Practical Concurrent and Parallel Programming 7

Peter Sestoft
IT University of Copenhagen

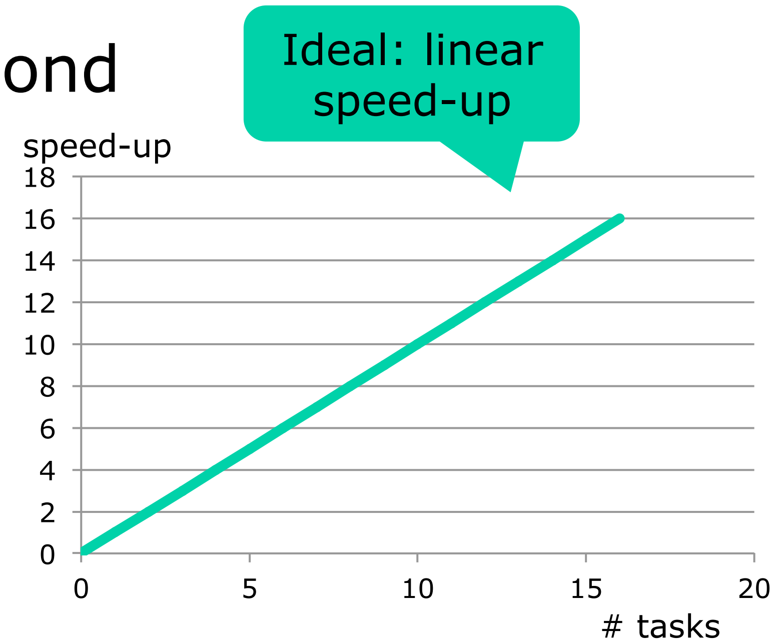
Friday 2014-10-10*

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
  - (A) Hash map à la Java monitor
  - (B) Hash map with lock striping
  - (C) Ditto with lock striping and non-blocking reads
- An atomic long with ”thread striping”
- Shared mutable state is slow on multicore

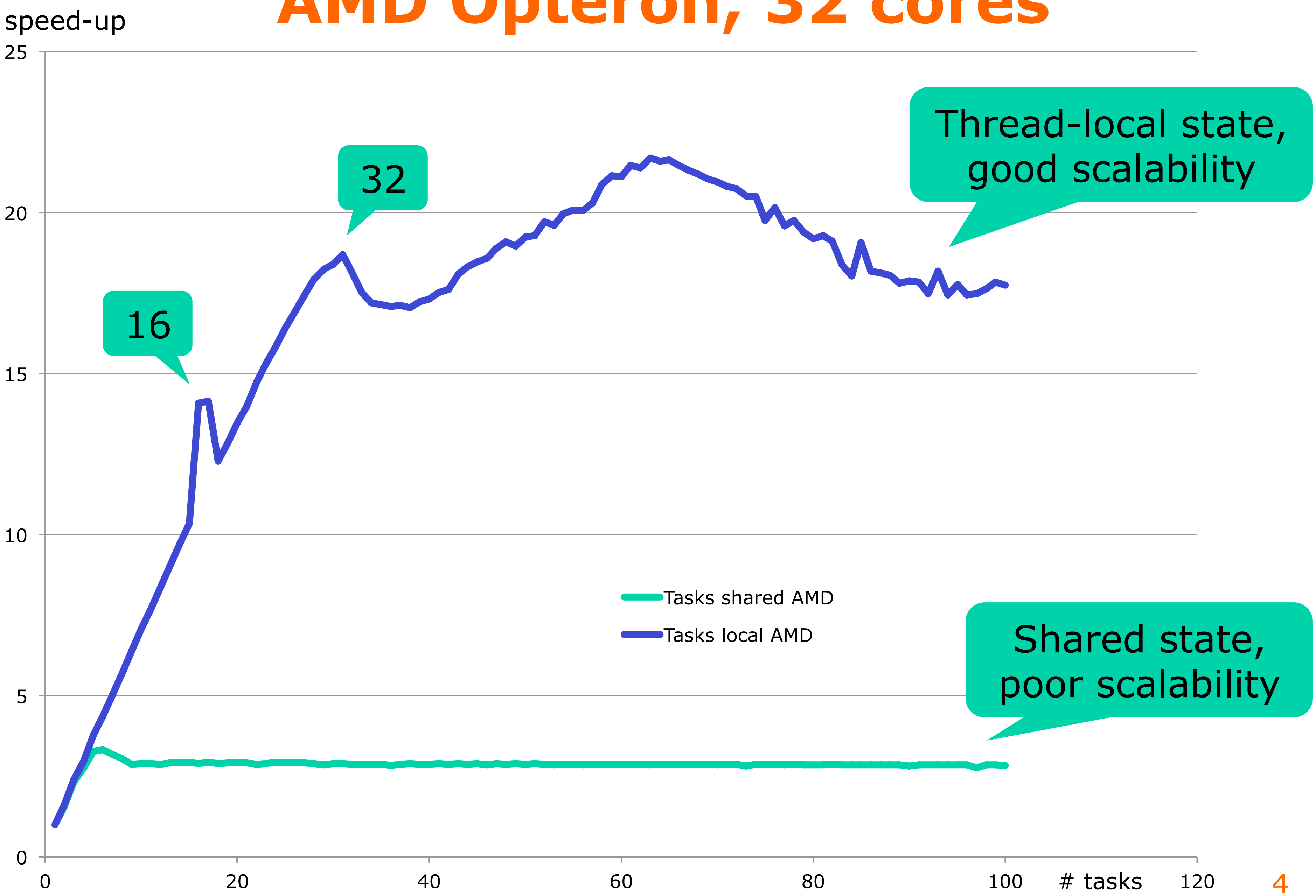
Performance versus scalability

- Performance
  - Latency: time till first result
  - Throughput: results per second
- Scalability
  - Improved throughput when more resources are added
  - Speed-up as function of number of threads or tasks



- One may sacrifice performance for scalability
  - OK to be slower on 1 core if faster on 2 or 4 or ...
  - Requires rethinking our “best” sequential code

Scalability of prime counting
AMD Opteron, 32 cores



What limits throughput?

- CPU-bound
  - Eg. counting prime numbers
  - To speed up, add more CPUs (cores)
- Memory-bound
  - Eg. make color histograms of images
  - To speed up, improve data locality; recompute more
- Input/output-bound
  - Eg. fetching webpages and finding links
  - To speed up, use more tasks
- Synchronization-bound
  - Eg. image segmentation using shared data structure
  - To speed up, improve shared data structure. How?

Much of this lecture

What limits scalability?

- Sequentiality of *problem*
  - Example: growing a crop
    - 4 months growth + 1 month harvest if done by 1 person
    - Growth (sequential) cannot be speeded up
    - Using 30 people to harvest, takes 1/30 month = 1 day
    - Maximal speed-up factor, using many many harvesters: 5/(4+1/30) = 1.24 times faster
  - Amdahl's law
    - F = sequential fraction of problem = 4/5 = 0.8
    - N = number of parallel resources = 30
    - Speed-up <= 1/(F+(1-F)/N) = 1/(0.8+0.2/30) = 1.24
- Sequentiality of *solution*
  - Solution slower than necessary because shared resources, eg. locking, sequentialize solution

Reduce lock duration

```
public class AttributeStore {
  private final Map<String, String> attributes = ...;
  public synchronized boolean userLocationMatches(String name,
                                                  String regexp)
  {
    String key = "users." + name + ".location";
    String location = attributes.get(key);
    return location != null && Pattern.matches(regexp, location);
  }
}
```

