# Chapter 5
# Parallel Programming for Linked Data Structures

## Linked Data Structures

- Examples: linked lists, trees, graphs, hash tables
- Commonly used in many non-numerical programs
- Common features:
  - Operations involve: node insertion, node deletion, node search
  - Traversal follows pointer chasing pattern -> loop-carried dependence
- Example:

```
void addValue(pIntList pList, int key, int x)
{
 pIntListNode p = pList->head;
 while (p != NULL) {
  if (p->key == key)
    p->data = p->data + x;
  p = p->next;
 }
}
```

## How to Parallelize LDS

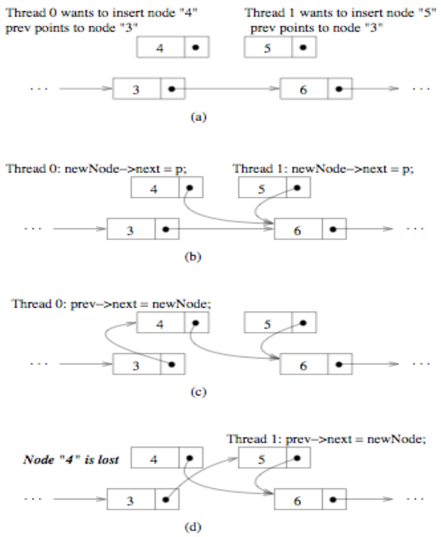
- Loop-level analysis does not reveal parallelism
- Look at algorithm level
- LDS is a data structure with operations that can be performed on them
  - node insertion
  - node deletion
  - node search
  - etc.
- Conceptually, we can allow several operations to be performed in parallel
- But how do we reason about the correctness of parallel operations?

## Correctness of Parallel LDS Operations

- Serializability = *a parallel execution of a group of operations or primitives is serializable if there is some seor primitives that produce an identical result*
- Suppose a node insertion and node deletion are performed in parallel
- The outcome must be equivalent to either
  - node insertion is performed after node deletion, or
  - node deletion is performed after node insertion
  - nothing else can be considered correct
- We will take a look at a simple case of a singly-linked list

# Serializability between 2 Insertions

*Conflict between 2 insertion operations*

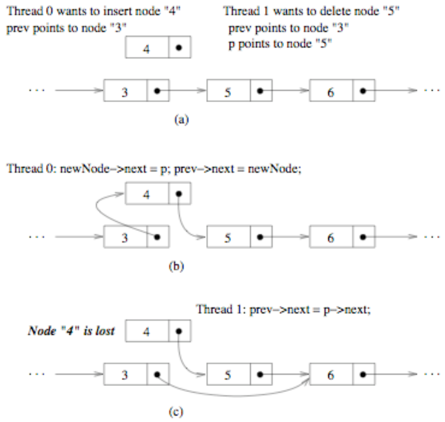


Thread 0 wants to insert node "4"
prev points to node "3"
Thread 1 wants to insert node "5"
prev points to node "3"
4
5
3
6
(a)
Thread 0: newNode–>next = p;
Thread 1: newNode–>next = p;
(b)
Thread 0: prev–>next = newNode;
(c)
Thread 1: prev–>next = newNode;
Node "4" is lost
(d)

- Serializable outcome:
  - insert 4, then insert 5
  - or insert 5, then insert 4
  - in both cases, both nodes 4 and 5 must be in the list at the end of execution
- Incorrect parallel execution may result in:
  - node 4 lost

