

5.15 谬误和陷阱

作为计算体系结构中的定量原则，存储器层次结构似乎不易受到谬误和陷阱的影响。但实际上却大相径庭，很多人不仅已经有了很多的谬误，还遭遇了陷阱，而且其中的一些还导致了很多负面的结果。下面以学生在练习和考试中经常遇到的陷阱开始讲解。

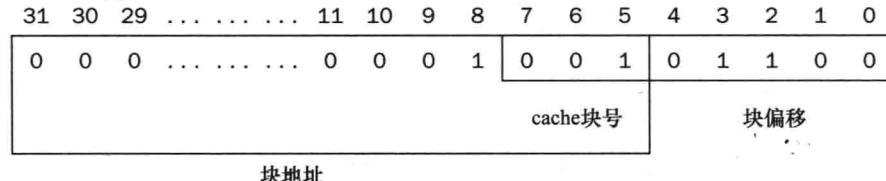
陷阱：在写程序或编译器生成代码时忽略存储系统的行为。

这可以很容易地写成一个谬误：“在写代码时，程序员可以忽略存储器层次”。图 5-19 中的排序和 5.14 节的 cache 分块技术证明了如果程序员在设计算法时考虑存储系统的行为，则可很容易地将性能翻倍。

陷阱：在模拟 cache 的时候，忘记说明字节编址或者 cache 块大小。

当模拟 cache 的时候（手动或者通过计算机），我们必须保证，在确定一个给定的地址被映射到哪个 cache 块中时，一定要说明字节编址和多字块的影响。例如，如果我们有一个容量为 32 字节的直接映射的 cache，块大小为 4 字节，则字节地址 36 映射到 cache 的块 1，因为字节地址 36 是块地址 9，而 $(9 \bmod 8) = 1$ 。另一方面，如果地址 36 是字地址，那么它就映射到块 $(36 \bmod 8) = 4$ 。要保证清楚地说明基准地址。

同样，我们必须说明块的大小。假设我们有一个 256 字节大小的 cache，块大小为 32 字节。那么字节地址 300 将落入哪一块中？如果我们将地址 300 划分成域，就能看到答案：



字节地址 300 是块地址

$$[300/32] = 9$$

cache 中的块数是

$$[256/32] = 8$$

块号 9 对应于 cache 块号 $(9 \bmod 8) = 1$ 。

许多人，包括作者（在早期的书稿中）和那些忘记自己预期的地址是字、字节或块号的教师们，都犯过这个错误。当你做练习时一定要注意这个易犯的错误。

陷阱：对于共享 cache，组相联度少于核的数量或者共享该 cache 的线程数。

如果不特别注意，一个运行在 2^n 个处理器或者线程上的并行程序为数据结构分配的地址可能映射到共享二级 cache 同一个组中。如果 cache 至少是 2^n 路组相联，那么通过硬件可以隐藏这些程序偶尔发生的冲突。如果不是，程序员可能要面对明显不可思议的性能缺陷——事实上是由于二级 cache 冲突缺失引起的——在程序迁移时发生，假定从一个 16 核的机器迁移到一个 32 核的机器上，并且如果它们都使用 16 路组相联的二级 cache。

陷阱：用存储器平均访问时间来评估乱序处理器的存储器层次结构。

如果处理器在 cache 缺失时阻塞，那么你可以分别计算存储器阻塞时间和处理器执行时间，因此可以使用存储器平均访问时间来独立地评估存储器层次结构（见 5.4 节第 2 个例子）。

如果处理器在 cache 缺失时继续执行指令，而且甚至可能维持更多的 cache 缺失，那么唯一可以用来准确评估存储器层次结构的办法是模拟乱序处理器和存储器结构。

陷阱：通过在未分段地址空间的顶部增加段来扩展地址空间。

在 20 世纪 70 年代，许多程序都变得很大，以至于不是所有的代码和数据都能仅用 16 位地址寻址。于是，计算机修改为 32 位地址，一种方法是直接使用未分段的 32 位地址空间（也称为平面地址空间），另一种方法是给已经存在的 16 位地址再增加 16 位长度的段。从市场观点来看，增加程序员可见的段，并且迫使程序员和编译器将程序划分成段，这样可以解决寻址问题。但遗憾的是，任何时候，一种程序设计语言要求的地址大于一个段的范围就会有麻烦，比如说大数组的索引、无限制的指针或者是引用参数。此外，增加段可以将每个地址变成两个字——一个是段号，另一个是段内偏移——这些在使用寄存器中地址时就会出现问题。

谬误：实际的磁盘故障率和规格书中声明的一致。

最近的两项研究评估了大量磁盘，目的是检查实际结果和规格之间的关系。其中一项研究了将近 100 000 个磁盘，他们声称其 MTTF 为 1 000 000 ~ 1 500 000 小时或者说具有 0.6% ~ 0.8% 的 AFR。他们发现 2% ~ 4% 的 AFR 是常见的，通常比设定的故障率高 3 ~ 5 倍 [Schroeder 和 Gibson, 2007]。另一项研究了 100 000 个磁盘，这些磁盘声称具有 1.5% 的 AFR，以及在第一年中，磁盘故障率为 1.7%，到第三年，磁盘的故障率上升到 8.6%，也就是说，大约是规格书中指定的故障率的 6 倍之多 [Pinheiro、Weber 和 Barroso, 2007]。

谬误：操作系统是调度磁盘访问的最好地方。

如 5.2 节所提到的，以高层接口为宿主操作系统提供逻辑块地址。假设在这样的高层抽象层 OS 可以通过将逻辑块的地址按照递增的顺序排序以得到最好的性能。然而，由于磁盘知道逻辑地址被映射到扇区、磁道上以及磁面上的实际物理地址，这样通过调度就可以减少旋转以及寻道的时间。

例如，假设以下工作负载是 4 个读操作 [Anderson, 2003]：

操作	LBA 的起始地址	长度
读	724	8
读	100	16
读	9 987	1
读	26	128

宿主 OS 可能对 4 个读操作重新进行调度，编排成逻辑块的读操作的顺序：

操作	LBA 的起始地址	长度
读	26	128
读	100	16
读	724	8
读	9 987	1

依赖于数据在磁盘中的相对位置，如图 5-50 所示，重新编排 I/O 顺序可能会使情况变得更糟。磁盘调度的读操作在磁盘的 3/4 的旋转周期就全部完成，而操作系统调度的读操作花费了 3 个旋转周期。

陷阱：在不为虚拟化设计的指令集体系结构上实现虚拟机监视器。

在 20 世纪 70 年代和 80 年代，很多计算机体系结构设计者并没有刻意去保证所有读写相关的硬件资源指令都是特权指令。这种放任的态度导致了 VMM 在这些体系结构上存在问题，包括 x86，这里我们就以它为例。

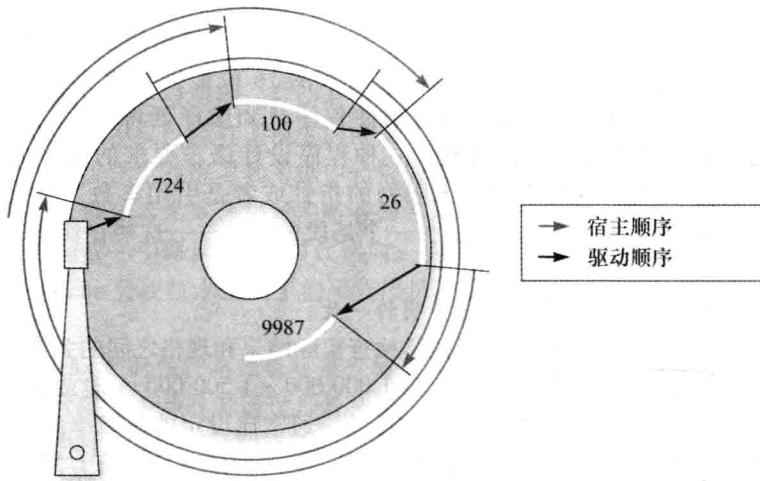


图 5-50 OS 调度与磁盘调度访问的例子，标记为宿主顺序和驱动顺序。前者完成 4 个读操作需要 3 个旋转周期，而后者完成 4 个读操作仅仅在一个 $3/4$ 的旋转周期即可完成（资料来源：Anderson[2003]）

图 5-51 指出了虚拟化产生问题的 18 条指令 [Robin 和 Irvine, 2000]。其中两大类指令是：

- 在用户模式下读控制寄存器，暴露了在虚拟机上运行的 guest 操作系统（如前面提到的 POPF）。
- 检查分段的体系结构所需的保护，但却假设操作系统在最高的特权级运行。

问题种类	x86 的问题指令
当运行在用户模式时，访问敏感寄存器无须 trap 中断	存储全局描述符表寄存器 (SGDT) 存储局部描述符表寄存器 (SLDT) 存储中断描述符表寄存器 (SIDT) 存储机器状态字 (SMSW) 标志入栈 (PUSHF, PUSHFD) 标志出栈 (POPF, POPFD)
在用户模式下访问虚拟存储机制时，x86 保护检查指令失效	从段描述符读取访问权限 (LAR) 从段描述符读取段的边界 (LSL) 如果段描述符可读，进行读校验 (VERR) 如果段描述符可写，进行写校验 (VERW) 段寄存器出栈 (POP CS, POP SS, ...) 段寄存器入栈 (PUSH CS, PUSH SS, ...) 远调用不同的特权级 (CALL) 远返回至不同的特权级 (RET) 远跳转至不同的特权级 (JMP) 软中断 (INT) 存储段选择寄存器 (STR) 移入/移出段寄存器 (MOVE)

图 5-51 虚拟化产生问题的 18 条 x86 指令的概述 [Robin 和 Irvine, 2000]。上面一组的前 5 条指令允许程序在用户模式下读控制寄存器，而无须 trap 中断，例如描述符表寄存器。标记出栈指令会修改包含敏感信息的控制寄存器，但在用户模式下将失效而无任何提示。x86 体系结构中段的保护检查在下面的一组指令中，当读取控制寄存器时，作为指令执行的一部分，都会隐式地检查特权级。进行检查时操作系统必须运行在最高特权级，但是对客户虚拟机并没有这样的要求。只有在移入段寄存器操作时会试图修改控制状态，但是，保护检查同样会阻止它这么做

为了简化在 x86 上实现 VMM，AMD 和 Intel 都提出通过新的模式扩展体系结构。Intel 的 VT-x 为虚拟机运行提供了一个新的执行模式、一个面向虚拟机状态的体系结构定义、快速虚拟机切换指令，以及一大组用来选择调入 VMM 环境的参数。总之，VT-x 在 x86 中加了 11 条新指令。AMD 的 Pacifica 做了相似的改进。

另一种方法通过修改硬件来对操作系统做细微的修改以简化虚拟化。这种技术称为泛虚拟化（paravirtualization），例如开源的虚拟机监视器 Xen 就是一个很好的例子。Xen 虚拟机监视器提供给客户操作系统一个抽象虚拟机，它仅仅使用了供虚拟机监视器运行的 x86 物理硬件中易于虚拟化的一部分。

5.16 本章小结

无论在最快的计算机还是最慢的计算机中，构成主存的原材料——DRAM 本质是相同的，并且是最便宜的，这使得构建一个和快速处理器保持同步的存储系统变得更加困难。

局部性原理可以用来克服存储器访问的长延迟——这个策略的正确性已经在存储器层次结构的各级都得到了证明。尽管层次结构中的各级从量的角度来看非常不同，但是在它们的执行过程中都遵循相似的策略，并且利用相同的局部性原理。

多级 cache 可以更方便地使用更多的优化，这有两个原因。第一，较低级 cache 的设计参数与一级 cache 不同。例如，由于较低级 cache 的容量一般很大，因此可能使用更大容量的块。第二，较低级 cache 并不像一级 cache 那样经常被处理器用到。这让我们考虑当较低级 cache 空闲时让它做一些事情以预防将来的缺失。

另一个趋势是寻求软件的帮助。使用大量的程序转换和硬件设备有效地管理存储器层次结构是增强编译器作用的主要焦点。现在有两种不同的观点。一种是重新组织程序结构以增强它的空间和时间局部性。这种方法主要针对以大数组为主要数据结构的面向循环的程序；大规模的线性代数问题就是一个典型的例子，例如 DGEMM。通过重新组织访问数组的循环增强了局部性——也因此改进了 cache 性能。

还有一种方法是预取（prefetching）。在预取机制中，一个数据块在真正被访问之前就被放入 cache 中了。许多微处理器使用硬件预取尝试预测访问，这对软件可能比较困难。

预取：使用特殊指令将未来可能用到的指定地址的 cache 块提前搬到 cache 中的一种技术。

第三种方法是使用优化存储器传输的特殊 cache 感知（cache-aware）指令。例如，在第 6 章的 6.10 节中，微处理器使用了一个优化设计：当发生写缺失时，由于程序要写整个块，因而并不从主存中取回一个块。对于一个内核来说，这种优化明显减少了存储器的传输。

我们将在第 6 章中看到，对并行处理器来说，存储系统也是一个重要的设计问题。存储器层次结构决定系统性能的重要性在不断增长，这也意味着在未来的几年内，这一领域对设计者和研究者来说将成为焦点。

5.17 历史观点和拓展阅读

本节网站中的内容描述了存储器技术的概况，从汞延迟线到 DRAM，存储器层次结构的发明，保护机制以及虚拟机，最后以操作系统的一个简单发展历史作总结，包括 CTSS、MULTICS、UNIX、BSD UNIX、MS-DOS、Windows 和 Linux。

5.18 练习题

- 5.1 在这个练习中，考虑矩阵计算中存储器的局部特性。下面的代码由 C 语言编写，在同一行中的元素被连续存放。假定每个字是 32 位整数。

480
481

482

```

for (I=0; I<8; I++)
    for (J=0; J<8000; J++)
        A[I][J]=B[I][0]+A[J][I];

```

5.1.1 [5] <5.1> 16 字节的 cache 块中可以存放多少 32 位的整数?

5.1.2 [5] <5.1> 访问哪些变量会显示出时间局部性?

5.1.3 [5] <5.1> 访问哪些变量会显示出空间局部性?

局部性同时受访问顺序和数据存放位置的影响。在同一列的矩阵元素是连续存放的情况下，同样的计算机也可以用下面的不同于 C 的 Matlab 来写。

```

for I=1:8
    for J=1:8000
        A(I,J)=B(I,0)+A(J,I);
    end
end

```

483

5.1.4 [10] <5.1> 存放全部将被访问的 32 位矩阵元素需要多少 16 字节的 cache 块?

5.1.5 [5] <5.1> 访问哪些变量会显示出时间局部性?

5.1.6 [5] <5.1> 访问哪些变量会显示出空间局部性?

5.2 cache 为处理器提供了一个高性能的存储器层次结构，因此十分重要。下面是一个 32 位存储器地址引用的列表，给出的是字地址。

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

5.2.1 [10] <5.3> 已知一个直接映射的 cache，有 16 个块，块大小为 1 个字。对于每次访问，请标识出二进制地址、标记以及索引。假设 cache 最开始为空，那么请列出每次访问是命中还是缺失。

5.2.2 [10] <5.3> 已知一个直接映射的 cache，有 8 个块，块大小为 2 个字。对于每次访问，请标识出二进制地址、标记以及索引。假设 cache 最开始为空，那么请列出每次访问是命中还是缺失。

5.2.3 [20] <5.3, 5.4> 对已知的访问来优化 cache 的设计。这里有三种直接映射的 cache 设计方案，每个容量都为 8 个字：C1 块大小为 1 个字，C2 块大小为 2 个字，C3 块大小为 4 个字。根据缺失率，哪种 cache 设计最好？如果缺失阻塞时间为 25 个周期，C1 的访问时间为 2 个周期，C2 为 3 个周期，C3 为 5 个周期，那么哪种 cache 设计最好？

这里有许多对 cache 整体性能很重要的不同的设计参数。下面列出了对于不同的直接映射 cache 设计的参数。

cache 数据量：32KiB

cache 块大小：2 个字

cache 访问时间：1 个周期

5.2.4 [15] <5.3> 假定 32 位的地址，计算上面列出的 cache 所需的总位数。给定总的大小，找出最接近的直接映像 cache 的总的大小，该 cache 块的大小为 16 个字长或更大。请解释为什么第二种 cache 比第一种 cache 的访问速度更慢，尽管第二种 cache 的数据量更大。

5.2.5 [20] <5.3, 5.4> 在一个 2KiB 的两路组相联 cache 上产生一系列读请求时的缺失率要比在表中 cache 上执行读请求的缺失率低。请给出一个可能的解决方案，使得表中列出的 cache 的缺失率等于或者低于 2KiB cache 的缺失率。讨论这种解决方案的优点和缺点。

5.2.6 [15] <5.3> 5.3 节的公式说明了用来索引直接映射 cache 的典型方法：(块地址) mod (cache 中的块数)。假设地址为 32 位，cache 中有 1024 个块，考虑一个不同的索引函数：(块地址 [31: 27] XOR 块地址 [26: 22])。可以使用这个公式来索引直接映射的 cache 吗？如果可以，请解释原因，并且讨论可能需要对 cache 做的一些改动。如果不可以，请解释原因。

5.3 对于一个 32 位地址的直接映射的 cache 设计，下面的地址位用来访问 cache。

标记	索引	偏移量
31 ~ 10	9 ~ 5	4 ~ 0

484

5.3.1 [5] <5.3> cache 块大小是多少（单位为字）？

5.3.2 [5] <5.3> cache 有多少项？

5.3.3 [5] <5.3> 这样的 cache 执行时所需的总位数与数据存储位数之间的比率是多少？

下表记录了从上电开始的 cache 访问的字节地址。

地址											
0	4	16	132	232	160	1 024	30	140	3 100	180	2 180

5.3.4 [10] <5.3> 有多少块被替换？

5.3.5 [10] <5.3> 命中率是多少？

5.3.6 [20] <5.3> 列出 cache 的最终状态，每个有效项以记录的形式 <索引，标记，数据> 表示出来。

5.4 回忆一下两个写策略和写分配策略，它们结合起来既可以在一级 cache 中执行，也可以在二级 cache 中执行。假定一级和二级 cache 如下：

一级 cache	二级 cache
写直达，写不分配	写回，写分配

5.4.1 [5] <5.3, 5.8> 在存储器层次结构中的不同层使用缓冲器来降低访问延迟。对这个给定的配置，列出一级 cache 与二级 cache 之间，以及二级 cache 与存储器之间可能需要的缓冲器。

5.4.2 [20] <5.3, 5.8> 描述处理一级 cache 写缺失的过程，考虑里面的组件以及替换一个脏块的可能性。

5.4.3 [20] <5.3, 5.8> 对于一个多级独占 cache（一个块只能存放在一个 cache 中，或者在一级 cache 中，或者在二级 cache 中）配置，描述处理一级 cache 写缺失的过程，考虑到里面的组件以及替换一个脏块的可能性。

考虑下面的方案和 cache 行为。

485

每 1000 条指令中 数据读的次数	每 1000 条指令中 数据写的次数	指令 cache 缺失率	数据 cache 缺失率	块大小（字节）
250	100	0.30%	2%	64

5.4.4 [5] <5.3, 5.8> 对于一个使用写直达法、写分配策略的 cache，如果 CPI 为 2，所需最小的读/写带宽是多少（以每周期字节数来度量）？

5.4.5 [5] <5.3, 5.8> 对于一个使用写回法、写分配策略的 cache，假定 30% 被替换的数据块为脏块，那么如果 CPI 为 2，所需最小的读/写带宽是多少？

5.4.6 [5] <5.3, 5.8> 如果要实现 $CPI = 1.5$ 的性能，所需的最小带宽是多少？

5.5 播放音频或视频文件的多媒体应用是一类被称为“流”的负载的一部分；即它们取回大量的数据，但是大部分数据都不会再使用。考虑一个视频流负载依次访问一个 512KiB 的工作集的情况，地址流如下：

0, 2, 4, 6, 8, 10, 12, 14, 16 ...

5.5.1 [5] <5.4, 5.8> 假设有一个 64KiB 的直接映射 cache，cache 块大小为 32 字节。那么对于上面的地址流，缺失率是多少？当 cache 大小或者工作集变化时，cache 的缺失率如何随之变化？根据 3C 模型，这些缺失如何被分类？

5.5.2 [5] <5.1, 5.8> 当 cache 块大小分别为 16 字节、64 字节和 128 字节时，重新计算缺失率。该负载所采用的是哪种局部性？

5.5.3 [10] <5.13> “预取”是一种技术：当一个特殊 cache 块被访问时，预测地址模式并推测地召回其他 cache 块。预取的一个例子是流缓冲区，当一个特定的 cache 块被取回时，将与其相邻的 cache 块也依次预取回到一个独立的缓冲区中。如果所需的数据在预取缓冲区中，那么看成是一

次命中并且将数据移入 cache，同时预取下一个 cache 块。假设一个流缓冲区有两项，并且假设 cache 延迟满足：在先前 cache 块的计算完成之前可以加载下一个 cache 块。那么对于上面的地址流，缺失率是多少？

cache 块大小 (B) 影响了缺失率和缺失延迟。假设有下面的缺失率表，并假定 CPI 为 1 的机器中，每条指令平均访问次数（指令和数据）为 1.35，给定不同容量的 cache 的缺失率，请找出最优的 cache 块大小。

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%
-------	--------	--------	----------	---------

5.5.4 [10] <5.3> 缺失延迟为 $20 \times B$ 个周期时，最佳的块大小是多少？

5.5.5 [10] <5.3> 缺失延迟为 $24 + B$ 个周期时，最佳的块大小是多少？

486 **5.5.6** [10] <5.3> 缺失延迟为恒定值时，最佳的块大小是多少？

5.6 在这个练习中，我们将研究不同容量对整体性能的影响。通常来说，cache 访问时间与 cache 容量成正比。假设访问主存需要 70ns，并且在所有指令中，有 36% 的指令需要访存。下表是 P1 和 P2 两个处理器各自的一级 cache 的数据。

	一级 cache 容量	一级 cache 缺失率	一级 cache 命中时间
P1	2KiB	8.0%	0.66ns
P2	4KiB	6.0%	0.90ns

5.6.1 [5] <5.4> 假定一级 cache 命中时间决定了 P1 和 P2 的周期时间，它们各自的时钟频率是多少？

5.6.2 [5] <5.4> P1 和 P2 各自的 AMAT（平均存储器访问时间）分别是多少？

5.6.3 [5] <5.4> 假定在没有任何存储器阻塞时基本的 CPI 为 1.0，P1 和 P2 各自的总 CPI 分别是多少？哪个处理器更快？

对下面的三个问题，我们考虑在 P1 中增加二级 cache，以弥补一级 cache 容量的限制。在解决这些问题时，依然使用上表中一级 cache 的容量和命中时间。二级 cache 缺失率是它的局部缺失率。

二级 cache 容量	二级 cache 缺失率	二级 cache 命中时间
1 MiB	95%	5.62ns

5.6.4 [10] <5.4> 增加二级 cache 后，P1 的 AMAT 是多少？有了二级 cache，AMAT 是更好还是更差了？

5.6.5 [5] <5.4> 假定在没有任何存储器阻塞时基本的 CPI 为 1.0，增加二级 cache 后，P1 的总的 CPI 是多少？

5.6.6 [10] <5.4> P1 有了二级 cache 后，哪个处理器更快？如果 P1 更快，那么 P2 中一级 cache 的缺失率需要为多少才能匹配 P1 的性能？如果 P2 更快，那么 P1 中一级 cache 的缺失率需要为多少才能匹配 P2 的性能？

5.7 这个练习研究了不同 cache 设计的效果，特别将关联的 cache 与 5.4 节中的直接映射的 cache 进行比较。练习中使用的是练习题 5.2 中的地址流。

5.7.1 [10] <5.4> 使用练习题 5.2 中的访问信息，对于一个 3 路组相联、块大小为 2 个字、总容量为 24 个字、使用 LRU 替换算法的 cache，指出 cache 中最终的内容。对每个访问，标识出索引位、标记位、块偏移位，以及当前访问是命中还是缺失。

5.7.2 [10] <5.4> 使用练习题 5.2 中的访问信息，对于一个全相联、块大小为 1 个字、总容量为 8 个字、使用 LRU 替换算法的 cache，指出 cache 中最终的内容。对每个访问，标识出索引位、标记位，以及当前访问是命中还是缺失。

5.7.3 [15] <5.4> 使用练习题 5.2 中的访问信息，对于一个全相联、块大小为 2 个字、总容量为 8 个字、使用 LRU 替换算法的 cache，请问缺失率是多少？如果替换为 MRU（最近最常使用）算法，

缺失率是多少？在这些替换策略下，最好的情况下，cache 缺失率是多少？

多级 cache 是一项重要技术，它在克服一级 cache 提供的有限空间的同时仍然保持了速度。假设一个处理器的参数如下：

没有存储器阻塞的基本 CPI	处理器速度	主存访问时间	每条指令的一级 cache 缺失率	直接映射的二级 cache 的速度	包含直接映射的二级 cache 时的全局缺失率	8 路组相联的二级 cache 的速度	包含 8 路组相联的二级 cache 时的全局缺失率
1.5	2GHz	100ns	7%	12 个周期	3.5%	28 个周期	1.5%

5.7.4 [10] <5.4> 计算表中处理器的 CPI：1) 只有一级 cache；2) 一个直接映射的二级 cache；3) 一个 8 路组相联的二级 cache。如果主存访问时间加倍，CPI 如何变化？如果主存访问时间减半，CPI 又如何变化？

5.7.5 [10] <5.4> 拥有比两级 cache 更多的 cache 层次是可能的。已知上述的处理器拥有一个直接映射的二级 cache，一个设计者希望增加一个三级 cache，其访问时间为 50 个周期，并且全局缺失率降低到 1.3%。这种设计能提供更好的性能吗？通常来说，增加一个三级 cache 的优点和缺点分别是什么？

5.7.6 [20] <5.4> 在以前的处理器中，如 Intel Pentium 或 Alpha 21264，二级 cache 在远离主处理器和一级 cache 的片外（放置在不同的芯片上）。这使得二级 cache 很大，访问延迟也高得多，同时由于二级 cache 以较低的频率运行，带宽通常也较低。假设一个 512KiB 的片外二级 cache 的全局缺失率为 4%。如果 cache 每增加 512KiB 容量可以降低 0.7% 的全局缺失率，并且 cache 总的访问时间为 50 个周期，那么 cache 容量为多大时才能匹配表中直接映射的二级 cache 的性能？如果匹配表中 8 路组相联的 cache 性能，cache 容量又需要是多少？

5.8 失效时间间隔（mean time between failure, MTBF）、替换时间（mean time to replacement, MTTR）和无故障工作时间（mean time to failure, MTTF）对于评估一个存储资源的可靠性和可用性非常有用。通过使用如下参数回答下列问题：

MTTF	MTTR
3 年	1 天

488

5.8.1 [5] <5.5> 计算表中每台设备的 MTBF。

5.8.2 [5] <5.5> 计算表中每台设备的可用性。

5.8.3 [5] <5.5> 如果 MTTR 接近于 0，则可用性如何变化？这是一个合理的情形吗？

5.8.4 [5] <5.5> 如果 MTTR 非常高，例如一台设备非常难维修，则可用性如何变化？这是否意味着该设备可用性很低？

5.9 本练习题检查纠正一位错检测两位错（SEC/DED）的汉明码。

5.9.1 [5] <5.5> 如果要对 128 位字采用 SEC/DED 编码进行保护，最少需要多少位的奇偶位？

5.9.2 [5] <5.5> 5.5 节指出，现代服务器存储器模块（DIMM）对于 64 位数据，使用 8 位奇偶位来实现 SEC/DEC ECC。计算该编码的开销/性能比，并与 5.9.1 进行比较。在这里开销是指所需的相对的奇偶位，性能是指相对的能够纠正的错误数量。哪种编码比较好？

