并行乱弹

风辰

自序

并行是指在具有多个处理单元的系统上，通过将计算划分为多个部分，将各个部分分配到不同的处理单元上，各处理单元相互协作，同时运行，以达到加快求解速度或者提高求解问题规模的目的。

有一个和并行相似的概念，称作并发，狭窄的说并发是指在一个处理单元上运行多个应用，各应用分时占用处理单元，是一种微观上串行、宏观上并行的模式，有时也称其为时间上串行、空间上并行。

一般来说，并发是为了满足应用的功能需求，比如在计算的同时，用户界面能够响应用户。而并行更多的是为了提高速度或为了解决更大规模的问题。

人类生活的方方面面存在着并行或者并发，边吃饭边看电视，双手同时拔草，甚至吃饭时，嘴巴的动作和手的动作也是并行的。和人类社会的广泛存在并行不同的是：计算机编程几乎一直都是串行的，绝大多数的程序只存在一个控制流。然而对于并行的研究却可以追溯到上世纪六十年代，但是直到近年来才得到广泛的关注，这主要是自2003年来，由于能耗和散热问题限制了CPU频率的提高导致多核处理器的广泛使用。

2003年以前，每18个月CPU的性能提升大约是2倍，这就是摩尔定律。在摩尔定律的作用下，软件开发人员只管写好软件，而性能就交给硬件了。2003年后，这种“免费的午餐”已经不复存在，为了生存，各硬件生产商不得不采用各种方式以提高硬件的计算能力，目前最流行的方式是在同一个芯片中集成多个处理单元，这种转变是不负责任的，而前途也并非一片光明。如著名计算机科学家高德纳（Donald Ervin Knuth）就认为：“在我看来，这种现象（多核或并行）或多或少是由于硬件设计者已经无计可施了导致的，他们将Moore定律失效的责任推脱给软件开发者，而他们给我们的机器只是在某些指标上运行得更快了而已。如果多线程的想法被证明是失败的，我一点都不会感到惊讶，也许这比当年的Itanium还要糟糕——人们基本上无法开发出它所需要的编译器。”但是这种转变已经在软件开发社区产生了巨大的影响。

目前绝大部分应用软件都是串行的，串行执行过程符合人类的思维习惯，易于理解、分析和验证。由于串行软件只能在多核CPU中的一个核上运行，这

和2003年以前的CPU没有多少区别，这意味着花多核CPU的价钱买到了单核的性能，是不能接受的。通过多核技术，硬件生产商成功的将提高实际计算能力的任务转嫁给软件开发人员，而软件开发人员没有选择。

有两种方式能够提升软件的性能:

1. 优化串行代码，
2. 采用并行技术。

前者并不能利用多核CPU的全部计算能力，但是它并不要求软件开发人员掌握并行开发技术，另外因为无须对软件做大的改动，而且串行代码优化有时能够获得非常好的性能，因此相比采用并行技术，应当优先选择串行代码优化。一般来说采用并行技术获得的性能加速不超过核数，这是一个非常大的限制，因为目前CPU硬件生产商最多只能集成十几、几十个核。

研究人员经常利用网络连接多台机器，多台机器协同解决问题，以增加计算能力，在这类机器上编程常用消息传递例程MPI，随着计算机价格的降低，普通开发人员越来越有机会接触到它们，但是这方面本人也不是非常熟悉，因此本文只给予一个简略的介绍，并不打算深入。

目前大多数计算机拥有两个或者四个核心，多核已经成为主流配置，因此本文将会详细的介绍基于共享存储器的多线程并行编程，pthread、OpenMP和最新的C/C++标准都在考虑之列。

从2006年开始，可编程的GPU越来越得到大众的认可，GPU是图形处理单元(Graphics Processing Unit)的简称，最初主要用于图形渲染。自20世纪90年代开始，NVIDIA、AMD(ATI)等GPU生产商对硬件和软件加以改进，GPU的可编程能力不断提高，GPU通用计算比以前的GPGPU(General-purpose computing on graphics processing units)容易许多，另外由于GPU具有比CPU强大的峰值计算能力，近来引起了许多科研人员和企业的兴趣，本文也会反映这一变化。

为了应对GPU 的挑战，Intel 终于推出了其众核硬件架构MIC。从架构上看，MIC 既有X86 CPU 的基因，又有GPU 架构的思想。

由于GPU和CPU硬件之间的不同，导致了基于GPU的并行编程和基于CPU的并行编程在细节上有许多不同的考虑，但是在大的设计方面是基本一致的。而CPU和GPU协作的异构并行计算越来越获得科研人员的青睐。

本系列不但包括多机、多核的优化及并行，还包括新出现的基于图形处理器(GPU 和MIC)和移动处理器的代码优化及并行。不但有实际的并行方式的介绍说明，还有理论的分析。作者希望通过这种方式能够让阅读本文的软件开发人员掌握并行编程方法。

整体而言，本书分为三个部分：

- 理论基础，本部分主要介绍并行软硬件基础，并行算法设计思想以及一些软件优化方法。
- 程序设计，本部分主要介绍目前主流的并行编程环境，如POSIX进程和线程、OpenMP、MPI、pthread，以及基于GPU的异构并行编程环境CUDA、OpenCL和OpenACC。
- 算法设计实践，本部分主要通过一些常见算法的并行化以加深读者对并行的理解，涉及的领域有线性代数、图像处理、物理模拟等。

本书希望通过这种方式能够让读者渐进的、踏实的拥有并行思维，并且能够写出优良的并行代码。

目　　录

表　　格

插　　图

第一部分

理论基础

第一章　绪论

在2003年以前，计算机性能的提升主要依赖CPU主频的提升，科研人员只要写好程序，几乎用不着优化，因为下一代CPU主频的提升会轻易的提升软件的性能，这使得计算机行业进入一个良性循环：由于性能的提升，人们能够使用计算机做更多的事，当人们习惯当前计算机的速度后，又会提出新的性能要求，以让计算机更快地做更多的事，而CPU生产商也乐于升级硬件以赚取更多的利润。但是由于CPU的功耗与频率的三次方成正比，这使得无限制的提升频率成为了不可能。为了能够卖出自己的产品，各CPU生产商纷纷通过其它各种方式提升计算能力，如提高指令级并行能力、在一个时钟周期内执行更多指令、超线程技术等。从长远来看，最有可能引领未来的是多核技术，多核采用在同一个芯片上集成多个核心的办法。而目前GPU通过将大量核心集成在一块硅片上以提升性能，这称为众核，由于众核集成的核心数量远远超过多核，因此其原生性能也超过多核。

1.1　并行的作用

并行的首要作用是尽量发挥硬件提供的全部计算能力。目前绝大多数软件都是串行的，虽然目前CPU使用的乱序执行、指令流水线等指令级并行技术使得程序的执行并不完全和串行的代码系列一致，但是软件依旧只有一个控制流。这就使得它们只能利用多核/众核提供的部分计算能力，为了利用多核/众核提供的全部计算能力，必须采用并行的方式编写软件，而现有的串行软件必须修改。