# Serializability b/w Insertion and Deletion

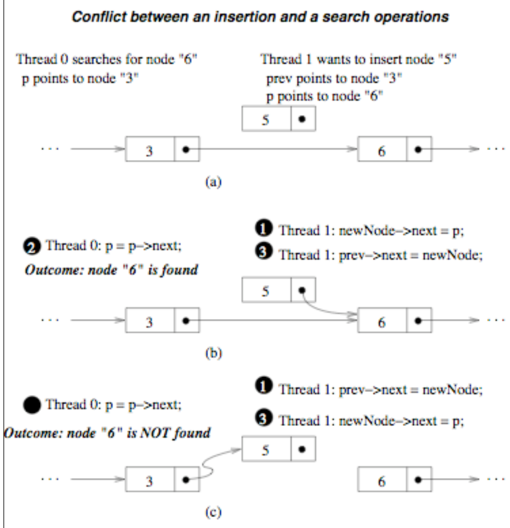
*Conflict between an insertion and a deletion operations*



Thread 0 wants to insert node "4"
prev points to node "3"
Thread 1 wants to delete node "5"
prev points to node "3"
p points to node "5"
4
3
5
6
(a)
Thread 0: newNode–>next = p; prev–>next = newNode;
(b)
Thread 1: prev–>next = p–>next;
Node "4" is lost
(c)

- Serializable outcome
  - insert 4, then delete 5
  - or delete 5, then insert 4
  - in both cases, at the end of execution, node 4 is in the list, node 5 is not in the list
- Incorrect parallel execution:
  - node 4 is lost, never inserted

## Serializability b/w Insertion and Search


Conflict between an insertion and a search operations
Thread 0 searches for node "6"
p points to node "3"
Thread 1 wants to insert node "5"
prev points to node "3"
p points to node "6"
5
3
6
(a)
❶ Thread 1: newNode->next = p;
❷ Thread 0: p = p->next;
❸ Thread 1: prev->next = newNode;
Outcome: node "6" is found
(b)
❶ Thread 1: prev->next = newNode;
● Thread 0: p = p->next;
❸ Thread 1: newNode->next = p;
Outcome: node "6" is NOT found
(c)

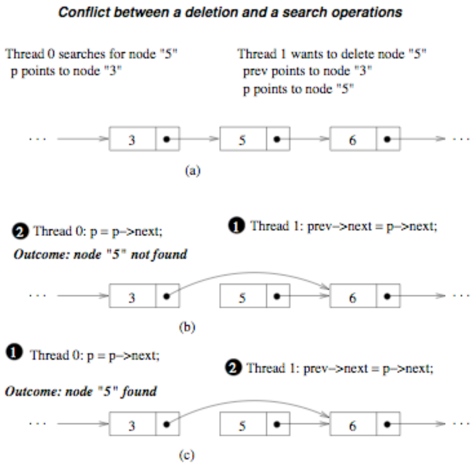
Insertion operation and node search operations

- insert 5, then search 6
- or search 6, then insert 5
- in both cases, at the end of execution, 5 must be in the list, and 6 is found

