互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第一部分 Part 1

做好学习 OpenMP 的准备

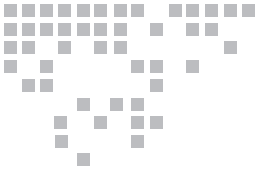
从计算的早期，乃至今天，同时做许多事情一直是提高性能的关键。从最初的 Cray 向量超级计算机中的流水线式执行单元、分布式内存工作站集群到单 CPU 中配置的多个核心，几十年来，并行性对于性能至关重要。

并行硬件只有在并行软件下才有用。如果能自动生成并行软件就好了。这已经尝试过很多次，除了极少数例外，其他都没有用，因此程序员需要编写并行软件。

这就造成了硬件和软件之间的“紧张”关系。新的硬件出现，程序员必须适应。这意味着编程语言和工具必须适应硬件。

在本书的第一部分，我们将描述并行程序员工作的世界。我们将为学习 OpenMP 做好准备，并将重点放在使用共享内存多处理器计算机这类重要的并行系统的程序员的需求上。这一讨论将引导我们了解这些机器所使用的编程模型以及 OpenMP 的历史渊源。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章



Chapter 1 第 1 章

并行计算

程序是在计算机上运行的指令流。计算机由包括内存、存储系统以及一个或多个处理单元在内的许多部分组成，而处理单元是计算机中实际进行“计算”和执行指令的部分。

顺序程序指一个程序在单个处理单元上运行，而并行程序同时在多个处理单元上运行多个指令流。这两种程序的基本思想很简单，但使用并行解决实际问题却不简单。

并行计算已经发展为计算机科学中一个独特的分支，有自己的专业术语、概念和硬件，当然还有自己的编程语言。它是在高性能计算（HPC）社区内发展起来的，如今已成为每一个 HPC 程序员所需掌握的核心知识的一部分。然而，并行计算中的许多理念对于 HPC 之外的广大程序员群体来说是陌生的。

我们希望每个人而不仅仅是 HPC 程序员能阅读本书并从中受益。因此，在本章中，我们讲解并行计算的语言和基础概念。对于有经验的 HPC 程序员来说，可以快速浏览本章以验证自己是否和我们使用了一样的并行计算专业术语。对于 HPC 新手或来自 HPC 社区之外的人，要仔细阅读本章，因为在本章中我们建立了学习 OpenMP 所需的基础概念。

1.1 并行计算的基本概念

如果你写过程序，可能对计算机的基本结构很熟悉。保存着程序数据和文本的地址空间，我们称其为计算机的内存。术语“数据”指的是一组变量，它命名内存中的地址，并引用其中存储的值。计算机的控制单元从存储在内存中的程序加载指令，并执行这个单指令流中的操作来产生结果。这种简单的顺序计算机概念后来被称为冯·诺依曼体系结构，

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

其可以追溯到 1945 年约翰·冯·诺依曼撰写的关于早期计算机设计的报告[14]。

所有的程序员都要学习如何编写适用于顺序计算机的程序。在最初，这就足够了。位于计算机核心的中央处理单元（CPU）能够提供市场所需的不断提高的性能增益。CPU 通过变得越来越复杂以解决限制性能的不同瓶颈来实现这一点。

让我们更详细地考虑 CPU。它由一个执行算术和逻辑运算的算术逻辑单元（ALU）以及一个管理数据和指令流的控制单元组成。现代 CPU 通过将指令分解为更小的微操作，并将其送入处理流水线从而在指令层面达到并行。由于控制单元跟踪微操作之间的依赖关系，因此它们可以并行执行，甚至可以不按顺序执行，却依旧能生成与原始顺序指令流相同的结果。这种并行形式的结果被称为超标量执行。幸运的是，超标量执行是由 CPU 代表程序员对指令进行管理，而程序员仍以单一的、顺序的指令流来思考。随着面向大众市场的商用成品 CPU 的兴起，在摩尔定律经济趋势的引领下，晶体管密度每两年翻一番，其性能也随之提升。除了最激进的超级计算应用外，几乎没有理由担心会超出顺序计算模型的能力。

硬件趋势的残酷现实

摩尔定律指出，每隔两年左右，半导体器件（或称“芯片”）上的晶体管数量将会翻倍。这是对经济趋势的预测，而不是物理定律。摩尔定律可追溯到 1965 年，直到 2004 年，芯片的处理速度不断提高。蚀刻在芯片上的工艺制程变得越来越小，切换晶体管状态所需的能量（动能）下降了。这被称为丹纳德微缩定律[3]，其指出芯片的制程变小时，电压会降低，芯片能以更高的频率驱动。

2004 年左右，摩尔定律不再提供更高的时钟速度。开关晶体管所需的能耗持续降低，漏电和其他静态能耗的需求并没有减少。最终，它们主导了驱动芯片所需的能耗，丹纳德微缩定律终结。尽管摩尔定律在继续缩小晶体管的尺寸，然而，随着丹纳德微缩定律的结束，增加晶体管数量的好处只能来自体系结构创新。这意味着更多的内核、更宽的向量单元、特殊用途的加速器等。

从硬件的角度来看，这是很好的。在后丹纳德微缩时代，硬件工程师乐趣更多，受苦的是软件界。硬件工程师把越来越复杂的东西扔给软件开发者，关心性能的程序员别无选择，他们必须面对这个残酷的硬件现实，并解决如何为这些并行、异构设备编写程序的问题。

这一切在 2004 年前后发生了变化。正如方框中所解释的那样，关心性能的软件开发人员别无选择，不得不编写并行代码。并行编程不是可有可无的，而应该是每个软件专业人员技能的一部分。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1.2 并发性的兴起

要理解并行计算，我们必须从与之密切相关的概念“并发性”（concurrency）开始。如果来自任何一个流的单个指令与来自其他流的指令相比是无序的，则这两个或多个指令流就被称为是并发的[7]。

这一点最好用一个简单的例子来解释。使用你喜欢的编辑器，输入图 1-1 中的代码。不要担心这段代码中 pragma 的含义，我们将在后面讲解它。

```
1  #include <stdio.h>
2  #include <omp.h>      // The OpenMP include file
3
4  int main()
5  {
6     #pragma omp parallel
7     {
8        printf(" Hello ");
9        printf(" World \n");
10    }
11 }
```

图 1-1 一个简单的“Hello World”C 程序，演示并发执行

使用支持 OpenMP 的编译器（如 GCC）编译这段代码。为了使编译器能够识别 OpenMP 指令，必须设置一个（编译器依赖的）标志。例如，使用 GCC 时，必须使用 -fopenmp 选项告诉编译器用 OpenMP 创建一个多线程程序。然后像运行其他可执行文件一样运行这个程序。

```
$ gcc -fopenmp hello.c
$ ./a.out
```

系统会给出默认的线程数，如果在现在典型的笔记本电脑上运行，将会有 4 个左右的线程（核的数量可以通过操作系统看到）。以串行编程的思路，你可能会期望程序的输出为：

```
Hello World
Hello World
Hello World
Hello World
```

但是，这四个线程是并发执行的。每个线程执行的指令相对于其他线程是无序的。每个线程中的 printf 语句遵循程序定义的顺序，但在不同线程之间，它们没有指定的顺序。所以输出的结果可能是这样的：

```
Hello Hello World
World
Hello  World
Hello  World
```

此外，每次运行程序输出的结果可能不一样。由于线程是并发的，所以每次操作系统调度线程执行时，输出操作的顺序可能会改变。另一种思考方式是，每一种合法的交错语

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

句方式为程序定义了一种可能的执行顺序。作为一个并行程序员，你的挑战是确保所有可能的交错都能产生正确的结果。

现在已经运行了第一个多线程程序，看到的是可用的线程都并发执行。让我们引入对并行性或“并行执行”概念的讨论。如果一组并发线程在不同的处理单元上执行，它们是同步推进的，这被称为并行执行。并发性定义了操作可以以任何顺序执行（即它们是无序的）。并行性使用多个硬件元件以使操作在同一时间运行。请注意，并发和并行具有不同的含义。尽管偶尔你可能会注意到这些术语被混淆，代表相同的意思，但这是不正确的。通过硬件的并行执行，允许同时执行并发任务。

1.3　并行硬件

随着丹纳德微缩定律的终结，计算机设计的重点从强调不断提高的时钟速度转移到利用并行性的巧妙架构特性上。这导致了各类系统的出现：分布式内存集群、可编程 GPU、从单一指令流驱动多个数据元素的向量单元（SIMD 或“单指令多数据”）和多处理器计算机。OpenMP 对除分布式内存集群外的上述系统都有效。

