

# Practical Concurrent and Parallel Programming 6

Riko Jacob  
IT University of Copenhagen

Friday 2017-10-13

# Plan for today

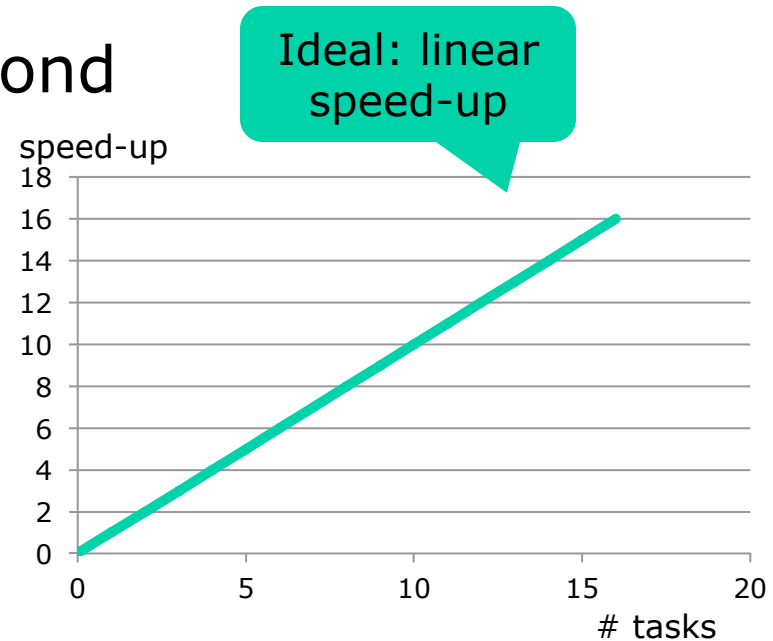
- Performance and scalability
- Reduce lock duration by lock splitting
- Hash maps, a scalability case study
  - (A) Hash map à la Java monitor
  - (B) Hash map with lock striping
  - (C) Ditto with lock striping and non-blocking reads
  - (D) Java 8 library's ConcurrentHashMap

