safe Register

Regular 1on = 1 : Atomic Register

sequential spec

N of values expected at each iteration,

on i, i is expected to contain 3 value

for i = 0 to i < V

if (AIi = 1) { break }

od

Return (i)

Atomic SW SR MV

if (lo < i) then

else if (hi < i) then

else


Atomic Snapshots

update(value)

scan

wait free

update(value)

SS = snapshot - scan
write(Reg: (TS, V, SS))

a = collect
b = collect
if a = b then return a
else return a

TS value

R1

R2

SS

SS

value

value
snapshot - scan

\[
\text{Moved} := \emptyset
\]

Do forever

\[
\bar{a} := \text{collect}
\]
\[
\bar{b} := \text{collect}
\]

if (\bar{a} = \bar{b}) \triangleright \text{return } \bar{a} \}

else

\[
\exists k \in \bar{a}, \bar{b} \neq b[k]
\]

if (k in moved)

\[
\text{return } b[k].\text{snapshot}
\]

else

\[
\text{moved} := \text{moved} \cup \{k\}
\]

end forever

Renaming

\[
1, \ldots, n \overset{\text{new}}{\mapsto} e', 1', \ldots, 2n-1 \quad e'
\]
$2n-2 \text{ or more for } 2n \text{ or more for } 3c$  

\[ \text{Reg} \text{ Li } j = 1 \]

\[ \text{Rooms} [i] = \text{null} \]

\[ \text{Reg} \text{ Li } j = 1 \]

\[ \text{Rooms} [i] = \text{snapshot scan} \]

if ($\forall k \neq i \text{ (Rooms} [k] \neq \text{null})$)

\[ \text{Return } \text{Li} 3 \]

else

\[ \text{Rooms} [i] = \text{null} \]

\[ \text{Reg} \text{ Li } j = 1 \]

\[ \text{Rooms} [i] = \text{snapshot} \]

Do forever

\[ \text{Rooms} [i] = \text{snapshot scan} \]

if ($\forall k \neq i \text{ (Rooms} [k] \neq \text{null})$)

\[ \text{Return } \text{Reg} \text{ Li } j \]

else

Let $r$ be my rank in active processes

\[ \text{Reg} \text{ Li } j = n^{th} \text{ free room in Rooms} \]

End forever
(1) Agreement (Wait-Free) in shared memory

- Asynchronous

* Wait-Free: Every processor does not wait for other processors.
Agreement in processors, with 1 fault

Read/Write memory (1-resilient) only:

(n-1 resilient) if we can wait-free

1-resilient reads are atomic, but

write is not equivalent. (why?)}

(why is the read atomic and
write not?)
Q: $P \& Q$ evidence

\begin{align*}
\text{loop} & \quad \text{if evidence of $P$ \& evidence of $Q$} \\
& \quad \text{then} \\
& \quad \text{loop if $P \& Q$ occur} \\
& \quad \text{end loop} \quad \text{end loop} \\
& \quad \text{end loop} \quad \text{end loop} \\
& \quad \text{end loop}
\end{align*}

}\text{Read} \quad 5 \quad \text{Read}

write (prop) \\
read ($s_1 = \text{prop, prop}$) \\
write ($sp$)

(Read)
read a file then read a korn model ok. 3

...
Immediate Snapshot

\[
\begin{align*}
\text{snapshot} & \quad \text{ski} \quad \text{write} \quad \text{read} \quad \text{test} \\
\text{sk, snapshot} & \quad \text{ox, test} \\
\text{sk, snapshot} & \quad \text{ox, test} \\
\end{align*}
\]

\[\text{sk, snapshot} \rightarrow \text{ox, test}\]
Do k = n down to 1
level[i] = k
S = snapshot [level[1] ... n]
if ( |e_j| ≤ k |S| ≥ k ) return

end Do
Process P.

previ = R
write(v + valuer(n) to R)
return(previ)

Consensus (my -input)

if t = swap (R, i) then return(IN[i])
write( IN[i] = IN[i-1]

else return(IN[i])

if t = swap (R, i) then return(IN[i])
write( IN[i] = IN[i-1]

else return(IN[i])

write(v + valuer(n) to R)
return(previ)
CAS (R, old-v, new-v)

t: = R
if t= old-value
then R: = new-value;
return true;
else
Return false;

Universal Construction.

