多线程并发访问无锁队列的算法研究

钱立兵　陈波　晏涛　徐云　孟金涛　刘涛

摘　要　FIFO队列是基本的、同时也研究最多的多线程并发数据机构之一。动态的内存分配实现并发无锁队列FIFO算法是高效实用的算法，是有Michael and Scott提出(被称为MS-queue)。由Edya Ladan-Mozes和 Nir Shavit提出的新的并发无锁FIFO队列算法进行研究，该算法是对MS-queue的改进，称为optimistic算法。MS-queue算法中的队列是用单链表表示的，指针的插入操作使用了代价昂贵的compare-and-swap(CAS)操作，Optimistic算法关键思想：使用双链表表示队列，用普通的、开销极低的store操作来插入指针，大大提高了算法的性能，尽管该算法在某些情况下可能会导致不一致性问题，但可以定位问题并改正，采用消隐技术于FIFO数据机构中，并且在保持无锁与线性一致的情况下，能够扩放多线程并发访问无锁队列性能。

关键词　CAS；MS-queue算法；Optimistic算法；并发数据结构；FIFO:无锁；消隐

1　引言

随着多核技术的发展，并行数据结构技术成为一个研究热点，采用有锁访问会带来一定的开销同时可能会产生死锁等，先进先出(FIFO)队列可以构建并发数据结构的基本块，如：Java并发包JSR-166并发队列是一种可线性化、支持enqueue/dequeue操作的FIFO数据结构。本文讨论动态内存分配实现的可线性化的多线程访问的无锁队列的FIFO队列。

Michael and Scott使用动态内存分配实现的并发无锁FIFO队列算法被认为是最高效和实用的算法(被称为MS-queue)。在共享内存的多核处理器上，这种基于Compare-and-swap(CAS)的算法在性能上要远远优于以前基于锁的算法，并且已经被Java并发包所采用。它的主要特点在于允许多线程并发的、无干扰的访问队列的头和尾。

本文从最著名的基于动态内存分配的无锁FIFO队列算法(MS-queue算法)出发，对最为有效的改进算法(我们称为Dominique Fober算法和optimistic算法)采用消隐技术改善多线程并发访问队列的性能。

Dominique Fober算法由于采用双字替换CAS中的单字，使得改进后的算法实际性能获得很大提升。而optimistic算法由于使用简单的store指令替换昂贵的CAS指令从而获得了比MS-queue算法更好的性能。

消隐技术在FIFO队列中的应用，消隐是由shavit and Touitou为在shared pool and counter 中实现可扩放性而收引入的一种技术，Hendler等提出了如何在消隐中引入退避机制，从而在保持线性一致性的条件下实现了LIFO栈的可扩放性。

消隐的工作机制，即让一对相反的操作，如入队出队，互相进行数据交换，而不用与中心数据结构进行同步。直观地，这种技术最适合于LIFO顺序的数据结构，例如一个入栈操作与一个出栈操作正好配对直接交换数据，而无需进入中心的栈结构。

FIFO队列的实现就基于这种技术，一般地，在低负载下，应更倾向于直接访问中心数据结构，因为此时较难找到消隐对；而在高负载下应更倾向于消隐，因为此时较易找到消隐对。利用消隐时，一个直观的方法是对中心队列引入退避机制，针对一个共享无锁队列的实现上使用一个简单的“消隐数组”来支持退避模式。

2　MS-queue算法及改善后的算法

2.1 MS-queue算法

MS-queue算法是1996年由Maged. M. Michael and M. L. Scott提出的，是最为经典的并发FIFO队列上的算法，目前很多对并发FIFO队列的研究都是基于这个算法来加以改进的。

MS-queue算法的队列用一个单链表来实现，包含两个基本的操作，enquene()和dequene()，新节点总是从队尾最后一个元素后面加入队列，节点元素总是从队头删除。包含两个指针，head和tail，head总是自相链表头部的节点，指向的这个

