计算机科学1998 Vol. 25№. 2

共享存储并行处理系统中的自适应同步路障算法*)

Adaptive Synchronization Barrier Algorithm in Shared-Memory Parallel Computing Systems

陈华平　黄刘生　陈国良

(中国科学技术大学计算机系　中国合肥高性能计算中心　合肥230027)

摘　要　The performance of barrier algorithms, which are usually implemented by using spin lock or block, varies with different parallel computing environment. Several adaptive barrier algorithms which combine the spin lock and block are suggested in this paper to avoid the degradation of synchronization performance suffered from varying computing environment.

关键词　Synchronization barrier, Algorithm, Adaptive

一、前言

在并行程序设计中，如何有效地协调各进程(子任务)之间的并行计算是直接影响整个系统执行性能的一个主要因素[1,2]。在许多基于共享存储器模型的并行处理系统中，通过设置同步路障(barrier)来协调并行计算是通常被采用的一种同步实现技术，它保证了在所有进程到达某一同步计算点以前，任何先行到达该同步计算点的进程不能先越过该点往下进行计算。一个应用程序中可设置多个同步路障点来协调各并行计算步(phase)。

目前，可用多种方法来实现同步路障机制，但基本上都是基于采用自旋锁和阻塞这两种方法。从理论上说来，如果由于阻塞引起的调度开销比预计的等待时间要长，或者没有其它进程等待使用处理器资源时，那么采用自旋锁方法效率要高些，否则的话采用阻塞更好一些。但计算环境(这里主要指进程数目和处理器数目)是可变的，这包括两个方面：①不同并行处理系统的处理器数目不同，在处理器较多的系统中较为有效的同步路障算法在处理器较少的系统中执行起来效率可能非常低；反之亦然；②在同一个机器上，一个应用程序在不同计算步(运行阶段)的进程数量不尽相同(由于中途动态创建或撤消一些进程)，所以在某些计算步上执行效率较高的路障算法，在另外一些计算步上执行效率可能较低。为此，我们提出了具有自适应性的同步路障算法，其基本思想是每次执行同步路障算法时，利用前面几次同步路障算法执行的一些启发信息和当前的一些调度信息，把自旋锁与阻塞结合起来决定本次路障算法的执行机制。

二、基本概念

1. 自适应路障算法的基本思想

正如上面所述，我们所执行的自适应路障算法的基本思想是先循环测试一个时间段(SpinLen)，然后阻塞，等待最后一个到达同步路障点进程的唤醒。SpinLen 的长度是可变的，两种极端情况是：①该时间段长度为零，这时路障算法就演变为单纯的阻塞操作。阻塞进程必须等待最后一个到达同步路障点进程的唤醒。②在 SpinLen 时间段内测试到所有进程已到达同步路障点，这时就不需要阻塞，而直接同其它所有进程一起直接进入下一计算步，因而避免了因阻塞引起的进程环境切换所带来的系统开销。但一般情况下，是先执行 SpinLen 长度的测试，然后阻塞。这里最为关键的地方是如何决定 SpinLen 的大小，我们采用的方法是结合前面几次同步路障算法执行的平均阻塞时间，以及当前非阻塞进程数的可用处理机数等一些调度信息来决定 SpinLen 的大小，正是基于这一点，我们称之为"自适应"的。

2. 路障的具体实现方式

*)本文受国家863重点项目(863-306-ZD-07-01)资助。陈华平　副教授，主要从事并行分布处理系统和并行程序设计环境的研究。黄刘生　副教授，现为中国合肥高性能计算中心副主任。陈国良　中国合肥高性能计算中心主任，博士生导师，现主要从事并行分布计算和智能计算的研究。

具体实现路障算法时需要考虑的一个主要问题是，各进程如何了解其它进程的一些同步信息，这些信息主要有“已有多少进程到达本次同步路障点”。最为简单的一种实现方法是采用“集中式”[3]，就是用一个全局共享变量来记录这些信息，然后每个执行路障算法的进程对这个共享变量进行测试和设置。这种方法的优点是实现起来比较简单，但当处理机数目较多的情况下，势必引起系统总线(MIMD-BUS 模型)或处理器/存储器互连网络(MIMD-MIN 模型)通讯的“拥挤”。另一种改进的实现方法是采用“组合树”式[4]，其基本思想是把处理器分组，给每组分配一个树叶(相当于一个共享变量)，每个处理器只对它所在组所对应的树叶状态进行测试和设置。如果发现它已是组中最后一个修改树叶状态的处理器，那么就把该树叶的最终状态结果传给它的父结点。以此类推，最后到达同步点的处理器最后把结果传播给树根结点。这样，可降低“集中式”中对某一共享变量的不断竞争带来的系统开销。另外，还有基于“蝶形网播送”和“二叉树竞争”等技术方式来实现的路障算法，这里不再详细说明，具体可参考文献[5][6]。

