Practical Concurrent and Parallel Programming VI
Performance and Scalability

Jørgen Staunstrup

Version after lecture

Agenda

- Performance versus scalability
- Scalability, speed-up and loss classification
  Example: QuickSort
- Executors and Future
  Example: count Prime Factors
- Hash maps, a scalability case study

Previously on PCPP

Week 3

Speedup for quicksort:	3.6 using 8 threads
	2.9 using 4 threads
Speedup for counting primes:	3.9 using 8 threads
	2.3 using 4 threads

Agenda

- **Performance versus scalability**
- Scalability, speed-up and loss classification
  Example: QuickSort
- Executors and Future
  Example: count Prime Factors
- Hash maps, a scalability case study

Performance and scalability

Performance (of software)

- **Latency:** time till first result (response time)
- **Throughput**: results per second

Scalability (one way to improve performance)

Improve throughput/latency by adding more resources

One may sacrifice performance for scalability

Maybe OK to be slower on 1 core, if faster on 2 or 4 or ...

Goetz chapter 11

What limits performance?

Suggestions?

What limits performance?

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

Input/output-bound
- Eg. reading from network
- To speed up, use more tasks (*inherent*)

Synchronization-bound
- Eg. Algorithm using shared data structure
- **To speed up, improve shared data structure**

(*Much of this lecture*)

Agenda

- Performance versus scalability
- **Scalability, speed-up and loss classification**
  Example: QuickSort
- Executors and Future
  Example: count primes
- Hash maps, a scalability case study

Quicksort

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 17 78 19 54 33 21 64 52 43 53

1 2 17 78 19 54 33 21 64 52 43 53

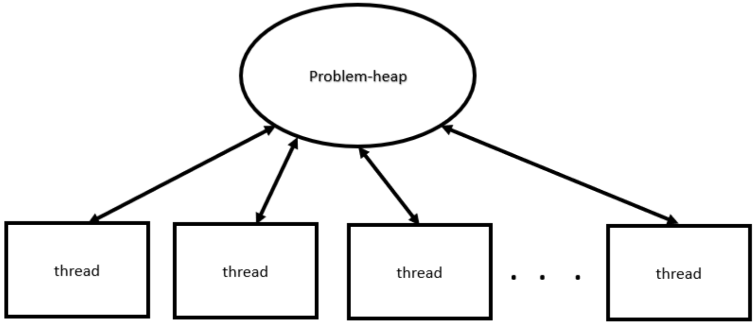
...

1 2 17 21 19 33 54 78 64 52 43 53

1 2 17 21 19 33 54 78 64 52 43 53

Two parts can be sorted independently

Multi-threaded version of Quicksort




```
class Problem {
      public int[] arr;
      public int low, high;
      ...
      }
}

class ProblemHeap {
      list<Problem> heap= new List<Problem>;
```

```
private static void qsort(Problem problem, ProblemHeap heap) {
  int[] arr= problem.arr;
  int a= problem.low;
  int.b= problem.high;
  ...
  heap.add(new Problem(arr, a, j); //qsort(arr, a, j);
  heap.add(new Problem(arr, i, b));//qsort(arr, i, b);
}
```

Java code for Quicksort (1)


```
public static void problemHeapStart(int threadCount, int pSize, int[] intArray) {
 ProblemHeap heap= new ProblemHeap(1);
 heap.add(new Problem(intArray, 0, pSize-1));

 for (int t=0; t<threadCount; t++) {
   threads[t]= new Thread( () -> { try {
       Problem newProblem= heap.getProblem();
       while (newProblem != null) {  // when newProblem == null alg stops
         qsort(newProblem, heap);
         newProblem= heap.getProblem();
       }
       }catch (InterruptedException exn) { }  //needed because getProblem may wait
   });
 }
}
```

Java code for Quicksort (2)

We use Mark8Setup to measure runtime

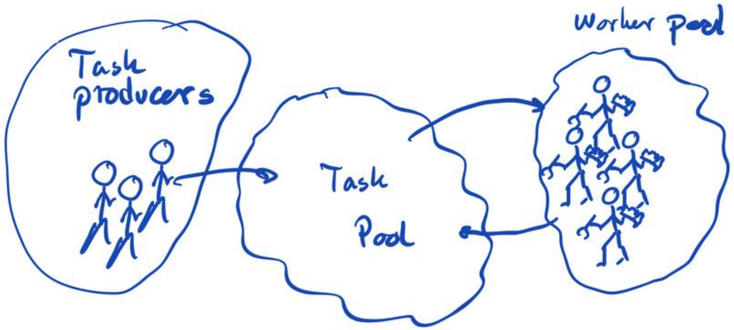
```
Benchmark.Mark8Setup("Problem heap quicksort",
          String.format("%2d", threadCount),
          new Benchmarkable() {
            public void setup() {
              SearchAndSort.shuffle(intArray);
              problemHeapStart(threadCount, pSize, intArray);
            }
            public double applyAsDouble(int i) {
              problemHeapFinish(threadCount, intArray); return 0.0;
            }
          }
    );
```

