第3讲并行软件及其程序设计

张永东

内容

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

并行软件

并行软件的主要任务

- 不再能通过硬件和编译器为应用提供性能上的稳定增长，要通过并行软件来发挥硬件性能
  - 硬件可以提供所需的速度
  - 编译器在发掘硬件能力已到极限
- 从今以后…
  - 在运行共享内存系统时:
    - 会启动一个单独的进程，然后派生（folk) 出多个线程。
    - 线程——任务的执行者。
  - 运行分布式内存程序时:
    - 会启动多个进程。
    - 进程——任务的执行者。

除非指明，执行任务者可以是进程，也可以是线程

讨论内容

- 只讨论MIMD软件
  - SPMD(Single Program, Multiple Data)
  - MPMD(Multiple Program, Multiple Data)
- 协调进程与线程
- 共享内存如何协调
- 分布内存如何协调
- 混合编程

SPMD

- SPMD单程序多数据流
  - 不是在每个核上运行不同的程序
  - SPMD程序仅包含一段可执行代码：
    - 通过使用条件转移语句，使这一段代码在执行时表现得像是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;
```

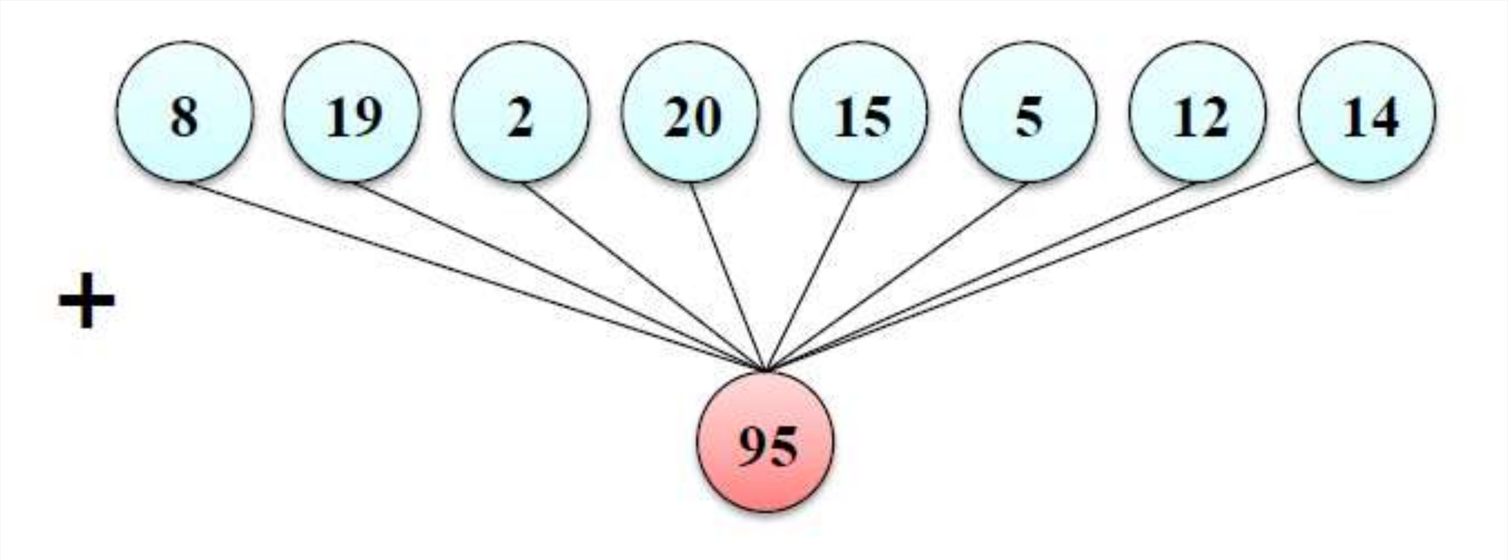
SPMD程序例

同一段代码如何像执行不同程序——父子进程