并行/并发的另一个作用是实现功能，比如需要软件在计算的同时能够响应用户的交互，此时就必须使用并行/并发，因为存在两个或多个控制流。

在科学计算中，物理模拟需要长时间的运行以获得更精确的结果，因此并行技术应用比较广泛。在这个领域，并行主要提供两方面的作用：

- 让程序算得更快，以节约时间，如果程序能够计算得更快的话，缩短了时间，就可以在同样的时间做更多的试验；

- 让程序能够计算更大规模的体系，大规模体系要求非常大的计算能力，因此对并行的需求更为迫切，而且大规模意味着可以更真实的模拟现实系综。

1.2 为什么要并行

从2003年开始，CPU频率的提升接近停止，那种每次硬件的更新都会提升软件的性能的“免费午餐”已经结束，为了提升软件的性能，软件开发人员不得不使用并行技术。

有些编译器作者想让编译器自动并行化串行程序，这一直是人们的奋斗目标。1980年代中期，基于依赖分析的自动向量化工具已经成熟，可以帮助程序员将Fortran语言代码移植到向量计算机上进行并行计算。后来的研究转向共享存储的MIMD（Multi instruction Multi Data，多指令多数据）和分布式存储结构的自动并行化，到目前为步，这种方法少有进展，这主要是因为编译器没有办法收集/分析并行所需的数据相关性等信息，必须要程序员干预。现在，研究重点又逐步转向基于语言的策略研究，即从用户那里获得更多信息，同时利用自动化并行技术来减轻程序设计的负担。

由于单核的性能已经不能大幅度提升和多核/众核技术的普及，只有并行才能利用多核/众核带来的性能提升，而人们总能发掘要求更高计算能力的应用（希望程序能够运行得更快或者能够计算更大规模的问题），这些应用对计算能力的需求推动着硬件和软件技术向前发展。

1.3 为什么并行难

并行编程方式和目前通行的串行软件开发方式的不一样，使得并行软件的开发难度远超串行软件的开发，主要原因有人为的也有技术方面的。

由于多核及并行技术的流行只是近几年的事情，软件开发人员还没有足够的经验来应对。而初学者也没有很好的资料及成熟的项目代码学习，另外并行编译器和调试工具的匮乏，这种现象和计算机编程早期一样，随着并行的流行，最终并行编程将会越来越简单。

技术方面的原因主要有：

- 遗留代码，过去几十年积累下来的代码是企业的巨大财富，没有人会放弃。

- 可扩展性，如果代码能够发挥双核的计算能力，那么4核、8核、16核，甚至百核呢？何况并行程序在百核上会发生什么事情也是未知数。

- 可维护性，并行代码的可读性通常不如串行代码，如何在开发人员离开后，接管的人员能够掌控就变得很重要。

- 任务/数据划分，并行意味着多个控制流同时执行，需要在各个控制流之间划分任务和数据，数据/任务的划分方式不但决定了编程时的难易，而由于划分带来的负载均衡和通信问题往往会对程序的最终性能产生决定性的影响。

- 并发访问控制，多个控制流需要访问不同的或相同的资源，如何协调对这些资源的访问就变得非常重要，也成为并行编程的一大难点。

- 资源划分，资源划分方法不但关系到编程的难易还关系到最终的性能。

- 与硬件交互，为了最好的发挥性能，软件开发人员通常会应用硬件的特性。

- 对软件开发人员的过高要求，开发工具不够智能，且市场不愿付出相应的薪水。

下面将详细述说各个方面。

1.3.1　遗留代码

一些项目拥有成千上万，甚至百万行代码，如何在这些代码中加入并行性就非常难。通常并行化的难度和代码的长度成线性关系，而且当前维护这些代码的人员通常并非原始的开发人员，这使得并行的代价和风险非常大。

对于遗留代码来说，基于编译制导的编译器会是一个比较安全的选择。编译制导能够保证原来的代码还能够运行，允许软件开发人员逐步的并行化现有代码，便于验证。

1.3.2 可扩展性

现在8核的机器已经开始普及，编写的程序在8核上可扩展性可能会比较好，但是如果把程序放到32核、64核上会发生什么事情，或许此时需要重新编写代码。而Amdal定律告诉我们程序是不太可能完全线性加速的。最终程序或者硬件会达到一个极限，在这个极限上，再增加核心数量就不会提升性能了。

可扩展性的主要解决办法是在开发项目时，要注意留下足够的设计文档。源码有足够的、准确的注释。

1.3.3 可维护性

由于在原有的串行逻辑中加入了多个控制流的调度内容，使得并行代码比串行代码难以维护。或者同时维护一个串行版本和一个并行版本，这就非常难以让两者保持一致。

1.3.4 任务/数据划分

由于并行需要将多个工作划分成几个小部分，然后每个控制流处理一个或多个部分。任务/数据划分时需要十分小心，划分方式不但影响编程的难易还影响程序最终的性能。比如，不均匀的划分会导致负载不均衡[1]。而某些划分方式会导致程序的很多代码顺序执行。另外，划分可能导致某些全局处理变得复杂，此时可能需要同步以安全的处理这些全局数据。

依据对任务和数据的划分方式的不同可将并行编程划分为不同的编程模式，本文会在第??章详细分析。

通常划分后各个控制流之间需要一些通信（易并行可能无须通信）。由于通信会引起开销，不成熟的划分方式可能使得通信的开销过大导致性能极端的下降。

基于CPU的并行编程中，控制流的数量必须加以控制，因为每个线控制流都会占用一些资源，比如缓存、虚拟存储器。如果过多的控制流同时在一个处理器核心上执行，每个控制流使用的资源数量就会减少，可能会引起过高的缓存不命中，从而降低性能。另一方面，大量的线程可能会带来大量冗余计算和I/O操作。

[1] 负载均衡能被用来在控制流之间重新分配任务/数据，以获得更好的性能

最后，并行会大量增加程序的状态空间，导致人脑难以理解，降低生产率。这一点通常可以通过采用成熟的软件工程方法予以克服。

1.3.5　并发访问控制

并行程序的多个控制流需要协调对某个资源的访问，比如打印机，如果不加以控制的话，并行程序打印出来的可能就是“天书”。

基于消息传递的编程模式允许各控制流拥有自己独立的存储器内容，此时数据的交流通过传递消息实现。此时要注意由于资源访问导致的死锁、活锁及饿死等问题。

基于共享存储器的编程模式只有一个存储器空间，这样各个控制流访问同一存储器地址时就有可能产生冲突，常见的有“读后写”、“写后读”和“写后写”等问题。对于这类问题，通常通过互斥[2]资源的访问解决。