3. 自适应路障算法的基本结构

这篇文章中我们主要讨论算法的自适应性，所以为了讨论简单起见，就采用“集中式”来访问共享状态变量。下面是算法用到的一些全局共享变量，及自适应算法的主要框架。

```
shared integer:count=0/*整型共享变量 count 用来记录
      已到达本次路障的进程数,初值为0*/
shared Boolean:g_state=true/*逻辑型共享变量 g_state
      */
private Boolean:l_state=true/*局部于每个进程的逻辑型
      变量 l_state*/
```

算法2.1

```
AdaptiveBarrier()
l_state=not l_state;/*每个进程改变其局部变量的状态,
  同时也是下一个路障点的初值*/
if fetch_and_increment(count)=numb_process-1 then
    initia_next();wake_up();/*设置下一个同步路障算
      法的初值,并唤醒阻塞进程*/
else
  for spin_time=1 to SpinLen do /*在 SpinLen 时间段内
                    对 g_state 重复测试*/
      if g_state=l_state then exit_barrier();/*如所
        有进程到达同步点,则进入下一计算步*/
  block();/*在 SpinLen 期间内没测试成功,有可能其它
    进程等待处理器资源,则阻塞*/
```

上述算法中，我们用 numb_process 来表示该计算步的进程总数，用 count 来记录已到达同步路障点的进程数。fetch_and_increment(count)是一个原语操作，访问共享变量 count 并执行加1操作，若是最后一个访问 count 的进程，那么重新设置下一个路障点的路障算法的共享变量初值。initia_next()主要包括 count=0和 g_state=l_state 这两个操作，这里要注意的是，相邻两个路障的 g_state 初值刚好相反，各进程的 l_state 初值也刚好相反，这一点非常重要，它不但确保所有进程已离开上一个同步路障点，而且确保所有进程必须均到达本次同步路障点。

三、几种不同的自适应路障算法

下面这几个算法的基本思想是一致的，主要区别在于自适应信息的选取方法有所不同，即如何根据路障算法前面几次的执行情况来决定本次路障算法的执行机制。下面算法中出现的变量和语句同上面算法中出现的同名变量和相同语句的含义是一样的，不再解释。

1. m-平均自适应路障算法

该算法主要利用路障算法前面 m 次的平均阻塞时间来决定进程在本次路障算法执行中的 SpinLen 大小。具体应用该算法时，可根据应用程序中使用同步路障的数目来选择 m 的大小，最简单的情况就是 m 为1的情况。

算法3.1

```
mAver_Barrier()
l_state=not l_state
if fetch_and_increment(count)<numb_process-1 then
    for spt=1 to SpinLen do
        if l_state=g_state then blktime=0; goto quit
          _barrier;
    curtime=Get Current Time();block(); blktime=
      GetCurrent Time()-curtime;
quit_barrier;/*下面是调整 SpinLen 的大小*/
    time[id][tcount[id] mod m]=blktime;tcount[id]=
      tcount[id]+1;
    if Average(tcount [id])<switch_time then/*id 为进
      程标识符*/
        SpinLen=min(switch_time, SpinLen+Adjust);
    else
        SpinLen=max(0,SpinLen-Adjust);
else
    initia_next();wake_up();/*设置下一个同步路障算
      法的初值,并唤醒阻塞进程,下同*/
```

上面算法以及下面的几个算法中，switch_time 是阻塞某进程而让另一进程占用处理器引起的环境切换所需要的时间。Adjust 是循环测试时间的调整量，它的取值同具体应用程序有关，一般说来，如果应用程序各计算步之间的关系比较紧密的话，Ajust 可选小一些(微调)，否则，可选大一些。blktime 是该进程在本计算步中的阻塞时间，Average 函数用于计算前面几次的平均阻塞时间。