1.3.1　多处理器系统

OpenMP 从关注多处理器系统开始。对于大多数 OpenMP 程序员来说，这些系统仍然是他们编写 OpenMP 程序时最关心的问题。一个多处理器计算机由多个处理器组成，它们可能共享同一个地址空间。处理器可用的内存是共享的，因此这就是它们通常被称为共享内存计算机的原因。为了理解多处理器系统，我们使用一个基本模型突出系统的核心元件，同时隐藏了被认为不那么重要的细节。对于多处理器系统，我们首先使用的模型是对称多处理器（SMP）模型，如图 1-2 所示。

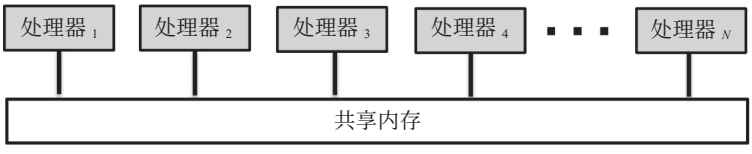


图 1-2　一个由 *N* 个处理器组成的对称多处理器计算机，共享一个内存区域

一个 SMP 有 *N* 个处理器，共享一个内存。硬件由操作系统（OS）管理，操作系统对所有处理器一视同仁。此外，对于任何处理器来说，访问内存中任何变量的成本都是一样的。换句话说，从操作系统和内存的角度来看，这个模型中的处理器是“对称”的。

SMP 模型被过度简化了。你不可能遇到一个看起来像 SMP 系统的计算机。让我们更

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

详细地考虑现代处理器。CPU 是一个通用处理器，位于计算机的单个插槽中。它被优化为快速地提供单个事件的结果，即 CPU 被优化为低延迟。现代 CPU 是一种多处理器计算机，多个被称为核的不同的处理器放在一个硅片上封装成一个多核 CPU。图 1-3 中展示的是多核 CPU 的示意图。

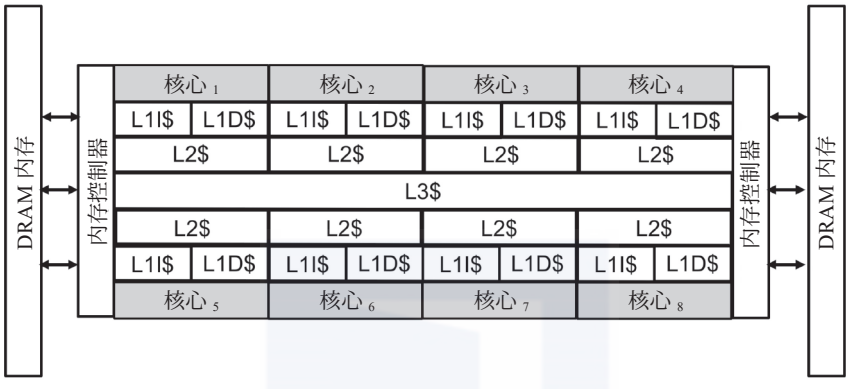


图 1-3 一个典型的有 8 个核的多核 CPU，每个核都有用于数据（L1D$）和指令（L1I$）的一级缓存、统一的二级缓存（L2$）和共享的三级缓存（L3$）。该 CPU 有两个内存控制器，每个控制器有三个通道，用于访问片外内存（DRAM）

多核系统的内存远比 SMP 系统的要复杂得多。所有的处理器共享可当作单一地址空间的内存。DRAM（动态随机存取存储器）为计算机中内存的具体实现，其比芯片内的核慢得多。因此，现代 CPU 包括紧密集成在核上的高速内存小区域，称为缓存。

缓存不是作为一个独立的地址空间来访问的。缓存中的存储空间被映射到 DRAM 中的地址上。这种映射的细节远超出了本书的范围，对于我们来说，把高速缓存看作进入 DRAM 中更大内存的一个高速窗口就可以了。缓存和 DRAM 之间的映射是以被称为缓存行的小分块形式进行的。一个典型的高速缓存行只有 64 个字节（即足够容纳 8 个双精度浮点数的空间）。将缓存组织成缓存行的原因是，在许多程序中，如果一条指令需要一个地址 N 处的值，下一条指令很可能将访问地址 N+1 处的值。利用这种空间局部性可以大大提升性能。

在现代系统中，内存是分级的。考虑一个典型的多核 CPU，如图 1-3 所示。每个处理器旁边都有一个直接相邻的小的缓存，分别用来存放程序的数据和指令。这些被称为一级缓存，包含 L1D$（数据）和 L1I$（指令）缓存，它们离核最近。每个核心都有一个二级缓存（L2$）存放数据和程序指令，因此这被称为统一缓存。最后，CPU 中的所有核可以共享一个额外的缓存，即三级缓存或 L3$。缓存的体积很小。对于一个典型的高端 CPU 来说，这些高速缓存的大小[⊖]是：

⊖ 一个字节是 8 个二进制位或比特。2^{10}B 是一个 KB 或 1024 个比特。一兆字节是 1024KB 或 1048576 个比特。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 每核 32KB 的 L1 级数据缓存
- 每核 32KB 的 L1 级指令缓存
- 统一的（保存数据和指令的）每核 256KB 的 L2 级高速缓存
- 核心之间共享 MB 量级的 L3 级高速缓存

在程序执行过程中，缓存行在内存层级结构中不断移动。多个核在任何时间可能访问任何一个缓存行，这就存在内存访问冲突的隐患。内置于 CPU 中的高速缓存一致性协议管理所有这些高速缓存行，以确保最终所有核都能在内存中看到相同的值。这里的关键词是“最终”，即在任何给定的时刻，核有可能看到任何给定地址的不同值。访问内存中的共享地址时，用于管控不同核可能看到的值的一组规则称为内存模型。我们将在本书后面的部分详细讨论内存模型。

SMP 模型的一个关键方面是，每个处理器对于访问内存中的任何地址都有相同的成本。即使粗略地看一眼图 1-3 中的多核芯片也会发现情况并非如此。访问内存中一个值所需的时间取决于该值在内存层级结构中的位置。考虑从内存中访问一个值所需的时间，即内存访问延迟。使用典型的高端 CPU 的各级延迟值，这些值在整个内存层级结构中的范围如下：

- L1 缓存延迟 = 4 个周期
- L2 缓存延迟 = 12 个周期
- L3 缓存延迟 = 42 个周期
- DRAM 访问延迟 = 约 250 个周期

因此，与其说内存系统具有均匀的内存访问时间，不如说系统具有非均匀的内存架构（NUMA），即现代多处理器系统不是 SMP，而是 NUMA 系统。它甚至比图 1-3 中多核 CPU 所展示的还要糟糕。高端服务器，尤其是 HPC 系统中使用的服务器，经常将多个 CPU 连接成一个大型的 NUMA 集群。例如，我们在图 1-4 中展示了一个用于 HPC 系统的通过高速点对点互连的 4 个 CPU 的服务器典型框图。集群中的每个 CPU 都有自己的内存控制器和 DRAM 块。4-CPU 集群中的所有内存都被组织在单一（共享）的地址空间中，并且系统中的任何核都可以访问。在此不赘述详细的时序，不难看出，当核访问映射到自己芯片的 DRAM 的缓存行，或访问与系统中其他一些芯片相关联的 DRAM 存储体的缓存行时，它们的访问内存成本会有很大的差异。

并行硬件可能非常混乱。幸运的是，随着可以根据需要移动高速缓存行的高速缓存一致性协议以及现代优化编译器的出现，可以在编写代码时摆脱 SMP 模型的束缚。作为以后的优化步骤，可以利用个人系统的 NUMA 特性修改程序。这包括直接的优化，如重新组织循环以提高缓存行数据的重用（缓存阻塞），以及更复杂的优化，如在以后可能会处理该数据的同一核上初始化数据。我们将在本书后面的部分讨论这些和其他相关的优化。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

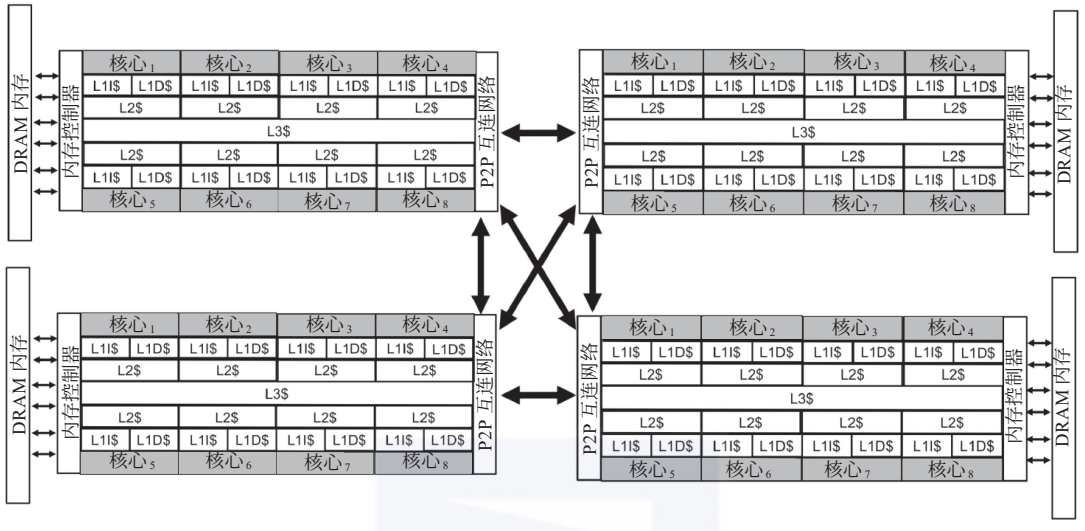


图 1-4 4 个 CPU 通过点对点（P2P）连接的 NUMA 系统。所有的 DRAM 都可以被所有的核访问，这意味着访问不同内存区域的成本在整个系统中差别很大