Code in: **`Week06: exercises/ProblemHeapSortingBenchmarkable.java`**

Java Executors (1)

Motivation

- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)



Task producers, and *Workers* are threads

Workers may themselves produce new tasks

The Task pool and Worker pool together is called an *Executor service*

Java Executors (2)

Goetz 6.2

```
new Thread(runnable1).start();
...
new Thread(runnable2).start();
...
new Thread(runnable3).start();
```

Threads are expensive !

```
ExecutorService pool;

pool.execute(runnable1);
...
pool.execute(runnable2);
...
pool.execute(runnable2);
```

Reuse of threads

https://howtodoinjava.com/java/multi-threading/java-fixed-size-thread-pool-executor-example/

Quicksort example


```
class runProblemHeap extends Thread {
  private final ExecutorService pool;
  public runProblemHeap(...) { pool = Executors.newFixedThreadPool(poolSize); }

  @Override
  public void run() {
    pool.execute( new solveProblem( new Problem(...), pool, c) );
  }
}

class solveProblem implements Runnable {
  ..
  @Override
  public void run() { qsort(... ); }

  // Quicksort
  public static void qsort(...,  ExecutorService pool) {   ...   }
}
```

Quicksort using Executors


```
class solveProblem implements Runnable {
 private ExecutorService pool;
 private static int threshold;
 public solveProblem(ExecutorService pool, int threshold, ...) {  }
 @Override
 public void run() { qsort(pool, ... ); }
 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) { ... }

   if ((j-a)>=threshold) pool.execute(new solveProblem(pool,threshold));
   else {  SearchAndSort.qsort(..., a, j);   }

   if ((b-i)>= threshold) pool.execute(new solveProblem(pool,threshold));
   else { SearchAndSort.qsort(..., i, b); }
  }
 }
}
```

Code in Week06: **Exercises** [.../QuicksortExecutor.java]()

Performance of Executor Quicksort


```
Executor quicksort  1     98003405.0 ns
Executor quicksort  2     53568593.9 ns
Executor quicksort  4     36397241.3 ns
Executor quicksort  8     21714103.7 ns
Executor quicksort 16     22237307.4 ns
Executor quicksort 32     22510681.9 ns
```

Speedup = 4.5

A bit better than using native Threads (slide 3)

Quicksort

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 17 78 19 54 33 21 64 52 43 53

1 2 17 78 19 54 33 21 64 52 43 53

...

1 2 17 21 19 54 33 78 64 52 43 53

1 2 17 21 19 33 54 78 64 52 43 53

Two parts can be sorted independently

What limits scalability?

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
- Speed-up using many harvesters: 5/(4+1/30) = 1.24 times faster

Amdahl's law (Goetz 11.2)

F = sequential fraction of problem = 4/5 = 0.8

N = number of threads (people) = 30

Speed-up <= 1/(F+(1-F)/N) = 1/(0.8+0.2/30) = 1.24

Other types of "loss" limiting scalability

Starvation loss

Separation loss (best threshold)

Saturation loss (locking common data structure)

Braking loss

Møller-Nielsen, P and Staunstrup, J, Problem-heap. A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63-74, 1987

Agenda

- Performance versus scalability
- Scalability, speed-up and loss classification
  Example: QuickSort
- **Executors and Future**
  Example: count Prime Factors
- Hash maps, a scalability case study

Termination in Executors

```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
//main (thread)
  setUpQS( … ) // creates thread
  finishQS( … ) //wait for all threads to finish
}
```

Need for a signal from the threads to the main thread !!!

Shut down

The ExecutorService has methods to help in shutting down.

```
 // Executor body
...
...

   pool.shutdown();
```

The challenge is: when to shut down?

