共享内存的多处理器上的可扩展同步算法

Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

简介

在共享内存的多线程程序中，互斥锁和屏障是两种常用的线程间同步机制。互斥锁是一种对共享资源的互斥访问机制；它支持获取锁（lock）和释放锁（unlock）两种操作，当一个线程获取锁后，另一个线程再试图获取锁时必须等待前面的线程先释放锁；这样可以保证一个共享资源在同一时间最多被一个线程访问。屏障是多阶段并行程序中常用的同步机制；在一个线程组中，如果其中一些线程执行到屏障处，它们必需等待线程组中所有线程都执行到屏障处，才能开始屏障之后的运算；这样可以保证在下一阶段运算开始前上一阶段已经完成。

在多核处理器上，当等待时间不算很长时，我们常常用忙等待来实现这样的同步机制。忙等待有着处理器资源占用的问题，但是比睡眠等待具有更高的性能。

在多核处理器上，同步机制的实现往往要依赖核间缓存一致性的维护，并且可能还要用到一些原子操作。读-修改-写的原子操作如原子加不能并行执行；当多个核同时执行原子加时本质上还是只能一个一个地计算加法。而缓存一致性的维护是更常见的影响可扩展性的因素，如果多个核共享地访问内存中同一个地址且其中一部分访问为写访问，那么核间通信网络就会忙于维护缓存一致性，使得多核性能下降。随着处理器核数的增多，原子操作和维护缓存一致性的开销也越来越大，这对互斥锁和屏障的性能来说，是巨大的挑战。扩展性较好的同步机制通常使用线程局部的变量来进行自旋，且该变量只有一个线程写，一个线程不断地读。

该文分析了已有互斥锁的实现，提出了MCS锁，并测试、比较了各互斥锁算法的在两种处理器上的性能；分析了已有屏障的实现，提出了基于树的屏障，并测试、比较了各屏障算法的在两种处理器上的性能。

可扩展的互斥锁实现

test-and-set锁

用一个布尔值表示锁是否被占用，可以利用原子交换[[1]](#footnote-1)操作来获取锁：即将值为局部变量与全局共享的变量交换值，可以得到交换前的值；若为，则表示成功获取锁，开始执行临界区，否则重新尝试；释放锁时只需要将的值置零即可。

这种性能问题在于，他在自旋的过程中还在不断地写。当为时，必须等待其它线程释放锁，但在等待过程中还在向写入，这种写毫无意义，却使得核间网络必须为此维护缓存一致性，产生巨大的性能开销。为此，一种改进方法是设置两个自旋阶段，第一阶段只读不写，读到了为之后，再进入之前提到的test-and-set自旋。这样的锁称为test-and-test-and-set锁，对核间网络造成的压力没有test-and-set锁那么大，但改进程度有限。另一种改进方法是在自旋中加延迟，使得自旋的频率减慢，从而减轻核间网络的压力。这种方法看似简单，但对于锁的可扩展性有较明显的提升。

门票锁

test-and-set锁还有一个功能上的问题，即它不满足先到先得原则。一个线程尝试获取锁失败后，若在它重新尝试之前锁被释放，则可能被一个新来的线程抢走。这使得在一些并行计算程序里，某些线程可能长时间获取不到锁，从而导致负载不均衡。

门票锁（ticket lock）通过门票机制实现了先到先得，并且在自旋过程中没有写操作。每个线程在尝试获取锁时都对一个整数进行原子加一，容易发现每个等待中的线程都有一个独特的编号，即为这里的门票。除了门票计数器外，门票锁的状态还包括一个整数，表示当前正在执行临界区的线程的门票号。在获取锁时，需要等待直到与自己的门票号相等；在释放锁时，将加一。

门票锁保证了自旋过程只读，避免了多余的缓存无效化。但是在读操作中，所有等待的线程读同一个变量。当被写时，虽然实际上成功获取锁的只有一个线程，但是所有等待的线程都要读取对应的缓存块，由于网络拥堵仍然会遭受较大的延迟。

基于等待队列的锁

扩展性较好的互斥锁通常需要有线程局部的数据结构，典型的数据结构就是队列，队列中一个元素为一个自旋变量，且该变量只有一个线程写，一个线程不断地读。

T.E.Anderson提出了基于循环队列的锁。其状态包括一个的布尔类型的数组和队列尾的下标。在获取锁时，首先将原子加一得到其值，在上自旋，直至其值为；在释放锁时，现将重置为，随后将置1。考虑到最坏情况，数组的长度应该不少于线程数；再考虑溢出的情况，的长度应该是的幂。

G.Graunke和S.Thakkar提出了另一种锁，可以将原子加操作替换为开销更小的原子交换操作。它要求每个线程有自己一个布尔型指示变量用于自旋，并有一个全局共享的指针指示当前的队尾。在获取锁时，通过原子交换更新的同时获取指向自己前驱的指示变量的指针，然后在指向的变量上自旋。释放锁时修改的值。

但是问题在于，如果将的值写为，那么何时将它置来准备下次使用呢？Graunke和Thakkar的解决方案是，释放锁时只将的值翻转，而将的旧值和全局共享的变量一起打包，原子交换整个结构体，得到前驱的指示变量的旧值后就再自旋。这样一来原子操作的位宽就增加了；为了在某些硬件平台上减少额外开销，可以将这些指示变量放在一个数组里，并下标代替指针，减少位宽。

基于数组的实现为了避免假共享，数组的一个元素必须最少占一个缓存块大小（通常64byte）；再加上必须为每个线程分配一个指示变量，一个锁的内存占用会较大。考虑到很多应用场景下会对一些数据结构加细粒度的锁，一些程序里会同时存在大量的互斥锁，这使得内存占用成为一个问题。

