Concepts and Models of Parallel and Data-centric Programming

Shared Memory XI

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. **Shared Memory**
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

l. Coarse-grained Synchronization
m. Fine-grained Synchronization
n. Optimistic Synchronization
o. Lazy Synchronization
p. Lock-free Synchronization

Lazy Synchronization

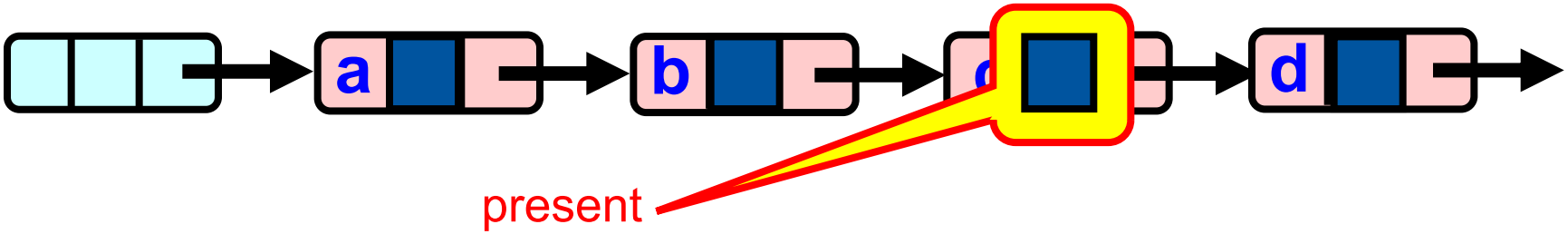
Lazy Synchronization / 1

- Strategy: postpone hard work
  - Approach is very similar to optimistic one, except
    - Scan once
    - `contains(x)` never locks …
- Key insight
  - Removing nodes causes trouble, hence do it “lazily”
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Finally do what needs to be done

Lazy Synchronization / 2

- `remove()`
  - Scans list (as before), locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

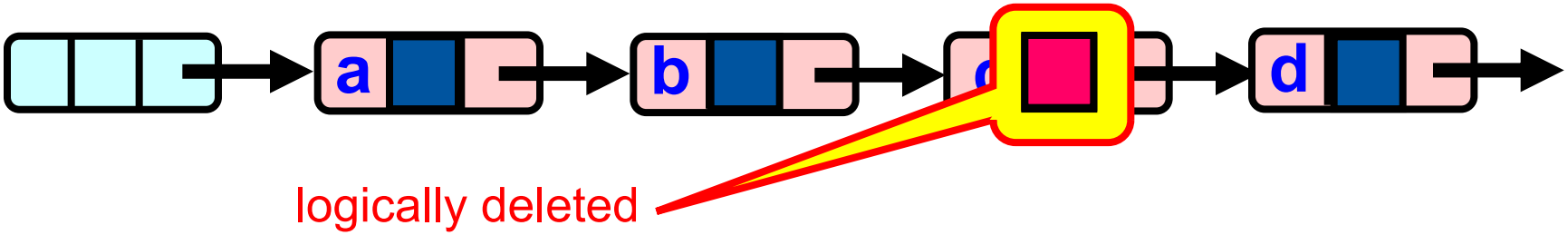
- Illustration:



Lazy Synchronization / 2

- `remove()`
  - Scans list (as before), locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

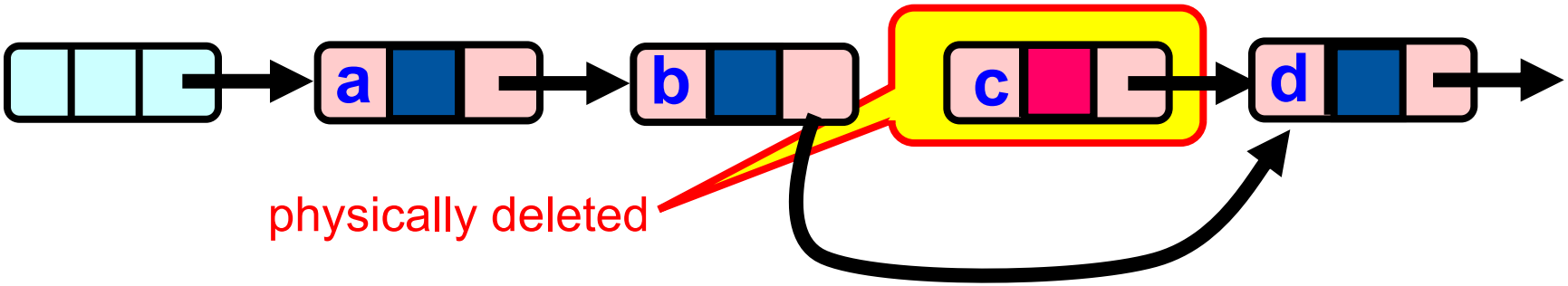
- Illustration:



