8.3 为多线程性能设计数据结构

8.1节中，我们看到了各种划分方法；并且在8.2节，了解了对性能影响的各种因素。如何在设计数据结构的时候，使用这些信息提高多线程代码的性能？这里的问题与第6、7章中的问题不同，之前是关于如何设计能够安全、并发访问的数据结构。在8.2节中，单线程中使用的数据布局就会对性能产生巨大冲击(即使数据并未与其他线程进行共享)。

关键的是，当为多线程性能而设计数据结构的时候，需要考虑*竞争*(contention)，*伪共享*(false sharing)和*数据距离*(data proximity)。这三个因素对于性能都有着重大的影响，并且你通常可以改善的是数据布局，或者将赋予其他线程的数据元素进行修改。首先，让我们来看一个轻松方案：线程间划分数组元素。

8.3.1 为复杂操作划分数组元素

假设你有一些偏数学计算任务，比如，需要将两个很大的矩阵进行相乘。对于矩阵相乘来说，将第一个矩阵中的首行每个元素和第二个矩阵中首列每个元素相乘后，再相加，从而产生新矩阵中左上角的第一个元素。然后，第二行和第一列，产生新矩阵第一列上的第二个结果，第二行和第二列，产生新矩阵中第二列的第一个结果，以此类推。如图8.3所示，高亮展示的就是在新矩阵中第二行-第三列中的元素产生的过程。

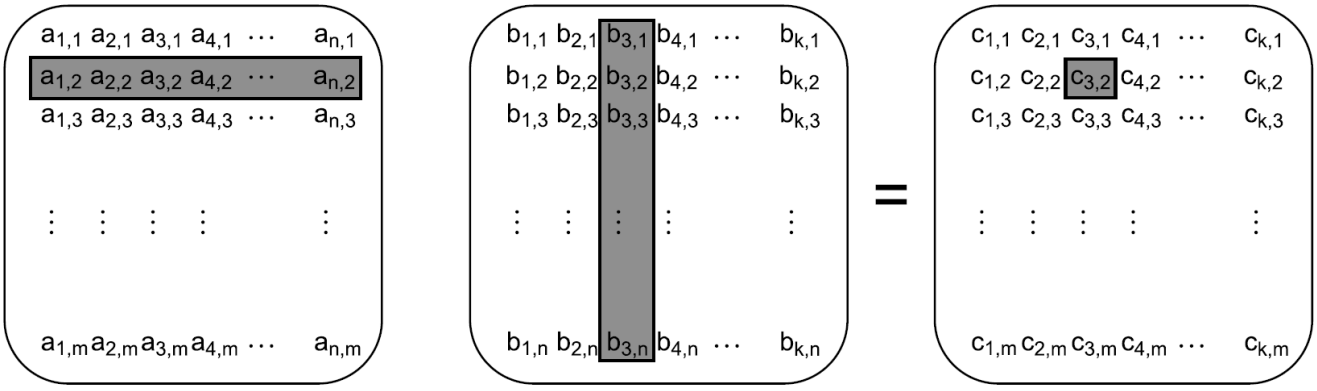

图8.3 矩阵相乘

现在，让我们假设两个矩阵都有上千行和上千列，为了使用多线程来优化矩阵乘法。通常，非稀疏矩阵可以用一个大数组来代表，也就是第二行的元素紧随着第一行的，以此类推。为了完成矩

阵乘法，这里就需要三个大数组。为了优化性能，你需要仔细考虑数据访问的模式，特别是向第三个数组中写入的方式。

线程间划分工作是有很多种方式的。假设矩阵的行或列数量大于处理器的数量，可以让每个线程计算出结果矩阵列上的元素，或是行上的元素，亦或计算一个子矩阵。

回顾一下8.2.3和8.2.4节，对于一个数组来说，访问连续的元素是最好的方式，因为这将会减少缓存的使用，并且降低伪共享的概率。如果要让每个线程处理几行，线程需要读取第一个矩阵中的每一个元素，并且读取第二个矩阵上的相关行上的数据，不过这里只需要对列的值进行写入。给定的两个矩阵是以行连续的方式存储，这就意味着当你访问第一个矩阵的第一行的前N个元素，然后是第二行的前N个元素，以此类推(N是列的数量)。其他线程会访问每行的的其他元素；很明显的，应该访问相邻的列，所以从行上读取的N个元素也是连续的，这将最大程度的降低伪共享的几率。当然，如果空间已经被N个元素所占有，且N个元素也就是每个缓存行上具体的存储元素数量，就会让伪共享的情况消失，因为线程将会对独立缓存行上的数据进行操作。

另一方面，当每个线程处理一组行，就需要读取第二个矩阵上的每一个数据，还要读取第一个矩阵中的相关行上的值，不过这里只需要对行上的值进行写入。因为矩阵是以行连续的方式存储，那么现在可以以N行的方式访问所有的元素。如果再次选择相邻行，这就意味着线程现在只能写入N行，这里就有不能被其他线程所访问的连续内存块。那么让线程对每组列进行处理就是一个改进，因为伪共享只可能有在一个内存块的最后几个元素和下一个元素的开始几个上发生，不过具体的时间还要根据目标架构来决定。