5.9.3 [5] <5.5> 考虑一个采用 4 位奇偶位来保护 8 位字的 SEC。如果读出值为 0x375，是否有错？如果有错，对错误进行纠正。

5.10 对于一个高性能系统如 B-tree 索引数据库，页的大小主要由数据量和磁盘性能决定。假设一个 B-tree 索引页（项数固定）使用了 70%。使用的页就是 B-tree 的深度，用 \log_2 （项数）来计算。下表显示的是 10 年前的一个拥有 16 字节表项的磁盘，延迟为 10ms，传输率为 10MB/s，最优的页大小是 16K。

页大小 (KiB)	页的使用/B-tree 深度 (保存的磁盘访问次数)	索引页的访问开销 (ms)	效用/代价
2	6.49 或 $\log_2(2048/16 \times 0.7)$	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

5.10.1 [10] <5.7> 如果项数为 128 字节，最佳的页大小是多少？

5.10.2 [10] <5.7> 根据练习题 5.10.1，如果页处于半满状态，最佳的页大小是多少？

5.10.3 [20] <5.7> 根据练习题 5.10.2，如果使用的是最新的磁盘，延时 3ms，而传输率为 100MB/s 时，最佳的页大小是多少？请解释为什么未来的服务器可能用较大的页？

489

在 DRAM 保存“频繁使用”的页（“热门”的页）可以避免磁盘访问，但是对于一个系统，我们如何判断“频繁使用”的精确含义？数据工程师利用 DRAM 和磁盘访问之间的开销比率对频繁使用页的重用时间阈值进行量化。磁盘访问的开销是 \$Disk/accesses_per_sec，将页保存在 DRAM 中的开销是 \$DRAM_MiB/page_size。在某些年代中，典型的 DRAM 和磁盘开销、典型的数据库页大小如下表所示：

年代	DRAM 开销 (\$/MiB)	页大小 (KiB)	磁盘开销 (\$/disk)	磁盘访问率 (访问/秒)
1987	5 000	1	15 000	15
1997	15	8	2 000	64
2007	0.05	64	80	83

5.10.4 [10] <5.1, 5.7> 对于这三种技术时代，重用时间阈值是多少？

5.10.5 [10] <5.7> 如果我们保持使用相同的 4K 大小的页，重用时间阈值是多少？这里趋势是什么？

5.10.6 [20] <5.7> 为了保持使用相同的页大小（因此避免软件重写），其他方面应该如何变化？讨论与当前技术和成本趋势的相似性。

5.11 如 5.7 节所述，虚拟存储器使用一个页表来追踪虚拟地址到物理地址之间的映射。这个练习说明了当地址被访问时，页表如何更新。下表是在一个系统上所看见的虚拟地址流。假设 4KiB 的页，一个 4 项的全相联 TLB，使用严格的 LRU 替换算法。如果必须从磁盘中取回页，那么增加下一次能取的最大页数：

4669, 2227, 13916, 34587, 48870, 12608, 49225

TLB

有效位	标记位	物理页号
1	11	12
1	7	4
1	3	6
0	4	9

490

页表

有效位	物理页/在磁盘上	有效位	物理页/在磁盘上
1	5	0	磁盘
0	磁盘	1	4
0	磁盘	0	磁盘
1	6	0	磁盘
1	9	1	3
1	11	1	12

5.11.1 [10] <5.7> 根据给出的地址流，以及 TLB、页表的初始状态，请给出系统的最终状态。对于每次访问，请列出是否在 TLB 中命中，是否在页表中命中或是发生缺页。

5.11.2 [15] <5.7> 重复练习题 5.11.1，但是这次使用 16KiB 的页来代替 4KiB 的页。使用更大的页的好处有哪些？缺点又有哪些？

5.11.3 [15] <5.4, 5.7> 如果使用两路组相联的 TLB，请指出 TLB 中最终的内容。如果 TLB 是直接映射的，同样指出 TLB 中最终的内容。讨论使用 TLB 来获得高性能的重要性。如果没有 TLB，那么如何处理虚拟存储器访问？

有一些参数对页表整个大小会有影响。下面就列出一些关键的页表参数。

虚拟地址位数	页的大小	页表项的大小
32 位	8KiB	4 字节

5.11.4 [5] <5.7> 已知上表中的参数，一个系统用了一半的内存来运行 5 个应用程序，计算该系统使用的页表总大小。

5.11.5 [10] <5.7> 已知上表中的参数，一个系统用了一半的内存来运行 5 个应用程序，给定一个两级的有 256 项的页表，计算该系统使用的页表总大小。假设主页表中每项是 6 字节，计算所需的最小和最大的内存容量。

5.11.6 [10] <5.7> 一名 cache 设计者要将一个 4KiB 的虚拟索引、物理标记的 cache 的容量增大，已知页的大小在上表中列出，那么能否构建一个 16KiB 的直接映射 cache，假设块大小为 2 个字？设计者如何增加 cache 的数据大小？

5.12 在这个练习中，我们将研究对页表进行空间/时间的优化。下表是一个虚拟存储器系统的参数。

虚拟地址 (位)	物理 DRAM	页大小	PTE 大小 (字节)
43	16GiB	4KiB	4

5.12.1 [10] <5.7> 对于一个单级页表，需要多少页表项 (PTE)？存放页表需要多少物理存储器？

5.12.2 [10] <5.7> 使用多级页表可以降低物理存储器中页表的消耗，它在物理存储器中只保存活跃的 PTE。这种情况下，需要多少级的页表？如果 TLB 缺失，地址转换需要访问多少次存储器？

5.12.3 [15] <5.7> 反置页表可以用来进一步优化空间和时间。存放页表需要多少 PTE？假设执行一个哈希表，当 TLB 缺失时，在正常情况下和最差情况下的存储器访问次数分别是多少？

下表是一个有 4 项内容的 TLB：

入口 ID	有效位	虚拟地址页	修改位	保护位	物理地址页
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

- 5.12.4** [5] <5.7> 在什么样的情况下第二项的有效位被置为 0?
- 5.12.5** [5] <5.7> 当一条指令写入虚拟地址页号为 30 处时，会发生什么情况？什么时候软件管理的 TLB 比硬件管理的 TLB 速度快？
- 5.12.6** [5] <5.7> 当一条指令写入虚拟地址页 200 时，会发生什么情况？
- 5.13** 在这个练习中，我们将研究替换策略是如何影响缺失率的。假设一个有 4 个块的 2 路组相联 cache。你会发现画表对于解决练习题中的问题很有帮助，如下面的示范，地址序列为“0, 1, 2, 3, 4”。

被访问的主存块的地址	命中/缺失	逐出的块	cache 块的内容			
			组 0	组 0	组 1	组 1
0	缺失		主存 [0]			
1	缺失		主存 [0]		主存 [1]	
2	缺失		主存 [0]	主存 [2]	主存 [1]	
3	缺失		主存 [0]	主存 [2]	主存 [1]	主存 [3]
4	缺失	0	主存 [4]	主存 [2]	主存 [1]	主存 [3]
...						

492

考虑如下的地址序列：0, 2, 4, 8, 10, 12, 14, 16, 0。

- 5.13.1** [5] <5.4, 5.8> 假定使用 LRU 替换策略，在这组地址序列中有多少次命中？
- 5.13.2** [5] <5.4, 5.8> 假定使用 MRU（最近最常使用）替换策略，在这组地址序列中有多少次命中？
- 5.13.3** [5] <5.4, 5.8> 通过掷硬币来模拟随机替换策略。例如，“正面”表示逐出组中第一块，“反面”表示逐出组中第二块。在这组地址序列中有多少次命中？
- 5.13.4** [10] <5.4, 5.8> 为了最大化命中次数，每次替换时哪个块应该被逐出？如果使用了这个“最优的”策略，在这组地址序列中有多少次命中？
- 5.13.5** [10] <5.4, 5.8> 请说明为什么实现这种对所有地址序列来说都是最优的 cache 替换策略很难？
- 5.13.6** [10] <5.4, 5.8> 假设在每次主存引用时，可以决定被请求的地址是否要被缓存，这对缺失率有什么影响？
- 5.14** 为了支持多虚拟机，需要对两级存储器进行虚拟化。每个虚拟机依然控制从虚拟地址 (VA) 到物理地址 (PA) 之间的映射，同时管理程序将每个虚拟机的物理地址 (PA) 映射到实际的机器地址 (MA)。为了加速映射过程，一种被称为“影子分页”(shadow paging) 的软件方法在管理程序中复制了每个虚拟机的页表，并且侦听从虚拟地址到物理地址的映射变化，以保证两个副本的一致性。为了消除影子页表 (shadow page table) 的复杂性，一种被称为嵌套页表 (nested page table, NPT) 的硬件方法可以支持两种页表 (VA \Rightarrow PA 和 PA \Rightarrow MA)，并且完全依靠硬件来查找这些表。
- 考虑下面的操作序列：(1) 创建进程；(2) TLB 缺失；(3) 缺页；(4) 上下文切换。
- 5.14.1** [10] <5.6, 5.7> 对于给定的操作序列，对影子页表和嵌套页表分别会发生什么情况？
- 5.14.2** [10] <5.6, 5.7> 假设一个基于 x86 架构的 4 级页表同时存放在客户页表 (guest page table) 和嵌套页表中，那么在处理本地页表 (native page table) TLB 缺失和嵌套页表 TLB 缺失时，分别需要多少次存储器访问？
- 5.14.3** [15] <5.6, 5.7> 在 TLB 缺失率、TLB 缺失延迟、缺页率、缺页处理延迟之间，对影子页表来说，哪些度量标准更重要？而对于嵌套页表来说，哪些度量标准更重要？

下表是影子分页系统的参数。

每 1 000 条指令 TLB 缺失数	嵌套页表 TLB 缺失延迟	每 1 000 条指令 缺页数	影子页表缺失代价
0.2	200 个时钟周期	0.001	30 000 个时钟周期

493

- 5.14.4** [10] <5.6>一个基准测试程序的本地执行 CPI 为 1，如果使用影子页表，CPI 是多少？如果使用嵌套页表（假设只有页表虚拟化开销），CPI 是多少？
- 5.14.5** [10] <5.6> 使用什么技术可以减少影子页表所带来的开销？
- 5.14.6** [10] <5.6> 使用什么技术可以减少嵌套页表所带来的开销？
- 5.15** 广泛使用虚拟机最大的障碍之一是运行虚拟机所导致的执行开销。下表列出了不同的性能参数和应用程序行为。

基本的 CPI	每 10 000 条指令中特权 O/S 访问次数	对客户 O/S 执行 trap 中断的性能影响	对 VMM 执行 trap 中断的性能影响	每 10 000 条指令中 I/O 访问次数	I/O 访问时间（包括 trap 中断客户 O/S 的时间）
1.5	120	15 个时钟周期	175 个时钟周期	30	1 100 个时钟周期

- 5.15.1** [10] <5.6> 对上面列出的系统计算 CPI，假设没有 I/O 访问。如果 VMM 性能影响加倍，CPI 是多少？如果减半呢？如果一个虚拟机软件公司希望获得 10% 的性能损失，那么对 VMM 执行 trap 中断的最长的时间代价是多少？
- 5.15.2** [10] <5.6> I/O 访问对系统整体性能有着很大的影响。使用上面性能特征值的机器的 CPI，假设是非虚拟化的系统。如果使用虚拟化的系统，CPI 又是多少？如果系统中 I/O 访问减半，那么这些 CPI 如何变化？请解释为什么 I/O 限制性应用受虚拟化影响很小。
- 5.15.3** [30] <5.6, 5.7> 比较并对比虚拟存储器和虚拟机的概念。各自的目标是什么？各自的利弊是什么？列出一些需要虚拟存储器的情况，以及一些需要虚拟机的情况。
- 5.15.4** [20] <5.6> 5.6 节讨论了虚拟化，其中假设虚拟化的系统和底层硬件运行相同的 ISA。然而，虚拟化的一种可能的用途是对非本地的 ISA 进行仿真。QEMU 就是这样一个例子，可以用来仿真多种 ISA，如 MIPS、SPARC 以及 PowerPC。与这种虚拟化相关的一些难点是什么？被模拟的系统可能比在本地 ISA 上运行得更快吗？
- 5.16** 在这个练习中，我们将研究处理器 cache 控制器中带写缓冲器的控制单元。可以使用图 5-40 的有限状态机作为设计有限状态机的出发点。假设 cache 控制器适用于 5.9 节图 5-40 所描述的简单的直接映射 cache，但是需要增加一个写缓冲器，其容量为 1 个块。

494

回忆一下，写缓冲器的目的是用来临时存储，因此处理器在发生脏块缺失时就不用等待两次存储器访问。比起在读新的块之前就写回脏块，写缓冲器缓存了脏块并且立即开始读新的块。而脏块随后被写入主存，同时处理器也在工作。

- 5.16.1** [10] <5.8, 5.9> 如果处理器发出一个请求并且在 cache 中命中，同时一个块将从写缓冲器被写回到主存，此时会发生什么？
- 5.16.2** [10] <5.8, 5.9> 如果处理器发出一个请求并且在 cache 中缺失，同时一个块将从写缓冲器被写回到主存，此时会发生什么？
- 5.16.3** [30] <5.8, 5.9> 设计一个能够使用写缓冲器的有限状态机。
- 5.17** cache 一致性考虑的是多个处理器看到的是同一个 cache 块。下表显示了两个处理器以及它们对一个 cache 块 X 中两个不同字的读/写操作（初始值 $X[0] = X[1] = 0$ ）。假定整数为 32 位。

P1	P2
$X[0] ++;$	$X[1] = 3;$

- 5.17.1** [15] <5.10> 当执行一个正确的 cache 一致性协议时，列出给定 cache 块可能的值。如果协议没有保证 cache 一致性，至少列出一个 cache 块可能的值。
- 5.17.2** [15] <5.10> 对于监听协议，列出每个处理器/cache 完成上面的读/写操作时正确的操作顺序。
- 5.17.3** [10] <5.10> 在最好和最差情况下，完成列出的读/写指令，cache 缺失次数分别是多少？
- 存储器一致性考虑的是看到的多个数据项。下表显示了两个处理器以及它们对不同的 cache 块

的读/写操作 (A 和 B 的初始值为 0)。

	P1				P2	
495	A = 1;	B = 2;	A += 2;	B ++;	C = B;	D = A;

- 5.17.4 [15] <5.10> 若使用 5.10 节列出的保证一致性协议的假设, 请列出 C 和 D 的值。
 5.17.5 [15] <5.10> 如果假设不成立, 那么至少列出一对 C 和 D 可能的值。
 5.17.6 [15] <5.3, 5.10> 对于写策略和写分配策略的多种组合, 哪些组合可以简化协议的执行?
 5.18 单芯片多处理器 (chip multiprocessor, CMP) 在单个芯片上有多个核和 cache。设计 CMP 的片上二级 cache 时都会进行权衡。下表列出了两个基准测试程序在私有二级 cache 和共享二级 cache 中的缺失率和命中延迟。假设每 32 条指令发生一次一级 cache 缺失。

	私有	共享
基准测试程序 A 的每指令缺失	0.30%	0.12%
基准测试程序 B 的每指令缺失	0.06%	0.03%

下表列出了命中延迟:

私有 cache	共享 cache	存储器
5	20	180

- 5.18.1 [15] <5.13> 对于每个基准测试程序, 哪种 cache 设计更好? 请用数据来支持你的结论。
 5.18.2 [15] <5.13> 共享 cache 的延迟随着 CMP 规模的增长而增长。如果共享 cache 的延迟加倍, 请选出最好的设计。当 CMP 中核的数量增加时, 片下带宽就变成瓶颈, 如果片下存储器访问延迟加倍, 请选出最好的设计。
 5.18.3 [10] <5.13> 讨论共享二级 cache 和私有二级 cache 对于执行单线程、多线程以及多道程序负载时的利弊情况; 如果还有片上三级 cache, 请重新考虑这些问题。
 5.18.4 [15] <5.13> 假设两个基准测试程序的基本 CPI 都为 1 (理想的二级 cache)。如果使用非阻塞 cache 能将同时发生二级 cache 缺失的平均次数从 1 提升到 2, 那么当使用共享二级 cache 时, 性能能提升多少? 如果是私有二级 cache, 性能又能达到多少?
 5.18.5 [10] <5.13> 假设新一代的处理器每 18 个月处理器核的数量就会翻倍。为了保证每个核的性能处于相同水平, 那么一个 3 年后的处理器需要多少片下存储器带宽?
 5.18.6 [15] <5.13> 考虑整个存储器层次结构, 哪种优化可以改进同时发生的缺失数量?

- 5.19 在这个练习中, 我们介绍了网络服务器日志的定义, 并且为了改进日志处理速度, 我们对代码优化进行了研究。日志的数据结构定义如下:

```
struct entry {
    int srcIP; // remote IP address
    char URL[128]; // request URL (e.g., "GET index.html")
    long long refTime; // reference time
    int status; // connection status
    char browser[64]; // client browser name
} log [NUM_ENTRIES];
```

假定日志的处理函数如下:

```
topK_sourceIP (int hour);
```

- 5.19.1 [5] <5.15> 对于给定的日志处理函数, 在一个日志项中哪些字段将被访问? 假设 cache 块为 64 字节, 没有预取, 那么给定的函数平均每个项会引发多少次 cache 缺失?
 5.19.2 [10] <5.15> 为了改善 cache 的应用和局部访问, 你会如何重新组织数据结构? 请给出结构代

码定义。

- 5.19.3 [10] <5.15> 请举例说明另一种不同数据结构的日志处理函数。如果两个函数都很重要，为了改进整体性能，将如何重写程序？补充对代码片段和数据的讨论。

对于下面的问题，每对基准测试程序使用的数据来自“SPEC CPU2000 基准测试程序测出的 cache 性能”(<http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/>)，如下表所示。

a.	Mesa / gcc
b.	mcf / swim

- 5.19.4 [10] <5.15> 64KiB 的数据 cache 使用不同的组相联度，对于每个基准测试程序，每种缺失类型（强制、容量和冲突缺失）相应的缺失率分别是多少？

- 5.19.5 [10] <5.15> 为两个基准测试程序共享的一级数据 cache 选择组相联度，其中 cache 大小为 64KiB。如果一级 cache 是直接映射的，那么为 1MiB 的二级 cache 选择组相联度。

- 5.19.6 [20] <5.15> 请列举一个缺失率表的例子说明较高的相联度实际上能增加缺失率。并构建一个 cache 配置以及访问流来给出证明。

01 小测验答案

5.1 1 和 4。（3 是错误的，因为每个计算机的存储器层次结构的开销是不同的，但是在 2013 年开销最高的通常是 DRAM。）

5.3 1 和 4。更低的缺失代价可以允许使用更小的 cache 块，因为没有更多的延迟；而更高的存储带宽通常导致更大的块，因为缺失代价只是稍微大了一些。

5.4 1。

5.7 1 - a, 2 - c, 3 - b, 4 - d。

5.8 2。（大容量的块和预取都能降低强制缺失，因此 1 是错误的。）

497
498

从客户端到云的并行处理器



多处理器和集群

我挥舞着大棒，用尽一切力量，我要么获得巨大的成功，要么一败涂地。我喜欢活得很洒脱。

——Babe Ruth, 美国棒球运动员

6.1 引言

“在月球的山脉上，沿着阴影笼罩的山谷，前进，勇敢的前进！” 阴影回答道
——“如果你在寻找理想国！”

——埃德加·爱伦·坡，《理想国》，第4节，1849

计算机设计者一直在寻求计算机设计的“黄金之城”（El Dorado）：只需将现有的多个较小计算机简单地连接在一起构成功能强大的计算机。这就是多处理器（multiprocessor）产生的根源。在理想情况下，用户可以按照自己的支付能力购买足够多的处理器，从而获得对应数量的性能。因而，多处理器软件必须设计为能在不同数量处理器的情况下工作。如第1章所述，无论是数据中心还是微处理器，功耗已经成为一个首要问题。在软件可以有效地使用每个处理器的情况下，用很多小型高性能处理器代替大型低效能处理器，可在每单位焦耳上获得更高的性能。这样，对多处理器而言，可以通过可伸缩的性能来提高功效。

- ② 多处理器：至少含有两个处理器的计算机系统。与之对应的概念是单处理器（uniprocessor），它仅含一个处理器。

由于多处理器软件支持可变数量的处理器，并且一些设计支持在受损硬件上正常工作；也就是说，如果在包含 n 个处理器的多处理器系统中有一个处理器失效，该系统将继续使用 $n - 1$ 个处理器提供服务。因此，多处理器也提高了可用性（见第5章）。

高性能意味着处理单独一个任务时的高吞吐量，这被称作任务级并行（task-level parallelism）或进程级并行（process-level parallelism）。这些任务是独立的单线程应用程序，这在多处理器计算机中非常重要并且普遍使用。与之相对的方法是在多个处理器上运行一个作业。我们使用术语

并行处理程序 (parallel processing program) 来表示同时运行在多个处理器上的单一程序。

- ② 任务级并行或进程级并行：通过同时运行独立程序的方法来利用多处理器。
- ③ 并行处理程序：同时运行在多个处理器上的单一程序。

在过去数十年里，很多科学问题都需要更快的计算机，同时这些问题也用于评价新型的并行计算机。这些问题中有些在今天处理起来很简单，使用由封装在不同独立服务器上的多个微处理器组成的集群 (cluster) 即可完成计算（见 6.7 节）。除了科学问题以外，集群还可以运行对等请求应用程序，如搜索引擎、Web 服务器、电子邮件服务器和数据库。

- ④ 集群：通过局域网连接的一组计算机，其作用等同于一个大型的多处理器。

如第 1 章所述，多处理器因功耗问题已成为研究焦点，未来处理器性能的提高显然不再是依赖更高的主频和改进 CPI，而是借助于硬件并行。就像我们在第 1 章所看到的，为了避免名称上的冗长，称为多核微处理器 (multicore microprocessor) 而不是多处理器微处理器。因此，处理器在多核芯片内一般称为核 (core)。核的数量预计以摩尔定律 (Moore's Law) 增长。这些多核处理器通常都是共享内存处理器 (shared memory processor, SMP)，因为它们通常共享一个单独的物理地址空间。我们会在 6.5 节更深入地讨论 SMP。

- ⑤ 多核微处理器：在单一集成电路上包含多个处理器（“核”）的微处理器。基本上目前所有的台式机和服务器都是多核微处理器。
- ⑥ 共享内存处理器：共享一个物理地址空间的并行处理器。

当今的技术意味着关心性能的编程人员必须成为并行编程的程序员，因为串行程序就意味着速度慢的程序。

业界面临的一大挑战是如何构建易于正确编写并行处理程序的软硬件系统，不仅程序能够有效执行，而且性能和功耗可以随着单芯片内核的数量而相应改变。

微处理器设计的这种突然转变导致很多设计人员措手不及，因而会有很多关于微处理器设计方面的术语及其内涵的困惑。图 6-1 试图阐述串行 (serial)、并行 (parallel)、顺序 (sequential) 和并发 (concurrent) 等术语之间的差异。该图中的每一列代表固有顺序或并发的软件，每一行表示串行或并行的硬件。例如，编写编译器的程序员认为编译器是顺序程序，因为编译的主要过程包含分析、代码生成和优化等。与之相反，编写操作系统的程序员一般认为操作系统是并发程序，因为操作系统需要协同处理一个计算机中多个独立作业产生的各种 I/O 事件。

		软件	
		顺序	并发
硬件	串行	在 Intel Pentium4 上运行的使用 Matlab 编写的矩阵乘法	在 Intel Pentium4 上运行的 Windows Vista 操作系统
	并行	在 Intel Core i7 上运行的使用 Matlab 编写的矩阵乘法	在 Intel Core i7 上运行的 Windows Vista 操作系统

图 6-1 硬/软件分类以及若干并发应用程序与并行硬件的对比实例

图 6-1 说明了以下两点：第一，并发软件可以运行于串行硬件上（如操作系统可以运行在 Intel Pentium 4 单处理器上），也可以运行于并行硬件上（如操作系统可以运行在 Intel Core i7 上）；第二，反过来顺序软件也是类似的，如 Matlab 程序员认为矩阵乘是顺序执行的，但是它可以串行地在 Intel Pentium 4 上运行，也可以并行地在 Intel Core i7 上运行。

503

也许你会认为并行的唯一挑战是如何使一个固有顺序执行的软件在并行硬件上获得更高性能，但实际上如何让并发程序在多处理器上随处理器数量增加而提高性能也是一个难点。为了加以区别，本章后面的部分使用并行处理程序（parallel processing program）或并行软件（parallel software）表示运行在并行硬件上的顺序软件或并发软件。本章下一节讲述为什么很难编写高效的并行处理程序。

在进一步讨论并行方法之前，我们需要回顾一下前面章节的下述内容：

- 第2章，2.11节：并行与指令：同步。
- 第3章，3.6节：并行性和计算机算术：字并行。
- 第4章，4.10节：指令级并行。
- 第5章，5.10节：并行与存储器层次结构：cache一致性。

01 小测验

是非判断题：为了从多处理器获得好处，应用程序必须是并发的。

6.2 创建并行处理程序的难点

并行的难点不在于硬件，而是目前只有极少量重要的应用程序经过重新编写后能在多处理器上，从而获得更快的执行时间。事实上，在多处理器上编写程序来提高执行效率很困难，而且随着处理器数量的增加会变得更加困难。

为什么会这样呢？为什么并行处理程序相对于顺序程序更加难开发呢？

首要原因是必须使用并行处理程序才能在多处理器上获取更高性能或功耗利用率；否则，就只能在单处理器上使用顺序程序，因为编写顺序程序相对比较简单。事实上，单处理器设计技术（如超标量和乱序执行）充分利用了指令级并行（见第4章），而且通常不需要程序员的介入。这些技术不需要改写程序，因此程序员不做任何事情就可以在新的计算机上获得更高性能。

504

为什么编写更快的并行处理程序非常困难（尤其是让执行速度可随处理器数量的增加而增加）？在第1章中我们打了个比方，让8个记者同时编写同一故事，希望获得8倍的速度完成该项工作。为了实现目标，任务必须被分解为等量的8份，否则会有一些记者处于空闲状态等待其他工作量较大的人员完成任务。另外一个影响性能的障碍是记者们必须花费大量时间进行互相交流，而不是专心编写自己所负责的那部分故事。无论是这个类比还是并行编程，都要面临如下挑战：调度、将任务分割成可并行的部分，负载均衡、同步时间和通信开销。而且，相对于使用更多记者完成一篇新闻报道，使用多处理器完成并行编程要复杂得多。

我们在第1章中还讨论了另外一个障碍，即Amdahl定律。它提示我们为了充分利用多核，程序中任何一个很小的部分都需要并行化。

01 例题·加速比的挑战

如果希望在100个处理器上获得加速比90，请问原始计算中最多有多少可以是顺序执行的呢？

01 答案

根据第1章描述的Amdahl定律：

改进后的执行时间 = 受改进影响的执行时间 / 改进量 + 未受改进影响的时间

使用加速比的形式重新表示Amdahl定律：

$$\text{加速比} = \frac{\text{改进前的执行时间}}{(\text{改进前的执行时间} - \text{受影响的执行时间}) + \frac{\text{受影响的执行时间}}{\text{改进量}}}$$

该公式通常被改写为假定改进前的执行时间为 1 个时间单元的形式，受改进影响的执行时间可以视作与原始执行时间的比值：

$$\text{加速比} = 1 / ((1 - \text{受影响的执行时间比例}) + \text{受影响的执行时间比例} / \text{改进量})$$

将加速比替换为 90，将改进量替换为 100，代入上述公式中：

$$90 = 1 / ((1 - \text{受影响的执行时间比例}) + \text{受影响的执行时间比例} / 100)$$

然后简化该公式并对受影响的执行时间比例进行求解：

$$90 \times (1 - 0.99 \times \text{受影响的执行时间比例}) = 1$$

$$90 - (90 \times 0.99 \times \text{受影响的执行时间比例}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{受影响的执行时间比例}$$

$$\text{受影响的执行时间比例} = 89 / 89.1 = 0.999$$

因此，为了在 100 个处理器上获得加速比 90，顺序执行部分最多占 0.1%。

然而，还是有大量具有固有并发特征的应用程序。□

01 例题·加速比的挑战：更大规模的问题

执行两个加法：一个加法是 10 个标量的求和，另一个加法是一对 10×10 二维矩阵的求和。我们目前假设只有矩阵求和可以并行化，以后会看到如何对标量求和进行并行化。使用 10 个和 40 个处理器达到的加速比分别是多少呢？如果矩阵维数是 20×20 呢？

