第5章

并行程序性能优化

前面几章介绍了几种主流的并行编程语言和接口，在掌握基本并行编程知识的基础上，本章将重点讨论常用的并行设计及性能优化方法，即如何使一个并行程序的性能更优、可扩展性更好。这在并行编程中常被称为调优（tuning），其中，狭义的性能调优是指并行程序编写完成后，进一步改进程序以优化其性能；但广义来讲，并行程序的性能优化应当覆盖并行编程的全过程，特别是程序设计初期的算法设计和并行划分，其对程序性能的影响远大于后期的一些局部改进优化，本章将从多个角度讨论这一问题。

5.1 Amdahl 定律

阿姆达尔（Amdahl）定律是计算机系统设计的重要原理，由 IBM 360 计算机的主要设计者 Gene Amdahl 于 1967 年提出。**该定律的基本思想是：由于系统中某一部件采用更快的执行方式后，整个系统性能的提升取决于这种执行方式的使用频率，或占总执行时间的比例。**

Amdahl 定律原本是设计计算机系统的指导性原则，这里从程序并行化的角度对该定律进行解读。其**核心思想是：设计并行程序或对并行程序进行性能优化时，应优先选取执行频度高，或在程序总执行时间中占比高的部分进行并行化或性能优化。**

根据 Amdahl 定律，一个程序包含两部分：并行部分和串行部分。假设并行部分的执行时间为 T_p，串行部分的执行时间为 T_s，则串行部分执行时间在总执行时间中的占比为

$$f=\frac{T_s}{T_p+T_s} \tag{5-1}$$

当使用 n 个处理器执行该程序时，其并行部分的执行时间可降为原来的 $1/n$，而串行部分的执行时间不受影响。这样，加速比计算公式为

$$S=\frac{1}{f+\frac{(1-f)}{n}} \tag{5-2}$$

根据以上公式，对于给定的处理器个数 n，如果 f 越小，则（$1-f$）越大，

可以获得的加速比也越高。这意味着：**一个程序的并行部分执行时间在总执行时间中占比越高，可以获得的加速比越高。**

图 5-1 给出了不同 f 取值下的加速比曲线。可以看出，f 越小，可以获得的加速比越接近于线性，当 $f=0.1\%$ 时，使用 256 个处理器可以获得超过 200 倍的加速比；而同样使用 256 个处理器，$f=1\%$ 和 $f=5\%$ 可以获得的加速比仅为 72 倍和 18 倍。之所以出现这种现象，是由于**程序中的串行部分拖累了整体的并行加速效果**。

如果从极限角度分析 Amdahl 定律的加速比计算公式（5-2）可以得出，当处理器个数 n 无限增大时，加速比将无限接近于 $1/f$，这实际上设定了加速比的上限值，故可以直观地将其理解为加速比的“天花板”。另一方面，还可以看到，$f=5\%$ 时的加速比上限值为 20 倍，当处理器个数为 32 时，就已经获得 12 倍的加速比，而在处理器个数从 32 增加到 256 后，加速比也仅从 12 倍增加到 18 倍。从这里也可以得出一个启示：**即使 f 较大，通过小规模并行也可以获得较好的加速效果。**