并行编程中，最常见的并发访问控制是文件，如果多个控制流同时读一个文件，就有可能读到错误的数据，常见的解决方法有：

- 由一个控制流读取文件，然后分发数据。
- 将文件分成多个子文件，每个控制流读取自己的子文件。

前一种方式编程简单，但是由于分发数据操作完全是串行的，有可能会导致过大的性能开销。而后一种则相反。

关于并发资源的访问，比较明智的做法是将访问分为写和读，由于不同的控制流可以同时读一个数据，因此此时无须访问控制。而多个控制流要写的数据必须要特殊处理。需要提醒读者的是，当对一个数据有些控制流读有些控制流写时，也必须特殊处理。

1.3.6　资源划分

如何在不同的控制流之间分配计算资源一直是并行的难题，这往往和负载均衡关联，如果分配给某个控制流的资源多，就可能要让其它的控制流等待它计算完成。在基于x86的处理器上，这往往只涉及到内存、共享文件的划分。在基于GPU的并行计算环境上，这个问题往往更加复杂。

[2]互斥是指某一时刻只允许一个控制流访问

资源划分和并发访问控制、通信密切相关，好的资源划分方式能够既减少通信又保证资源访问的局部性，这通常意味着优秀的性能和可扩展性。

资源划分通常依赖于应用。数值计算频繁的将矩阵按行、列或者子矩阵进行划分。控制流可能会静态的分割数据，或者随时间改变。资源划分非常有效，但是随之带来的复杂的数据结构的处理也非常有挑战性。

1.3.7 与硬件交互

并行编程要求软件开发人员对机器的配置比较了解，只有这样才能避开硬件的缺陷。涉及到新的硬件特性时，经常需要直接与这些硬件打交道。当需要榨取系统的最后一点性能时，通常需要直接访问硬件。

由于不能的硬件其设计方法、发挥硬件性能的编程方式及硬件设计上容易成为性能瓶颈的地方都不相同，这些因素可能会导致在某一硬件上性能很好的算法，在另一硬件上性能却非常差。

基于多机系统编程时，网络的拓扑结构和网线的传输速度非常重要；基于多核编程时，核心和缓存之间的组织比较重要[3]；基于GPU编程时，GPU硬件的组织更为重要，如核心之间缓存的组织、DRAM的组织、核心的组织以及程序如何映射到硬件上执行。在这些情况下，软件开发人员需要根据目标硬件，协调程序各个方面的设计。

1.3.8 对软件开发人员的要求

目前编译器及开发环境对并行的支持能力比较差，主要包括以下三个方面：

- 不能自动并行化，
- 不能找出并行冲突的地方，
- 不能协调资源访问。

Intel 的parallel studio工具系列能够发掘出程序的简单的并行性并识别读写冲突，另外其具有简单的自动并行化能力（能够自己决定是否使用SSE指令及OpenMP）。

[3]这个方面经常出现的是伪共享问题

由于编译器缺乏相应的功能，软件开发人员不得不自己来做。软件开发人员需要自己发掘应用的并行性，并且处理共享资源的访问冲突。另外由于不同的并行化方法可能利用了硬件/软件不同的特性，因此其性能更难以把握。

最后目前的调试器对并行的支持非常的差且不可靠，软件开发人员缺乏工具导致生产率上不去，这就导致了雇主不愿使用并行开发。

由于广为人知的原因，硬件生产商极力的、不负责任的吹嘘并行编程是如此简单，使得很多雇主认为只要付给并行软件开发人员和串行软件开发人员一样的工资就够了，而且一般而言并行软件的开发周期比串行软件开发要长得多，这也导致了软件开发人员不愿意使用并行技术[4]。

1.4　并行的替代方法

由于并行编程的难度和软件开发周期长，很多人不愿意使用并行，但这并不意味着他们不能享有并行的好处，另外一些方法也能够像并行一样提升软件性能。下面给出了几种简单方法：

- 运行同一程序的多个实例。
- 利用已有的并行库。
- 优化串行程序。

1.4.1　运行同一程序的多个实例

如果需要计算同一条件作用在不同的数据集上的效果，或者要计算同一数据集在不同的配置条件下的运行结果，软件开发人员可以在系统上同时为每个数据集运行一次串行程序，这样多个进程可占用计算资源。或者为每一种配置条件运行串行程序的一个实例，这样运行每个配置条件的多个实例可占有多个计算资源。或者同时在一台机器上运行多个程序，虽然这没有减少单个程序的运行时间（甚至增大了），但是整体的吞吐量得到了提高。

[4]个人认为市场应当给经验丰富、能力强的并行软件开发人员2倍以上的同等能力的串行软件开发人员的工资，否则开发优秀的并行软件只能是一句空话

1.4.2 利用已有的并行库

目前已经有许多函数库实现了并行，如Intel的MKL、IPP、TBB和NVIDIA开发的基于其CUDA计算环境的NPP、cufft、cublas等，这些库简化了并行的设计，使用这些库能够方便的利用并行。

1.4.3 优化串行程序

优化串行程序通常应当比并行化更吸引人，因为一方面与核心的数量没有直接的关系。另一方面其性能提升可能是指数级，而并行化带来的性能提升通常是和核心的数量成线性的。本书第??章专门述说串行程序性能优化的技巧。

1.5 性能优化准则

并行也是一种性能优化方法，也遵守优先并行占用时间最长的模块的准则。对于这个准则的一个简明的解释就是如果一个程序用时10秒，无论如何优化耗时1秒的模块，总的时间不会少于9秒。而如果优化耗时9秒的模块，更容易优化到总耗时9秒以下。

1.5.1 加速比与并行效率

通常使用加速比来定义优化效果，其表示优化前程序的运行时间与优化后运行时间的比值，计算方式如下：

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

其中T_s表示串行程序的执行时间，T_p表示并行化后的执行时间。加速比定理说明了并行优化的效果，但是其无法说明并行优化的可扩展性，而并行效率表达了这一概念。并行效率定义为加速比与计算单元数的比例，计算公式如下：

$$P = \frac{S}{M_p}$$

其中M_p表示使用的核心数量。一般而言，如果并行效率低于0.5就说明并行优化是失败的[5]，此时应当减少核心数目而非相反。一般并行效率在0.75以上就已经非常好了。

并行效率和可扩展性紧密相连，并行效率越高，可扩展性通常就越好。

有时并行化获得的加速比会大于处理器数目，这称为超线性加速。超线性加速的原因之一是缓存，因为每个核心通常有自己的一级缓存，如果单个处理器没有办法将数据全部放到缓存的时候，多个处理器划分后的数据有可能放入缓存中，如果缓存的效果超越多线程的开销时，就会出现超线性加速现象。

1.5.2 Amdahl定律和Gustafson定律

Amdahl定律描述了在固定问题规模的前提下，对某个模块获得了加速比S后对程序整体性能的影响，假设并行的部分在未优化前占整体比例为f，则程序的整体加速比为：

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

S'对f求导可知，在$[0, 1]$范围内，函数递增，这表明可并行的部分越多，整体加速比就会越大。

Gustafson定律描述了增加处理器数目的同时增大问题的规模对加速比的影响，Gustafson定律认为此时加速比是线性的，实际上这种情况只在解决问题的时间和规模之间存在线性关系的时候成立，如果其关系非线性，那么就不成立。

