

# 第 11 章 并行性和局部性优化

本章将介绍一个编译器如何增强处理数组的计算密集型程序中的并行性和局部性，以便提高目标程序在多处理器系统上的运行速度。很多科学、工程和商业领域的应用对计算能力的要求是永无止境的。这些例子包括气象预报，用于药物设计的蛋白质折叠，用于设计航空推进系统的流体力学和用于高能物理中强相互作用研究的量子色动力学。

加快计算过程的方法之一就是使用并行技术。遗憾的是，开发可以利用并行机器的软件并不是容易的事情。把计算过程分割为多个可以在不同并行处理器上执行的单元已经是很困难的事情了，但是这样的分割还不一定能保证提高速度。我们还必须把处理器之间的通信量减到最小，因为通信开销很容易使并行代码运行得甚至比串行代码还慢！

尽可能降低通信开销可以被当作是提高程序的数据局部性 (data locality) 的一个特殊情况。一般来说，如果一个处理器经常访问它最近使用的同一组数据，我们就说这个程序具有良好的数据局部性。如果并行机上的一个处理器具有良好的局部性，它就不需要和其他处理器频繁通信。因此，并行性和数据局部性必须放在一起考虑。数据局部性本身对于单个处理器的性能也是很重要的。现代处理器的内存层次结构中都有一层或多层高速缓存，一次内存访问可能会需要几十个机器周期，而在高速缓存命中命中的运算只需要几个机器周期。如果一个程序没有良好的数据局部性，并经常在缓存访问中脱靶，那么它的性能就会受到影响。

在本章中同时处理并行性和数据局部性的另一个理由是它们使用的理论相同。如果我们知道如何优化数据局部性，也就知道了并行性在哪里。在本章，你将看到第 9 章中为进行数据流分析而使用的程序模型对并行化和局部性优化来说是不够的。原因是在数据流分析中的工作假设我们不需要区分到达一个给定语句的不同路径。实际上，第 9 章中的那些技术利用了不需要区分同一个语句的（例如，在循环中的）不同执行的事实。为了实现代码并行化，需要考虑同一语句的不同动态执行实例之间的依赖关系，以决定它们是否可以在不同处理器上同时执行。

本章关注的是用于优化某一类数值应用的技术。这类应用使用数组作为数据结构，并且以一种简单且规则的模式访问这些数据结构。更明确地说，我们研究的程序中包含的数组访问与外围循环的下标变量之间具有仿射关系。例如，如果  $i$  和  $j$  是外围循环的下标变量，那么  $Z[i][j]$  和  $Z[i][i+j]$  都是仿射访问。如果关于一个或多个变量  $x_1, x_2, \dots, x_n$  的函数可以被表示为一个常数加上常数乘以这些变量的和，即  $c_0 + c_1x_1 + c_2x_2 + \dots + c_nx_n$ ，其中  $c_0, c_1, c_2, \dots, c_n$  是常数，那么这个函数就是仿射的。仿射函数通常称为线性函数，虽然严格地讲线性函数不能有  $c_0$  项。

下面是这个领域内的循环的一个简单的例子：

```
for (i = 0; i < 10; i++) {  
    Z[i] = 0;  
}
```

因为这个循环的各个迭代对不同的内存位置进行写运算，不同的处理器可以并发地执行不同的迭代。另一方面，如果有另一个语句  $Z[j] = 1$  正在执行，我们就要担心  $i$  是否可能和  $j$  相同，以及如果相同的话，要按照什么顺序来执行这两个具有相同数组下标值的语句的实例。

知道哪些迭代可能指向同一个内存位置是很重要的。这个知识使我们可以描述调度代码时必须遵守的数据依赖，不管被调度的代码是在单处理器上运行还是在多处理器上运行。我们的目标是找到一个遵守所有数据依赖关系的调度方案，使得访问相同内存位置或高速缓存线的运

算尽可能靠近执行；并且在多处理器的情况下，把这些代码放在同一个处理器上执行。

本章中给出的理论是基于线性代数和整数规划技术的。我们把一个深度为  $n$  的循环嵌套结构中的迭代建模为一个  $n$  维多面体。该多面体的边界用代码中循环的界限来描述。仿射函数把每个迭代映射成为它所访问的数组位置。我们可以使用整数线性规划技术来确定是否存在可能指向同一个位置的两个迭代。

我们在这里讨论的代码转换方法的集合可以分成两类：仿射分划 (affine partitioning) 和分块 (blocking)。仿射分划把迭代的多面体分割成多个部分，在不同的机器上执行各个部分，或者一个一个地顺序执行各个部分。另一方面，分块技术创建了一个由迭代组成的层次结构。假设有一个以逐行方式扫描整个数组的循环。我们可以把这个数组分成多个块，并且逐块访问其中的元素。最后得到的代码由遍历这些块的外层循环和扫描各块中元素的内层循环组成。线性代数技术用来确定最好的仿射分划和最好的分块方案。

在接下来的内容中，我们先在 11.1 节中概述关于并行计算和局部性优化的概念。然后，在 11.2 节中给出一个具体例子——矩阵乘法。这个例子用于说明循环转换，即对一个循环内的计算过程进行重新排序，是如何既提高局部性又提高并行化效率的。

11.3 节到 11.6 节给出了循环转换所必需的基本信息。11.3 节介绍如何对一个循环嵌套结构中的各个迭代进行建模；11.4 介绍如何对数组下标函数建模。这类函数把每个循环迭代映射到被该迭代访问的数组位置；11.5 节介绍如何使用标准线性代数算法来确定一个循环中的哪些迭代访问了相同的数组位置或高速缓存线；11.6 节说明如何找到一个程序中的数组引用之间的所有数据依赖关系。

本章的其余部分应用这些基本技术来进行优化。11.7 节首先考虑一个比较简单的问题，即寻找不需要同步的并行性。为了找到最佳的仿射分划方案，我们只需要找到满足下面约束的解：具有数据依赖关系的运算必须被分配到同一个处理器上。

当然，没有多少程序能够在不需要任何同步的情况下实现并行化。因此，在 11.8 节到 11.9.9 节将探讨寻找需要同步的并行性的一般情况。我们引入了流水线化的概念，说明如何寻找能够达到一个程序所允许的最大流水线化程度的仿射分划。我们将在 11.10 节中说明如何优化数据局部性。最后，我们讨论如何把仿射变换用于其他形式的并行性。

## 11.1 基本概念

本节介绍了一些和并行化及局部性优化相关的基本概念。如果运算可以并行执行，它们也可以为了其他目的（比如局部性）而重新排序。反过来，如果一个程序中的数据依赖关系决定了一个程序中的指令必须串行执行，这个程序显然不具有并行性，同时也没有任何机会对指令重新排序以提高数据局部性。因此，并行化分析也可以寻找可用的机会，为了提高数据局部性而进行移动代码。

为了尽可能降低并行代码之间的通信量，我们把所有相关的运算都组合在一起，并把它们分配给同一个处理器。因此得到的代码必然具有数据局部性。在单处理器上获得良好数据局部性的一个粗略的方法是让该处理器顺序执行在并行化时分配给各个处理器的代码。

在本节中，我们首先概述并行计算机体系结构。然后给出并行化的基本概念。并行化是一种可以引起巨大变化的转换技术；同时会介绍一些对并行化有用的概念。然后我们讨论了如何把这些类似的考虑用于优化数据局部性。最后，我们将非正式地介绍本章中使用到的数学概念。

### 11.1.1 多处理器

最流行的并行机体系结构是对称多处理器 (Symmetric MultiProcessor, SMP) 结构。高性能个

人计算机通常有两个处理器，而很多服务器有四个、八个，甚至几十个处理器。不仅如此，因为现在已经能够实现把多个高性能处理器集成在一个芯片上，所以多处理器得到了更加广泛的应用。

一个对称多处理器结构中的各个处理器共享同一个地址空间。进行通信时，一个处理器把数据写到某个内存位置，然后由其他处理器来读取。对称多处理器的名字就是源于所有的处理器能够以统一的访问时间来访问系统中所有的内存。图 11-1 给出了一个多处理器的高层体系结构。各个处理器都拥有自己的第一层、第二层高速缓存，有时甚至拥有第三层高速缓存。

最高层的高速缓存通过通常的共享总线连接到物理内存。

对称多处理器使用一致缓存协议 (coherent cache protocol) 来对程序员隐藏高速缓存的存在。在这样的协议下，多个处理器可以同时保存同一高速缓存线的多个拷贝<sup>⊖</sup>，前提是它们都只是读取数据。当一个处理器想向一个高速缓存线写数据时，所有其他的高速缓存拷贝都被删除。当一个处理器请求的数据不在高速缓存中时，该请求会向外传递到共享总线，并从内存或其他处理器的高速缓存中获取数据。

一个处理器和另一个处理器通信所花的时间大约是内存访问时间的两倍。以缓存线为单位的数据必须首先从源处理器的高速缓存写到内存中，然后再从内存取出放到第二个处理器的高速缓存中。你可能认为处理器之间的通信相对便宜，因为它只需要大约两倍于内存访问的时间。但是，必须记住，相对于在高速缓存中命中的访问运算，内存访问已经非常昂贵了——内存访问可能慢几百倍。这个分析十分清楚地说明了高效并行化和局部性分析之间的相似性。不管是单独工作的处理器还是多处理器环境下的处理器，要使它高效工作就必须使它能够在高速缓存中找到运算所需的数据。

在 21 世纪早期，对称多处理器的设计超不过几十个处理器的规模，原因是受到共享总线或任何其他用于此目的的互联设施的限制。它们在速度上不能应对不断增加的处理器数量。为了使得处理器设计可不断扩大规模，体系结构设计师们又在内存层次结构中引入了新的一层。他们不再使用和各个处理器距离一样远的内存，而是把内存分配给各个处理器，以便每个处理器能够快速访问它的局部内存，如图 11-2 所示。因此远

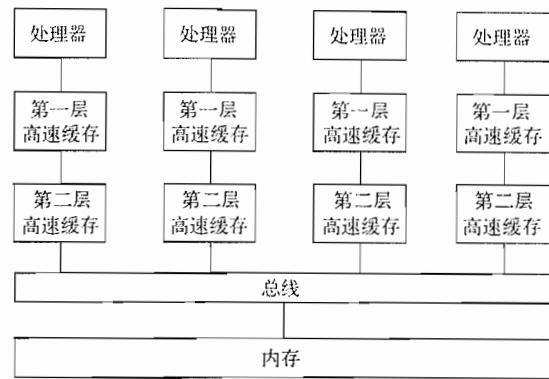


图 11-1 对称多处理器体系结构

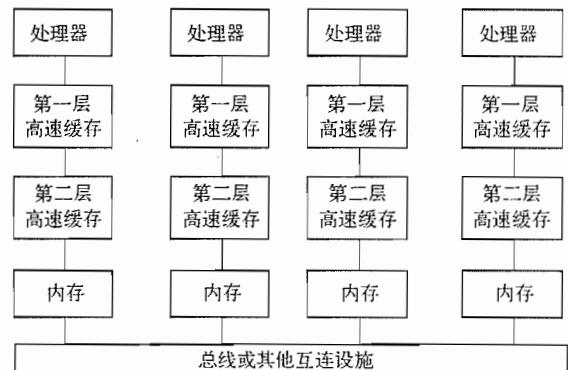


图 11-2 分布式内存的机器

程内存成为内存层次的下一级，它们的总量要比局部内存大，但是访问时花费的时间也更长。和

<sup>⊖</sup> 你可以复习一下 7.4 节中对高速缓存和高速缓存线的讨论。

内存体系结构设计时高速存储设备必然容量较小的原理类似，支持处理器之间快速通信的机器所拥有的处理器数量也必然较少。

有两种不同的带有分布式内存的并行机：NUMA(NonUniform Memory Access, 不一致内存访问)机器以及消息传递机器。NUMA 体系结构为软件提供了一个共享地址空间，允许处理器通过读写共享内存来通信。然而，在消息传递机器上的处理器有各自的地址空间，处理器之间通过相互传递消息来通信。请注意，虽然为共享内存机器写代码要相对简单，但任何一种机器要想具有好的性能，都要求软件具有良好的局部性。

### 11.1.2 应用中的并行性

我们使用两种高层次的度量来估计一个并行应用性能运行的好坏：并行性覆盖率和并行性粒度，前者指明了并行执行的计算过程所占的百分比；后者指出了各个处理器在不和其他处理器通信或同步的情况下能够运行的计算量。一个特别有吸引力的并行化目标是循环：一个循环可能有很多个迭代，并且如果它们之间相互独立，我们就找到了一个很大的并行性的源头。

#### Amdahl 定律

并行性覆盖率的重要性可以用 Amdahl 定律简洁地表示。Amdahl 定律的内容是，如果  $f$  是被并行化代码的比率，并且如果并行化版本在一个有  $p$  个处理器的机器上运行，且没有任何通信或者并行化开销，那么此时的加速比是：

$$\frac{1}{(1-f)+(f/p)}$$

比如，如果有一半的计算是串行执行的，那么不管我们使用多少个处理器，计算过程最多能够以双倍速度运行。如果我们有 4 个处理器，可达到的加速比是 1.6。即使并行化覆盖率达到 90%，我们在 4 个处理器上最多能够得到 3 倍的加速比，而在无限多的处理器上得到的加速比最多为 10。

#### 并行化的粒度

理想情况是一个应用的全部计算过程能够被划分成为很多粗粒度的独立任务，因为我们可以直接把不同的任务分配给不同的处理器。这样的例子之一是 SETI(Search for Extra Terrestrial Intelligence, 寻找外星智慧生物)项目，这个实验使用很多通过 Internet 相连的家庭计算机来并行分析射电望远镜数据的不同部分。每一个工作单元只需要少量的输入并生成少量的输出，并且可以独立于所有其他单元完成。由于这些原因，虽然 Internet 具有相对高的通信延时和低带宽，但这样的计算工作仍然在与 Internet 连接的机器上运行得很好。

大部分应用要求处理器之间有更多的通信和交互，但仍然支持粗粒度的并行性。比如，考虑一个 Web 服务器，它负责响应大量独立的对一个公共数据库的请求。我们可以在一个多处理器机器上运行这个应用，使用一个线程来实现数据库访问，并使用其他线程来对用户请求提供服务。其他的例子包括药物设计和机翼动力学模拟。在这些例子中，人们可以独立地求解针对不同的参数的结果。在模拟的时候，有时即使对于一组参数求解也会花很长时间，因此期望能够使用并行化技术进行加速。当一个应用中的可用并行性的粒度降低时，就需要更好的处理器之间的通信支持和更多的编程工作量。

很多运行时间很长的科学及工程应用具有简单的控制结构和很大的数据集合，因此它们要比上面讨论的应用更容易实现更细粒度的并行化。因此，本章主要考虑的是那些用于数值应用的技术，特别是那些把大部分时间用于多维数组中的数据运算的应用。下面我们就探讨一下这类程序。

### 11.1.3 循环层次上的并行性

循环是并行化的主要目标，在使用数组的应用中更是如此。运行时间较长的应用通常具有大型数组，从而产生具有很多迭代的循环。其中的每一个迭代处理数组中的一个元素。迭代之间相互独立的循环并不难找到。我们可以把这类循环的大量迭代分配给多个处理器。如果每个迭代的工作量基本相同，那么简单地在处理器之间平均分配迭代就可以得到最大的并行性。例 11.1 是一个特别简单的例子，它说明我们如何利用循环层次的并行性。

#### 例 11.1 下面的循环

```
for (i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

计算了向量  $X$  和  $Y$  的元素之间的平方差，并把它们存放到数组  $Z$  中。这个循环是可并行化的，因为每个迭代访问不同的数据集合。我们可以在具有  $M$  个处理器的计算机上执行这个循环。给每个处理器赋予唯一的 ID  $p = 0, 1, 2, \dots, M-1$ ，并让每个处理器执行同样的代码：

```
b = ceil(n/M);
for (i = b*p; i < min(n, b*(p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

我们把这个循环中的迭代平均分配给各个处理器，第  $p$  个处理器被分配执行第  $p$  组迭代。请注意，迭代数目不一定能够被  $M$  整除，因此我们通过在程序中引入一个求最小值的运算来保证最后一个处理器执行的时候不会越过原来的循环界限。□

#### 任务层次的并行

有可能在一个循环的迭代之外找到并行性。比如，我们可以把两个不同的函数调用或两个独立的循环分配给两个处理器。这种形式的并行性称为任务并行性 (task parallelism)。和循环层次的并行性相比，任务层次的并行性作为一个并行性来源的吸引力较弱。原因是对于每个程序来说，其独立任务的数量是固定的，并且不能随着数据大小的增加而增加。而一个典型循环的迭代次数则会随数据的增加而增加。不仅如此，这些任务的大小通常并不相等，因此难以让所有的处理器在所有时间都有事可做。

例 11.1 中显示的并行代码是一个 SPMD (Single Program Multiple Data，单程序多数据) 程序。所有的处理器都执行相同的代码，只是这些代码都带有各个处理器的唯一标识作为参数，因此不同的处理器完成不同的动作。通常会有一个被称为主处理器 (master) 的处理器来执行计算中的所有串行部分。在到达代码中已并行化的部分时，主处理器激活所有从处理器 (slave)。所有从处理器同时执行代码中已经被并行化的区域。在每个并行化代码区域的尾部，所有这些处理器参与栅障同步 (Barrier Synchronization)。只有等到所有处理器都已经执行完它们进入一个同步栅障之前的全部运算之后，各个处理器才会被允许离开这个栅障并执行栅障之后的运算。

如果我们只是把类似于例 11.1 中的简单循环并行化，那么得到的代码通常具有较低的并行性覆盖率和相对较细的并行性粒度。我们倾向于把一个程序的最外层循环并行化，因为这样会得到最粗的并行性粒度。比如，考虑二维 FFT 变换的应用，它在一个  $n \times n$  的数据集上运行。这个程序对各行数据执行  $n$  次 FFT 变换，然后再对各列数据执行  $n$  次 FFT 变换。我们倾向于把  $n$  个独立 FFT 变换中的每一个分配给一个处理器，而不是使用多个处理器协作完成一次 FFT 变换。

这样做使得代码容易书写，算法的并行性覆盖率达到 100%，并且代码具有很好的数据局部性，因为在计算一个 FFT 时不需要进行任何通信。

很多应用没有可并行化的大的最外层循环。然而，这些应用的执行时间通常由耗时的内核决定。这些内核可能具有几百行代码，包含了不同嵌套层次的循环。有时可以单独地处理内核部分，通过集中考虑它的局部性，重新组织它的计算过程并把它分划成为多个独立的单元。

#### 11.1.4 数据局部性

在对程序进行并行化的时候，需要考虑两种略微不同的数据局部性概念。当同一个数据在短时间内被多次使用时就产生了时间局部性 (temporal locality)。当位置相近的不同数据元素在短时间内被使用的时候就产生了空间局部性 (spatial locality)。空间局部性的一个很重要的形式是在同一个高速缓存线中出现的所有数据元素被一起使用。这种形式之所以重要的理由是当需要一个来自某高速缓存线的元素时，这个高速缓存线的所有元素都会被加载到高速缓存中。如果很快就会使用这些元素，那么它们很可能还在高速缓存中。这种空间局部性的效果使得高速缓存脱靶的概率降到最小，也使得该程序得到了较好的加速比。

程序的内核经常可以使用多种不同的方式来书写。它们之间在语义上等价，但是数据局部性和性能却相差很大。例 11.2 给出了例 11.1 中的计算过程的另一种表示方法。

**例 11.2** 和例 11.1 类似，下面的代码也能够计算得到向量  $X$  和  $Y$  的元素之间的差的平方。

```
for (i = 0; i < n; i++)
    Z[i] = X[i] - Y[i];
for (i = 0; i < n; i++)
    Z[i] = Z[i] * Z[i];
```

第一个循环计算元素之间的差，第二个循环计算差的平方。这样的代码经常会在实际程序中遇到，因为这就是我们为向量机 (vector machine) 优化程序的办法。向量机是一种超级计算机，拥有可以一次性对整个向量进行简单算术运算的指令。我们可以看到，这里的两个循环在例 11.1 中被融合 (fuse) 为一个循环。

我们已经知道这两个程序完成同样的计算工作，那么哪一个程序比较好呢？例 11.1 中被融合在一起的循环拥有比较好的性能，因为它具有较好的数据局部性。每个差值一生成就立刻进行平方运算。实际上，我们可以将差值存放在一个寄存器中，求它的平方，并把结果一次性写入内存位置  $Z[i]$ 。反过来，本例中的代码需要获取  $Z[i]$  值一次，并对它写两次。不仅如此，如果数组的大小比高速缓存大，那么当  $Z[i]$  在这个例子中被第二次使用时，需要从内存中重新获取这个值。因此，这个代码的运行速度可能低很多。□

**例 11.3** 假设我们希望把按行存放（回忆一下 6.4.3 节）的数组  $Z$  设置为全零。图 11-3a 和图 11-3b 分别对这个数组进行逐列和逐行扫描。我们可以对图 11-3a 中的循环进行变换得到图 11-3b 的循环。从空间局部性的角度看，以逐行方式执行置零运算是比较好的，因为在一个高速缓存线中的所有字都被顺序置零了。在逐列处理的方法中，虽然每个高速缓存线都被外层循环的下一个迭代复用，但当列的大小比高速缓存大的时候，高速缓存线在被复用之前就会被抛出高速缓存。为了得到最好的性能，我们使用类似于例 11.1 中所使用的方法，把图 11-3b 的外层循环并行化，结果如图 11-3c 所示。□

上面的两个例子解释了和操作数组的数值应用相关的几个重要性质：

- 数组代码经常有很多可并行化的循环。
- 当循环具有并行性的时候，它们的迭代可以按照任意的顺序执行。它们可以通过重新排序大幅提高数据局部性。
- 当我们创建出大的相互独立的并行计算单元时，以串行方式执行它们往往能得到良好的

数据局部性。

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        Z[i, j] = 0;
```

a) 逐列将一个数组置零

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[i, j] = 0;
```

b) 逐行将一个数组置零

```
b = ceil(n/M);
for (i = b*p; i < min(n, b*(p+1)); i++)
    for (j = 0; j < n; j++)
        Z[i, j] = 0;
```

c) 并行地按照逐行方式将一个数组置零

图 11-3 将一个数组置零的顺序代码和并行代码

### 11.1.5 仿射变换理论概述

编写正确且高效的串行程序是困难的，编写正确且高效的并行程序则更加困难。编写的难度随着所利用的并发性粒度的降低而增长。我们在上面看到过，程序员必须注意数据局部性以获取高性能。此外，把一个已有的串行程序并行化的任务是极端困难的。找出程序中的所有依赖关系很困难，当这个程序并不是我们所熟悉的类型时尤其如此。调试一个并行程序也很困难，因为错误可能是不确定的。

在理想情况下，一个并行的编译器自动地把普通的串行程序翻译成高效的并行程序，并对这些程序的局部性进行优化。遗憾的是，编译器并不知道有关这个应用的高层知识，它只能保持原算法的语义，而这些算法未必适合进行并行化。而且，程序员也可能随意地做出选择，结果限制了程序的并行性。

一些 Fortran 数值应用显示了并行化和局部性优化的成功。这些应用以仿射访问的方式对数组进行操作。因为没有指针和指针运算，所以对 Fortran 的分析相对容易。请注意，不是所有的应用都有仿射访问。最值得注意的是，很多数值应用是在稀疏矩阵上运算的。这些矩阵的元素是通过另一个数组间接访问的。本章关注的是内核的并行化和优化，这些内核通常由几十行代码组成。

如例 11.2 和例 11.3 所示，并行化和局部性优化需要我们考虑一个循环的不同实例以及实例之间的关系。这个情况和数据流分析有着很大的不同。在数据流分析中，我们把所有实例的相关信息组合在一起考虑。

对于带有数组访问的循环的优化问题，我们使用三种类型的空间。每个空间可以看成是一维或多维栅格中的点集。

1) 迭代空间 (iteration space) 是在一次计算过程中动态执行实例的集合，也就是各个循环下标的取值的组合。

2) 数据空间 (data space) 是被访问的数组元素的集合。

3) 处理器空间 (processor space) 是系统中的处理器的集合。通常情况下，这些处理器使用整数或者整数向量进行编号，以便相互区分。

优化问题的输入是各个迭代被执行的串行顺序以及一个仿射的数组访问函数（例如， $X[i, j + 1]$ ）。这个函数描述了迭代空间中的哪个实例访问数据空间中的哪个元素。

这个优化的输出也是用仿射函数表示的，它定义了每个处理器在什么时候做什么事情。为了指明每个处理器所做的工作，我们使用一个仿射函数把原迭代空间中的实例映射到各个处理

器上。为了描述什么时候执行迭代，我们使用一个仿射函数把迭代空间中的实例映射成为一个新的顺序。通过分析程序中的数组访问函数所蕴涵的数据依赖关系和复用模式，就可以得到调度方案。

下面的例子将说明上述三个空间——迭代空间、数据空间和处理器空间。它也非正式地介绍使用这些空间来并行化代码时涉及的重要概念和需要解决的问题。在后面的各节中将详细介绍这些概念。

**例 11.4** 图 11-4 解释了下面程序中用到的不同空间和这些空间之间的关系。

```
float Z[100];
for (i = 0; i < 10; i++)
    Z[i+10] = Z[i];
```

这三个空间和它们之间的映射如下：

1) **迭代空间**：迭代空间是循环的迭代的集合。各个迭代的 ID 通过循环下标变量的取值给出。一个深度为  $d$  的循环嵌套结构(即有  $d$  层嵌套的循环)有  $d$  个下标变量，因此被建模为一个  $d$  维空间。迭代空间通过循环下标变量的上下界来限定。这个例子的循环定义了一个由 10 个迭代组成的一维空间。空间中的迭代用循环下标变量的值： $i = 0, 1, \dots, 9$  表示。

2) **数据空间**：数据空间由数组声明直接给出。在这个例子里，数组中的元素用  $a = 0, 1, \dots, 99$  作为下标。虽然所有的数组在一个程序的地址空间中是线性存放的，我们还是把  $n$  维向量当作  $n$  维空间处理，并假设各个下标总是在它们的界限之内。当然，这个例子里的数组是一维的。

3) **处理器空间**：在我们开始并行化时，假设系统中有无限多个虚拟处理器。这些处理器以多维空间的方式进行组织，每一个维度对应于我们试图并行化的循环嵌套结构中的一个循环。在并行化完成之后，如果我们拥有的物理处理器少于虚拟处理器，就把虚拟处理器等分成多个块，每一块分配给一个物理处理器。在这个例子中，我们只需要 10 个处理器，循环的每一个迭代分配一个处理器。在图 11-4 中，我们假设处理器被组织成一个一维空间且用  $0, 1, \dots, 9$  进行编号。循环的第  $i$  个迭代被分配给处理器  $i$ 。假如只有五个处理器，我们可以把迭代 0 和 1 分配给处理器 0，迭代 2 和 3 分配给处理器 1，以此类推。因为迭代是独立的，所以只要五个处理器中的每一个分得两个迭代，我们怎么进行分配并不重要。

4) **仿射数组下标函数**：代码中的每个数组访问都描述了一个从迭代空间中的迭代到数据空间中的数组元素的映射。如果这个访问函数是把各个循环下标变量乘以常量并求和，然后加上一些常量值，那么这个函数就是仿射的。本例中的两个数组下标函数  $i + 10$  和  $i$  都是仿射的。我们可以根据访问函数求出被访问数据的维度(dimension)。在本例中，因为每个下标函数只有一个循环变量，所以被访问数组元素的空间是一维的。

5) **仿射分划**：我们使用一个仿射函数把一个迭代空间中的各个迭代分配给处理器空间中的各个处理器，由此把一个循环并行化。在我们的例子中，我们直接把迭代  $i$  分配给处理器  $i$ ，也可以使用仿射函数来指定一个新的执行顺序。如果我们希望以相反的顺序串行地执行上面的循环，那么可以用一个仿射表达式  $10 - i$  明确指定排序函数。这样，第九个迭代就是被第一个执行的迭代。

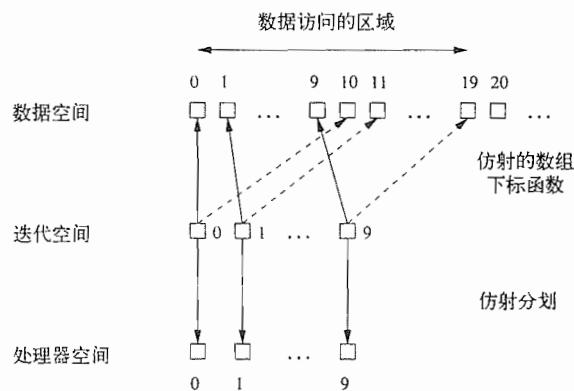


图 11-4 例 11.4 的迭代空间、数据空间和处理器空间

代，以此类推。

6) 被访问数据的区域：知道一个迭代所访问数据的区域有助于找到最好的仿射分划。把迭代空间的信息和数组下标函数结合起来，我们就可以得出被访问数据的区域。在本例中，数组访问  $Z[i+10]$  触及的空间是  $\{a \mid 10 \leq a < 20\}$ ，而数组访问  $Z[i]$  触及的空间是  $\{a \mid 0 \leq a < 10\}$ 。

7) 数据依赖：为了确定被处理的循环是否可被并行化，我们需要知道在各个迭代之间是否存在数据依赖关系。在这个例子中，我们首先考虑该循环中的写访问之间的依赖关系。因为访问函数  $Z[i+10]$  把不同迭代映射到不同的数组位置，不同迭代向数组写数据的顺序上不存在依赖关系。那么在读访问和写访问之间存在依赖关系吗？因为只有  $Z[10], Z[11], \dots, Z[19]$ （通过访问  $Z[i+10]$ ）被写，而只有  $Z[0], Z[1], \dots, Z[9]$ （通过访问  $Z[i]$ ）被读，因此一个读访问和一个写访问的相对顺序不存在依赖关系。因此，这个循环是可并行化的。也就是说，这个循环的各个迭代独立于其他所有的迭代，我们可以并行地执行这些迭代，或者按照我们选择的任意顺序执行这些迭代。但是请注意，如果我们做一些细微的改动，比如把循环下标  $i$  的上界改成 10 或者更大，那么就会存在依赖关系。因为数组  $Z$  的某些元素会在一个迭代中被写，然后在 10 个迭代之后被读。在那种情况下，这个循环不能被完全地并行化，我们将不得不仔细考虑如何在处理器之间分配迭代，以及如何对迭代进行排序。□

把并行化问题写成多维空间和这些空间之间的仿射映射使得我们可以使用标准的数学技术来解决并行化和局部性优化问题。比如，可以通过使用 Fourier-Motzkin 消除算法消除相应的变量，找出被访问数据的区域。已经证明数据依赖性和整数线性规划问题等价。最后，寻找仿射分划的问题则对应于对一组线性约束求解。如果你不熟悉这些概念也不用着急，因为从 11.3 节开始将对这些概念进行解释。

## 11.2 矩阵乘法：一个深入的例子

我们将使用一个较大的例子来介绍并行编译器所使用的很多技术。在本节中，我们将研究大家熟悉的矩阵相乘算法，以说明即使对一个简单且易于并行化的程序进行优化也不是轻而易举的事。我们将看到代码改写可以如何提高数据局部性。也就是说，和选择直接运行该程序相比，处理器只需要通过少得多的（根据不同的体系结构，和全局内存或其他处理器之间的）通信量就可以完成它们的工作。我们也将讨论如何利用可以存放多个连续数据元素的高速缓存线的特性来改进像矩阵乘法这样的程序的运行时间。

### 11.2.1 矩阵相乘算法

在图 11-5 中，我们看到一个典型的矩阵乘法程序<sup>⊖</sup>。它的输入是两个  $n \times n$  的矩阵  $X$  和  $Y$ ，它产生的输出存放在第三个  $n \times n$  矩阵  $Z$  中。注意， $Z_{ij}$ ，即矩阵  $Z$  在第  $i$  行第  $j$  列上的元素将变成  $\sum_{k=1}^n X_{ik} Y_{kj}$ 。

图 11-5 中的代码生成  $n^2$  个结果，每个结果都是矩阵  $X$  的某行和矩阵  $Y$  的某列的内积。显然，矩阵  $Z$  的每一个元素的计算都是独立的，因此可

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        Z[i,j] = 0.0;
        for (k = 0; k < n; k++)
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
    }
```

图 11-5 基本的矩阵相乘算法

<sup>⊖</sup> 在本章中的程序中，我们通常将使用 C 语言的语法，但是为了使得多维数组访问（这是本章大部分内容的中心问题）的代码更加易于理解，我们将使用 Fortran 风格的数组访问，也就是说，使用  $Z[i,j]$  而不是  $Z[i][j]$ 。

以并行执行。

$n$  的值越大，算法访问各个元素的次数就越多。也就是说，这三个矩阵中有  $3n^2$  个位置，但是这个算法执行  $n^3$  次运算，每次运算把  $X$  的一个元素和  $Y$  的一个元素相乘并把乘积加到  $Z$  的一个元素中。因此算法是计算密集型的，从原则上来讲内存访问不应该成为瓶颈。

### 矩阵乘法的串行执行

我们首先考虑这个程序在单处理器上顺序运行时是如何工作的。最内层循环读写  $Z$  的同一个元素，并使用  $X$  的一行和  $Y$  的一列。可以很容易地把  $Z[i,j]$  放到一个寄存器中，不需要进行内存访问。不失一般性，假设这个矩阵是按行存放的，并假设  $c$  是高速缓存线中的数组元素的个数。

图 11-6 给出了当我们执行图 11-5 中的外层循环的一个迭代时的访问模式。这张图显示的是第一个迭代的情况，此时  $i=0$ 。每次我们从  $X$

的第一行的一个元素移动到下一个元素时，都会访问  $Y$  的某一列中的各个元素。在图 11-6 中可以看到假设的将各个矩阵组织成高速缓存线的方法。也就是说，每个小矩形表示了一个存放四个数组元素的高速缓存线（即在此图中， $c=4$  且  $n=12$ ）。

对  $X$  的访问几乎没有增加高速缓存的负担。 $X$  的一行只分布在  $n/c$  个高速缓存线中。

如果这一行元素可以被一起放到高速缓存中，对于一个给定的下标  $i$  的值只会发生  $n/c$  次高速缓存脱靶，而整个  $X$  的脱靶数量在最少情况下为  $n^2/c$ （为方便起见，我们假设  $n$  可以被  $c$  整除）。

但是，当使用  $X$  的一行时，该矩阵相乘算法逐列访问  $Y$  的所有元素。也就是说，当  $j=0$  时，内层循环把  $Y$  的整个第一列都搬到了高速缓存中。请注意，该列的所有元素存放在  $n$  个不同的高速缓存线中。如果高速缓存大到（或者  $n$  小到）可以存放所有这  $n$  个高速缓存线，并且没有对高速缓存的其他使用使得某些高速缓存线被清除出高速缓存，那么当需要  $Y$  的第二列时，对应于  $j=0$  的列仍然在高速缓存中。在此情况下，在  $j=c$  之前就不会产生  $n$  次对  $Y$  的脱靶。当  $j=c$  时，我们需要把对应于  $Y$  的另一组完全不同的高速缓存线载入到高速缓存中。因此，完成外层循环的第一个迭代（即  $i=0$ ）所碰到的高速缓存脱靶次数在  $n^2/c$  到  $n^2$  之间。具体的脱靶次数取决于  $Y$  的高速缓存线列能否从第二个循环的一次迭代存活到下一个迭代。

不仅如此，当我们完成  $i=1, 2, \dots$  外层循环时，在读取  $Y$  的元素时可能会碰到更多的高速缓存脱靶，也可能完全没有脱靶。如果高速缓存大到可以把  $Y$  的所有  $n^2/c$  个高速缓存线一起存放在高速缓存中，那么就不会再碰到任何脱靶。因此，整个脱靶次数是  $2n^2/c$ ，一半在访问  $X$  时发生，另一半在访问  $Y$  时发生。但是，如果目标机器的高速缓存只能存放  $Y$  的一列，而不是全部  $Y$ ，那么每次执行外层循环的一个迭代时，我们需要把  $Y$  的所有元素再次载入到高速缓存中。也就是说，高速缓存脱靶的次数是  $n^2/c + n^3/c$ ，其中的第一项是访问  $X$  时的脱靶次数，而第二项是访问  $Y$  时的脱靶次数。在最坏情况下，如果我们甚至不能把  $Y$  的一列一起存放在高速缓存中，那么外层循环的每次迭代都有  $n^2$  个高速缓存脱靶，总计有  $n^2/c + n^3$  次脱靶。

### 逐行并行化

现在我们考虑可以如何使用多个处理器（比如说  $p$  个）来加快图 11-5 中程序的执行。一个很显然的并行化矩阵乘法的方法是把  $Z$  的各行分配给不同的处理器。每个处理器负责  $n/p$  个连续

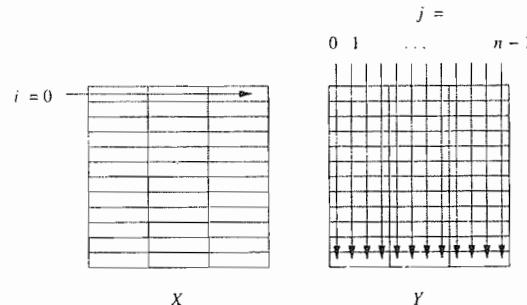


图 11-6 在矩阵乘法中的数据访问模式

的行(为方便起见,我们假设  $n$  可以被  $p$  整除)。通过这样分配工作量,每个处理器只需要访问矩阵  $X$  和  $Z$  的  $n/p$  行,但是要访问整个  $Y$  矩阵。每个处理器将计算  $Z$  的  $n^2/p$  个元素。为了计算得到这些元素,它需要进行  $n^3/p$  次乘 - 加法运算。

这样做之后,虽然计算时间减少和  $p$  成正比,但通信开销的增长实际上和  $p$  成正比。也就是说,每个  $p$  处理器必须读取  $n^2/p$  个  $X$  的元素,但是要读取  $Y$  的所有  $n^2$  个元素。必须被加载到这  $p$  个处理器的高速缓存中的高速缓存线的总数最少为  $n^2/c + pn^3/c$ ,这两项分别对应于加载  $X$  和加载  $Y$  副本的数量。当  $p$  的值趋近于  $n$  时,计算时间变为  $O(n^2)$ ,但是通信开销为  $O(n^3)$ 。也就是说,在内存和处理器高速缓存之间移动数据的总线成为性能瓶颈。因此,对于给定的数据布局而言,使用大量的处理器来平分计算量实际上会降低计算速度,而不是加快计算速度。

### 11.2.2 优化

图 11-5 中的矩阵相乘算法说明了这样的事实:即使一个算法可能复用同样的数据,它的数据局部性仍然可能很差。要使得一次数据复用能够在高速缓存中命中,它就必须在该数据被转移出高速缓存之前发生。在本例中,  $n^2$  个乘 - 加法把对矩阵  $Y$  中同一个元素的复用分开了,因此数据局部性变得很差。实际上,  $n$  次运算把对  $Y$  中同一高速缓存线内的数据的复用分开。另外,在一个多处理器系统中,只有当数据被同一个处理器复用时才可能发生高速缓存命中事件。当我们在 11.2.1 节中考虑一个并行实现时,我们看到  $Y$  的元素必须被每一个处理器使用。因此,对  $Y$  的复用并没有转化为局部性。

#### 改变数据布局

改善一个程序的局部性的方法之一是改变它的数据结构的布局。比如,把  $Y$  按列存放将提高对矩阵  $Y$  的高速缓存线的复用。这个方法的应用范围是有限的,因为同一个矩阵通常会在不同的运算中使用。如果在另一个矩阵乘法运算中  $Y$  扮演了  $X$  的角色,那么又会因为按列存放而损害效率,因为在乘法运算中的第一个矩阵按行存放比较好。

#### 分块

有时可以通过改变指令执行的顺序来改善数据局部性。但是循环互换的技术并不能提高矩阵乘法例程的效率。假设把这个例程改写成每次生成矩阵  $Z$  的一列,而不是一行。也就是说,把  $j$  循环变成外层循环而  $i$  循环变成内层循环。假设这些矩阵仍旧按行存放,矩阵  $Y$  就有比较好的空间局部性和时间局部性,但会损害矩阵  $X$  的局部性。

分块(blocking)是另一种对循环中的迭代重新排序的方法,它可以大大提高一个程序的局部性。我们不再一次计算结果矩阵的一行或者一列,而是按照图 11-7 中所示把矩阵分割成为子矩阵,或者说块。然后,我们对运算重新排序,使得整个块只在一小段时间内使用。通常情况下,这些块是边长为  $B$  的正方形。如果  $B$  整除  $n$ ,那么所有的块都是正方形。如果  $B$  不能整除  $n$ ,那么在矩阵下面或右面的边上的块的一条或者两条边的长度小于  $B$ 。

图 11-8 显示了基本矩阵相乘算法的一个版本。在这个版本中,全部三个矩阵被划分为边长为  $B$  的正方形。和图 11-5 中一样,假设  $Z$  已经被初始化为全 0 矩阵。我们假设  $B$  整除  $n$ ,如果不是这样的话,就需要把第(4)行中的上限修改为  $\min(ii + B, n)$ ,并对第 5 和 6 行做类似的修改。

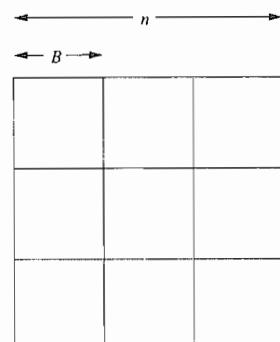


图 11-7 一个被分割成边长为  $B$  的块的矩阵