节点被当作是哑节点或哨兵节点，它保存的值是多少并无意义；tail总是指向链表中的一个节点，不一定是队尾元素。每个节点包含两个数据域值信息，即存放的数值信息和指向下一个节点的指针。每个指针对象，除了包含一个指向节点的指针外，还包含一个时间戳，初试时时戳为零，每修改一次指针，时戳增加一，在64位系统中，无需考虑时戳溢出的影响。

2.2 改进的Optimistic算法描述

Optimistic算法对于上面提到的MS-queue算法的改进就在于使用普通的store指令代替代价昂贵的CAS指令。

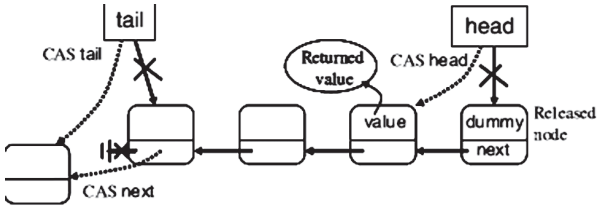

图1 单链表的队列

Figure1描述了用单链表表示的并发操作队列.相对于optimistic算法，MS-queue算法低效的主要原因在于为了成功的完成一次出队(dequeue)操作，需要执行一次成功的CAS操作；为了成功的完成一次入队(enqueue)操作，需要执行两次成功的CAS操作.optimistic算法的出队操作同样需要一次成功的CAS操作，但入队操作却只需要一次成功的CAS操作.表面上看，这种改进似乎并不重要，但是在多线程并发执行时两次成功的CAS操作比一次成功的CAS操作会大大增加CAS操纵失败的次数，况且一次成功的CAS操纵的时间要比普通的load/store操作时间多出一个数量级.因为CAS操作需要独占Cache行并且要刷新处理器的写缓冲区。

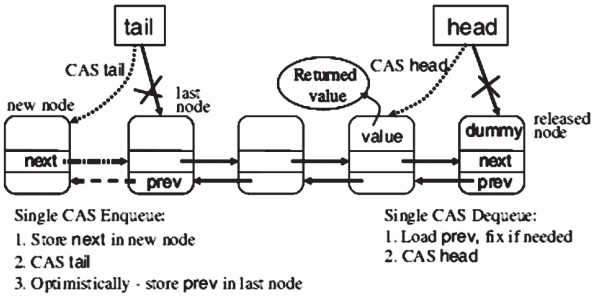

图2 双链表的队列

Figure2描述了optimistic算法的实现方案.它使用双链表表示队列，在进行出队操作时需要使用前向指针prev，但对prev指针的修改却使用普通的store指令完成的。在某些情况下，prev指针可能会不一致，但在必要的时候可以通过执行fixList方法纠正不一致的prev指针.造成这种prev指针的不一致性是由于线程在执行入队操作的过程中遇到长延迟时间，而不是由于线程的争用。当然，对fixList方法的调用频率是很低的。这样一个基于双链表的optimistic算法在修改next和prev指针时使用store指令，入队和出队操作分别只需要一次成功的CAS操作。

2.2.1 算法细节

Optimistic算法的高效性在于使用双向链表表示队列，并且入队和出队操作都只需要一次成功的CAS操作。该算法保证链表总是连接的，next指针总是一致的，当prev指针出现不一致时通过调用fixList方法能够恢复到一致性状态。

同MS-queue算法一样，optimistic算法也用到了原子化的指令Compare-and-swap(CAS)，CAS(a，p，n)，原子化的将内存地址a中的值与p进行比较，如果二者相等，就将n写入地址a中并返回true，否则返回false。由于optimistic算法使用了CAS指令，所以经典的ABA问题同样会出现，解决方案同MS-queue相同，即使用标签机制。

