1 Feilds

2 Queue

3 Search+Append

4 Propagate

5 Index+Get

6 MaxByProcess

7 Help

8 FreeMemory

9 PBRT

Algorithm Tree Fields Description

◇ *Shared*

- A binary tree of Nodes with one leaf for each process. root is the root node.
- *MaxbyProcess* lastDequeuedFrom Index of the most recent block in the root that has been deqeueued from.

◇ *Local*

- *Node* leaf: process's leaf in the tree.
- *int* garbageCollectRound

▶ *Node*

- **Node* left, right, parent : Initialized when creating the tree.
- *PBRT* blocks : Initially blocks[0] contains an empty block with all fields equal to 0.
- *int* head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.

▶ *Block*

- *int* super : approximate index of the superblock, read from parent.head when appending the block to the node

▶ *InternalBlock* extends *Block*

- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: indices of the last subblock of the block in the left and right child
- *int* $\texttt{sum}_{\texttt{enq-left}}$: #enqueues in left.blocks[1..$\texttt{end}_{\texttt{left}}$]
- *int* $\texttt{sum}_{\texttt{deq-left}}$: #dequeues in left.blocks[1..$\texttt{end}_{\texttt{left}}$]
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #enqueues in right.blocks[1..$\texttt{end}_{\texttt{right}}$]
- *int* $\texttt{sum}_{\texttt{deq-right}}$: #dequeues in right.blocks[1..$\texttt{end}_{\texttt{right}}$]

▶ *LeafBlock* extends *Block*

- *Object* element : Each block in a leaf represents a single operation. If the operation is enqueue(x) then element=x, otherwise element=null.
- *int* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: # enqueue, dequeue operations in this block and its previous blocks in the leaf
- *object* response

▶ *RootBlock* extends *InternalBlock*

- *int* size : size of the queue after performing all operations in this block and its previous blocks in the root

Algorithm *Queue*

```
void Enqueue(Object e)                          ▷ Creates a block with element e and adds it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq
    leaf.Append(newBlock)
end Enqueue

▷ Creates a block with null value element, appends it to the tree and returns its response.
Object Dequeue()
    block newBlock= new(LeafBlock)
    newBlock.element= null
    newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
    leaf.Append(newBlock)
    <b, i>= IndexDequeue(leaf.head, 1)
    output= FindResponse(b, i)
    return output
end Dequeue

▷ Returns the response to D_i(root, b), the ith Dequeue in root.blocks[b].
element FindResponse(int b, int i)
    if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then     ▷ Check if the queue is empty.
        lastDequeudFrom.update(b)
        return null
    else                                        ▷ The response is E_e(root), the eth Enqueue in the root.
        e= i + (root.blocks[b-1].sum_enq-root.blocks[b-1].size)
        <x, y>= root.DoublingSearch(e, b)
        lastDequeudFrom.update(x)
        return root.GetEnqueue(x,y)
    end if
end FindResponse
```

Algorithm *Node*

⇝ Precondition: `blocks[start..end]` contains a block with $\texttt{sum}_{\texttt{enq}}$ greater than or equal to `x`

▷ Update needed: search on RBT.

```
int BinarySearch(int x, int start, int end)
    while start<end do
        int mid= floor((start+end)/2)
        if blocks[mid].sum_enq<x then
            start= mid+1
        else
            end= mid
        end if
    end while
    return start
end BinarySearch
```

Algorithm *Root*

⇝ Precondition: $\texttt{root.blocks[end].sum}_{\texttt{enq}} \geq \texttt{e}$

▷ Returns `<b,i>` such that $E_{\texttt{e}}(\texttt{root}) = E_{\texttt{i}}(\texttt{root}, \texttt{b})$, i.e., the `e`th `Enqueue` in the `root` is the `i`th `Enqueue` within ▷ the `b`th block in the `root`.

```
<int, int> DoublingSearch(int e, int end)
    start= end-1
    while root.blocks[start].sum_enq>=e do
        start= max(start-(end-start), 0)
    end while
    b= root.BinarySearch(e, start, end)
    i= e- root.blocks[b-1].sum_enq
    return <b,i>
end DoublingSearch
```