强扩展和弱扩展是和Amdahl定律和Gustafson定律相关的另外两个概念。强扩展是指在固定问题规模的前提下，随着处理器数目增加，其性能或加速比的变化情况。而弱扩展是指在固定每个处理器的问题规模的前提下，随着处理器数目的增加，其所能够处理的总的问题规模的变化情况。大致来说，强扩展对应Amdahl定律，而弱扩展对应着Gustafson定律。

[5]这意味着双核的性能还比不上单核，当然如果你有几十个核，可能会认为并行效率为0.5以下也是成功的

1.6 进程、线程与处理器

现代处理器和进程、线程两个概念紧密关联。进程的概念简化了程序设计、存储器管理。并且提供了一种大粒度并行的方法。线程存在于进程之中，进程中的所有线程共享进程的资源，因此更易于通信。本节介绍与此相关的进程、线程、超线程和处理器。

通常基于进程的是像MPI一样的分布式存储器编程模式，基于线程的是像pthread、OpenMP等的基于共享存储器编程模式。由于分布式计算的各节点有其独立的存储器，因此基于进程的消息传递通信更适合，而多核等由于共享存储器，因此基于线程的共享存储器更易于通信。

1.6.1 进程

当程序在系统上运行时，操作系统会提供一种假象，就好像系统中只有这个程序在运行，只有该程序在使用核心、DRAM和设备。如果真的只有这个程序一直在使用核心的话，在单核心的系统上，用户就必须得等待当前正在执行的程序运行完才能输入下一条指令。现代系统并非这样的，这是通过进程的概念实现的。

进程是对操作系统正在运行的程序的一种抽象。多个进程可以同时运行在一个核心上，通过时间片轮转，进程就好像它一直在使用核心。系统内的多个进程通过时间片轮转并发执行，这就要求有一种机制能够保存正在运行的进程的状态并重启另一个进程的运行，这称为上下文切换。

上下文是指保持进程运行所需要的寄存器、缓存和DRAM。在任何一个时刻一个核心只能运行一个进程，当操作系统需要在某个核心上运行另一个进程时，就会进行上下文切换。

通过上下文切换进程获得了并发的特性，而在多核心上的多进程又获得了并行的特性，由于进程的上下文切换和通信比较耗时，因此基于进程的并发往往只适合于大粒度的任务并行。

进程和程序有关系也有不同，程序是静态的指令的集合，而进程是程序正在运行的状态。进程是资源拥有的独立单位，不同的进程拥有不同的虚拟地址空间，不能够直接访问其它进程的上下文资源。

1.6.2　线程

进程之中可以有许多线程，这些线程共享进程的上下文，如存储器空间和文件，但是独立执行且可通过存储器进行通信。当进程终止时，进程内的所有线程也会同时终止。另外线程也有其私有存储器栈和指令指针PC。

由于线程共享进程资源，因此新线程的建立、销毁比进程高效，逐渐比进程更引人关注。

由于进程的存储器资源是独立的而线程的存储器资源是共享的，因此通常基于进程的并行编程更简单，但是基于线程的并行在多核处理器上通常更高效。在多机系统中，不同的计算机天然适合多进程。因此在多核应优先选择线程级并行，而多机系统应选择进程级并行。实际上，许多现代系统及大规模程序充分利用这两种优势，在节点间使用进程级并行，在节点内的多核上使用线程级并行，这称为混合或超级并行。

目前基于GPU的并行编程也使用基于线程的开发环境，但是一种“硬件线程”，其线程的创建、调度和销毁开销接近为0。

多线程程序在多核和单核上执行时具有明显的差别。由于在单核上多线程通过分时共享执行，这使得一些长延迟的操作如锁、IO访问不会导致核心空闲。事实上，网络服务器就是通过多线程技术来提升系统的吞吐量。

1.6.3　超线程

Intel的一些高端机器支持称为超线程（Hyper-Threading，HT）的技术，超线程通过双倍一些资源(PC和寄存器)来减少线程的切换代价，但是只有一份执行单元，因此其峰值计算能力并没有提高。对于那些指令类型丰富且多的应用，超线程能够很好的提升性能。但是超线程不是万能的，在某些应用上性能可能会下降，而在绝大多数应用上提升不会超过20%。

超线程技术将一个物理处理器核模拟成两个逻辑核，可并行执行两个线程，能够在单个时针周期内在两个线程间切换，让单核都能使用线程级并行计算，减少了CPU的闲置时间，提高CPU的运行效率。采用超线程，应用程序可在同一时间里使用芯片的不同功能单元。单线程核心在任一时刻只能够对一条指令进行操作，而超线程技术可以让一个核心同时进行多线程处理。

Intel表示，超线程技术让（P4）处理器在只增加5%的芯片面积的情况下，

就可以换来15%至30%的效能提升[6]。超线程技术需要主板芯片组和操作系统的配合，才能充分发挥效能。

1.6.4 进程线程退出及返回

进程指一个运行中的程序，进程退出或返回意味着程序执行结束或终止。进程终止/退出通常是调用exit函数，进程返回指其中的主线程执行结束（return ）。函数返回指函数调用return或代码执行完，而pthread线程执行的函数return指线程返回，而pthread_exit则会退出线程。

1.6.5 阻塞和同步

在并行编程中，阻塞和非阻塞，同步和异步是非常常见的名词。在某些文献中，阻塞与同步、非阻塞与异步的含义是一样的。

具体来说，阻塞是相对于进程或线程本身而言，如果一个操作并不阻碍进程或线程接着执行代码，称这个操作为非阻塞，反之则为阻塞。相对非阻塞来说，阻塞更为常见，因为非阻塞可能会带来数据一致性问题。

同步或异步则相对于交互的多个进程或线程，如果一个进程或线程与其它进程线程交互时，不需要其它线程做好准备，称之为异步，反之则为同步。

阻塞和同步的具体含义可能会依据不同的编程环境、语言有微小的不同。

1.7 并行硬件平台

对于并行应用来说，如果选对了并行硬件平台，性能会比较好，编程也会简单。下面列出一些常用的并行硬件平台，并试图说明其适合的编程模式及优缺点。

1.7.1 机群

通过使用网线依据某种拓扑方式将多台机器互联以获取更大的计算能力，这种系统通常称为机群。目前所有的超级计算机都是机群系统。由于程序运行时需要通过网线在各个节点间交换数据，因此MPI成为这类平台的首选。