```
void main() {
    int child=fork(); //
   if (child == 0)
   {// 子进程
          printf("Now it is in child process.\n");
   }
    else
   {     // 父进程
         printf("Now it is in parent process.\n");
    }
   exit();
 }
```

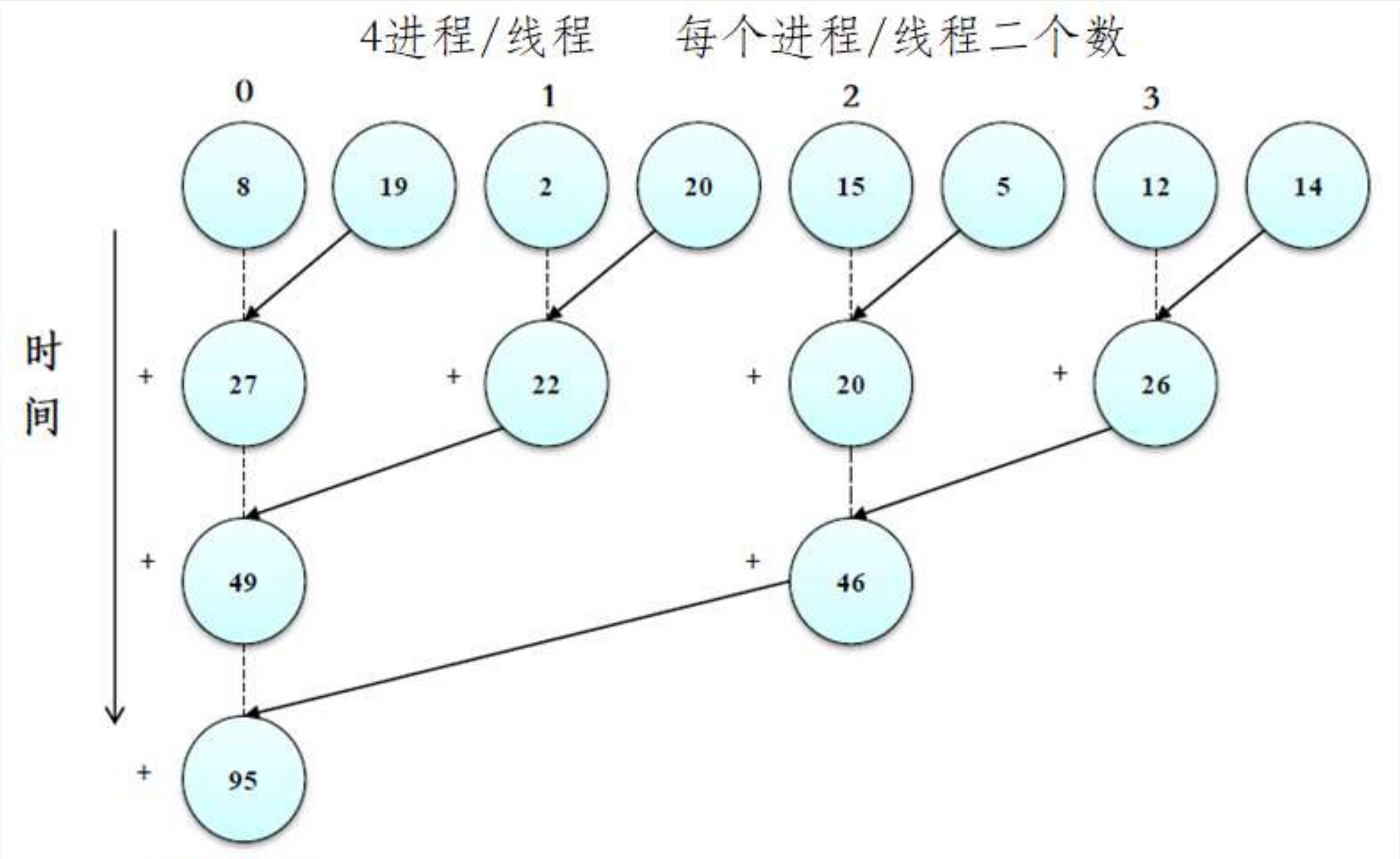
协调进程与线程

- 获得极高的开行性能是十分复杂的。
- 为了了解并行软件的主要任务，看简单的求和例子：



并行程序

- 一个并行求和算法



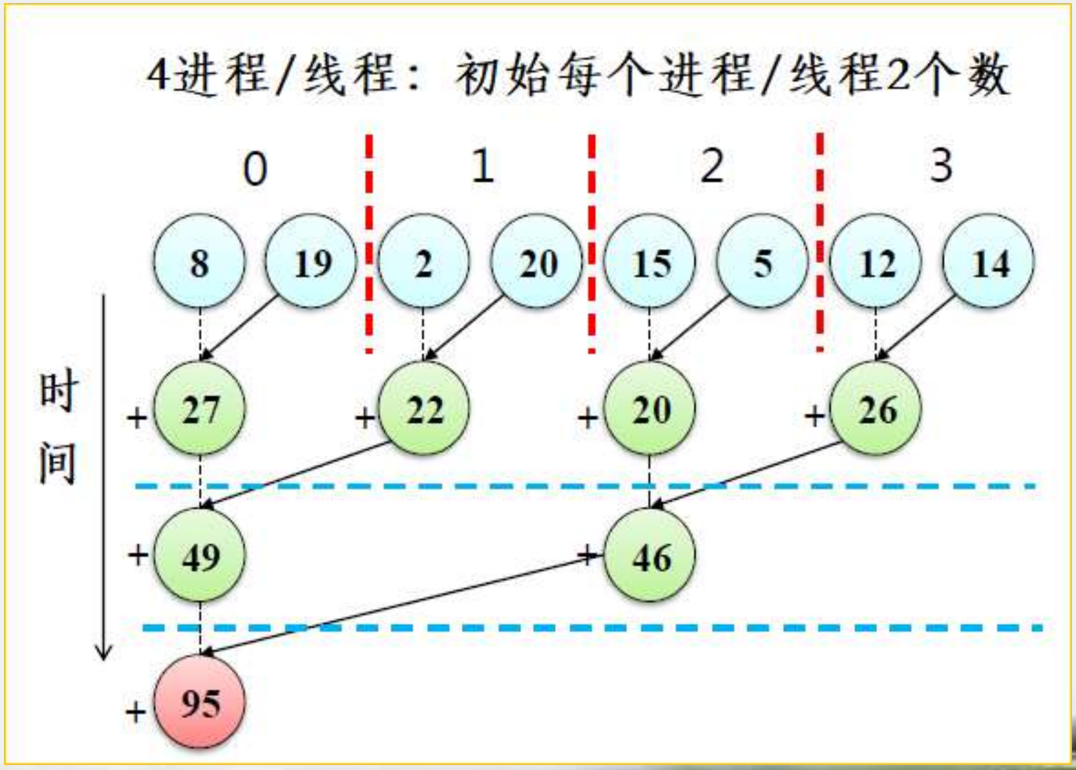
协调进程与线程

- 并行软件的主要任务：
  - 分别考虑两种系统——共享内存和分布内存

  1.任务划分：将任务在进程/线程之间分配

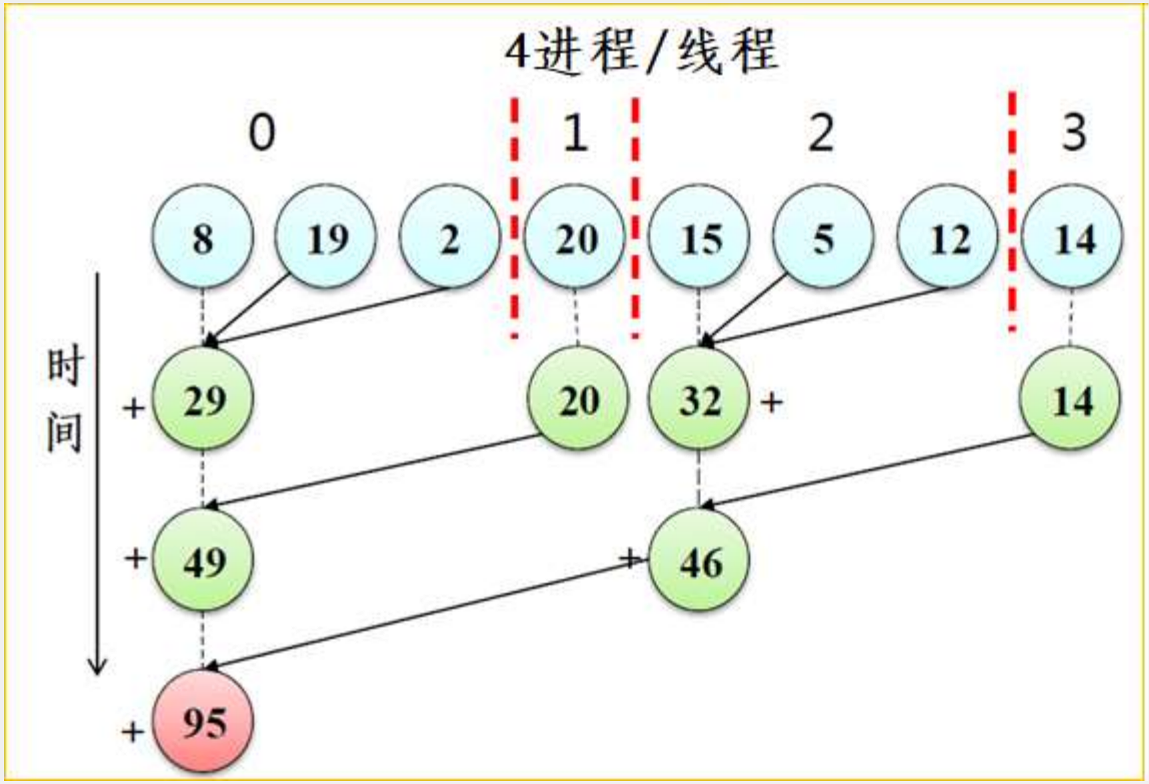
  2.同步

  3.通信



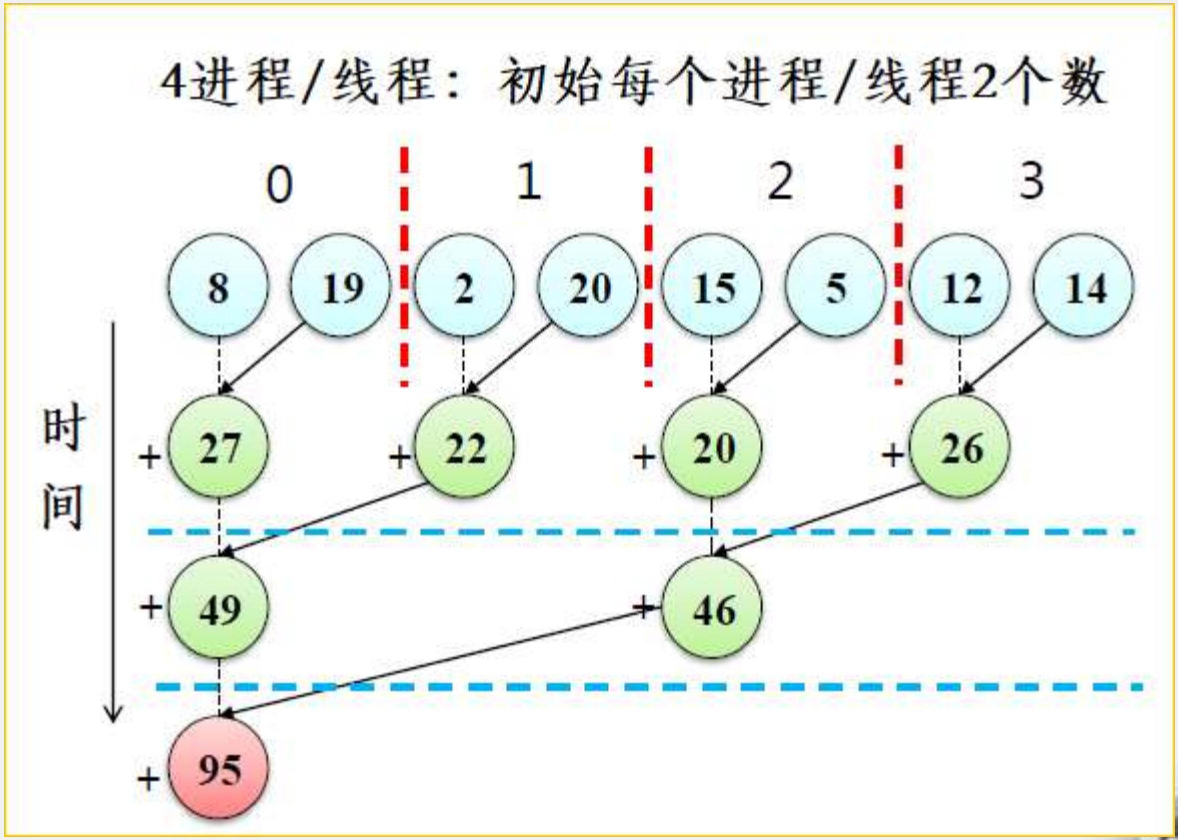
协调进程与线程

- 并行软件的主要任务：
  - 任务划分
    - 负载平衡
    - 使通信量最小
  - 同步
  - 通信



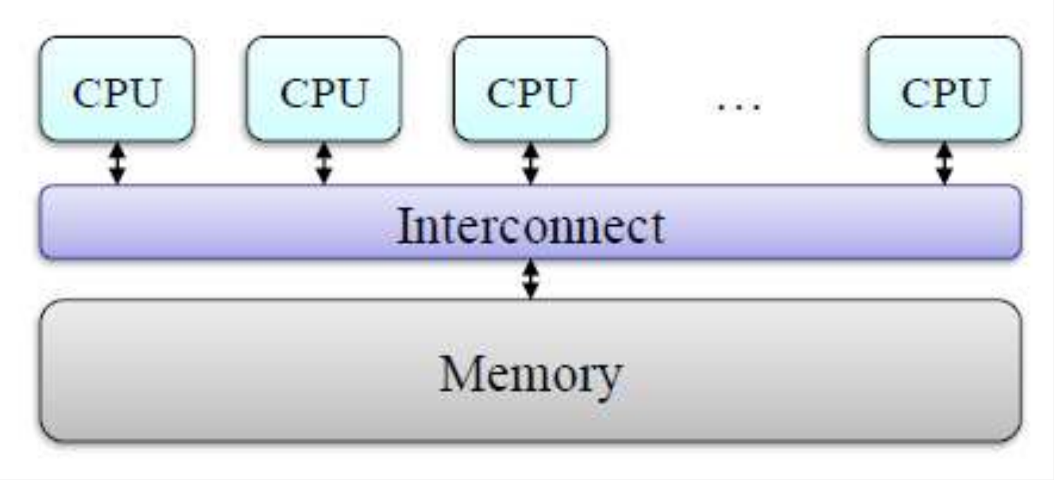
协调进程与线程

- 并行软件的主要任务：
  - 任务划分
  - 同步
  - 通信



共享内存系统中线程协调

- 共享内存系统中通常使用线程
- 共享内存系统的特点
  - 变量分为共享的和私有的：
    - 共享变量可以被任何线程读、写
    - 私有变量只能被单个线程访问
  - 通信是隐式的: 通过共享变量实现的



共享内存系统中线程协调

- 动态线程和静态线程
- 不确定性
- 线程安全

共享内存系统中线程协调

- 动态线程和静态线程
  - 动态线程：由主线程在程序运行过程中动态生成，完成一定计算任务后释放回收的线程。
  - 静态线程：由主线程按需要生成，但直到程序退出前才释放回收的线程。
- 如果有足够的资源，性能上，静态线程要优于动态线程。

共享内存系统中线程协调

- 不确定性
  - 给定的输入能产生不同的输出，这种计算称为不确定性。
  - 在任何一个MIMD系统中，如果处理器异步执行，那么很可能会引发不确定性。
  - 例子：
    - 两个线程同时输出私有变量，输出的次序不确定：
    - 情况1
      - Thread 0 > my_val = 7
      - Thread 1 > my_val = 19
    - 情况2
      - Thread 1 > my_val = 19
      - Thread 0 > my_val = 7

不确定性

