并行编程平台

从传统的逻辑角度来看，顺序计算机包括一个通过数据路径连接到处理器的存储器。处理器、内存和数据路径这三个组成部分都是计算机系统整体处理速度的瓶颈。多年来，一些架构创新解决了这些瓶颈问题。其中最重要的创新之一就是处理单元、数据路径和内存单元的多重性。这种多重性要么对程序员完全隐藏（如隐式并行），要么以不同形式暴露给程序员。在本章中，我们将概述与并行处理相关的重要架构概念。目的是为程序员提供足够详细的信息，使他们能够在各种平台上编写高效的代码。我们开发了成本模型和抽象概念，用于量化各种并行算法的性能，并找出各种编程结构导致的瓶颈。

在讨论并行平台时，我们首先要概述串行和隐式并行架构。这是因为通常可以通过简单的程序转换来重新设计代码，从而实现显著的速度提升（2 到 5 倍的未优化速度）。对次优串行代码进行并行化往往会带来不可靠的速度提升和误导运行时间的不良后果。因此，我们主张在尝试并行化之前先优化代码的串行性能。正如我们将在本章中展示的，串行优化和并行优化的任务往往具有非常相似的特点。在讨论了串行和隐式并行架构之后，我们将在本章的其余部分讨论并行平台的组织、算法的底层成本模型以及可移植算法设计的平台抽象。希望直接了解并行架构的读者可以跳过第 2.1 和 2.2 节。

2.1 隐式并行 - 微处理器架构的趋势

在过去的十年中，微处理器技术在时钟速度方面取得了显著的进步，但同时也暴露出其他各种性能瓶颈。为了缓解这些瓶颈，微处理器设计人员探索了其他途径，以实现具有成本效益的性能提升。在本节中，我们将概述其中的一些趋势，以便了解它们的局限性以及对算法和代码开发的影响。本节的目的不是全面介绍处理器架构。参考书目中提到的几篇优秀文章都涉及这一主题。

- **2.1.1 流水线和超标量**
- **2.1.2 超长指令字处理器**

2.2 内存系统性能的限制

计算机程序的有效运行不仅取决于处理器的速度，还取决于内存系统向处理器提供数据的能力。在逻辑层面上，内存系统（可能由多级缓存组成）接收一个内存字的请求，并在 1 纳秒后返回一个大小为 b 的数据块，其中包含所请求的字。这里的 l 指的是内存的**延迟**时间。数据从内存传送到处理器的速度决定了内存系统的**带宽**。

- **2.2.1 利用缓存改善有效内存延迟**
- **2.2.2 内存带宽的影响**
- **2.2.3 隐藏内存延迟的其他方法**
- **2.2.4 多线程与预取的权衡**

2.3 并行计算平台的两部分

在前面的章节中，我们指出了影响串行或隐式并行程序性能的各种因素。当前微处理器的峰值性能和可持续性能之间的差距越来越大，内存系统性能的影响，以及许多问题的分布式性质，都是并行化的主要动机。现在，我们将从高层次介绍并行计算平台的要素，这些要素对于面向性能和可移植的并行编程至关重要。为了便于我们讨论并行平台，我们首先探讨了基于并行平台的逻辑组织和物理组织的两部分。逻辑组织指的是程序员对平台的看法，而物理组织指的是平台的实际硬件组织。从程序员的角度来看，并行计算的两个关键组成部分是表达并行任务的方法和指定这些任务之间交互的机制。前者有时也被称为**控制结构**，后者则被称为**通信模型**。

- **2.3.1 并行平台的控制结构**
- **2.3.2 并行平台的通信模式**

2.4 并行平台的物理结构

在本节中，我们将讨论并行机的物理架构。我们将从理想架构开始，概述与实现这一模型相关的实际困难，并讨论一些传统架构。

- **2.4.1 理想并行计算机的结构**
- **2.4.2 并行计算机互连网络**
- **2.4.3 网络拓扑**
- **2.4.4 评估静态互连网络**
- **2.4.5 评估动态互连网络**
- **2.4.6 多处理器系统的缓存一致性**

2.5 并行计算中的通信开销

并行程序执行过程中的主要开销之一来自处理元件之间的信息通信。通信成本取决于多种特性，包括编程模型语义、网络拓扑结构、数据处理和路由选择，以及相关的软件协议。这些问题是我们在此讨论的重点。

- **2.5.1 并行计算中的消息传递开销**
- **2.5.2 并行计算中的内存共享开销**

2.6 互联网络中的路由机制

将信息路由到目的地的高效算法对并行计算机的性能至关重要。路由机制决定了信息通过网络从源节点到达目的节点的路径。它将信息的源节点和目的节点作为输入。它还可以使用网络状态信息。它返回一条或多条从源节点到目的节点的网络路径。

2.7 进程到处理器的映射技术及其影响

正如我们在第 2.5.1 节中所讨论的，程序员通常无法控制逻辑进程如何映射到网络中的物理节点。因此，即使通信模式本身并不拥塞，也可能造成网络拥塞。下面我们举例说明：

- **2.7.1 Mapping Techniques for Graphs**
- **2.7.2 Cost-Performance Tradeoffs**

2.1 隐式并行 - 微处理器架构的趋势

在过去的十年中，微处理器技术在时钟速度方面取得了显著的进步，但同时也暴露出其他各种性能瓶颈。为了缓解这些瓶颈，微处理器设计人员探索了其他途径，以实现具有成本效益的性能提升。在本节中，我们将概述其中的一些趋势，以便了解它们的局限性以及对算法和代码开发的影响。本节的目的不是全面介绍处理器架构。参考书目中提到的几篇优秀文章都涉及这一主题。

在过去的 20 年中，微处理器的时钟速度令人印象深刻地提高了两到三个数量级。然而，内存技术的限制严重削弱了这些时钟速度的提升。与此同时，更高的设备集成度也导致晶体管数量庞大，这就提出了一个显而易见的问题，即如何更好地利用晶体管。因此，能够在一个时钟周期内执行多条指令的技术开始流行起来。事实上，这种趋势在当前新一代的微处理器（如 Itanium、Sparc Ultra、MIPS 和 Power4）中都很明显。在本节中，我们将简要探讨各种处理器支持多指令执行的机制。

2.1.1 流水线和超标量执行

长期以来，处理器一直依赖流水线来提高执行速度。通过重叠指令执行中的各个阶段（获取、调度、解码、操作数获取、执行、存储等），流水线实现了更快的执行速度。装配线的比喻很适合用来理解流水线。如果一辆汽车的组装需要 100 个时间单位，可以分成 10 个流水线阶段，每个阶段 10 个单位，那么一条流水线每 10 个时间单位就能生产一辆汽车！这比完全串行地一辆接一辆生产汽车的速度提高了 10 倍。从这个例子中还可以看出，要想提高单条流水线的速度，就必须将任务分解成越来越小的单元，从而延长流水线的长度，增加执行中的重叠。就处理器而言，由于任务变小了，时钟频率也随之加快。例如，工作频率为 2.0 GHz 的奔腾 4 处理器就有 20 级流水线。请注意，单个流水线的速度最终受限于流水线中最大的原子任务。此外，在典型的指令跟踪中，每隔五到六条指令就是一条分支指令。因此，长指令流水线需要有效的技术来预测分支目的地，以便流水线能被推测性地填满。由于需要刷新的指令数量较多，因此流水线越深，预测错误的惩罚就越大。这些因素限制了处理器流水线的深度和由此带来的性能提升。

要提高指令执行率，一个显而易见的方法就是使用多条流水线。在每个时钟周期内，多条指令并行地流水线进入处理器。这些指令在多个功能单元上执行。我们通过一个例子来说明这一过程。

- **例2.1 超标量执行**

  考虑一种具有两条流水线并能同时发出两条指令的处理器。这些处理器有时也被称为超流水线处理器。处理器在同一周期内发出多条指令的能力被称为超标量执行。由于图2.1所示的架构允许在每个时钟周期内发出两条指令，因此也被称为双向超标量或双指令执行。

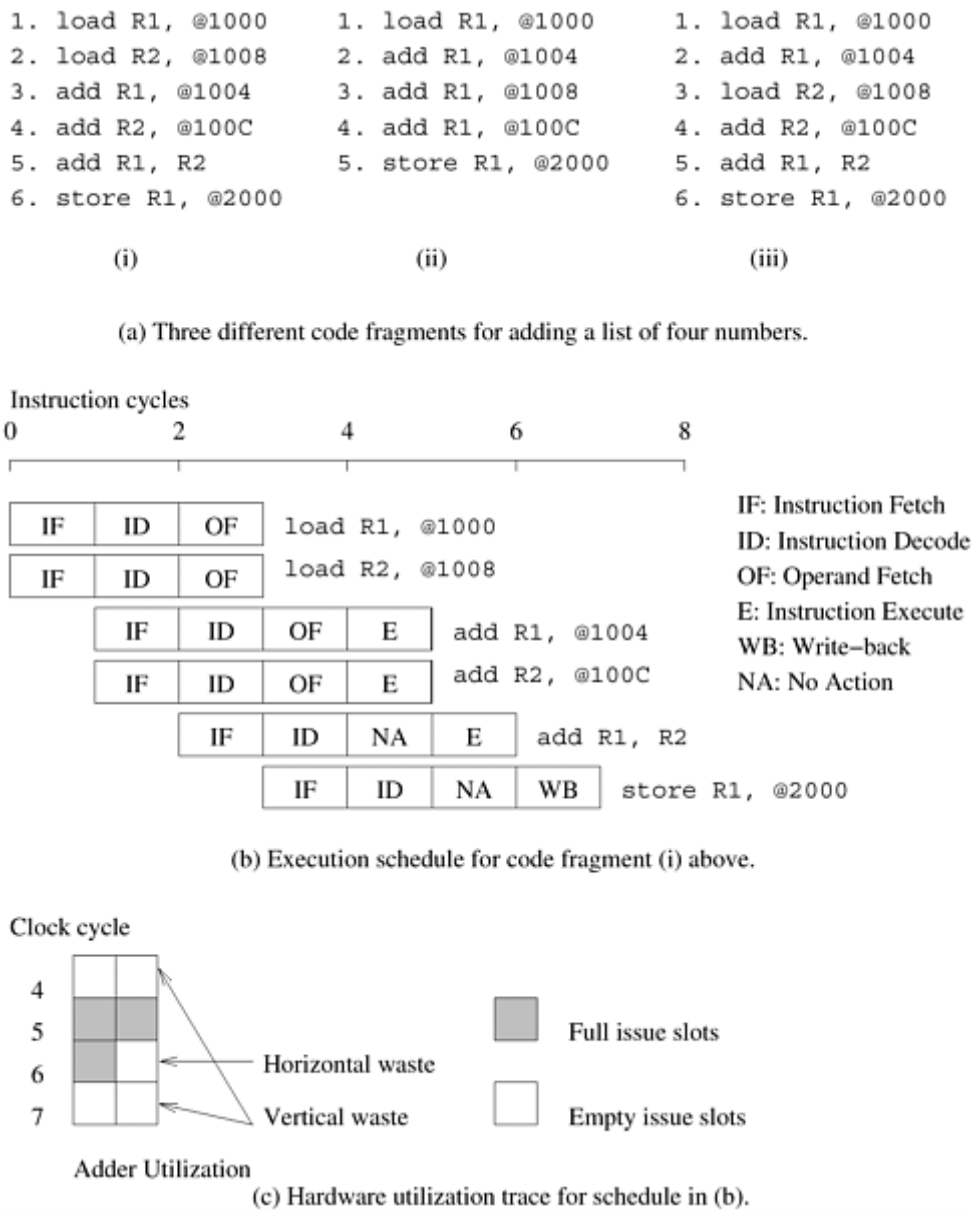


图2.1 双向超标量执行指令示例

图2.1 中第一个代码片段的执行情况，该代码用于四个数字的加法运算。第一条指令和第二条指令是独立的，因此可以同时执行。在 $t = 0$ 时同时发出的指令 `load R1, @1000` 和 `load R2, @1008` 就说明了这一点。接下来的两条指令， `add R1, @1004` 和 `add R2, @100C` 也是相互独立的，尽管它们必须在前两条指令之后执行。因此，由于处理器采用流水线设计，它们可以在 $t = 1$ 时同时发出。这些指令在 $t = 5$ 时终止。接下来的两条指令，即 `add R1, R2` 和 `store R1, @2000` 不能同时执行，因为前一条指令的结果（寄存器 `R1` 的内容）被后一条指令使用。因此，只能在 $t = 2$ 时发出 `add` 指令，在 $t = 3$ 时发出 `store` 指令。需要注意的是，只有在前两条指令执行完毕后，才能执行 `add R1, R2` 指令。指令时间表如图2.1(b) 所示。该时间表假定每次内存访问只需一个周期。实际情况可能并非如此。第 2.2 节将讨论这一假设对内存系统性能的影响。

原则上，超标量执行似乎很自然，甚至很简单。然而，有一些问题需要解决。首先，如例2.1所示，程序中的指令可能相互关联。一条指令的结果可能是后续指令所需要的。这被称为真正的数据依赖性。例如，图 2.1中的第二个代码片段，该代码用于四个数字的相加。 `load R1, @1000` 和 `add R1, @1004` 之间存在真正的数据依赖关系，后续指令之间也存在类似的数据依赖关系。这种依赖关系必须在同时发出指令之前解决，这有两层含义：首先，由于解析是在运行时完成的，因此必须在硬件中受支持，这种硬件的复杂性可能很高。其次，程序中指令级并行性的数量通常是有限的，这与编码技术有关。在第二个代码片段中，指令不能同时运行，从而导致资源利用率低下。图 2.1(a)中的三个代码片段也说明，在大多数情况下，通过

对指令重新排序和修改代码，可以更适合并行。需要注意的是在本例中，代码重组相当于以一种可被指令发布机制使用的形式暴露出并行性。

指令之间的另一个依赖性来源于不同流水线共享的有限资源。举例来说，考虑在具有单浮点运算单元的双发机器（Dual Issue Machine）上共同调度两个浮点运算。虽然指令之间可能不存在数据依赖关系，但由于这两条指令都需要浮点运算单元，因此不能同时进行调度。这种两条指令竞争单个处理器资源的依赖形式被称为**资源依赖（Resource Dependency）**。

程序中的控制流会在指令之间产生第三种形式的依赖关系。考虑一下有条件分支指令的执行。由于只有在执行时才知道分支的目的地，因此事先跨分支安排指令可能会导致错误。这些依赖关系被称为 **分支依赖（Branch Dependencies）** 关系或 **程序依赖（Procedural Dependencies）** 关系，通常通过在分支间进行推测性调度并在出现错误时进行回滚来处理。对典型跟踪的研究表明，平均每五到六条指令就会遇到一条分支指令。因此，与填充指令流水线一样，准确的分支预测对于高效超标量执行至关重要。

处理器检测和调度并发指令的能力对于超标量性能至关重要。例如，图 2.1 中的第三个代码片段也是计算四个数的和。读者会发现，这只是对第一个代码片段进行了语义等价的重新排序。然而，在这种情况下，前两条指令 - `load R1, @1000` 和 `add R1, @1004` 之间存在数据依赖关系。因此，这些指令不能同时发出。但是，如果处理器具有前瞻能力，它就会意识到可以将第三条指令（`load R2, @1008`）与第一条指令安排在一起。在下一个执行周期中，指令二和指令四可以同时执行，以此类推。这样，第一个和第三个代码片段就可以得到相同的执行时间表。不过，处理器需要具备 **不按顺序（Out-of-Order）** 发布指令的能力，以完成所需的重新排序。如本例所示，**按顺序（In-Order）** 发出指令的并行性可能会受到很大限制。目前大多数微处理器都能实现无序指令的发布和完成。这种模式也称为 **动态指令发布（Dynamic Instruction Issue）**，可最大限度地利用指令级并行性。处理器使用一个指令窗口，从中选择指令同时执行。该窗口与调度程序的前瞻性相对应。

超标量架构的性能受到可用指令级并行性的限制。请看图 2.1 中的示例。为便于讨论，我们忽略该示例中的流水线问题，重点关注程序的执行问题。假设有两个执行单元（乘加单元，Multiply-Add Units），图中显示有几个零Issue周期（浮点单元空闲的周期）。从执行单元的角度来看，这些周期基本上是浪费周期。如果在某个周期内，执行单元没有发出任何指令，则称为 **纵向浪费（Vertical Waste）**；如果在一个周期内只使用了部分执行单元，则称为 **横向浪费（Horizontal Waste）**。在本例中，有两个周期存在垂直浪费，一个周期存在水平浪费。总之，八个可用周期中只有三个用于计算。这意味着代码片段的计算量不会超过处理器额定 FLOP 数峰值的八分之三。通常，由于并行性有限、资源依赖或处理器无法提取并行性，超标量处理器的资源利用率严重不足。目前的微处理器通常最多支持四组超标量执行。

2.1.2 超长指令字处理器

超标量处理器的并行性往往受到指令前瞻性的限制。在传统微处理器上，用于动态依赖性分析的硬件逻辑通常占总逻辑的 5-10%（在四路超标量 Sun UltraSPARC 上约占 5%）。这种复杂性随着问题数量的增加而呈四倍增长，并可能成为瓶颈。超长指令字（Very Long Instruction Word，VLIW）处理器中使用的另一种利用指令级并行性的概念是依靠编译器在编译时解决依赖性和资源可用性问题。可同时执行的指令被打包成组，并作为单个长指令字分发给处理器，以便在多个功能单元上同时执行。

与超标量处理器相比，VLIW 概念既有优势也有劣势。VLIW 概念最早用于多流跟踪（Multiflow Trace，约 1984 年），后来作为英特尔 IA64 架构的一个变体。由于调度是在软件中完成的，因此 VLIW 处理器的解码和指令发布机制更为简单。与硬件发行单元相比，编译器有更大的范围来选择指令，并能使用各种转换来优化并行性。编译器通常可以使用额外的并行指令来控制并行执行。然而，编译器没有动态程序状态（如分支历史缓冲区）可用来做出调度决策。这就降低了分支和内存预测的准确性，但允许使用更复杂的静态预测方案。其他运行时情况，如高速缓存未命中导致的数据获取停滞，则极难准确预测。这就限制了基于编译器的静态调度的范围和性能。

最后，VLIW 处理器的性能对编译器检测数据和资源依赖性以及读写危险的能力，以及为最大并行性安排指令的能力非常敏感。循环展开、分支预测和推测执行都对 VLIW 处理器的性能起着重要作用。虽然超标量和 VLIW 处理器在利用隐式并行性方面取得了成功，但它们通常仅限于四路至八路并行性范围内的较小并发规模。

> **Note**
>
> **流水线**：CPU支持按照流水线模式同时执行多条命令，以提高计算资源的利用率。
>
> **超标量执行**：通过动态指令发布，乱序执行命令，在解决资源依赖的同时提高计算资源的利用率。
>
> **超长指令字处理器**：在程序编译的时候就将可并行的指令做打包，但是不支持动态调度。

2.2 内存系统性能限制

计算机程序的有效运行不仅取决于处理器的速度，还取决于内存系统向处理器提供数据的能力。在逻辑层面上，内存系统（可能由多级缓存组成）接收一个内存字的请求，并在 1 纳秒后返回一个大小为 b 的数据块，其中包含所请求的字。这里的 l 指的是内存的 **延迟** 时间。数据从内存传送到处理器的速度决定了内存系统的**带宽**。

了解延迟和带宽之间的区别非常重要，因为解决这两个问题需要不同的技术，而且往往是相互竞争的技术。打个比方，如果消防栓打开 2 秒钟后，消防水带的末端就有水流出，那么系统的延迟时间就是 2 秒钟。一旦水流开始，如果水管以每秒 1 加仑的速度抽水，那么水管的 "带宽 "就是每秒 1 加仑。如果我们需要立即灭火，我们可能希望延迟时间更短。这通常需要消防栓提供更高的水压。另一方面，如果我们希望扑灭更大的火灾，我们可能需要更高的流量，这就需要更宽的水管和消防栓。我们将在这里看到，这种类比对内存系统也很有效。延迟和带宽在决定内存系统性能方面都起着至关重要的作用。我们将通过几个例子分别对它们进行更详细的研究。

为了研究内存系统延迟的影响，我们在下面的示例中假设一个内存块由一个字组成。稍后，我们将在研究内存带宽的作用时放宽这一假设。由于我们主要关注的是可实现的最高性能，因此我们还假设了最佳情况下的高速缓存替换策略。有关内存系统设计的详细讨论，请参阅参考书目。

- **例2.2 内存延迟对性能的影响**

  考虑一个工作频率为 $1GHz$（$1ns$ 时钟）的处理器，它与一个延迟时间为 $100ns$（无缓存）的 DRAM 相连。假设该处理器有两个乘加单元，能够在每个 $1ns$ 周期内执行四条指令。因此，处理器的峰值为 $4GFLOPS$。由于内存延迟等于 100 个周期，块大小为一个字，因此每次内存请求发出后，处理器必须等待 100 个周期才能处理数据。考虑在这样的平台上计算两个向量的点积问题。点积计算对两个矢量元素进行一次乘加运算，即每次浮点运算需要获取一次数据。不难看出，这种计算的峰值速度仅限于每 $100ns$ 进行一次浮点运算，或 $10MFLOPS$ 的速度，仅占处理器峰值额定值的很小一部分。这个示例突出说明，要实现高计算速度，需要有效的内存系统性能。

2.2.1 利用缓存改善有效内存延迟

为解决处理器和 DRAM 速度不匹配的问题，内存系统设计中出现了许多架构创新。其中一项创新是通过在处理器和 DRAM 之间放置一个更小、更快的存储器来解决速度不匹配问题。这种内存被称为高速缓冲存储器，是一种低延迟、高带宽的存储设备。处理器所需的数据首先被获取到高速缓存中。对缓存中数据项的所有后续访问都由缓存提供服务。因此，原则上说，如果一个数据被重复使用，缓存可以减少该内存系统的有效延迟。高速缓存满足的数据引用比例称为系统计算的高速缓存命中率。许多应用的有效计算速度并不取决于 CPU 的处理速度，而是取决于向 CPU 输入数据的速度。这种计算被称为内存约束计算。内存绑定程序的性能受到高速缓存命中率的严重影响。