In the Quicksort example a counter (=size of problem heap) was used to determine when to shutdown

Solution 1: shared counter


```
class countProblems{
  private int c= 1; // counting active threads + problems in heap
  synchronized void incr() { c++; }
  synchronized void decr() { c--; }
  synchronized void reset() { c= 1; }
  synchronized boolean isZero() { return c==0; }

  // The semaphore finished signals termination to main thread
  public Semaphore finished= new Semaphore(0);
}
```

Using the counter


```
public static void qsort(.., countProblems c) {
if (a < b) {
    ...
  if ((j-a)>= threshold) {
    c.incr();
    pool.execute(new solveProblem(new Problem(arr, a, j), pool, c) );
  };
  if ((b-i)>= threshold) {
    c.incr();
    pool.execute(new solveProblem(new Problem(arr, i, b), pool, c) );
  }
  c.decr(); // problem solved decrement c

  if ( c.isZero() ) { /* signal termination*/ pool.shutdown(); }
}
```

Using the counter


```
public static void qsort(.., countProblems c) {
if (a < b) {
    ...
  if ((j-a)>= threshold) {
    c.incr();
    pool.execute(new solveProblem(new Problem(arr, a, j), pool, c) );
  };
  if ((b-i)>= thre
    c.incr();
    pool.execute(new sol                                       c) );
  }
  c.decr(); // problem                decrement c

  if ( c.isZero() ) { /* signal termination*/ pool.shutdown(); }
}
```

How can we be sure that no other threads are active?

Termination in Executors


```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

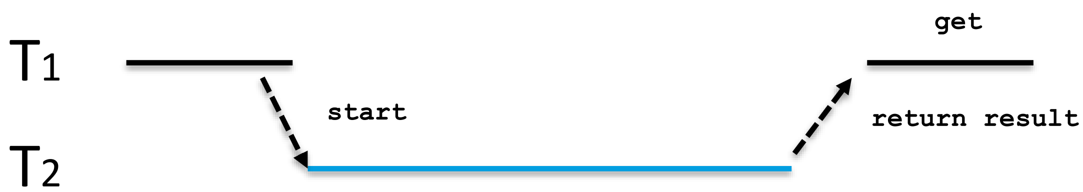
```
class solveProblem implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
//main (thread)
  setUpQS( … ) // creates thread
  finishQS( … ) //wait for all threads to finish
}
```

```
private static void finishQS(countProblems c) {
  try { c.finished.acquire(); }
  catch (java.lang.InterruptedException e) { ... ); }
}
```

Solution 2: Future




```
T2:
  public Future<Integer> calculate(Integer input) {
    return executor.submit(() -> {
      ... // compute result
      return result;
    });
  }
}

T1:
  Future<Integer> future = T2.calculate(  );  // start
     ...
  future.get();
```

Complete example

https://www.baeldung.com/java-future

```
private ExecutorService executor
     = Executors.newSingleThreadExecutor();

   public Future<Integer> calculate(Integer input) {
       return executor.submit(  //Callable
          () -> {...      return ... ;   }
       );
   }
```

Code in Week06: **LectureCode** [**…/futureExample.java**]

Runnable vs. Callable

Both are used to specify the code of a thread.

- Runnable cannot return a result
- Callable returns a result (via a Future)

As illustrated by the Quicksort example, Runnables may use shared data (e.g., to deliver a result)

Futures are an example of message passing

countPrimes


```
private static long countParallelN(int range, int taskCount) {
  List<Callable<Long>> tasks= new ArrayList<Callable<Long>>();

  for (int t= 0; t<taskCount; t++) {
    final int from= ...,
              to= ...;
    tasks.add(() -> {
      long count = 0;  // Task-local counter
      for (int i=from; i<to; i++) if (isPrime(i))  count++;
      return count;
    });
  }

  long result = 0;
  try {
    List<Future<Long>> futures = executor.invokeAll(tasks);
    for (Future<Long> fut : futures)
      result += fut.get();
    } catch ...
    return result;
  }
```

Agenda

- Performance versus scalability
- Scalability, speed-up and loss classification
  Example: QuickSort
- Executors and Future
  Example: count Prime Factors
- **Hash maps, a scalability case study**

Scalability of Java Collections

A *collection* is simply an object that groups multiple elements into a single unit

Package: java.util

Examples: ArrayList, HashMap, TreeSet, …

https://docs.oracle.com/javase/tutorial/collections/intro/index.html

Methods: add, remove, size, contains, …

Many of the classes have synchronized/concurrent implementations

https://www.baeldung.com/java-synchronized-collections

Example: synchronized ArrayList


```
import java.util.*;
public class syncCollectionExample {
  public static void main(String[] args) {  new syncCollectionExample(); }

  public String getLast(ArrayList<String> l) {
    int last= l.size()-1;
    return l.get(last);
  }

  public static void delete(ArrayList<String> l) {
    int last= l.size()-1;
    l.remove(last);
  }

  public syncCollectionExample() {
    ArrayList<String> a= new ArrayList<String>();
    a.add("A");  ...