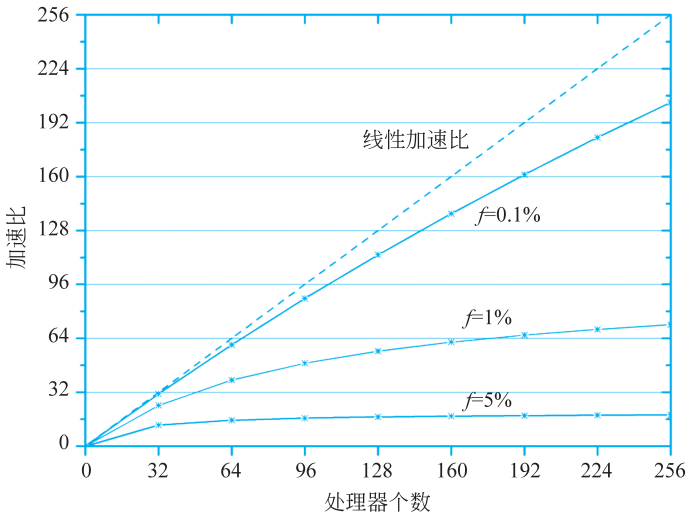


图 5-1　不同串行时间占比下的加速比

Amdahl 定律除了告诉人们获得线性加速比的难度之外，还给出了程序性能优化的重要启示，即**进行并行性能优化时，应聚焦在程序执行时间中占比最高的部分**。注意这里所说的“占比”不是指代码行数的多少，而是执行时间的占比。对于大多数计算复杂度较高的程序，其执行时间往往呈现出 90/10 特征，即程序执行时间的 90% 耗费在 10% 的代码上，这就是通常所说的**“密集计算部分”**。根据 Amdahl 定律，开发者应优先选择“密集计算部分”进行并行化，或对其进行性能优化。设想一个使用多重循环进行计算的程序，其多重循环部分的代码行数在总代码行数中仅占很小比例，但执行时间却占了总执行时间的绝大部分，这就是典型的“密集计算部分”。通过对多重循环进行并行化，就能以较小的代价获得更大的收益（加速效果）；又如，多重循环的最内层循环语句执行频率高，在并行化基础上对这部分语句进一步优化（如减少互斥 / 同步、改进数据局部性等）也可能获得较为明显的加速效果。

5.2 影响性能的主要因素

性能是并行计算最重要的目标之一。限制并行性能的因素有很多，包括并行开销、并行粒度、负载均衡等。评价性能的指标一般为计算的加速比。更具体地说，性能有两个衡量指标：**延迟**和**吞吐量**。延迟即完成单位工作所花费的时间，吞吐量即单位时间内完成的工作量。

并行性通常用来提高吞吐量，第 1 章中提到的指令级并行就是提高吞吐量的例子。假设流水线微处理器将指令执行分为五个阶段，通过指令级并行可以同时执行五条指令，因此吞吐量就有可能提高五倍，这也意味着程序执行时间减少。并行性也可以用来隐藏延迟，例如，处理器可以无须等待长时间执行的操作，而通过切换上下文执行另一个进程或线程。注意这种延迟隐藏技术实际上并没有减少等待时间，而只是隐藏了等待而导致的执行时间的损失。下文将分别介绍影响并行性能的主要因素。

5.2.1 并行开销

相比串行方案，并行方案产生的任何成本都被看作是并行开销。首先，创建或释放线程和进程会产生开销，由于内存分配及其初始化的开销很大，因此进程相比线程会产生更多的开销。除此之外，并行开销还有一些其他来源，主要包括通信、同步和资源争用等。下面对此进行分析。

1. 通信

线程和进程之间的通信是并行开销的主要来源。由于串行计算中处理器之间不需要通信，因此所有通信都可以被看作并行开销的一种。如果观察单个处理器上单个任务的计算时间，会发现它可以被粗略地划分为计算时间、通信时间和闲置时间。当任务需要等待系统传输的某条消息时，很可能会出现闲置。如果将进程和线程之间的通信时间考虑在内，则可以对并行程序的性能给出更实际的预测。

通信的具体开销取决于硬件。共享内存系统和消息传递系统的硬件通信开销构成有很大不同。在共享内存系统中，通信开销的主要来源是访存延迟、一致性操作、互斥和争用；而在消息传递系统中，通信开销的主要来源则是网络延迟、数据编组、消息形成、数据解组和争用。

2. 同步

当一个线程或进程必须等待另一线程或进程上的事件时，就会产生同步开销。对于消息传递系统而言，由于消息的传送必须在消息的接收之前，因此大多数情况下同步是隐式的。而对于共享内存系统而言，同步往往是显式的，这意味着开发者必须显式地指定某段代码需要在线程之间互斥执行（即临界区）。如图 5-2 所示，一个线程可能等待其他线程完成计算或释放资源。线程获取和释放锁的过程也是同步开销，因为串行代码不需要这些操作。如果获取锁的次数过多或频繁地在所有线程间同步，那么由此产生的同步开销往往非常巨大。

