Practical Concurrent and Parallel Programming VII

Performance and scalability using locking

IT University of Copenhagen

Friday 2020-10-02

Kasper Østerbye

Starts at 8:00

Plan for today

Follow up on last week

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

Follow up and loose ends

- Thanks for the questions in the forum! Thanks to Holger for answering many of them, but you could too
- Perhaps some follow up on particular strange exercises

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

- Maybe OK to be slower on 1 core if faster on 2 or 4 or …
- Requires rethinking our “best” sequential code speed-up 18

What limits performance?

CPU-bound

- Eg. counting prime numbers
- To speed up, add more CPUs (cores)

Input/output-bound

- Eg. fetching webpages or reading from disk
- To speed up, use more tasks

Synchronization-bound

- Eg. image segmentation using shared data structure
- **To speed up, improve shared data structure** (*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

AtomicInteger and AtomicLong

Not implemented using locks! - optimistic concurrency

More efficient in case of low contention

```
// in class AtomicLong - pre Java 8
public final long incrementAndGet() {
    for (;;) {
        long current = get();
        long next = current + 1;
        if (compareAndSet(current, next))
          return next;
    }
} // [https://dzone.com/articles/how-cas-compare-and-swap-java]
```

The operation `ai,compareAndSet(a,b)` is **atomic**. If the current value of `ai` equals `a`, then `ai` is set to `b`.

The operation `ai.compareAndSet(a,b)` has support in the hardware of most modern CPU's.

From Java8 and on, the method `incrementAndGet()` is implemented natively in the JVM.

Back to locks

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); // here we must lock
    return location != null && Pattern.matches(regexp, location);
  } // Holds lock for the entire method
}
```

Better (shorter locking period)

```
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) { // Only locks during get
      location = attributes.get(key);
    }
    return location != null && Pattern.matches(regexp, location);
  }
}
```

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) { // locks server
    users.add(u);
  }
  public synchronized void addQuery(String q) { // locks server
    queries.add(q);
  }
  public synchronized void removeUser(String u) {
  ...
```

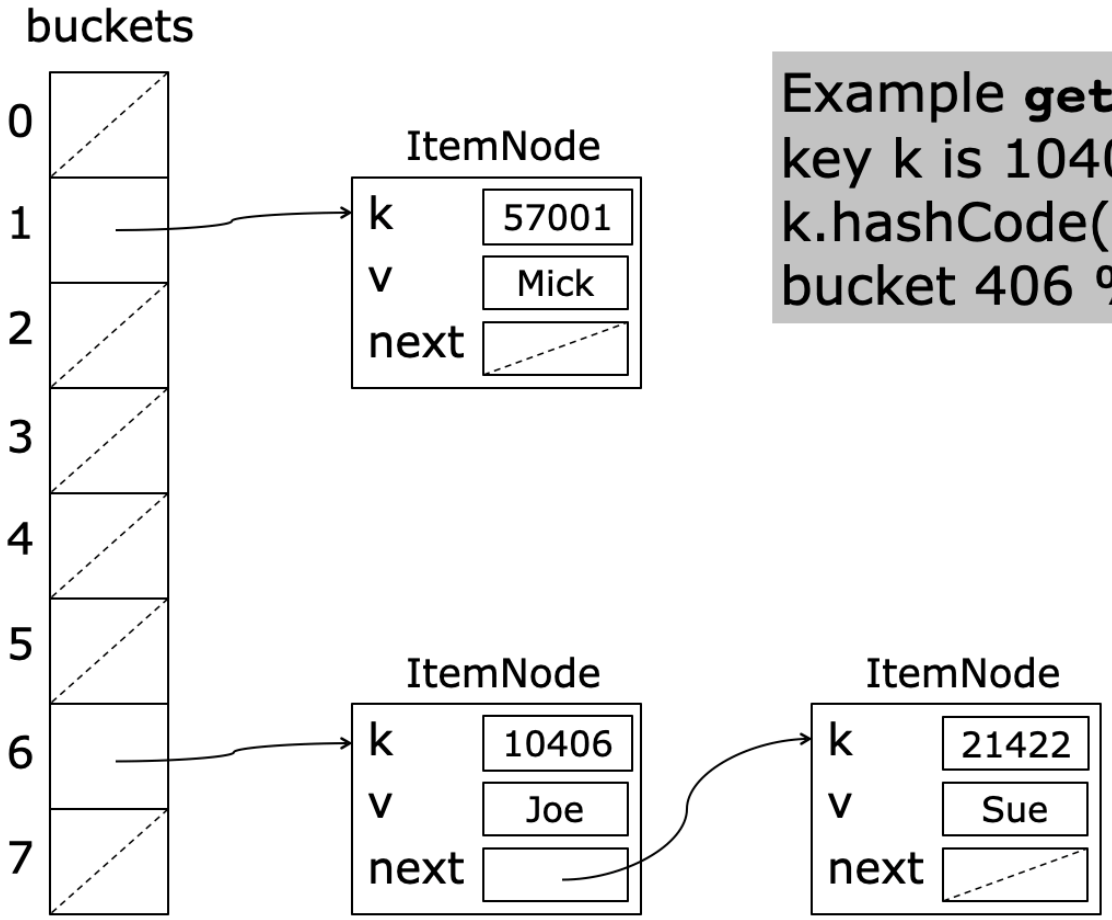
Better (addUser and addQuery can run concurrently)

```
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); } // locks only users
  }
  public void addQuery(String q) {
    synchronized (queries) { queries.add(q); } // locks only queries
  }
  ...
```