```
. . .
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

不确定性

//x是共享变量，my_val是私有的

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

不确定性的原因

- 竞争条件（Race condition）
  - 当线程/进程尝试同时访问一个资源时，这种访问会引发错误，常说程序有竞争条件，因为线程/进程处于竞争状态下。即程序的输出依赖于赢得竞争的进程/线程。
  - 共享变量竞争
  - 消息传递竞争
- 动态进程调度
- 不确定的系统调用

不确定性的解决

- 建立临界区，人为排除其他线程/进程中断临界区：互斥进入临界区执行
- 互斥锁(mutual exclusion lock)
- 忙碌等待（ busy-waiting）
- 信号量（Semaphores ）

```
//使用互斥锁解决
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```

使用忙等待解决

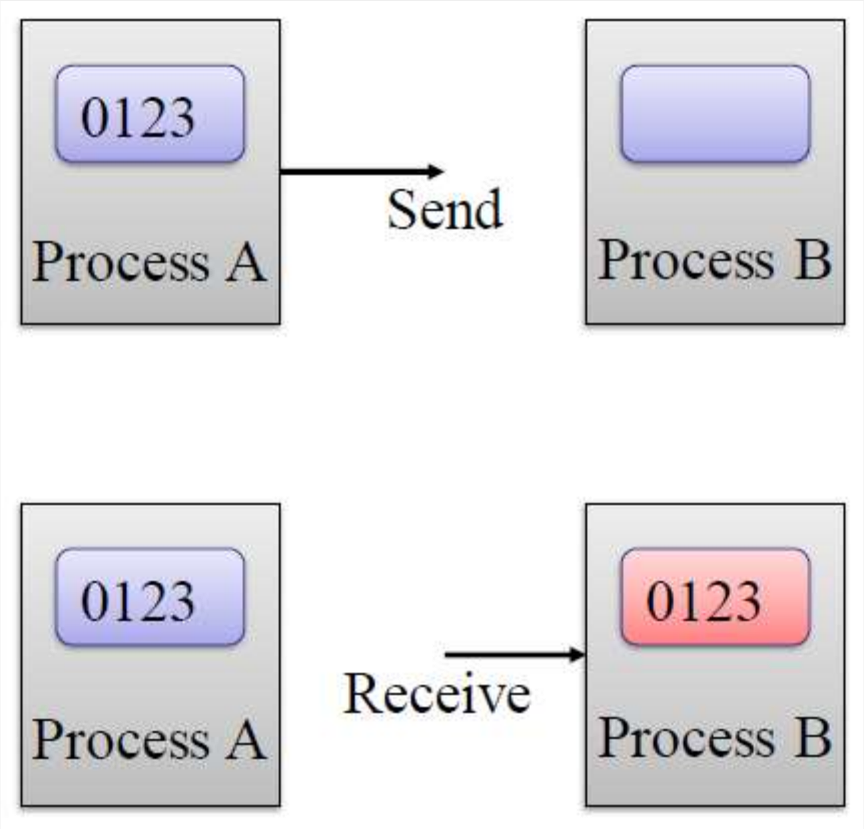
```
my_val = Compute_val ( my_rank ) ;
i f ( my_rank == 1)
    whi l e ( ! ok_for_1 ) ; /* Busy−wait loop */
x += my_val ; /* Critical section */
i f ( my_rank == 0)
    ok_for_1 = true ; /* Let thread 1 update x */
```

分布内存系统中协调

- 消息传递（Message-passing）
- 单向通信（One-sided Communication）
- 划分的全局地址空间(Partitioned Global Address Space, PGAS)模型

分布内存系统中协调

- 消息传递(Message-passing)

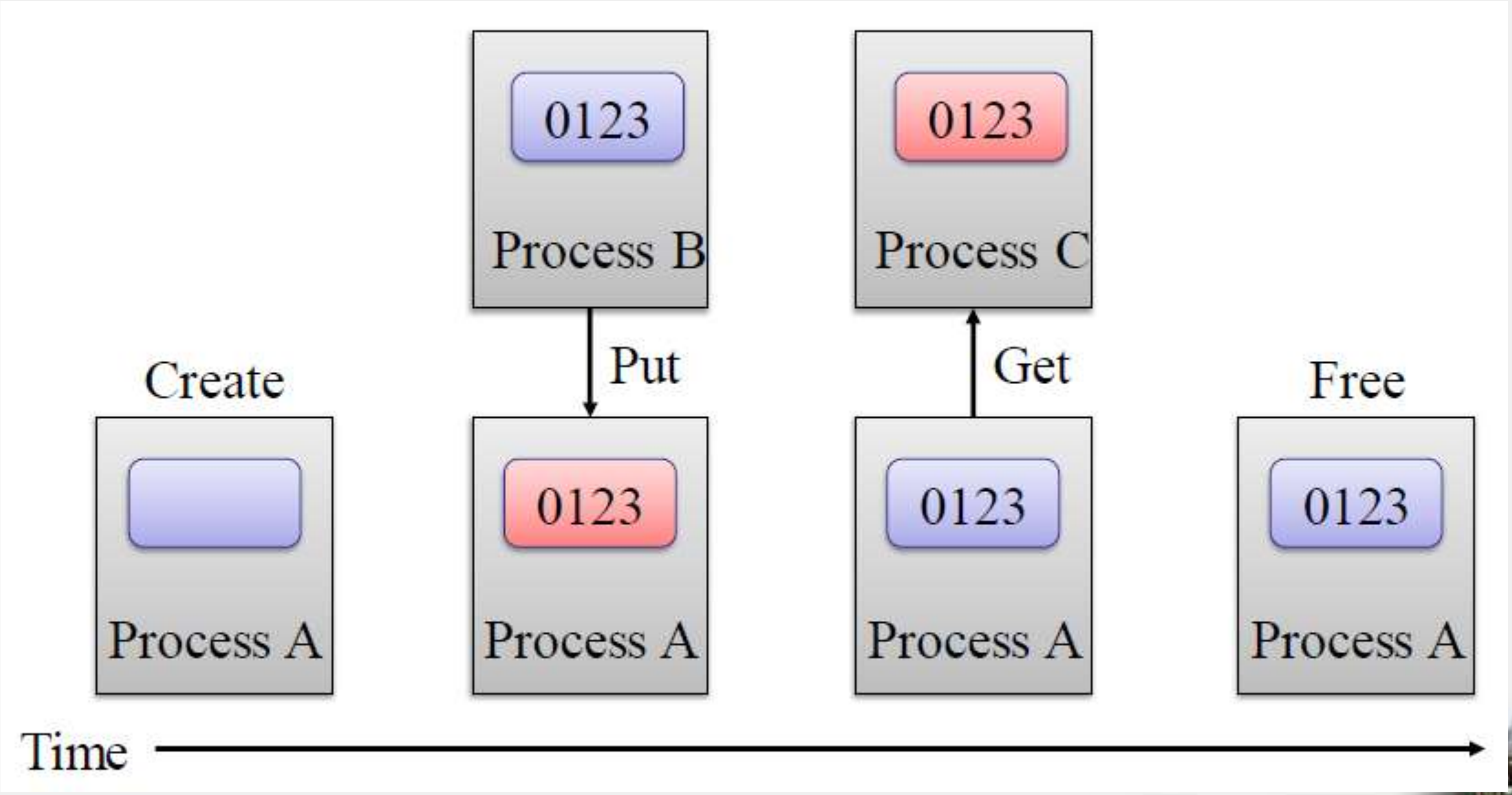


使用消息传递解决