Algorithm *Leaf*

```
void Append(block B)                              ▷ Only called by the owner of the leaf.
    if root.head%p^2<p and garbageCollectRound<floor(root.head/p^2) then
        Help()
        root.FreeMemory()(lastDequeuedFrom.Get()-1)
        garbageCollectRound=floor(root.head/p^2)
    end if
    blocks[head]= B
    head= head+1
    parent.Propagate()
end Append
```

Algorithm *Node*

```
    ▷ n.Propagate propagates operations in this.children up to this when it terminates.
void Propagate()
    if not Refresh() then
        Refresh()
    end if
    if this is not root then
        parent.Propagate()
    end if
end Propagate

    ▷ Creates a block containing new operations of this.children, and then tries to append it to this.
boolean Refresh()
    h= head
    for each dir in {left, right} do
        h_dir= dir.head
        if dir.blocks[h_dir]!=null then
            dir.Advance(h_dir)
        end if
    end for
    new= CreateBlock(h)
    if new.num==0 then return true
    end if
    result= blocks[h].CAS(null, new)
    this.Advance(h)
    return  result
end Refresh
```

Algorithm *Node*

```
void Advance(int h)                                   ▷ Sets blocks[h].super and increments head from h to h+1.
    h_p= parent.head
    blocks[h].super.CAS(null, h_p)
    head.CAS(h, h+1)
end Advance


Block CreateBlock(int i)                              ▷ Creates and returns the block to be installed in blocks[i].
    block new= new(InternalBlock)
    for each dir in {left, right} do
        index_prev= blocks[i-1].end_dir
        new.end_dir= dir.head-1                       ▷ new contains dir.blocks[blocks[i-1].end_dir..dir.head-1].
        block_prev= dir.blocks[index_prev]
        block_last= dir.blocks[new.end_dir]
        new.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        new.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        new.type= InternalBlock-->RootBlock
        new.size= max(root.blocks[i-1].size  + new.num_enq- new.num_deq, 0)
    end if
    return new
end CreateBlock
```

Algorithm *Node*

```
⇝ Precondition: blocks[b].num_enq ≥ i ≥ 1
element GetEnqueue(int b, int i)                      ▷ Returns the element of E_i(this, b).
    if this is leaf then
        return blocks[b].element
    else if i <= blocks[b].num_enq-left then          ▷ E_i(this, b) is in the left child of this node.
        subblockIndex= left.BinarySearch(i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1,
                       blocks[b].end_left)
        return left.GetEnqueue(subblockIndex, i)
    else
        i= i-blocks[b].num_enq-left
        subblockIndex= right.BinarySearch(i+blocks[b-1].sum_enq-right, blocks[b-1].end_right+1,
                       blocks[b].end_right)
        return right.GetEnqueue(subblockIndex, i)
    end if
end GetEnqueue

⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_deq ≥ i.
<int, int> IndexDequeue(int b, int i)    ▷ Update needed: return null when superblock in the root was not found.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n ? left: right)
        superblockIndex= parent.blocks[blocks[b].super].sum_deq-dir > blocks[b].sum_deq ?
                         blocks[b].super: blocks[b].super+1
        if dir is left then
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-left
        else
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-right
            i+= parent.blocks[superblockIndex].num_deq-left
        end if
        return this.parent.IndexDequeue(superblockIndex, i)
    end if
end IndexDequeue
```

Algorithm *MaxByProcess*

```
int[p] lastDequeuedbyProcess

int Get
    return max(lastDequeuedbyProcess)
end Get

Update(int b)
    if lastDequeuedbyProcess[pid]<b then
        lastDequeuedbyProcess[pid]=b
    end if
end Update
```

Algorithm *Tree*

```
int Help
    for each process P
    h=P.leaf.head
    if P.leaf.blocks[h].num_deq==1 and P.leaf.IndexDequeue(h,1)!=null then
        <b, i>= IndexDequeue(h, 1)
        output= FindResponse(b, i)
        P.leaf.blocks[h].response= output
    end if
    end for
end Help
```