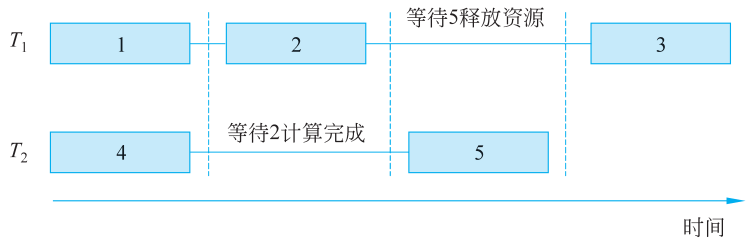


图 5-2　线程之间的同步

3. 资源争用

对共享资源（如 cache、总线、内存、锁等）的访问冲突会产生资源争用。作为并行开销的一种特殊情况，资源争用可能会导致系统性能明显下降，甚至会使并行执行的性能比串行执行还差。例如，当两个或多个处理器交替地重复更新同一个缓存行（cache line）时会发生 cache 争用。cache 争用有两个主要来源：内存争用和伪共享（false sharing）。内存争用即两个或多个处理器尝试更新同样的变量；伪共享即多个处理器更新占用同一个缓存行的不同变量。伪共享会导致同一个缓存行在不同的缓存之间不断被置为无效并最终访问主存。在这种情况下，总线流量的增加会影响所有处理器的性能。锁争用可能在内存中产生高负载，进而降低性能。如果为自旋锁（spin lock），则等待线程会重复检查该锁的可用性，从而增加总线流量。与伪共享类似，这种争用会影响所有尝试访问共享总线的线程。

5.2.2　负载均衡

负载均衡（load balance）是指在并行程序运行过程中，系统的每个处理器始终都在计算。如果存在任务量分配不均的情形，例如，有的处理器任务计算时间长，而其他处理器任务计算时间短，那么就会出现有些处理器进行计算而另一些处理器空闲的情形，这被称为负载不均衡（load imbalance）。图 5-3 给出了一个负载不均衡的例子，如图所示，一个程序在 4 个处理器上并行执行，在整个执行过程中，处理器 4 始终在计算，处理器 1 仅有少量时间空闲，而处理器 2 和处理器 3 在程序启动后不久就先后进入空闲状态（这通常意味着分配给它们的进程 / 线程已完成了计算）。这种负载不均衡会导致计算资源闲置，必然会对程序的性能造成不利影响。