```

1) for (ii = 0; ii < n; ii = ii+B)
2)   for (jj = 0; jj < n; jj = jj+B)
3)     for (kk = 0; kk < n; kk = kk+B)
4)       for (i = ii; i < ii+B; i++)
5)         for (j = jj; j < jj+B; j++)
6)           for (k = kk; k < kk+B; k++)
7)             Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];

```

图 11-8 基于分块技术的矩阵乘法

外面的三层循环，即第 1 行到第 3 行，使用下标变量  $ii$ 、 $jj$  和  $kk$ 。这些变量的增量总是  $B$ ，因此它们总是表示了某个块的左面或上面的边。对于固定的  $ii$ 、 $jj$ 、 $kk$  的值，第 4 行到第 7 行计算了以  $X[ii, kk]$  和  $Y[kk, jj]$  为左上角的两个块的乘积并加到以  $Z[ii, jj]$  为左上角的块中去。

当不能够把  $X$ 、 $Y$ 、 $Z$  一起放到高速缓存中时，如果我们选择了适当的  $B$  值，和基本算法相比，我们可以明显地降低高速缓存脱靶的数量。选择  $B$  使得可以把每个矩阵的各个块一起放到缓存中去。因为上面的算法中各个循环的顺序，我们实际上只需要把  $Z$  中的每个块放到高速缓存中一次就可以了。因此（像我们在 11.2.1 节中对基本算法所作的分析那样）我们将不需要计算因为  $Z$  而产生的高速缓存脱靶。

把  $X$  或  $Y$  的一个块载入到高速缓存需要  $B^2/c$  次高速缓存脱靶。请记住， $c$  是一个高速缓存线中的元素的个数。但是，对于确定的分别来自  $X$  和  $Y$  的块，我们在图 11-8 的第 4 行到第 7 行中进行了  $B^3$  次乘 - 加法运算。因为整个矩阵乘法需要  $n^3$  次乘 - 加法运算，所以把两个块加载进缓存的次数是  $n^3/B^3$ 。因为每次加载一个块时会碰到  $2B^2/c$  次高速缓存脱靶，所以总的缓存脱靶数量是  $2n^3/Bc$ 。

把这个数字  $2n^3/Bc$  和 11.2.1 节中给出的估计值相比是很有意思的。在那一节中，我们说，如果整个矩阵可以一起放到高速缓存中的话，就将出现  $O(n^2/c)$  次高速缓存脱靶。然而，在那种情况下，我们可以令  $B=n$ ，即把每个矩阵当成一个块。我们仍然可以得到前面估算的  $O(n^2/c)$  次高速缓存脱靶。另一方面，我们观察到如果不能把整个矩阵放到高速缓存中，就需要  $O(n^3/c)$  次高速缓存脱靶，甚至  $O(n^3)$  次脱靶。在这种情况下，假设我们可以选择相当大的  $B$  值（比如  $B$  可以是 200，此时仍然可以把三个 8 字节数字的块放到一个 1 MB 的高速缓存中），在矩阵乘法中使用分块技术有很大的优越性。

分块技术可以被应用到内存层次结构的各个层次上。比如，我们可能希望通过把一个  $2 \times 2$  矩阵乘法的运算分量都放到寄存器中，以优化对寄存器的使用。对于不同层次的高速缓存和物理内存，我们使用逐渐增大的分块尺寸。

类似地，我们可以把各个块分布到不同的处理器上，以便使数据流量达到最小。实验显示，这样的优化在单处理器情况下的性能加速比可以达到 3，而在多处理器系统上的加速比几乎和所使用的处理器数量成线性关系。

### 基于块的矩阵乘法的另一个视点

我们可以想象图 11-8 中的矩阵  $X$ 、 $Y$ 、 $Z$  并不是  $n \times n$  的浮点数的矩阵，而是  $(n/B) \times (n/B)$  的矩阵，而这个矩阵的元素本身又是  $B \times B$  的浮点数的矩阵。那么，图 11-8 中的第 1 行到第 3 行就像是图 11-5 中的基本算法的三个循环，但是它们处理的矩阵的大小是  $n/B$ ，而不是  $n$ 。我们可以把图 11-8 的第 4 行到第 7 行看作是图 11-5 中的单个乘 - 加法运算的实现。请注意，在这个运算中的单个乘法步骤对应于一个矩阵乘法步骤，使用了图 11-5 中对元素为浮点数的两个矩阵相乘的基本算法。矩阵加法就是各个元素上的浮点数加法。

### 11.2.3 高速缓存干扰

遗憾的是，对于高速缓存的利用还有一些问题要解决。大部分高速缓存不是完全结合的（见 7.4.2 节）。在一个直接映射的高速缓存中，如果  $n$  是高速缓存大小的倍数，那么一个  $n \times n$  的矩阵的同一行中的各个元素将竞争同样的高速缓存位置。在那种情况下，把某列的第二个元素加载进高速缓存将会把第一个元素的高速缓存线挤出高速缓存。即使高速缓存有能力同时存放所有这些高速缓存线，这样的情况仍然会发生。这种情况被称为高速缓存干扰(cache interference)。

对于这个问题有多种解决方法。第一个方法是一劳永逸地重新排列数据，使得被访问的数据放在连续的数据位置上。第二个方法是把  $n \times n$  的数组放在一个较大的  $m \times n$  的数组中，我们可以选择适当的  $m$  来最小化干扰问题。第三种方法是选择一个可以保证避免干扰的分块大小。

### 11.2.4 11.2 节的练习

练习 11.2.1：和图 11-5 中的代码不同，图 11-8 中的基于块的矩阵相乘算法中不包含把矩阵  $Z$  的所有元素清零的初始化部分。在图 11-8 中加入把  $Z$  初始化为全零矩阵的步骤。

## 11.3 迭代空间

本节的研究动机是充分利用 11.2 节中提到的优化技术。对于一些简单情况，比如矩阵乘法，这些技术是相当简单明了的。对于更加一般的情况，同样的技术仍然可用，但此时它们的应用就远没有那么直观了。然而，通过应用一些线性代数技术，我们可以使得这些技术能够完成对一般情况的优化。

11.1.5 节中讨论过，我们的转换模型中有三种空间：迭代空间、数据空间和处理器空间。这里我们首先从迭代空间开始。一个循环嵌套结构的迭代空间被定义为该嵌套结构中所有循环下标变量取值的组合。

和图 11-5 中的矩阵乘法的例子一样，迭代空间常常是矩形的。在那种情况下，每个嵌套中的循环具有下界 0 和上界  $n - 1$ 。但是，在更复杂但相当现实的循环嵌套结构中，一个循环下标的上界和下界可能依赖于较外层循环的下标值。我们很快会看到这样的一个例子。

### 11.3.1 从循环嵌套结构中构建迭代空间

让我们首先描述一下即将学习的技术能够处理哪些类型的循环嵌套结构。每个循环有一个唯一的循环下标，我们假设每次迭代这个下标增加 1。这个假设并没有失去一般性，因为如果每次迭代的增量是大于 1 的整数  $c$ ，那么总是可以把对下标  $i$  的使用替代为  $ci + a$ ，其中  $a$  是某个正或负的常数，然后循环中的每次迭代将  $i$  加 1。这个循环的上下界必须写成其外层循环的下标的仿射表达式。

#### 例 11.5 考虑下面的循环

```
for (i = 2; i <= 100; i = i+3)
    Z[i] = 0;
```

该循环的每轮运行把循环下标  $i$  加 3，它的效果是把各个数组元素  $Z[2]$ ,  $Z[5]$ ,  $Z[8]$ , ...,  $Z[98]$  设置为 0。我们可以使用下面的循环来得到同样的效果：

```
for (j = 0; j <= 32; j++)
    Z[3*j+2] = 0;
```

也就是说，我们用  $3j + 2$  替代了  $i$ 。下界  $i = 2$  变成了  $j = 0$ （只需要求解方程  $3j + 2 = 2$  就可得到  $j$  的值），而上界  $i \leq 100$  变成了  $j \leq 32$ （将  $3j + 2 \leq 100$  简化可得  $j \leq 32.67$ ，又因为  $j$  必须是整数，所以要舍弃小数部分）。□

通常情况下，我们将在循环嵌套结构中使用 for 循环结构。对于一个 while 循环或者 repeat 循环，如果存在一个下标以及该下标的上下界，那么它就可以被替换为一个 for 循环。图 11-9a 中的循环就是这样的情况。一个像 `for (i = 0; i < 100; i++)` 这样的 for 循环可以达到同样的目标。

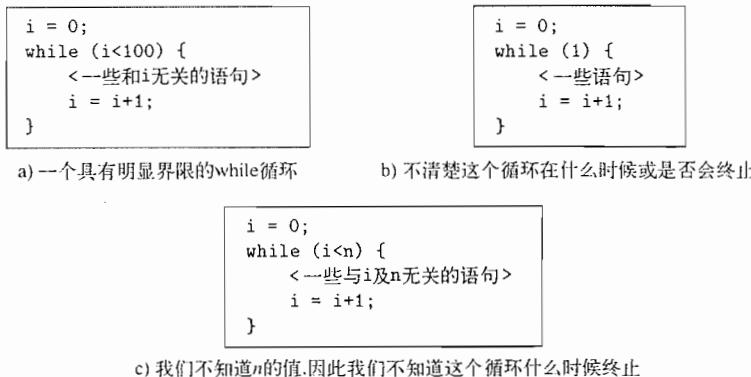


图 11-9 一些 while 循环

但是有些 while 循环或 repeat 循环没有明显的界限。比如，图 11-9b 中的例子可能会中止，也可能不会终止，但是没有办法指出在未知的循环体中  $i$  满足什么条件时程序会跳出该循环。图 11-9c 是另一个会出现问题的情况。例如，变量  $n$  可能是一个函数的参数。我们知道循环迭代  $n$  次，但是在编译时刻我们不知道  $n$  的值是什么。实际上，我们可能期望该循环在不同的执行中迭代的次数不同。在图 11-9b 和图 11-9c 这样的情况下，我们必须把  $i$  的上界当作无限来处理。

一个深度为  $d$  的循环嵌套结构可以被建模为一个  $d$  维空间。空间的各个维是有序的，第  $k$  维表示该嵌套结构中从最外层循环起的第  $k$  个循环。这个空间中的一个点  $(x_1, x_2, \dots, x_d)$  表示所有这些循环下标的值，最外层循环下标的值是  $x_1$ ，第二个循环下标的值是  $x_2$ ，以此类推。最内层循环下标的值是  $x_d$ 。

但并不是这个空间中的每个点都代表了一个在该循环嵌套结构执行时实际出现的下标取值组合。作为外层循环下标的一个仿射函数，每个循环的上下界都定义了一个不等式，它把空间分成两半：对应于循环迭代的部分（即正的半空间）和不对应于迭代的部分（即负的半空间）。所有线性不等式的交（逻辑 AND）表示这些正的半空间的交集，该交集定义了一个凸多面体（convex polyhedron）。我们把这个多面体称为这个循环嵌套结构的迭代空间（iteration space）。一个凸多面体具有以下性质：如果两个点在该多面体内，那么它们之间的连线上的所有点都在该多面体内。多面体使用循环界限不等式描述。循环的每个迭代都可以由该多面体中的具有整数坐标的点表示。反过来，在多面体内的每个整数点都代表了该循环嵌套结构在某个时候执行的一个迭代。

**例 11.6** 考虑图 11-10 中的二维循环嵌套结构。我们可以使用图 11-11 中显示的二维多面体对这个深度为 2 的循环嵌套结构建模。图中的两个轴表示循环下标  $i$  和  $j$  的值。下标  $i$  可以取  $0 \sim 5$  之间的任何整数值；下标  $j$  可以取满足  $i \leq j \leq 7$  的任何整数值。

```

for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
  
```

□ 图 11-10 一个二维循环嵌套结构

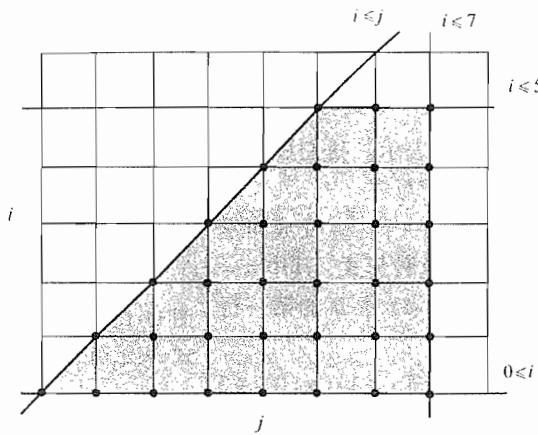


图 11-11 例 11.6 的迭代空间

### 迭代空间和数组访问

在图 11-10 的代码中，迭代空间也是数组  $Z$  中被该代码访问到的部分。这种类型的访问是很常见的，它们的数组下标就是按某种顺序排列的循环下标。但是，我们不应该把迭代空间和数据空间相混淆。迭代空间的各个维度是各循环下标。假设我们在图 11-10 的代码中使用一个类似于  $Z[2 * i, i + j]$  的数组访问来替换  $Z[j, i]$ ，那么迭代空间和数据空间之间的区别就很明显了。

### 11.3.2 循环嵌套结构的执行顺序

一个循环嵌套结构的顺序执行按照上升的词典顺序逐个执行它的迭代空间中的各个迭代。一个向量  $i = [i_0, i_1, \dots, i_n]$  按照词典排序小于另一个向量  $i' = [i'_0, i'_1, \dots, i'_n]$ ，记为  $i < i'$ ，当且仅当存在一个  $m < \min(n, n')$  使得  $[i_0, i_1, \dots, i_m] = [i'_0, i'_1, \dots, i'_m]$  并且  $i_{m+1} < i'_{m+1}$ 。请注意， $m$  可以等于 0，实际上这种情况很常见。

**例 11.7** 把  $i$  当作外层循环，例 11.6 中的循环嵌套结构的迭代按照图 11-12 所示的顺序执行。

□

[0, 0],	[0, 1],	[0, 2],	[0, 3],	[0, 4],	[0, 5],	[0, 6],	[0, 7]
[1, 1],	[1, 2],	[1, 3],	[1, 4],	[1, 5],	[1, 6],	[1, 7]	
[2, 2],	[2, 3],	[2, 4],	[2, 5],	[2, 6],	[2, 7]		
[3, 3],	[3, 4],	[3, 5],	[3, 6],	[3, 7]			
[4, 4],	[4, 5],	[4, 6],	[4, 7]				
[5, 5],	[5, 6],	[5, 7]					

图 11-12 图 11-10 中的循环嵌套的迭代的执行顺序

### 11.3.3 不等式组的矩阵表示方法

在一个深度为  $d$  的循环嵌套中的迭代可以用数学方式表示为

$$\{i \text{ 在 } Z^d \text{ 中 } |Bi + b \geq 0|\} \quad (11.1)$$

其中：

- 1)  $Z$ （按照数学惯例）表示整数的集合——包括正整数、负整数和零。
- 2)  $B$  是一个  $d \times d$  的整数矩阵。

- 3)  $b$  是一个长度为  $d$  的整数向量。  
 4)  $\theta$  是一个由  $d$  个零组成的向量。

**例 11.8** 我们可以把例 11.6 中的不等式写成图 11-13 中的形式。也就是说， $i$  的范围用  $i \geq 0$  和  $i \leq 5$  表示； $j$  的范围用  $j \geq i$  和  $j \leq 7$  表示。我们要求这些不等式都能写成  $ui + vj + w \geq 0$  的形式。然后， $[u, v]$  变成了不等式(11.1)中矩阵  $B$  的一行， $w$  变成向量  $b$  中的相应元素。比如， $i \geq 0$  就是这种形式，其中  $u = 1, v = 0, w = 0$ 。这个不等式用图 11-13 中  $B$  的第一行和  $b$  的顶端元素表示。

看另一个例子，不等式  $i \leq 5$  等价于  $(-1)i + (0)j + 5 \geq 0$ ，它由图 11-13 中  $B$  和  $b$  的第二行表示。另外， $j \geq i$  变成了  $(-1)i + (1)j + 0 \geq 0$ ，由第三行表示。最后， $j \leq 7$  变成  $(0)i + (-1)j + 7 \geq 0$ ，由图中矩阵和向量的最后一行表示。□

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

图 11-13 矩阵向量乘法和一个向量的不等式，表示用于定义一个迭代空间的不等式组

### 处理不等式

为了像例 11.8 中那样转换不等式，我们可以像处理等式一样进行转换。比如，在不等式两边都增加或减少同样的值，或将两边都乘以同样的常量。我们必须记住的唯一特殊规则是，当我们把不等式两边都乘以一个负数的时候，必须改变不等号的方向。因此， $i \leq 5$  乘以  $-1$  就变成  $-i \geq -5$ 。给不等式两边都加上 5 得到  $-i + 5 \geq 0$ ，实际上就是图 11-13 的第二行。

### 11.3.4 混合使用符号常量

有时我们需要对涉及某些变量的循环嵌套结构进行优化，这些变量对于该嵌套中的所有循环都是循环不变的。我们把这样的变量称为符号常量 (symbolic constant)，但是为了描述一个迭代空间的边界，我们需要把它们当作变量进行处理，并在循环下标变量组成的向量中为它们创建一个条目。这个向量就是通用不等式(11.1)中的向量  $i$ 。

**例 11.9** 考虑下面的简单循环：

```
for (i = 0; i <= n; i++) {
    ...
}
```

这个循环定义了一个一维的迭代空间，下标变量是  $i$ ，界限是  $i \geq 0$  和  $i \leq n$ 。因为  $n$  是一个符号常量，我们需要把它当作一个变量包括进来，得到一个循环下标的向量  $[i, n]$ 。按照矩阵 - 向量的形式，这个迭代空间被定义为

$$\left\{ i \text{ 在 } Z \text{ 中 } \mid \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

请注意，虽然数组下标的向量具有两个维度，但它们中只有表示  $i$  的第一维是输出部分，即迭代空间的点集。

### 11.3.5 控制执行的顺序

从一个循环体的上下界中抽取到的线性不等式定义了一个凸多面体上的迭代的集合。这个表示方法并没有假定在迭代空间中的迭代之间的任何执行顺序。原程序在迭代之上强加了一个串行顺序，该顺序就是按照从外到内方式排列的循环下标变量取值的词典排序。但是，只要遵守它们之间的数据依赖关系（即循环嵌套结构中不同赋值语句所执行的对任一数组元素的写/读操

作的顺序没有改变），就可以按照任何顺序执行该空间中的迭代。

如何选择一个既遵守数据依赖关系，又能优化数据局部性和并行性的顺序是很困难的问题，我们稍后将从 11.7 节开始处理这个问题。这里我们假设已经有了一个合法且令人满意的排序，说明如何生成遵守这个顺序的代码。首先让我们讨论例 11.6 中的另一个排序。

**例 11.10** 在例 11.6 的程序中，迭代之间没有数据依赖关系。因此，可以用串行或者并行的方式执行这些迭代。因为在此代码中迭代  $[i,j]$  访问了元素  $Z[j,i]$ ，原程序按照图 11-14a 中的顺序访问数组。为了提高空间局部性，我们更愿意像图 11-14b 所显示的那样连续地访问数组中的邻近元素。

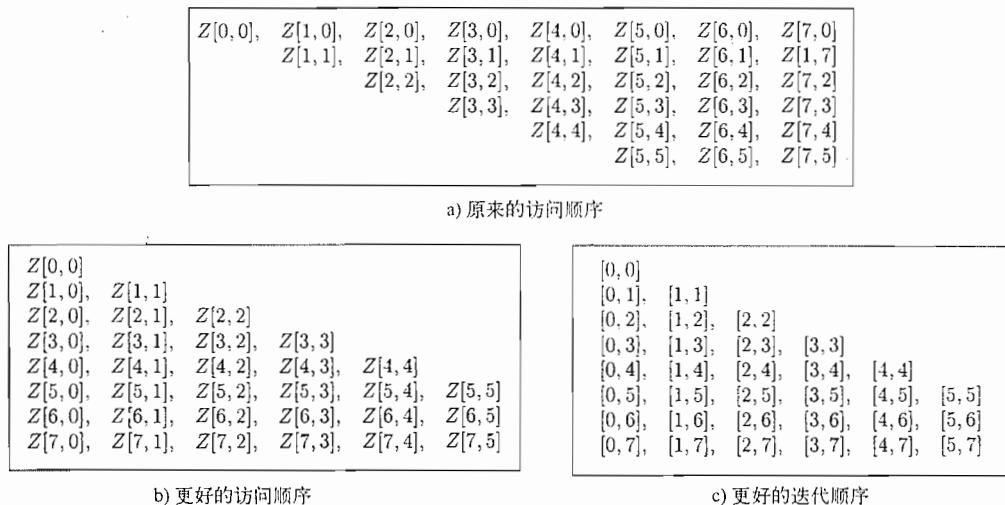


图 11-14 一个循环嵌套结构的访问和迭代的重新排序

如果我们按照图 11-14c 中的顺序执行循环的迭代，就能够得到上面的访问模式。也就是说，我们垂直地（而不是水平地）扫描图 11-11 中的迭代空间，因此  $j$  变成了外层循环的下标。按照上面的顺序执行这些迭代的代码是

```
for (j = 0; j <= 7; j++)
    for (i = 0; i <= min(5,j); i++)
        Z[j,i] = 0;
```

□

给定一个凸多面体和一个下标变量的排序，我们该如何生成循环的界限，使得循环能够按照这些变量的词典排序扫描这个迭代空间？在上面的例子中，约束  $i \leq j$  在原程序中是作为内层循环下标  $j$  的下界出现的，但是在转换得到的程序中它作为内层循环下标  $i$  的上界出现。

最外层循环的界限是用符号常量和常量的线性组合来表示的，它定义了该循环下标的全部取值的范围。内层循环的下标变量的界限用较外层循环的下标变量、符号常量和常量的线性组合来表示。对于给定的较外层循环下标变量的每个取值组合，它们定义了该循环下标变量的取值范围。

### 投影

从几何学的角度来讲，我们可以把表示迭代空间的凸多面体投影（projection）到该空间中对应于较外层循环的维度上，以得到一个深度为 2 的循环嵌套结构中外层循环下标变量的循环界限。直观地讲，一个多面体在一个较低维度空间的投影就是该物体在这个空间中的影子。图 11-11 中的二维迭代空间到  $i$  轴上的投影是从 0 到 5 的垂直线；而到  $j$  轴的投影是从 0 到 7 的水平

线。当我们把一个 3 维物体沿着  $z$  轴投影到一个二维的  $x - y$  的平面上时，我们消除变量  $z$ ，失去了各个点的高度信息，而仅仅记录下该物体在  $x - y$  平面上的二维覆盖区域。

循环界限生成只是投影的多种用途之一。投影的正式定义如下。令  $S$  为一个  $n$  维多面体。 $S$  到它的前  $m$  个维度的投影是满足如下条件的点  $(x_1, x_2, \dots, x_m)$ ：存在  $x_{m+1}, x_{m+2}, \dots, x_n$  使得向量  $[x_1, x_2, \dots, x_n]$  在  $S$  中。我们可以使用 Fourier-Motzkin 消除算法来计算投影。下面介绍该算法。

### 算法 11.11 Fourier-Motzkin 消除算法。

**输入：**一个带有变量  $x_1, x_2, \dots, x_n$  的多面体  $S$ 。也就是说， $S$  是关于变量  $x_i$  的一组线性约束。一个给定的变量  $x_m$  是被指定需要消除的变量。

**输出：**一个关于变量  $x_1, x_2, \dots, x_{m-1}, x_{m+1}, \dots, x_n$  (即除  $x_m$  之外的所有  $S$  的变量) 的多面体  $S'$ 。 $S'$  是  $S$  到除第  $m$  个维度之外的所有维度的投影。

**方法：**令  $C$  是  $S$  中所有涉及  $x_m$  的约束的集合。执行下列步骤：

1) 对于  $C$  中关于  $x_m$  的每一对上界和下界，比如

$$\begin{aligned} L &\leq c_1 x_m \\ c_2 x_m &\leq U \end{aligned}$$

建立一个新的约束

$$c_2 L \leq c_1 U$$

请注意， $c_1$  和  $c_2$  是整数，但  $L$  和  $U$  可能是关于除  $x_m$  之外的其他变量的表达式。

2) 如果整数  $c_1$  和  $c_2$  有公因子，将上面约束的两边都除以这个因子。

3) 如果新的约束是不可满足的，那么  $S$  无解，即多面体  $S$  和  $S'$  都是空的空间。

4)  $S'$  是约束集合  $S - C$  加上在第 2 步中生成的所有约束。

顺便说一下，如果  $x_m$  具有  $u$  个下界和  $v$  个上界，消除  $x_m$  最多会产生  $uv$  个不等式。  $\square$

在算法 11.11 的第一步中引入的约束对应于约束集合  $C$  所蕴涵的对系统中其余变量的约束。因此， $S'$  中有一个解的充要条件是  $S$  中至少有一个对应的解。给定  $S'$  中的一个解，把约束集合  $C$  中除了  $x_m$  之外的所有变量替换为它们在这个解中的实际取值，就可以得到  $x_m$  的取值范围。

**例 11.12** 考虑定义了图 11-11 中的迭代空间的不等式组。假设我们希望使用 Fourier-Motzkin 消除算法来消除  $i$  维度，从而把这个二维空间投影到  $j$  维度上。存在一个  $i$  的下界  $0 \leq i$  和两个上界  $i \leq j$  和  $i \leq 5$ 。这可以生成两个约束  $0 \leq j$  和  $0 \leq 5$ 。后一个约束是永真式，可以忽略。前一个约束给出了  $j$  的下界， $j$  的上界就是原来的上界  $j \leq 7$ 。  $\square$

### 循环界限的生成

既然我们已经定义了 Fourier-Motzkin 消除算法，生成循环界限来遍历一个凸多面体的算法（算法 11.13）就很容易得到了。我们按照从最内层到最外层循环的顺序计算循环界限。所有涉及最内层循环下标变量的不等式都被改写成为该变量的下上界的形式。然后，我们通过投影消除代表了最内层循环的维度，得到减少了一个维度的多面体。我们重复这个过程，直到找出所有循环下标变量的界限。

### 算法 11.13 给定一组变量的顺序，计算这些变量的界限。

**输入：**一个在变量  $v_1, v_2, \dots, v_n$  之上的凸多面体  $S$ 。

**输出：**每个变量  $v_i$  的下界  $L_i$  和上界  $U_i$ ，这些界限只使用排在  $v_i$  之前的变量  $v_j (j < i)$  来表示。

**方法：**该算法在图 11-15 中描述。  $\square$

```

 $S_n = S; /* 使用算法 11.11 来计算循环界限 */$ 
 $\text{for } (i = n; i \geq 1; i--) \{$ 
     $L_{v_i} = S_i$  中  $v_i$  的所有下界;
     $U_{v_i} = S_i$  中  $v_i$  的所有上界;
     $S_{i-1}$  = 将算法 11.11 应用于消除约束集合  $S_i$  中的  $v_i$  后得到
        的约束集合;
}
/* 消除冗余性 */
 $S' = \emptyset;$ 
 $\text{for } (i = 1; i \leq n; i++) \{$ 
    消除所有由  $S'$  蕴涵的  $L_{v_i}$  和  $U_{v_i}$  中的界限;
    把  $L_{v_i}$  和  $U_{v_i}$  中其余关于  $v_i$  的约束加到  $S'$  中。
}

```

图 11-15 按照一个给定的变量顺序表示变量界限的代码

**例 11.14** 我们应用算法 11.13 来生成用于垂直扫描图 11-11 中的迭代空间的循环界限。下标变量的顺序是  $j, i$ 。该算法生成了如下的界限:

$$\begin{aligned} L_i &: 0 \\ U_i &: 5, j \\ L_j &: 0 \\ U_j &: 7 \end{aligned}$$

我们要满足所有这些约束，因此  $i$  的上界是  $\min(5, j)$ 。这个例子中没有冗余的界限。  $\square$

### 11.3.6 坐标轴的变换

请注意，上面讨论的对迭代空间进行水平扫描或垂直扫描只是两种最常见的访问迭代空间的方法。还有很多其他的可行方法，比如，我们可以按照逐条斜线的方式来扫描例 11.6 中的迭代空间。下面的例 11.15 就讨论这种扫描方法。

**例 11.15** 我们可以按照逐条斜线的方式来扫描图 11-11 中的迭代空间，使用的顺序如图 11-16 所示。每条斜线上的点的坐标  $j$  和  $i$  之间的差值是一个常量。开始的时候这个差值是 0，而结束的时候是 7。因此，我们定义一个新的变量  $k = j - i$ ，并按照针对  $k, j$  的词典顺序来扫描上面的迭代空间。在不等式中用  $j - k$  替代  $i$ ，我们得到：

$$\begin{aligned} 0 &\leq j - k \leq 5 \\ j - k &\leq j \leq 7 \end{aligned}$$

[0, 0],	[1, 1],	[2, 2],	[3, 3],	[4, 4],	[5, 5]
[0, 1],	[1, 2],	[2, 3],	[3, 4],	[4, 5],	[5, 6]
[0, 2],	[1, 3],	[2, 4],	[3, 5],	[4, 6],	[5, 7]
[0, 3],	[1, 4],	[2, 5],	[3, 6],	[4, 7]	
[0, 4],	[1, 5],	[2, 6],	[3, 7]		
[0, 5],	[1, 6],	[2, 7]			
[0, 6],	[1, 7]				
[0, 7]					

要为上面描述的顺序建立循环界限，可以对上面的图 11-16 图 11-11 的迭代空间的斜向排序不等式集合按照变量顺序  $k, j$  应用算法 11.13，得到

$$\begin{aligned} L_j &: k \\ U_j &: 5 + k, 7 \\ L_k &: 0 \\ U_k &: 7 \end{aligned}$$

根据这些不等式，我们可以生成下列代码。在访问数组的时候， $i$  被替换为  $j - k$ 。

```

for (k = 0; k <= 7; k++)
    for (j = k; j <= min(5+k, 7); j++)
        Z[j, j-k] = 0;

```

$\square$

一般来说，我们可以通过创建新的循环下标变量并定义这些变量的顺序，从而改变一个多面体的坐标轴。这些新的循环下标变量表示了原来变量的仿射组合。这个问题的难点在于如何选择适当的坐标轴，使得在满足数据依赖关系的同时达到并行性和局部性目标。我们将从 11.7 节开始讨论这个问题。在这里，我们的结果表明一旦选择好坐标轴，就可以像例 11.15 所示那样直接生成想要的代码。

还有很多遍历 - 迭代的顺序不能使用这个技术处理。比如，我们可能希望首先访问一个迭代空间中的奇数行，然后再访问其偶数行。或者我们可能想从迭代空间的中间开始然后逐渐到达边缘地带。但是，对于具有仿射访问函数的应用程序而言，这里描述的技术覆盖了人们期望的大部分迭代排序。

### 11.3.7 11.3 节的练习

**练习 11.3.1：**把下面的循环转换成为另一种形式，其中循环下标每次增加 1：

- 1) 

```
for (i=10; i<50; i=i+7) X[i,i+1] = 0;
```
- 2) 

```
for (i= -3; i<=10; i=i+2) X[i] = X[i+1];
```
- 3) 

```
for (i=50; i>=10; i--) X[i] = 0;
```

**练习 11.3.2：**画出或描述下面的每个循环嵌套结构的迭代空间：

- 1) 图 11-17a 中的循环嵌套结构。
- 2) 图 11-17b 中的循环嵌套结构。
- 3) 图 11-17c 中的循环嵌套结构。

```
for (i = 1; i < 30; i++)
    for (j = i+2; j < 40-i; j++)
        X[i,j] = 0;
```

a) 练习 11.3.2(1) 的循环嵌套结构

```
for (i = 10; i <= 1000; i++)
    for (j = i; j < i+10; j++)
        X[i,j] = 0;
```

b) 练习 11.3.2(2) 的循环嵌套结构

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100+i; j++)
        for (k = i+j; k < 100-i-j; k++)
            X[i,j,k] = 0;
```

c) 练习 11.3.2(3) 的循环嵌套结构

图 11-17 练习 11.3.2 的循环嵌套结构

**练习 11.3.3：**按照(11.1)的形式写出图 11-17 中的每个循环嵌套结构所蕴涵的约束。也就是给出向量 **i** 和 **b** 以及矩阵 **B** 的值。

**练习 11.3.4：**颠倒图 11-17 中的各个嵌套结构的循环嵌套顺序。

**练习 11.3.5：**使用 Fourier-Motzkin 消除算法从练习 11.3.3 中得到的各个约束集合中消除 *i*。

**练习 11.3.6：**使用 Fourier-Motzkin 消除算法从练习 11.3.3 中得到的各个约束集合中消除 *j*。

**练习 11.3.7：**对于图 11-17 中的每个循环嵌套结构，改写相应的代码，使得坐标轴 *i* 被替换为主对角线，即新的坐标轴可以用 *i* = *j* 描述。新的坐标轴应该对应于最外层循环。

**练习 11.3.8：**对于下列的坐标轴变换，重复练习 11.3.7：

- 1) 将 *i* 替换为 *i* + *j*，即新的坐标轴的方向是 *i* + *j* 等于常量的直线。新的坐标轴对应于最外层的循环。
- 2) 将 *j* 替换为 *i* - 2*j*。新的坐标轴对应于最外层循环。