5: c obj out. CAS active mjce, common part
almost like NIDS, sequential specification to arcs E =
sequential specification of CAS similar to this.
sequential specification of CAS similar to this.
sequential specification of CAS similar to this.
stack, swap, Fetch&add, Test&Set, 2-consensus

N: push null onto, 2-consensus

Read-Write Regs

-2k-2 renaming

-2k-2 O(1) -- 2k-2 O(1)

(set-consensus)

N: pop null from, 2-consensus

Write/Read/Write/Read

Immediate snapshot

3 to 2

1838
lıklar סלקו proprietor על יד וידי 2 (3
oka על אייקון, נקודה 8/ב, קוח ל-(11)

View: \((1,1), (1,2), (1,1), (2,1), (2,1)\)

3 × ניתן לשהות בקטים של נזק
לשם שיר חדש, ומה שהופך
בנושאים של עולי-ה לעמוד
וביצועי ה-

teens - read only write 8

(snapshot) read ski write
3.3.1 Advanced Topic:

Algorithm 15.3 A non-blocking universal algorithm using a compare-and-swap
object, code for processor \( p_i \), \( 0 \leq i \leq n - 1 \).
Initially \( Head \) points to the anchor record

when inv occurs:  // operation invocation, including parameters
1: allocate a new opr record pointed to by point with point.inv := inv
2: repeat
3: h := Head
4: point.new-state, point.response := apply(inv,h,new-state)
5: point.before := h
6: until compare\&swap Head, h, point = h
7: enable the output indicated by point.response  // operation response

Algorithm 15.4 A non-blocking universal algorithm using consensus objects:

code for processor \( p_i \), \( 0 \leq i \leq n - 1 \).
Initially \( Head[j] \) points to anchor, for all \( 0 \leq j \leq n - 1 \)

when inv occurs:  // operation invocation, including parameters
1: allocate a new opr record pointed to by point with point.inv := inv
2: for \( j := 0 \) to \( n - 1 \) do  // find record with highest sequence number
3: if \( Head[j].seq > Head[i].seq \) then \( Head[i] := Head[j] \)
4: repeat
5: win := decide(Head[i].after,point)  // try to thread your record
6: win.seq := Head[i].seq + 1
7: win.new-state, win.response := apply(win.inv,Head[i].new-state)
8: Head[i] := win  // point to the record at the head of the list
9: until win = point
10: enable the output indicated by point.response  // operation response

Algorithm 15.5 A wait-free universal algorithm using consensus objects:

code for processor \( p_i \), \( 0 \leq i \leq n - 1 \).
Initially \( Head[j] \) and \( Announce[j] \) point to the anchor record,
for all \( 0 \leq j \leq n - 1 \)

when inv occurs:  // operation invocation, with parameters
1: allocate a new opr record pointed to by \( Announce[i] \)

with \( Announce[i].inv := inv \) and \( Announce[i].seq := 0 \)
2: for \( j := 0 \) to \( n - 1 \) do  // find highest sequence number
3: if \( Head[j].seq > Head[i].seq \) then \( Head[i] := Head[j] \)
4: while \( Announce[i].seq = 0 \) do
5: priority := Head[i].seq + 1 mod n  // id of processor with priority
6: if \( Announce[priority].seq = 0 \)  // check if help is needed
7: then point := Announce[priority]  // choose other record
8: else point := Announce[i]  // choose your own record
9: win := decide(Head[i].after,point)  // try to thread chosen record
10: win.seq := Head[i].seq + 1
11: win.new-state, win.response := apply(win.inv,Head[i].new-state)
12: Head[i] := win  // point to the record at the head of the list
13: enable the output indicated by win.response  // operation response

simulation using consensus

nation are relatively simple.
no new state or the response,
times). A processor detects
threaded onto the linked
the sequence number of the
process p:

\[ A[1] := \text{input}; \]

\[ \text{Do } s := \text{snapshot}(A) \]  
\[ \text{until } |s| \geq n-t \]  
\[ \text{Decide (minimum (s))} \]

\[ t \geq 0 \text{ and } t \text{-resilient} \]

\[ w/A \text{ p o p o p 3 \ e } \text{ p o p o p s p s p s} \]

\[ q(p9) \]

\[ p \rightarrow \text{p o p o p s p s p s} \]
Adaptive Renaming \((f(k), \sigma(n), n)\)