01 答案

我们假定性能是加法时间 t 的函数，并且假定有 10 次加法不能从并行处理器中获益，100 次加法可以获益。如果在单处理器上的执行时间为 $110t$ ，那么在 10 个处理器上的执行时间是

$$\text{改进后的执行时间} = \text{受影响的执行时间} / \text{改进量} + \text{未受影响的执行时间}$$

$$\text{改进后的执行时间} = 100t / 10 + 10t = 20t$$

所以使用 10 个处理器的加速比是 $110t / 20t = 5.5$ 。使用 40 个处理器的执行时间是

$$\text{改进后的执行时间} = 100t / 40 + 10t = 12.5t$$

所以使用 40 个处理器的加速比是 $110t / 12.5t = 8.8$ 。因此，对于该问题规模，我们使用 10 个处理器达到了潜在加速比的 55%，但是使用 40 个处理器仅达到了潜在加速比的 22%。□

当增大矩阵规模时会发生什么。顺序程序的执行时间为 $10t + 400t = 410t$ 。10 个处理器的执行时间是

$$\text{改进后的执行时间} = 400t / 10 + 10t = 50t$$

所以 10 个处理器的加速比是 $410t / 50t = 8.2$ 。40 个处理器的执行时间是

$$\text{改进后的执行时间} = 400t / 40 + 10t = 20t$$

所以 40 个处理器的加速比是 $410t / 20t = 20.5$ 。因此，对于较大的问题规模，我们使用 10 个处理器获得了大约 82% 的潜在加速比，使用 40 个处理器获得了超过 51% 的潜在加速比。□

这些例子说明为了在多处理器上获得更高加速比，保持问题规模不变相对于增加问题规模会更加困难。为此我们引入两个术语来描述按比例缩放的方式。

强比例缩放（strong scaling）指保持问题规模固定所测得的加速比。**弱比例缩放**（weak scaling）指问题规模随处理器数量按比例增加所获得的加速比。假定问题规模 M 是主存中的工作集，处理器数量为 P ，那么每个处理器所占用的内存对于强比例缩放大约是 M/P ，对于弱比例缩放大约是 M 。

○ 强比例缩放：在多处理器上不需增加问题规模即可获得的加速比。

○ 弱比例缩放：在多处理器上增加处理器数量的同时按比例增加问题规模所能获得的加速比。

注意，传统认知认为弱比例缩放会比强比例缩放简单，但是存储器层次结构可能对这一传统认知产生影响。例如，如果弱比例缩放数据组不再适用于多核微处理器高速缓冲存储器的最后一层，会导致系统的性能比使用强比例缩放更加糟糕。

可根据不同的应用程序选择不同的比例缩放方法。例如，TPC-C 借贷数据库基准测试程序需要按与每分钟内的事务处理次数成比例地增加客户数量。这是因为如果银行装备了更快的计算机，我们也不能假定客户从此以后每天使用 100 次 ATM，很明显这是没有实际意义的。因此，如果希望证明系统可以将每分钟内处理的事务次数提高 100 倍，应当在顾客数量提高 100 倍的情况下进行实验。更大规模的问题需要更多的数据，这是弱缩放方法的特征。

最后一个例子说明了负载均衡的重要性。

01 例题·加速比的挑战：负载均衡

507 在上个例子中，我们使用 40 个处理器在较大问题规模中实现了加速比 20.5，其中假定了负载是完全均衡的。也就是说，40 个处理器中每一个都完成 2.5% 的工作。事实上，如果一个处理器的负载高于其他处理器，则加速比会受到影响。请计算其中一个处理器完成两倍于负载（5%）和五倍于负载（12.5%）时的加速比。对于其他处理器的利用率如何？

01 答案

如果一个处理器负责 5% 的并行负载，那么它需要完成 5% 乘以 400，即 20 次加法，其他的 39 个处理器分担剩余的 380 次加法。由于它们是同时运算的，我们可以取两者工作时间的最大值。

$$\text{改进后的执行时间} = \text{Max}(380t/39, 20t/1) + 10t = 30t$$

加速比从 20.5 降低至 $410t/30t = 14$ 。剩下的 39 个处理器的利用率不及原来的一半：当等待任务最重的处理器完成 $20t$ ，它们只执行了 $380t/39 = 9.7t$ 。

如果一个处理器完成 12.5% 的负载，它必须执行 50 次加法。公式为：

$$\text{改进后的执行时间} = \text{Max}(350t/39, 50t/1) + 10t = 60t$$

加速比进一步降低至 $410t/60t = 7$ 。其余的处理器的利用率不到 $(9t/50t)$ 的 20%。这个例子说明了负载均衡的重要性：仅在一个处理器的负载是其他处理器的两倍时，加速比几乎降低了三分之一；一个处理器的负载是其他处理器的 5 倍时，加速比几乎降低到了原来的三分之一。□

现在我们对并行处理的目标和挑战有了更好的理解，我们在这里给出本章后面内容的一个概览。下一节（6.3 节）介绍了一个比图 6-1 更古老的分类方法。另外，该节也给出了两种可以使串行程序运行在并行硬件上的指令集，即单指令多数据指令（SIMD）和向量指令（vector）。6.4 节介绍了多线程（multithreading），这个概念常常容易与多进程（multiprocessing）混淆，一部分原因是由于它们都依赖于程序中相似的并行性。6.5 节介绍了基本并行硬件的两种类型，它们的区别在于系统中所有处理器是否采用单一的物理地址。这两种类型的常见形式分别是共享存储多处理器（shared memory multiprocessor）和集群（cluster），而该节讲述的是前者。6.6 节介绍了一种来自图形硬件处理领域的相对较新的计算机，称为图形处理单元（GPU），它同样是共享一个物理地址空间的。（附录 C 更详细地介绍了 GPU）。6.7 节介绍了集群，这是使用多个物理地址空间的一个很常见的例子。6.8 节介绍了将多个处理器（可以是集群中的多个服务节点，也可以是微处理器中的多个核）链接起来的分类方法。6.9 节介绍了通过以太网使集群中的多个节点进行通信的硬件和软件。该节展示了如何使用用户软件和硬件优化性能。我们接下来在 6.10 节探讨了寻找并行测试集程序的困难。该节也包含了一个简单但是很有启发意义的性能模型，这个模型可用于辅助应用程序及体系结构的设计。在 6.11 节，我们同时使用该模型和并行测试集程序对一个双核计算机和一个 GPU 进行比较。6.12 节展示了加速矩阵乘法这一旅程的最后也是最庞大的一个步骤。对于无法在 cache 中放下的矩阵，使用 16 个核的并行处理得到 14 倍的性能加速。本

章最后解析了一些常见谬误和陷阱，并进行了总结。

在下一节，我们介绍代表不同类型的并行计算机的英文首字母缩写，这些你可能以前已经见过了。

01 小测验

是非判断：强比例缩放不遵守 Amdahl 定律。

6.3 SISD、MIMD、SIMD、SPMD 和向量机

20世纪60年代提出了并行硬件的一种分类方法，并且一直沿用至今。该分类基于指令流的数量和数据流的数量。图6-2给出了该分类方法。这样，常规的单处理器具有单一的指令流和单一的数据流，而常规的多处理器具有多个指令流和多个数据流。这两种类别分别称为SISD和MIMD。

- SISD：单指令流单数据流的单处理器。
- MIMD：多指令流多数据流的多处理器。

		数据流	
		单	多
指令流	单	SISD: Intel Pentium 4	SIMD: x86 的 SSE 指令
	多	MISD: 至今没有实例	MIMD: Intel Core i7

图6-2 基于指令流和数据流数量的硬件分类和实例：SISD、SIMD、MISD 和 MIMD

在MIMD计算机上可以编写独立的程序并运行在不同的处理器上，而且这些程序可以协同完成一个共同的大型目标。但是编程人员通常仅编写单一程序，将其运行在MIMD计算机的所有处理器上，然后使用条件控制语句使不同的处理器执行不同的代码段。这种风格被称作单程序多数据（single program multiple data, SPMD），它是MIMD计算机编程的正常方式。

- SPMD：单程序多数据流。传统的MIMD编程模型，其中一个程序运行在所有处理器之上。

最接近多指令流单数据流（MISD）的处理器应该算是“流处理器”了，流处理器在流水线中对一个单独的数据流执行一系列计算：从网络中解析输入，分析数据，解压数据，查找匹配，等等。相反，SIMD更常见一些。SIMD计算机对向量数据进行操作。例如，一个单一的SIMD指令可以把64个数相加，只需要把64个数据流发送到64个ALU，就可以在一个时钟周期内得到64个和。我们在3.6节和3.7节中见到的子字并行指令是另一个SIMD的例子；实际上，Intel的SSE中的第二个S就代表SIMD。

- SIMD：单指令流多数据流。同样的指令在多个数据流上操作，和在向量处理器中的一样。

SIMD的优点是所有并行执行单元都是同步的，它们都对源自同一程序计数器（PC）的同一指令作出响应。从程序员的角度来看，非常接近于已经熟悉的SISD。尽管每个单元都执行相同指令，但是每个执行单元都有自己的地址寄存器，这样每个单元都有不同的数据地址。因此，根据图6-1，一个顺序应用程序编译后可能运行于串行硬件上并按SISD组织，也可能运行于并行硬件上按SIMD组织。

SIMD的初衷是在几十个执行单元之间均摊控制单元成本。另外一个优点是降低指令宽度和空间——SIMD只需要同时执行代码的一个副本，而消息传递的MIMD可能需要在每个处理器都有一份副本，共享存储器MIMD可能需要多个指令缓存。

SIMD 在使用 `for` 循环语句处理数组时最为有效。因此，为了在 SIMD 中并行工作，必须有大量相同结构的数据，一般称之为**数据级并行**（data-level parallelism）。SIMD 在使用 `case` 或 `switch` 语句时效率最低，此时每个执行单元必须根据不同的数据执行不同的操作。带有错误数据的执行单元必须被摒弃，而带有正确数据的执行单元将继续执行。若有 n 个 `case`，SIMD 处理器将会以最高 $1/n$ 的性能运行。

● 数据级并行：对不同数据执行相同操作所获得的并行。

激发了 SIMD 类型产生的阵列处理器正逐渐成为历史（见网站上的 6.15 节），但是直到现在 SIMD 的两种表示依然并存。

6.3.1 在 x86 中的 SIMD：多媒体扩展

正如第 3 章所述，在 1996 年，对于窄位宽整型数据的子字并行化激发了 x86 指令集中的多媒体扩展（MMX）指令的产生。随着摩尔定律的发展，加入了更多的指令，产生了最初的 SSE 扩展，现在为 AVX 扩展。AVX 支持同时执行 4 个 64 位的浮点数据。操作和寄存器的位宽编码到多媒体指令的操作码中。随着操作和寄存器位宽的变大，多媒体指令的操作码数量也在增加，现在已经有数百条 SSE 和 AVX 指令（见第 3 章）。

6.3.2 向量机

510 SIMD 的一个更加古老和优雅的称呼是向量体系结构，它几乎等同于西摩克雷（Seymour Cray）在 20 世纪 70 年代制造的计算机。向量机结构与具有大量数据并行的问题非常匹配。除了具有 64 个 ALU 可以同时计算 64 次加法之外，与早期的阵列处理器类似，向量体系结构将 ALU 流水化，从而在低成本下获得高性能。向量体系结构的基本理念是从存储器中收集数据元，并将它们按顺序放到一大组寄存器中，然后在寄存器中使用流水化的执行单元对它们依次操作，最后将结果写回存储器。向量体系结构的关键特征是拥有一组向量寄存器。这样，向量体系结构可能拥有 32 个向量寄存器，每个寄存器包含 64 个 64 位宽的数据元。

01 例题·向量机与常规处理器在代码上的区别

假设我们基于 MIPS 指令集体系结构进行扩展，增加向量指令和向量寄存器。向量操作的名称与 MIPS 原有操作相同，但是在其后增加一个字母“V”。例如，`addv.d` 表示将两个双精度向量相加。向量指令的输入可以是一对向量寄存器（`addv.d`），也可以一个是向量寄存器一个是标量寄存器（`addvs.d`）。对于后者，标量寄存器的值被用于所有操作的输入——`addvs.d` 操作将会把标量寄存器的内容加到向量寄存器中每个数据元上。关键词 `lv` 和 `sv` 分别代表向量的读入和写回，它们完成整个双精度数据向量的读入或写回。`lv` 和 `sv` 的一个操作数是要读入或写回的向量寄存器；另一个操作数是一个 MIPS 的通用寄存器，用来给出向量在存储器中的起始地址。在简要说明之后，我们看下面的一小段代码如何从常规的 MIPS 代码转换成向量 MIPS 代码：

$$Y = a \times X + Y$$

其中 X 和 Y 是 64 位双精度浮点数的向量，并且最初保存在存储器中； a 是一个双精度标量。（这个例子就是所谓的 DAXPY 循环，其构成了 Linpack 基准测试程序的内部循环。DAXPY 表示 double precision $a \times X$ plus Y 。）假定 X 和 Y 的起始地址分别保存在 $\$s0$ 和 $\$s1$ 中。

01 答案

针对 DAXPY 的常规 MIPS 代码是：

```

l.d      $f0,a($sp)      :load scalar a
addiu   $t0,$s0,#512    :upper bound of what to load
loop: l.d      $f2,0($s0)    :load x(i)
      mul.d   $f2,$f2,$f0    :a x x(i),
      l.d      $f4,0($s1)    :load y(i)
      add.d   $f4,$f4,$f2    :a x x(i) + y(i)
      s.d      $f4,0($s1)    :store into y(i)
      addiu   $s0,$s0,#8     :increment index to x
      addiu   $s1,$s1,#8     :increment index to y
      subu   $t1,$t0,$s0     :compute bound
      bne    $t1,$zero,loop   :check if done

```

针对 DAXPY 的向量 MIPS 代码是：

```

l.d      $f0,a($sp)      :load scalar a
lv      $v1,0($s0)        :load vector x
mulvs.d $v2,$v1,$f0      :vector-scalar multiply
lv      $v3,0($s1)        :load vector y
addv.d  $v4,$v2,$v3      :add y to product
sv      $v4,0($s1)        :store the result

```

511

□

针对上面两段代码有几点值得注意。最引人注目的是向量处理器大大降低了动态指令带宽需求，仅用 6 条指令就完成了接近 600 条 MIPS 指令的工作。降低的原因一是向量操作是在 64 个数据元上同时进行的，二是 MIPS 中接近一半开销的循环指令在向量机代码中不存在了。正如你所想的一样，取指和执行次数的降低也会节省能耗。

另外一个重要的不同点是流水线相关的频率（见第 4 章）。在我们直接编写的 MIPS 代码中，每个 add.d 必须等待 mul.d，并且每个 s.d 必须等待 add.d，另外每个 add.d 和 mul.d 必须等待 l.d。在向量处理器中，每条向量指令只会在每个向量的起始数据元阻塞，在随后的数据元都会顺畅地通过流水线。因此，流水线阻塞在每次向量操作时只会发生一次，而不是每次对向量数据元进行操作时都会发生一次。在这个例子中，MIPS 中的流水线阻塞频率大约比 MIPS 的向量版本高 64 倍。当然，MIPS 可以采用循环展开技术降低流水线阻塞频率（见第 4 章），但是指令带宽的巨大差异是无法减小的。

由于向量元素是相互独立的，它们可以并行地执行，很像 AVX 指令的子字并行。现在所有的向量计算机都带有多个并行流水线（称为向量通道，见图 6-2 和图 6-3）的向量功能单元，每个流水线在一个时钟周期可以产生两个甚至更多的结果。

01 精解 上面的例子中循环次数恰好等于向量长度。当循环次数更小时，向量体系结构可以使用降低向量操作长度的寄存器。当循环次数更大时，我们可以增加记录代码来迭代全长度向量操作，最后处理剩余部分。后面的处理过程称作条状开发法（strip mining）。

6.3.3 向量与标量的对比

与常规的指令集体系结构（本部分将其称为标量体系结构）相比，向量指令具有几个重要的属性：

- 一条向量指令指定了大量需要完成的工作——它等价于执行一个循环。因而对取指和译码带宽的需求显著降低了。
- 通过使用向量指令，编译器或程序员隐含指明向量中每个结果的计算与同一向量中其他结果的计算是不相关的，因而硬件无需检查一条向量指令内的数据相关。
- 相对于 MIMD 多处理器，包含数据级并行的应用程序采用向量体系结构和编译器能够更加容易地编写高效代码。

512

- 硬件只需在两条向量指令之间对每个向量操作数检查一次数据相关，而不是对向量内每个数据元检查一次。相关检查次数的降低也会使得能耗降低。
- 访问存储器的向量指令具有确定的存取模式。如果向量的每个元素的地址都是连续的，那么从一组交叉存储器组中取回一个向量将会很快。因此，对整个向量而言，主存延迟的开销看上去只有一次，而不是对向量中每个字都有一次。
- 因为整个循环用具有预定义行为的向量指令所替换，循环转移所引起的控制相关就不存在了。
- 节省的指令带宽和相关检查以及存储器带宽的有效使用，使得向量体系结构在能耗方面优于标量体系结构。

由于这些原因，在同样的数据量前提下，向量操作比一组标量操作序列更快，并且如果应用程序可以频繁使用这些向量操作，就会促使设计者加入向量单元。

6.3.4 向量与多媒体扩展的对比

与 x86 AVX 多媒体指令扩展类似，向量指令可以指定多种操作。然而，多媒体扩展一般仅指定几种操作，而向量可以指定几十种操作。与多媒体扩展不同的是，向量操作中分量的数量不在操作码中，而是在一个单独的寄存器中。这个区别意味着不同版本的向量体系结构只需修改该寄存器的值，就能够实现不同的分量数量，并且能够保持二进制代码的兼容性。相比之下，在 x86 的多媒体扩展体系结构中（MMX、SSE、SSE2、AVX、AVX2、…），每次“向量”长度改变时都需要加入大量新的操作码。

还有一点与多媒体扩展不同，数据传输不必是连续的。向量同时支持按步长存取（strided access）和变址存取（indexed access），前者是硬件每隔 n 个存储器中的数据元读取一次，后者是按照数据项地址读取到向量寄存器中。变址存取也称作聚集分散（gather-scatter），变址的读取操作将内存中的数据元素聚集成连续的向量元素，变址的存储操作将向量元素分散到内存中。

与多媒体扩展类似，向量机可以灵活地支持不同数据宽度，因此它既可以在 32 个 64 位数据上进行向量操作，也可以在 64 个 32 位数据、128 个 16 位数据或者 256 个 8 位数据上进行向量操作。向量指令的并行特性可以使其采用深度流水的功能单元、并行功能单元阵列或并行功能单元与流水功能单元的组合来实现。图 6-3 说明了如何采用并行流水线执行一条向量加法指令来提高向量的性能。

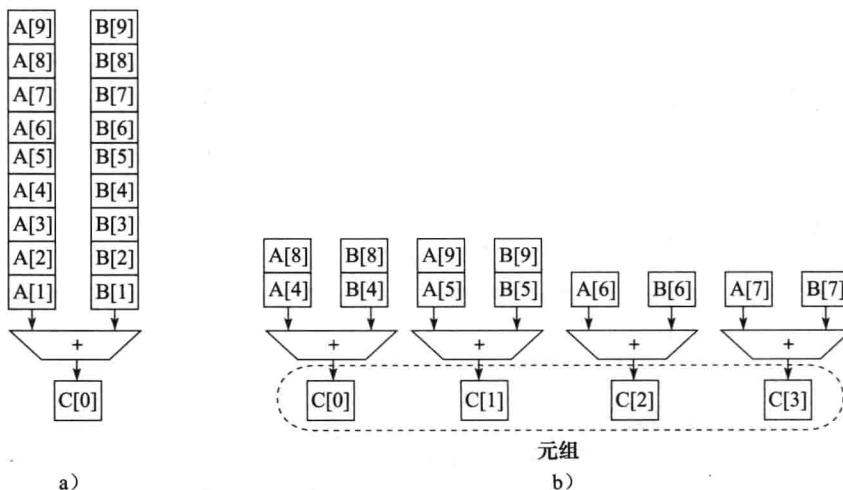


图 6-3 使用多个功能单元来提升单个向量加法指令， $C = A + B$ 。左侧的向量处理器有一条加法流水线，并且可以在一个周期完成加法操作。右侧的向量处理器有 4 条加法流水线，并且可以在一个周期完成 4 条加法操作。一条向量加法指令的数据元素被分叉地放到 4 个通道中

向量算术运算指令通常只允许一个向量寄存器的元素 N 与另一个向量寄存器的元素 N 进行计算。通过多个向量通道（vector lane）的方式构建高度并行化的向量单元，极大地简化了高度并行化的向量单元的实现。由于有高速的通道，我们可以通过增加更多的通道数量来提高向量单元的最大吞吐率。图 6-4 展示了一个四通道的向量单元的结构。因此，通过将通道数从一个增至 4 个，使每条向量指令的周期数减少了大约 4 倍。由于多通道具有优势，故应用程序和体系结构都必须支持长向量。否则，指令会很快地执行完毕却得不到新的指令去执行，而去要求像第 4 章介绍的指令级并行提供足够的向量指令。

- ② 向量通道：一个或多个向量功能单元与一部分向量寄存器。由为提高交通流量的高速公路的道路数启发而来，多个通道同时执行向量操作。

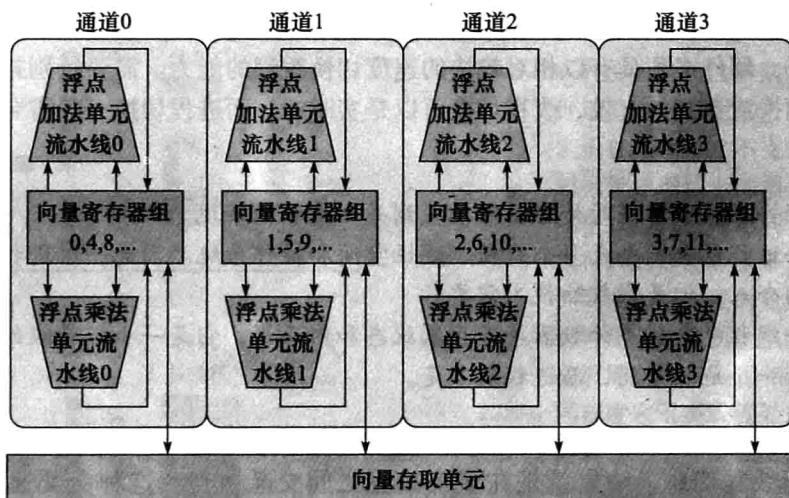


图 6-4 有 4 个通道的向量处理器的结构。向量寄存器等量地分配给每个通道，每个通道所占有的寄存器直接相隔为 4。图中画出了三个向量功能单元：一个浮点加法器、一个浮点乘法器和一个存取单元。每一个向量算术单元都包含 4 个执行流水线，每个通道一个，每条流水线执行一条指令。注意向量寄存器的每一部分是如何只需要为自己的通道提供足够的读和写端口（见第 4 章）的

总的来说，向量体系结构是执行数据平行处理程序的一种有效途径；相对多媒体扩展，向量机与编译器技术更加接近；并且相对于对 x86 体系结构进行多媒体扩展，向量技术更加容易随时间推移而得到不断改进。

给出了这些经典的分类方法，我们接下来看看如何发掘指令的并行流来提高一个单独处理器的性能，我们还会将该方法应用到多处理器中。

01 小测验

是非判断：以 x86 为例，多媒体扩展可以被视作一种采用短向量的仅支持顺序向量数据传输的向量体系结构。

01 精解 在了解了向量体系结构如此之多的优点之后，为何向量机却没有在高性能计算领域之外流行呢？主要原因包括：向量寄存器的巨大状态增加了上下文切换时间；向量存取产生的缺页故障难以处理；SIMD 指令也可以获得向量指令的部分优势。另外，只要指令级并行可以提供摩尔定律要求的性能提升，就没有理由要去改变体系结构的类型。

01 精解 向量和多媒体扩展的另外一个优点是一个标量指令集体体系结构更易于扩展，从而使其提高数据并行操作的性能。

01 精解 Intel 的 Haswell x86 处理器支持 AVX2 指令集，AVX2 指令级只有聚集（gather）操作而没有分散（scatter）操作。

6.4 硬件多线程

从程序员的角度来看，硬件多线程（hardware multithreading）就是一个和 MIMD 相关的概念。MIMD 依靠多个进程（process）或线程（thread）来努力使多个处理器处于忙碌状态，而硬件多线程允许多个线程以重叠的方式共享一个处理器的功能单元，以有效地利用硬件资源。为了支持共享，处理器必须为每个线程复制独立的状态。例如，每个线程必须拥有寄存器文件和 PC 的独立备份。存储器自身可以通过虚拟存储器机制实现共享，多道程序设计中已经支持这种方法。此外，硬件必须具有以相对较快的速度切换线程的能力。需要特别指出的是，线程切换相对进程切换应该更加有效，线程切换可以是实时的，而进程切换一般需要数百个到数千个处理器周期。

- ② 硬件多线程：在线程阻塞时处理器可切换到另一线程的实现。
- ② 进程：一个进程包含一个或多个线程、地址空间和操作系统。因此一次进程切换通常需要操作系统的介入，但是线程切换不需要。
- ② 线程：一个线程包含程序计数器、寄存器状态和内存栈。它是一个轻量级的进程；多个线程通常共享一个地址空间，而进程则不是。

硬件多线程主要有两种实现方法。细粒度多线程（fine-grained multithreading）在每条指令执行后都进行线程切换，结果就是在多个线程之间交叉执行。这种交叉通常以循环方式进行，并在循环时，在每个时钟周期跳过处于阻塞状态的线程。为了实现细粒度多线程，处理器必须能够在每个时钟周期进行线程切换。细粒度多线程的一个主要优点是可同时隐藏由短阻塞和长阻塞引起的吞吐量损失，因为当一个线程阻塞时可以执行其他线程的指令。细粒度多线程的主要缺点是降低了单个线程的执行速度，因为就绪状态的线程会因为其他线程而延迟执行。

- ② 细粒度多线程：硬件多线程的一种形式，每条指令执行之后都进行线程切换。

粗粒度多线程（coarse-grained multithreading）是细粒度多线程的一种替代方案。粗粒度多线程仅在高开销阻塞时才进行线程切换，如最后一级缓存缺失。这种改变对高速的线程切换降低了要求，并且几乎不会降低单个线程的执行速度，因为粗粒度多线程仅在当前线程遇到高开销阻塞时才会发射其他线程的指令。然而，粗粒度多线程有一个严重的缺点：它在隐藏吞吐量损失的能力方面受限，特别是短阻塞。这种限制源自粗粒度多线程中的流水线启动开销。因为粗粒度多线程处理器从单一线程发射指令，在阻塞发生时，必须清空或冻结流水线。阻塞之后开始执行的新线程必须在指令能够完成之前填充流水线。由于启动开销，粗粒度多线程更加适合用来降低高开销阻塞带来的性能损失，因为在这种情况下，与阻塞时间相比，流水线重新填充时间是可以忽略的。

- ② 粗粒度多线程：硬件多线程的一种形式，仅在一些重要事件（如最后一级缓存缺失）之后进行线程切换。

同时多线程 (simultaneous multithreading, SMT) 是硬件多线程的一个变种, 它使用多发射动态调度流水线处理器的资源来挖掘线程级并行, 并同时保持指令级并行 (见第 4 章)。提出 SMT 的主要原因是在多发射处理器中通常有单线程难以充分利用的多个并行功能单元。而且, 借助于寄存器重命名和动态调度 (见第 4 章), 不需考虑它们之间的相关性即可发射来自不同线程的多条指令; 相关性的解决可以由动态调度机构来处理。