! 练习 11.3.9: 在下列循环中, 令  $A, B$  和  $C$  为常整数并且  $C > 1, B > A$ :

```
for (i = A; i <= B; i = i + C)
    Z[i] = 0;
```

改写这个循环使得该循环的下标变量的增量为 1, 并且初值为 0。也就是说, 新循环的形式如下:

```
for (j = 0; j <= D; j++)
    Z[E*j + F] = 0;
```

其中  $D, E, F$  为整数。把  $D, E, F$  表示为  $A, B, C$  的表达式。

练习 11.3.10: 对于一个通用的深度为 2 的循环嵌套结构

```
for (i = 0; i <= A; i++)
    for(j = B*i+C; j <= D*i+E; j++)
```

其中  $A$  到  $E$  是常整数。以矩阵 - 向量的形式, 即  $Bi + b = 0$  的形式写出这个循环嵌套结构的迭代空间的约束。

练习 11.3.11: 对于如下的带有整数符号常量  $m$  和  $n$  的通用的深度为 2 的循环嵌套结构

```
for (i = 0; i <= m; i++)
    for(j = A*i+B; j <= C*i+n; j++)
```

重复练习 11.3.10。和前面一样,  $A, B$  和  $C$  表示特定的整数常量。只有  $i, j, m$  和  $n$  可以在未知量的向量中出现。另外请记住, 只有  $i$  和  $j$  是表达式的输出变量。

## 11.4 仿射的数组下标

本章关注的是仿射数组访问, 即每个数组下标都被表示为循环下标和符号常量的仿射表达式。仿射函数提供了从迭代空间到数据空间的简明的映射关系, 这使得我们容易确定哪些迭代被映射到同一个数据或同一个高速缓存线。

就像一个循环的仿射上下界可以表示成一个矩阵 - 向量的计算一样, 我们可以使用同样的方法来处理仿射访问函数。只要把仿射访问函数表示成矩阵 - 向量的形式, 我们就可以应用标准的线性代数技术来寻找相关的信息, 比如被访问数据的维度以及哪些迭代指向同一个数据。

### 11.4.1 仿射访问

如果下列条件成立, 我们就说一个循环中的一个数组访问是仿射的。

- 1) 该循环的上下界被表示为外围循环变量和符号常量的仿射表达式, 且
- 2) 该数组的每个维度的下标也是外围循环变量和符号常量的仿射表达式。

**例 11.16** 假设  $i$  和  $j$  是循环下标变量, 其上下界通过仿射表达式给出。仿射数组访问的例子有  $Z[i]$ ,  $Z[i+j+1]$ ,  $Z[0]$ ,  $Z[i,i]$  和  $Z[2*i+1, 3*j-10]$ 。如果  $n$  是一个循环嵌套结构中的符号常量, 那么  $Z[3*n, n-j]$  是仿射数组访问的另一个例子。但是  $Z[i*j]$  和  $Z[n*j]$  不是仿射访问。 □

每个仿射数组访问可以用两个矩阵和两个向量来描述。第一个矩阵 - 向量对是  $B$  和  $b$ , 它们以式(11.1)中的不等式的方式描述了该访问的迭代空间。我们通常用  $F$  和  $f$  来表示第二对矩阵 - 向量对。它们表示循环下标变量的函数, 这些函数生成了在数组访问的不同维度中使用的数组下标。

正式地说, 我们把使用了下标变量向量  $i$  的一个循环嵌套结构中的数组访问表示为一个四元组  $\mathcal{F} = \langle F, f, B, b \rangle$ ; 它把界限

$$Bi + b \geq 0$$

中的向量  $i$  映射到数组元素位置

$$Fi + f$$

**例 11.17** 图 11-18 中是一些常见的用矩阵表示法表示的数组访问。两个循环下标  $i$  和  $j$  组成了向量  $i$ 。另外,  $X$ 、 $Y$  和  $Z$  分别是维度为 1、2 和 3 的数组。

第一个数组访问  $X[i-1]$  由一个  $1 \times 2$  的矩阵  $F$  和一个长度为 1 的向量  $f$  表示。请注意, 当我们执行矩阵 - 向量乘法并加到  $f$  中时, 我们得到一个函数  $i-1$ 。这个函数就是前面提到的对一维数组  $X$  进行访问所使用的公式。同时请注意, 第三个访问  $Y[j, j+1]$  在进行矩阵 - 向量乘法和加法之后, 得到一个函数对  $(j, j+1)$ 。它们就是这个二维数组访问的下标。

最后, 我们观察第四个数组访问  $Y[1, 2]$ 。这个访问是一个常量, 毫无疑问, 矩阵  $F$  是全零矩阵。因此, 循环下标的向量  $i$  没有出现在访问函数中。  $\square$

#### 11.4.2 实践中的仿射访问和非仿射访问

在数值计算程序中, 有一些常见的数据访问模式不是仿射的。涉及稀疏矩阵的程序是一个重要的例子。稀疏矩阵的常用表示方法之一是只保存一个向量中的非零元素, 并使用辅助的下标数组来记录某一行从哪里开始, 以及哪些列包含非零元素。访问这样的数据时要使用间接数组访问。这种类型的访问, 比如  $X[Y[i]]$ , 是对数组  $X$  的非仿射访问。如果矩阵的稀疏情况是有规律的, 比如在一个带状矩阵中只有在对角线周围才有非零元素, 那么可以使用紧密数组来表示带有非零元素的子区域。在这样的情况下, 数组访问仍可能是仿射的。

另一个常见的非仿射访问的例子是线性化的数组。程序员有时使用一个线性数组来存放一个在逻辑上多维的对象。这么做的原因之一是多维数组的维度可能在编译时刻未知。在正常情况下写成  $Z[i, j]$  形式的访问现在变成了  $Z[i * n + j]$ , 其下标是一个二次函数。如果对线性化数组的每个访问都可以被分解为不同维度的分量并保证每个分量都不会超过维度界限, 那么我们可以把这个线性访问转换成为一个多维的访问。最后, 如例 11.18 所示, 我们注意到可以使用归纳变量分析把一些非仿射访问转换成为仿射访问。

**例 11.18** 我们可以把下面的代码

```
j = n;
for (i = 0; i <= n; i++) {
    Z[j] = 0;
    j = j+2;
}
```

写成

```
j = n;
for (i = 0; i <= n; i++) {
    Z[n+2*i] = 0;
}
```

这样做使得这个对矩阵  $Z$  的访问变成仿射的。  $\square$

#### 11.4.3 11.4 节的练习

**练习 11.4.1:** 对于下面的每个数组访问, 给出描述它们的向量  $f$  和矩阵  $F$ 。假设下标向量  $i$

数组访问	仿射表达式
$X[i-1]$	$[1 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$
$Y[i, j]$	$[1 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [0 \ 0]$
$Y[j, j+1]$	$[0 \ 1] \begin{bmatrix} i \\ j \end{bmatrix} + [0 \ 1]$
$Y[1, 2]$	$[0 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [1 \ 2]$
$Z[1, i, 2*i+j]$	$[0 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [1 \ 0]$

图 11-18 一些数组访问和它们的矩阵 - 向量表示

是  $i, j, \dots$ , 并且所有的循环下标都有仿射的界限。

- 1)  $X[2 * i + 3, 2 * j - i]$
- 2)  $Y[i - j, j - k, k - i]$
- 3)  $Z[3, 2 * j, k - 2 * i + 1]$

## 11.5 数据复用

从数组访问函数中我们得到了两种可用于局部性优化和并行化的有用信息：

1) **数据复用**: 对于局部性优化, 我们希望识别出访问相同数据或相同高速缓存线的迭代集合。

2) **数据依赖**: 为了并行化和局部性循环转换的正确性, 我们希望找出代码中的所有数据依赖关系。回顾一下, 如果两个(不一定不同的)访问的实例可能指向相同的内存位置, 并且其中至少有一个是写运算, 那么这两个访问之间具有数据依赖关系。

在很多情况下, 只要我们找到了复用相同数据的迭代, 就知道它们之间必然存在数据依赖关系。

只要存在数据依赖关系, 显然就会有相同的数据被复用。比如, 在矩阵乘法中, 输出数组中的同一个元素被写  $O(n)$  次。这些写运算必须按照原来的顺序执行<sup>⊖</sup>, 我们可以分配一个寄存器, 令它在计算输出数组的一个元素时存放该元素。这个就可以利用这个数据复用机会。

不过, 并不是所有的数据复用都可以用到局部性优化中, 下面的例子说明了这个问题。

**例 11.19** 考虑下面的循环:

```
for (i = 0; i < n; i++)
    Z[7*i+3] = Z[3*i+5];
```

我们观察到这个循环在每次迭代时都对不同的内存位置进行写运算, 因此在不同的写操作之间没有复用或者依赖关系。但是, 这个循环从位置 5、8、11、14、17、…读取数据, 而向位置 3、10、17、24…写入数据。不同迭代的读运算和写运算访问了共同的元素 17、38 和 59…。也就是说, 对于  $j = 0, 1, 2, \dots$ , 形如  $17 + 21j$  ( $j = 0, 1, 2, \dots$ ) 的整数就是所有既可以写作  $7i_1 + 3$ , 又可以写作  $3i_2 + 5$  的整数, 这里  $i_1, i_2$  是两个整数。但是这种复用很少发生, 因此即使有可能利用这种复用, 也不容易做到。□

数据依赖和复用分析的不同之处在于: 具有数据依赖关系的访问中必须有一个访问是写访问。更重要的是, 数据依赖关系必须既正确又精确。为了保持正确性, 它必须找到所有的依赖关系。但是, 它又不应该找出假的依赖关系, 因为这些假依赖关系会引起不必要的串行执行。

考虑数据复用时, 我们只需要找出大部分可利用的复用在哪里。这个问题就简单多了, 因此我们在本节中就讨论这个主题, 接下来再讨论数据依赖问题。因为循环界限很少改变复用区域的形状, 所以可以通过忽视循环界限来简化对复用的分析。可以被仿射分划利用的很多数据复用位于相同数组访问的不同实例之间, 以及使用相同的系数矩阵(即在仿射下标函数中通常被称为  $\mathbf{F}$  的矩阵)的访问之间。上面介绍过, 像  $7i + 3$  和  $3i + 5$  这样的访问模式没有令人感兴趣的复用。

### 11.5.1 数据复用的类型

我们首先用例 11.20 来说明不同种类的数据复用。在下面的内容中, 我们需要区分作为程序

<sup>⊖</sup> 这里有一个微妙之处。根据加法的交换率, 不管我们按照什么顺序执行加法, 我们依然得到相同的结果。但是, 这种情况是很特别的。一般来说, 让编译器来决定在一个写运算之前的一系列算术运算步骤完成哪些计算过于复杂。我们也不能依赖于会有任何代数规则来帮助我们安全地重新排列这些步骤。

中的一个指令的访问（比如  $x = Z[i, j]$ ）和我们执行循环嵌套结构时指令的多次执行。为了强调它们之间的区别，我们将把访问指令本身称为静态访问（static access），而当我们执行该循环嵌套结构时该语句的多次迭代称为动态访问（dynamic access）。

数据复用可以分为自复用和组复用两种。如果复用同样数据的多个迭代源于同一个静态访问，我们就把这样的复用称为自复用；如果它们源于不同的静态访问，那么我们称这个复用为组复用。如果一个复用指向完全相同的位置，那么它就是时间复用；如果指向同一个高速缓存线，那么它就是空间复用。

**例 11.20** 考虑下面的循环嵌套结构：

```
float Z[n];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        Z[j+1] = (Z[j] + Z[j+1] + Z[j+2])/3;
```

数组访问  $Z[j]$ 、 $Z[j+1]$  和  $Z[j+2]$  都具有自空间复用性，因为同一个访问的连续迭代指向连续的数组元素。我们假定连续元素很可能存放在同一个高速缓存线中。另外，这些访问都具有自时间复用性，因为在外层循环的每次迭代中，同一个元素被多次使用。再者，它们都具有同样的系数矩阵，因此具有组复用性。在不同的访问之间具有组复用性，而且既是时间性复用，又是空间性复用。当这些复用都可以利用时，虽然在代码中有  $4n^2$  次访问，我们只需要把大约  $n/c$  个高速缓存线加载到高速缓存中即可，其中  $c$  是一个高速缓存线中的内存字的数量。因为具有自空间复用性，我们从  $4n^2$  中消除了一个因子  $n$ ；因为存在空间局部性，我们又把加载次数降低了  $c$  倍；最后因为组复用的原因又降低了 4 倍。□

下面我们说明如何使用线性代数从仿射数组访问中抽取复用信息。我们感兴趣的不仅仅是找出有多大的提高性能的潜力，而且要找出哪些迭代在复用数据，以便把这些迭代移近从而利用这些复用。

### 11.5.2 自复用

通过利用自复用可以有效节约在内存访问方面的开销。如果一个静态访问所指向的数据具有  $k$  个维度，且这个访问嵌套在一个深度为  $d$  ( $d > k$ ) 的循环结构中，那么同一个数据可以被复用  $n^{d-k}$  次。其中， $n$  是每个循环的迭代次数。比如，如果一个深度为 3 的循环嵌套结构访问一个数组的某一列，那么访问节约系数就可能达到  $n^2$ 。实际上，一个访问的维度和这个访问的系数矩阵的秩相对应。我们可以通过寻找该矩阵的零空间来找出哪些迭代指向同一个位置。具体方法在下面解释。

#### 矩阵的秩

矩阵  $F$  的秩是  $F$  的线性无关列（或者等价地，行）的最大数目。一个向量集合被称为线性无关（linearly independent）的条件是没有向量可以被写成该集合中有限多个其他向量的线性组合。

**例 11.21** 考虑矩阵

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

请注意，第二行是第一和第三行的和，而第四行是第三行减去第一行的两倍。但是，第一行和第三行是线性独立的；其中的任何一行都不是另一行的倍数。因此，矩阵的秩是 2。

我们也可以通过检查各列来得到这个结果。第三列是第二列的两倍减去第一列。另一方面，

任何两列都是线性独立的。我们同样可以确定矩阵的秩为 2。  $\square$

**例 11.22** 我们看一下图 11-18 中的数组访问。第一个访问  $X[i-1]$  的维度为 1，因为矩阵  $[10]$  的秩为 1。也就是说，唯一的一行是线性独立的，同样第一列也是线性独立的。

第二个访问  $Y[i,j]$  的维度为 2。原因是矩阵

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

具有两个独立的行(当然，因此也具有两个独立的列)。

第三个访问  $Y[j,j+1]$  的维度为 1，因为矩阵

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

的秩为 1。请注意，矩阵中的两行是相同的，因此只有一行是线性独立的。等价地，第一列是第二列乘以 0，因此这两列不是独立的。直观地讲，在一个大的正方形数组  $Y$  中，所有被访问的元素都排列在紧靠主对角线之上的一条斜线上。

第四个访问  $Y[1,2]$  的维度为 0，因为一个全零矩阵的秩为 0。请注意，对于这样的一个矩阵，我们找不出非零的矩阵的行(哪怕只有一行)的线性和。最后一个访问  $Z[i,i,2*i+j]$  的维度为 2。请注意在这个访问的矩阵

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

中，最后两行是线性独立的，任何一行都不是另一行的倍数。但是，第一行是另两行的线性“和”，其中的系数都是 0。  $\square$

### 矩阵的零空间

在一个深度为  $d$  的循环嵌套结构中的秩为  $r$  的数据引用在  $O(n^d)$  个迭代中访问了  $O(n^r)$  个数据元素，因此平均一定有  $O(n^{d-r})$  个迭代指向同一个数组元素。哪些迭代访问了同一个数据呢？假设在这个循环嵌套结构中的一个访问用  $F$  和  $f$  的矩阵 - 向量组合来表示。令  $i$  和  $i'$  为指向同一个数组元素的两个迭代，那么  $Fi + f = Fi' + f$ 。重新排列这个等式中的各项，我们得到

$$F(i - i') = \mathbf{0}$$

有一个众所周知的线性代数概念可以刻画  $i$  和  $i'$  在什么时候满足上述等式。满足等式  $Fv = \mathbf{0}$  的所有解的集合称为  $F$  的零空间。因此，如果两个迭代的循环下标向量的差属于矩阵  $F$  的零空间，那么它们指向同一个数组元素。

显然，零向量  $v = \mathbf{0}$  总是满足  $Fv = \mathbf{0}$ 。也就是说，如果两个向量的差为  $\mathbf{0}$ ，那么它们一定指向同一个数组元素。换句话说，如果它们实际上是同一个迭代，它们就指向同一个元素。另外，零空间确实是一个向量空间。也就是说，如果  $Fv_1 = \mathbf{0}$  且  $Fv_2 = \mathbf{0}$ ，那么  $F(v_1 + v_2) = \mathbf{0}$  且  $F(cv_1) = \mathbf{0}$ 。

如果矩阵  $F$  是全秩的(fully ranked)，也就是说它的秩为  $d$ ，那么  $F$  的零空间只包含零向量。在这种情况下，一个循环嵌套的各个迭代指向不同的数据。一般来说，零空间的维度，或者说零数(nullity)，就是  $d - r$ 。如果  $d > r$ ，那么对于每个元素，访问该元素的迭代组成一个  $(d - r)$  维空间。

零空间可以用它的基本向量表示。一个  $k$  维的零空间可以由  $k$  个独立的向量表示，任何可以被表示为这些基本向量的线性组合的向量都属于这个空间。

**例 11.23** 重新考虑例 11.21 的矩阵

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix}$$

在例 11.21 中，我们确定这个矩阵的秩为 2，因此其零数为  $3 - 2 = 1$ 。在这个例子中，零空间的基本向量必然是一个长度为 3 的非零向量。为了找到这个零空间的基本向量，我们假设零空间中的一个向量为  $[x, y, z]$ ，并尝试求解下面的方程

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 4 & 5 & 6 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

如果我们将前面两行乘以未知向量，就得到两个方程

$$x + 2y + 3z = 0$$

$$5x + 7y + 9z = 0$$

我们也可以根据第三和第四行写出这样的方程，但是因为不存在三个线性独立的行，所以增加方程不会对  $x, y$  和  $z$  增加新的约束。比如，我们从第三行得到的方程  $4x + 5y + 6z = 0$  就是通过从第二个方程中减去第一个方程得到的。

我们必须尽可能地从上面的等式中消除变量。首先使用第一个方程求解  $x$ ，得到  $x = -2y - 3z$ 。然后在第二个方程中替换  $x$ ，得到  $-3y = 6z$ ，即  $y = -2z$ 。由  $x = -2y - 3z$  且  $y = -2z$  可知  $x = z$ 。因此，变量  $[x, y, z]$  实际上是  $[z, -2z, z]$ 。我们可以选择任意的非零  $z$  值，得到这个零空间的唯一基本向量。比如，我们可以选择  $z = 1$  并把  $[1, -2, 1]$  当作这个零空间的基本向量。□

**例 11.24** 例 11.17 中的所有数组访问的秩、零数和零空间显示在图 11-19 中，请注意，在所有情况下秩和零数的和都等于该循环嵌套的深度 2。因为数组访问  $Y[i, j]$  和  $Z[1, i, 2 * i + j]$  的秩都是 2，因此所有的迭代都指向不同的位置。

访问	仿射表达式	秩	零数	零空间的基本向量
$X[i-1]$	$[1 \ 0] \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$	1	1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$Y[i, j]$	$\begin{bmatrix} 1 \ 0 \\ 0 \ 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$	2	0	
$Y[j, j+1]$	$\begin{bmatrix} 0 \ 1 \\ 0 \ 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	1	1	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
$Y[1, 2]$	$\begin{bmatrix} 0 \ 0 \\ 0 \ 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	0	2	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$Z[1, i, 2*i+j]$	$\begin{bmatrix} 0 \ 0 \\ 1 \ 0 \\ 2 \ 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	2	0	

图 11-19 仿射访问的秩和零数

数组访问  $X[i-1]$  和  $Y[j, j+1]$  的矩阵的秩都是 1，因此  $O(n)$  个迭代指向同一个位置。在前一种情况下，迭代空间的一整行都指向同一个位置。换句话说，仅仅  $j$  值不同的所有迭代指向同一个位置。这一事实由相应零空间的基本向量  $[0, 1]$  明确表示。对于  $Y[j, j+1]$ ，迭代空间中的整列指向同一个位置。这个事实由相应零空间的基本向量  $[1, 0]$  明确表示。

最后，数组访问  $Y[1, 2]$  在所有迭代中指向同一个位置。相应的零空间有两个基本向量  $[0, 1]$  和  $[1, 0]$ ，这表示这个循环嵌套中的任何一对迭代都准确地指向同一个位置。□

### 11.5.3 自空间复用

空间复用的分析依赖于矩阵的数据布局。C 语言的矩阵是按行存放的，而 Fortran 语言的矩阵按列存放。换句话说，在 C 语言中数组元素  $X[i, j]$  和  $X[i, j+1]$  相邻，而在 Fortran 语言中  $X[i, j]$  和  $X[i+1, j]$  相邻。不失一般性，在本章余下的部分，我们将选用 C 语言的数组布局方式（按行存放方式）。

首先作出如下的近似，当且仅当两个数组元素位于一个二维数组的同一行中时，我们认为它们共享一个高速缓存线。更加一般地讲，在一个  $d$  维数组中，如果两个元素只在最后一维的下标值上有所不同，我们就认为它们共享一个高速缓存线。因为对于通常的数组和高速缓存线，多个数组元素可以被放到同一高速缓存线中，以整行的方式顺序访问一个数组可以明显提高访问速度。虽然严格地讲，我们有时还需要等待加载一个新的高速缓存线。

发现和利用自空间复用的技巧是不考虑系数矩阵  $\mathbf{F}$  中的最后一行。如果得到的截短后的矩阵的秩小于循环嵌套结构的深度，那么我们就可以确保最内层循环只改变数组的最后下标的值，从而保证空间局部性。

**例 11.25** 考虑图 11-19 中的最后一个数组访问  $Z[1, i, 2 * i + j]$ 。如果我们删除最后一行，就会得到下面的截短后的矩阵

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

这个矩阵的秩显然是 1。因为该循环嵌套结构的深度为 2，所以存在空间复用的机会。在这个例子中，因为  $j$  是内层循环的下标且  $Z$  是按行存放的，所以内层循环访问  $Z$  的连续元素。让  $i$  成为内层循环的下标不会产生空间局部性。因为当  $i$  改变时，第二和第三个维度都会改变。□

确定是否存在自空间复用的一般规则如下。如我们一直所做的，假设各循环的下标和系数矩阵的各列顺序对应，其中最外层循环对应于第一列，最内层循环对应于最后一列。然后，为了确保存在空间复用，向量  $[0, 0, \dots, 0, 1]$  必须在被截短的矩阵的零空间中。理由是如果这个向量在零空间中，那么当我们把除了最内层下标之外的所有下标都固定下来的时候，就知道在最内层循环的一次执行中所有的动态访问都只在最后的数组下标上取不同的值。如果数组是按行存放的，那么这些元素之间距离接近，有可能在同一高速缓存线中。

**例 11.26** 请注意， $[0, 1]$ （转置为一个列向量）位于例 11.25 中的被截短矩阵的零空间中。因此，我们期望当把  $j$  当作内层循环下标时会出现空间局部性。另一方面，如果我们颠倒循环的顺序使得  $i$  成为内层循环，那么系数矩阵就变成

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

现在， $[0, 1]$  就不在这个矩阵的零空间中了。相反地，零空间由基本向量  $[1, 0]$  生成。因此，如我们在例 11.25 中所建议的，如果  $i$  是内层循环，我们不再期望这个循环具有空间局部性。

但是，我们应该观察到向量  $[0, 0, \dots, 0, 1]$  在零空间里远不足以保证空间局部性。比如，假设

该访问不是  $Z[1, i, 2 * i + j]$  而是  $Z[1, i, 2 * i + 50 * j]$ 。那么在内层循环的一次运行中， $Z$  的每 50 个元素只有一个元素会被访问，除非一个高速缓存线长到足以保存 50 个以上的元素，否则我们将无法复用一个高速缓存线。□

#### 11.5.4 组复用

我们只在同一个循环中的具有相同系数矩阵的数组访问之间计算组复用。给定两个动态访问  $\mathbf{F}i_1 + f_1$  和  $\mathbf{F}i_2 + f_2$ ，它们复用相同的数据的条件是

$$\mathbf{F}i_1 + f_1 = \mathbf{F}i_2 + f_2$$

或者说

$$\mathbf{F}(i_1 - i_2) = (f_2 - f_1)$$

假设  $v$  是这个等式的一个解，如果  $w$  是  $F$  的零空间中的任意向量，那么  $w + v$  也是一个解。实际上，这样的向量就是该方程的全部解。

**例 11.27** 下面的深度为 2 的循环嵌套结构

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        Z[i, j] = Z[i-1, j];
```

有两个数组访问  $Z[i, j]$  和  $Z[i-1, j]$ 。请注意，这两个访问都可以使用系数矩阵

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

来刻画。这个矩阵和图 11-19 中的第二个访问  $Y[i, j]$  的矩阵一样。这个矩阵的秩为 2，因此没有自时间复用。

但是，每个访问都展示了自空间复用。如 11.5.3 节中所述，当我们删除该矩阵的最下面一行后，只留下最上面的一行  $[1, 0]$ ，其秩为 1。因为  $[0, 1]$  位于这个被截短矩阵的零空间中，所以我们期望找到空间复用。内层循环下标  $j$  的每次增加都会把第二个下标的值增加 1，实际上确实访问了连续的数组元素，并将充分利用每个高速缓存线。

虽然两个访问都没有自时间复用，请注意这两个访问  $Z[i, j]$  和  $Z[i-1, j]$  所访问的几乎是同一个集合的数组元素。也就是说，除了  $i=1$  的情况之外，数组访问  $Z[i-1, j]$  所读取的数据和数组访问  $Z[i, j]$  所写入的数据相同，因此它们之间存在组时间复用。这个简单的访问模式对于整个迭代空间都成立，可以利用这个模式来提高代码的数据局部性。正式地讲，如果不考虑循环界限，那么只要

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

成立，分别位于迭代  $(i_1, j_1)$  和迭代  $(i_2, j_2)$  中的两个数组访问  $Z[i, j]$  和  $Z[i-1, j]$  指向同一个位置。改写这些项，我们得到

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

也就是说， $j_1 = j_2$  且  $i_2 = i_1 + 1$ 。

请注意，这个复用是沿着迭代空间的  $i$  轴发生的。也就是说，迭代  $(i_2, j_2)$  在迭代  $(i_1, j_1)$  发生之后的  $n$  次（内层循环的）迭代之后才发生。因此，在被写入数据被复用之前要执行很多个迭代。此时这个数据有可能在（也有可能不在）高速缓存中了。如果高速缓存中存放了矩阵  $Z$  的连续两行，那么数组访问  $Z[i-1, j]$  不会发生高速缓存脱靶现象，整个循环嵌套结构的总的高速缓存脱靶数量为  $n^2/c$ ，其中  $c$  是每个高速缓存线中的元素数量。否则，脱靶次数将会为原来的两倍，因

为这两个静态访问对于每  $c$  个动态访问都要求加载一个新的高速缓存线。  $\square$

**例 11.28** 假设在一个深度为 3 的循环嵌套结构中有两个访问  $A[i, j, i+j]$  和  $A[i+1, j-1, i+j]$ 。该嵌套结构的下标从外到内分别是  $i, j, k$ 。那么对于两个访问  $i_1 = [i_1, j_1, k_1]$  和  $i_2 = [i_2, j_2, k_2]$ , 只要

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

成立, 它们就能复用同一个数组元素。

这个方程成立时, 向量  $v = [i_1 - i_2, j_1 - j_2, k_1 - k_2]$  的一个解为  $v = [1, -1, 0]$ 。也就是说  $i_1 = i_2 + 1, j_1 = j_2 - 1$  且  $k_1 = k_2$ 。<sup>⊖</sup> 然而, 矩阵

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

的零空间是由基本向量  $[0, 0, 1]$  生成的。也就是说, 第三个循环下标  $k$  可以是任意值。因此, 上面方程的解  $v$  可以是  $[1, -1, m]$ , 其中  $m$  为任意值。换句话说, 在一个下标为  $i, j, k$  的循环嵌套结构中,  $A[i, j, i+j]$  的一个动态访问不仅被  $A[i, j, i+j]$  的具有同样  $i, j$  值和不同  $k$  值的其他动态访问所复用, 也被  $A[i+1, j-1, i+j]$  的其循环下标值为  $i+1, j-1$  和任意  $k$  值的动态访问所复用。  $\square$

我们可以用类似的方法来考虑组空间复用, 虽然不会在这里这么做。和针对自空间复用的讨论一样, 我们只需要舍弃被考虑矩阵的最后一维就可以了。

对于不同种类的复用, 复用的程度是不同的。自时间复用的好处最多: 一个具有  $k$  维零空间的数组访问对同一个数据会复用  $O(n^k)$  次。自空间复用的程度受到高速缓存线长度的限制。最后, 组复用的程度受一个组中共享该复用的数组访问数目的限制。

### 11.5.5 11.5 节的练习

**练习 11.5.1:** 计算图 11-20 中各个矩阵的秩。并给出每个矩阵的最大线性独立列的集合, 以及最大的线性独立行的集合。

**练习 11.5.2:** 找出图 11-20 中各个矩阵的零空间的基本向量。

**练习 11.5.3:** 假设一个迭代空间的维度(变量)为  $i, j$  和  $k$ 。对于下面的每个访问, 描述指向下列数组元素的子空间:

- 1)  $A[i, j, i+j]$
- 2)  $A[i, i+1, i+2]$
- 3)  $A[i, i, j+k]$

$$\begin{array}{c} \begin{bmatrix} 0 & 1 & 5 \\ 1 & 2 & 6 \\ 2 & 3 & 7 \\ 3 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 12 & 15 \\ 3 & 2 & 2 & 3 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 5 & 6 & 3 \end{bmatrix} \\ \text{a)} \qquad \qquad \qquad \text{b)} \qquad \qquad \qquad \text{c)} \end{array}$$

图 11-20 计算这些矩阵的秩和零空间

<sup>⊖</sup> 在这里可以观察到一件很有意思的事情。虽然这个例子有一个解, 但如果我们将第三个分量  $i+j$  改成  $i+j+1$ , 解就不存在了。也就是说, 在这个给定的例子中, 两个访问所触及的数组元素都存在于一个二维的子空间  $S$  中。该空间可以定义为“第三个分量是前面两个分量的和”。如果我们把  $i+j$  改成  $i+j+1$ , 则第二个访问所触及的元素都不在  $S$  中, 因此也不存在任何复用。

! 4)  $A[i-j, j-k, k-i]$

! 练习 11.5.4: 假设数组  $A$  按行存放, 并在下面的循环嵌套结构中被访问:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        for (k = 0; k < 100; k++)
            <某个对 A 的访问>
```

对于下列的各个访问, 指出是否可能改写该循环结构, 使得对  $A$  的访问具有自空间复用特性。也就是说, 整个高速缓存线被连续使用。如果可以, 指明如何改写这个循环。注意, 对循环的改写可以包括对下标变量重新排序或引入新的循环下标。但是不能改变数组的布局, 比如把数组改成按列存放的。还要注意的是, 一般来说, 循环下标的重新排序可能是合法的, 也可能是非法的。我们将在下一节给出判断重新排序是否合法的标准。但是在这个例子中, 因为每个访问的效果就是把一个数组元素设置为 0, 所以不需要担心循环重新排列的效果会影响程序的语义。

- 1)  $A[i+1, i+k, j] = 0$
- !! 2)  $A[j+k, i, i] = 0$
- 3)  $A[i, j, k, i+j+k] = 0$
- !! 4)  $A[i, j-k, i+j, i+k] = 0$

练习 11.5.5: 在 11.5.3 节中, 我们说如果最内层循环只改变一个数组访问的最后一个下标值, 我们就能获得空间局部性。但是这个断言依赖于: 数组是按行存放的假设。如果数组是按列存放的, 那么什么样的条件可以保证空间局部性?

! 练习 11.5.6: 在例 11.28 中, 我们看到两个相似的数组访问之间是否存在复用很大程度上依赖于特定的数组下标表达式。将在例 11.28 中观察到的事实进行推广, 并决定对什么样的函数  $f(i, j)$ , 访问  $A[i, j, i+j]$  和  $A[i+1, j-1, f(i, j)]$  之间存在复用。

! 练习 11.5.7: 在例 11.27 中, 我们指出, 如果矩阵  $Z$  的行的长度很长, 以至于不能一起存放到高速缓存中, 就会产生更多的不必要的高速缓存脱靶。如果出现了这样的情况, 应怎样改写循环嵌套以保证组空间复用?