Hash maps, a scalability case study

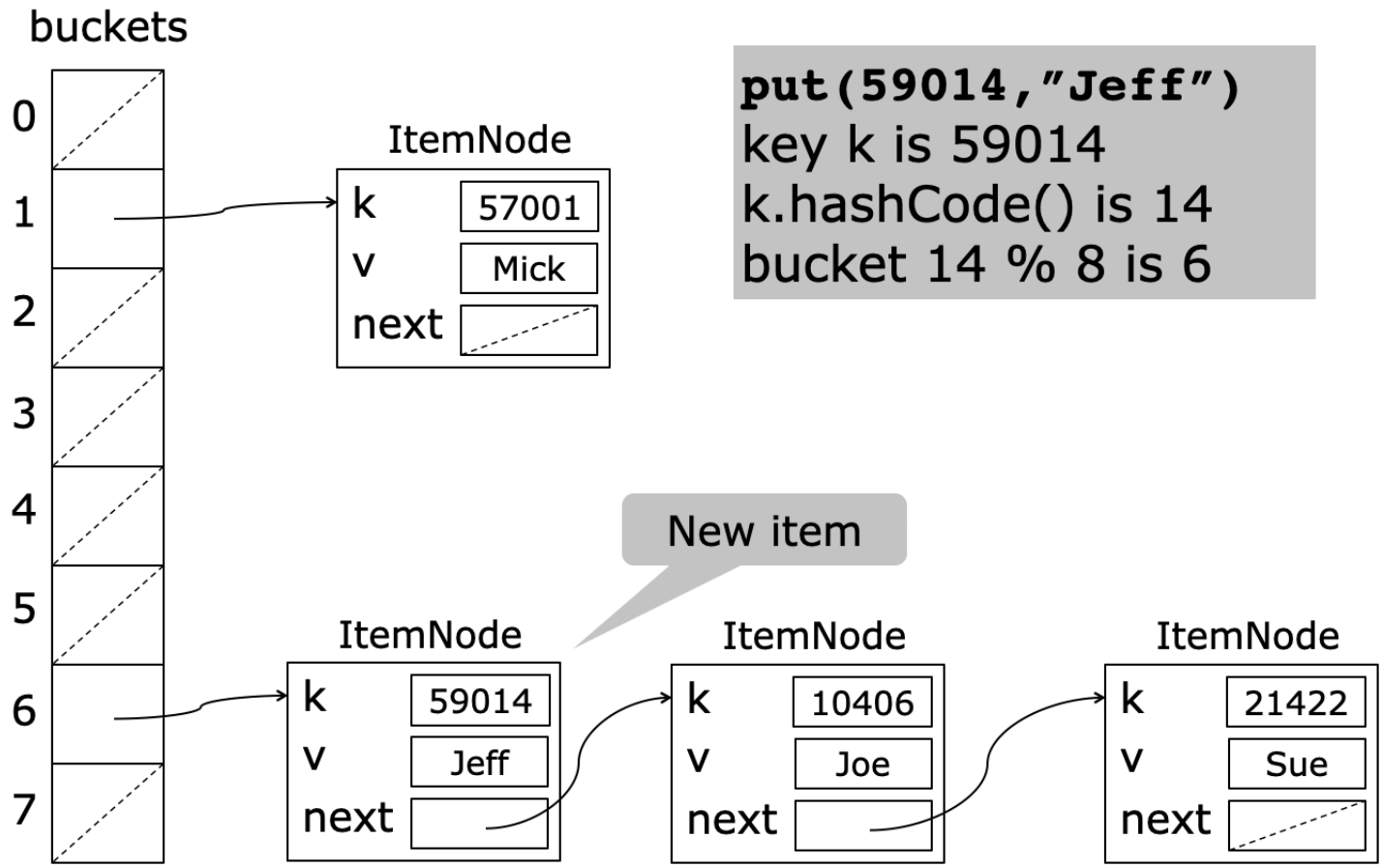
- (A) Hash map à la Java monitor
- (B) Hash map with lock striping
- (C) Ditto with lock striping and non-blocking reads
- (D) Java 8 library's ConcurrentHashMap

