Intoducing the Best Shared Queue Ever

February 14, 2022

1 Description of the algorithm

1.1 `Queue`

1.1.1 Definition

`Queue` is an ordered set of elements.

1.1.2 Methods

`Enqueue(element v)` Appends `v` after the last appended `element` to the queue.

`Dequeue()` Removes the first not dequeued `element` in the queue and returns it.

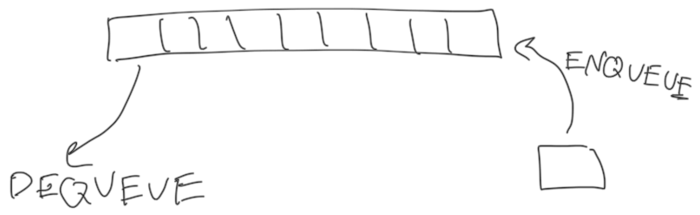

Figure 1: `Queue`

1.1.3 Implementation

We use a `Block tree` and a `Queue helper` implement a `Queue`. To do an enqueue, it is sufficient to append an enqueue operation to the block tree. For dequeues, we do the same but at last, compute the head using `Queue helper`, and if the queue is not empty, we return the corresponding enqueue argument.

1.1.4 Complexity

`Enqueue(v)` and `Dequeue()` each take O(log Q+log^{2}p) steps.

1.2 `Block tree`

1.2.1 Definition

`Block tree` is a data structure that linearizes invocations of `operation`s by some procesess.

Linearization Operations are linearized when they are added to the root. The operations in a root block are linearized as the block's ordering. `TODO`problem: where should we define the linearization ordering?

1.2.2 Methods

`Append(operation op)` Appends op to the linearization.

`Get(int i)` Returns ith linearized operation.

`Index(operation op)` If Append(op) is terminated, Index(op) returns the order of op in the linearization.

`Mark(int i)` If operation i is marked Get(i) is not going to be invoked in future anymore.

1.2.3 Implementation

`Block tree` is a binary tree of `node`s, such that a leaf is assigned to each process. Process `p` adds `op` to its leaf and propagates it to the `root node`, which stores the total ordering.

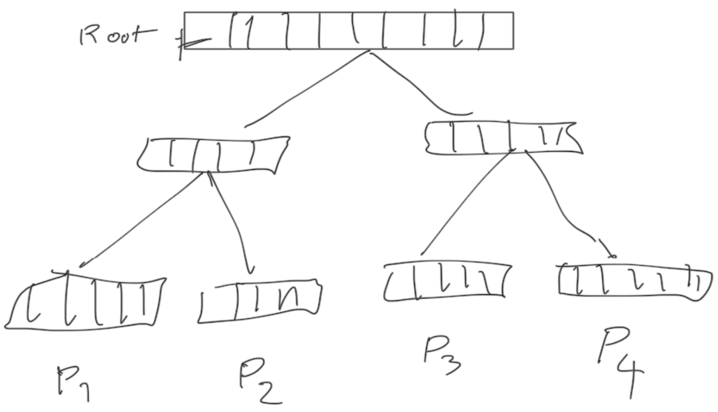

Figure 2: `Block tree`

`Append(op)` Adds a block for op to the invoking process's leaf.

`Propagate(node n)` When `Propagate(node n)` is terminated, operations in n before calling `Propagate(node n)` are in the root.

1. Refresh(n)
2. If 1 was unsuccessful, do Refresh(n) one more time.
3. Propagate(n.parent)

`Get(i)`

1. B= find the root block of containing ith op
2. Get(B, i-pre(B), root)

`pre(B)` is the number of operations before block B in B's node.

`Index(op)`

1. B= the block in a leaf containing operation op
2. Index(n,B,i+pre(B))

`Mark(i)` Mark i in the root node.

1.2.4 Complexity

`Append(op)`, `Get(i)` and `Index(op)` each take O(log^{2}p), O(log (n-#marked)+log^{2}p), O(log^{2}p) steps.

1.3 `Queue helper`

1.3.1 Definition

Queue helper tells us the head of the queue at the time some dequeue operation has been linearized.

1.3.2 Methods

`ComputeHead(i)` Computes head of the queue when ith dequeue occurs.

`Augment(i)` Augments root block i.