## 11.6 数组数据依赖关系分析

并行化或局部性优化经常对原程序中执行的运算重新排序。和所有的优化一样, 只有当对运算的重新排序不会改变程序输出时才可以对这些运算重新排序。一般来说, 我们不可能深入理解一个程序到底做了什么, 代码优化通常选用一个较简单的、保守的测试方法来决定在什么时候可以肯定程序的输出不会受到优化的影响: 检查在原程序中和在修改后的程序中, 对同一内存位置的各个运算被执行的顺序是否一样。在当前的研究中, 我们关注的是数组访问, 因此数组元素就是需要考虑的内存位置。

如果两个访问(不管是读还是写)指向两个不同的位置, 显然它们是相互独立的(可以被重新排序)。另外, 读运算不会改变内存的状态, 因此各个读运算之间是独立的。根据 11.5 节的介绍, 如果两个访问指向同一个内存位置并且其中至少有一个写运算, 那么就说这两个访问是数据依赖的。为了保证修改后的程序和原程序做同样的事情, 每一对有数据依赖关系的运算在原程序中的执行顺序必须在新的程序中得到保持。

回顾一下 10.2.1 节, 可知存在三种类型的数据依赖:

- 1) 真依赖, 一个写运算后面跟一个对同一个内存位置的读运算。
- 2) 反依赖, 一个读运算后面跟一个对同一个内存位置的写运算。

3) 输出依赖，是两个针对同一个位置的写运算。

在上面的讨论中，数据依赖是针对动态访问定义的。只要一个程序的某个静态访问的某个动态实例依赖于另一个静态访问的某个动态实例，我们就说第一个静态访问依赖于第二个静态访问<sup>⊖</sup>。

我们可以很容易看出数据依赖关系如何应用到并行化中。比如，如果在一个循环的各个访问之间没有发现数据依赖关系，那么就可以很容易地把不同的迭代分配给不同的处理器。11.7节将讨论如何系统化地将这个信息应用到并行化中。

### 11.6.1 数组访问的数据依赖关系的定义

让我们考虑对同一个数组的两个静态访问，它们可能位于不同的循环中。第一个访问用访问函数和界限表示为  $\mathcal{F} = \langle F, f, B, b \rangle$ ，它位于一个深度为  $d$  的循环嵌套结构中；第二个访问表示为  $\mathcal{F}' = \langle F', f', B', b' \rangle$ ，它位于一个深度为  $d'$  的程序嵌套结构中。如果下面的条件成立，这两个访问就是数据依赖的。

- 1) 它们中至少有一个是写运算，且
- 2) 存在  $Z^d$  中的向量  $i$  和  $Z^{d'}$  中的向量  $i'$  使得
  - ①  $Bi + b \geq 0$
  - ②  $B'i' + b \geq 0$
  - ③  $Fi + f = F'i' + f'$

因为一个静态访问通常会产生多个动态访问，所以考虑它的多个动态访问是否可能指向同一个内存位置也是有意义的。为了寻找同一个静态访问的不同实例之间的依赖关系，我们假设  $\mathcal{F} = \mathcal{F}'$  并在上面的定义中加入附加条件  $i \neq i'$  以剔除平凡解。

**例 11.29** 考虑下面的深度为 1 的循环嵌套结构：

```
for (i = 1; i <= 10; i++) {
    Z[i] = Z[i-1];
}
```

这个循环具有两个数组访问： $Z[i-1]$  和  $Z[i]$ 。第一个访问是读运算，而第二个访问是写运算。为了找到这个程序中的所有数据依赖关系，我们需要检查这个写运算和它自身以及上面的读运算之间是否具有依赖关系：

1)  $Z[i-1]$  和  $Z[i]$  之间的数据依赖关系。除了第一个迭代，每个迭代都会读取前一个迭代写入的值。从数学的角度看，因为存在整数  $i$  和  $i'$  使得

$$1 \leq i \leq 10, 1 \leq i' \leq 10, \text{ 且 } i-1 = i'$$

所以我们知道它们之间存在一个依赖关系。上面的约束系统有九个解： $(i=2, i'=1)$ ,  $(i=3, i'=2)$ , 等等。

2)  $Z[i]$  和它自身之间的依赖关系。可以看到，这个循环的不同迭代向不同的位置写入数据。也就是说，写访问  $Z[i]$  的各个实例之间不存在数据依赖关系。从数学的角度看，因为不存在整数  $i$  和  $i'$  满足条件

$$1 \leq i \leq 10, 1 \leq i' \leq 10, i = i', \text{ 且 } i \neq i'$$

因此我们知道实例之间不存在依赖关系。请注意，之所以有第三个条件  $i = i'$  是因为要求  $Z[i]$  和  $Z[i']$  必须在同一个位置上。和这个条件矛盾的第四个条件  $i \neq i'$  是因为要求依赖关系必

---

⊖ 回忆一下静态访问和动态访问之间的区别。一个静态访问是程序中某个位置上的数组引用，而一个动态访问是这个引用的一次执行。

须是非平凡的——必须是不同动态访问之间的依赖关系。

任意两个读访问总是独立的，因此不需要考虑上面的读引用  $Z[i - 1]$  和它自身之间的依赖关系。  $\square$

### 11.6.2 整数线性规划

对数据依赖关系的分析要求找出是否存在一些整数满足由等式和不等式组成的约束系统。其中的等式是从数组访问的矩阵 - 向量表示中得到的；不等式是从循环界限中得到的。等式可以用不等式表示：等式  $x = y$  可以用两个不等式  $x \geq y$  和  $y \geq x$  表示。

因此，数据依赖关系问题可以被表示为寻找满足一组线性不等式的整数解，这个问题就是众所周知的整数线性规划 (integer linear programming)。整数线性规划是一个 NP 完全问题。虽然没有已知的多项式复杂性的算法，但人们研发了多种启发式解法来解决涉及很多变量的线性规划问题。这些解法在很多情况下运行得是相当快的。遗憾的是，这样的标准启发式解法并不适合数据依赖关系分析。在数据依赖分析中，问题的难点在于如何解决很多小且简单的整数线性规划，而不是大型的复杂整数线性规划。

数据依赖关系分析算法由三个部分组成：

1) 使用丢番图方程的理论，应用 GCD (Greatest Common Divisor, 最大公约数) 测试来检验是否存在满足问题中所有等式的整数解。如果没有整数解，那么就不存在数据依赖关系，否则就用等式来替换其中的某些变量，从而得到较简单的不等式组。

2) 使用一组简单的启发规则来处理大量的典型不等式。

3) 在少数情况下，这些启发式规则可能还解决不了问题。此时，我们使用线性整数规划求解程序来解决问题。这个程序基于 Fourier-Motzkin 消除算法，使用了一种分支并设限的方法来求解。

### 11.6.3 GCD 测试

第一个子程序检验是否存在满足约束中各个等式的整数解。只考虑整数解的方程称为丢番图方程 (Diophantine equation)。下面的例子说明了只考虑整数解会带来什么问题；同时它也说明，虽然很多例子中每次只涉及单个循环嵌套结构，数据依赖关系的公式表达还可以被应用到位于不同循环中的数组访问。

**例 11.30** 考虑下面的代码片段：

```
for (i = 1; i < 10; i++) {
    Z[2*i] = ...;
}
for (j = 1; j < 10; j++) {
    Z[2*j+1] = ...;
}
```

访问  $Z[2 * i]$  只触及  $Z$  的偶数号元素，而访问  $Z[2 * j + 1]$  只触及奇数号元素。显然，如果省略号表示的右部不涉及  $Z$  的运算，那么不管循环的界限如何，这两个访问之间没有数据依赖关系。我们可以在第一个循环执行之前就执行第二个循环，或者交叉执行这两个循环的迭代。这个例子看起来是人为设计的、没有实际意义，其实不然。数组的偶数号元素与奇数号元素被分开处理的一个实际例子是复数数组，其中各个复数的实部和虚部各占一个元素，并列存放。

为了证明这个例子中没有数据依赖关系，我们做如下论证。假设存在整数  $i$  和  $j$  使得  $Z[2 * i]$  和  $Z[2 * j + 1]$  是同一个数组元素，我们得到丢番图方程

$$2i = 2j + 1$$

没有整数  $i$  和  $j$  可以满足上面的方程。证明如下：如果  $i$  是一个整数，那么  $2i$  就是偶数。如

果  $j$  是一个整数，那么  $2j$  是偶数，因此  $2j+1$  是奇数。没有哪个偶数同时也是奇数。因此，这个方程没有整数解，因此这两个写访问之间没有依赖关系。□

为了描述一个线性丢番图方程什么时候有解，我们需要引入两个或多个整数的最大公约数的概念。多个整数  $a_1, a_2, \dots, a_n$  的 GCD，记为  $\gcd(a_1, a_2, \dots, a_n)$ ，是能够整除这些整数的最大整数。GCD 可以使用著名的欧几里德算法（见下面的“欧几里德算法”部分）快速地计算。

**例 11.31**  $\gcd(24, 36, 54) = 6$ ，因为  $24/6, 36/6$  和  $54/6$  的余数都是 0，而且用任何大于 6 的整数去除  $24, 36, 54$  时，至少有一个余数非零。比如，12 能够整除 24 和 36，但是不能整除 54。□

GCD 的重要性体现在下面的定理中。

**定理 11.32** 线性丢番图方程

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

有  $x_1, x_2, \dots, x_n$  的一个整数解，当且仅当  $\gcd(a_1, a_2, \dots, a_n)$  能够整除  $c$ 。□

**例 11.33** 在例 11.30 中，我们看到线性丢番图方程  $2i = 2j + 1$  无解。我们可以把这个方程写作

$$2i - 2j = 1$$

现在  $\gcd(2, -2) = 2$  且 2 不能整除 1。因此方程无解。

再看另一个例子，考虑方程

$$24x + 36y + 54z = 30$$

因为  $\gcd(24, 36, 54) = 6$  且  $30/6 = 5$ ，因此存在  $x, y$  和  $z$  的整数解。其中的一个解是  $x = -1, y = 0$  且  $z = 1$ ，但是存在无穷多个其他的解。□

数据依赖关系问题的第一步是使用一个诸如高斯消除算法的标准方法来求解给定的方程组。每构造出一个线性方程，就应用定理 11.32 尽可能地排除整数解的存在。如果我们能够排除这样的整数解，那么答案就是“否”。否则我们使用这些方程的解来减少不等式中的变量数目。

**例 11.34** 考虑两个方程

$$\begin{aligned} x - 2y + z &= 0 \\ 3x + 2y + z &= 5 \end{aligned}$$

从各个方程本身来看是存在解的。对于第一个方程， $\gcd(1, -2, 1) = 1$  能够整除 0，而对于第二个方程， $\gcd(3, 2, 1) = 1$  能够整除 5。但是，如果我们求解第一个方程得到  $z = 2y - x$ ，并以此替代第二个方程中的  $z$ ，我们得到  $2x + 4y = 5$ 。因为  $\gcd(2, 4) = 2$  不能整除 5，所以这个丢番图方程无解。□

### 欧几里德算法

欧几里德算法按照下面的方法找出  $\gcd(a, b)$  的值。首先，假设  $a$  和  $b$  为正整数，且  $a \geq b$ 。请注意，多个负数的 GCD，或一个负数与一个正数的 GCD 等于它们的绝对值的 GCD，因此可以假设所有的整数都是正的。

如果  $a = b$ ，那么  $\gcd(a, b) = a$ 。如果  $a > b$ ，令  $c$  为  $a/b$  的余数。如果  $c = 0$ ，那么  $b$  整除  $a$ ，因此  $\gcd(a, b) = b$ 。否则，计算  $\gcd(b, c)$  得到的结果也是  $\gcd(a, b)$ 。

为了计算  $n > 2$  时的  $\gcd(a_1, a_2, \dots, a_n)$ ，使用欧几里德算法来计算  $\gcd(a_1, a_2) = c$ ，然后递归地计算  $\gcd(c, a_3, a_4, \dots, a_n)$ 。

### 11.6.4 解决整数线性规划的启发式规则

数据依赖关系问题需要求解很多简单的整数线性规划问题。现在我们讨论处理简单不等式组的几个技术，以及一个可以利用在数据依赖关系分析时发现的相似性的技术。

#### 独立变量测试

从数据依赖关系分析中得到的很多整数线性规划问题由多个只涉及一个未知量的不等式组成。这类规划问题的解法很简单，只需要分别测试常量上界和常量下界之间是否存在整数即可。

#### 例 11.35 考虑嵌套循环结构

```
for (i = 0; i <= 10; i++)
    for (j = 0; j <= 10; j++)
        Z[i, j] = Z[j+10, i+11];
```

为了找出  $Z[i, j]$  和  $Z[j+10, i+11]$  之间是否存在数据依赖关系，我们考虑是否存在整数  $i, j, i'$  和  $j'$ ，使得

$$0 \leq i, j, i', j' \leq 10$$

$$i = j' + 10$$

$$j = i' + 11$$

对其中的方程应用 GCD 测试可以确定可能存在一个整数解。这些方程的整数解可表示如下：

$$i = t_1, j = t_2, i' = t_2 - 11, j' = t_1 - 10$$

其中， $t_1$  和  $t_2$  是任意整数。把变量  $t_1$  和  $t_2$  代入上面的线性不等式，我们得到

$$0 \leq t_1 \leq 10$$

$$0 \leq t_2 \leq 10$$

$$0 \leq t_2 - 11 \leq 10$$

$$0 \leq t_1 - 10 \leq 10$$

这样，把根据后两个不等式得到的下界与根据前两个不等式得到的上界组合起来，我们推出

$$10 \leq t_1 \leq 10$$

$$11 \leq t_2 \leq 10$$

因为  $t_2$  的下界大于它的上界，因此不存在整数解，也就没有数据依赖关系。这个例子说明，即使存在涉及多个变量的等式，GCD 测试（原文如此，实际应该是独立变量测试，译者注）依然可以构造出每个不等式只涉及一个变量的线性不等式组。□

#### 无环测试

另一个简单的启发式规则是寻找是否存在一个其上界或下界为常量的变量。在某些情况下，我们可以安全地用这个常量来替换这个变量。简化后的不等式组有一个整数解当且仅当原来的不等式组有一个整数解。明确地说，假设  $v_i$  的每个下界都具有如下形式：

对于某个  $c_i > 0, c_0 \leq c_i v_i$

同时  $v_i$  的上界都具有如下形式：

$$c_i v_i \leq c_0 + c_1 v_1 + \cdots + c_{i-1} v_{i-1} + c_{i+1} v_{i+1} + \cdots + c_n v_n$$

其中， $c_i$  是非负整数。那么我们可以把变量  $v_i$  替换为最小的可能整数值。如果没有这样的下界，我们可以把  $v_i$  替换为  $-\infty$ 。类似地，如果涉及  $v_i$  的所有约束都可以表示成上面的两种形式，但是不等号的方向相反，那么我们可以把变量  $v_i$  替换为最大的可能整数值，或者在没有常量上界时替换为  $\infty$ 。可以重复这个步骤对不等式不断化简，在某些情况下可以确定不等式无解。

**例 11.36** 考虑下面的不等式：

$$1 \leq v_1, v_2 \leq 10$$

$$0 \leq v_3 \leq 4$$

$$v_2 \leq v_1$$

$$v_1 \leq v_3 + 4$$

变量  $v_1$  的下界由  $v_2$  确定，而上界由  $v_3 + 4$  确定。但是，界定  $v_2$  下界的只有常量 1，界定  $v_3$  上界的只有常量 4。因此，在这些不等式中把  $v_2$  替换为 1 并把  $v_3$  替换为 4，我们得到

$$1 \leq v_1 \leq 10$$

$$1 \leq v_1$$

$$v_1 \leq 8$$

现在这个不等式组可以很容易地使用独立变量测试的方法求解。  $\square$

### 循环残数测试

现在让我们考虑每个变量的上下界都由其他变量确定的情况。在数据依赖分析中经常会碰到形如  $v_i \leq v_j + c$  的约束。这种情况可以使用 Shostak 提出的循环残数测试 (loop-residue test) 的一个简化版本来求解。这样的一组约束可以用一个有向图表示。这个图的结点标号为不等式中的变量。对于每一个约束  $v_i \leq v_j + c$ ，都有一条从结点  $v_i$  到  $v_j$  的标号为  $c$  的对应边。

我们把一条路径的权重 (weight) 定义为该路径上所有边的标号的和。图中的每条路径表示此约束系统中的一组约束的组合。也就是说，只要存在一条从  $v$  到  $v'$  的权重为  $c$  的边，我们就可以推断出  $v \leq v' + c$ 。图中的一条权重为  $c$  的环表示对环中的每个结点  $v$  都存在约束  $v \leq v + c$ 。如果我们能够在图中找到一个权重为负的环，那么就可以推断出  $v < v$ ，而这是不可能成立的。此时，我们断定不等式组无解，因此也就没有依赖关系。

我们也可以把形如  $c \leq v$  或  $v \leq c$  的约束放到循环残数测试中去，这里  $v$  是一个变量， $c$  是一个常量。我们向不等式系统引入一个新的哑变量  $v_0$ 。这个哑变量被加到每个常量上界和常量下界上。当然  $v_0$  的值一定是 0，但是因为循环残数测试只寻找图中的环，变量的实际值并不重要。为了处理常量上下界，我们把

$$v \leq c \text{ 替换为 } v \leq v_0 + c$$

$$c \leq v \text{ 替换为 } v_0 \leq v - c$$

**例 11.37** 考虑不等式

$$1 \leq v_1, v_2 \leq 10$$

$$0 \leq v_3 \leq 4$$

$$v_2 \leq v_1$$

$$2v_1 \leq 2v_3 - 7$$

变量  $v_1$  的常量上下界变成了  $v_0 \leq v_1 - 1$  和  $v_1 \leq v_0 +$

10； $v_2$  和  $v_3$  的常量界限也可以做类似的处理。然后，把最后一个约束转换成  $v_1 \leq v_3 - 4$ ，我们就得到

图 11-21 中显示的图。环  $v_1, v_3, v_0, v_1$  的权重为 -1，因此这个不等式组无解。  $\square$

### 记忆模式

因为一些简单的访问模式会在整个程序中重复出现，我们经常需要重复求解类似的数据依赖关系问题。提高数据依赖关系的处理速度的重要技术之一是使用记忆模式 (memoization)。这

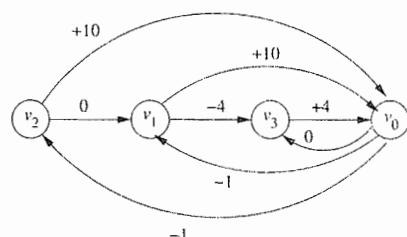


图 11-21 例子 11.37 中的约束的图形表示

个模式在生成一个问题的结果之后就把这个结果用表格记录下来。每次处理此类问题的时候，算法都会查询这个表。只有在表中找不到被处理问题的结果时才需要从头求解这个问题。

### 11.6.5 解决一般性的整数线性规划问题

现在我们描述一个解决整数线性规划问题的一般性方法。这个问题是 NP - 完全的。我们的算法使用了一个分支 - 界定方法，这种方法在最坏情况下花费的时间为指数级。但是，11.6.4 节中的启发式规则未能解决问题的情况很少出现。并且即使我们需要应用本节中的算法，也很少需要执行算法中的分支 - 界定步骤。

这个方法首先检查不等式组是否存在有理数解。这个问题是标准的线性规划问题。如果不等式组不存在有理数解，问题中的数组访问所触及的数据区域就一定不相交，因此一定不存在数据依赖关系。如果存在有理数解，我们首先试图证明存在一个整数解（通常会有这样的解）。如果不能证明这一点，我们就把这个不等式组界定的多面体分割为两个较小的问题，并递归地解决问题。

**例 11.38** 考虑下面的简单循环：

```
for (i = 1; i < 10; i++)
    Z[i] = Z[i+10];
```

访问  $Z[i]$  所触及的元素是  $Z[1], \dots, Z[9]$ ，而访问  $Z[i+10]$  所触及的元素是  $Z[11], \dots, Z[19]$ 。这两个范围并不相交，因此不存在数据依赖关系。更严格地讲，我们需要说明不存在两个动态访问  $i$  和  $i'$  满足  $1 \leq i \leq 9, 1 \leq i' \leq 9$  且  $i = i' + 10$ 。如果存在这样的整数  $i$  和  $i'$ ，那么可以用  $i' + 10$  来替代  $i$ ，并得到四个关于  $i'$  的约束： $1 \leq i' \leq 9$  和  $1 \leq i' + 10 \leq 9$ 。但是， $i' + 10 \leq 9$  蕴含  $i' \leq -1$ ，这和  $1 \leq i'$  矛盾。因此不存在这样的整数  $i$  和  $i'$ 。□

算法 11.39 描述了如何基于 Fourier-Motzkin 消除算法来确定是否可以找到一组线性不等式的整数解。

**算法 11.39 整数线性规划问题的分支界定解法。**

**输入：**一个变量  $v_1, \dots, v_n$  上的多面体  $S_n$ 。

**输出：**如果  $S_n$  有一个整数解，输出“yes”，否则输出“no”。

**方法：**图 11-22 中给出的算法。□

```

1) 对  $S_n$  应用算法 11.11，把变量
    $v_n, v_{n-1}, \dots, v_1$  按顺序通过投影消除；
2) 令  $S_i$  为通过投影消除掉  $v_{i+1}$  之后得到的多面体，其中
    $i = n-1, n-2, \dots, 0$ ；
3) if  $S_0$  为空 return “no”;
   /* 如果只涉及常量的  $S_0$  具有不可满足的约束,
      就不存在有理数解 */
4) for (i = 1; i <= n; i++) {
5)     if ( $S_i$  不包含整数解) break;
6)     令  $c_i$  为  $S_i$  中  $v_i$  的取值范围正中的整数值;
7)     把  $v_i$  替换为  $c_i$ ，修改  $S_i$ ;
8)
9)     if (i == n + 1) return “yes”;
10)    if (i == 1) return “no”;
11)    令  $S_i$  中  $v_i$  的下界和上界分别为  $l_i$  和  $u_i$ ；
12)    对  $S_n \cup \{v_i \leq l_i\}$  和  $S_n \cup \{v_i \geq u_i\}$  递归应用
          这个算法且  $S_n \cup \{v_i \geq u_i\}$  ;
13)    if (有一个结果为“yes”) return “yes” else return “no”;
```

图 11-22 寻找不等式的整数解

第 1 行到第 3 行试图找出不等式组的一个有理数解。如果没有有理数解，就没有整数解。如果找到一个有理数解，就表明这个不等式组定义了一个非空的多面体。这样的一个多面体不包含整数解的情况相对较少——要是出现这种情况，这个多面体在某些维度上必然很薄，而且位于整数点之间。

因此，第 4 行到第 9 行试图快速检查是否存在一个整数解。Fourier-Motzkin 消除算法的每一步都会产生一个多面体，其维度比前一个多面体的维度小 1。我们反向考虑这些多面体。我们从只有一个变量的多面体开始，在可能的情况下，向这个变量赋予一个大概处于它的取值范围中间的整数值。然后，我们在所有其他的多面体中用这个值来替代这个变量，把这些多面体的未知量的数目减一。不断重复这个过程，直到所有的多面体都得到处理，或者找到了一个没有整数解的变量。在前一种情况下，我们可以找到一个整数解。

如果我们甚至不能为第一个变量找到整数值，那么整个不等式组就不存在整数解（第 10 行）。否则，我们所知道的全部情况就是没有哪个整数解会包含至今为止我们为一些变量选择的特定整数值。这个结论不是决定性的。第 11 行到第 13 行表示算法的分支 - 界定步骤。如果发现变量  $v_i$  具有有理数解但是没有整数解，就把问题中的多面体分成两个多面体，第一个要求  $v_i$  必须是小于已找到的有理数的整数，第二个要求  $v_i$  必须是一个大于此有理数解的整数。如果两个多面体都没有整数解，那么就不存在依赖关系。

### 11.6.6 小结

我们已经说明一个编译器能够从数组引用中收集到的信息的主要部分和某些标准数学概念等价。给定一个访问函数  $\mathcal{F} = \langle \mathbf{F}, \mathbf{f}, \mathbf{B}, \mathbf{b} \rangle$ ：

1) 被访问的数据区域的维度由矩阵  $\mathbf{F}$  的秩给出。访问同一位置的访问空间的维度就是  $\mathbf{F}$  的零数。如果两个迭代向量的差值属于  $\mathbf{F}$  的零空间，那么这两个迭代指向同一个数组元素。

2) 同一个数组访问的具有自时间复用关系的多个迭代之间的差距是  $\mathbf{F}$  的零空间中的向量。自空间复用可以用类似的方式计算得到，但不是考虑两个迭代何时使用同一个元素，而是考虑它们何时使用同一行元素。两个访问  $\mathbf{F}\mathbf{i}_1 + \mathbf{f}_1$  和  $\mathbf{F}\mathbf{i}_2 + \mathbf{f}_2$  沿着向量  $\mathbf{d}$  的方向具有易于利用的局部性，其中  $\mathbf{d}$  是方程  $\mathbf{Fd} = (\mathbf{f}_1 - \mathbf{f}_2)$  的某个解。特别是当  $\mathbf{d}$  的方向和最内层循环对应时，即  $\mathbf{d}$  为向量  $[0, 0, \dots, 0, 1]$  时，如果数组是按行存放的，那么就会存在空间局部性。

3) 数据依赖关系问题——两个引用是否可能指向同一个位置——和整数线性规划等价。两个访问函数之间具有数据依赖关系的条件是存在值为整数的向量  $\mathbf{i}$  和  $\mathbf{i}'$ ，使得  $\mathbf{Bi} \geq 0$ ,  $\mathbf{B}'\mathbf{i}' \geq 0$ ，并且  $\mathbf{Fi} + \mathbf{f} = \mathbf{F}'\mathbf{i}' + \mathbf{f}'$ 。

### 11.6.7 11.6 节的练习

**练习 11.6.1：** 找出下列整数集合的 GCD：

- 1)  $\{16, 24, 56\}$ 。
- 2)  $\{-45, 105, 240\}$ 。
- 3)  $\{84, 105, 180, 315, 350\}$ 。

**练习 11.6.2：** 对于下面的循环

```
for (i = 0; i < 10; i++)
    A[i] = A[10-i];
```

指出所有的

- 1) 真依赖关系（即写运算后跟着对同一个位置的读运算）。
- 2) 反依赖关系（即读运算后跟着对同一个位置的写运算）。
- 3) 输出依赖关系（即写运算后跟着对同一个位置的另一个写运算）。

! 练习 11.6.3: 在介绍欧几里得算法的部分中, 我们未经证明就给出了一些断言。证明下面的每一个断言:

1) 该部分所述的欧几里得算法总是能够工作。特别地,  $\gcd(b, c) = \gcd(a, b)$ , 其中  $c$  是  $a/b$  的非零余数。

2)  $\gcd(a, b) = \gcd(a, -b)$ 。

3) 当  $n > 2$  时,  $\gcd(a_1, a_2, \dots, a_n) = \gcd(\gcd(a_1, a_2), a_3, \dots, a_n)$ 。

4) GCD 实际上是一个整数集合上函数, 即整数的顺序并不重要。说明 GCD 的交换率:  $\gcd(a, b) = \gcd(b, a)$ 。然后证明更加困难的结论, 即 GCD 的结合律:  $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$ 。最后, 说明上述这些定律蕴含了下面的性质: 不管按照什么样的顺序来计算各个整数对的 GCD, 得到的一个整数集合的 GCD 总是相同的。

5) 如果  $S$  和  $T$  都是整数集合, 那么  $\gcd(S \cup T) = \gcd(\gcd(S), \gcd(T))$ 。

! 练习 11.6.4: 找出例 11.33 中的第二个丢番图方程的另一个解。

练习 11.6.5: 在下面的情况中应用独立变量测试。循环嵌套结构是

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        for (k=0; k<100; k++)
```

在嵌套结构中是一个关于数组访问的赋值语句。确定下面的每一个语句是否会引起某些数据依赖关系。

1)  $A[i, j, k] = A[i+100, j+100, k+100]$

2)  $A[i, j, k] = A[j+100, k+100, i+100]$

3)  $A[i, j, k] = A[j-50, k-50, i-50]$

4)  $A[i, j, k] = A[i+99, k+100, j]$

练习 11.6.6: 在下面的约束中, 通过把  $x$  替换为  $y$ (原文如此, 译者注) 的常量下界来消除  $x$ 。

$$1 \leq x \leq y - 100$$

$$3 \leq x \leq 2y - 50$$

练习 11.6.7: 对下面的约束集合应用循环残数测试:

$$0 \leq x \leq 99 \quad y \leq x - 50$$

$$0 \leq y \leq 99 \quad z \leq y - 60$$

$$0 \leq z \leq 99$$

练习 11.6.8: 对下面的约束集合应用循环残数测试:

$$1 \leq x \leq 99 \quad y \leq x - 50$$

$$0 \leq y \leq 99 \quad z \leq y + 40$$

$$0 \leq z \leq 99 \quad x \leq z + 20$$

练习 11.6.9: 对下面的约束集合应用循环残数测试:

$$0 \leq x \leq 99 \quad y \leq x - 100$$

$$0 \leq y \leq 99 \quad z \leq y + 60$$

$$0 \leq z \leq 99 \quad x \leq z + 50$$

## 11.7 寻找无同步的并行性

我们已经得到了关于仿射数组访问, 访问之间对数据的复用, 以及它们之间的依赖关系的理论。现在我们将应用这些理论来对实际程序进行并行化及优化处理。如 11.1.4 节中所讨论的,

在找到并行性的同时保证处理器之间通信量的最小化是很重要的。我们首先研究如何在完全不允许处理器之间进行通信或同步的情况下实现并行化的问题。这个约束可能看起来像是一个纯学术的练习，我们有多大的机会会碰到具有这种形式的并行性的程序或过程？实际上，在现实生活中存在很多这样的程序，因此解决这个并行化问题的算法本身就是有用的。另外，可以扩展解决这个问题时使用的概念以处理同步和通信。

### 11.7.1 一个介绍性的例子

图 11-23 中显示的是从一个 5000 行的 Fortran 代码程序中摘录的并以 C 语言表示的程序片段。为清晰起见，代码中仍然保留了 Fortran 风格的数组访问语法。原来的程序实现了用来解决三维欧拉方程的多重网格算法。这个程序的大部分运行时间都花费在少数几个如图所示的子程序上。它是很多数值程序的典型代表。这些数值程序经常由很多处在不同嵌套层次上的 for 循环组成。它们包含了很多数组访问，所有数组访问的下标都是外围循环下标的仿射表达式。为了使这个例子比较简短，我们已经从原来的程序中删除了一些具有类似性质的代码行。

```

for (j = 2; j <= jl; j++) {
    for (i = 2, i <= il, i++) {
        AP[j,i]      = ...;
        T            = 1.0/(1.0 + AP[j,i]);
        D[2,j,i]     = T*AP[j,i];
        DW[1,2,j,i] = T*D[1,2,j,i];
    }
    for (k = 3; k <= kl-1; k++)
        for (j = 2; j <= jl; j++)
            for (i = 2; i <= il; i++) {
                AM[j,i]      = AP[j,i];
                AP[j,i]      = ...;
                T            = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...
                D[k,j,i]     = T*AP[j,i];
                DW[1,k,j,i] = T*(DW[1,k,j,i] + DW[1,k-1,j,i])...
            }
        ...
        for (k = kl-1; k >= 2; k--)
            for (j = 2; j <= jl; j++)
                for (i = 2; i <= il; i++)
                    DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
    }
}

```

图 11-23 一个多重网格算法的代码片断

图 11-23 的代码在一个标量变量  $T$  和一些具有不同维度的多个数组上运行。我们首先来看一下对变量  $T$  的使用。因为一个循环中的每个迭代使用同一个变量  $T$ ，我们不能并行执行这些迭代。但是  $T$  只是用于存放在一个迭代中使用两次的公共子表达式的值。在图 11-23 中的前两个循环嵌套中，最内层循环的各个迭代向  $T$  中写入一个值，然后立刻在同一个迭代中两次使用这个值。我们可以把对  $T$  的每次使用替换为前面对  $T$  的赋值语句的右部表达式，从而在不改变程序语义的前提下消除依赖关系。我们也可以把标量  $T$  替换为一个数组。然后我们让每个迭代  $(j, i)$  使用它自己的数组元素  $T[j, i]$ 。

经过这样的修改，每个赋值语句中对一个数组元素的计算只依赖于最后两个下标分量值（分别是  $j$  和  $i$ ）相同的其他数组元素。因此，我们可以把对各个数组的第  $(j, i)$  个元素进行操作的所有运算组合成为一个计算单元，并按照原来的串行顺序执行它们。这个修改产生了  $(jl-1) \times (il-1)$  个相互独立的计算单元。请注意，原程序里第二和第三个循环嵌套结构涉及下标为  $k$  的第三个循环。但是，因为具有同样的  $j$  和  $i$  值的动态访问之间不存在依赖关系，所以可以安全地在  $j$  和  $i$  的

循环内部——就是说在一个计算单元中——执行  $k$  的循环。

知道这些计算单元是独立的，我们就可以对代码进行多个合法的转换。比如，一个单处理器系统可以不按照原来的代码执行，而是逐个执行独立的运算单元，最终完成同样的计算工作。转换所得代码显示在图 11-24 中。这个代码具有更好的时间局部性，因为计算中生成的结果立刻就被用掉了。

```

for (j = 2; j <= j1; j++) {
    for (i = 2; i <= i1; i++) {
        AP[j,i]      = ...;
        T[j,i]       = 1.0/(1.0 + AP[j,i]);
        D[2,j,i]     = T[j,i]*AP[j,i];
        DW[1,2,j,i] = T[j,i]*DW[1,2,j,i];
        for (k = 3; k <= k1-1; k++) {
            AM[j,i]   = AP[j,i];
            AP[j,i]   = ...;
            T[j,i]     = ...AP[j,i] - AM[j,i]*D[k-1,j,i]...
            D[k,j,i]   = T[j,i]*AP[j,i];
            DW[1,k,j,i] = T[j,i]*(DW[1,k,j,i] + DW[1,k-1,j,i])...
        }
        ...
        for (k = k1-1; k >= 2; k--)
            DW[1,k,j,i] = DW[1,k,j,i] + D[k,j,i]*DW[1,k+1,j,i];
    }
}

```

图 11-24 经过改写的图 11-23 的代码，最外层是并行循环

这些独立的计算单元也可以被分配给不同的处理器并行执行。这些处理器之间不需要任何同步或通信。因为有  $(j_1 - 1) \times (i_1 - 1)$  个相互独立的计算单元，我们最多可以利用  $(j_1 - 1) \times (i_1 - 1)$  个处理器。我们可以按照二维数组的方式组织处理器，每个处理器的 ID 是  $(j, i)$ ，其中  $2 \leq j \leq j_1, 2 \leq i \leq i_1$ 。由各个处理器执行的 SPMD 程序就是图 11-24 中的内层循环的循环体。

上面的例子说明了寻找无同步的并行性的基本方法。我们首先把计算任务分解成尽可能多的独立单元。这个分解揭示了可行的调度选择。然后，我们依照拥有的处理器数目把这些计算单元分配给各个处理器。最后，我们生成一个在各个处理器上执行的 SPMD 程序。

### 11.7.2 仿射空间分划

如果一个循环嵌套结构内部具有  $k$  个可并行化的循环，那么这个循环嵌套结构就有  $k$  度的并行性。一个可并行化循环的不同迭代之间不存在数据依赖关系。比如，图 11-24 中的代码就具有 2 度并行性。把具有  $k$  度并行性的计算任务分配给一个  $k$  维的处理器阵列是很方便的。

一开始，我们将假设处理器阵列的每个维度上的处理器数目和相应循环的迭代个数一样多。在找到所有这些独立计算单元后，我们将把这些“虚拟”处理器映射到实际的处理器上。在实践中，每个处理器要负责相当多的迭代，否则就没有足够的工作量来分摊并行化所带来的额外开销。

我们把需要并行化的程序分解成为类似于三地址语句这样的基本语句。对于每个语句，我们找出一个仿射空间分划 (affine space partition)。这个分划把这些语句的各个动态实例映射到一个处理器 ID。这些动态实例使用其循环下标来标记。

**例 11.40** 如上面所讨论的，图 11-24 的代码具有 2 度的并行性。我们把处理器阵列看作一个二维空间。令  $(p_1, p_2)$  为这个阵列中的一个处理器的 ID。在 11.7.1 节中所讨论的并行化方案可以使用一个简单的仿射分划函数来描述。在第一个循环嵌套结构中的所有语句都有下面的仿射分划：

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

在第二个和第三个循环嵌套结构中的所有语句共享如下的相同的仿射分划：

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ j \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
□

寻找无同步并行性的算法由三个步骤组成：

1) 为程序中的每个语句寻找一个能够最大化并行性度数的仿射分划。请注意，我们通常把语句(而不是单个访问)作为计算的单元。对于一个语句中的所有访问必须应用同样的仿射分划。这种对访问的分组方法是有意义的，因为在同一个语句中的访问之间几乎总是存在依赖关系。

2) 在处理器之间分配由第1步得到的独立计算单元，并选择在每个处理器上执行的各个步骤之间的交替顺序关系。这个分配主要考虑局部性。

3) 生成一个将在各个处理器上执行的SPMD程序。

下面我们将讨论如何寻找仿射分划函数，如何生成一个顺序程序来串行执行各个分划，以及如何生成一个在不同处理器上执行各个分划的SPMD程序。在从11.8节到11.9.9节讨论了如何处理带有同步的并行性之后，我们将在11.10节回到上面的第2步，并讨论如何针对单处理器和多处理器系统进行局部性优化。

### 11.7.3 空间分划约束

因为要求没有通信，所以每一对具有数据依赖关系的运算都必须被分配在同一个处理器上。我们把这些约束称为“空间分划约束”。任何满足这些约束的映射所创建的分划都是相互独立的。请注意，只要把所有运算都放到一个分划单元，就可以满足这样的约束。遗憾的是，这样的“解”没有给出任何并行性。我们的目标是在满足这些空间分划约束的同时得到尽可能多的独立分划。也就是说，只有在必要的时候才会把不同的运算放到同一个处理器上。

当我们限制自己只考虑仿射分划时，可以将并行性的度数(即维度)最大化，而不是将独立单元的数目最大化。如果我们使用分段(piecewise)仿射分划，有时有可能创建出更多的独立单元。一个分段仿射分划把单个访问的实例分割成为不同的集合，并允许对每个集合使用不同的仿射分划。但是这里我们不考虑这样的选项。

正式地讲，一个程序的仿射分划是无同步的(synchronization free)当且仅当对于两个具有数据依赖关系的(不一定不同的)访问，即在循环嵌套结构 $d_1$ 中的语句 $s_1$ 中的访问 $\mathcal{F}_1 = \langle F_1, f_1, B_1, b_1 \rangle$ 和循环嵌套结构 $d_2$ 中的语句 $s_2$ 中的访问 $\mathcal{F}_2 = \langle F_2, f_2, B_2, b_2 \rangle$ ，对语句 $s_1$ 和 $s_2$ 的分划 $\langle C_1, c_1 \rangle$ 和 $\langle C_2, c_2 \rangle$ 满足下面的空间分划约束(space-partition constraint)：

- 对于所有满足下列条件的 $Z^{d_1}$ 中的 $i_1$ 和 $Z^{d_2}$ 的 $i_2$ ：

- 1)  $B_1 i_1 + b_1 \geq 0$
- 2)  $B_2 i_2 + b_2 \geq 0$
- 3)  $F_1 i_1 + f_1 = F_2 i_2 + f_2$

$C_1 i_1 + c_1 = C_2 i_2 + c_2$ 一定成立。

并行化算法的目标是为每个语句找出满足这些约束的具有最高秩的分划。

图11-25中的图说明了空间分划约束的本质。假设有两个静态访问分别处于两个循环嵌套结构中，它们的下标向量分别为 $i_1$ 和 $i_2$ 。假设它们共同访问了至少一个数组元素，且至少其中之一

是写运算，那么它们之间具有依赖关系。根据仿射访问函数  $F_1 i_1 + f_1$  和  $F_2 i_2 + f_2$ ，该图显示了在两个循环中恰巧访问同一个数组元素的动态访问。除非这两个静态访问的仿射分划  $C_1 i_1 + c_1$  和  $C_2 i_2 + c_2$  把它们的动态访问分配到同一个处理器上，否则不同处理器之间必须进行同步。

如果我们选择一个仿射分划，它的秩为所有语句的秩的最大值，那么就得到了最大可能的并行性。但是，在这种划分下，有些处理器有时可能会空闲，而其他处理器却忙于执行那些具有较小秩的仿射分划的语句。如果执行这些语句的时间相对较短，这种情况还是可接受的。否则，我们可以选择秩小于最大可能值的仿射分划，只要这个分划的秩大于 0 即可。

在例 11.41 中，我们给出了一个用于说明这个技术的功能的小程序。实际应用通常要比这个程序简单，但是它们的边界条件可能和这里显示的一些问题类似。我们将在本章的各个部分使用这个例子来说明下面的事实：具有仿射访问的程序具有相对简单的空间分划约束，这些约束可以通过标准线性代数技术来解决，并且最终需要的 SPMD 程序能够从仿射分划中机械化地生成。

**例 11.41** 这个例子说明了我们如何把一个程序的空间分划约束用公式表达出来。这个程序显示在图 11-26 中，它由具有两个语句  $s_1$  和  $s_2$  的小循环嵌套结构组成。

