Previous Work

	Type	Progress Property	Conditions	Amortized Time Complexity	Space

Table 1:

Goal

The model consists of p processes. And the problem is to implement linearizable shared queue Q supporting $< ENQ, DEQ >$ operations between them.

If *op* finishes before *op*$'$ starts, it has to take effect on the Q before another. For concurrent operations from processes on Q, the implementation can decide which to put before the other. So we want to design an algorithm that gives processes responses and does their operation on the Q. Since the linearization is a complete ordering among given operations, our problem is to agree on a linearization of the operations.

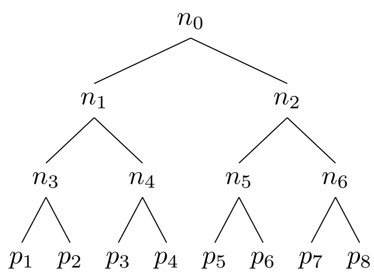


Figure 1: The total ordering of all requests operation propagated up to the root is stored in the root. A request it done if it has reached to the root and has been added to the history. In each node we store which requests has been propagated up to it.

A usual way [?] to agree on something is to use tournament trees. Consider a tournament tree where each process is assigned to one of its leaves. Each process adds an operation to its leaf and tries to propagate it up to the root. The first one that reaches the root is the first operation. Thus the tournament tree is like a competition between processes to determine the total ordering. Propagating an operation from a node means trying to move it one level higher than its parent. Sometimes, two operations from two nodes are propagated up to the parent concurrently; we have to choose one to be before the other arbitrarily.

Now two problems arise:

- Since the tournament tree is shared among all the processes, then at some point, there may be multiple processes trying to propagate their operations at the same node. So we need a lock-free procedure for propagating operations from children of a node to it.
- One way to implement a shared object with the tournament tree is to store in each node the ordering of operations propagated up to that node. Merging of the two children of the root gives us the total ordering at some point. But keeping all these data is not memory-efficient. For example, all dequeues straight after a queue that gets empty can return null without knowing the whole history before.

First Try

In the first attempt, we store each subtree ordering of operation in its root. At each propagation step, we append children's new operations to the parent ordering. The merging step is heavy, and this way is not memory-efficient. Each ordering merge step is $O(p)$, and there will be $\log(p)$ propagate steps, and it will take $O(\#operations)$ to compute the operation response.

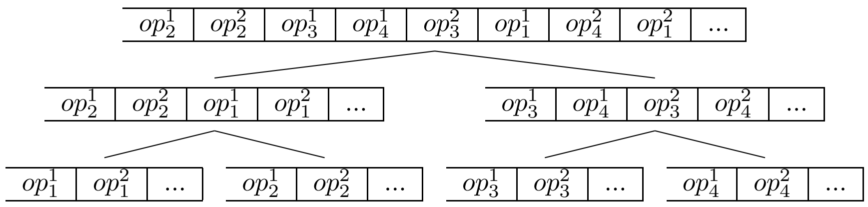


Figure 2: First Try: We show operation i from process j with op_j^i. In each node, we store the ordering of all the operations propagated up to it.

Algorithm 1 First Try Algorithm

```
procedure Do(node n, operation op)
    n.leaf.append(op)
    Propagate(n)
    return Compute(op)
end procedure

procedure Propagate(node n)
    if r≠root then
        MergeChilrenOrderingInto(n.parent)
        Propagate(r.parent)
    end if
end procedure

procedure MergeChilrenOrderingInto(node n)
    new=n.children.new-operations
    n.ordering.append(new)
end procedure
```

Second Try

On the first try, we store the ordering of all the operations in the subtree in its root. But it's not necessary for concurrent operations. If two operations are propagated to a node simultaneously, they will be propagated up to the root together so we can store their ordering in the root. This improvement does not change the time complexity of merge steps.

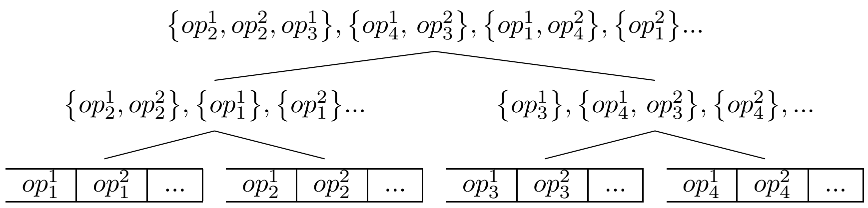


Figure 3: Second Try: In each internal node, we store the set of all the operations propagated together up to it, and one can arbitrarily linearize the total of concurrent operations.

Third Try

We call sets in the ordering from Algorithm 2 "blocks". Here we are proposing that if you store some constant statistics in each set, you can compute the set. If you know how many operations from the left child and right child are in block b, then with knowing the count of operations before block b in node n, we can find out which operation from each child of n has propagated to block b. This leads us to the main algorithm. Using this approach, we can make merge steps $O(\log p)$. As we said before, if we know some data like what is the size of the queue, it helps us to compute results faster. In this way, we propose our algorithm to compute the dequeue operation in $O(\log^2 p)$.

Merge Step

In each merge step on node n, we read n.children new operations and try to append them to the n's ordering. Here we propose a wait-free approach using two CAS operations. First, we create a block of newly added operations to n's children(exist in n.children but not in n). After that, we try to append them to the last block; if it wasn't successful, we try again. If one of the CAS operations was successful, then the block is propagated up to the root; otherwise, it means after the first unsuccessful CAS, another operation has come, and it reads current operations.