Based on slides by  
Peter Sestoft

# 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

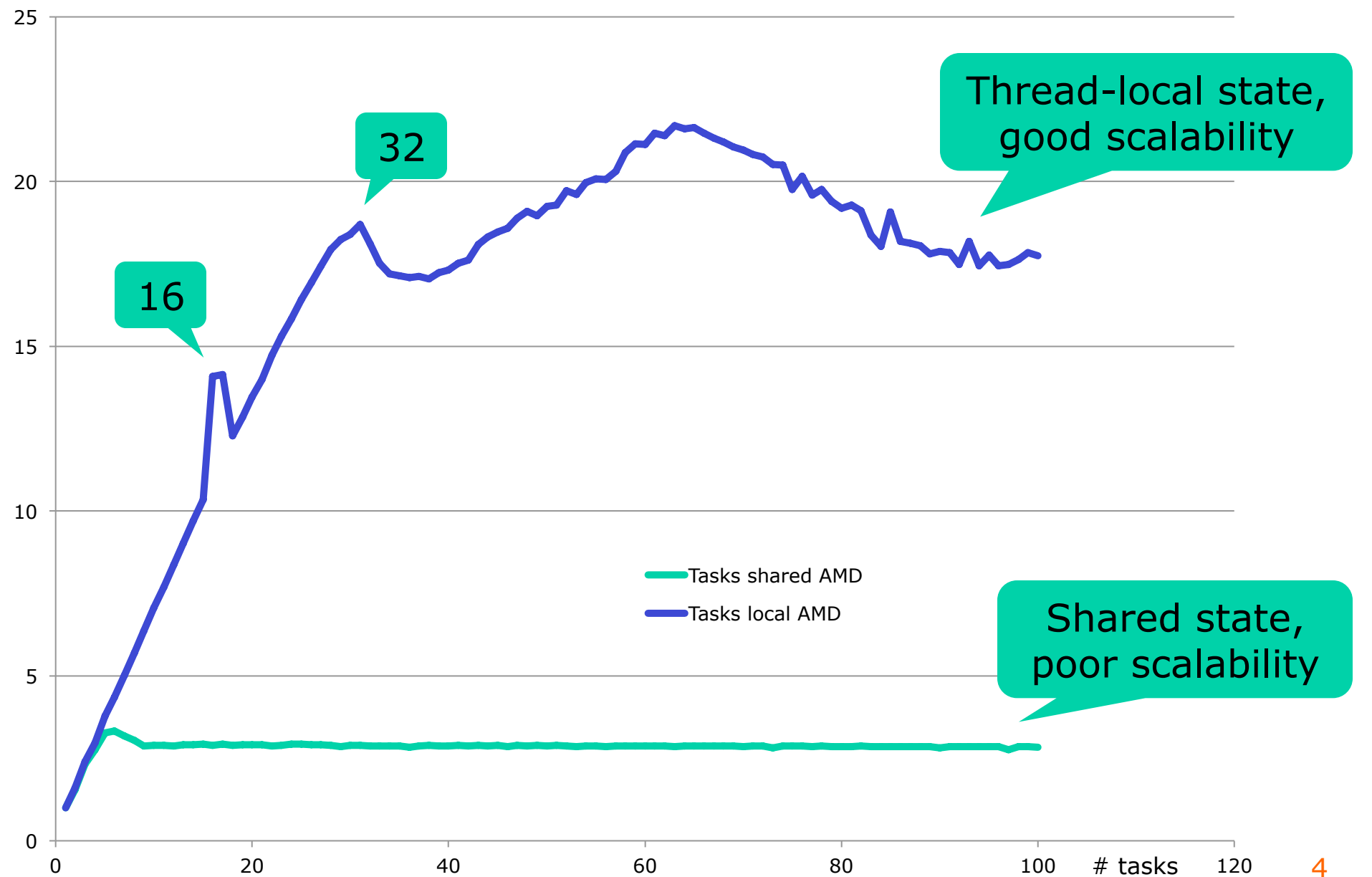


- One may sacrifice performance for scalability
  - OK to be slower on 1 core if faster on 2 or 4 or ...
  - Requires rethinking our “best” sequential code

# Scalability of prime counting

## AMD Opteron, 32 cores

speed-up



# 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  $\leq 1/(F+(1-F)/N) = 1/(0.8+0.2/30) = 1.24$
- 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