- 同时多线程: 多线程的一种形式, 利用多发射、动态调度微体系结构中的资源实现多线程, 从而降低多线程的开销。

SMT 依赖于现有的动态机制, 且不用每个周期切换资源。事实上, SMT 总是执行来自多个线程的指令, 由硬件将指令槽和重命名寄存器与适当的线程关联起来。

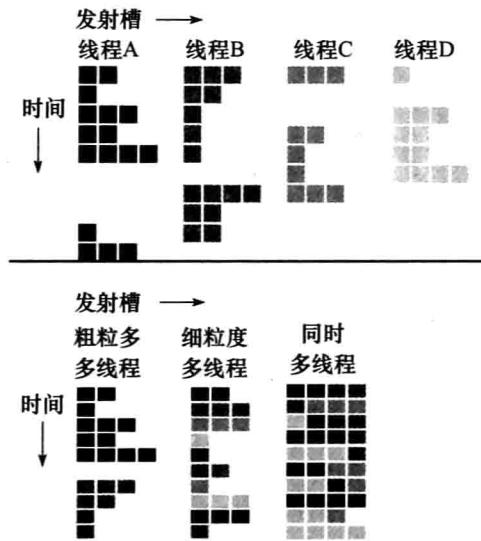


图 6-5 4 个线程如何以不同方式利用超标量处理器中的发射槽。上面的 4 个线程表示独立运行在不支持多线程的标准超标量处理器上的情况。下面给出了三个线程以三种不同多线程模式一起执行时的情况。水平方向表示每个时钟周期的指令发射量。垂直方向表示时钟周期的序列。空块 (白块) 表示在该周期没有利用相应的发射槽。不同灰度表示多线程处理器中的 4 个不同线程。尽管粗粒度多线程中额外的流水线启动开销在本图中没有标识, 但其会导致更多的吞吐量损失

图 6-5 说明了使用一个处理器的不同配置开发超标量资源的能力上的差别。上面的部分表示 4 个线程如何在不支持多线程的超标量处理器上独立运行。下面的部分表示 4 个线程如何以 3 种不同的多线程方式在处理器上更加有效地运行:

- 支持粗粒度多线程的超标量。
- 支持细粒度多线程的超标量。
- 支持同时多线程的超标量。

在不支持硬件多线程的超标量处理器中, 指令发射槽的使用受到指令级并行性的限制。而且, 绝大多数阻塞, 如指令缓存缺失, 会使整个处理器空闲。

在粗粒度多线程超标量处理器中, 通过切换到其他使用该处理器资源的线程可以部分隐藏长阻塞。尽管这能降低完全空闲的时钟周期数量, 但是流水线的启动开销仍然会带来空闲周期, 并使 ILP 受到限制, 也就是说, 并非所有发射槽都能得到有效利用。在细粒度多线程中, 线程的交叉执行几乎不会出现发射槽全空的情况。但是, 由于在一个给定的时钟周期仅有单一线程发射指令, 指令级并行的限制仍会导致某些时钟周期出现空闲发射槽。

在 SMT 中, 线程级并行和指令级并行都得到充分利用, 在一个时钟周期多个线程共同使用发射槽。理想情况下, 发射槽的使用仅受多个线程间资源失衡和资源可用性的限制。实际上, 还有一些其他因素限制可用发射槽的多少。尽管图 6-5 大大简化了这些处理器的真实操作

情况，但是它确实从整体上给出了多线程潜在的性能优势，特别是 SMT。

图 6-6 画出了在 Intel Core i7 960 的一个处理器上运行多线程时的性能和能耗优势，Intel Core i7 960 支持两个线程。平均加速比为 1.31，这对于有少量额外资源执行硬件多线程的情况来说不算坏。平均能耗效率提升为 1.07，效果很好。总之，对于在能耗不变的前提下性能得到了提升总是使人高兴的。

现在我们看到了如何通过多个线程更有效地使用一个处理器的资源，接下来看看如何利用多线程来发掘多处理器的资源。

01 小测验

1. 是非判断：多线程和多核都依赖并行来获得更高效率。
2. 是非判断：同时多线程（SMT）使用线程提高动态调度的乱序处理器的资源使用率。

6.5 多核和其他共享内存多处理器

尽管硬件多线程在很小的代价下提升了处理器的效率，但在过去的 10 年中的主要挑战是通过有效地编程利用单个芯片上数量不断增长的处理器以使性能以摩尔定律继续增长。

由于对原有程序进行重写，使之在并行硬件上很好地运行有困难，一个自然的问题是计算机设计者如何简化该工作。一种方法是为所有处理器提供一个共享的单一物理地址空间，以便程序不必考虑它们的数据在哪里，只要知道程序能够并行执行就可以了。在这种方法中，一个程序的所有变量对其他任何处理器在任何时刻都是可见的。另一种方法是每个处理器采用独立的地址空间，则必须进行显式共享；我们将在 6.7 节描述这种情况。当物理地址空间公用时，通常由硬件提供 cache 一致性，以便保证共享存储器的一致性（参见 5.8 节）。

综上所述，为程序员提供跨越所有处理器的单一物理地址空间的多处理器称为共享内存多处理器（SMP）——对于多核芯片总是这样——尽管更加准确的术语应该是共享地址多处理器（shared-address multiprocessor）。处理器通过存储器中的共享变量互相通信，所有处理器都能通过存取指令访问任何存储器位置。图 6-7 给出了 SMP 的典型组成。注意，即使这些系统共享同一个物理地址空间，它们仍然可以在自己的虚拟地址空间中单独地运行程序。

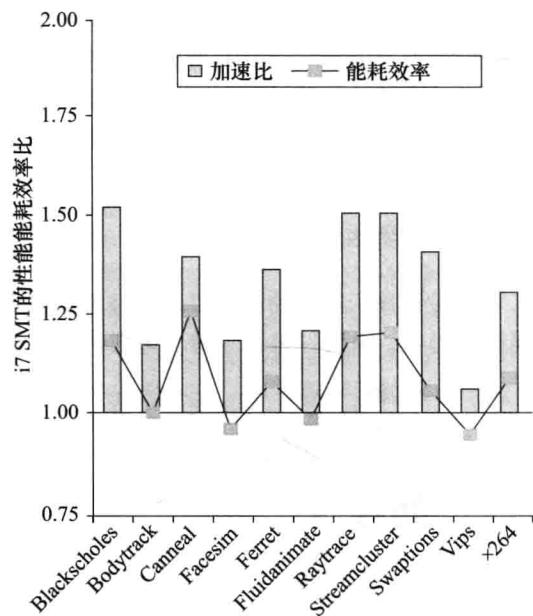


图 6-6 在 i7 处理器上的一个核上使用多线程运行 PARSEC 测试集程序（见 6.9 节），平均加速比为 1.31，功耗效率提升 1.07。该数据是由 Esmaeilzadeh 等。
[2011] 收集分析得来

518
519

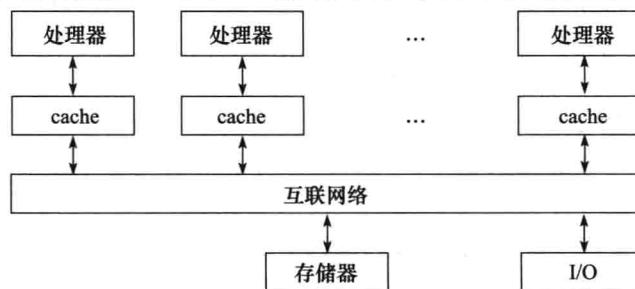


图 6-7 一个共享内存多处理器的典型组成

单一地址空间的多处理器有两种类型。第一种类型的访存时间不依赖于是哪个处理器提出访存请求，也无论要访存哪个字。这类机器称为统一存储访问（Uniform Memory Access, UMA）多处理器。对于第二种类型，一些访存请求会比其他的快，这取决于是哪个处理器访问哪个字，这是由于主存被分割并分配给同一个芯片上的不同的处理器或内存控制器。这类机器称为非统一存储访问（Nonuniform Memory Access, NUMA）多处理器。NUMA 多处理器的编程难度要高于 UMA 多处理器，但 NUMA 机器可以扩展到更大规模，并且 NUMA 访问附近的存储器时具有较低的延迟。

- ➊ 统一存储访问：无论访存的是哪个处理器，也无论访存的是哪个字，访存时间都大致相同的多处理器。
- ➋ 非统一存储访问：使用单一地址空间多处理器的一种类型，某些存储访存速度高于其他访存，访存速度与访问哪个处理器及访问哪个字相关。

由于处理器并行执行一般都需要共享数据，所以它们在操作共享数据时需要进行协调；否则，一个处理器可能会在其他处理器尚未完成对共享数据的操作时就开始使用该数据了。这种协调称为同步（synchronization），正如我们在第 2 章看到的。在使用单一地址空间支持的共享时，必须提供一套独立的同步机制。一种方法是为每个共享变量使用锁（lock）。在一个时刻只能有一个处理器获得锁，其他需要操作该共享数据的处理器必须等待，直到该处理器解锁该变量为止。第 2 章 2.11 节描述了 MIPS 中关于锁操作的指令。

- ➌ 同步：对可能运行于不同处理器上的两个或者更多进程的行为进行协调的过程。
- ➍ 锁：一个时刻仅允许一个处理器访问数据的同步装置。

520

01 例题·一个共享地址空间的简单并行处理程序

假设我们需要在一个处理器数量为 64 的共享存储多处理器计算机上对 64 000 个数求和，该计算机具有统一的存储器访问时间。

01 答案

第一步是保证每个处理器的负载是均衡的，所以我们将这组数分成等量的子集。由于该机器具有单一的存储器空间，因此我们不把这些子集分配到不同的存储器空间上；我们只给每个处理器分配不同的起始地址。用 P_n 表示不同处理器的编号，取值范围在 0 ~ 63 之间。所有处理器启动程序运行一个循环来完成它们子集中数的求和：

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
    sum[Pn] += A[i]; /*sum the assigned areas*/
```

（注意，在 C 程序代码中， $i + = 1$ 是 $i = i + 1$ 的简写形式。）

下一步是将这 64 个部分和加起来，称为约简（reduction）。我们采用分而治之的方法。首先用一半处理器对部分和求和，然后再用四分之一处理器对新的部分和求和，以此类推直到获得最终的和。图 6-8 对约简的过程进行了说明。

- ➊ 约简：处理一个数据结构并返回单一值的函数。□

在该例子中，“消费者”处理器在读取由“生产

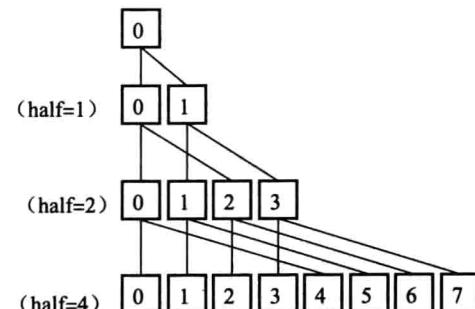


图 6-8 自底向上的最后 4 级求和过程。对于所有编号 i 小于 $half$ 的处理器，将自己产生的部分和与编号 $i + half$ 的处理器产生的部分相加。

521 者”处理器写入结果的存储器位置之前必须同步；否则，消费者可能读取到数据的旧值。我们希望每个处理器拥有自己的循环计数器变量 *i*，因此我们将其声明为“私有”变量。下面是相应的代码（*half* 也是私有变量）：

```
half = 64; /*64 processors in multiprocessor*/
do
    synch(); /*wait for partial sum completion*/
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
    /*Conditional sum needed when half is
     odd; Processor0 gets missing element */
    half = half/2; /*dividing line on who sums */
    if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

01 硬件/软件接口 由于人们长久以来对并行编程都有着浓厚的兴趣，现在已经有了上百种创建并行编程系统的尝试。一个有局限性但是很常用的例子就是 OpenMP。它只是一个带有一些编译器提示、环境变量和动态链接库的对现有标准语言进行扩展的应用程序接口（Application Programmer Interface，API）。它为共享存储器的多处理器提供了一个便于携带的、可伸缩并且简单的编程模型。它的最初目标是对循环和递归进行并行化。

大部分 C 语言编译器已经提供了对 OpenMP 的支持。在 UNIX 下的 C 语言编译器中使用 OpenMP API 的命令如下：

```
cc -fopenmp foo.c
```

OpenMP 使用 pragma 对 C 语言进行扩展，就像 C 宏预处理器命令 #include 和 #define 一样。与上面的例子中一样，我们要使用 64 个处理器，命令如下：

```
#define P 64 /* define a constant that we'll use a few times */
#pragma omp parallel num_threads(P)
```

这样，运行时库就会使用 64 个并行线程。

要将一个串行的 for 循环变为一个并行的 for 循环，并且要把任务等份地分割成我们指定的线程数，我们只需要写如下代码（这里假设 sum 初始为 0）：

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
    for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
        sum[Pn] += A[i]; /*sum the assigned areas*/
```

522 对于递归，我们可以使用另一个命令告诉 OpenMP 什么是递归操作符和用什么变量代替递归运算的结果。

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* Reduce to a single number */
```

注意，现在就要靠 OpenMP 库来找到使用 64 个处理器来完成 64 个数字相加的最佳代码了。

尽管 OpenMP 使得编写并行代码更加简单，但是对于调试并不是很有帮助，所以很多并行程序员使用比 OpenMP 更复杂的并行编程系统，就像今天有很多程序员使用比 C 语言效率更高的编程语言一样。

③ OpenMP：一个为运行在 UNIX 或 Microsoft 平台上的 C、C++ 或 Fortran 语言的共享存储器多处理器的 API。它包含了一些给编译器的提示、一个库和一些运行时提示。

以上给出了一个经典的 MIMD 硬件和软件的例子，我们的下一个关于 MIMD 的例子继承于一个十分不同的体系结构，并且对于并行编程更具有挑战性。

01 小测验

是非判断：共享存储多处理器不能利用任务级并行性的优势。

01 精解 一些作者使用 SMP 作为同步处理器（symmetric multiprocessor）的简称，以此来说明无论哪个处理器访问存储器，延时都是一样的。这么做是为了和大规模 NUMA 多处理器做区别，因为两者都是共享一个地址空间。由于集群比大规模 NUMA 多处理器更为常见，在本书中，我们仍然使用 SMP 来表示它最原始的含义（即共享内存多处理器），并用它来区别使用多个地址空间的处理器，例如集群。

01 精解 除了共享物理地址空间之外，还有一种方法是使用独立的物理地址空间，但共享同一虚地址空间，由操作系统负责处理通信。这种方法已经有过尝试，但为了向注重性能的程序员提供一个实用的共享存储器抽象，它的开销显得过大。

523

6.6 图形处理单元简介

在现有体系结构中增加 SIMD 指令的一个最初理由是许多微处理器都连接到 PC 或工作站中的图形显示设备上，并且用于图形显示的处理时间所占比例越来越大。因此，当微处理器设计中可用晶体管数量随着摩尔定律的发展增加时，提高图形处理能力就变得有必要了。

提高图形处理能力的主要动力是计算机游戏产业，包括 PC 和专用的游戏终端（如 Sony PlayStation）。快速增长的游戏市场使许多公司增加了快速图形硬件方面的研发，这种正反馈使得图形处理能力的增长超过了主流微处理器的通用处理能力。

由于图形和游戏开发与微处理器开发有着不同的目标，故而图形处理采用了自己的一套处理风格和术语。随着图形处理器地位的上升，它们将自己命名为图形处理单元（Graphics Processing Unit，GPU），以便区分子 CPU。

如今人们可以只花几百美元就能买到带有上百个并行浮点运算单元的 GPU，这使得进行高性能计算更容易了。这种趋势与程序语言相结合，促进了人们对 GPU 计算的兴趣的增长同时使得 GPU 更易于编程。因此，很多科学计算和多媒体程序的编程人员开始犹豫是使用 CPU 还是 GPU。

（本节专注于使用 GPU 进行计算。要想知道使用 GPU 进行计算是如何与其作为图形加速卡这一原始功能进行结合的，请参阅附录 C。）

下面是 GPU 与 CPU 的几个主要差别：

- GPU 是补充 CPU 的加速器，因此它们不必执行 CPU 的全部任务。这种定位使得它们专注于图形方面的资源。对于一个同时具有 GPU 和 CPU 的系统来说，GPU 可以对某些任务执行效率很低甚至不能完成时，可以让 CPU 在必要的时候完成。
- GPU 解决的问题规模通常为几百 MB 到 GB，而不是几百 GB 到 TB。

这些差异导致体系结构的设计风格不同：

- 也许最大的不同就是 GPU 不像 CPU 一样依赖多级缓存来隐藏访问存储器的长延迟。事实上，GPU 依赖硬件多线程（6.4 节）来隐藏到访存的延迟。也就是说，在存储器请求和数据到达之间，GPU 会执行数以百计甚至数以千计的与该请求无关的线程。
- GPU 的主存是面向带宽的而不是面向延迟的。甚至有面向 GPU 的特殊图形 DRAM 芯片，相对于面向 CPU 的 DRAM，它的宽度更大并能提供更大带宽。除此之外，GPU 存

524

储器历来都小于常规微处理器的存储器。在 2013 年，GPU 一般有不超过 4~6GiB 的存储器，而 CPU 一般在 32~256GiB 之间。最后，需要注意对于通用计算，必须将数据在 CPU 存储器和 GPU 存储器之间的传输时间包含进来，因为 GPU 是一个协处理器。

- 考虑到 GPU 是通过多线程并行执行来获取高存储器带宽的，除了多线程，GPU 还可以提供许多并行处理器（MIMD）。因此，每个 GPU 相比于 CPU 有更多的线程，并且拥有更多的处理器。

01 硬件/软件接口 尽管 GPU 是为众多应用程序中很小一部分设计的，但是一些程序员希望能以某种形式编制他们的应用，以利用 GPU 内潜在的高性能。在厌倦了使用图形 API 语言描述问题之后，他们开发了类 C 编程语言，可以直接在 GPU 上编程。NVIDIA 的 CUDA (compute unified device architecture) 是其中一个例子，它使得程序员可以编写直接在 GPU 运行的 C 程序，尽管仍有一些限制。附录 C 给出了 CUDA 代码的例子。（OpenCL 是一个由多个公司发起的一种轻型编程语言，它可以提供很多 CUDA 中的功能。）

NVIDIA 决定将所有形式的并行都定义为 CUDA 线程（CUDA thread）。将这种最底层的并行作为编程原语，编译器和硬件可以在 GPU 上将上千个 CUDA 线程聚集起来使用各种类型的并行去执行：多线程、MIMD、SIMD 和指令级并行。这些线程被聚集成线程块，以 32 个为一组一起执行。GPU 内部的多线程处理器执行这些线程块，一个 GPU 一般由 8~32 个这种处理器组成。

6.6.1 NVIDIA GPU 体系结构简介

我们使用 NVIDIA 系统作为 GPU 体系结构的代表。特别地，我们使用 CUDA 并行编程语言中的术语并使用 Fermi 体系结构作为例子。

525 与向量体系结构一样，GPU 只对数据级并行问题才有效果。这两种体系结构都有聚集 - 分散数据传输，但是 GPU 处理器有比向量处理器更多的寄存器。与向量体系结构不同，GPU 也依赖于多线程 SIMD 处理器中的硬件多线程以隐藏访存延时（见 6.4 节）。

多线程 SIMD 处理器与向量处理器很相似，但是前者有更多的并行功能单元，而不像后者只有少数几个高度并行化的功能单元。

就像前面提到过的，一个 GPU 包含多个多线程的 SIMD 处理器；也就是说，GPU 是一个由多个多线程 SIMD 处理器组成的 MIMD 处理器。例如，NVIDIA 的 Fermi 结构有 4 种不同的配置，根据价格的不同分别含有 7、11、14 或 15 个多线程 SIMD 处理器。对于含有不同数量多线程 SIMD 处理器的 GPU，为了实现透明的伸缩性，GPU 使用线程块调度器（thread block scheduler）将线程块分配给多线程 SIMD 处理器。图 6-9 给出了一个简化的多线程 SIMD 处理器的结构图。

526 再往下深入一层，硬件产生、管理、调度并执行的机器目标代码是一个由 SIMD 指令组成的线程（thread of SIMD instruction），我们也称其为一个 SIMD 线程（SIMD thread）。它就是一个传统意义上的线程，但是它包含这些相互独立的 SIMD 指令。这些 SIMD 线程有它们自己的程序计数器并且它们运行在一个多线程 SIMD 处理器上。SIMD 线程调度器（SIMD thread scheduler）含有一个控制器，这个控制器可以告诉调度器哪些 SIMD 指令线程已经准备就绪可以执行了，并且将这些线程送给分派单元，然后分派到多线程 SIMD 处理器上执行。这个调度器同传统多线程处理器中的硬件线程调度器（见 6.4 节）一样，只是它调度的是 SIMD 指令。因此，GPU 硬件有两层硬件调度器：

- 1) 线程块调度器（thread block scheduler）将线程块分配到多线程 SIMD 处理器上。
- 2) 当 SIMD 线程准备就绪时，SIMD 处理器内部的 SIMD 线程调度器进行调度。

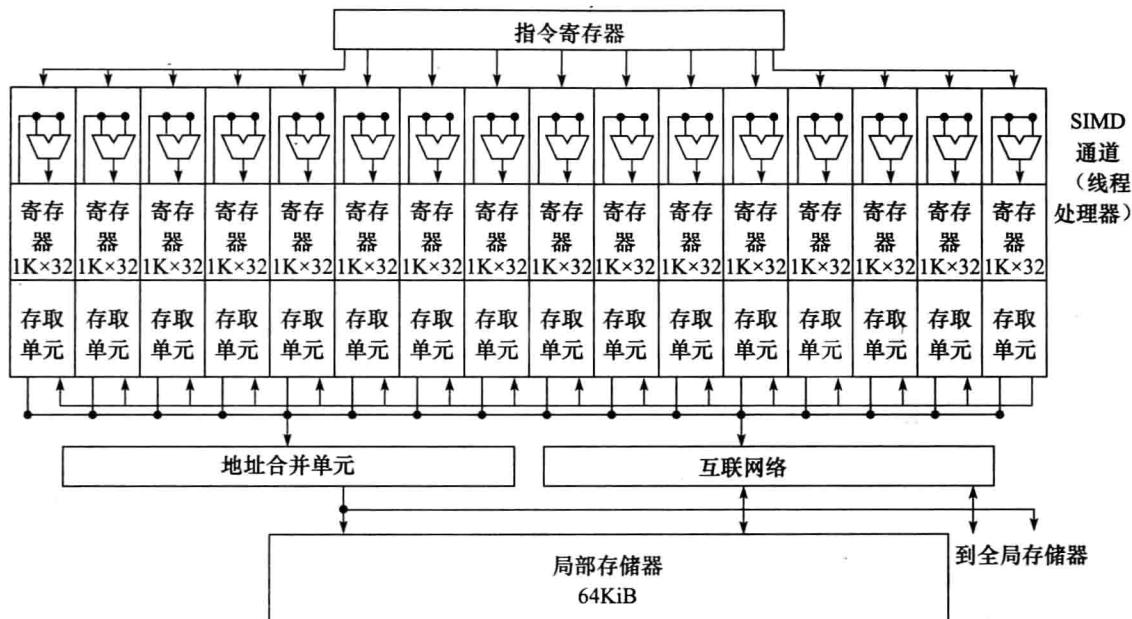


图 6-9 多线程 SIMD 处理器的数据通路的简略结构图。它有 16 个 SIMD 通道。SIMD 线程调度器用很多相互独立的 SIMD 线程来调度执行

这些线程的 SIMD 指令的宽度为 32，所以每一个 SIMD 指令线程都会对 32 个元素进行计算。由于线程是由 SIMD 指令组成，SIMD 处理器必须有并行功能单元来执行这些操作。我们称为 SIMD 通道 (SIMD lane)，它们同 6.3 节的向量通道非常相似。

01 精解 GPU 版本的不同，每个 SIMD 处理器中的通道数量也有所不同。对于 Fermi，每个 32 宽度的 SIMD 指令线程被映射到 16 个 SIMD 通道上。所以 SIMD 指令线程中的每一条指令需要两个时钟周期来完成。每一个 SIMD 指令线程都是同步执行。继续将 SIMD 处理器比作一个向量处理器，我们可以说它有 16 个通道，并且向量宽度为 32。这种宽且浅的特性使得我们称之为 SIMD 处理器而不是向量处理器，因为这样更直观一些。

根据定义，SIMD 指令线程之间是相互独立的，SIMD 线程调度器可以挑选任何准备就绪的线程去执行，而不需要考虑同一线程中指令的顺序。因此，若使用 6.4 节中的术语，则它使用的是细粒度多线程。

为了保存数据元素，一个 Fermi SIMD 处理器有着多达 32 768 个 32 位寄存器。就像向量处理器一样，这些寄存器根据向量通道（或称为 SIMD 通道）进行逻辑划分。每个 SIMD 线程有至多 64 个寄存器，所以我们可以认为一个 SIMD 线程有至多 64 个向量寄存器，且每个向量寄存器可以存放 32 个 32 位宽的数据元素。

由于 Fermi 有 16 个 SIMD 通道，所以共有 2048 个寄存器。每个线程可以从其中的一个向量寄存器中获得一个数据。注意，一个 CUDA 线程只是将一个 SIMD 指令中的线程进行纵向划分，对应于一个 SIMD 通道执行一个数据元素。请注意 CUDA 线程同 POSIX 线程非常不同；因为不能在一个 CUDA 线程中执行系统调用和同步操作。

6.6.2 NVIDIA GPU 存储结构

图 6-10 展示了一个 NVIDIA GPU 的存储结构。我们称每个多线程 SIMD 处理器专用的片上存储器为局部存储器 (local memory)。它是由同一个多线程 SIMD 处理器中的所有 SIMD 通道所共享的，但是多个多线程 SIMD 处理器之间不共享它。我们称整个 GPU 和所有线程块共享的片外存储器为 GPU 存储器 (GPU memory)。

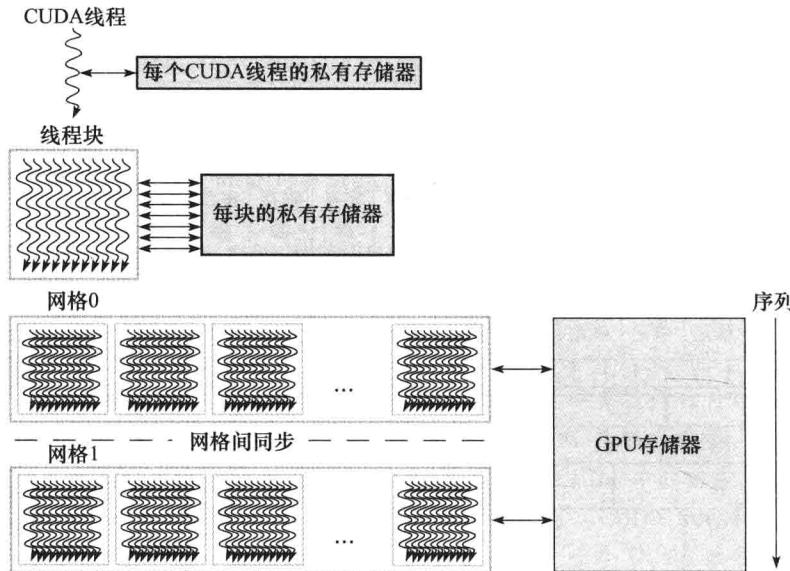


图 6-10 GPU 存储器结构。GPU 存储器被向量化的循环所共享。同一个线程块中的所有线程共享局部存储器

GPU 不依赖于大容量的 cache 来保存整个应用程序的工作集，而是依赖于小容量的流 cache 和大量的 SIMD 多线程来隐藏访问 DRAM 的访存延时，因为这些工作集通常为上百 MB。因此这些数据无法在多核处理器的最后一级 cache 中放下。为了使用硬件多线程来隐藏访问 DRAM 时的延时，就将处理器中用来放置 cache 的芯片面积在 GPU 中替换为计算资源和大量的寄存器，用来执行大量的 SIMD 线程。