1.3.2 图形处理单元

计算机图形学的核心问题是为一个场景选用合适的模型，并将其转化为像素呈现给（通常是人类）视觉处理系统。利用计算机科学、物理学、数学和心理学的组合，所产生的图像将向观众传达信息。图形学中的处理方式一般为数据并行的：一个场景被分解成几何图形，这些图形又被进一步转化为分片的集合。这些分片流经处理流水线后被渲染成一组像素显示在屏幕上。我们大大简化了这个过程，但总的处理流水线由以下几个阶段组成：

- 遍历每一个 3D 对象，用多边形平铺可见表面。
- 遍历每一个多边形，分解成三角形。
- 遍历每一个三角形，计算遍历组成三角形的每一个像素点的颜色。

这里的关键思想是遍历每一个（foreach），后面跟着某种集合的名称（例如，多边形的集合或三角形的集合）。该函数（例如，分解成三角形）被应用于集合（例如，多边形集合）的每个成员。集合中每个成员的计算是独立的并且可以同时进行，即并行是在数据中进行的（因此称为数据级并行）。

支持某种版本的图形流水线的专用硬件可以追溯到 20 世纪 70 年代。随着时间的推移，对一组对象的 foreach 成员进行的处理变得更加复杂。相较于在硬件中固定该功能，能够对这些单元进行编程变得更加重要。渐渐地，用于图形的硬件不再是“硬件专用”，而是变得越来越可编程。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

2006 年，市场上出现了完全可编程的 GPU。涉及 GPGPU 或通用 GPU 编程的 GPU 厂商提供了应用程序接口（API）。GPGPU 的发展速度很快。在 GPGPU 设备首次出现在市场上两年后，一种名为 OpenCL[11] 的 GPGPU 编程标准编程语言发布。这些基于标准的解决方案（OpenCL 和 OpenMP）和专有解决方案已经得到了快速的发展，使得 GPGPU 编程成为性能导向型程序员的主流技能。

显然，详细讨论 GPU 以及如何对其进行编程已经远超出了一本关于学习 OpenMP 的书的范围。然而，我们想描述一下 GPGPU 编程的关键特性，帮你全面了解并行编程的基础知识。然后在后面的 12.3 节中，我们将重新讨论这个话题，并讨论如何使用 OpenMP 进行 GPGPU 编程。

区分 GPGPU 与 CPU 编程的最根本问题是吞吐量与延迟。当一个帧流显示在显示器上时，你并不关心更新一个像素点需要多长时间。你关心的只是整个帧的生成速度，即它们的生成速度是否足够快到使视频看起来流畅。换句话说，你关心的是在屏幕上移动的帧的吞吐量。与提交给电脑的命令的响应相比，如果在屏幕上的某一点上点击鼠标，你希望得到立即的响应。你关心的是单个计算的延迟。

大量的电路被使用以维持通用处理器的低延迟要求。例如，CPU 包括高速缓存层级结构，专门用于将内存放置在与之共同工作的处理元件附近。GPU 针对吞吐量进行优化。晶体管被设计为将许多处理元件集成到一个芯片上。该芯片的内存系统支持高吞吐量，但它缺乏类似 CPU 用以支持低延迟的复杂多级缓存。

吞吐量优化的 GPU 编程模型的基本思想是将 foreach 语句的索引范围变成索引空间。foreach 语句的主体被转化为称为核的函数。核的实例在索引空间（一个 NDRange 或网格）中的每一个点运行。我们称这个核实例为工作项（work-item）。数据被对齐到同一个 NDRange 上，所以我们将数据放置在靠近工作项的地方，处理元件将对其进行操作。

为了使 GPU 成为一个专门的吞吐量引擎，GPU 内置的调度器知道核函数所需要的数据。工作项被分组到工作组（work-groups）中，工作组在它的数据可用之前等待执行。如果工作组比处理元件多得多，它们会排队等待。在等待数据流过系统时，有很多工作可以让处理元件保持忙碌状态。这是延迟隐藏的一个例子，即将内存移动与计算重叠，这样就不会产生访问单个内存块的高成本。

当与针对高带宽进行优化的内存系统相结合时，GPU 可以为数据并行工作负载提供高吞吐量。它们并不能处理所有的工作负载。如果工作项需要交互，编程就会变得更加复杂，在某些时候，基于用工作组的深度队列来隐藏内存延迟的方法会崩溃。通过大量的创新，各种各样的算法被映射到 GPGPU 数据并行计算模型上。正如我们将在 12.3 节中所看到的，你可以在 OpenMP 中实现所有这些。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

1.3.3 分布式内存集群

在科学计算中，给定一个大小为 N 的问题，在该科学领域工作的人很可能想运行一个大小为 $10\times N$ 或更大的问题。在科学探究的顶端，人们似乎对更多更大的计算量有着无法满足的需求。这意味着单个处理器——无论是 CPU 还是 GPU——始终无法跟上需求。

将多个处理器集成到单一存储器域中是可能的。例如，图 1-4 中的 NUMA 系统就做到了这一点。创建一个 NUMA 系统需要专门的硬件工程，而且非常昂贵。随着对计算需求的增多，NUMA 系统也将难以应对。

解决方案是限制 NUMA 系统的不断推进。取而代之的是，把标准化的现成服务器联网成一个大型系统，建造成大规模数据中心。然后用软件将它们捆绑在一起，成为大型并行计算机。像这样用现成的组件构建的计算机被称作集群（也称机群，cluster）。只要能支付运行集群的电费，并有一个足够大的建筑来容纳它们，集群就可以达到想要的规模并支持真正不可思议的计算速率。

集群作为一种并行计算机，与我们迄今为止所考虑到的共享内存系统有很大的不同。集群中的服务器位于集群网络中的节点上。当节点专门用于密集计算时，也许是通过将一个 GPU 与一个高端 CPU 配对，或者将多个 CPU 组合成一个 NUMA 域，它们有时被称为计算节点，以区别于为更典型的数据中心操作而优化的节点（有时称为服务器节点）。节点间不共享物理内存，因此内存分布在系统周围。节点之间通过相互传递消息进行交互。关键的思想是“双向”通信，即一个节点向另一个节点发送命令并接收返回的消息。用于在分布式内存机器（如集群）上编程的标准 API 称为消息传递接口（MPI）。更高级的分布式内存机器的编程系统是基于单向通信、分区全局地址空间（PGAS）和 map-reduce 框架的。然而，对于 HPC 来说，无处不在的共同点是 MPI。

我们在本书中不会涉及 MPI。如果你在 HPC 领域工作，你将需要在某些时候学习 MPI。HPC 中的主流模型是节点间的 MPI 和节点内的 OpenMP，我们称之为 MPI/OpenMP 混合模型。即使使用高级 PGAS 语言或一些较新的任务驱动编程模型，它们背后的原理也是 MPI，即使不是 OpenMP，也是类似 OpenMP 的多线程模型。

1.4 多处理器计算机的并行软件

多处理器计算机已经有 50 多年的历史了。我们已经知道了如何组织软件来管理这些系统，这极大地简化了程序员的生活，因为关于如何组织这些系统的软件的共同观点可能适用于任何系统。

操作系统代表系统的用户（包括程序员）管理硬件。当启动一个程序时，操作系统会创

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

建一个进程。与这个进程相关联的是一块内存区域和对系统资源的访问权（例如，文件系统）。一个进程会分叉为一个或多个线程，这些线程都是该进程的一部分。每个线程都有自己的内存块，但所有线程都共享进程的内存和进程可用的系统资源。这些线程代表该进程执行程序的指令。

操作系统对线程的执行进行调度。在一个系统中，线程比处理器多得多。基本的思想是让操作系统将并发线程交换执行。这样，如果一个线程在等待一些高延迟事件（如文件访问）时被阻塞，其他准备执行的线程就可以交换进来使用可用的处理器，目标是有效地利用多处理器计算机上的处理器，使没有一个处理器在相当长的时间内处于空闲状态。