通常机群通过TCP/IP协议通信，使用物理网络互联，基本示意图如1.1。

[6]实际上，在某些程序或非多线程程序而言，超线程反而会降低性能

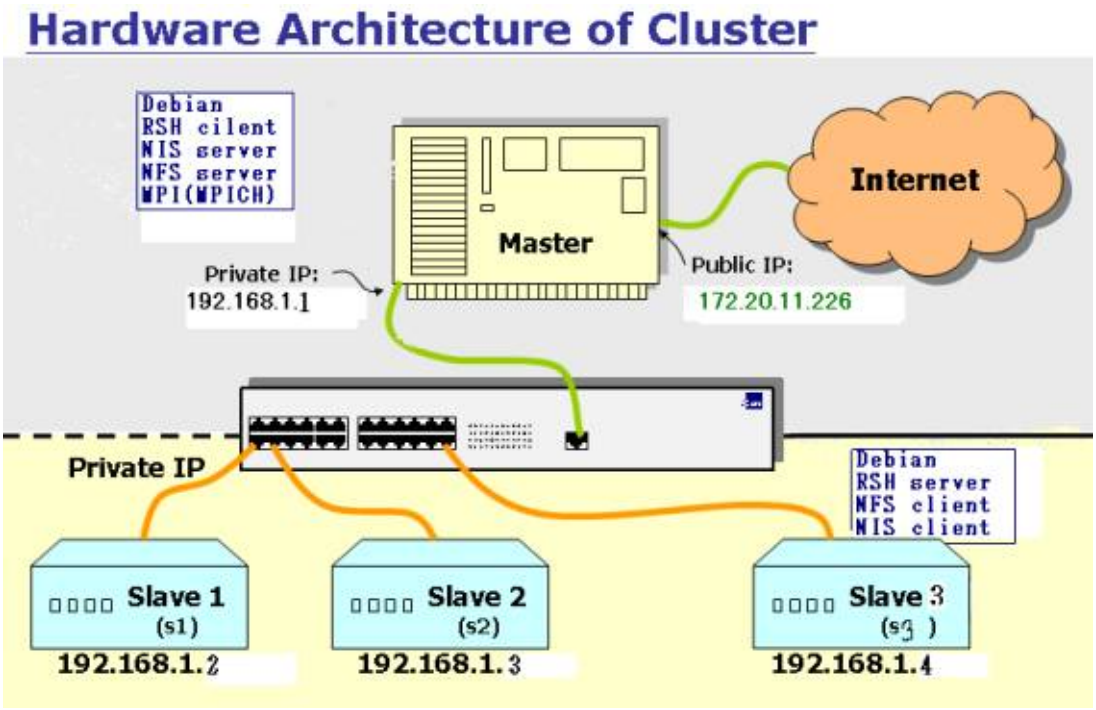

图 1.1: 简单机群结构，图片来自Internet

目前常用于机群通信的网线主要有千兆以太网和infiniBand，其中千兆以太网的最高速度为125MB/s，而infiniBand可达7GB/s，这大约是现行处理器速度的百/千分之一，极易在计算时成为瓶颈。

将多台机器互联在一起的方式称为网络互联，目前流行的互联方式有星形、环形、树形、网格和超立方。不同的互联方式对数据传递的速度影响非常大，如网格就适合具有局部性的数据传递应用。

由于普通软件开发人员并不拥有建立机群的能力，但考虑到某些研究者的需要，本文会在第??章简略介绍基于机群的MPI编程。

1.7.2　多核

多核（multi-core）即多微处理器核心，是将两个或更多的独立处理器核封装在一个集成电路（IC）芯片中的一种方案。多核可以执行线程级并行处理（Thread-Level Parallelism，TLP）。由于生产商大量生产这种多个核心集成的芯片，因此硬件随处可得，近年来越来越获得开发人员的重视。

相比超线程，多核是真正的线程级并行设备。多核与超线程技术相结合，可能会进一步提高系统的吞吐量。

多核的每个核心里面具有独立的一级缓存，共享或独立的二级缓存，有些机器还有三级缓存，所有核心共享内存DRAM。如Intel Core i7处理器具有4个核，高端版本还支持超线程，其中每个核心具有独立的一级数据缓存和指令缓存，统一的二级缓存，并且所有的核心共享统一的三级缓存。

由于共享三级缓存，因此使用多核编程时，平均每核占用的缓存要小，这使得某些时候可扩展性看起来没那么好。由于每个核心都有独立的一级，有时还有二级缓存，这是超线性加速的原因之一。

图1.2展示了某个AMD多核的组织结构。

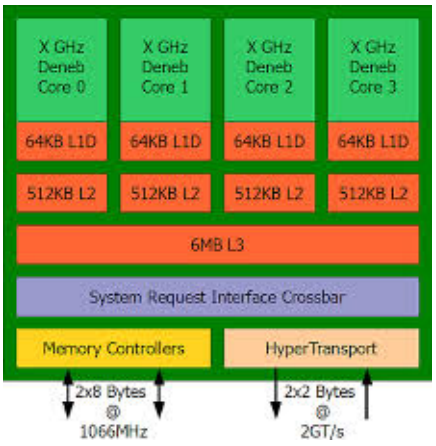

图 1.2: 多核结构示意，图片来自Internet

硬件生产商还将多个多核芯片封装在一起，称之为多路，多路之间共享内存。由于多路之间缺乏缓存，因此其通信代价高昂。一些多核也将内存控制器封装进多核之中，直接和内存相连，以提供更高的访存带宽。

多路上还有两个和内存相关的概念：UMA（均匀内存访问）和NUMA（非均匀内存访问）。UMA是指多个核心访问内存中的任何一个位置的延迟是一样的，NUMA和UMA相对，访问离核心近的内存其延迟要小。如果程序的局部性很好，应当开启硬件的NUMA支持。

多核编程通常使用OpenMP和pthread等线程级并行工具，容易产生的性能问题主要是伪共享。

本文将在第??章和第??章分别详细介绍基于多核的pthread、OpenMP编程。

1.7.3　neon/SSE/AVX/MIC

neon 是ARM 处理器支持的向量指令，具有16 个长度为128 位的向量寄存器[7]，处理器可同时操作向量寄存器的16 个字节，因此具有更高的性能和带宽。本文第??章简略介绍了如何使用它们。

SSE是X86 处理器支持的向量指令，具有16个长度为128位（16个字节）的向量寄存器，处理器能够同时操作向量寄存器中的16个字节，因此具有更高的带宽和计算性能。AVX将SSE的向量长度延长为256位（32字节）。由于采用了SIMD编程模型，SSE/AVX的使用范围非常有限，而且使用其编程实在是一件痛苦的事情，本文第??章简略介绍它们。

MIC是Intel的众核架构，它拥有大约60左右个X86核，每个包括向量单元和标量单元。向量单元包括32个长度为512位（64字节）的向量寄存器，支持16个32位或8个64位数同时运算。目前的MIC的核为按序的，因此其性能优化方法和基于乱序执行的X86有很大不同。

为了减小使用SIMD指令的复杂度，Intel寄希望于编译器，实际上Intel的编译器向量化能力非常不错。在MIC上编程时，软件开发人员的工作由显式的使用向量指令转化为改写C代码以让编译器产生优化的向量指令。