- **例2.3 缓存对内存系统性能的影响**

  与上一示例相同，考虑一个 $1GHz$ 处理器和一个 $100ns$ 延迟的 DRAM。在这种情况下，我们引入一个大小为 $32KB$ 的高速缓存，其延迟时间为 $1ns$ 或一个周期（通常在处理器本身）。我们使用这种设置对两个矩阵 A 和 B 进行乘法运算，矩阵尺寸为 32×32。我们精心选择了这些数字，以便高速缓存足够大，可以存储矩阵 A 和 B 以及结果矩阵 C。我们再次假设了一种理想的缓存放置策略，即所有数据项都不会被其他数据项覆盖。将两个矩阵取入高速缓存相当于取 $2K$ 个字，大约需要 $200\mu s$。我们从基本算法中得知，两个 $n \times n$ 矩阵相乘需要 $2n^3$ 次运算。对于我们的问题，这相当于 $64K$ 次运算，以每个周期四条指令计算，可以在 $16K$ 个周期（或 $16\mu s$）内完成。因此，计算的总时间约为加载/存储操作时间与计算本身时间的总和，即 $200 + 16\mu s$。这相当于 $64K/216$ 或 $303MFLOPS$ 的峰值计算速度。请注意，这比上一个示例提高了 30 倍，但仍不到处理器峰值性能的 10%。从这个例子中我们可以看到，通过放置一个小型高速缓冲存储器，我们可以大大提高处理器的利用率。

缓存对性能的提升是基于对同一数据项的重复引用这一假设。这种在较小的时间窗口内重复引用数据项的概念被称为引用的时间局部性。在我们的示例中，数据访问次数为 $O(n^2)$，计算次数为 $O(n^3)$。数据重用对高速缓存性能至关重要，因为如果每个数据项只使用一次，那么每次使用时仍需从 DRAM 中提取一次，因此每次操作都要支付 DRAM 延迟。

2.2.2 内存带宽的影响

内存带宽是指数据在处理器和内存之间移动的速度。它由内存总线和内存单元的带宽决定。提高内存带宽的一个常用技术是增加内存块的大小。举例说明，让我们放宽对内存块大小的简化限制，假设单个内存请求返回一个包含四个字的连续内存块。在这种情况下，由四个字组成的单个单元也被称为高速缓存行。传统计算机通常会将 2 到 8 个字一起取入高速缓存。我们将看到这对数据重用受限的应用性能有何帮助。

- **例2.4 内存块大小的影响：两个向量的点积**

  再考虑一个具有单周期高速缓存和 100 周期延迟 DRAM 的内存系统，处理器工作频率为 $1GHz$。如果块大小为一个字，处理器取每个字需要 100 个周期。对于每一对字，点积执行一次乘加运算，即两个 FLOP。因此，如例 2.2 所示，该算法每 100 个周期执行一次 FLOP，峰值速度为 $10MFLOPS$。

  现在让我们考虑一下，如果块的大小增加到 4 个字，即处理器每 100 个周期可以获取一个 4 个字的高速缓存行，会发生什么情况。假设向量在内存中是线性排列的，那么在 200 个周期内可以执行 8 个 FLOP（4 个乘加）。这是因为单次内存访问可获取矢量中的四个连续字。因此，两次访问可获取每个向量的四个元素。这相当于每 $25ns$ 完成一个 FLOP，峰值速度为 $40MFLOPS$。请注意，将块大小从一个字增加到四个字并没有改变内存系统的延迟。但带宽却增加了四倍。在该算法完全没有数据重用的情况下，内存系统带宽的增加使我们能够加速点积算法。

  另一种快速估算性能边界的方法是估算缓存命中率，用它来计算每个字的平均访问时间，并通过底层算法将其与 FLOP 率联系起来。例如，在本例中，算法每访问八次数据，就会有两次 DRAM 访问（缓存未命中）。这相当于 75% 的高速缓存命中率。假设主要的开销来自高速缓存未命中，则未命中的平均内存访问时间为 25%，即 $100ns$（或 $25\ ns/word$）。由于点积的运算量为 $1\ operation/word$，因此计算速率与之前一样为 $40MFLOPS$。对这一计算率的更精确估计是，平均内存访问时间为 $0.75 \times 1 + 0.25 \times 100$ 即 $25.75\ ns/word$，相应的计算速度为 $38.8MFLOPS$。

从物理角度看，例 2.4 中的情况相当于一条宽数据总线（4 个字或 128 位）连接到多个内存库（Memory Banks）。实际上，这种宽总线的构建成本很高。在更实用的系统中，在取回第一个字后，会在随后的总线周期中向内存总线发送连续的字。例如，对于 32 位数据总线，第一个字在 $100ns$ 后放入总线，随后每个总线周期放入一个字。由于整个高速缓存行在 $100 + 3x\ ns$（内存总线周期）后才可用，因此上述计算结果会稍有变化。假设数据总线的工作频率为 $200MHz$，则高速缓存行的访问时间将增加 $15ns$。这并不会明显改变我们对执行速率的限制。

上述例子清楚地说明了带宽的增加如何带来更高的峰值计算速度。这些示例还做出了某些对程序员具有重要意义的假设。假设数据布局是这样的：内存中连续的数据字被连续的指令使用。换句话说，如果我们以计算为中心，那么内存访问存在空间位置性。如果从以数据布局为中心的角度来看，计算是有序的，因此连续的计算需要连续的数据。如果计算（或访问模式）不具有空间位置性，那么有效带宽可能会比峰值带宽小得多。

这种访问模式的一个例子是，当密集矩阵在内存中以行为主（row-major）的方式存储时，按列读取矩阵。编译器通常可以很好地重组计算，以利用空间局部性。

- **例2.5 跨步访问的影响**

  请看下面的代码片段：

```
for (i = 0; i < 1000; i++) {
    column_sum[i] = 0.0;
    for (j = 0; j < 1000; j++) {
        column_sum[i] += b[j][i];
    }
}
```

  代码片段将矩阵 *b* 的列求和为一个向量 *column_sum*。有两点值得注意：

  1. 向量 *column_sum* 较小，很容易放入高速缓存；
  2. 如图 2.2(a)所示，按列顺序访问矩阵 *b*。对于以行为主的顺序存储的 1000×1000 大小的矩阵，这相当于访问每 1000 个条目。

因此，从内存中获取的每个高速缓存行可能只使用一个字，上述代码片段的性能可能会很差。

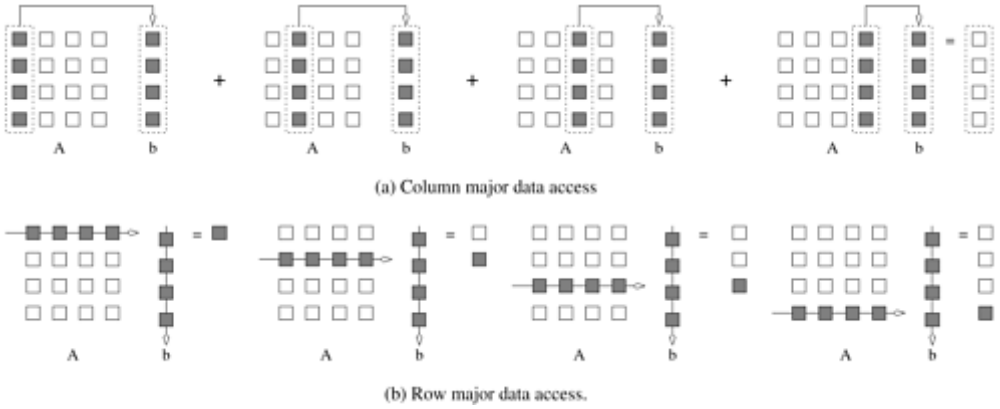


图2.2 矩阵与矢量相乘：(a)逐列相乘；(b)计算矩阵的每一行与矢量的点积结果中的每个元素

上面的例子说明了跨步访问（步长大于 1）的问题。计算中缺乏空间定位会导致内存系统性能低下。通常情况下，可以通过调整计算结构来消除跨步访问。在上述示例中，可以对循环进行如下简单重写：

- **例2.6 消除跨步访问**

  下面是对列和片段的重组：

```
for (i = 0; i < 1000; i++) {
    column_sum[i] = 0.0;
}
for (j = 0; j < 1000; j++) {
    for (i = 0; i < 1000; i++) {
        column_sum[i] += b[j][i];
    }
}
```

  在这种情况下，矩阵是按行顺序遍历的，如图 2.2(b)所示。不过，读者会注意到，这个代码片段依赖于这样一个事实，即向量 *column_sum* 可以通过循环保留在高速缓存中。事实上，在这个特定的例子中，我们的假设是合理的。如果向量更大，我们就必须将迭代空间分割成块，每次计算一个块的乘积。这一概念也被称为迭代空间**平铺**。这个循环的性能改进留给读者去练习。

因此，下一个问题是，我们是否已经有效解决了内存延迟和带宽带来的问题。在过去的几十年里，处理器的峰值运算速率大幅增长，但内存延迟和带宽却没有跟上增长的步伐。因此，对于典型计算机而言，峰值 FLOPS 速率与峰值内存带宽的比率介于 $1MFLOPS/MB$（该比率表示每兆字节/秒带宽的 FLOPS 速率）与 $100MFLOPS/MB$ 之间。较低的数字通常与大型矢量超级计算机相对应，较高的数字则与基于微处理器的快速计算机相对应。这个数字非常有启发性，因为它告诉我们，一个字在被取入全带宽存储（通常是 *L1* 高速缓存）后，平均必须重复使用 100 次才能达到处理器的全部利用率。在此，我们将全带宽定义为计算所需的数据传输速率，以使其与处理器绑定。

本节介绍的一系列示例说明了以下概念：

- 利用应用中的空间和时间位置性，对于摊销内存延迟和增加有效内存带宽至关重要。
- 某些应用本身就比其他应用具有更强的时间局部性，因此对低内存带宽的容忍度更高，操作次数与内存访问次数的比率是衡量内存带宽预期承受能力的一个很好指标。
- 内存布局和适当组织计算可对空间和时间局部性产生重大影响。

2.2.3 隐藏内存延迟的其他方法

想象一下，在网络流量高峰时段坐在电脑前浏览网页的情景。您可以使用以下三种简单的方法之一来缓解浏览器缺乏响应的问题：

1. 我们提前预知要浏览的网页，并提前向它们发出请求；
2. 我们打开多个浏览器，并在每个浏览器中访问不同的网页，因此，当我们等待一个网页加载时，我们可能正在阅读其他网页
3. 我们一次性访问一大堆网页 - 在各种访问中摊销延迟。

第一种方法称为预取，第二种称为多线程，第三种则对应于访问内存字的空间位置性。在这三种方法中，内存访问的空间位置性之前已经讨论过。在本节中，我们将重点讨论作为延迟隐藏技术的预取和多线程技术。

多线程隐藏延迟

线程是程序流程中的单个控制流，我们用一个简单的例子来说明线程：

- **例2.7 矩阵乘法的线程执行**

  请看以下代码段，用于将 $n \times n$ 矩阵 a 与向量 b 相乘，得到向量 c。

```
for(i = 0; i < n; i++) {
    c[i] = dot_product(get_row(a, i), b);
}
```

  请注意，每个点积都相互独立，因此代表了一个并发执行单元。我们可以安全地将上述代码段重写为：

```
for(i = 0; i < n; i++) {
    c[i] = create_thread(dot_product, get_row(a, i), b);
}
```

  这两段代码之间的唯一区别是，我们明确指定了点积计算的每个实例都是一个线程。(我们将在第 7 章中了解到，有许多指定线程的 API。我们只是为创建线程的函数选择了一个直观的名称）。现在，请考虑函数 *dot_product* 的每个实例的执行情况。该函数的第一个实例访问一对向量元素并等待它们。在此期间，该函数的第二个实例可以在下一个循环中访问另外两个向量元素，以此类推。经过 l 个单位的时间（l 为内存系统的延迟时间）后，第一个函数实例从内存中获取了所需数据，并可以执行所需的计算。在下一个周期，下一个函数实例的数据项到达，依此类推。这样，在每个时钟周期内，我们都可以执行一次计算。

例 2.7 中的执行时间表基于两个假设：内存系统能够满足多个未处理请求，处理器能够在每个周期切换线程。此外，它还要求程序以线程的形式明确说明并发性。多线程处理器能够保持多个计算线程的上下文，这些线程具有未处理请求（内存访问、I/O 或通信请求），并在请求得到满足时执行这些请求。HEP 和 Tera 等机器所依赖的多线程处理器可以在每个周期内切换执行上下文。因此，只要有足够的并发性（线程）来防止处理器空转，它们就能有效地隐藏延迟。并发性和延迟之间的权衡将是本文许多章节中反复出现的主题。

预取隐藏延迟

在一个典型的程序中，处理器会在一个很小的时间窗口内加载并使用一个数据项。如果加载导致缓存丢失，那么使用就会停滞。解决这个问题的一个简单办法是提前加载操作，这样即使出现缓存缺失，数据也有可能在使用时已经到达。但是，如果数据项在加载和使用之间被覆盖，则需要重新加载。请注意，这种情况不会比未提前加载的情况更糟。仔细研究这种技术就会发现，预取的工作原理与多线程的工作原理大致相同。在提前加载时，我们试图找出与其他线程没有资源依赖性（即使用相同寄存器）的独立执行线程。许多编译器都会积极尝试提前加载，以掩盖内存系统的延迟。

- **例2.8 通过预取隐藏延迟**

  考虑使用一个 *for* 循环添加两个向量 a 和 b 的问题。在循环的第一次迭代中，处理器请求 $a[0]$ 和 $b[0]$。由于这两个向量不在高速缓存中，处理器必须支付内存延迟。在处理这些请求的同时，处理器还请求 $a[1]$ 和 $b[1]$。假设每个请求在一个周期（$1ns$）内产生，而内存请求在 $100ns$ 内满足，那么在 100 次此类请求后，内存系统将返回第一组数据项。随后，每个周期将返回一对矢量分量。这样，在随后的每个周期中，都可以执行一次加法运算，而不会浪费处理器周期。

2.2.4 多线程与预取的权衡

多线程和预取似乎可以解决所有与内存系统性能有关的问题，但它们却受到内存带宽的严重影响。

- **例2.9 带宽对多线程程序的影响**

  考虑在一台机器上运行的计算，其时钟频率为 $1GHz$，缓存行为 4 字，缓存访问为单周期，到 DRAM 的延迟为 $100ns$。计算在 $1KB$ 缓存大小时的缓存命中率为 25%，在 $32KB$ 缓存大小时的缓存命中率为 90%。考虑两种情况：第一种是单线程执行，串行上下文可使用整个高速缓存；第二种是多线程执行，有 32 个线程，每个线程有 $1KB$ 的高速缓存空间。如果计算在每 $1ns$ 周期内发出一个数据请求，在第一种情况下，DRAM 的带宽需求为每 $10ns$ 一个字，因为其他字来自高速缓存（高速缓存命中率为 90%）。这相当于 $400MB/s$ 的带宽。在第二种情况下，对 DRAM 的带宽要求增加到每个线程每四个周期三个字（缓存命中率为 25%）。假设所有线程都表现出类似的缓存行为，这相当于 $0.75\ words/ns$，或 $3GB/s$。

例 2.9 说明了一个非常重要的问题，即由于每个线程的缓存驻留时间较小，多线程系统的带宽需求可能会大幅增加。在本例中，虽然 $400MB/s$ 的持续 DRAM 带宽是合理的，但 $3.0GB/s$ 的带宽已超过了目前大多数系统所能提供的带宽。此时，多线程系统就不再受延迟限制，而是受带宽限制。必须认识到，**多线程和预取只能解决延迟问题，往往会加剧带宽问题**。

另一个问题与有效使用预取和多线程所需的额外硬件资源有关。假设我们向寄存器进行了 10 次高级加载。这些加载要求 10 个寄存器在持续时间内保持空闲。如果中间的指令覆盖了寄存器，我们就必须重新加载数据。与没有预取的情况相比，这不会增加取数的延迟。但是，现在我们要两次获取相同的数据项，导致对内存系统带宽的需求增加了一倍，这种情况与例 2.9 中说明的缓存限制类似。通过使用更大的寄存器文件和高速缓存来支持预取和多线程，可以缓解这种情况。

> **Note**
>
> 内存系统的带宽和延迟能影响并行计算，改善其对并行计算的影响的方法如下：
>
> 1. 使用缓存可以降低延迟，但是存在缓存命中率的问题
> 2. 使用更高带宽的内存系统，但是意味着成本更高
> 3. 使用多线程技术，意味着对内存带宽的需求和缓存容量的需求增加
> 4. 使用预取技术提前加载需要的资源
>
> 此外内存的跨步访问也会导致并行计算的性能下降，同时可能导致缓存命中率降低。

2.3 并行计算平台的两部分

在前面的章节中，我们指出了影响串行或隐式并行程序性能的各种因素。当前微处理器的峰值性能和可持续性能之间的差距越来越大，内存系统性能的影响，以及许多问题的分布式性质，都是并行化的主要动机。现在，我们将从高层次介绍并行计算平台的要素，这些要素对于面向性能和可移植的并行编程至关重要。为了便于我们讨论并行平台，我们首先探讨了基于并行平台的逻辑组织和物理组织的两部分。逻辑组织指的是程序员对平台的看法，而物理组织指的是平台的实际硬件组织。从程序员的角度来看，并行计算的两个关键组成部分是表达并行任务的方法和指定这些任务之间交互的机制。前者有时也被称为控制结构，后者则被称为通信模型。

2.3.1 并行平台的控制结构

并行任务可以按不同的粒度进行指定。一个极端是，一组程序中的每个程序都视为一个并行任务。在另一个极端，每个程序中的单个指令也可视为并行任务。在这两个极端之间，有一系列用于指定程序控制结构和相应架构支持的模型。

- **例2.10 多处理器上单指令的并行性**

  请看下面这段添加两个向量的代码：

```
for (i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i];
}
```

  在本例中，循环的各种迭代是相互独立的，即 `c[0] = a[0] + b[0];`、`c[1] = a[1] + b[1];` 等等，都可以相互独立地执行。因此，如果有一种机制可以执行相同的指令，在本例中，在所有处理器上加上适当的数据，我们就可以更快地执行这个循环。

并行计算机中的处理单元要么在单个控制单元的集中控制下运行，要么独立工作。在被称为SIMD（Single Instruction Stream, Multiple Data Stream，单指令流多数据流）的体系结构中，单个控制单元向每个处理单元发送指令。图 2.3(a) 展示了典型的 SIMD 架构。在 SIMD 并行计算机中，所有处理单元同步执行同一指令。在例 2.10 中，加法指令被分派给所有处理器，并由它们同时执行。一些最早的并行计算机，如 Illiac IV、MPP、DAP、CM-2 和 MasPar MP-1 都属于这类计算机。最近，这一概念的变种被用于协同处理单元，如英特尔处理器中的 MMX 单元和 Sharc 等 DSP 芯片。英特尔奔腾处理器的 SSE（流 SIMD 扩展）提供了许多在多个数据项上执行相同指令的指令。这些架构改进依赖于底层计算的高度结构化（规则）性质，例如在图像处理和图形处理中，以提高性能。

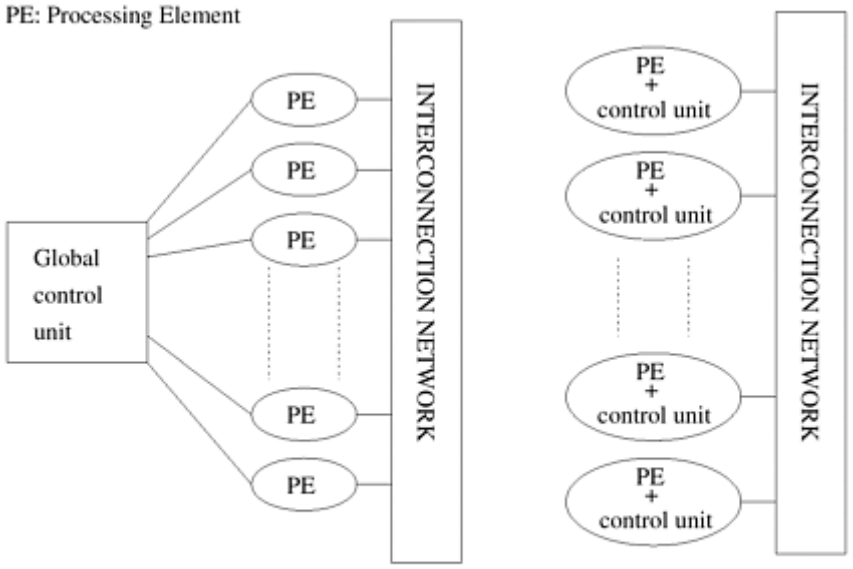


图2.3 典型的SIMD架构(a)和典型的MIMD架构(b)

虽然 SIMD 概念对数组等并行数据结构的结构化计算非常有效，但通常需要有选择地关闭对某些数据项的操作。因此，大多数 SIMD 编程范式都允许使用"活动掩码（Activity Mask）"。这是一个与每个数据项和操作相关联的二进制掩码，用于指定该数据项是否参与操作。`where (condition) then <stmnt> <elsewhere stmnt>` 等原语用于支持选择性执行。条件执行可能会损害 SIMD 处理器的性能，因此必须谨慎使用。

与 SIMD 体系结构不同，每个处理元件都能独立于其他处理元件执行不同程序的计算机被称为MIMD（Multiple Instruction Stream, Multiple Data Stream，多指令流多数据流）计算机。图 2.3(b) 描述了典型的 MIMD 计算机。这种模式的一个简单变体被称为SPMD (Single Program, Multiple Data，单程序多数据)模式，它依赖于同一程序的多个实例在不同数据上执行。不难看出，SPMD 模型与 MIMD 模型具有相同的表达能力，因为多个程序中的每个程序都可以插入一个大的 `if-else` 块中，其条件由任务标识符指定。SPMD 模型被许多并行平台广泛使用，只需最低限度的架构支持。这类平台包括 Sun Ultra 服务器、多处理器 PC、工作站集群和 IBM SP。

SIMD 计算机所需的硬件比 MIMD 计算机少，因为它们只有一个全局控制单元。此外，由于 SIMD 计算机只需存储一份程序，因此所需的内存也较少。相比之下，MIMD 计算机则在每个处理器上存储程序和操作系统。然而，SIMD 处理器作为通用计算引擎并不受欢迎，这可能要归咎于其专用的硬件架构、经济因素、设计限制、产品生命周期和应用特性。与此相反，支持 SPMD 模式的平台可以在短时间内利用廉价的现成组件构建，所需的工作量相对较少。SIMD 计算机需要大量的设计工作，因此产品开发时间较长。由于底层串行处理器变化如此之快，SIMD 计算机很快就会被淘汰。许多应用的不规则性也使 SIMD 体系结构不太适用。例 2.11 举例说明了 SIMD 体系结构在条件执行时资源利用率较低的情况。

- **例2.11 在 SIMD 架构上执行条件语句**

  考虑执行图 2.4 所示的条件语句。图 2.4(a) 中的条件语句分两步执行。第一步，所有 `B == 0` 的处理器执行指令 `C = A`，所有其他处理器处于空闲状态。第二步，执行指令的 `else` 部分（`C = A/B`）。在第一步中处于活动状态的处理器现在变成空闲。这说明了 SIMD 架构的一个缺点。