A hash map = buckets table + item node lists



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

Hashmap insert



Our map interface

Reduced version of Java interface Map

```
 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);
}
```

```
map.forEach((k, v) -> System.out.printf("%10d maps to %s%n", k, v));
// Same as...
for (Entry (k,v) : map) System.out.printf(...);
```

Synchronized map implementation

```
static class ItemNode<K,V> { //  Visibility depends on synchronization elsewhere
private final K k;
private V v;
private ItemNode<K,V> next;
public ItemNode(K k, V v, ItemNode<K,V> next) { ... }
}
```

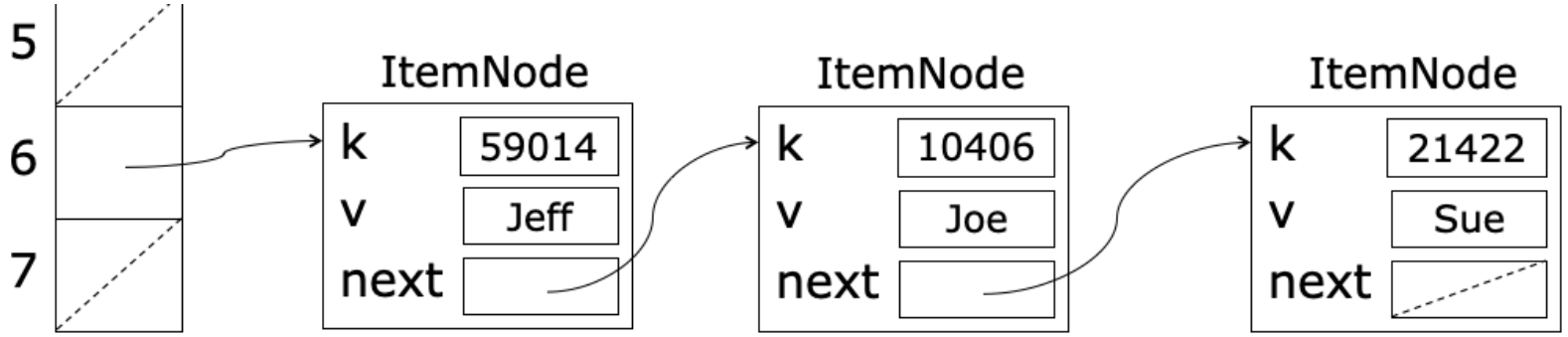
```
// implemented using Java monitor pattern
class SynchronizedMap<K,V> implements OurMap<K,V> {
  private ItemNode<K,V>[] buckets; // guarded by this
  private int cachedSize; // guarded by this
  public synchronized V get(K k) { ... }
  public synchronized boolean containsKey(K k) { ... }
  public 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

```
public synchronized boolean containsKey(K k) {
  final int h = getHash(k); // helper method to prevent neg. hashcodes
  final int bucket = h % buckets.length; //
  return ItemNode.search(buckets[bucket], k) != null;
}
```

Linked list implementation with search loop

```
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;
}
```



Implementing putIfAbsent