01 精解 尽管隐藏访存延时是基本的原则，但是注意，最新的 GPU 和向量处理器都增加了 cache。例如，最近的 Fermi 结构增加了 cache，但是并不把它们作为为了减少访问 GPU 存储器访问次数的带宽过滤器，或为了少数不能通过多线程来隐藏访存延时的变量的加速器。为栈帧、函数调用和寄存器划分而设计的局部存储器可以被认为是 cache，因为它们的延时对函数调用有影响。cache 也对减少能耗有好处，因为访问片上的 cache 需要的能耗比访问多个片外的 DRAM 要小很多。

6.6.3 GPU 展望

在高层次上，拥有 SIMD 指令扩展的多核计算机的确与 GPU 有一些共同特点。图 6-11 总结了它们之间的相似点与不同点。尽管 GPU 有更多的处理器和更多的通道数，但是它们两者都是通过使用多个 SIMD 通道来实现 MIMD 功能的。尽管 GPU 的硬件支持更多的线程，但是两者都是通过使用硬件多线程来提高处理器利用率的。尽管 GPU 使用的是小容量的流 cache，而多核计算机使用的是尽量把整个工作集都放进去的大容量多层次结构的 cache，但是两者都有 cache 结构。尽管 GPU 的物理主存小很多，但是两者都使用 64 位的地址空间。尽管 GPU 提供

特点	使用 SIMD 的多核	GPU
SIMD 处理器数目	4 ~ 8	8 ~ 16
SIMD 通道数目 / 处理器	2 ~ 4	8 ~ 16
支持 SIMD 线程的硬件数量	2 ~ 4	16 ~ 32
最大的 cache 容量	8MiB	0.75MiB
存储器地址大小	64 位	64 位
主存容量	8GiB ~ 64GiB	4 ~ 6GiB
页面级存储保护?	是	是
需要分页?	是	否
cache 一致性?	是	否

图 6-11 带有多媒体 SIMD 扩展的多核处理器与最近的 GPU 的相似点与不同点

页面集的内存保护，但是它目前还不支持请求页面调度。

SIMD 处理器也很像向量处理器。GPU 中的多个 SIMD 处理器像独立的 MIMD 核一样工作，就像向量计算机有多个向量处理器一样。这样，可以将 Fermi GTX 580 认为是一个带有硬件多线程功能的 16 个核的机器，每个核含有 16 个通道。两者最大的区别在于多线程，多线程对于 GPU 是最基本的概念，而在向量处理器中却不存在。

GPU 结构和 CPU 结构并没有相同的祖先；并没有缺少了哪一环来解释这种现象。这种不同寻常的继承关系，使得 GPU 并没有使用 CPU 领域中常用的术语，这让人们开始对 GPU 是什么以及 GPU 是如何工作的产生困惑。为了解决这个问题，图 6-12（从左到右）列出了本节使用过的更具有说明性的术语，首先是主流计算领域中最接近的术语，然后是 NVIDIA GPU 官方的术语（如果你感兴趣），最后是该术语简单的解释。这个“GPU 罗塞塔石碑”可以将本节的内容和想法与附录 C 中介绍的更传统的 GPU 描述联系起来。·

分类	描述性名称	传统描述名称	CUDA/NVIDIA 官方术语	教科书的定义
程序抽象	可向量化循环	可向量化循环	网格	在 GPU 上执行的一个可向量化循环由一个或多个可并行执行的线程块（循环展开后的程序块）组成
	向量化了的循环体/循环展开后的程序块	(切分) 向量化循环体/循环展开后的程序块	线程块	在 SIMD 多线程处理器上执行的被展开的循环体（向量化了的循环），由一个或多个 SIMD 指令线程组成。这些线程间数据通信通过局部存储单元实现
	SIMD 单元执行序列	标量循环的一次迭代	CUDA 线程	一个 SIMD 指令线程对应于一个 SIMD 的标量元素执行单元的执行序列。其计算结果的保存依赖于屏蔽寄存器以及预测寄存器的内容
机器目标代码	SIMD 指令的一个线程	向量指令线程	Warp 块	一个仅包含 SIMD 指令的传统线程，在 SIMD 多线程处理器中执行。其结果的保存依赖于单元素屏蔽寄存器的内容
	SIMD 指令	向量指令	PTX 指令	横跨多个 SIMD 元素执行单元（SIMD Lanes，一个 SIMD 执行单元包含对应向量内标量元素数量的执行模块）上执行的一个 SIMD 指令
处理硬件	多线程 SIMD 处理器	(多线程) 向量处理器	流多线程处理器	执行 SIMD 指令线程，一个 SIMD 多线程处理器独立于其他 SIMD 处理器
	线程块调度器	标量处理器	主线程调度引擎	分配多个线程块（循环展开后的程序块）到多个 SIMD 处理器上
	SIMD 线程调度器	一个多线程处理器中的线程调度器	Warp 调度器	当 SIMD 线程可执行时，硬件单元调度并发射 SIMD 指令，调度机制由一个用于确定 SIMD 线程执行的记分牌构成
	SIMD 元素执行单元	向量元素执行单元	线程处理器	SIMD 元素执行单元执行的一个 SIMD 线程对应于向量中的单一标量元素。其计算结果的保存依赖于屏蔽寄存器的内容
存储器硬件	GPU 片外存储器	主存储器	全局存储器	GPU 中所有多线程处理器均可访问的 DRAM 存储器
	本地存储器	本地存储器	共享存储器	一个多线程 SIMD 处理器私有的快速本地 SRAM 存储器，其他 SIMD 处理器不能访问
	SIMD 元素执行单元寄存器	向量元素执行单元寄存器	线程处理器寄存器	横跨一个完整线程块（被展开的循环体），SIMD 元素执行单元内的寄存器

图 6-12 GPU 术语的快速介绍。第一列列出了硬件术语。这 12 个术语被分成 4 组。从上到下为：程序抽象、机器目标代码、处理硬件和存储器硬件

528
529

尽管 GPU 正在向主流计算进军，但是它们还是不能舍弃继续发展图形加速卡的功能。由于有很多硬件是为了加速图形处理的，因此只有当架构师开始考虑如何扩展 GPU 以使它可以进行更多类型的计算时，对 GPU 进行设计才会更有意义。

本节给出了两种共享地址空间的 MIMD 类型计算机，我们下节将要介绍每一个处理器都有自己独立的地址空间的并行处理器，这会使构建更大型的系统变得更简单。你每天都在使用的 Internet 服务就是依靠这些系统工作的。

01 精解 尽管传统上的 GPU 是具有独立内存空间的处理器，但 AMD 和 Intel 都宣称已经有可以让 GPU 和 CPU 共享一个内存的“混合”产品。对于这种混合结构来说，挑战就是如何维护一个高的存储带宽，这也是 GPU 的基本问题。

01 小测验

是非判断题：GPU 依靠图形 DRAM 芯片来减少访存延时，并以此来提高图形应用程序的性能。

6.7 集群、仓储级计算机和其他消息传递多处理器

对处理器来说，另一个共享地址空间的方法是每个处理器具有自己私有的物理地址空间。图 6-13 给出了具有多个私有地址空间的多处理器的典型组成。这种多处理器必须通过显式的消息传递（message passing）进行通信，传统上也把这类计算机称为消息传递计算机。只要系统提供发送消息例程（send message routine）和接收消息例程（receive message routine），协调工作就可以通过消息传递来完成，因为发送处理器知道何时发送消息，接收处理器也知道何时消息到达。如果发送者需要确认消息已经送达，那么接收处理器可以向发送者返回一个确认消息。

- ⌚ 消息传递：通过显式发送和接收信息的方式在多个处理器之间的通信。
- ⌚ 发送消息例程：具有私有存储器的机器中一个处理器将消息发送给另一个处理器的例程。
- ⌚ 接收消息例程：具有私有存储器的机器中一个处理器接收来自其他处理器消息的例程。

530
531

历史上曾经有过几次基于高性能消息传递网络构建大规模计算机的尝试。相对于使用局域网构建的集群，它们确实可以提供更高的性能。确实，今天很多超级计算机使用自己特有的网络。但是问题是它们比以太网这样的局域网花销更大。很少有应用程序能够为更高的性能支付更多的成本。

01 硬件/软件接口 相比于共享一致性 cache，依赖于消息传递机制的计算机对于硬件设计者来说更容易构建（见 5.8 节）。这对于编程人员来说也有好处，那就是通信都是显式的，这意味着相比于共享一致性 cache 的隐式通信来说性能方面的惊喜更少。对于编程人员的坏处是把顺序程序移植到消息传递机制的计算机上更困难，因为所有的通信都需要提前指出，否则程序就不会工作。cache 一致性共享存储器允许硬件指明哪些数据需要进行通信，这使得移植更容易。考虑到隐式通信的优点与不足，关于哪种方式最有利于获得高性能存在分歧，但

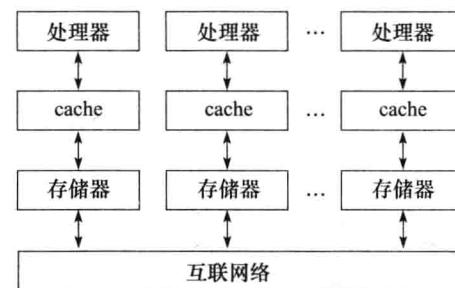


图 6-13 具有两个私有地址空间的多处理器的组成，传统上称为消息传递多处理器。与图 6-7 中的 SMP 不同，互联网络不是在 cache 和存储器之间，而是在处理器 - 存储器的节点之间

是在今天的市场上却并没有这种困惑。多核微处理器使用共享物理内存机制进行通信，而集群的节点之间使用消息传递机制进行通信。

一些并发程序可以在并行硬件上运行地很好，而与该硬件提供的是共享地址机制还是消息传递机制无关。特别的，使用任务级并行和通信比较少的应用程序——如 Web 搜索、邮件服务器和文件服务器——不需要共享地址机制就可以运行地很好。结果是，集群（cluster）已经变成了当今基于消息传递机制的并行计算机最普遍使用的例子。由于有独自的存储器，集群的每一个节点都运行操作系统的一个独立的副本。相反，微处理器的所有核在一个芯片上通过一个高速的互联网络相连，并且多个芯片共享的存储系统通过存储互联网络通信。存储互联网络有更高的带宽和更小的延时，可以让共享内存的多处理器有更好的通信性能。

- 集群：通过标准的网络开关将 I/O 链接起来的计算机集合，以此来构建消息传递机制的多处理器。

从并行编程的角度来讲，将用户内存的弱点分配到各个独立的部分提高了系统的可靠性（见 5.5 节）。由于集群是由通过局域网络连接起来的独立计算机组成，所以相比于共享存储器多处理器而言，在不影响整个系统性能的前提下替换其中的某一个计算机比较容易。从根本上讲，共享地址意味着很难在不需要大量操作系统和服务器设计进行大量工作情况下将每个处理器隔离开来并且进行替换。当一个服务器坏掉时，整个系统很容易降级，因此提高了可靠性。由于集群上的软件是运行在每个独立计算机上的操作系统中的，所以很容易断开一个计算机与互联网络的连接并且进行替换。

由于集群是由多个计算机和独立可配置的互联网络组成的，这种隔离使得在不卸载运行在集群上的软件的前提下对系统进行扩展变得很容易。

尽管与大规模共享存储器多处理器相比，集群在通信方面性能很弱，但它的低成本、高可用性和快速可扩展性使得它对于 Internet 服务提供者具有很大的吸引力。上亿人每天都在使用的搜索引擎就依赖于该技术。Amazon、Facebook、Google、Microsoft 和其他商业巨头都有多个数据中心，而每个数据中心都是由含有成千上万个服务器的集群构成。很显然，将多个处理器应用在 Internet 服务公司的做法已经获得了巨大的成功。

仓储级计算机

每个人都可以构建一个快速的 CPU，关键是如何构建一个快速的系统。

——Seymour Cray，超级计算机之父

为了支持上面提到的 Internet 服务，必须构建新的建筑、电力系统以及对 100 000 台服务器进行冷却。尽管它们可以归类为大型的集群，但是它们的体系结构和操作复杂很多。它们就像一个巨大的计算机，连接和安放 50 000 ~ 100 000 台服务器，造价、电力和冷却系统、服务器成本以及互联设备总共需要 15 亿美元。我们把这归类于一类新的计算机，叫做仓储级计算机（Warehouse-Scale Computer，WSC）。

- 硬件/软件接口 WSC 中批处理最流行的架构是 MapReduce [Dean, 2008] 和它们开源孪生兄弟 Hadoop。受 Lisp 中同名函数的启发，Map 首先对每个逻辑输入记录使用一个程序员提过的函数，然后 Map 在数以千计的服务器上运行并产生一个由关键值对组成的中间结果。Reduce 将这些分布的任务的输出结果收集起来，并使用另外一个程序员提供的函数来对其进行压缩。通过适当的软件处理，这两部分可以高度并行化并且易于使用。在

30 分钟之内，一个编程新手就可以在上千个服务器上运行 MapReduce。

例如，一个 MapReduce 程序要计算一大堆文档中每个单词出现的次数。下面是该程序的一个简化版本，它只给出了最内层的循环，并且假设在一个文档中所有单词只出现一次。