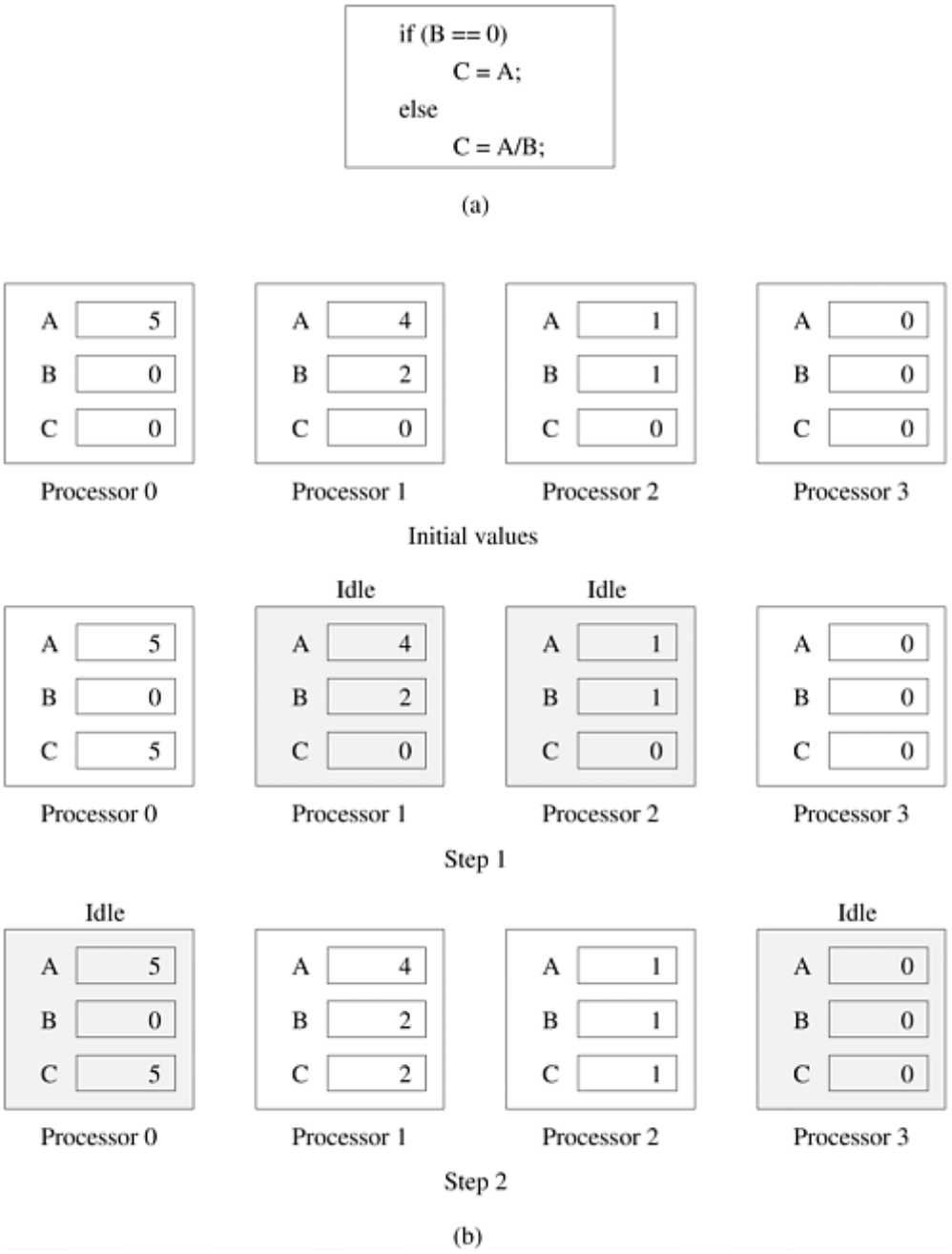


图2.4 在有四个处理器的 SIMD 计算机上执行条件语句：(a)条件语句；(b)分两步执行语句。

2.3.2 并行平台的通信模式

并行任务之间的数据交换主要有两种形式 - 访问共享数据和消息传递。

共享地址空间平台

并行平台的 "共享地址空间（shared-address-space）"支持所有处理器都能访问的通用数据空间。处理器通过修改存储在共享地址空间中的数据对象进行交互。支持 SPMD 编程的共享地址空间平台也被称为多处理器。共享地址空间平台中的内存可以是本地的（处理器独占），也可以是全局的（所有处理器共有）。如果处理器访问系统中任何内存字（全局或局部）所需的时间相同，则该平台被归类为UMA（Uniform Memory Access，统一内存访问）多核计算机。另一方面，如果访问某些内存字所需的时间比其他内存字长，该平台就被称为NUMA（Non-Uniform Memory Access，非统一内存访问）多计算机。图 2.5(a)和图 2.5(b)展示的是 UMA 平台，而图 2.5(c) 展示的是 NUMA 平台。

图 2.5(b) 展示了一个有趣的案例。在这里，访问高速缓存中的内存字比访问内存中的位置更快。然而，我们仍将其归类为 UMA 架构。原因在于目前所有的微处理器都有高速缓存分层。因此，如果考虑缓存访问时间，即使是单处理器也不能称为 UMA。因此，我们仅根据内存访问时间而非高速缓存访问时间来定义 NUMA 和 UMA 架构。SGI Origin 2000 和 Sun Ultra HPC 服务器等机器属于 NUMA 多处理器。UMA 和 NUMA 平台之间的区别非常重要。如果访问本地内存的成本比访问全局内存的成本低，算法就必须建立本地性，并相应地构建数据和计算结构。

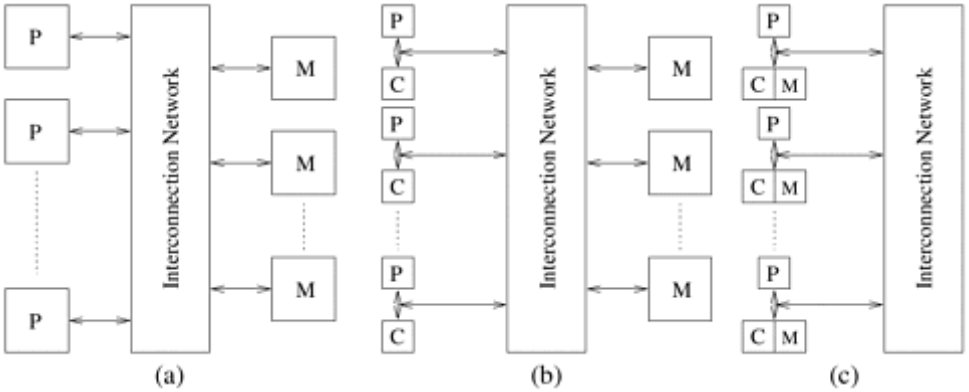


图2.5 典型的共享地址空间架构：(a)统一内存访问的共享地址空间计算机；(b)带缓存和内存的统一内存访问共享地址空间计算机；(c)仅带本地内存的非统一内存访问共享地址空间计算机。

全局内存空间的存在使此类平台的编程变得更加容易。程序员看不到所有只读交互，因为它们的编码方式与串行程序并无不同。这大大减轻了编写并行程序的负担。不过，读/写交互比只读交互更难编程，因为这些操作需要互斥才能实现并发访问。因此，共享地址空间编程范式（如线程（POSIX、NT）和指令（OpenMP））支持使用 `lock` 和相关机制进行同步。

处理器上缓存的存在也提出了两个或多个处理器同时操作单个内存字的多个副本的问题。在这种情况下，支持共享地址空间涉及两项主要任务：提供一种地址转换机制，用于定位系统中的内存字；确保对同一内存字的多个副本进行的并发操作具有定义明确的语义。后者也被称为 **高速缓存一致性（Cache Coherence）** 机制。第 2.4.6 节将详细讨论这一机制及其实现。支持高速缓存一致性需要大量的硬件支持。因此，一些共享地址空间机器只支持地址转换机制，而将确保一致性的任务留给了程序员。这类平台的本地编程模型包括 `get` 和 `put` 等原语。这些原语允许处理器获取（和放入）存储在远程处理器中的变量。

但是，如果该变量的一个副本发生变化，其他副本不会自动更新或失效。

必须注意两个常用但经常被误解的术语之间的区别 - 共享地址空间和共享内存计算机。**共享内存计算机（Shared-Memory Computer）** 一词历来用于在不同处理器之间物理共享内存的体系结构，即每个处理器对任何内存段的访问都是平等的。这与我们刚才讨论的 UMA 模式相同。这与分布式内存计算机形成鲜明对比，在分布式内存计算机中，不同的内存段在物理上与不同的处理元件相关联。共享内存计算机与分布式内存计算机的两部分与计算机的物理组织有关，将在第 2.4 节中详细讨论。无论是共享内存还是分布式内存，这两种物理模型都可以呈现不相连或共享地址空间平台的逻辑视图。分布式内存共享地址空间计算机与 NUMA 机器相同。

消息传递平台

消息传递平台的逻辑机器视图由 p 个处理节点组成，每个节点都有自己的专用地址空间。每个处理节点既可以是单处理器，也可以是共享地址空间的多处理器 - 这是现代消息传递并行计算机迅速发展的趋势。这种观点的实例自然来自集群工作站和非共享地址空间多处理器。在这些平台上，运行在不同节点上的进程之间的交互必须通过消息来完成，因此也被称为 **消息传递（Message Passing）**。这种消息交换用于在进程间传输数据、工作和同步操作。在最一般的形式中，消息传递范式支持在每 p 个节点上执行不同的程序。

由于交互是通过发送和接收信息完成的，因此这种编程范式的基本操作是 `send` 和 `receive`（各应用程序接口的相应调用可能不同，但语义基本相同）。此外，由于发送和接收操作必须指定目标地址，因此必须有一种机制来为执行并行程序的多个进程中的每个进程分配唯一的标识或 ID。程序通常可以使用 `whoami` 等函数来获取 ID，该函数会向调用进程返回其 ID。通常还需要一个函数来完成基本的消息传递操作 - `numprocs`，该函数用于指定参与集合的进程数量。有了这四个基本操作，就可以编写任何消息传递程序了。不同的消息传递应用程序接口，如消息传递接口（MPI）和并行虚拟机（PVM），以不同的函数名称

支持这些基本操作和各种更高级的功能。支持消息传递范例的并行平台包括 IBM SP、SGI Origin 2000 和工作站集群。

在具有相同节点数的共享地址空间计算机上模拟包含 p 个节点的消息传递架构非常容易。假定使用单处理器节点，可以将共享地址空间划分为 p 个互不相连的部分，并将其中一个部分专门分配给每个处理器。这样，一个处理器就可以通过写入或读取另一个处理器的分区来 "发送 "或 "接收 "信息，同时使用适当的同步原语来通知其通信伙伴它已完成数据的读取或写入。然而，在消息传递计算机上模拟共享地址空间架构的成本很高，因为访问另一个节点的内存需要发送和接收消息。

> **Note**
>
> 并行计算的平台主要分为两个部分：控制结构和通信模式
>
> **控制结构**:
>
> 1. **SIMD**（Single Instruction Stream, Multiple Data Stream，单指令流多数据流）
> 2. **MIMD**（Multiple Instruction Stream, Multiple Data Stream，多指令流多数据流）
> 3. **SPMD** (Single Program, Multiple Data，单程序多数据）- MIMD的一个简单变体
>
> **通信模式**:
>
> 1. 共享地址空间
>   - **UMA**（Uniform Memory Access，统一内存访问）
>   - **NUMA**（Non-Uniform Memory Access，非统一内存访问）
>   - 需要特别注意读写锁和缓存一致性的问题
> 2. 消息传递
>   - 通过 `send` / `receive` 等操作实现数据传输

2.4 并行平台的物理结构

在本节中，我们将讨论并行机的物理架构。我们将从理想架构开始，概述与实现这一模型相关的实际困难，并讨论一些传统架构。

2.4.1 理想并行计算机的结构

串行计算模式的自然扩展由 p 个处理器和一个大小不等的全局存储器（Random Access Machine，RAM）组成，所有处理器均可访问该存储器。所有处理器访问相同的地址空间。处理器共享一个时钟，但在每个周期内可能执行不同的指令。这种理想模式也被称为 **并行随机访问设备（Parallel Random Access Machine，PRAM）**。由于 PRAM 允许并发访问不同的内存位置，根据处理并发内存访问的方式，PRAM 可分为四个子类。

1. **独占读写 (Exclusive-Read, Exclusive-Write，EREW) PRAM**：在这类内存中，对内存位置的访问是排他性的。不允许同时进行读或写操作。这是最薄弱的 PRAM 模型，可提供最小的内存访问并发性。
2. **并发读取、独占写入（Concurrent-Read, Exclusive-Write，CREW）PRAM**：在该类中，允许对内存位置进行多次读取访问。但是，对一个内存位置的多次写入访问会被序列化。
3. **独占读取、并发写入 (Exclusive-Read, Concurrent-Write，ERCW) PRAM**：允许对一个内存位置进行多次写入访问，但多次读取访问会被序列化。
4. **并发读写 (Concurrent-Read, Concurrent-Write，CRCW) PRAM**：该类别允许对一个共同的内存位置进行多次读写访问。这是最强大的 PRAM 模型。

允许并发读取访问不会在程序中产生任何语义差异。但是，并发写入内存位置的访问则需要仲裁。有几种协议可用于解决并发写入问题。最常用的协议如下：

- **通常（Common）**：如果处理器试图写入的所有值都相同，则允许并发写入。
- **竞争（Arbitrary）**：在这种情况下，选取任意一个处理器进行写入操作，其他处理器则失败
- **优先级（Priority）**：即所有处理器被编入一个预定义的优先级列表，优先级最高的处理器成功，其他处理器失败。
- **总和（Sum）**：其中写入所有数量的总和，类似Reduce操作。（基于总和的写入冲突解决模型可以扩展到在正在写入的数量上定义的任何关联运算符）

理想模型的结构复杂性

考虑将 EREW PRAM 作为一台共享内存计算机来实现，它有 p 个处理器和 m 个字的全局存储器。处理器通过一组开关与内存相连。这些开关决定了每个处理器访问的内存字。在 EREW PRAM 中，只要内存字不被多个处理器同时访问，组合中的 p 个处理器都可以访问任何一个内存字。为确保这种连通性，开关总数必须为 $\Theta(mp)$。对于合理的内存大小，构建如此复杂的开关网络非常昂贵。因此，PRAM 计算模型在实践中是不可能实现的。

2.4.2 并行计算机互连网络

互连网络为处理节点之间或处理器与内存模块之间的数据传输提供了机制。互联网络的黑盒视图由 n 个输入和 m 个输出组成。输出可能有别于输入，也可能没有。典型的互连网络是利用链路和交换机构建的。链路对应于物理介质，如一组能够传输信息的电线或光纤。影响链路特性的因素有很多。对于基于导电介质的链路，导线之间的电容耦合限制了信号的传播速度。这种电容耦合和信号强度的衰减是链路长度的函数。

互连网络可分为 **静态** 和 **动态** 两种。静态网络由处理节点之间的点对点通信链路组成，也称为 **直接网络（Direct Network）**。而动态网络则是利用交换机和通信链路构建的。通信链路通过交换机动态地相互连接，在处理节点和内存库之间建立路径，动态网络也称为 **间接网络（Indirect Network）**。图 2.6(a) 展示了一个由四个处理元件或节点组成的简单静态网络。每个处理节点通过一个网络接口与其他两个节点

相连，形成网状配置。图 2.6(b) 展示了一个由四个节点组成的动态网络，这些节点通过交换机网络与其他节点相连。

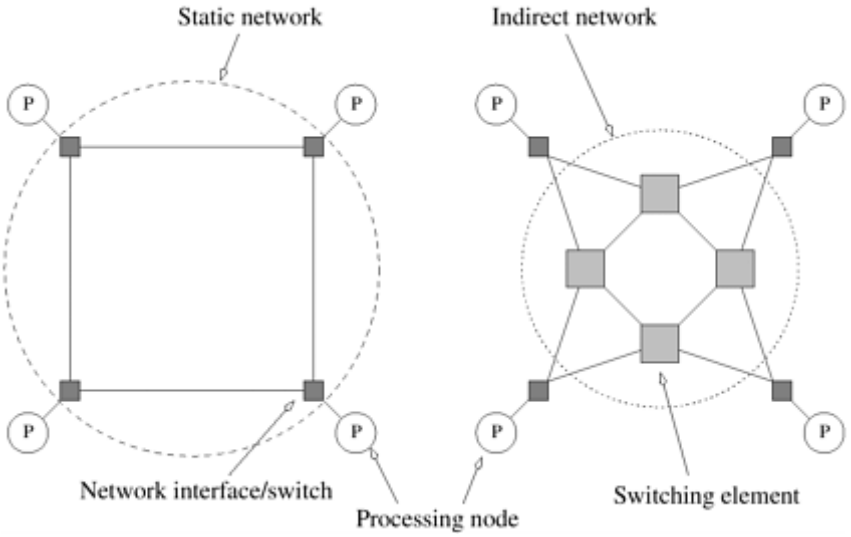


图2.6 互联网络的分类：(a) 静态网络；(b) 动态网络。

互连网络中的单个交换机由一组输入端口和一组输出端口组成。交换机提供一系列功能。交换机提供的最小功能是从输入端口到输出端口的映射。交换机的端口总数也称为交换机的 **度（Degree）**。交换机还可支持内部缓冲（当请求的输出端口繁忙时）、路由（缓解网络拥塞）和组播（多个端口上的相同输出）。从输入端口到输出端口的映射可通过基于物理横梁、多端口存储器、多路复用器-解多路复用器和多路复用总线的各种机制来实现。交换机的成本受映射硬件、外设硬件和封装成本的影响。映射硬件的成本通常是开关度的平方，外围硬件的成本是开关度的线性增长，而封装成本则是引脚数量的线性增长。

节点与网络之间的连接由网络接口提供。网络接口有输入和输出端口，将数据导入和导出网络。它通常负责对数据进行分组、计算路由信息、缓冲输入和输出数据以匹配网络和处理元件的速度，以及进行错误检查。处理元件与网络之间的接口位置也很重要。传统的网络接口挂在 I/O 总线上，而紧密耦合并行机的接口则挂在内存总线上。由于 I/O 总线通常比内存总线慢，因此后者可以支持更高的带宽。

> **Note**
>
> **直接网络（Direct Network）**：由处理节点之间的点对点通信链路组成，也称为静态网络
>
> **间接网络（Indirect Network）**：通信链路通过交换机动态地相互连接，在处理节点和内存库之间建立路径，也称为动态网络
>
> **度（Degree）**：交换机的端口总数

2.4.3 网络拓扑

互联网络中使用了多种网络拓扑结构。这些拓扑结构试图在成本和可扩展性与性能之间进行权衡。虽然纯拓扑结构具有吸引人的数学特性，但在实践中，互连网络往往是本节讨论的纯拓扑结构的组合或修改。

总线网络

总线网络可能是最简单的网络，它由所有节点共用的共享介质组成。总线具有一个理想特性，即网络成本与节点数 p 成线性关系。此外，网络中任意两个节点之间的距离是恒定的（$O(1)$）。总线也是在节点间广播信息的理想选择。由于传输介质是共享的，因此与点对点信息传输相比，广播的开销很小。然而，随着节点数量的增加，总线的带宽限制了网络的整体性能。典型的基于总线的机器只能有几十个节点。Sun 企业服务器和基于共享总线的英特尔奔腾多处理器就是这种架构的例子。

在典型的程序中，访问的大部分数据都是节点本地数据，利用这一特性可以降低对总线带宽的需求。对于此类程序，可以为每个节点提供一个缓存。私人数据在节点上缓存，只有远程数据通过总线访问。

- **例2.12 利用缓存减少共享总线带宽**

  图 2.7(a) 展示了 p 个处理器共享一条总线到内存的情况。假设每个处理器访问 k 个数据项，每次数据访问耗时 t_{cycle}，则执行时间的下限为 $t_{cycle} \times kp$ 秒。现在考虑图 2.7(b) 的硬件组织结构。假设 50% 的内存访问（$0.5k$）是对本地数据的访问。这些本地数据位于处理器的专用内存中。假设访问专用内存的时间与访问全局内存的时间相同，即 t_{cycle}。在这种情况下，总执行时间的下限为

$0.5 \times t_{cycle} \times k + 0.5 \times t_{cycle} \times kp$。这里，第一项是访问本地数据的结果，第二项是访问共享数据的结果。不难看出，当 p 变大时，图 2.7(b) 的组织结构会导致下限接近 $0.5 \times t_{cycle} \times kp$。与图 2.7(a) 相比，执行时间下限提高了 50%。

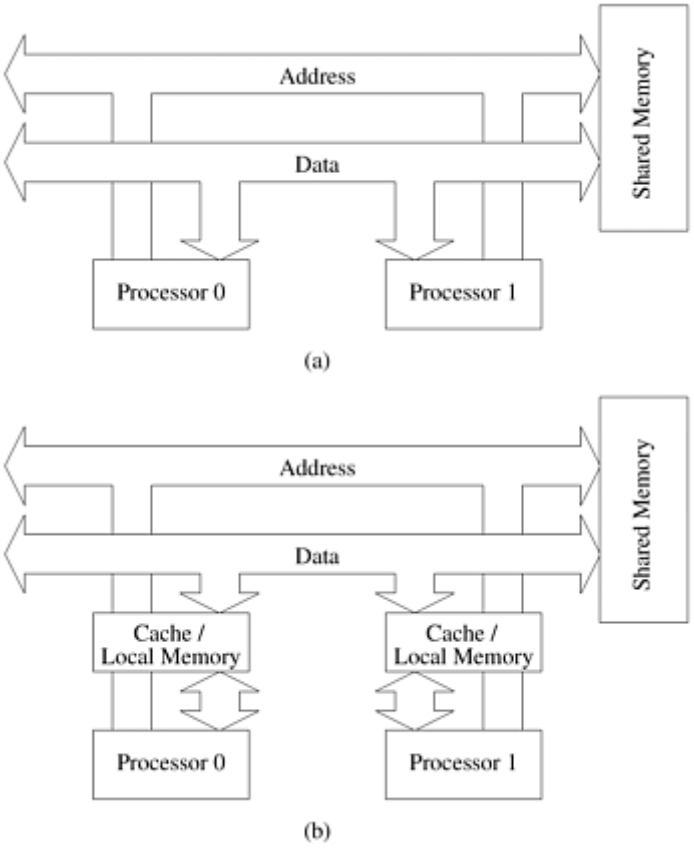


图2.7 基于总线的互连 (a) 无本地缓存；(b) 有本地内存/缓存。

实际上，共享数据和私有数据的处理方式更为复杂。第 2.4.6 节将简要讨论缓存一致性问题。

> **Note**
>
> **优点**：对广播场景支持度高、任意两点的距离均为恒定的$O(1)$
>
> **缺点**：随着节点数量的增加，总线的带宽限制了网络的整体性能