```
public synchronized V putIfAbsent(K k, V v) {
  final int h = getHash(k), bucket = h % buckets.length;
  // Search bucket's node list
  ItemNode<K,V> node = ItemNode.search(buckets[bucket], k);
  if (node != null) {
    return node.v; // If key exists, return value
  } else { // Add new item node at front of list
    buckets[bucket] = new ItemNode<K,V>(k, v, buckets[bucket]);
    cachedSize++;
    return null;
  }
}
```

- 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)

Insert figure

Reallocate buckets 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) { \\ for each item node
      // compute new hash (new bucket number)
      final int newHash = getHash(node.k) % newBuckets.length;
      // Insert into new bucket (we do not allocate a new Node!)
      ItemNode<K,V> next = node.next;
      node.next = newBuckets[newHash];
      newBuckets[newHash] = node;
      node = next;
    }
  }
  buckets = newBuckets; // Swap in new buckets
}
```

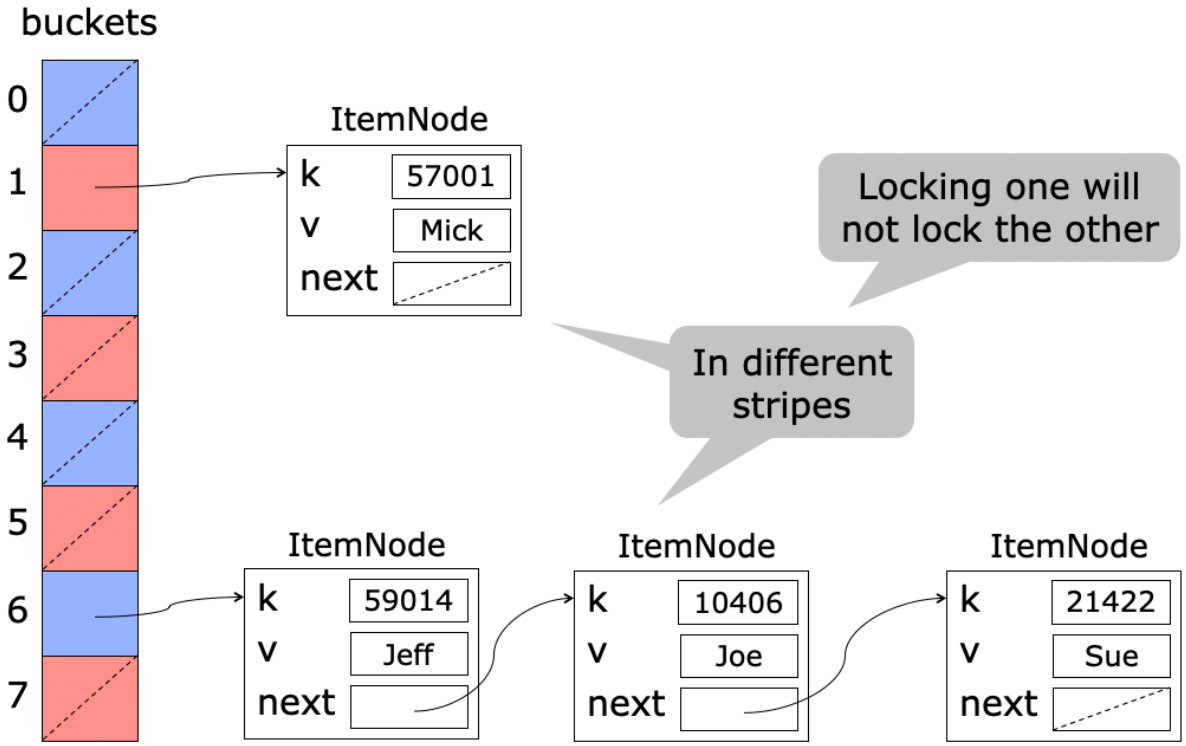
- Seems efficient: reuses each ItemNode
- Links it into an new item node list
- Discards the old item node list
- Read access impossible during reallocation
- Good 1-core performance, but bad scalability

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 `Q: Why?`
  - guard every 16^{th} bucket with the same lock
  - locks[0] guards bucket 0, 16, 32, …
  - locks[1] guards bucket 1, 17, 33, …
- With high probability `Q: What do we mean?`
  - 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 buckets



