

# **Proposals for a Component Architecture and Non-Blocking Synchronization**

**Kenneth Chiu  
Indiana University**

# Software Components

- Deployable by end user
- Relatively large
- Possibly long-lived
- Distributed

# Examples

- Enterprise Java Beans
- CORBA
- DCOM
- AVS
- Common Component Architecture (CCA)  
DOE effort to provide a standard, high-performance component standard.
- Common Component Architecture Toolkit (CCAT)  
IU implementation of a CCA-compliant architecture

# CCA

- Services object.

Functions as the CCA “ORB”.

- Components communicate through ports.

- *Provides-ports*

- Essentially an interface provided by a component.

- *Uses-ports*

- Essentially a bindable, named, remote reference.

# CCAT

- Globus-based Java and C++ implementation of CCA.
- Components instantiated and connected through a GUI.
- Linear System Analyzer (LSA)  
Allows users to connect and run various linear system components to reorder, scale, etc.
- Portals Notebook Project  
Allows scientists to access and connect applications through the web.

# Two Views of Components

- As a rapid application development methodology.  
The end user is completely unaware of the component nature of the application.
- As a problem-solving environment.  
The end-user directly manipulates components on a daily basis.

These two views are orthogonal. A monolithic implementation could present a component-based UI, while a monolithic UI could be implemented using components.

# Verdant Component Architecture

- Separation of ports from connections

Ports serve only as rendezvous points. Multiple connections to a single port result in multiple communication end points.

- 2-way connections

A single interface can specify the complete relationship between two peers.

- Remote invocation channels

Every remote call also opens a channel attached to the remote invocation. Can be used for streaming.

# Ports and Connections

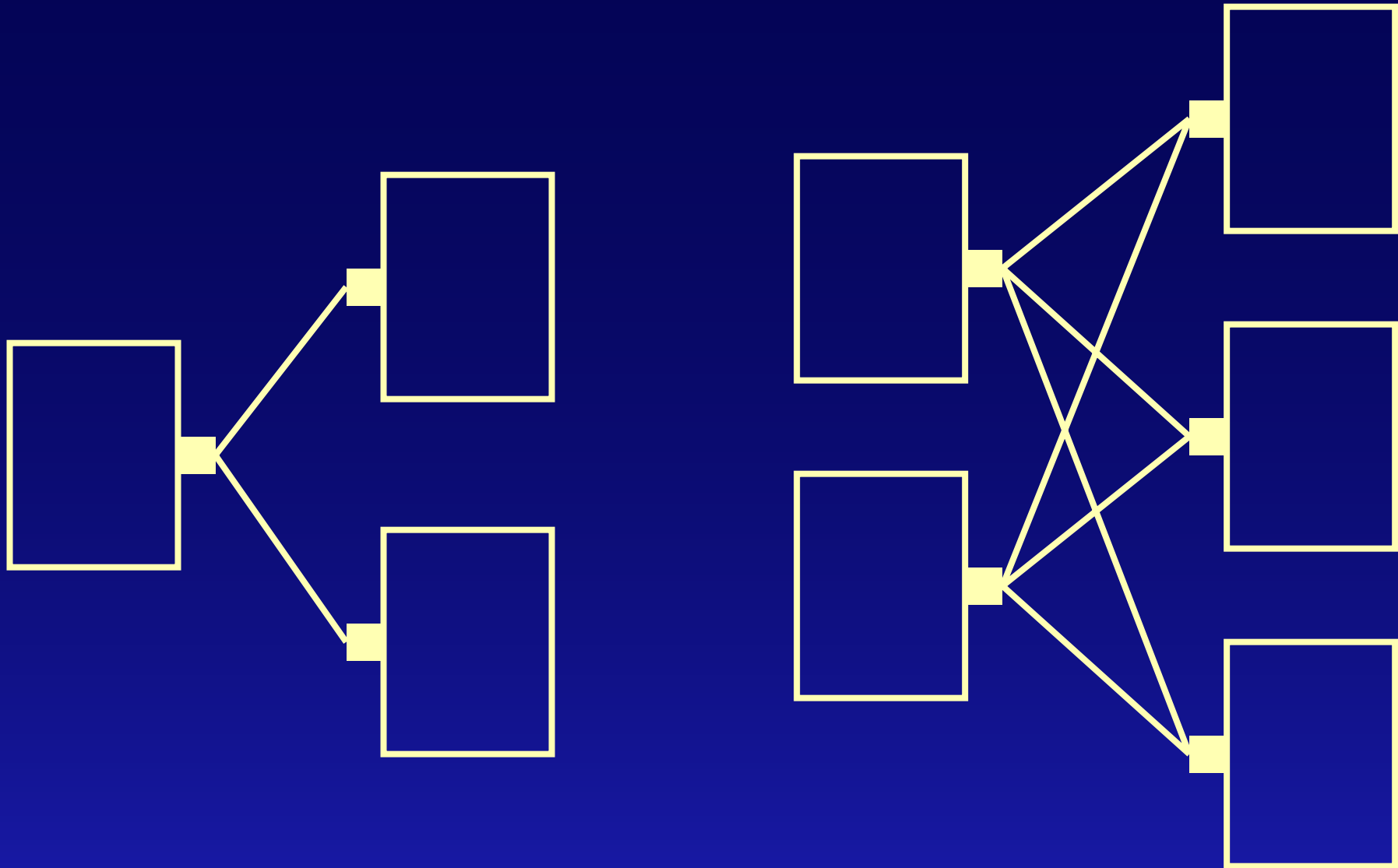
Component ports are intended as an analog to hardware ports.

Simulation

Visualizer

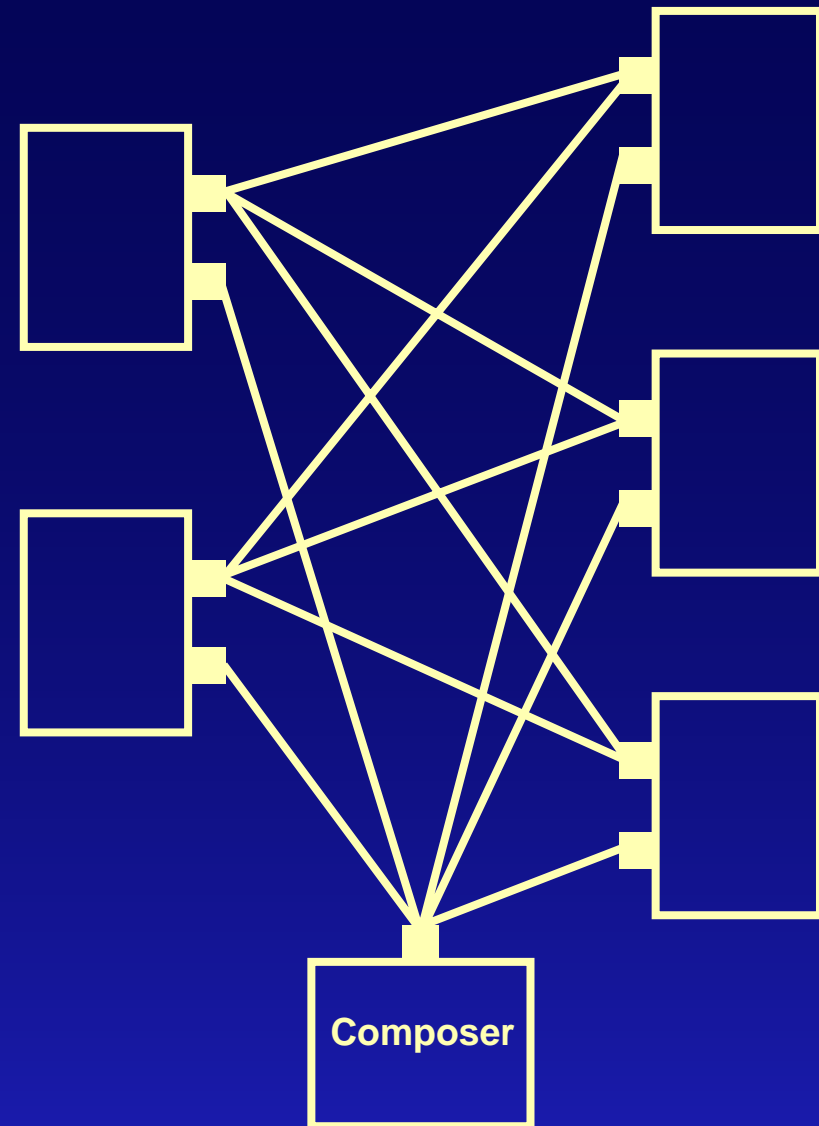


# Multiple Connections

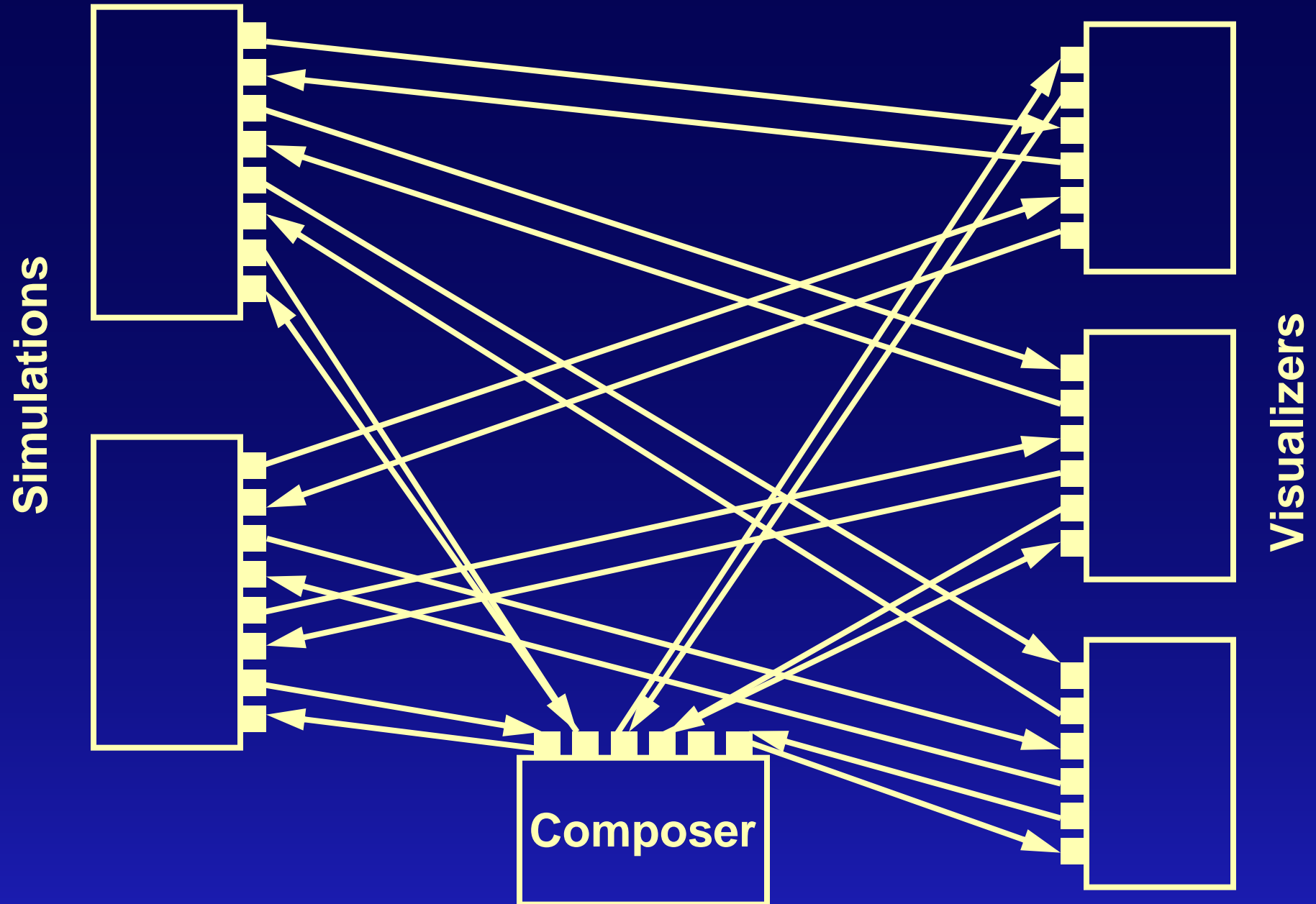


# Awkward for CCA

- Calls on multiply connected uses-ports “broadcast”.  
Must use separate port.
- Invocations are one way.



