An Architecture for Concurrent, Peer-to-Peer Components

Kenneth Chiu
Indiana University

Software Components

- Deployable by end user
- Relatively large
- Possibly long-lived
- Distributed
- Interactions constrained
- Simple enough to be developed by domain scientists

Examples

- CORBA Component Model (CCM)
- DCOM
- 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

Ports

- Components communicate through ports.
  - *Provides*-ports

    An interface provided by a component.
  - *Uses*-ports

    An unbound, named, remote reference.

  Connecting a uses port to a provides binds the reference.

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

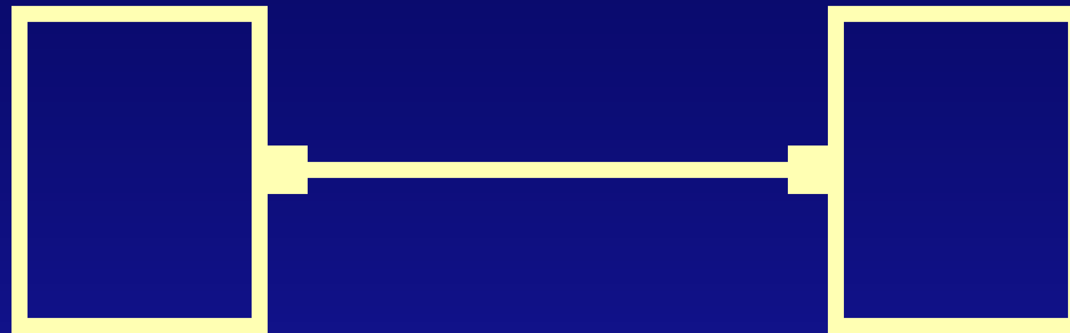
  A single interface can specify the complete relationship between two peers.

- Streams

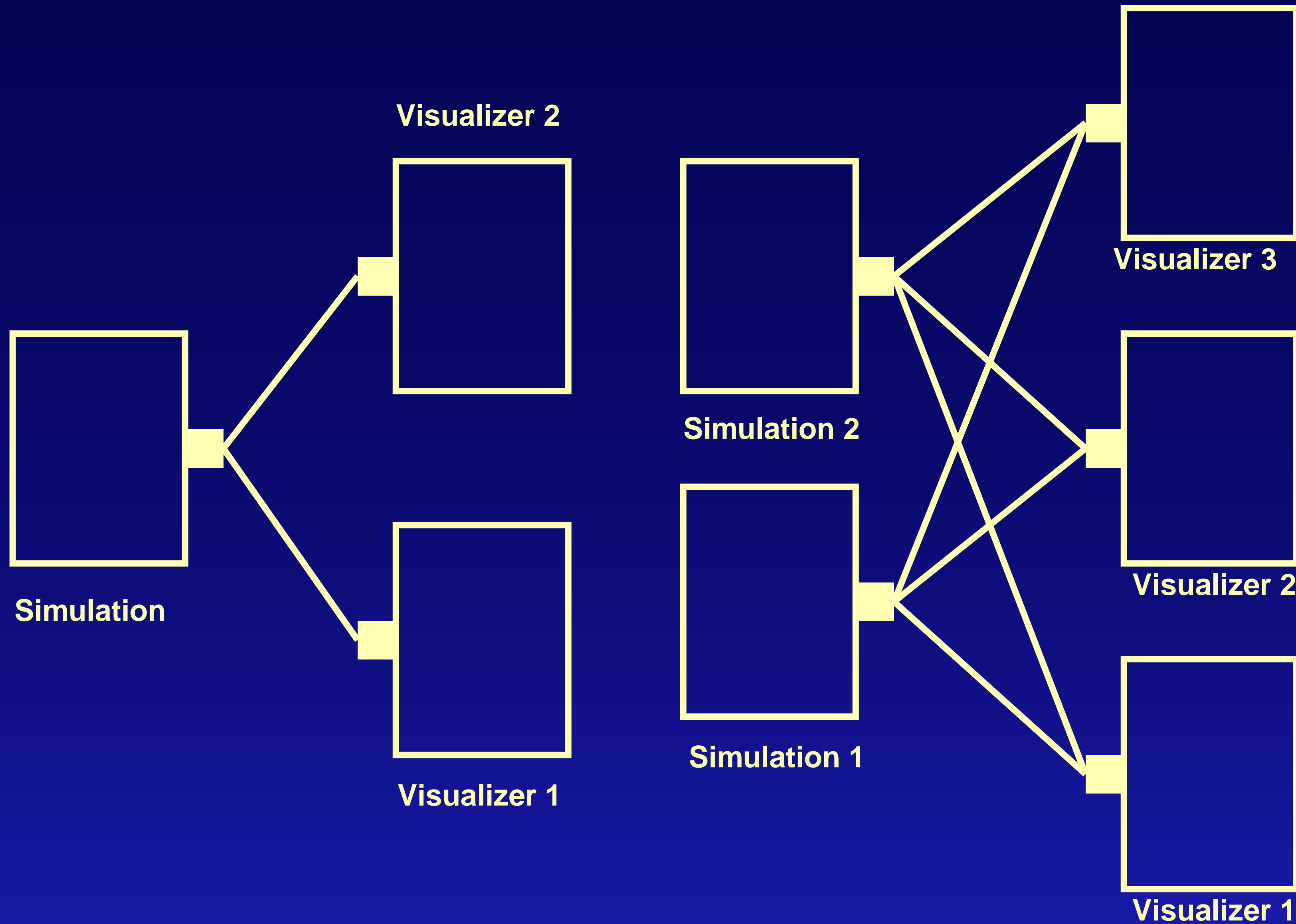
  Stream objects can be passed as arguments to remote methods. During the call, the stream can be used to send data from the caller to the callee.

Ports and Connections

Component ports are intended as an analog to hardware ports.



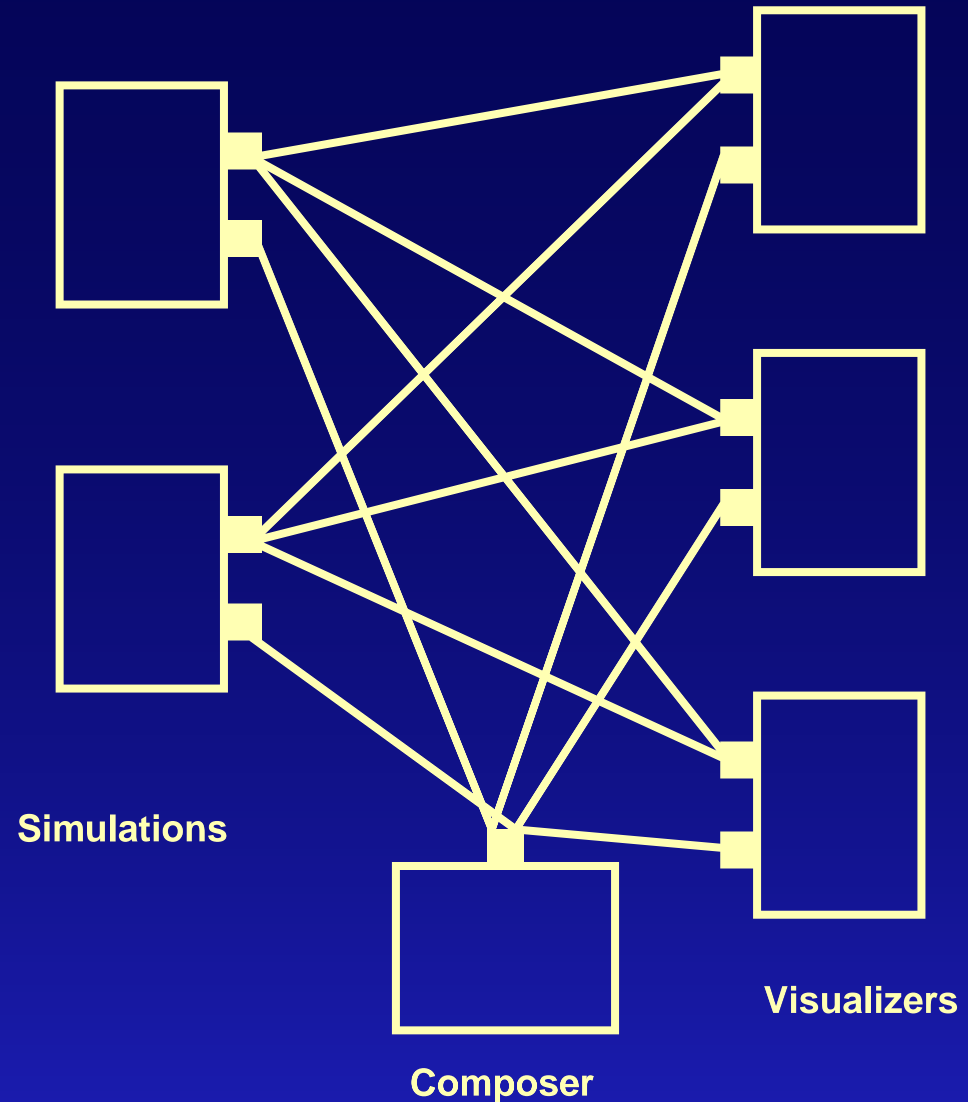
Multiple Connections



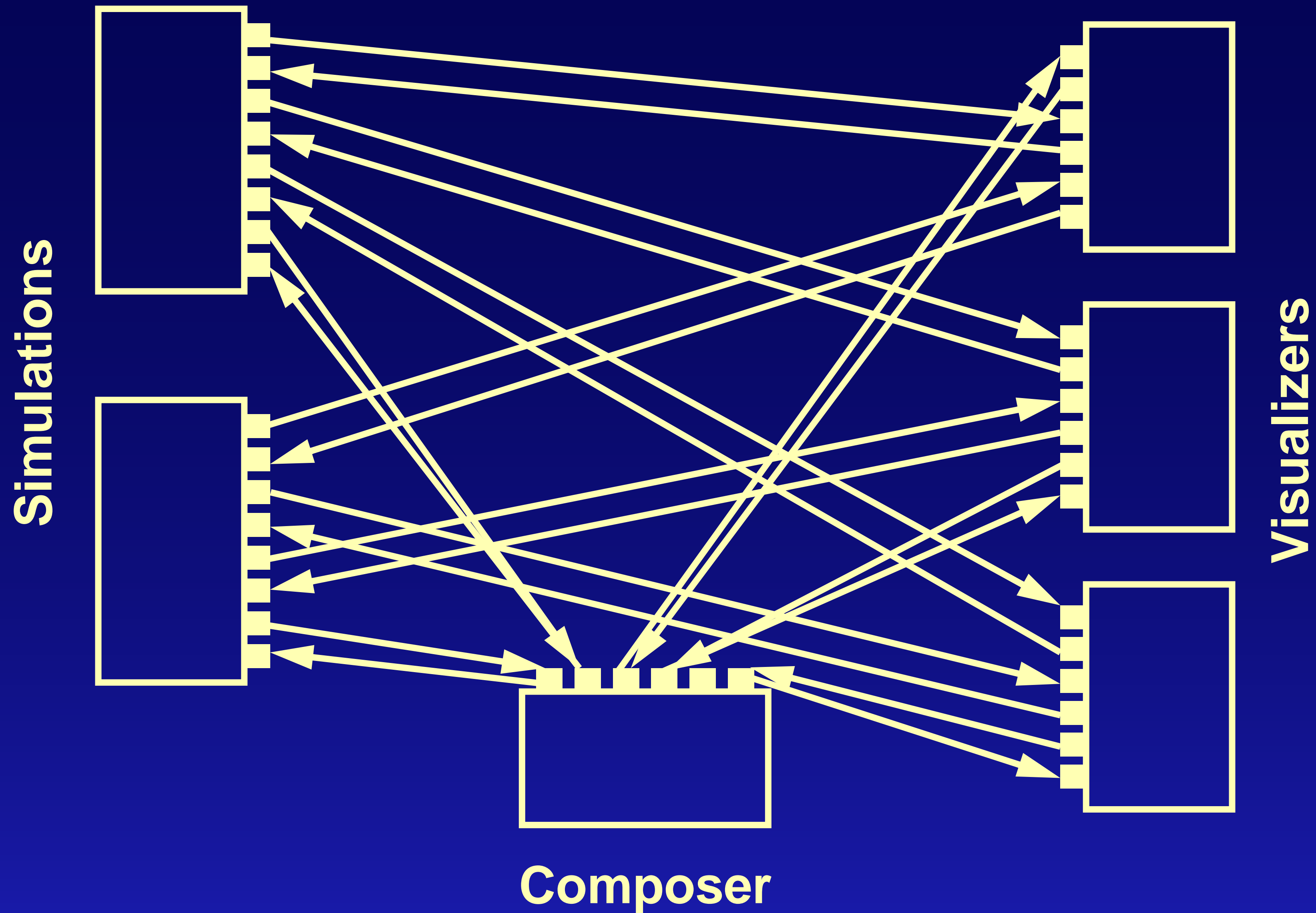
Awkward for CCA-Like Archs

- Calls on multiply connected ports do not maintain identity.