1.3.3 Implementation

By adding the size and number of non-null dequeues in a block, we can compute the head in constant time.

1.3.4 Complexity

Constant time.

1.4 Node

1.4.1 Definition

A node contains the ordering of the blocks appended to it by time.

1.4.2 Methods

`Append(Block b)` Adds `block b` to the end of the node's ordering.

`GetHead()` Returns the index of the last block added.

`Refresh(Node n)` Creates a block from `n`'s children's blocks which are not already in `n`, and tries to append it to `n`.

`Get(B,i,n)` Returns the ith operation in block B in node n.

`Index(n,B,i)` Returns the superblock in n.parent containing block B in node n.

1.4.3 Implementation

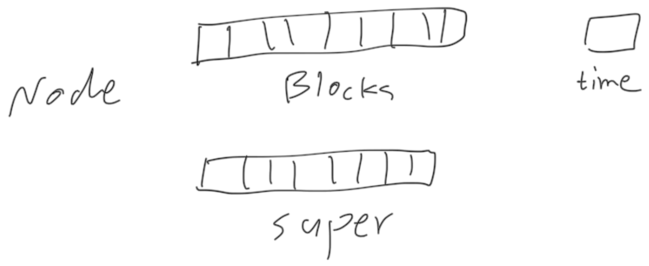

Figure 3: `Node`

`blocks[i]` stores the ith block. `time` shows the number of groups propagated to the parent. `super[i]` is the index of the superblock of the group i.

`Append(Block b)` `CAS`es b to the first empty cell in A.

`GetHead()` Returns the head field, it is updated every successful Refresh().

`Get(B,i,n)`

1. SB= find the subblock of B containing ith op
2. Get(SB, i-pre(SB), B's node)

`Index(n,B,i)`

1. SB= find the block in n's parent that contains ith op in block B of node n
2. Index(n,B,i+pre(SB))

`Refresh()`

1. Create a block from n's children's blocks which are not already in n
2. Append the block to n
3. If 2 was successful, update the head field

1.4.4 Requirements

If Append() fails it means another successful instance of Append exists that has some time in common.

1.4.5 Complexity

Each level of Get() takes log p steps, and there are log p levels. So Get may take O(log^2 p) steps. Complexity of finding the superblock of a block is O(log p).

1.5 `Root node`

A data structure that contains an ordering of not-marked blocks appended to it.

`Append(Block b)` Adds `block b` to the end of the `Root node`'s ordering.

`Mark(Block b)` Marks block b.

`Get(i)` Returns the block containing ith operation.

1.6 `Block`

1.6.1 Definition

A block is an ordering of some operations in a node. We can merge some blocks into a block, the initial blocks are called subblocks, and the latter is called superblock.

`TODO` problem: where should we put end pointer of blocks that help get become faster?

1.6.2 Methods

`CreateBlock(operation op)` Returns a `block` that contains `op`.

`CreateBlock(node n, int start, int last)` Returns a `block` that contains the blocks from n.start to n.last. The order of the input blocks remains the same.

`GetEnd()` Returns the last left and right subblock of the block.

1.6.3 Implementation

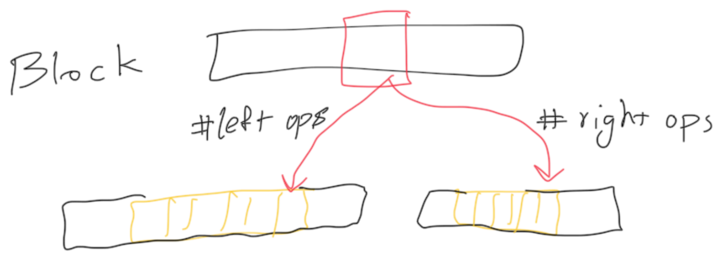

Figure 4: `Block`

We can do the told before methods on blocks only by knowing

1. #enqueue operation from left
2. #enqueue operation from right
3. #dequeue operation from left
4. #dequeue operation from right
5. index of the last block from left
6. index of the last block from left
7. group number which tells the blocks which have propagated to the parent together

`TODO` problem: how to explain the separation of the enqueues from the dequeues? till now, we considered all operations from just one type

1.7 `Operation`

1.7.1 Implementation

Each operation contains the arguments of the operation, the invoking process, and the position in the invoking's process operations.