API 是程序员在编写软件时可以使用的函数、数据类型和任何其他构件的接口，其目的是将系统中复杂的和潜在动态的功能隐藏在一个固定的 API 后面。操作系统提供了一个低级的 API 来管理线程。大多数操作系统，包括基于 Linux 的操作系统（其中包括苹果的 OSX），都支持名为 pthreads 的 IEEE POSIX 线程模型。与任何低级 API 一样，直接使用 pthreads 可以在处理线程时获得更低的开销和最大的灵活性。然而，直接使用 POSIX pthreads API 编写代码是很烦琐的，容易出错，而且不是大多数应用程序员愿意做的事情。例如，在图 1-5 中，展示了在图 1-1 中讨论的“ Hello World ”程序的 pthreads 版本。我们的目的不是教你如何使用 pthreads 写代码，而只是想突出 pthreads 编程的一些高层特性，并强调大多数程序员并不想这样写代码。

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define NUM_THREADS 4
5
6  void *PrintHelloWorld(void *InputArg)
7  {
8      printf(" Hello ");
9      printf(" World \n");
10 }
11
12 int main()
13 {
14     pthread_t threads[NUM_THREADS];
15     int id;
16     pthread_attr_t attr;
17     pthread_attr_init(&attr);
18     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
19
20     for (id = 0; id < NUM_THREADS; id++) {
21         pthread_create(&threads[id], &attr, PrintHelloWorld, NULL);
22     }
23
24     for (id = 0; id < NUM_THREADS; id++){
25         pthread_join(threads[id], NULL);
26     }
27
28     pthread_attr_destroy(&attr);
29     pthread_exit(NULL);
30 }
```

图 1-5 一个使用 pthreads 的简单的“ Hello World ”C 程序

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

pthreads 编程的第一步是将在线程中执行的代码以函数的形式隔离出来。在本例中，这个函数是 `Print_HelloWorld(void *)`。在主函数内部，设置了一个数组来存放线程 ID，并设置了一个 pthreads 不透明的对象来存放线程的属性。然后，通过一个循环以创建我们想要的线程数量，并创建（即“ fork ”）线程，传递一个指向每个线程应该运行的函数的指针。之后，程序等待每个线程完成，此时线程退出。实质上，执行与程序的主流程相接，一旦所有线程完成了接入，主程序就会清理 pthreads 对象并终止。

这是一个最简单的 pthreads 编程的例子。当把参数传递给线程，用额外的属性控制线程，并在线程之间有序地进行内存操作时，代码就变得复杂多了。当需要完全控制线程以最大限度地提高性能，或者管理线程与系统交互的所有方式时，那么或许只能在 pthreads 的层次上进行编码。然而，大多数程序员不能容忍在如此低的层次上进行编程，他们需要更高层次的抽象。

这也是引入 OpenMP 标准的一个主要原因。

对构建应用程序而不是操作系统或低级服务感兴趣的程序员来说，编码是基于高级模型进行的。关键是要选择一个很适合目标硬件的模型。请记住，硬件每隔几年就会更换一次，而一个好的应用程序会使用几十年。因此，在选择高级模型时，重要的是选择一个不与单一厂商的硬件挂钩的模型。一个所有相关厂商都支持的标准编程模型是必不可少的。例如，MPI、pthreads 和 OpenMP 可以从多个来源获得（包括开源实现），并能在所有主流平台上运行。

编程模型也必须与应用程序中的算法或基本设计模式相匹配。例如，OpenMP 能够很好地匹配围绕嵌套循环结构组织的程序和利用共享内存的任务级程序。而对于分布式内存架构和具有硬性实时约束的应用来说，它不是一个好的选择。

最后，编程模型是由规范定义的，但不能用规范来编译一个程序。需要一个编程环境，或者在 OpenMP 的情况下，需要支持该规范的编译器。对于编程环境来说，要花很多年的时间才能在不同的架构上完全支持一个新版本的编程模型规范，这是很常见的，也是令人沮丧的。由于有关键厂商的积极参与，OpenMP 在跟进新版本的标准方面做得很好。大多数情况下，在 OpenMP 规范的新版本发布后的一年内，就可以实现完全符合标准。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 2 章 Chapter 2

性能语言

编写并行程序的原因只有两个：用较少的时间解决一个固定大小的问题，或者用合理的时间解决一个较大的问题。无论上述哪种情况都是为了提高性能。OpenMP 是一种用于编写并行程序的编程语言。在某种层面上，它总是要回到性能上。

性能是一个如此简单的词汇。然而，这个词隐藏着多层复杂性，并根据使用环境的不同而具有不同的含义。性能的原始评判标准是以时间为基础的，但即使是“时间”这个看似毫不含糊的概念也有细微的差别，如“ CPU 时间”（CPU 频率乘以 CPU 在执行程序时消耗的周期数）与“墙钟时间”（由计算机外部时钟测量的时间，即“墙上”的时钟）。性能作为一个独立的数字很少有意义，我们通常将性能作为一种比较性的衡量标准来关注以突出性能趋势。

这种复杂性导致了丰富的性能语言。并行程序员谈论的是加速比、效率、可扩展性、并行开销、负载均衡以及一系列用于推理性能的概念。本章的目标是解释这些术语。

2.1 基础：FLOPS、加速比和并行效率

我们写一个并行程序来减少“求解时间”时，关心的是我们经历的时间，即墙钟时间。当我们把一个程序转换为并行程序时，墙钟时间应该减少。随着处理器的增加，墙钟时间应该继续减少直到硬件或算法所提供的潜在并行性耗尽。

我们可以用展示性能不同方面的方式来呈现计时数据。例如，如果用速率来表示性能，即墙钟走一秒所执行的计算次数，就可以立即将结果与计算机的峰值性能或通过算法得出的希望的理论性能值进行比较。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

高性能计算（HPC）在很大程度上是以浮点数的运算为中心的。因此，我们经常以墙钟时间一秒钟内可以完成的浮点运算来考虑性能。这种衡量的单位叫作 FLOPS ：每秒执行的浮点运算。可在 FLOPS 上加上相应的前缀，如 Mega（10^6）、Giga（10^9）、Tera（10^{12}）、Peta（10^{15}）等，使单位的使用更方便。除非另有说明，否则在谈论 FLOPS 时，默认的浮点类型是双精度。在大多数计算机系统上，双精度是一个 64 位的量。虽然 HPC 关注的是 FLOPS，但当从 HPC 常见的数值仿真进入更广泛的计算领域时，关注点有时会转移到每秒执行的操作（OPS）或每秒执行的指令（IPS），但大多数 OpenMP 程序员都是关注 FLOPS 的。