  Must use separate port.
- Invocations are one way.



The Real Picture



Connecting Two Ports

```
// Create new unique ports.
my_pp = new MyPPort(..);
MyPPortInfo my_ppi("my_pport");
svcs->addProvidesPort(my_pp, &my_ppi);
MyUPortInfo my_upi(...);
svcs->addUsesPort(&my_upi);
// Get remote unique ports.
conn_svc->connect(vis, "my_reg", sim, "sim_reg");
my_reg = getPort("my_reg");
(sim_uport, sim_pport) = my_reg->getUniquePorts();
svcs->releasePort("my_reg");
// Connect new ports.
conn_svc->connect(me, "my_uport", sim, sim_pport);
conn_svc->connect(sim, sim_uport, me, "my_pport");
my_uport = getPort("my_uport");
my_uport->method(...);
```

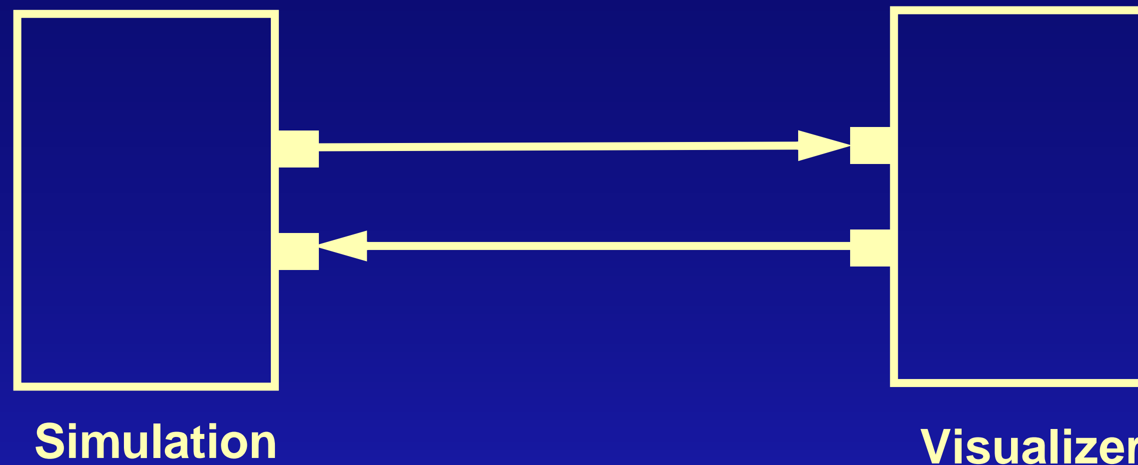
Separate Connection From Port

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

```
Port *port = ...;
PortAddress remote_port = ...;
RenderConnection *conn;

...
conn = port->connect(remote_port);
conn->render(...);
```

Connections

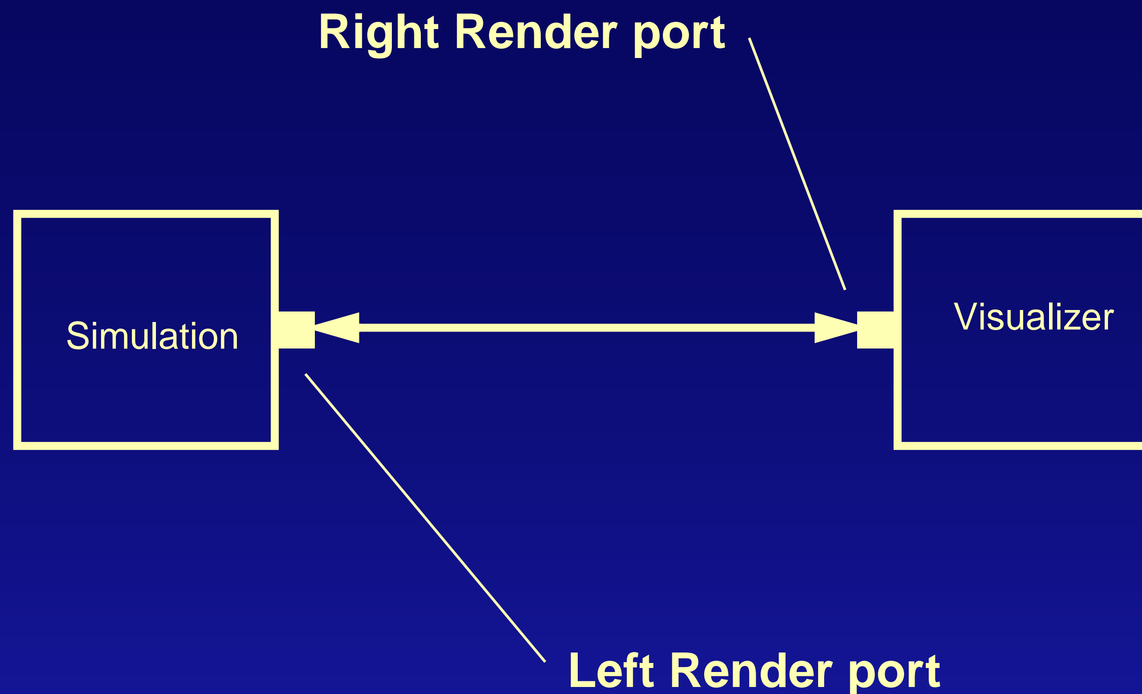
- Distributed computing is increasingly peer-to-peer.
- Peer-to-peer relationships often require invocations to each peer from the other.
- Usually requires two interfaces, four ports and two connections.



```
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;
}
```

Two-Way Connections

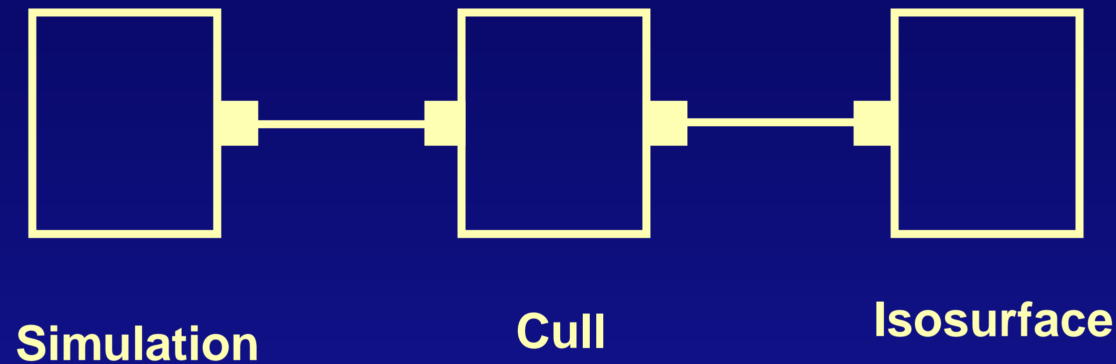
Easier to understand, maintain, and use if one interface is used to specify the contract.



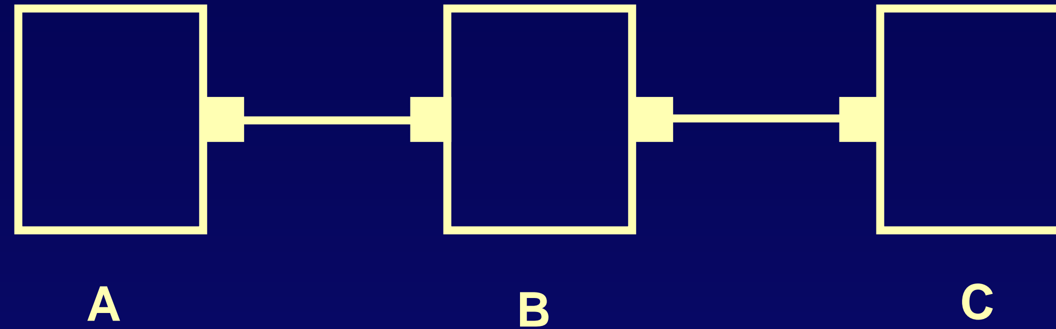
```
port Render
     render(...) right;
     set_decimation(...) left;
     partial_results(...) left;
};

component Sim {
     left port Render render;
}
component Vis {
     right port Render render;
}
```

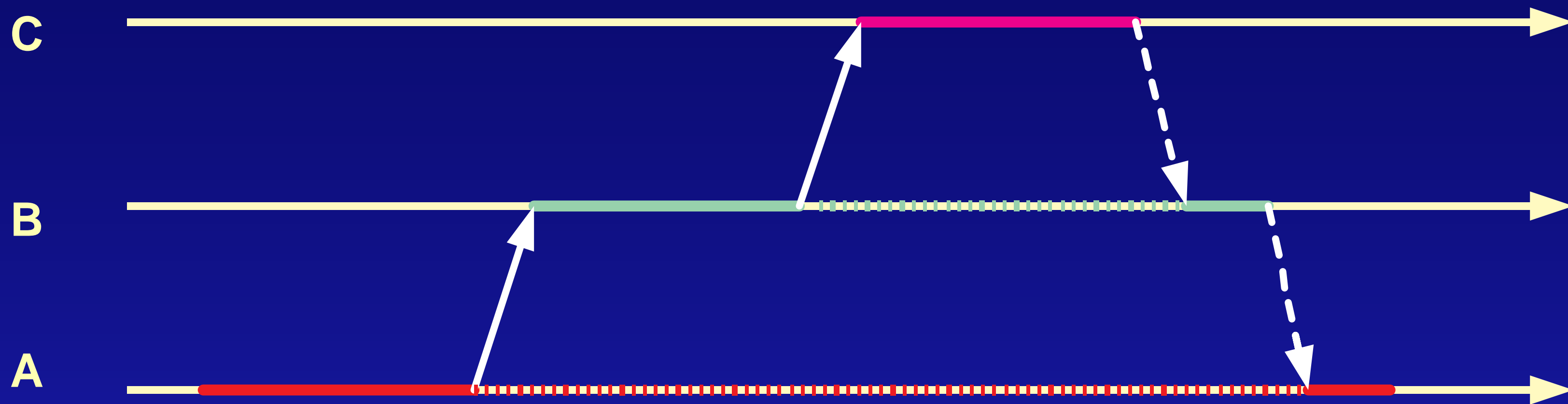
Concurrency

- Part of the motivation for components is pipelined concurrency.