交叉网络

将 p 个处理器连接到 b 个内存库的简单方法是使用交叉条网络。如图 2.8 所示，交叉网络采用网格状的开关或交换节点。横条网络是一种非阻塞网络，即一个处理节点与一个内存库的连接不会阻塞其他处理节点与其他内存库的连接。

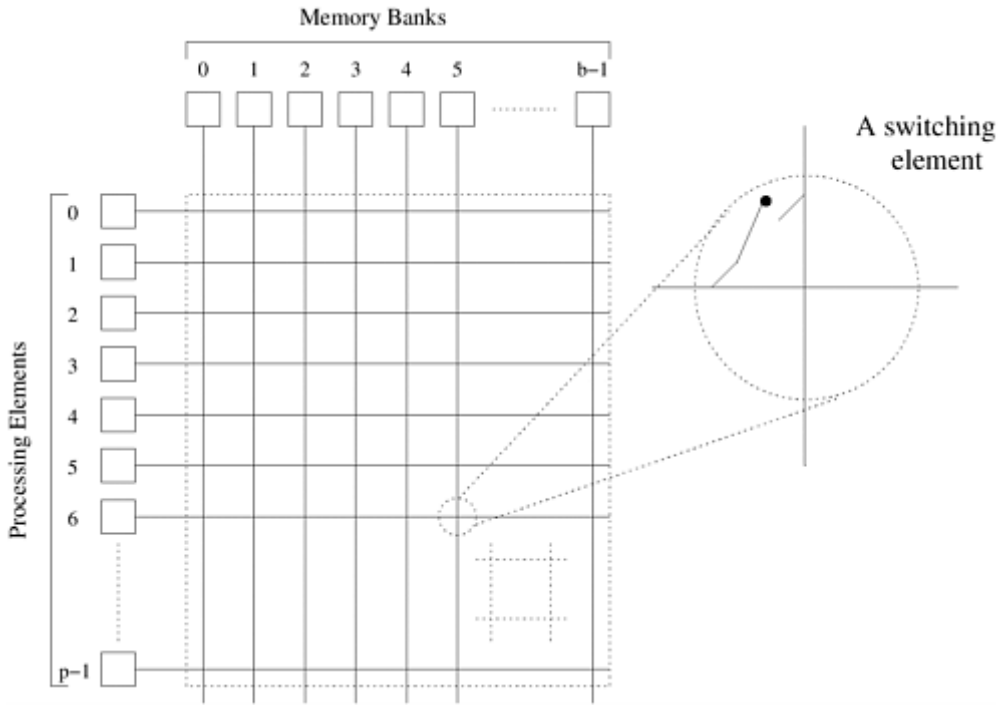


图2.8 连接 p 个处理器和 b 个内存库的完全无阻塞的交叉网络

实现这样一个网络所需的交换节点总数为 $\Theta(pb)$。可以合理地假设内存库 b 的数量至少为 p；否则，在任何给定时间内，都会有一些处理节点无法访问任何内存库。因此，随着 p 值的增加，交换网络的复杂性（组件数）也会随着 $\Omega(p^2)$ 的增加而增加。随着处理节点数量的增加，这种交换复杂性很难在高数据速率下实现。因此，就成本而言，交叉网络的可扩展性并不高。

> **Note**
>
> **优点**：结构简单、小规模性能较好
>
> **缺点**：随着处理节点数量的增加，交换网络的复杂性也会随着增加，在高数据速率下实现，可扩展性不高

多级网络

交叉互连网络在性能方面是可扩展的，但在成本方面是不可扩展的。相反，共享总线网络在成本方面是可扩展的，但在性能方面是不可扩展的。在这两个极端之间有一类中间网络，称为 **多级互连网络（Multistage Interconnection）**。就性能而言，它比总线网络更具可扩展性，而就成本而言，它比交叉网络更具可扩展性。

由 p 个处理节点和 b 个内存库组成的多级网络的一般示意图如图 2.9 所示。常用的多级连接网络是 **Omega 网络（Omega Network）**。该网络由对数 p 级组成，其中 p 是输入（处理节点）和输出（存储库）的数量。Omega 网络的每一级都由连接 p 个输入和 p 个输出的互连模式组成；如果以下条件为真，则输入 i 和输出 j 之间存在连接：

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i + 1 - p, & p/2 \le i \le p - 1 \end{cases} \tag{1}$$

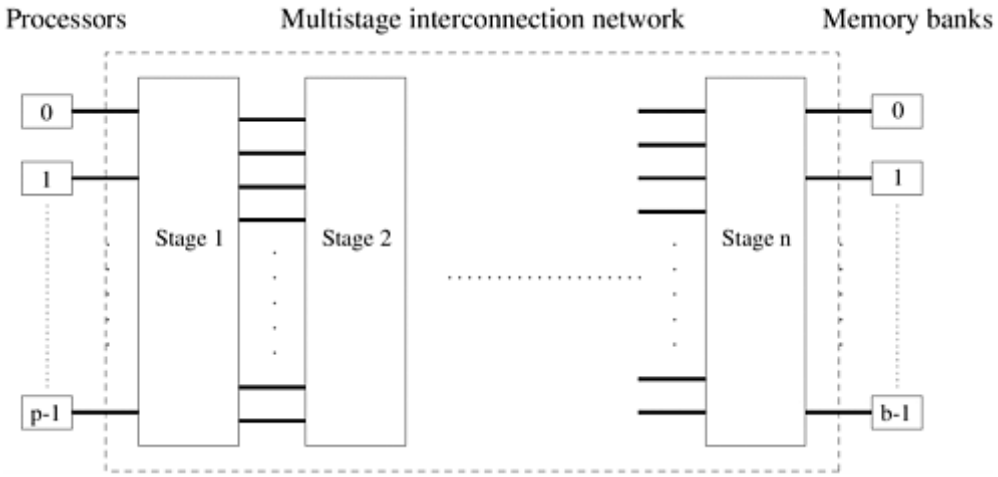


图2.9 典型的多级互连网络示意图

上式表示对 i 的二进制表示进行左旋转操作，从而得到 j。图 2.10 显示了八个输入和输出的 **完美随机（Perfect Shuffle）** 互联模式。在Omega网络的每个阶段，完美随机互联模式都会输入一组 $p/2$ 开关或开关节点。每个开关处于两种连接模式之一。在其中一种模式下，输入端直通输出端，如图 2.11(a)所示。这就是所谓的 **直通连接（Pass-Through）** 。在另一种模式中，开关节点的输入被交叉后送出，如图 2.11(b)所示。这就是所谓的 **交叉连接（Cross-Over）** 。

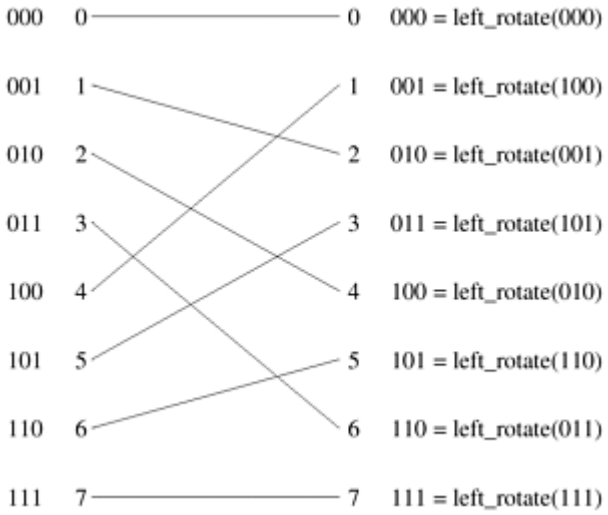


图2.10 八个输入和输出的完美随机互连

图2.11 2 x 2 交换机的两种交换配置：(a)直通；(b)交叉

Omega网络有 $p/2 \times \log p$ 个交换节点，其成本按 $\Theta(p \log p)$ 增长。需要注意的是，这一成本小于完整交叉条网络的 $\Theta(p^2)$ 成本。图 2.12 显示了八个处理器（左侧二进制数表示）和八个内存库（右侧二进制数表示）的Omega网络。Omega网络中的数据路由是通过一个简单的方案实现的。假设 s 是一个处理器的二进制表示，该处理器需要将一些数据写入内存库 t。如果 s 和 t 的最有效位相同，则数据由交换机以直通模式路由。如果这些位不同，则数据以交叉模式路由。在下一个切换阶段，使用下一个最显著位重复这一方案。遍历对数 p 个阶段时，将使用 s 和 t 二进制表示中的所有对数 p 位。

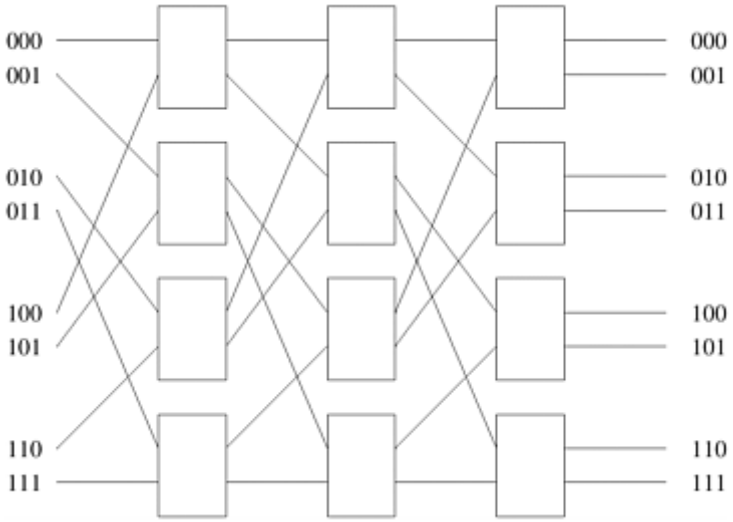


图2.12 一个完整的Omega网络连接八个输入和八个输出

图 2.13 显示了从处理器 2（010）到内存库 7（111）以及从处理器 6（110）到内存库 4（100）的Omega网络数据路由。该图还说明了该网络的一个重要特性。当处理器 2 (010) 与存储器组 7 (111) 通信时，会阻断处理器 6 (110) 到存储器组 4 (100) 的路径。两条通信路径都使用通信链路 *AB*。因此，在Omega网络中，一个处理器对内存库的访问可能会阻止另一个处理器对另一个内存库的访问。具有这种特性的网络被称为 **阻塞网络（Blocking Networks）** 。

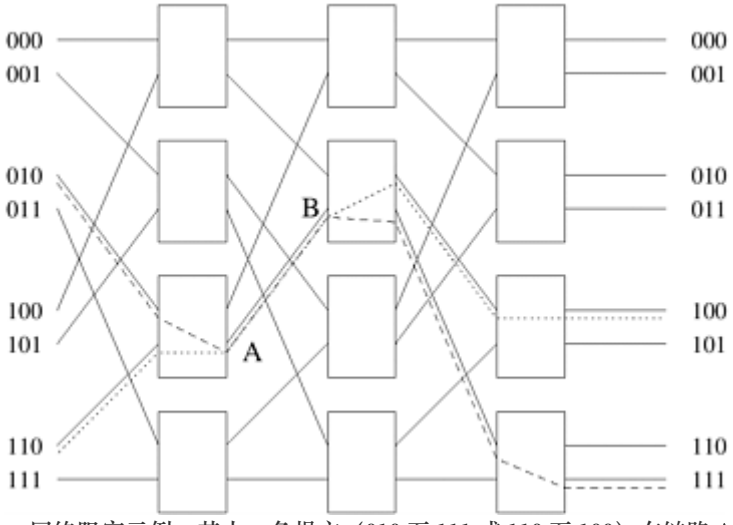


图2.13 Omega网络阻塞示例：其中一条报文（010 至 111 或 110 至 100）在链路 AB 上被阻塞

> **Note**
>
> **优点**：成本和性能都比较综合（相比较于总线网络和交叉网络）
>
> **缺点**：多级网络属于阻塞网络

全连接网络

在 **完全连接的网络（Completely-Connected Network）** 中，每个节点都与网络中的其他节点有直接的通信连接。图 2.14(a) 展示了一个由八个节点组成的完全连接网络。这个网络非常理想，因为节点之间存在通信链路，一个节点只需一步就能向另一个节点发送信息。完全连接网络是交叉交换网络的静态网络，因为在这两种网络中，任何输入/输出对之间的通信都不会阻碍其他任何输入/输出对之间的通信。

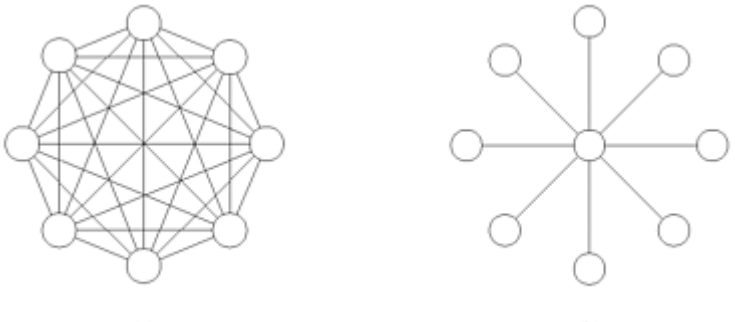


图2.14 (a)由8个节点组成的完全连接网络；(b)由9个节点组成的星形连接网络

> **Note**
>
> **优点**：无阻塞，任意两点之间均可以通信
>
> **缺点**：存在大量的链接，实现起来较困难

星形连接网络

在 **星形连接网络（Star-Connected Network）** 中，一台处理器充当中央处理器。其他每个处理器都有一条通信链路与该处理器相连。图 2.14(b) 显示了一个由9个处理器组成的星形连接网络。星形连接网络与总线型网络类似。任何一对处理器之间的通信都要经过中央处理器，就像共享总线构成了总线型网络中所有通信的媒介一样。中央处理器是星型拓扑结构中的瓶颈。

> **Note**
>
> **优点**：结构简单，实现容易
>
> **缺点**：任何一对处理器之间的通信都要经过中央处理器，中央处理器是星型拓扑结构中的瓶颈

线性阵列、网格和k-d网格

由于完全连接的网络中存在大量链接，因此通常使用较稀疏的网络来构建并行计算机。此类网络的一个系列横跨线性阵列和超立方体空间。**线性阵列（Linear Arrays）** 是一种静态网络，其中每个节点（除了两端的两个节点）都有两个邻居，左右各一个。线性阵列（图 2.15(a)）的一个简单扩展是环或一维环（图 2.15(b)）。环形结构在线性阵列的两端之间有环绕连接。在这种情况下，每个节点都有两个邻居。

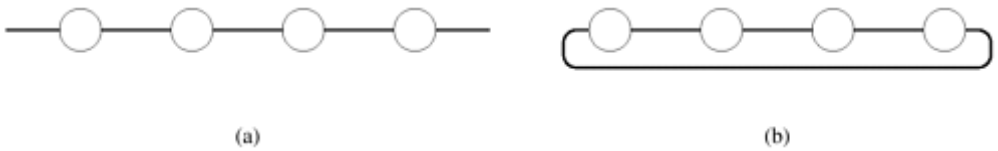


图2.15 线性阵列：(a)无环绕链接；(b)有环绕链接

图 2.16(a) 所示的二维 **网格（Mesh）** 是线性数组向二维的扩展。每个维度都有$\sqrt{p}$个节点，节点由两个元组 (i,j) 标识。每个节点（外围节点除外）都与其他四个节点相连，这些节点的索引在任意维度上都相差一个。二维网格具有可在二维空间中布局的特性，因此从布线的角度来看很有吸引力。此外，各种规则结构的计算都能非常自然地映射到二维网格上。因此，二维网格经常被用作并行机器中的互连。如图 2.16(b) 所示，二维网格可通过环绕链接形成二维环。如图 2.16(c)所示，三维立方体是二维网格向三维的扩展。三维立方体中的每个节点元素（外围的节点元素除外）都与其他六个节点相连，三个维度各两个。通常在并行计算机上执行的各种物理模拟（如三维天气建模、结构建模等）都可以自然地映射到三维网络拓扑结构中。因此，三维立方体常用于并行计算机的互连网络中（如 Cray T3E）。

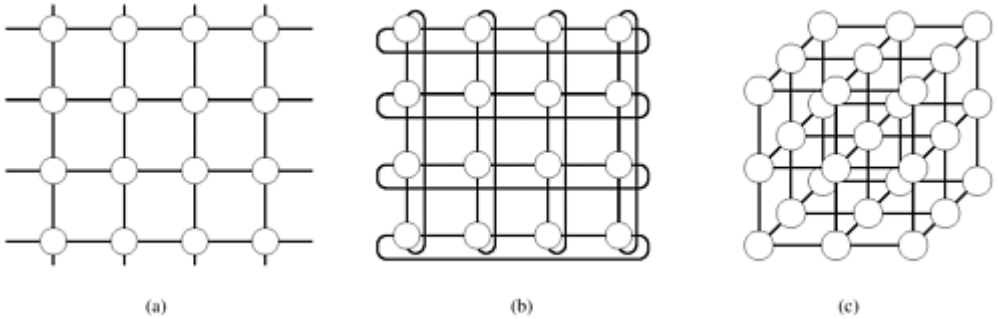


图2.16 二维和三维网格：(a)无环绕的二维网格；(b)有环绕链接的二维网格；(c)无环绕的三维网格

一般的k-d网格是指由 d 个维度组成的拓扑结构，每个维度有 k 个节点。正如线性阵列构成了k-d网格家族的一个极端，另一个极端则由一种名为超立方体的有趣拓扑构成。超立方体拓扑结构每个维度有两个节点，维数为对数 p。超立方体的构造如图 2.17 所示。零维超立方由 2^0 个节点组成。一个一维超立方体由两个零维超立方体连接而成。一个有四个节点的二维超立方体是由两个一维超立方体通过连接相应的节点构造而成的。一般来说，一个 d 维超立方体是由两个 $(d-1)$ 维超立方体的相应节点连接而成的。图 2.17 展示了四维超立方体中最多 16 个节点的情况。

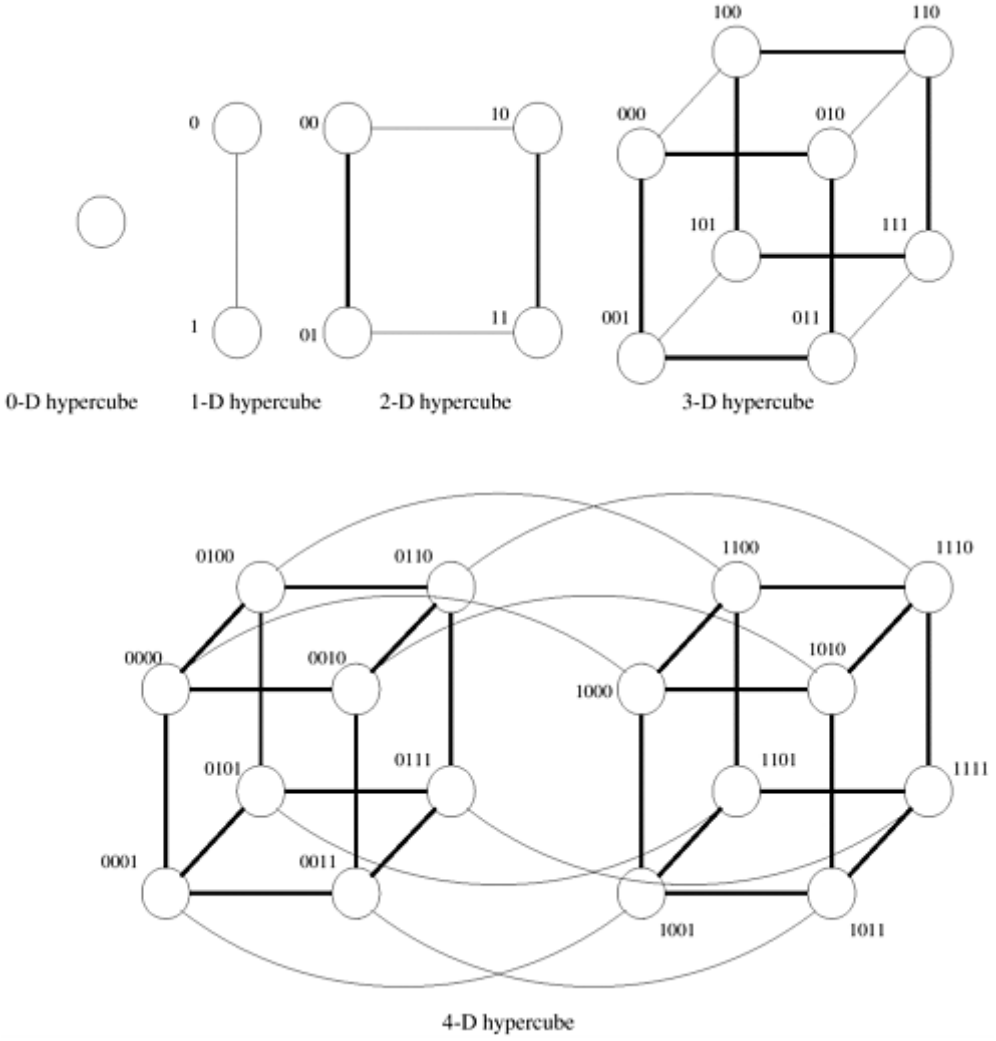


图2.17 从低维度超立方体构建超立方体

为超立方体中的节点导出一个编号方案是非常有用的。从超立方体的构造中可以推导出一个简单的编号方案。如图 2.17 所示，如果我们有两个由 $p/2$ 个节点组成的子立方体的编号，我们可以通过在其中一个子立方体的标签前加上 "0"，在另一个子立方体的标签前加上 "1"，来推导出由 p 个节点组成的立方体的编号方案。这种编号方案有一个有用的特性，即两个节点之间的最小距离由两个标签中不同的比特数给出。例如，标记为 0110 和 0101 的节点相距两个链路，因为它们在后两个比特位置上不同。这一特性有助于为超立方架构推导出许多并行算法。

> **Note**
>
> **优点**：比全连接网络稀疏，适用于特定通信算法
>
> **缺点**：结构比较复杂，一般是静态网络

树状网络

树状网络是指任意一对节点之间只有一条路径的网络。线性阵列和星形连接网络都是树状网络的特例。图 2.18 显示了基于完整二叉树的网络。静态树状网络的每个节点都有一个处理元件（图 2.18(a)）。树状网络也有动态网络。在动态树状网络中，中间层的节点是切换节点，叶节点是处理元件（图 2.18(b)）。