让我们来看看 FLOPS 的范围。在 20 世纪 80 年代末，当时最快的超级计算机中的一个处理单元（如 Cray 2）运行时的峰值性能为 500 MegaFLOPS（或 500 MFLOPS）。今天常见的 iPhone 运行 Linpack 1000 ㊀测试程序（一种用于跟踪 HPC 系统性能的基准）时，其性能超过 1200 MegaFLOPS。美国国家能源研究科学计算中心（NERSC）的 Cray XC40 超级计算机 Cori 系统在 2018 年以 14 PetaFLOPS 的速度运行 MPLinpack 基准，核心数量为 622 336 个（https://www.top500.org/system/178924）。到 21 世纪 20 年代早期，世界上最快的超级计算机应该会跨越 ExaFLOP 障碍（10^{18} FLOPS）。

用 FLOPS 表示的原始性能很有趣，但对并行程序员来说，这还不够。我们想知道程序对并行系统资源的利用情况。我们想衡量程序在增加处理器时运行速度有多快。我们想知道程序的加速比情况。

加速比是一个程序的运行时间的比率。理想情况下，在一个处理器上用可获得的最好的串行算法运行一个程序，然后用并行软件在 P 个处理器上再次运行它。如果我们将 T_s 定义为串行程序的运行时间，把 T_P 定义为在 P 个处理器上的并行程序的运行时间，那么加速比的定义为：

$$S(P) = \frac{T_s}{T_P}$$

并不总是能够运行优化的串行程序来测量 T_s。在许多情况下，我们实现了一个并行算法，却无法实现相应的串行算法。在这些情况下，可以用运行在一个处理器上的并行程序的时间 T_1 来代替 T_s。在其他情况下，串行代码可能不适合放在单个处理器的内存中，因此我们可以只与运行在最小数量处理器上的并行程序进行比较。由于加速比的定义存在这种潜在的歧义，因此在报告加速比时，定义要比较的参考“串行执行”很重要。

㊀ Linpack 基准求解了一个稠密系统的线性方程组：著名的 $Ax = b$ 问题。我们曾经为固定阶数的矩阵（如 Linpack 1000）运行该基准，但现在人们用与自己的机器匹配的最大的问题规模运行该基准，即 MPLinpack 基准。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在理想的情况下，加速比等于处理器的数量。如果把处理器的数量增加一倍，性能就应该增加一倍。当一个程序遵循这种加速比趋势时，我们称它具有“完美线性加速比”。我们用一个被称为并行效率的指标，来衡量可缩放的加速比与完美线性加速比的接近程度。

$$\text{eff} = \frac{S(P)}{P}$$

我们在图 2-1 中展示了加速比和效率与处理器数量（在本例中，HPC 集群单个节点中的核）的函数关系。这个程序（octo-tiger：一个使用快速多极子算法进行引力场模拟的程序）使用了一个名为 HPX 的任务驱动系统 [5]，并显示出了极好的加速。请注意，对于处理器数量较多（如图 2-1），我们经常对水平轴和垂直轴使用对数标尺。

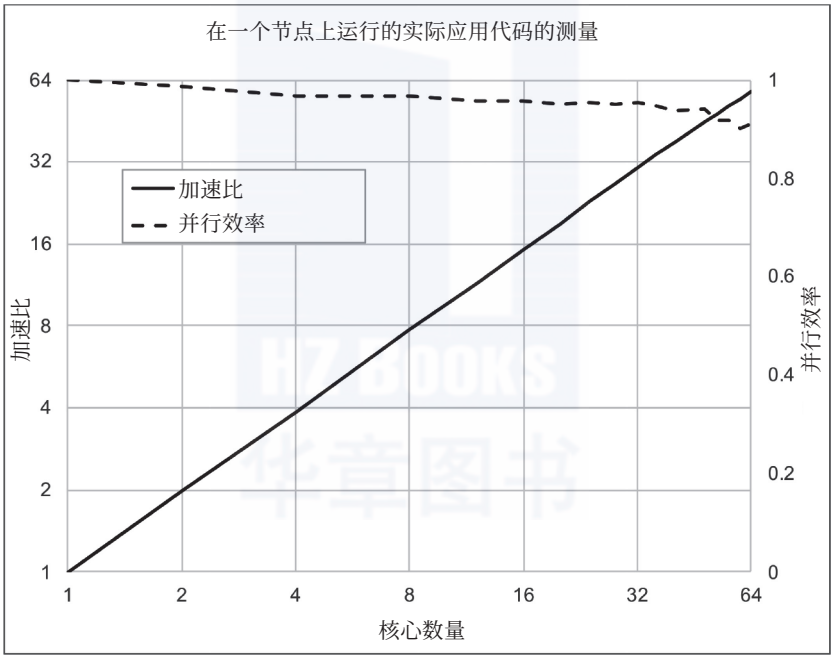


图 2-1　加速比和效率与核心数量的函数关系。对于集群中单个节点的性能测量，*X* 轴和 *Y* 轴使用以 2 为底的对数，相应的并行效率使用对数线性标尺。相关的应用是用 HPX 任务驱动编程模型并行化的快速多极代码 octo-tiger。节点为运行频率为 1.4GHz 的 Intel Xeon Phi7250 68C 处理器

加速比对于理解并行算法以及它如何随着处理器数量的增加而扩展是很重要的。但千万不要忽视，最终的目标是减少处理问题的时间。高性能计算中最古老的“骗术”之一就是选择一个慢的算法，而这个算法恰好有很低的串行比例。它会显示出极好的加速比，

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

但如果开始使用的算法很差，那么巨大的加速比其实是在误导你，掩盖了解决方案本质上的低质量。因此，请测量加速比，并了解算法的扩展性。然而，始终要花时间确保你使用的是一个好的（即使不是最优的，也应是快速的）算法，以减少计算的整体运行时间。

2.2 阿姆达尔定律

正如我们所看到的，并行计算的一个核心问题是了解性能如何随着处理器数量的增加而提升。如果我们增加处理元件的数量，性能是否会无限制地提高？这个问题由 Gene Amdahl 在 1967 年的一篇论文 [1] 中进行了论述，那时正是并行计算机刚出现的时候。他的目标是限制人们对通过组合许多较小的处理器来构建大型计算机这一想法的热情。这导致了一个一般原则，我们称之为阿姆达尔定律。

为了推导出阿姆达尔定律，我们从一个基本的简化开始。假设一个程序有一个占比为 α 的工作，这部分基本上是串行的。对于程序的这一部分，增加处理元件时，它不会运行得更快。我们称 α 为串行比例。如果 α 是一个问题的串行比例，那么 1–α 就是并行比例。令串行代码的运行时间为 T_s，则使用 P 个处理器的并行代码的运行时间为 T_p。在这个模型中，并行运行时间为：

$$T_p = \alpha \times T_s + (1-\alpha) \times \frac{T_s}{P}$$

执行可并行部分代码的时间减少为原来的 1/P 倍。加速比可由以下式子给出：

$$\text{Speedup} = \frac{T_s}{T_p} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

当 P 趋近无穷大时，并行项接近零，阿姆达尔定律就剩下如下的式子：

$$\text{Speedup} = \frac{1}{\alpha}$$

例如，如果算法可以为程序提速 95%，那么串行比例为 0.05，在处理器数量不受限制的情况下，能达到的最佳加速比为 20。

把加速比的变化看作 P 的函数。在图 2-2 中，我们展示了不同并行比例的最大可能的加速比。根据阿姆达尔定律，如果只有 90% 的代码可以并行化，那么串行比例就是 10%，无论使用多少处理器，都不可能做到比 10 更好的加速比。代码的串行比例的影响和它不能从额外的处理器中获益的事实在我们达到 10 的极限之前就已经降低了加速比。作为一个好的经验法则，良好的扩展性要求串行比例比阿姆达尔定律所建议的极限小一个数量级。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

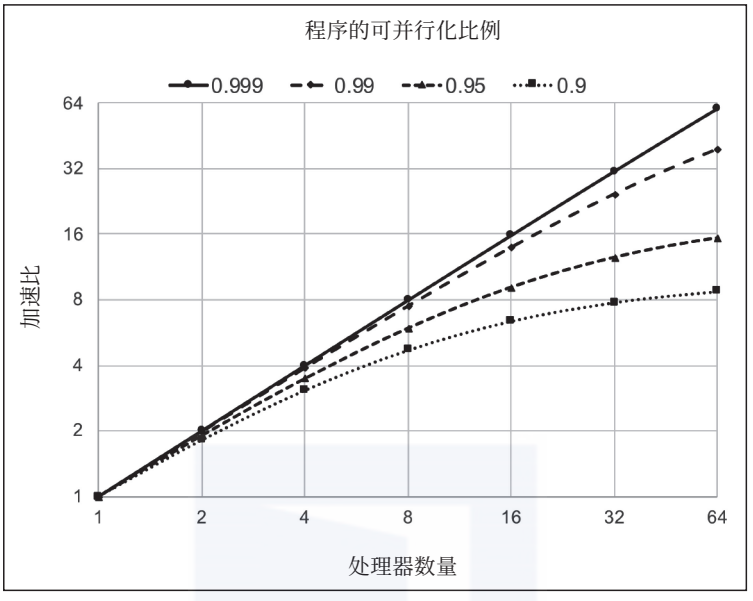


图 2-2　对于程序的可并行化比例的不同值，加速比与处理器数量的函数关系。我们绘制了加速比的对数与处理器数量的对数之间的曲线，请注意，当程序可并行化比例从 0.999 降到 0.9 时，加速比下降得很快

2.3　并行开销

由于算法的限制，阿姆达尔定律限制了并行的收益。如果算法中的某些部分不能利用多个处理器，那么额外的处理器将无济于事。这是并行程序员必须努力解决的一个严重的限制。不幸的是，这个问题比阿姆达尔定律提出的问题更严重。除去算法中串行比例的影响之外，还有并行开销的问题。

并行开销是指管理并行应用程序中的线程所花费的时间。例如，在本书的后面，我们将学习大量关于线程协调执行的方式。它们被创建、销毁，有时它们会在开始执行时等待其他线程完成一个动作。对于程序的运行时间而言，这就多浪费了一些时间，或者我们说它给程序的执行增加了开销。在这种情况下，这种开销会随着线程数量的增加而增加。