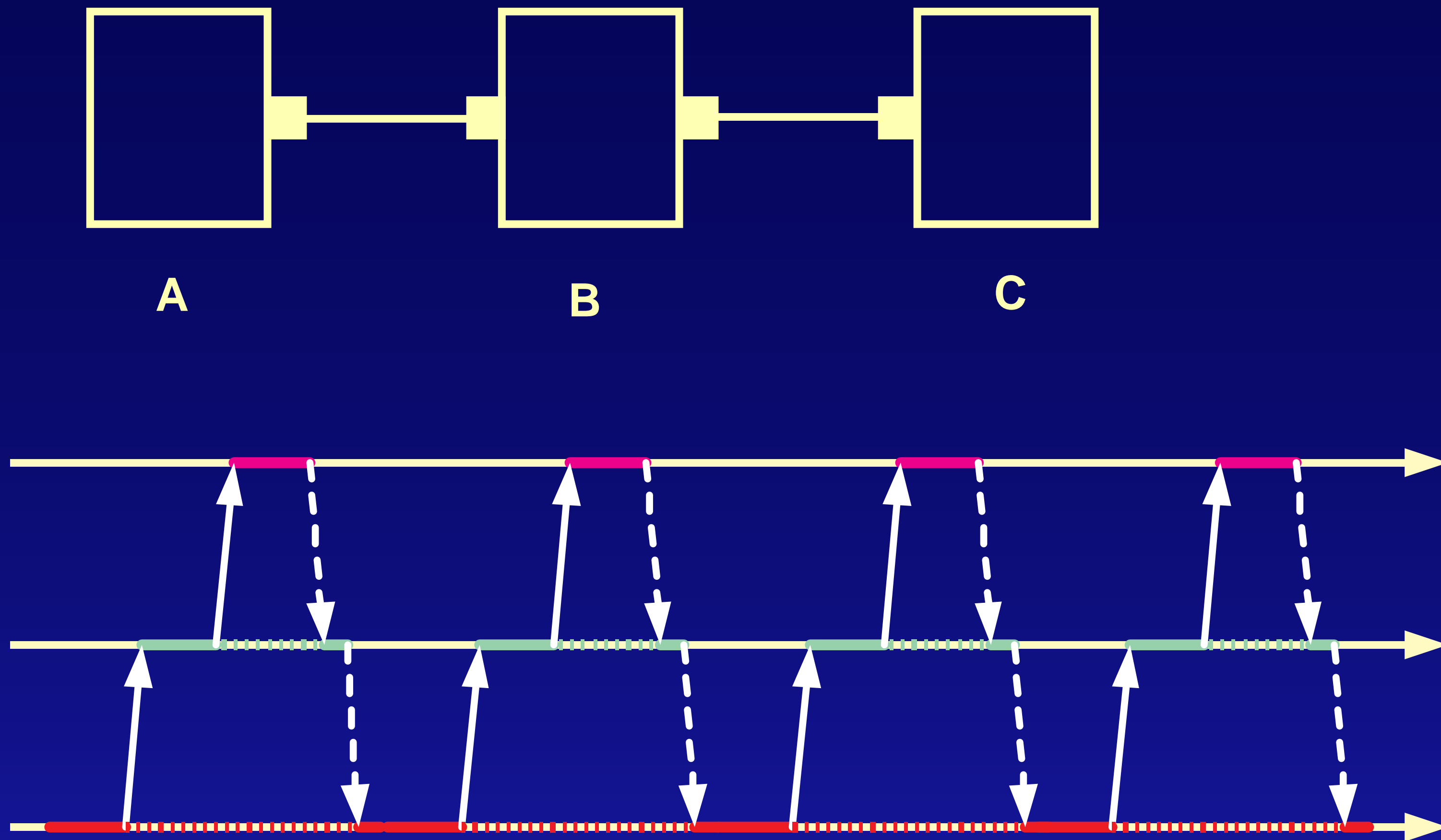


- Obtaining it is not always easy, however.

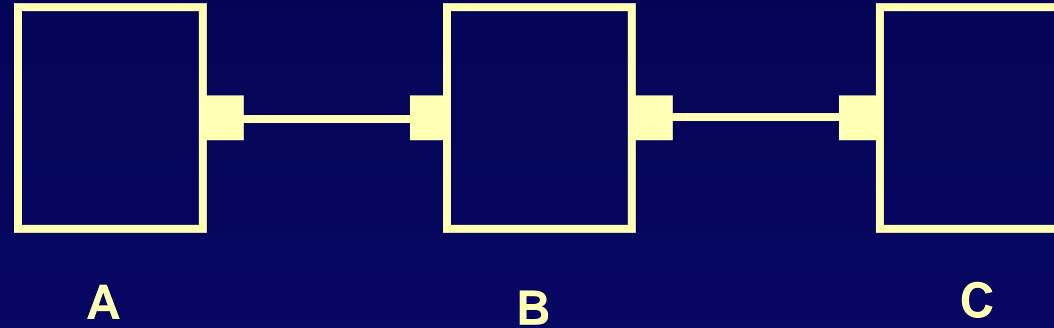


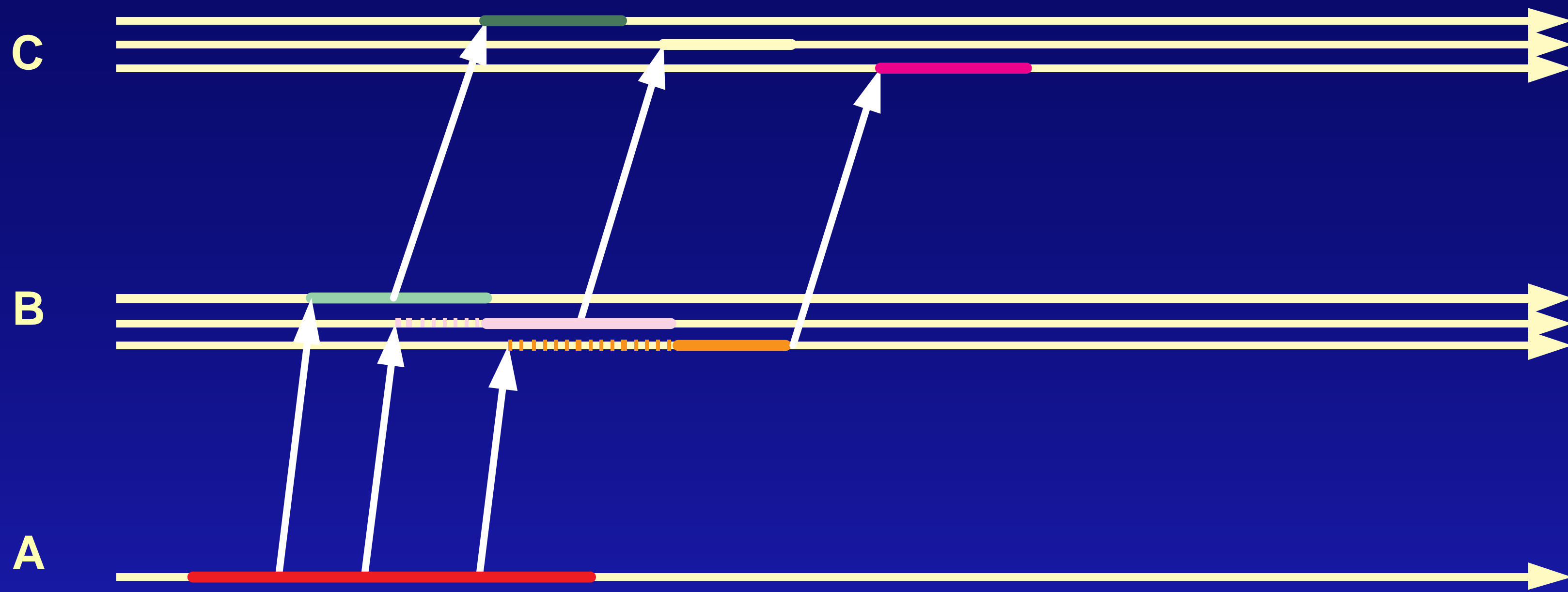


- So we try “chunking” the data.



- Asynchronous “chunking”





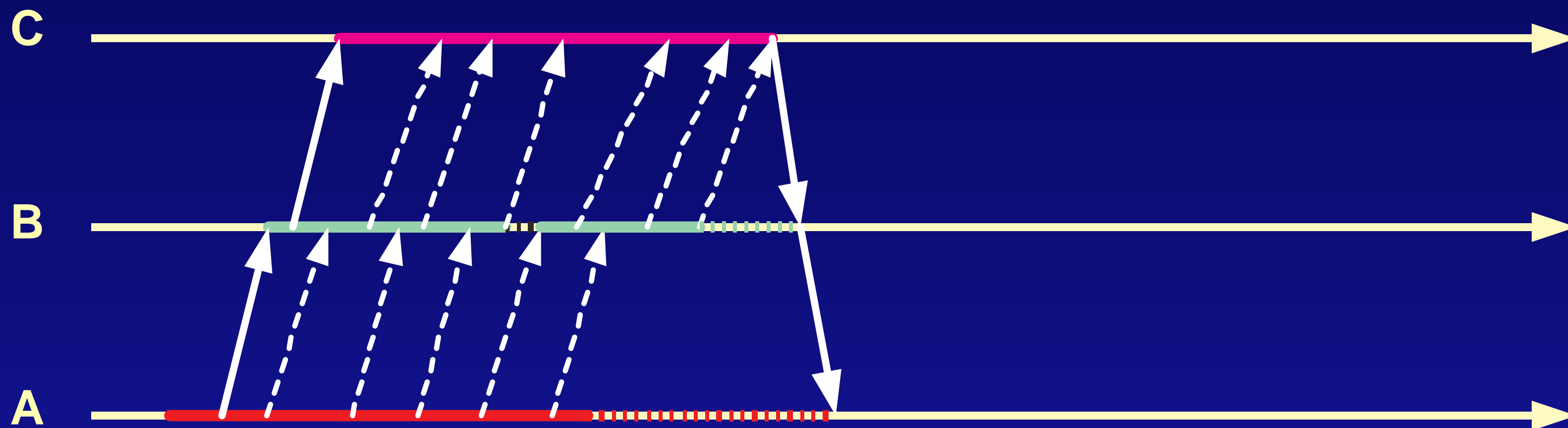
What Is Our Goal?

- 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.

Streams

- Borrows from message passing.
- Stream objects can be passed as arguments, and used to send data during the call.



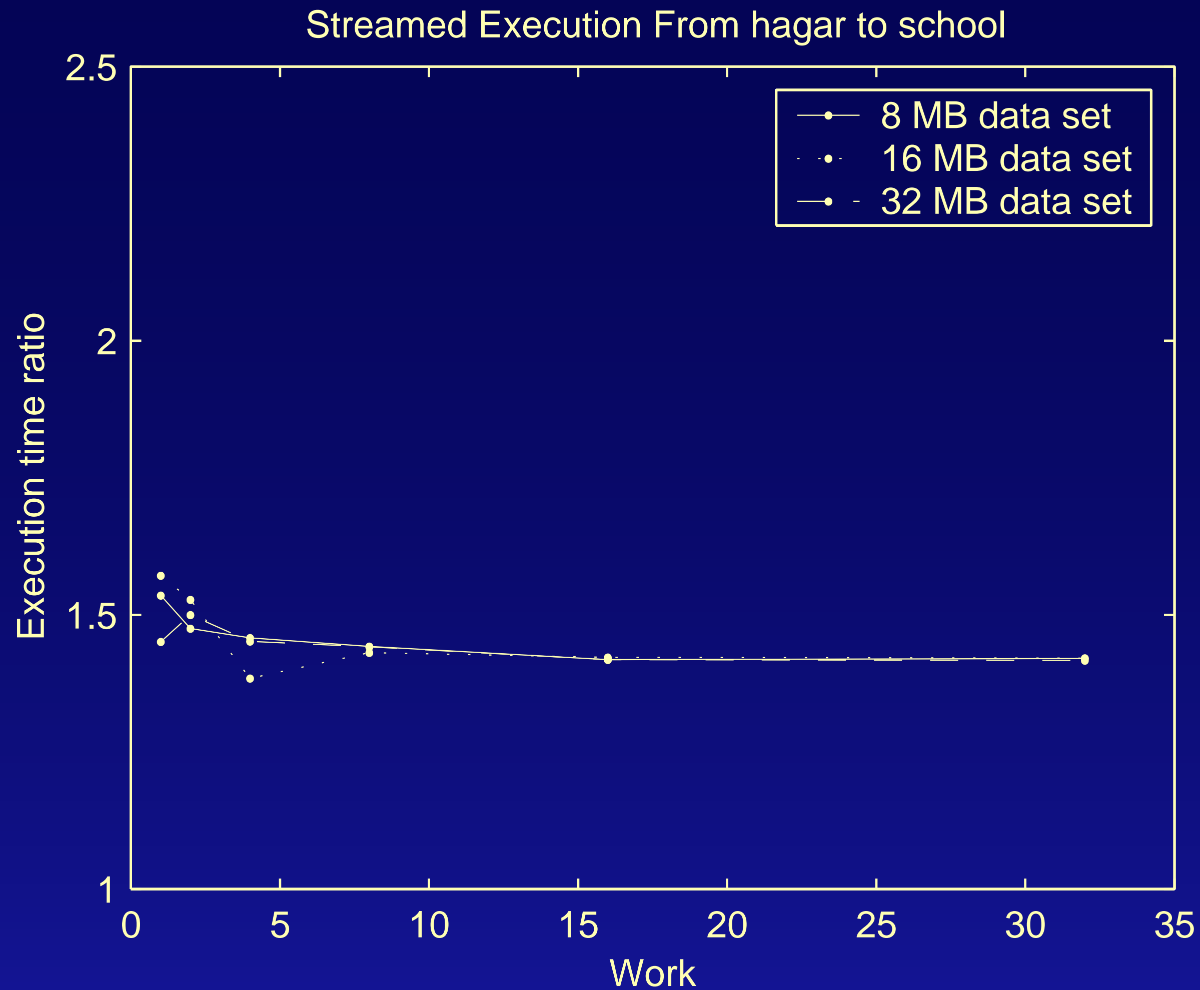
Caller

```
RenderConn *conn;
SendStream<double> *s;
Call<double> *call;
call = conn->render(&s);
    while (...) {
        s->write(...);
    }
flux = call->complete();
```