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1"); // Produce list of all words
    reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each v in values:
            result += ParseInt(v); // get integer from key-value pair
        Emit(AsString(result));
```

533

Map 函数中使用的 EmitIntermediate 函数可以将文档中的每一个单词输出，同时还带有一个 1 值。然后 Reduce 函数逐个将同一个单词的所有 value 值加起来，这其中使用 ParseInt() 函数得到所有文档中每个单词的 value 值。MapReduce 运行时环境将 map 任务和 reduce 任务调度到 WSC 中的服务器上。

在这种极端的规模下，WSC 需要在电源分布、冷却、监控和操作上都做出创新，WSC 可以算是 20 世纪 70 年代的超级计算机的后代——这使得 Seymour Cray 成为了当今 WSC 体系结构之父。他的超级计算机可以解决其他计算机无法解决的问题，但是却过于昂贵，以至于只有少数几个公司有能力购买它。而现在 WSC 的目标是为全世界提供信息技术，而不是专门为科学家和工程师提供高性能计算。因此，WSC 在今天的社会中扮演着比 Cray 的超级计算机在那个年代更加重要的角色。

尽管它们都与服务器在目标上有一些共同点，但是 WSC 还是有以下 3 点主要区别：

1) 大量简单的并行：对于服务器架构师，一个需要着重考虑的因素是目前市场上的应用程序是否具有能在并行硬件上运行得足够的并行性，以及为了发掘这些并行性而使用足够多的硬件是否代价过于高昂。但是 WSC 的架构师没有这方面的顾虑。首先，像 MapReduce 这样的批处理程序可以从大量需要独立处理的数据集中获利，例如网页抓取中数以亿计的网页。其次，交互式 Internet 服务应用，也称作软件即服务（software as a service, SaaS），可以从数以百万计的相互独立的交互式 Internet 服务用户中获利。在 SaaS 中，读和写之间依赖关系很少，所以 SaaS 基本上不使用同步操作。例如，查找操作只使用一个只读的索引，电子邮件通常是读和写独立的信息。我们将这种简单的并行称作请求级并行（request-level parallelism），因为很多独立的工作可以很自然地执行，并且只需要很少的通信和同步操作。

2) 软件即服务：相比于出售那些安装并运行在用户计算机上的软件，软件是运行在远程的一个站点上的，并且是通过 Internet（通常是面向用户的网页接口）向用户提供服务。SaaS 用户是基于是否使用而非是否拥有来收费的。

3) 经营成本：传统上，服务器架构师通常在有限的开销上设计可以达到最佳性能的系统，并且要时刻确保冷却系统的容量不会超过它的额定面积。他们经常忽略服务器的经营开销，就像经营开销和购买的开销比起来暗淡无光一样。WSC 拥有更长的寿命——建筑和电力以及冷却系统通常使用超过 10 年的分期付款来购买——所以经营成本加起来：能耗、电源分布和冷却系统在这 10 年中总共超过 WSC 价格的 30%。

4) 规模以及规模带来的机遇和问题：为了建造一个 WSC，你必须购买 100 000 台服务器

以及相关的设施，这意味着会有总额折扣。因此，WSC 的内部如此庞大以至于即使没有很多的 WSC 你也会得到规模经济。这种规模经济导致了云计算（cloud computing）出现，因为每个单元更低的花费意味着云计算公司可以用比用户自己购买这些服务更低的价格将服务出租给用户以获得利润。规模经济的负面影响就是需要解决这种规模下的故障率。即使一个服务器有着高达 25 年（200 000 小时）的平均无故障时间（mean time to failure），WSC 架构师也要考虑每天会出现 5 个服务器故障的可能。5.15 节提到了 Google 测试得到的年均磁盘故障率（AFR）为 2% ~ 4%。如果每个服务器有 4 个磁盘并且它们的年均故障率为 2%，WSC 架构师应该每小时都发现一个磁盘故障。因此，容错性对于 WSC 架构师比服务器架构师要重要得多。

WSC 带来的规模经济使得长久以来人们梦寐以求的将计算变成一种设施的梦想成为现实。云计算意味着任何有好的想法、一个商业模式以及一个信用卡的人都可以在任何地点使用数以千计的服务器向全世界传播他的想法。当然，也存在阻挡云计算发展的障碍——例如安全、隐私、标准化以及 Internet 带宽的增长率——但是我们可以预见这些障碍都会被解决，并且 WSC 和云计算终将繁荣起来。

考虑到云计算的发展率，2012 年亚马逊网络服务宣布它每天都会增加相当于 2003 年亚马逊的所有设施的服务器数量，那时候亚马逊是一个拥有 6 000 名员工的年收入额为 52 亿美元的公司。

对于云计算而言，现在我们理解了消息传递机制多处理器的重要性，接下来要介绍将 WSC 中个节点连接起来的方法。归功于摩尔定律和每芯片上不断增加的核数，我们现在在芯片内部也需要互联网络，所以这些拓扑结构无论在小规模计算机上还是大规模计算机上都很重要。

01 精解 MapReduce 架构在 Map 阶段的最后将关键值对进行移动和分类，并生成所有共享相同关键值的组。然后这些组会被传递给 Reduce 阶段。

01 精解 另一种大规模计算是网格计算（grid computing），所有计算机被分布在很大的范围之内，运行于其上的软件通过长距离的网络进行通信。最常见并且特殊的网格计算形式要属 SETI@ home 项目设计的。当数以百万计的计算机在空闲的时候，若有人开发了一个可以使用这些计算机的软件，并且将能够将任务分成独立的部分分配给这些计算机运行，这些计算机就会被征集起来并获得很好的利用。最早的一个例子是搜索地外文明的项目（Search for ExtraTerrestrial Intelligence，SETI），该项目是在 1999 年在加州大学伯克利分校启动的。超过 200 个国家的 500 万计算机用户签署了 SETI@ home 项目，其中有超过 50% 的非美国用户。到 2011 年年底，SETI@ home 网格计算的平均性能为 3.5PetaFLOPS。

01 小测验

1. 是非判断题：同 SMP 相同，消息传递机制的计算机基于锁来进行同步操作。
2. 是非判断题：集群有独立的存储器，所以每个都有一份操作系统的副本。

6.8 多处理器网络拓扑简介

多核芯片需要使用片上网络将各个核连接到一起，集群需要局域网将服务器连接到一起。本节讨论不同互联网络拓扑的优点与缺点。

网络成本包括开关的数量、每个开关连接到网络上的链路数量、每条链路的宽度（比特数）以及网络映射到芯片时链路的长度。例如，某些核或服务器可能是相邻的，而其他的可能

在芯片或数据中心的另一端。网络性能也是多方面的。它包括在一个无负载的网络中发送和接收消息的延迟，按照在给定时间周期内能够传输的最大消息数量所给出的吞吐量，由于网络冲突导致的延迟，以及由通信模式决定的可变性能。网络的另一责任是容错，因为系统可能需要在存在部件受损的情况下继续工作。最后，在这芯片设计受能耗限制的时代里，不同组织结构具有不同功效，能耗可能超越其他考虑因素而成为主导因素。

网络通常绘制为图形表示，图中的每条边表示通信网络中的一条链路。在本章的图中，处理器-存储器节点用一个黑色方块表示，而开关用一个灰色圆形表示。我们假设所有链路都是双向的；也就是说，信息可以向两个方向流动。所有网络都由开关构成，开关负责建立处理器-存储器节点和其他开关的链接。第一个网络将若干节点组成的序列连接到了一起：



该拓扑叫作环（ring）。由于一些节点不是直接连接的，一些信息将不得不经过中间节点最终到达目标节点。

和总线不同——总线允许一组连线向所有相连的节点发送广播——环可以同时进行多个传输。
536

因为有众多的拓扑可以选择，所以需要辨别这些不同设计的性能度量。主要有两个常用的性能度量。第一个是总网络带宽（network bandwidth），它是每条链路带宽与链路数量的乘积。该度量表示带宽的峰值。对于上面的环网络，如果处理器数量为 P ，那么总网络带宽就是一条链路带宽的 P 倍；一条总线的总网络带宽仅仅是该总线的带宽。

- ② 网络带宽：非正式用语，用于表示网络传输速度的峰值；既可以指单一链路的速度，也可以指网络中全部链路的共同的传输速度。

为了不只评估最好情况下的性能，我们引入一个接近于最差情况的度量：切分带宽（bisection bandwidth）。它的计算是通过将机器分割为两半，然后将跨越假想分割线的链路带宽加起来。环的切分带宽是链路带宽的两倍，是总线链路带宽的一倍。如果单一链路和总线一样快，那么环在最差情况下是总线速度的两倍，而在最好情况下是总线的 P 倍。

- ② 切分带宽：多处理器中两个相等部分之间的带宽。这种测量可以表示对多处理器的最差拆分情况。

某些网络拓扑是非对称的，那么在切分网络时会产生一个问题：在哪里进行假想切分。由于切分带宽是一个针对最差情况的度量，因此答案就是选择会导致最差网络性能的切分方式。换句话说，就是计算所有可能的切分带宽，然后选择其中最小的一个作为最终结果。我们之所以选择这种最差情况，是因为并行程序常常受通信链中最薄弱链路的限制。

相对于环的另一个极端是全连接网络（fully connected network），其中每个处理器都与其他处理器具有一个双向链路。对全连接网络，总网络带宽是 $P \times (P - 1)/2$ ，而切分带宽是 $(P/2)^2$ 。

- ② 全连接网络：通过专用通信链路连接所有处理器-存储器节点的网络。

全连接网络对性能的极大提升被成本的急剧增加所抵消了。这激励工程师不断创造出介于环的成本和全连接网络的性能之间的新型拓扑。评估是否成功，很大程度上依赖于计算机上所运行的并行程序负载的通信特征。

各种公开发布的不同拓扑可能难以计数，但是只有少数几个已经用于商业并行处理器中。图 6-14 给出了两种常见拓扑。

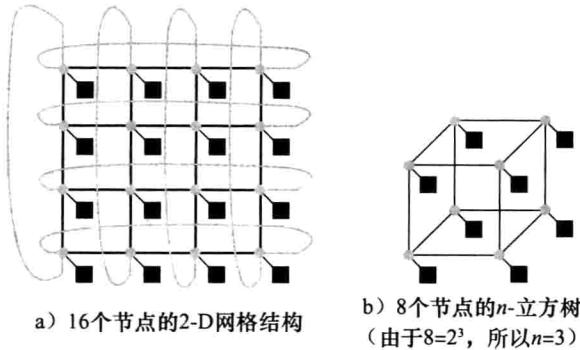


图 6-14 已经出现在商业并行处理器中的网络拓扑。其中灰色圆形表示开关，而黑色方块表示处理器 - 存储器节点。尽管一个开关可以有多个链路，但是通常只有一个连接到处理器。布尔 n 维立方体拓扑是一个使用 $2n$ 个节点构成的 n 维互连，每个开关需要 n 个链路（并加上一个处理器链路），因而存在 n 个最近相邻节点。这些基本拓扑常常会补充一些额外链路，从而提高性能和可靠性

除了在网络中每个节点都放置一个处理器之外，也可以在某些节点只保留开关。这些开关相对处理器 - 存储器 - 开关节点更小，因此可以放置得更密集一些，进而缩短距离提高性能。这样的网络一般称为多级网络 (multistage network)，因为信息需要多级传输才能到达目的地。多级网络的类型和单级网络是一样的；图 6-15 给出了两种常见的多级结构。全连接网络或交叉开关网络 (crossbar network) 允许任何节点一次就可以通过网络与其他任何节点通信。Omega 网络相对交叉开关网络使用更少的硬件（前者需要 $2n\log_2 n$ 个开关，后者需要 n^2 个开关），但是消息之间可能会发生冲突，这取决于通信模式。例如，图 6-15 中的 Omega 网络在从 P_0 向 P_6 发送信息的同时，不能从 P_1 向 P_4 发送信息。

- ② 多级网络：每个节点提供一个小开关的网络。
- ② 交叉开关网络：任何一个需一次即可与其他任意一个节点通信的网络。

537

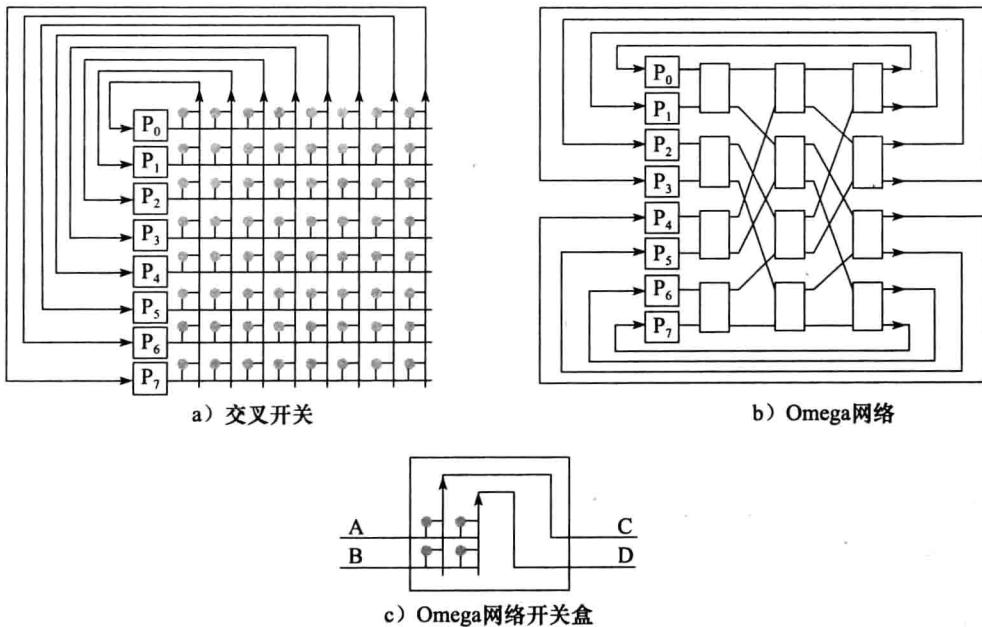


图 6-15 常见的八节点多级网络拓扑。本图中的开关相对前面的更加简单，因为本图的链路是单向的；数据从左边进入，从右边的链路退出。图 c 中的开关盒可以将 A 传送到 C、将 B 传送到 D，或将 B 传送到 C、将 A 传送到 D。交叉开关使用 n^2 个开关，其中 n 是处理器的数量，而 Omega 网络需要 $2n\log_2 n$ 个大的开关盒，其中每个开关盒逻辑上由 4 个更小的开关组成。在这种情况下，交叉开关网络需要 64 个开关，而 Omega 网络需要 12 个开关盒，相当于 48 个开关。但是，交叉开关网络可以支持处理器消息传递的任意组合，而 Omega 网络却不能

网络拓扑实现

本节对所有网络简单分析的时候，忽略了在网络构建时需要考虑的实际因素。在高速时钟下，链路的距离影响通信的成本——一般来说，距离越长，在高速时钟下的成本越大。较短的距离也会更加容易地将更多的连线增加到同一链路中，因为连线越短，驱动连线的能耗就会越低。较短的连线也比较长的连线便宜。另外一个实际限制是三维拓扑连线必须映射到芯片的二维媒介上。最后一点需要考虑的是能耗。例如，能耗可能迫使多核芯片必须采用简单网格拓扑。总之，在黑板上画上很美的拓扑，在使用硅工艺或数据中心制造时可能是不切实际的。

现在我们已经了解了集群的重要性，并且看到了将它们连接起来的方法，接下来我们要看一看网络与处理器的软硬件接口。

01 小测验

538

是非判断：对于一个有 P 个节点的环，总网络带宽与切分带宽的比为 $P/2$ 。

6.9 与外界通信：集群网络

这个网上章节讲述了用来连接集群节点的网络硬件和软件。例子中使用了采用 PCIe 连接到计算机上的 10Gb/s 的以太网。这个例子展示了软硬件优化如何提升互联网络的性能，包括零拷贝消息传递、用户空间通信、使用轮询机制代替 I/O 中断以及使用硬件计算校验总和。尽管例子是互联网络，但是本节介绍的这些技术也可以应用到存储控制器和其他的 I/O 设备。

539

在这个网上章节中从底层详细讲述了互联网络的性能后，下节从更高的层次介绍了如何测试评价各种类型的多处理器。

6.10 多处理器测试集程序和性能模型

在第 1 章中我们看到，基准测试系统一直是一个敏感话题，因为它是判断哪个系统更好的一种最为直观的方式。测试结果不仅影响商业系统的销售，而且影响这些系统设计者的声誉。因此，每个参加测试者都希望自己获胜，但是如果别人获胜，他们也希望确信获胜者的系统真正是一个更好的系统。这些期望导致测试结果不能只是针对测试程序的简单伎俩，而应该能够真正促进实际应用程序性能的提高。

为了避免可能的作弊，一个典型的原则是你不能修改基准测试程序。源代码和数据集是固定的，并且只有唯一的正确结果。对这些原则的任何违反都会使得测试结果无效。

许多多处理器基准测试程序都遵守这些惯例。一个共同的例外是允许增加问题规模，这样你就可以在有不同数量处理器的系统上运行。也就是说，许多基准测试程序允许弱比例缩放而不是强比例缩放，但即便如此，在比较不同问题规模的程序结果时仍要小心。

图 6-16 是对几种并行基准测试程序的总结。描述如下：

- Linpack 是一组线性代数例程，这些例程执行高斯消元。前面示例中给出的 DGEMM 例程就是 Linpack 基准测试程序中的一小部分代码片段，但是它占用了该基准测试程序的大部分执行时间。它允许弱比例缩放，让用户选择任何规模的问题。而且，它允许使用者以几乎任何形式和任何语言重写 Linpack，只要保持计算结果的正确性以及对于同样规模大小的问题进行相同次数的浮点运算。每隔两年计算 Linpack 最快的 500 台计算机会公布在 www.top500.org 上。排名第一的被新闻界认为是世界上最快的计算机。
- SPECrate 是一个基于 SPEC CPU 基准测试程序（如 SPEC CPU 2006，见第 1 章）的吞吐量度量。SPECrate 不是报告单个程序的性能，而是同时运行该程序的很多副本。因此，它主要测量任务级并行，因为这些任务之间没有通信。程序的副本数量是不受限制的，因此这也是弱比例缩放的形式。

540

- SPLASH 和 SPLASH 2 (Stanford Parallel Applications for Shared Memory) 是 20 世纪 90 年代斯坦福大学的研究成果，目的是提供类似于 SPEC CPU 一样的并行基准测试程序。它由核心程序和应用程序构成，许多都来自高性能计算领域。尽管该程序提供了两组数据集，但仍需要强比例缩放。
- NAS (NASA Advanced Supercomputing) 并行基准测试程序是 20 世纪 90 年代以来对多处理器基准测试程序的另一尝试。它由 5 个核心程序构成，都是来源于流体动力学。它允许通过定义几个数据集实现弱比例缩放，像 Linpack，这些基准测试程序可以被重写，但是编程语言只能使用 C 或 Fortran。
- 最近的 PARSEC (Princeton Application Repository for Shared Memory Computer) 基准测试程序集由采用 Pthread (POSIX 线程) 和 OpenMP (Open MultiProcessing, 见 6.5 节) 的多线程程序组成。它们主要专注于计算领域，由 9 个应用程序和 3 个核构成。其中 8 个依赖数据并行，3 个依赖流水并行，另外一个依赖非结构化并行。
- 在云的前端，Yahoo! Cloud Serving Benchmark (YCSB) 的目标是比较云数据服务的性能。它通过使用 Cassandra 和 HBase 作为具有代表性的例子，提供了一个易于让用户评测新数据服务的框架。[Cooper, 2010]

⑤ Pthread：创建和操作线程的一个 UNIX API。它被组织成一个库的形式。

基准测试程序	可扩展性	可编程性	描述
Linpack	弱	是	稠密矩阵线性代数 [Dongarra, 1979]
SPECrate	弱	否	独立任务并行化 [Henning, 2007]
面向共享存储器的 斯坦福并行应用 SPLASH2 [Woo et al. 1995]	强（虽然提供了 两种问题规模）	否	复杂 1D FFT 模块化 LU 分解 模块化稀疏丘拉斯基分解 整数基数排序 巴尔内斯小屋 适应性快速多极算法 海洋仿真 光线跟踪 声音渲染器 空间数据结构的水仿真 非空间数据结构的水仿真
NAS 并行 Benchmark [Bailey et al., 1995]	弱	是（只能是 C 或 Fortran）	EP：非空间数据结构的水仿真 MG：简化的多重网格计算 CG：面向共轭梯度方法的非结构化网格 FT：使用 FFT 的 3-D 偏微分方程 IS：大型整数排序
PARSEC Benchmark 集 [Bienia et al. 2008]	弱	否	Blackscholes——使用毕苏期权定价模式的期权定价 Bodytrack——人体跟踪 Canneal——使用 cache 感知的模拟退火进行布线优化 Dedup——采用数据去重的下一代压缩 Facesim——人脸运动仿真 Ferret——内容相似性搜索服务器 Fluidanimate——SPH 方法的流体动力学动画 Freqmine——常见物品集合的数据挖掘 Streamcluster——输入流的在线分类 Swaptions——Pricing of a portfolio of swaptions Vips——图像处理 x264——H. 264 视频编码

图 6-16 并行基准测试程序的实例

基准测试程序	可扩展性	可编程性	描述
伯克利设计数据集 [Asanovic et al. 2006]	强或弱	是	有限状态机 组合逻辑 图的遍历 结构化网格 稠密矩阵 稀疏矩阵 波谱法 动态程序设计 N 体问题 MapReduce：（云计算架构的一个专有名词） 反向跟踪/分支与边界 图模型推导 非结构化网格

图 6-16 （续）

基准测试程序原有约束所造成的负面影响是创新被局限到体系结构和编译器中。更好的数据结构、算法、编程语言等通常不能使用，因为这些可能导致容易误解的结果。这样系统可能不是由于硬件或编译器的原因获得更高性能，例如算法。

尽管这些准则在计算基础相对稳定时是可以理解的——因为它们是在 20 世纪 90 年代提出的，而且是在 90 年代的前 5 年——但是，这些准则在编程变革中就不合时宜了。为了变革的成功，我们需要鼓励在所有层次上的创新。

加利福尼亚大学伯克利分校的研究人员提出了一个最新的方法。他们确定了 13 个面向未来应用程序的设计模式。这些设计模式使用框架或核心实现。一些实例包括稀疏矩阵、结构化网格、有限状态机、映射规约和图遍历等。通过将定义保持在高级别层次，他们希望鼓励在系统的任何层次进行创新。因此，速度最快的稀疏矩阵求解器的系统除了使用新型体系结构和编译器之外，还可以使用任何数据结构、算法和编程语言。

6.10.1 性能模型

和测试集程序相关的一个话题就是性能模型。就像我们在本章中看到的不断增加的体系结构多样性——多线程、SIMD、GPU——如果我们能拥有一个简单的模型来分析不同体系结构设计的性能，将是十分有益的。这个模型不需要是完美的，只要有所见地就行。

第 5 章的对于 cache 性能测试的 3C 模型是一个性能模型的例子，它并不是一个完美的性能模型，因为它忽略了一些潜在的重要因素，如块尺寸大小、块分配策略和块替换策略。而且，它还含有一些含糊其辞的地方。例如，在一个设计中缓存缺失的原因可能是因为容量，但在另一个相同大小的缓存中可能是因为冲突。然而 3C 模型已经流行了 25 年，因为它提供了深刻理解程序行为的一个途径，有助于体系结构设计者和程序员基于模型的洞察来改进他们的创新。

为了找到这样一个并行计算机的模型，让我们从小的核心程序开始，就像图 6-16 中的 13 个 Berkeley 设计模式。尽管这些核心程序的不同数据类型有许多版本，但是浮点在几种实现中最常见的。因此，在给定的计算机上峰值浮点性能是这类核心程序的速度瓶颈。对于多核芯片，峰值浮点性能是芯片上所有处理器核峰值性能的总和。如果系统中包含多处理器，那么应当将每芯片的峰值性能与芯片数量相乘。

对存储器系统的需求可以用峰值浮点性能除以每访问一字节所包含浮点操作数的平均值来估算：

$$\text{浮点操作数 / 秒} \div \text{浮点操作数 / 字节} = \text{字节 / 秒}$$

存储器每访问一字节所包含的浮点操作比例称作算术密度 (arithmetic intensity)。它的计算可以用程序中总的浮点操作数除以程序执行期间主存传输数据总的字节数。图 6-17 给出了图 6-16 中几种 Berkeley 设计模式的算术密度。

- 算术密度：一个程序中浮点操作数量与访问主存字节数量的比值。

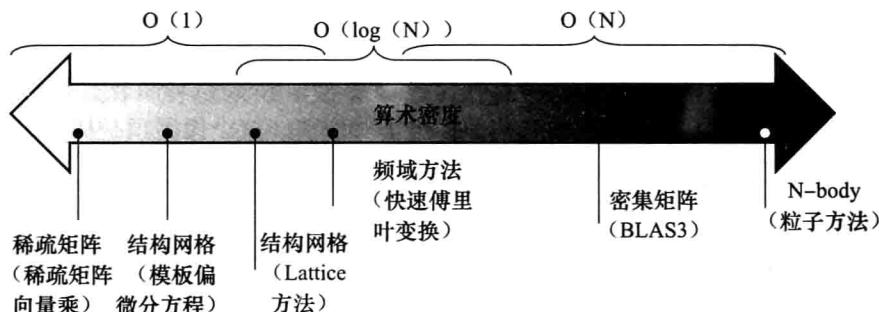


图 6-17 算术密度，计算方式为用运行程序中总的浮点操作数除以访问主存总的字节数 [Williams, Patterson, 2009]。一些核心程序的算术密度与问题规模成比例扩展，如 Dense Matrix，但是也有许多核心程序与问题规模无关。对于前者，弱比例缩放会导致不同的结果，因为它对存储系统的需求不是很大

543

6.10.2 Roofline 模型

本节提出的简单模型将浮点性能、算术密度和存储性能联系在一个二维图中 [Williams, Waterman, Patterson, 2009]。峰值浮点性能可以在上面谈到的硬件规格说明书中找到。我们这里考虑的核心程序的工作集不适合使用片上缓存，因此峰值存储器性能可以使用缓存之后的存储器来定义。获得峰值存储性能的一种方法是使用 Stream 基准测试程序。（见第 5 章的精解。）

图 6-18 给出了针对一台计算机的模型，注意不是针对每个核心程序的模型。纵轴 Y 表示浮点性能，从 0.5 到 64.0 GFLOPs/s。横轴 X 表示算术密度，从 1/8 FLOPs/DRAM 字节到 16 FLOPs/DRAM 字节。注意该图采用 log-log 的比例。

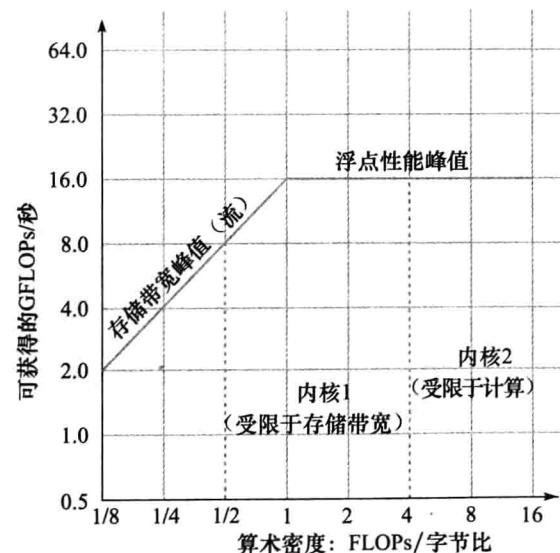


图 6-18 Roofline 模型 [Williams, Waterman, Patterson, 2009]。本例具有 16GFLOPS/s 的峰值浮点性能和 16GB/s 的峰值存储带宽，该数据来自流测试程序（由于流实际上是 4 次测量，图中的线是 4 次的均值）。左边的灰色点垂线标识内核 1，其计算密度为 0.5FLOPs/byte。在 Operon X2 上，受限于低于 8GFLOPS/s 的存储器带宽。右边的点垂线标识内核 2，计算密度为 4FLOPs/byte，它只受限于 16GFLOPS/s 的计算。（该数据基于 AMD Opteron X2 (版本 F)，使用运行在双socket系统中的 2GHz 的双核）

对给定的核心程序，我们可以基于其算术密度在 X 轴找到对应点。如果我们在该点画一条垂直线，那么该核心程序在计算机上的性能一定在该垂直线的某个位置上。我们可以画一条水平线表示该计算机的峰值浮点性能。显然，实际的浮点性能不会超过该水平线，因为这是一个硬界限（hardware limit）。

544

我们如何画出峰值存储性能呢（单位为字节/秒）？由于 X 轴是 FLOPs/byte，Y 轴是 FLOPs/s，所以 byte/s 只是图中一条 45° 的对角线。因此，我们画出第三条线来表示对于给定的算术密度该计算机存储系统所能支持的最大浮点性能。我们可以用下面的公式表示该界限，以便在图 6-18 中画出该线：

$$\text{可达到的 GFLOPs/s} = \text{Min}(\text{峰值存储器带宽} \times \text{算术密度}, \text{峰值浮点性能})$$

水平线和对角线给出了简单模型的名称并标出了对应值。这个像屋顶一样的轮廓线设定了一个核心程序在不同算术密度下的性能上界。给定一个计算机的 Roofline 模型，你可以重复地使用它，因为它不会随核心程序而变化。

如果我们认为算术密度是支撑屋顶的一个杆，那么它要么支撑屋顶的平坦部分，这表示性能受计算限制；要么支撑屋顶的倾斜部分，这表示性能受存储器带宽限制。在图 6-18 中，核心程序 1 属于前者，而核心程序 2 属于后者。

需要注意的是“脊点”，它是屋顶平坦部分与倾斜部分的交叉点，这对计算机来说是一个关键点。如果它过于靠右，那么只有极高算术密度的核心程序才能获得最大性能。如果它过于靠左，那么几乎所有核心程序都可以达到最大性能。

6.10.3 两代 Opteron 的比较

四核的 AMD Opteron X4（Barcelona）是两核 Opteron X2 的后续版本。为了简化主板设计，它们使用了相同的插座。因此，它们具有相同的 DRAM 通道，也就具有相同的峰值存储带宽。除了将核心程序数量加倍之外，Opteron X4 还将每核的峰值浮点性能提高到原来的两倍：Opteron X4 核每时钟周期可发射两条浮点 SSE2 指令，而 Opteron X2 核最多只能发射一条。由于我们比较的两个系统具有接近的时钟频率——Opteron X2 为 2.2GHz，Opteron X4 为 2.3GHz——所以 Opteron X4 的峰值浮点性能是 Opteron X2 的 4 倍，而两者 DRAM 带宽完全相同。Opteron X4 还有 2MiB 的三级缓存，而 Opteron X2 没有。

图 6-19 比较了两个系统的 Roofline 模型。正如我们所期望的那样，脊点向右进行了移动，从 Opteron X2 的 1 移到了 Opteron X4 的 5。因此，为了看到下一代 Opteron 处理器性能的改进，核心程序的算术密度必须大于 1，或者核心程序的工作集必须适合 Opteron X4 的缓存。

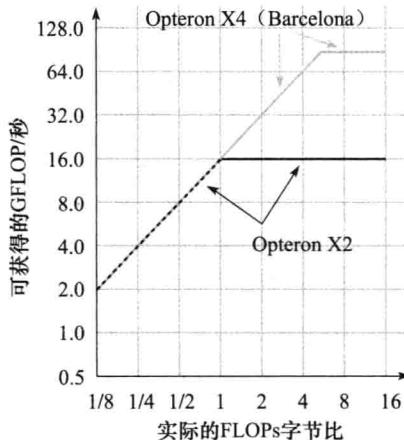


图 6-19 两代 Opteron 的 Roofline 模型。Opteron X2 的屋顶线与图 6-18 相同，使用黑色绘制，而 Opteron X4 的屋顶线使用灰色绘制。Opteron X4 更大的脊点意味着原来在 Opteron X2 中是计算受限的核心程序在 Opteron X4 中可能是存储性能受限

Roofline 模型给出了性能的上界。假设你的程序远远低于该上界，那么你应进行哪些优化呢？这些优化的优先级顺序是什么？

为了克服计算瓶颈，下面的两种优化可以改进几乎任何核心程序：

1) 浮点操作混合。对一台计算机而言，峰值浮点性能一般需要几乎同时到达的等量加法和乘法。这种均衡不仅是因为计算机支持融合的乘加指令（见第 3 章的精解），也因为浮点单元具有相同数量的浮点加法器和浮点乘法器。最佳性能也需要大部分浮点操作和非整数指令混合。

2) 提高指令级并行并应用 SIMD。对当代的体系结构，最高性能在每个时钟周期取指、执行并提交 3~4 条指令时才能获得（见 4.10 节）。这一步的目标是从编译器上改进代码来增加 ILP。一种方法是循环展开，就像我们在 4.12 节看到的。对 x86 体系结构而言，一个单一的 AVX 指令可以对 4 个双精度操作数进行操作，因此它们应该被尽量使用（见 3.7 节和 3.8 节）。

为了克服存储瓶颈，可以采用下面的两种优化方法：

1) 软件预取指（software prefetching）。最高性能通常需要保持许多存储器操作一直运行，这使得通过执行软件预取指令来预测访存更加容易，而不用等到计算需要该数据时才进行访存。

2) 内存关联（memory affinity）。现在大多数的微处理器都在片内包含了内存控制器，它能提高存储器层次的性能。如果系统中含有多个芯片，这就会使一些地址访问本地 DRAM，而其他地址需要通过芯片互连才能访问对于其他芯片是本地的 DRAM。这种分割导致了我们在 6.5 节介绍的非一致性访存。通过另一个芯片进行访存会降低性能。第二种优化方法是分配数据后尽量让线程操作属于同一存储器-处理器对上的数据，这样处理器几乎不会访问其他芯片上的存储器。

Roofline 模型可以帮助决定选用哪些优化，以及优化的实施顺序。我们可以认为这些优化方法中的每一个都是适当屋顶线下面的一层“天花板”，也就是说，在没有实施相应优化的情况下不能突破该层天花板。

计算性能屋顶线可以在手册中找到，而存储屋顶线则可以通过运行流基准测试程序获得。计算性能天花板，如浮点均衡，也可来自该计算机的手册。存储天花板，例如存储器关联，需要在每台计算机上运行实验，从而决定它们之间的间隙。一个好消息是这一过程在每台计算机上只需进行一次，只要有人完成了对该计算机天花板的评估，任何人都可以将该结果用于指导该计算机优化的先后次序。

图 6-20 相对于图 6-18 中的屋顶线模型增加了天花板，其中上图给出了计算天花板，下图给出了存储带宽天花板。尽管较高的天花板没有标记，但是其隐含使用了全部优化手段；为了突破最高的天花板，首先必须突破所有下面的天花板。

天花板之间间隙的宽度和下一个更高的限制表示优化之后的收益。因此，图 6-20 建议优化 2 和 4。其中 2 是改善 ILP，对于改善该计算机的计算有很大益处；4 是改善内存关联，对于改善该计算机的存储带宽有很大益处。

图 6-21 将图 6-20 中的天花板整合到一张图中。核心程序的算术密度决定了优化的区域，优化区域反过来又给出了哪些优化手段可以尝试。需要注意的是，对大多数算术密度，计算优化和存储带宽优化都是重叠的。图 6-21 中有三处不同的阴影标记，用于区分不同的优化策略。例如，核心程序 2 落在右边灰色梯形区域，表示只工作在计算优化上。核心程序 1 落在灰色与浅灰色平行四边形区域，表示两种优化均可尝试。而且，它建议从优化 2 和优化 4 开始。注意到核心程序 1 的垂直线低于浮点失衡优化，因此优化 1 是没有必要的。如果核心程序落在左下角的浅灰色三角形区域，则表示只需进行存储优化即可。

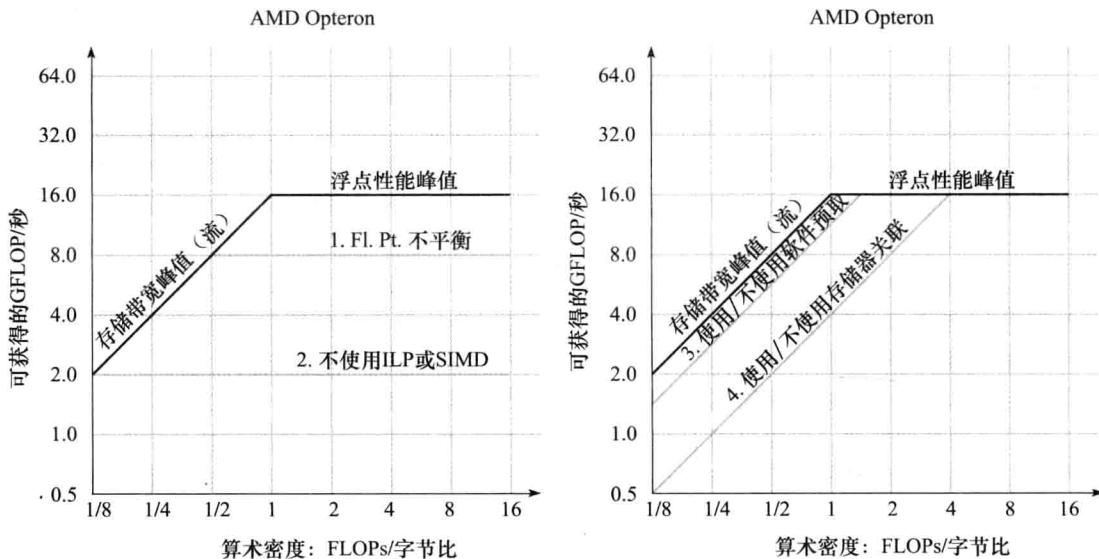


图 6-20 带有天花板的 Roofline 模型。其中上面的图表示计算性能的“天花板”，1 表示浮点操作混合失衡情况下性能为 8GFLOPs/s，2 表示同时未使用 ILP 和 SIMD 下的性能为 2GFLOPs/s。下面的图表示存储带宽的天花板，其中 3 表示没有软件预取指时的带宽为 11GB/s，4 表示同时没有优化内存关联的带宽为 4.8GB/s

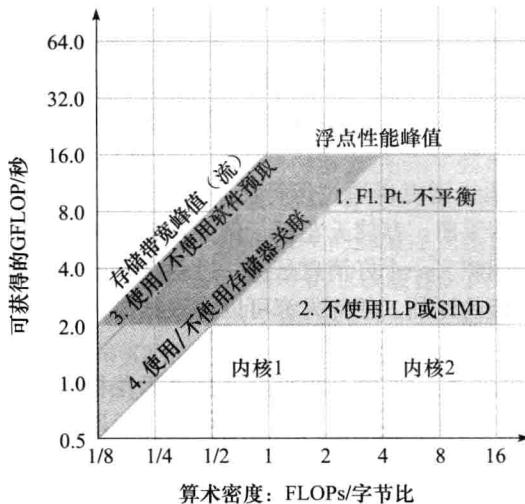


图 6-21 将图 6-18 中两图重叠的 Roofline 模型。算术密度处于右边灰色梯形区域的核心程序应当着重于计算优化，而处于浅灰色三角形区域的核心程序应当着重于存储带宽优化。处于灰色和浅灰色平行四边形区域的核心程序两种优化都应当考虑。例如核心程序 1 落在中间的平行四边形中，可尝试优化 ILP 和 SIMD、内存关联、软件预取指等。核心程序 2 落在右边的梯形区域，可尝试优化 ILP 和 SIMD 以及浮点操作均衡

到目前为止，我们一直假定算术密度是固定的，但是实际情况并非如此。首先，有些核心程序的算术密度会随问题规模增长，如稠密矩阵和多体问题（见图 6-17）。事实上，这就是程序员处理弱比例缩放比强比例缩放更成功的原因之一。第二，存储器层次结构的效应影响存储器的访问次数，因此改善缓存性能的优化也能改善算术密度。一个例子是通过循环展开，并将使用相近地址的语句分组到一起来改善时间局部性。许多计算机提供特殊的缓存指令，故可以先将数据分配到缓存中，而不用先从存储器中填充，因为它可能很快被改写。这些优化降低了存储器流量，因此可以将算术密度乘以一个系数（如 1.5）向右移动。这种右移会使核心程序移到一个不同的优化区域。

虽然上面的例子展示的是如何帮助程序员提高程序的性能，但同时架构师也可以利用这个

模型决定硬件的哪些部分应该优化，以提升他们认为重要的核心程序的性能。

下一节使用 Roofline 模型分析一个多核微处理器和一个 GPU 的性能差异，以及这些差异是否反映了真实程序的性能。

01 精解 天花板是分层次的，最低的天花板是最容易优化的。显然，程序员可以按任意顺序优化，但是遵从建议的顺序可以避免将时间浪费在因其他约束而无效的优化上。和 3C 模型类似，只要模型进行了抽象，就会存在一些理想的假设。例如，屋顶线模型是假定程序在所有处理器间负载均衡的。

01 精解 一种替换 Stream 基准测试程序的方法是使用原始 DRAM 带宽作为屋顶线。尽管原始带宽构成了硬件上界，但是存储器的实际性能往往与此相差甚远，因此可用性不高。也就是说，没有程序能够接近该上界。使用 Stream 的不利因素是非常仔细的编程有可能获得高于 Stream 的结果，因此存储器屋顶线不像计算屋顶线那样坚实。我们坚持使用 Stream 是因为很少有程序员能够做到这一点。

01 精解 尽管屋顶线模型是针对多核处理器的，但是它也可以用于单处理器。

01 小测验

是非判断：评测并行计算的常规方法的主要缺陷是确保公平性的同时压制了创新。

6.11 实例：评测 Intel Core i7 960 和 NVIDIA Tesla GPU 的 Roofline 模型

一组 Intel 的研究人员发表了一篇论文 [Lee 等, 2010]，对带有多媒体 SIMD 扩展的四核 Intel Core i7 960 与前一代的 GPU (NVIDIA Tesla GTX 280) 进行了对比。图 6-22 列出了两个系统的特点。这两个系统都是在 2009 年秋天购买的。Core i7 使用的是 Intel 的 45 纳米半导体工艺，而 GPU 使用的是 TSMC 的 65 纳米工艺。虽然让一个中立机构或一个对两种产品都感兴趣的机构进行评估可能更公平一些，但是本章的重点不是为了说明哪个产品运行地更快，快的比慢的快多少等问题，而是尝试着理解这两种截然不同的结构的特性。

	Core i7-960	GTX 280	GTX 480	280/i7 的比例	480/i7 的比例
处理单元的数量 (核或线程数量)	4	30	15	7.5	3.8
时钟频率 (GHz)	3.2	1.3	1.4	0.41	0.44
芯核尺寸	263	576	520	2.2	2.0
工艺	Intel 45nm	TSMC 65nm	TSMC 40nm	1.6	1.0
功率 (芯片, 非模块)	130	130	167	1.0	1.3
晶体管数量	700M	1 400M	3 030M	2.0	4.4
存储带宽 (G字节/秒)	32	141	177	4.4	5.5
单精度SIMD宽度	4	8	32	2.0	8.0
双精度SIMD宽度	2	1	16	0.5	8.0
峰值单精度向量FLOPS (GFLOP/秒)	26	117	63	4.6	2.5
峰值单精度SIMD FLOPS (GFLOP/秒)	102	311到933	515或1344	3.0–9.1	6.6–13.1
(SP 1加或乘)	不支持	(311)	(515)	(3.0)	(6.6)
(SP 1融合乘加的指令)	不支持	(622)	(1 344)	(6.1)	(13.1)
(Rare SP双通道融合乘加和乘)	不支持	(933)	不支持	(9.1)	–
Peak双精度SIMD FLOPS (GFLOP/秒)	51	78	515	1.5	10.1

图 6-22 Intel Core i7-960、NVIDIA GTX 280 以及 GTX 480 的指标。最右面的两列展示了 Tesla GTX 280 和 Fermi GTX 480 与 Core i7 的对比。尽管例子研究的是 Tesla 280 和 i7，但是我们也给出了 Fermi 480 与 Tesla 280 的对比，因为本章中介绍到了这点。注意这里的存储带宽比图 6-23 中的要高，因为这里的是 DRAM 引脚的带宽，而图 6-23 中的是通过测试程序得到的处理器中的带宽（本表来自于 Lee 等. [2010] 中的表 2）

图 6-23 中给出的 Core i7 960 和 GTX 280 的 Roofline 模型展示了这两个计算机的区别。GTX 280 不仅有更高的存储带宽和双精度浮点性能，而且它的双精度拐点更靠左。GTX 280 的双精度拐点在 0.6，而 Core i7 的在 3.1。上面曾提到，若 Roofline 模型的拐点更靠左，则更容易达到它的性能峰值。对于单精度性能，这两个计算的观点都更靠右一些，所以要想达到最佳的单精度计算性能会更难。注意，算术运算强度是基于访问主存的字节数的，而不是基于访问 cache 的字节数。因此，就像之前提到的，若大部分访存都访问 cache，则缓存可以改变一个核心程序在计算机上的运算强度。再次注意，对于这两种结构，带宽是单位步长的访问。我们即将看到，真实的聚集 - 分散地址在 GTX 280 和 Core i7 上会更慢一些。

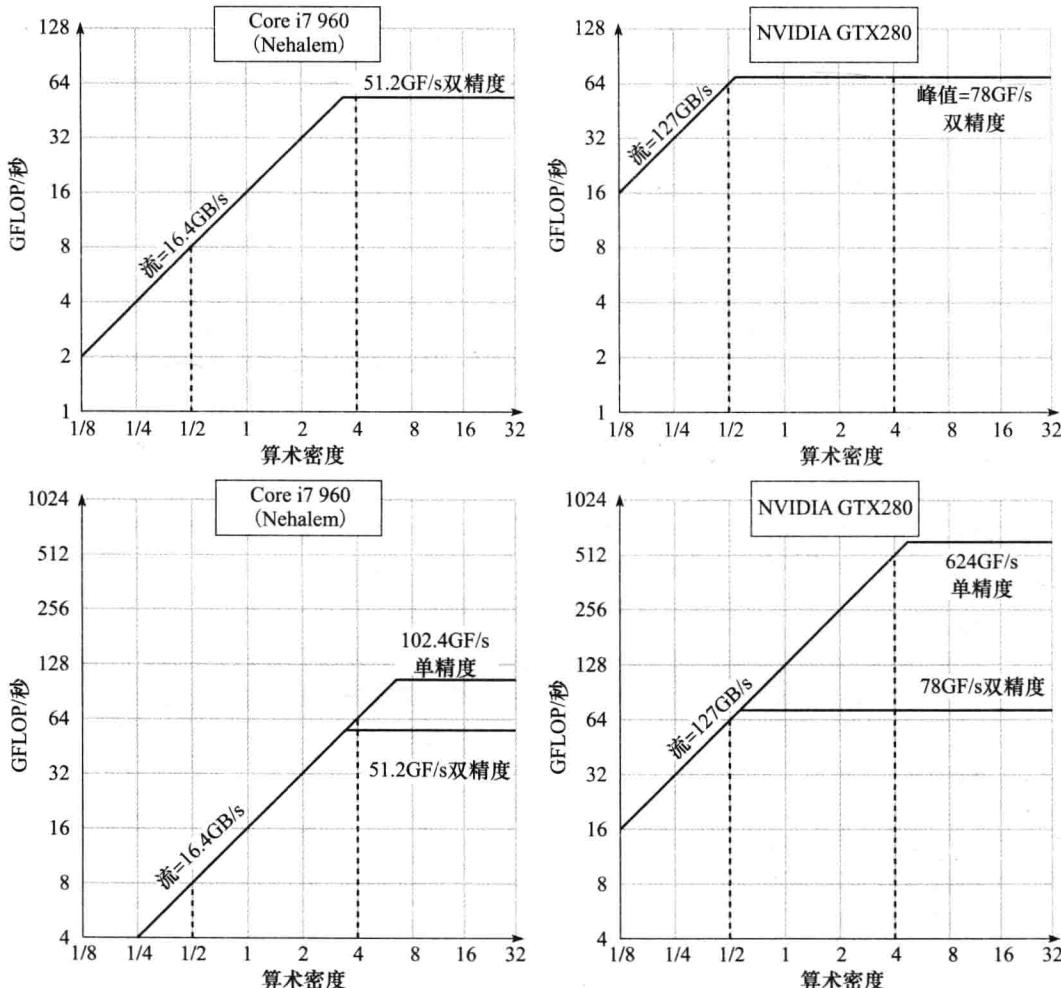


图 6-23 Roofline 模型 [Williams, Waterman, Patterson, 2009]。上面一行给出了双精度浮点性能，下面一行给出了单精度浮点性能。（双精度浮点性能的屋顶线同时也在下面一行的图中画出了，已给出对比。）左边的 Core i7 960 的最高双精度浮点性能为 51.2GFLOP/s，最高单精度浮点性能为 102.4GFLOP/s，最高存储带宽为 16.4GB/s。NVIDIA GTX 280 的最高双精度浮点性能为 78GFLOP/s，最高单精度浮点性能为 624GFLOP/s，最高存储带宽为 127GB/s。图中左侧的垂直虚线表示 0.5FLOP/byte 的算术运算强度。Core i7 的算术运算强度受存储带宽的限制不能超过 8GFLOP/s（对于双精度浮点和单精度浮点都如此）。图中右侧的垂直虚线表示 4FLOP/byte 的运算强度。在 Core i7 上限制在 51.2GFLOP/s（双精度浮点性能）到 102.4GFLOP/s（单精度浮点性能）之间，在 GTX 280 上限制在 78GFLOP/s（双精度浮点性能）到 512GFLOP/s（双精度浮点性能）之间。为了在 Core i7 上达到最高的运算率，需要使用全部 4 个核和 SSE 指令，并且乘法和加法的数量相等。对于 GTX 280，你需要在所有多线程的 SIMD 处理器上使用混合的乘加指令。

通过分析，研究者们最近提出了 4 个测试集程序的计算和访存特性，然后对可以捕获这些特性的吞吐计算核心程序集建模来选择测试用的程序。图 6-24 给出了性能结果，数字越大表明速度越快。Roofline 模型帮助解释了本例子中的相对性能。

内核	单位	Core i7-960	GTX 280	GTX 280 i7-960
SGEMM	GFLOP/秒	94	364	3.9
MC	十亿条路径/秒	0.8	1.4	1.8
Conv	百万像素/秒	1 250	3 500	2.8
FFT	GFLOP/秒	71.4	21.3	3.0
SAXPY	G字节/秒	16.8	88.8	5.3
LBM	百万次查找/秒	85	426	5.0
Solv	帧/秒	103	52	0.5
SpMV	GFLOP/秒	4.9	9.1	1.9
GJK	帧/秒	67	1 020	15.2
Sort	百万个元素/秒	250	198	0.8
RC	帧/秒	5	8.1	1.6
Search	百万次查询/秒	50	90	1.8
Hist	百万像素/秒	1 517	2 583	1.7
Bilat	百万像素/秒	83	475	5.7

图 6-24 对两个平台测量得到的原始和相对性能数据。在这项研究中，SAXPY 只被用来对存储带宽进行测量，所以右边的单位为 GBytes/s 而不是 GFLOP/s（基于 [Lee 等. 2010] 中的表 3。）

GTX 280 的标准性能在慢 2.5 倍（时钟频率）到快 7.5 倍（每个芯片上的核数）之间，而它的实际性能为慢 2.0 倍到快 15.2 倍之间，Intel 的研究者们决定找到其中的原因：

- **存储带宽：** GPU 有 4.4 倍的存储带宽，这解释了为什么 LBM 和 SAXPY 运行地快 5.0 倍和 5.3 倍；它们的工作集为几百兆字节，因此数据不能在 Core i7 的 cache 中全放下。（所以由于集中地访存，它们刻意不使用第 5 章中的缓存阻塞技术）。因此，Roofline 模型的斜率解释了它们的性能。SpMV 同样也有一个大的工作集，但是它的运行只快了 1.9 倍，这是因为 GTX 280 的双精度浮点运算只比 Core i7 的快 1.5 倍。
- **计算带宽：** 剩下的核心程序中有 5 个是受计算制约的：SGEMM、Conv、FFT、MC 和 Bilat。GTX 运行这 5 个程序时分别快 3.9、2.8、3.0、1.8 和 5.7 倍。这其中的前三个使用单精度浮点运算，而且 GTX 280 的单精度运算要快 3~6 倍。MC 使用双精度浮点运算，这解释了为什么它只快 1.8 倍，因为双精度浮点运算性能只比 Core i7 快 1.5 倍。Bilat 使用 GTX 280 直接支持的超越函数。Core i7 执行 Bilat 时，三分之二的时间用来计算超越函数，所以 GTX 280 要快 5.7 倍。这些观察帮助指明了使用硬件支持负载的特定操作（双精度浮点运算、超越函数）有何意义。
- **从 cache 获得的好处：** 在 GTX 上 RC 运行地只快 1.6 倍，只是因为在 GPU 上 Core i7 的缓存阻塞技术可以防止程序变得受存储带宽的影响（参见 5.4 节和 5.14 节），而在 GTX 上会出现这种情况。缓存阻塞技术还可以帮助 Search 程序。如果索引树小到可以在 cache 中装下，则 Core i7 的速度会快 2 倍。大的索引树会使程序受存储带宽的限制。总体来说，GTX 在运行 Search 程序时快 1.8 倍。缓存阻塞技术也对 Sort 程序有利。尽管大部分程序员不会在一个 SIMD 处理器上运行 Sort，但是它可以用称作 split 的 1 位 Sort 原语编写。但是，split 算法比标量 Sort 程序多执行更多的指令。结果是，Core i7 比 GTX 280 快 1.25 倍。注意，cache 对于运行在 Core i7 上的其他核心程序也有帮助，因为缓存阻塞技术允许 SGEMM、FFT 和 SpMV 变为受计算制约的程序。这项观察再次强调了第 5 章介绍的缓存阻塞技术的重要性。
- **聚集 - 分散：** 如果数据被分散到主存的各个地方，多媒体 SIMD 扩展就不会有太大的

用途；只有访问的数据是 16 字节对齐时才会获得最佳的性能。因此 GJK 程序在 Core i7 上从 SIMD 中获得的好处很少。就像前面提到过的，GPU 提供向量结构支持而大部分 SIMD 扩展不支持的聚集 - 分散技术。内存控制器甚至会将访问同一个 DRAM 页的请求打包（见 5.2 节）。这两点使得 GTX 280 在运行 GJK 时比 Core i7 快 15.2 倍，这比图 6-22 中的任何物理参数都要大。这项观察再一次证实了聚集 - 分散技术对于缺乏 SIMD 扩展的向量和 GPU 结构的重要性。

- 同步：同步的性能受原子更新的制约，这占据了 Core i7 28% 的总执行时间，尽管 Core i7 有硬件读取并自增（fetch-and-increment）指令。因此，Hist 程序在 GTX 280 上只比 Core i7 快 1.7 倍。Solv 程序使用少量指令和一个同步操作来解决一批独立的限制条件。Core i7 可以从原子指令以及一致性存储模型之中获益，一致性存储模型可以保证结果是正确的，尽管不是所有之前的访存指令都已经执行完毕。由于没有一致性存储模型，当 GTX 280 从系统处理器那里得到一批操作时，它的性能只是 Core i7 的 0.5 倍。这项观察指出了同步的性能对于一些数据并行问题是多么重要。

由 Intel 研究者选择的这些核心程序揭示出的 Tesla GTX 280 的不足，在 Tesla 的后续版本中都得到了解决：Fermi 拥有更好的双精度浮点性能、更快的原子操作以及 cache。另一个有趣的是，支持向量结构的聚集 - 分散机制早于 SIMD 指令几十年就出现了，并且对 SIMD 扩展的有效执行非常重要，这在这次比较之前就已经预言到了。Intel 的研究者们注意到这 14 个核心程序中的 6 个可以在对 Core i7 提供足够的聚集 - 分散支持的前提下更有效地发掘 SIMD。这项研究当然也证实了缓存阻塞技术的重要性。

554

现在我们看到了评价不同多处理器得出的很多结果，接下来让我们看看我们需要对 DGEMM 程序的 C 代码进行多大的修改才可以发挥多处理器的性能优势。

6.12 运行更快：多处理器和矩阵乘法

这一节是我们根据 Intel Core i7（Sandy Bridge）已经改变结构的 DGEMM 程序以获得性能提升的最后一步，也是最大的一步。每个 Core i7 有 8 个核，我们用的计算机有 2 个 Core i7。所以我们有 16 个核来运行 DGEMM 程序。

图 6-25 给出了使用这些核的 OpenMP 版本的 DGEMM 程序。注意，第 30 行是相对于图 5-48 唯一增加的一行代码，以使程序可以运行在多处理器上：使用了一个 OpenMP 的 pragma 语句告诉编译器对最外层 for 循环使用多线程。它告诉计算机将最外层 for 循环的任务分配给所有线程去执行。

图 6-26 画出了一个经典的多处理器加速比图，它展示了当线程数量增加时，相对于单线程的性能提升。这个图让人们很容易地看到强比例缩放相对于弱比例缩放的挑战。当所有数据都可以放入一级数据 cache 中时，例如 32×32 矩阵，增加线程的数量实际上会损坏性能。在这种情况下，16 个线程的 DGEMM 程序的性能只是单线程的一半。相反，最大的那两个矩阵在使用 16 个线程时，性能提升了 14 倍，所以得到了图 6-26 最上面的两条线。

图 6-27 给出了当我们增加线程的数量时的绝对性能增长。对于 960×960 的矩阵，DGEMM 程序以 174GLOPS 的速率执行。图 3-21 给出的未经任何优化的 C 版本的 DGEMM 程序是以 0.8GFOPS 的速率执行，所以通过第 3 ~ 6 章根据硬件对代码进行的优化，性能提升了 200 倍！

接下来我们给出了多进程的谬误与陷阱。计算机系统结构的墓地充满了忽略这些谬误与陷阱的并行项目。

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6 {
7     for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8         for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9             __m256d c[4];
10            for ( int x = 0; x < UNROLL; x++ )
11                c[x] = _mm256_load_pd(C+i+x*4+j*n);
12 /* c[x] = C[i][j] */
13            for( int k = sk; k < sk+BLOCKSIZE; k++ )
14            {
15                __m256d b = _mm256_broadcast_sd(B+k+j*n);
16 /* b = B[k][j] */
17                for (int x = 0; x < UNROLL; x++)
18                    c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                                           _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20            }
21
22            for ( int x = 0; x < UNROLL; x++ )
23                _mm256_store_pd(C+i+x*4+j*n, c[x]);
24 /* C[i][j] = c[x] */
25        }
26    }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32         for ( int si = 0; si < n; si += BLOCKSIZE )
33             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34                 do_block(n, si, sj, sk, A, B, C);
35 }