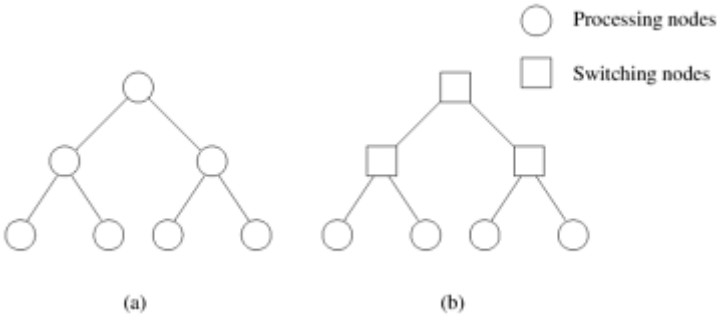


图2.18 完整的二叉树网络：(a)静态树网络；(b)动态树网络

要在树中路由信息，源节点会将信息向上发送，直到到达包含源节点和目的节点的最小子树根部的节点。然后，信息沿着树的方向向目的地节点发送。

树状网络的较高层存在通信瓶颈。例如，当节点左侧子树的许多节点与右侧子树的节点通信时，根节点必须处理所有信息。在动态树状网络中，可以通过增加通信链路的数量和切换更靠近根节点的节点来缓解这一问题，这种网络也称为 **胖树（Fat Tree）**，如图 2.19 所示。

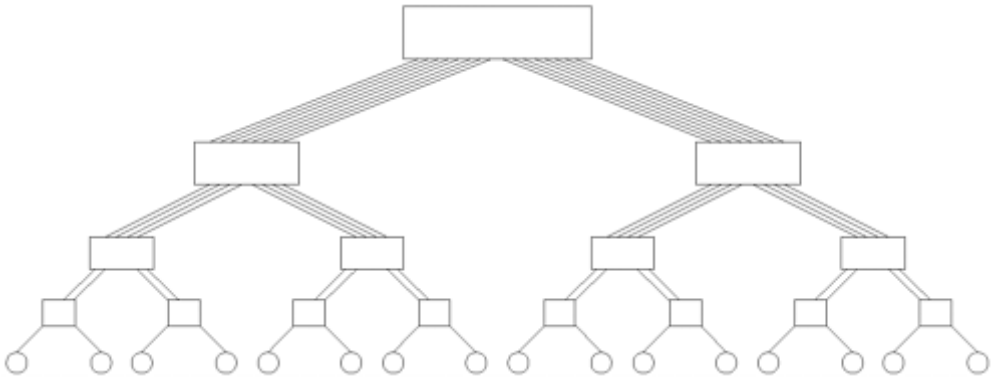


图2.19 由 16 个处理节点组成的胖树网络

> **Note**
>
> **优点**：结构简单，既可以是静态网络也可以是动态网络
>
> **缺点**：较高层存在通信瓶颈，在动态树状网络中，可以通过增加通信链路的数量和切换更靠近根节点的节点来缓解这一问题，这种网络也称为 **胖树**。

2.4.4 评估静态互连网络

我们现在讨论用于描述静态互连网络成本和性能的各种标准。我们将使用这些标准来评估上一小节介绍的静态网络。

直径（Diameter） 网络的直径是网络中任意两个处理节点之间的最大距离。两个处理节点之间的距离定义为它们之间的最短路径（以链接数计）。完全连接网络的直径为 1，星形连接网络的直径为 2。环形网络的直径为 $\lfloor p/2 \rfloor$。无环绕连接的二维网格的直径为对角线上两个节点的直径 $2(\sqrt{p}-1)$，环绕网格的直径为 $2\lfloor \sqrt{p}/2 \rfloor$。超立方体连接网络的直径为 $\log p$，因为两个节点标签最多相差 $\log p$ 个位置。完整二叉树的直径为 $2\log((p+1)/2)$，因为两个通信节点可能分别位于根节点的不同子树中，信息可能要一直传到根节点，然后再传到另一个子树。

连通性（Connectivity） 网络的连通性是衡量两个处理节点之间路径多寡的标准。具有高连接性的网络是理想的，因为它能减少对通信资源的争夺。连通性的一个衡量标准是，要将网络分成两个互不相连的网络，必须从网络中移除的弧的最小数目，这就是网络的 **弧连通性（Arc Connectivity）**。对于线性阵列以及树形和星形网络，弧连通性为 1。对于环形和无环绕的二维网格，弧连通性为 2；对于二维环绕网格，弧连通性为 4；对于 d 维超立方体，弧连通性为 d。

对分宽度和对分带宽（Bisection Width and Bisection Bandwidth） 网络的 **对分宽度（Bisection Width）** 是指将网络分成相等的两半所必须移除的通信链路的最小数目。环网的分段宽度为 2，因为任何对分都只跨越两个通信链路。同样，不带环绕连接的二维 p 节点网格的分段宽度为 $\sqrt{p}$，带环绕连接的二维 p 节点网格的分段宽度为 $2\sqrt{p}$。树形和星形的对分宽度为 1，而由 p 个节点组成的完全连接网络的分段宽度为 $p^2/4$。超立方体的分段宽度可以从其构造中推导出来。我们通过连接两个 $(d-1)$ 维超立方体的相应链接来构建一个 d 维超立方体。由于每个子立方体包含 $2^{(d-1)}$ 或 $p/2$ 个节点，因此至少有 $p/2$ 条通信链路必须穿过超立方体的任何分区，将其分成两个子立方体。

连接两个节点的链路可同时通信的比特数称为 **信道宽度（Channel Width）**。信道宽度等于每个通信链路中的物理线数。单根物理线缆传输比特的峰值速率称为 **信道速率（Channel Rate）**。通信链路两端数据通信的峰值速率称为 **信道带宽（Channel Bandwidth）**，信道带宽是信道速率和信道宽度的乘积。

网络的 **对分带宽（Bisection Bandwidth）** 被定义为网络任何两部分之间允许的最小通信量。它是对分宽度与信道带宽的乘积。网络的分段带宽有时也称为 **交叉带宽（Crosssection Bandwidth）**。

成本（Cost） 评估网络成本的标准有很多。定义网络成本的一种方法是网络所需的通信链路数或电线数。线性阵列和树状网络只需 $p-1$ 个链接来连接 p 个节点。一个 d 维环绕网格有 dp 个链接。超立方体连接网络有 $(p\log p)/2$个链接。

网络的对分带宽也可用作成本的衡量标准，因为它提供了二维包装中面积或三维包装中体积的下限。如果网络的对分带宽为 w，则二维包装的面积下限为 $\Theta(w^2)$，三维包装的体积下限为 $\Theta(w^{3/2})$。根据这一标准，超立方体和完全连接网络比其他网络更昂贵。

表2.1 连接 p 个节点的各种静态网络拓扑结构的特点汇总

网络	直径	对分宽度	弧连通性	成本（链接数）
全连接	1	$p^2/4$	$p-1$	$p(p-1)/2$
星形连接	2	1	1	$p-1$
完全二叉树	$2\log((p+1)/2)$	1	1	$p-1$
线性阵列	$p-1$	1	1	$p-1$
无环二维网络	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
有环二维网络	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
超立方体	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
有环k-d立方体	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

我们在表 2.1 中总结了各种静态网络的特点，其中突出强调了各种性价比权衡。

2.4.5 评估动态互连网络

动态网络的一些评估指标源自静态网络的相应指标。由于经过交换机的信息会产生开销，因此除了处理节点外，将每个交换机视为网络中的一个节点也是合乎逻辑的。网络直径现在可以定义为网络中任意两个节点之间的最大距离。这表示信息在选定的一对节点之间传输时将遇到的最大延迟。实际上，我们希望该指标是任意两个处理节点之间的最大距离；不过，对于所有相关网络而言，这等同于任意（处理或交换）一对节点之间的最大距离。

动态网络的连通性可以用节点连通性（Node Connectivity）或边缘连通性（Edge Connectivity）来定义。节点连通性是将网络分割成两个部分所必须失效（从网络中移除）的最小节点数。和以前一样，我们应该只考虑切换节点（而不是所有节点）。不过，考虑所有节点可以很好地近似动态网络中路径的多重性。网络的弧连通性可以类似地定义为：要将网络分割成两个无法到达的部分，必须失效（从网络中删除）的最少边数。

动态网络的对分宽度必须比直径和连通性定义得更精确。就对分宽度而言，我们考虑将 p 个处理节点划分为两个相等部分的任何可能情况。请注意，这并不限制交换节点的划分。对于每个这样的分区，我们选择一个开关节点的诱导分区，使穿过该分区的边的数量最小。任何此类分区的最小边数就是动态网络的分段宽度。另一种直观的分段宽度思维方式是，必须从网络中移除最少数量的边，才能将网络划分为处理节点数量相同的两半。我们在下面的示例中进一步说明这一概念：

- **例2.13 动态网络的对分宽度**

  请看图 2.20 所示的网络。我们在此展示了 A、B 和 C 三个分段，每个分段都将网络划分为两组，每组有两个处理节点。请注意，这些分区并不一定要将网络节点平均分配。在本例中，每个分区都会产生四条切边。因此，我们可以得出结论：该图的分割宽度为 4。

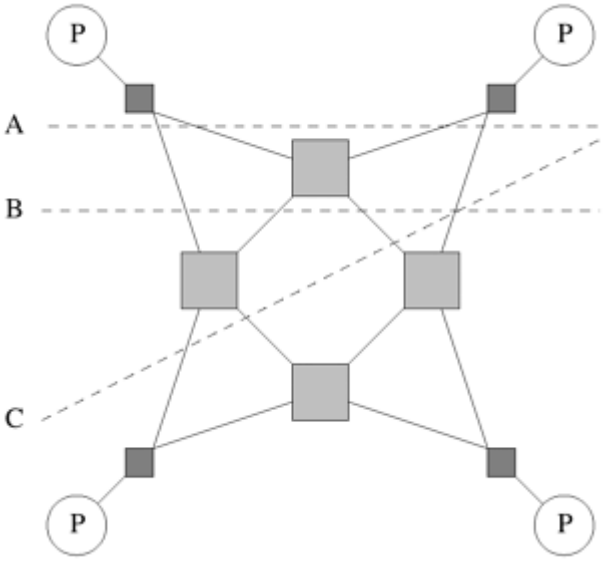


图2.20 动态网络的分割宽度是通过检查处理节点的各种等分区，并选择穿越该分区的最小边数计算得出的。在本例中，每个分区产生的边切为四条，因此，该图的分段宽度为4

与静态网络一样，动态网络的成本由链路成本和交换机成本决定。在典型的动态网络中，交换机的阶数是恒定的。因此，链路和交换机的数量近似相同。此外，在典型网络中，交换机成本超过链路成本。因此，动态网络的成本通常由网络中交换节点的数量决定。

表 2.2 总结了各种动态网络的特点：

表2.2 连接 p 个处理节点的各种动态网络拓扑结构的特点概述

网络	直径	对分宽度	弧连通性	成本（链接数）
交叉网络	1	p	1	p^2
Omega网络	$\log p$	$p/2$	2	$p/2$
动态树	$2\log p$	1	2	$p-1$

2.4.6 多处理器系统的缓存一致性

虽然互连网络提供了基本的消息（数据）通信机制，但在共享地址空间计算机中，还需要额外的硬件来保持多个数据副本之间的一致性。具体来说，如果存在两个数据副本（在不同的缓存/内存元件中），我们如何确保不同的处理器以符合预定义语义的方式对这些副本进行操作？

与单处理器系统相比，多处理器系统中保持缓存一致性的问题要复杂得多。这是因为除了单处理器系统中的多个副本外，还可能有多个处理器在修改这些副本。如图 2.21 所示的一个简单场景。两个处理器 P_0 和 P_1 通过共享总线连接到一个全局可访问内存。两个处理器都加载了同一个变量。现在该变量有三个副本。一致性机制现在必须确保在这些副本上执行的所有操作都是可串行化的（即存在某种与并行时间表相对应的串行指令执行顺序）。当处理器更改其变量副本的值时，必须发生以下两种情况之一：其他副本必须失效，或者其他副本必须更新。如果做不到这一点，其他处理器可能会使用变量的错误（陈旧）值工作。这两个协议被称为失效协议和更新协议，如图 2.21(a)和图 2.21(b)所示。

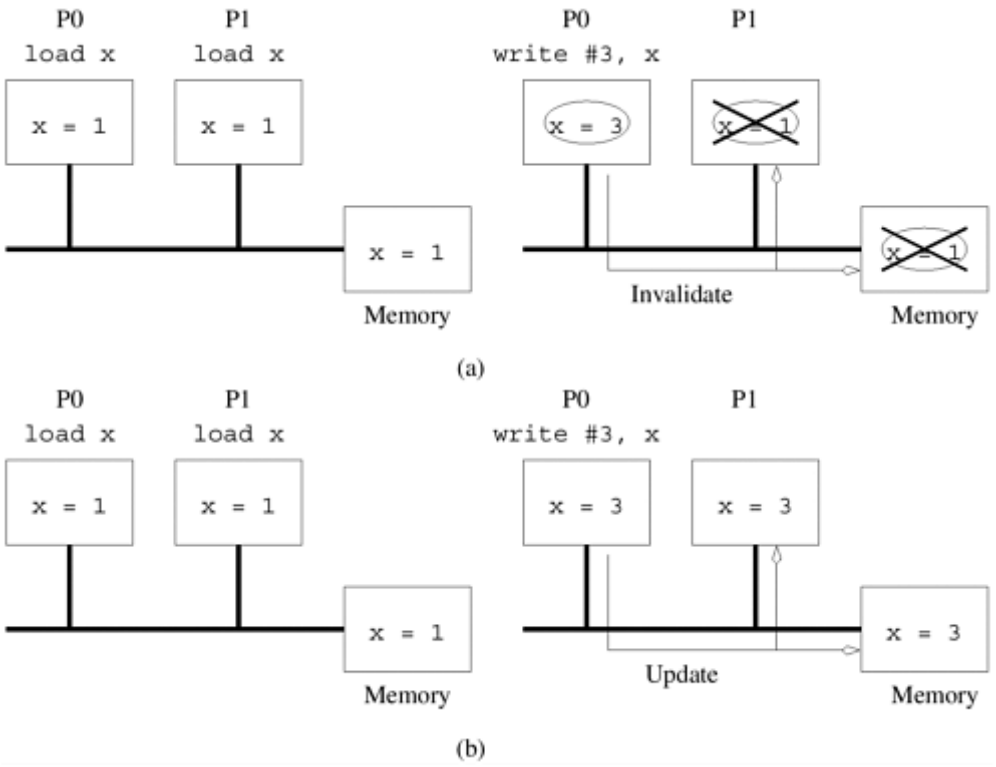


图2.21 多处理器系统的缓存一致性：(a)失效协议；(b)共享变量的更新协议

在更新协议中，每当写入一个数据项，系统中的所有副本都会被更新。因此，如果一个处理器只读取一次数据项而从不使用它，那么其他处理器对该数据项的后续更新就会造成过多的源延迟和网络带宽开销。另一方面，在这种情况下，失效协议会在远程处理器的第一次更新中使数据项失效，因此无需对该副本执行后续更新。

影响这些协议性能的另一个重要因素是 **错误共享（False Sharing）**。错误共享指的是不同处理器更新同一高速缓存行的不同部分。因此，虽然更新不是在共享变量上进行的，但系统并不能检测到这一点。在失效协议中，当一个处理器更新其缓存行的一部分时，该缓存行的其他副本就会失效。当其他处理器试图更新其缓存行部分时，必须从远程处理器获取该行。不难看出，虚假共享会导致高速缓存行在不同处理器之间 "乒乓 "作响。在更新协议中，这种情况稍好一些，因为所有读取操作都可以在本地执行，而写入操作必须更新。这就节省了一个无效操作，否则就会浪费。

失效和更新方案之间的权衡是通信开销（更新）和空闲（失效中的停滞）之间的经典权衡。目前的高速缓存一致性机器通常依赖于失效协议。因此，我们在讨论多处理器高速缓存系统的其余部分时，将假定使用失效协议。

使用无效协议保持一致性（Maintaining Coherence Using Invalidate Protocols） 单一数据项的多个副本可通过跟踪副本数量和每个副本的状态来保持一致。我们在此讨论与数据项相关的一组可能状态，以及触发这些状态之间转换的事件。请注意，这组状态和转换并不是唯一的。我们还可以定义其他状态和相关的转换。

让我们重温一下图 2.21 中的示例。最初，变量 x 位于全局内存中。两个处理器执行的第一步都是对该变量进行加载操作。此时，变量的状态可以说是 **共享的（Shared）**，因为它被多个处理器共享。当处理器 P_0 对该变量执行存储操作时，它会将该变量的所有其他副本标记为无效。它还必须将自己的副本标记为已修改或已脏。这样做是为了确保其他处理器对该变量的所有后续访问都由处理器 P_0 而不是内存来完成。此时，处理器 P_1 对 x 执行了另一个加载操作，处理器 P_1 尝试获取该变量，由于处理器 P_0 已将该变量标记为 **Dirty**，因此处理器 P_0 为该请求提供服务。处理器 P_1 和全局存储器中该变量的副本被更新，变量重新进入共享状态。因此，在这个简单的模型中，缓存行会经历 **共享（Shared）**、**无效（Invalid）** 和 **脏（Dirty）** 三种状态。

简单三态协议的完整状态图如图 2.22 所示。实线表示处理器的操作，虚线表示一致性操作。例如，当处理器对无效数据块执行读操作时，该数据块将被获取，并从无效数据块过渡到共享数据块。同样，如果处理器对共享区块执行写操作，一致性协议会在该区块上传播 `C_write`（一致性写）。这将触发所有其他区块从共享到无效的转换。

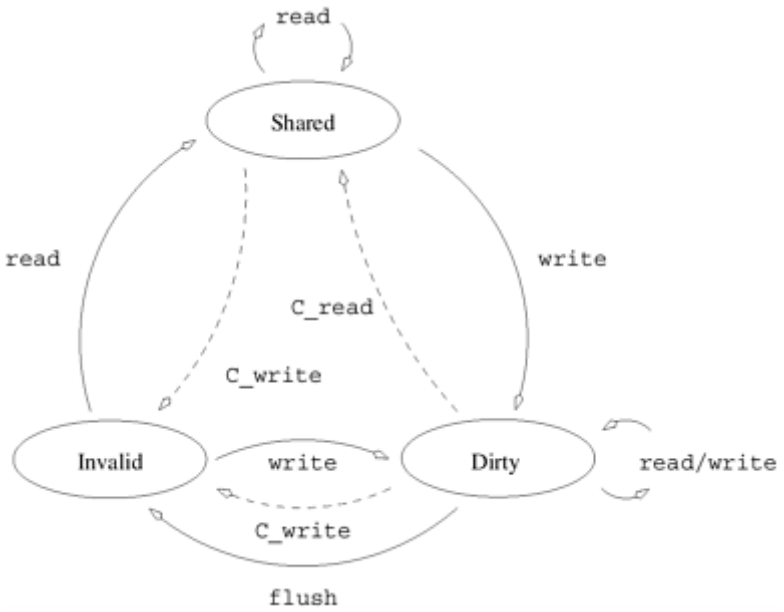


图2.22 简单三态一致性协议的状态图

- **例2.14 使用简单的三态协议保持一致性**

  以处理器 P_0 和 P_1 正在执行的两个程序段为例，如图 2.23 所示。系统由处理器 P_0 和 P_1 的本地存储器（或高速缓存）以及全局存储器组成。本例中假设的三态协议与图 2.22 所示的状态图相对应。该系统中的缓存线可以是Shared、Invalid或Dirty的。假定每个数据项（变量）位于不同的高速缓存行上。起初，两个变量 x 和 y 被标记为脏变量，全局存储器中只有这两个变量的副本。图 2.23 展示了每条指令执行时的状态转换和变量副本的值。

Time ↓	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D y = 12, D
	read x		x = 5, S		x = 5, S
		read y		y = 12, S	y = 12, S
	x = x + 1		x = 6, D		x = 5, I
		y = y + 1		y = 13, D	y = 12, I
	read y		y = 13, S	y = 13, S	y = 13, S
		read x	x = 6, S	x = 6, S	x = 6, S
	x = x + y		x = 19, D	x = 6, I	x = 6, I
		y = x + y	y = 13, I	y = 19, D	y = 13, I
	x = x + 1		x = 20, D		x = 6, I
		y = y + 1		y = 20, D	y = 13, I

图2.23 使用第2.4.6节中讨论的简单三态一致性协议执行并行程序的示例

一致性协议的执行可以使用各种硬件机制 - Snoopy系统、基于目录的系统或它们的组合。

Snoopy缓存系统

Snoopy缓存通常与基于广播互连网络（如总线或环网）的多处理器系统相关联。在这种系统中，所有处理器都会Snoopy（监控）总线上的事务。这样，处理器就可以对其缓存块进行状态转换。图 2.24 展示了一个典型的基于窥探总线的系统。每个处理器的高速缓存都有一组与之相关的标签位，用于确定高速缓存块的状态。这些标签根据与一致性协议相关的状态图进行更新。例如，当窥探硬件检测到对其拥有脏拷贝的缓存块发出了读取请求时，就会断言对总线的控制并输出数据。同样，当窥探硬件检测到对其拥有副本的缓存块进行了写操作时，就会使缓存块失效。其他状态转换也是以这种方式在本地完成的。

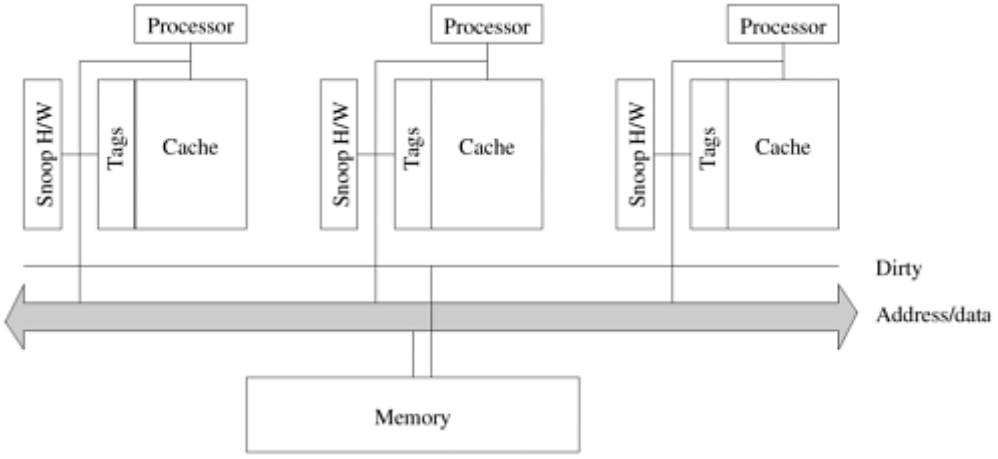


图2.24 一个简单的基于Snoopy总线的缓存一致性系统

Snoopy缓存的性能 Snoopy协议已被广泛研究并用于商业系统。这主要是因为它们简单易用，而且现有的基于总线的系统可以升级以适应Snoopy协议。Snoopy系统的性能提升源于这样一个事实，即如果不同的处理器对不同的数据项进行操作，这些数据项可以被缓存起来。一旦这些数据项被标记为 "Dirty"，所有后续操作都可以在本地缓存中执行，而不会产生外部流量。同样，如果一个数据项被多个处理器读取，它就会过渡到缓存中的共享状态，所有后续读取操作都会变成本地操作。在这两种情况下，一致性协议都不会增加任何开销。另一方面，如果多个处理器读取和更新同一个数据项，就会产生跨处理器的一致性函数。由于共享总线的带宽有限，在单位时间内只能执行一定数量的一致性操作。这就成为基于Snoopy总线的系统的一个基本瓶颈。