Must lock

May be slow, holds lock unnecessarily

Goetz p. 233-234

- Better:

```
public class BetterAttributeStore {
  private final Map<String, String> attributes = ...;
  public boolean userLocationMatches(String name, String regexp) {
    String key = "users." + name + ".location";
    String location;
    synchronized (this) {
      location = attributes.get(key);
    }
    return location != null && Pattern.matches(regexp, location);
  }
}
```

Lock only here

Does not hold lock

Lock splitting

Goetz p. 236

```
public class ServerStatusBeforeSplit {
  @GuardedBy("this") public final Set<String> users = ...;
  @GuardedBy("this") public final Set<String> queries = ...;
  public synchronized void addUser(String u) {
    users.add(u);
  }
  public synchronized void addQuery(String q) {
    queries.add(q);
  }
  public synchronized void removeUser(String u) { .
}
```

Lock server status object

Lock server status object

- Better, (addUser and addQuery can run concurrently)

```
public class ServerStatusAfterSplit {
  @GuardedBy("users")   public final Set<String> users = ...;
  @GuardedBy("queries") public final Set<String> queries = ...;
  public void addUser(String u) {
    synchronized (users) { users.add(u); }
  }
  public void addQuery(String q) {
    synchronized (queries) { queries.add(q); }
  }
  ...
}
```

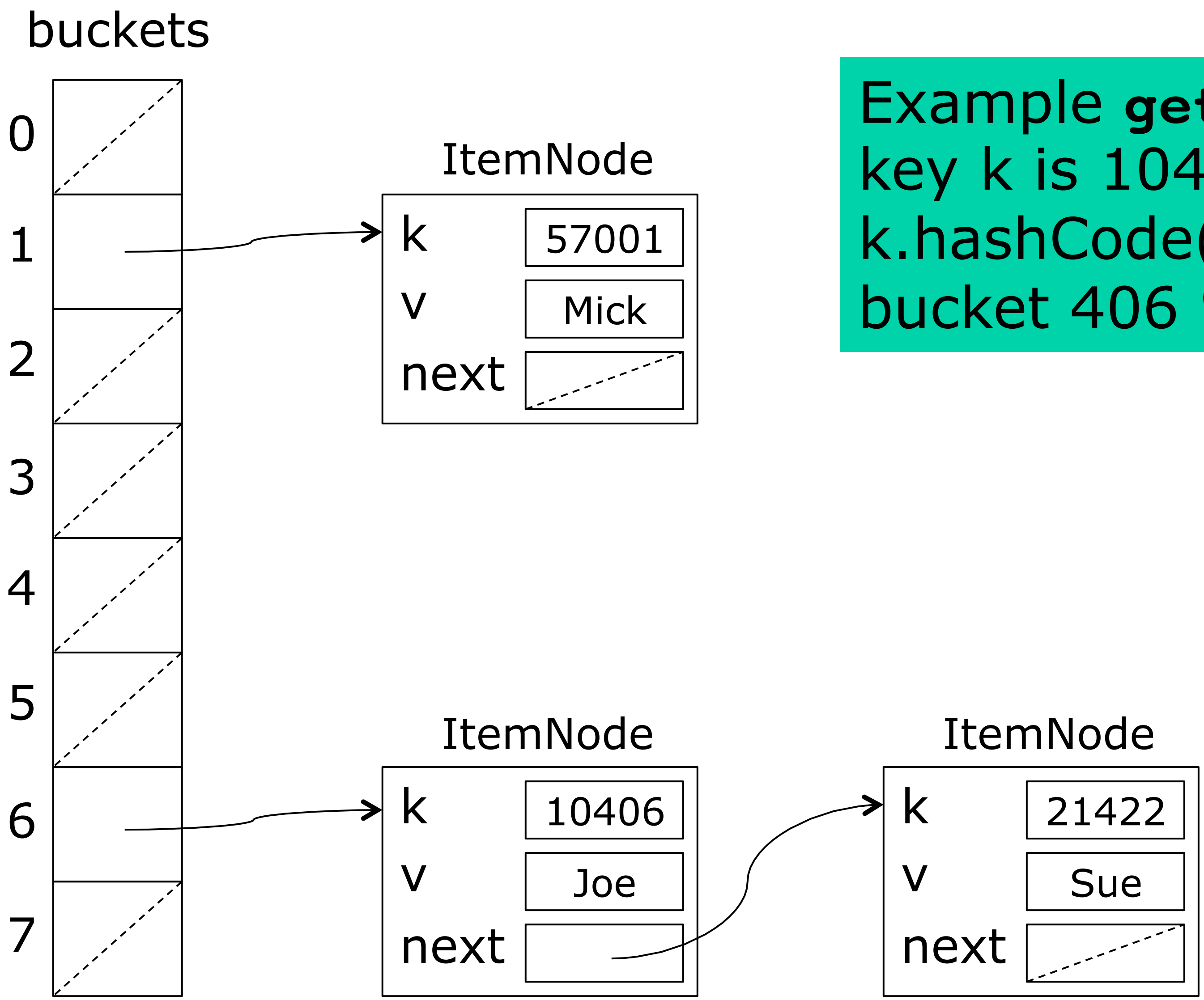
Lock only users set

Lock only queries set

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- **Hash maps, a scalability case study**
  - (A) Hash map à la Java monitor
  - (B) Hash map with lock striping
  - (C) Ditto with lock striping and non-blocking reads
- An atomic long with ”thread striping”
- Shared mutable state is slow on multicore

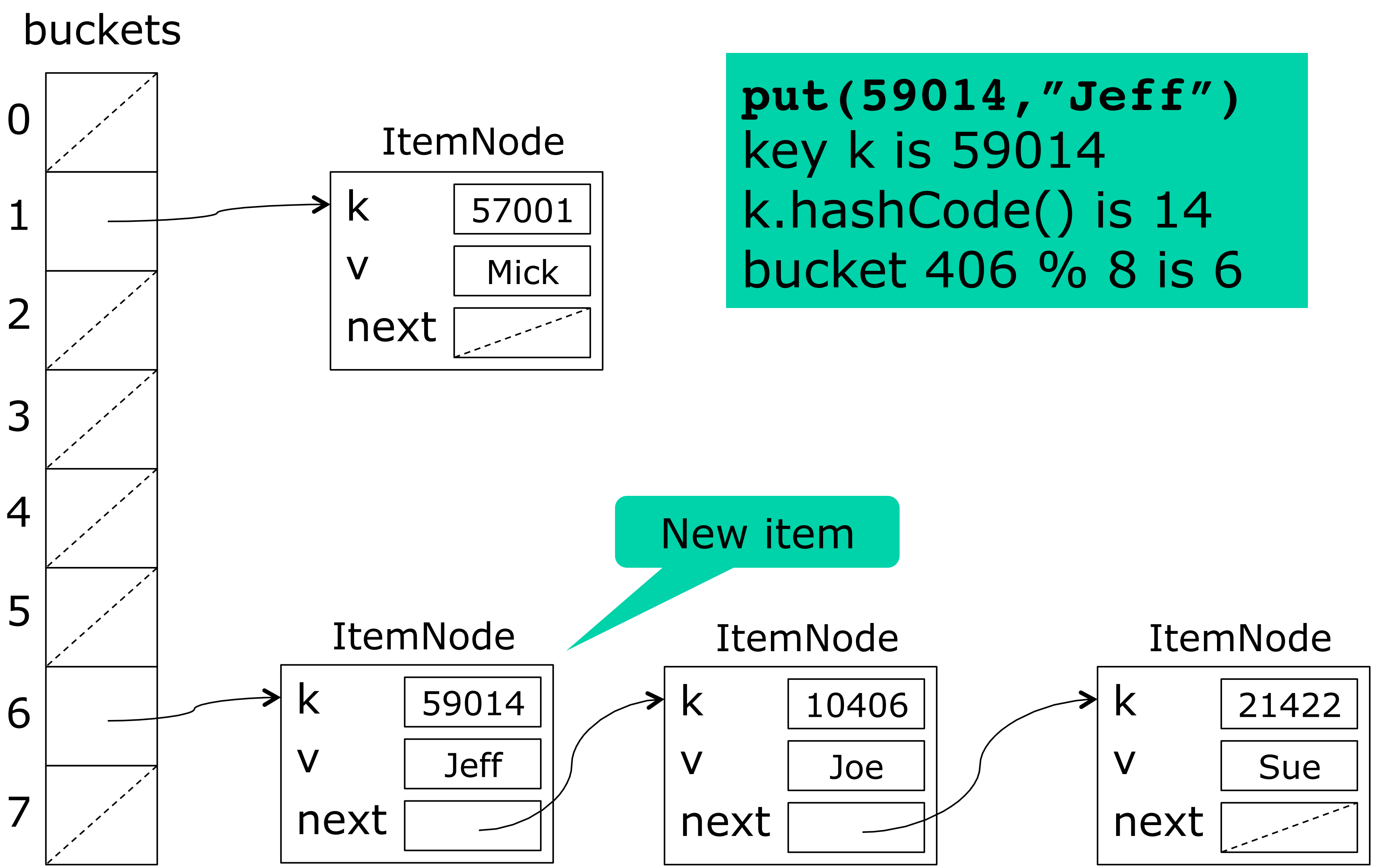
A hash map = buckets table + item node lists