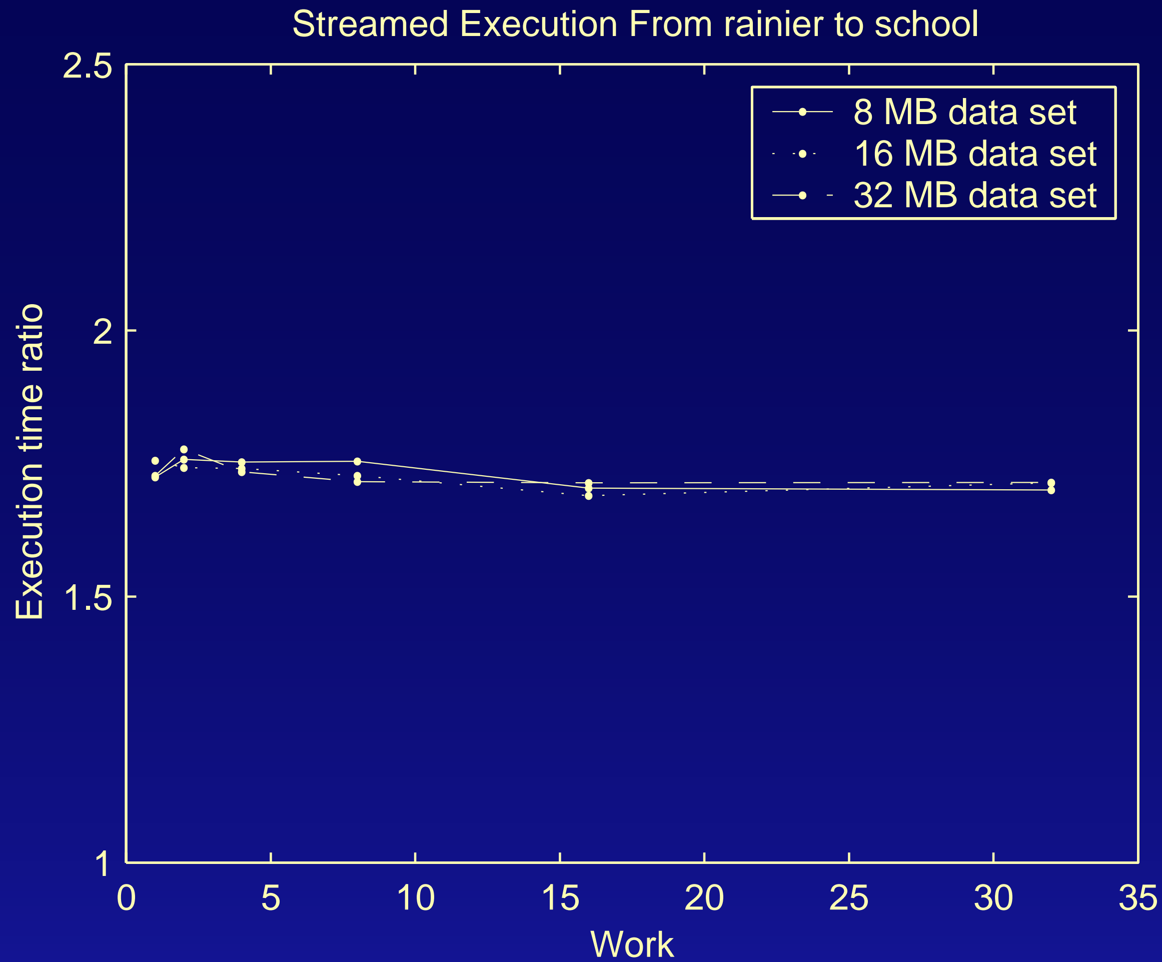
Callee

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

Performance Results



hagar is PII-400, Linux. school is UltraSPARC II-400, Solaris.



rainier is UltraSPARC II-295, school is UltraSPARC II-400

Limitations

- Collocation and streams.
- Streams cannot be returned.

  Allow to live past call completion.
- Does not eliminate the need for threads.

Future Work

- CCA-like component programming is relatively new.

  Fulfilling its promise involves social as well as technical factors.
- Investigate a more general message passing model.
- Extend specification to include broadcast/multicast semantics.

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 (lock-free)

  One process guaranteed to complete.
- Type-stable memory (TSM)

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

  An object has a consensus number N if it can be used to solve the consensus problem for at most N processes.

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 (∞)

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

- COMPARE&SWAP (∞)

  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 discontiguous 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
- Linearizable
- Assume TSM
- Assume double-word COMPARE&SWAP

Linearizability

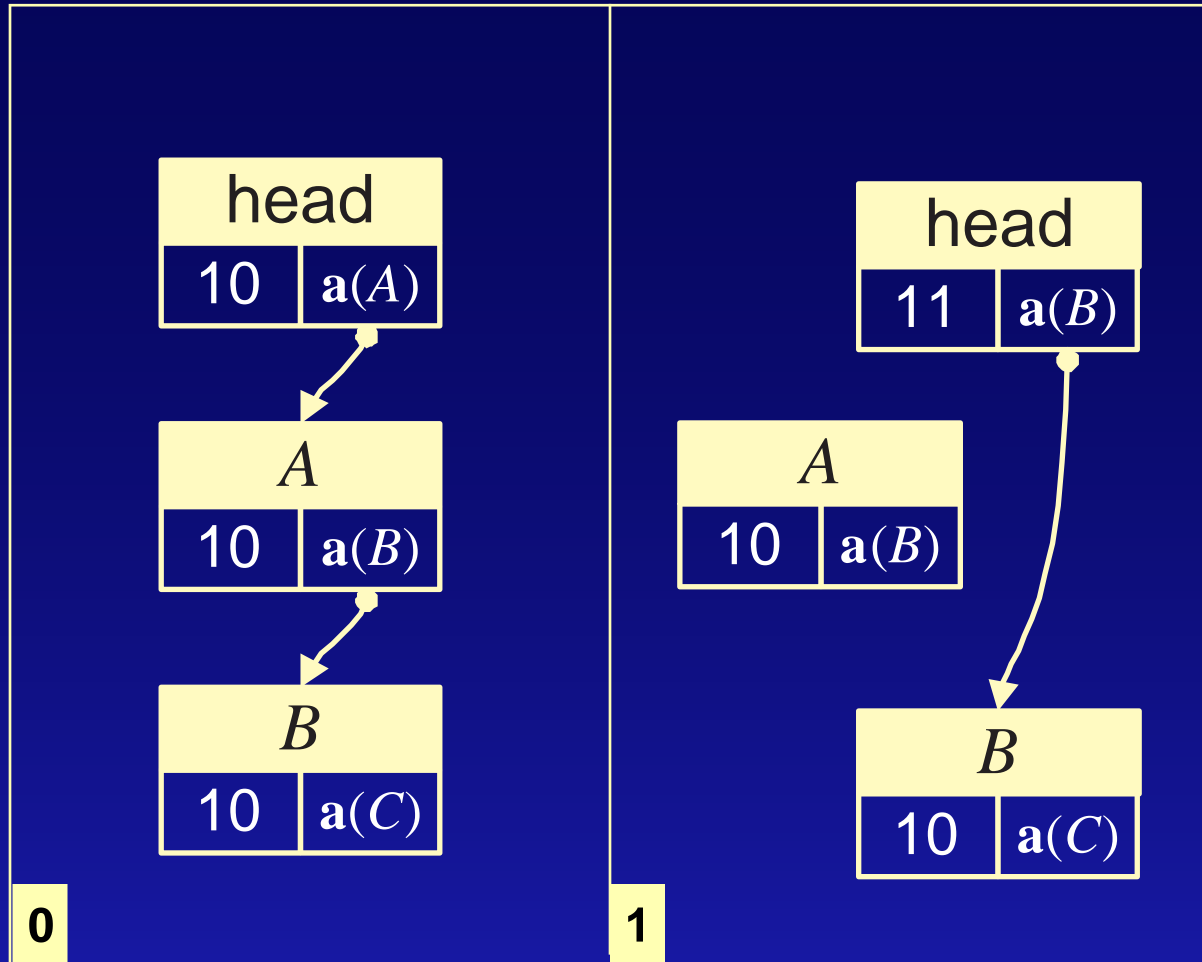
- An object is linearizable if it preserves ordering of non-concurrent operations, and executes concurrent operations as if they were in some sequential order.

Not linearizable	Linearizable
A: Push(1)	A: Push(1)
B: Push(2)	B: Push(2)
B: Ok	B: Ok
A: Ok	A: Ok
C: Push(3)/Ok	C: Push(3)/Ok
D: Pop/Ok(1)	D: Pop/Ok(3)
D: Pop/Ok(3)	D: Pop/Ok(1)
D: Pop/Ok(2)	D: Pop/Ok(2)

- Contrast with sequential consistency.

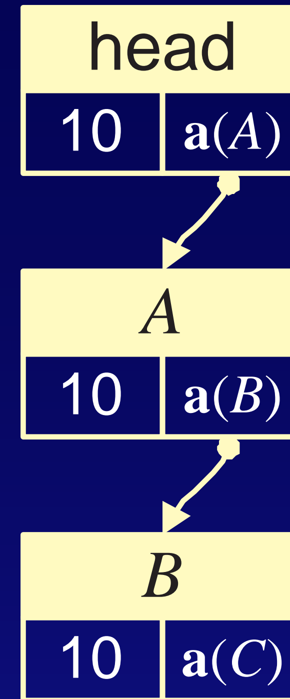
Example of NBS

Popping an element from a stack.

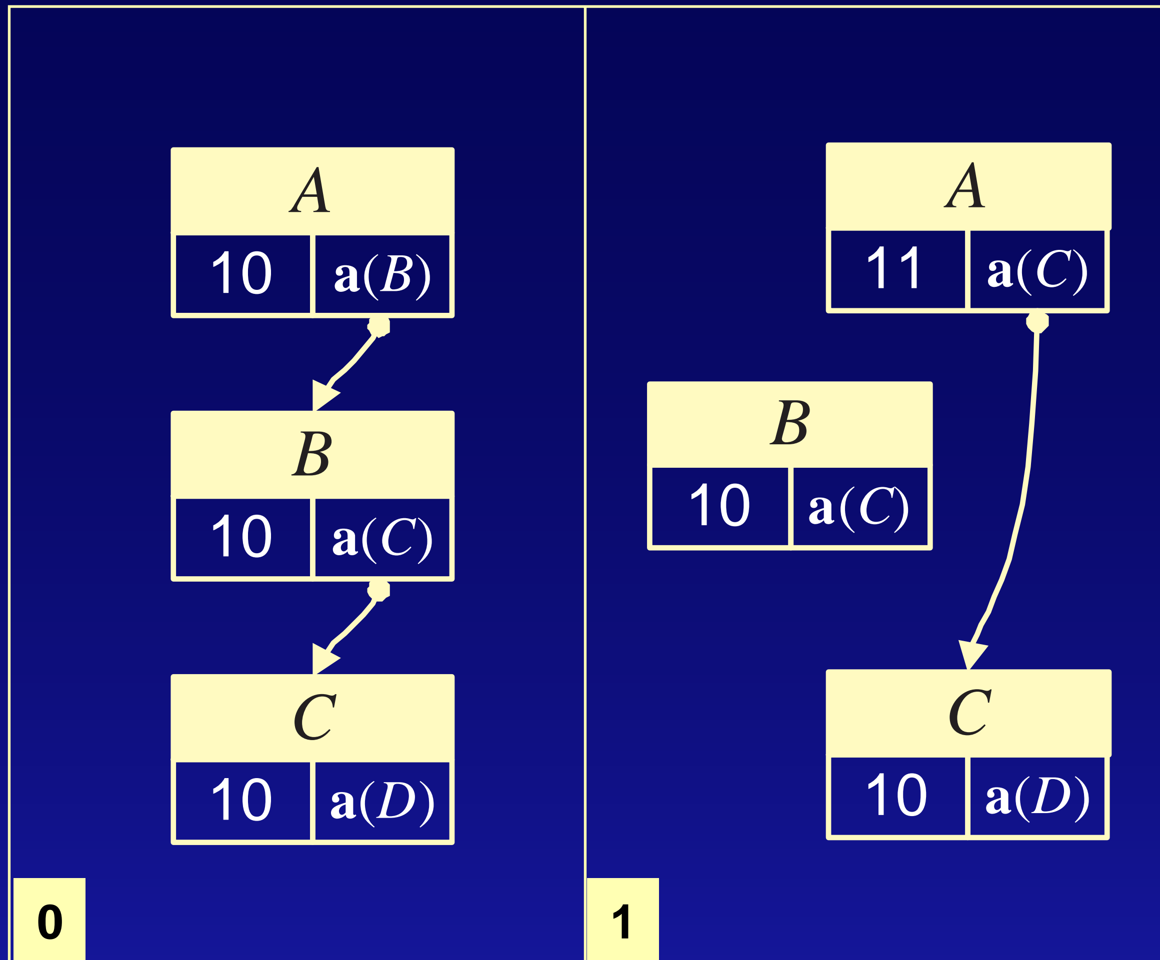


$\mathbf{a}(E)$ = address of element E

```
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));
```