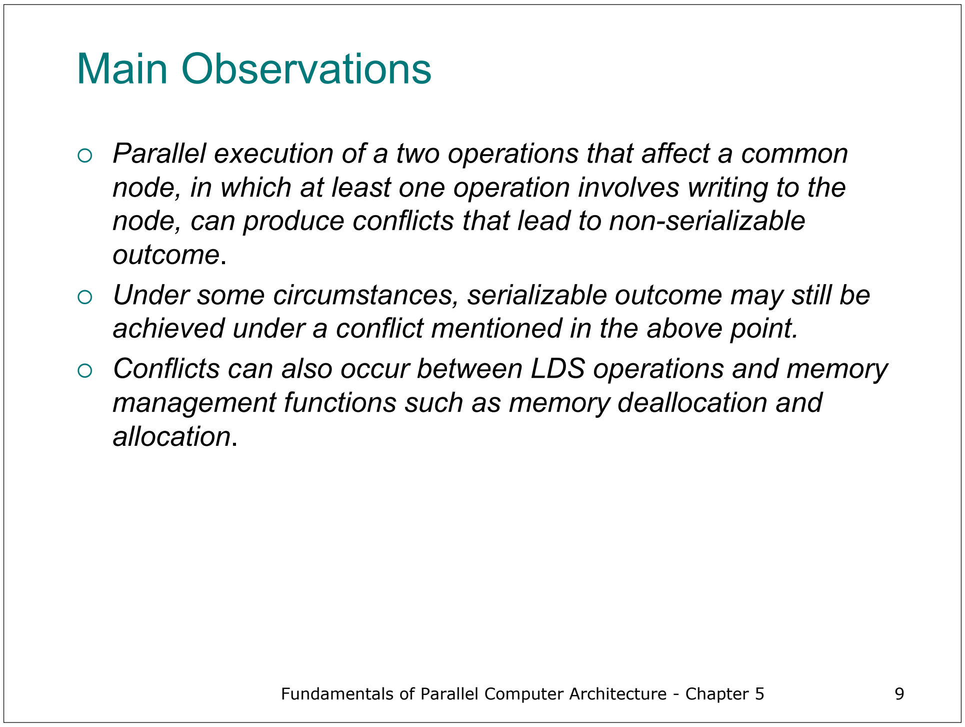
Incorrect parallel execution

- node 6 may be found
- but node 6 may also not be found, and the linked list is broken

## Serializability between Deletion and Search

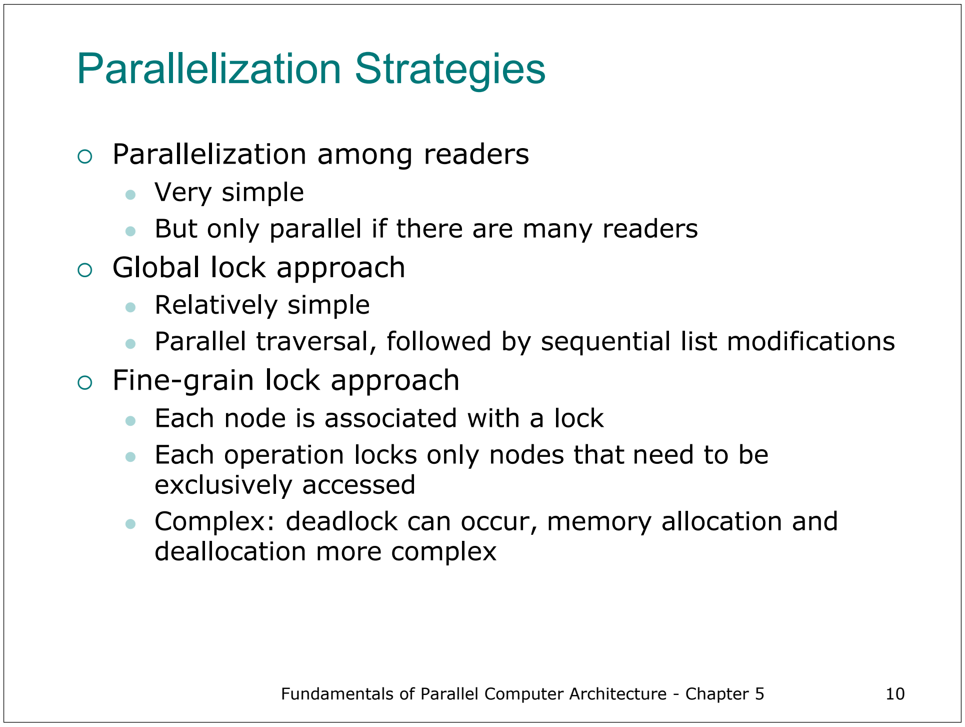

Conflict between a deletion and a search operations
Thread 0 searches for node "5"
p points to node "3"
Thread 1 wants to delete node "5"
prev points to node "3"
p points to node "5"
3
5
6
(a)
❷ Thread 0: p = p->next;
Outcome: node "5" not found
❶ Thread 1: prev->next = p->next;
(b)
❶ Thread 0: p = p->next;
❷ Thread 1: prev->next = p->next;
Outcome: node "5" found
(c)

- Deletion and search
  - delete 5, then search 5
  - search 5, then delete 5
- Outcome is serializable
  - Node 5 may be found or not found
  - Node 5 is deleted from the list