2.2.2 算法的数据结构

共享队列由头指针、尾指针和一些结点组成.每个结点有三个域：value、next指针、prev指针.当一个新结点被创建时，它的next和prev指针被赋空(null)，当队列被初始化时，有一个标记结点，该结点的value域无意义，头指针head和尾指针tail都指向它。

```
struct pointer_t {
  <ptr, tag>: <node_t *, unsigned integer>
};

struct node_t {                          struct queue_t {
  data_type value;                         pointer_t tail;
  pointer_t next;                          pointer_t head;
  pointer_t prev;                        };
};

void init_queue(queue_t* q)
I01: node_t* nd = new_node()      # Allocate a new node
I02: nd->next = <null, 0>         # next points to null with tag 0
I03: q->tail = <nd, 0>;           # tail points to nd with tag 0
I04: q->head = <nd, 0>;           # head points to nd with tag 0
```

图3 算法的数据结构

2.2.3 算法的入队操作

如Figure4代码所示，每次进行入队操作时，首先会创建一个结点并将要入队的元素存入其中；接

着，读取当前的尾指针，将新结点的next指针指向尾结点，将next指针的标记加1；然后调用CAS操作试图修改队列的尾指针，如果成功再将原来队列尾指针指向新的尾结点，如果失败，则重头再来。

从代码中可以看出，optimistic算法的入队操作只需要一次CAS操作，这也是该算法的性能优于MS-queue算法的原因。

```
void enqueue(queue_t* q, data_type val)
E01: pointer_t tail
E02: node_t* nd = new_node()                        # Allocate a new node
E03: nd->value = val                                # Set enqueued value
E04: while(TRUE){                                   # Do till success
E05:   tail = q->tail                               # Read the tail
E06:   nd->next = <tail.ptr, tail.tag+1>            # Set node's next ptr
E07:   if CAS(&(q->tail), tail, <nd, tail.tag+1>){ # Try to CAS the tail
E08:     (tail.ptr)->prev = <nd, tail.tag>          # Success, write prev
E09:     break                                      # Enqueue done!
E10:   }
E11: }
```

图4 入队操作

2.2.4 算法的出队操作

如Figure5代码所示，每次进行出队操作时，首先会读取当前队列的头指针、尾指针和头指针指向结点的pre指针域firstNodePre；接着会判断头指针是否发生变化，如果没变，再判断队列是否为空，如果队列不为空，再接着判断头指针head的标记和firstNodePre指针的标记是否相等，如果相等，则取出firstNodePre指向结点的value域，并调用CAS操作试图修改头指针，如果成功，则释放原来的头指针所指向的结点并返回新指针所指结点的value值。

如果队列为空则返回null，其他失败情况则重新再来。

```
data_type dequeue(queue_t* q)
D01: pointer_t tail, head, firstNodePrev
D02: data_type val
D03: while(TRUE){                              # Try till success or empty
D04:   head = q->head                          # Read the head
D05:   tail = q->tail                          # Read the tail
D06:   firstNodePrev = (head.ptr)->prev        # Read first node prev
D07:   if (head == q->head){                   # Check consistency
D08:     if (tail != head){                    # Queue not empty?
D09:       if (firstNodePrev.tag != head.tag){  # Tags not equal?
D10:         fixList(q, tail, head)            # Call fixList
D11:         continue                          # Re-iterate (D04)
D12:       }
D13:       val = (firstNodePrev.ptr)->value    # Read the value to return
D14:       if CAS(&(q->head), head, <firstNodePrev.ptr,head.tag+1>){# CAS
D15:         free (head.ptr)                   # Free the node at head
D16:         return val                        # Dequeue done!
D17:       }
D18:     }
D19:     else{                                 # Only one node
D20:       return NULL                         # Empty queue, done!
D21:     }
D22:   }
D23: }
```