```
char message [ 1 0 0 ] ;
. . .
my_rank = Get_rank ( ) ;
i f ( my_rank == 1) {
    sprintf ( message , "Greetings from process 1" ) ;
    Send ( message , MSG_CHAR , 100 , 0 ) ;
} e l s e i f ( my_rank == 0) {
    Receive ( message , MSG_CHAR , 100 , 1 ) ;
    printf ( "Process 0 > Received: %s\n" , message ) ;
}
//输出次序不变
```

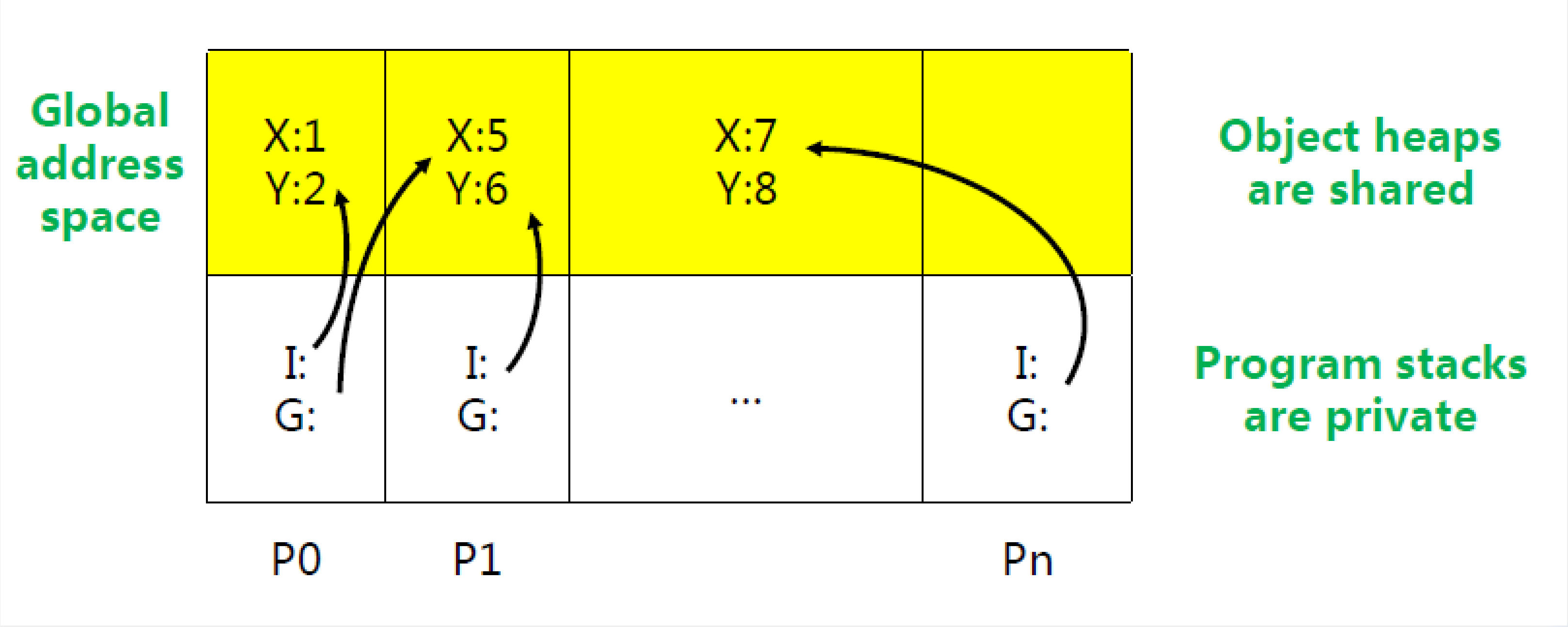
分布内存系统中协调

- 单向通信(One-sided communication)或者远程内存访问（remote memory access



分布式内存硬件上的共享内存技术

允许在分布式内存硬件上使用共享内存技术的并行编程语言。将分布式系统中所有分散的内存看做一个大内存，访问的粒度无法控制，性能不可预测——通常很差。



分布式内存硬件上的共享内存技术

将分布式系统中所有分散的内存看做一个大内存，访问的粒度无法控制，性能不可预测——通常很差。