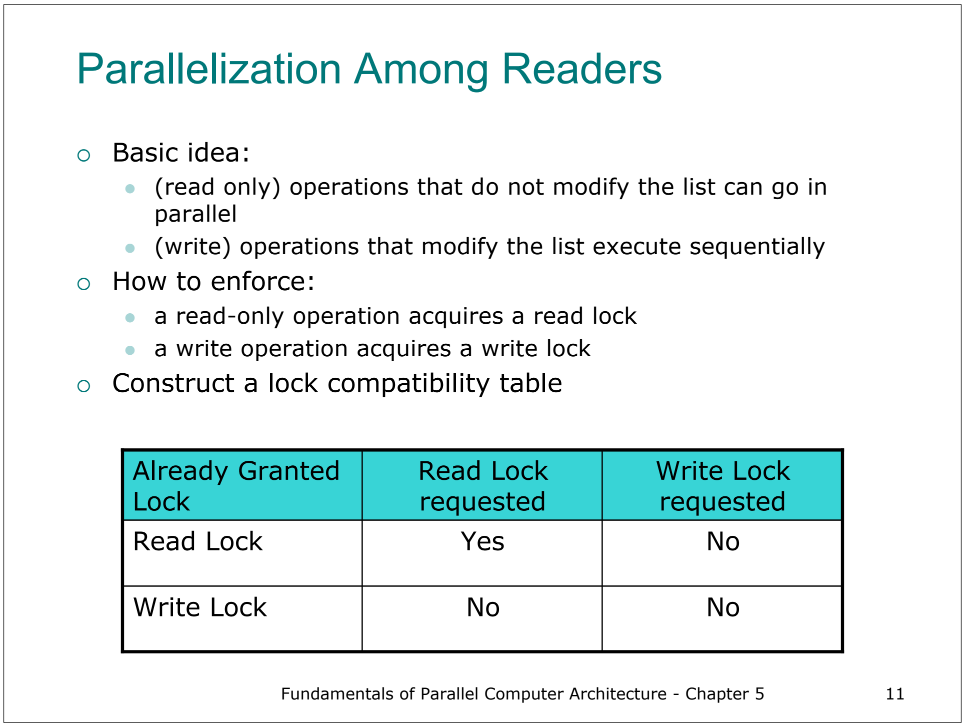
## Main Observations

- *Parallel execution of a two operations that affect a common node, in which at least one operation involves writing to the node, can produce conflicts that lead to non-serializable outcome*.
- *Under some circumstances, serializable outcome may still be achieved under a conflict mentioned in the above point.*
- *Conflicts can also occur between LDS operations and memory management functions such as memory deallocation and allocation*.



