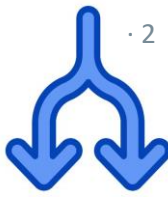


Practical Concurrent and Parallel Programming IX

Performance and Scalability

Jørgen Staunstrup



- **Executor and Future**
- Scalability, speed-up and loss (of scalability) classification
Example: QuickSort
- Lock striping
 - A case study with Hash maps



Thread:

```
threads[t] = new Thread( () -> {  
    ...  
        for (int i=from; i<to; i++)  
            if (isPrime(i)) lc.increment();  
});
```

Task:

```
public class countPrimesTask implements Runnable {  
    ...  
    @Override public void run() {  
        if ((high-low) < threshold) {  
            for (int i=from; i<=to; i++)  
                if (isPrime(i)) lc.increment();  
        }  
    }
```

Prime counter task (skeleton)



```
public class countPrimesTask implements Runnable {  
    private final int low;  
    private final int high;  
    private final ExecutorService pool;  
  
    @Override public void run() {  
  
        int mid= low+(high-low)/2;  
        pool.submit( new countPrimesTask(low, mid, pool) );  
        pool.submit( new countPrimesTask(mid+1, high, pool) );  
  
    }  
}
```



Shortcomings:

1. How to stop?
2. Will create too many "small" tasks
3. Returning result (# primes)



@Override

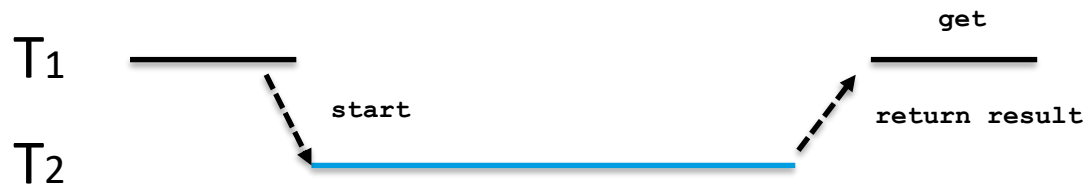
```
public void run() {  
    if ((high-low) < threshold) {  
        for (int i=low; i<=high; i++) if (isPrime(i)) lc.increment();  
    } else {  
        int mid= low+(high-low)/2;  
        pool.submit(new countPrimesTask(lc, low, mid, pool, threshold) );  
        pool.submit(new countPrimesTask(lc, mid+1, high, pool, threshold) );  
    }  
}
```

Shortcomings:

1. How to stop?
- ~~2. Will create too many "small" tasks~~
3. Returning result (# primes)



Futures are executors that deliver a result



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

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

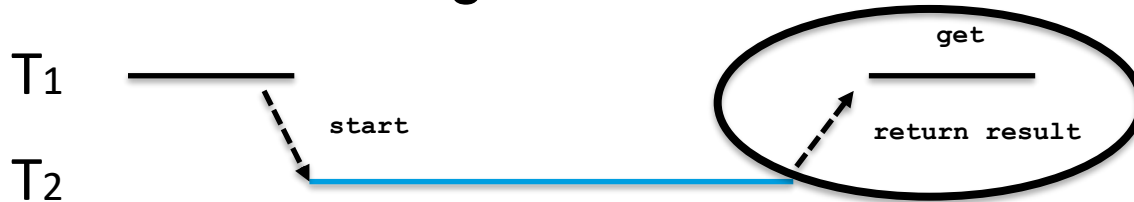
get() is a blocking call !!!!

Splitting tasks



```
public void run() {  
    if ((high-low) < threshold) {        ...  
    } else {  
        int mid= low+(high-low)/2;  
        Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,  
                                                    pool, threshold) );  
        Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,  
                                                    pool, threshold) );  
    }  
}
```

But how do we get the results from a future?