- 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

```
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

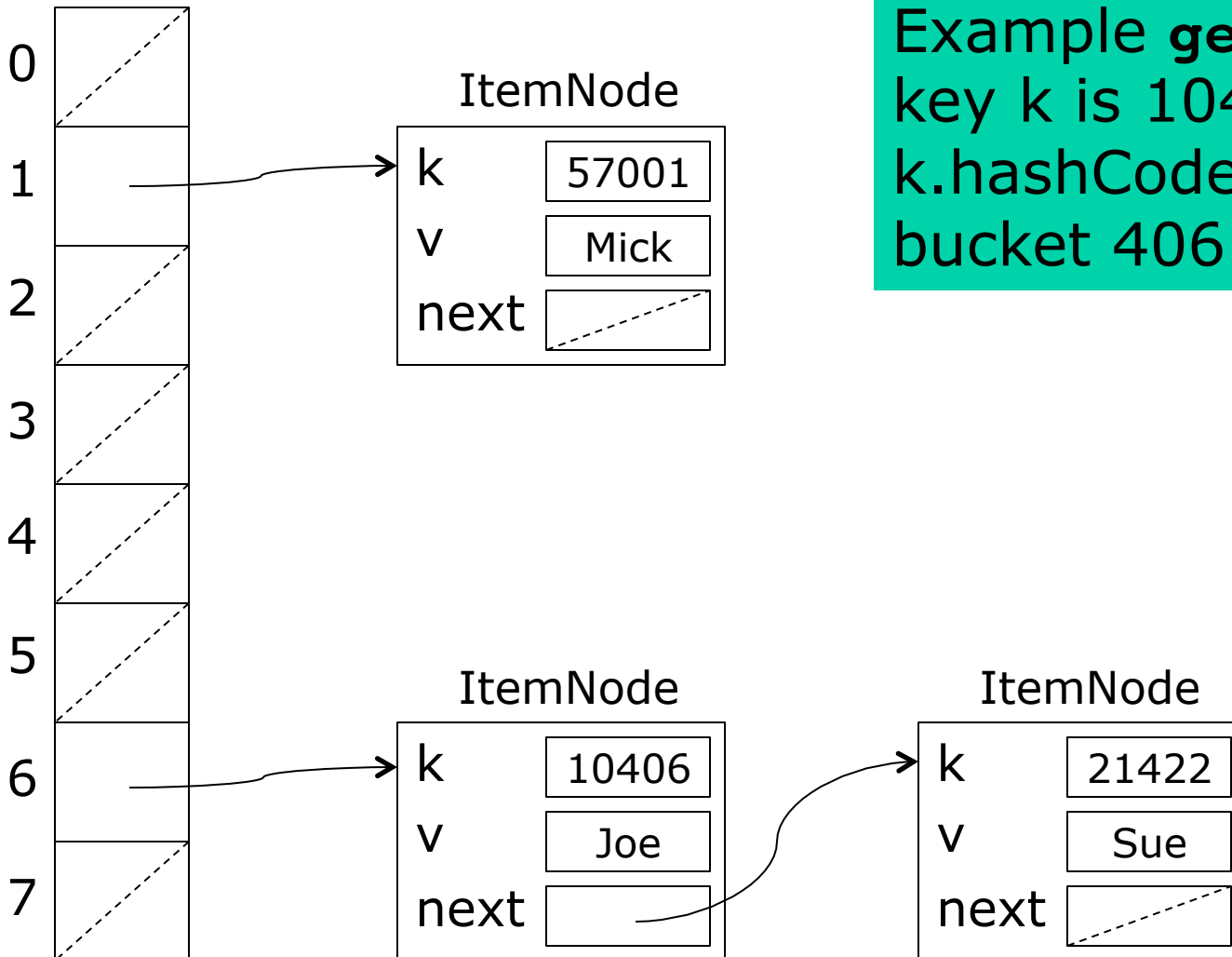


# Plan for today

- 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

# A hash map = buckets table + item node lists

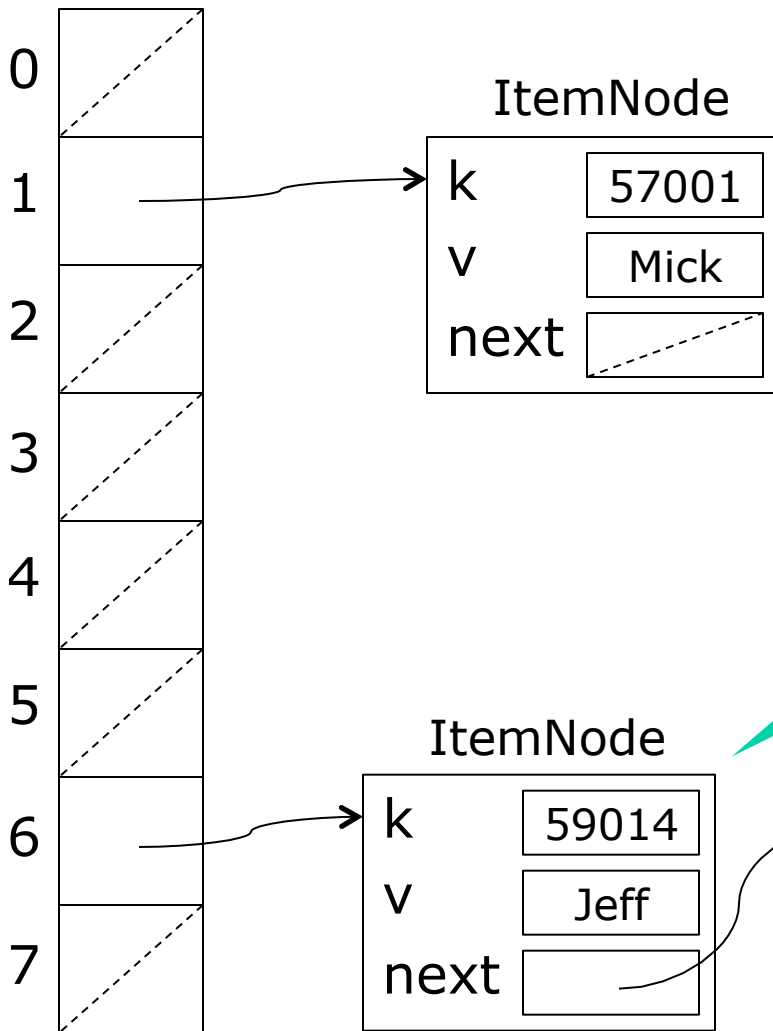
buckets



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

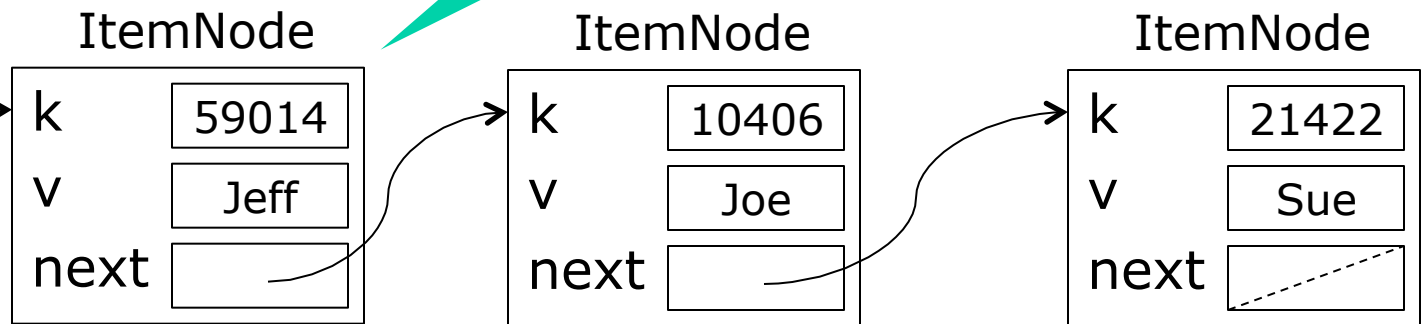