## Parallelization Strategies

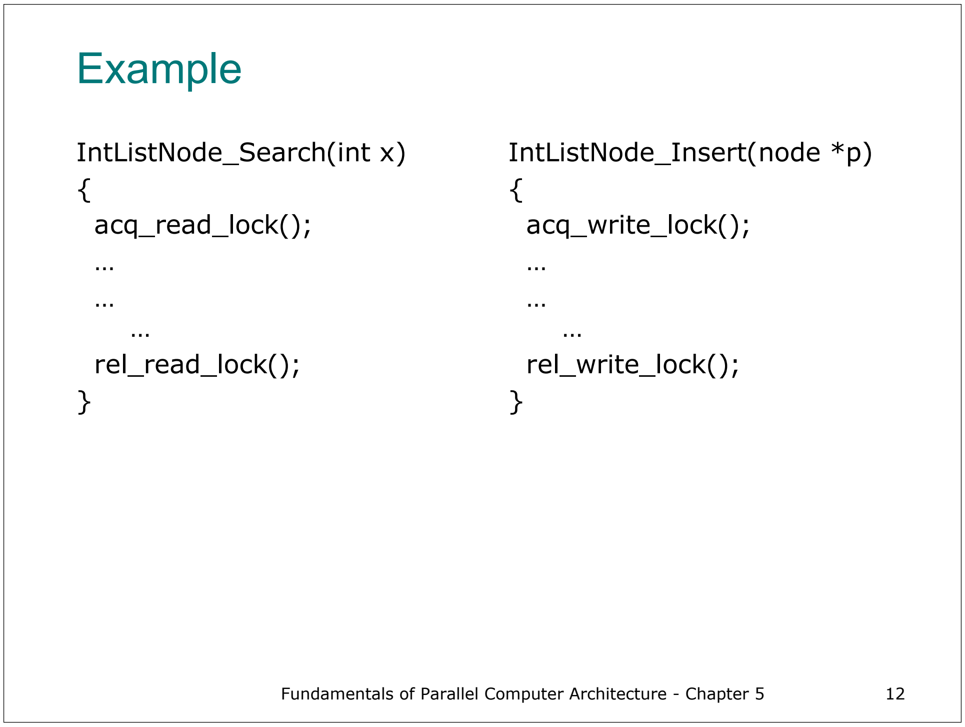
- Parallelization among readers
  - Very simple
  - But only parallel if there are many readers
- Global lock approach
  - Relatively simple
  - Parallel traversal, followed by sequential list modifications
- Fine-grain lock approach
  - Each node is associated with a lock
  - Each operation locks only nodes that need to be exclusively accessed
  - Complex: deadlock can occur, memory allocation and deallocation more complex



## Parallelization Among Readers

- Basic idea:
  - (read only) operations that do not modify the list can go in parallel
  - (write) operations that modify the list execute sequentially
- How to enforce:
  - a read-only operation acquires a read lock
  - a write operation acquires a write lock
- Construct a lock compatibility table

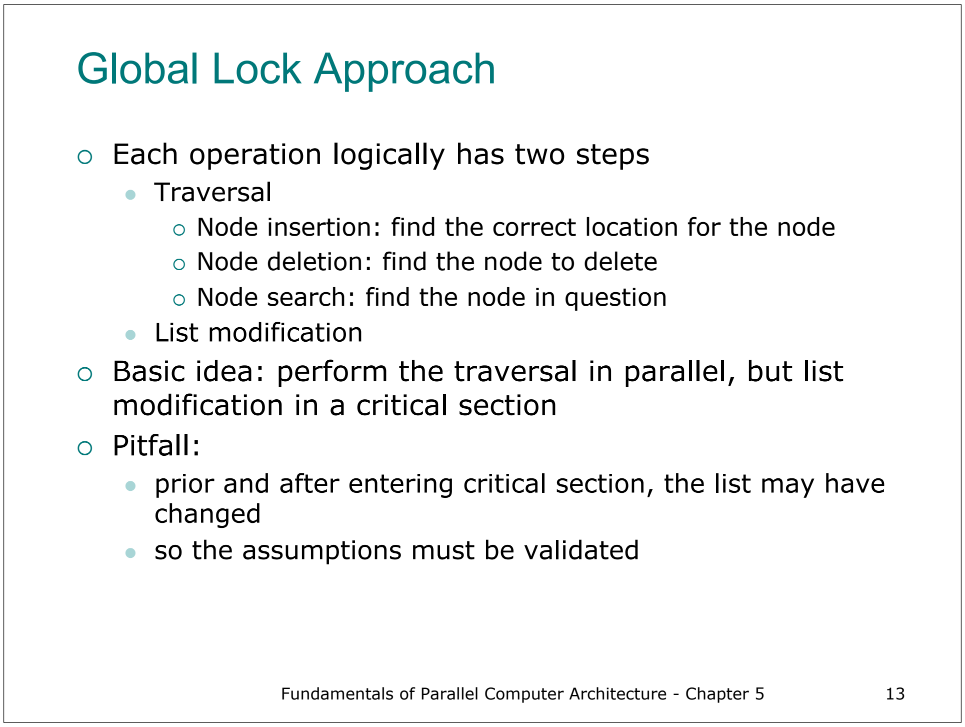
| Already Granted Lock | Read Lock requested | Write Lock requested |
|---|---|---|
| Read Lock | Yes | No |
| Write Lock | No | No |