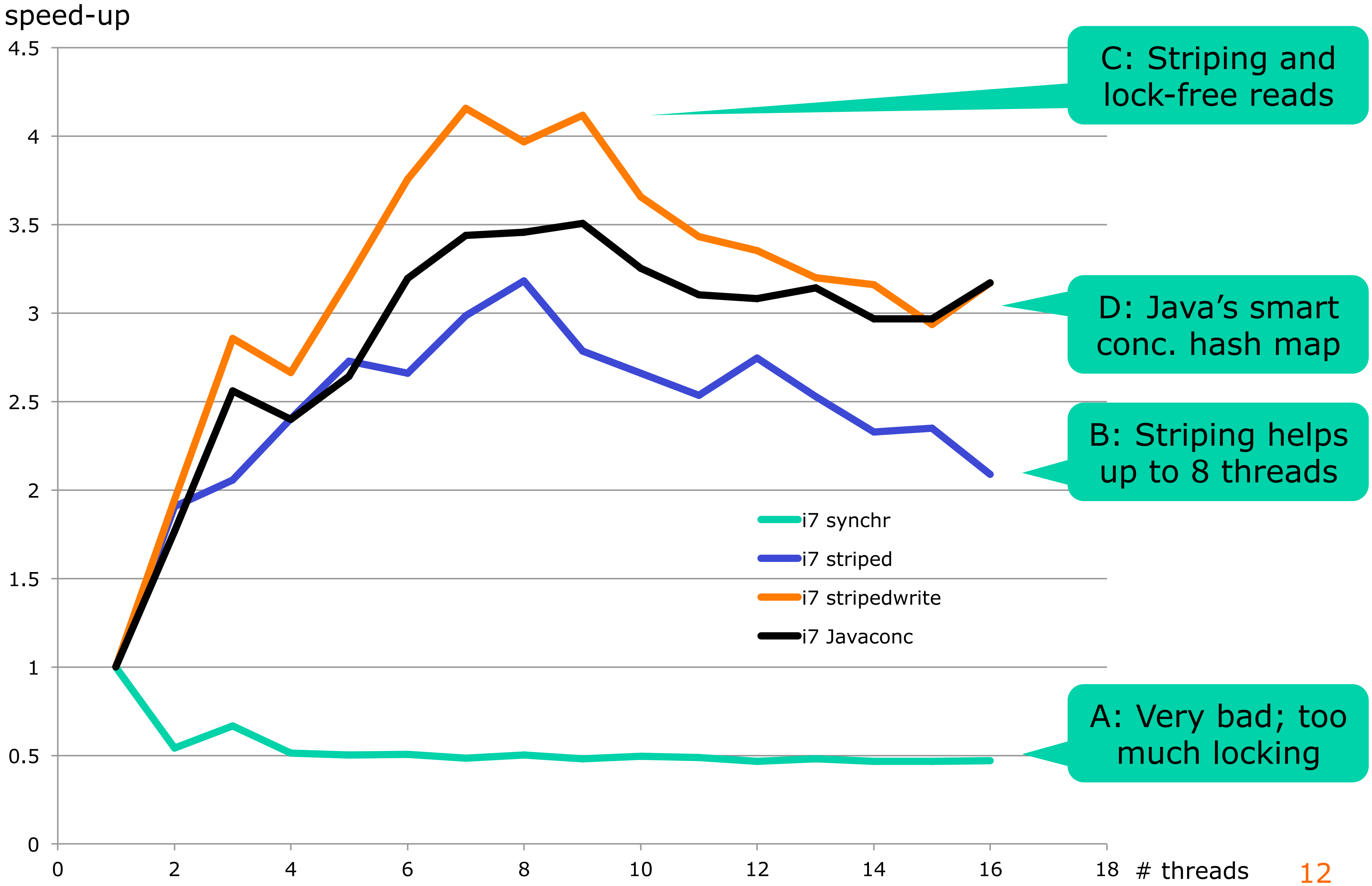
Example **`get(10406)`**
key k is 10406
k.hashCode() is 406
bucket 406 % 8 is 6

Insertion into the hashmap

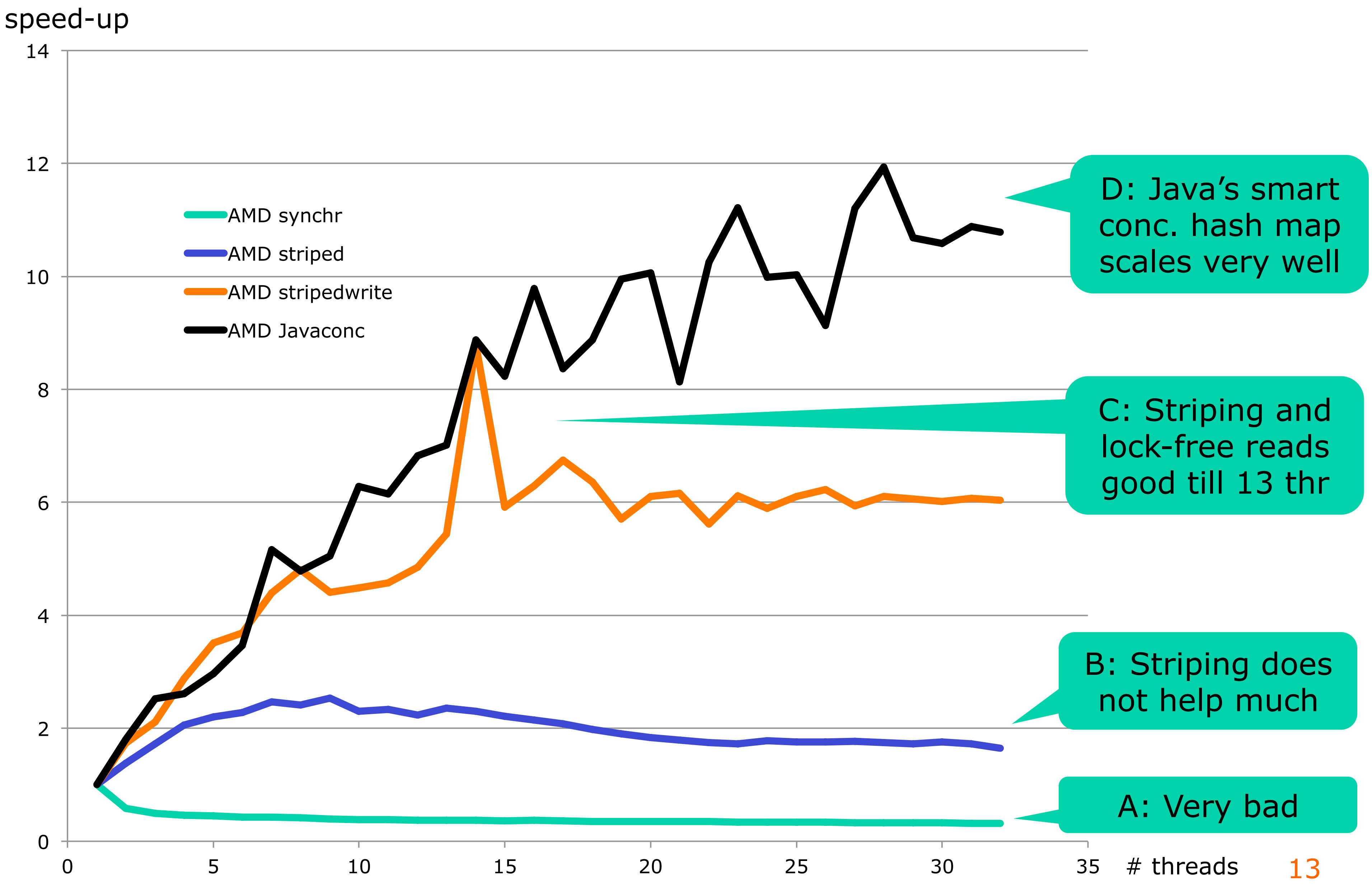
put(59014,"Jeff")
key k is 59014
k.hashCode() is 14
bucket 406 % 8 is 6