# Insertion into the hashmap

buckets



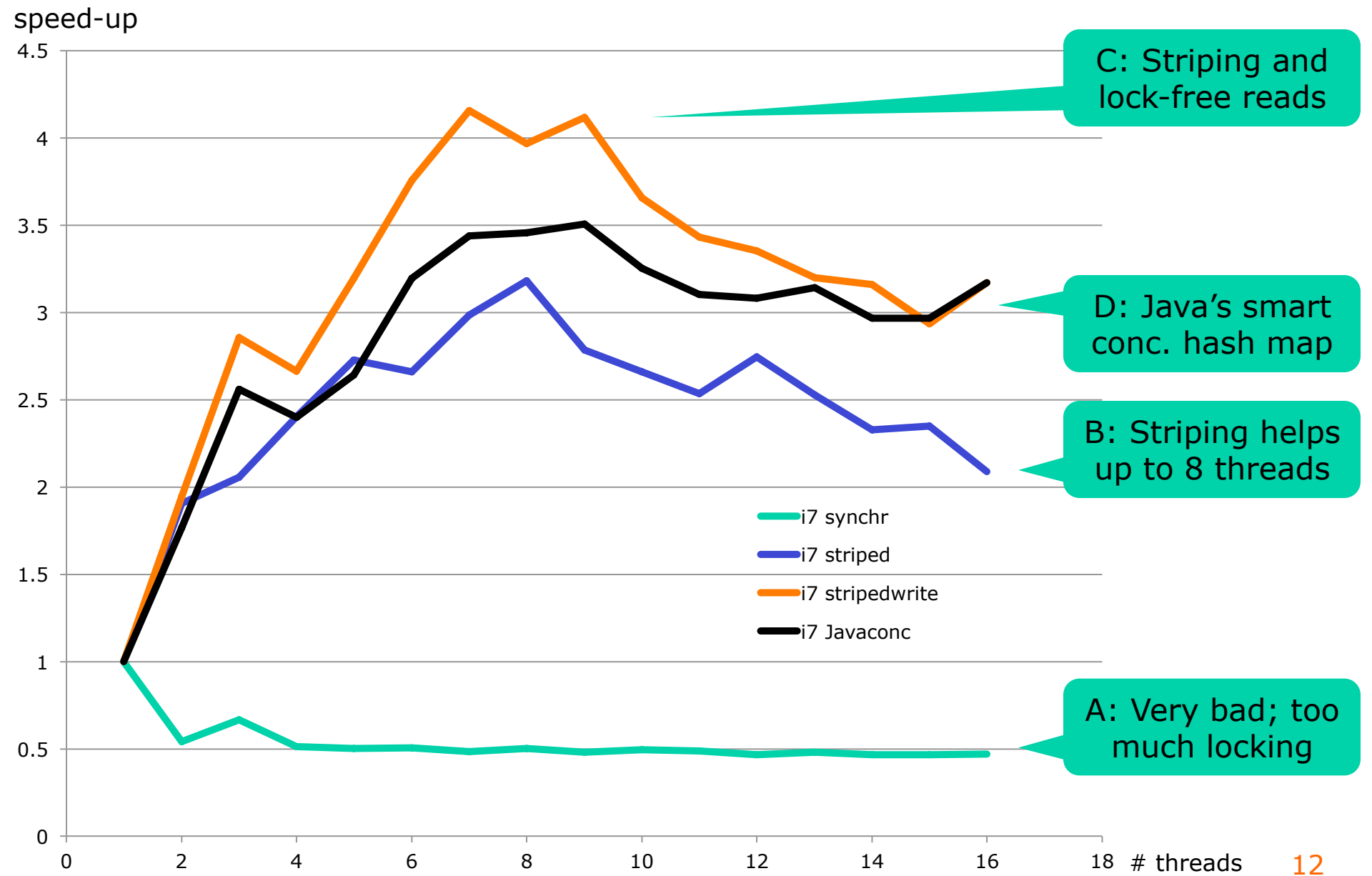
```
put(59014, "Jeff")  
key k is 59014  
k.hashCode() is 14  
bucket 14 % 8 is 6
```

New item



# Scalability of hash maps

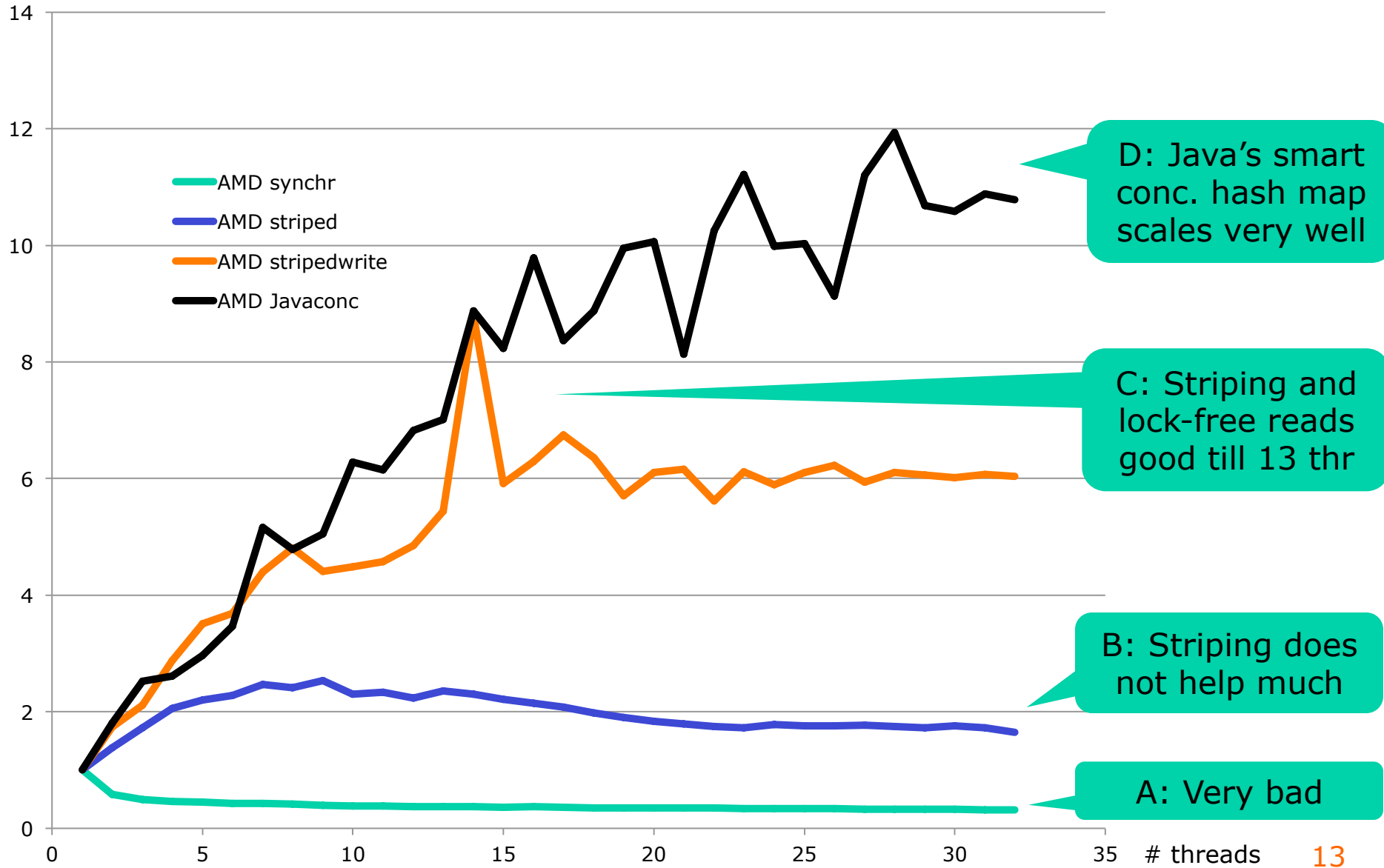
## Intel i7 w 4 cores & hyperthreading



# Scalability of hash maps

## AMD Opteron w 32 cores

speed-up




# Our map interface

- Reduced version of Java interface Map<K,V>