我们已经提到了管理线程的开销。其他引起开销的原因来自管理数据。如果数据必须分布在处理器之间，当处理器的数量增加时，它们之间的数据移动也会增加。内存速度和网络速度远远低于处理器的速度，因此数据移动会迅速增长并失去了并行处理的好处。对于分布式内存集群来说，这种数据移动开销要大得多，而这些数据移动效应也会影响共享

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

内存计算机。

我们可以非常粗略地模拟并行开销带来的影响，通过向并行程序运行时间方程中加入一个并行开销项。按如下的阿姆达尔定律：

$$T_p = \alpha \times T_s + (1-\alpha) \times \frac{T_s}{P}$$

将并行开销作为一个新的项，得到如下的公式，其随着处理器数量 P 的增加而增加：

$$T_p = (\alpha + \gamma \times P) \times T_s + (1-\alpha) \times \frac{T_s}{P}$$

其中，我们将开销建模为一个按 P 缩放的小常数 γ，并行开销的结果是如图 2-3 中的加速比曲线。加速比迅速攀升，并且总体上增长受制于串行比例（如阿姆达尔定律所预测）。随着处理器的增加，并行开销会增长，最终以比阿姆达尔定律限制所建议的更极端的速度将这个数字向下拉。有时能够创建并行开销很小且串行比例极小的程序。这些情况会导致如图 2-1 所示的加速比曲线。然而，更多的时候，曲线会更像图 2-3 中的曲线。有经验的程序员会预料到这种形状，并会进行仔细的可扩展性研究，来看看加速比曲线在什么地方下降，以确保他们了解其并行程序可扩展性的极限。

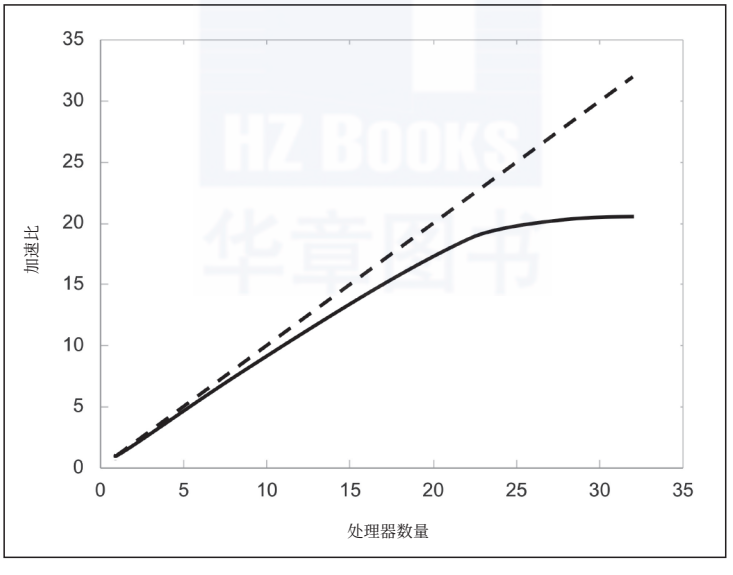


图 2-3 加速比作为带有并行开销项的处理器数量的函数。我们使用线性标尺来表示加速比，用对数标尺来表示处理器数量。虚线是我们预期的完美线性加速比线。它是为了理解观察到的加速比数据而提供一个可视化的参考。实线是一个并行比例为 0.995，并行开销项为 0.0005 的情况下的加速比曲线

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

2.4　强扩展与弱扩展

到目前为止，在本章中，我们已经考虑了对于一个固定规模的问题，性能是如何随着处理器数量的增加而变化的。这被称为强扩展。由于问题大小是固定的，所以随着处理器数量的增加，每个处理器要处理的数据量就会减少。最终随着处理器数量的不断增加，将可能没有足够的工作量使额外的处理器保持忙碌状态。

强扩展难以持续。这也是阿姆达尔在发表以他名字命名的定律时提出的关键点之一。然而，对于如何使用并行计算机这件事，这也许是一种过于悲观的思考方式。在许多问题领域，如果选择一个大小为 N 的问题，该领域的专家可以展示一个更有趣的大小为 $2N$ 或更大的问题。换句话说，基于问题域的需求，通常有很好的理由让问题的大小增长。

为什么要这样做呢？让我们回到当问题的执行中有一些部分是串行的时候，对加速比的分析。如果这个串行比例 α 不是常数会怎样？如果它是问题大小 N 的递减函数呢？这意味着加速比现在是 P 和 N 的函数。

$$\text{Speedup}(P,N)=\frac{T_s}{T_p}=\frac{T_s}{\alpha(N)+\dfrac{(1-\alpha(N))}{P}T_s}$$

如果在 N 足够大（N_{large}）的情况下，串行比例接近零，我们得到

$$\text{Speedup}(P, N_{\text{large}}) = P$$

事实证明，对于很多问题来说，$\alpha(N)$ 确实是 N 的递减函数。例如，在很多密集的线性代数问题中，工作规模为 N^3，其中 N 是问题中矩阵的阶数。如果串行比例来自起初设置的矩阵，那就会随着大小为 N^2 矩阵的大小而缩放。对于足够大的 N，$O(N^3)$ 的工作项远超过了 $O(N^2)$ 的任何项，所以我们可以忽略基于串行比例的项的影响。

这导致了在可扩展性研究中，每个处理器的工作量是固定的。随着处理器的增加，整个问题的大小也在增加。这就是所谓的弱扩展。如果每个处理器的问题大小是固定的并且并行开销可以忽略不计，那么理想弱扩展问题的时间是固定的。换句话说，对于弱扩展，把运行时间作为处理器数量的函数，绘制出来所得到的曲线，理想情况下应该是平坦的。

2.5　负载均衡

我们可以把程序定义的工作看作是程序所执行的全部操作。当我们创建一个并行版本的程序时，我们将工作分成若干块，并将这些块分配给线程去执行。如果有多个处理器使线程可以同时执行，我们就会以并行的方式进行工作以减少程序的整体执行时间。

当一个并行程序的最后一个线程完成时，它才算完成。这是一个需要理解的重要观点，所以

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

我们将重复这一点，但方式略有不同：最慢的线程决定了所有线程何时完成。假设现在所有的处理器都以同样的速度运行，并且假设我们现在运行在一台 SMP 机器上，所以我们可以忽略访问不同区域内存的成本差异，那么当一个线程比其他线程有更多的工作要做时，它就是“慢”的。

因此，我们设计一个并行程序的目标是让所有线程在同一时间完成，这意味着我们希望它们都有相同的工作量。如果把工作看成是给线程施加负载，我们说作为算法设计者的工作就是“均衡线程之间的负载”。

在学习并行编程的过程中，我们将花费大量的时间来讨论负载均衡。我们现在将阐述这个主题的大致观点，而把细节留到以后再讨论。我们对负载均衡的选择定义了四种不同的情况。我们将它们分为两对对立项：

- 显式与自动：是由程序员计算出一个固定的公式来生成负载均衡，还是在计算过程中自动出现负载均衡？
- 静态与动态：工作的分解和安排执行的方式在编译时是固定的吗？还是在程序运行时动态发生的？

我们将定义负载均衡的四种案例，并为每种案例举一个简单的例子。然而，我们并不打算太深入地赘述这个主题。我们希望在本章中为这个重要的话题打下基础，但真正的学习将在后续，在 OpenMP 程序中应用这些概念时开始。

案例 1，显式、静态：程序员根据程序中的逻辑来定义分块。这些表达式在编译时是固定的。因此随着程序的运行，它调整负载均衡方式的能力有限。例如，程序员可能有四个线程，并将工作分解到四个挑选出来的块中，这样工作在每个线程上需要的时间就差不多。然后给每个线程分配一个块，实现有效负载。

案例 2，显式、动态：程序员在代码中写下了决定工作分配方式的逻辑，但当程序运行时，它时常暂停，并重新审视该逻辑以动态地重新分配负载。这种情况会发生在引力模拟中，鉴于系统会不断发展，一些区域会变得更多，而另一些区域则会变得更少。这就需要在运行时调整区域映射到线程的方式（因此这被认为是动态的）。

案例 3、自动、动态：程序创建了一系列的块，并把它们放在一个队列中。线程抓取一个工作块，完成它，然后回到队列中进行更多的工作。工作在各线程之间是动态平衡的，但是程序员不需要决定哪个线程得到了多少分块。这种方法对于工作高度可变和不可预测的问题很有价值。在处理器需要以不同速度运行线程时也很有价值（例如，在一个集群中，有些节点是新的，而其他节点则落后一两代）。在这种情况下，速度较快的处理器就会自然而然地比速度较慢的处理器承担更多的工作。

案例 4，自动、静态：静态负载均衡策略的本质是在工作开始之前，工作的分布就已经固定了。在工作依赖于输入数据集，但系统的性质却得益于静态工作分配的问题中，有时让一个进程在运行时检查计算，以确定一个由线程集合使用的静态调度是很有利的。例如，