另外，现代64位CPU还利用SSE指令执行浮点运算。

1.7.4　GPU+CPU

近年来GPU（Graphics Processing Unit，图形处理器）的晶体管集成度和（多核并行）处理能力的发展速度都远远快于CPU（Central Processing Unit，中央处理器），CPU与GPU的融合是芯片技术发展的一种大趋势。Intel和AMD都在其CPU中集成GPU，而Nvidia和ATI（AMD）则在其GPU中增加CPU的功能。

GPGPU是一种利用本来用于处理图形任务的GPU来完成原本由CPU处理的（与图形处理无关的）通用计算任务。由于现代GPU强大的并行处理能力和可编程流水线，令其可以处理非图形数据。特别在面对单指令流多数据流

[7]这些寄存器以q 开头，也可表示为32 个64 位寄存器，以d 开头

（SIMD），且数据处理的运算量远大于数据调度和传输的需要时，GPGPU在性能上大大超越了传统的CPU应用程序。

NVIDIA和AMD持续的改进GPU的编程能力，尤其是CUDA和OpenCL推出后，基于CPU+GPU的异构并行计算越来越得到大家的重视。

GPU是为了渲染大量像素而设计的，因此带宽比延迟更重要。考虑到渲染的大量像素之间通常并不相关，因此GPU将大量的晶体管用于并行计算，故在同样数目的晶体管上，具有比CPU更高的计算能力，如图1.3。

图 1.3: GPU将更多晶体管用于计算，图片来自Internet

CPU和GPU的硬件架构完全不同，因此其编程方法很不相同，很多使用CUDA的开发人员有机会重新回顾学习汇编语言的痛苦经历。GPU的编程能力还不够强，因此必须要有对GPU特点的详细了解，知道哪些能做，哪些不能做，才不会出现项目开发途中发觉有一个功能无法实现而导致中止的情况。

由于GPU的访存带宽与CPU内存带宽的比例要比峰值计算性能比小[8]，因此那些计算访存比低的应用不适合在GPU上实现。CPU+GPU异构计算需要在GPU和CPU之间传输数据，而这个带宽比内存的访问带宽还要小，因此需要在GPU和CPU之间进行大量数据交互的解决方案可能不适合在GPU上实现。

本书将在第??章介绍基于GPU的异构并行计算环境CUDA和OpenCL。

1.7.5 移动并行计算

目前高端的智能手机、平板使用多个ARM 核心和一个GPU ，越来越多的移动处理器支持使用OpenCL 进行并行计算，如何有效的利用这些资源进行计算也是一个值得研究的课题。

[8]目前GPU的访存带宽大约在200GB/s左右，CPU的访存带宽大约50GB/s，但是相比计算能力，GPU的带宽还是很小

1.8　如果摩尔定律失效

Intel在2003年曾经预测能够在2010年采用10纳米或更小的制作工艺、开发出30 GHz的计算机，实现万亿指令级别的性能（即每秒钟处理一万亿条指令）。但是现在Intel和其他的生产商在使用22 nm技术生产主频低于4 GHz的处理器[9]。现在看来5 GHz是硅技术的极限，虽然Intel和AMD的超频技术使得计算机的瞬间主频远远超过5 GHz，但是这不可持续且会降低硬件寿命。

由发热和能量消耗带来的不可预测的问题，已经对处理器的时钟频率或指令的处理速率造成了实际限制。硬件生产商进而采用多核技术，并且宣称多核技术延续了摩尔定律。多核技术解决了发热和能耗问题，但是这种解决方案也引来一个棘手的软件问题：如何编程以发挥多核的计算能力？

大多数程序设计时并没有考虑多核。多核需要一个全新的编程模型和环境。而且即使程序能够匹配双核或四核，但是将来出现数百核心时，编程还是一样吗？实际上多核可能只是一种过渡。由于应用和硬件中串行处理部分的存在，每增加一个处理器，并行系统的效率就会降低一些。对于某个应用，当使用的核心数量达到一定程度时，增加核心反而会减慢应用的速度，硬件上也存在这种问题。可能在制造几百核心的计算机前，多核可能已经达到了实际限制。

对于软件开发人员来说，一些问题很容易并行化，但是也有一些不行；有些问题适合并行，但是程序却不容易编写。很难将算法划分为几百几千个控制流，因为人脑很难维护这些控制流的状态空间。实际上自动并行化是解决这些问题的首选[10]，但是现在的自动并行化工具仍旧非常弱。

抽象是软件开发中的有效技术。但是，编写并行程序的时候，抽象就不太好用。

其实即使摩尔定律失效又有什么关系。尽管有些程序需要多核/多处理器的能力，但是大多数人不会需要它们。其实对不断增长的处理能力的需求绝大多数商业的[11]。从2003年到现在，主频并没有提升多少，但是又有多少人抱怨他们的机器太慢？摩尔定律的失效必将带来软件开发的复兴，因为那时才能更

[9] 2004年，时任Intel总裁贝瑞特因为没能制造出4 GHz的计算机，在电视节目中半真半假的向观众下跪

[10] 软件开发人员编写串行代码，编译器或硬件给多个处理器有效地分发指令

[11] 半导体和计算机公司、软件供应商和手机制造商需要卖出产品

关注软件实践而非硬件，才能最终创建在稳定持久的平台上运行的软件[12]。

房子的生存期是几十几百年，书本大约几年几十年，而一台计算机则大约三年。而这些设备中计算机的购买、维护费用最高并且使用期结束后还没有多少剩余价值，而且实际上计算机的更新换代不是因为它们不能用了，而是因为我们觉得它们过时了[13]。只有摩尔定律失效，计算机平台生存期足够长，软件产业才会得到充足的发展。

本节参考了以前摘录的某篇文献中的内容，由于我现在没有找到它的出处，在此向原作者表示抱歉。

[12]其实主频提升的停止导致了很多企业去优化他们的软件

[13]可是什么才是过时的标准，是因为它不是双核或四核？

第二章　整形与浮点数据

相邻整数之间的距离为1，这是精确的，因此可以精确表示，但是整数数目却是无穷的，要表示这无穷个数据，需要无数个二进制位。为了在存储量和大小之间均衡，编程语言都支持不同大小的整数数据类型。由于采用了有限的位数存储整数，其计算和现实中的整数计算不一样，本章会详细说明C语言中整数的表示方式以及其运算和现实运算的不同之处。

现代X86 处理器使用SSE 或x87 指令做浮点计算，为了可移植性，在32 位机器上编译器通常将浮点运算编译成x87 指令，而在64 位机器上则编译成SSE 指令。浮点表示具有不同于整形表示的特点，因为整形相邻数据之间的差异为1，而浮点数据的差异为无穷小，这就意味着要准确表示浮点数值，需要无穷二进制位，这并不现实。目前处理器普遍采用的浮点表示方法是IEEE-754标准，本章会详细的说明该标准的主要内容。