图5 出队操作

2.2.5 修复prev指针

在进行入队操作时，对结点prev指针的修改是使用普通的store指令完成的。从入队操作的代码中不难看出，如果调用CAS操作修改尾指针成功，就将前一个尾指针所指结点的prev指针指向新加入的尾结点.不幸的是，线程在修改prev指针时会由于某种原因而被延迟。可能这时有一个线程在执行出队操作，而他要删除的结点正好是还没有设置prev指针的结点，这种情况称为prev指针不一致。

为了消除prev指针的不一致，特别定义了一个方法fixList，该方法从尾结点开始顺next链一次修改结点的prev指针和其标记，直到头结点。如果在修改过程中，发现头指针发生了变化说明另外一个线程已经执行完了fixList方法，该线程停止。具体代码如图6所示。

```
F01:void fixList(queue_t* q, pointer_t tail, pointer_t head)
F02: pointer_t curNode , curNodeNext
F03: curNode = tail                                  # Set curNode to tail
F04: while((head == q->head) && (curNode != head)){  # While not at head
F05:   curNodeNext = (curNode.ptr)->next             # Read curNode next
F06:   (curNodeNext.ptr)->prev = <curNode.ptr, curNode.tag-1>; # Fix
F07:   curNode = <curNodeNext.ptr, curNode.tag-1>    # Advance curNode
F08: }
```

图6 修复prev指针

2.2.6 通过标记机制避免ABA问题

当算法中使用了CAS操作时，很有可能会引起ABA问题，通常解决ABA问题的经典方案就是给每个指针加上一个标记，每操作指针一次就改变其标记。在optimistic算法中，对指针标记的修改满足以下三条规则：

- 当对头指针或尾指针执行CAS操作时，它的标记(tag)增加1。
- 当一个新结点加入到队列中时，其next指针的标记设置为当前尾指针的标记加1。
- 同一个结点的next和prev指针的标记相等。

下面解释如何应用上述规则来避免ABA问题。

考虑这样一种情形：假设当前尾指针指向结点A，当一个线程P读取了尾指针所指向的结点后由于某种原因而被挂起，在线程P醒来之前，结点A被其他线程删除了，此外，有线程依次向队列加入了结点B和结点A，这时被挂起的线程P醒来了。如果没有标签机制，线程P无法区分在它被挂起和被唤醒时队列的不同状态，接着会成功的执行CAS操作，导致出错。如果采用了标记机制则线程P在执行CAS操作时会失败而重头再来。

2.2.7 Optimistic算法是可线性化的

`对于入队操作，由于队列是无界的，所以入队操作最终会成功。从入队操作的代码可以看出，当且仅当执行入队操作的线程对尾指针成功的执行了CAS操作时，要插入的结点才算正真入队了。因此成功

的入队操作线性化的时间点是入队线程成功的执行了CAS操作的时刻。

出队操作可能成功也可能失败，当队列为空时，执行出队操作的线程会失败，此时失败的出队操作可线性化的时间点是该线程完成了读取尾指针的时刻。如果队列非空，则出队操作也最终会成功，当且仅当出队线程对头指针成功的执行了CAS操作.因此成功的出队操作可线性化的时间点是执行出队操作的线程成功的完成对头指针的CAS操作。

2.2.8 Optimistic算法是非阻塞(non-blocking)的

从上面的入队和出队操作的代码中不难看出，线程执行入队或出队操作失败的原因是线程在对尾指针或头指针调用CAS操作失败，这意味着队列的头指针或尾指针已经被其它线程更改过，即这些线程成功的执行了入队或出队操作.因此，系统作为一个整体仍然在运行。

2.2.9 Optimistic算法与MS-queue算法的性能比较

图7中，线程交替的执行入队和出队操作.Figure8中线程随机的执行入队和出队操作，但入队和出队的次数相等.图中的“new with pre-backoff”表示线程在执行入队或出队操作失败时会退避一段时间的optimistic算法，“new-no pre-backoff”表示没有使用退避的optimistic算法。