2. 粗调自适应路障算法

如果每个计算步的并行粒度较大，那么不同路障点的路障算法执行情况的差别相对来说也比较大，这时可把调整 SpinLen 的幅度加大，对有些应用

程序来说，它是非常有效的。

算法3.2

```
CoarAju_Barrier()
l_state=not l_state
if fetch_and_increment(count)<num_process-1 then
    for spt=1 to SpinLen do
        if l_state=g_state then blktime=0; goto quit_
          barrier;
     curtime = GetCurrentTime ( ); block ( ); blktime =
           GetCurrent Time()-curtime;
    quit_barrier;
    if blktime<2 * switch_time then
        SpinLen=switch_time
    else
        SpinLen=0
else
    initia-next();wake-up();
```

该算法只利用了上一次的阻塞时间。实际上一个进程阻塞时，除了把处理器切换给其它进程使用会带来一些开销外，到达本次路障点以后，还需把该阻塞进程唤醒，并且该进程可能马上抢占到处理器，又要把环境切换过来，所以与该进程不阻塞相比，实际时间开销增加了2 * switch_time。如果 blktime 大于2 * switch_time，那么表示由于阻塞该进程而把处理器让给其它进程使用的时间要长些，因而整体执行效率要高一些。所以，这时可把 SpinLen 置为零。

3. 基于调度信息的自适应路障算法

该算法主要根据当前非阻塞进程数与处理器数的情况来决定是阻塞还是循环测试。一个进程到达同步路障点时，如果发现有其它进程在等待使用处理器资源，那么就阻塞让给其它进程使用，否则就执行 spin 操作。该算法的主要优点是不需要决定 SpinLen 的大小，但通过检查来获取这些调度信息会增加一些额外的开销。

算法3.3

```
ScheInfo_barrier(0
l_state=not l_state
if fetch_and_increment(count)<num_process-1 then
     if (numb_procs-num_blocked < numb_processors)
        then/ * numb_processors 为处理器数 * /
         while l_state< > g_state do spin();
    else
        block();
else
    initia_next();wake_up();
```

在一般情况下，假定有 P 个处理器，现在有 N 个进程($N \geqslant P$)欲通过同步路障，如果每个进程的执行工作量差不多的情况下，那么可把这 N 个进程按 P 个一组划分，每一组的进程可同时执行。当组中 P 个进程到达同步点后，它们可以循环测试完剩余时间，或者阻塞让下面 P 个进程执行。如果是这种情况，那么显然最佳方案是让前 N-P 个进程阻塞，这样剩余的 P 个进程能运行。但是最后 P 个进程就没必要阻塞，因为在该计算步内已没其它进程需要使用处理器资源。这样，阻塞所带来的开销只是所有进程到达同步点后，把阻塞进程转换过来。假定 overhead 表示检查非阻塞进程数和空闲处理器数所需要的开销，c 是切换进程环境所需的开销，那么整个同步路障的执行开销为 $Cost_{sched}=\lceil (N-P)/P \rceil \times c+overhead$。

结束语 本文提出了在共享存储器模型的并行处理系统中进行并行程序设计时使用的自适应同步路障算法，这些算法的最大优点是避免了因计算环境变化而引起的并行程序中同步法执行效率的降低。也就是说，使用这些自适应同步算法的并行程序在执行同步机制时具有可伸缩性，因为这些自适应路障算法会随处理器数的变化(在不同并行机上运行)和应用程序的规模(进程数量)的变化而调整算法的具体执行机制，以取得较高的同步执行效率。

参考文献

[1] A. Tucker and A. Gupta, Process control and scheduling issues for multiprogrammed shared-memory multiprocessors, In Proc. of the 12th Symposium on Operating Systems Principles, 1989. ACM

[2] Leonidas Kontothanassis and Robert W. Wisniewski, Using Scheduler Information to Acheive Optimal Barrier Synchronization Performance, 4th ACM PPOPP, 5/1993

[3] Anderson T. E. ,The performance of spin lock alternatives for shared-memory multiprocessors, IEEE Trans. Parallel Distributed Syst. 1, Jan. 1990

[4] John M. et al. , Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Syst, Vol. 9,Feb. 1991

[5] Graunke G. and Thakkar S. , Synchronization Algorithms for Shared-memory Multiprocessors. Computer, 23(6)1990

[6] Sanders B. A. , The information structure of distributed mutual exclusion algorithms, ACM Trans. Comput. Syst. Vol. 5, Aug. 1987