Snoopy协议与基于广播网络（如总线）的多计算机密切相关。这是因为所有处理器都必须窥探所有信息。显然，将一个处理器的所有内存操作广播给所有处理器并不是一个可扩展的解决方案。解决这个问题的一个显而易见的办法是，只向必须参与操作的处理器（即拥有相关数据副本的处理器）传播一致性操作。这种解决方案要求我们跟踪哪些处理器拥有各种数据项的副本，以及这些数据项的相关状态信息。这些信息存储在一个目录中，基于这些信息的一致性机制被称为基于目录的系统。

基于目录的系统

考虑一个简单的系统，在它的全局内存中增加一个目录，该目录维护着代表缓存块和缓存块所在处理器的位图（图 2.25）。这些位图条目有时被称为 **存在位（Presence Bits）**。如前所述，我们假定采用三状态协议，即 **Invalid状态**、**Dirty状态** 和 **Shared状态**。基于目录方案的性能关键在于一个简单的观察结果，即只有持有特定区块（或正在读取该区块）的处理器才会参与一致性操作引起的状态转换。需要注意的是，处理器读取、写入或刷新（从缓存中删除一行）可能会触发其他状态转换，但这些转换可在本地处理，操作反映在目录中的存在位和状态中。

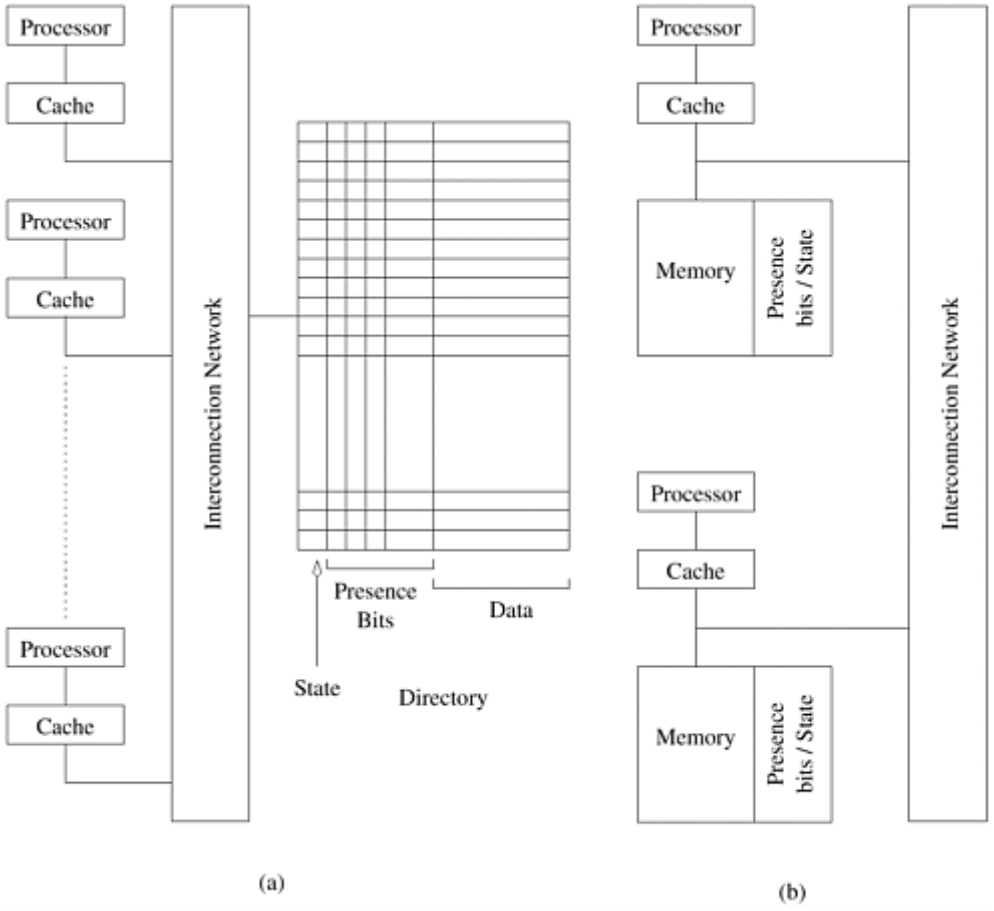


图2.25 典型的基于目录的系统架构：(a)集中式目录；(b)分布式目录。

重温图 2.21 中的代码段，当处理器 P_0 和 P_1 访问与变量 x 对应的块时，块的状态变为共享，存在位更新以表示处理器 P_0 和 P_1 共享该块。当 P_0 对变量执行存储操作时，目录中的状态变为 Dirty，P_1 的存在位被重置。处理器 P_0 对该变量执行的所有后续操作均可在本地进行。如果其他处理器读取该值，目录会注意到 Dirty 标记，并使用存在位将请求导向相应的处理器。处理器 P_0 会更新内存中的数据块，并将其发送给提出请求的处理器。存在位会被修改以反映这一情况，状态也会转换为共享状态。

基于目录的方案的性能 与Snoopy协议一样，如果不同的处理器对不同的数据块进行操作，这些数据块就会在各自的缓存中变脏，第一次操作之后的所有操作都可以在本地执行。此外，如果多个处理器读取（但不更新）一个数据块，该数据块会以共享状态复制到缓存中，随后的读取操作不会触发任何一致性开销。

当多个处理器试图更新同一个数据项时，就会启动一致性操作。在这种情况下，除了必要的数据移动外，一致性操作还会以传播状态更新（失效或更新）和从目录中生成状态信息的形式增加开销。前者以通信开销的形式出现，后者则增加了争用。通信开销是需要状态更新的处理器数量和传播状态信息的算法的函数。争用开销在本质上更为根本。由于目录位于内存中，而内存系统在单位时间内只能处理一定数量的读/写操作，因此状态更新的数量最终会受到目录的限制。如果并行程序需要大量一致性操作（大量读/写共享数据块），目录将最终限制其并行性能。

最后，从成本的角度来看，随着处理器数量的增加，存储目录所需的内存容量本身就可能成为瓶颈。回想一下，目录大小的增长为 $O(mp)$，其中 m 是内存块的数量，p 是处理器的数量。一种解决方案是增大内存块（从而在给定内存大小的情况下减少 m）。然而，这会增加其他开销，如错误共享，即两个处理器更新程序中不同的数据项，但数据项恰好位于同一内存块中。这种现象将在第 7 章中详细讨论。

由于目录是争论的中心点，自然要把保持一致性的任务分解给多个处理器。其基本原理是，假设内存块在处理器之间进行物理（或逻辑）分区，让每个处理器维护自己内存块的一致性。这就是分布式目录系统的原理。

分布式目录方案 在可扩展架构中，内存在物理上分布在各个处理器上。内存块的相应存在位也是分布式的。每个处理器负责维护自己内存块的一致性。这种系统的架构如图 2.25(b)所示。由于每个内存块都有一个所有者（通常可以通过块地址计算出来），所有处理器都隐含地知道其目录位置。当处理器首次尝试读取一个内存块时，它会请求该内存块的所有者。所有者会根据本地可用的存在和状态信息适当地引导这一请求。同样，当处理器向内存块中写入内容时，它会向所有者发出无效请求，所有者再将无效请求转发给所有拥有该内存块缓存副本的处理器。通过这种方式，目录是分散的，与中央目录相关的争用也得到了缓解。需要注意的是，与状态更新信息相关的通信开销并没有减少。

分布式目录方案的性能 显然，分布式目录允许同时进行 $O(p)$次一致性操作，前提是底层网络能够承受相关的状态更新信息。从这一点来看，分布式目录的可扩展性本质上要高于窥探式系统或集中式目录系统。网络的延迟和带宽成为此类系统的基本性能瓶颈。

> **Note**
>
> **Snoopy缓存系统的瓶颈**：由于共享总线的带宽有限，在单位时间内只能执行一定数量的一致性操作。
>
> **集中式目录系统的瓶颈**：由于目录位于内存中，而内存系统在单位时间内只能处理一定数量的读/写操作，因此状态更新的数量最终会受到目录的限制，如果并行程序需要大量一致性操作（大量读/写共享数据块），目录数量将最终限制其并行性能。
>
> **分布式目录系统的瓶颈**：分布式目录的可扩展性本质上要高于窥探式系统或集中式目录系统，网络的延迟和带宽成为此类系统的基本性能瓶颈。

2.5 并行计算中的通信开销

并行程序执行过程中的主要开销之一来自处理元件之间的信息通信。通信成本取决于多种特性，包括编程模型语义、网络拓扑结构、数据处理和路由选择，以及相关的软件协议。这些问题是我们在此讨论的重点。

2.5.1 并行计算中的消息传递开销

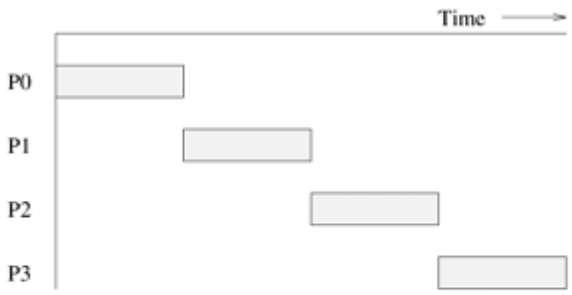
网络中两个节点之间传输信息所需的时间是准备传输信息所需的时间与信息穿越网络到达目的地所需的时间之和。决定通信延迟的主要参数如下：

1. **启动时间（Startup Time）** t_s：启动时间是发送节点和接收节点处理信息所需的时间。这包括准备报文的时间（添加报文头、预告片和纠错信息）、执行路由算法的时间以及在本地节点和路由器之间建立接口的时间，单次信息传输只产生一次延迟。
2. **每跳时间（Per-hop Time）** t_h：信息离开一个节点后，到达路径上的下一个节点所需的时间是有限的。信息头在网络中两个直接连接的节点之间传输所需的时间称为每跳时间，它也称为 **节点延迟（Node Latency）**。每跳时间与路由交换机内确定信息应转发到哪个输出缓冲区或通道的延迟直接相关。
3. **每字传输时间（Per-word Transfer Time）** t_w：如果信道带宽为每秒 r 个字，那么每个字在链路上的传输时间为 $t_w = 1/r$。这段时间称为每字传输时间。该时间包括网络和缓冲开销。

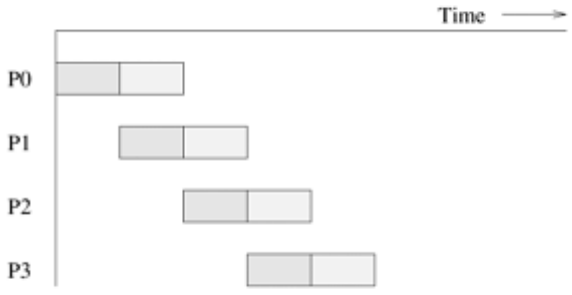
现在我们讨论并行计算机中使用的两种路由技术 - 存储转发路由和穿透路由。

存储转发路由

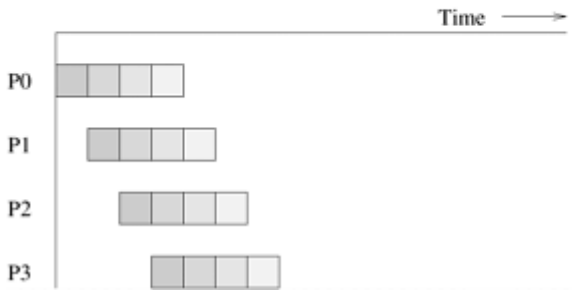
在 **存储转发路由（Store-and-Forward Routing）** 中，当报文穿越有多个链路的路径时，路径上的每个中间节点都会在接收并存储整个报文后，将报文转发给下一个节点。图 2.26(a) 显示了信息通过存储转发网络进行通信的过程。



(a) A single message sent over a store-and-forward network



(b) The same message broken into two parts and sent over the network.



(c) The same message broken into four parts and sent over the network.

图2.26 (a)通过存储转发通信网络将信息从节点P0传递到 P3；(b)和(c)将这一概念扩展到直通路由。阴影区域表示信息在传输过程中的时间（与信息传输相关的启动时间假定为零）。

假设一个大小为 m 的信息正在通过这样一个网络传输。假设它要穿越 l 个链路。在每个链路上，信息头的传输成本为 t_h，其余部分的传输成本为 $t_w m$。由于有 l 个这样的链路，所以总时间为 $(t_h + t_w m)l$ 。因此，对于存储转发路由，大小为 m 个字的信息穿越 l 个通信链路的总通信成本为：

$$t_{comm} = t_s + (mt_w + t_h)l \tag{1}$$

在当前的并行计算机中，每跳时间 t_h 非常小。对于大多数并行算法来说，即使 m 的值很小，t_h 也小于 $t_w m$，因此可以忽略不计。对于使用存储转发路由的并行平台，上式所给出的时间可简化为：

$$t_{comm} = t_s + mlt_w \tag{2}$$

数据包路由

存储转发路由对通信资源的利用率很低。只有在收到整个信息后，信息才会从一个节点发送到下一个节点（图 2.26(a)）。考虑一下图 2.26(b)所示的情况，在这种情况下，原始信息在发送前被分成大小相等的两部分。在这种情况下，中间节点只需等待原始信息的一半到达即可将其发送出去。从图 2.26(b)中可以明显看出，通信资源的利用率提高了，通信时间缩短了。图 2.26(c) 更进一步，将信息分成四个部分。除了更好地利用通信资源外，这一原则还具有其他优点 - 降低数据包丢失（错误）造成的开销、数据包可能走不同的路径以及更好的纠错能力。由于这些原因，这种技术成为互联网等长途通信网络的基础，因为在这些网络中，错误率、跳数和网络状态的变化可能会更高。当然，这里的开销是每个数据包必须携带路由、纠错和排序信息。

考虑通过网络传输一个 m 字的信息。网络接口编程和计算路由信息等所需的时间与报文长度无关。这些时间汇总为信息传输的启动时间 t_s。我们假设路由表在信息传输过程中是静态的（即所有数据包都经过相同的路径）。虽然这种假设并非在所有情况下都有效，但它有助于激发信息传输的成本模型。信息被分解成数据包，数据包与错误、路由和排序字段组装在一起。现在，数据包的大小为 $r + s$，其中 r 是原始信息，s 是数据包中携带的附加信息。将信息打包所需的时间与信息的长度成正比。我们用 mt_{w1} 表示这个时间。如果网络能够每 t_{w2} 秒传送一个字，每跳的延迟为 t_h，第一个数据包经过 l 跳，那么这个数据包到达目的地需要 $t_h l + t_{w2}(r + s)$ 时间。此后，目的节点每隔 $t_{w2}(r + s)$秒会收到一个额外的数据包。由于有 $m/r - 1$个附加数据包，因此总通信时间为：

$$\begin{aligned} t_{comm} &= t_s + t_{w1}m + t_h l + t_{w2}(r + s) + (\frac{m}{r} - 1)t_{w2}(r + s) && (3)\\ &= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\frac{s}{r}m && (4)\\ &= t_s + t_h l + t_w m && (5) \end{aligned}$$

其中：

$$t_w = t_{w1} + t_{w2}(1 + \frac{s}{r}) \tag{6}$$

数据包路由适合于具有高度动态状态和较高错误率的网络，如局域网和广域网。这是因为单个数据包可能会选择不同的路由，而且可以对丢失的数据包进行定位重传。

直通路由

在并行计算机的互连网络中，可以对信息传输施加额外的限制，以进一步减少与数据包交换相关的开销。通过强制所有数据包走相同的路径，我们可以消除随每个数据包传输路由信息的开销。通过强制按顺序传送，可以消除排序信息。通过在信息层而不是数据包层关联错误信息，可以减少与错误检测和纠正相关的开销。最后，由于并行机互连网络中的错误率极低，因此可以使用精简的错误检测机制来代替昂贵的纠错方案。

由这些优化产生的路由方案称为 **直通路由（Cut-through Routing）**。在直通路由中，信息被分解成固定大小的单位，称为 **流量控制位（Flow Control Digits, or Flits）**。由于比特不包含数据包的开销，因此比特可以比数据包小得多。首先从源节点向目的节点发送跟踪器，以建立连接。一旦建立了连接，就会一个接一个地发送比特。所有传输片都以对接方式沿相同路径传输。中间节点在转发信息之前不会等待整个信息到达。一旦中间节点收到一个比特，该比特就会被转发到下一个节点。与存储转发路由不同，每个中间

节点不再需要缓冲空间来存储整个信息。因此，直通式路由占用中间节点的内存和内存带宽更少，速度更快。

假设有一条信息正在穿越这样的网络。如果信息穿越 l 个链路，t_h 是每跳时间，那么信息头到达目的地需要 lt_h 时间。如果信息长 m 个字，那么整个信息在信息头到达后的 $t_w m$ 时间内到达。因此，直通式路由的总通信时间为：

$$t_{comm} = t_s + lt_h + t_w m \tag{7}$$

由于与跳数和字数相对应的项是加法，而不是前者的乘法，因此与存储转发路由相比，这个时间有所改进。需要注意的是，如果通信是在最近的邻居之间进行（即 $l = 1$），或者信息量很小，那么存储转发路由方案和直通路由方案的通信时间是相似的。

目前大多数并行计算机和许多局域网都支持直通路由。比特的大小由各种网络参数决定。控制电路必须以单位速率运行。因此，如果我们选择非常小的单位大小，在给定链路带宽的情况下，所需的单位速率就会变得很大。这给路由器的设计带来了相当大的挑战，因为它要求控制电路以非常高的速度运行。另一方面，随着比特大小变大，内部缓冲区的大小也会增加，信息传输的延迟也会增加。这两种情况都不可取。在最近的直通式互连网络中，单位大小从 4 位到 32 字节不等。在许多主要依赖短信息（如高速缓存行）的并行编程模式中，信息的延迟至关重要。在这些情况下，长报文在链路中穿行时会耽误短报文的时间，这是不合理的。路由器采用多线穿通路由技术来解决这种情况。在多线穿通路由选择中，单个物理通道被分成多个虚拟通道。

报文常量 t_s、t_w 和 t_h 由硬件特性、软件层和报文语义决定。与消息传递等范例相关的消息传递语义最好使用长度可变的消息，而其他范例则使用固定长度的短消息。对于前者，有效带宽可能至关重要，而对于后者，减少延迟则更为重要。这些范例的信息传递层经过调整，以反映这些要求。

在穿越网络时，如果报文需要使用当前正在使用的链接，那么报文就会被阻塞。这可能会导致死锁。图 2.27 展示了直通路由网络中的死锁。报文 *0*、*1*、*2* 和 *3* 的目的地分别是 *A*、*B*、*C* 和 *D*。报文 *0* 的飞信占用了链路 *CB*（及相关缓冲区）。然而，由于来自报文 *3* 的信道占用了链路 *BA*，因此来自报文 *0* 的信道被阻塞。同样，由于正在使用链路 *AD*，来自报文 *3* 的飞信也被阻塞。我们可以看到，任何报文都无法在网络中传输，网络陷入死锁。在直通网络中，可以通过使用适当的路由技术和报文缓冲区来避免死锁。第 2.6 节将对此进行讨论。

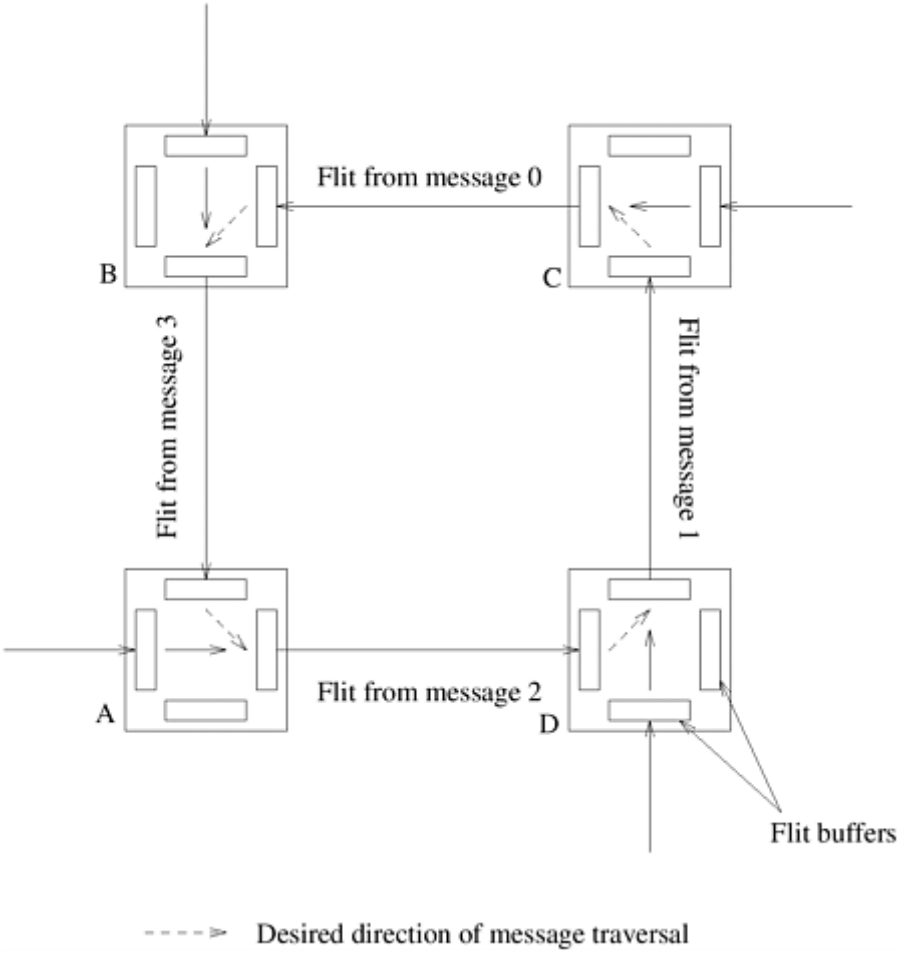


图2.27 直通路由网络中的死锁示例

消息传递的简化开销模型

正如我们在第 2.5.1 节中看到的，使用直通式路由在相距 l 跳的两个节点之间传输信息的开销为：

$$t_{comm} = t_s + lt_h + t_w m \tag{8}$$

这个等式意味着，为了优化信息传输的成本，我们需要：

1. **批量通信（Communicate in Bulk）** 也就是说，我们不希望发送小信息并为每条信息支付启动成本 t_s，而是希望将小信息汇聚成一条大信息，并在一条更大的信息中分摊启动延迟。这是因为在集群和消息传递机器等典型平台上，t_s 的值远大于 t_h 或 t_w 的值。
2. **尽量减少数据量（Minimize the volume of data）** 为了尽量减少每个字传输时间 t_w 的开销，最好尽可能减少通信数据量。
3. **最小化数据传输距离（Minimize distance of data transfer）** 尽量减少信息必须经过的跳数 l。