```
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 ] ;
```

划分全局地址空间的语言 (Partitioned Global Address Space, PGAS)

- 语言提供了一些共享内存程序的机制。
  - 给程序员提供了一些工具，避免上面讨论的问题发生。
  - 私有变量在运行程序的核的局部内存空间中分配，
  - 共享数据结构中数据的分配由程序员控制。
    - 程序员知道共享数组中哪个元素是在进程的本地内存中。

混合编程

- 分布内存模型中的节点可以是共享内存的并行系统
- 所以对于更高的性能要求，还可能混合共享内存和分布内存进行编程
- 混合编程的接口常复杂

输入和输出

输入和输出的一些问题

- 当从多个进程调用printf函数时，我们想要结果输出在某一个单一系统的显示屏上。
- 调用scanf函数输入应该在各个进程间平分吗？或者只有一个进程能够调用scanf吗？
- 当多个进程能够访问stdout、stderr或者stdin时，输入的分布和输出的顺序是非确定的。怎么办？

并行的输入输出规范

- 在分布式内存程序中，只有进程0可以访问标准输入stdin。在共享内存程序中，只有主线程或线程0将访问stdin。
- 在分布式内存和共享内存程序中，所有进程/线程都可以访问标准输出stdout和stderr。

并行的输入输出规范

- 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的操作将只使用一个进程/线程。
- 调试输出应始终包含生成输出的进程/线程的列组或ID。

并行的输入输出规范

- 只有一个进程/线程将尝试访问stdin、stdout或stderr以外的任何单个文件。
  - 每个进程/线程都可以打开自己的私有文件进行读写
  - 没有两个进程/线程会打开同一个文件。

性能

性能

- 计时
- 加速比和效率
- 阿姆达尔定律（ Amdahl’s law ）
- Gustafson定律
- 可扩展性
  - 扩展性比较
  - 等效率模型

性能——计时

- 计时的目的
  - 并行程序计时的两个目的：
    - 调试程序；
    - 测试程序性能，进行调优。
  - 不只测量总时间，还要测量算法不同过程的时间；

性能——计时

- 计时**注意！！**
  - CPU时间（clock函数）不包括进程空闲等待的时间；
  - 时间的精度：不同的计时函数有不同的精度；
  - 时间的变化性：每次运行程序的时间都是不同的；
  - 如果程序中一个核会运行多个线程，会增加程序时间的变化性。

性能——计时

- 计时 例子

实际时间函数是：
MPI_Wtime
omp_get_wtime

使用挂钟时间计时

```
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);
```

性能——计时

- 计时 例子

并行计时-1

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/*    同步所有进程/线程       */
Barrier();
my_start = Get_current_time();
/*    要计时的代码              */
. . .
```

性能——计时

- 计时 例子

并行计时-2

```
my_finish = Get_current_time();
my_elapsed = my_finish - my_start;
/* 在所有进程/线程的运行时间中找最大值 */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
    printf("The elapsed time = %e seconds\n",
global_elapsed);
```

处理器核与进程/线程的关系

- 进程/线程——程序执行体
- 核——执行进程/线程的硬件
  - 与核对应，进程/线程也称为虚拟处理机
- 进程/线程与核的对应（映射）关系：m个进程/线程对应p个核
  - m>p ： 为隐藏访存或网络时延
    - p=1：多个进程/线程对应一个核
    - p>1：多个进程/线程对应多个核
  - m<p：核未充分利用
  - m=p：进程/线程与核1-1对应
  - 对算法性能的讨论时，假定m=p

性能——加速比

- 加速比S：串行计算时间与并行算时间比值。
  - T_{serial} = 串行程序运行时间；p=并行计算的进程/线程数目；$T_{parallel}$ = 并行程序运行时间

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

- 线性加速比：计算速度随进程/线程数增加呈线性增长。

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

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

性能——超线性加速比

- 加速比大于处理器数目的情况
- 成因之一——资源引起的超线性：更大的缓存或更高的存储器带宽会导致更高的缓存命中率，从而引起超线性
  - 处理器数量增加，处理器的cache总量也增加
    - 大幅降低cache缺失率：访存时间大幅降低，对实际计算产生了额外的加速效果。

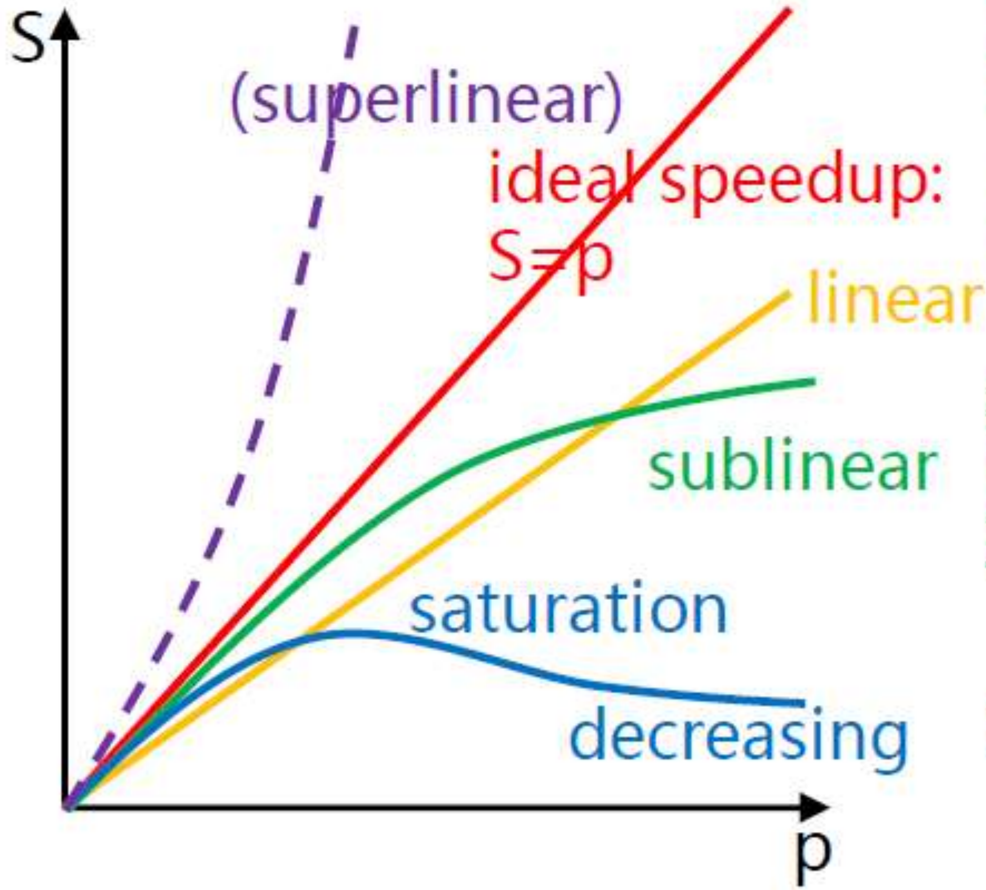
性能——超线性加速比

- 例子：一个拥有64KB缓存的处理器有 80%的缓存命中率。如果使用两个处理器，则每个处理器面对的问题规模变小了，缓存命中率上升到了90%。剩下未命中的10%，8%需要操作本地存储器而2%需要操作远端存储器。

➢ 如果DRAM的存取时间是100ns，缓存的存取时间是2ns，而远端的存储器存取时间是400ns，那么相应的加速比是2.43！

  ✓ 单处理器：DRAM的有效存取时间=2*0.8+100*0.2=21.6ns
  ✓ 2个处理器：有效存取时间=2*0.9+100*0.08+400*0.02=17.8ns
  ✓ 计算：1FLOPS/访存时间秒
  ✓ 单处理器：1FLOPS/21.6ns=1/(21.6/1000M) FLOPS=46.3M
  ✓ 2个处理器：2*1FLOPS/17.8ns=2*1000/17.8MFLOPS
  ✓ =112.36M
  ✓ 加速比=112.36M/46.3M=2.43 > 2