Scalability of hash maps
Intel i7 w 4 cores & hyperthreading



Scalability of hash maps
AMD Opteron w 32 cores



Our map interface

- Reduced version of Java's Map<K,V>

TestStripedMap.java

```
interface OurMap<K,V> {
  boolean containsKey(K k);
  V get(K k);
  V put(K k, V v);
  V putIfAbsent(K k, V v);
  V remove(K k);
  int size();
  void forEach(Consumer<K,V> consumer);
  void reallocateBuckets();
}
```

```
interface Consumer<K,V> {
  void accept(K k, V v);
}
```

```
for (Entry (k,v) : map)
  System.out.printf(...);
```

```
map.forEach((k, v) ->
  System.out.printf("%10d maps to %s%n", k, v));
```

Map A

Synchronized map implementation

```
static class ItemNode<K,V> {
  private final K k;
  private V v;
  private ItemNode<K,V> next;
  public ItemNode(K k, V v, ItemNode<K,V> next) { ... }
}
```

Visibility depends on synchronization

TestStripedMap.java

Java monitor pattern

```
class SynchronizedMap<K,V> implements OurMap<K,V>  {
  private ItemNode<K,V>[] buckets;  // guarded by this
  private int cachedSize;           // guarded by this
  public synchronized V get(K k) { ... }
  public synchronized boolean containsKey(K k) { ... }
  public synchronized int size() { return cachedSize; }
  public synchronized V put(K k, V v) { ... }
  public synchronized V putIfAbsent(K k, V v) { ... }
  public synchronized V remove(K k) { ... }
  public synchronized void forEach(Consumer<K,V> consumer) { ... }
}
```

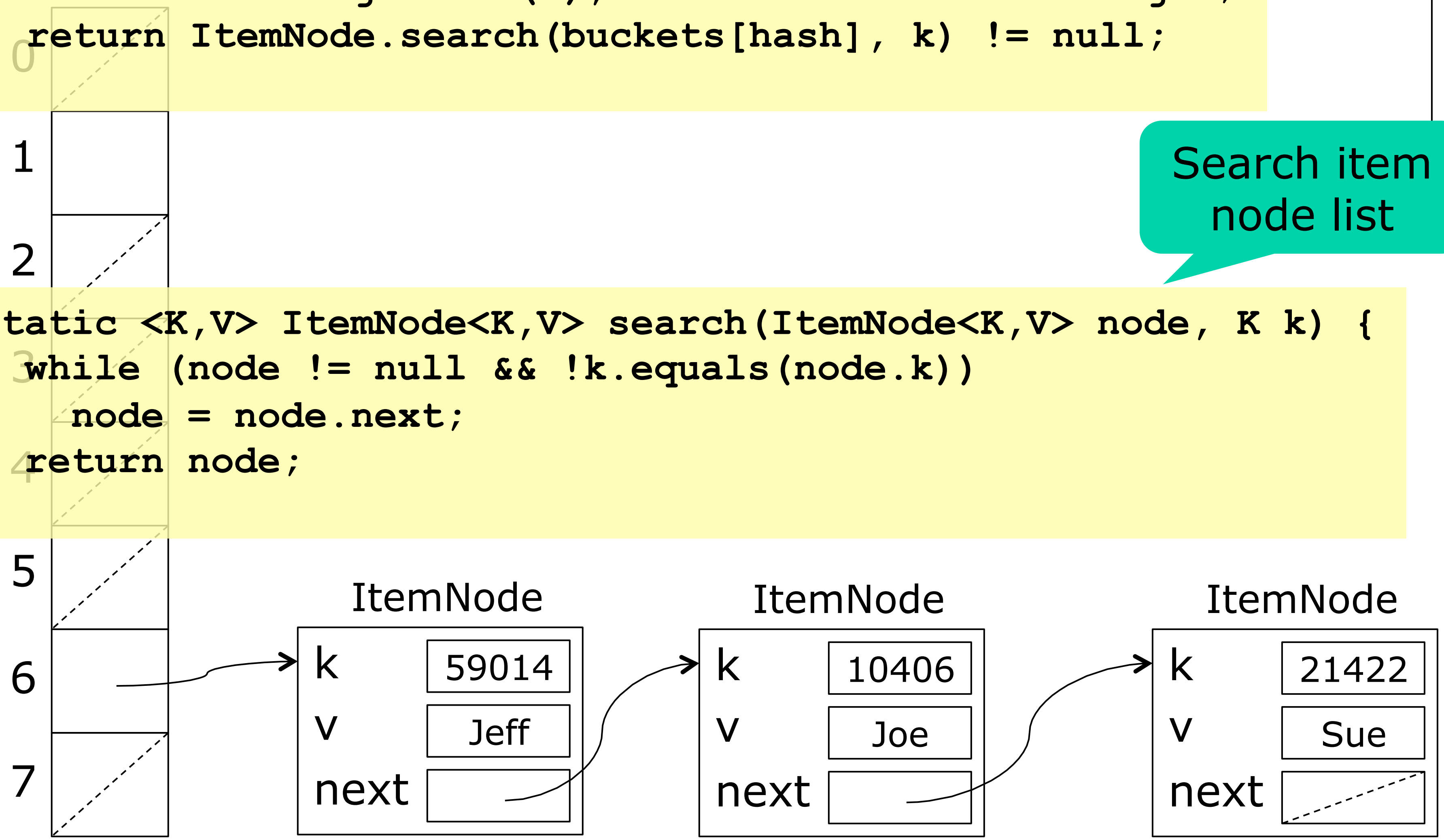
Implementing containsKey

Find bucket

```
public synchronized boolean containsKey(K k) {
  final int h = getHash(k), hash = h % buckets.length;
  return ItemNode.search(buckets[hash], k) != null;
}
```

Search item node list

```
static <K,V> ItemNode<K,V> search(ItemNode<K,V> node, K k) {
  while (node != null && !k.equals(node.k))
    node = node.next;
  return node;
}
```