# The Real Picture



# Connecting Two Ports

```
UsesPort *sim1, vis1;  
PortID sim_uses, vis_prov, sim_prov, vis_uses;  
sim_uses = sim1->new_uses();  
sim_prov = sim1->new_provides();  
vis_uses = vis1->new_uses();  
vis_prov = vis1->new_provides();  
sim1->start_getPort(); // Starts thread on sim.  
vis1->start_getPort(); // Starts thread on sim.  
conService->connect(sim_uses, vis_prov);  
conService->connect(vis_uses, sim_prov);
```

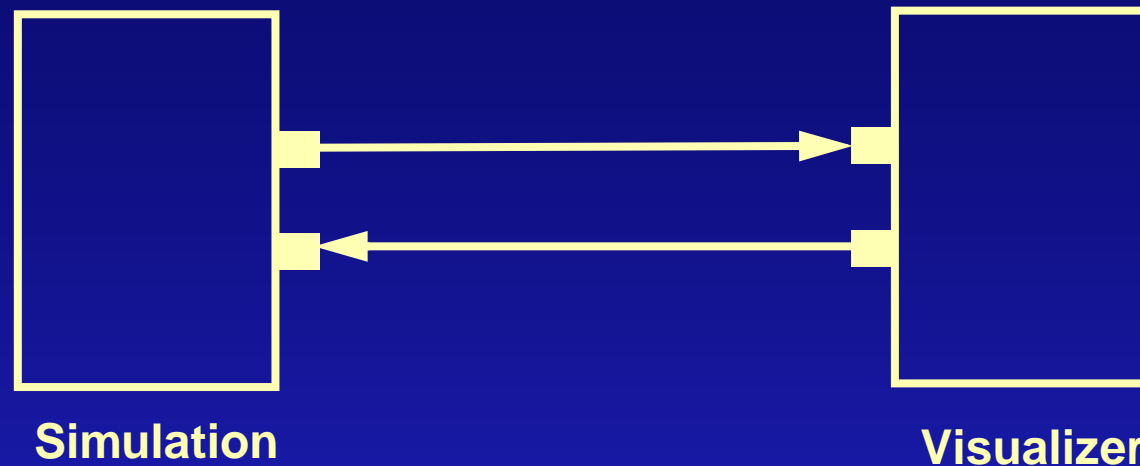
# Separate Connection From Port

- Port is just for rendezvous.
- Actual communication channel is a connection.
- Similar to Berkeley sockets.

```
Port *port;  
PortHandle remote_port;  
RenderConnection *conn;  
  
...  
conn = port->connect(remote_port);  
conn->render(...);
```

# 2-Way Connections

- Distributed computing is increasingly peer-to-peer.
- Peer-to-peer relationships often require invocations to each peer from the other.
- Usually requires two interfaces, four ports and two connections.



```
port Render {
    render(...);
}
port Control {
    set_decimation(...);
    partial_results(...);
}
component Sim {
    uses Render render_receptacle;
    provides Control control_facet;
}
component Vis {
    uses Control control_receptacle;
    provides Render render_facet;
}
```

- Easier to understand, maintain, and use if one interface is used to specify the contract.



```
port Render
  render(...) right;
  set_decimation(...) left;
  partial_results(...) left;
};

component Sim {
  left Render render;
}

component Vis {
  right Render render;
}
```

# Connecting Two Ports

- One-way ports

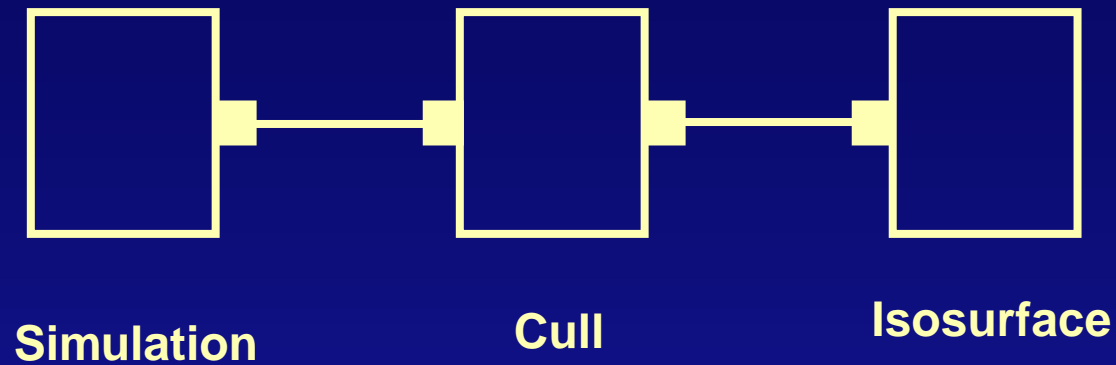
```
UsesPort *sim1, vis1;  
PortInfo sim_use, vis_prov, sim_prov, vis_use;  
sim_use = sim1->new_uses();  
sim_prov = sim1->new_provides();  
vis_use = vis1->new_uses();  
vis_prov = vis1->new_provides();  
sim1->start_getPort();  
vis1->start_getPort();  
conService->connect(sim_use, vis_prov);  
conService->connect(vis_use, sim_prov);
```

- Two-way connections

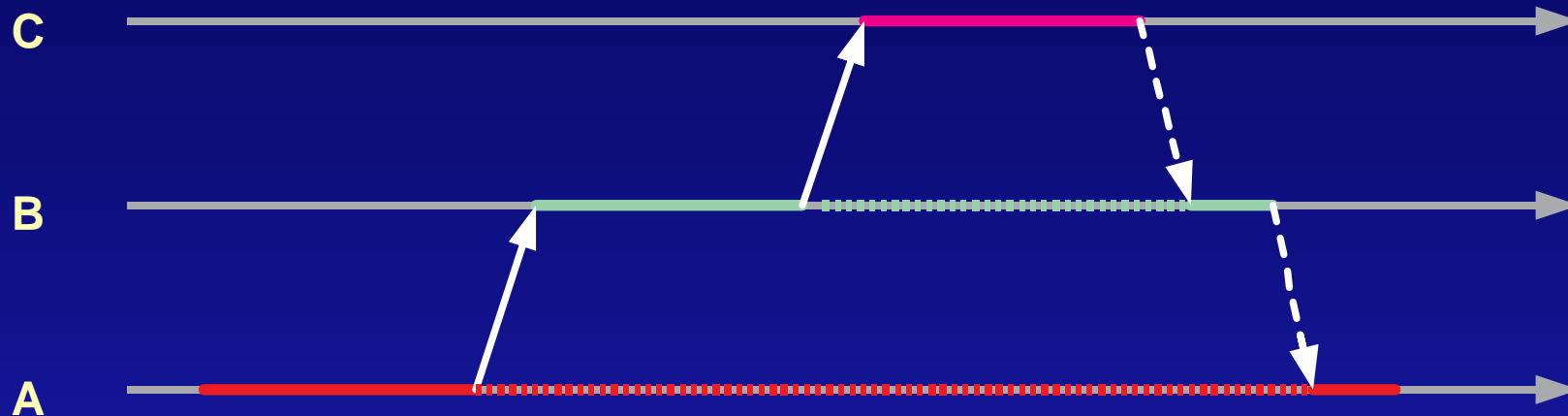
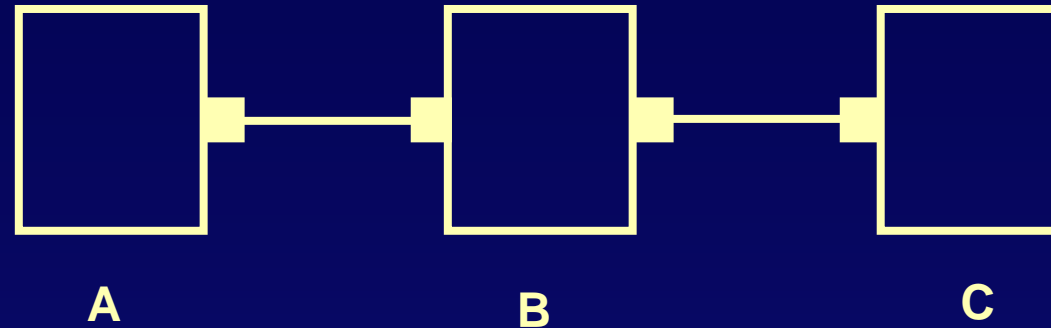
```
PortHandle *sim1, *vis1;  
sim1->connect(vis1);
```

# Concurrency

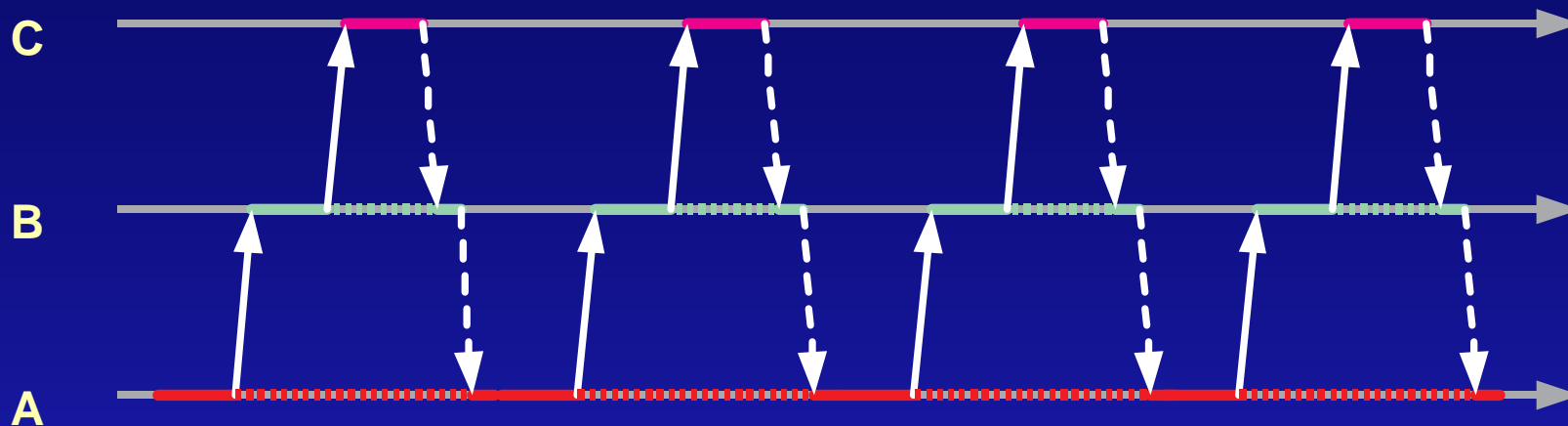
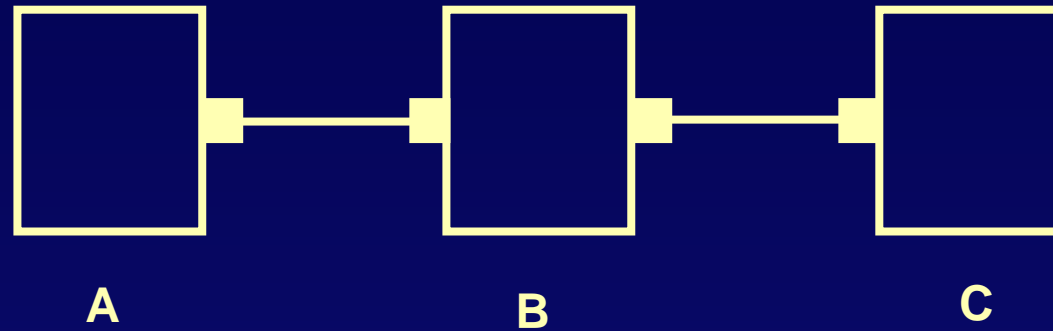
- Part of the motivation for components is pipelined concurrency.