```
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));
```

# 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

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) { ... }  
}
```

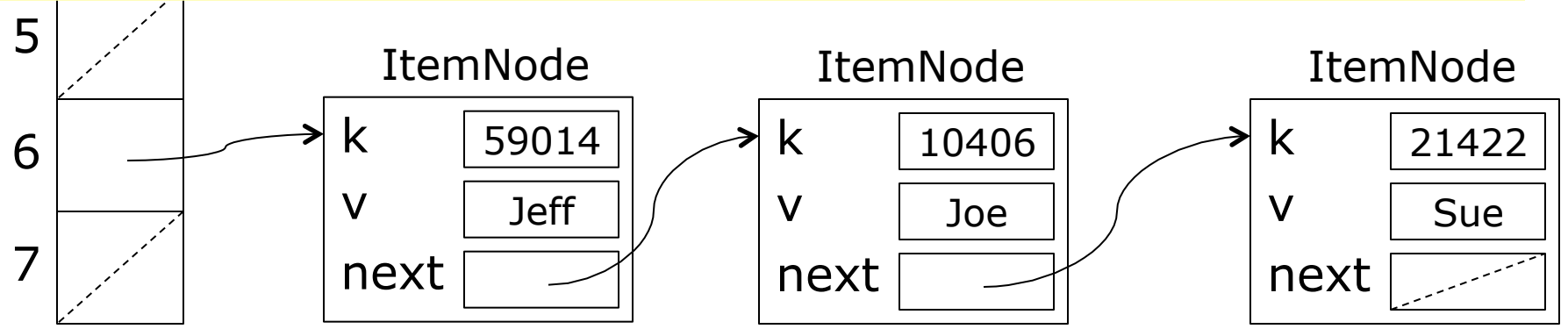
# Implementing containsKey

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



# 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

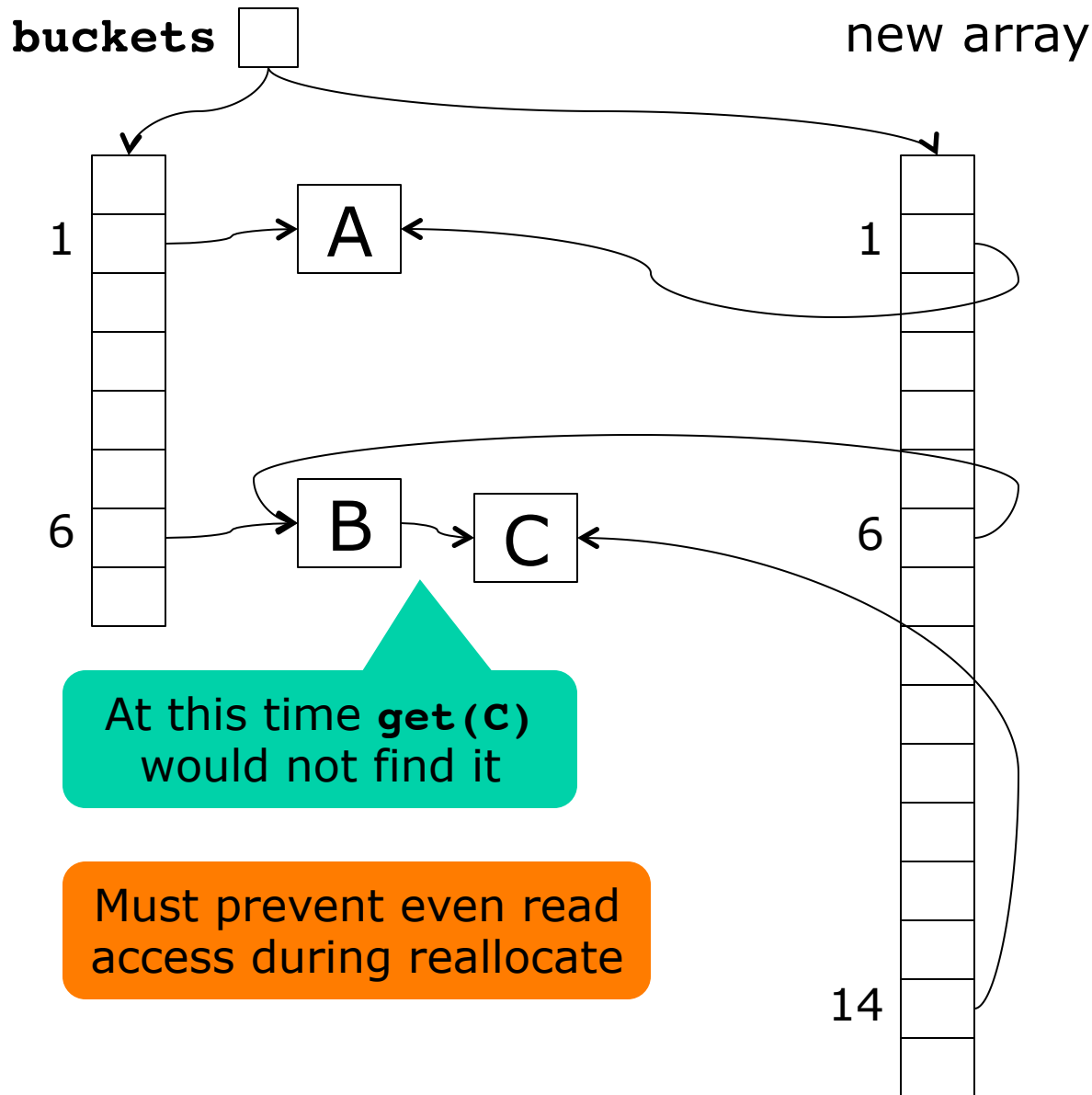
Else add new  
item node at  
front of list