StripedMap.java

Map A

Implementing putIfAbsent

```
public synchronized V putIfAbsent(K k, V v) {
  final int h = getHash(k), hash = h % buckets.length;
  ItemNode<K,V> node = ItemNode.search(buckets[hash], k);
  if (node != null)
    return node.v;
  else {
    buckets[hash] = new ItemNode<K,V>(k, v, buckets[hash]);
    cachedSize++;
    return null;
  }
}
```

Search bucket's node list

If key exists, return value

Else add new item node at front of list

- All methods are synchronized
  - atomic access to buckets table and item nodes
  - all writes by put, putIfAbsent, remove, reallocateBuckets are visible to containsKey, get, size, forEach

Reallocating buckets

- Hash map efficiency requires short node lists
- When item node lists become too long, then
  - Double buckets array size to newCount
  - For each item node (k,v)
    - Recompute **`newHash = k.hashCode() % newCount`**
    - Link item node into new list at **`newBuckets[newHash]`**

- This is a dramatic operation
  - Must lock the entire data structure
  - Can happen at any insertion

Map A

ReallocateBuckets implementation

TestStripedMap.java

```
public synchronized void reallocateBuckets() {
  final ItemNode<K,V>[] newBuckets = makeBuckets(2 * buckets.length);
  for (int hash=0; hash<buckets.length; hash++) {
    ItemNode<K,V> node = buckets[hash];
    while (node != null) {
      final int newHash = getHash(node.k) % newBuckets.length;
      ItemNode<K,V> next = node.next;
      node.next = newBuckets[newHash];
      newBuckets[newHash] = node;
      node = next;
    }
  }
  buckets = newBuckets;
}
```

For each item node

Compute new hash

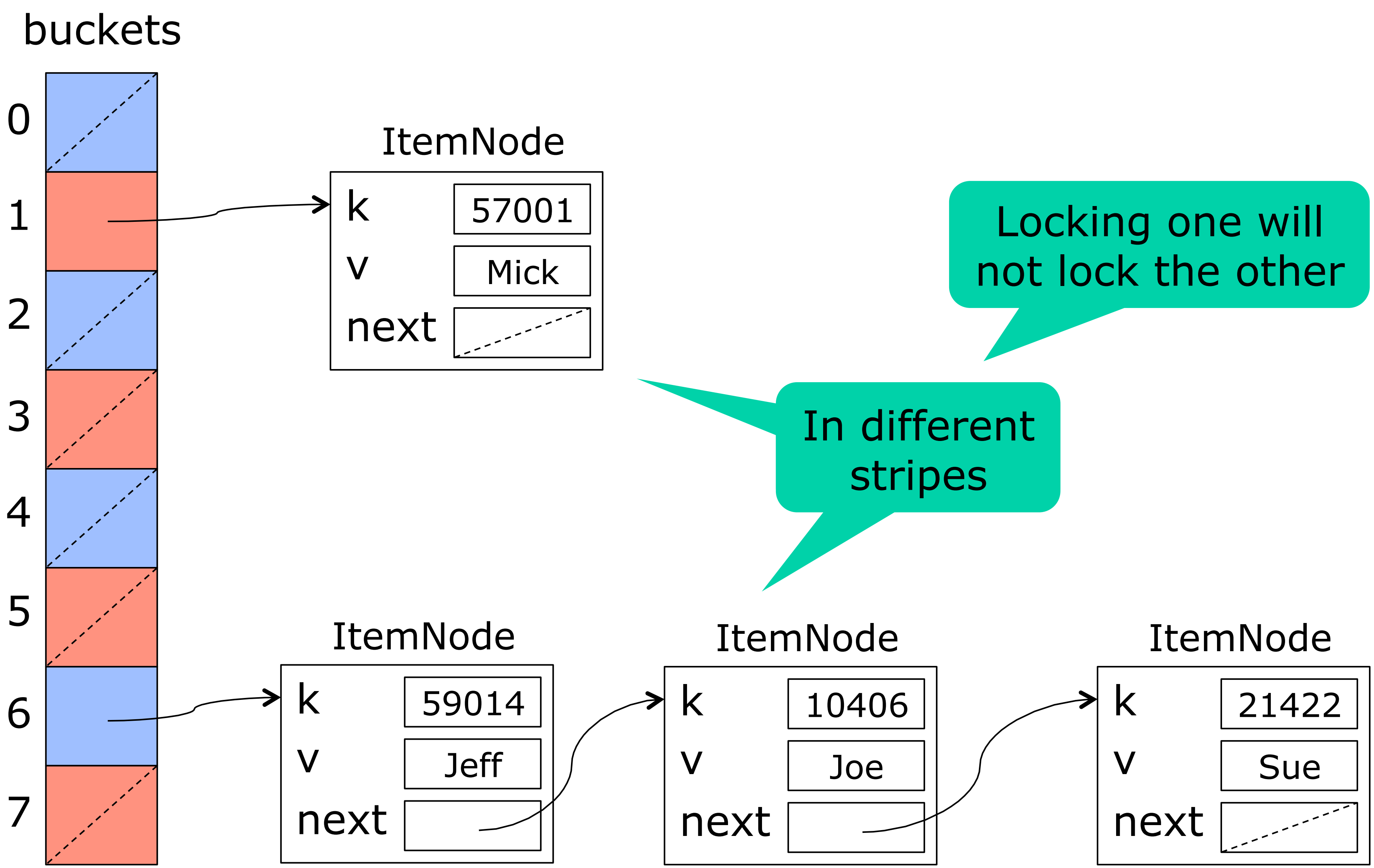
Link into new bucket

- Seems efficient: reuses each ItemNode
  - Links it into an new item node list
  - So destructs the old item node list
  - So read access impossible during reallocation
  - Good 1-core performance, but bad scalability

Better scalability: Lock striping

- Guarding the table with a single lock works
  - ... but does not scale well (actually **very** badly)
- Idea: Each bucket could have its own lock
- In practice
  - use a few, maybe 16, locks
  - guard every 16th bucket with the same lock
  - locks[0] guards bucket 0, 16, 32, ...
  - locks[1] guards bucket 1, 17, 33, ...
- With high probability
  - two operations will work on different stripes
  - hence will take different locks
- Less lock contention, better scalability

Lock striping in hash map
Two stripes 0 = blue and 1 = red



Map B

Striped hashmap implementation

NB!

```
class StripedMap<K,V> implements OurMap<K,V> {
  private volatile ItemNode<K,V>[] buckets;
  private final int lockCount;
  private final Object[] locks;
  private final int[] sizes;

  public boolean containsKey(K k) { ... }
  public V get(K k) { ... }
  public int size() { ... }
  public V put(K k, V v) { ... }
  public V putIfAbsent(K k, V v) { ... }
  public V remove(K k) { ... }
  public void forEach(Consumer<K,V> consumer) { ... }
}
```

Methods **not** synchronized

TestStripedMap.java

- Synchronization on **`lock[stripe]`** ensures
  - atomic access within each stripe
  - visibility of writes to readers

Map B

Implementation of containsKey

```
public boolean containsKey(K k) {
  final int h = getHash(k), stripe = h % lockCount;
  synchronized (locks[stripe]) {
    final int hash = h % buckets.length;
    return ItemNode.search(buckets[hash], k) != null;
  }
}
```

TestStripedMap.java

- Compute key’s hash code
- Lock the relevant stripe
- Compute hash index, access bucket
- Search node item list

- What if buckets were reallocated between computing “stripe” and locking?