```

for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }

```

图 11-26 一个用以说明相互依赖的长运算链的循环嵌套结构

我们在图 11-27 中显示了程序中的数据依赖关系。在图中，每个黑点表示语句  $s_1$  的一个实例，而每个白点表示了语句  $s_2$  的一个实例。在坐标  $(i,j)$  处的点表示该语句在下标变量的取值为  $(i,j)$  时的实例。但是请注意， $s_2$  的实例位于对应于相同  $(i,j)$  对的  $s_1$  的实例的下方，因此图中对应于  $j$  的垂直刻度要比对应于  $i$  的水平刻度长。

请注意， $X[i,j]$  是由  $s_1(i,j)$  写入的， $s_1(i,j)$  就是语句  $s_1$  对应于循环下标值  $i$  和  $j$  的实例。之后它被  $s_2(i,j+1)$  读出，因此  $s_1(i,j)$  必须在  $s_2(i,j+1)$  之前执行。这个事实解释了图中从黑点到白点的垂直箭头。类似地， $Y[i,j]$  被  $s_2(i,j)$  写入再

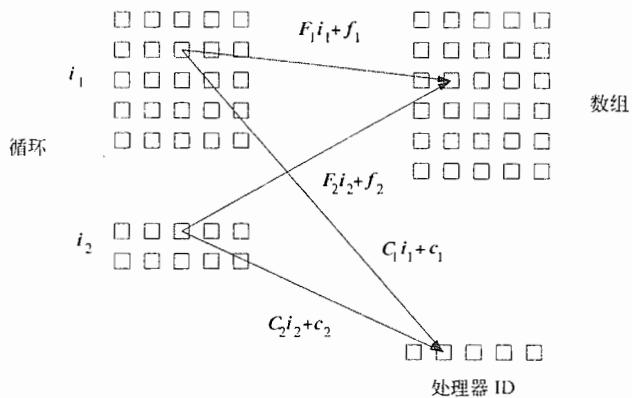


图 11-25 空间分划约束

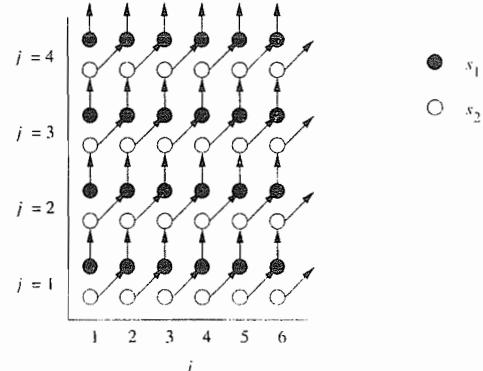


图 11-27 例 11.41 的代码中的依赖关系

由  $s_1(i+1, j)$  读出，这个事实解释了从白点到黑点的箭头。

从图中很容易看出，代码可以被并行化为无同步关系的几个部分，方法是把各个相互依赖的运算链分配给同一个处理器。但是，写出一个实现这样的映射方案的 SPMD 程序不容易。在原来的程序中每个循环有 100 个迭代，因此存在 200 个运算链。在这些运算链中，其中的一半由  $s_1$  开始并以  $s_1$  结束，另一半从  $s_2$  开始并以  $s_2$  结束。这些链的长度从 1 ~ 100 个迭代不等。

因为有两个语句，所以我们需要为每个语句寻找一个仿射分划。我们只需要表示出一维仿射分划的空间分划约束。这些约束稍后将由试图寻找所有独立的一维仿射分划的解方法所使用。这个方法还将把这些一维分划组合起来得到多维仿射分划。因此，我们可以把每个语句的仿射分划表示为一个  $1 \times 2$  的矩阵和一个  $1 \times 1$  的向量，并把下标向量  $[i, j]$  转换成为一个处理器的编号。令  $\langle [C_{11} C_{12}], [c_1] \rangle, \langle [C_{21} C_{22}], [c_2] \rangle$  分别为语句  $s_1$  和  $s_2$  的一维仿射分划。

我们将应用六个数据依赖测试：

- 1) 语句  $s_1$  中的写访问  $X[i, j]$  和其自身之间的依赖关系。
- 2) 语句  $s_1$  中的写访问  $X[i, j]$  和读访问  $X[i, j]$  之间的依赖关系。
- 3) 语句  $s_1$  中的写访问  $X[i, j]$  和语句  $s_2$  中的读访问  $X[i, j-1]$  之间的依赖关系。
- 4) 语句  $s_2$  中的写访问  $Y[i, j]$  和其自身之间的依赖关系。
- 5) 语句  $s_2$  中的写访问  $Y[i, j]$  和读访问  $Y[i, j]$  之间的依赖关系。
- 6) 语句  $s_2$  中的写访问  $Y[i, j]$  和语句  $s_1$  中的写访问  $Y[i-1, j]$  之间的依赖关系。

我们可以看到，这些依赖关系测试都很简单，而且高度重复。这个代码中出现的依赖关系发生在第(3)种情况下访问  $X[i, j]$  和  $X[i, j-1]$  的实例之间以及第(6)种情况下访问  $Y[i, j]$  和  $Y[i-1, j]$  的实例之间。

由语句  $s_1$  中的  $X[i, j]$  和  $s_2$  中的  $X[i, j-1]$  之间的数据依赖关系而导致的空间分划约束可以表示成下列各项：

对于所有满足下面条件的  $(i, j)$  和  $(i', j')$

$$\begin{aligned} 1 \leq i \leq 100 & \quad 1 \leq j \leq 100 \\ 1 \leq i' \leq 100 & \quad 1 \leq j' \leq 100 \\ i = i' & \quad j = j' - 1 \end{aligned}$$

我们有

$$[C_{11} \quad C_{12}] \begin{bmatrix} i \\ j \end{bmatrix} + [c_1] = [C_{21} \quad C_{22}] \begin{bmatrix} i' \\ j' \end{bmatrix} + [c_2]$$

也就是说，前四个条件是说  $(i, j)$  和  $(i', j')$  处于这个循环嵌套结构的迭代空间中，后两个条件是说动态访问  $X[i, j]$  和  $X[i, j-1]$  触及同一个数据元素。我们可用类似的方法得到针对语句  $s_2$  中的访问  $Y[i-1, j]$  和语句  $s_1$  中的访问  $Y[i, j]$  的空间分划约束。□

#### 11.7.4 求解空间分划约束

一旦抽取得空间分划约束之后，我们就可以使用标准线性代数技术来寻找满足这个约束的仿射分划。让我们首先说明如何找出例 11.41 的解。

**例 11.42** 我们可以使用下面的步骤来找出例 11.41 的仿射分划：

- 1) 建立例 11.41 中显示的空间分划约束。我们在决定数据依赖关系的时候使用了循环界限，但是在算法的其余部分不再使用循环界限。
- 2) 在不等式中的未知变量是  $i, i', j, j', C_{11}, C_{12}, c_1, C_{21}, C_{22}$  和  $c_2$ 。根据访问函数可得到等式  $i = i'$  和  $j = j' - 1$ 。使用这些等式来减少未知量的数目。我们使用高斯消除法来完成这个工

作，它把四个变量减少为两个变量，也就是说  $t_1 = i = i'$  和  $t_2 = j = j' - 1$ 。分划的等式变为

$$[C_{11} - C_{21} \quad C_{12} - C_{22}] \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + [c_1 - c_2 - C_{22}] = 0$$

3) 上面的等式对于所有的  $t_1$  和  $t_2$  组合都成立。因此必然有下面的结论：

$$C_{11} - C_{21} = 0$$

$$C_{12} - C_{22} = 0$$

$$c_1 - c_2 - C_{22} = 0$$

如果我们对访问  $Y[i-1, j]$  和  $Y[i, j]$  之间的约束执行同样的处理步骤，我们得到

$$C_{11} - C_{21} = 0$$

$$C_{12} - C_{22} = 0$$

$$c_1 - c_2 + C_{21} = 0$$

把所有这些约束一起进行简化，我们得到下面的关系：

$$C_{11} = C_{21} = -C_{22} = -C_{12} = c_2 - c_1$$

4) 找出那些只涉及系数矩阵中的未知量的等式的所有独立解。在这一步中忽略常量向量中的未知量。在系数矩阵中只有一个独立的选择，因此我们寻找的仿射分划的秩最多为一。为了使得分划尽量简单，我们把  $C_{11}$  设置为 1。我们不能把 0 赋值给  $C_{11}$ ，因为这会建立一个零秩的系数矩阵。零秩矩阵会把所有的迭代都映射到同一个处理器上。由  $C_{11} = 1$  可得  $C_{21} = 1$ ， $C_{22} = -1$ ， $C_{12} = -1$ 。

5) 找出常数项。我们知道常数项之间的差  $c_2 - c_1$  必须是 -1。但是我们必须选择实际的值。为了使分划简单，我们选择  $c_2 = 0$ ，因此  $c_1 = -1$ 。

令  $p$  为执行迭代  $(i, j)$  的处理器的 ID。这个仿射分划用  $p$  表示就是

$$s_1 : [p] = [1 \quad -1] \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$$

$$s_2 : [p] = [1 \quad -1] \begin{bmatrix} i \\ j \end{bmatrix} + [0]$$

也就是说， $s_1$  的第  $(i, j)$  个迭代被分配给处理器  $p = i - j - 1$ ；而  $s_2$  的第  $(i, j)$  个迭代被分配给处理器  $p = i - j$ 。□

**算法 11.43** 找出一个程序的具有最高秩的无同步仿射分划。

**输入：**一个带有仿射数组访问的程序。

**输出：**一个分划。

**方法：**执行下列步骤：

1) 找出程序中所有的具有数据依赖关系的访问对。对于每一对具有数据依赖关系的访问：嵌套在循环  $d_1$  中的语句  $s_1$  的访问  $\mathcal{F}_1 = \langle F_1, f_1, B_1, b_1 \rangle$  和嵌套在循环  $d_2$  中的语句  $s_2$  的访问  $\mathcal{F}_2 = \langle F_2, f_2, B_2, b_2 \rangle$ 。令  $\langle C_1, c_1 \rangle$  和  $\langle C_2, c_2 \rangle$  分别是语句  $s_1$  和  $s_2$  的（当前未知的）分划。相应的空间分划约束表明对于分别处于各自循环界限中的  $i_1$  和  $i_2$ ，如果

$$F_1 i_1 + f_1 = F_2 i_2 + f_2$$

那么

$$C_1 i_1 + c_1 = C_2 i_2 + c_2$$

我们将扩展迭代的域，使之包含  $Z^{d_1}$  中的所有  $i_1$  和  $Z^{d_2}$  中的所有  $i_2$ 。也就是说，假设所有的界限都

是从负无穷大到正无穷大。这样的假设是有道理的，因为一个仿射分划不能利用如下的性质：一个下标变量的取值范围是一个有限整数集合。

2) 对于每一对相互依赖的访问，我们减少其下标向量中的未知量的数目。

④ 请注意  $\mathbf{F}i + \mathbf{f}$  和向量

$$[\mathbf{F} \quad \mathbf{f}] \begin{bmatrix} i \\ 1 \end{bmatrix}$$

相同。也就是说，通过在列向量  $i$  的底部加上一个额外的分量 1，我们可以使列向量  $f$  成为附加到矩阵  $F$  中的最后一列。这样，可以把访问函数的等式  $\mathbf{F}_1 i_1 + \mathbf{f}_1 = \mathbf{F}_2 i_2 + \mathbf{f}_2$  改写为

$$[\mathbf{F}_1 - \mathbf{F}_2 \quad (\mathbf{f}_1 - \mathbf{f}_2)] \begin{bmatrix} i_1 \\ i_2 \\ 1 \end{bmatrix} = 0$$

⑤ 一般来说，上面的等式具有多个解。但是，我们仍然可以使用高斯消除法尽可能地求解这个关于分量  $i_1$  和  $i_2$  的方程组。也就是说，尽量多地消除变量，直到只剩下无法消除的变量为止。最后得到的  $i_1$  和  $i_2$  的解将具有如下形式

$$\begin{bmatrix} i_1 \\ i_2 \\ 1 \end{bmatrix} = \mathbf{U} \begin{bmatrix} t \\ 1 \end{bmatrix}$$

其中  $\mathbf{U}$  是一个上三角形矩阵， $t$  是一个由取值范围为所有整数的自由变量组成的向量。

⑥ 我们可以使用步骤 2④ 中的技巧来改写关于分划的等式。用步骤 2⑤ 的结果替代向量  $(i_1, i_2, 1)$ ，我们可以把关于分划的约束写成

$$[\mathbf{C}_1 - \mathbf{C}_2 \quad (\mathbf{c}_1 - \mathbf{c}_2)] \mathbf{U} \begin{bmatrix} t \\ 1 \end{bmatrix} = 0$$

3) 舍弃和分划无关的变量。上面的等式对所有的  $t$  都成立的条件是

$$[\mathbf{C}_1 - \mathbf{C}_2 \quad (\mathbf{c}_1 - \mathbf{c}_2)] \mathbf{U} = 0$$

把这些等式改写成  $Ax = 0$  的形式，其中  $x$  是一个由仿射分划的所有未知系数组成的向量。

4) 找出这个仿射分划的秩并求解系数矩阵。因为一个仿射分划的秩和分划中常量项的值无关，所以消除所有来自于  $c_1$  和  $c_2$  等常量向量的未知量，从而把  $Ax = 0$  替换为经过简化的约束  $A'x' = 0$ 。找出  $A'x' = 0$  的解，并把它们表示为  $B$ ，也就是可以生成  $A'$  的零空间的一组基本向量的集合。

5) 找出常量项。从  $B$  中的每个基本向量中得到所求仿射分划的一行，并使用  $Ax = 0$  来获得常量项。□

请注意，步骤 3 忽略了因循环界限而加在变量  $t$  上的约束。由此而得到的仿射分划约束会更加严格，因此这个算法一定是安全的。也就是说，我们在假设  $t$  可取任意值的情况下生成了对  $C$  和  $c$  的约束。可以想象，如果考虑到对变量  $t$  的约束会使得  $t$  不可能取某些值，那么可能还会存在一些  $C$  和  $c$  的其他解。没有搜寻这样的解会使我们失去一些优化的机会，但不会使得被处理的程序所完成工作和原程序不同。

### 11.7.5 一个简单的代码生成算法

算法 11.43 生成了能够把计算任务分割成独立分划单元的仿射分划。因为分划单元之间是相互独立的，因此它们可以被任意分配到不同处理器上。一个处理器可以被分配给多个分划单元，并且处理器可以交替执行分配给它的分划单元。但是每个分划单元中的运算需要顺序执行，

这是因为它们之间通常具有数据依赖关系。

为一个给定的仿射分划生成一个正确的程序相对容易一些。我们首先介绍算法 11.45。这是一个简单的代码生成方法，它能够为单处理器系统生成顺序地执行各个独立分划单元的代码。这样的代码优化了时间局部性，因为对相同数组元素的多次数组访问在时间上相当靠近。不仅如此，这个代码很容易被转换成为一个 SPMD 程序，这个程序在不同的处理器上执行各个分划单元。遗憾的是，这样生成的代码是低效的，我们下一步将讨论使这些代码高效执行的优化方法。

我们的基本思想如下。我们已经知道了一个循环嵌套结构的各个下标变量的界限，也在算法 11.43 中确定了某个语句  $s$  中的访问的分划。假设我们希望生成一个能够顺序执行各个处理器上的动作的代码，那么可以创建一个最外层的循环，该循环遍历各个处理器 ID。也就是说，这个循环的每个迭代执行了分配给某个处理器 ID 的运算。原来的程序作为这个循环的循环体被插入到代码中。另外，对代码中的每个运算都增加了一个测试条件作为卫式，以保证每个处理器只执行赋予它的运算。通过这个方法，我们保证一个处理器按照原来的顺序执行了所有赋予它的指令。

**例 11.44** 我们希望生成能够顺序执行例 11.41 中的各个独立分划单元的代码。原来的顺序程序来自图 11-26，我们在图 11-28 中重复这段代码。

在例 11.42 中，仿射分划算法找到了度数为一的并行性。因此，处理器空间可以用单个变量  $p$  表示。请回忆一下，我们在那个例子中选择了如下的仿射分划，对于所有满足  $1 \leq i \leq 100$  和  $1 \leq j \leq 100$  的下标变量  $i$  和  $j$ ：

- 1) 将语句  $s_1$  的实例  $(i, j)$  分配给处理器  $p = i - j - 1$ 。
- 2) 将语句  $s_2$  的实例  $(i, j)$  分配给处理器  $p = i - j$ 。

我们可以分三步生成代码：

1) 对于每个语句，找出所有参与该语句计算的处理器的 ID。我们把约束  $1 \leq i \leq 100$  及  $1 \leq j \leq 100$  和等式  $p = i - j - 1$  和  $p = i - j$  中的一个组合起来，并通过投影消除  $i$  和  $j$ ，得到新的约束。

① 如果我们使用为语句  $s_1$  得到的函数  $p = i - j - 1$ ，那么得到  $-100 \leq p \leq 98$ 。

② 如果我们使用语句  $s_2$  的函数  $p = i - j$ ，那么得到  $-99 \leq p \leq 99$ 。

2) 找出所有参与了任一语句的计算工作的处理器的 ID。我们取这些范围的并集，得到  $-100 \leq p \leq 99$ ，这些界限足以覆盖语句  $s_1$  和  $s_2$  的所有的实例。

3) 生成能够顺序遍历每个分划单元中的计算工作的代码。图 11-29 中显示的代码有一个外层循环，它迭代遍历了所有参与计算的分划单元的 ID(第 1 行)。在第 2 行和第 3 行，每个分划单元都会经历原串行程序生成所有迭代下标的过 程，然后它可以选出应该由处理器  $p$  执行的迭代。第 4 行和第 6 行的代码保证了只有当处理器  $p$  应该执行语句  $s_1$  和  $s_2$  时，这两个语句才可以执行。

虽然生成的代码是正确的，但是特别低效。首先，虽然每个处理器最多执行 99 个迭代的计算任务，但是它生成了  $100 \times 100$  个迭代的循环下标值，比必须的下标值数目多了一个量级。其次，最内层循环的每一个加法都带有一个条件测试，在原有开销上又增加了一个常量因子。这两种低效率问题的处理将分别在 11.7.6 节和 11.7.7 节中处理。□

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        Y[i,j] = Y[i,j] + X[i,j-1]; /* (s2) */
    }
```

图 11-28 重复图 11-26

```

1)   for (p = -100; p <= 99; p++)
2)     for (i = 1; i <= 100; i++)
3)       for (j = 1; j <= 100; j++) {
4)         if (p == i-j-1)
5)           X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
6)         if (p == i-j)
7)           Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
8)       }

```

图 11-29 图 11-28 中的代码的简单改写，它在执行时遍历处理器空间

虽然图 11-29 的代码看起来被设计成在单处理器上执行的代码，但我们将把第 2 行到第 8 行的内层循环拿出来在 200 个不同的处理器上执行它们。每个处理器都有一个不同的从  $-100 \sim 99$  的  $p$  值。只要我们安排得当，使得每个处理器都知道各自负责  $p$  的哪些值，并且只执行对应于这些值的第 2 行到第 8 行代码，那么就可以在少于 200 个处理器上分划内层循环的计算。

#### 算法 11.45 创建顺序执行一个程序的各个分划单元的代码。

**输入：**一个具有仿射数组访问的程序  $P$ 。程序中的每个语句  $s$  具有形如  $B_s i + b_s \geq 0$  的界限，其中  $i$  是  $s$  所在循环嵌套结构的循环下标变量的向量。每个语句  $s$  还附有一个分划  $C_s i + c_s = p$ ，其中  $p$  是一个由表示处理器 ID 的变量组成的  $m$  维向量。 $m$  是程序  $P$  中的各个语句的分划的秩的最大值。

**输出：**一个等价于  $P$  的程序，但是它在处理器空间上(而不是原来的循环下标上)进行迭代遍历。

**方法：**执行下列各步骤：

- 1) 对于每个语句，使用 Fourier-Motzkin 消除法从界限中通过投影消除所有的循环下标变量。
- 2) 使用算法 11.13 来决定分划单元 ID 的界限。

3) 为处理器空间的  $m$  个维度中的每一维生成一个循环。令  $p = [p_1, p_2, \dots, p_m]$  为这些循环的变量的向量。也就是说，对于处理器空间的每一个维度都有一个变量。每个循环变量  $p_i$  遍历程序  $P$  中所有语句的分划空间的并集。

请注意，分划空间的并集不一定是凸的。为了保证算法简单，我们不必做到只枚举那些确实有计算任务的分划单元；我们可以把每个  $p_i$  的下界设置为由各个语句确定的全部下界的最小值，并把每个  $p_i$  的上界设置为由各个语句确定的全部上界的最大值。因此  $p$  的某些取值可能没有运算。

由每个分划单元执行的代码就是原来的串行程序。但每个语句都带有一个断言作为卫式条件，以保证只有属于这个分划单元的代码才会被执行。□

我们很快就会给出算法 11.45 的一个例子。但是请记住，要得到典型例子的优化代码还有很多工作要做。

#### 11.7.6 消除空迭代

现在我们讨论生成高效 SPMD 代码所必须的两个转换中的第一个。每个处理器执行的代码循环遍历原程序中的所有迭代，并选择应该由它执行的运算。如果代码具有  $k$  度的并行性，这么做的后果就是每个处理器的工作量增大了  $k$  个数量级。第一个转换的目的是收紧循环的界限以便消除所有的空迭代。

首先我们逐条考虑程序中的语句。由一个分划单元执行的一个语句的迭代空间是原来的迭代空间加上仿射分划给出的约束。我们可以把算法 11.13 应用到新的迭代空间，为每个语句生成一个紧致的界限。新的下标向量和原顺序程序的下标向量类似，但加上了处理器 ID 作为最外层

的下标。请注意，这个算法会为每个下标生成以外围下标表示的紧致的界限。

在找到不同语句的迭代空间之后，我们按照逐个循环的方式把它们组合起来，使得下标的界限为各个语句的界限的并集。如下面的例 11.46 所示，最后有些循环可能只有一个迭代，我们可以简单地消除这个循环，并直接把循环下标设置为该迭代对应的值。

**例 11.46** 对于图 11-30a 中的循环，算法 11.43 将生成仿射分划

$$s_1: p = i$$

$$s_2: p = j$$

算法 11.45 将生成图 11-30b 中的代码。对语句  $s_1$  应用算法 11.13 得到界限  $p \leq i \leq p$ ，即  $i = p$ 。类似地，这个算法确定对于语句  $s_2$  有  $j = p$ 。这样我们就得到了图 11-30c 所示的代码。对变量  $i$  和  $j$  的传播将会消除不必要的测试而得到图 11-30d 中的代码。□

```
: for (i=1; i<=N; i++)
    Y[i] = Z[i]; /* (s1) */
  for (j=1; j<=N; j++)
    X[j] = Y[j]; /* (s2) */
```

a) 初始代码

```
for (p=1; p<=N; p++) {
    for (i=1; i<=N; i++)
        if (p == i)
            Y[i] = Z[i]; /* (s1) */
    for (j=1; j<=N; j++)
        if (p == j)
            X[j] = Y[j]; /* (s2) */
}
```

b) 应用算法 11.45 后得到的代码

```
for (p=1; p<=N; p++) {
    i = p;
    if (p == i)
        Y[i] = Z[i]; /* (s1) */
    j = p;
    if (p == j)
        X[j] = Y[j]; /* (s2) */
}
```

c) 应用算法 11.13 后得到的代码

```
for (p=1; p<=N; p++) {
    Y[p] = Z[p]; /* (s1) */
    X[p] = Y[p]; /* (s2) */
}
```

d) 最终的代码

图 11-30 例 11.46 的代码

现在我们回到例 11.44，并说明把不同语句的多个迭代空间合并到一起的步骤。

**例 11.47** 现在让我们收紧例 11.44 中代码的循环界限。由分划单元  $p$  执行的语句  $s_1$  的迭代空间由下面的等式和不等式定义：

$$-100 \leq p \leq 99$$

$$1 \leq i \leq 100$$

$$1 \leq j \leq 100$$

$$i - p - 1 = j$$

对上面的算式应用算法 11.13 生成了图 11-31a 中显示的约束。算法 11.13 根据  $i - p - 1 = j$  和  $1 \leq j \leq 100$  生成约束  $p + 2 \leq i \leq 100 + p + 1$ ，并把  $p$  的上界收紧为 98。类似地，对于语句  $s_2$  的各个变量的界限在图 11-31b 中显示。

图 11-31 中语句  $s_1$  和  $s_2$  的迭代空间是相似的，但是如图 11-27 中所期望的，两个空间在某些界限上相差 1。图 11-32 中的代码在这两个迭代空间的并集上运行。比如，使用  $\max(1, p + 1)$  作为  $i$  的

下界,  $\min(100, 100 + p + 1)$  作为其上界。请注意, 最内层循环在第一次和最后一次执行时只有一个迭代, 而在其他情况下有两个迭代。生成循环下标的开销因此降低了一个数量级。因为被执行的迭代空间比  $s_1$  和  $s_2$  的迭代空间都大, 在执行这些语句的时候仍然需要使用条件判断来进行选择。□

$\begin{array}{l} j : i - p - 1 \leq j \leq i - p - 1 \\ \quad 1 \leq j \leq 100 \\ \\ i : p + 2 \leq i \leq 100 + p + 1 \\ \quad 1 \leq i \leq 100 \\ \\ p : -100 \leq p \leq 98 \end{array}$	$\begin{array}{l} j : i - p \leq j \leq i - p \\ \quad 1 \leq j \leq 100 \\ \\ i : p + 1 \leq i \leq 100 + p \\ \quad 1 \leq i \leq 100 \\ \\ p : -99 \leq p \leq 99 \end{array}$
--	---

a) 语句  $s_1$  的界限b) 语句  $s_2$  的界限图 11-31 图 11-29 中  $p$ 、 $i$  和  $j$  的较紧致的界限

### 11.7.7 从最内层循环中消除条件测试

第二个转换是从内层循环中消除条件测试。如上面的例子所示, 如果循环中各个语句的迭代空间相交但是不重合, 就需要保留条件测试语句。为了消除对条件测试的需求, 我们把迭代空间分割成为子空间, 每个子空间执行同样的语句集合。这个优化过程要求复制代码, 且只应该用于消除内层循环的条件测试。

为了分割一个迭代空间以消除内层循环的条件测试, 我们重复应用下列步骤直到消除内层循环中的所有条件测试:

- 1) 选择一个由界限不同的多个语句组成的循环。
- 2) 使用一个条件来分割循环, 使得某个语句从至少一个分循环中被剔除。我们从相互重叠的不同多面体的边界中选择这个条件。如果某个语句的所有迭代都只位于这个条件的某个半个平面中, 那么这个条件就是有用的。
- 3) 为每一个迭代空间生成代码。

**例 11.48** 让我们从图 11-32 的代码中删除条件测试。除了在两端的边界分划单元, 语句  $s_1$  和  $s_2$  被映射到同一个分划单元 ID。因此, 我们把分划空间分成三个子空间:

- 1)  $p = -100$
- 2)  $-99 \leq p \leq 98$
- 3)  $p = 99$

然后, 就可以针对每个子空间中所包含的  $p$  的值对它的代码进行特化。图 11-33 中显示了这三个迭代空间的代码。

请注意, 第一个和第三个空间不需要  $i$  或  $j$  的循环, 因为定义这两个空间的  $p$  值是确定的, 这些循环都是退化的, 它们只有一个迭代。比如, 在空间 1 中, 在循环界限中把  $p$  替换为  $-100$  会把  $i$  限制为 1, 从而把  $j$  限定为 100。在空间 1 和 3 中对  $p$  的赋值显然是死代码, 因此可以被消除。

下面我们在空间 2 中分割下标为  $i$  的循环。循环下标  $i$  的第一次和最后一次迭代是不同的。因此, 我们把这个循环分割为三个子空间:

```

for (p = -100; p <= 99; p++)
    for (i = max(1, p+1); i <= min(100, 101+p); i++)
        for (j = max(1, i-p-1); j <= min(100, i-p); j++) {
            if (p == i-j-1)
                X[i, j] = X[i, j] + Y[i-1, j]; /* (s1) */
            if (p == i-j)
                Y[i, j] = X[i, j-1] + Y[i, j]; /* (s2) */
        }
    }
}

```

图 11-32 通过收紧循环界限进行改进之后的

图 11-29 的代码

- 1)  $\max(1, p+1) \leq i < p+2$ , 其中只有  $s_2$  被执行。
- 2)  $\max(1, p+2) \leq i \leq \min(100, 100+p)$ , 其中  $s_1$  和  $s_2$  都被执行。
- 3)  $101+p < i \leq \min(101+p, 100)$ , 其中只有  $s_1$  被执行。

图 11-33 中第二个空间的循环嵌套因此可以被写成图 11-34a 所示的代码。

图 11-34b 显示了经过优化的程序。我们已经用图 11-34a 替换了图 11-33 中相应的循环迭代结构。我们也已经把对  $p$ 、 $i$  和  $j$  的赋值传播到数组访问中。当在中间代码层次上进行优化时, 其中的一些赋值会被识别为公共子表达式, 并从数组访问代码中重新抽取出来。□

```

/* 空间(1) */
p = -100;
i = 1;
j = 100;
X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */

/* 空间(2) */
for (p = -99; p <= 98; p++)
    for (i = max(1,p+1); i <= min(100,101+p); i++)
        for (j = max(1,i-p-1); j <= min(100,i-p); j++) {
            if (p == i-j-1)
                X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
            if (p == i-j)
                Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
        }

/* 空间(3) */
p = 99;
i = 100;
j = 1;
Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */

```

图 11-33 根据  $p$  的值分割迭代空间

```

/* 空间(2) */
for (p = -99; p <= 98; p++) {
    /* 空间(2a) */
    if (p >= 0) {
        i = p+1;
        j = 1;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* 空间(2b) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        j = i-p-1;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
        j = i-p;
        Y[i,j] = X[i,j-1] + Y[i,j]; /* (s2) */
    }
    /* 空间(2c) */
    if (p <= -1) {
        i = 101+p;
        j = 100;
        X[i,j] = X[i,j] + Y[i-1,j]; /* (s1) */
    }
}

```

a) 根据  $i$  的值分割空间(2)

图 11-34 例 11.48 的代码

```

/* 空间(1); p = -100 */
X[1,100] = X[1,100] + Y[0,100]; /* (s1) */

/* 空间(2) */
for (p = -99; p <= 98; p++) {
    if (p >= 0)
        Y[p+1,1] = X[p+1,0] + Y[p+1,1]; /* (s2) */
    for (i = max(1,p+2); i <= min(100,100+p); i++) {
        X[i,i-p-1] = X[i,i-p-1] + Y[i-1,i-p-1]; /* (s1) */
        Y[i,i-p] = X[i,i-p-1] + Y[i,i-p]; /* (s2) */
    }
    if (p <= -1)
        X[101+p,100] = X[101+p,100] + Y[101+p-1,100]; /* (s1) */
}
/* 空间(3); p = 99 */
Y[100,1] = X[100,0] + Y[100,1]; /* (s2) */

```

b) 和图11-28等价的优化代码

图 11-34 (续)

### 11.7.8 源代码转换

我们已经看到如何根据各个语句的简单仿射分划得到和原来的源代码明显不同的程序。但是, 至今为止看到的例子中都没有明确显示出仿射分划是如何与源代码层次上的改变联系起来的。本节将说明, 通过把仿射变换分解成为一系列基本变换, 我们可以相对容易地论证对源代码的修改。

#### 七个基本仿射转换

每个仿射分划可以表示为一个由的基本仿射转换组成的序列。每个基本仿射转换对应于源代码层次上的一个简单改变。总共有七个基本仿射转换: 前四个基本转换在图 11-35 中说明, 后三个转换被称为幺模转换(unimodular transform), 在图 11-36 中解释。

图中给出了每个基本转换的一个例子: 一个源代码、一个仿射分划和一个结果代码。我们也画出了转换之前和之后的代码中的数据依赖关系。在数据依赖关系图中, 我们看到每个基本转换对应于一个简单的几何转换, 对应于一个简单的代码转换。这七个基本转换为:

- 1) 融合(fusion)。融合转换的特点是把原程序中的多个循环下标映射到同一个循环下标上。新的循环融合了来自不同循环的语句。
- 2) 裂变(fission)。裂变转换是融合的逆向转换。它把不同语句的同一个循环下标映射到转换得到的代码中的不同循环下标。这个转换把原来的一个循环分解为多个循环。
- 3) 重新索引(re-indexing)。重新索引技术把一个语句的动态执行偏移固定多个迭代。这个仿射变换有一个常量项。
- 4) 比例变换(scaling)。源程序中的连续迭代被一个常量因子隔开。这个仿射变换具有一个正的非单元系数。
- 5) 反置(reversal)。按照相反顺序执行循环中的迭代。反置转换的特点是有一个系数为 -1。
- 6) 交换(permuation)。交换内层循环和外层循环。这个仿射变换由单位矩阵中的经过交换的各行组成。
- 7) 倾斜(skewing)。沿着一个角度来遍历循环的迭代空间。这个仿射变换是一个幺模矩阵, 其对角线上都是 1。

源代码	分划	转换后的代码
<pre>for (i=1; i&lt;=N; i++)     Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)     X[j] = Y[j]; /*s2*/ </pre>	融合 $s_1 : p = i$ $s_2 : p = j$	<pre>for (p=1; p&lt;=N; p++){     Y[p] = Z[p];     X[p] = Y[p]; }</pre>
<pre>for (p=1; p&lt;=N; p++){     Y[p] = Z[p];     X[p] = Y[p]; } </pre>	裂变 $s_1 : i = p$ $s_2 : j = p$	<pre>for (i=1; i&lt;=N; i++)     Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)     X[j] = Y[j]; /*s2*/ </pre>
<pre>for (i=1; i&lt;=N; i++) {     Y[i] = Z[i]; /*s1*/     X[i] = Y[i-1]; /*s2*/ } </pre>	重新索引 $s_1 : p = i$ $s_2 : p = i - 1$	<pre>if (N&gt;=1) X[1]=Y[0]; for (p=1; p&lt;=N-1; p++){     Y[p]=Z[p];     X[p+1]=Y[p]; } if (N&gt;=1) Y[N]=Z[N]; </pre>
<pre>for (i=1; i&lt;=N; i++)     Y[2*i] = Z[2*i]; /*s1*/ for (j=1; j&lt;=2*N; j++)     X[j] = Y[j]; /*s2*/ </pre>	比例变换 $s_1 : p = 2 * i$ $(s_2 : p = j)$	<pre>for (p=1; p&lt;=2*N; p++){     if (p mod 2 == 0)         Y[p] = Z[p];     X[p] = Y[p]; } </pre>

图 11-35 基本仿射转换(I)

### 幺模转换

一个幺模转换仅由一个幺模系数矩阵组成，没有常量向量。幺模矩阵是一个正方形矩阵，其行列式为  $\pm 1$ 。幺模转换的重要性在于它把一个  $n$  维迭代空间映射到另一个  $n$  维的多面体，并且两个空间之间的迭代具有一一对应关系。

### 并行化的几何解释

在上面的例子中，除裂变之外，其他的仿射转换都是通过把无同步仿射分划算法应用到各自的源代码上而得到的。（下一节中将讨论如何把裂变转换应用于带有同步的代码的并行化。）在每一个例子中，生成的代码有一个（最外层的）可并行化循环，这个循环的各个迭代可以分配给不同的处理器，并且不需要同步。

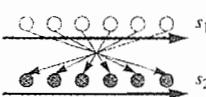
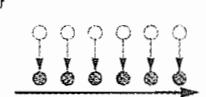
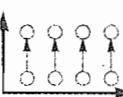
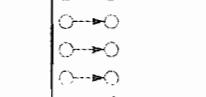
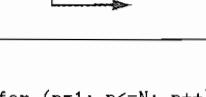
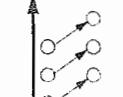
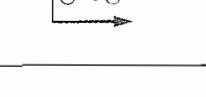
源代码	分划	转换后的代码
<pre>for (i=0; i&lt;=N; i++)     Y[N-i] = Z[i]; /*s1*/ for (j=0; j&lt;=N; j++)     X[j] = Y[j]; /*s2*/ </pre>	<p>反置  <math>s_1 : p = N - i</math>  <math>(s_2 : p = j)</math></p>	<pre>for (p=0; p&lt;=N; p++){     Y[p] = Z[N-p];     X[p] = Y[p]; }</pre> 
<pre>for (i=1; i&lt;=N; i++)     for (j=0; j&lt;=M; j++)         Z[i,j] =             Z[i-1,j]; </pre>	<p>交换</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$ 	<pre>for (p=0; p&lt;=M; p++)     for (q=1; q&lt;=N; q++)         Z[q,p] = Z[q-1,p]; </pre>
<pre>for (i=1; i&lt;=N+M-1; i++)     for (j=max(1,i+N);          j&lt;=min(i,M); j++)         Z[i,j] =             Z[i-1,j-1]; </pre>	<p>倾斜</p> $\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ 	<pre>for (p=1; p&lt;=N; p++)     for (q=1; q&lt;=M; q++)         Z[p,q-p] =             Z[p-1,q-p-1]; </pre>