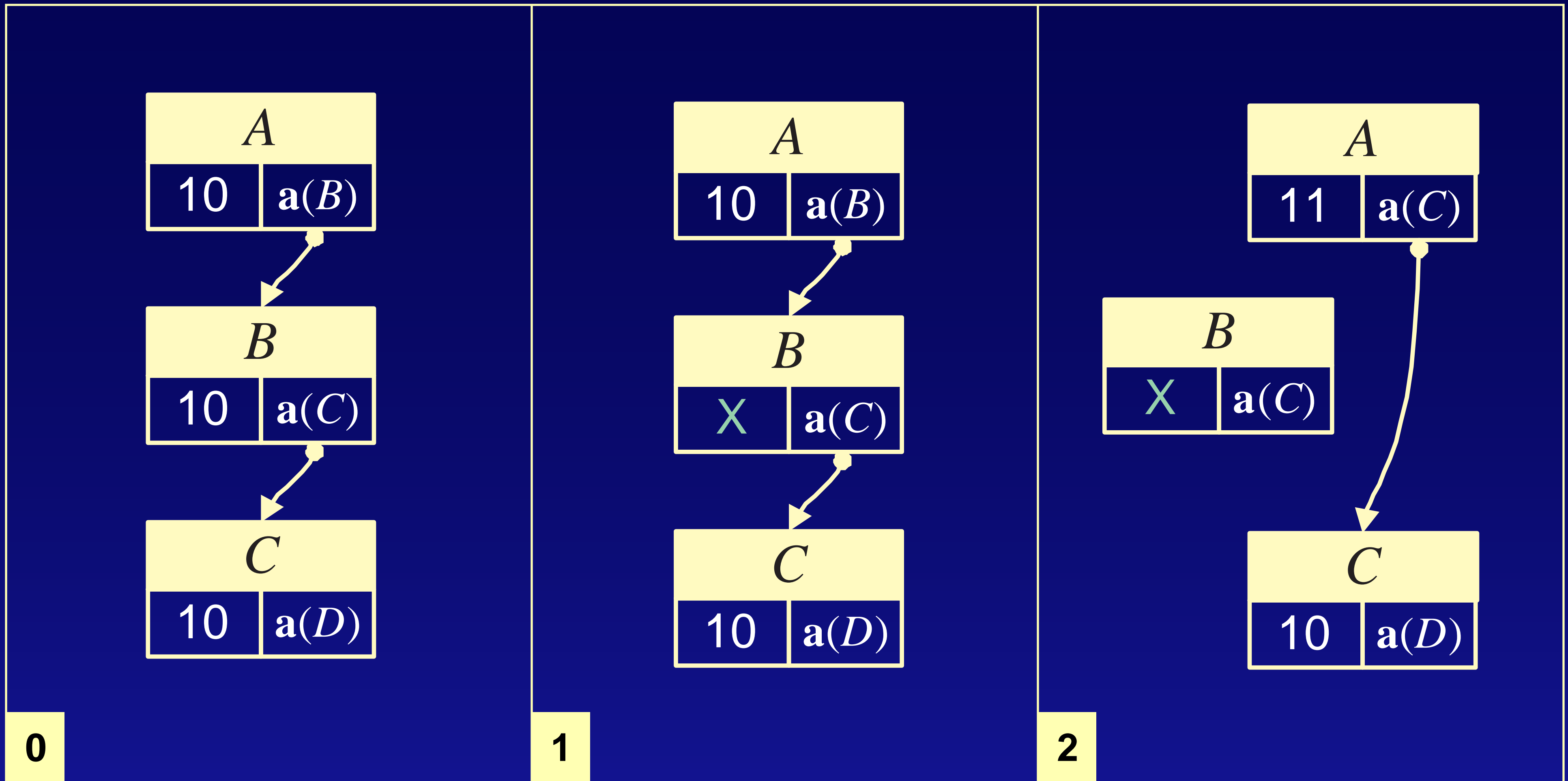


Does This Work for Lists?



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

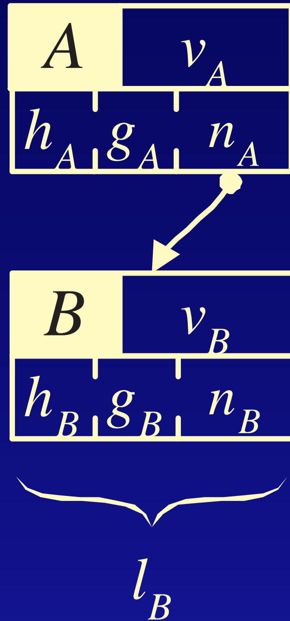
Marking



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.



$$g_E = \text{generation of element } E$$

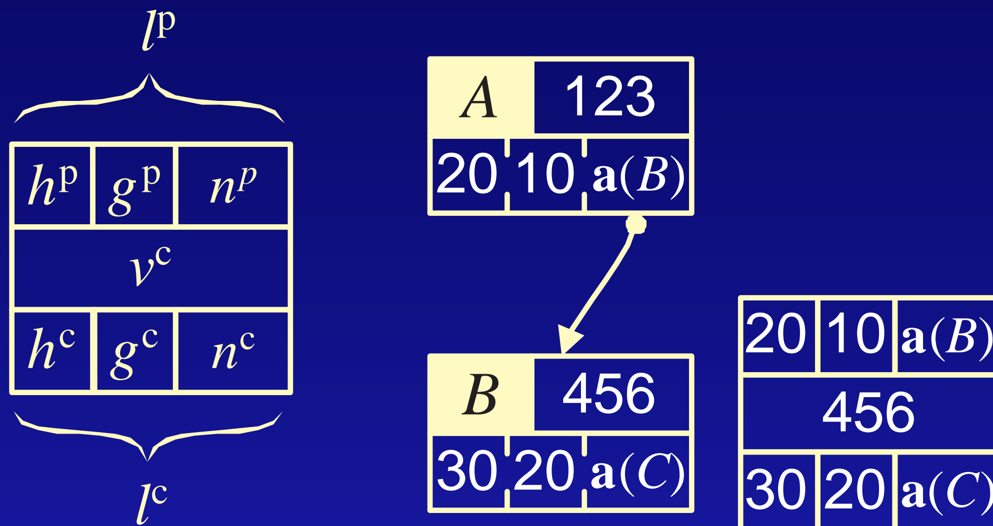
$$h_E = \text{generation of successor to } E$$

$$n_E = \text{pointer to successor}$$

$$l_E = (g_E, h_E, n_E)$$

Cursor

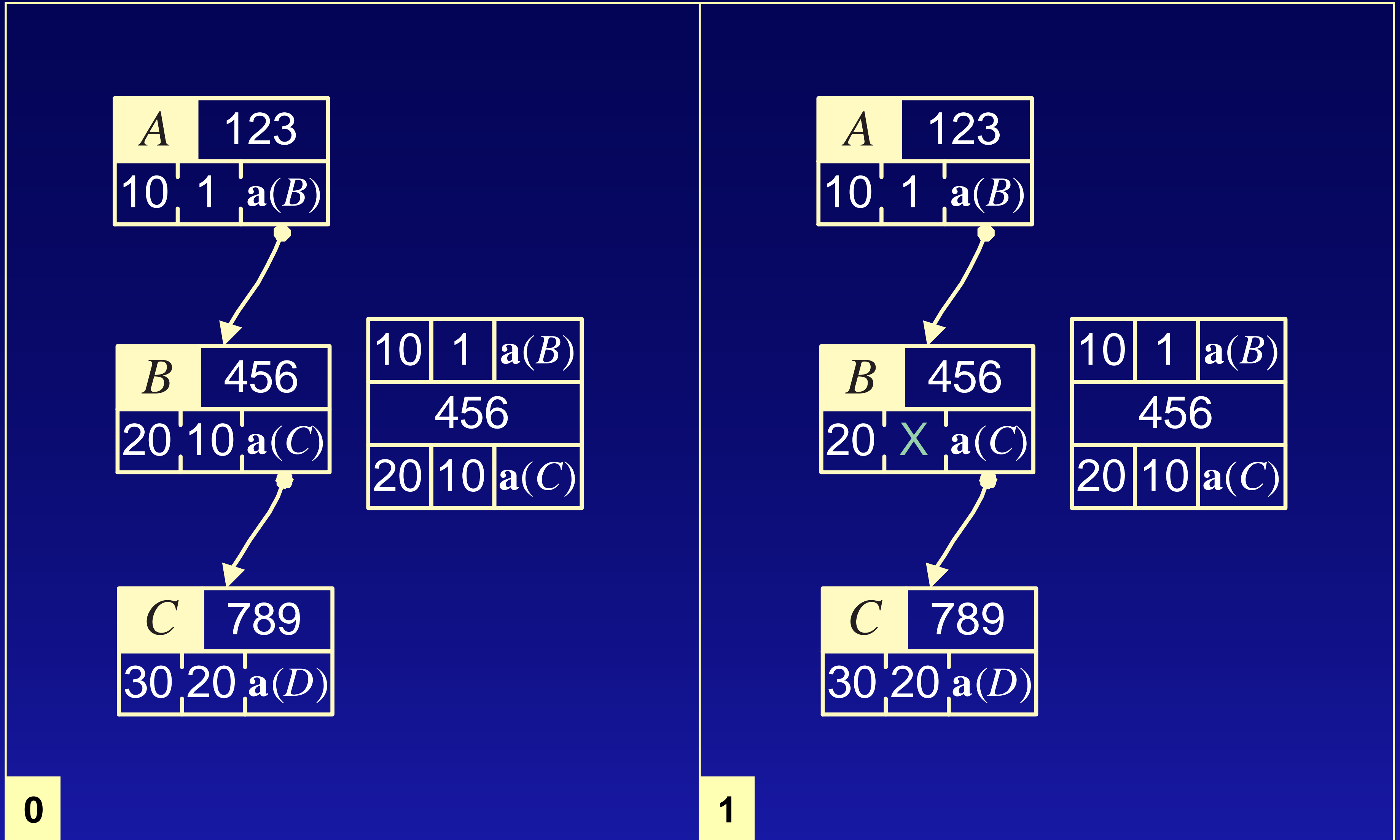
- Represents a snapshot in “time.”
- Contains two links and the value of the visited element.
- Aids detection of conflicts.