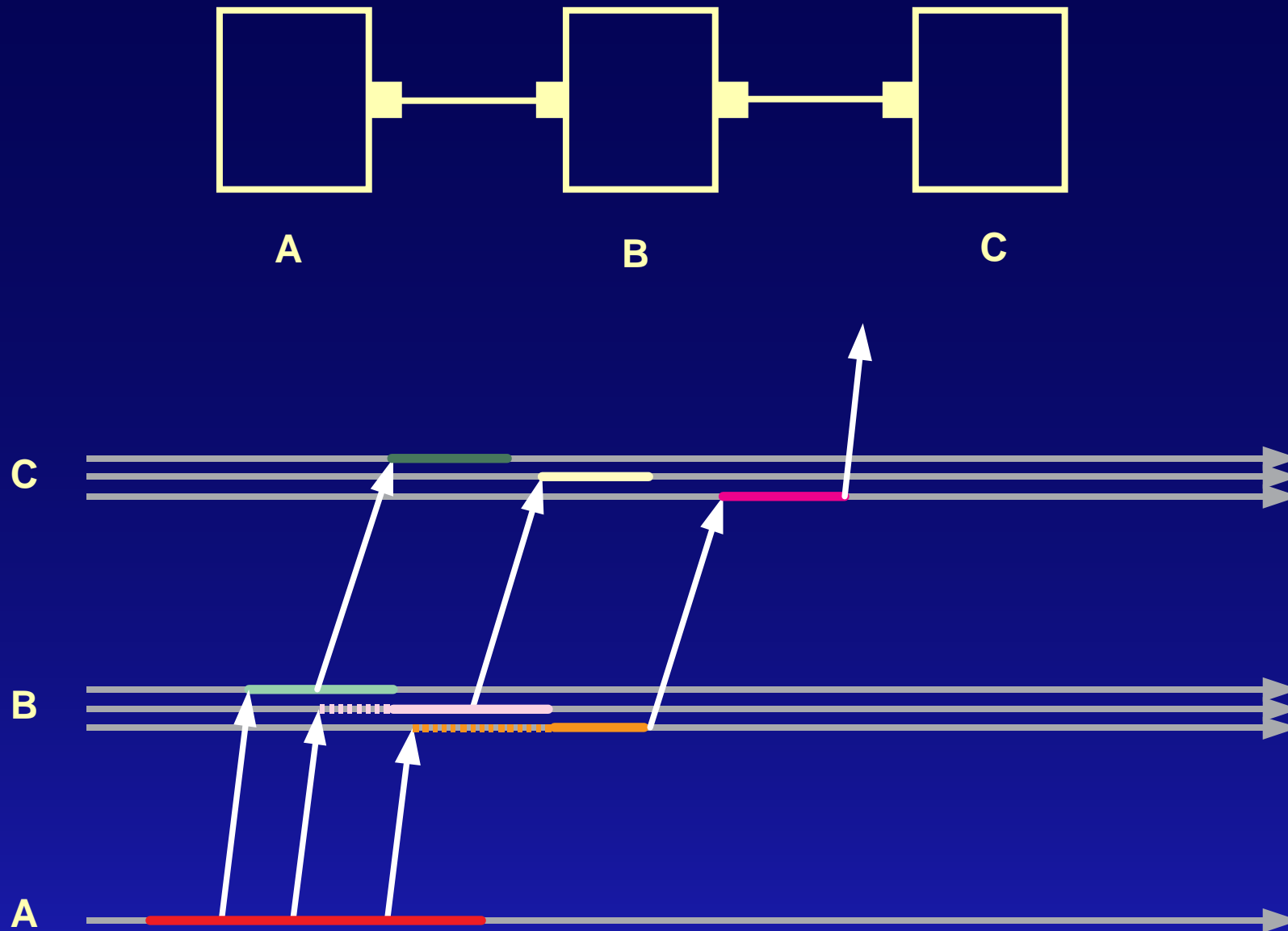
- Obtaining it is not always easy, however.



- So we try “chunking” the data.



- Asynchronous “chunking”



# What Is Our Goal?

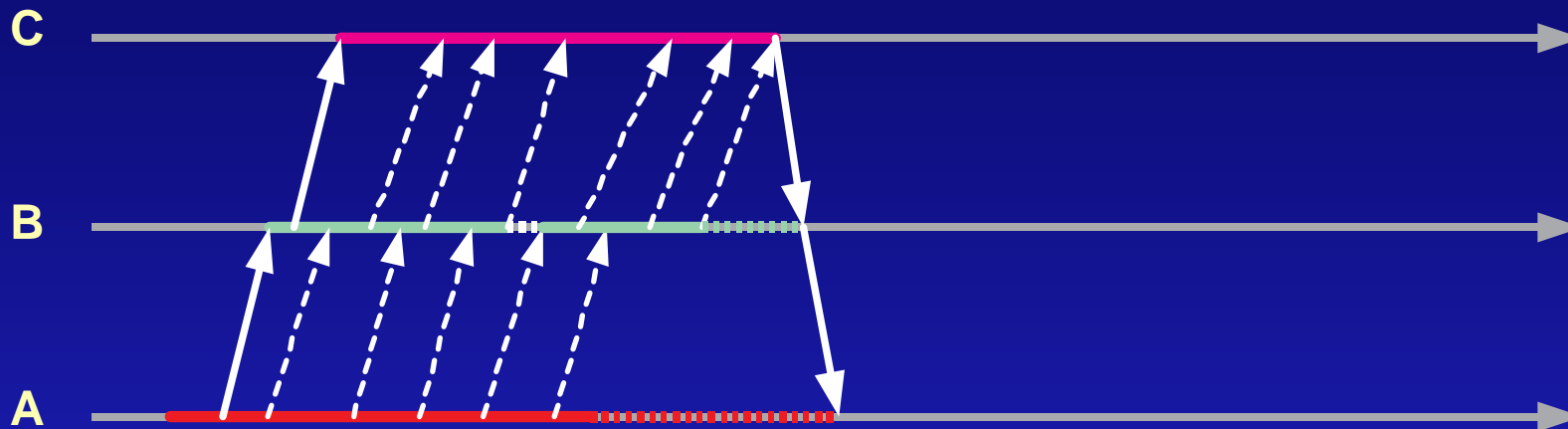
- Trying to obtain concurrency.
- RMI is intended to behave like a function call.
- Function calls generally do not exhibit concurrency.

We tend to think of a function call as being executed by a single thread of control.

- Would like to use control flow to maintain state.

# Remote Invocation Channel

- Borrows from message-passing.
- Each call carries a channel. Channel can be used to send and receive data within context of a single call.
- Similar to a mini-MPI session in each RMI.



## Caller

```
RenderConn *conn;  
Channel *ch;  
ch = conn->render(...);  
    while (...) {  
        ch->write(...);  
    }  
rc = ch->complete();
```

## Callee

```
double render(Channel *ch, ...) {  
    ...  
    while (!ch->closed()) {  
        ch->read(&data);  
    }  
    return flux;  
}
```

# Conclusions

- Need to validate on real application.  
X-Ray crystallography project.
- CCA-like component programming is relatively new.  
Perhaps paradigms besides RMI might be more appropriate. Really depends on how components are used.
- Implementation of stubs and skeletons somewhat more complex.  
I believe most of this can be hidden from the component programmer, however.

# Non-Blocking Synchronization

# Problems with Mutual Exclusion

- Fault-intolerance

A process which fails while holding a lock leaves the system in an inconsistent state.

- Priority inversion

A low-priority process can indefinitely block a high-priority process (Mars Pathfinder).

- Deadlock

- Low throughput

High-contention, too many writes.

# Terms

- Wait-free

All processes deterministically guaranteed to complete.

- Non-blocking

One process guaranteed to complete.

- Type-stable memory (TSM)

A formalism of the idea that an object remains valid even after it is “freed.”

# Synchronization Instructions

- TEST&SET (2)

Atomically read a shared variable and set it to one.

- FETCH&ADD (2)

Atomically read a shared variable and increment it.

- LOAD-LINKED/STORE-CONDITIONAL ( $\infty$ )

The first instruction reads a variable. The second instruction writes it only if no other writes intervene.

- COMPARE&SWAP ( $\infty$ )

Given a shared variable  $V$ , an old value  $O$  and a register  $N$  containing the new value, the instruction

COMPARE&SWAP  $V,O,N$

will swap  $V$  with  $N$  only if  $V = O$ .

Two variations of COMPARE&SWAP

- Double COMPARE&SWAP

Two discontinuous words. Not commonly available.

- Double-word COMPARE&SWAP

One aligned double-word. Commonly available.

