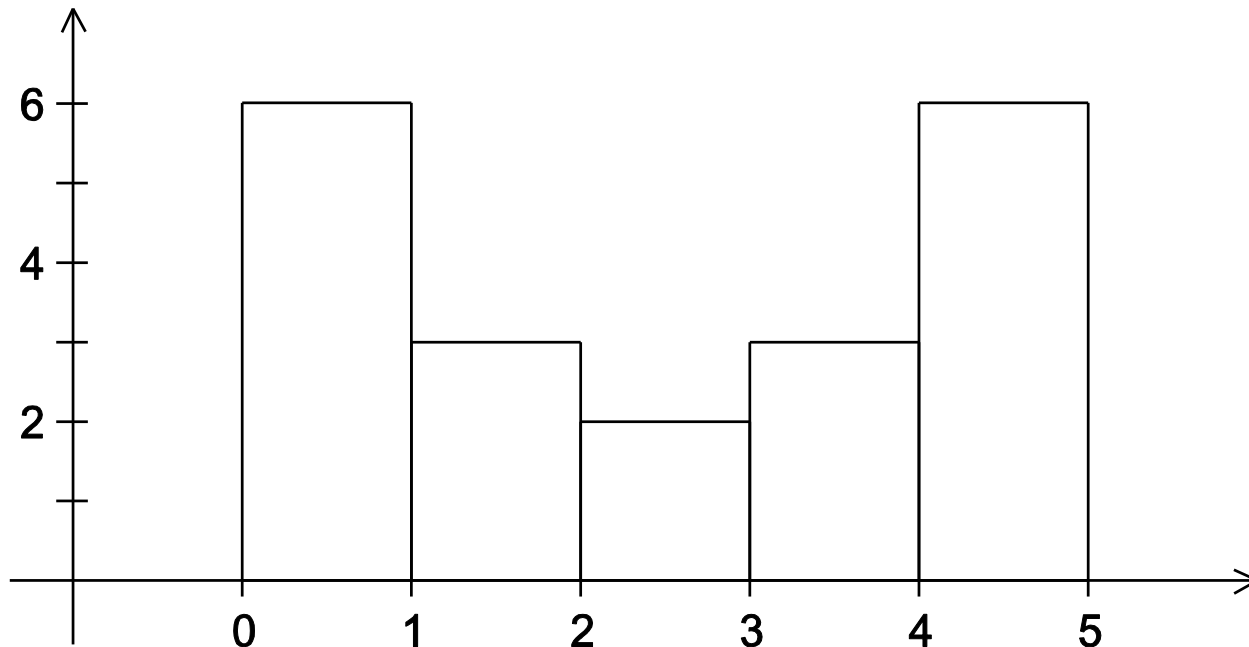
- 映射策略：指定任务到哪个处理器上去执行，其主要目标是减少算法的总执行时间，策略有二：
 - 把那些可并发执行的任务放在不同的处理器上以**增强并行度**
 - 把那些需频繁通信的任务置于同一个处理器上以**提高局部性**
- 负载均衡：使得所有处理器完成等量的任务
 - 减少同步等待的时间，这包括等待其它进程结束运行的时间和串行执行的代码部分（包括临界区代码和因数据相关造成的串行执行）
 - 通常采用的一种策略是在任务分配中，先集中目标使负载尽量均衡，然后再对任务分配进行调整，使得交互尽量少

任务分配与调度

- **静态调度：**任务到进程的算术映射
 - 静态地为每个处理器分配连续的循环迭代（要求处理器计算能力同构）
 - 轮转（将第 i 个循环迭代分配给第 $i \bmod P$ 个处理器）
- **动态调度：**动态调度技术可以取得较好的负载均衡效果（但开销较大）
 - 基本自调度SS (Self Scheduling)：每个处理器空闲时从全局队列取一个任务
 - 块自调度BSS (Block Self Scheduling)：每次取 k 个任务（块）
 - 指导自调度GSS (Guided Self Scheduling)：每次取剩余任务的 $1/P$
 - 因子分解调度FS (Factoring Scheduling)： $C_i = R_i / 2P$ ，每阶段所有处理器任务大小相等
 - 梯形自调度TSS (Trapezoid Self Scheduling)：连续的块之间的差距固定不变
 - 安全自调度SSS (Safe Self Scheduling)：任务分配使得累计执行时间刚刚超过平均负载
 - 亲和性调度AS (Affinity Scheduling)：分布式任务队列（本地调度+远程调度）
 - 自适应耦合调度AAS (Adapt Affinity Scheduling)：初始分配不平衡、计算能力异构
 - 重载、轻载、常载