We could have made first half blue, last half red

Striped hashmap implementation

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

  // Methods no longer synchronized
  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) { ... }
}
```

- 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; // Does this need to be synchronized?
    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 while calling containsKey?

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]) { // Why do we need to lock?
    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);
  final int stripe = h % lockCount;

  synchronized (locks[stripe]) { // lock only stripe
    final int hash = h % buckets.length;
    final ItemNode<K,V> node = ItemNode.search(buckets[hash], k);
    if (node != null) { // If k exists, update value to v, return old
      ...
    } else { // Else add new item node (k,v)
      ...
    }
  }
}
```

Striped put(k,v)

```
public V put(K k, V v) {
  final int h = getHash(k), stripe = h % lockCount;
  synchronized (locks[stripe]) { // lock only stripe
    final int hash = h % buckets.length;
    final ItemNode<K,V> node = ItemNode.search(buckets[hash], k);
    if (node != null) { // If k exists, update value to v, return old
      V old = node.v;
      node.v = v;
      return old;
    } else { // Else add new item node (k,v)
      buckets[hash] = new ItemNode<K,V>(k, v, buckets[hash]);
      sizes[stripe]++; // add to size of stripe
      return null;
    }
  }
}
```

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);
}
// This method is the one doing the work
private void lockAllAndThen(int nextStripe, Runnable action) {
  if (nextStripe >= lockCount)
    action.run();
  else
    synchronized (locks[nextStripe]) {
      lockAllAndThen(nextStripe + 1, action);
}}
```

Overall effect of calling lockAllAndThen(0, action)

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

All locks held when calling action.run()

Why recursion?

- The only way to lock is by using `synchronized(lock)`.

```
private void lockAllAndThen(Runnable action) {
  for (int i = 0; i < lockCount; i++){
    // uups, no way to say lock(locks[i])
  }
  action.run();
  for (int i = lockCount-1; i >= lockCount; i--){
    // uups, no way to say unlock(locks[i])
  }
```

Passing an action as parameter

Caller (of `lockAllAndThen`) knows:
  What to do (the `runnable`), but not when

Implementer knows:
  When, but not what

This is an important idea in making your programs general.

Also with exceptions.

- Implementer of method know when something goes wrong, but not what to do
  - Throws exception
- Caller of method know how to handle, but not when something goes wrong
  - catches exeption

Read-write locks

Allowing many to *read*, while allowing only one to *write* is common.

The java library has support for this in the form of `ReadWriteLock` with an implementation called `ReentrantReadWriteLock`. It is described in Goetz chap 13.5.

```
public class ReadWriteMap <K,V> {
    private final Map<K, V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock(), w = lock.writeLock();

    public V get(K key) {
        r.lock();
        V res = return map.get(key);
        r.unlock();
        return res;
    }

    public V put(K key, V value) {
        w.lock();
        V old =  map.put(key, value);
        w.unlock();
        return old;
    }
    ...
}
```

Small detour

Notice the pattern:

```
public V someMethod( arguments) {
    r.lock();
    // Do stuff
    r.unlock();
}
```

we rewrite this to the following, making sure to release the lock