    Collection<String> synColl = Collections.synchronizedCollection(a);
    ...
  }
}
```

Goetz p. 80

Thread-safety (from week 4)

It is very important to note that:

Program where all classes are thread-safe $\not\Rightarrow$ thread-safe <u>program</u>

Making the synchronized ArrayList thread safe


```
import java.util.*;
public class syncCollectionExample {
  public static void main(String[] args) {  new syncCollectionExample(); }

  public String getLast(ArrayList<String> l) {
    synchronized(l) {
      int last= l.size()-1;
      return l.get(last);
    }
  }

  public static void delete(ArrayList<String> l) {
    synchronized(l) {
      int last= l.size()-1;
      l.remove(last);
    }
  }

  public syncCollectionExample() {
   ...
  }
}
```

Goetz p. 80

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
...

Would not work if everything I "synchronized"

What can we do?

Reduce locking !!

Example: A huge HashMap

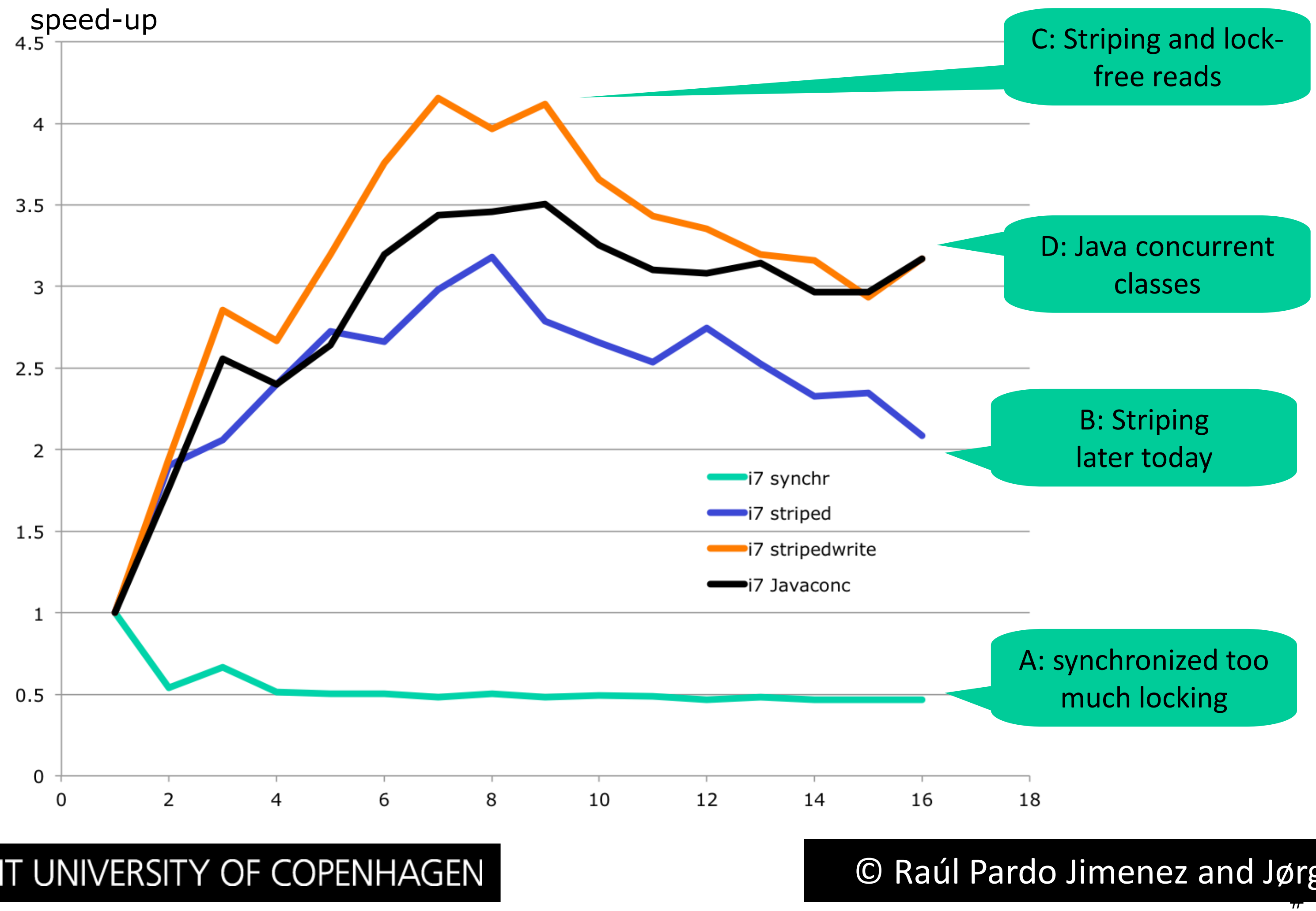
Key	Value
Peter	20487612
Anna	51251218
Lena	34458318
Holger	89545010
Lisa	94959500

Key value pairs: <k1, v1>, <k2, v2>, …

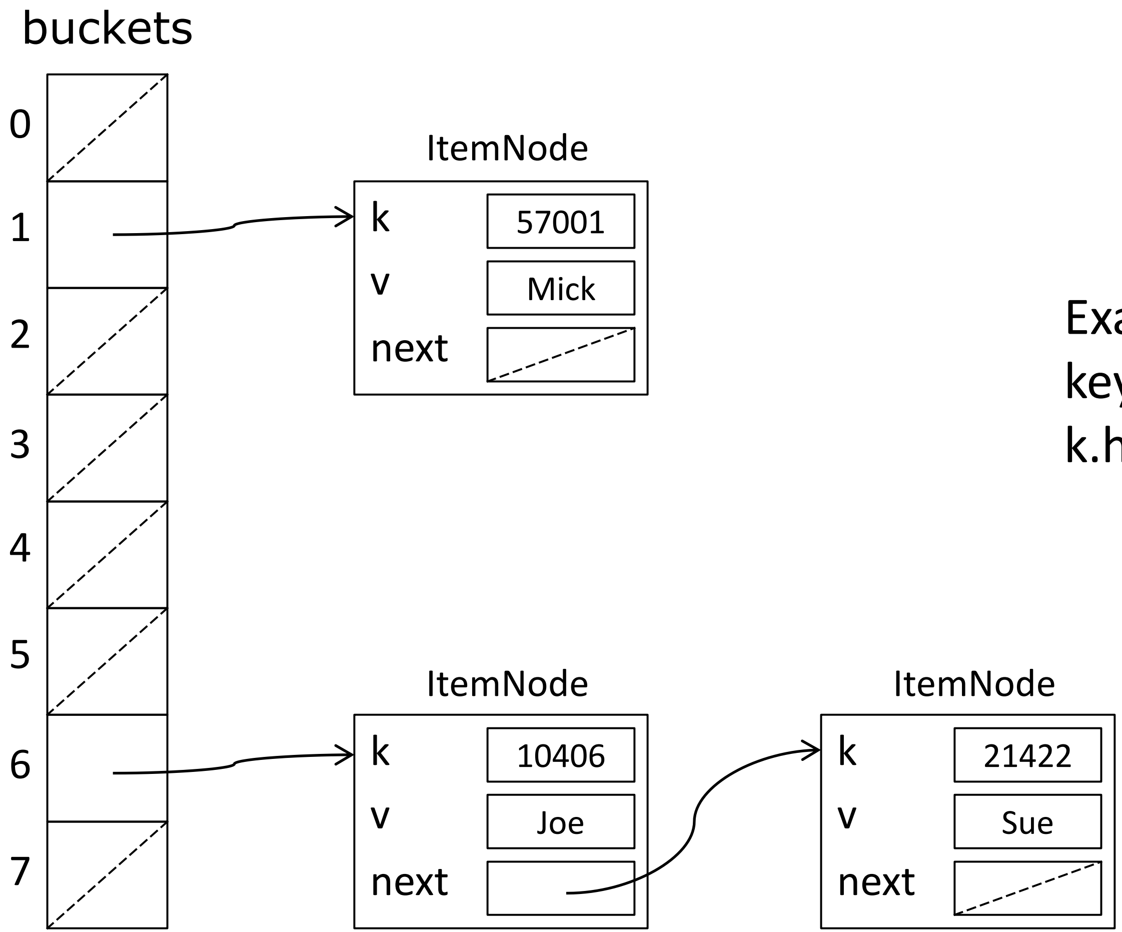