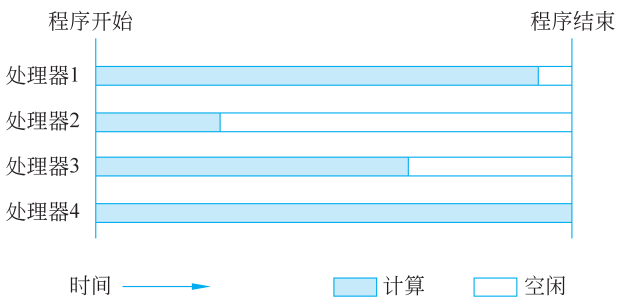


图 5-3　处理器间负载不均衡示意

对于较为规则的计算任务，通过静态的任务划分就可以实现负载均衡。例如，要实现

并行的矩阵相乘，无论是按行、按列、按块划分数据，每个处理器分到的计算量都将是相同的。对于不规则或者动态的计算任务，要实现负载均衡的难度就更大一些。例如，在求解线性方程组的 LU 分解中，虽然同样是针对矩阵进行计算，但高斯消去过程是从矩阵的左上角向右下角逐步进行的，此时如果仍然按矩阵的行 / 列 / 块平均划分数据，则随着计算过程的进行，负责矩阵左上部分的处理器在完成计算任务后就会先后进入空闲状态，从而导致负载不均衡；又如，在对树结构进行搜索时，不同子树的宽度或深度可能存在较大差异，进而导致不同子树的搜索计算量大小不一，而且这种差异无法事先确定。因此，在静态数据划分难以保证负载均衡的情形下，就需要采取更加灵活的数据划分方法或动态任务分配策略。例如，不按照处理器个数等分数据，而是将数据划分成数量更多的小块，然后分配给各处理器；又如，在计算量动态变化难以预知的计算中，维护一个动态的待计算任务列表或任务池，然后根据处理器负载情况动态地分配任务。

5.2.3 并行粒度

并行粒度（grain）又被称为并行颗粒，它由线程或进程之间交互的频率确定，可根据交互指令的数量来衡量。业内常用粗粒度（coarse grain）、细粒度（fine grain）来定性地描述程序的并行粒度。**粗粒度**指的是线程或进程很少依赖其他进程或线程的事件，而**细粒度**则需要线程或进程频繁交互。每次交互过程中都会引入通信或同步的开销。另外，并行粒度也可以用**大**和**小**描述。例如，可以把循环中的所有迭代分布到多个处理器上并行执行，处理器数量越多，意味着并行粒度可以被拆分得越小。

在实际应用中，并不存在适用于所有情形的固定并行粒度。相反地，并行粒度需要与硬件平台及应用特点相匹配。例如，共享内存系统可以支持更细粒度的计算；而对于消息传递系统，由于节点间通信延迟大，粗粒度往往更好。除硬件系统外，并行粒度的选择还与应用的计算模式密切相关。例如，扫描（前缀求和）操作是一种细粒度的计算，其计算量较少但涉及相邻线程的细粒度交互，在这种情形下，运行在多核芯片上的线程可以在处理器之间实现低延迟通信。又如，在流体模拟中，计算域通常被表示成具有离散位置的网格；在消息传递系统中，这些网格将被分配给不同的进程以计算网格点数值；此时，域中边界节点的更新需要一个或多个相邻进程的信息，在下一次域更新之前，所有节点需要与相邻子网格节点进程进行通信；由于只需在每次迭代时进行一轮通信，这种粗粒度的计算在消息传递系统上也可以获得很好的性能表现。

5.2.4 并行划分

并行划分就是将计算和数据分为多个部分的过程。良好的并行划分会将计算和数据分成许多小块以提高并行性。对于共享内存系统，**数据划分**往往被隐含于**计算划分**中。尽管如此，由于内存的物理分布特征和数据之间的复杂相关性，数据划分在大部分情况下仍需要由开发者完成。在进行数据划分时，要尽量减小数据块之间的通信开销；此外，并行划分还要平衡处理器间的工作负载。如果处理不当，处理器分配的工作量可能不成比例，从而引起严重的负载不均衡问题。

以矩阵相乘 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$ 为例，有几种可选的并行划分方案。最直接的方案是将矩阵 $\boldsymbol{C}$ 划分为行数据块的集合（即按行划分）。由矩阵相乘的运算规则可知，计算矩阵 $\boldsymbol{C}$ 的每一

个行需要矩阵 ***A*** 和矩阵 ***B*** 的相应行。在这种方案下，单元任务就是计算矩阵 ***C*** 中行的元素。更高效的方案是将三个矩阵都划分成若干个数据块或子矩阵，这样每次计算时不需要复制整个矩阵 ***A***。因此，单元任务变成了矩阵 ***C*** 中数据块的更新。随着计算的进行，矩阵 ***A*** 和矩阵 ***B*** 的数据块周期性地出现在单元任务中。这种并行划分方案增加了编程的复杂性，且需要结合硬件架构而在通信和计算间做出权衡。