```
public V someMethod( arguments) {
    withLockDo(r, () -> {
      // Do stuff
    });
}
```

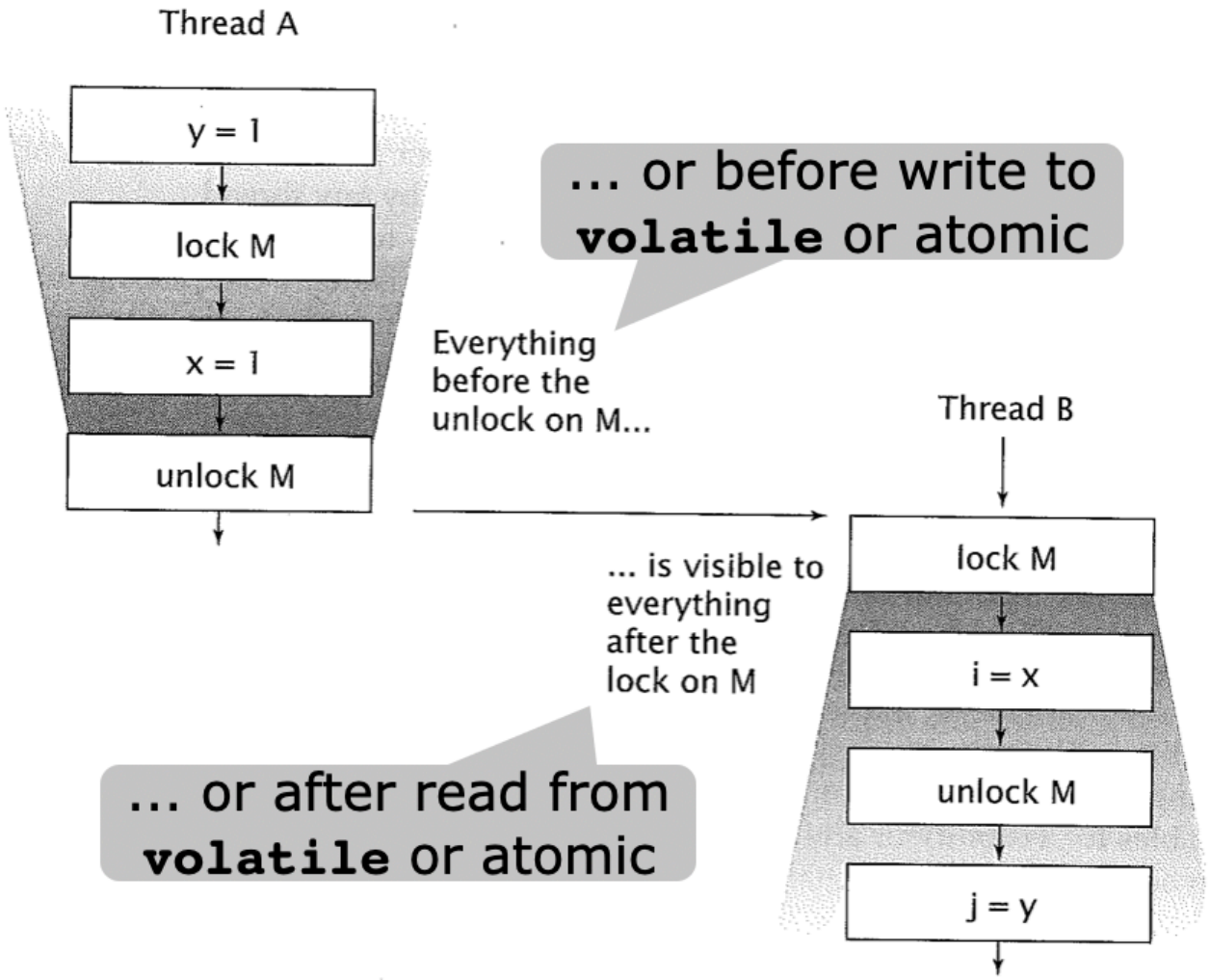
```
public static void withLockDo(Lock l, Runnable action){
  l.lock();
  try {
      action.run();
  } finally {
      l.unlock();
  }
}
```

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 (*atomically*) hash = getHash(k) % buckets.length
  - Access (*atomically*) 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

Visibility by lock, volatile, or atomic



Immutable ItemNode

```
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) ...
}
```

Immutables are often a good idea for parallelism

- if you have more reads then updates
- it trades copying nodes against locks

Holder is used to implement **out** parameters in Java.

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

In short - an **out** paramter is used to return a value from a method.
It is used in languages which can only return one value.

StripedWriteMap < K ,V >

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

Class AtomicIntegerArray
An int array in which elements may be updated atomically.

Stripe-locking 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**