图 11-36 基本仿射转换(Ⅱ)

这些例子说明可以使用几何学的方法来简单地解释并行化技术是如何工作的。依赖边总是从一个较早的实例指向较晚的实例。因此，嵌套在不同循环中的不同语句之间的依赖关系遵循程序文本中的顺序；嵌套在同一循环中的语句之间的依赖关系遵循词典顺序。从几何学角度讲，一个二维循环嵌套结构的依赖关系总是在 $[0^\circ, 180^\circ]$ 的范围之内，也就是说这些依赖关系的角度必然低于 $180^\circ$ ，但是不小于 $0^\circ$ 。

仿射转换改变了迭代的顺序，使得只有最外层循环的同一迭代之内的运算之间才有依赖关系。换句话说，在最外层循环的迭代边界上没有依赖边。在对简单的源代码进行并行化时，我们可以画出它们的依赖关系，并用几何方法找出这样的转换。

### 11.7.9 11.7 节的练习

#### 练习 11.7.1：对于下面的循环

```
for (i = 2; i < 100; i++)
    A[i] = A[i-2];
```

- 1) 最多可以用多少个处理器来有效运行这个循环？
- 2) 以处理器编号  $p$  作为参数改写这个代码。
- 3) 给出这个循环的空间分划约束，并找出这个约束的一个解。
- 4) 这个循环的具有最高秩的仿射分划是什么？

练习 11.7.2：对于图 11-37 中的循环嵌套结构重复练习 11.7.1。

```
for (i = 0; i <= 97; i++)
    A[i] = A[i+2];
```

a)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + C[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1];
        }
```

b)

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++)
        for (k = 1; k <= 100; k++) {
            A[i,j,k] = A[i,j,k] + B[i-1,j,k];
            B[i,j,k] = B[i,j,k] + A[i,j-1,k];
            C[i,j,k] = C[i,j,k] + A[i,j,k-1] + B[i,j,k];
        }
```

c)

图 11-37 练习 11.7.2 的代码

练习 11.7.3：改写下面的代码

```
for (i = 0; i < 100; i++)
    A[i] = 2*A[i];
for (j = 0; j < 100; j++)
    A[j] = A[j] + 1;
```

使得新代码只包含一个循环。以处理器编号  $p$  为下标改写这个循环，使得代码可以在 100 个处理器之间分配，其中第  $p$  个迭代由处理器  $p$  执行。

练习 11.7.4：在下面的代码中

```
for (i = 1; i < 100; i++)
    for (j = 1; j < 100; j++)
/* (s) */    A[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4;
```

唯一的约束是作为该循环嵌套结构的循环体的语句  $s$  必须首先执行迭代  $s(i-1,j)$  和  $s(i,j-1)$ ，然后执行迭代  $s(i,j)$ 。证明这些约束就是全部的必要约束。然后改写代码使得最外层循环的下标变量为  $p$ ，并且所有满足  $i+j=p$  的实例  $s(i,j)$  都在外层循环的第  $p$  个迭代上执行。

练习 11.7.5：重复练习 11.7.4，但是重新安排执行方案，使得  $s$  的满足  $i-j=p$  的实例在外层循环的第  $p$  个迭代上运行。

! 练习 11.7.6：把下面的循环

```
for (i = 0; i < 100; i++)
    A[i] = B[i];
for (j = 98; j >= 0; j = j-2)
    B[i] = i;
```

合并为一个循环，要求保持所有的依赖关系。

练习 11.7.7：证明矩阵

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

是幺模的。描述一下它对一个二维循环嵌套结构所做的转换。

练习 11.7.8：对下面的矩阵重复练习 11.7.7。

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

## 11.8 并行循环之间的同步

如果我们不允许处理器之间进行任何同步，很多程序就没有任何并行性。但是通过向一个程序中增加少量固定多个同步运算之后，可以找到更多的并行性。在本节中，我们将首先讨论因为引入固定多个同步运算而获得的并行性。下一节中将讨论一般情况，即把同步运算嵌入到循环中的情况。

### 11.8.1 固定多个同步运算

没有无同步并行性的程序可能包含一系列循环。如果独立地考虑这些循环，其中的某些循环是可以并行化的。我们可以在这些循环执行之前和之后引入同步栅障，从而把这些循环并行化。例 11.49 说明了这一点。

**例 11.49** 图 11-38 给出了一个实现了 ADI(交替方向隐式方法，Alternating Direction Implicit) 积分算法的典型程序。它没有无同步并行性。在第一个循环嵌套结构中的依赖关系要求每个处理器在数组  $X$  的一列上工作；但是在第二个循环嵌套结构中的依赖关系要求每个处理器在数组  $X$  的一行上工作。如果要求没有通信运算，整个数组必须放在同一个处理器上，因此不存在并行性。但是，我们观察到两个循环都是可以单独并行化的。

```

for (i = 1; i < n; i++)
    for (j = 0; j < n; j++)
        X[i,j] = f(X[i,j] + X[i-1,j]);
for (i = 0; i < n; i++)
    for (j = 1; j < n; j++)
        X[i,j] = g(X[i,j] + X[i,j-1]);

```

图 11-38 两个顺序的循环嵌套结构

并行化代码的方法之一是在第一个循环中让不同的处理器在数组的不同列上工作，同步并等待所有的处理器完成任务后，各个处理器再在各个行上进行运算。使用这个方法，只需要引入一个同步操作就可以使算法中的所有计算都被并行化。但是，我们注意到虽然只进行了一次同步，但是这个并行化方案要求几乎所有的矩阵  $X$  中的数据在不同的处理器之间传递。通过引入更多的同步计算有可能降低通信量。我们将在 11.9.9 节中讨论这个问题。□

看起来，这个方法可能只适用于由一系列循环嵌套组成的程序。但是，我们可以通过代码转换创造出更多的优化机会。我们可以应用循环裂变转换把原程序中的循环分解成为几个较小的循环。利用同步栅障把它们隔开，然后逐一将它们并行化。我们用例 11.50 来解释这个技术。

**例 11.50** 考虑下面的循环：

```

for (i=1; i<=n; i++) {
    X[i] = Y[i] + Z[i]; /* (s1) */
    W[A[i]] = X[i]; /* (s2) */
}

```

因为不知道数组  $A$  中的值，我们必须假设语句  $s_2$  中的访问可以写到  $W$  的任何元素上。因此， $s_2$  的实例的执行顺序必须和它们在原程序中的顺序一致。

代码中没有无同步的并行性，算法 11.43 将简单地把所有的计算任务都赋予同一个处理器。但是，至少语句  $s_1$  的实例可以并行执行。我们可以把这个代码的一部分并行化，方法是让不同的处理器执行语句  $s_1$  的不同实例。然后在另一个独立的顺序循环中，用一个处理器（比如说 0 号处理器）执行  $s_2$ 。相应的 SPMD 代码显示在图 11-39 中。□

```

X[p] = Y[p] + Z[p]; /* (s1) */
/* synchronization barrier */
if (p == 0)
    for (i=1; i<=n; i++)
        W[A[i]] = X[i]; /* (s2) */

```

图 11-39 例 11.50 中的循环的 SPMD 代码，其中  $p$  是存放处理器 ID 的变量

### 11.8.2 程序依赖图

为了找出所有可以通过加入固定多个同步运算而变得可用的并行性，我们可以尽可能地对原程序应用裂变转换。把循环尽可能分解为独立循环，然后独立地并行化每个循环。

为了揭示出所有可以进行循环裂变的机会，我们使用程序依赖图（Program Dependence Graph, PDG）的抽象表示方法。程序的程序依赖图中的各个结点是程序的赋值语句，图中的边表示语句之间的数据依赖关系以及依赖的方向。只要语句  $s_1$  的某个动态实例和后面的语句  $s_2$  的一个动态实例之间存在数据依赖关系，就存在一条从语句  $s_1$  到语句  $s_2$  的边。

构造一个程序的 PDG 时，我们首先找出每一对语句中的两个静态访问之间的数据依赖关系。一个语句对中的两个语句可以相同，这两个静态访问也可以相同。假设确定了语句  $s_1$  中的访问  $\mathcal{R}_1$  和语句  $s_2$  中的访问  $\mathcal{R}_2$  之间有依赖关系。请注意，一个语句的实例可以使用下标向量  $\mathbf{i} = [i_1, i_2, \dots, i_m]$  来刻画，其中  $i_k$  是该语句所在循环嵌套结构中从最外层开始的第  $k$  个循环的下标。

1) 如果有两个语句实例， $s_1$  的实例  $\mathbf{i}_1$  和  $s_2$  的实例  $\mathbf{i}_2$ ，它们之间具有数据依赖关系，并且在原程序中， $\mathbf{i}_1$  在  $\mathbf{i}_2$  之前执行，记作  $\mathbf{i}_1 \prec_{s_1, s_2} \mathbf{i}_2$ ，那么有一条从  $s_1$  到  $s_2$  的边。

2) 类似地，如果有两个语句实例， $s_1$  的实例  $\mathbf{i}_1$  和  $s_2$  的实例  $\mathbf{i}_2$ ，它们之间具有数据依赖关系，记作  $\mathbf{i}_2 \prec_{s_1, s_2} \mathbf{i}_1$ ，那么有一条从  $s_2$  到  $s_1$  的边。

请注意，有可能根据两个语句  $s_1$  和  $s_2$  之间的数据依赖关系，在 PDG 中既生成了从  $s_1$  到  $s_2$  的边，又生成了从  $s_2$  到  $s_1$  的边。

在语句  $s_1$  和  $s_2$  相同的特殊情况下， $\mathbf{i}_1 \prec_{s_1, s_2} \mathbf{i}_2$  当且仅当  $\mathbf{i}_1 \prec \mathbf{i}_2$ （即  $\mathbf{i}_1$  按照词典排序比  $\mathbf{i}_2$  小）。在一般情况下， $s_1$  和  $s_2$  可以是不同的语句，有可能属于不同的循环嵌套结构。

**例 11.51** 对于例 11.50 中的程序，在语句  $s_1$  的实例之间没有依赖关系。但是，语句  $s_2$  的第  $i$  个实例必须在语句  $s_1$  的第  $i$  个实例之后发生。更糟糕的是，因为数组引用  $W[A[i]]$  可以对数组的  $W$  的每个元素进行写运算， $s_2$  的第  $i$  个实例依赖于所有之前的  $s_2$  的实例。也就是说，语句  $s_2$  依赖于它本身。例 11.50 的程序的 PDG 显示在图 11-40 中。请注意图中有一个只包含  $s_2$  的环。



图 11-40 例 11.50 的程序的 PDG

程序依赖图使得我们可以很容易地确定是否可以分割一个循环中的多个语句。在一个 PDG 中，一个环所连接的各个语句不能被分割开。如果  $s_1 \rightarrow s_2$  是一个环中两个语句之间的依赖关系，那么  $s_1$  的某些实例必须在  $s_2$  的某些实例之后发生，反过来也成立。请注意，只有当  $s_1$  和  $s_2$  嵌入同一个循环中的时候才可能有这种相互依赖关系。因为有这种相互依赖关系，我们不能先执行完一个语句的所有实例之后再执行另一个语句的所有实例，因此不允许进行循环裂变转换。另一方面，如果依赖关系  $s_1 \rightarrow s_2$  是单向的，我们就可以对这个循环进行分割，首先执行  $s_1$  的所有实例，然后执行  $s_2$  的实例。

**例 11.52** 图 11-41b 显示了图 11-41a 中程序的程序依赖图。图中语句  $s_1$  和  $s_3$  属于一个环，因此不能被放到不同的循环中去。但是我们可以把语句  $s_2$  分割出去，并在执行其他计算之前执行它的所有实例，如图 11-42 所示。第一个循环是可以并行化的，但是第二个循环不能被并行化。我们可以在第一个循环的并行执行之前和之后放上一个同步栅障，从而把第一个循环并行化。

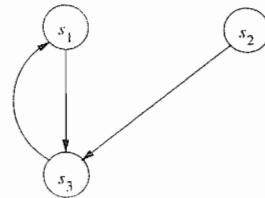
□

```

for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];           /* (s1) */
    for (j = i; j < n; j++) {
        X[i,j] = Y[i,j]*Y[i,j];  /* (s2) */
        Z[j] = Z[j] + X[i,j];    /* (s3) */
    }
}

```

a) 一个程序



b) 它的依赖图

图 11-41 例 11.5.2 的程序和依赖图

```

for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
        X[i,j] = Y[i,j]*Y[i,j];  /* (s2) */
for (i = 0; i < n; i++) {
    Z[i] = Z[i] / W[i];           /* (s1) */
    for (j = i; j < n; j++)
        Z[j] = Z[j] + X[i,j];    /* (s3) */
}

```

图 11-42 对一个循环嵌套结构的强连通子图进行分组

### 11.8.3 层次结构化的时间

在一般情况下，关系  $\prec_{s_1 s_2}$  的计算是很困难的。但是对于某些类型的程序而言，我们有一个直接的方法来计算这种依赖关系。本节中的优化技术经常被应用于这一类程序。假设这个程序是块结构的，由循环和简单的算术运算组成，并且不包含其他控制结构。该程序中的语句要么是一个赋值语句，要么是一个语句序列，要么是一个其循环体为单个语句的循环结构。这样，这个程序的控制结构就形成了一个层次结构。这个层次结构的顶层结点表示对应于整个程序的语句。单个赋值语句是一个叶子结点。如果某语句是一个语句序列，那么它的子结点就是该序列中的语句。这些子结点按照语句的词典排序从左到右排列。如果某语句是一个循环结构，那么它的子结点就是循环体对应的子图，通常是由一个或多个语句组成的序列。

**例 11.53** 图 11-43 中程序的层次结构显示在图 11-44 中。执行序列的层次结构特性在图 11-45 中着重显示。语句  $s_0$  的唯一实例在所有其他运算之前进行，因为它是被执行的第一个语句。接下来，我们执行来自外层循环的第一个迭代的所有指令，然后再执行第二个迭代中的指令，这样一直向前执行。对于循环下标  $i$  的值为 0 的所有动态实例，语句  $s_1, L_2, L_3$  和  $s_5$  按照正文顺序执行。我们可以重复上面的讨论，生成执行顺序的其他部分。□

我们可以用一种层次结构化的方式来解决由两个不同语句生成的两个实例之间的顺序问题。如果两个语句处于同一个循环中，我们从最外层循环开始比较它们的共同循环的下标变量的值。当我们发现它们的某个下标具有不同值时，这个差值就决定了它们之间的顺序。只有当较外层循环的下标值都相同的时候，我们才需要比较下一个内层循环

```

s0;
L1: for (i = 0; ...) {
    s1;
    L2: for (j = 0; ...) {
        s2;
        s3;
    }
    L3: for (k = 0; ... )
        s4;
    s5;
}

```

图 11-43 一个按层次结构组织的程序

的下标值。这个过程类似于我们比较以小时/分钟/秒的方式所表示的时间。比较两个时间时，我们首先比较小时数，只有当它们的小时数相同的时候，我们才比较分钟，以此类推。如果所有公共循环的下标值都相同，那么我们根据它们的相对正文位置来决定它们之间的顺序。因此，我们一直在讨论的简单嵌套循环程序的执行顺序经常被称为“层次结构化”的时间。

令  $s_1$  为一个嵌套在深度为  $d_1$  的循环中的语句，而  $s_2$  嵌套在深度为  $d_2$  的循环中，它们之间有  $d$  个公共(外层)循环。当然  $d \leq d_1$  且  $d \leq d_2$ 。假设  $i = [i_1, i_2, \dots, i_{d_1}]$  为  $s_1$  的一个实例，而  $j = [j_1, j_2, \dots, j_{d_2}]$  为  $s_2$  的一个实例。

$i \prec_{s_1, s_2} j$  当且仅当下列条件之一成立：

- 1)  $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$ ，或者
- 2)  $[i_1, i_2, \dots, i_d] = [j_1, j_2, \dots, j_d]$ ，且在正文上  $s_1$  出现在  $s_2$  之前。

断言  $[i_1, i_2, \dots, i_d] \prec [j_1, j_2, \dots, j_d]$  可以写成如下的线性不等式的析取式。

$$\begin{aligned} & (i_1 < j_1) \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \dots \vee \\ & (i_1 = j_1 \wedge \dots \wedge i_{d-1} = j_{d-1} \wedge i_d < j_d) \end{aligned}$$

只要数据依赖关系的条件和上面的析取式中的某个子句同时成立，就存在一个从  $s_1$  到  $s_2$  的 PDG 边。因此，我们可能需要求解多达  $d$  或  $d+1$  个整数线性规划问题来决定某一条边是否存在。要求解的问题个数依赖于语句  $s_1$  是否按照正文顺序出现在  $s_2$  之前。

#### 11.8.4 并行化算法

现在我们给出一个简单的并行化算法。它首先把计算任务分解到尽可能多的不同循环中，然后独立地并行化各个循环。

**算法 11.54** 在允许  $O(1)$  次同步的情况下最大化并行性的度数。

**输入：**一个带有数组访问的程序。

**输出：**带有固定多个同步栅障的 SPMD 代码。

**方法：**

1) 构造程序的程序依赖图，并把语句分划为强连通分量(SCC)。回忆一下 10.5.8 节介绍过，一个强连通分量是原图的一个满足下列条件的最大的分量：其中的每个结点都可以到达所有其他结点。

2) 转换代码，使之按照拓扑顺序执行各个 SCC。必要时可以应用裂变转换。

3) 对每个 SCC 应用算法 11.43，寻找出所有的无同步并行性。在每个被并行化的 SCC 的前后都插入同步栅障。□

虽然算法 11.54 能够找到带有  $O(1)$  次同步的所有并行性度数，它仍然存在一些缺点。第一，它可能引入不必要的同步。作为一个现实问题，如果我们把这个算法应用到一个可以被无同步地并行化的程序上，这个算法会对每个语句进行并行化，并且在执行各个语句的并行循环之间引入同步栅障。第二，虽然只会有固定多个同步，但得到的并行化方案会在每次同步的时候在不同

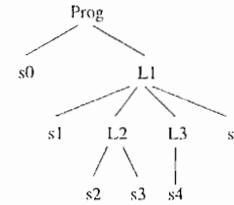


图 11-44 例 11.53 中的程序的层次结构

1:	$s_0$
2:	$L_1 \quad i = 0 \quad s_1$
3:	$\quad L_2 \quad j = 0 \quad s_2$
4:	$\quad \quad \quad s_3$
5:	$\quad \quad \quad j = 1 \quad s_2$
6:	$\quad \quad \quad s_3$
7:	$\quad \quad \quad \dots$
8:	$\quad \quad \quad L_3 \quad k = 0 \quad s_4$
9:	$\quad \quad \quad \quad \quad k = 1 \quad s_4$
10:	$\quad \quad \quad \quad \quad \dots$
11:	$\quad \quad \quad s_5$
12:	$\quad \quad \quad i = 1 \quad s:$
13:	$\quad \quad \quad \dots$

图 11-45 例 11.53 中的程序的执行顺序

的处理器之间传递很多数据。有些情况下，通信开销会使得并行化的代价太过昂贵，有时甚至不如在一个单处理器上顺序执行这个程序。在后面的各节中，我们将继续给出提高数据局部性的手段，从而降低通信量。

### 11.8.5 11.8 节的练习

练习 11.8.1：把算法 11.54 应用到图 11-46 的代码上。

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        B[i,j] = Y[i,j] + A[i] + A[j]; /* (s2) */

```

图 11-46 练习 11.8.1 的代码

练习 11.8.2：把算法 11.54 应用到图 11-47 的代码上。

练习 11.8.3：把算法 11.54 应用到图 11-48 的代码上。

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    B[i] = B[i] + A[i]; /* (s2) */
    for (j=0; j<100; j++)
        C[j] = Y[j] + B[j]; /* (s3) */
}

```

图 11-47 练习 11.8.2 的代码

```

for (i=0; i<100; i++)
    A[i] = A[i] + X[i]; /* (s1) */
for (i=0; i<100; i++) {
    for (j=0; j<100; j++)
        B[j] = A[i] + Y[j]; /* (s2) */
    C[i] = B[i] + Z[i]; /* (s3) */
    for (j=0; j<100; j++)
        D[i,j] = A[i] + B[j]; /* (s4) */
}

```

图 11-48 练习 11.8.3 的代码

## 11.9 流水线化技术

在流水线化技术中，一个任务被分成数个阶段，各个阶段在不同的处理器上进行。比如，一个具有  $n$  个迭代的循环可以被构造一个  $n$  阶段的流水线。每个阶段被分配给不同的处理器，当一个处理器完成了它负责的阶段后，结果就作为输入传送到流水线中的下一个处理器。

下面我们首先更加详细地解释流水线化的概念。然后我们在 11.9.2 节中给出了一个实际生活中的被称为“连续过松弛方法”的数值算法。我们用这个例子说明在什么样的情况下可以应用流水线化技术。然后我们在 11.9.6 节中正式定义需要求解的约束，并在 11.9.7 节中描述一个求解这些问题的算法。如果一个程序的时间分划约束具有多个独立解，那么就可以认为它具有最外层的完全可交换循环(fully permutable loop)。正如 11.9.8 节中将讨论的，这样的循环可以很容易地被流水线化。

### 11.9.1 什么是流水线化

前面我们尝试对一个循环嵌套结构进行并行化时，我们对这个循环嵌套结构中的迭代进行分划，使得任意两个共享数据的迭代都被分配给同一个处理器上。流水线化技术允许不同处理器共享数据，但是一般只能以“局部的”方式来共享数据，数据只能从一个处理器传递到所在处理器空间中的邻近处理器上。下面给出一个简单的例子。

#### 例 11.55 考虑循环：

```

for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        X[i] = X[i] + Y[i,j];

```

这个代码把  $Y$  的第  $i$  行中的值相加，并把结果加到  $X$  的第  $i$  个元素上。其中的内层循环对应于求和过程。因为数据依赖的原因，这个循环必须顺序执行<sup>⊖</sup>，但是不同的求和过程之间是独立的。我们可以让每个处理器执行一个独立的求和过程，从而实现代码的并行化。处理器  $i$  访问  $Y$  的第  $i$  行并修改  $X$  的第  $i$  个元素。

我们还可以把多个处理器组织成一个流水线来执行这个求和过程，并通过求和过程的重叠来获取并行性，如图 11-49 所示。更明确地讲，内层循环的每个迭代都可以被当作流水线的一个阶段：第  $j$  个阶段获取在前一阶段生成的  $X$  的一个元素，将它和  $Y$  的一个元素相加，并把结果传递到下一个阶段。请注意在这种情况下，每个处理器访问  $Y$  的一列，而不是一行。如果  $Y$  是按列存放的，那么通过按列分区（而不是按行分区）就可以提高局部性。

第一个处理器处理完前一个任务的第一个阶段之后，我们就可以立刻启动一个新的任务。流水线在开始时是空的，只有第一个处理器在执行第一个阶段。在它完成处理之后，结果被传送到第二个处理器，同时第一个处理器开始处理第二个任务，如此继续。按照这种方式，流水线被逐渐填满，直到所有的处理器都进入忙状态。当第一个处理器完成了最后一个任务后，流水线开始排空，越来越多的处理器进入空闲状态，直到最后一个处理器完成最后一个任务。在稳定状态下， $n$  个任务在由  $n$  个处理器组成的流水线中并行执行。□

把流水线技术和不同处理器处理不同任务的简单并行性进行比较是很有意思的：

- 流水线化技术只能应用于深度至少为 2 的循环嵌套结构。我们可以把外层循环的每个迭代当作一个任务，而把内层循环的各个迭代当作任务的各个阶段。
- 在一个流水线中运行的任务可以具有数据依赖关系。属于各个任务的同一个阶段的信息被存放在同一个处理器上。因此，由一个任务的第  $i$  个阶段生成的结果可以直接被后继任务的第  $i$  个阶段使用，不会产生通信开销。类似地，由不同任务的同一个阶段所使用的每个输入数据元素必须存放在同一个处理器内，如例 11.55 所示。
- 如果任务是独立的，那么简单的并行化方案具有较好的处理器利用率，原因是各个处理器可以一起开始执行，而不会产生填满和排空流水线的开销。但是，如例 11.55 所示，在一个流水线方案中的数据访问模式和简单并行化方案中的模式不同。如果流水线化技术可以降低通信量，那么就应该选择这个技术。

### 11.9.2 连续过松弛方法：一个例子

连续过松弛法 (Successive Over Relaxation, SOR) 是一个在使用松弛方法求解联立线性方程式时加快收敛速度的技术。图 11-50a 中显示的相对简单的模板解释了这个技术的数据访问模式。在这里，数组中的一个元素的新值依赖于它的相邻元素的值。这个运算会被重复执行，直到满足某种收敛标准为止。

图 11-50b 中显示的是关键数据依赖关系。我们没有显示能够从该图中已包含的依赖关系推导出的依赖关系。比如，迭代  $[i, j]$  依赖于迭代  $[i, j-1], [i, j-2]$ ，等等。从这个依赖关系可以清

时间	处理器		
	1	2	3
1	$X[1] += Y[1,1]$		
2	$X[2] += Y[2,1]$	$X[1] += Y[1,2]$	
3	$X[3] += Y[3,1]$	$X[2] += Y[2,2]$	$X[1] += Y[1,3]$
4	$X[4] += Y[4,1]$	$X[3] += Y[3,2]$	$X[2] += Y[2,3]$
5		$X[4] += Y[4,2]$	$X[3] += Y[3,3]$
6			$X[4] += Y[4,3]$

图 11-49 例 11.55 中的流水线化的执行过程，  
其中  $m = 4, n = 3$

⊖ 请记住，我们没有利用加法的交换率和结合率。

楚地看出不存在无同步并行性。因为最长的依赖关系链包含了  $O(m+n)$  个边，通过引入同步，我们应该可以找到度数为 1 的并行性，并在  $O(m+n)$  个时间单位内执行  $O(mn)$  运算。

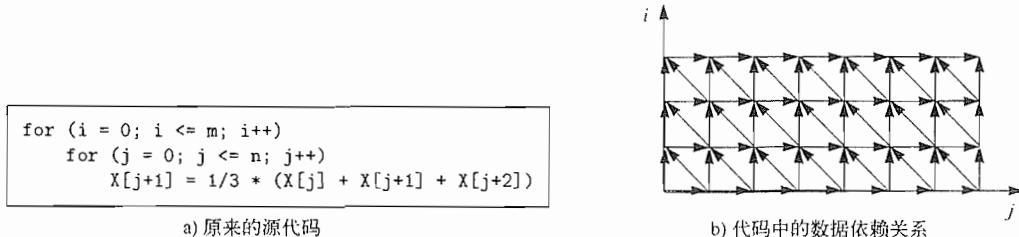


图 11-50 连续过松弛法(SOR)的例子

特别地，我们看到图 11-50b 中角度为  $150^\circ$ <sup>⊖</sup> 的斜线上的各个迭代之间没有数据依赖关系。它们只依赖于比较靠近原点的斜线上的迭代。因此，我们可以从原点上的斜线开始逐步向外，逐条斜线地执行线上的迭代，从而达到并行化这个代码的目的。我们把一个斜线上的全部迭代称为波阵面(wave front)，而这样的并行化方案被称为波阵面推进(wavefronting)。

### 11.9.3 完全可交换循环

我们首先介绍一下完全可交换(full permutability)的概念。这个概念对于流水线化和其他一些优化技术都是有用的。多个循环是完全可交换的条件是它们可以任意地排列而不会改变原程序的语义。一旦多个循环具有完全可交换的性质，我们可以很容易地把相应的代码流水线化，并对代码应用某些转换(比如分块技术)来提高数据局部性。

图 11-50a 中给出的 SOR 代码不是完全可交换的。如 11.7.8 节所示，交换两个循环的位置意味着原来的迭代空间中的迭代按照逐列(而不是逐行)的方式执行。比如，原来在迭代[2,3]中的计算将会在迭代[1,4]的计算之前执行，这就违反了图 11-50b 中的依赖关系。

然而，我们可以通过代码转换使得上面的 SOR 代码变成完全可交换的。对这个代码应用仿射转换

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

可得到图 11-51a 中所示代码。经过转换得到的代码是完全可交换的，交换后的版本如图 11-51c 所示。我们在图 11-51b 和图 11-51d 中分别显示了这两个程序的迭代空间和数据依赖关系。从这个图中可以很容易看出这个重新排序保持了每一对具有数据依赖关系的访问之间的相对顺序。

当我们交换循环时，我们极大地改变了最外层循环的各个迭代所执行的运算集合。我们在调度这些运算时具有这样的自由度，说明在对程序的运算进行排序时有很大的回旋余地。调度的余地意味着存在并行化的的机会。在本节的稍后我们将说明如果一个循环嵌套结构具有  $k$  个最外层的完全可交换循环，我们仅仅需要引入  $O(n)$  个同步运算，就可以得到  $O(k-1)$  度的并行性( $n$  是一个循环中的迭代的个数)。

⊖ 即不断下移一步再右移两步所得到的点的序列。

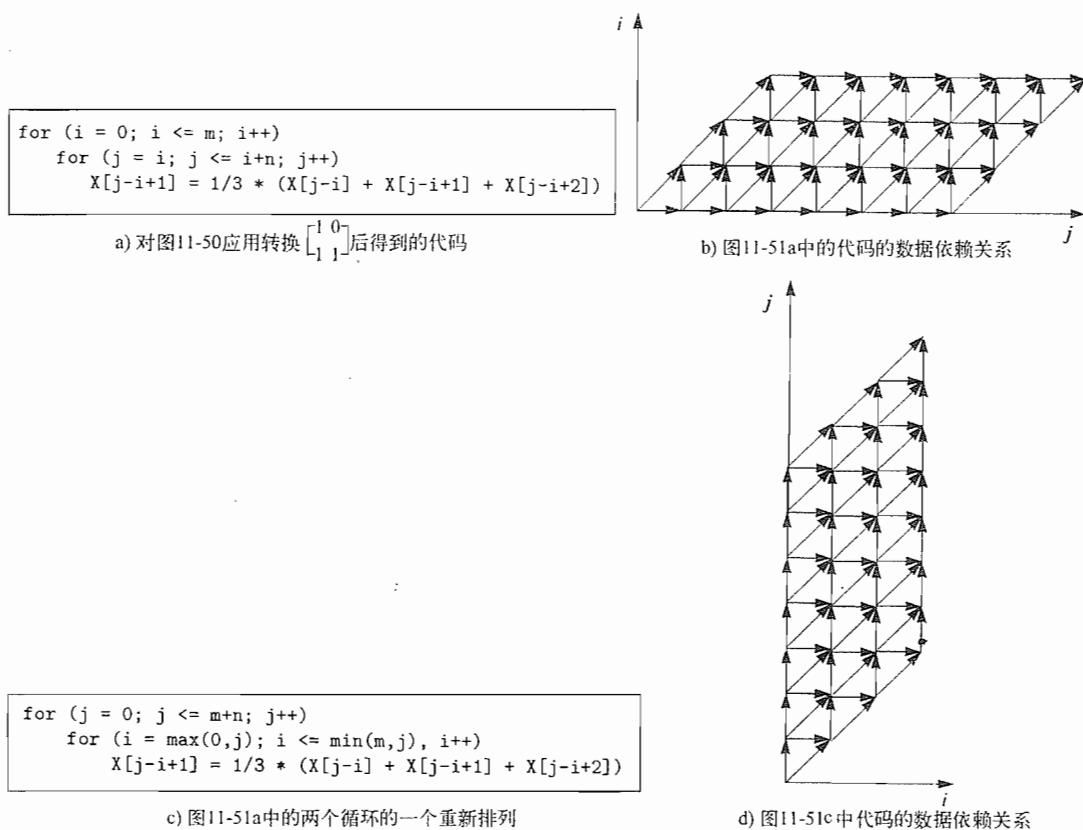


图 11-51 图 11-50 中代码的完全可交换版本

#### 11.9.4 把完全可交换循环流水线化

一个具有  $k$  个最外层完全可交换循环的循环嵌套结构可以被构造为一个  $O(k - 1)$  维的流水线。在 SOR 的例子中,  $k = 2$ , 因此我们可以把处理器构造成一个线性流水线。

我们可以用两种不同的方法对上面的 SOR 代码进行流水线化, 如图 11-52a 和图 11-52b 所示。这两种流水线化方案分别对应于图 11-51a 和图 11-51c 所示的两种可能的排列。在每一种情况下, 相应迭代空间的每一列组成一个任务, 而每一行组成一个流水线阶段。我们把第  $i$  个阶段分配给处理器  $i$ , 因此每个处理器执行内层循环的代码。不考虑边界条件, 只有在处理器  $p - 1$  执行了迭代  $i - 1$  之后, 处理器  $p$  才可以执行迭代  $i$ 。

假设每个处理器用同样的时间来执行一个迭代, 并且同步运算在瞬时

```
/* 0 <= p <= m */
for (j = p; j <= p+n; j++) {
    if (p > 0) wait (p-1);
    X[j-p+1] = 1/3 * (X[j-p] + X[j-p+1] + X[j-p+2]);
    if (p < min (m,j)) signal (p+1);
}
```

a) 把处理器分配给各行

```
/* 0 <= p <= m+n */
for (i = max(0,p); i <= min(m,p); i++) {
    if (p > max(0,i)) wait (p-1);
    X[p-i+1] = 1/3 * (X[p-i] + X[p-i+1] + X[p-i+2]);
    if (p < m+n) & (p > i) signal (p+1);
}
```

b) 把处理器分配给各列

图 11-52 图 11-51 中的代码的两个流水线化实现

发生。这两个流水线化方案将并行执行同样的迭代，唯一的区别是它们的处理器分配方法不同。在图 11-51b 中的迭代空间中，所有并行执行的迭代处于  $135^\circ$  的斜线上，这些斜线对应于原迭代空间中的  $150^\circ$  斜线，见图 11-50b。

但是在实践中，带有高速缓存的处理器执行同一代码所花的时间并不总是相同的，用于同步运算的时间也会有所变化。使用同步栅障将使所有的处理器按照一致的步调进行运算。和同步栅障方法不同，流水线化技术最多要求处理器和另外两个处理器进行同步和通信。因此，流水线化的波阵面更加松弛，允许有些处理器领先而其他处理器暂时落后。这个灵活性降低了处理器在等待其他处理器时所花的时间，提高了并行性能。

可以把这个计算任务流水线化的方法有很多，上面显示的流水线化方案只是其中的两个。我们说过，只要一个循环嵌套结构是完全可交换的，我们在选择代码并行化方案方面就有很大的自由度。第一个流水线方案把迭代  $[i, j]$  映射到处理器  $i$ ；第二个方案把迭代  $[i, j]$  映射到处理器  $j$ 。只要  $c_0$  和  $c_1$  是正常数，我们就可以把迭代  $[i, j]$  映射到处理器  $c_0 i + c_1 j$ ，从而得到其他的流水线化方案。这样的方案将创建出不同的流水线，它们的松弛波阵面位于  $90^\circ$ （不含）到  $180^\circ$ （不含）之间。

### 11.9.5 一般性的理论

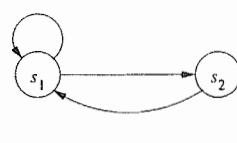
刚刚讨论的例子解释了关于流水线化的一般性理论：如果我们能够在一个循环嵌套结构中找到至少两个不同的最外层循环，并满足所有的依赖关系，那么就可以把这个计算过程流水线化。一个具有  $k$  个最外层完全可交换循环的循环嵌套结构具有  $k - 1$  度的流水线化并行度。

不能被流水线化的循环嵌套结构没有可交换的外层循环。例 11.56 给出了这样的例子。为了遵循所有的依赖关系，在最外层循环中的每个迭代必须精确执行原来代码中的计算任务。但是，这样的代码仍然可能在内层循环中包含并行性。要利用这种并行性，我们必须引入至少  $n$  个同步运算，其中  $n$  是最外层循环中的迭代个数。