对矩阵相乘来说，其计算复杂度为 $O(N^3)$，访存次数为 $O(N^2)$。可以看出，访存相对于计算的比率较大。对于此类问题，并行划分方案要考虑对内存的访问模式以提升缓存命中率。具体来说，可以使用细粒度的数据块划分，并调整数据块的大小，使其更适用于缓存。在特定硬件架构（如 GPU）中，开发者也可以显式地将重用率较高或线程共用的数据放置在访存更快的内存层次中（如 shared memory）。

5.2.5　依赖关系

1. 依赖关系简介

为了保证计算的正确性，有些时候开发者必须保证计算步骤之间的先后顺序，这就是依赖关系（dependency），又被称为相关性。依赖关系可以分为控制依赖和数据依赖，但无论是控制依赖还是数据依赖都要求计算步骤必须按顺序进行，这在很大程度上限制了并行性。

代码中一般有两种常见的依赖关系：**循环承载依赖**和**内存承载依赖**。对于循环承载依赖，一个循环体的前后轮循环之间存在依赖关系，这要求执行下一轮循环之前必须得到上一轮的计算结果；对于内存承载依赖，对内存中同一位置的两次访问必须确定先后顺序。依赖在很大程度上限制了并行性。因此，依赖可以用作分析潜在性能损失的来源。假设确知存在依赖关系，即使不知道计算的哪个部分导致了先后顺序关系，也可以推理出并行化后的性能结果。接下来将以并行计算中最为常见的数据依赖为例，进一步阐述依赖的表现形式。

2. 数据依赖

数据依赖即一组内存操作的顺序，必须保持该顺序以保证程序的正确性。

如表 5-1 所示，数据依赖分为 4 种类型：**真依赖**（true dependence）、**反依赖**（anti dependence）、**输出依赖**（output dependence）和**输入依赖**（input dependence）。真依赖即写之后读，其代表了内存操作的基本顺序；反依赖即读之后写，其必须在覆盖数据前读取该数据；输出依赖即写之后写，写的顺序决定了该数据的最终结果；输入依赖即读之后读，其不存在依赖，读的顺序不影响该数据的最终结果。反依赖和输出依赖被统称为**伪依赖**，因为它们是由内存重用而不是操作的基本顺序引起的。尽管输入依赖对于内存操作没有顺序约束，但其可以被用于推理时间局部性。

表 5-1　数据依赖的类型

比较项	真　依　赖	反　依　赖	输 出 依 赖	输入依赖
含义	写之后读	读之后写	写之后写	读之后读
代码示例	① b＝a＋1; ② c＝b＋1;	① b＝a＋1; ② a＝c＋1;	① c＝a＋1; ② c＝b＋1;	① b＝a＋1; ② c＝a＋1;
说明	后续计算需使用前期计算结果，必须遵守计算顺序	属于伪依赖，可消除	属于伪依赖，可消除	可以并行

反依赖和输出依赖都属于伪依赖，可以通过增加变量来消除。对于表 5-1 所示的反依赖，也就是读后写操作，第 1 条语句读取变量 *a*，而第 2 条语句写入变量 *a*，如果这两条语句并行执行，一旦第 2 条语句先执行，就会导致第 1 条语句读取错误的变量值。要消除这种伪依赖关系，则可以增加一个变量，让第 2 条语句使用新的变量，而不是重用现有变量。修改后的代码如表 5-2 所示。输出依赖的消除也可以采用类似的方法，由于前后两条语句都写入同一个变量 *a*，在顺序执行的情况下，第 1 条语句的结果将被第 2 条语句覆盖，使第 1 条语句没有存在意义，而一旦并行执行这两条语句，就可能导致不确定的结果，为此，同样可以增加一个变量，让第 1 条语句写入新的变量，这样既消除了两条语句之间的依赖关系，又可以保留第 1 条语句的结果。

表 5-2　消除伪依赖的示例

比　较　项	消除反依赖	消除输出依赖
消除措施	语句②写入新增变量 *x*	语句①写入新增变量 *x*
修正后的 代码示例	① b＝a＋1; ② x＝c＋1;	① x＝a＋1; ② c＝b＋1;