该论文作者Mellor Crummey和Michael Scott提出了基于链表的MCS锁，利用了链表可变长的特点。在尝试获取锁之前，线程应自己分配一个链表节点，并在释放锁之后回收节点内存。考虑到它的生存周期是确定的，可以在栈上分配链表节点。和G&T的算法一样，获取锁时通过原子交换得到队尾指针的旧值，若发现队列不为空，随后需要将队尾的指针指向，再等待上的指示变量置。

但是在释放锁的时候，如何知道自己在队列中有没有后继呢？首先应该查看指针，但有可能出现查看完之后指针再被更新的情况。在这里，作者根据原子交换与原子比较-交换在不同硬件平台上开销的不同，设计了依赖原子交换的和依赖原子比较-交换的两种算法，来保证释放锁的正确性。

在支持快速原子比较-交换的平台上，当发现为空时，对队尾指针做原子比较-交换，若发现队尾指针指向时，则交换成功，队尾指针已被置为空；否则等待指针变为非空。若最终指针非空，则将其指向的节点的指示变量置。

在不支持快速原子比较-交换的平台上，当发现为空时，对队尾指针做原子交换，将队尾指针置空，同时取出其旧值。若旧值指向，则说明没有后继；否则需要把取来的、原本在队列中的节点放回队尾指针处。在两次改动队尾指针之间，有可能又有新的节点入队，所有将取来的节点放回时也应用原子交换操作，拿到最新的队尾；若非空，则使插队到的后面。在两次改动队尾指针之间又有新节点入队的概率很小，所有有很大概率不会出现插队，即大多数情况下满足先到先得。

后者由于可能进行多次原子交换，对核间网络造成的压力更大。

可扩展的屏障实现

计数器屏障

计数器屏障是最简单最容易想到的。用一个计数器来表示已到达的线程数，每个线程到达时进行原子加一，若发现自己不是最后一个到达，则在一个指示变量上自旋；若是最后一个，则重置计数器并将更新指示变量。

为了屏障重复使用，更新指示变量的方式为翻转。由于每个线程调用屏障的次数相同，指示变量的旧值不难得到。以下的屏障算法中也都用翻转的方式实现同步。

组合树屏障

计数器屏障的可扩展性瓶颈在于原子加和全局共享的指示变量。为了增强可扩展性，P.C.Yew，N.F.Tzeng和D.H.Lawrie提出组合树屏障。将线程分组，组内用计数器屏障同步，组内最后一个到达者在翻转指示变量之前，还需通过一个组间的屏障。这种结构可以递归地划分从而形成树状结构，每个线程对应一个叶子节点。

锦标赛屏障

组合树仍然有原子加的开销。D.Hensgen，R.Finkel和U.Manber提出，在已经划分好树结构之后，当组内只有很少几个线程（2到4个）的时候，不必让所有线程等待最后一个到达者，而是随意选定一个代表w，所有线程到达时通知w，之后等待w将其唤醒；而w先等待所有其他线程，之后参与上一层同步，然后唤醒其他线程。在锦标赛屏障中称w为小组赛胜者，称参与上一层同步为晋级。

循环屏障

E.D.Brooks提出了蝶形屏障。对于个线程，每个线程进行轮点到点同步，在第轮，号线程与号线程为一对，需要次同步。

D.Hensgen，R.Finkel和U.Manber在此基础上提出循环屏障，将同步次数次。号线程在到达屏障后需要通知，并等待，其中编号为循环编号。

新的基于树的屏障

作者提出了一类基于树的屏障算法。首先，和锦标赛树不同的是，不再只有叶子节点对应一个线程，而是每个树节点都对应一个线程。在到达过程中，每个线程先等待子节点上的线程（如果不存在则跳过，下同），然后通知父节点，并进入唤醒过程；在唤醒过程中，每个线程等待父节点上的线程，然后通知子节点。

注意到这两个过程是完全分离的，可以两个过程用根节点是同一个线程的两棵不同的树，或者可以只有其中一个过程使用树。事实上，实验表明，在某些支持广播的缓存一致性协议中，唤醒过程可以用和计数器屏障同样的中心化唤醒，因为中心化唤醒虽然造成核间网络拥堵，但只有一步，而树状唤醒的步数取决于唤醒树的高度（）。总的开销因硬件平台而异。因此我们有到达树和到达树+唤醒树两个版本的屏障算法。

实验

作者的实验

作者在BBN Butterfly 1和Sequent Symmetry Model B上实验了上述各种算法，得出结论：

1. 在两种多核平台上，MCS锁都是最快的互斥锁；
2. 在BBN Butterfly 1上，循环屏障和到达树+唤醒树屏障是最快的屏障；
3. 在Sequent Symmetry Model B上，计数器屏障和到达树屏障是最快的屏障。

两种硬件平台都不支持原子比较-交换，尽管如此MCS锁还是达到了很高的性能。

对于两种不同的硬件平台，屏障性能差异的主要原因就在于缓存一致性协议上。Sequent Symmetry Model B对于一写多读的情况有较好的优化，一次一写多读快于多次一写一读。

我们的实验

在24核的Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz上重复了该实验，得到同样的结论：

1. MCS锁是最快的互斥锁；
2. 计数器屏障和到达树屏障是最快的屏障。

该平台支持原子比较-交换，缓存一致性协议中也支持广播。

1. 原子地交换一个寄存器与一个内存单元中的值 [↑](#footnote-ref-1)