```

图 6-25 图 5-48 中的 DGEMM 程序的 OpenMP 版。第 30 行是唯一一条 OpenMP 语句，它使最外层的 for 循环并行执行。这一行代码是与图 5-48 中的唯一区别

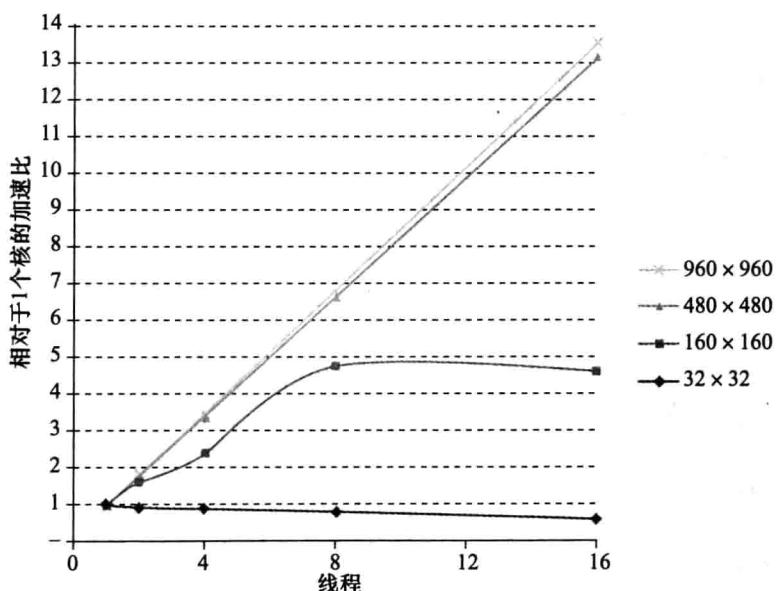


图 6-26 与单线程相比，当线程数增多时的性能提升。最客观的方法是拿多线程的性能与最优的单线程的性能相比，这也是我们的做法。这个图是与图 5-48 中没有使用 OpenMP 的#pragma语句的代码相比较的。

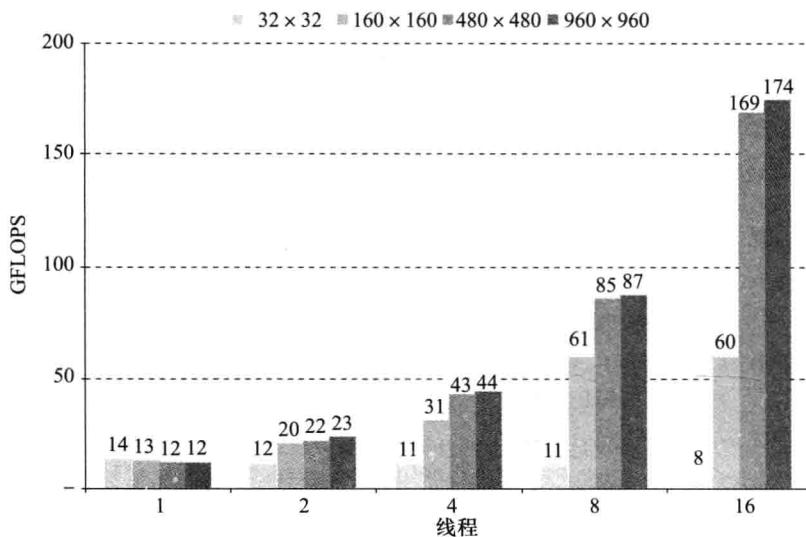


图 6-27 4 个不同大小的矩阵的 DGEMM 程序的性能。对于 960×960 矩阵，在使用 16 个线程时与图 3-21 中的未经任何优化的代码相比，性能提升了 212 倍

01 精解 这些结论是在 Turbo 模式关闭的情况下得到的。在这个系统中，我们使用的是一个双芯片系统，所以意料之中，我们无论使用 1 个线程（只使用一个芯片上的一个核）还是 2 个线程（每个芯片上使用一个核）都能得到完整的 Turbo 加速 ($3.3/2.6 = 1.27$)。当我们增加线程数时（使用的核数也增多），从 Turbo 模式的获益将减少，因为会有更少的功耗来使用这些核。对于 4 个线程，平均 Turbo 加速比对于 4 个线程、8 个线程和 16 个线程分别为 1.23、1.13 和 1.11。

01 精解 虽然 Sandy Bridge 支持每个核两个硬件线程，但当使用 32 个线程时我们无法得到更多的性能提升。这是因为一个 AVX 硬件被同一个核上的两个线程所共享，所以当为一个核分配两个线程时实际上会对性能有所损害。

555
557

6.13 谬误与陷阱

十多年来，一直有人在争论单处理器的组织形式已经到达了性能极限，并且性能的真正改进只能通过将多台计算机互连从而以这种方式支持协同计算……事实证明单处理器的性能一直在不断增长……

—— Gene Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, Spring Joint Computer Conference, 1967

对并行处理的大量研究揭示了诸多谬误和陷阱。我们在这里讨论其中 4 个。

谬误：Amdahl 定律不适用于并行计算机。

1987 年，一个研究组织的负责人宣称 Amdahl 定律已经被多处理器所打破。为了试图理解这些媒体报道的依据，我们首先看一下对 Amdahl 定律的相关引用 [1967, p. 483]：

此时可以得出的一个相当直观的结论是：花费在获得高并行处理速度上的努力都是无用的，除非顺序处理速度提高的数量级也与其十分接近。

这句话依然是正确的；程序中被忽视的部分必然限制性能。该定律的一种解释可得到下面一条引理：每个程序中都有一部分是顺序的，因此必然有一个经济的处理器数量上界——比如说是 100。通过给出使用 1 000 个处理器也可以达到线性增长，证明该引理是错误的；因而得出了 Amdahl 定律被打破的结论。

这些研究人员的方法是使用弱比例缩放：他们不是在相同的数据集上将速度提高 1 000 倍，而是在可比较时间内将计算量提高 1 000 倍。对于他们的算法，程序中顺序执行的比例是常数，与问题的输入规模无关，而其余部分则是完全并行的——因此，使用 1 000 个处理器时依然为线性增长。

Amdahl 定律显然也适用于并行处理器。这项研究确实指出了更快的计算机主要用途之一是完成更大规模的问题。只要确保用户真的关心这些问题并把这些作为购买更贵的计算机的理由，而不是关心为找到可以使很多处理器保持忙碌的问题。

谬误：峰值性能可代表实际性能。

超级计算机业界在市场中曾经使用该度量方法，并且该谬误在并行机中更加严重。市场营销人员不仅在单处理器节点使用这种几乎不可能达到的峰值性能指标，而且还将其乘以处理器的总个数，从而假定并行机可以达到完美的加速度！Amdahl 定律指出达到两种峰值是多么困难；将两者相乘就错上加错了。屋顶线模型有助于达到合乎比例的峰值性能。

陷阱：在利用和优化多处理器体系结构时不开发软件。

在很长的时间里并行软件一直落后于并行硬件，可能是因为软件问题困难得多。我们给出一个例子说明这一问题，但是可供选择的例子还有很多！

558

在将为单处理器设计的软件移植到多处理器环境时经常会遇到这样一个问题。例如，SGI 操作系统最初假定页分配不频繁，从而通过一个锁来保护页表。在单处理器中，这不是一个性能问题。在多处理器中，对某些程序会成为一个主要的性能瓶颈。考虑一个程序在启动时需要初始化大量页的情况，正如 UNIX 为静态分配页所做的操作那样。假设该程序被并行化以便多核进程分配页。由于页的分配需要使用页表，而页表在每次使用时必须锁定，即使操作系统内核支持多线程，如果这些进程试图同时请求分配页（这恰好就是我们在初始化时所预期的情况）也会因此串行执行。

页表的串行操作影响了初始化时的并行，并对整个并行性能有着很大的影响。该性能瓶颈甚至在任务级并行中也存在。例如，假设我们将并行处理程序分为若干独立的作业并分别在一个处理器上运行一个作业，这样在不同作业之间就没有任何共享。（这恰好是一个用户的做法，因为他合乎情理地相信性能问题是由于应用程序中非预期的共享或冲突所造成的。）不幸的是，锁机制依然将所有工作串行化——因此说明即使互相独立的工作性能也会很低。

该陷阱说明当软件在多处理器上运行时，这种微妙但对性能有极大影响的错误会显现出来。和其他许多主要软件一样，操作系统的算法和数据结构在多处理器上需要重新考虑。在页表的更小区域加锁可以有效地避免这个问题。

谬误：可以在不提升存储器带宽的前提下得到好的向量计算性能。

从 Roofline 模型中可以看到，存储带宽对各种体系结构都很重要。DAXPY 每个浮点操作需要 1.5 个存储访问，对于很多科学计算代码这是一个很标准的比例。即使浮点操作不需要花费时间，一个 Cray-1 计算机也不会增加 DAXPY 向量序列的性能，因为它的存储受到限制。当编译器使用阻塞机制改变计算，以使数据可以保存在向量寄存器中时，Cray-1 运行 Linpack 的性能有了跳跃式提升。这个方法降低了每个浮点运算的访存次数并使性能提升了将近两倍。因此，Cray-1 的存储带宽对于之前有更多带宽需求的循环来说足够了，这正是 Roofline 模型所预言的。

559

6.14 本章小结

我们正在将未来产品的开发专注于多核设计。我们相信这对工业界是一个重要转折点。……这不是一场竞争。这是计算的翻天覆地的变化……

——Paul Otellini, Intel 总裁, Intel 开发者论坛, 2004

自从计算开始之日起，人们就梦想着通过简单的集成若干处理器构建计算机。然而，构建并充分有效利用并行处理器的进程是缓慢的。其原因之一是受软件难点的限制，另一方面是为

了提高可用性和效率，多处理器的体系结构在不断改进。本章中我们讨论了许多软件方面的挑战，包括编写由于 Amdahl 定律可获得高加速比程序的难点。不同并行体系结构之间往往存在巨大差异，所取得的性能提升也非常有限，而且过去许多并行体系结构的生命周期非常短暂，这些因素使得软件更加困难。网站上的 6.15 节讨论了这些多处理器的历史。要对本章所讲述的主题有更深入的理解，请参阅《计算机体系结构：量化研究方法》第 5 版第 4 章中的更多关于 GPU 以及 CPU 与 GPU 之间进行对比的内容，还有第 6 章中关于 WSC 的内容。

正如第 1 章所述，信息技术业的未来与并行计算是紧密联系在一起的。像过去一样，尽管有很多努力会失败，但是依然有很多理由让我们充满希望：

- 显然，软件即服务（SaaS）的重要性正在增长中，并且集群已经被证实为提供此类服务的一种非常成功的方法。通过提供高层次的冗余，包括地理分布的数据中心，此类服务可以为全世界的客户提供 $24 \times 7 \times 365$ 的可用性。
- 我们相信仓储级计算机正在改变服务器设计的目标和原则，就像移动客户端的需求正在改变微处理器设计的目标和原则一样。这两者同样也都在革新软件行业。移动客户端硬件和 WSC 硬件都受一美元所能带来多少性能和每焦耳所能带来多少性能的驱动，并行是提供这些目标的关键。
- SIMD 和向量操作很适合在后 PC 时代占据重要地位的多媒体应用。它们比经典的并行 MIMD 编程更简单并且在功耗方面效率更高。为了认识到 SIMD 与 MIMD 的重要性，图 6-28 给出了随着时间的增长 MIMD 中的核数，以及 x86 计算机中 SIMD 模式下每时钟周期的 32 位和 64 位操作个数。对于 x86 计算机，我们期待每两年在芯片上增加两个核并且每四年 SIMD 宽度增加一倍。在这些假设的前提下，在下一个十年后，SIMD 的并行加速会是 MIMD 并行的两倍。给出了 SIMD 对于多媒体的效率和它在后 PC 时代越来越大的重要性，这个推论可能是正确的。因此，至少应理解 SIMD 并行和 MIMD 并行同等重要，尽管后者已经得到了更多的关注。