**例 11.56** 图 11-53 是我们在例 11.50 中所见程序的一个更复杂的版本。如图 11-53b 的程序依赖图所示，语句  $s_1$  和  $s_2$  属于同一个强连通分量。因为我们不知道矩阵  $A$  中的内容，所以必须假设语句  $s_2$  中的访问可以读取  $X$  的任何元素。从语句  $s_1$  到语句  $s_2$  之间有一个真依赖关系，并且从语句  $s_2$  到语句  $s_1$  存在一个反依赖关系。这两个语句都没有进行流水线化的机会，因为属于外层循环的迭代  $i$  的所有运算必须在属于迭代  $i + 1$  的所有运算之前进行。为了找到更多的并行性，我们对内层循环重复并行化过程。第二个循环中的迭代可以被无同步地并行化。因此，我们需要 200 个同步栅障，在内层循环的每次执行之前和之后各需要一个。 □

```
for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++)
        X[j] = X[j] + Y[i, j]; /* (s1) */
        Z[i] = X[A[i]]; /* (s2) */
}
```

a)



b)

图 11-53 一个顺序化的外层循环(参见图 a)以及它的 PDG 图(参见图 b)

### 11.9.6 时间分划约束

现在我们关注寻找流水线化并行性的问题。我们的目标是把一个计算任务转变成为一组可流水线化的任务。为了找到流水线化的并行性，我们没有像处理循环并行性时那样直接求出各

个处理器上将执行哪些计算，而是提出了下面的根本性问题：所有可能的遵循循环中原有数据依赖关系的执行序列有哪些？显然，原来的执行序列满足所有的数据依赖关系。问题是是否存在这样的仿射转换，由它可以创建另一种调度，使得最外层循环的各个迭代执行的运算集合和原来不同，但是依然满足所有的依赖关系。如果我们能够找到这样的转换，就能够把这个循环结构流水线化。要点在于如果在调度运算时存在自由度，那么就存在并行性。后面将会解释如何从这样的转换中获取流水线化并行性的细节问题。

为了找出可接受的外层循环的重新排序，我们希望为每个语句找到一个一维仿射变换，这个变换把原来的循环下标值映射到最外层循环的迭代编号上。如果这样的分配能够满足程序中的所有数据依赖关系，那么变换就是合法的。下面显示的“时间分划约束”就是说如果一个运算依赖于另一个运算，那么分配给前一个运算的最外层循环的迭代必须不早于分配给第二个运算的迭代。如果它们被分配到同一个迭代，我们都应该在此迭代中第一个运算必须在第二个之后执行。

程序的一个仿射分划映射是一个合法的时间分划(legal-time partition)当且仅当对于任意两个具有依赖关系的(可能相同的)访问，比如嵌套在循环结构  $d_1$  中的语句  $s_1$  中的访问

$$\mathcal{F}_1 = \langle F_1, f_1, B_1, b_1 \rangle$$

和嵌套在循环结构  $d_2$  中的语句  $s_2$  的访问

$$\mathcal{F}_2 = \langle F_2, f_2, B_2, b_2 \rangle$$

分别对应于语句  $s_1$  和  $s_2$  的一维分划映射  $\langle C_1, c_1 \rangle$  和  $\langle C_2, c_2 \rangle$  满足下面的时间分划约束(time-partition constraint)：

- 对于满足下列条件的  $Z^{d_1}$  中所有的  $i_1$  和  $Z^{d_2}$  中的所有  $i_2$

- 1)  $i_1 \prec_{s_1 s_2} i_2$
  - 2)  $B_1 i_1 + b_1 \geq 0$
  - 3)  $B_2 i_2 + b_2 \geq 0$
  - 4)  $F_1 i_1 + f_1 = F_2 i_2 + f_2$
- 必然有  $C_1 i_1 + c_1 \leq C_2 i_2 + c_2$ 。

图 11-54 中显示的这个约束和空间分划约束看起来非常相似。它是空间分划约束的一个放松版本。如果两个迭代指向同一个位置，这个约束不要求它们被映射到同一个分划单元。我们只要求这两个迭代之间的相对执行顺序保持不变。也就是说，在空间分划约束中使用 = 的地方，这个约束使用  $\leq$ 。

我们知道，这个时间分划约束至少存在一个解。我们可以把最外层循环的每个迭代中的运算映射到同一个迭代中去，此时所有的数据依赖关系都得到满足。对于那些不能被流水线化的程序，这个解是它们的时间分划约束的唯一解。另一

方面，如果我们能够找到时间分划约束的多个独立解，这个程序就能够被流水线化。每个独立解对应于最外层完全可交换循环嵌套结构中的一个循环。如你所期望的，因为例 11.56 中的程序没

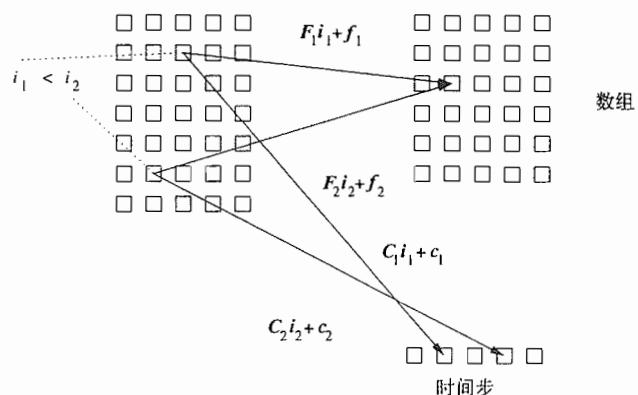


图 11-54 时间分划约束

有可流水线化的并行性，因此从中抽取得到的时间分划约束只有一个独立解。而上面的 SOR 代码的例子则存在两个独立解。

**例 11.57** 我们考虑一下例 11.56，特别是考虑语句  $s_1$  和  $s_2$  中对数组  $X$  的引用之间的依赖关系。因为语句  $s_2$  中的访问不是仿射的，所以在涉及语句  $s_2$  的依赖分析中，我们把矩阵  $X$  建模为一个标量变量，从而近似地处理这个访问。令  $(i, j)$  为  $s_1$  的一个动态实例的下标值，令  $i'$  为  $s_2$  的一个动态实例的下标值。令语句  $s_1$  和  $s_2$  的计算任务的映射分别为  $\langle [C_{11}, C_{12}], c_1 \rangle$  和  $\langle [C_{21}], c_2 \rangle$ 。

让我们首先考虑从  $s_1$  到  $s_2$  的依赖关系所决定的时间分划约束。这样，如果  $i \leq i'$ ，那么转换之后的  $s_1$  的第  $(i, j)$  个迭代不得晚于转换之后的  $s_2$  的第  $i'$  个迭代。也就是说

$$\begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + c_1 \leq C_{21} i' + c_2$$

展开后得到

$$C_{11}i + C_{12}j + c_1 \leq C_{21}i' + c_2$$

因为  $j$  和  $i$  及  $i'$  无关，所以可以取任意大的值，因此  $C_{12} = 0$  必须成立。可知，这个约束的一个可能的解是

$$C_{11} = C_{21} = 1 \text{ 且 } C_{12} = c_1 = c_2 = 0$$

对于从  $s_2$  到  $s_1$  以及从  $s_2$  到其自身的依赖关系的分析将得到类似的结果。外层循环的第  $i$  个迭代由  $s_2$  的第  $i$  个实例和  $s_1$  的所有实例  $(i, j)$  组成。在这个特定的解中，这个迭代将被分配给第  $i$  个时间步骤。选择其他的  $C_{11}, C_{12}, C_{21}, c_1, c_2$  的值会得到类似的分配方法，但是会存在一些不进行任何运算的时间步骤。也就是说，调度这个外层循环的所有方法都要求其中的迭代按照与原代码同样的顺序进行。不管全部的 100 个迭代是在同一个处理器上执行，还是在 100 个不同的处理器上执行，又或在 1~100 之间的任意多个处理器上运行，上面的论断都成立。□

**例 11.58** 在图 11-50a 中显示的 SOR 代码中，写引用  $X[j+1]$  和它本身，以及代码中的三个读引用之间具有依赖关系。我们要为该赋值语句寻找计算任务的映射  $\langle [C_1, C_2], c \rangle$ ，使得如果存在从  $(i, j)$  到  $(i', j')$  的依赖关系，那么

$$\begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [c] \leq \begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} + [C]$$

根据定义， $(i, j) \prec (i', j')$ ，也就是说，要么  $i < i'$ ，要么  $(i = i' \wedge j < j')$ 。

让我们考虑三对数据依赖关系：

1) 从写访问  $X[j+1]$  到读访问  $X[j+2]$  之间的真依赖关系。因为它们的实例必须访问同一个位置，因此  $j+1 = j'+2$ ，或者说  $j = j'+1$ 。把  $j = j'+1$  替换到时间约束中，我们得到

$$C_1(i' - i) - C_2 \geq 0$$

由  $j = j'+1$  可知  $j > j'$ ，上面的先后关系约束归约为  $i < i'$ 。因此

$$C_1 - C_2 \geq 0$$

2) 从读访问  $X[j+2]$  到写访问  $X[j+1]$  的反依赖关系。这里  $j+2 = j'+1$ ，即  $j = j'-1$ 。把  $j = j'-1$  代入时间约束我们得到

$$C_1(i' - i) + C_2 \geq 0$$

当  $i = i'$  时得

$$C_2 \geq 0$$

当  $i < i'$  时，因为  $C_2 \geq 0$ ，我们得到

$$C_1 \geq 0$$

3) 从写访问  $X[j+1]$  到自身的输出依赖。这里  $j=j'$ 。时间约束被归约为

$$C_1(i' - i) \geq 0$$

因为只有  $i < i'$  是相关的，我们还是得到

$$C_1 \geq 0$$

其余的依赖关系没有产生任何新的约束。总共有三个约束：

$$C_1 \geq 0$$

$$C_2 \geq 0$$

$$C_1 - C_2 \geq 0$$

下面是这些约束的两个独立解：

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

第一个解保持了最外层循环的迭代执行顺序。图 11-50a 中原来的 SOR 代码和图 11-51a 中转换得到的代码都是这种安排的例子。第二个解把沿着  $135^\circ$  斜线上的迭代放置在外层循环的同一个迭代中。图 11-51b 中显示的是具有这种最外层循环组成方式的代码的一个例子。

请注意，存在多个其他可能的独立解对，比如

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

也是具有同样约束的独立解。我们选择最简单的向量来简化代码转换。□

### 11.9.7 用 Farkas 引理求解时间分划约束

因为时间分划约束和空间分划约束类似，那么是否可以用一个类似的算法来求解这些约束呢？遗憾的是，两类问题之间的少许不同使得两个解决方法在技术上存在很大不同。算法 11.43 直接求解  $C_1, c_1, C_2$  和  $c_2$  的满足下面条件的值，使得对于所有  $Z^{d_1}$  中  $i_1$  和  $Z^{d_2}$  中的  $i_2$ ，如果

$$F_1 i_1 + f_1 = F_2 i_2 + f_2$$

成立，那么

$$C_1 i_1 + c_1 = C_2 i_2 + c_2$$

根据循环界限而得到的线性不等式只用于确定两个引用是否具有数据依赖关系，没有其他用途。

为了找出时间分划约束的解，我们不能忽略线性不等式  $i < i'$ ，忽略它们经常会使得只存在平凡解，而平凡解会把所有的迭代都放到同一个分划单元中。因此，寻找时间分划约束解的算法必须同时处理等式和不等式。

我们希望解决的一般性问题是：给定一个矩阵  $A$ ，找出一个向量  $c$  使得对于所有满足  $Ax \geq 0$  的向量  $x$ ，都有  $c^T x \geq 0$ 。换句话说，我们在寻找向量  $c$ ，使得  $c$  和  $Ax \geq 0$  所定义的多面体之内任何向量的内积总是大于 0。

这个问题可以用 Farkas 引理解决。令  $A$  为一个  $m \times n$  的实数矩阵，且  $c$  为一个实数、非零的  $n$  维向量。Farkas 引理说要么原不等式系统

$$Ax \geq 0 \quad c^T x < 0$$

具有一个实数解  $x$ ，要么相应的对偶系统

$$A^T y = c, \quad y \geq 0$$

具有一个实数解  $y$ ，但是两者不会同时成立。

这个对偶系统可以用 Fourier-Motzkin 消除法进行处理，通过投影消除变量  $y$ 。这个引理保证，对于每个对偶系统中有解的向量  $c$ ，原系统不存在解。换句话说，我们可以找到对偶系统  $A^T y = c, y \geq 0$  的解，从而证明系统的否命题，即对于所有满足  $Ax \geq 0$  的  $x$ ，都有  $c^T x \geq 0$ 。

### 关于 Farkas 引理

关于这个引理的证明可以在很多关于线性规划的标准课本中找到。最早在 1901 年被证明的 Farkas 引理是择一性定理之一。这些定理相互等价，但是尽管经过了多年的尝试，人们仍然没有找到有关这个引理或者它的某个等价定理的简单、直观的证明。

**算法 11.59.** 为一个外层的顺序循环找到一个合法的最大线性独立的仿射时间分划映射。

**输入：**一个带有数组访问的循环嵌套结构。

**输出：**线性独立时间分划映射的最大集。

**方法：**算法包括以下步骤：

- 1) 找出程序中所有具有数据依赖关系的访问对。
- 2) 对于每一对具有数据依赖的访问，在循环结构  $d_1$  中的语句  $s_1$  中的访问  $\mathcal{F}_1 = \langle F_1, f_1, B_1, b_1 \rangle$  和嵌套在循环结构  $d_2$  中的语句  $s_2$  的访问  $\mathcal{F}_2 = \langle F_2, f_2, B_2, b_2 \rangle$ ，令  $\langle C_1, c_1 \rangle$  和  $\langle C_2, c_2 \rangle$  分别为语句  $s_1$  和  $s_2$  的（未知的）时间分划映射。回顾一下，时间分划约束是说：

- 如果  $Z^{d_1}$  中所有的  $i_1$  和  $Z^{d_2}$  中的  $i_2$  满足下列条件，

- 1)  $i_1 \prec_{s_1 s_2} i_2$
- 2)  $B_1 i_1 + b_1 \geq 0$
- 3)  $B_2 i_2 + b_2 \geq 0$
- 4)  $F_1 i_1 + f_1 = F_2 i_2 + f_2$

那么必然有  $C_1 i_1 + c_1 \leq C_2 i_2 + c_2$

因为  $i_1 \prec_{s_1 s_2} i_2$  是多个子句的析取式，因此我们可以为每个子句创立一个约束系统，并单独对它们求解。方法如下：

- (a) 和算法 11.43 的步骤 2(a) 类似，对方程

$$F_1 i_1 + f_1 = F_2 i_2 + f_2$$

应用高斯消除法把向量

$$\begin{bmatrix} i_1 \\ i_2 \\ 1 \end{bmatrix}$$

归约为某个未知量的向量  $x$ 。

- (b) 令  $c$  为这个分划映射中的所有未知量。把因为分划映射而产生的线性不等式约束可写成

$$c^T D x \geq 0$$

其中  $D$  为一个矩阵。

- (c) 把循环下标变量的先后关系约束和循环边界表示为

$$A x \geq 0$$

其中  $A$  为一个矩阵。

- (d) 应用 Farkas 引理。找到满足上面两个约束的  $x$  的任务等价于寻找满足下列条件的  $y$ ：

$$A^T y = D^T c \text{ 且 } y \geq 0$$

请注意，这里的  $c^T D$  就是 Farkas 引理中的  $c^T$ ，而且我们使用的是这个引理的否定形式。

④ 在这个形式中，应用 Fourier-Motzkin 消除法把  $y$  的变量通过投影消除，并把关于系数  $c$  的约束表示为  $Ec \geq 0$ 。

⑤ 令  $E'c' \geq 0$  为不包含常量项的约束。

3) 使用附录 B 中的算法 B.1，找出  $E'c' \geq 0$  的线性独立解的最大集合。这一复杂的算法的基本思路是跟踪每个语句的当前解集，并通过插入一些约束不断寻找更多的独立解。这些被插入的约束会保证相应的解至少对于一个语句来说是线性独立的。

4) 根据找到的每个解  $c'$  得到一个仿射时间分划映射。映射的常量项通过不等式  $Ec \geq 0$  得到。  $\square$

**例 11.60** 例 11.57 的约束可以写作

$$\begin{bmatrix} -C_{11} & -C_{12} & C_{21} & (c_2 - c_1) \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq 0$$

$$\begin{bmatrix} -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ 1 \end{bmatrix} \geq 0$$

Farkas 引理说这些约束和

$$\begin{bmatrix} -1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} z \end{bmatrix} = \begin{bmatrix} -C_{11} \\ -C_{12} \\ C_{21} \\ c_2 - c_1 \end{bmatrix} \text{ 且 } z \geq 0$$

等价。解这个不等式系统，我们得到

$$C_{11} = C_{21} \geq 0 \text{ 且 } C_{12} = c_2 - c_1 = 0$$

请注意，我们在例 11.57 中得到的特定解满足这些约束。  $\square$

### 11.9.8 代码转换

如果一个循环嵌套结构的时间分划约束存在  $k$  个独立解，那么就可能把这个循环嵌套结构转换成为具有  $k$  个最外层完全可交换循环的结构。可以对这个结构进行转换得到  $k-1$  度的流水线，或得到  $k-1$  个可并行化的内层循环。而且，我们还可以对完全可交换循环应用分块技术，以提高单处理器系统的数据局部性或降低并行执行中的处理器之间的同步开销。

#### 利用完全可交换循环

如果一个循环嵌套结构的时间分划约束具有  $k$  个独立解，我们就可以容易地根据这些解生成一个循环嵌套结构，其最外层的  $k$  个循环是完全可交换的循环。通过直接把第  $k$  个解变成新转换的第  $k$  行，我们就可以得到这样的转换。一旦构造出这个仿射变换，我们就可以使用算法 11.45 来生成代码。

**例 11.61** 在例 11.58 中为我们的 SOR 例子找到的解是

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

令第一个解为转换后的第一行且第二个解为转换后的第二行，我们得到转换

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

这个转换生成图 11-51a 中的代码。

如果我们把第二个解作为第一行，我们可以得到转换

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

它生成图 11-51c 中的代码。□

显然，这样的转换产生了一个合法的顺序程序。第一行按照第一个解来分划整个迭代空间。时间约束保证这样的分解不会违反任何数据依赖关系。然后，我们根据第二个解对各个最外层循环中的迭代进行分划。这个分划必然是合法的，原因是处理的是原来的迭代空间的子集。对于矩阵中的其余各行，以上讨论仍然成立。因为我们可以任意排列这些解，所以这些循环是完全可交换的。

### 利用流水线化技术

我们可以轻易地把一个具有  $k$  个最外层完全可交换循环的循环嵌套结构转换成为一个具有  $k - 1$  度流水线并行性的代码。

**例 11.62** 让我们回到 SOR 的例子。在例子中的循环都被转换为完全可交换的循环之后，我们知道只要在迭代  $[i_1, i_2 - 1]$  和  $[i_1 - 1, i_2]$  执行之后，迭代  $[i_1, i_2]$  就可以被执行。我们可以用如下方法在一个流水线中保证这个顺序。我们把迭代  $i_1$  分配给处理器  $p_1$ 。每个处理器按照原来的顺序执行内层循环中的迭代，因此保证了迭代  $[i_1, i_2]$  在迭代  $[i_1, i_2 - 1]$  之后执行。另外，我们要求处理器  $p$  在执行迭代  $[p, i_2]$  之前必须等待处理器  $p - 1$  的信号，这个信号表明处理器  $p - 1$  已经执行了迭代  $[p - 1, i_2]$ 。这个技术可以根据图 11-51a 和图 11-51b 中的完全可交换循环分别生成图 11-52a 和图 11-52b 中的代码。□

一般来说，给定  $k$  个最外层的完全可交换循环，具有下标值  $(i_1, \dots, i_k)$  的迭代可以执行且不违反数据依赖约束的前提是下列迭代

$$[i_1 - 1, i_2, \dots, i_k], [i_1, i_2 - 1, i_3, \dots, i_k], \dots, [i_1, \dots, i_{k-1}, i_{k-1}]$$

已经执行完毕。因此，我们可以按照如下方法把这个迭代空间的前  $k - 1$  个维度的分划分配到  $O(n^{k-1})$  个处理器上。每个处理器负责一个迭代的集合，该集合中迭代的下标值在前  $k - 1$  个维度上相同，而第  $k$  个下标值则包括了该下标的全部可能值。每个处理器顺序地执行第  $k$  个循环中的迭代。前  $k - 1$  个循环下标值  $[p_1, p_2, \dots, p_{k-1}]$  所对应的处理器可以执行第  $k$  个循环的第  $i$  个迭代的前提是它收到了处理器

$$[p_1 - 1, p_2, \dots, p_{k-1}], \dots, [p_1, \dots, p_{k-2}, p_{k-1} - 1]$$

发出的信号，表明它们已经执行完了各自的第  $k$  个循环中的第  $i$  个迭代。

### 波阵面化

根据一个具有  $k$  个最外层完全可交换循环的循环结构生成  $k - 1$  个可并行化内层循环是比较容易的。虽然我们更倾向于使用流水线化，但为完整起见，我们仍在这里给出这个方法。

我们使用一个新的下标变量  $i'$  来分划一个具有  $k$  个最外层完全可交换循环的循环结构的计算任务，其中  $i'$  被定义为这  $k$  个可交换循环中所有下标的某种组合。比如， $i' = i_1 + \dots + i_k$  就是这样的一个组合。

我们创建一个最外层的顺序循环，该循环以升序遍历这个  $i'$  分划，在各个分划单元中的计算任务依然按以前的顺序执行。每个分划单元中的前  $k - 1$  个循环一定是可并行化的。直观地讲，

如果给定一个二维的迭代空间，这个转换沿着 $135^{\circ}$ 的斜线把迭代组合起来，作为外层循环的一次执行。这个策略保证了在最外层循环中的各个迭代之间没有数据依赖。

### 分块

一个深度为  $k$  的完全可交换的循环嵌套结构可以在  $k$  个维度上进行分块。我们可以把多个迭代的块组合成为一个单元，而不是根据最外层或者最内层的循环下标值把迭代分配给处理器。分块技术可以用于增强数据局部性并最小化流水线的开销。

假设我们有一个二维完全可交换的循环嵌套结构，如图 11-55a 所示，且我们希望把这个结构的计算任务分成  $b \times b$  块。分块后的代码的执行顺序如图 11-56 所示，等价的代码显示在图 11-55b 中。

```
for (i=0; i<n; i++)
    for (j=1; j<n; j++) {
        <S>
    }
```

a) 一个简单的循环嵌套结构

```
for (ii = 0; ii<n; ii+=b)
    for (jj = 0; jj<n; jj+=b)
        for (i = ii*b; i <= min(ii*b-1, n); i++)
            for (j = ii*b; j <= min(jj*b-1, n); j++) {
                <S>
            }
        }
```

b) 这个循环嵌套结构的分块版本

图 11-55 一个二维循环嵌套结构和它的分块版本

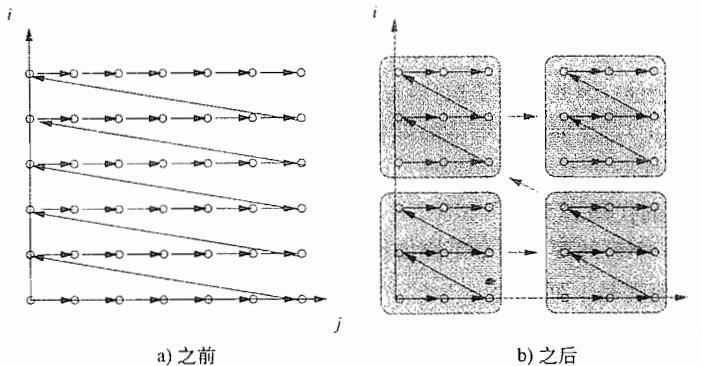


图 11-56 在对一个深度为 2 的循环嵌套结构分块之前和分块之后的执行顺序

如果我们把每个块分配给一个处理器，那么在同一个块中从一个迭代到另一个迭代的数据传递不需要处理器之间的通信。我们还可以把块的一列分配给一个处理器，以便加粗流水线的粒度。请注意，每个处理器只在块的边界上和它的前驱及后继进行通信。因此，分块的另一个优点是程序只需要在块和它的邻居块的边界上交换被访问的数据。处于块内部的数据仅由一个处理器处理。

**例 11-63** 现在我们使用一个真实的数值算法——Cholesky 分解——来说明算法 11.59 是如何在只有流水线化并行性的情况下处理单循环嵌套结构的。图 11-57 中显示的代码实现了一个  $O(n^3)$  的算法，该算法对一个二维数据数组进行运算。被执行的迭代

```
for (i = 1; i <= N; i++) {
    for (j = 1; j <= i-1; j++) {
        for (k = 1; k <= j-1; k++)
            X[i,j] = X[i,j] - X[i,k] * X[j,k];
        X[i,j] = X[i,j] / X[j,j];
    }
    for (m = 1; m <= i-1; m++)
        X[i,i] = X[i,i] - X[i,m] * X[i,m];
    X[i,i] = sqrt(X[i,i]);
}
```

图 11-57 Cholesky 分解

空间是一个三角形的金字塔结构，因为  $j$  只会遍历到外层循环下标  $i$  的值，而  $k$  只会遍历到  $j$  的值。这个循环结构有四个语句，各个语句都嵌套在不同的循环中。

对这个程序应用算法 11.59 可以找到三个合法的时间维度。它把所有的运算都嵌入到一个三维的完全可交换的循环嵌套结构中去。其中的某些运算在原程序中是嵌套在深度为 1 或 2 的循环结构中的。图 11-58 中显示了得到的代码和相应的映射。

```

for (i2 = 1; i2 <= N; i2++)
    for (j2 = 1; j2 <= i2; j2++) {
        /* 处理器(i2,j2)的代码的开始 */
        for (k2 = 1; k2 <= i2; k2++) {

            // 映射: i2 = i, j2 = j, k2 = k
            if (j2 < i2 && k2 < j2)
                X[i2,j2] = X[i2,j2] - X[i2,k2] * X[j2,k2];

            // 映射: i2 = i, j2 = j, k2 = j
            if (j2 == k2 && j2 < i2)
                X[i2,j2] = X[i2,j2] / X[j2,j2];

            // 映射: i2 = i, j2 = i, k2 = m
            if (i2 == j2 && k2 < i2)
                X[i2,i2] = X[i2,i2] - X[i2,k2] * X[i2,k2];

            // 映射: i2 = i, j2 = i, k2 = i
            if (i2 == j2 && j2 == k2)
                X[k2,k2] = sqrt(X[k2,k2]);
        }
        /* 处理器(i2,j2)的代码的结束 */
    }
}

```

图 11-58 写成完全可交换循环结构的图 11-57 的代码

代码生成例程保证了运算的执行都位于原来的循环界限中，以保证新的程序只执行原来代码中的运算。我们可以把这个代码流水线化，方法是把这个三维结构映射到二维的处理器空间中。迭代  $(i_2, j_2, k_2)$  被分配给 ID 为  $(i_2, j_2)$  的处理器。每个处理器执行循环下标为  $k_2$  的最内层的循环。在它执行第  $k$  个迭代之前，这个处理器会等待 ID 为  $(i_2 - 1, j_2)$  和  $(i_2, j_2 - 1)$  的处理器发来的信号。在它执行了它的迭代之后，它会给处理器  $(i_2 + 1, j_2)$  和  $(i_2, j_2 + 1)$  发出信号。□

### 11.9.9 具有最小同步量的并行性

在前面的三节中，我们已经描述了三个功能强大的并行化算法：算法 11.43 可以找出所有不需要同步的并行性，算法 11.54 找出了所有只需要固定多次同步的并行性，而算法 11.59 找出了所有需要  $O(n)$  次同步的可流水线化的并行性，其中  $n$  是最外层循环的迭代数量。粗略地说，我们的目标是尽可能多地把一个计算过程并行化，同时尽量少地引入同步运算。

下面的算法 11.64 从最粗糙的并行性粒度开始，找出了一个程序中存在的所有并行度。在实践中，在为某个多处理器系统并行化一个代码时，我们并不需要利用所有层次上的并行性。我们总是利用最外层的并行性，直到所有的计算任务都被并行化，并且所有的处理器都被完全利用为止。

**算法 11.64** 找出一个程序中存在的所有并行度，同时所有并行性的粒度都尽可能地粗糙。

**输入：**一个要进行并行化的程序。

**输出：**同一个程序的并行化版本。

**方法：**完成下列步骤：

- 1) 找出不需要同步运算的并行性的最大度数，对这个程序应用算法 11.43。
- 2) 找出需要  $O(1)$  次同步运算的并行性的最大度数：对步骤 1 中找出的所有空间分划单元应用算法 11.54。（如果在步骤 1 中没有找到无同步的并行性，那么所有的计算任务都在同一个分划单元中。）
- 3) 找出需要  $O(n)$  次同步运算的最大并行性度数。对步骤 2 中找到的每个分划单元应用算法 11.59，以找出可流水线化的并行性。然后对分配给各个处理器的分划单元逐个应用算法 11.54。如果前面没有找到流水线化的并行性，就对串行循环的循环体应用算法 11.54。
- 4) 在逐步增加同步度数的情况下寻找最大的并行性度数。递归地把步骤 3 应用到上一步生成的各个空间分划单元中的计算任务上。  $\square$

**例 11.65** 现在让我们回到例 11.56。算法 11.64 的步骤 1 和 2 都没有找到并行性。也就是说，为了并行化这个代码，我们需要的同步量大于一个常量。在步骤 3 中，应用算法 11.59 确定了只有一个合法的外层循环，这个循环就是图 11-53 中的原代码中的循环。因此，这个循环不具有可流水线化的并行性。在步骤 3 的第二部分，我们应用算法 11.54 来并行化内层循环。我们像处理整个程序那样来处理一个分划单元中的代码，不同之处仅在于我们像处理符号常量那样处理了分划单元的编号。在这个例子中，我们发现内层循环是可并行化的，因此这个代码可以使用  $n$  个同步栅障进行并行化。  $\square$

算法 11.64 找出了一个程序中在各个同步层面上的并行性。这个算法优先求出需要较少同步量的并行化方案，但是最少同步运算并不表示通信量是最少的。这里我们讨论这个算法的两个扩展，以解决此算法的弱点。

#### 考虑通信开销

如果没有发现无同步的并行性，算法 11.64 的步骤 2 对每个强连通分量独立地进行并行化。但是，某些这样的分量仍然可能在无同步和通信的情况下被并行化。解决方法之一是尽可能地在程序依赖图中共享大部分数据的子集之间寻找无同步的并行性。

如果强连通分量之间的通信是必须的，我们注意到有些通信的开销要高过其他通信的开销。比如，转置一个矩阵的开销要比两个相邻处理器之间通信的开销高得多。假设  $s_1$  和  $s_2$  分别是两个分离的强连通图中的语句，它们分别在迭代  $i_1$  和  $i_2$  中访问相同的数据。如果我们不能分别为语句  $s_1$  和  $s_2$  找到分划映射  $\langle C_1, c_1 \rangle$  和  $\langle C_2, c_2 \rangle$  使得

$$C_1 i_1 + c_1 - C_2 i_2 - c_2 = 0$$

我们就试图满足约束

$$C_1 i_1 + c_1 - C_2 i_2 - c_2 \leq \delta$$

其中  $\delta$  是一个小的常量。

#### 用通信量交换同步量

有些时候，我们宁愿多进行一些同步运算以降低通信量。例 11.66 讨论了一个这样的例子。因此，如果我们不能在只允许相邻的强连通分量之间进行通信的情况下对一个代码进行并行化，我们将试图把这个计算任务流水线化，而不是独立地并行化各个分量。如例 11.66 所示，流水线化技术可以被应用到一个循环序列中。

**例 11.66** 对于例 11.49 中的 ADI 集成算法，我们已经说明对第一和第二个循环嵌套结构进行独立优化可以在每个嵌套结构中找到并行性。但是，这样的方案要求在循环之间进行矩阵转置，从而出现  $O(n^2)$  的数据流量。如果我们使用算法 11.59 来寻找可流水线化的并行性，就可以把整

个程序转换成为一个完全可交换的循环嵌套结构，如图 11-59 所示。然后，我们可以应用分块技术来降低通信开销。这个方案将带来  $O(n)$  次的同步，但是需要的通信量要小很多。□

### 11.9.10 11.9 节的练习

```
for (j = 0; j < n; j++)
    for (i = 1; i < n+1; i++) {
        if (i < n) X[i,j] = f(X[i,j] + X[i-1,j])
        if (j > 0) X[i-1,j] = g(X[i-1,j], X[i-1,j-1]);
    }
```

图 11-59 例 11.49 的代码的一个完全可交换循环嵌套结构

**练习 11.9.1：**在 11.9.4 节中，我们讨论了使用倾斜的轴，而不是用水平轴或垂直轴来将图 11-51 中的代码流水线化的能力。对于以下度数的斜线，写出和图 11-52 中的循环类似的代码：  
①  $135^\circ$ ，②  $120^\circ$ 。

**练习 11.9.2：**如果  $b$  能够整除  $n$ ，那么图 11-55b 可以进一步简化。按照这种假设改写代码。

**练习 11.9.3：**图 11-60 中是一个计算 Pascal 三角形的前 100 行的程序。也就是说，对  $0 \leq j \leq i < 100$ ， $P[i,j]$  将变成从  $i$  个物体中选择  $j$  个物体的方法总数。

1) 把这个代码改写为单一的、完全可交换的循环嵌套结构。

2) 在一个流水线中使用 100 个处理器来实现这个代码。为每个处理器  $p$  写出其代码，并指出必要的同步运算。

3) 使用边长为 10 个迭代的块改写

这个代码。因为迭代空间形成了一个三角形，总共只有  $1 + 2 + \dots + 10 = 55$  个块。用  $p_1, p_2$  作为参数来表示一个处理器  $(p_1, p_2)$  的代码，该处理器被分配给  $i$  方向上的第  $p_1$  个块和  $j$  方向上的第  $p_2$  个块。

**练习 11.9.4：**对图 11-61 中的代码重复练习 11.9.3。但是请注意，这个问题的迭代形成了一个边长为 100 的三维立方体。因此，问题 3 中的块应该是  $10 \times 10 \times 10$ ，且有 1000 个块。

**练习 11.9.5：**让我们对一个简单的时间分划约束的例子应用算法 11.59。在下面的内容中假设向量  $i_1$  是  $(i_1, j_1)$ ，向量  $i_2$  是  $(i_2, j_2)$ 。从技术上讲，这两个向量都是转置的。条件  $i_1 \prec_{s_1 s_2} i_2$  由下列子句的析取式构成：

①  $i_1 < i_2$ ，或者

②  $i_1 = i_2$  且  $j_1 < j_2$

其他的等式和不等式是

```
for (i=0; i<100; i++) {
    P[i,0] = 1; /* s1 */
    P[i,i] = 1; /* s2 */
}
for (i=2; i<100; i++)
    for (j=1; j<i; j++)
        P[i,j] = P[i-1,j-1] + P[i-1,j]; /* s3 */
```

图 11-60 计算 Pascal 三角形

```
for (i=0; i<100; i++) {
    A[i, 0, 0] = B1[i]; /* s1 */
    A[i, 99, 0] = B2[i]; /* s2 */
}
for (j=1; j<99; j++) {
    A[0, j, 0] = B3[j]; /* s3 */
    A[99, j, 0] = B4[j]; /* s4 */
}
for (i=0; i<99; i++)
    for (j=0; j<99; j++)
        for (k=1; k<100; k++)
            A[i, j, k] = 4*A[i, j, k-1] + A[i-1, j, k-1] +
                A[i+1, j, k-1] + A[i, j-1, k-1] +
                A[i, j+1, k-1]; /* s5 */
```

图 11-61 练习 11.94 的代码

$$2i_1 + j_1 - 10 \geq 0$$

$$i_2 + 2j_2 - 20 \geq 0$$

$$i_1 = i_2 + j_2 - 50$$

$$j_1 = j_2 + 40$$

最后, 时间分划不等式如下, 其中  $c_1, d_1, e_1, c_2, d_2$  和  $e_2$  为未知量:

$$c_1 i_1 + d_1 j_1 + e_1 \leq c_2 i_2 + d_2 j_2 + e_2$$

- 1) 对情况①, 即  $i_1 < i_2$ , 求解这个时间分划约束。特别地, 你需要尽可能地消除  $i_1, j_1, i_2$  和  $j_2$ , 并像算法 11.59 中那样设置矩阵  $D$  和  $A$ 。然后对得到的矩阵不等式应用 Farkas 引理。
- 2) 对于情况②, 即  $i_1 = i_2$  且  $j_1 < j_2$ , 重复问题 1。

## 11.10 局部性优化

不管一个处理器是不是某个处理器系统的一部分, 它的性能和它的高速缓存的行为密切相关。高速缓存中的脱靶可能要花费几十个时钟周期, 因此高速缓存的高脱靶率会使处理器性能降低。在带有公共内存总线的多处理器系统的背景下, 对总线的竞争会进一步加剧低劣的数据局部性所带来的损失。

我们将看到, 即使我们只是希望提高单处理器系统的数据局部性, 用于并行化的仿射分划算法也是有用的。它是一个有效的寻找循环转换机会的工具。在本节中, 我们描述了三种在单处理器系统和多处理器系统中提高数据局部性的技术。