与伪依赖不同的是，真依赖不能通过增加变量来消除。为保证程序正确性，无论怎样表示，变量的写操作都必须在读操作之前完成。

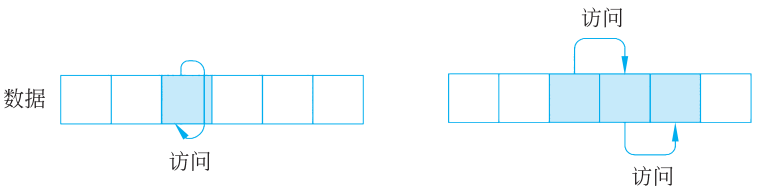
以上关于数据依赖关系的讨论使用了非常简单的语句。在实际程序应用中，依赖关系的表现可能更加复杂。例如，对同一变量的访问在前后两段代码之间，甚至两个软件模块调用之间产生了依赖关系。此时，开发者仍然可以使用上述原则和方法对程序进行优化，在保证正确性的前提下消除伪依赖，以实现并行。例如，在一个程序中，前后两段代码之间存在反依赖，也就是前一段代码读取某个变量，而后一段代码写入相同的变量，这时，如果用 OpenMP 的 Sections 构造将这两段代码分别定义成两个 section，那么这两个代码块的并行执行就可能导致错误的结果，而通过在后面的代码块中使用新增的变量、消除这种反依赖关系，就可以在保证程序正确性的前提下实现并行。

5.2.6　局部性

由于处理器和存储器之间的性能差距，在等待访存操作（如从主存中读取数据）时，处理器可能会停顿。存储器系统性能包括**带宽**和**延迟**两方面。存储器带宽即数据从存储器到 CPU 的传输速率（单位是 GB/s）；延迟即将数据项从存储器传送到 CPU 的等待时间，通常是数百个 CPU 周期。尽管摩尔定律保证了芯片性能的稳定增长，但存储器性能的增长速度仍然低于处理器，这将导致处理器和存储器的性能差距越来越大（即第 1 章中所介绍的“存储墙”）。在层次化存储体系中，cache 的性能与处理器较为接近，因此，充分利用 cache 可以大幅缓解存储墙对程序性能的不利影响。但由于 cache 容量较小，只有当程序的数据局部性较好时才能保证较高的 cache 命中率，进而提升性能。

局部性分为**时间局部性**（temporal locality）和**空间局部性**（spatial locality）。如图 5-4 所示，时间局部性表征数据访问在时间上的聚集特性，即程序当前访问的数据很可能会被再次访问，如“加 – 规约”中的累加和变量、循环中的循环变量等；而空间局部性表征数据访问在空间上的聚集特性，即程序未来很可能会访问当前数据的邻近数据，例如，程序

遍历一维或二维数组。对于时间局部性来说，当一块数据从主存被加载到 cache 后，如果程序在一段时间内频繁访问该块数据，则可以显著提升 cache 命中率；反之，如果一块数据被加载到 cache 后，其在相当长时间内不再被访问，那么其就很容易被 cache 替换策略换出，以后程序访问该块数据时，需要再次从主存加载到 cache，这显然是对性能不利的。对于空间局部性来说，当程序访问一个变量（如数组中的一个元素）时，该变量所在的整块数据将从主存被加载到 cache，如果程序后续访问邻近数据（如按顺序遍历数组），就会形成 cache 命中；而如果程序对数据的访问是跳跃式甚至是随机的，那么 cache 命中率将难以得到保证。



(a) 时间局部性　　(b) 空间局部性

图 5-4　局部性的分类

局部性较差的内存访问通常导致较多的主存访问开销，从而限制了并行化的性能。频繁访问主存还会使多个处理器（核）竞争互连网络或内存带宽，进一步降低性能。在并行计算中，局部性可以提高 cache 的命中率和有效内存带宽，因此，改善局部性是优化程序性能的重要手段。