- 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 = getHash(k) % 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

# Double buckets array (mutating)



# ReallocateBuckets implementation

```
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

TestStripedMap.java

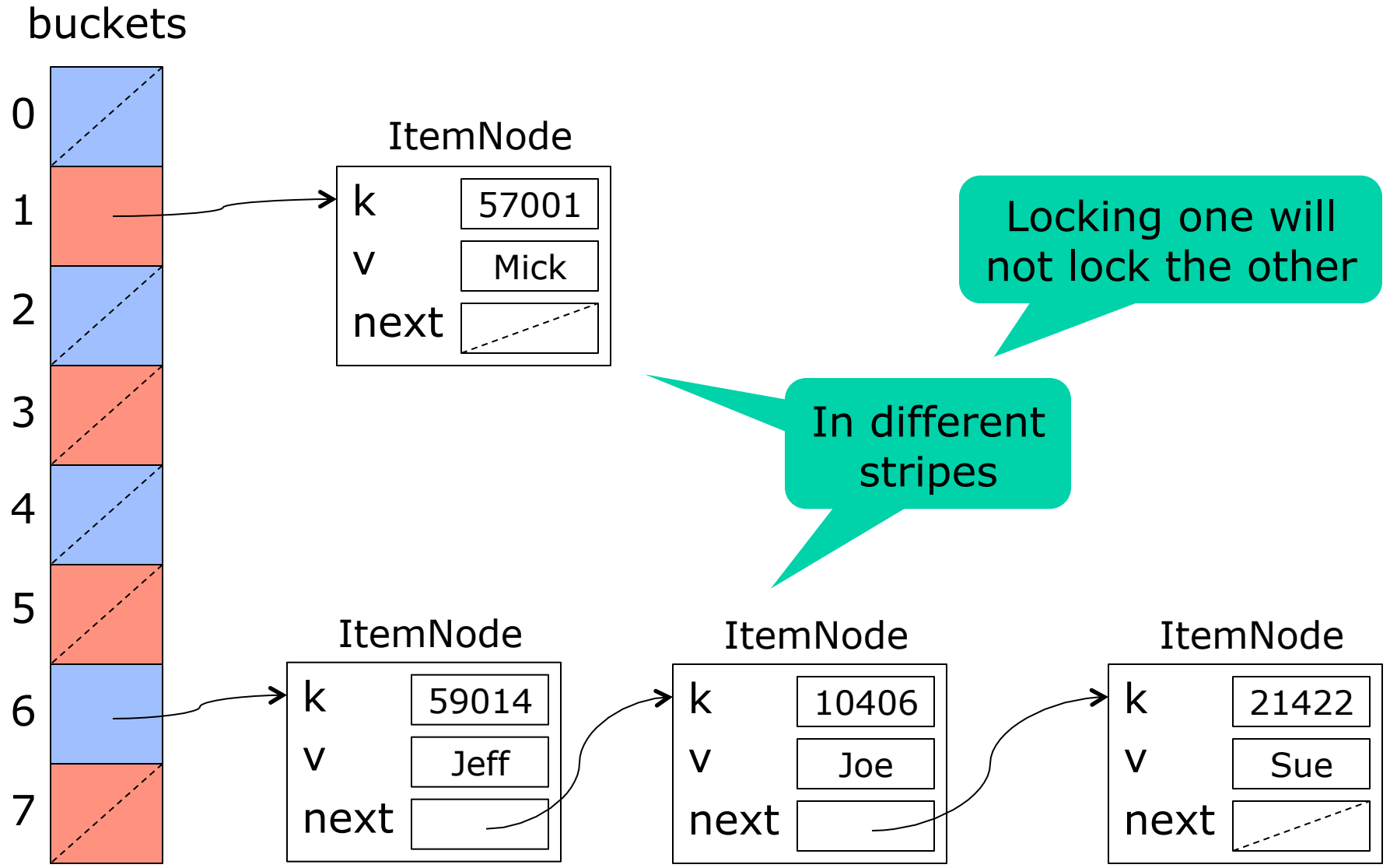
- Seems efficient: reuses each ItemNode
  - Links it into an new item node list
  - So destructs the old item node list
  - So read access impossible during reallocation
  - Good 1-core performance, but bad scalability

# 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 16<sup>th</sup> 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



# 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 **locks[stripe]** ensures
  - atomic access within each stripe
  - visibility of writes to readers

# 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;
    }
}
```