3 采用消隐技术的消隐队列

3.1 算法思想

对一个MS-queue，要做一定的修改，使之可以查询到过去有多少个入队和出队操作已经成功完成，这个信息可以用来决策一个入队操作是否达到一定的时机可以被消隐。

算法的关键是需要在队列负载高的情况下来使用消隐，在这种情况下如果一个入队操作尝试访问中心队列没有成功，它会在重试之前进行一个退避。在这期间，若此入队操作开始时队列中所有的值均已出队，那么可以假设此入队操作已成功的把自己的值加到队列的尾部，并且已达到队列的头，因而可以与一个出队操作进行成对消隐。因此，可以用退避的时间来给未成功的入队操作打上时间标签，以等到时机成熟里可以消隐。关键是达到足够时间，入队操作才可以被消隐。

为进一步理解怎样才是时机成熟，图9出这个执行过程的一个实例。从一个空队列开始，首先1成功入队，然后有一个并发的入队操作，分别是2，3，4. 4成功入队的同时，2，3的入队操作失败了，它们选择退避，此时队列中只有1，4. 这时两个出队操作开始了，第一个成功取出1，而第二个失败了，也选择退避。1出队以后，2已经等到时机成熟了，因为它的前面的所有值已经成功出队，这时可以认为2已经在4的前面队列的首了，这时就可以与出队操作进行消隐了，而无需访问中心队列。

3.2 中心队列的转换

如何确定时机是否成熟到可以执行消隐，这时定义一个抽象的counting queue，它可以支持对时机的检测。一个counting queue提供EnqueueAttempt 和 DequeueAttempt操作，语义上与Enqueue和Dequeue相同，仅仅是多了一个为“fail”的返回值，因为并发操作的相互干涉的结果。它还提供NumDeqs和NumEnqs操作，这两个操作可以返回之前有多少个入队或出队操作已成功。

3.3 算法的数据结构

```
structure node_t  {value: valtype, seq: uint}
structure ptrctr_t  {node: pointer to node_t, ver: uint}
structure Queue_t  {Q: counting_queue_t,
                    Collisions: array of ptrctr_t}
```

图10 算法的数据结构

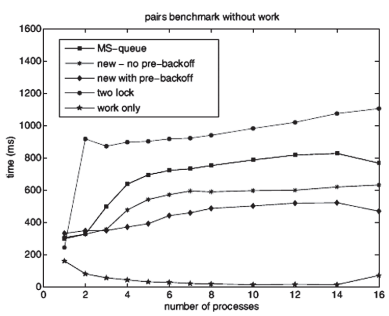

图7 交替的入队与出对

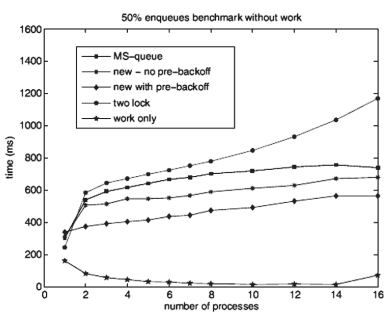

图8 随机的入队与出对

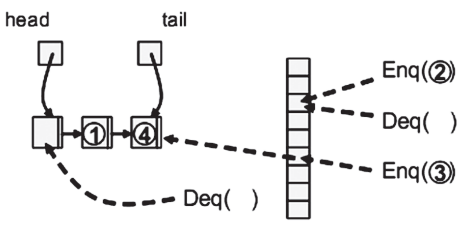

图9 消隐队列，它包括一个MS-queue，和一个消隐数组

由图10知，Node_t 类型包括一个要入队的值和一个序列号，值就是从要消隐的入队操作传递到要消隐的出队操作的那个值，序列号用来用来确定要消隐的入队出队操作队是否可以安全的消隐。FIFO队列Queue_t 由一个计数队列和一个消隐数组组成。node_t类型，带有两个特殊值，”EMPTY”,”DONE”,用来区别于“真”节点。