# Previous Work

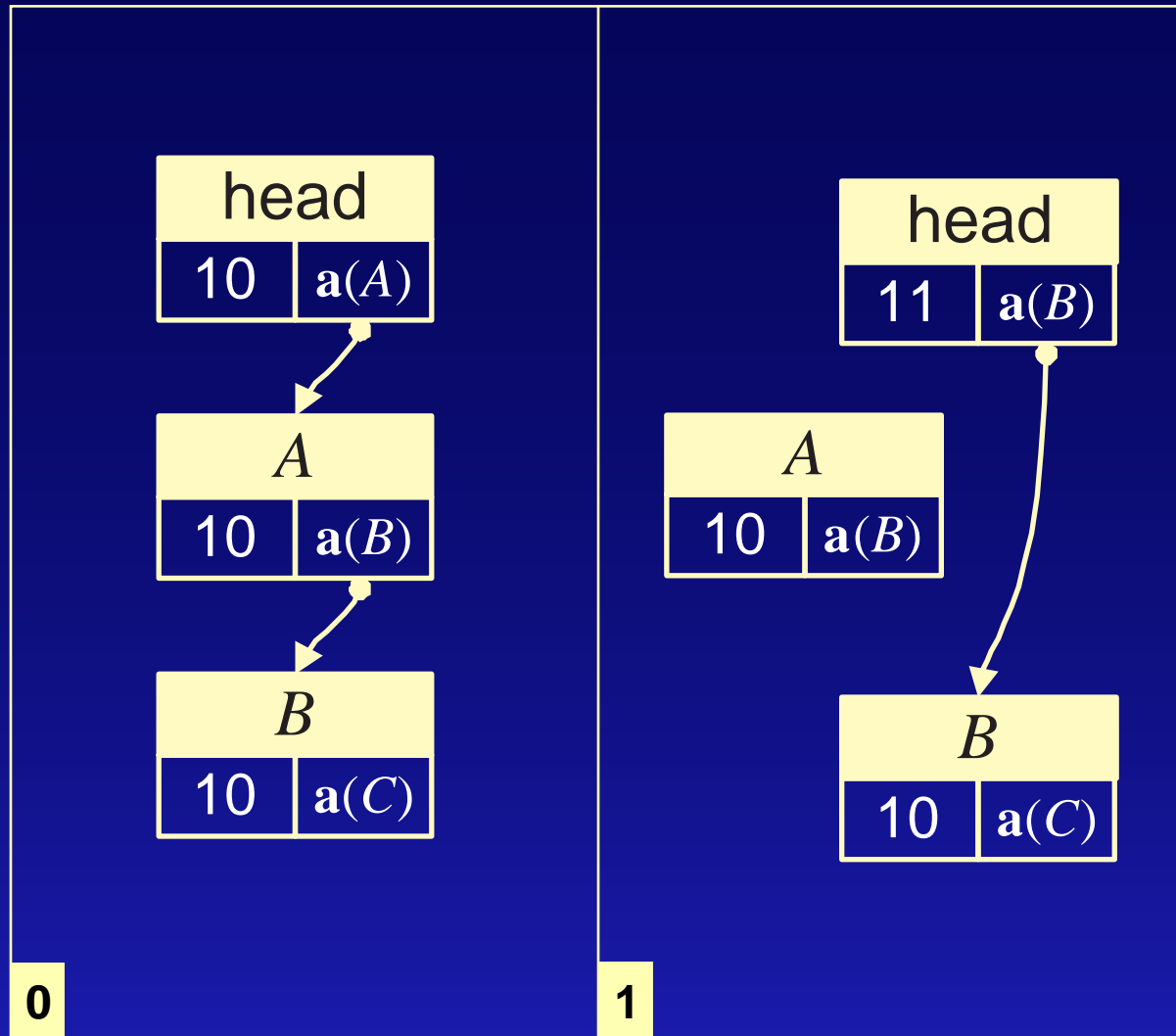
- Herlihy (1991)  
General transformation techniques. Consensus hierarchy.
- Valois (1995)  
Linked lists using COMPARE&SWAP. Traversal requires writes.
- Michael and Scott (1996)  
FIFO queues using double-word COMPARE&SWAP.
- Greenwald and Cheriton (1996)  
Linked list using double COMPARE&SWAP.

# Problem Definition

- Singly-linked list
- Arbitrary insertion
- Arbitrary removal
- Assume TSM
- Assume double-word COMPARE&SWAP

# Example of NBS

Popping an element from a stack.



$a(E)$  = address of element  $E$

```

struct Link {
    int gen;
    Element *ptr;
} head;

```

```

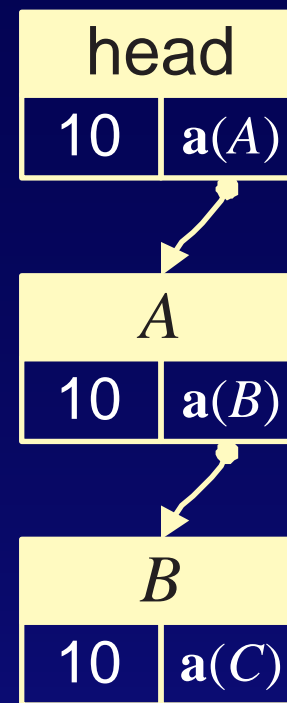
struct Element {
    ...
    Link next;
};

```

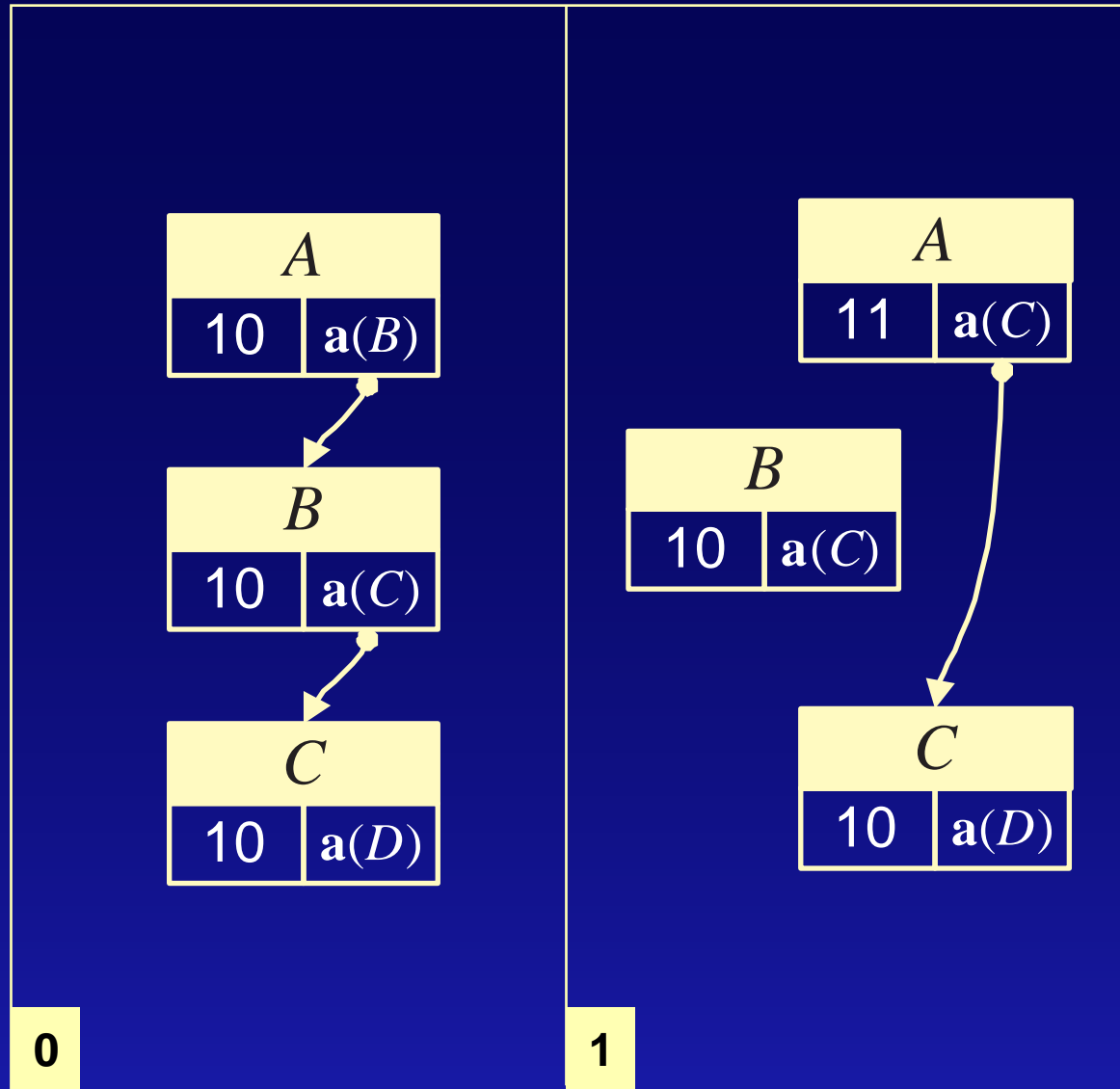
```

do {
    Link new_head = org_head = head;
    new_head.ptr = new_head.ptr->next.ptr;
    new_head.gen++;
} while (!cas(&head, org_head, new_head));

```

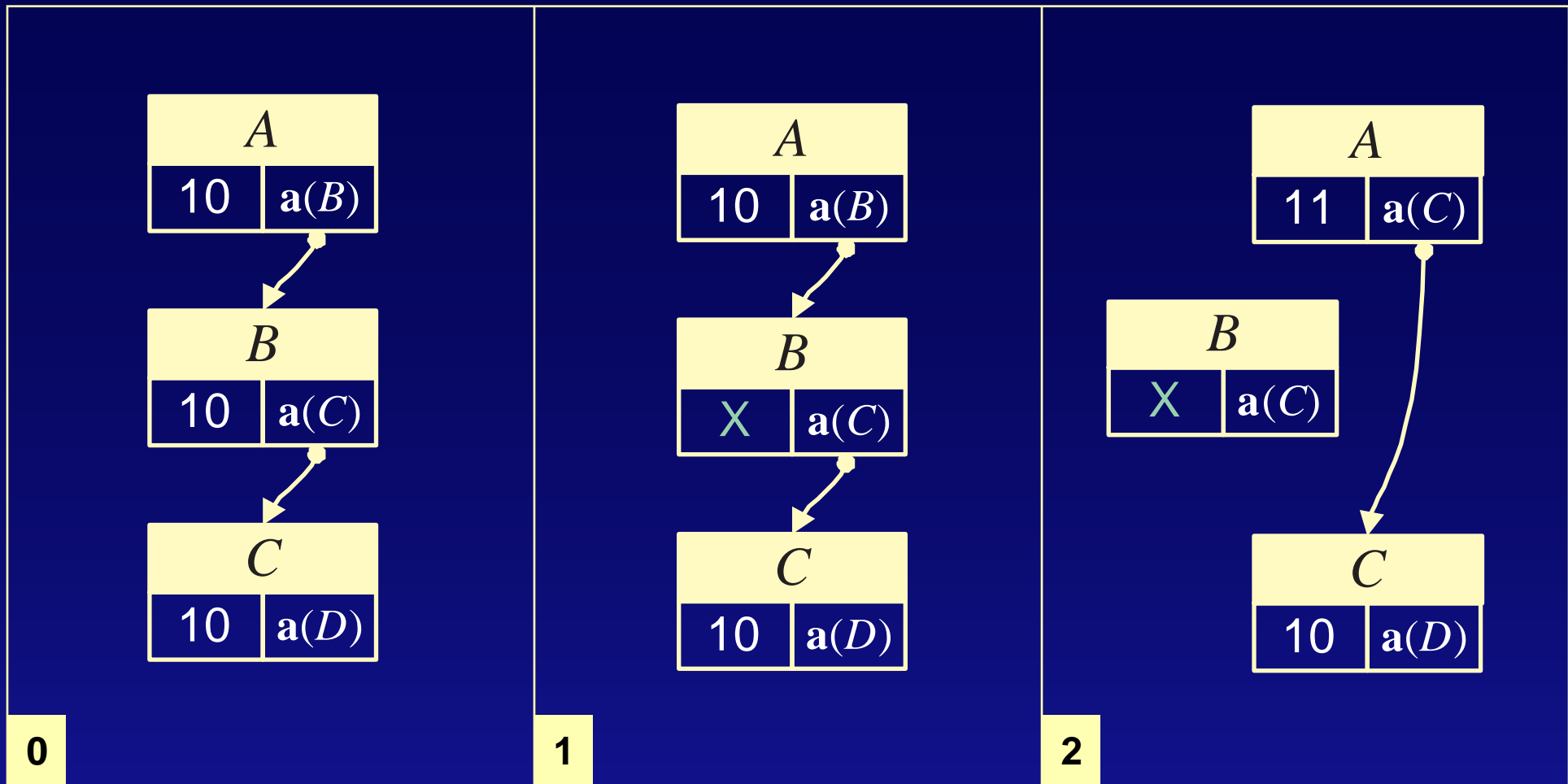


# Does This Work for Lists?



Key difference is that elements are removed only from the head of a stack.