- 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

# 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

# Reallocating buckets

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

```
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);
        }
}
```

TestStripedMap.java

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

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

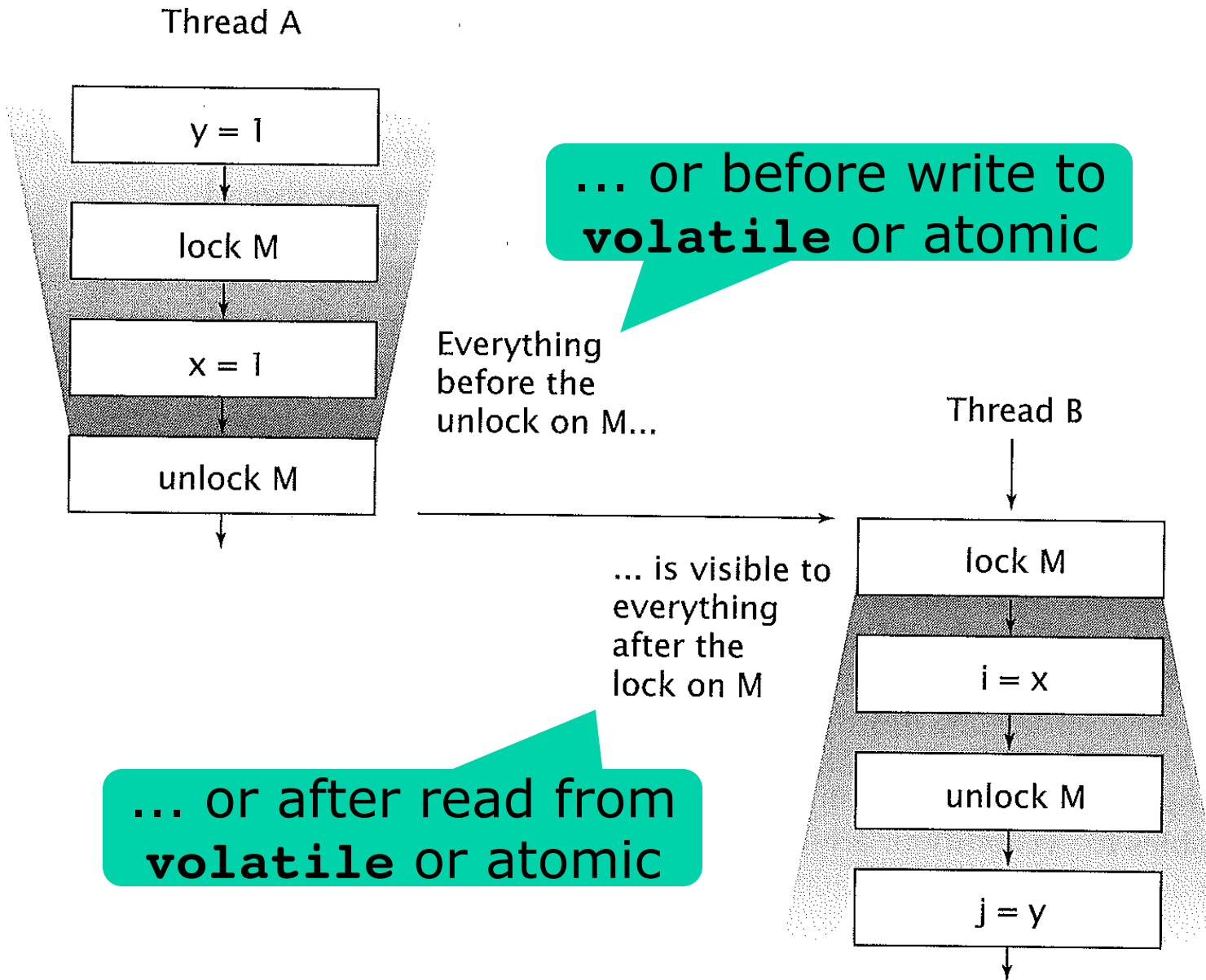
All locks held when  
calling `action.run()`

# 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



# Locking the stripes only on write

```
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>  
}
```

TestStripedMap.java

```
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) ...  
}
```