3.4 算法的实现细节

```
Enqueue(Q: pointer to Queue_t, value: valtype)

 1: uint seen_tail = NumEnqs(Q)
 2: pointer to node_t node = new_node(value)
 3: loop
 4:    if DecideWhetherToAccessQueue() and
                          EnqueueAttempt(Q, node) then
 5:       return
 6:    else
 7:       if TryToEliminateEnqueue(Q, node,
                                        seen_tail) then
 8:          return
 9:       end if
10:    end if
11: end loop

TryToEliminateEnqueue(Q: pointer to Queue_t,
            pointer to node_t, seen_tail: uint):boolean

 1: node→seq = seen_tail;
 2: i = random(collision_array_size)
 3: ⟨colnode,ver⟩ = Q→Collisions[i]
 4: if colnode == EMPTY then
 5:    if CAS(&Q→Collisions[i], ⟨EMPTY,ver⟩,
                                    ⟨node,ver+1⟩) then
 6:       ShortDelay()
 7:       colnode = Q→Collisions[i].node
 8:       if (colnode == DONE) or
            (not CAS(&Q→Collisions[i], ⟨colnode,ver+1⟩,
                                    ⟨EMPTY,ver+1⟩)) then
 9:          Q→Collisions[i] = ⟨EMPTY,ver+1⟩
10:          return true
11:       end if
12:    end if
13: end if
14: return false

Dequeue(Q: pointer to Queue_t,
                    pvalue: pointer to valtype):boolean

 1: loop
 2:    if DecideWhetherToAccessQueue() then
 3:       res = DequeueAttempt(Q, pvalue)
 4:       if res == SUCCESS then
 5:          return true
 6:       else if res == QUEUE_EMPTY then
 7:          return false
 8:       end if
 9:    else
10:       if TryToEliminateDequeue(Q, pvalue) then
11:          return true
12:       end if
13:    end if
14: end loop

TryToEliminateDequeue(Q: pointer to Queue_t,
                    pvalue: pointer to valtype):boolean

 1: seen_head = NumDeqs(Q)
 2: i = random(collision_array_size)
 3: ⟨node,ver⟩ = Q→Collisions[i]
 4: if node ∉ {EMPTY, DONE} then
 5:    if node→seq ≤ seen_head then
 6:       *pvalue = node→value
 7:       if CAS(&Q→Collisions[i],⟨node,ver⟩,
                                    ⟨DONE,ver⟩) then
 8:          return true
 9:       end if
10:    end if
11: end if
12: return false
```

入队操作首先要确定中心队列上有多少个入队操作已成功，从而可以确定一个入队操作是否时机已成熟可以与一个出队操作消隐。一个要入队的值先初始化为一个结点，然后入队操作不断的尝试要么访问中心队列，要么在一个启发示的函数DecideWhetherToAccessQueue的引导下与出队操作进行消隐。

TryTo EliminateEnqueue保存先前在中心队列上的入队操作的个数，然后尝试在消隐数组上找个空位。它随机的找一个空位，判断空位是否包含EMPTY值。如果没有，则消隐失败。否则，线程将尝试用一个CAS，把EMPTY值替换成指向自己结点的指针。如果CAS失败，那么消隐失败。否则，入队操作成功的把自己的结点放入到消隐数组中，等待一个出队操作与之消隐，然后把该结点的指针值置为DONE。

当一个出队操作尝试消隐，它先在消隐数组上随机选一个位置，检查那里是否有一个入队操作等待消隐。如果没有，则消隐失败。否则出队操作尝试把节点指针改为DONE，表明已与那个入队操作成功进行了消隐。此时，只需返回节点的值即可。但是，与一个入队操作的消隐过程并非都是安全的，因此要首先确认在中心队列上的出队操作数至少要达到当前入队操作之前的所有入队操作数。这个过程，只要比较入队操作在中心队列上调用NumDeqs函数的返回结果即可。

