

# 并行计算

## (Parallel Computing)

# 并行硬件和并行软件

## 学习内容：

- 背景知识
- 对冯诺依曼架构的改进
- 并行硬件
- 并行软件
- 输入和输出

# 并行硬件和并行软件

## 学习内容：

- 性能
- 并行软件设计
- 编写和运行并行程序

*classic von Neumann*

*not covered*

<b>S I S D</b> Single Instruction stream Single Data stream	<b>S I M D</b> Single Instruction stream Multiple Data stream
<b>M I S D</b> Multiple Instruction stream Single Data stream	<b>M I M D</b> Multiple Instruction stream Multiple Data stream

## 4. 并行软件

- 除了操作系统、数据库系统和Web服务器之外，目前很少有商用软件能广泛使用并行硬件
- 软件如何使用并行硬件
  - 对于共享内存程序
    - 启动一个进程（process）并 fork 线程（threads）
    - 线程负责执行任务
  - 对于分布式内存程序
    - 启动多个进程
    - 进程负责执行任务

## 4. 并行软件

- SPMD – single program multiple data

- 由一个可执行程序组成，通过使用条件分支，表现为多个不同的程序

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



## 4. 并行软件

### ●SPMD – single program multiple data

- 由一个可执行程序组成，通过使用条件分支，表现为多个不同的程序

```
if (I'm thread/process 0)
    operate on the first half of the array;
else /* I'm thread/process 1 */
    operate on the second half of the array;
```

## 4. 并行软件

### ●编写并行程序

#### ➤在进程/线程之间分配工作

- 每个进程/线程获得的工作量大致相同
- 最小化进程/线程间的通信

#### ➤同步进程/线程

#### ➤进程/线程间的通信

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



## 4. 并行软件

### ●共享内存

- 线程之间的通信通常通过共享变量完成，因此通信是隐式的，而不是显式的
- 动态和静态线程
  - 在许多环境下，共享内存程序使用动态线程（dynamic threads）
  - master 线程等待请求，请求到达后 fork 工作线程
  - 有效利用系统资源：线程所需的资源仅在线程实际运行时使用

## 4. 并行软件

### ● 共享内存

#### ➤ 动态和静态线程

- 静态线程（ static threads ）：所有线程由主线程创建，并且线程一直运行到所有工作完成为止
- 在线程 join 主线程之后，主线程会进行一些清理（例如，释放内存）
- 就资源使用而言，效率较低：如果线程空闲，则无法释放其资源
- 但是，fork 和 join 操作相当耗时。因此，如果有资源可用，静态线程可以获得比动态线程更好的性能

## 4. 并行软件

### ●共享内存

#### ➤ 不确定性（Nondeterminism）

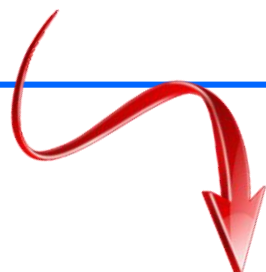
- 在处理器异步执行的 MIMD 系统中，存在不确定性
- 一个计算是不确定的：相同的输入，不同的输出

## 4. 并行软件

### ● 共享内存

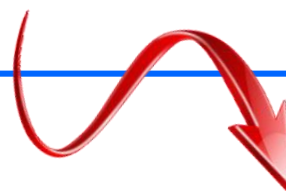
#### ➤ 不确定性 (Nondeterminism)

```
...  
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

# 4. 并行软件

●共享内存

➤不确定性（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

## 4. 并行软件

### ●共享内存

#### ➤不确定性（Nondeterminism）

- 线程独立执行并与操作系统交互，一个线程完成一个语句块的时间因执行而异，所以这些语句的完成顺序是不可以预测的
- 两个线程试图同时更新同一内存位置，产生竞争条件（race condition）
- 一次只能由一个线程执行的代码块称为临界区域（critical section）
- 应确保对 critical section 的互斥（mutually exclusive）访问

## 4. 并行软件

### ●共享内存

#### ➤ 不确定性 (Nondeterminism)

- 确保互斥访问的常用机制是 mutex 或 lock

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

## 4. 并行软件

### ●共享内存

#### ➤ 不确定性 (Nondeterminism)

- busy-waiting
- Semaphores

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



## 4. 并行软件

### ●共享内存

#### ➤线程安全（Thread safety）

- C 程序中的静态局部变量（static local variables）
- 静态变量在调用函数的线程间共享，会产生不想要的结果
- strtok: static char \*
- 线程0第一次调用strtok，线程1在线程0完成字符串拆分之前调用strtok，线程0的字符串将丢失或被覆盖
- 当代码块不是线程安全时，通常是因为不同的线程访问共享的数据
- reentrant code

## 4. 并行软件

### ● 分布式内存

➤ 消息传递 ( message-passing)

- SPMD
- 数组message属于进程私有变量
- 进程 0 对 stdout 操作