Immutable

```
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

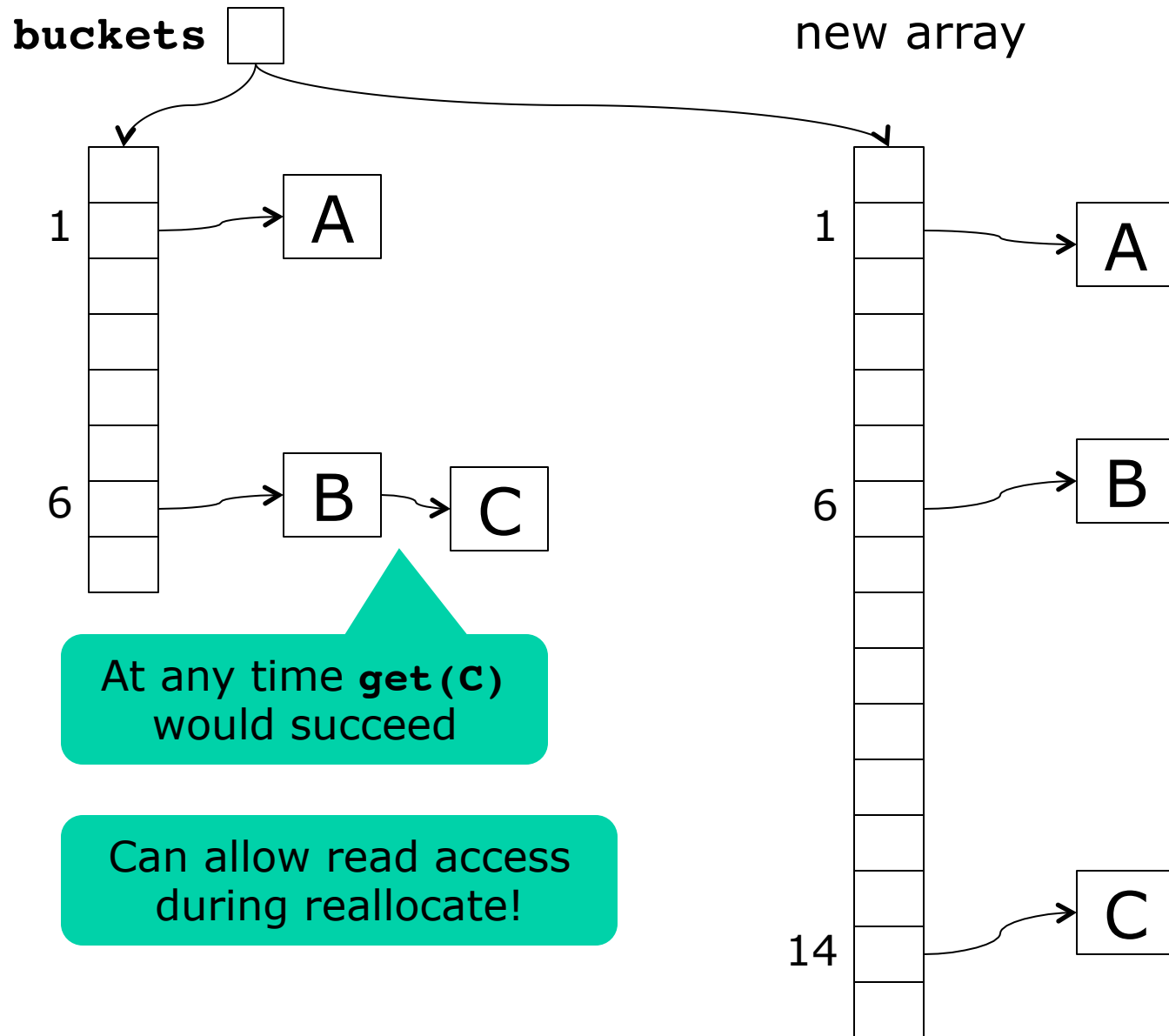
Else add 0 for  
visibility only

## • 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



# Double buckets array (non-mutating)



# StripedWriteMap in perspective

- StripedWriteMap design
  - incorporates ideas from Java's ConcurrentHashMap
  - yet is much simpler (Java's uses optimistic concurrency, compare-and-swap, week 10-11)
  - but also less scalable
- Is it correct?
  - I think so ...
  - Various early versions weren't ☹
- Can we test it?
  - Yes, week 8

# Comparison of concurrent hashmaps

	A	B	C	D
Concurrent reads	✗	✓	✓	✓
Concurrent reads and writes	✗	✓	✓	✓
Reads during reallocate	✗	✗	✓	?
Writes during reallocate	✗	✗	✗	?

- (A) Hash map à la Java monitor
- (B) Hash map with lock striping
- (C) Ditto with lock striping and non-blocking reads
- (D) Java 8 library's ConcurrentHashMap

# This week

- Reading
  - Goetz et al chapter 11, 13.5
- 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 (9 October)
  - Goetz et al chapter 9

# Massively parallel hardware

- Nvidia Tesla K40 GPGPU
  - GPGPU = General Purpose Graphics Processing Unit
  - has 15 Symmetric Multiprocessors
  - each has 192 cores, in total 2880 float cores
  - clock speed 745 MHz
  - total throughput 4,290,000,000,000 float ops/sec
  - or 1,430,000,000,000 double ops/sec
  - but difficult to achieve in practice
- Standard 4-core Intel i7 CPU
  - $2.4 \text{ GHz} \times 4 \times 2 = 19,200,000,000$  double ops/sec, using vector AVX instructions
- GPU is 74 x faster than CPU, in theory

# Nvidia GK110 GPGPU