560

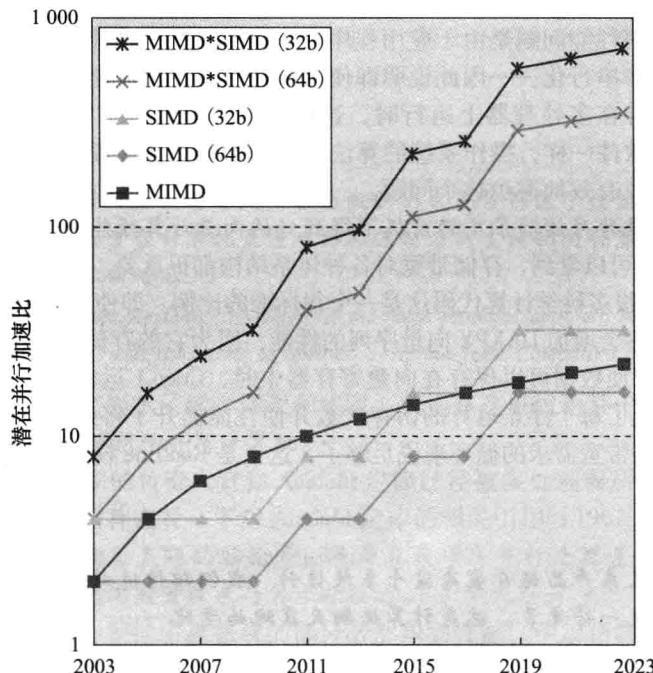


图 6-28 MIMD 和 SIMD 的潜在并行加速比，以及 x86 计算机的 MIMD 和 SIMD 随着时间的变化。该图假设每两年单芯片上 MIMD 的核数增加两个，并且每四年 SIMD 操作的数目增加一倍

- 并行处理在科学计算和工程计算等领域中非常普遍。此类应用领域对计算能力几乎充满无限的渴望。而且有很多应用具有天然的并行性。集群再一次地占据了此类应用领域。例如，根据 2012 Top 500 报告，集群占据了 500 个最快的 Linpack 报告的 80%。
- 为了获得更高性能，所有的桌面和服务器微处理器制造商正在构建多处理器，顺序应用程序不会像过去一样再有获取更高性能的捷径。正如我们之前所说的，串行程序现在就是慢的程序。因此，需要更高性能的程序员必须将自己的代码并行化，或者编写全新的并行处理程序。
- 在过去，微处理器和多处理器在成功上的定义是不同的。当缩放单处理器性能时，如果单线程性能随硅面积的开方增长，微处理器设计者会感觉很满意。也就是说，他们满足于性能随资源数量的亚线性增长。多处理器的成功在过去通常定义为与处理器数量相关的线性加速比函数，并假定 n 个处理器的购买成本或管理成本是单一处理器的 n 倍。目前并行正在以片上多核的形式实现，我们可以使用已经获得成功的传统微处理器标准来获得亚线性的性能提升。
- 运行时编译技术和自动运行的成功使得软件更容易适应每芯片上核数量的增长，提供受静态编译器限制所不能提供的灵活性。
- 与过去不同的是，开放源代码运动已经成为软件业的一个关键部分。这项运动可以改善工程解决方案，促进开发者之间的知识共享。它也鼓励创新，在改变旧软件时欢迎新的语言和软件产品。这种开放式的文化必将有益于目前日新月异的时期。

为使读者接受这个改变，我们通过快速浏览第 3 ~ 6 章的章节来展示如何通过 Intel Core i7 (Sandy Bridge) 处理器发掘矩阵乘法的潜在并行：

- 第 3 章中的数据级并行通过使用 256 位的 AVX 指令并行执行 4 个 64 位浮点运算使性能提升了 3.85 倍，这展示了 SIMD 的价值。
- 第 4 章中的指令级并行 4 次展开循环给乱序执行的硬件提供了更多的指令去调度，这又使性能提升了 2.3 倍。
- 第 5 章中的 cache 优化使用 cache 阻塞来减少 cache 缺失，这对不能放进 L1 cache 的矩阵性能提升了 2.0 ~ 2.5 倍。
- 本章中的线程级并行通过使用多核芯片上的所有 16 个核使无法放入单一 L1 cache 的矩阵的性能提升了 4 ~ 14 倍，这展示了 MIMD 的价值。我们是通过加入了一行 OpenMP pragma 语句实现的。

使用本书中的方法并且根据该计算机对软件进行改变，在 DGEMM 程序上加了 24 行代码。对于 32×32 、 160×160 、 480×480 和 960×960 的矩阵，通过这几行代码和本书的方法得到的总性能加速比为 8、39、129 和 212！

软硬件接口上的并行变革也许是过去 60 年来所面临的最大挑战。你也可以把它当做最大的机遇，就像我们快速浏览各章章节时所展示的。这个变革在 IT 界内外提供了大量研究和商业前景，并且主导多核的公司并不一定与主导单处理器的公司相同。在理解了硬件发展的潮流以及学会了如何根据硬件来改变软件之后，也许你就会抓住其中的机会，成为创新者中的一员。我们期待从你的发明创造中获益！

6.15 历史观点和拓展阅读

本节在网上主要给出了近 50 年来多处理器的发展历史。

561

562

参考文献

- G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP offloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.
- B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB, In: Proceedings of the 1st ACM Symposium on Cloud computing, June 10–11, 2010, Indianapolis, Indiana, USA, doi:10.1145/1807128.1807152.

6.16 练习题

6.1 首先写一个每周你通常需要完成的日常活动的列表。例如，你可能会起床、淋浴、穿衣服、吃早饭、弄干头发、刷牙。确保列表中至少包含 10 项活动。

563 6.1.1 [5] <6.2> 考虑哪些活动已经利用了某种形式的并行性（例如，是同时刷多颗牙还是一次只刷一颗牙，是一次只带一本书到学校，还是将所有书装到背包里一次“并行”携带）。对每个活动都分析是否已经并行工作，如果没有，分析其原因。

6.1.2 [5] <6.2> 接下来考虑哪些活动可以并发执行（例如，吃早餐和听新闻）。对每个活动都分析哪些活动可以与其配对并发执行。

6.1.3 [5] <6.2> 对练习题 6.1.2，可以通过改变现有系统（例如，淋浴设备、衣服、电视机、汽车等）中的什么来让我们并行执行更多的任务？

6.1.4 [5] <6.2> 如果你能尽可能多地并行执行任务，估计完成这些任务可以缩短的时间是多少？

6.2 假设需要你制作 3 块蓝莓蛋糕。蛋糕的配料如下：

1 杯黄油，软化后再用

1 杯糖

4 个大鸡蛋

1 茶匙香草精

0.5 茶匙盐

0.25 茶匙肉豆蔻

1.5 杯面粉

1 杯蓝莓

蛋糕的制作流程如下：

第 1 步：烤箱预热至 160°C (325°F)。在烤盘上抹黄油和一层薄薄的面粉。

第 2 步：在一只大碗中使用搅拌器以中速将奶油和糖混合在一起，直到松发。再加鸡蛋、香草精、盐和肉豆蔻，搅拌到完全混合。将搅拌器降到低速，一次加入 0.5 杯面粉，搅拌到完全混合。

第 3 步：最后慢慢加入蓝莓，将蛋糕均匀地放在烤盘中，烘烤约 60 分钟。

6.2.1 [5] <6.2> 你的任务是尽可能高效率地完成 3 块蛋糕。假定只有一个能容纳一块蛋糕的烤箱、一个大碗、一个烤盘、一个搅拌器，请做出合理的调度以尽可能快地完成任务，并分析瓶颈所在。

6.2.2 [5] <6.2> 假设你现在有 3 个碗、3 个蛋糕盘子和 3 个搅拌器。你拥有这些增加的资源后，现在的工序加快了多少？

6.2.3 [5] <6.2> 假设你现在有两个朋友，可帮你烹饪，并且你有一个可容纳 3 个蛋糕的大烤箱。这些将对练习题 6.2.1 中的计划有何改变？

6.2.4 [5] <6.2> 将制作蛋糕与并行计算机中的循环迭代进行类比。分析制作蛋糕的循环中存在的数据级并行和任务级并行。

6.3 许多计算机应用程序需要在一组数据中进行搜索和对数据进行排序。为了减少这些任务的执行时间，已经实现了几种高效的搜索和排序算法。在本练习中，我们将考虑如何将这些任务的并行最大化。

- 6.3.1** [10] <6.2> 请看下面的二进制搜索算法（一种经典的分而治之算法），该算法可以在已经排序的 N 元素数组 A 中搜索值 X ，并返回匹配项的索引号：

```
BinarySearch(A[0..N-1], X) {
    low = 0
    high = N -1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] > X)
            high = mid -1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}
```

假设 `BinarySearch` 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 N 。请问预期的加速比是多少？请画图表示。

- 6.3.2** [5] <6.2> 接下来，假设 Y 与 N 相同，这会对你前面的结论有何影响？如果要求你获得尽可能高的加速比（强比例缩放），请问该如何修改代码？

- 6.4** 请看下面的 C 代码片段：

```
for (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

565

与之对应的 MIPS 代码如下所示：

```
addiu    $s2,$zero,7992
addiu    $s1,$zero,16
loop:   l.d      $f0, -16($s1)
        l.d      $f2, -8($s1)
        add.d    $f4, $f0, $f2
        s.d      $f4, 0($s1)
        addiu   $s1, $s1, 8
        bne     $s1, $s2, loop
```

每种指令的延迟如下（以周期为单位）：

add.d	l.d	s.d	addiu
4	6	1	2

- 6.4.1** [10] <6.2> 执行一次循环所有的指令需要多少周期？
- 6.4.2** [10] <6.2> 在循环中，如果后面重复执行的指令会依赖于前面指令产生的结果，我们说循环内重复存在循环进位相关性（loop-carried dependence）。请分析上面代码中的循环进位相关性，识别其中相关的程序变量和汇编级寄存器。可忽略循环变量 j 。
- 6.4.3** [10] <6.2> 第 4 章中描述了循环展开。对此循环进行展开，并考虑将此代码运行在一个 2 节点的基于消息传递的分布式存储器系统中。假定我们采用 6.7 节描述的消息传递机制，操作 `send(x, y)` 可向节点 x 发送值 y ，操作 `receive()` 可等待正在发送的数。再假定 `send` 操作的发射需要 1 个周期（也就是说，同一节点的后续指令可在下个周期执行），而接收节点需要 10 个周期接收。接收指令会阻塞接收节点上指令的执行，一直等到接收节点完成消息接收为止。假设循环会执行 4 次，请计算在该基于消息传递的系统中完成循环所需的周期数。
- 6.4.4** [10] <6.2> 互联网络的延迟是决定消息传递系统效率的重要因素之一。请问为了让练习题 6.4.3 中的分布式系统能获得任意加速比，互联网络需要提供多快的速度？
- 6.5** 考虑下面的归并排序算法（另一种经典的分而治之算法）。归并排序由 John Von Neumann 于 1945

566

年首先提出。其基本思想是将含有 m 个元素的未排序序列 x 分为两个子序列，其中每个序列长度都大约是原来的一半。然后对每个子序列重复类似的动作，直到每个子序列的长度均为 1。再从长度为 1 的子序列开始，将两个子序列“归并”为一个排序的序列。

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
    return result
```

下面的代码实现归并步骤：

```
Merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
        if length(left) > 0
            append rest(left) to result
        if length(right) > 0
            append rest(right) to result
    return result
```

- 6.5.1** [10] <6.2> 假设 MergeSort 运行在具有 Y 个核的多核处理器上，且 Y 远远小于 m （长度）。请问预期的加速比是多少？请画图表示。
- 6.5.2** [10] <6.2> 接下来，假设 Y 与 m （长度）相同，这会对你前面的结论有何影响？如果要求你获得尽可能高的加速比（例如，强比例缩放），请问该如何修改代码？

567

- 6.6** 矩阵乘在大量应用中都扮演重要角色。两个矩阵可以相乘的条件是第一个矩阵的列数和第二个矩阵的行数相同。
- 假设我们有一个 $m \times n$ 的矩阵 A ，还有一个 $n \times p$ 的矩阵 B 与之相乘。乘法结果为一个 $m \times p$ 的矩阵 AB （或 $A \cdot B$ ）。如果令 $C = AB$ ， $C_{i,j}$ 代表在矩阵 (i, j) 位置处的值，则 $1 \leq i \leq m$ 且 $1 \leq j \leq p$ 。现在我们考虑是否可以将 C 的计算并行化。假设矩阵在存储器中的存放顺序为： $a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}, \dots$
- 6.6.1** [10] <6.5> 假设我们分别在单核/四核共享存储器的系统计算 C ，请问四核相对于单核的预期加速比是多少？可忽略存储器相关的问题。
- 6.6.2** [10] <6.5> 如果对 C 的更新会导致 cache 缺失（例如更新一行中连续的元素时可能引起伪共享），重新计算练习题 6.6.1 中的问题。
- 6.6.3** [10] <6.5> 有什么办法消除可能出现的伪共享问题？
- 6.7** 下面的两个程序同时运行在一个包含 4 个处理器的 SMP（对称多核处理器）中。假设在开始运行之前， x 和 y 的初值均为 0。
- 核 1： $x = 2;$
 核 2： $y = 2;$

核 3: $w = x + y + 1;$

核 4: $z = x + y;$

6.7.1 [10] <6.5> w 、 x 、 y 、 z 所有可能的结果分别是什么？对每种可能的情况，通过分析指令的交错情况，解释其产生的原因。

6.7.2 [5] <6.5> 采用什么措施能让执行变成更有确定性，以便只产生一种结果。

6.8 哲学家就餐问题是一个经典的同步和并发问题。该问题假设就座于一个圆桌周围的哲学家们可以做两件事之一：吃饭或思考。当他们吃饭时，他们不能思考，反之亦然。在圆桌中心有一碗通心粉。每两个哲学家之间有一只叉子，这样每个哲学家左面有一把叉子，右面也有一把叉子。按照吃通心粉的方式，哲学家需要两把叉子才能吃通心粉，而且只能使用紧挨着他左右的两把叉子。哲学家不能和其他人说话。

6.8.1 [10] <6.7> 请描述没有任何哲学家可以吃通心粉的情景。什么样的事件序列会导致该问题发生？

6.8.2 [10] <6.7> 如何通过引入优先级的概念来解决这一问题？这样可以对所有哲学家公平对待吗？请解释原因。

现在假定我们增加一个服务员负责为哲学家们分配叉子。只有在服务员允许之下他们才可以拿起叉子。服务员也知道所有叉子的状态。而且我们要求所有哲学家总是先请求拿起左边的叉子再请求拿起右边的叉子，这样可以避免死锁。

6.8.3 [10] <6.7> 对服务员请求的实现，可以将请求放入一个队列，也可以让请求周期性地重试。采用队列方式，请求可以按收到的顺序依次处理。使用队列的问题是即使请求排在队列的最前面，我们也不能保证总是为其提供服务，因为可能缺乏所需的资源。试想使用 1 个队列为 5 个哲学家服务的情景，即使有的哲学家有两把叉子都可用但仍然不能为其服务（因为他的请求排在队列的后部）。

6.8.4 [10] <6.7> 如果我们让请求周期性地重试直到资源变为可用，这样是否就解决了练习题 6.8.3 中的问题？请给出原因。

6.9 请看下面的 3 种 CPU 结构：

CPU SS: 一个双核超标量微处理器，支持在两个功能单元 (FU) 上的乱序发射。每个核只能运行单一线程。

CPU MT: 一个细粒度的多线程处理器，支持来自两个线程中指令的并发执行（也就是说，有两个功能单元），尽管每个周期只能从一个线程发射一条指令。

CPU SMT: SMT 处理器支持来自两个线程的指令并发执行（也就是说，有两个功能单元），并且发射的指令可来自任一线程或者两个线程。

假定我们在这些 CPU 上运行线程 X 和线程 Y，具体操作如下：

线程 X	线程 Y
A1: 需三个周期执行 A2: 无相关性 A3: 与 A1 使用的一个功能单元冲突 A4: 需要 A3 的结果	B1: 需两个周期执行 B2: 与 B1 使用的一个功能单元冲突 B3: 需要 B2 的结果 B4: 无相关性并且需要两个周期执行

568

除非特别标记或者遇到相关阻塞，假定所有的指令都是单周期执行。

6.9.1 [10] <6.4> 如果使用一个 SS CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

6.9.2 [10] <6.4> 如果使用两个 SS CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

6.9.3 [10] <6.4> 如果使用一个 MT CPU，执行这两个线程需要多少个周期？相关阻塞浪费了多少发射槽？

569

6.10 虚拟化软件正在用于降低管理高性能服务器的成本。包括 VMWare、Microsoft 和 IBM 公司在内的很多公司正在开发一系列的虚拟化产品。第 5 章中介绍的管理程序层 (hypervisor layer) 位于硬件和操作系统之间，使得多个操作系统可以共享同一物理硬件。管理程序层负责分配 CPU 和存储器资源，同时处理原本由操作系统完成的服务（如 I/O）。

虚拟化为宿主操作系统和应用软件提供了底层硬件的一个抽象层，使得若干操作系统可并行运行在共享的 CPU 和存储器上。我们需要重新考虑未来如何设计多核和多处理器系统来对此进行支持。

6.10.1 [30] <6.4> 选择市场上的两种管理程序，比较它们虚拟化和管理底层硬件（CPU 和存储器）的方式。

6.10.2 [15] <6.4> 为了更好地满足未来多核 CPU 平台的资源需求，可采取哪些措施？例如，多线程技术是否可以减轻计算资源间的竞争？

6.11 我们将讨论如何高效地执行下面的代码。假设我们有两种不同的机器，一种是 MIMD，另一种是 SIMD。

```
for (i=0; i < 2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

6.11.1 [10] <6.3> 对一个包含 4 个 CPU 的 MIMD 机器，请给出每个 CPU 上执行的 MIPS 指令序列。此 MIMD 机器的加速比是多少？

6.11.2 [20] <6.3> 对一个宽度为 8 的 SIMD 机器（也就是说，包含 8 个并行的 SIMD 功能单元），使用你自己的对 MIPS 的 SIMD 扩展编写一个执行该循环的汇编程序，并比较 SIMD 和 MIMD 上执行指令的数量。

6.12 MISD 机器的一个例子是脉动阵列 (systolic array)。它是一个由数据处理单元构成的流水线网络或波阵面。这些单元都不需要程序计数器，因为执行是通过数据到达触发的。时钟脉动阵列以与每个处理器相“锁步”的方式进行计算，而这些处理器承担了交替的计算和通信。

6.12.1 [10] <6.3> 分析脉动阵列的各种实现机制（可以在互联网或出版物中查找相关资料），然后使用 MISD 模型对练习题 6.11 中的循环进行编程，并对遇到的问题进行讨论。

6.12.2 [10] <6.3> 应用数据级并行中的各种术语，分析 MISD 和 SIMD 之间的相似点和不同点。

6.13 假定我们要执行本章讲述 NVIDIA 8800 GTX GPU 时提到的 DAXPY 循环。在这一问题中，我们假定所有算术操作是单精度浮点数运算（因此我们将其重新命名为 SAXPY）。假定指令的执行周期数如下所示。

Loads	Stores	Add. s	Mult. s
5	2	3	4

6.13.1 [20] <6.6> 请描述一下在一个 8 核处理器中如何构建 warp 来完成 SAXPY 循环？

6.14 从 http://www.nvidia.com/object/cuda_get.html 下载 CUDA Toolkit 和 SDK。注意使用代码的 emurelease (Emulation Mode) 版本（此版本可在没有 NVIDIA 硬件的情况下运行）。编译 SDK 中提供的示例程序，并确认它们运行在仿真器上。

6.14.1 [90] <6.6> 以 SDK 的示例程序为起点，编写一个完成如下向量操作的 CUDA 程序：

- 1) $a - b$ (向量减法)
- 2) $a \cdot b$ (向量点积)

向量 $a = [a_1, a_2, \dots, a_n]$ 和 $b = [b_1, b_2, \dots, b_n]$ 的点积定义如下：

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

运行编写的程序并验证结果是否正确。

6.14.2 [90] <6.6> 如果你有可用的 GPU 硬件，请完成对程序的性能分析，并查看在向量大小不同的情况下 GPU 和一个 CPU 版本的计算时间，并解释其中的原因。

6.15 AMD 最近宣布将把 GPU 与 x86 核集成到一个封装中，尽管两者的时钟不同。这是我们能在不远的将来看到的一种异构多处理器系统商业化产品。设计的关键之一是如何支持 CPU 和 GPU 之间的高速数据通信。当前，必须在 CPU 和 GPU 之间进行通信，这在 AMD 的 Fusion 体系结构中发生了变化。目前的计划是采用多个（至少 16 个）PCI express 通道来实现高速通信。Intel 也使用 Larrabee 芯片进行了类似的研究，通信计划采用 QuickPath 互联技术。

6.15.1 [25] <6.6> 比较这两种互联技术的带宽和延迟。

6.16 参照图 6-14b 中给出的 3 阶 n 维立方体互连拓扑结构，其将 8 个节点进行了互连。 n 维立方体互连拓扑的一个优势是在部分互连损坏的情况下依然可以保持连接性。

6.16.1 [10] <6.8> n 维立方体中最多有多少互连损坏时还能保证任何节点依然能够连接？请写出计算公式。

6.16.2 [10] <6.8> 比较 n 维立方体和全互联网络的可靠性。画图比较两种拓扑分别在多少连接损坏增加时会导致连接失效。

6.17 基准测试程序用于在指定的计算平台上运行有代表性的工作负载，从而比较不同系统之间的性能。在本练习题中，我们将比较两种 benchmark：Whetstone CPU benchmark 和 PARSEC Benchmark suite。从 PARSEC 中选择一个程序。所有程序都可从网上免费下载。考虑将 Whetstone 的多份备份或 PARSEC Benchmark 运行在 6.11 节中描述的各个系统上。

6.17.1 [60] <6.10> 两种工作负载运行在这些多核系统上的本质区别是什么？

6.17.2 [60] <6.10> 使用 Roofline 模型的相关术语，分析在运行了这些 benchmark 时，运行情况与工作负载中共享和同步的数量相关性有多大？

6.18 在计算稀疏矩阵时，存储器的延迟至关重要。由于稀疏矩阵缺乏矩阵操作中常见的空间局部性，所以需要研究新的矩阵表示方法。

572

最早的稀疏矩阵表示方法之一是 Yale 稀疏矩阵格式。它使用 3 个一维数组存储维数 $m \times n$ 的矩阵 M 。令 R 代表 M 中的非零项数目。我们构造一个长度为 R 的数组 A 存储 M 中的所有非零项（按照从左到右、从上到下的顺序）。我们再构造一个长度为 $m + 1$ 的数组 IA 。 $IA(i)$ 包含第 i 行中第一个非零项在 A 中的索引号。原矩阵中的第 i 行的元素可从 $A(IA(i))$ 到 $A(IA(i+1)-1)$ 中得到。第三个数组 JA 包含 A 中每个元素的列号，因此它的长度也为 R 。

6.18.1 [15] <6.10> 分析下面的稀疏矩阵 X ，并编写 C 程序将其存储为 Yale 稀疏矩阵格式。

```
Row 1 [1, 2, 0, 0, 0, 0]
Row 2 [0, 0, 1, 1, 0, 0]
Row 3 [0, 0, 0, 0, 9, 0]
Row 4 [2, 0, 0, 0, 0, 2]
Row 5 [0, 0, 3, 3, 0, 7]
Row 6 [1, 3, 0, 0, 0, 1]
```

6.18.2 [10] <6.10> 在存储空间方面，假定矩阵 X 中的每个元素都是单精度浮点格式，如果用 Yale 稀疏矩阵格式存储上面的矩阵，请计算共需多少存储空间。

6.18.3 [15] <6.10> 执行下面给出的矩阵 X 和矩阵 Y 的矩阵乘。

```
[2, 4, 1, 99, 7, 2]
```

将该计算放入循环中，并对执行过程进行计时。确保增加循环执行的次数，以在你的时间测量中获得较好的分辨率。比较矩阵的原始表示的运行时间和 Yale 稀疏矩阵格式的运行时间。

6.18.4 [15] <6.10> 你是否能够找到更加有效的稀疏矩阵表示方法（考虑空间和计算开销）？

6.19 在未来的系统中，我们期待能够看到由异构 CPU 构成的异构计算平台。在嵌入式处理相关市场，一些同时包含浮点 DSP 和微控制 CPU 的多芯片模块包的系统已经开始呈现。

假定你有三类 CPU：

CPU A——每周期可执行多条指令的中速多核 CPU（具有浮点单元）。

CPU B——每周期可执行单条指令的快速单核整型 CPU（例如，无浮点单元）。

CPU C——每周期可执行同样指令的多个副本的慢速向量 CPU（具备浮点能力）。

573

假定我们的处理器在下面的频率运行：

CPU A	CPU B	CPU C
1GHz	3GHz	250MHz

在每个时钟周期，CPU A 可以执行 2 条指令，CPU B 可以执行 1 条指令，CPU C 可以执行 8 条指令（尽管是相同指令）。假定所有的操作在单周期延迟中完成执行，且没有任何冒险。

三个 CPU 均可执行整型算术，尽管 CPU B 不能直接执行浮点算术。CPU A 和 B 具有与 MIPS 处理器相似的指令集。CPU C 仅能执行浮点加、减和存储器存、取操作。假定所有 CPU 均可访问共享存储器，并且同步的开销为零。

我们的任务是比较两个矩阵， X 和 Y ，它们每个都包含 1024×1024 个浮点元素。输出结果应是指示矩阵 X 中何处的值比矩阵 Y 中的值大或相等的一系列数。

- 6.19.1** [10] <6.11> 请描述如何划分该问题到 3 个不同的 CPU 上，以获得最佳性能。
6.19.2 [10] <6.11> 你会向向量 CPU C 中增加哪类指令，以获得更好的性能？
6.20 假定一个四核计算机系统可以处理每秒钟具有稳定状态率的数据库事务。同时假定，每个事务平均花费固定的时间来处理。下表给出了几对事务延迟和处理速率。

平均事务延迟	最大事务处理速率
1ms	5 000/s
2ms	5 000/s
1ms	10 000/s
2ms	10 000/s

对于表中的每一对数据，回答如下问题：

- 6.20.1** [10] <6.11> 在任意给定的瞬间，平均有多少请求被处理？
6.20.2 [10] <6.11> 如果移到 8 核的系统中，理想情况下，系统的吞吐量将发生什么变化（例如，计算机每秒处理多少请求）？
6.20.3 [10] <6.11> 讨论为什么通过简单地增加核的数量，我们很少获得这种加速？

01 小测验答案

- 6.1** 错误。任务级并行可以帮助串行应用，可以使串行应用在并行硬件上运行，尽管会有很多挑战。
6.2 错误。弱缩放可以补偿程序的串行部分，强缩放的缩放性会被串行部分所限制。
6.3 正确。但是它们缺少可以提升向量体系结构性能的特性，如聚集 - 分散和向量长度寄存器。（就像这节中的精解中提到的，AVX2 SIMD 扩展通过聚集操作提供了变址加载但不通过分散操作提供变址存储。Haswell x86 微处理器是第一个支持 AVX2 的处理器。）
6.4 1. 正确。2. 正确。
6.5 错误。由于共享地址是物理地址，且多任务中的每个任务都在它们自己的虚拟地址空间中，因而可在共享存储器多处理器上良好地运行。
6.6 错误。图形 DRAM 因其更高的带宽而被赞扬。
6.7 1. 错误。发送和接收消息是一个隐式的同步，同样也是一种共享数据的方式。2. 正确。
6.8 正确。
6.10 正确。我们或许需要在硬件的所有层次和软件栈上进行革新，以使并行计算成功。