示例 – 直方图histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



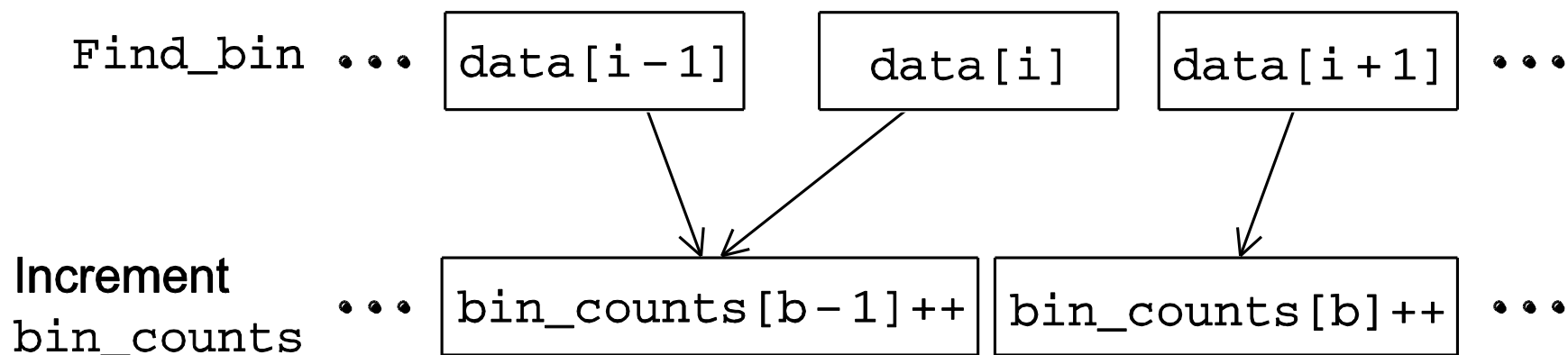
Serial program - input

1. 数据的个数: **data_count**
2. 一个大小为**data_count**的浮点数数组:
data
3. 包含最小值的桶中的最小值: **min_meas**
4. 包含最大值的桶中的最大值: **max_meas**
5. 桶的个数: **bin_count**

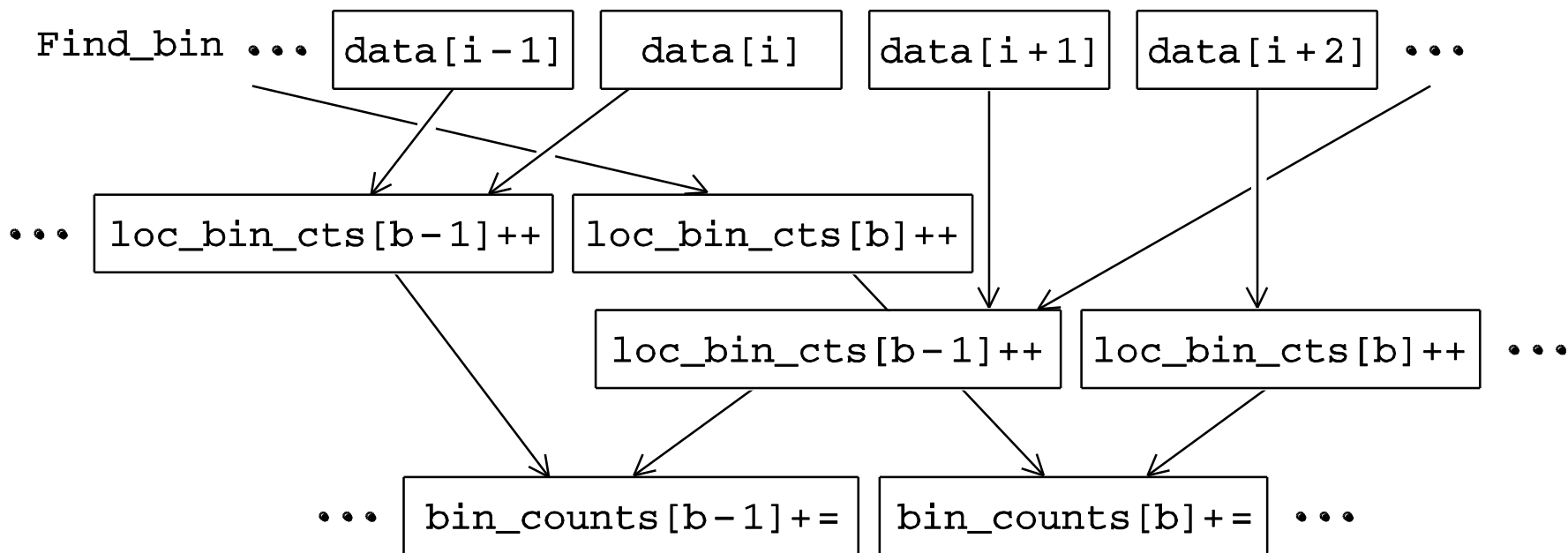
Serial program - output

1. **bin_maxes** : 一个大小为**bin_count**的浮点数数组，存储的是每个桶的上界
2. **bin_counts** : 一个大小为**bin_count**的整数数组，存储的是落在每个桶里的数据的个数