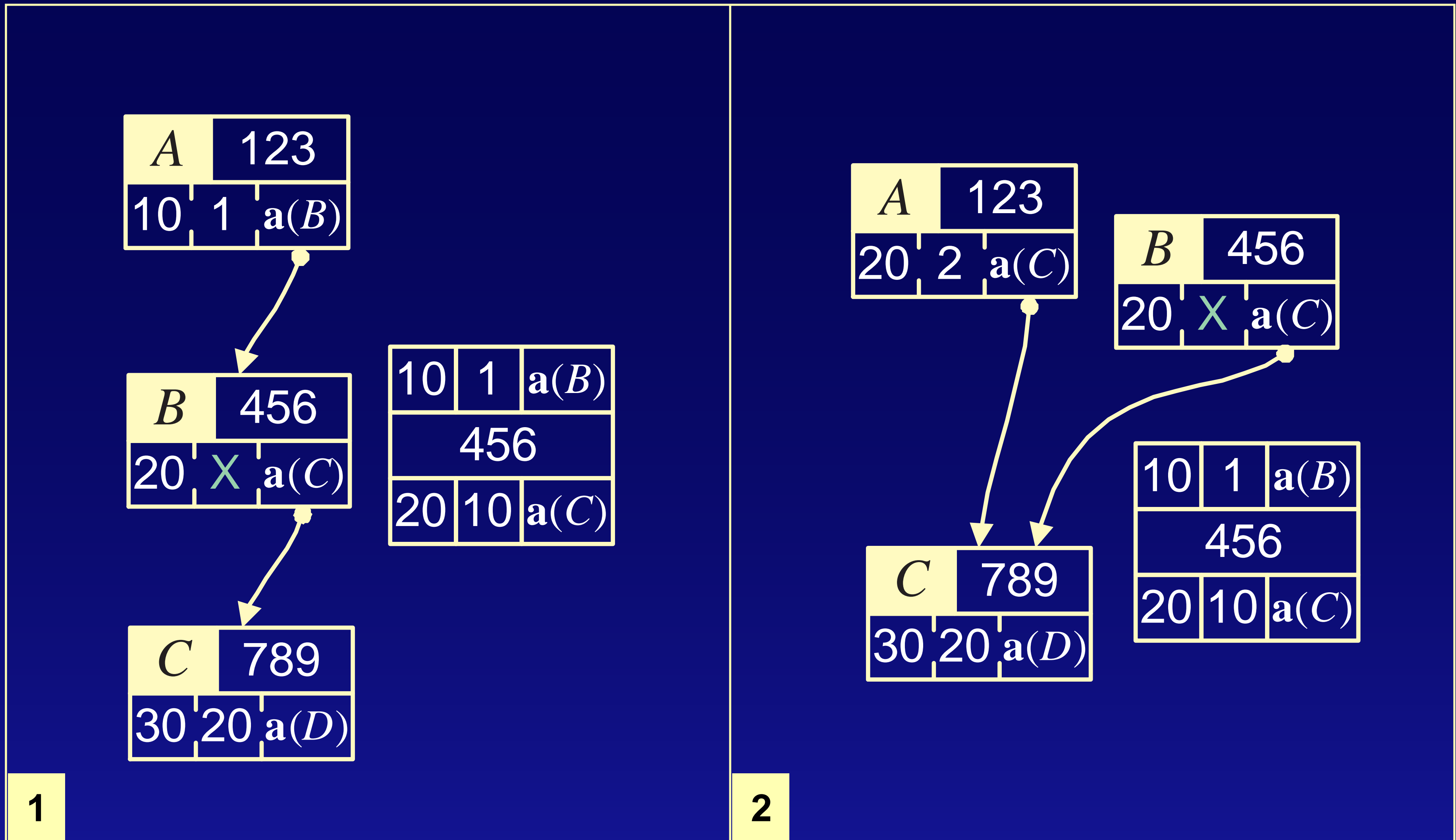


If the cursor is visiting B, then normally

$$l^p = l_A$$
$$l^c = l_B$$
$$v^c = v_B$$

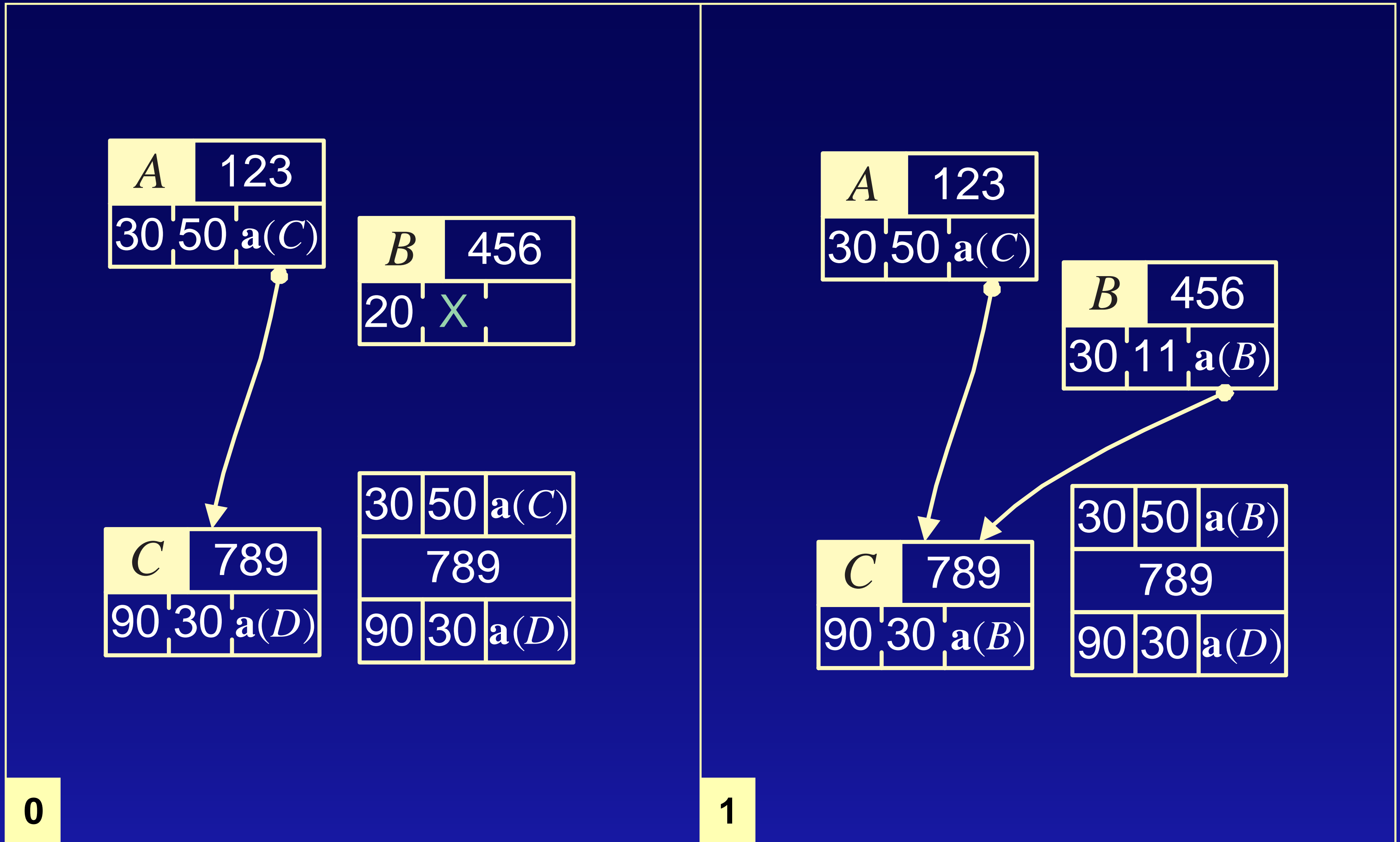
Removal

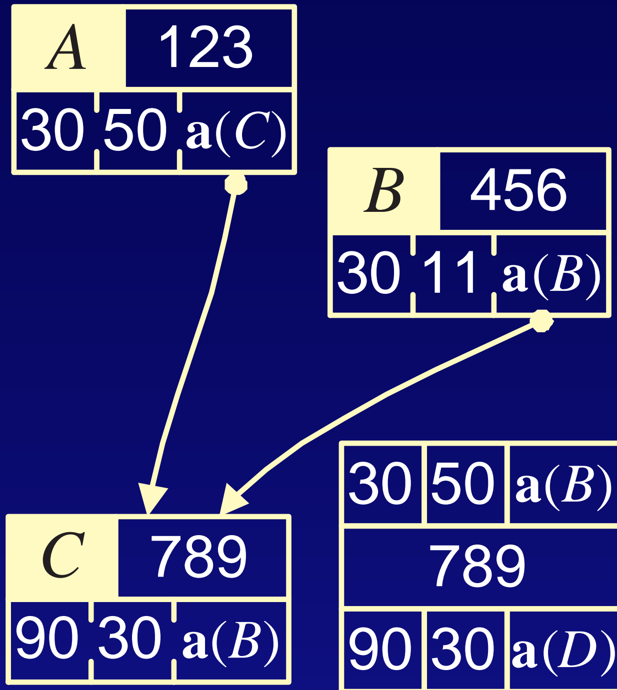


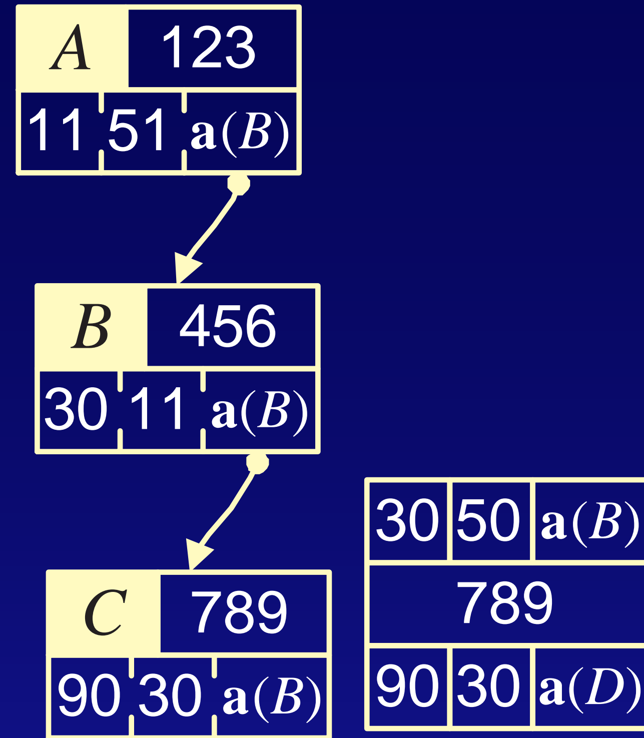


If a conflict is detected, must restart from head of list. Minor conflicts can be fixed-up with restarting.

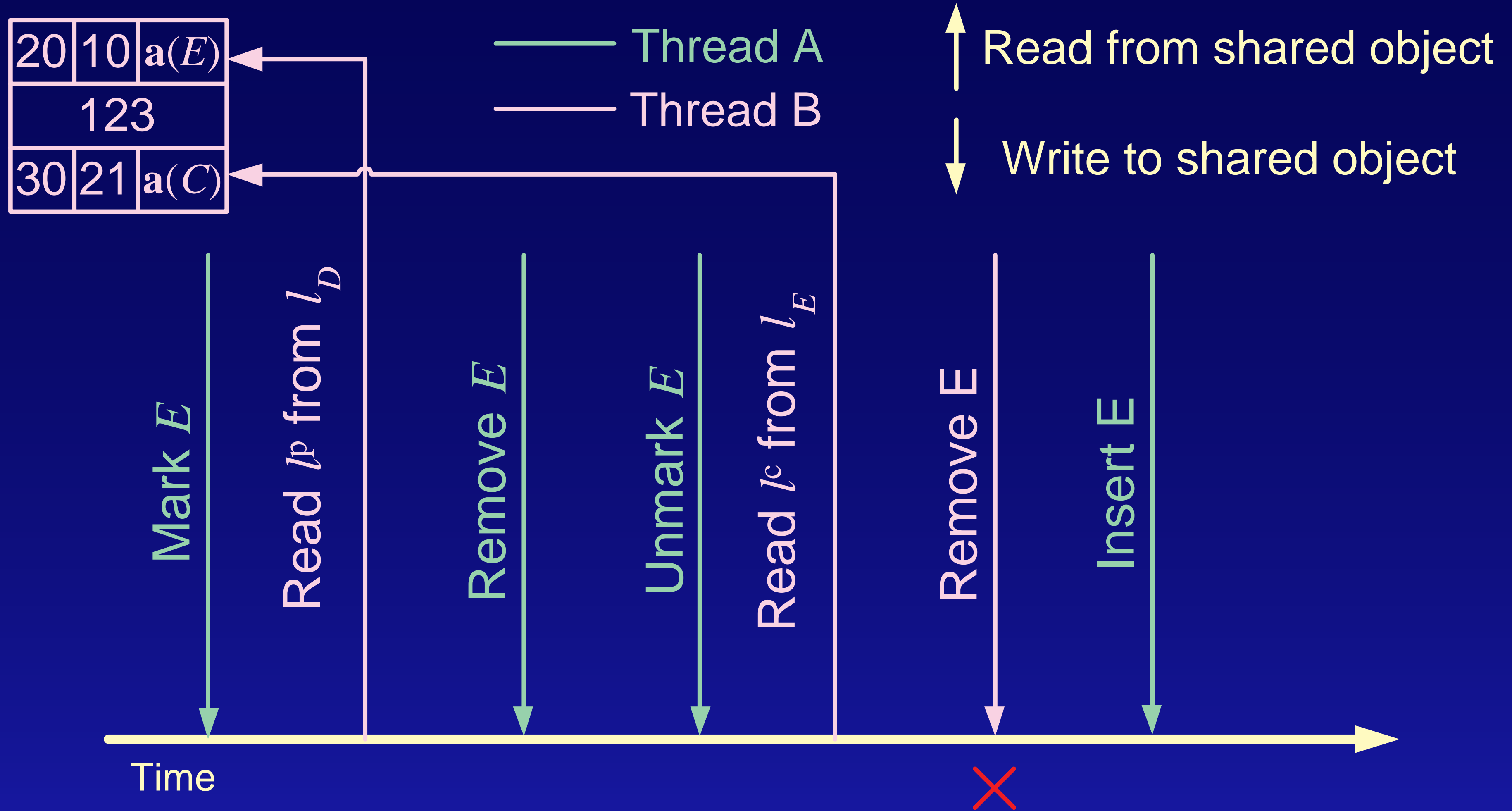
Insertion




A
123
30
50
a(C)
B
456
30
11
a(B)
C
789
90
30
a(B)
30
50
a(B)
789
90
30
a(D)
1


A
123
11
51
a(B)
B
456
30
11
a(B)
C
789
90
30
a(B)
30
50
a(B)
789
90
30
a(D)
2

Purpose of h



Algorithm Outline

- Ordered insert

```
Element *e = ...;

RESTART:

    Cursor c = begin();

    while (!proper_position()) {
        if (!c.advance()) {
            goto RESTART;
        }
    }

    if (!c.insert(e)) {
        goto RESTART;
    }
```