性能——加速比曲线



细粒度并行
(例：矩阵相乘, LU分解，光线跟踪)
接近理想$S = p$

工作(计算)约束算法
线性$S \in \Theta(p)$, work-optimal

类树状结构的算法
(例： 全局最大值/和)
亚线性$S \in \Theta\left({}^{p}/_{\log p}\right)$

通信约束算法 $S = {}^{1}/_{f_n(p)}$

性能——并行程序的效率

效率：加速比与进程/线程数的比值。

- 理论上p个进程/线程应有p倍加速，实际S<=p。所以，效率为S占p的百分比，也就是说算法发挥了p个核的计算能力的百分比。

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

性能——加速比与效率的关系

- 对**n=256**个数计算的例子

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

性能
——加速比、效率与规模的关系

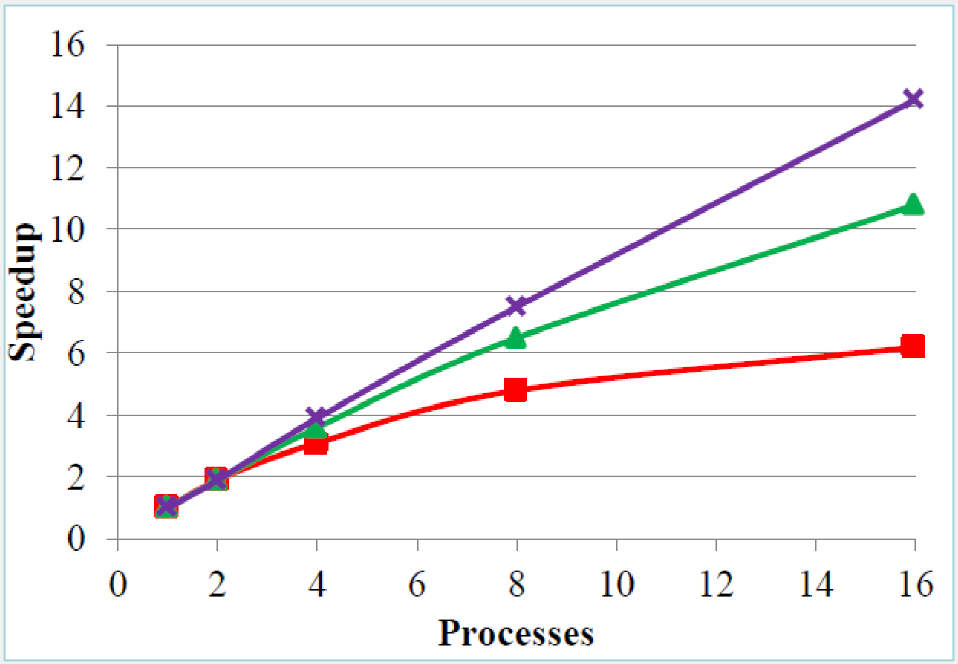
- 对**n=256**个数计算的例子

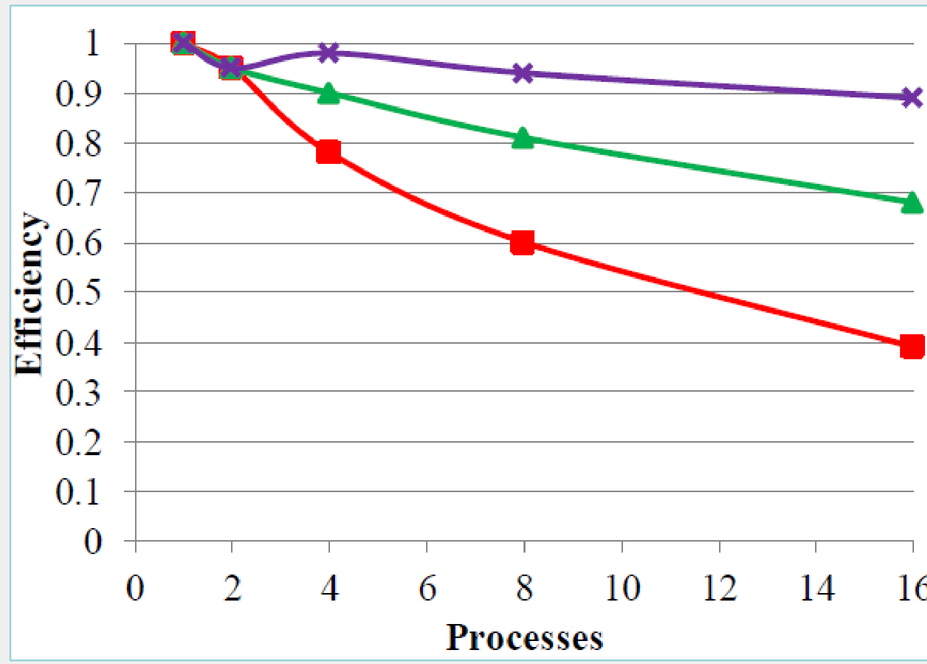
	p	1	2	4	8	16
Half n=128	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original n=256	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double n=512	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

性能

——加速比、效率与规模的关系

- 对**n**个数计算的例子





性能
——阿姆达尔定律(Amdahl’s Law)

- 除非串行程序都是完全可并行的，即没有不可并行部分，否则可能的加速将是极其有限的——无论使用多少进程/线程(核)。

阿姆达尔定律——例

- 假设：
  - 有一个可以并行90%的串行程序
  - 并行部分的时间对于节点数是线性增长的
  - 串行计算的时间T_{serial}是20s
- 可推出
  - 并行部分的运行时间：$0.9 \times T_{serial} / p = 18 / p$
  - 不可并行部分的运行时间：$0.1 \times T_{serial} = 2$
  - 整个并行计算的运行时间：

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

阿姆达尔定律——例(cont.)

- 加速比：

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

$$S < \lim_{p \to \infty} \left(\frac{20}{18/_{p} + 2} \right) = 10$$

- 即此例的加速比S不会超过10。

性能——阿姆达尔定律

- 阿姆达尔定律是固定负载（计算总量不变时）时的量化标准。假定单位负载的计算时间为固定常数
- 设W=Ws+Wp是串行程序的工作负载
  - Ws是其不可并行部分的负载——串行分量
  - Wp是其可并行部分的负载——可并行分量
- 加速比 $S = \dfrac{W}{Ws+\dfrac{Wp}{p}} = \dfrac{Ws+Wp}{Ws+\dfrac{Wp}{p}}$
- 只要注意到当时 $p\rightarrow\infty$，上式的极限是 $\dfrac{W}{Ws}$。
- 意味着无论我们如何增大处理器数目，加速比是无法高于这个数的。

性能——阿姆达尔定律

- 阿姆达尔定律揭示了
  - 无论我们如何增大处理器数目，加速比是存在极限的。
    - 通常情况下，5到10的加速比能达到要求。
  - 用串行分量比例改写加速比
    - 串行分量比例

$$f=\frac{W_s}{W}$$

    - 加速比

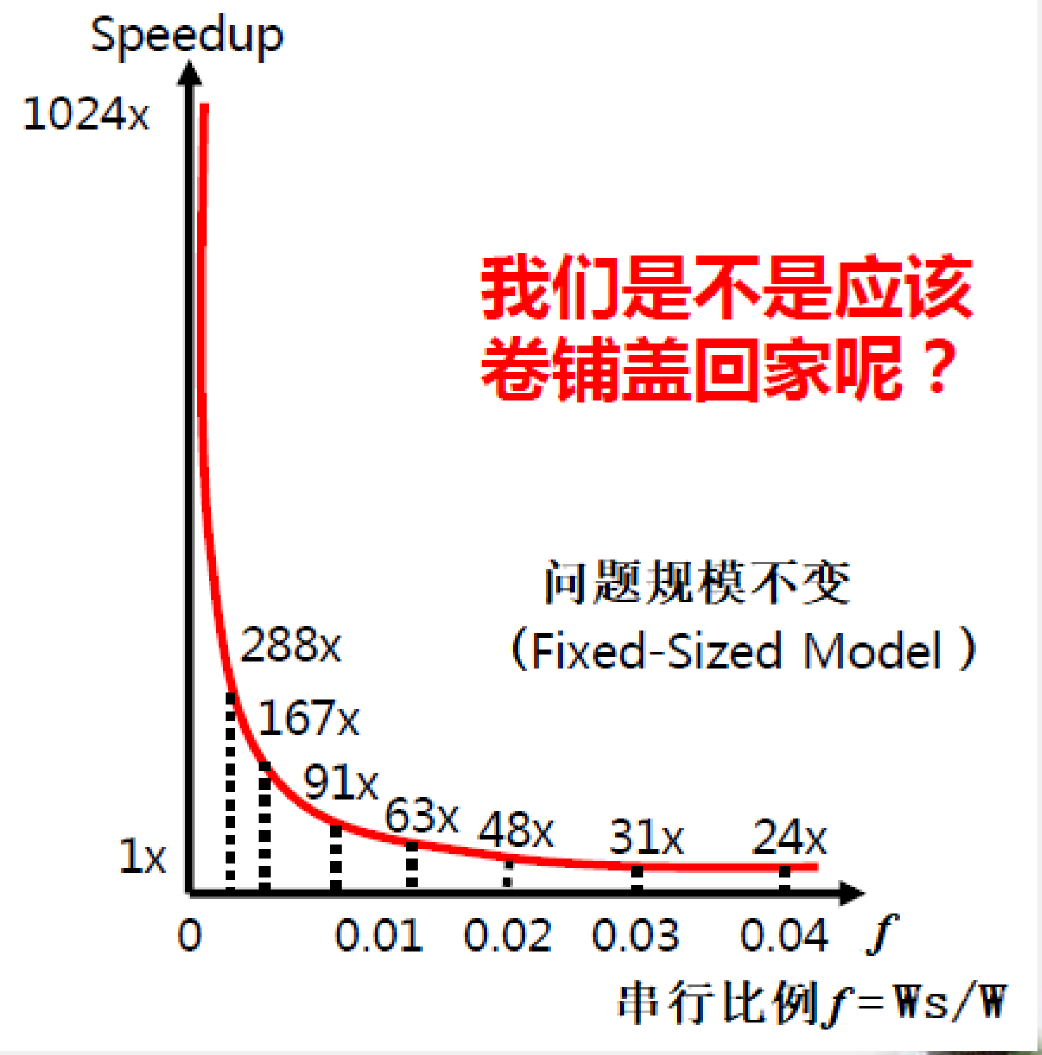
$$S=\frac{W}{W_s+W_\mathrm{p}/p}=\frac{1}{f+(1-f)\,/p}\to\frac{1}{f}\quad(p\to\infty)。$$