虽然前两个目标相对容易实现，但最大限度减少通信节点距离的任务却很困难，而且在很多情况下会给算法设计者带来不必要的负担。这是并行平台和范式的以下特点造成的直接后果：

- 在许多消息传递库（如 MPI）中，程序员几乎无法控制进程与物理处理器的映射。在这种模式下，虽然任务可能有定义明确的拓扑结构，并且只能在任务拓扑结构中的邻居之间进行通信，但将进程映射到节点可能会破坏这种结构。
- 许多架构依赖于随机（两步）路由，即首先将信息从源节点发送到一个随机节点，然后再从这个中间节点发送到目的地。这可以缓解网络上的热点和竞争。在随机路由网络中，跳数最小化不会带来任何好处。
- 每跳时间（t_h）通常由小型报文的启动延迟（t_s）或大型报文的每字分量（$t_w m$）决定。由于大多数网络中的最大跳数（l）相对较小，因此可以忽略每跳时间，而几乎不会损失准确性。

所有这些都指向一个更简单的成本模型，即在网络的两个节点之间传输信息的成本由以下公式给出：

$$t_{comm} = t_s + t_w m \tag{9}$$

这一表达式对独立于架构的算法设计以及运行时预测的准确性具有重要意义。由于这种成本模型意味着任何一对节点之间的通信时间都是相同的，因此它对应的是一个完全连接的网络。我们无需为每种特定架构（如网状、超立方或树状）设计算法，而是可以在设计算法时考虑到这种成本模型，并将其移植到任何目标并行计算机上。

这就提出了一个重要问题，即当算法从我们的简化模型（假设网络完全连接）移植到实际的机器架构时，预测的准确性会有损失。如果我们最初的假设成立，即 t_h 项通常由 t_s 或 t_w 项主导，那么准确性的损失应该是最小的。

不过，需要注意的是，我们的基本成本模型仅适用于不拥堵的网络。不同的架构会有不同的拥塞阈值，例如，线性阵列的拥塞阈值比超立方低得多。此外，不同的通信模式会对特定网络造成不同程度的拥塞。因此，我们的简化成本模型只有在基本通信模式不造成网络拥塞的情况下才有效。

- **例2.15 阻塞对通信开销的影响**

  考虑一个 $\sqrt{p} \times \sqrt{p}$ 的网状结构，其中每个节点只与其最近的邻居通信。由于网络中没有链路用于一次以上的通信，因此这一操作的时间为 $t_s + t_w m$，其中 m 为通信字数。这个时间与我们的简化模型一致。考虑另一种情况，即每个节点与随机选择的节点进行通信。这种随机性意味着有 $p/2$ 次通信（或 $p/4$ 次双向通信）发生在机器的任何等分区中（因为被通信的节点可能以相等的概率出现在任何一半中）。根据我们对分段宽度的讨论，我们知道二维网格的分段宽度为 $\sqrt{p}$。根据这两点，我们可以推断出，假设有双向通信通道，一些链接现在至少要携带 $\frac{p/4}{\sqrt{p}} = \sqrt{p}/4$ 的信息。这些信息必须在链路上序列化。如果每条信息的大小为 m，那么这一操作所需的时间至少为 $t_s + t_w m \times \sqrt{p}/4$。这个时间与我们的简化模型不符。

上面的例子说明，对于给定的架构，有些通信模式可能不会造成拥塞，而有些则可能会造成拥塞。这就使得通信成本建模任务不仅取决于架构，还取决于通信模式。为此，我们引入了 **有效带宽（Effective Bandwidth）** 的概念。对于不会造成网络拥塞的通信模式，有效带宽与链路带宽相同。然而，对于网络拥塞的通信操作，有效带宽是链路带宽按最拥塞链路的拥塞程度缩减后的值。这通常很难估算，因为它是

进程到节点映射、路由算法和通信时间表的函数。因此，我们使用信息通信时间的下限。相关的链路带宽按系数 p/b 缩减，其中 b 是网络的对分宽度。

在本文的其余部分，我们将使用简化的通信模型来处理消息传递和有效的每字时间 t_w，因为它允许我们以一种与体系结构无关的方式来设计算法。我们还将具体说明算法中的通信操作何时会造成网络拥塞，以及如何将其影响计入并行运行时间。书中的通信时间适用于k-d网格的一般类别。虽然这些时间也可以在其他架构上实现，但这是底层架构的函数。

2.5.2 并行计算中的内存共享开销

将通信成本与并行程序联系起来的主要目的是为程序设定一个优劣值，以指导程序开发。与消息传递或非缓存相干架构相比，缓存一致的共享内存的这一任务要困难得多。原因如下：

- 内存布局通常由系统决定。程序员对特定数据项位置的控制微乎其微，只能通过排列数据结构来优化访问。这一点在分布式内存共享地址空间架构中尤为重要，因为很难识别本地和远程访问。如果本地数据项和远程数据项的访问时间相差很大，那么通信成本就会因数据布局的不同而大相径庭。
- 有限的高速缓存大小可能会导致高速缓存崩溃。考虑这样一种情况：节点需要全部数据的某一部分来计算其结果。如果这部分数据小于本地可用的缓存，则可以在首次访问时获取数据并进行计算。但是，如果这部分数据超过了可用缓存的容量，那么数据的某些部分可能会被覆盖，从而被多次访问。随着问题规模的增大，这种开销会导致程序性能急剧下降。为了解决这个问题，程序员必须改变执行计划，以尽量减小工作集的大小。虽然这个问题在串行和多处理器平台上都很常见，但在多处理器情况下，由于每次失误都可能涉及一致性操作和处理器间通信，因此代价要高得多。
- 与失效和更新操作相关的开销很难量化。数据项被处理器取入高速缓存后，可能会在另一个处理器上进行各种操作。例如，在失效协议中，缓存行可能会被远程处理器的写入操作失效。在这种情况下，数据项的下一次读取操作必须再次支付远程访问延迟成本。同样，与更新协议相关的开销可能会因数据项的副本数量不同而有很大差异。数据项的并发拷贝数和指令执行时间表通常不在程序员的控制范围内。
- 空间定位很难建模。由于高速缓存行一般比一个字长（从 4 个字到 128 个字），即使是第一次访问，不同的字也可能有不同的访问延迟。如果高速缓存行尚未被覆盖，那么访问先前获取的字的邻居可能会非常快。同样，程序员对此的控制能力微乎其微，只能通过改变数据结构来最大限度地提高数据引用的空间位置性。
- 预取可以减少与数据访问相关的开销。编译器可以提前加载，如果有足够的资源，与这些加载相关的开销可能会被完全掩盖。由于这是编译器、底层程序和可用资源（寄存器/缓存）的函数，因此很难准确建模。
- 错误共享通常是许多程序中的重要开销。不同处理器（在不同处理器上执行的线程）使用的两个字可能位于同一高速缓存行上。这可能会导致一致性操作和通信开销，即使没有任何数据可能被共享。程序员必须充分填充不同处理器使用的数据结构，以尽量减少错误共享。
- 共享访问中的争用通常是共享地址空间机器的主要开销。遗憾的是，争用是执行进度的函数，因此很难准确建模（与调度算法无关）。虽然可以通过计算共享访问的次数来获得松散的渐近估计值，但这种约束往往意义不大。

共享地址空间机器的任何成本模型都必须考虑所有这些开销。将这些费用纳入一个单一的成本模型，会导致该模型过于繁琐，难以设计程序，而且过于针对单个机器，无法普遍适用。

作为一阶模型，我们不难看出，访问一个远程字会导致一个高速缓存行被取入本地高速缓存。与此相关的时间包括一致性开销、网络开销和内存开销。一致性开销和网络开销是底层互连的函数（因为一致性操作必须有可能传播到远程处理器，而且必须提取数据项）。在不知道特定访问与哪些一致性操作相关以及字从哪里来的情况下，我们将访问共享数据的缓存行与恒定开销联系起来。为了与消息传递模型保持一致，我们将这一开销称为 t_s。由于在现代处理器架构中实施了各种延迟隐藏协议（如预取），我们假定，即使 m 大于缓存行大小，在开始访问共享数据的 m 字连续块时，也会产生一个不变的成本 t_s。我们进一步假设，访问共享数据的成本要高于访问本地数据（例如，在 NUMA 机器上，本地数据可能存在于本地内存模块中，而 p 个处理器共享的数据需要从至少 $p-1$ 个处理器的非本地模块中获取）。因此，我们为共享数据分配的每字访问成本为 t_w。

根据上述讨论，我们可以使用相同的表达式 $t_s + t_w m$ 来计算在共享内存和消息传递模式下一对处理器之间共享单块 m 字的成本，不同的是，在共享内存机器上，常数 t_s 相对于 t_w 的值可能比在分布式内存机器上小得多（在 UMA 机器上，t_w 可能接近于零）。需要注意的是，$t_s + t_w m$ 的代价是假定只读访问时没有竞争。如果多个进程访问同一数据，那么成本将乘以进程数，就像在消息传递中，拥有数据的进程需要向每个接收进程发送一条消息一样。如果是读写访问，那么除写入的进程外，其他进程的后续访问也会产生成本。这与消息传递模型再次等价。如果一个进程修改了它接收到的消息内容，那么它就必须把它发回给随后需要访问刷新数据的进程。虽然这个模型在共享地址空间机器的背景下显得过于简化，但我们注意到，这个模型很好地估算了在一对处理器之间共享由 m 个字组成的数组的成本。

上述简化模型主要考虑了远程数据访问，但没有模拟其他各种开销。对共享数据访问的争夺必须通过计算共同调度任务之间对共享数据的访问次数来明确计算。该模型没有明确包括许多其他开销。由于不同机器的高速缓存大小不同，因此很难确定工作集大小超过高速缓存大小的时间点，从而导致独立于体系结构的高速缓存崩溃。因此，本成本模型忽略了有限缓存所产生的影响。最大化空间位置性（高速缓存行效应）没有明确包含在成本中。假共享是指令调度和数据布局的函数。成本模型假定共享数据结构经过适当填充，因此不包括错误共享成本。最后，该成本模型不考虑通信和计算的重叠。人们还提出了其他模型来模拟重叠通信。不过，为这些模型设计简单的算法也很麻烦。与此相关的单个处理器上的多个并发计算（线程）问题也没有在表达式中建模。相反，每个处理器都被假定为执行单个并发计算单元。

2.6 互联网络的路由机制

将信息路由到目的地的高效算法对并行计算机的性能至关重要。**路由机制（Routing Mechanism）** 决定了信息通过网络从源节点到达目的节点的路径。它将信息的源节点和目的节点作为输入。它还可以使用网络状态信息。它返回一条或多条从源节点到目的节点的网络路径。

路由机制可分为 **最小（Minimal）** 路由机制和 **非最小（Non-Minimal）** 路由机制。最小路由机制总是选择信源和目的地之间的一条最短路径。在最小路由方案中，每条链路都能使信息更接近目的地，但该方案可能会导致部分网络拥塞。与此相反，非最小路由方案可能会沿着较长的路径路由信息，以避免网络拥塞。

路由机制还可根据其使用网络状态信息的方式进行分类。**确定性路由（Deterministic Routing）** 方案根据信息的来源和目的地为信息确定唯一的路径。它不使用任何有关网络状态的信息。确定性方案可能会导致网络通信资源的不均衡使用。与此相反，**自适应路由（Adaptive Routing）** 选择方案使用有关网络当前状态的信息来确定信息的路径。自适应路由可检测网络拥塞情况，并绕过拥塞情况路由信息。

一种常用的确定性最小路由技术称为维度有序路由。**维度有序路由（Dimension-Ordered Routing）** 根据由通道维度决定的编号方案，为信息的穿越分配连续的通道。二维网格的维度有序路由技术称为 **XY 路由（XY-Routing）**，超立方体的维度有序路由技术称为 **E 立方体路由（E-Cube Routing）**。

考虑一个没有环绕连接的二维网格。在 XY 路由方案中，信息首先沿 X 维发送，直到到达目的节点的列，然后沿 Y 维发送，直到到达目的地。让 $P_{Sy,Sx}$ 表示源节点的位置，$P_{Dy,Dx}$ 表示目的节点的位置。任何最小路由方案都应返回一条长度为 $|Sx - Dx| + |Sy - Dy|$ 的路径。假设 $Dx \geq Sx$，$Dy \geq Sy$。在 XY 路由方案中，信息沿 X 维通过中间节点 $P_{Sy,Sx+1}$、$P_{Sy,Sx+2}$、……、$P_{Sy,Dx}$，然后沿 Y 维通过节点 $P_{Sy+1,Dx}$、$P_{Sy+2,Dx}$、……、$P_{Dy,Dx}$ 到达目的地，这条路径的长度确实是$|Sx - Dx| + |Sy - Dy|$。

超立方连接网络的 E 立方路由工作原理类似。考虑一个由 p 个节点组成的 d 维超立方体。让 P_s 和 P_d 分别代表源节点和目的节点的标签。我们从第 2.4.3 节得知，这些标签的二进制表示长度为 d 比特。此外，这些节点之间的最小距离由 $P_s \oplus P_d$ 中 1 的个数给出（其中 $\oplus$ 表示位运算中的异或运算）。在 E 立方算法中，节点 P_s 计算 $P_s \oplus P_d$ 并沿维度 k 发送信息，其中 k 是 $P_s \oplus P_d$ 中最小有效非零比特的位置。在每个中间步骤中，接收到信息的节点 P_i 会计算 $P_i \oplus P_d$，并沿最小有效非零位对应的维度转发信息。这一过程一直持续到信息到达目的地。例 2.16 展示了三维超立方网络中的 E 立方路由。

- **例2.16 超立方网络中的 E 立方路由选择**

  考虑图 2.28 所示的三维超立方体。让 $P_s = 010$ 和 $P_d = 111$ 分别代表一条信息的源节点和目的节点。节点 P_s 计算出 $010 \oplus 111 = 101$。第一步，P_s 沿着与最小有效位对应的维数将信息转发给节点 011。节点 011 沿着与最有效位对应的维度发送信息（$011 \oplus 111 = 100$）。信息到达节点 111，即信息的目的地。

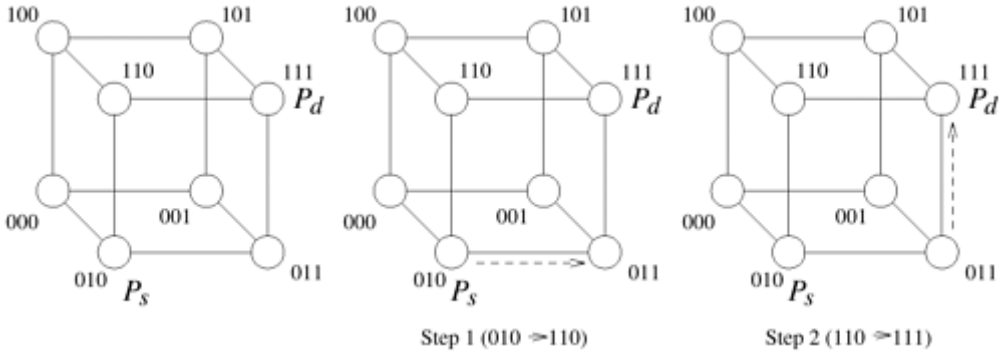


图2.28 在三维超立方体中使用 E 立方体路由将信息从010路由到111

在本书的其余部分，我们假定在分析并行算法时采用确定性的最小报文路由。

2.7 进程到处理器的映射技术及其影响

正如我们在第 2.5.1 节中所讨论的，程序员通常无法控制逻辑进程如何映射到网络中的物理节点。因此，即使通信模式本身并不拥塞，也可能造成网络拥塞。下面我们举例说明：

- **例2.17 处理器映射的影响**

  请看图 2.29 所示的情况。底层架构是一个 16 节点的网状结构，节点标号从 1 到 16（图 2.29(a)），算法以 16 个进程（标号从 "*a*"到 "*p*"）的形式实现（图 2.29(b)）。该算法是为在网格上执行而调整的，因此不会出现拥塞的通信操作。我们现在来看看流程与节点的两种映射，如图 2.29(c)和图 2.29(d)所示。图 2.29(c)是一种直观的映射，底层架构中的单个链路只携带与进程间单个通信通道相对应的数据。另一方面，图 2.29(d)对应的是进程随机映射到处理节点的情况。在这种情况下，很容易看到机器中的每条链路都在进程间携带多达六条数据通道。如果进程间通信通道所需的数据传输速率较高，这可能会导致通信时间大大增加。

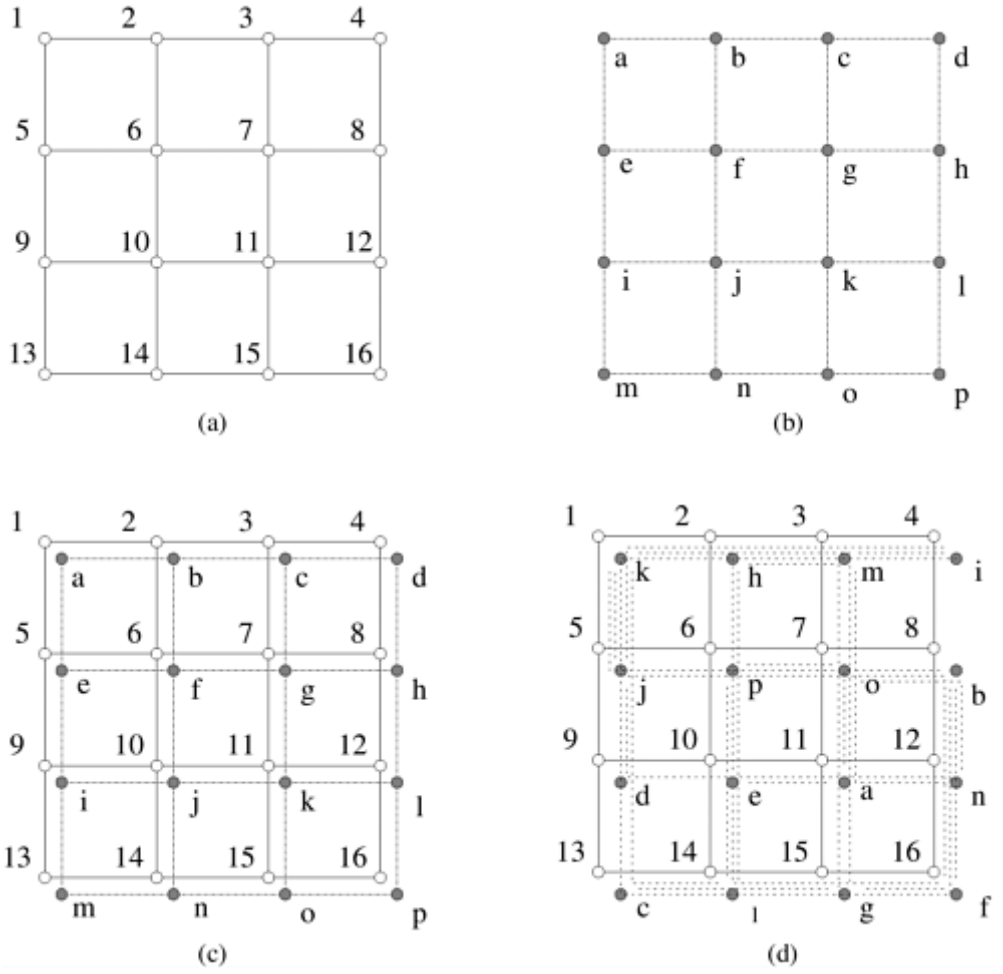


图2.29 进程映射对性能的影响：(a)底层架构；(b)进程及其之间的关系；(c)进程与节点的直接映射；(d)进程与节点的随机映射

从上面的例子可以看出，虽然算法可以由不拥塞的通信操作组成，但将进程映射到节点实际上可能会导致网络拥塞，造成性能下降。

2.7.1 图的映射技术

虽然程序员通常无法控制进程到处理器映射，但了解这种映射的算法非常重要。这是因为这些映射可用于确定算法性能的下降。给定两个图 $G(V,E)$和 $G'(V',E')$，将图 G 映射到图 G'中，可以将集合 V 中的每个顶点映射到集合 V'中的一个顶点（或顶点集合），将集合 E 中的每条边映射到 E'中的一条边（或边集合）。在将图 $G(V,E)$映射到 $G'(V',E')$ 时，有三个参数非常重要。首先，E 中可能有不止一条边被映射到 E' 中的一条边上。映射到 E' 中任何一条边上的最大边数称为映射的 **拥挤度（Congestion）**。在例 2.17中，图 2.29(c) 中的映射拥挤度为 1，图 2.29(d) 中的映射拥挤度为 6。其次，E 中的一条边可以映射到 E' 中的多条连续边上。这一点很重要，因为相应通信链路上的流量必须穿越多个链路，从而可能造成网络拥塞。E' 中任何一条边被映射到的最大链路数称为映射的 **扩张（Dilation）**。第三，集合 V 和 V' 可能包含不同数量的顶点。在这种情况下，V 中的一个节点对应 V' 中的多个节点。集合 V' 中的节点数与集合 V 中的节点数之比称为映射的 **扩展（Expansion）**。在进程到处理器映射中，我们希望映射的扩展与虚拟和物理处理器的比例相同。

在本节中，我们将讨论一些常见图的嵌入，如二维网格（第 8 章中的矩阵运算）、超立方体（第 9 章和第 13 章中分别介绍了排序和 FFT 算法）和树（第 4 章中的广播和屏障）。我们的讨论范围仅限于集合 V 和 V' 包含节点数相等（即扩展为 1）的情况。

线性阵列嵌入的超立方体

通过将线性数组的节点 i 映射到超立方体的节点 $G(i,\ d)$ 上，可以将由 2^d 个节点（标记为 0 至 2^d-1）组成的线性数组（或环）嵌入到 d 维超立方体中。函数 $G(i,x)$ 的定义如下：

$$G(0,1)=0 \tag{1}$$

$$G(1,1)=1 \tag{2}$$

$$G(i,x+1)=\begin{cases} G(i,x), & i<2^x \\ 2^x+G(2^{x+1}-1-i,x), & i\geq 2^x \end{cases} \tag{3}$$