```
class HashMap<K,V> {
  ...  // datastructure
  public V get(K k) { ... }
  public V put(K k, V v) { ... }
  public boolean containsKey(K k) { ... }
  public int size() { return cachedSize; }
  public V remove(K k) { ... }
  ...
}
```

How to make it thread-safe?

Scaling a HashMap

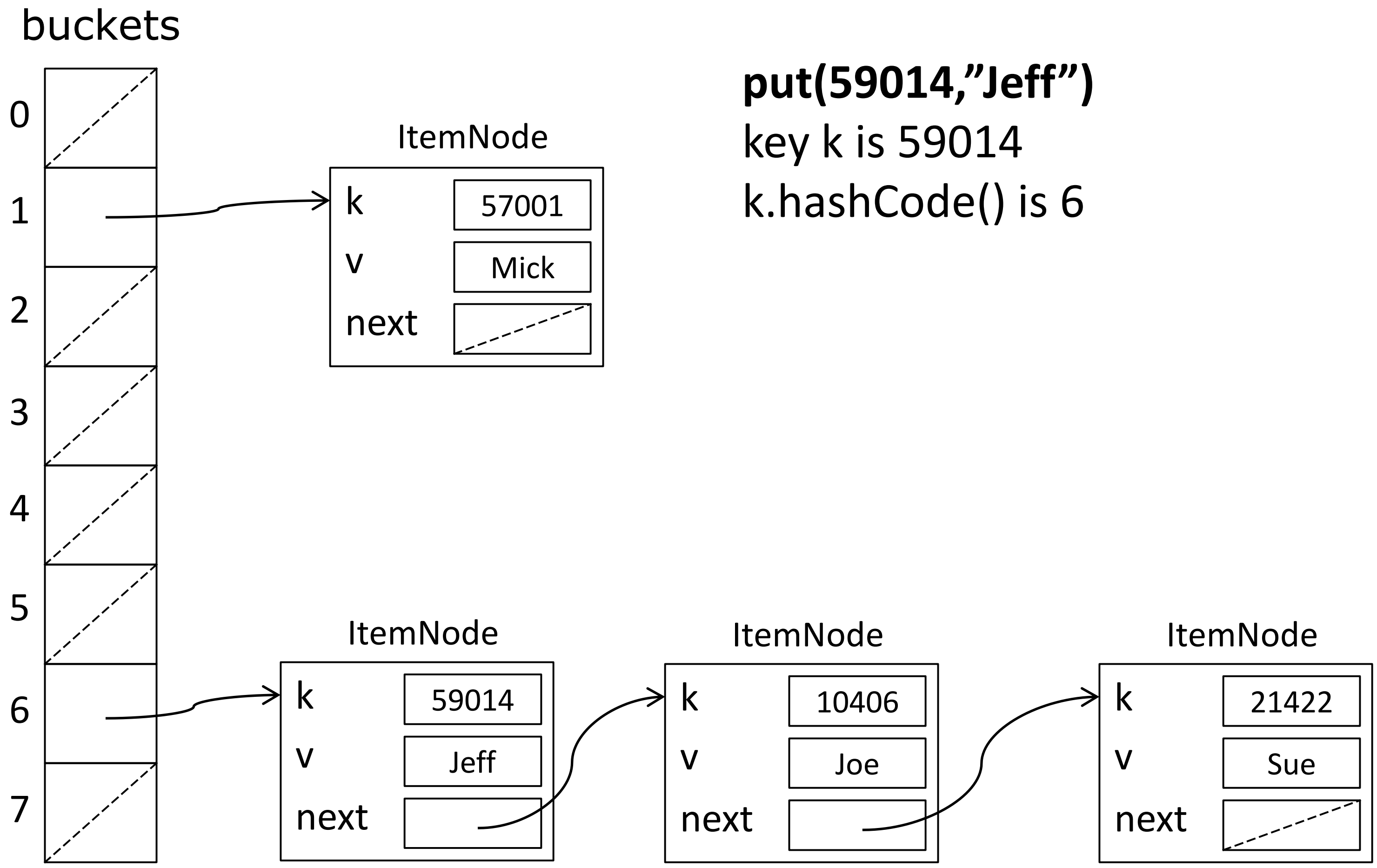


HashMap implementation



Example **get(10406)**
key k is 10406
k.hashCode() is 6

HaspMap put



put(59014,"Jeff")

key k is 59014

k.hashCode() is 6

Synchronized implementation

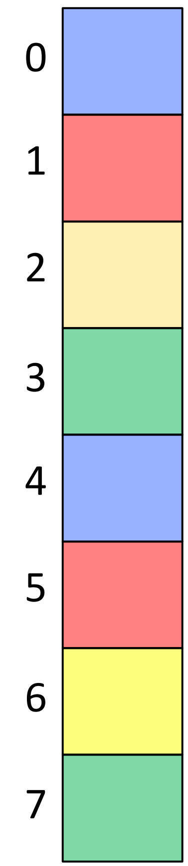