性能——阿姆达尔定律

- 前例使人气馁：

  即使做了这么多工作，却也是徒劳。

  – 随串行分量比例 f=Ws/W 的增加，加速比减小



性能——阿姆达尔定律

- 阿姆达尔定律的局限性
  - 没有考虑问题规模的变化：假定了规模不变
- 通常问题规模增大的时候，串行分量占的比例会减少——古斯塔夫森(Gustafson)定律。

性能——Gustafson定律

- Gustafson提出问题规模可变的加速比模型
  - p个进程并行执行时，串行负载W_s执行时间不变，而并行负载扩大为pW_p，此时加速比

$$S = \frac{W_s + \mathrm{p}W_\mathrm{p}}{W_s + \mathrm{p}W_\mathrm{p}/p} = f + (1 - f)\,p,$$

  - 加速比与处理器数目成正比，串行分量不再是瓶颈

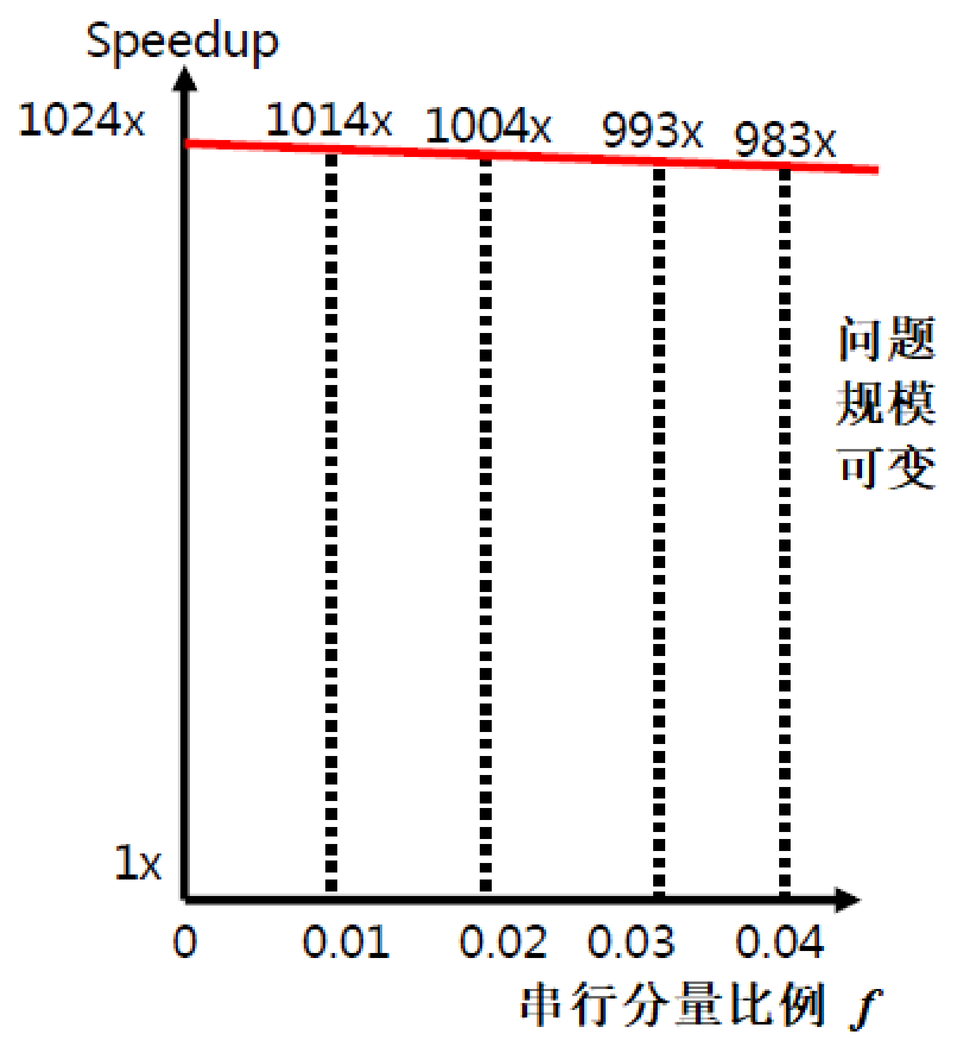
性能——Gustafson定律

- Gustafson加速比模型

$$S = \frac{W_s + pW_p}{W_s + pW_p/p}$$

$$= f + (1 - f)\,p,$$

  - 加速比与处理器数目成正比，串行分量不再是瓶颈
  - $E = \frac{f}{p} + (1 - f)$
  - $E \rightarrow 1 - f, (p \rightarrow \infty)$



性能——可扩展性定义

- 如果我们同时增加问题规模和进程/线程数，并行程序的效率能基本保持不变，就说这个程序是可扩展的。
- 当我们增加进程/线程数，为了维持效率而增加的问题规模不大时，就说这个程序有强可扩展性。
- 如果问题规模的增加的比率与进程/线程数增加比率一致，就说这个程序有弱可扩展性。

性能——可扩展性比较

- 如果我们同时增加问题规模和进程/线程数，并行程序的效率能基本保持不变，就说这个程序是可扩展的。
  - 这里并没有要求问题规模增加速度和进程/线程数增加速度一样。注意：前述的强弱可扩展性规定了速度。
- 问题：两个算法，哪一个的可扩展性好？

  下图：p不变，效率随n增大; n不变，效率随p减小

用p个处理器相加n个数时，效率作为n和p的函数

n	$P=1$	$P=4$	$P=8$	$P=16$	$P=32$
64	1.0	0.80	0.57	0.33	0.17
192	1.0	0.92	0.80	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	0.80	0.62

性能——可扩展性比较

- 问题：两个算法，哪一个的可扩展性好？
  三个参数：效率E，进程/线程数p，问题规模n，
  效率是p和n的函数：E=f(p,n)。
  - 对同样的p和n，E更大的好（S=Ep更大）
  - 对同样的E和n，p更大的好（S=Ep更大）
  - 对同样的E和p，n更小的算法可扩展性好，或者说，若E相等，p增速一样，则n增速慢的好

➢ 若E不变（为常数）， 由E=f(p,n)，可得n=g(p)：这时g(p)小的可扩展性好。

■ 如何计算g(p)后面再讨论这个问题

性能——可扩展性

- 强可扩展性——问题规模增加不大，不妨假定规模固定，其效率

$$E = \frac{S}{p} = \frac{W}{p\left(W_s + \frac{W_p}{p}\right)} = \frac{1}{fp + (1-f)}$$

  - 如果$f \neq 0$，即有串行分量，则

$$E = \frac{1}{fp + (1-f)} \rightarrow 0 \text{（}p \rightarrow \infty\text{）。}$$

  - 除非程序都是完全可并行的（即没有不可并行部分），其效率随p下降至0.