Algorithm *Node*

```
FreeMemory(int b)
    if not leaf then
        left.FreeMemory(blocks[i].end_left)
        right.FreeMemory(blocks[i].end_right)
    end if
    while !blocks.CAS(blocks.splitGreater(i))
end FreeMemory
```

Algorithm *PBRT*

```
PRBT prbt
nodes store <key, sum_enq-> block
[i] -> GetByBlock(i)
GetByBlock(int i)
    return rbt.get(i)
    if not found then
        return written response
    end if
end FreeMemory
```

10 Description

In our original algorithm an `Enqueue` or a `Dequeue` remains in the `blocks` array in the tree nodes even after they terminate. This makes the space used by the algorithm factor of the number of operations of invoked on the queue. In this section here we want to free the memory allocated by the operations that are no longer needed and make the space used polynomial of $p+q$.

One way of handling garbage collection is to reallocate the space taken by each each operation right after it is not needed anymore, but it might cost too much to do that. So we garbage collect by batching. If we do the garbage collection every p^2 block appended to the root, its cost is amortized over p^2 blocks which is $O(p^3)$ and $\Omega(p^2)$.

In our design a process attempts to do `Garbage Collect` every p^2 block appended to the root. `Garbage Collect` corresponded to the kp^2 root block is called the kth round. To know if it is kth round or nor we can use an if condition on the value of `root.head`. But the problem is that a process might miss a round not because it is on idle but because it reads `root.head` not to be kp^2. We handle that by checking with a window.

Lemma 1. *The number of the blocks in the* `blocks` *is* $O(p^2+q)$.

Lemma 2. *When* $\frac{\texttt{root.head}}{p^2}$ *incremnts from* r *to* $r+1$*, at least one operation has executed Lines 48,49.*

An `Enqueue` operation cannot be remove after it is terminated since later it might be dequeues, however it can be remove from the tree after it has been computed to be the response of a `Dequeue`. A `Dequeue` operation can be removed from the tree safely after it has computed its response `Enqueue`. Note that it is not needed for the `Dequeue` to be terminated but only it is sufficient if it has computed its response. We say a block in the `root` is *finished* if all of its operations can be removed.

Lemma 3. *There are at most $p^2 + q$ not finished blocks in the* `root.blocks` *at a time.*

The good property in a fifo queue is that if ith `Enqueue` gets dequeued then it means 1 to $i-1$th enqueue operation have also dequeued. This gives us the idea that if a block is finished all the blocks before it are also finished. However if an operation in a block goes to sleep for a long time then the block remains not finished. Since there are at most p idle operations we can help them and then remove all the finished blocks safely.

Lemma 4. *If all current operations are helped, then there is a block in the root that all of its previous blocks are finished.*

The idea above leads us to a poly-log data structure that supports throwing away all the values smaller than an index. Red-black trees do this for us. We can create a shared red-black tree just creaing a new path for the operation and then using `CAS` to change the root of the tree. See [this] for more.

Observation 5. *PBRT supports....*

Lemma 6. *If we replace the arrays we used to implement* `blocks` *with red-black trees the amortized complexity of the algorithm would be $PolyLog(p, q)$. And also the algorithm is correct.*

We can help a `Dequeue` by computing its response and writing it down. If the process in future failed to execute, it can read the helped value written down.

Lemma 7. *The* `response` *written is correct.*

But how can we know which blocks in each node are finished or not? We can keep track of the last finished block.

Lemma 8. *If all current operations are helped, then the blocks before the newest block that some* `Enqueue` *has been dequeued from is safe to remove. If the most current* `Deuque` *returned* `null` *then all the blocks before the block containing the* `Dequeue` *can be removed.*

There is a shared array among processes which they write the last block dequeud from in it.

Lemma 9. `lastDequeuedFrom`*- index of the last finished block in the root is $O(p)$.*

Lemma 10. *After* `FreeMemory`*, the space taken by each node is $O(p^2 + q)$*