Lazy Synchronization / 2

- `remove()`
  - Scans list (as before), locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

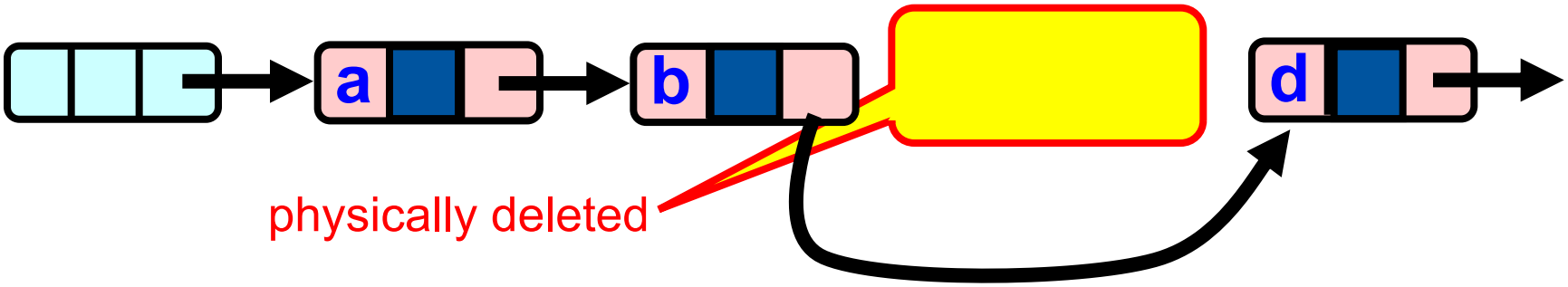
- Illustration:



Lazy Synchronization / 2

- `remove()`
  - Scans list (as before), locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

- Illustration:



Implementation sketch

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
  - Must still lock pred and curr nodes
- Validation method (no rescan)
  - Check that pred and curr are not marked and that pred points to curr
- Extended correctness criterium:
  - S(head) = { x | there exists node a such that
    - a reachable from head and
    - a.item = x and
    - a is unmarked
      }

Illustration: validation

```
bool validate(Node *pred, Node *curr)
{
   return !pred->marked && !curr->marked &&
          pred->next->key == curr->key);
}
```

Illustration: contains

```
bool contains(Item *item)
{
   Node *curr = this->head;
   while (curr != NULL && curr->item->key != item->key)
   {
      curr = curr->next;
   }
   return curr != NULL && curr->item->key == item->key &&
          !curr->marked;
}
```

traverse without locking

Illustration: contains

```
bool contains(Item *item)
{
   Node *curr = this->head;
   while (curr != NULL && curr->item->key != item->key)
   {
      curr = curr->next;
   }
   return curr != NULL && curr->item->key == item->key &&
          !curr->marked;
}
```

traverse without locking

present and not logically removed?

Summary: Lazy Sync.

- Summary:
  - Use of marker
  - Lazy add() and remove() + Wait-free contains()

- Good:
  - contains() doesn't lock
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse

- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays

Traffic Jam?!?

- Any concurrent data structure based on mutual exclusion has a weakness:

- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!

Synchronization: (preliminary) Summary

Comparison

	add()	remove()	contains()
Coarse-grained Sync.	whole object locked	whole object locked	whole object locked
Fine-grained Sync.	chain of pair-wise acquire and release	chain of pair-wise acquire and release	no lock
Optimistic Sync.	lock targets only, but validate with traversal	lock targets only, but validate with traversal	chain of acquire and release
Lazy Sync.	mark and lock targets only, retry if conflict	mark and lock targets only, retry if conflict	no lock (check for marker)
Lock-free Sync.	no lock	no lock	no lock