Representing hash map size

- Could use a single AtomicInteger **`size`**
  - might limit concurrency
- Instead use one **`int`** per stripe
  - read and write while holding the stripe's lock

```
public int size() {
  int result = 0;
  for (int stripe=0; stripe<lockCount; stripe++)
    synchronized (locks[stripe]) {
      result += sizes[stripe];
    }
  return result;
}
```

- A stripe might be updated right after we read its size, before we return the sum
  - This is acceptable in concurrent data structures

Map B

Striped put(k,v)

```
public V put(K k, V v) {
  final int h = getHash(k), stripe = h % lockCount;
  synchronized (locks[stripe]) {
    final int hash = h % buckets.length;
    final ItemNode<K,V> node = ItemNode.search(buckets[hash], k);
    if (node != null) {
      V old = node.v;
      node.v = v;
      return old;
    } else {
      buckets[hash] = new ItemNode<K,V>(k, v, buckets[hash]);
      sizes[stripe]++;
      return null;
    }
  }
}
```

Lock stripe

If k exists, update value to v, return old

Else add new item node (k,v)

And add 1 to stripe size

Map B

Reallocating buckets

- Must lock all stripes; how take **`nlocks`** locks?
  - Use recursion: each call takes one more lock

TestStripedMap.java

```
private void lockAllAndThen(Runnable action) {
  lockAllAndThen(0, action);
}
private void lockAllAndThen(int nextStripe, Runnable action) {
  if (nextStripe >= lockCount)
    action.run();
  else
    synchronized (locks[nextStripe]) {
      lockAllAndThen(nextStripe + 1, action);
    }
}
```

```
synchronized(locks[0]) {
  synchronized(locks[1]) {
    ...
    synchronized(locks[15]) {
      action.run();
} ... } }
```

Overall effect of calling **`lockAllAndThen(0, action)`**

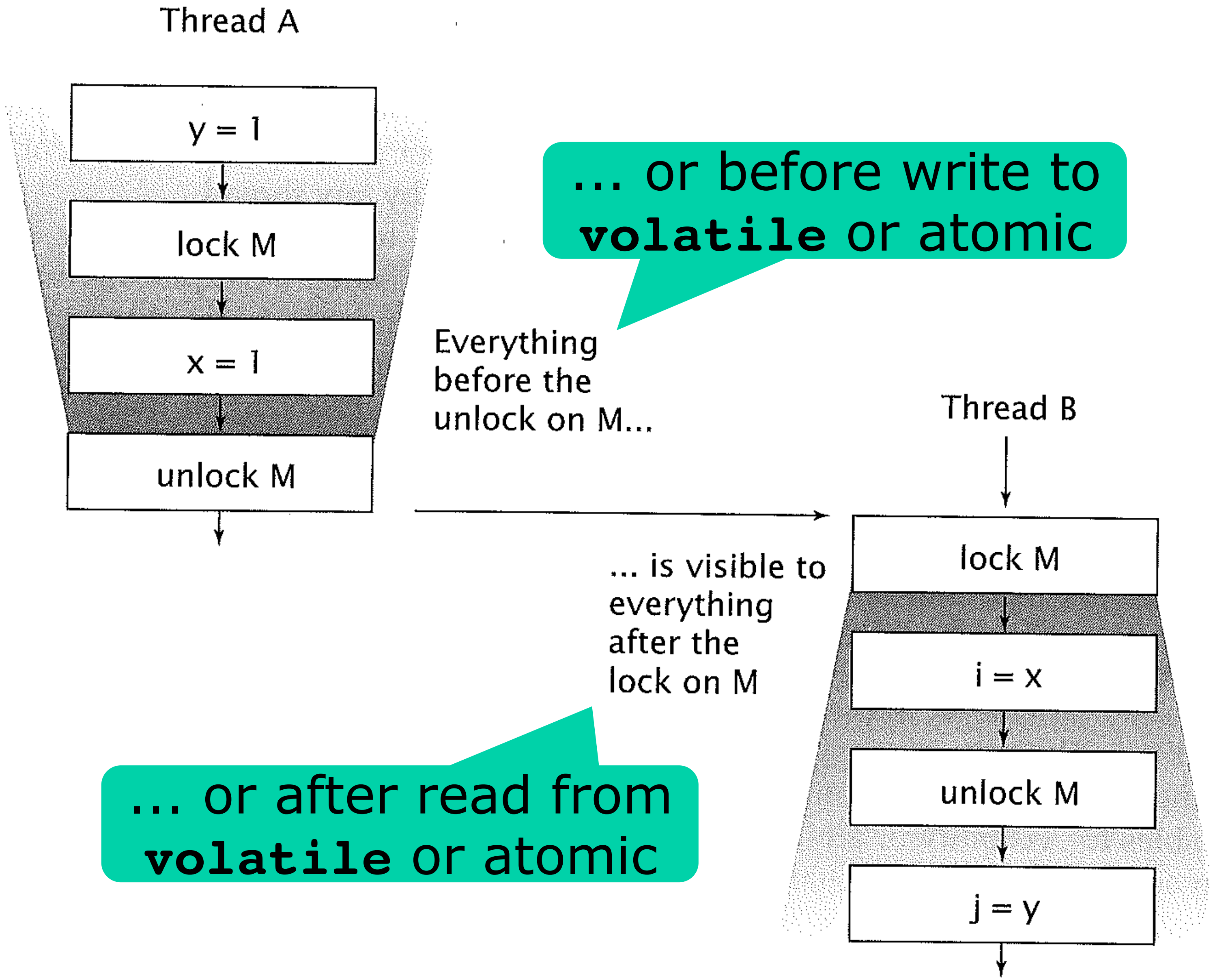
All locks held when calling **`action.run()`**

Idea: Immutable item nodes

- We can make read access lock free
- Good if more reads than writes
- A *read* of key k consists of
  - Compute **`hash = getHash(k) % buckets.length`**
  - Access **`buckets[hash]`** to get an item node list
  - Search the immutable item node list
- (1) Must make **`buckets`** access *atomic*
  - Get local reference: **`final ItemNode<K,V>[] bs = buckets;`**
- (2) No lock on reads, how make writes *visible*?
  - Represent stripe sizes using AtomicIntegerArray
  - A hash map write must write to stripe size, **last**
  - A hash map read must read from stripe size, **first**
  - Also, declare **`buckets`** field **`volatile`**

Must be atomic

Visibility by lock, volatile, or atomic



Goetz p. 37

Locking the stripes only on write

TestStripedMap.java

```
class StripedWriteMap<K,V> implements OurMap<K,V> {
  private volatile ItemNode<K,V>[] buckets;
  private final int lockCount;
  private final Object[] locks;
  private final AtomicIntegerArray sizes;
  ... non-synchronized methods, signatures as in StripedMap<K,V>
}
```

Immutable

```
static class ItemNode<K,V> {
  private final K k;
  private final V v;
  private final ItemNode<K,V> next;

  static boolean search(ItemNode<K,V> node, K k, Holder<V> old) ...
  static ItemNode<K,V> delete(ItemNode<K,V> node, K k, Holder<V> old) ...
}
```

```
static class Holder<V> { // Not threadsafe
  private V value;
  public V get() { return value; }
  public void set(V value) { this.value = value; }
}
```

To hold ”out” parameters

Lock-free ContainsKey

```
public boolean containsKey(K k) {
  final ItemNode<K,V>[] bs = buckets;
  final int h = getHash(k), stripe = h % lockCount,
    hash = h % bs.length;
  return sizes.get(stripe) != 0 && ItemNode.search(bs[hash], k, null);
}
```

Read volatile field, once ...