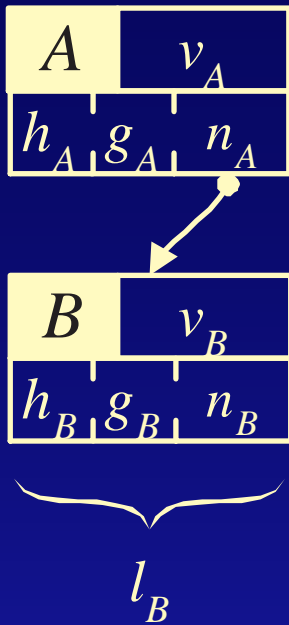
# Marking



To preserve non-blocking property, other processes are allowed to remove a marked element.

# Two Tags

- Two tags are stored along with each pointer. This triple constitutes a link.



$g_E$  = generation of element  $E$

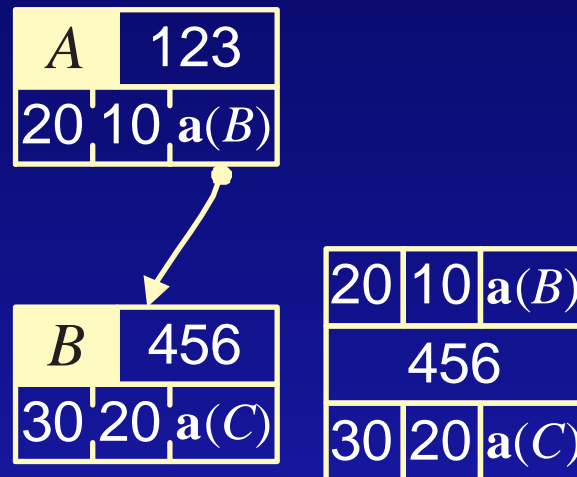
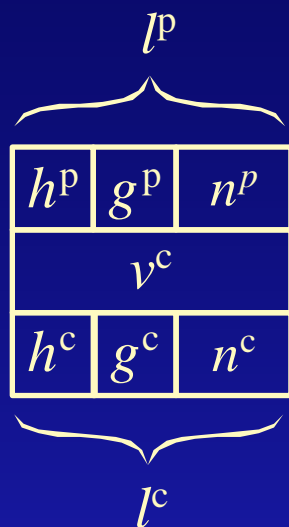
$h_E$  = generation of successor to  $E$

$n_E$  = pointer to successor

$l_E = (g_E, h_E, n_E)$

# Cursor

- Represents a snapshot in “time.”
- Contains two links and the value of the visited element.
- Aids detection of conflicts.



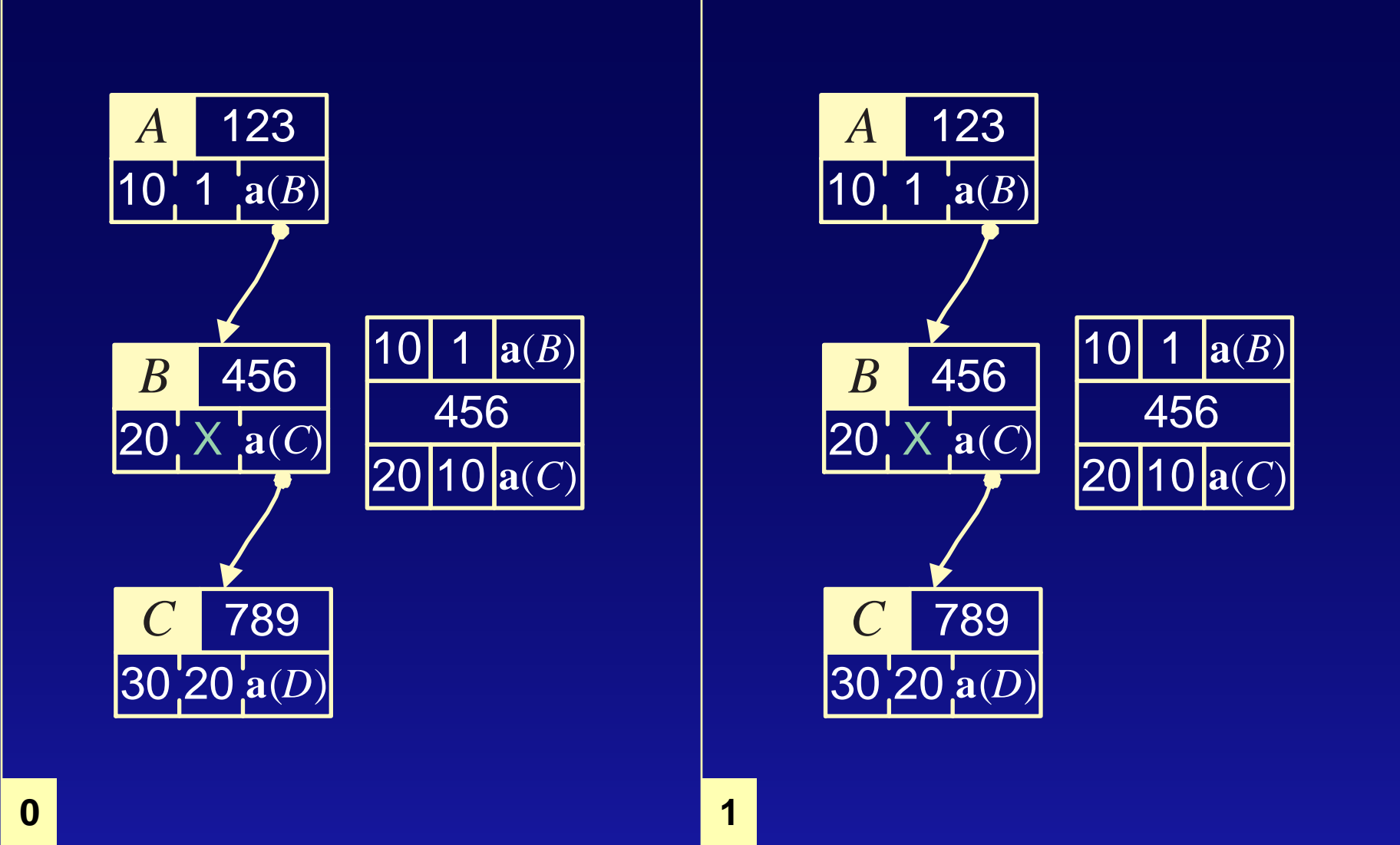
If the cursor is visiting  $B$ ,  
then normally