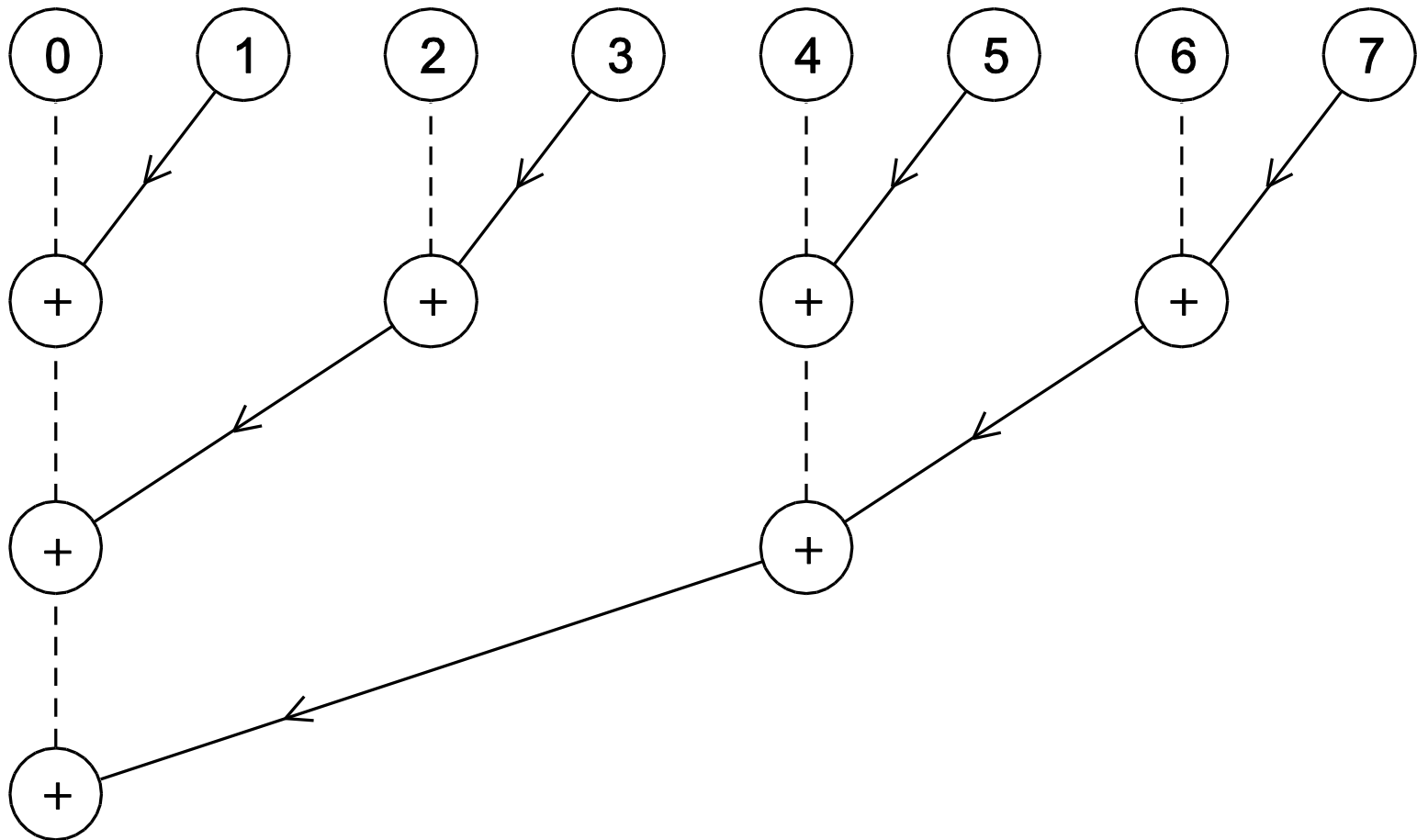
Foster方法最开始的两个阶段



任务与通信的另一种定义方式



本地数组相加



总结

- 串行系统 **Serial systems**
 - 计算机的标准硬件模型是冯·诺依曼结构
- 并行硬件 **Parallel hardware**
 - **Flynn**分类法
- 并行软件 **Parallel software**
 - **MIMD**系统的软件开发。此类系统中，大部分程序时单个程序，并通过分支语句实现并行
 - 单程序多数据流程序 **SPMD**

总结

- 输入和输出 **Input and Output**
 - 我们学习的编程，只关注其中可以有一个进程/线程访问标准输入 **stdin**，二所有进程可以访问标准输出 **stdout** 和标准错误输出 **stderr**
 - 但是在调试输出的时候，只让一个进程/线程访问标准输出 **stdout**

总结

- 性能评价
 - 加速比
 - 效率
 - Amdahl定律
 - 可扩展性
- 并行程序设计
 - Foster方法