1) 我们试图在计算结果生成之后尽快地使用它们, 以提高计算结果的时间局部性。提高时间局部性的方法是把一个计算任务分割成为独立的分划单元, 并把各个分划单元中具有依赖关系的运算紧靠在一起执行。

2) 数组收缩(array contraction)技术降低了一个数组的维度, 且降低了被访问内存位置的数目。如果在任意给定时刻该数组只有一个位置被使用, 我们就可以应用数组收缩技术。

3) 除了提高计算结果的时间局部性, 我们也需要优化计算结果的空间局部性, 同时优化只读数据的时间和空间局部性。我们可以交替执行多个分划单元, 而不是一个接一个地执行它们。这样做可以使得分划单元之间的复用更加靠近。

### 11.10.1 计算结果数据的时间局部性

仿射分划算法把所有相互依赖的运算放到一起。通过串行执行这些分划, 我们提高了计算结果数据的时间局部性。让我们回顾一下 11.7.1 节中讨论的多网格的例子。应用算法 11.43 来并行化图 11-23 中的代码, 找到了 2 度的并行性。图 11-24 中的代码包含两个外层循环, 它们顺序遍历了各个独立的分划单元。转换得到的这个代码具有较好的局部性, 因为计算得到的结果立刻就在同一个迭代中使用。

因此, 即使我们的目标是优化顺序执行, 使用并行化技术找出这些相关的运算并把它们放到一起也是很有好处的。我们在这里使用的算法和算法 11.64 类似, 它从最外层循环开始寻找所有的并行性粒度。如 11.9.9 节中讨论的, 如果我们在每个层次上都不能找到无同步的并行性, 这个算法就对各个强连通分量单独进行并行化。这个并行化方法常常会增加通信量。因此, 如果被单独并行化的强连通分量之间存在复用, 我们就尽可能地把它们组合起来。

### 11.10.2 数组收缩

数组收缩优化给出了另一个在存储和并行性之间进行折衷处理的例子。这种折衷方案首先在 10.2.3 节中讨论指令级并行性时引入。使用更多寄存器可以得到更大的指令级并行性。同样, 使用更多的内存可以得到更多的循环级并行性。如 11.7.1 节中的多网格例子所示, 把一个临时的标量变量变成一个数组就可以允许不同的迭代使用这个临时变量的不同实例, 也就允许它们同时执行。反过来, 如果我们有一个每次只操作一个数组元素的顺序执行过程, 就可以收缩这个数组, 把它替换为一个标量, 并让各个迭代使用同一个位置。

在图 11-24 中显示的转换得到的多网格程序中，内层循环的每个迭代生成并消耗了  $AP$ 、 $AM$ 、 $T$  以及  $D$  的一行中的不同元素。如果这些数组不会在这个代码段之外使用，这些迭代就可以串行地复用同一个数据存储位置，而不是把这些值分别存放在不同的元素中或者不同行中。图 11-62 显示了减少这些数组的维度之后的结果。这个代码比原来的代码运行得更快，因为它读写的数据更少。特别是当一个数组被归约成一个标量变量时，我们可以把这个变量放在一个寄存器中，并完全消除了对内存访问的需求。

```

for (j = 2, j <= jl, j++) {
    for (i = 2, i <= il, i++) {
        AP      = ...;
        T       = 1.0/(1.0 + AP);
        D[2]   = T * AP;
        DW[1,2,j,i] = T * DW[1,2,j,i];
        for (k=3, k <= kl-1, k++) {
            AM      = AP;
            AP      = ...;
            T       = ... AP - AM * D[k-1] ...;
            D[k]   = T * AP;
            DW[1,k,j,i] = T * (DW[1,k,j,i] + DW[1,k-1,j,i]) ...
        }
        ...
        for (k=kl-1, k>=2, k--)
            DW[1,k,j,i] = DW[1,k,j,i] + D[k] * DW[1,k+1,j,i];
    }
}

```

图 11-62 对图 11-23 进行分划(图 11-24)和  
数组收缩之后的得到的代码

因为使用的内存更少，所以可用的并行性也变少了。转换得到的图 11-62 中的代码的迭代之间有了数据依赖关系，因此不能再并行执行。为了把代码在  $P$  个处理器上并行执行，我们可以把每个标量变量扩展出  $P$  个副本，并让每个处理器访问自己的副本。这样，内存扩展的数量就和被利用的并行性直接相关了。

通常，要寻找数组收缩机会的理由有三个：

1) 用于科学应用的高级程序设计语言(比如 Matlab 和 Fortran90)支持数组层次的运算。数组运算的每个子表达式都生成一个临时数组。因为这些数组可能很大，每个数组运算，比如乘法或加法，需要读写很多内存位置，同时对算术运算的需求相对较少。因此，我们对运算进行重新排序以便数据被生成后立刻就被消耗掉，同时也就把这些数组收缩成了标量变量。这样的处理是很重要的。

2) 在 20 世纪 80 和 90 年代构造的超级计算机都是向量机，因此那时开发的很多科学计算应用都是针对这样的机器进行优化的。虽然存在向量化编译器，但很多程序员依然把他们的代码写成每次完成一次向量运算的方式。本章中多网格代码的例子就是这种风格的程序的例子。

3) 收缩优化的机会也会由编译器引入。如多重网格例子中的变量  $T$  所演示的，一个编译器可能会扩展数组以提高并行性。如果这种空间扩展是不必要的，那么我们就必须对这个数组进行收缩处理。

### 例 11.67 数组表达式 $Z = W + X + Y$ 被翻译成为

```

for (i=0; i<n; i++) T[i] = W[i] + X[i];
for (i=0; i<n; i++) Z[i] = T[i] + Y[i];

```

把这个代码改写成

```
for (i=0; i<n; i++) { T = W[i] + X[i]; Z[i] = T + Y[i] }
```

可以极大地提高代码的运行速度。当然，在 C 代码的层次上我们甚至不需要使用临时变量  $T$ ，而是可以把对  $Z[i]$  的赋值语句写成单个语句。但这里我们正试图对中间代码层次进行建模。在这个层次上一个向量处理器将会处理这些运算。□

**算法 11.68** 数组收缩。

**输入：**一个由算法 11.64 转换得到的程序。

**输出：**一个等价的程序，但降低了数组的维度。

**方法：**一个数组的维度可以被收缩为一个元素的条件如下：

- 1) 每个独立的分划单元只使用这个数组的一个元素。
- 2) 这个元素在分划单元入口处的值没有被这个分划单元使用，且
- 3) 这个元素的值在这个单元的出口处不活跃。

找出可收缩的维度，也就是满足上面三个条件的维度，并把它们替换为单个元素。  $\square$

算法 11.68 假设这个程序首先由算法 11.64 进行转换，把所有相互依赖的运算都分配到同一个分划单元中，并顺序地执行这些分划单元。它找出了其元素在不同迭代中活跃范围不相交的数组变量。如果这些变量在循环之后不再活跃，它就可以收缩这个数组并让处理器在同一个标量位置上进行运算。在数组收缩之后，可能还有必要选择性地扩展一些数组，以应对并行性和其他局部性优化问题。

这里要进行的活跃性分析比 9.2.5 节中所描述的分析更加复杂。如果数组被定义为一个全局变量，或者它是一个参数，我们就需要使用过程间分析技术来保证不使用出口处的值。不仅如此，我们还需要计算单个数组元素的活跃性，保守地把数组当作一个标量进行活跃性分析会使结果不够精确。

### 11.10.3 分划单元的交织

一个循环中的不同分划单元经常读取同样的数据，或者读写同样的高速缓存线。在本节和接下来的两节中，我们将讨论当发现了分划单元之间的复用时如何优化局部性。

#### 最内层块的复用

我们采用一个简单的模型，即如果一个数据在少量迭代之后就被复用，那么就可以在高速缓存中找到这个数据。如果最内层循环具有很大或未知的界限，那么只有最内层循环的迭代之间的复用才能够带来更好的局部性。分块处理过程创建了具有较小且已知界限的内层循环，使得我们可以充分利用整个计算块之内或块之间的复用。因此，分块技术具有促进更多维度复用的作用。

**例 11.69** 考虑图 11-5 中显示的矩阵乘法代码以及图 11-7 中该代码的分块版本。矩阵乘法在它的三维迭代空间中的每一个维度上都有复用。在原来的代码中，最内层循环具有  $n$  个迭代，其中  $n$  是未知的，且可能很大。我们的简单模型假设只有跨越最内层循环迭代的被复用数据才可以在高速缓存中找到。

在分块版本中，最内层的三个循环执行了一个三维的计算任务块。这个三维块的每条边长都是  $B$  个迭代。这个块的大小是由编译器选择的。这个大小必须足够小，使得在计算分块时读写的所有高速缓存线能够一起放到高速缓存中。因此，跨越自外而内的第三层循环的迭代的复用数据可以在高速缓存中找到。  $\square$

我们把具有较小且已知界限的最内层循环集合称为最内层分块 (innermost block)。如果有可能，我们期望最内层块包含所有可能带有数据复用的迭代空间的维度。把分块边长最大化并不重要。以矩阵乘法为例，三维分块技术把对每个矩阵的数据访问量降低了  $B^2$  倍。如果存在数据复用，使用高维度小边长的分块要比低维度大边长的分块更好。

我们可以对具有复用关系的循环进行分块，从而优化最内层完全可交换循环嵌套结构的局

部性。我们也可以把分块概念泛化，以利用在较外层并行循环的迭代之间找到的复用。请注意，分块技术主要是交错地执行内层循环的少量实例。在矩阵乘法中，最内层循环的每个实例计算出结果数组的一个元素，总共有  $n^2$  个这样的元素。分块技术把一个块的实例执行交织起来，每次计算每个实例中的  $B$  个迭代。类似地，我们可以把并行循环中的迭代交替执行，以利用它们之间的数据复用。

下面我们定义了两个基本方法，它们可以降低跨越迭代的复用之间的距离。我们从最外层循环开始反复应用这两个基本方法，直到所有的复用都被移动到最内层块的相邻位置上。

#### 在一个并行循环中交织内层循环

考虑一个外层可并行化循环包含一个内层循环的情况。为了利用外层循环迭代之间的数据复用，我们把固定多个内层循环的实例交织在一起执行，如图 11-63 所示。通过创建二维内层分块，这个转换降低了外层循环的连续迭代之间的数据复用之间的距离。

<pre>for (i=0; i&lt;n; i++)     for (j=0; j&lt;n; j++)         &lt;S&gt;</pre>	<pre>for (ii=0; ii&lt;n; ii+=4)     for (j=0; j&lt;n; j++)         for (i=ii; i&lt;min(n, ii+4); i+=4)             &lt;S&gt;</pre>
a) 源程序	b) 转换得到的代码

图 11-63 把内层循环的 4 个实例交织执行

#### 把一个循环

```
for (i=0; i<n; i++)
    <S>
```

变成

```
for (ii=0; ii<n; ii+=4)
    for (i=ii; ii<min(n, ii+4); ii+=4)
        <S>
```

的步骤称为条状挖掘(stripmining)。当图 11-63 中的外层循环的界限较小且已知时，我们不需要对它进行条状挖掘，而是直接交换原程序中的两个循环。

#### 交织执行一个并行循环中语句

考虑一个可并行化循环包含一个语句序列  $s_1, s_2, \dots, s_m$  的情况。如果其中的一些语句本身也是循环，那么连续迭代的语句之间可能还是相隔了很多运算。我们可以使用交织运行这些语句的方法来利用迭代之间的数据复用，如图 11-64 所示。这个转换在不同的语句之间分布那些经过了条状挖掘的循环。如果外层循环只有少量的固定多个迭代，我们不需要对循环进行条状挖掘，而是直接在各个语句上分布原来的循环。

<pre>for (i=0; i&lt;n; i++) {     &lt;S1&gt;     &lt;S2&gt;     ... }</pre>	<pre>for (ii=0; ii&lt;n; ii+=4) {     for (i=ii; i&lt;min(n, ii+4); i++) {         &lt;S1&gt;         for (i=ii; i&lt;min(n, ii+4); i++) {             &lt;S2&gt;             ...         }     } }</pre>
a) 源程序	b) 转换后的代码

图 11-64 交织语句的转换

我们使用  $s_i(j)$  来表示语句  $s_i$  在第  $j$  个迭代中的运行。代码的执行不是按照图 11-65a 中显示的原执行顺序，而是按照图 11-65b 中显示的顺序执行。

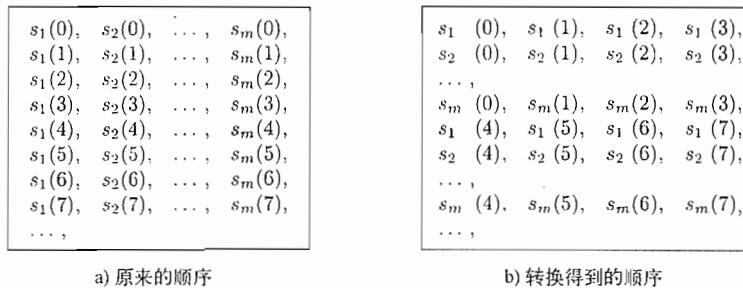


图 11-65 分布一个经过条状挖掘的循环

**例 11.70** 我们现在回到多网格的例子，说明如何利用外层并行循环的迭代之间的数据复用。我们观察到，图 11-62 的代码的最内层循环中的引用  $DW[1,k,j,i]$ 、 $DW[1,k-1,j,i]$  和  $DW[1,k+1,j,i]$  的空间局部性很差。根据 11.5 节中讨论的复用分析可知，下标变量为  $i$  的循环具有空间局部性，而下标为  $k$  的循环具有组复用性。下标为  $k$  的循环已经是最内层循环了，因此我们感兴趣的交织执行来自一个具有连续  $i$  值的分划块中的对  $DW$  的运算。

我们应用这个转换来交织这个循环中的语句，得到图 11-66 中的代码，然后应用这个转换来交织内层循环，得到图 11-67 中的代码。请注意，当我们把来自下标为  $i$  的循环的  $B$  个迭代交错执行时，我们需要把变量  $AP$ 、 $AM$ 、 $T$  扩展为数组，以便一次存放  $B$  个结果。  $\square$

```

for (j = 2, j <= jl, j++) {
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib          = i-ii+1;
            AP[ib]      = ...;
            T           = 1.0/(1.0 +AP[ib]);
            D[2,ib]     = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (i = ii; i <= min(ii+b-1,il); i++) {
            for (k=3, k <= kl-1, k++) {
                ib          = i-ii+1;
                AM          = AP[ib];
                AP[ib]      = ...;
                T           = ...AP[ib]-AM*D[ib,k-1]...
                D[ib,k]     = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...
            }
            ...
            for (i = ii; i <= min(ii+b-1,il); i++)
                for (k=kl-1, k>=2, k--) {
                    DW[i,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
                    /* Ends code to be executed by processor (j,i) */
                }
        }
    }
}

```

图 11-66 对图 11-23 的代码进行分划、数组收缩、内层循环交错和分块后所得的部分代码

```

for (j = 2, j <= jl, j++) {
    for (ii = 2, ii <= il, ii+=b) {
        for (i = ii; i <= min(ii+b-1,il); i++) {
            ib          = i-ii+1;
            AP[ib]      = ...;
            T           = 1.0/(1.0 +AP[ib]);
            D[2,ib]     = T*AP[ib];
            DW[1,2,j,i] = T*DW[1,2,j,i];
        }
        for (k=3, k <= kl-1, k++) {
            for (i = ii; i <= min(ii+b-1,il); i++) {
                ib          = i-ii+1;
                AM          = AP[ib];
                AP[ib]      = ...;
                T           = ...AP[ib]-AM*D[ib,k-1]...
                D[ib,k]     = T*AP;
                DW[1,k,j,i] = T*(DW[1,k,j,i]+DW[1,k-1,j,i])...
            }
            ...
            for (k=kl-1, k>=2, k--) {
                for (i = ii; i <= min(ii+b-1,il); i++)
                    DW[1,k,j,i] = DW[1,k,j,i] +D[iw,k]*DW[1,k+1,j,i];
                /* Ends code to be executed by processor (j,i) */
            }
        }
    }
}

```

图 11-67 对图 11-23 的代码进行分划、数组收缩和分块后所得的部分代码

#### 11.10.4 合成

算法 11.71 对一个单处理器系统的局部性进行了优化，而算法 11.72 则对一个多处理器系统的并行性和局部性进行了优化。

**算法 11.71** 在一个单处理器系统上优化数据局部性。

**输入：**一个带有仿射数组访问的程序。

**输出：**一个最大化数据局部性的等价程序。

**方法：**执行下列步骤：

- 1) 应用算法 11.64 来优化计算结果的时间局部性。
- 2) 应用算法 11.68 在可能的时候收缩数组。
- 3) 利用 11.5 节中描述的技术，确定可能共享相同数据或高速缓存线的迭代子空间。对于每个语句，找出具有数据复用的外层并行循环的维度。
- 4) 对每个带有数据复用的外层并行循环，重复使用基本的交织方法，把一个迭代分块移动到最内层块中。
- 5) 对位于那些带有复用的最内层的完全可交换循环中的维度的子集应用分块技术。
- 6) 对外层完全可交换循环嵌套结构进行分块，其目的是利用内存层次结构中的更高层存储设备，比如第三层高速缓存或物理内存。
- 7) 在必要的地方按照块的边长扩展标量或者数组。

□

**算法 11.72** 针对多处理器系统优化并行性和数据局部性。

**输入：**一个带有仿射数组访问的程序。

**输出：**一个最大化并行性和数据局部性的等价程序。

**方法：**执行下列步骤：

- 1) 使用算法 11.64 对这个程序进行并行化，并创建一个 SPMD 程序。
- 2) 对步骤 1 中生成的 SPMD 程序应用算法 11.71，以优化它的局部性。□

### 11.10.5 11.10 节的练习

练习 11.10.1：对下面的向量运算进行数组收缩变换：

```
for (i=0; i<n; i++) T[i] = A[i] * B[i];
for (i=0; i<n; i++) D[i] = T[i] + C[i];
```

练习 11.10.2：对下面的向量运算进行数组收缩变换：

```
for (i=0; i<n; i++) T[i] = A[i] + B[i];
for (i=0; i<n; i++) S[i] = C[i] + D[i];
for (i=0; i<n; i++) E[i] = T[i] * S[i];
```

练习 11.10.3：以 10 为宽度对下面的外层循环进行条状挖掘：

```
for (i=n-1; i>=0; i--)
  for (j=0; j<n; j++)
```

## 11.11 仿射转换的其他用途

至今为止，我们的注意力都集中在共享内存的计算机体系结构上，但是仿射循环转换的理论还有很多其他的应用。我们可以把仿射转换应用到其他形式的并行性上，包括分布式内存计算机、向量指令、SIMD(Single Instruction Multiple Data，单指令多数据)指令以及多指令发送计算机等。本章中介绍的复用分析技术也可以用于数据预取(prefetching)。数据预取是一个可以提高内存性能的有效技术。

### 11.11.1 分布式内存计算机

在分布式内存计算机中，处理器通过发送消息和其他处理器进行通信。对于这类机器，给各个处理器分配大型的、独立的计算单元显得更加重要。仿射分划算法可以生成这样的单元。除了计算任务的分划，还存在其他一些编译问题需要处理：

1) 数据分配。如果处理器使用的是一个数组的不同部分，每一个处理器只需要分配足够的空间以存放各自使用的部分。我们可以使用投影的方式来决定每个处理器使用数组的那个部分。在决定数据分配时，输入是一个线性不等式系统，该系统表示循环界限，数组访问函数，以及把迭代映射到处理器 ID 的仿射分划。我们通过投影消除循环下标，并找出每个处理器 ID 所使用的数组位置。

2) 通信代码。我们需要明确生成向其他处理器发送以及从其他处理器接收数据的代码。在每个同步点上，

① 确定存放在某个处理器上且其他处理器需要使用的数据。

② 生成具有如下功能的代码：找出所有将被发送的数据并把它们打包放到一个缓冲区中。

③ 类似地，确定这个处理器需要的数据，解开接收到的消息的数据包，把数据移动到适当的内存位置。

如果所有的访问都是仿射的，那么这些任务仍然可以由编译器使用仿射框架来完成。

3) 优化。并不是所有的通信都必须在同步点上进行。比较好的做法是每个处理器在数据可用时立刻发送数据，而每个处理器只有在需要数据时才开始等待。必须对这个优化和另一个目标(即不能生成太多消息)加以权衡，因为处理每个消息的开销都比较大。

这里描述的技术还有其他用途。比如，一个专用的嵌入式系统可能使用协处理器来减轻某些计算负担。嵌入式系统也可能使用一个独立的控制器把数据加载进高速缓存或其他数据缓冲区，或从中卸载，而不是自己要求把数据调入高速缓存；在独立控制器调动数据时，处理

器可同时在其他数据上执行运算。在这些情况下，我们可以使用类似的技术来生成移动数据的代码。

### 11.11.2 多指令发送处理器

我们也可以使用仿射循环转换来优化多指令发送计算机的性能。10.5节讨论过，一个软件流水线化循环的性能受到两个因素的限制：先后关系约束中的环，以及对关键资源的使用。通过改变最内层循环的组成，我们可以改进这些限制。

首先，我们可以使用循环转换来创立最内层的可并行化循环，从而完全消除先后关系约束中的环。假设一个程序有两个循环，其中的外层循环是可并行化的，而内层循环不可并行化。我们可以交换这两个循环，使得内层循环变成可并行化的，从而创造出更多的指令级并行化机会。请注意，我们并不要求最内层循环的迭代之间一定是完全可并行化的。只要其依赖关系所确定的环短到可以充分利用硬件资源就足够了。

我们也可以通过改进一个循环中资源使用的平衡性来放松因资源使用而引起的限制。假设一个循环只使用加法器，而另一个只使用乘法器。假设一个循环因为内存而受到制约，另一个循环因为计算量而受到制约。比较好的做法是把这些例子中的循环对融合到一起，以便同时充分利用所有的功能单元。

### 11.11.3 向量和 SIMD 指令

除了多指令问题之外，还有其他两种重要的指令级并行性：向量和 SIMD 运算。在这两种情况下，发送一个指令可以对一个数据向量的所有元素进行相同运算。

前面提到过，很多早期的超级计算机使用了向量指令。向量运算以流水线化的方式执行，该向量的元素被串行获取，对不同元素的计算相互重叠。在先进的向量计算机中，向量运算可以链接起来：当生成结果向量的元素时，它们立刻被另一个向量指令的运算消耗掉，不需要等待所有的结果都计算完成。不仅如此，在具有散播/收集(scatter/gather)硬件的先进计算机中，向量的元素不要求是连续的，可以用一个下标向量确定这些元素该放在哪里。

SIMD 指令指定了对连续内存位置执行的相同运算。这些指令从内存中并行加载数据，把它们存放在宽寄存器中，并使用并行硬件来计算它们。很多媒体、图形和数字信号处理应用可以利用这些运算。低端媒体处理器只需要一次发射一个 SIMD 指令就可以获得指令级并行性。高端处理器可以把 SIMD 和多指令发射结合起来以获取更好的性能。

SIMD 及向量指令生成和数据局部性优化之间具有很多相似性。当我们找到在连续内存位置上运算的独立分划单元时，就对这些迭代进行条状挖掘，并把最内层循环中的运算交织起来。

生成 SIMD 指令有两个难点。首先，有些机器要求从内存中获取的 SIMD 数据是位对齐的。比如，它们可能要求将 256 字节的 SIMD 运算分量放在为 256 的倍数的地址上。如果源循环只在一个数据数组上运算，我们可以生成一个主循环来处理对齐的数据，而这个循环的前面和后面都有附加的代码来计算边界上的元素。但是对于在多个数组上运算的循环，就有可能无法同时对齐所有的数据。第二，一个循环的连续迭代所使用的数据可能不是连续的。这种例子包括很多重要的数字信号处理的算法，比如 Viterbi 解码器和快速傅里叶变换。要利用 SIMD 指令的话，有可能需要一些额外的用于移动数据的指令。

### 11.11.4 数据预取

没有哪个数据局部性优化方法可以消除所有的内存访问。首先，第一次使用的数据必须从

内存中获取。为了隐藏内存访问的延时，预取指令 (prefetch instruction) 被很多高性能处理器采用。数据预取指令被用来向处理器指明某些数据有可能很快就会被用到，因此如果它现在还没有在高速缓存中，期望能把它加载到高速缓存中。

11.5 节中描述的复用分析可以用于估计什么时候可能发生高速缓存脱靶。当生成预取指令时，有两个重要问题需要考虑。如果将要访问连续的内存位置，我们只需要为每个高速缓存线发出一个预取指令。我们必须足够早地发出预取指令，以保证在使用这个数据时，它已经在高速缓存中了。但是，我们不应该过早地发出预取指令。预取指令可能会把高速缓存中还需要使用的数据转移出高速缓存，而预取到的数据也可能会因此在使用之前就被调出高速缓存了。

#### 例 11.73 考虑下面的代码：

```
for (i=0; ii<3; i++)
    for (j=0; j<100; j++)
        A[i, j] = ...;
```

假设目标机器有一个预取指令。该指令可以一次预取两个字的数据，而一个预取指令的延时大约等于上面的循环中六次迭代的执行时间。图 11-68 中显示了这个例子的使用预取指令的代码。

我们把最内层的循环展开两次，使得可以为每个高速缓存线发出一个预取指令。我们使用软件流水线化概念来保证在数据被使用的六个迭代之前预取数据。流水线的前言部分获取了前 6 个迭代中使用的数据，稳定状态循环在它进行计算的同时提前预取 6 个迭代。尾声部分没有预取指令，只是直接执行余下的迭代。

```
for (i=0; ii<3; i++) {
    for (j=0; j<6; j+=2)
        prefetch(&A[i, j]);
    for (j=0; j<94; j+=2)
        prefetch(&A[i, j+6]);
    A[i, j] = ...;
    A[i, j+1] = ...;
}
for (j=94; j<100; j++)
    A[i, j] = ...;
```

图 11-68 为预取数据而修改的代码



## 11.12 第 11 章总结

- 数组的并行性和局部性。对于并行性和基于局部性的优化而言，最重要的机会来自于访问数组的循环。在这些循环中，对数组元素的各个访问之间的依赖关系通常是有局限的，并且通常按照一个正则的模式访问数组元素。这些因素使程序可以获得很好的数据局部性，高效使用缓存。
- 仿射访问。几乎所有的并行化及数据局部性优化的理论和技术都假设对数组的访问是仿射的：这些数组下标的表达式是循环下标的线性函数。
- 迭代空间：一个具有  $d$  个循环的循环嵌套结构定义了一个  $d$  维的迭代空间。该空间中的点都是值的  $d$  元组，元组中的值对应于该嵌套循环结构运行时各个循环下标的取值。在仿射情况下，各个循环下标的界限是较外层循环下标的线性函数，因此迭代空间是一个多面体。
- Fourier-Motzkin 消除算法*。对迭代空间的关键操作之一是把定义该空间的各个循环重新排列。这么做要求把一个多面体迭代空间投影到它的部分维度上。*Fourier-Motzkin 算法*把一个给定变量的上下界替换成为关于这些界限的不等式。
- 数据依赖与数组访问。在为了并行性和局部性优化的目的而处理循环时，我们需要解决的一个中心问题是确定两个数组访问之间是否具有数据依赖关系（也就是它们是否可能触及同一个数组元素）。如果这些访问以及循环界限都是仿射的，迭代空间就可以被定义为一个多面体。而上面的问题可以被表示为一个特定的矩阵 - 向量方程是否具有位于该

多面体中的解。

- 矩阵的秩和数据复用。用来描述一个数组访问的矩阵可以给出多个关于该数组访问的重要信息。如果该矩阵的秩达到最大值(即矩阵的行数和列数的最小值)，那么当这个循环迭代运行时，数据访问不会两次触及同一个元素。如果数组是按行(列)存放的，那么删除掉最后(最前)一行后得到的矩阵的秩可以告诉我们这个访问是否具有良好的局部性，即单个高速缓存线中的元素被几乎同时访问。
- 数据依赖关系和丢番图方程。如果我们仅仅知道对同一数组的两个访问触及该数组的同一区域，我们并不能判定它们是否真的访问了某个公共元素。原因是每个访问都可能跳过某些元素。比如，一个访问读写偶数号元素，另一个访问读写奇数号元素。为了确定是否存在数据依赖关系，我们必须求一个丢番图方程(只要整数解)的解。
- 解丢番图线性方程。关键技术是计算各个变量的系数的最大公约数(GCD)。只有当这个最大公约数能够整除常量项时，方程才可能存在整数解。
- 空间分划约束。为了并行化一个循环嵌套结构的执行过程，我们需要把这个循环的迭代映射到一个处理器空间。这个处理器空间可能具有一个或多个维度。空间分划约束是说如果不同迭代中的两个访问之间具有数据依赖关系(即它们访问了同一个数据元素)，那么它们必须被映射到同一个处理器上。只要这个从迭代到处理器的映射是仿射的，我们就可以把这个问题用矩阵-向量的方式表示出来。
- 基本代码转换。用来并行化具有仿射数组访问的程序的转换是七个基本转换的组合，它们是：循环融合、循环裂变、重新索引(给循环下标加上一个常量)、比例变换(将循环下标乘以一个常量)、反置(倒转一个循环的下标)、交换(交换循环的顺序)和倾斜(改写循环使得迭代空间中的扫描线不再和某个坐标轴同向)。
- 并行运算的同步。有时，如果我们在一个程序的步骤之间插入同步运算，就可以获得更多的并行性。比如，相邻的两个循环嵌套结构之间可能具有数据依赖关系，但是在这两个循环之间的同步运算可以使得各个循环被单独并行化。
- 流水线化。这个并行化技术允许处理器共享数据，方法是把某些数据(通常是数组元素)从一个处理器同步传递到处理器空间中的相邻的处理器。这个方法可以提高每个处理器所访问数据的局部性。
- 时间分划约束。为了找到流水线化的机会，我们要求出时间分划约束的解。这些约束是说只要两个数组访问会触及同一个数组元素，那么在此流水线中，首先发生的迭代中的访问所分配到的流水线阶段不得晚于第二个访问所分配的流水线阶段。
- 求解时间分划约束。Farkas 引理提供了一个有力的求解技术。它可以找出一个带有数组访问的给定循环嵌套结构所允许的所有仿射时间分划映射。这个技术实质上是把原来的表达时间分划约束的线性不等式公式替换成为它的对偶系统。
- 分块。这个技术把一个循环嵌套结构中的每个循环都分割成为两个循环。这个技术的优点在于可以使得我们在一个多维数组的小段(块)上进行计算，每次处理一个块。这么做提高了程序的局部性，使处理单个块时需要的数据都在高速缓存中。
- 条状挖掘。和分块技术类似，这个技术只把一个循环嵌套结构中的一部分循环分解开，每个循环分成两个循环。这么做的好处是一个多维数组被一条一条地访问，从而得到最好的高速缓存利用率。

### 11.13 第 11 章参考文献

要得到关于多处理器体系结构的详细讨论，读者可参阅 Hennessy 和 Patterson 的教科书[9]。

Lamport[13]和Kuck、Muraoka 及 Chen[10]引入了数据依赖分析的概念。早期的数据依赖分析测试使用启发式规则，通过确定丢番图方程和线性实数不等式系统是否无解来确定一对引用是否独立：[5, 6, 26]。Maydan、Hennessy 和 Lam[18]把数据依赖关系测试写成了整数线性规划的形式，并证明这个问题在实践中可以精确高效地求解。本章描述的数据依赖关系分析基于 Maydan、Hennessy、Lam[18]和 Pugh 及 Wonnacott[23]的工作。这些分析技术使用了 Fourier – Motzkin 消除算法[7]和 Shostak 的算法[25]。

在 20 世纪 70 年代和 80 年代早期已经有人利用循环转换来改进向量化和并行化：循环融合[3]、循环裂变[1]、条状挖掘[17]和循环互换[28]。在当时进行了三个主要的实验性的并行化/向量化项目：在 Illinois Urbana-Champaign 大学由 Kuck 领导的 Parafrase 项目[21]，由 Rice 大学的 Kennedy 领导的 PFC 项目[4]和在 IBM 研究院由 Allen 领导的 PTRAN 项目[2]。

McKellar 和 Coffman[19]最先讨论了使用分块技术来提高数据局部性的理论。Lam、Rothbert 和 Wolf[12]率先在现代体系结构的高速缓存上对分块技术进行了深入的实验分析。Wolf 和 Lam[27]使用线性代数技术来计算循环中的数据复用。Sarkar 和 Gao[24]引入了数组收缩优化技术。

Lamport[13]首先把循环建模为迭代空间，并使用混合规划技术（仿射转换的一个特殊情况）来为多处理器系统寻找并行性。仿射转换的最原始出处是心跳阵列算法的设计[11]。作为直接实现在 VLSI 上的并行算法，心跳阵列要求在并行化的同时最小化通信量。代数技术用于把计算映射到空间和时间坐标上。仿射调度方案的概念以及在仿射转换中使用 Farkas 引理首先由 Feautrier[8]提出。本章描述的仿射转换算法基于 Lim 等人的工作[15, 14, 16]。

Porterfield 提出了第一个预取数据的编译器算法。Mowry、Lam 和 Gupta[20]应用复用分析来使预取数据的开销降到最小，并在整体上提高了性能。

1. Abu-Sufah, W., D. J. Kuck, and D. H. Lawrie, “On the performance enhancement of paging systems through program analysis and transformations,” *IEEE Trans. on Computing* C-30:5 (1981), pp. 341–356.
2. Allen, F. E., M. Burke, P. Charles, R. Cytron, and J. Ferrante, “An overview of the PTRAN analysis system for multiprocessing,” *J. Parallel and Distributed Computing* 5:5 (1988), pp. 617–640.
3. Allen, F. E. and J. Cocke, “A Catalogue of optimizing transformations,” in *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1–30, Prentice-Hall, 1972.
4. Allen, R. and K. Kennedy, “Automatic translation of Fortran programs to vector form,” *ACM Transactions on Programming Languages and Systems* 9:4 (1987), pp. 491–542.
5. Banerjee, U., *Data Dependence in Ordinary Programs*, Master’s thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.
6. Banerjee, U., *Speedup of Ordinary Programs*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1979.

7. Dantzig, G. and B. C. Eaves, "Fourier-Motzkin elimination and its dual," *J. Combinatorial Theory, A*(14) (1973), pp. 288–297.
8. Feautrier, P., "Some efficient solutions to the affine scheduling problem: I. One-dimensional time," *International J. Parallel Programming* 21:5 (1992), pp. 313–348,
9. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.
10. Kuck, D., Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Transactions on Computers* C-21:12 (1972), pp. 1293–1310.
11. Kung, H. T. and C. E. Leiserson, "Systolic arrays (for VLSI)," in Duff, I. S. and G. W. Stewart (eds.), *Sparse Matrix Proceedings* pp. 256–282. Society for Industrial and Applied Mathematics, 1978.
12. Lam, M. S., E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms," *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 63–74.
13. Lamport, L., "The parallel execution of DO loops," *Comm. ACM* 17:2 (1974), pp. 83–93.
14. Lim, A. W., G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," *Proc. 13th International Conference on Supercomputing* (1999), pp. 228–237.
15. Lim, A. W. and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), pp. 201–214.
16. Lim, A. W., S.-W. Liao, and M. S. Lam, "Blocking and array contraction across arbitrarily nested loops using affine partitioning," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001), pp. 103–112.
17. Loveman, D. B., "Program improvement by source-to-source transformation," *J. ACM* 24:1 (1977), pp. 121–145.
18. Maydan, D. E., J. L. Hennessy, and M. S. Lam, "An efficient method for exact dependence analysis," *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 1–14.
19. McKeller, A. C. and E. G. Coffman, "The organization of matrices and matrix operations in a paged multiprogramming environment," *Comm. ACM*, 12:3 (1969), pp. 153–165.
20. Mowry, T. C., M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proc. Fifth International Conference on*

- Architectural Support for Programming Languages and Operating Systems* (1992), pp. 62–73.
21. Padua, D. A. and M. J. Wolfe, “Advanced compiler optimizations for supercomputers,” *Comm. ACM*, **29**:12 (1986), pp. 1184–1201.
  22. Porterfield, A., *Software Methods for Improving Cache Performance on Supercomputer Applications*, Ph.D. Thesis, Department of Computer Science, Rice University, 1989.
  23. Pugh, W. and D. Wonnacott, “Eliminating false positives using the omega test,” *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 140–151.
  24. Sarkar, V. and G. Gao, “Optimization of array accesses by collective loop transformations,” *Proc. 5th International Conference on Supercomputing* (1991), pp. 194–205.
  25. R. Shostak, “Deciding linear inequalities by computing loop residues,” *J. ACM*, **28**:4 (1981), pp. 769–779.
  26. Towle, R. A., *Control and Data Dependence for Program Transformation*, Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1976.
  27. Wolf, M. E. and M. S. Lam, “A data locality optimizing algorithm,” *Proc. SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 30–44.
  28. Wolfe, M. J., *Techniques for Improving the Inherent Parallelism in Programs*, Master’s thesis, Department of Computer Science, University of Illinois Urbana-Champaign, 1978.