```
public V put(K k, V v) {
  final int h = getHash(k), stripe = h % lockCount;
  synchronized (locks[stripe]) { // lock 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 is a copy of the list with node containing k removed
    // if k is not found, delete returns the existing list
    final ItemNode<K,V> newNode = ItemNode.delete(node, k, old);
    bs[hash] = new ItemNode<K,V>(k, v, newNode); // Add
    // if newNode == previous node, nothing was deleted
    // update size for visibility in any case
    sizes.getAndAdd(stripe, newNode == node ? 1 : 0);
    return old.get();
  }
}
```

Double buckets array (non-mutating)

StripedWriteMap in perspective

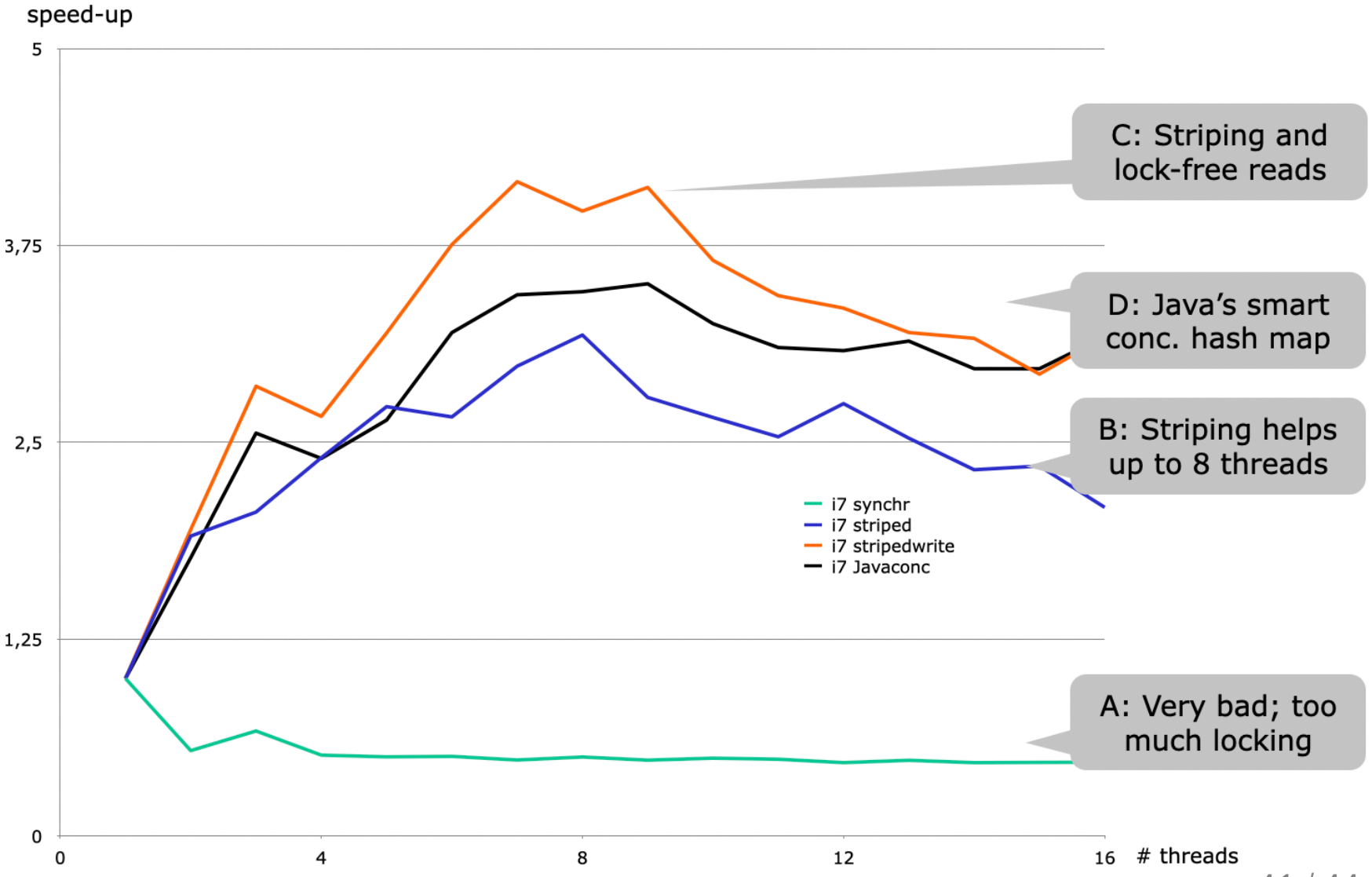
- StripedWriteMap design