```
char message [ 100] ;  
...  
my_rank = Get_rank ( ) ;  
if ( my_rank == 1) {  
    sprintf ( message , "Greetings from process 1" ) ;  
    Send ( message , MSG_CHAR , 100 , 0 ) ;  
} else if ( my_rank == 0) {  
    Receive ( message , MSG_CHAR , 100 , 1 ) ;  
    printf ( "Process 0 > Received: %s\n" , message ) ;  
}
```

## 4. 并行软件

### ● 分布式内存

#### ➤ 消息传递（message-passing）

- Send
  - Receive 调用开始接收数据前，Send 阻塞（block）
  - Send 函数将 message 中的内容拷贝到自己的存储区后返回
- Receive：最常见的是直到接收 message 结束前，Receive 函数阻塞

## 4. 并行软件

### ● 分布式内存

#### ➤ 消息传递（message-passing）

- 广播（broadcast）：一个进程向所有进程发送同一数据
- 约减（reduction）：各个进程的结果组合成一个结果，如：将每个进程的结果相加
- 被广泛使用的消息传递 API：MPI（Message-Passing Interface）
- 消息传递有时被称为“并行编程中的汇编语言”

## 4. 并行软件

### ● 分布式内存

#### ➤ 单向通信（One-sided communication）

- 在消息传递中 Send 和 Receive 是成对出现的，通信需要两个进程参与
- 单向通信或远程内存访问（remote memory access）：单个进程调用函数，用另一个进程的值更新本地内存，或用调用进程的值更新远程内存
- *何时可以安全的更新对方进程的数据？*
- *对方如何能够知道数据被更新？*
- 远程内存操作带来的错误很难被跟踪

## 4. 并行软件

### ● 分布式内存

- 全局地址划分语言 (Partitioned global address space languages)
  - 程序员认为共享内存编程比消息传递和单向通信更吸引人
  - 开发编程语言允许用户使用共享内存技术来为分布式内存硬件编程
  - 不可预测的性能: *访问的是本地内存还是远程内存?*

## 4. 并行软件

### ● 分布式内存

#### ➤ 全局地址划分语言

- x 和 y 属于同一 core 的进程的内存
- x 和 y 属于不同 core 的进程的内存