性能——并行开销(overhead)

- 将串行程序并行化时，将W分为Ws和Wp，实际做不到：
  - Ws难于计算；
  - 在并行化Wp时，仍会引入串行计算部分：同步互斥和通信等局部或全局串行部分，情况复杂，不能简单地归为W的串行部分
- 引入并行开销(overhead)：易于计算

性能——并行开销(overhead)

- 并行计算的总成本 = **p**$T_{parallel}$

——相当于将p个进程串行执行时间。

总并行开销 = $pT_{parallel}$ - T_{serial}

- 或者这样说

———每个进程的执行时间应为T_{serial} / p，但实际为$T_{parallel}$，每个进程增加的部分称为（额外）并行开销：

$$T_{overhead} = T_{parallel} - T_{serial} / p$$

即，$T_{parallel} = T_{serial} / p + T_{overhead}$

总并行开销 = $pT_{overhead} = pT_{parallel} - T_{serial}$

额外增加的主要是额外的计算、通讯、空闲和资源争用导致的开销

并行程序的开销来源

- 进程间的交互： 任何非平凡的并行系统都要求其处理器交互和传送数据，包括资源争用同步互斥时
- 空闲：处理器会由于负载平衡，同步或程序的串行部分而空闲
- 额外计算：这部分计算在串行版本中是没有的。
  - 差的串行算法，但并行性好（好的串行算法，但并行性差），所带来的额外计算。
  - 利用重复计算来减少通讯——这里重复计算导致的额外计算。

性能——可扩展性——用并行开销表示

- 强可扩展性——不妨假定问题规模固定，其效率

$$E = \frac{S}{p} = \frac{T_{serial}}{p(T_{serial}/p + T_{overhead})} = \frac{1}{1+p\frac{T_{overhead}}{T_{serial}}}$$

  - 如果$T_{overhead} \neq 0$，即有并行开销，则

$$E = \frac{1}{1+p\frac{T_{overhead}}{T_{serial}}} \rightarrow 0 \text{（} p \rightarrow \infty \text{）。}$$

  - 除非程序都是完全可并行的（并行开销为0），其效率随p下降至0.

性能——可扩展性

- 弱可扩展性——问题规模n的增加的比率与进程/线程数p增加比率一致时，其效率E不变。
- 然而，效率是p和n的函数E=f(p,n)。在大多数情况下，n与p之间具有某个更一般的函数关系时，效率E不变。即存在函数g，n=g(p)时，有f(p,g(p))=常数。
  - 问题：为了保持效率恒定，问题规模n要以怎样的速度随处理器个数p增加而增加?
    - **找出这个速度，即找出函数g**
      - 精确求出这个函数是困难的，但可以近似估计
- 先引入符号Θ、O、Ω，进而引入（渐近）成本最优概念

阶的符号Θ、O、Ω

- 为方便，引入阶的符号：
  - Θ符号：给定非负函数$g(x)$, 我们说 $f(x)= \Theta(g(x))$，当且仅当存在常数$c_1,c_2>0$和点x_0, 并且对所有$x>x_0$,函数$f(x)$满足不等式

$$c_1 g(x) \le f(x) \le c_2 g(x)。$$

  - O符号：给定非负函数$g(x)$, 我们说 $f(x)= O(g(x))$，当且仅当存在常数$c>0$和点x_0, 并且对所有$x>x_0$,函数$f(x)$满足不等式

$$f(x) \le c g(x)。$$

  - Ω符号：给定非负函数$g(x)$, 我们说 $f(x)= \Omega(g(x))$，当且仅当存在常数$c>0$和点x_0, 并且对所有$x>x_0$,函数$f(x)$满足不等式

$$c g(x) \le f(x)。$$

并行算法的成本和最优成本

- 并行算法(总)成本 = $pT_{parallel}$
  - 成本反映了用于解决问题的累计总时间
- 串行算法(总)成本 = T_{serial}
  - 用求解同样问题的已知最快的串行算法与并行算法比较
- 如果一个并行算法的成本与串行成本渐近相等，则称它为(渐近)成本最优的:
  - $T_{parallel} = \Theta(T_{serial}/p)$ 或 $T_{serial} = \Theta(pT_{parallel})$。
  - $E = \dfrac{T_{Serial}}{p\,T_{parallel}} = \Theta(1)$。
  - 这里渐近的意思是上面等式在p充分大时成立。

性能——可扩展性

- 如果我们用时间单位计量负载W，则
  - $W = T_{serial}$
  - 简记 $T_o = pT_{overhead}$
  - 效率 $E = \frac{1}{1+\frac{T_o}{W}}$
  - 总并行开销$T_o=T_o(W,p)$是W和p的函数
    - 可用W表示问题规模——假定W是n的递增函数的（当n充分大时），其他非平凡情形均可近似为这样。
    - 同理，假定总并行开销函数T_o 是p的递增函数。
    - 对于给定的问题规模（即， W保持不变），当我们增加了处理器个数时，T_o增加。并行程序的效率随之下降。这对所有非平凡的并行程序都是适用的。

并行程序的扩展特性：例子

- 考虑使用***p***个处理器进行***n***个数的加法运算（按树形计算）
- 我们可以看到

$$T_{parallel} = \frac{n}{p} + 2\log p$$

$$S = \frac{n}{\frac{n}{p} + 2\log p}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$

可扩展性与成本最优

- 同时增加问题规模和进程/线程数，并行程序的效率能基本保持不变——程序是可扩展的。
  - 问题规模和进程/线程数按照某种函数关系增加
  - 并行系统的可扩展性与成本最优性是相关的
    - 如果进程/线程数p和计算规模W适当地选定，一个可扩展的并行程序总是可以成为成本最优的

并行程序的扩展特性：例子

- 考虑使用***p***个处理器进行***n***个数的加法运算（按树形计算）
- 我们可以看到

$$T_{parallel} = \frac{n}{p} + 2\log p$$

$$S = \frac{n}{\frac{n}{p} + 2\log p}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}}$$

当n= Θ(plog p)时，算法成本最优。

并行程序的扩展特性：例子

用p个处理器相加n个数时，效率作为n和p的函数

n	$P=1$	$P=4$	$P=8$	$P=16$	$P=32$
64	1.0	$\boxed{0.80}$	0.57	0.33	0.17
192	1.0	0.92	$\boxed{0.80}$	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	$\boxed{0.80}$	0.62

- n=Θ(plog p)
- n=64，p=4，E=0.8，此时n=8plogp
- p=8， n=8*8log8=192 → E=0.8
- p=16，n=8plogp=512， → E=0.8

可扩展性——等效率模型

由前述推导，程序效率

$$E = \frac{1}{1 + T_o(W,p)/W},$$

- 保持 T_o / W 为一个常量时，即可获得固定的效率——程序是可扩展的。

可扩展性——等效率模型

上式移项得

$$\frac{T_o(W,p)}{W} = \frac{1-E}{E}, \quad \rightarrow \quad W = \frac{E}{1-E}T_o(W,p).$$

- 效率E不变，则E是常量，那么 $K = \frac{E}{1-E}$ 也是常量，因而T_o 是 W 和 p 的函数，

$$W = KT_o(W,p).$$

由此等式可获得W关于p的函数，即等效率函数。

- 较小的等效率函数意味着较好的可扩展性。
- 若W随p线性增长，则扩展性很好；
- 若W随p指数增长，则扩展性较差。

等效率函数：例子

- 在p个进程/线程上进行 n个数的加法的总并行开销函数大约是$2p\ \log\ p$。
- 将前式中的T_o 换为 $2p\ \log\ p$ ， 可得

$$W = K2p\log p.$$

- 因此，这个并行系统的渐近等效率函数是 $\Theta(p\log p)$

如果处理器个数从***p***增加到***p'***， 问题规模（在这个例子中是***n***）必须以(***p'* log *p'***) / (***p* log *p***) 的比例增长，才能在***p'***个处理器上获得相同的加速比。