第三个选择——将矩阵分成小矩阵块？这可以看作先对列进行划分，再对行进行划分。因此，划分列的时候，同样有伪共享的问题存在。如果你可以选择内存块所拥有行的数量，就可以有效的避免伪共享；将大矩阵划分为小块，对于读取来说是有好处的：就不再需要读取整个源矩阵了。这里，只需要读取目标矩形里面相关行列的值就可以了。具体的来看，考虑1,000行和1,000列的两个矩阵相乘。就会有1百万个元素。如果有100个处理器，这样就可以每次处理10行的数据，也就是10,000个元素。不过，为了计算着10,000个元素，就需要对第二个矩阵中的全部内容进行访问(1百万个元素)，再加上10,000个相关行(第一个矩阵)上的元素，大概就要访问1,010,000个元素。另外，硬件能处理100x100的数据块(总共10,000个元素)，这就需要对第一个矩阵中的100行进行访问(100x1,000=100,000个元素)，还有第二个矩阵中的100列(另外100,000个)。这才只有200,000个元素，就需要五轮读取才能完成。如果这里读取的元素少一些，缓存缺失的情况就会少一些，对于性能来说就好一些。

因此，将矩阵分成小块或正方形的块，要比使用单线程来处理少量的列好的多。当然，可以根据源矩阵的大小和处理器的数量，在运行时对块的大小进行调整。和之前一样，当性能是很重要的指标，就需要对目标架构上的各项指标进行测量。

如果不做矩阵乘法，该如何对上面提到的方案进行应用呢？同样的原理可以应用于任何情况，这种情况就是有很大的数据块需要在线程间进行划分；仔细观察所有数据访问的各个方面，以及确

定性能问题产生的原因。各种领域中，出现问题的情况都很相似：改变划分方式就能够提高性能，而不需要对基本算法进行任何修改。

OK，我们已经了解了访问数组是如何对性能产生影响的。那么其他类型的数据结构呢？

8.3.2 其他数据结构中的数据访问模式

根本上讲，同样的考虑适用于想要优化数据结构的数据访问模式，就像优化对数组的访问：

- 尝试调整数据在线程间的分布，就能让同一线程中的数据紧密联系在一起。
- 尝试减少线程上所需的数据量。
- 尝试让不同线程访问不同的存储位置，以避免伪共享。

当然，应用于其他数据结构上会比较麻烦。例如，对二叉树划分就要比其他结构困难，有用与没用要取决于树的平衡性，以及需要划分的节点数量。同样，树的的属性决定了其节点会动态的进行分配，并且在不同的地方进行释放。

现在，节点在不同的地方释放倒不是一个严重的问题，不过这就意味着处理器需要在缓存中存储很多东西，这实际上是有好处的。当多线程需要旋转树的时候，就需要对树中的所有节点进行访问，不过当树中的节点只包括指向实际值的指针时，处理器只能从主存中对数据进行加载。如果数据正在被访问线程所修改，这就能避免节点数据，以及树数据结构间的伪共享。

这里就和用一个互斥量来保护数据类似了。假设你有一个简单的类，包含一些数据项和一个用于保护数据的互斥量(在多线程环境下)。如果互斥量和数据项在内存中很接近，对与一个需要获取互斥量的线程来说是很理想的情况；需要的数据可能早已存入处理器的缓存中了，因为在之前为了对互斥量进行修改，已经加载了需要的数据。不过，这还有一个缺点：当其他线程尝试锁住互斥量时(第一个线程还没有是释放)，线程就能对对应的数据项进行访问。互斥锁是当做一个“读-改-写”原子操作实现的，对于相同位置的操作都需要先获取互斥量，如果互斥量已锁，那就会调用系统内核。这种“读-改-写”操作，可能会让数据存储在缓存中，让线程获取的互斥量变得毫无作用。从目前互斥量的发展来看，这并不是个问题;线程不会直到互斥量解锁，才接触互斥量。不过，当互斥量共享同一缓存行时，其中存储的是线程已使用的数据，这时拥有互斥量的线程将会遭受到性能打击，因为其他线程也在尝试锁住互斥量。

一种测试伪共享问题的方法是：对大量的数据块填充数据，让不同线程并发的进行访问。比如，你可以使用：

```
1 struct protected_data
2 {
```

```
  std::mutex m;
  char padding[65536];  // 65536字节已经超过一个缓存行的数量级
  my_data data_to_protect;
};
```

用来测试互斥量竞争或

```
struct my_data
{
  data_item1 d1;
  data_item2 d2;
  char padding[65536];
};
my_data some_array[256];
```

用来测试数组数据中的伪共享。如果这样能够提高性能，你就能知道伪共享在这里的确存在。

当然，在设计并发的时候有更多的数据访问模式需要考虑，现在让我们一起来看一些附加的注意事项。