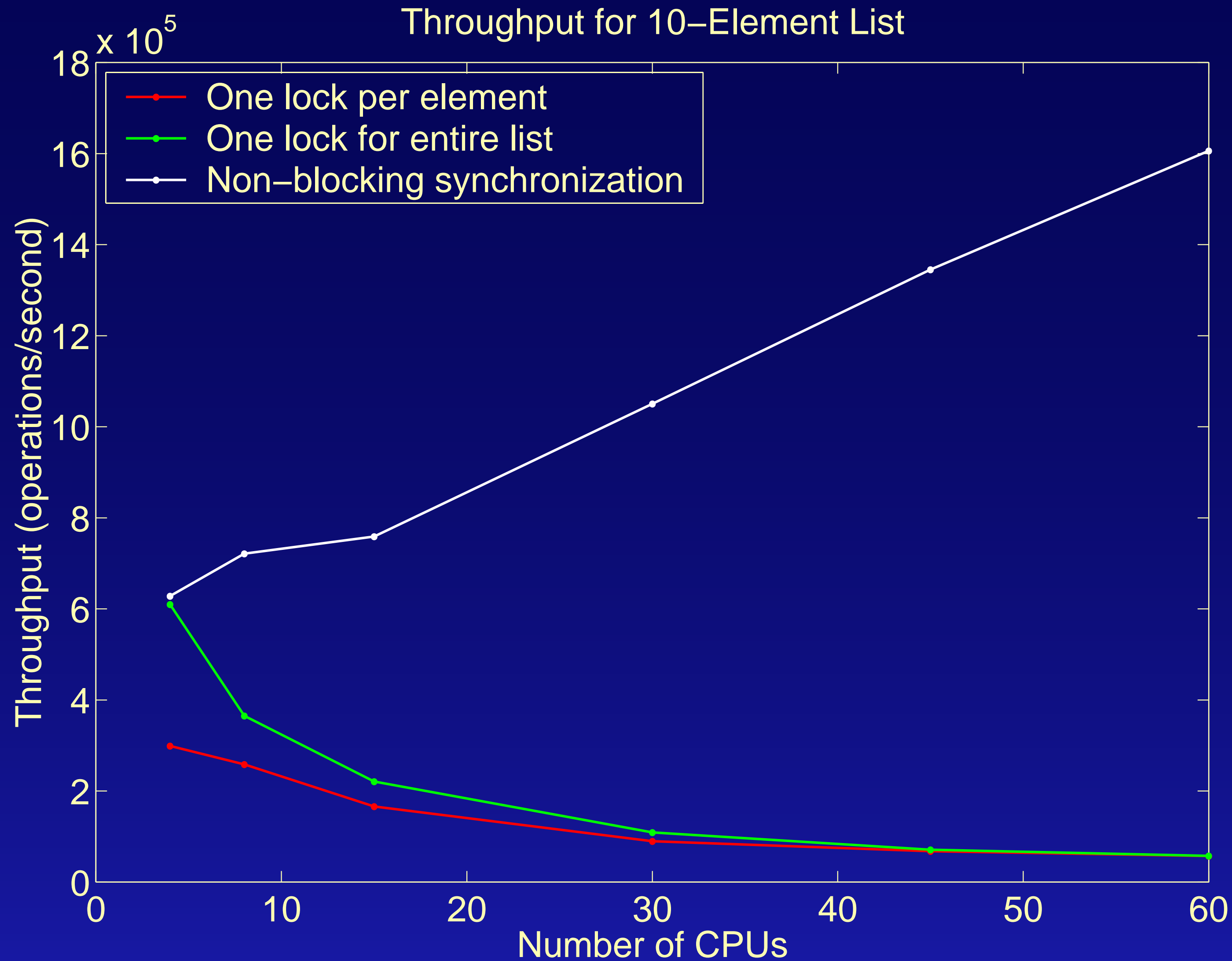
Performance

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

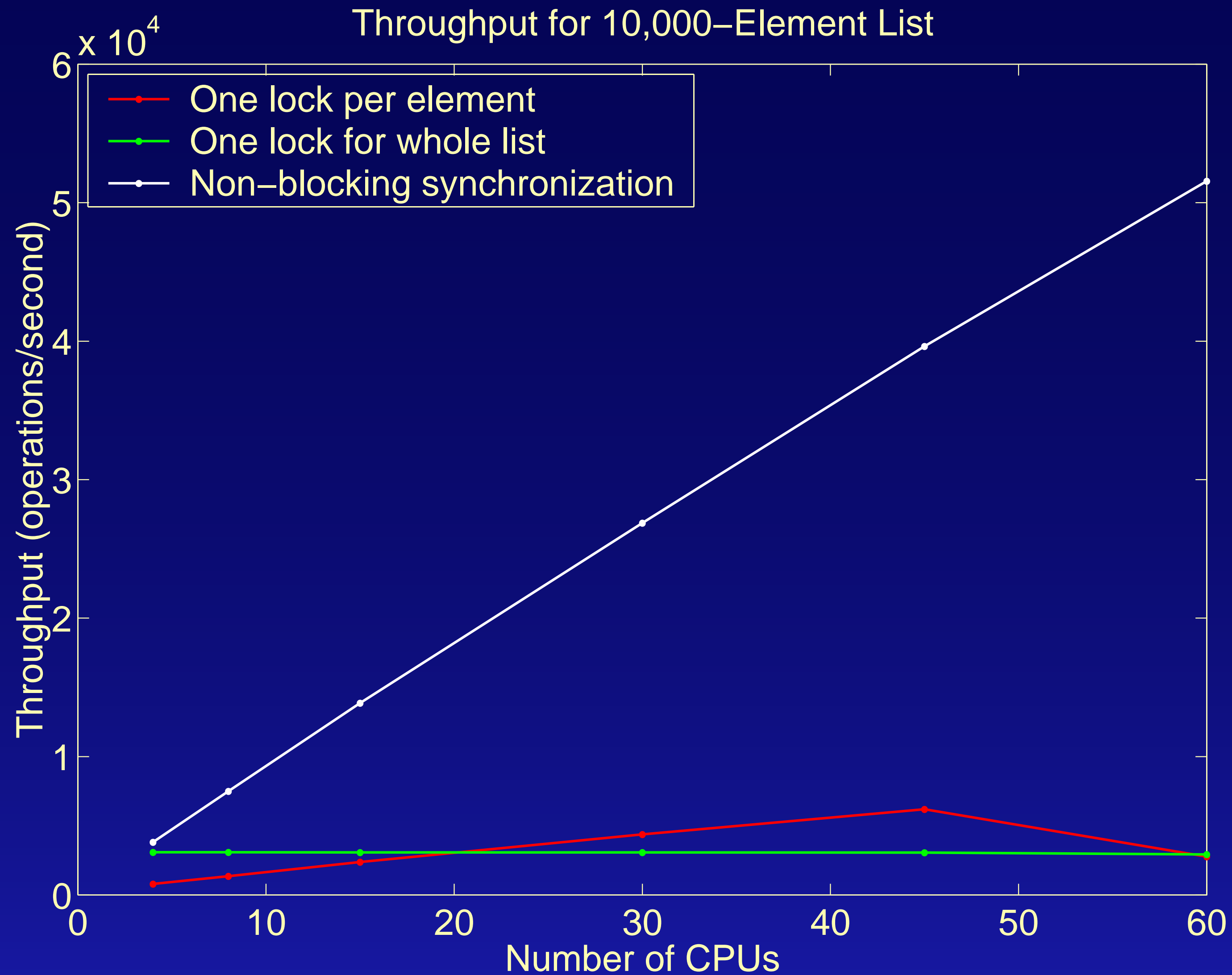
```
while (true) {
    r = random();
    Element *e = list.remove(r);
    if (e != NULL) {
        e->value = random();
        list.insertSorted(e);
    }
}
```

- 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.