```
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 */  
...  
for( i = my_first_element ; i <= my_last_element ; i++)  
    x [ i ] += y [ i ] ;
```

## 5.输入和输出

- 在分布式内存程序中，只有进程0可以访问stdin；在共享内存程序中，只有主线程或线程0可以访问stdin
- 在分布式内存和共享内存程序中，所有进程/线程都可以访问stdout 和stderr
- 但是，由于输出到stdout的顺序不确定，在大多数情况下，除了调试输出之外，所有输出到stdout的操作都只使用一个进程/线程



## 5.输入和输出

- 调试输出应始终包含生成输出的进程/线程的ID
- 应只有一个进程/线程尝试访问stdin、 stdout或stderr以外的任何单个文件。例如，每个进程/线程都可以打开自己的私有文件进行读写，但是不应有两个进程/线程打开同一个文件

## 6.性能

### ●加速和效率（Speedup and efficiency）

#### ➤加速

- Number of cores =  $p$
- Serial run-time =  $T_{\text{serial}}$
- Parallel run-time =  $T_{\text{parallel}}$

linear speedup

$$T_{\text{parallel}} = T_{\text{serial}} / p$$



## 6.性能

### ●加速和效率 (Speedup and efficiency)

#### ➤加速

linear speedup

$$T_{\text{parallel}} = T_{\text{serial}} / p$$

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$



## 6.性能

### ●加速和效率（Speedup and efficiency）

#### ➤效率

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

## 6.性能

### ●加速和效率（Speedup and efficiency）

$p$	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

# 6.性能

●加速和效率（Speedup and efficiency）

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

## 6.性能

- 加速和效率（Speedup and efficiency）

$$T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}}$$

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