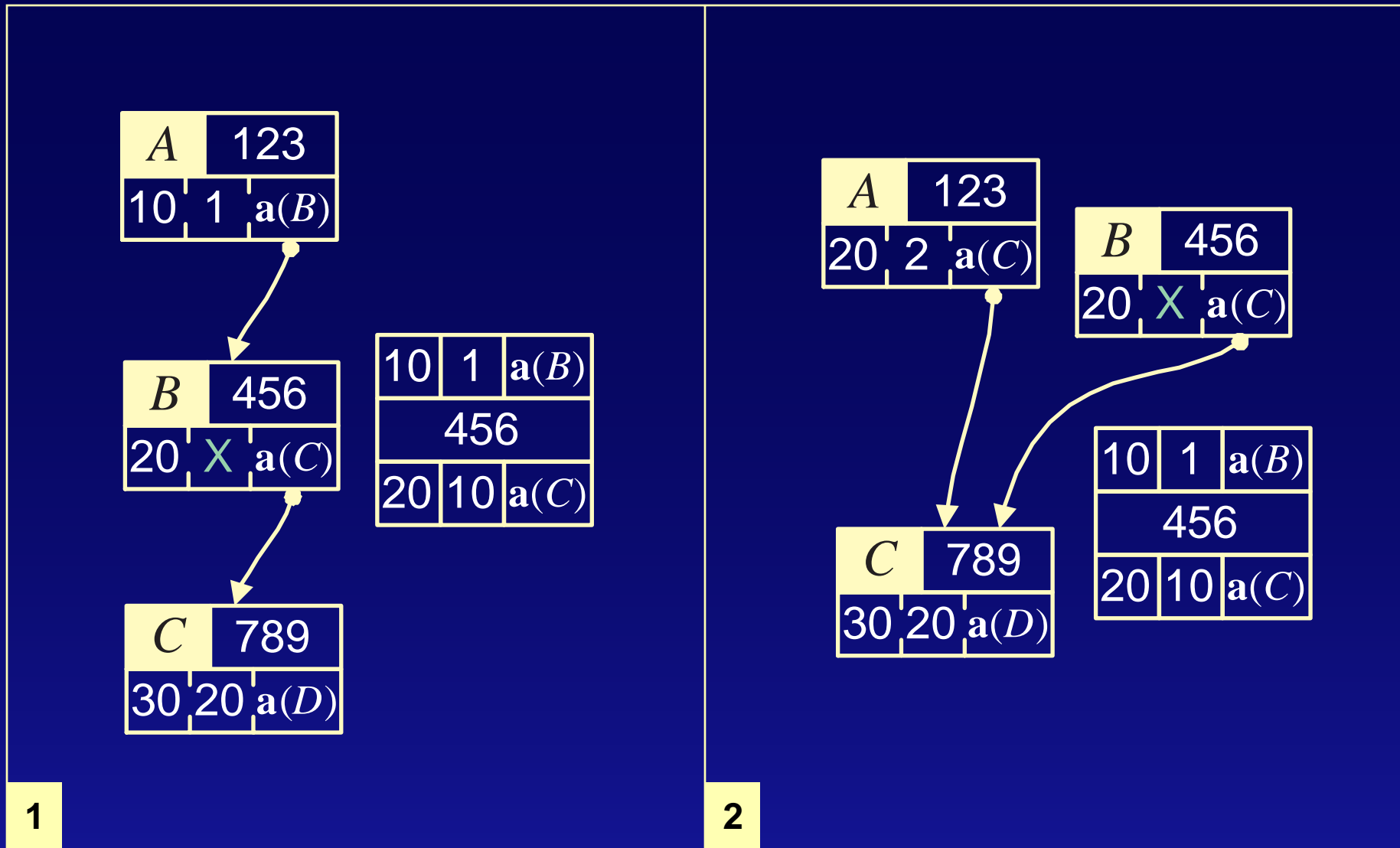
$$l^p = l_A$$

$$l^c = l_B$$

$$v^c = v_B$$

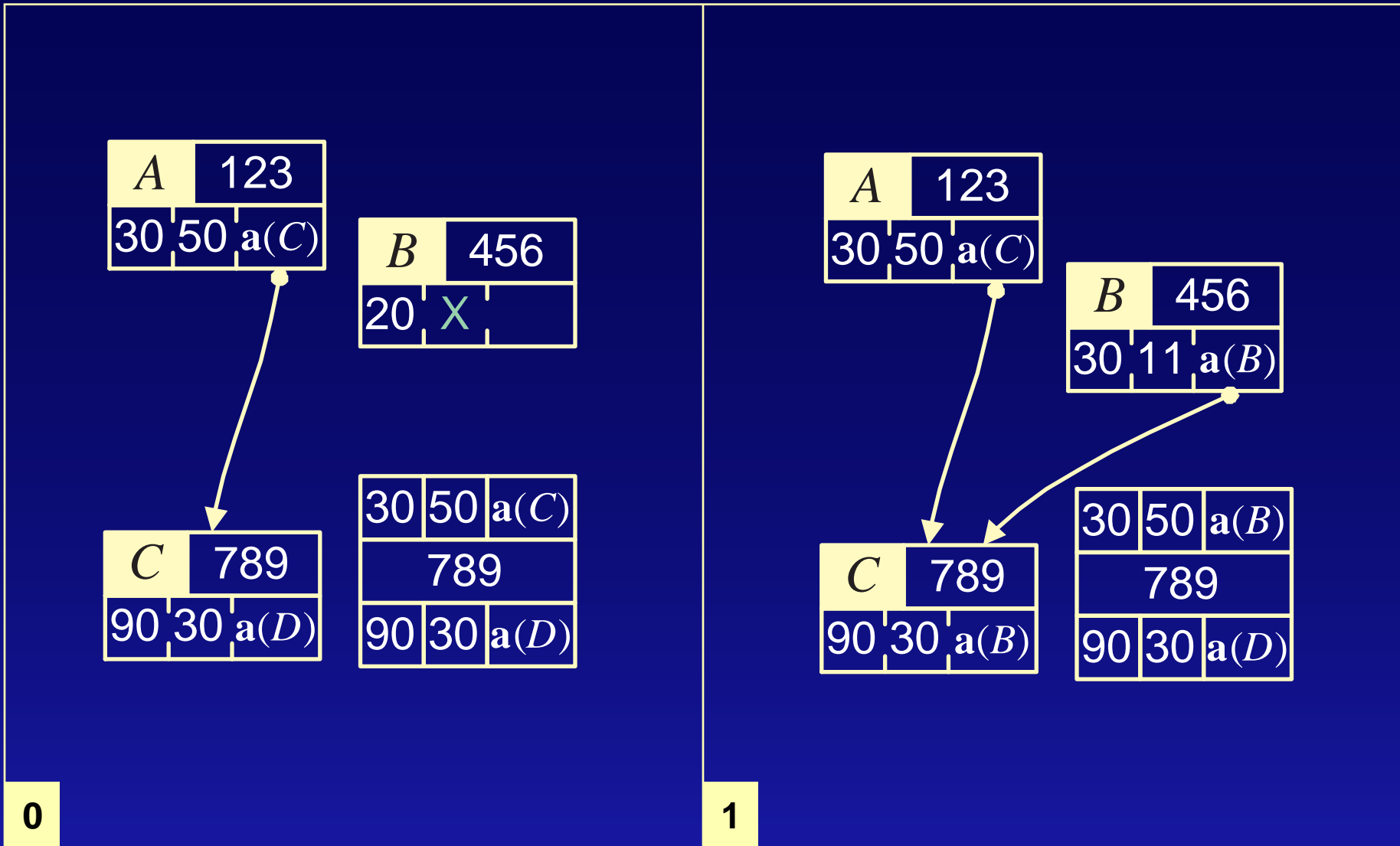
# Removal

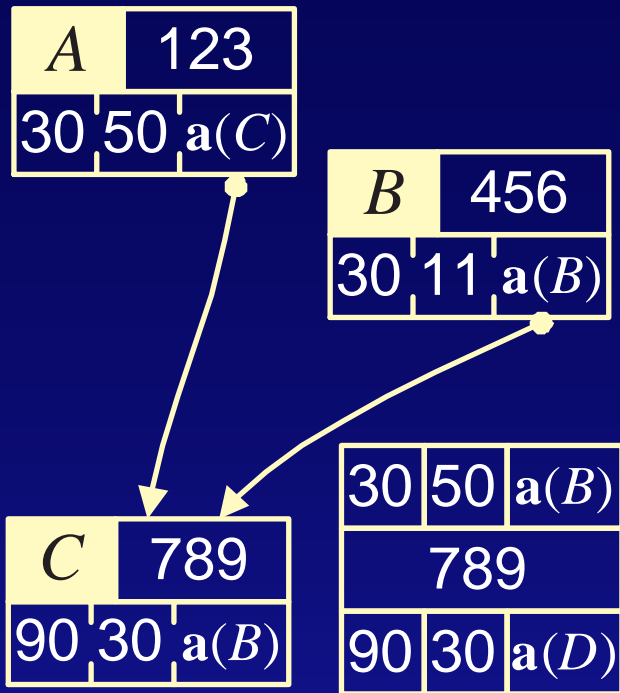




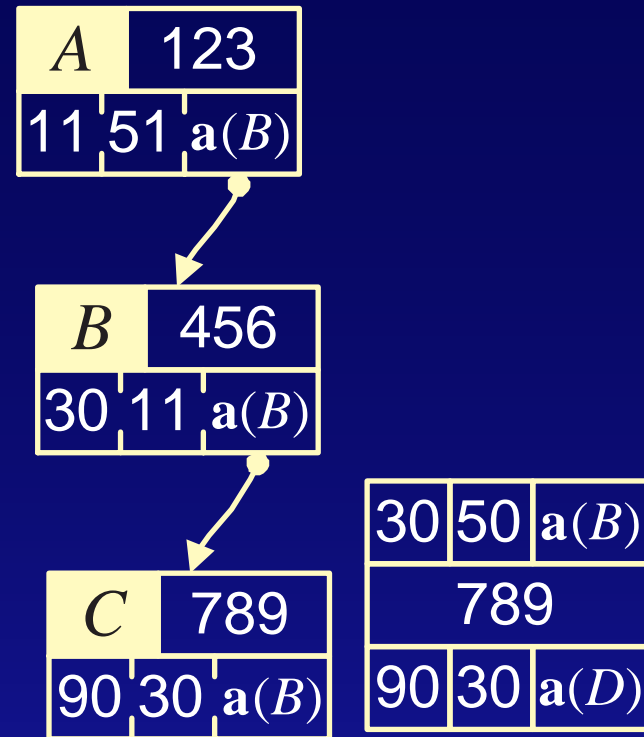
If a conflict is detected, must restart from head of list.  
 Minor conflicts can be fixed-up with restarting.