2.1　C语言数据类型

C语言中的整数分为有符号和无符号两种类别，有符号表示其值可以表示正数和负数，而无符号则只表示正数。C中的浮点数有单精度和双精度两种，除了精度之外，两者没有区别。表2.1列出了C语言支持数据类型及其在IA32和x86-64机器上所占据的字节数。

表 2.1: C语言的数据类型及在IA32和x86-64上位数

类型	IA32	x86-64
char	8	8
short	16	16
int	32	32
long	32	64
float	32	32
double	64	64
void*	32	64

2.2 整型运算

由于使用有限的位表示整型，因此其运算可能出现溢出，溢出的主要原因是运算结果改变了最高位或者超过了表示能力。和浮点表示不同的是：整型的表示是精确的，其满足分配律和结合律。

2.2.1 整型的补码表示

C语言中整数的表示有原码、反码和补码三种，三者在表示正数时没有区别，区别的只是如何表示负数。虽然C语言标准没有规定整数的编码标准，但是现在的编译器都使用补码表示，因此本文也只介绍补码表示。

对于一个二进制表示为$(x_{n-1}, x_{n-2}, ..., x_0)$、二进制长度为n 的整型数据，其补码表示的数据为：$-x_{n-1}2^{n-1} + \sum_{i=0}^{i=n-2} x_i 2^i$，其表示的无符号数据是：$\sum_{i=0}^{i=n-1} x_i 2^i$ 。对于有符号整数，通常称x_{n-1} 为符号位，其值为1 表示负数，为0 表示正数。

本节使用函数r表示C语言中某种运算的结果，而表达式表示无限精度运算结果。

2.2.2 加减

对于二进制长度为n位的两个数的加减运算，产生的结果可能是$n+1$位数据，而实际上C还是使用n位表示[1]，这就可能导致某些奇怪的行为。

无符号数的加减结果还是无符号数，但是两个无符号数的和可能小于两者中任何一个，两个无符号数减的结果可能大于被减数。无符号数加减的结果如式2.1所示。

$$r(x \pm y) = \begin{cases} x \pm y - 2^n & x \pm y \geq 2^n \\ x \pm y + 2^n & x \pm y < 0 \\ x \pm y & 0 < x \pm y < 2^n \end{cases} \tag{2.1}$$

[1]原因是提供无限精度的表示代价太大

两个正的有符号数和可能为负数，两个负的有符号数的和可能是正数。有符号数加减的结果如式2.2所示。

$$r(x \pm y) = \begin{cases} x \pm y - 2^n & x \pm y \geq 2^{n-1} \\ x \pm y + 2^n & x \pm y < -2^{n-1} \\ x \pm y & -2^{n-1} \leq x \pm y < 2^{n-1} \end{cases} \tag{2.2}$$

2.2.3　取负

由于补码表示的数不是对称的[2]，因此取负存在特例。无符号数取负却是正数，而且0是其特例。无符号数的负数如式2.3所示。

$$r(-x) = \begin{cases} 0 & x = 0 \\ 2^n - x \end{cases} \tag{2.3}$$

而有符号数的负数如式2.4所示。

$$r(-x) = \begin{cases} x & x = -2^{n-1} \\ -x \end{cases} \tag{2.4}$$

2.2.4　乘法

两个n位二进制的乘法结果最多可能为$2n$位，但是C只使用了其低n位而丢弃高n位，因此其结果可能和直观不一致。整数乘法的结果如式2.5所示。

$$r(x \times y) = (x \times y)\%2^n \tag{2.5}$$

对于有符号数的乘法来说，其运算相当于先将两个乘数的位表示解释成无符号数，再执行乘法，然后再将结果的位表示解释成有符号数，因此有符号数和无符号数的乘法可使用同一个指令。

整数乘以常整数时，可以将其转化为左移和加减法，因为通常乘法需要几个时钟周期而移位和加法一个周期能够处理多个。如$x * 8$可转化为$x << 3$，而$x * 9$可转化为$(x << 3) + x$。但是$x * 7$有两种表示方式：$(x << 3) - x$和$(x << 2) + (x << 1) + x$，很明显前一个更高效。

[2]最小的负数没有对应的正数表示

2.2.5 除法

C语言中，整数除法的结果还是整数，因此存在一个舍入的问题，C语言默认采用向零舍入，即(int)3.5 = 3, (int)-3.5 = -3。

除法运算需要的周期数远超乘法，通常要几百个。除以2的n次方时，可以将其转化为右移，但是有一点需要注意：除法是向零舍入，而右移是向下舍入，因此如果被除数是负数时，需要特殊处理。

```
int r = ((x > 0 ? x : (x+(1<<n)-1)) >> n);
```

2.2.6 移位

C语言中，有两种移位操作，一是左移，用符号<<表示，左移一位相当于乘以2，左移产生的右侧空位填充0，左移产生的进位会被忽略，其行为和乘法一致；二是右移，用符号>>表示，右移一位相当于除以2，对于右移产生的高位空位有两种处理方式，一种是填充0，这称为逻辑右移；一种是填充最高位，这称为算术右移。

关于在采用何种填充方式，现行的C标准没有对此做出规定，但是所有的编译器对此的处理表现出惊人的一致：对于无符号数使用逻辑右移，对于有符号数使用算术右移。这种处理能够保证移位和乘除法方便的进行转换。另外，右移是向下舍入，而除法是向零舍入的，关于向下舍入和向零舍入的概念，请参考2.3.4。

2.2.7 数据类型转换

C支持强制数据类型转换，由于强制类型转换可能会改变数据的表示，因此结果可能会发生变化。另外由于C支持隐式数据类型转换，就导致了许多潜在的错误，而且非常隐蔽，难以debug。下面列出了C中支持的数据类型转换类别。

- 大转小，此时会去掉高位，留下低位，如int转化为char，便只会留下最低8位。
- 小转大，存在两种高位填充模式：一种是填充0，无符号数大小转换采用这种；另一种是以符号位填充高位，有符号数转换采用这种方式，它能

够保证其数字结果和原来一致。这一点和编译器对右移的处理保持了一致。

- 有符号转无符号，此时会保持数据的位表示，只是对位做重新解释。

- 混合，同时有数据大小转换和符号转换的同时，C会先进行数据类型大小转换，然后再进行符号转换。如将int强制转换成unsigned short，会先将int转换成short，再将short转换成unsigned short。对于混合转换时，特别需要注意小数据转大数据的情况，尤其是同时执行有符号数转无符号数，如(unsigned int)(short -2)，这将会产生一个非常大的无符号数。