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

通常的做法是，当在 GPU 上解决稀疏线性代数问题时，对于这些问题来说，了解数组中非零元素的分布以及计算过程中的填充模式是至关重要的。

我们在探索 OpenMP 的过程中会反复遇到这些情况，尤其是案例 1（显式、静态）和案例 3（自动、动态）。

2.6　用 roofline 模型理解硬件

加速比曲线描述了并行程序中随着处理器数量增加而出现的趋势。然而，当考虑性能时，最终需要了解相对于硬件提供的原始性能的性能。与其研究性能如何随着处理器数量的增加而变化，不如考虑计算的绝对速度，以及在给定的算法和硬件细节的情况下是否可以接受。

为了探讨这些问题，我们使用了一个 roofline 模型 [16]。roofline 模型是一种可视化工具，用于帮助理解计算机系统相对于特定算法特性的局限性。在模型中，你可以绘制性能与算术强度的关系。

- 算术强度是指程序执行的浮点运算次数（Flops）与支持这些运算所需的数据移动的比率。
- 性能用一个比率来表示：每秒浮点运算（FLOPS）。当计算的速率以内存移动为主时，性能就会受到内存带宽的约束，绘制的速率就变成了每秒移动的浮点数。

理想情况下，可以为正在使用的特定系统和算法构建一个 roofline 模型。基于系统的峰值性能来构建的通用 roofline 图是很有效的。我们在图 2-4 中展示了一个典型的通用模型。

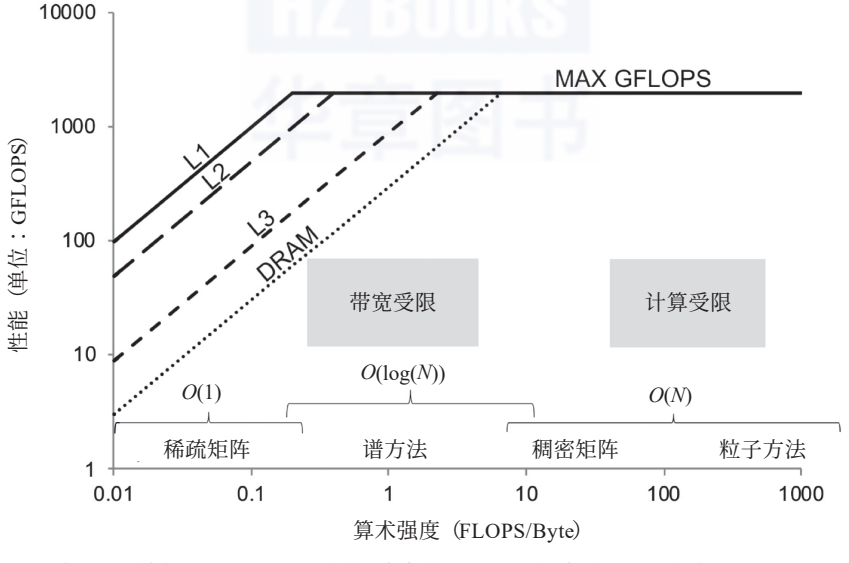


图 2-4　具有三级缓存和 DRAM 内存的系统的 roofline 性能模型。线条显示了不同的性能的极限值，上方的水平线是受浮点运算限制的程序的性能上限，向左下倾斜的斜线显示的是受内存层次结构中不同级别限制的程序的性能界限

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

具有高算术强度的算法（如密集矩阵上的线性代数和粒子方法中的短程力），从内存中访问的每一个字节都要进行大量的计算，以至于它们受到系统峰值浮点性能（最大 GigaFLOPS）的限制。这些计算是有计算约束的。在 roofline 图中为最大 GigaFLOPS 使用适当的值很重要。该值必须与被研究的计算中使用的精度（例如，半精度、单精度或双精度）和主导计算的运算（例如，标量或向量运算或更复杂的运算，如融合乘加）的精度相匹配。

另一个极端是低算术强度的问题，它受到数据在内存层次结构中移动的约束。我们用每秒移动的浮点数来表示存储器层次结构中不同层级的性能。当我们从 L1 到 L2 再到 L3 再到内存（DRAM）时，内存的大小会增加，但带宽会减少。如果问题被分解成适合 L1 缓存的块，并且内存移动到 L1 与计算重叠，那么性能就会受到 L1 带宽的约束。如果 L1 的重用性很差，我们就会下移到 L2 甚至是 L3 缓存。同样主要取决于计算能分解成适合缓存的块的程度。在最坏的情况下，对于带宽约束的计算，性能受限于主内存（DRAM）的数据移动速度。

基于 FFT 和频谱方法的计算属于中等算术强度。如果有正确的缓存管理，它们可以在系统的峰值浮点性能附近运行。但更典型的是，它们最终会在一定程度上受到内存层次结构的限制。

roofline 图的构建是基于系统特点的。评估算法以估计其算术强度，然后测量观察到的性能，看看在 roofline 图上的位置。如果达到了 roofline 图的那个区域可用的峰值性能，你知道已经完成了，且额外的优化工作不太可能得到回报。然而，如果性能远远低于峰值，那么 roofline 图表明，研究不同的方法来重组计算以提高性能是有意义的。roofline 图也可以用来提示可能需要改变算法的地方。如果算术强度处于倾斜的内存约束线之下，是否可以改变算法来增加算术强度以直接进入 roofline 图的更高性能区域？从本质上讲，目标是利用 roofline 模型引导一条优化路径，让代表你的性能的“图上的点”向上、向右移动。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 3 章 Chapter 3

什么是 OpenMP

OpenMP 是一个用于编写并行程序的应用编程接口。虽然它一开始专注于 SMP 计算机的多线程程序，但经过多年的发展，它已经可以应对 NUMA 系统和 GPU 等外设。在本章中，我们将探讨 OpenMP 的历史，并讨论这个重要标准的高层结构。

3.1 OpenMP 的历史

20 世纪 80 年代，市场上出现了少量的共享内存计算机。为这些计算机编写应用程序的程序员很快就发现需要为这些系统提供一个可移植的 API。20 世纪 80 年代和 90 年代，人们为创建这样的 API 做了一些努力，但都失败了。

这些早期标准化工作的问题是，当时对共享内存计算机的关注度并不高。基于分布式内存的系统主导了超级计算领域。在这些系统中，有一个通信网络，计算机位于网络的节点上。每台计算机都有自己的内存，因此它们被称为分布式内存系统。如果这些分布式内存系统有共享的内存节点，则这些节点只需要几个处理器。这样，在每个节点上运行额外的分布式内存进程并完全忽略共享内存通常更容易。

1995 年年底，共享内存计算机的计算环境发生了变化。英特尔发布了一款在 SMP 配置中支持最多 4 个 CPU 的芯片组。这使得 SMP 计算机从 HPC 专用计算机进入主流市场。这也意味着，用于高性能计算的工作站 – 集群中的节点更有可能是具有更多 CPU 数量的 SMP 节点。也是在同一时间，SGI 收购了 Cray Research。这两家公司各自有对共享内存机器进行编程的方式。一旦它们成为一家公司，就只需要围绕着一个共享编程模型将两个产品线结合在一起。最后的关键因素来自应用程序员社区。社区中的人们在美国加速战略计

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

算计划（ASCI）的领导下团结起来，共同推动厂商为 SMP 系统编程定义一个标准。

ASCI 应用程序员在 1996 年年底和 1997 年与主要的 HPC 厂商密切合作，定义了 OpenMP，并在 1997 年 11 月发布了 1.0 版本。由于早期共享内存标准化工作的努力，并借鉴了自 20 世纪 80 年代以来少数共享内存系统厂商的经验，OpenMP 的基本设计很快就完成了。指导原则是：

- 标准化现有的实践，而不是建立一个研究议程。我们希望有一个 API，让厂商可以快速实现，而不需要经过一个漫长的研究过程。
- 支持可移植的、高效的、可理解的共享内存并行程序。
- 为 Fortran、C 和 C++ 提供一致的 API，这样程序员就可以轻松地在不同语言之间转换。
- 创建一个小型的 API，其规模足以表达重要的控制并行模式。
- 保证严格的向后兼容性，这样程序员就不需要重写代码来适应新版本的标准。
- 支持编写串行等效的代码，即在并行运行或作为串行程序运行时产生相同结果的代码。

在标准发布的一年内，所有主流的 HPC、共享内存厂商都提供了 OpenMP 编译器。应用程序员只需写一次代码，只需重新编译程序就可以从一台共享内存计算机转移到另一台计算机。以今天的标准来看，这似乎很平凡，但在 20 世纪 90 年代末 OpenMP 出现时，它是革命性的。

当创建 OpenMP 时，我们就知道它需要成为一种“活的”语言，可以随着硬件和算法的发展而发展。因此，我们创建了一个非营利性的公司来维护这个标准，保护它不被任何一个厂商不适当地操纵，并指导它的持续发展。这个组织被称为 OpenMP 架构审查委员会（ARB）。在 ARB 的领导下，不断产生新的规范版本。我们在图 3-1 中总结了发展情况，图中显示了规范的页数（忽略了前面的材料和附录）。