# Insertion



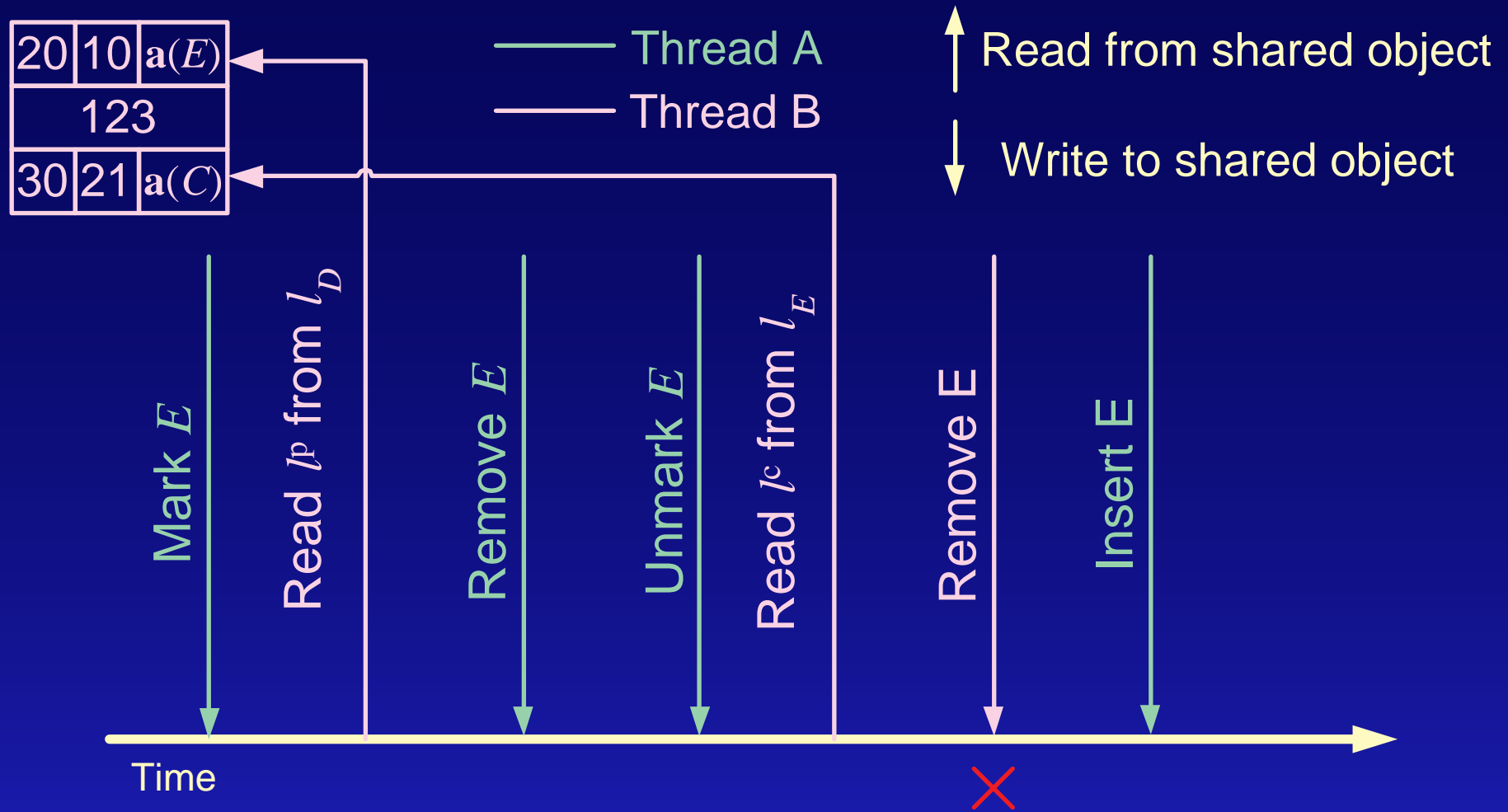


1



2

# Purpose of $h$



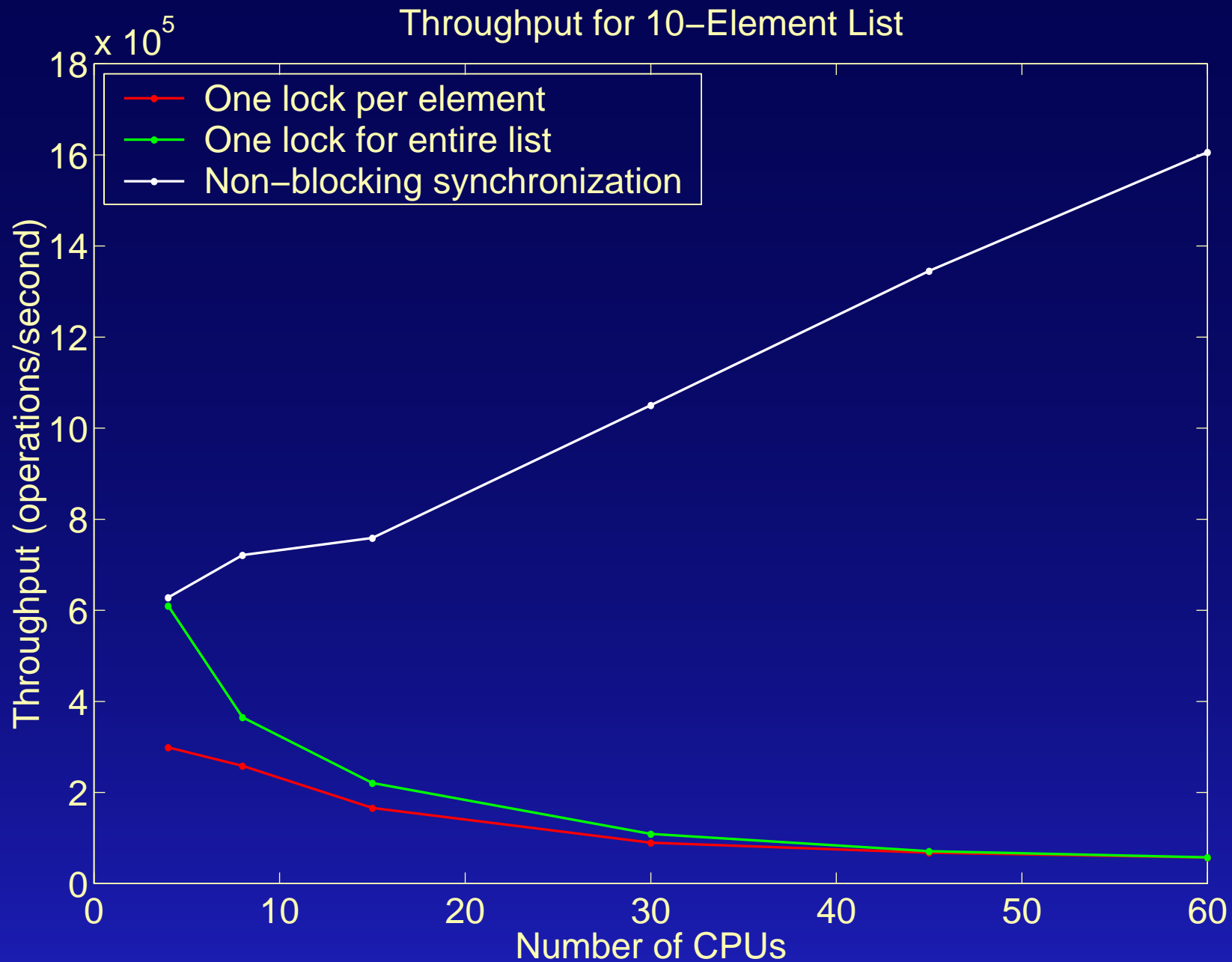
# Performance

- Tests conducted on an E10000.
- Operations are a mixture of search&removal, and insertion.

```
while (true) {  
    r = random();  
    Element *e = list.remove(r);  
    if (e != NULL) {  
        e->value = random();  
        list.insertSorted(e);  
    }  
}
```

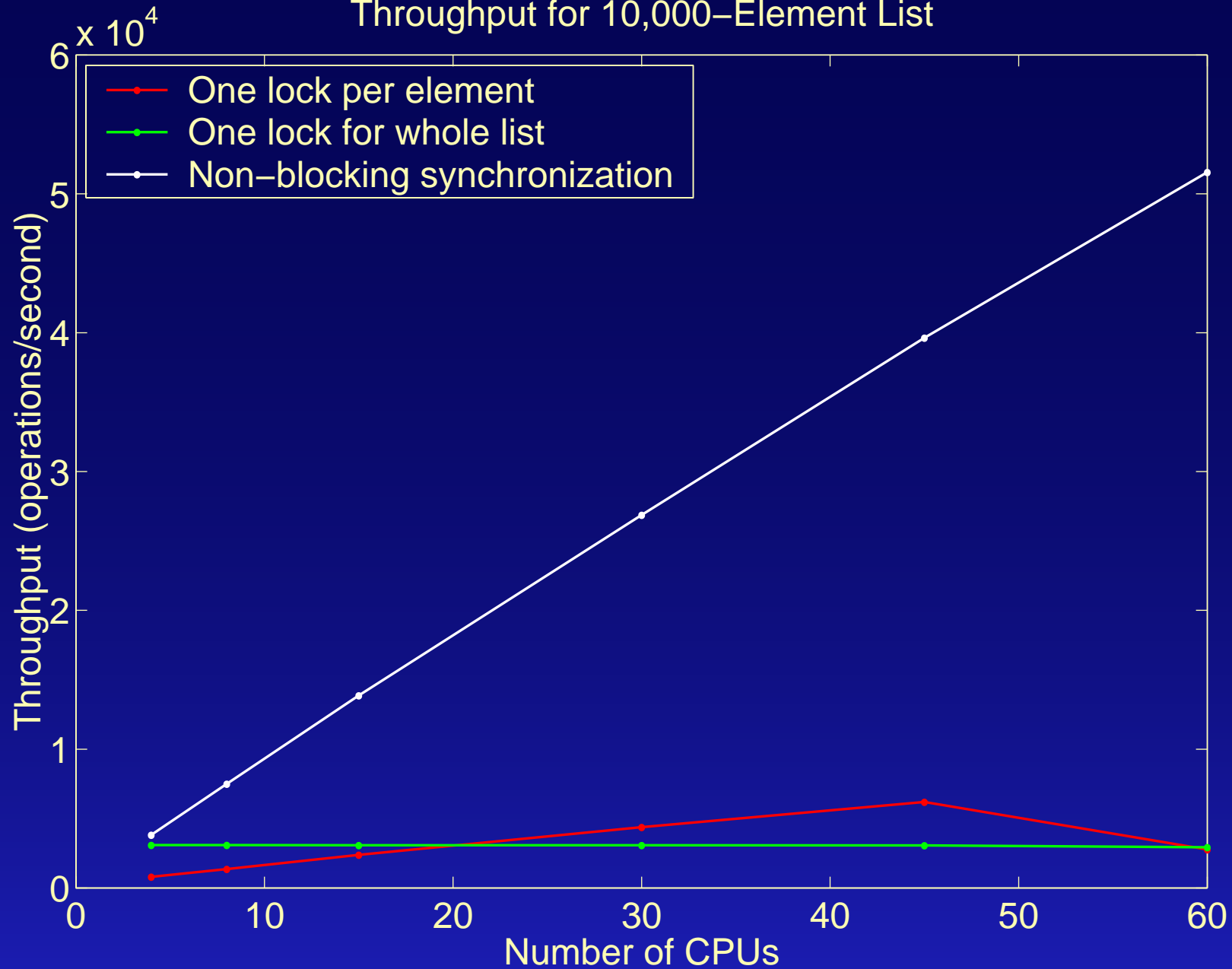
- Tested against one lock for each element and one lock for the entire list. Locks in the locking algorithms are spin locks spinning on read.
- Comparison against locking methods intended merely as a baseline sanity check.