First read sizes, to make previous writes visible

... so that hash and array are consistent

- In class ItemNode, a plain linked list search:

```
static <K,V> boolean search(ItemNode<K,V> node, K k, Holder<V> old) {
  while (node != null)
    if (k.equals(node.k)) {
      if (old != null)
        old.set(node.v);
      return true;
    } else
      node = node.next;
  return false;
}
```

Item nodes are immutable and so threadsafe

If k found, may return v here

Stripe-locking put(k,v)

```
public V put(K k, V v) {
  final int h = getHash(k), stripe = h % lockCount;
  synchronized (locks[stripe]) {
    final ItemNode<K,V>[] bs = buckets;
    final int hash = h % bs.length;
    final Holder<V> old = new Holder<V>();
    final ItemNode<K,V> node = bs[hash],
      newNode = ItemNode.delete(node, k, old);
    bs[hash] = new ItemNode<K,V>(k, v, newNode);
    sizes.getAndAdd(stripe, newNode == node ? 1 : 0);
    return old.get();
  }
}
```

Lock stripe

If k exists, delete, return (new) list

Add (k,v) to list

Add 1 to size if k wasn't already in

Else add 0 for visibility only

- To **put(k,v)**
  - Delete existing entry for **k**, if any
    - This may produce a new list of item nodes (immutable!)
  - Add new **(k,v)** entry at head of item node list
  - Update stripe size, *also* for visibility

Map C

StripedWriteMap in perspective

- StripedWriteMap design
  - incorporates ideas from Java’s ConcurrentHashMap
  - yet is much simpler (Java’s uses optimistic concurrency, compare-and-swap, week 11-12)
  - but also less scalable
- Is it correct?
  - I think so ...
  - Various early versions weren’t ☹
- Can we test it?
  - We can see if we can break it, week 9
  - Too subtle for ThreadSafe tool (visibility)?

Why is coarse locking so expensive?

- Limited concurrency
  - In SynchMap only 1 thread can work at a time
  - Hence 3, or 31, CPU cores may sit idle
- Increased thread scheduling overhead
  - If lock unavailable, the thread moves to Locking, then to Enabled, then to Running
  - *Context switch* is slow, also causes cache misses
- Atomic operations may be slow on multicore
  - ... and lock taking requires an atomic operation
  - Clearly worse on AMD Opteron than on Intel i7

Goetz p. 229

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
  - (A) Hash map à la Java monitor
  - (B) Hash map with lock striping
  - (C) Ditto with lock striping and non-blocking reads
- **An atomic long with ”thread striping”**
- Shared mutable state is slow on multicore

A striped thread-safe long

- Use case: more writes (**`add`**) than reads (**`get`**)
- Vastly different scalability
  - (a) Java 5's AtomicLong
  - (b) Java 8's LongAdder
  - (c) Home-made synchronized LongCounter
  - (d) Home-made striped long using AtomicLongArray
  - (e) Home-made striped long with scattered allocation
- Ideas
  - (d,e) Use thread's hashCode to reduce update collisions
  - (e) Scatter AtomicLongs to avoid false cache line sharing

TestLongAdders.java

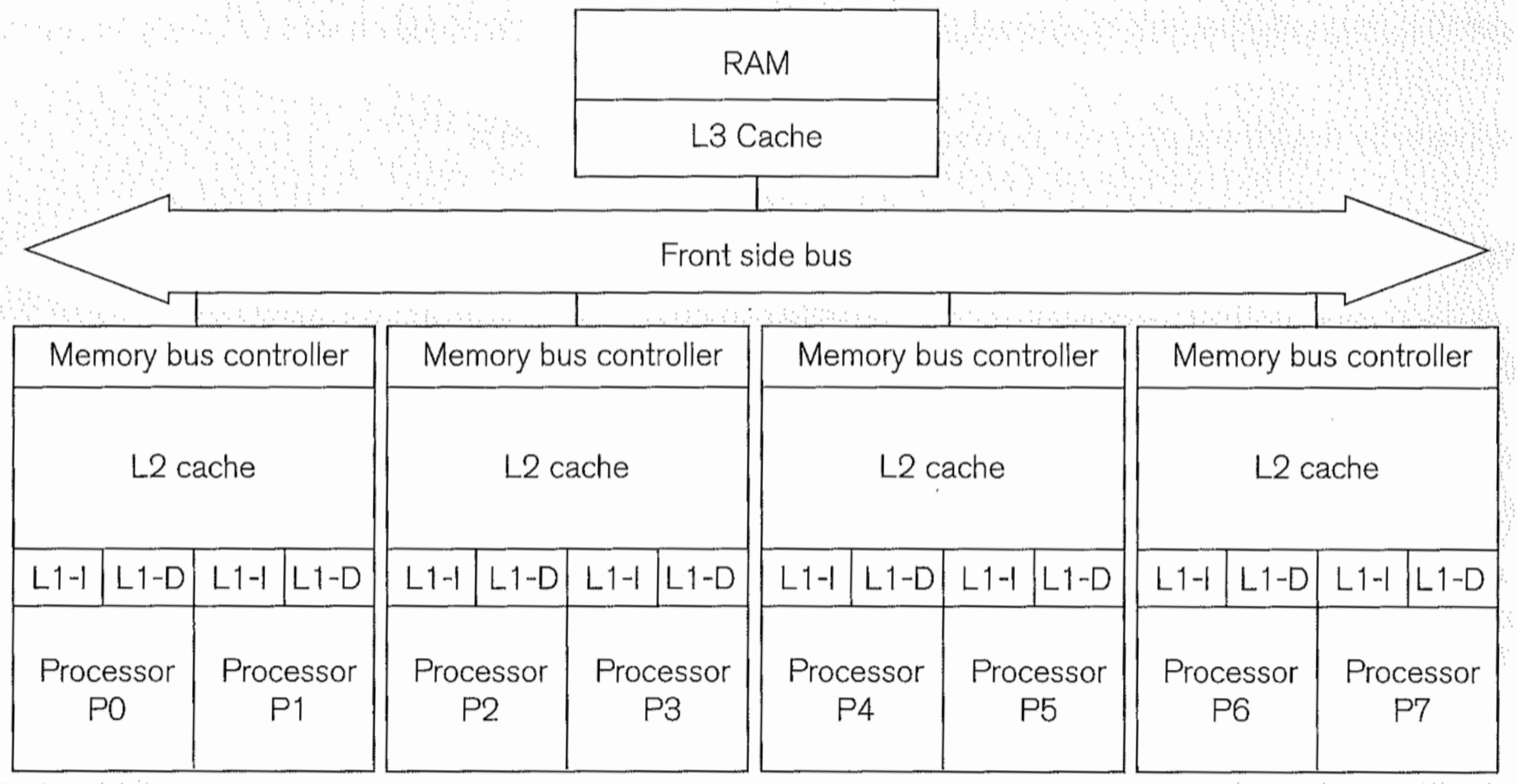
	i7 4c	AMD 32c
(a)	942	3011
(b)	65	54
(c)	1450	14921
(d)	427	1611
(e)	108	922

Wall clock time (ms) for 32 threads making 1 million additions each

Plan for today

- Performance and scalability
- Reduce lock duration, use lock splitting
- Hash maps, a scalability case study
  - (A) Hash map à la Java monitor
  - (B) Hash map with lock striping
  - (C) Ditto with lock striping and non-blocking reads
- An atomic long with ”thread striping”
- **Shared mutable state is slow on multicore**