5.3　并行程序的可扩展性及性能优化方法

5.3.1　什么是并行程序的可扩展性？

1. 并行程序可扩展性的概念

可扩展性（scalability）又被称为可伸缩性，并行程序的可扩展性是指程序的性能是否可以随着所用计算部件数量的增加而提升。这里的计算部件可以是处理器（核）、异构加速部件、计算节点等。

可扩展性是衡量并行程序设计好坏的重要指标。换句话说，一个好的并行程序设计应当具有良好的可扩展性。为什么可扩展性如此重要？主要是因为可扩展性影响了并行程序的适应性。宽泛地讲，并行程序需要适应不同数量的计算部件及各种问题规模（数据集）。

如何衡量一个并行程序的可扩展性呢？回想本书第 1 章介绍的加速比指标，对于相同的问题规模，通过测试程序在不同进程 / 线程数时的加速比可以画出加速比曲线，该曲线就可以比较直观地反映程序的可扩展性。

2. 强可扩展与弱可扩展

如果更深入地分析可扩展性，可以发现它包含两个方面：第一，对相同的问题规模（数据集），通过增加计算部件的数量，是否能够缩短程序的计算时间；第二，并行程序是否对各种问题规模（数据集）都能表现出良好的性能，以矩阵乘法为例，随着矩阵规模的增大，

人们往往希望通过使用更多的处理器或计算节点使计算时间保持稳定或在可接受范围内。

一个理想的并行程序应当同时满足上述两个方面的要求，但现实世界中，很多并行程序受限于应用模型和算法，只能满足一个方面的要求，这时，仍然可以认为该程序具有可扩展性，并使用强可扩展和弱可扩展加以区分。

强可扩展：所谓强可扩展（strongly scalable），就是对于相同的问题规模，通过增加进程 / 线程的数量（当然也意味着增加计算部件的数量）缩短程序的执行时间。如果用加速比指标衡量，强可扩展就意味着在问题规模不变的情况下，程序的加速比接近于线性。

以矩阵相乘为例，由于计算过程中进程 / 线程间几乎没有同步和通信，在矩阵足够大的情况下，通过增加进程 / 线程个数可以持续提升程序性能，因此可以说矩阵相乘是强可扩展的。

需要说明的是，强可扩展通常是有条件、有限度的。仍然以矩阵相乘为例，设想保持矩阵规模不变，进程 / 线程数量增加到一定程度后，每个进程 / 线程负责的计算量过小（也就是粒度过细），加速比就会逐渐走平甚至下降。这时，要保证程序性能可扩展，就需要增加矩阵规模，也就是增加问题规模，这就是下面所说的弱可扩展。

弱可扩展：所谓弱可扩展（weakly scalable），就是随着进程 / 线程数量的增加（当然也意味着增加计算部件的数量），通过增大问题规模使程序的执行时间保持相对稳定。

一般来说，强可扩展对并行程序的要求比弱可扩展更高。换句话说，设计强可扩展并行程序的难度更大一些。但这并不意味着弱可扩展就比强可扩展“差”，因为实际应用中的很多模型和算法本身就不具有强可扩展性。

回忆第 1 章介绍过的加速比概念，该指标常被用于衡量程序的并行可扩展性。理想的可扩展并行程序就意味着它可以获得线性加速比。然而，正如 5.2 节所介绍的影响并行性能的诸多因素那样，这些因素会限制程序的并行可扩展性。

有些计算可以很方便地被分成多个完全独立的计算任务，在计算过程中，这些任务之间没有交互，因而可以获得很好的并行可扩展性。有一个专门的术语用来描述这类计算：**易并行计算**（**embarrassingly parallel**）。一个典型的易并行计算例子是采用蛮力攻击方法破解密码，假设一种加密算法使用的密钥长度为 1024b，则密钥的状态空间为 2^{1024}，蛮力攻击就是对密钥状态空间进行穷举式搜索，以寻找正确的密钥。这显然是一个非常容易实现并行的计算：将密钥状态空间分成多个子空间，并将其分配给多个任务进行并行计算。由于各个子空间的搜索可以独立进行，并且划分子空间的数量也可以根据需要被灵活调整，因此，这个问题就属于典型的易并行计算。