3.5 避免ABA问题

为了避免ABA问题，消隐数组中的指针要加以扩展，给它们增加一个版本号，每当一个节点被加入到消隐数组中时，版本号增加。

3.6 算法性能

消隐队列的吞吐量随着并发程度的增加而增加，而MS-queue则不具备这种性质。测试结果如图3几种算法执行的性能对比。

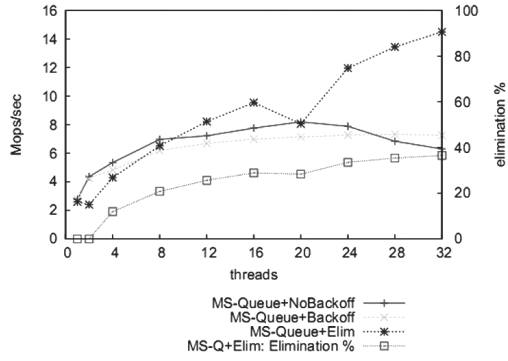

图11 几种算法执行的性能

4 总结

随着多核技术的发展和性能的优化，在算法与数据结构上都要进行改善，如堆栈、队列等数据结构在新的体系结构上要加以改善。文章从无锁机制方面入手进行队列的并发访问，由MS-queue算法到Optimistic算法的改进，采用消隐技术，扩充FIFO性能。也是当前的研究重要的方向。

参考文献

[1] Maged. M. Michael and M. L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” 15th ACM Symp. on Principles of Distributed Computing (PODC), May 1996. pp. 267 – 275.

[2] Dominique Fober, Yann Orlarey, Stéphane Letz. Optimised Lock-Free FIFO Queue.GRAME - Computer Music Research Lab. Technical Report - TR010101,sept. 30 2003.

[3] Edya Ladan-Mozes and Nir Shavit.An optimistic approach to lock-free FIFO queues. Department of Computer Science, Tel-Aviv University, Israel. Distributed Computing, 30 November 2007.

[4] Mark Moir, Daniel Nussbaum, Ori shalev, Nir Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues,SPAA 05,July 18-20,2005,Las Vegas, Nevada,USA.

[5] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 30:645–670, 1997.

[6] B.M. Gittings, S.C. Roche, Parallel polygon line shading: the quest for more computational power from an existing GIS algorithm, IJGIS 10 (6) (1996) 731–746.

[7] A. Clematis, B. Falcidieno, M. Spagnuolo, Parallel processing on heterogeneous networks for GIS applications, IJGIS 10 (6) (1996) 747–767.

[8] D. Xiong, D.F. Marble, Strategies for real-time spatial analysis using massively parallel SIMD computers: an application to urban traffic flow analysis, IJGIS 10 (6) (1996) 769–789.

[9] Stan Openshaw, Developing Automated and Smart Spatial Pattern Exploration Tools for Geographical Information Systems Applications, The Statistician, vol. 44, No. 1, pp. 3-16, 1995.

[10] N. Katayama and S. Satoh, “The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries,” Proc. ACM SIGMOD Int'l Conf. Management of Data, pp. 369-380, 1997.

作者简介

钱立兵 中国科学技术大学计算机软件系统设计硕士 国家高性能计算中心(合肥)实验室 中国科学院深圳先进院客座学生 研究方向为高性能计算与体系结构 。

陈　波 中国科学技术大学计算机专业硕士 国家高性能计算中心(合肥)实验室。

晏　涛 中国科学技术大学计算机专业硕士 国家高性能计算中心(合肥)实验室

徐　云 博士，副教授 国家高性能计算中心(合肥)实验室研究方向并行与分布式计算，随机算法，生物信息学 1999年12月至今，在中国科学技术大学计算机系任教。

孟金涛 华中师范大学计算机专业硕士，现就职于深圳先进技术研究院，主要研究方向为高性能计算，无线传感器网络，高速网络拥塞控制。

刘　涛 西安交通大学计算机专业博士，2008年聘任深圳先进技术研究院助理研究员，主要从事高性能计算及应用的研究工作。