2.3　IEEE-754浮点格式

通常任一浮点数据x都可以表示成如式2.6所示：

$$x = (-1)^S \times M \times 2^E \tag{2.6}$$

其中S 表示x 的符号，等于1 或者0，因此只要一位便可；M 是一个二进制数，称为尾数，它的范围是$[0, 2)$；E 为阶码，可以是负数也可以是正数。

IEEE-754规定32位的单精度数据中E由8位表示，M由23位表示；而64位的双精度数据中E由11位表示，M由52位表示；半精度数据类型，其M由10位表示，E由5位表示。本文使用k 表示阶码的位数。

2.3.1　规格化数

如果E的二进制表示非全0或全1，表示规格化数。此时M的范围是$[1, 2)$，可以省略1的表示，称为隐含的1，因此实际的10/23/52位只表示小数部分，其第一位表示是否有0.5，第二位表示是否有0.25，其余类推。而此时E表示$e - bia$，e是E的二进制表示的无符号值，bia表示$2^{k-1} - 1$，其中k表示E的位数。

可以知道最小的正规格化数为：M全0，表示1.0；E的最低有效位为1，表示$2 - 2^{k-1}$，故最小的正规格化数为$2^{2-2^{k-1}}$。

2.3.2 非规格化数

如果E的二进制表示全0，这称为非规格化数。非规格化数的尾数M由其二进制小数表示（没有隐含的1），故其大小范围为$[0,1)$，而此时E大小为$2-2^{k-1}$，故最大的正非规格化数为：M全1，表示$1-2^{-n}$，n表示M的位数，值为$(1-2^{-n})*2^{2-2^{k-1}}$。可以看出最大的正非规格化数和最小的正规格化数非常接近。

非规格化数提供了一种表示非常小的数的方式，且使得这些小数近似对称的分布在0左右。IEEE的表示有一个非常好的优点：二进制相邻[3]表示的浮点数并不均匀，越靠近0，其二进制相邻表示的浮点数之间的距离越小，这就能够保证更好的精度。

现在处理器处理非规格化数时，有一个称为下溢的术语，它表示，一个存在但是无法表示的数被当成了0处理，下溢会造成精度损失。由于最小的非规格化数为: M最后一位为1，即2^{-n}，值为$2^{2-2^{k-1}-n}$，其绝对值非常小，虽然其绝对值非常小，但是下溢会损失精度，尤其是在参与运算的数绝对值大小差距明显时。

2.3.3 特殊值

当E 的二进制表示全1，M 全0 时，表示正负无穷大，无穷大表示数据类型不能表示但确实存在的数据；当E全1，而M非全0时，结果表示nan，意味着不能表示的数据，通常意味着除以0或无穷相关运算得到的结果。

怪异的是：有两种0表示法，一种是正0，一种是负0，对应着S 为0 或1。在标准看来这两种0是不同的。

2.3.4 舍入模式

由于IEEE-754采用有限位来表示浮点数据，因此有许多数据不能精确表示，此时需要采用舍入规则使用一个离它很近的值来表示。常用的舍入模式有：向上舍入、向下舍入、舍入到偶、向零舍入。

- 向上舍入是指将结果舍入到第一个比它大的可表示的数，类似于标准数学函数ceil，只是舍入的是最低有效位；

[3]即二进制x和x+1

- 向下舍入指将结果舍入到第一个比它小的可表示的数，类似于floor函数；
- 舍入到偶是指将结果向上或者向下舍入，使得结果的最低有效位为偶数；这是标准支持的默认舍入模式。
- 向零舍入是指最后的结果向0靠近，如果是负数就是向上舍入，如果是正数就是向下舍入。

将浮点数转型为整数采用的舍入模式是向零舍入。一些标准库为其某些函数提供了支持这些舍入模式的版本。

2.3.5　浮点运算特性

由于IEEE-754标准是近似的表示浮点数，因此其计算和符号计算有许多不同的地方。

- 整形和单精度浮点型之间转换会有精度损失，即使将单精度浮点转成8字节的整形，因为有些浮点值是整形无法表示的，同样有些整形也无法使用浮点值表示。大致说来，在C语言中，将整型转化为浮点会选择一个最接近整型的浮点可表示的浮点值。将浮点型强制转换为整型的法则是直接去掉小数即向零舍入，而不管浮点数本身是正是负，如果超出了整数的表示范围，则使用整数的最大或最小值。如果需要对十分位进行四舍五入，则其公式是：

$$x > 0 \quad ? \quad (int)(x + 0.5) \quad : \quad (int)(x - 0.5) \tag{2.7}$$

下面的代码展示了这一公式。

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    printf("(int)8.3=%d\n", (int)8.3);
    printf("(int)8.7=%d\n", (int)8.7);
    printf("(int)(8.3+0.5)=%d\n", (int)(8.3+0.5));
```

```
    printf("(int)(8.7+0.5)=%d\n", (int)(8.7+0.5));

    printf("(int)-8.3=%d\n", (int)-8.3);
    printf("(int)-8.7=%d\n", (int)-8.7);
    printf("(int)(-8.3+0.5)=%d\n", (int)(-8.3+0.5));
    printf("(int)(-8.7+0.5)=%d\n", (int)(-8.7+0.5));
    printf("(int)(-8.3-0.5)=%d\n", (int)(-8.3-0.5));
    printf("(int)(-8.7-0.5)=%d\n", (int)(-8.7-0.5));

    return 0;
}
/*
(int)8.3=8
(int)8.7=8
(int)(8.3+0.5)=8
(int)(8.7+0.5)=9
(int)-8.3=-8
(int)-8.7=-8
(int)(-8.3+0.5)=-7
(int)(-8.7+0.5)=-8
(int)(-8.3-0.5)=-8
(int)(-8.7-0.5)=-9
*/
```

- 将int转换为double没有精度损失，因为double能够表示每一个int。
- 将单精度转成双精度不会有损失，但是将双精度转化成单精度有损失，甚至可能产生无穷大。浮点(双精度和单精度)之间的转化与整型之间的转化的不同在于：浮点之间的转化尽量保证转化前后结果一致或相近，而没有符号位扩展的问题。
- 浮点加法和乘满足交换律，但是不满足结合律与分配律，在某些情况下，这阻止了编译器的优化。在必要的时候可以通过加括号来告诉编译器某

些操作可乱序或同时执行。如下面的代码中下一条就比上一条高效。

```
float x = a + b + c +d;
float x = (a+b) + (c+d)
```

2.3.6　不同条件下浮点运算结果

对于涉及到浮点运算的算法实现，直接拿不同硬件的结果一步一步地比较是不恰当的，因为即使在同一台机器上同一程序源码在不同的编译器上、不同的操作系统上、不同的编译选项上、串行和并行、结果都会有差别，这个问题的根本原因在于现行的IEE-754浮点标准，浮点加减乘不满足结合律和分配律，这种问题在操作数为非规格化数时更为明显。由于浮点运算不满足分配律和结合律，由不同的人实现的算法结果也可能有差别。通常这并不意味着程序是错误的，而是说结果的确存在着差异。

由于浮点运算存在误差，在某种对精确度要求非常严格的应用中，对误差的控制就非常有必要。