```
static class ItemNode<K,V> {
  private final K k;
  private V v;
  private ItemNode<K,V> next;
  public ItemNode(K k, V v, ItemNode<K,V> next) { ... }
}
```

```
class SynchronizedMap<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 remove(K k) { ... }
}
```

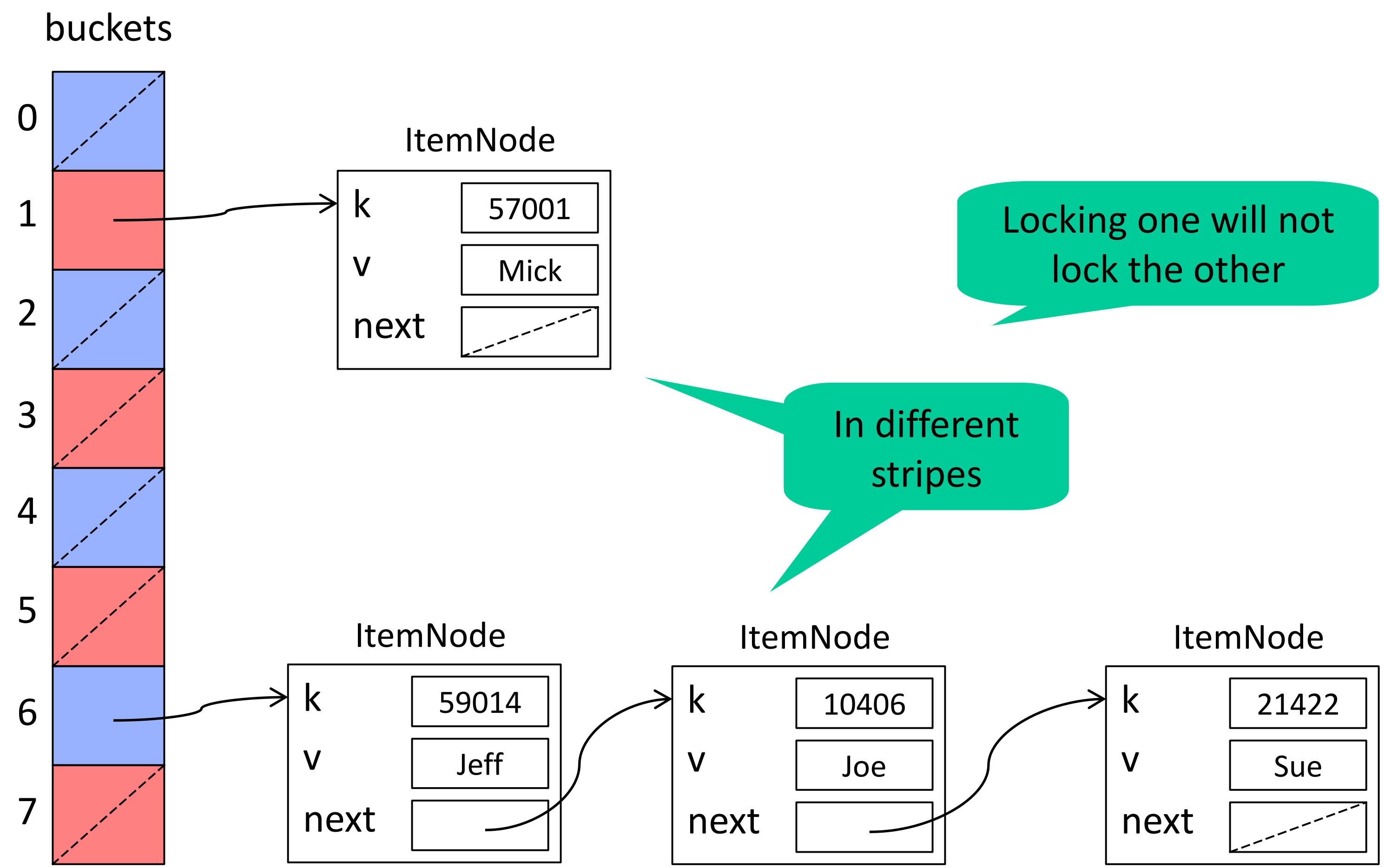
Improving scalability

- 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 fewer, to illustrate we use 4, locks
  - guard every 4th bucket with the same lock
  - locks[0] guards bucket 0, 4, 8, ... (stripe 0)
  - locks[1] guards bucket 1, 5, 9, ... (stripe 1) et