  - incorporates ideas from Java's ConcurrentHashMap
  - yet is much simpler (Java's uses optimistic concurrency, compare-and-swap)
  - but also less scalable
- Is it correct?
  - I think so …
- Can we test it? * Yes, we will look at that later in this course

Comparison of concurrent hashmaps

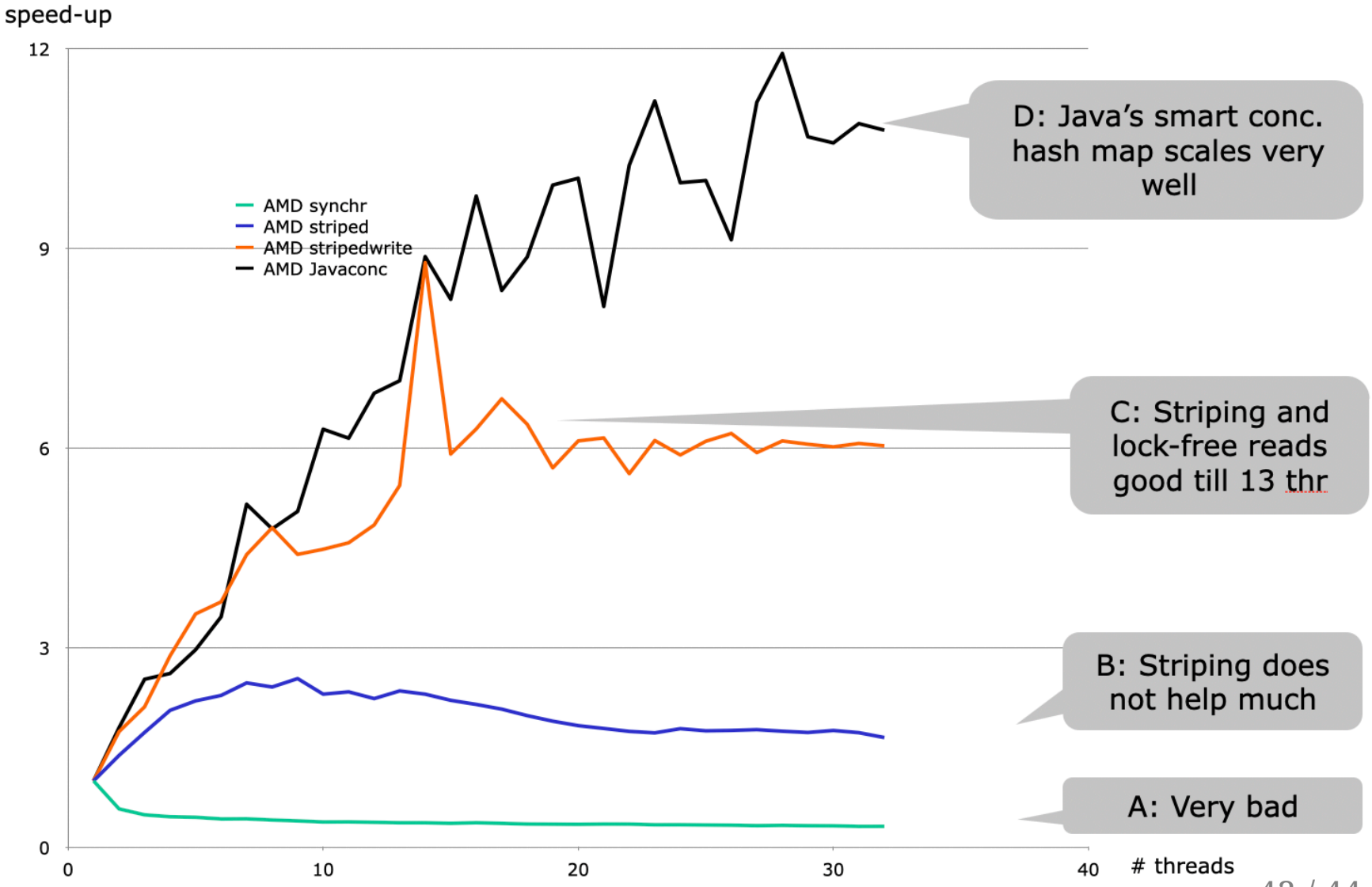
	A	B	C	D
Concurrent reads	✖	✔	✔	✔
Concurrent reads and writes	✖	✔	✔	✔
Reads during reallocate	✖	✖	✔	?
Writes during reallocate	✖	✖	✖	?

(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

Scalability of hash maps - Intel i7 w 4 cores & hyperthreading



Scalability of hash maps - AMD Opteron w 32 cores



Can you stipe other things?

- Histogram
- Potentially any indexed structure

Hashmaps using open addressing