5.3.2 确保并行程序可扩展性的重要原则：独立计算块

前文介绍了可扩展性的概念，那么如何设计出可扩展性好的并行程序呢？这是一个涉及多方面因素、需要综合考虑的问题。这里讨论其中一个重要的独立计算块原则。

从可扩展性考虑，**一个理想的并行程序应当是每个进程 / 线程负责足够大并且独立的计算任务，这被称为独立计算块原则。**这里所说的计算任务“足够大”就是指本章前文所讲的并行粒度，只有并行粒度足够大，才足以抵消创建进程 / 线程、分发数据等并行开销。而计算任务“独立”是指进程 / 线程之间的同步和通信尽可能少。要知道，同步和通信会导致进程 / 线程间相互等待，因而降低并行执行的效率，而且随着进程 / 线程数量的增加，

这种影响会更加严重。因此，同步 / 通信是制约可扩展性的重要因素，需要开发者在设计并行程序时给予足够重视。

1. 独立计算块示例一：并行统计数据

下面通过一个简单的例子展示独立计算块原则的应用。

假设一个并行程序需要对大量数据进行统计分析，并使用一个全局变量记录统计结果。并行程序创建多个线程，每个线程负责一块数据的统计，由于多个线程都需要写入结果变量，故需使用临界区进行互斥。图 5-5 给出了该程序的两种 OpenMP 伪代码实现，分别用代码 A 和代码 B 表示。代码 A 和代码 B 都使用一个 for 循环对数据进行统计，并在 for 循环处使用多线程进行工作共享。两种代码实现的区别是：代码 A 在循环内部直接更新结果变量 result，而代码 B 则增加了一个线程私有变量 result_p（注意 private 子句），在循环处理过程中，用该变量记录本线程的统计结果，并在循环结束后把该结果变量累计到全局结果变量 result。由于代码 A 在每次循环都有一个线程间互斥操作，所以这显然会严重影响并行效果，而代码 B 在循环过程中没有线程间交互，只在循环结束后进行一次互斥操作，因而可以显著提升并行效果。对于代码 B 来说，每个线程的循环过程都是独立计算块。

```
int  result;  //统计结果变量
int  length;  //待统计的数据个数
int  i;

#pragma omp parallel shared(…)
{
   #pragma omp for private( i )
   for  ( i = 0; i < length; i++ )
   {
       统计数据;
       #pragma omp critical
       更新 result;
    }
}
```

(a) 代码A

```
int  result;  //统计结果变量
int  length;  //待统计的数据个数
int  i;

#pragma   omp   parallel   shared(…)
private(result_p)
{
   初始化 result_p;
   #pragma omp for private( i )
   for  ( i = 0; i < length; i++ )
   {
       统计数据;
       更新 result_p;
    }
   #pragma omp critical
      根据 result_p 更新 result;
```

(b) 代码B

图 5-5　独立计算块的简单示例

上述例子非常简单，但其体现出的独立计算块原则适用于所有并行程序的设计。事实上，这一原则不应只体现在程序设计环节，还应在并行算法设计甚至应用模型设计上得到充分考虑。

2. 独立计算块示例二：规约（reduce）计算

下面再看一种体现独立计算块原则的例子：规约计算。

以最常见的“加 – 规约”为例，假设存在一个包含 n 个数的数组，要对数组的所有元素进行累加求和，这是一个典型的“加 – 规约”计算。该计算的最简单实现方式就是使用一个 for 循环对这 n 个数进行顺序累加，如图 5-6（a）所示。由于计算过程形成了一个树状操作结构，故这类计算又被称为树操作。

当规约计算的数据量很大，或者规约计算使用非常频繁时，就需要将其并行化。然而，在图 5-6（a）所示的顺序求和计算中，由于前后循环之间存在依赖关系，故开发者无法直