## 6.性能

### ●阿姆达尔定律（Amdahl's Law）

- 除非串行程序所有部分都可以被并行化，否则可能的加速将是非常有限的，不管可用的内核数是多少





## 6.性能

### ●阿姆达尔定律 (Amdahl's Law)

- 例如：我们可以并行化一个串程序的 90%
- $T_{\text{serial}} = 20 \text{ seconds}$
- 并行部分的运行时间：  $0.9 \times T_{\text{serial}} / p = 18 / p$
- 不能被并行化部分的运行时间：  $0.1 \times T_{\text{serial}} = 2$
- 总的并行运行时间：  $T_{\text{parallel}} = 0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}} = 18 / p + 2$

## 6.性能

### ●阿姆达尔定律（Amdahl's Law）

➤ 加速：

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

## 6.性能

- 古斯塔夫森定律 (Gustafson's law)

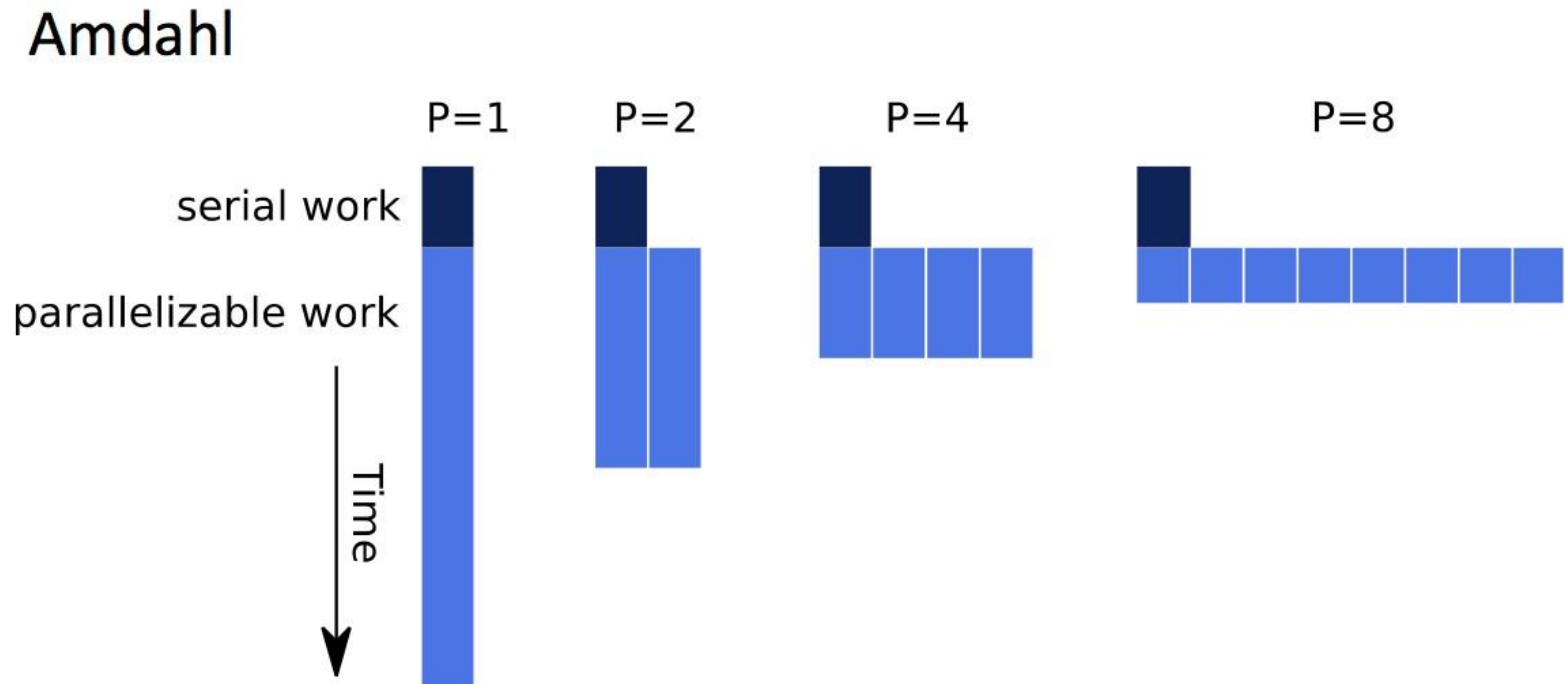
- 顺序计算的比例通常随着问题规模的增加而减少：

$$S = p - \alpha (p - 1)$$

- $p$  为处理器的数量， $\alpha$  为串行部分的比例

## 6.性能

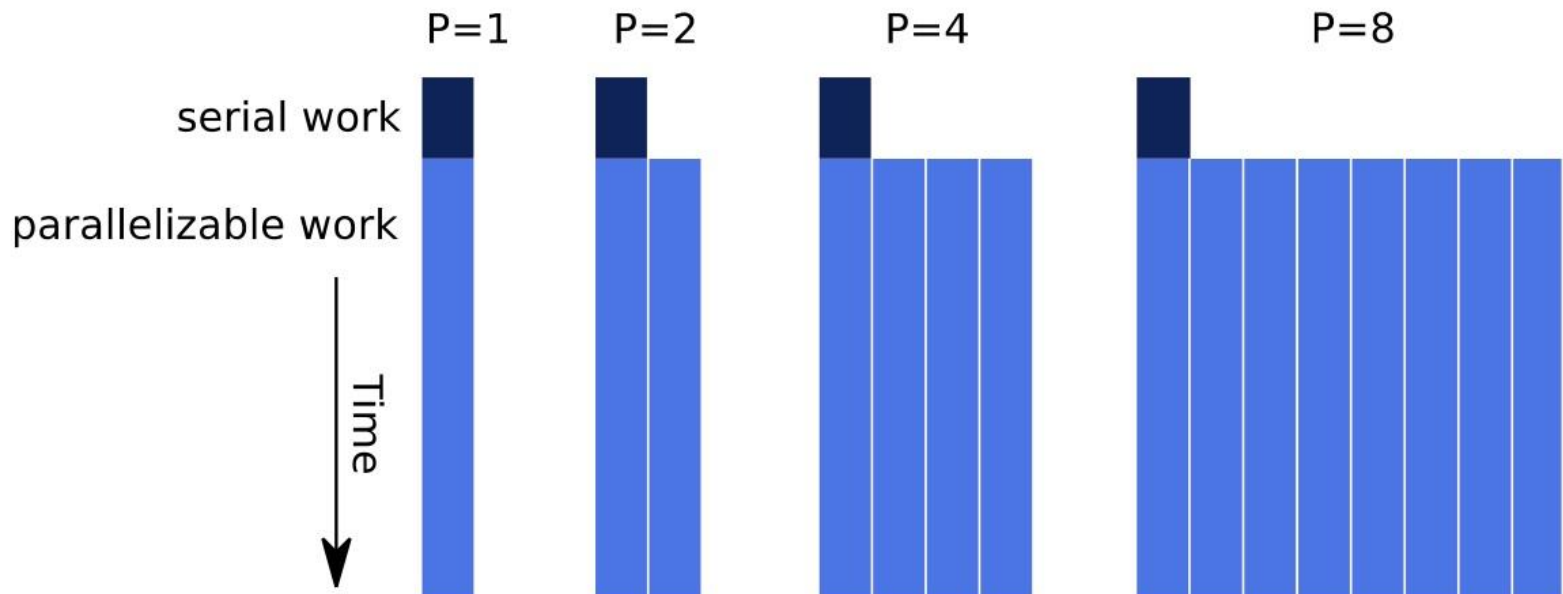
- 阿姆达尔定律 (Amdahl's Law) vs. 古斯塔夫森定律 (Gustafson's law)



## 6.性能

- 阿姆达尔定律 (Amdahl's Law) vs. 古斯塔夫森定律 (Gustafson's law)

### Gustafson-Baris



## 6.性能

### ●可扩展性（Scalability）

- 一般来说，如果能够处理日益增加的问题大小，则问题是可伸缩的
- 如果我们增加进程/线程的数量并在不增加问题大小的情况下保持效率不变，则问题具有强扩展性（ *strongly scalable* ）
- 如果我们通过以增加进程/线程数量相同的速率增加问题大小来保持效率不变，则问题具有弱扩展性（ *weakly scalable* ）

## 6.性能

### ●计时（Taking Timings）

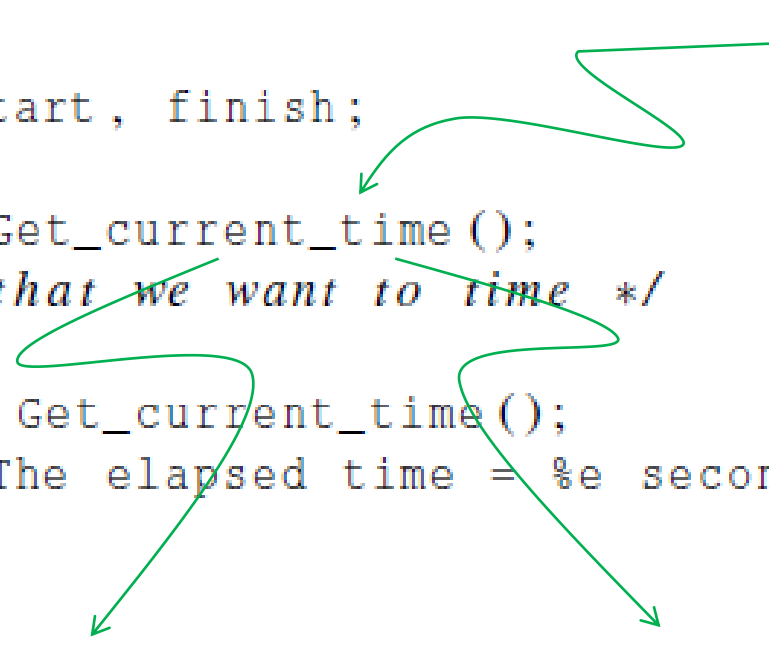
- 记录开始到结束的时间？
- 记录感兴趣的程序片段时间？
- 记录 CPU 的时间？
- 记录挂钟（Wall clock）的时间？



## 6.性能

### ●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



## 6.性能

### ●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);
```