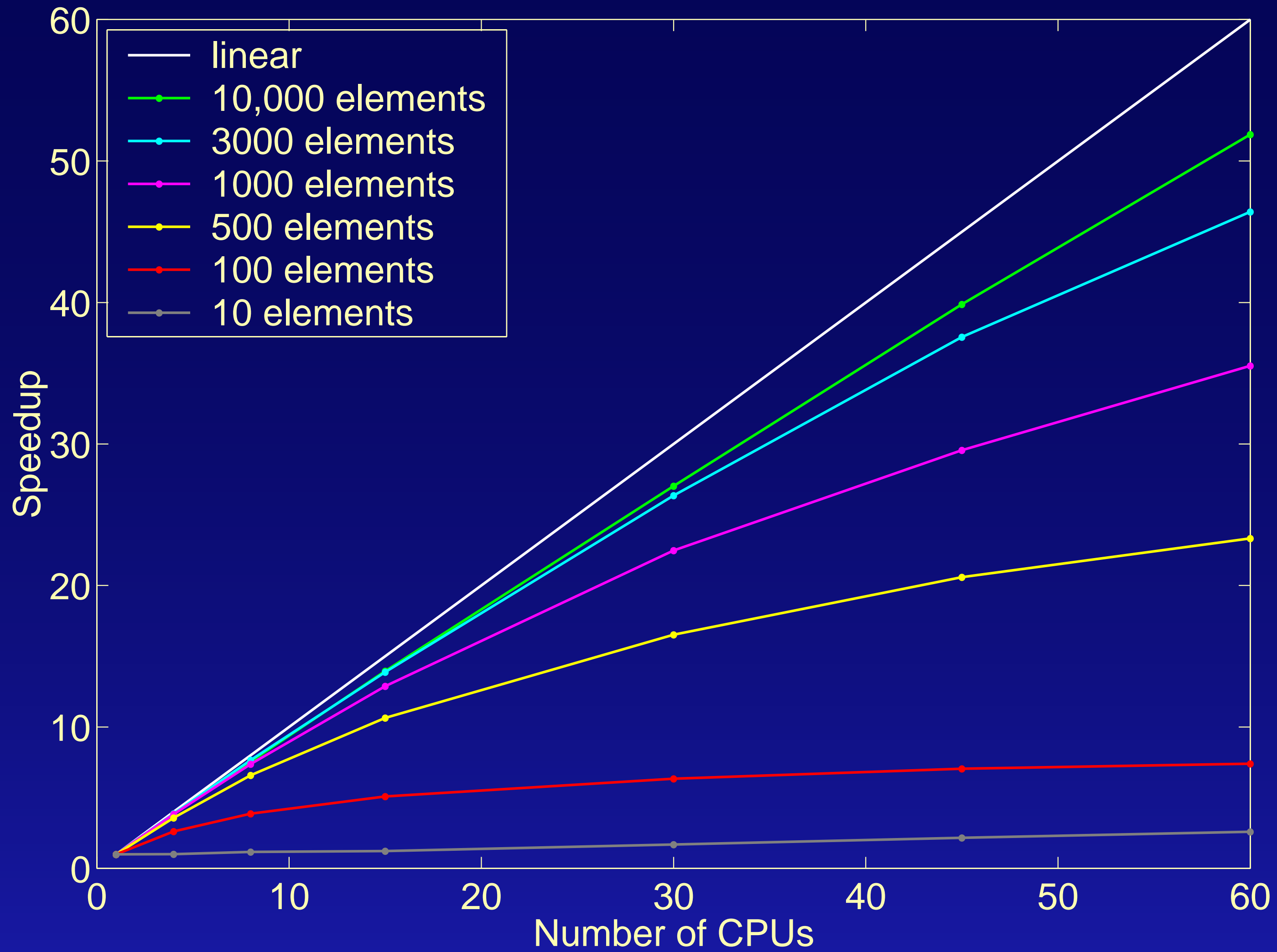
10 Elements



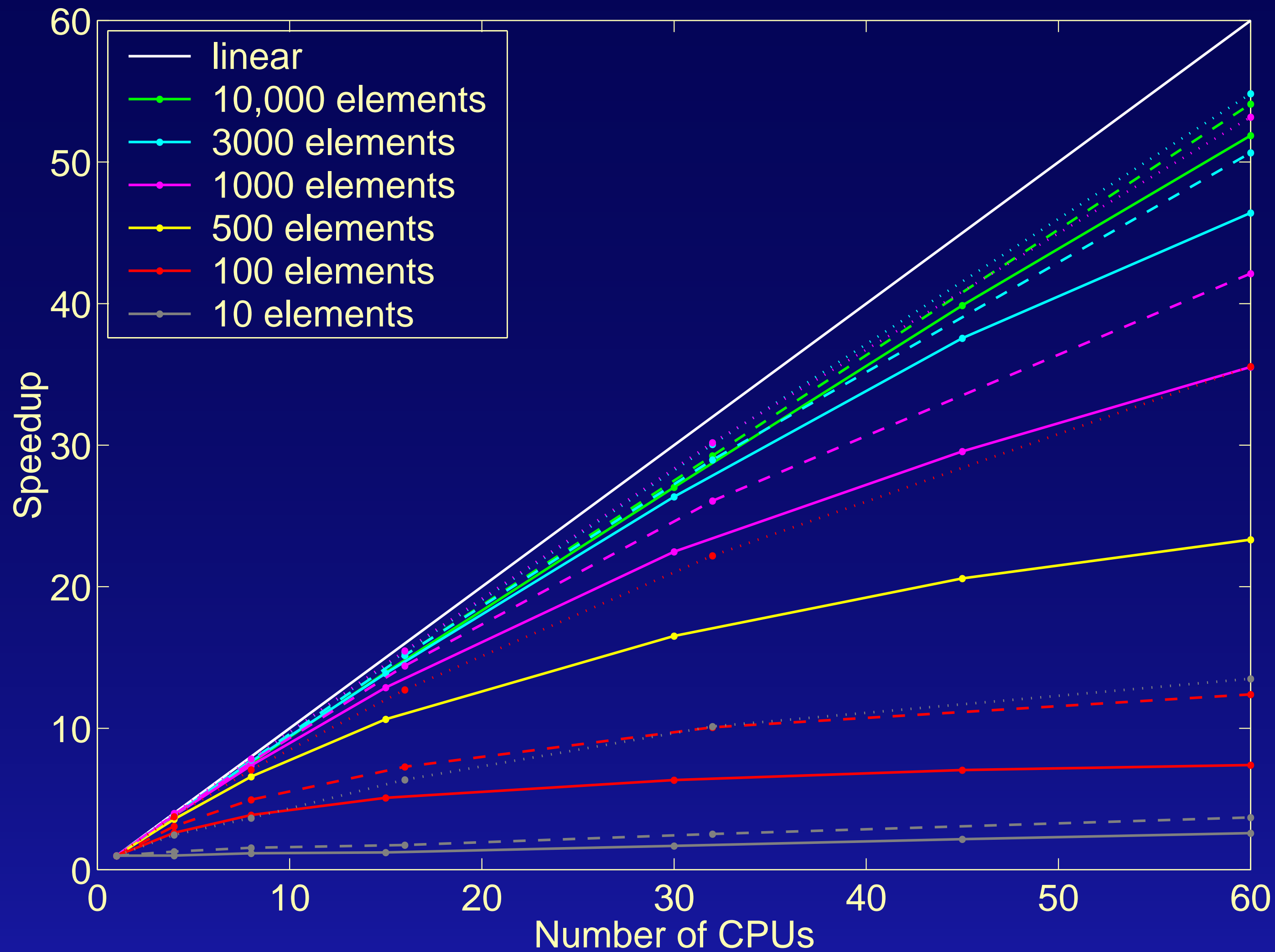
10,000 Elements



Speedup



Expensive Compares



To Do

- Formal verification

  A number of incorrect algorithms have been published. Would like to use machine-assisted.

- Performance enhancements

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

- Relaxed memory models

  Will require memory barriers, acquire/release. Verification more difficult than sequential.

- Investigate performance on NUMA architectures

Scratch Slides

Model Checking

Traditional model checking does not seem well-suited.

- Can only check finite instances.
- Difficult to express safety properties as temporal logic expression over state. TLA (Lamport) might be more expressive.

Connecting Two Ports

- One-way ports

```
// Create new unique ports.
my_pp = new MyPPort(..);
MyPPortInfo my_ppi("my_pport");
svcs->addProvidesPort(my_pp, &my_ppi);
MyUPortInfo my_upi(...);
svcs->addUsesPort(&my_upi);
// Get remote unique ports.
conn_svc->connect(vis, "my_reg", sim, "sim_reg");
my_reg = getPort("my_reg");
(sim_uport, sim_pport) = my_reg->getUniquePorts();
svcs->releasePort("my_reg");
// Connect new ports.
conn_svc->connect(me, "my_uport", sim, sim_pport);
conn_svc->connect(sim, sim_uport, me, "my_pport");
my_uport = getPort("my_uport");
my_uport->method(...);
```

- Two-way connections

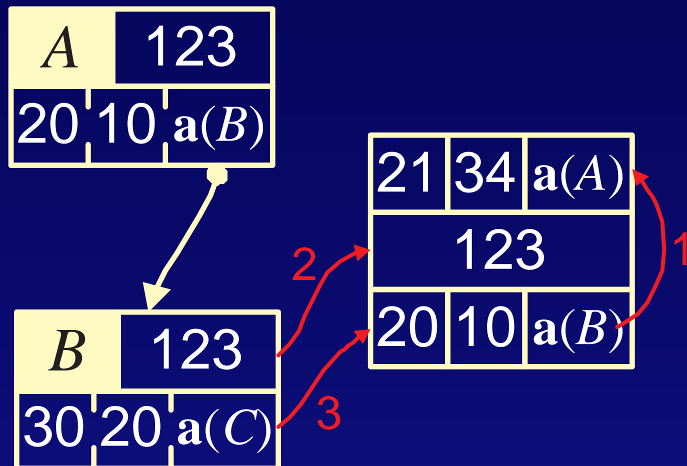
```
Port *sim1 = ...; PortAddress vis1 = ...;
sim1->connect(vis1);
```

Consensus Hierarchy

- Some primitives are more powerful than others.
- An object with consenus number N can be used to implement any wait-free object for N or few processes.
- The consensus number defines a robust hierarchy.

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	$\mathbf{a}(B)$
123		
30	X	$\mathbf{a}(C)$

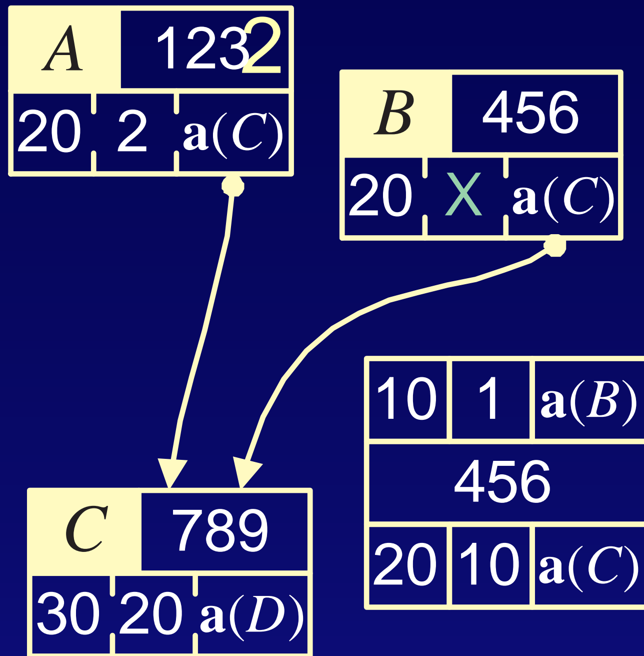
If the current element has been marked, then we need to remove it before continuing. If, upon trying to do so, we detect that gi-1 != gp, then this indicates the current element may have been removed and reinserted. This may cause portions of the list to be skipped, so a restart is necessary. If only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

Another condition is that h may be out-of-date. That is, hp != 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.

<table>
<tr><td>20</td><td>10</td><td>$\mathbf{a}(B)$</td></tr>
<tr><td colspan="3">123</td></tr>
<tr><td>30</td><td>20</td><td>$\mathbf{a}(C)$</td></tr>
</table>

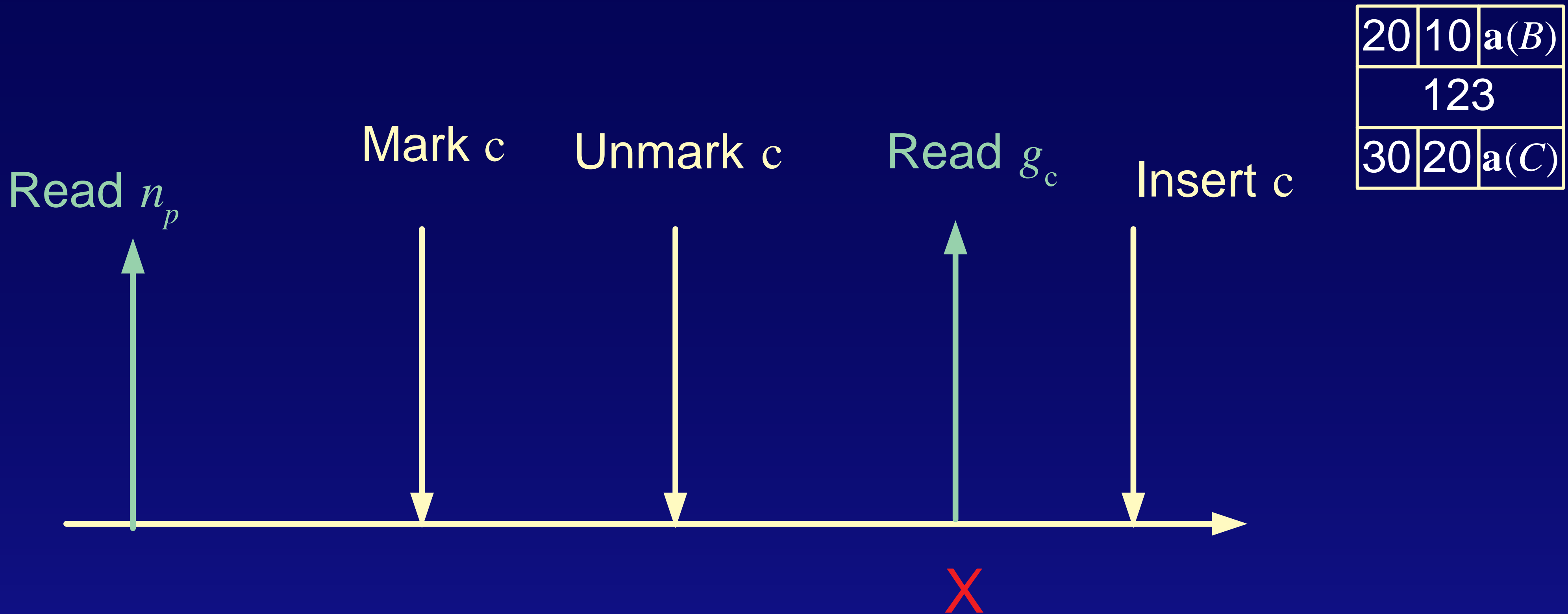


If the COMPARE&SWAP fails, it might be for a number of reasons. Must retraverse the whole list. If we don't find this element again, then we can be sure it has been removed properly.

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	$\mathbf{a}(B)$
123		
30	20	$\mathbf{a}(C)$