## Example

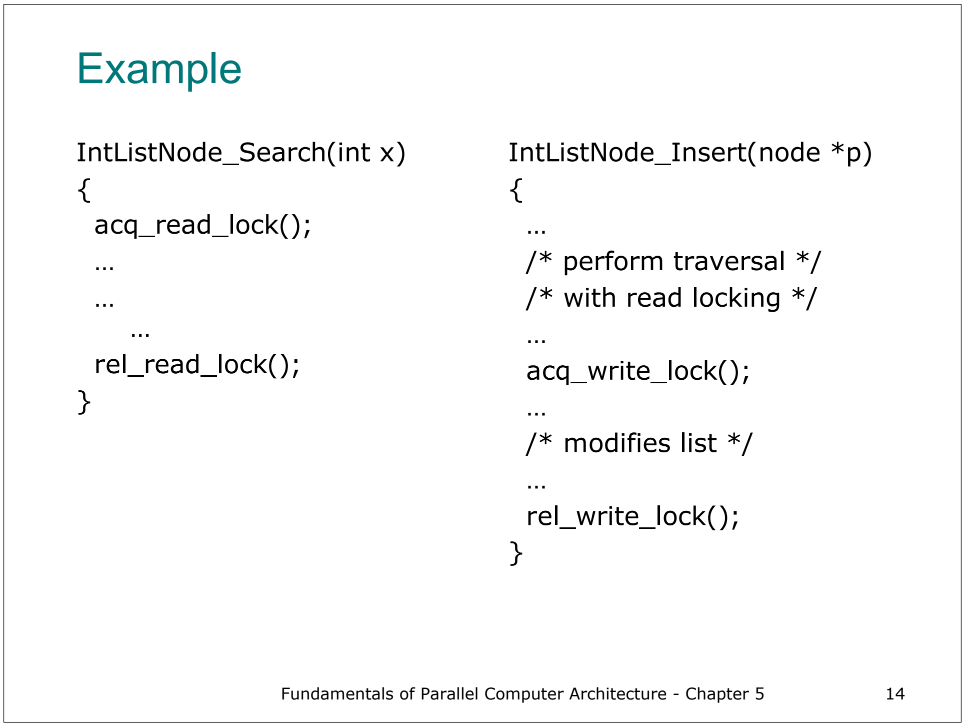
```
IntListNode_Search(int x)
{
  acq_read_lock();
  …
  …
      …
  rel_read_lock();
}
```

```
IntListNode_Insert(node *p)
{
  acq_write_lock();
  …
  …
      …
  rel_write_lock();
}
```



## Global Lock Approach

- Each operation logically has two steps
  - Traversal
    - Node insertion: find the correct location for the node
    - Node deletion: find the node to delete
    - Node search: find the node in question
  - List modification
- Basic idea: perform the traversal in parallel, but list modification in a critical section
- Pitfall:
  - prior and after entering critical section, the list may have changed
  - so the assumptions must be validated



## Example

```
IntListNode_Search(int x)
{
  acq_read_lock();
  …
  …
      …
  rel_read_lock();
}
```

```
IntListNode_Insert(node *p)
{
  …
  /* perform traversal */
  /* with read locking */
  …
  acq_write_lock();
  …
  /* modifies list */
  …
  rel_write_lock();
}
```

## Fine-Grain Locking Approach

- Associate each node with a lock (read, write)
- Each operation locks only needed nodes
- (Read and write) operations execute in parallel except when they conflict on some nodes

- Nodes that will be modified are write locked
- Nodes that are read and must remain unchanged are read locked

- Pitfall: deadlocks becomes possible
- Deadlocks can be avoided by imposing a global lock acquisition order

## Example

- Refer to code example in the book