函数 G 称为 **二进制反射格雷码（Binary Reflected Gray Code，RGC）**。其中 $G(i,d)$ 表示 d 位格雷码序列中的第 i 个条目。$d+1$ 比特的灰色编码是从 d 比特的灰色编码表中导出的，方法是对表进行反射，并在反射条目前加上 1，在原始条目前加上 0。这个过程如图 2.30(a) 所示。

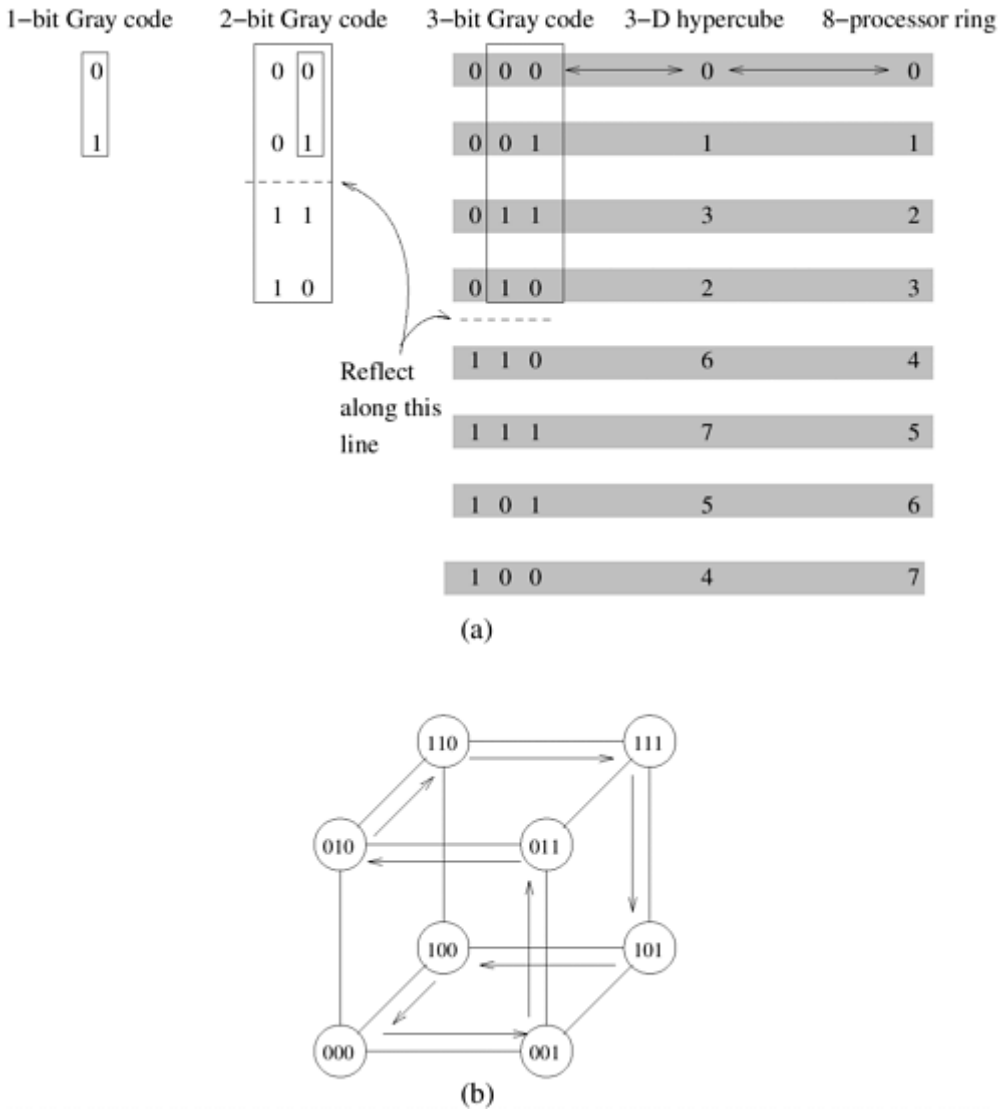


图2.30 (a)三比特反射格雷码环；(b)嵌入三维超立方体

仔细观察格雷码表就会发现，两个相邻条目（$G(i,d)$ 和 $G(i+1,d)$）之间只有一个比特位置不同。由于线性数组中的节点 i 被映射到节点 $G(i,d)$，而节点 $i+1$ 被映射到 $G(i+1,d)$，因此超立方体中有一个直接链接与线性数组中的每个直接链接相对应。(回想一下，在超立方体中，标签只差一个比特位置的两个节点有一个直接链接）。因此，函数 G 指定的映射具有 1 的扩张和 1 的拥塞。图 2.30(b) 展示了将八节点环嵌入三维超立方体的过程。

网格嵌入的超立方体

将网格嵌入超立方体是将环嵌入超立方体的自然延伸。我们可以通过将网格的节点 (i,j) 映射到超立方体的节点 $G(i,r-1)||G(j,s-1)$ 上（其中 $||$ 表示两个灰色编码的连接），将 $2^r \times 2^s$ 的环绕网格嵌入到 2^{r+s} 节点的超立方体中。需要注意的是，网格中的近邻节点被映射到超立方体节点上，这些节点的标签正好相差一个比特位置。因此，这种映射的扩张率为 1，拥塞率为 1。

例如，将一个 2×4 的网格嵌入一个八节点的超立方体中。r 和 s 的值分别为 1 和 2。网格的节点 (i,j) 映射到超立方体的节点 $G(i,1)||G(j,2)$。因此，网格的节点 $(0,0)$ 映射到超立方体的节点 000，因为 $G(0,1)$ 是 0，$G(0,2)$ 是 00；将这两个节点连接起来就得到了超立方体节点的标签 000。同样，网格的节点 $(0,1)$ 映射到超立方体的节点 001，依此类推。图 2.31 展示了将网格嵌入超立方体的过程。

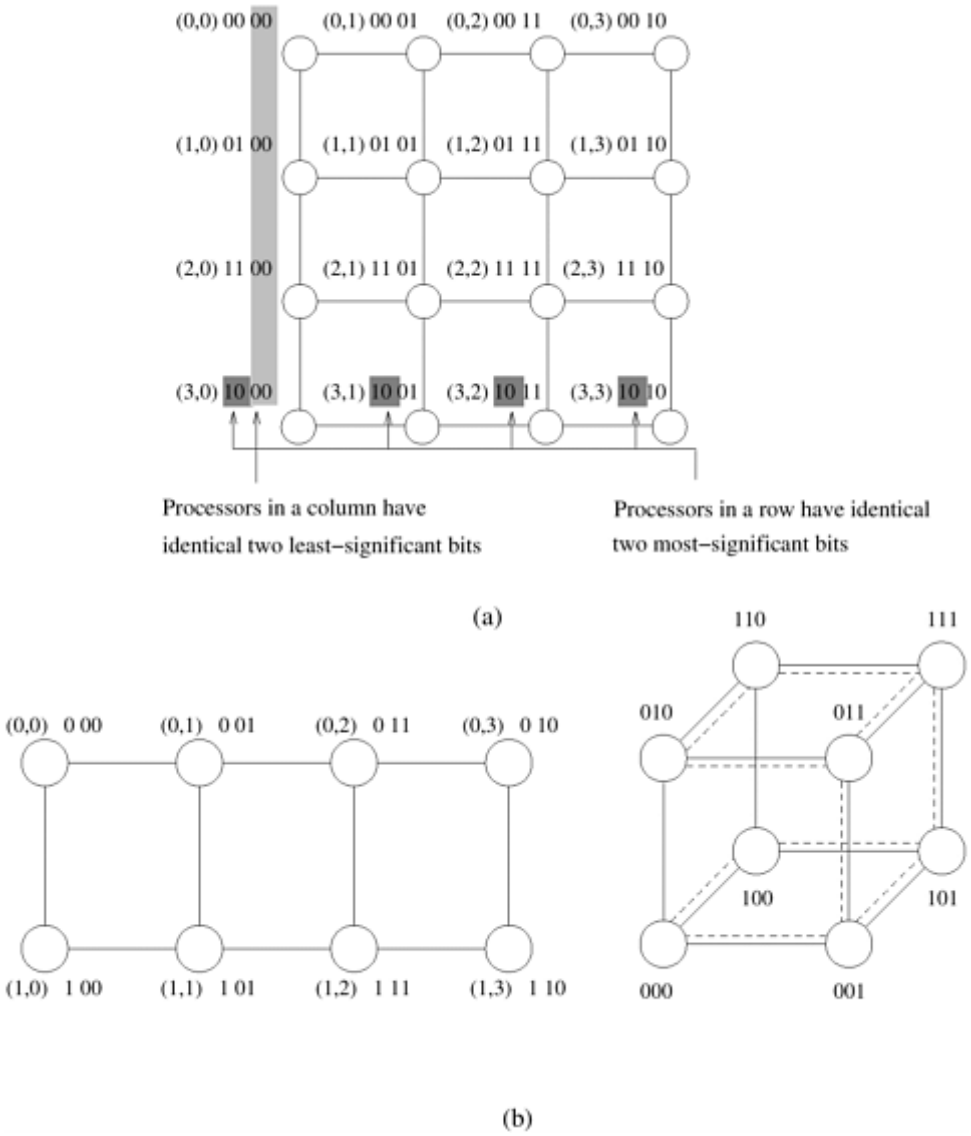


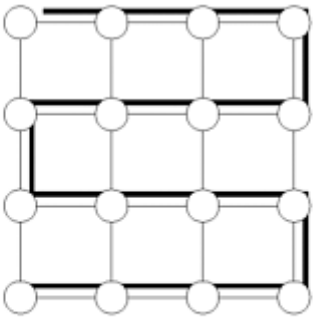
图2.31 (a)4 x 4网格，说明网格节点与四维超立方体节点的映射关系；(b)嵌入三维超立方体的 2 x 4 网格

这种将网格映射为超立方体的方法具有某些有用的特性。网格同一行中的所有节点都被映射为超立方体节点，这些节点的标签有 r 个相同的最有效位。从第 2.4.3 节中我们知道，在一个 $(r+s)$ 维超立方体的节点标签中固定任意 r 个比特，就会产生一个具有 2^s 个节点的 s 维子立方体。由于每个网格节点都映射到超立方体中一个唯一的节点上，而网格中的每一行都有 2^s 个节点，因此网格中的每一行都映射到超立方体中一个不同的子立方体上。同样，网格中的每一列都映射到超立方体中的一个不同子立方体。

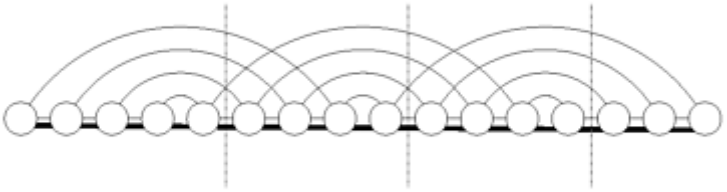
网格嵌入的线性阵列

在此之前，我们一直在考虑将较稀疏的网络嵌入到较密集的网络中。一个 2-D 网格有 $2 \times p$ 个链接。相比之下，一个 p 节点线性阵列有 p 个链接。因此，这种映射必然会产生拥塞。

首先考虑将线性阵列映射到网格中。我们假设网格和线性阵列都没有环绕连接。线性阵列与网格的直观映射如图 2.32 所示。在这里，实线对应线性阵列中的链接，法线对应网格中的链接。从图 2.32(a)中不难看出，线性阵列与网格的映射可以是拥塞为1、扩张为1。



(a) Mapping a linear array into a 2D mesh (congestion 1).



(b) Inverting the mapping – mapping a 2D mesh into a linear array (congestion 5)

图2.32 (a)将16节点线性阵列嵌入2-D网格；(b)相反的映射关系。加粗的实线对应线性阵列中的链接，细线对应网格中的链接

现在考虑这种映射的逆映射，即我们给定一个网格，然后使用相同映射函数的逆映射将网格顶点映射到线性数组中的顶点。图 2.32(b) 展示了这种映射。如前所述，实线对应线性数组中的边，法线对应网格中的边。从图中可以明显看出，这种情况下映射的拥挤度为五，即没有一条实线携带超过五条法线。一般来说，很容易看出这种（反向）映射的拥塞 $\sqrt{p}+1$ 是针对一般的 p 节点映射而言的（通往下一行的每 $\sqrt{p}$ 个边缘都有一条拥塞，还有一条额外的边缘）。

虽然这是一个简单的映射，但目前的问题是我们是否能做得更好。为了回答这个问题，我们使用了两个网络的分段宽度。我们知道，不带环绕链接的二维网格的分段宽度为 $\sqrt{p}$，而线性阵列的分段宽度为 1。

假设二维网格到线性阵列的最佳映射的拥挤度为 r。这意味着，如果我们将线性阵列切成两半（从中间），我们将只切掉一个线性阵列链接，或不超过 r 个网格链接。我们认为，r 必须至少等于网格的分割宽度。这是因为将线性阵列等分为二的同时，也将网格等分为二。因此，根据分段宽度的定义，至少有 $\sqrt{p}$ 个网格链接必须穿过分段。因此，连接两半的一个线性阵列链路必须至少携带 $\sqrt{p}$ 个网格链路。因此，任何映射的拥塞下限都是 $\sqrt{p}$。这与图 2.32(b)中的简单（逆）映射几乎相同。

当把密度较高的网络映射到密度较低的网络时，上文确定的下限具有更普遍的适用性。我们可以合理地认为，将具有 x 个链接的网络 S 映射到具有 y 个链接的网络 Q 时，拥塞下限为 x/y。在从网状网络映射到线性阵列的情况下，下限为 $2p/p$，即 2。然而，这个下限过于保守。事实上，通过研究两个网络的平分宽度，可以得到更严格的下限。我们将在下一节进一步说明。

超立方体嵌入的二维网格

考虑将 p 节点超立方体嵌入 p 节点二维网格。为方便起见，我们假设 p 是 2 的偶次幂。在这种情况下，可以将超立方体可视化为 $\sqrt{p}$ 子立方体，每个子立方体都有 $\sqrt{p}$ 个节点。具体做法如下：设 $d=\log p$ 为超立方体的维数。根据假设，我们知道 d 是偶数。我们取 $d/2$ 个最小有效位来定义节点的各个子立方体。例如，在 4D 超立方体的情况下，我们使用较低的两位定义子立方体为 $(0000,0001,0011,0010)$、$(0100,0101,0111,0110)$、$(1100,1101,1111,1110)$ 和 $(1000,1001,1011,1010)$。此时请注意，如果我们固定所有这些子立方体的 $d/2$ 个最小有效位，我们就会得到由 $d/2$ 个最有效位定义的另一个子立方体。例如，如果我们将所有子立方体的低两位固定为 10，就会得到节点(0010、0110、1110、1010)。读者可以验证这对应的是一个二维子立方体。

超立方体到网格的映射现在可以定义如下：每 $\sqrt{p}$ 个节点子立方体映射到网格的一个 $\sqrt{p}$ 节点行。我们只需将线性数组到超立方体的映射倒置即可。$\sqrt{p}$ 节点超立方体的分段宽度为 $\sqrt{p}/2$，$\sqrt{p}$ 节点行对应的二分宽度为1。因此，子立方体行映射的拥挤度为 $\sqrt{p}/2$（在连接两半行的边缘处）。图 2.33(a) 和图 2.33 (b) 展示了 $p = 16$ 和 $p = 32$ 的情况。通过这种方式，我们可以将每个子立方体映射到网格中的不同行。请注意，虽然我们已经计算了子立方体到行的映射所产生的拥塞，但还没有解决列映射所产生的拥塞。我们将超立方体节点映射到网格中的方式是，将超立方体中具有相同 $d/2$ 最小有效位的节点映射到同一列。这就形成了子立方体到列的映射，其中每个子立方体/列都有 $\sqrt{p}$ 个节点。使用与子立方体到行映射相同的论证，这将导致 $\sqrt{p}/2$ 的拥塞，因为行和列映射产生的拥塞影响到不相连的边集，所以该映射的总拥塞为 $\sqrt{p}/2$。

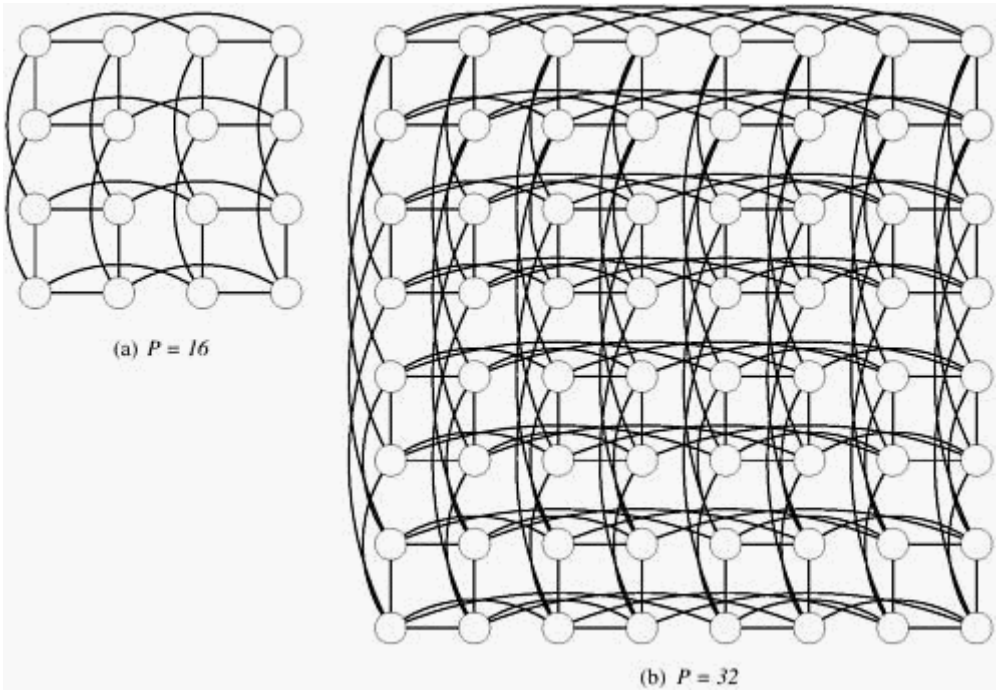


图2.33 在二维网格中嵌入超立方体

为了确定拥塞下限，我们沿用第 2.7.1 节中的论证方法。由于超立方体的分割宽度是 $p/2$，而网格的分割宽度是 $\sqrt{p}$，因此拥塞下限就是它们的比值，即 $\sqrt{p}/2$。我们注意到，我们的映射产生了这个拥塞下限。

进程到处理器映射和互连网络设计

前几节的分析表明，可以将较密集的网络映射为较稀疏的网络，并产生相关的拥塞开销。这意味着，增加链接带宽以补偿拥塞的稀疏网络，其性能有望与密集网络不相上下（模数膨胀效应）。例如，链接速度快 $\sqrt{p}/2$ 倍的网格，其性能可与超立方体相媲美。我们称这种网格为胖网格。胖网格的分段带宽与超立方体相同，但直径更大。正如我们在第 2.5.1 节中看到的，通过使用适当的信息路由技术，可以将节点距离的影响降至最低。值得注意的是，高维度网络涉及更复杂的布局、导线交叉和可变导线长度。基于这些原因，低维网络为设计互连提供了极具吸引力的替代方法。现在，我们将对并行架构的成本-性能权衡进行更正式的研究。

2.7.2 成本与性能的权衡

现在，我们将探讨如何利用各种成本指标来研究互连网络中的性价比权衡问题。我们通过分析成本相同的网状网络和超立方网络的性能来说明这一点。

如果网络的成本与导线数成正比，那么每个通道有 $(\log p)/4$ 根导线的 p 节点正方形环绕网格的成本与每个通道只有一根导线的 p 节点超立方体的成本相同。让我们比较一下这两个网络的平均通信时间。二维环绕网中任意两个节点之间的平均距离 l_{av} 为 $\sqrt{p}/2$，超立方体中的平均距离 l_{av} 为 $(\log p)/2$。在使用直通路由的网络中，相隔 l_{av} 跳的节点之间发送大小为 m 的信息所需的时间为 $t_s + t_h l_{av} + t_w m$。由于网格的信道宽度放大了 $(\log p)/4$ 倍，因此每个字的传输时间也减少了相同的系数。因此，如果超立方体上的每字传输时间为 t_w，那么具有增大通道的网格上的相同时间为 $4t_w/(\log p)$。因此，超立方体的平均通信延迟时间为 $t_s + t_h(\log p)/2 + t_w m$，而成本相同的环绕网格的平均通信延迟时间为 $t_s + t_h\sqrt{p}/2 + 4t_w m/(\log p)$

现在让我们研究一下这些表达式的行为。在节点数固定的情况下，随着信息量的增加，t_w 引起的通信项将占主导地位。比较这两种网络的 t_w，我们会发现，如果 p 大于 16，且信息大小 m 足够大，则环绕网格的时间 $(4t_w m/(\log p))$小于超立方体的时间 $(t_w m)$。在这种情况下，随机节点对之间的大信息点对点通信在带穿透路由的环绕网格上所需的时间比在相同成本的超立方体上所需的时间要少。此外，对于适合网格通信的算法，每个信道的额外带宽会带来更好的性能。需要注意的是，如果采用存储转发路由，网格的成本效益就不再高于超立方体。对于 k-d 立方体的一般情况，也可以分析类似的性价比权衡。

上述通信时间是在网络负载较轻的情况下计算得出的。随着报文数量的增加，网络上会出现争用现象。与超立方网络相比，争用对网状网络的不利影响更大。因此，如果网络负载较重，超立方体网络的性能将优于网状网络。

如果网络的成本与分段宽度成正比，那么每个通道 $\sqrt{p}/4$ 根导线的 p 节点环绕网格的成本等于每个通道一根导线的 p 节点超立方体的成本。让我们进行与上述类似的分析，利用这一成本指标来研究成本与性能之间的权衡。由于网状信道的宽度是 $\sqrt{p}/4$ 倍，因此每个字的传输时间也会降低相同的系数。因此，成本相同的超立方网络和网状网络的通信时间分别为 $t_s + t_h(\log p)/2 + t_w m$ 和 $t_s + t_h\sqrt{p}/2 + 4t_w m/\sqrt{p}$。同样，在节点数一定的情况下，当信息大小 m 变大时，t_w 项将占据主导地位。比较这两个网络的这个项，我们会发现，对于 $p > 16$ 且信息量足够大的情况，网状结构优于相同成本的超立方体。因此，对于足够大的信息量，只要网络负载较轻，网格总是优于相同成本的超立方体。即使网络负载很重，网状结构的性能也与相同成本的超立方体相似。