`This new algorithm needs proof`

Algorithm 2 Merge Step

procedure MergeChilrenOrderingInto(node n)
    block=CreateBlock(n.left,n.right)
    last=n.last ▷ Index of the first empty cell of n's array of blocks.
    if CAS(n.last, last, last+1) **then** n[last]=block
    else if
        CAS(n.last, last+1, last+2) **then** n[last]=block
    end if
end procedure

Data structure details

Here we are talking about details of the information stored in blocks and the root.

Tree Structure:

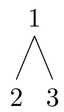
- Leaf l of the tournamnet tree is the list of the operations of process p.
- Interval node n stores an array of blocks($n.blocks$) and index of the first empty cell of the array($n.last$).
- Root like interval nodes stores blocks with two additional data: size(size of the queue after the block), req(#returning deques in each block).

In each block we store:

- Accumulative #left enqs,#right enqs,#left deqs,#right deqs
- pointers to last block merged from left child and righ child

How to find the ith enq among all operations? Find the bock containing ith operation in the root using binary search. Decide the operation is in which child and continue recursively.

`how to draw lists as nodes of the tree and draw edges from cells?`



And these coloumns in the root:

enq	5,6	5,8	6,3	3,7	10,2	1,2	2,3	...
deq	3,11	3,6	5,10	7,9	3,1	2,0	9,8	...
rdeq	11	9	13	10	4	2	14	...
size	0	4	0	0	8	9	0	...

Complete Algorithm with the Logic related to the Queue

Algorithm 3 Main Algorithm

```
function Do(operation op)
    add p to this.ops
    Propagate(this.ops)
                                                ▷ When is op added to the root?
    if op is a deq then
        before-size: size of the block before block containing op
        e: #enqs in the block containing op
        d: #deqs in the block containing op before op
        if before-size + e - d < 1 then
            return null
        else
            d: #rdeqs before the op in all the ordering
            return enq(d+1) #d+1th enq value in all the enqs
        end if
    end if
end function

function Propagate(node n)
    b=Create-Block(n)
    if !tryAppend(b, n) then tryAppend(b, n)
    end if
    Propagate(n.parent)
end function

function Create-Block(n)
        ▷ constructs block of the new operation in children of n. if n is the root, it has extra fields: size, rdeqs.
end function

function tryAppend(b, n)
                                    ▷ tries to append b to the last of the n's list.
end function

function Index(operation op, level ∈ nodes, type ∈ {block, operation})
    ▷ returns index of op in the given level, e.g Index(op, root, block) return ordering of the block containing op
    in the root blocks.
end function

function Access(i, level ∈ nodes, type ∈ {enq, deq})
                        ▷ returns i-th operation of given type in the given node subtree.
end function

function Prefix-Sum(i, level ∈ nodes, type∈ {enq, deq, rdeq})
    ▷ computes how many of the given type operations are before the ith operation in the given level. For rdeq it
    will only get root level.
end function
```

Algorithm 4 Block Tree

▶ leaf l_i: list of operations
▶ internal node n of BT: list of blocks and index *last*
▶ block b: 4 statistics of concurrent operations aggregated together to a block in a REFRESH consisting [$left$:#left ops, $right$: #right ops, $left - sum$: prefix sum left ops, $right - sum$: prefix sum right ops]
▶ index *last*: index of last block of node n

```
1: function VOID(Append)operation op, pid i
2:     l_i.append(op)
3:     PROPAGATE(parent of l_i)
4: end functionAppend

5: function VOID(Propagate)node n
6:     if n==root then return
7:     else
8:         new=CREATEBLOCK(n)
9:         if CAS(n.last, now, now+1) then last block of n=now
10:        else if CAS(n.last, now, now+1) then last block of n=now
11:        end if
12:    end if
13:    PROPAGATE(parent of n)
14: end functionPropagate

15: function BOOLEAN(CreateBlock)node n
16:     current= last block of n
17:     start= BSEARCH(crrent.left-sum)
18:     left=0
19:     for each block do start: first null block of left child of n
20:         left+=block.left
21:     end for
22:     left − sum=current.left-sum+left
23:     do lines 14 to 19 for right
24:     return [left,right,left-sum,right-sum]
25: end functionCreateBlock

26: function ELEMENT(GetIndex)block b, index i
27:     n:=b.node
28:     if i≤b.left then
29:         sb=BSEARCH(left child of n, i-b.right.sum)
30:         GETINDEX(sb, b.left-sum)
31:     else sb= BSEARCH(right child of n, i-b.left.sum)
32:         GETINDEX(sb, b.right-sum)
33:     end if
34: end functionGetIndex

35: function LIST(GetElements)block b
36:     for each block in GETSUBBLOCKS(b) do
37:         result.append(GETELEMENTS)
38:     end for
39:     return result
40: end functionGetElements

41: function LIST(GetSubBlocks)b
42:     n:=b.node
43:     b[-1]=b's previous block
44:     for each direction {left, right} do
45:         init=BSEARCH(n.direction, b[-1].direction)
46:         end=BSEARCH(n.direction, b.direction)
47:         result.append([init:end])
48:     end for
49:     return result
50: end functionGetSubBlocks
```