## 6.性能

### ●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);
```

## 7.并行程序设计

### ●Ian Foster 的方法论 《 Designing and Building Parallel Programs 》

#### 1. 划分（Partitioning）

将要执行的计算和由计算所操作的数据划分为小任务

这里的重点是确认可以并行执行的任务

#### 2. 通信（Communication）

确定在上一步所划分的任务之间需要进行通信

## 7.并行程序设计

### ●Ian Foster 的方法论 《 Designing and Building Parallel Programs 》

#### 3. 聚集（Agglomeration or aggregation）

将第一步中确定的任务和通信组合成更大的任务。

例如：

如果任务 A 必须在任务 B 之前执行，则将它们组合成单个任务可能更合理

## 7.并行程序设计

### ●Ian Foster 的方法论 《 Designing and Building Parallel Programs 》

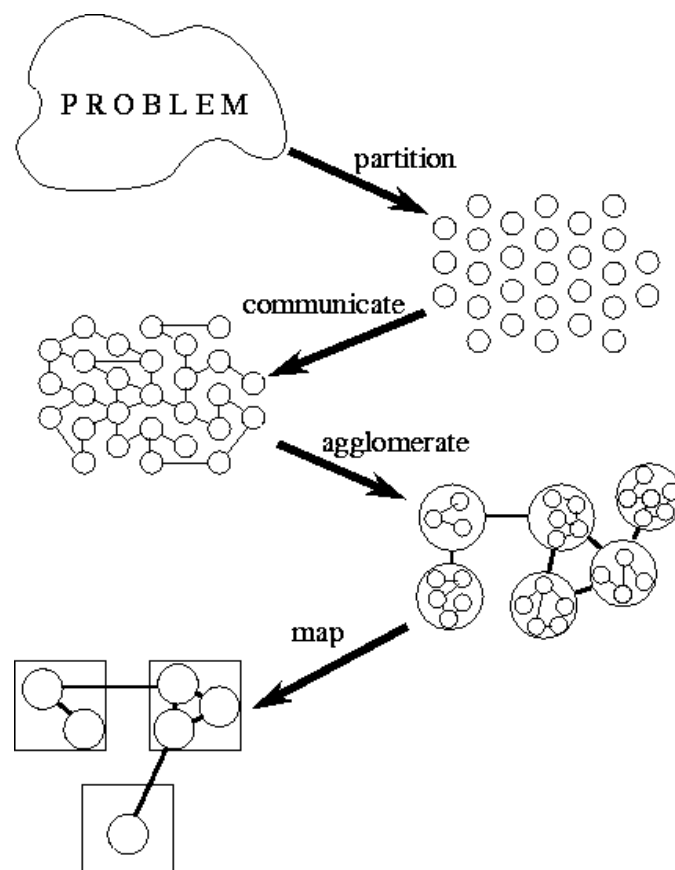
#### 4. 映射（Mapping）

将上一步中标识的复合任务分配给进程/线程

应该使通信最小化，并且每个进程/线程获得大致相同的工作量

## 7.并行程序设计

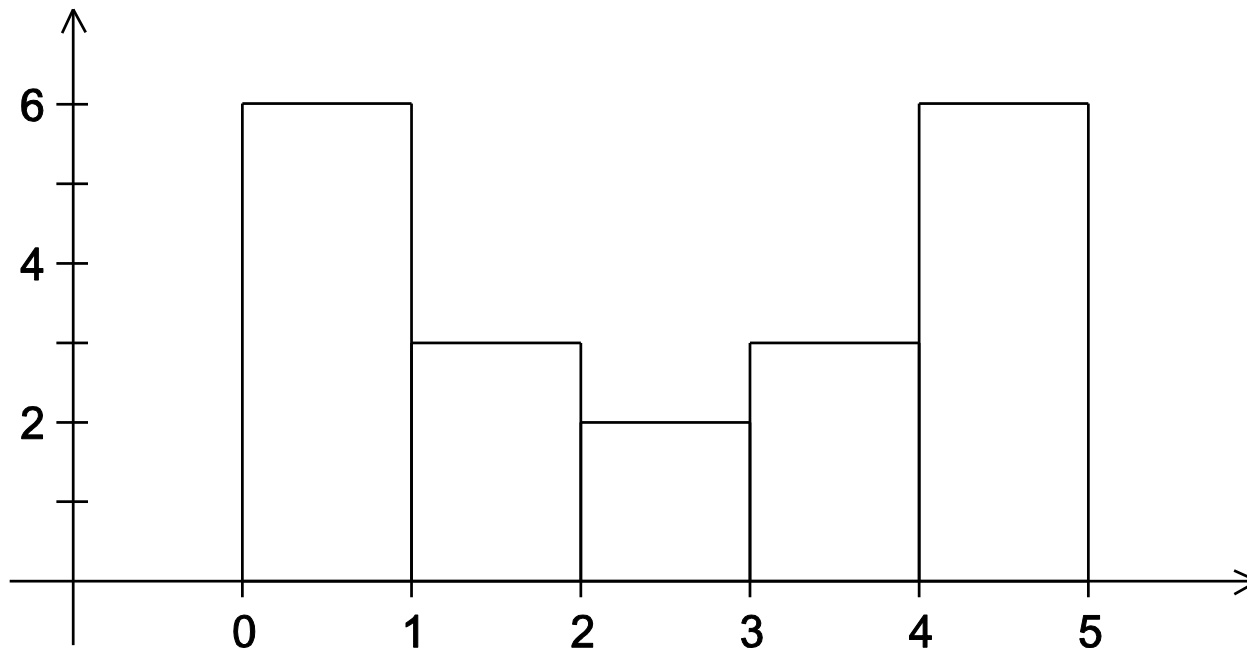
### ●Ian Foster 的方法论 《Designing and Building Parallel Programs》



## 7.并行程序设计

●An example: 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



## 7.并行程序设计

### ●An example: histogram – 串行程序（输入）

- 数据量: data\_count
- 存放数据的 float 类型数组: data
- 包含数据最小值的 bin 的最小值: min\_meas
- 包含数据最大值的 bin 的最大值: max\_meas
- bin 的数量: bin\_count



## 7.并行程序设计

### ●An example: histogram – 串行程序（输出）

➤ 数组： 包含每个bin中数据元素的个数

- bin\_maxes: 存放 bin\_count 个 float 类型数的数组
- bin\_counts: 存放 bin\_count 个 int 类型数的数组

**$\text{bin\_width} = (\text{max\_meas} - \text{min\_meas}) / \text{bin\_count}$**

**for (b = 0; b < bin\_count; b++)**

**bin\_maxes[b] = min\_meas + bin\_width \* (b+1);**

## 7.并行程序设计

●An example: histogram – 串行程序