# 10 Elements



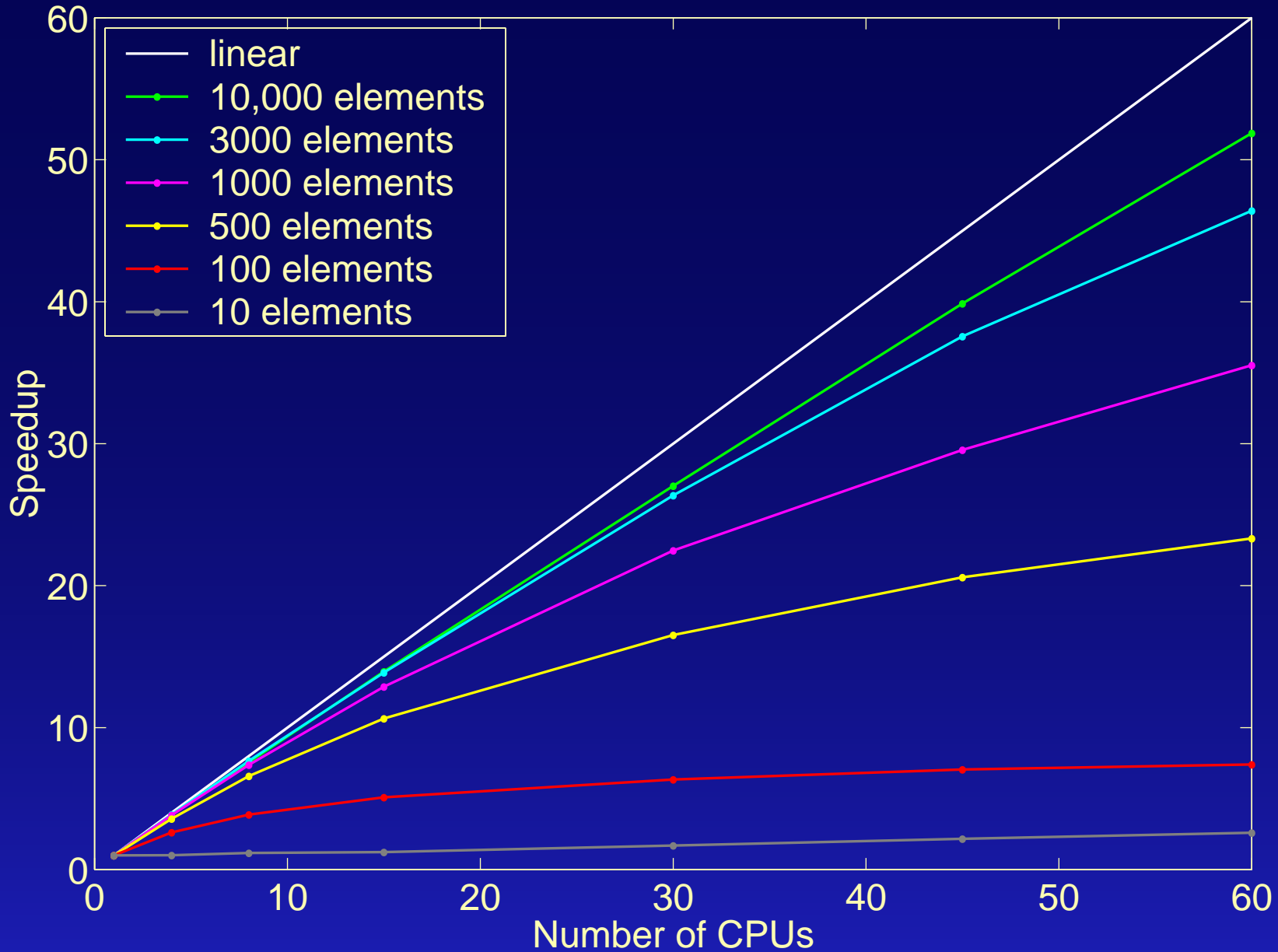
# 10,000 Elements

Throughput for 10,000-Element List



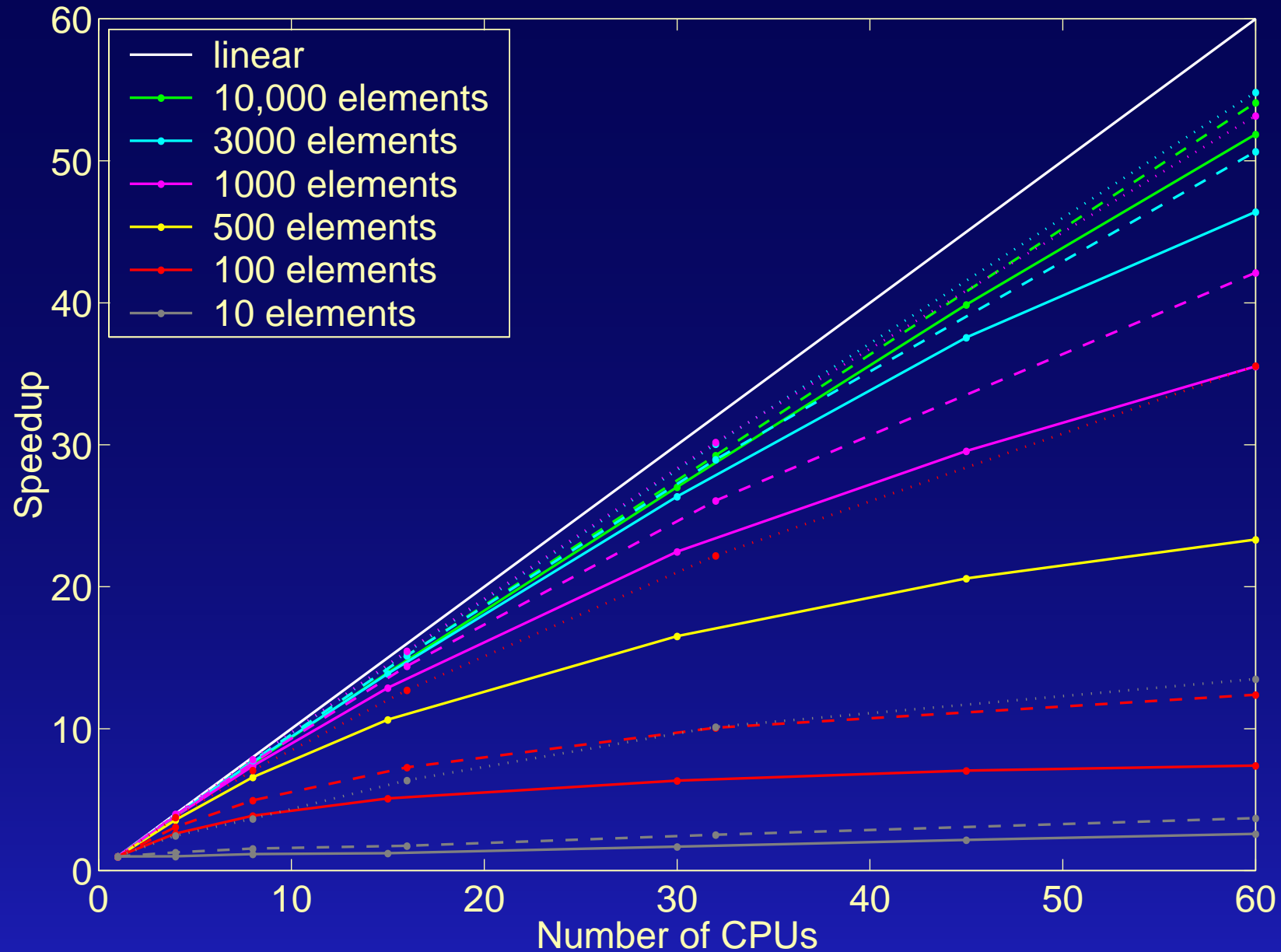
# Speedup

Effect of List Size on Speedup



# Expensive Compares

Effect of Comparison Cost on Speedup



# Future Work

- Formal verification

A number of incorrect algorithms have been published.

- Performance enhancements

Reduce memory operations. Reduce restarts. Perhaps wait bit if encounter a marked element.

- Relaxed memory models

These allow operations to be drastically reordered. Will probably require extra memory barriers.

- NUMA architectures

- Other data structures

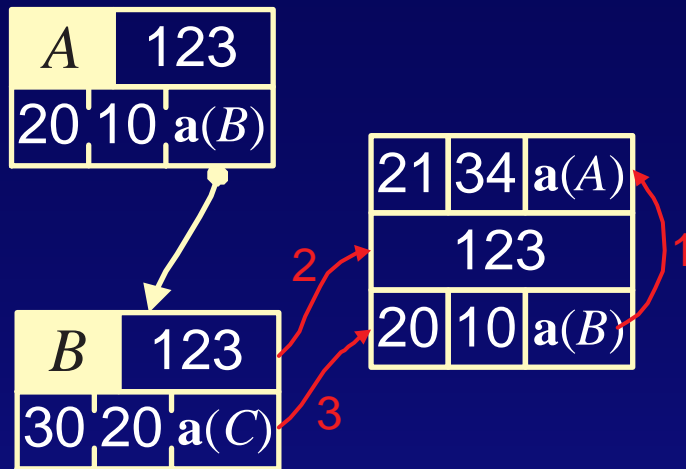
# Scratch Slides

# Consensus Hierarchy

- Some primitives are more powerful than others.

# Advancing a Cursor

The cursor below is being advanced from  $E_1$  to  $E_2$ .



$$l_p \leftarrow l_c$$
$$v_c \leftarrow v_2$$
$$l_c \leftarrow l_2$$

Clearly not atomic, but ignore that for now.

# Validating a Cursor

Advancing a cursor is not an atomic operation. It may no longer be valid. Validation is the process of fixing it up. Two conditions require fixing.

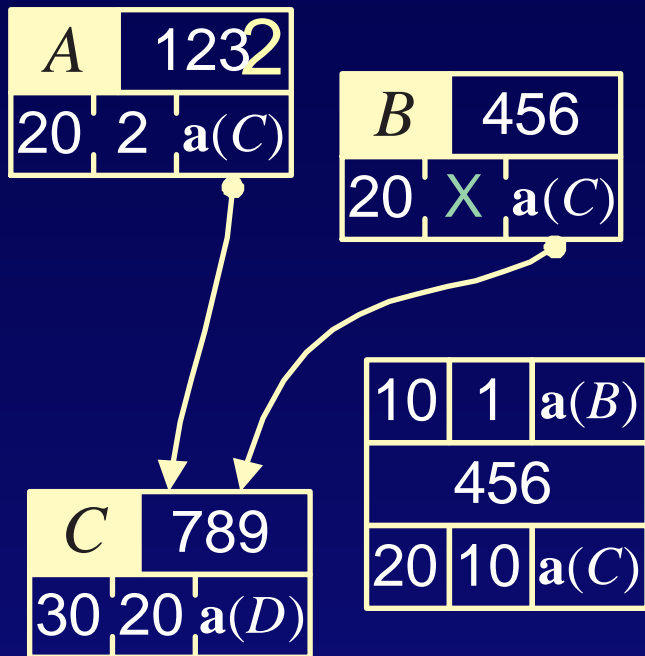
20	10	a(B)
123		
30	X	a(C)

If the current element has been marked, then we need to remove it before continuing. If, upon trying to do so, we detect that  $g_{i-1} \neq g_p$ , then this indicates the current element may have been removed and reinserted. This may cause portions of the list to be skipped, so a restart is necessary. If only  $h$  differs, fix and continue.

Another condition is that  $h$  may be out-of-date. That is,  $hp \neq gc$ . We try to correct this situation. If it fails because  $h$  has changed, we can reload with the new value of  $h$  and try again.

If it fails because  $g$  has changed, many things could have happened, and we restart.

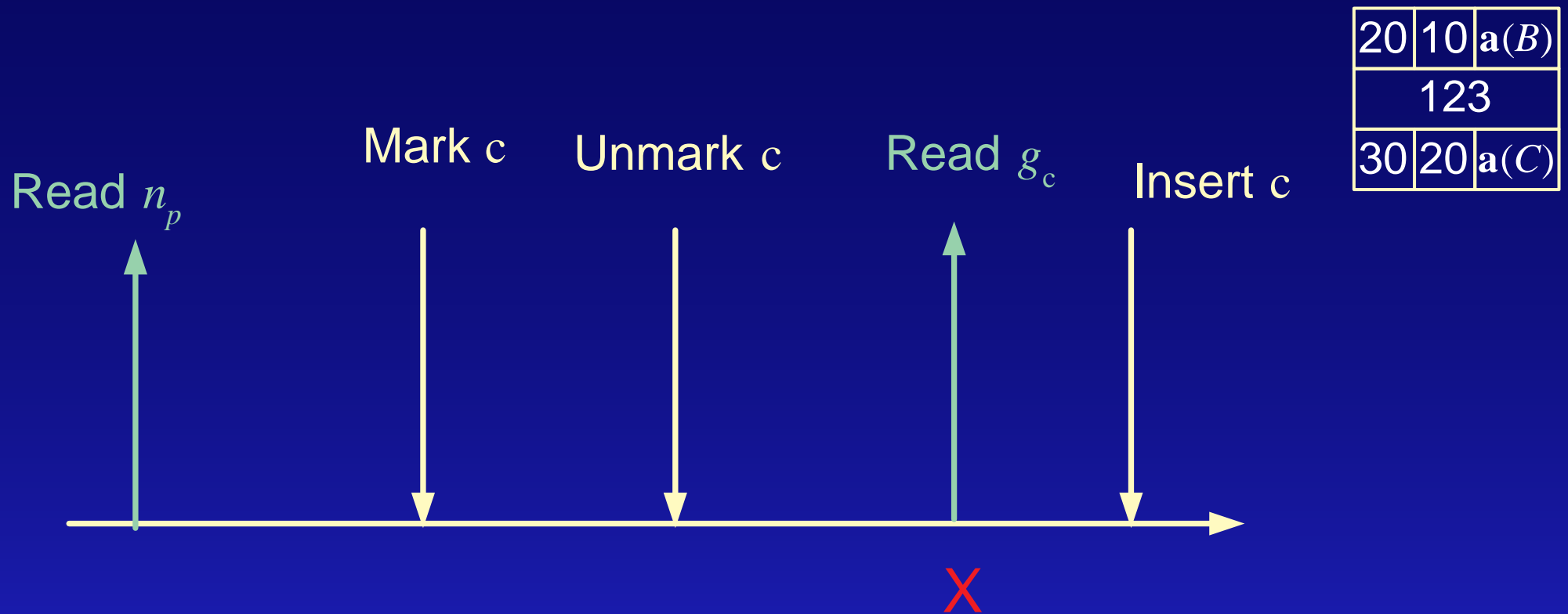
20	10	$a(B)$
123		
30	20	$a(C)$



If the COMPARE&SWAP fails, it might be for a number of reasons. Must retrace the whole list. If we don't find this element again, then we can be sure it has been removed properly.

# The $h$ Field

The  $h$  field of the link serves at least two important functions. The first is that it XXXX that when a cursor is advanced, the value read from the element is consistent with the links.



The second important function is that it prevents an element about to be inserted from being prematurely remarked for removal.

20	10	a(B)
123		
30	20	a(C)