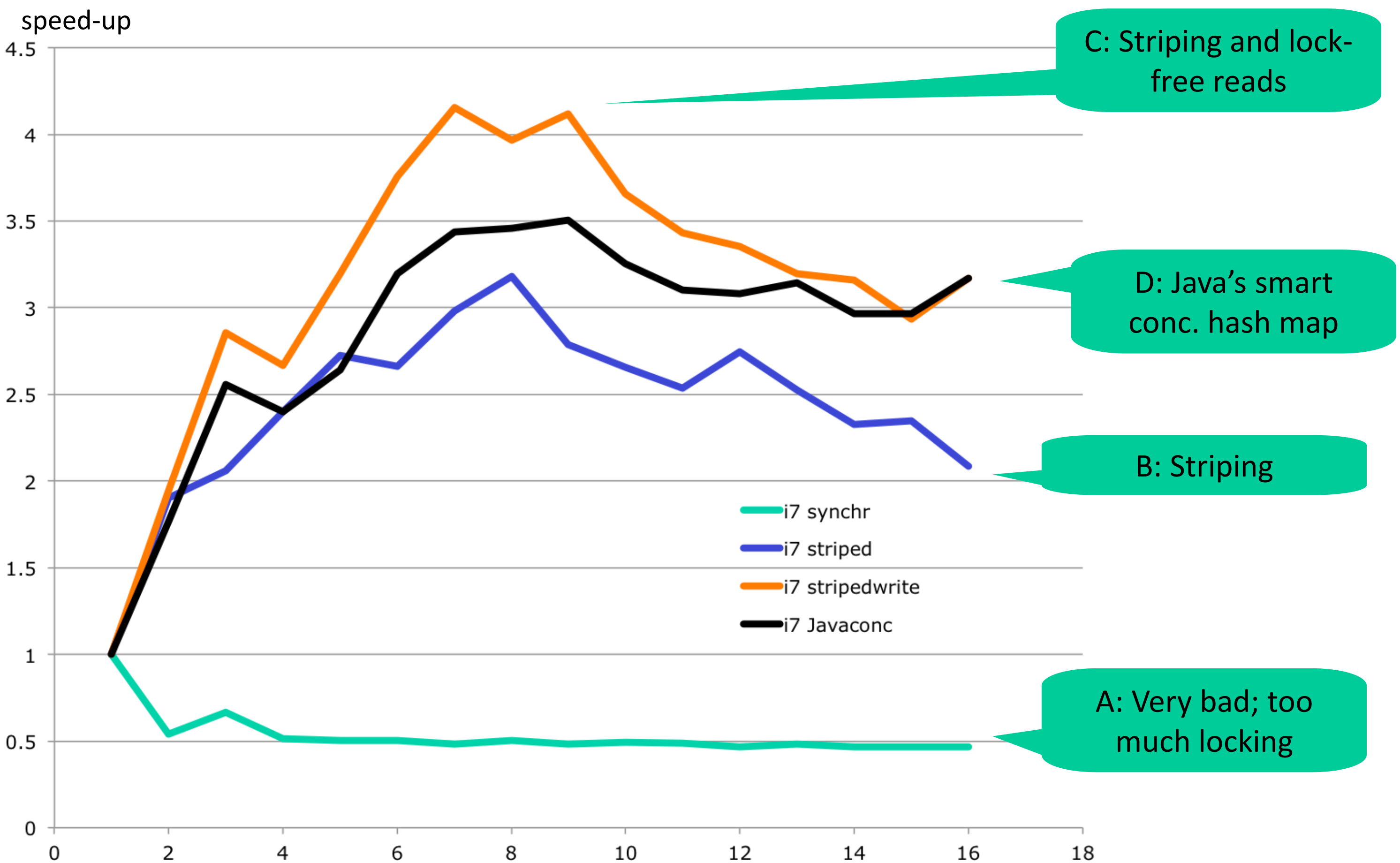
  - With high probability
  - two operations will work on different stripes
  - hence will take different locks
- Less lock contention, better scalability



Bucket idea


buckets
0
1
2
3
4
5
6
7
ItemNode
k
57001
v
Mick
next
Locking one will not lock the other
In different stripes
ItemNode
k
59014
v
Jeff
next
ItemNode
k
10406
v
Joe
next
ItemNode
k
21422
v
Sue
next

Reducing locking



Ultimate scalability

A web-shop, Facebook, …

We must give up thread safety,

but still maintain some sort of consistency

Week 13