```
for (i = 0; i < data_count; i++) {  
    bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);  
    bin_counts[bin]++;  
}
```

Find\_bin返回的bin应满足:  $\text{bin\_maxes}[\text{bin} - 1] \leq \text{data}[i] < \text{bin\_maxes}[\text{bin}]$

## 7.并行程序设计

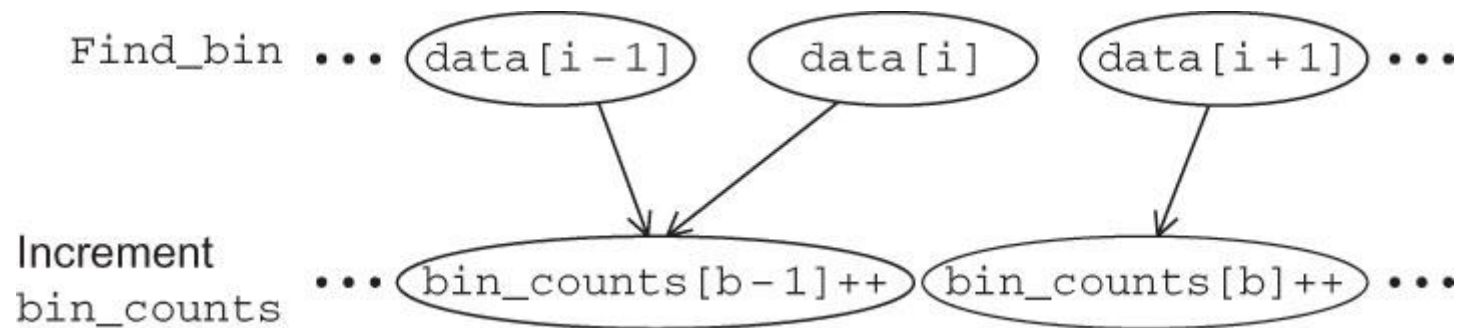
### ●An example: histogram – 并行化

- 假设 data\_count 远远大于 bin\_count
- 大部分工作集中在 Find\_bin 函数中的循环中
- 应用 Ian Foster 方法划分任务
  - 找到数据属于哪个 bin
  - 相应 bin 中的计数值累加

## 7.并行程序设计

### ●An example: histogram – 并行化

➤ 应用 Ian Foster 方法确定通信



## 7.并行程序设计

### ●An example: histogram – 并行化

#### ➤ 应用 Ian Foster 方法聚合

- 由于第二个任务只有在第一个任务结束后才能开始
- 两个任务可以合并

## 7.并行程序设计

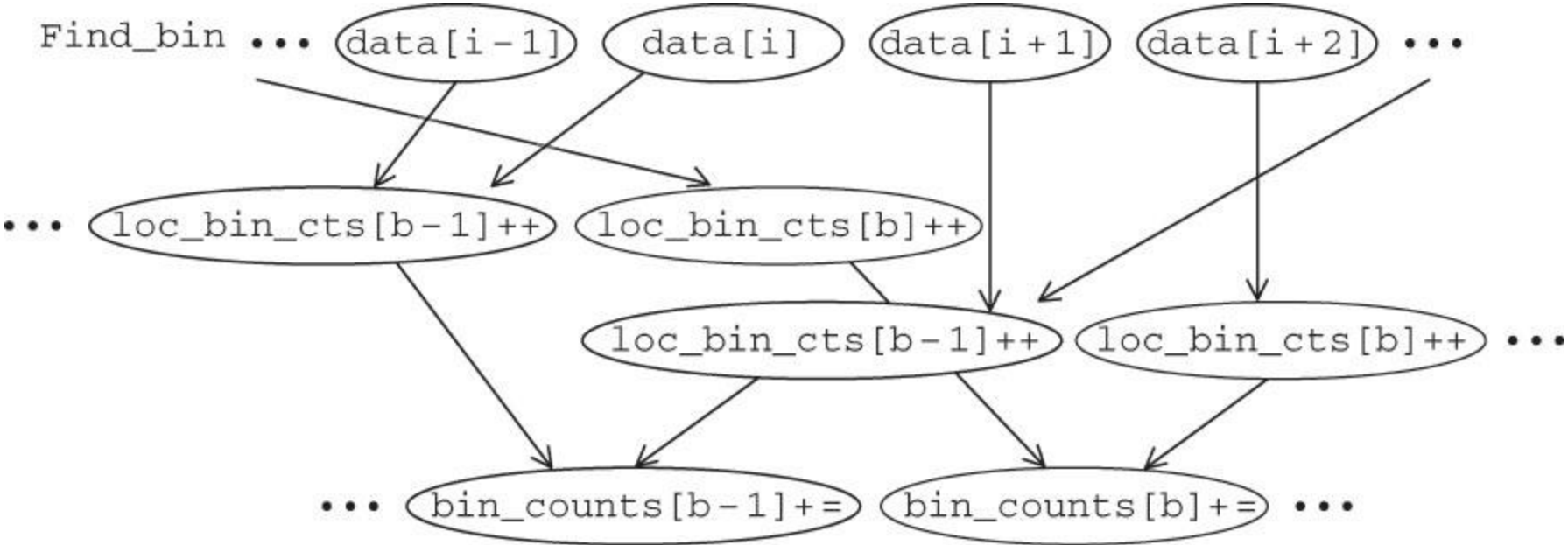
### ●An example: histogram – 并行化

#### ➤应用 Ian Foster 方法映射

- 如果 bin\_counts 为共享内存，会产生竞争
- 如果 bin\_counts 被划分到不同的进程/线程，会产生通信
- 在本地存放 bin\_counts 的拷贝，累加这些局部拷贝

# 7.并行程序设计

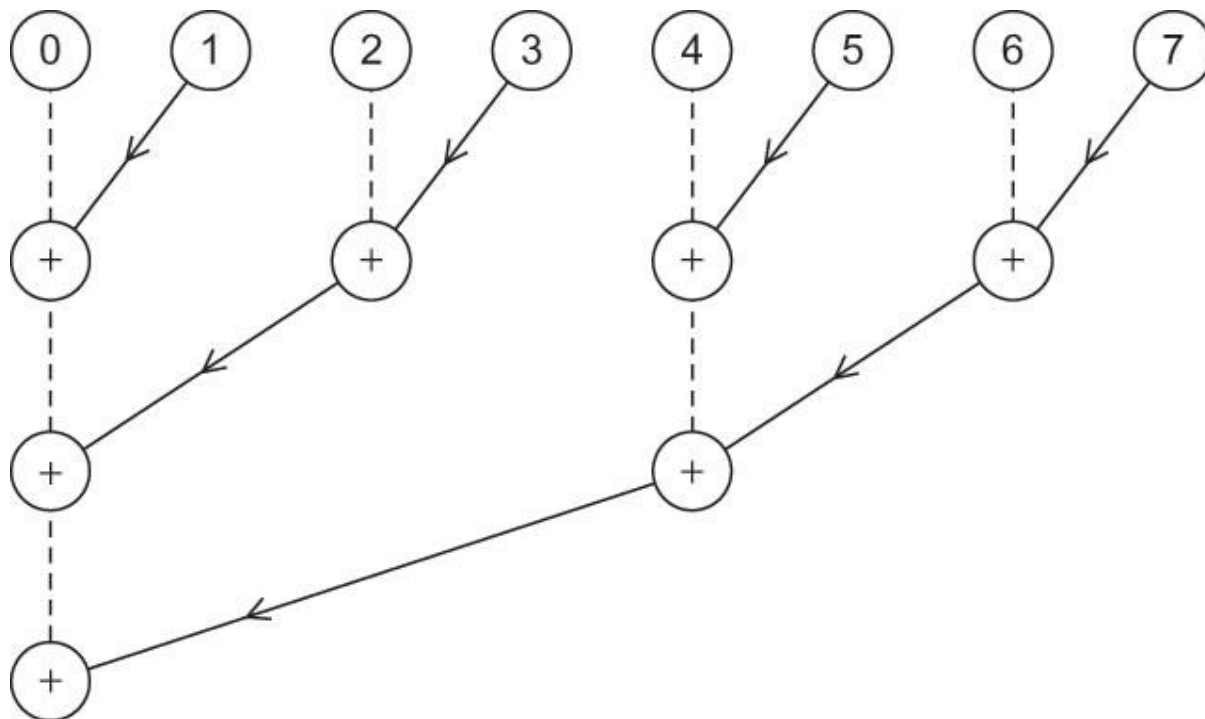
●An example: histogram – 并行化



## 7.并行程序设计

### ●An example: histogram – 并行化

#### ➤ 累加局部数组





## 8. 编写和运行并行程序

- 过去采用文本编辑器编写并行程序，在命令行编译、运行、调试程序
- 目前可以采用 IDE，如：Microsoft、Eclipse等
- 对于小型的共享内存系统，操作系统负责调度进程到可用的 core 上
- 对于大型系统，可以采用批处理调度程序，用户对 core 的数量提出需求，指定可执行程序的路径和输入输出的位置（通常为文件）

## 8. 编写和运行并行程序

- 对于典型的分布式系统和混合系统，主机负责分配节点给用户，批处理或者与用户交互