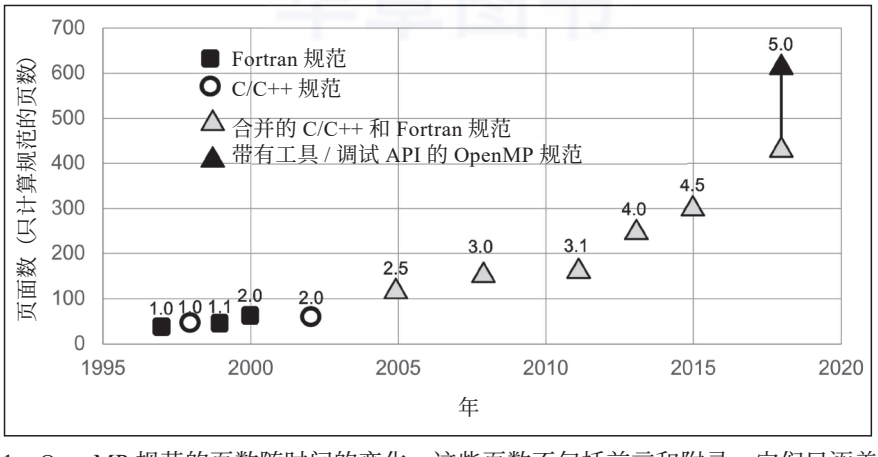


图 3-1 OpenMP 规范的页数随时间的变化。这些页数不包括前言和附录，它们只涵盖了规范中定义 OpenMP 的部分。对于 OpenMP 5.0，区分了单独的 API 的页数和包括新工具接口的页数

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

OpenMP 始于 1997 年 Fortran 的 1.0 版本。它的目标是基本的循环级并行，只需要 40 页就可以完全定义。我们继续在两个方向上完善和开发新功能：一个方向是 Fortran，另一个方向是 C/C++。这两个方向是“串行”的，因为是同样的人在研究 Fortran 和 C/C++ 规范。因此，如果在开发 Fortran 规范时发现了 OpenMP 中的一个问题，那么在 C/C++ 中可能要等上 3 到 4 年才能显示出所需的修改，因为必须在完成 Fortran 规范后才能将任何修改反映到 C/C++ 文档中。为了解决这个问题，我们把 C/C++ 和 Fortran 规范合并成一个文档。这就是 2005 年发布的 OpenMP 2.5，共有 117 页。

从 OpenMP 1.0 到 2.5，关注的焦点仍然是并行性，表现为在并行循环中使用线程。对于 2008 年发布的 151 页的 OpenMP 3.0，我们给 OpenMP 增加了任务。有了任务，就可以超越常规循环，考虑递归和其他不规则算法。

在 OpenMP 4.0（2013 年发布，共 248 页）中，增加了一个主机设备模型。利用这个模型，程序员可以通过优化的处理器（如 GPU）来表达数据并行的算法。这是对 OpenMP 先前技术的巨大突破，使我们进入了具有多地址空间的复杂系统世界。我们还在 OpenMP 4.0 中添加了一些方法来显式控制处理器在向量单元上的执行，即所谓的 SIMD（单指令多数据）并行（将在 12.2 节中详细讨论）。

2018 年 11 月，ARB 发布了 OpenMP 5.0。它有 600 多页，其中 437 页定义了 API，其余规定了一个新的工具接口。它扩展了 OpenMP 中任务的功能，并针对 C++ 和 Fortran 的最新发展更新了 API。它对整个 API 增加了额外的功能以便对管理复杂的内存层次结构有更好的支持，扩展了 GPU 支持，增加了迭代器以支持现代 C++ 的编程风格等。

这个讨论涉及 OpenMP 发展中的主要特性。我们的目标不是提供一个详细的发展历史（可以在论文 [2] 中找到），而是想说明 OpenMP 作为一种活的语言，在过去 20 多年的时间里，其复杂性是如何增长的。开始的时候，我们的目标是让并行编程尽可能简单，由于 OpemMP 是向后兼容的，所以现在仍然如此。然而，由于 5.0 规范的庞大规模，保持开始制定 OpenMP 时的核心简洁特性变得越来越困难。

3.2　通用核心

我们认为，需要改变讲解 OpenMP 的方式。不应该遵循规范的发展 —— 这是我们过去教 OpenMP 的方式，而应该分离出 OpenMP 的核心，抓住 API 固有的简洁性。事实证明，大多数程序员很少甚至没有超越 OpenMP 的核心：大多数 OpenMP 程序使用的基本元素。我们称之为 OpenMP 通用核心。表 3-1 中给出了 OpenMP 中的通用核心项。

我们还没有讨论这些概念，所以不要花时间去理解它们，后面将会介绍。

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

表 3-1 构成 OpenMP 通用核心的编译指令、运行时库函数和子句以及相关的多线程计算基本概念

OpenMP 编译指令、函数或子句	概　念
#pragma omp parallel	并行区域、线程组、结构化块和跨线程交错执行
int omp_get_thread_num() int omp_get_num_threads() void omp_set_num_threads()	SPMD 模式：创建并行区域，使用线程数和线程 ID 分割工作
double omp_get_wtime()	代码的定时块、加速比和阿姆达尔定律
export OMP_NUM_THREADS=N	内部控制变量和用环境变量设置默认线程数
#pragma omp barrier #pragma omp critical	交错执行、竞争条件和同步所隐含的操作
#pragma omp for #pragma omp parallel for	共享工作、并行循环和循环携带依赖
reduction(op: list)	跨组内线程的值归约
schedule(static [,chunk]) schedule(dynamic) [,chunk])	循环调度、循环开销和负载平衡
private(list) firstprivate (list) shared(list)	OpenMP 数据环境：默认规则和修改默认行为的子句
default(none)	每个变量的存储属性的强制显式定义
nowait	禁用共享工作构造的隐含栅栏、栅栏的高成本以及刷新内存
#pragma omp single	由单线程完成的工作
#pragma omp task #pragma omp taskwait	任务、任务完成和用于任务的数据环境

3.3 OpenMP 的主要组件

在完成对 OpenMP 的概述，开始探索 OpenMP 通用核心之前，我们描述一下 OpenMP 的高层结构，以及它是如何与典型的共享内存计算机相适应的。图 3-2 展示了这一点。

我们从底层的硬件开始。OpenMP 通用核心采用了共享内存计算机的 SMP 模型。它不包括任何为解决 NUMA 计算机而增加的更高级的 OpenMP 特性。硬件之上是系统层。操作系统用某种线程模型来支持共享内存计算机。OpenMP 使用操作系统提供的任何一种线程模型，在大多数情况下是 pthreads。在操作系统层之上是 OpenMP 运行时系统。它由支持 OpenMP 程序执行的低层库和软件组件组成。它不是由 OpenMP 规范定义的，而是由

互动出版网 www.china-pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

OpenMP 的实现者编写的。

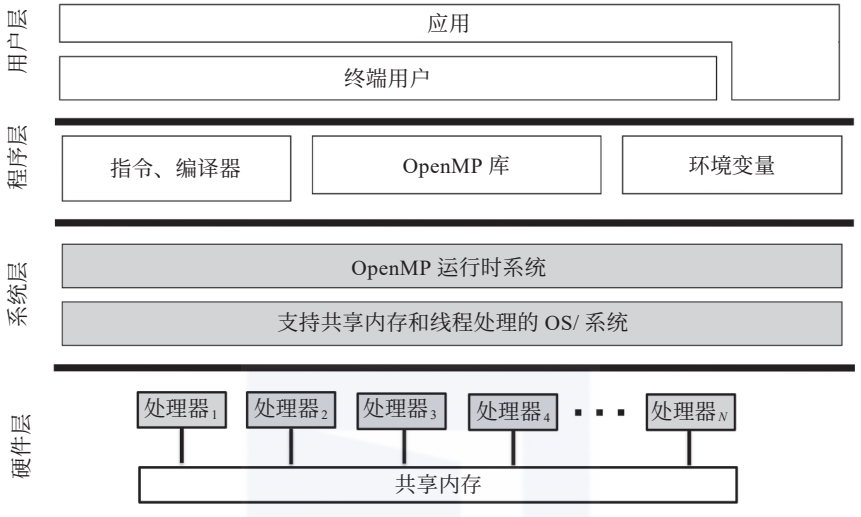


图 3-2　用于 OpenMP 通用核心的解决方案栈

下一层是程序层。这是 OpenMP 规范定义的 OpenMP API。它由三组基本项组成。

- 指令和 OpenMP 感知编译器：指令代表 OpenMP 程序员告诉编译器如何创建多线程程序。
- OpenMP 库：由 API 定义的函数，用于在程序执行时与计算机交互。它解决了在编译时无法解决的问题，如线程数以及控制程序执行的低级原语。
- 环境变量：控制执行程序的特征，并在运行时设置默认参数。

最后一层是用户层，这里的“用户”指的是运行（而不是创建）OpenMP 程序的人。顶层的“终端用户”框的形状表明，当应用程序与 OpenMP 的全部项目交互时，用户可以通过环境变量直接与 OpenMP 运行时交互。

我们对 OpenMP 及其历史的概述到此结束。你现在已经知道了所有需要的背景，下面开始探索 OpenMP 通用核心。