Also Adaptive Renaming, where for each

\[ f(k) = 2k - 2 \]

\[ f(k) = 2k - 1 \]

Symmetry Breaking

Look and check if

No risk symmetry breaking

Also, consider the following

\(a_0, a_1, a_2, \ldots\)

2m - 2 when

also renaming 2n - 2 in \(\sigma(n)\)
Leader Election

1. Choose $b$ with probability $\frac{1}{n}$
   0 otherwise

```
b := coin
send b clockwise
upon receiving (S)
if |S| = n
    then I am only '1'
else
    send S + b clockwise
```

$$n \cdot \frac{1}{n} \cdot (1 - \frac{1}{n})^{n-1} \Rightarrow \frac{1}{e}$$

By now we have shown that the probability of selecting a leader in
leader election is $\frac{1}{e}$.
Random Walk

counter = 0

for x in range(100):
    if random.random() < 0.5:
        counter += 1
    else:
        counter -= 1

print(counter)
E = Agreement + read-write + 1/5.0

- [min(input), max(input)]
- [0, 5]
- [0, 1]
- 
- termination = wait-free

- 0.2 x 0.2 x 0.2

- 0.1

- 0.2

- 0.1

- 0.2

- 0.1
Adaptive Algorithms

1. Initially, X, Y belong to Win or are in Win registers
2. if \( Y = 1 \) then move right \( Y \)
3. \( Y \) initially 0
4. go to 1

Wait Tree - I have it anymore

- If splitter continues, then copy from left to right.
- Splitter continues.
Algorithm 14.3 The consensus algorithm:

Code for processor $p_i$ with input $x_i$, $0 \leq i \leq n - 1$.

Initially $r = 1$ and $preff = x$ // prefers is initially the input

1: while true do
2: $r := \text{send}((\text{vote}, \text{prefer}, r), \text{reliable})$ // phase $r$
3: wait for phase $r$ (vote) messages from at least $n - f$ processors
4: let $v$ be the majority of phase $r$ vote values received
5: if all phase $r$ vote values received are $v$ then $y := v$ // decide $v$
6: $r := \text{send}((\text{outcome}, v, r), \text{reliable})$
7: wait for phase $r$ (outcome) messages from at least $n - f$ processors
8: if all phase $r$ outcome values received are $w$ then $\text{prefer} := w$
9: else $\text{prefer} := \text{common-coin}(t)$
10: $r := r + 1$

Shared data:
- counter $a_0$ with range $[0, n]$, initialized to 0
- counter $a_1$ with range $[0, n]$, initialized to 0
- counter $c$ with range $[-n, n]$, initialized to 0

procedure $\text{consensus}$(input)

increment($a_i$)
repeat
read($a_0, a_1, c$)
if $c \leq -2n$ then decide 0
elseif $c \geq 2n$ then decide 1
elseif $c \leq -(a_0 + a_1)$ or $a_1 = 0$ then decrement($c$)
elseif $c \geq (a_0 + a_1)$ or $a_0 = 0$ then increment($c$)
else
if coin() = 0 then decrement($c$)
else increment($c$)
fi
fi
end
end

Algorithm 17.2 Asynchronous round $r \geq 1$ of wait-free $\varepsilon$-approximate agreement algorithm for known input range. Code for processor $p_i$, $0 \leq i \leq n - 1$.

Initially $y = x$ and $\text{maxRound} = \left\lceil \log_2 \frac{n}{\varepsilon} \right\rceil + 1$, where $D$ is the maximal spread of inputs

1: $\text{update}(\text{ASO}, n)$
2: $\text{values}(r) := \text{scan}(\text{ASO})$
3: $v := \text{midpoint}($values$(r))$
4: if $r = \text{maxRound}$ then $y := v$ and terminate // decide

Below, we present two wait-free algorithms for approximate agreement; a simple algorithm that depends on knowing the range of possible input values, and an adaptive algorithm that does not require this knowledge.