- 由于任务的启动通常包含与远程系统的通信，通常采用脚本启动，如：MPI 程序采用 mpirun 或 mpiexec 脚本启动

# 小结

## ● 串行系统

- 冯诺依曼架构：CPU进行计算，主存储存指令和数据
- 瓶颈：CPU 与 主存分离
- 一个运行的程序称为进程
- 线程由进程创建，启动和停止要快于进程
- cache位于CPU寄存器和主存之间，减少主存访问的延迟
- 储存在cache中的数据遵循“局部性”原则

# 小结

## ● 串行系统

- cache 的 hit 和 miss
- cache 直接由计算机硬件管理，程序员只能间接控制缓存
- 主存可以起到 secondary storage 的缓存作用
- 由硬件和操作系统通过 virtual memory 来管理
- swap space、virtual address、page table、TLB

# 小结

## ● 串行系统

- 指令级别的并行（ILP）允许处理器同时执行多条指令：pipelining和multiple issue
  - pipelining：处理器的功能单元按顺序排列
  - multiple issue：功能单元独立运行，处理器试图同时执行程序中的不同指令
- 线程级别的并行（TLP）

# 小结

## ●并行硬件

### ➤ 费林分类法 (Flynn's Taxonomy)

- 冯诺依曼架构：SISD
- SIMD：通常用于数据并行程序；向量处理器和GPU通常被分类为SIMD系统
- MIMD：通常由多个自主的处理器按照各自的步伐运行；共享内存、分布式内存。大型的MIMD系统通常为混合系统，由node构成；一些MIMD系统为异构系统，如：CPU + GPU

# 小结

## ●并行硬件

### ➤ 处理器之间的连接

- bus、crossbar、toroidal mesh
- 衡量同时通信的数量： bisection width or the bisection bandwidth
- 节点之间的通信： latency and bandwidth

### ➤ 共享内存的潜在问题： 缓存一致性（cache coherence）

- snooping and the use of directories
- 伪共享（false sharing）

# 小结

## ●并行软件

- 重点关注为同质（homogeneous）的MIMD系统开发软件
  - SPMD
  - threads for shared-memory; processes for distributed-memory
  - load balance, communication, and synchronization
  - 共享内存中的 race condition: critical section -> mutex
  - 共享内存中的 thread safety