A typical multicore CPU with three levels of cache

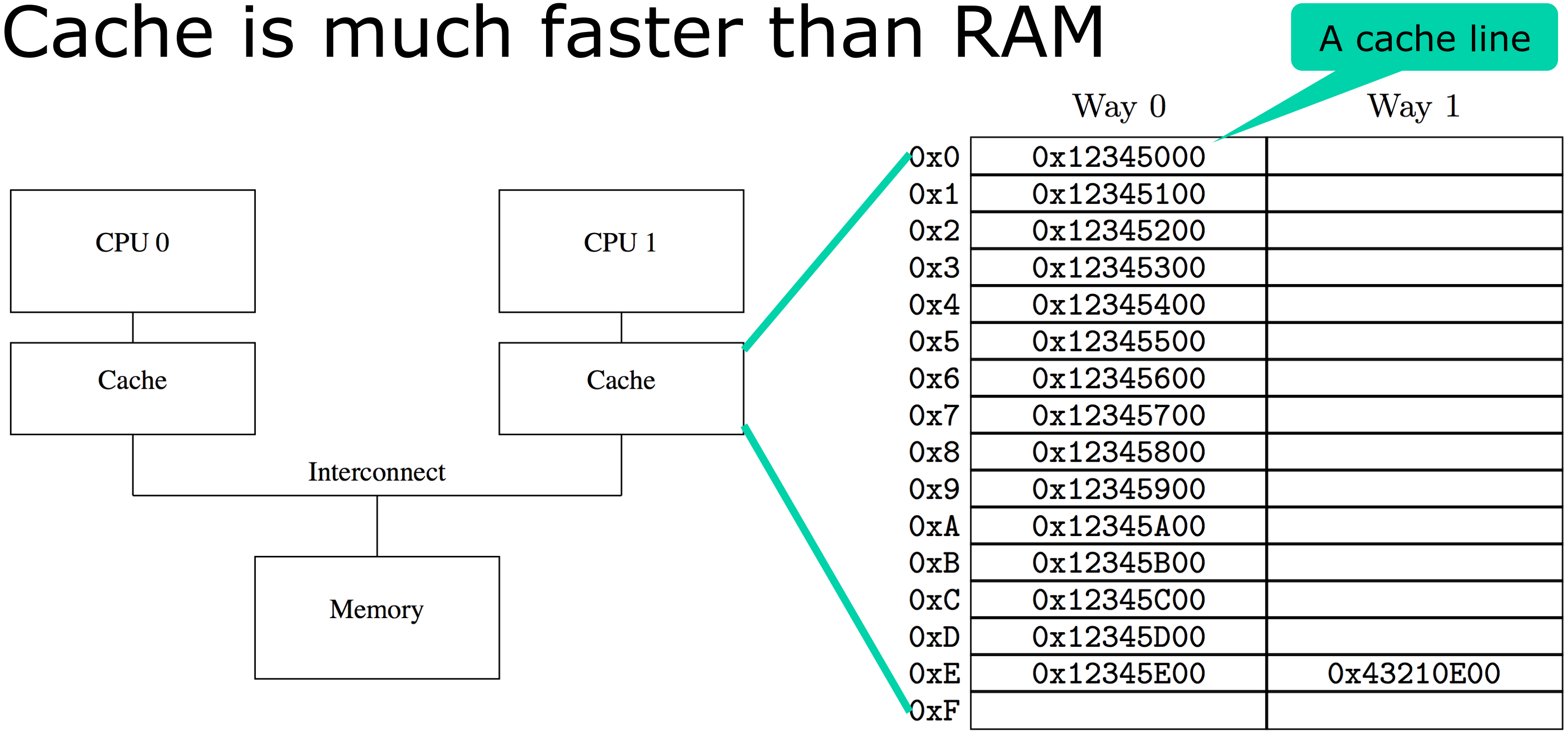


Lin & Snyder 2009, p. 16

- Floating-point add or mul: 0.4 ns
- RAM access: > 100 ns

Fix 1: Each processor core has a cache

- Cache = simple hardware hashtable
- Stores recently accessed values from RAM
- Cache is much faster than RAM



McKenney 2010: Memory barriers

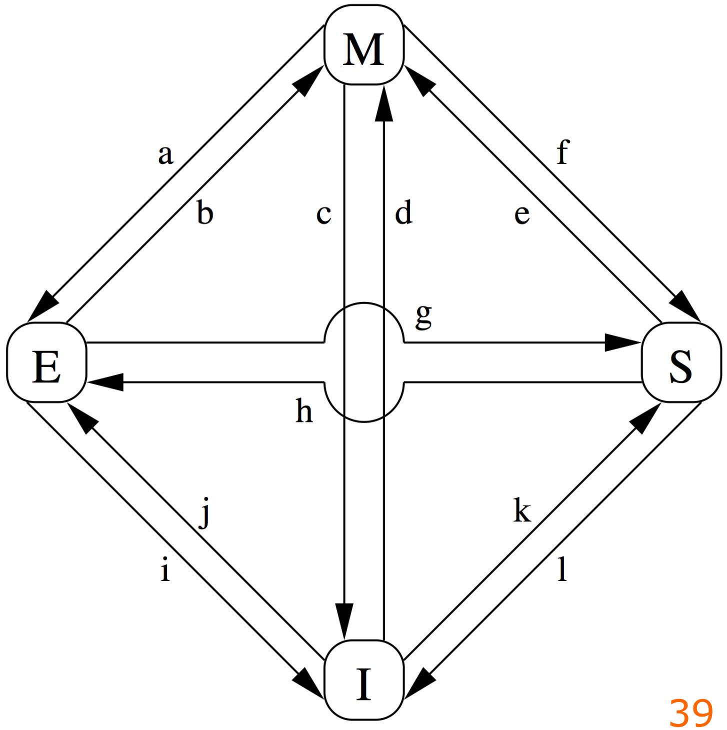
- Two caches may have different values for a given memory address

Fix 2: Get all caches to agree

- Cache coherence; cache line state = M,E,S,I

State	Cache line	Excl	RAM	Read	Write
Modified	Modified by me	Y	not OK	from cache	to cache
Exclusive	Not modified	Y	OK	from cache	to cache -> M
Shared	Others have it	N	OK	from cache	send invalidate
Invalid	Not in use	-	-	elsewhere	send invalidate

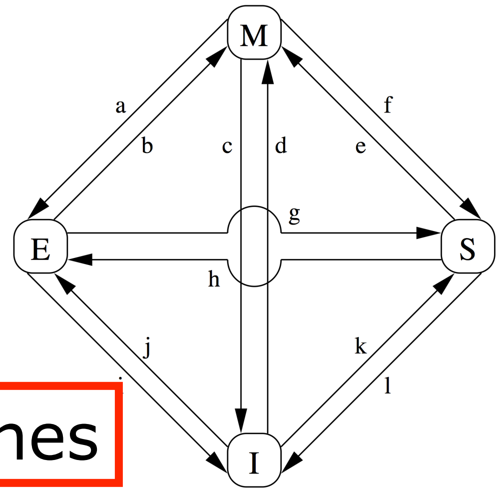
- A cache line
  - has 4 states
  - and 12 transitions a-l
- Cache messages
  - sent by cores to others
  - via memory bus
  - to make caches agree



Transitions and messages

A write in a non-exclusive state requires acknowledge ack* from *all other* cores

Shared mutable state is slow on big machines



		Cause	I send	I receive	My response
M	a	(Send update to RAM)	writeback	-	-
E	b	Write	-	-	-
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack

One more performance problem: "false sharing" because of cache lines

- A cache line typically is 32 bytes
  - gives better memory bus utilization
  - prefetches data (in array) that may be needed next
- Thus invalidating one (8 byte) long may invalidate the neighboring 3 longs in an array
- Frequently written memory locations should not be on the same cache line

```
for (int stripe=0; stripe<NSTRIPES; stripe++) {
  // Believe it or not, this may speed up the code,
  // presumably because it avoids false sharing:
  new Object(); new Object(); new Object(); new Object();
  counters[stripe] = new AtomicLong();
}
```

This week

- Reading
  - Goetz et al chapter 11 + 13.5
  - Optional: McKenney: *Memory barriers*

- Exercises
  - Make sure you can write well-performing and scalable software using lock striping, immutability, Java atomics, and visibility rules; finish StripedMap and StripedWriteMap classes

- Read before next lecture (24 October)
  - Goetz et al chapter 9