Design & Notation

A tournament tree in which each process is assigned to one of its leaves. Each node of the tree stores a string of form " $< operation, process >^*$ " and has a regex parser for reading its words. Each process has a local copy of the history of the given sequential object and a local copy o.

We show set subtraction with notation "-" in the algorithm. And "." for concatenation on two strings. In the rest of the paper, when we say node n we mean *n.value*. Unless we explicitly call tree related attributes like *n.children* or *n.parent*.

Algorithm 1

```
Shared Objects
    tree T
Local Objects
    Sequential Object o
    List snapshot

function Do(operation op)
    l= p's assigned leaf in tree
    l.< op, p >
    Propagate(l)
    history= Read(root)
    changeset= histroy-snapshot
    snapshot= history
    for all < op′, p′ >: changeset do
        if p′=p then
            res=o.op′
        else
            o.op′
        end if
    end for
    return res
end function

function Propagate(node n)
    if n==root then return
    else if !Refresh(n.parent) then
        Refresh(n.parent)
    end if
    Propagate(n.parent)
end function

function Refresh(node n)
    old= Read(n)
    for all child: n.children do
        posted.(child-n)
            ▷ ops that child contains but n doesn't
    end for
    new+= old.posted
    res=CAS_n(old, new)
    return res
                ▷ true: Success, false: Failure
end function
```

Correctness

We're going to show our design satisfies correctness and progress. Do(op) finishes in finite steps, so our design is wait-free. It remains to show Algorithm 1 is linearizable.

Lemma 1. *If op is in node n and then a process calls a successful* Refresh(n.parent) *or 2 consecutive* Refresh(n.parent)*s, op will be in n.parent after those procedures terminates.*

Proof. In case of one successful Refresh(), the CAS_n has added *op* to *n.parent*. Consider 2 consecutive failed $refresh_1$, $refresh_2$ on node n. For each failed REFRESH() r there is at least a REFRESH() w which has succeeded to do CAS with the same old vale read as r. From Lemma 1 let the successful REFRESH()es corresponding to $refresh_1$ and $refresh_2$ be $refresh_3$ and $refresh_4$ respectively. If $refresh_3$ has read *op* then *op* will be in *n.parent* after $refresh_3$. Otherwise $refresh_4$'s READ cannot be before CAS of $refresh_3$. And since the CAS of $refresh_3$ is after concatenating *op* to n then $refresh_4$ has read *op* and stores it in *n.parent*. □

Lemma 2. *Consider an instance of* PROPAGATE(n). *When it terminates all operations that were in* n *before* PROPAGATE(n) *will be in the root.*

Proof. We prove this by induction on depth of node n. From Lemma 1, all operations in n will be in *n.parent* after one recursive call of PROPAGATE(n). By induction hypothesis to PROPAGATE(n.parent) which is then called recursively, PROPAGATE(n) satisfies the claim. □

Theorem 3. *Algorithm 1 is linearizable.*

Proof. Lemma 2 shows us that if PROPAGATE procedure of a DO(op) is terminated then *op* will be in root. let the linearization point of a finished DO(op) be when *op* is concatenated to the root by some PROPAGATE procedure. If several operations are propagated at the same time then they are linearized in the order of adding to root. Since every process runs operation on o in order of the *history* stored in the root, then results are consistent with linearization order.

If do_1 is finished before do_2 begins, then do_1 is linearized before do_2. From Lemma 2 we know that after line 4 of Algorithm 1 do_1 operations are propagated to the root. So do_1 is linearized before do_2. □

Analysis & Conclusion

Each DO(op) has a PROPAGATE which uses at most $2\log(p)$ CAS operations. The history is repeated by each process locally. So after n DO(op) operations among p processes there will be $\Omega(\log p + np)$ steps per DO(op). However the number of shared memory accesses is $\Omega(\log p)$. Since CAS operations work with large words the algorithm is not practical, but it shows that without restricting size of shared words the lower bound on universal construction shared memory accesses is lower than $\Omega(\log p)$.