  - 分布式内存系统的API: message passing



# 小结

## ●输入和输出

- 只有一个进程或线程可以访问 stdin
- 除了调试输出外，只有一个进程或线程可以访问 stdout

# 小结

## ●性能

### ➤ Speedup and Efficiency

- 固定问题大小，E通常随着p的增加而减小
- 固定进程或线程的数量，S和E通常随着问题大小的增加而增加

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

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

# 小结

## ●性能

### ➤ Speedup and Efficiency

- 阿姆达尔定律（Amdahl's Law）给出了 Speedup 的上限
- 但它没有考虑到在实际情况中，串行部分相对于并行部分的比例随着问题的增大而减小

- ### ➤ 为计算并行时间，通常在启动计时器之前同步进程/线程，并且在停止计时器之后，在所有进程/线程中找到最大运行时间

# 小结

## ● 并行程序设计

### ➤ Foster's methodology

- 划分（partitioning）问题，标识任务
- 确认任务之间的通信（communication）
- 聚集（agglomeration or aggregation）任务
- 将任务映射（mapping）到进程或线程

# 作业1

## ●问题1

- a. 假设一个共享内存系统使用窥探缓存一致性（snooping cache coherence）和写回（write-back）机制，core 0 的缓存中有变量  $x$ ，并执行赋值语句  $x=5$ ，假设 core 1 的缓存中没有  $x$ ，在core 0 更新  $x$  后，core 1 尝试执行  $y=x$ ， $y$  的值是什么？为什么？
- b. 假设上面的共享内存系统使用基于目录的缓存一致性（Directory-based cache coherence）， $y$  的值会是什么？为什么？
- c. 如何解决上面出现的问题？

# 作业1

## ●问题2

假设一个串程序的运行时间为： $T_{\text{serial}} = n^2$ （单位为毫秒），这个程序的并行版本运行时间为： $T_{\text{parallel}} = n^2/p + \log_2(p)$ ，编写一个程序获得在 $n$ 和 $p$ 为不同值时，并行版本的加速和效率（ $n = 10, 20, 40, \dots, 320$  和  $p = 1, 2, 4, \dots, 128$ ）。当固定 $n$ ， $p$ 增加时，加速和效率如何变化？当固定 $p$ ， $n$ 增加时，加速和效率如何变化？