Combining tasks



.8

```
public void run() {
    if ((high-low) < threshold) {        ...
    } else {
        int mid= low+(high-low)/2;
        Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,
                                                    pool, threshold) );
        Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,
                                                    pool, threshold) );

        try {  f1.get();f2.get(); }
        catch (InterruptedException | ExecutionException e) {    }
    }
}
```

1: Does the order of f1.get and f2.get matter?


```
public void run() {  
    if ((high-low) < threshold) {        ...  
    } else {  
        int mid= low+(high-low)/2;  
        Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,  
                                                         pool, threshold) );  
        Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,  
                                                         pool, threshold) );  
  
        try {    f1.get();f2.get(); }  
        catch (InterruptedException | ExecutionException e) {    }  
    }  
}
```

Shortcomings:

- ~~1. How to stop?~~
- ~~2. Will create too many "small" tasks~~
3. Returning result (# primes)



```
public void run() {
    if ((high-low) < threshold) {        ...
    } else {
        int mid= low+(high-low)/2;
        Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,
                                                    pool, threshold) );
        Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,
                                                    pool, threshold) );

        try {  f1.get();f2.get(); }
        catch (InterruptedException | ExecutionException e) {    }
    }
}
```

Shortcomings:

- ~~1. How to stop?~~
- ~~2. Will create too many "small" tasks~~
3. Returning result (# primes)

**How do we get the result
primes ?**

Counting the primes

· 12



```
public void run() {  
    if ((high-low) < threshold) {        ... lc.increment();  
    } else {  
        int mid= low+(high-low)/2;  
        Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,  
                                                         pool, threshold) );  
        Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,  
                                                         pool, threshold) );  
  
        try {  f1.get();f2.get(); }  
        catch (InterruptedException | ExecutionException e) { }  
    }  
}
```

Shortcomings:

- ~~1. How to stop?~~
- ~~2. Will create too many "small" tasks~~
- ~~3. Returning result (# primes)~~

Final value in lc



```
public class countPrimesTask implements Runnable {
    private final int low;
    ...

    private static boolean isPrime(int n) {
        ...
    }

    public countPrimesTask(PrimeCounter lc, int low, int high,
        ExecutorService pool, int threshold) {
        this.lc      = lc;
        ...
    }

    @Override
    public void run() {
        ...
    }
}
```

- Ideally, tasks should be independent i.e. not update shared variables



```
public class countPrimesTask implements Runnable {
    private final int low; private final int high;
    private final ExecutorService pool;
    private final PrimeCounter lc;

    private static boolean isPrime(int n) {...}
}

public countPrimesTask(PrimeCounter lc... ) {this.lc= lc;... }

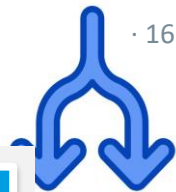
@Override
public void run() {
    if ((high-low) < threshold) {
        for (int i=low; i<=high; i++) if (isPrime(i)) lc.increment();
    } else {
        int mid= low+(high-low)/2;
        Future<?> f1= pool.submit( new countPrimesTask(lc, ... ) );
        Future<?> f2= pool.submit( new countPrimesTask(lc, ... ) );
        try { f1.get();f2.get(); }
        catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
    }
}
}
```

2: Are countPrimesTasks independent?

Code in countPrimesTask.java

countPrimesTask class

· 16



```
public class countPrimesTask implements Runnable {
    private final int low; private final int high; private final
    private final ExecutorService pool;
    private final PrimeCounter lc;

    private static boolean isPrime(int n) {...}
}

public countPrimesTask(PrimeCounter lc... ) {this.lc= lc;... }

@Override
public void run() {
    if ((high-low) < threshold) {
        for (int i=low; i<=high; i++) if (isPrime(i)) lc.increment();
    } else {
        int mid= low+(high-low)/2;
        Future<?> f1= pool.submit( new countPrimesTask(lc, ... ) );
        Future<?> f2= pool.submit( new countPrimesTask(lc, ... ) );
        try { f1.get();f2.get(); }
        catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
    }
}
}
```

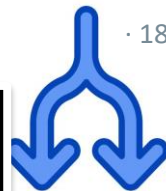
Not really, they all
update lc



```
class PrimeCounter {  
  
    private int count= 0;  
    public synchronized void increment() {  
        count= count + 1;  
    }  
    public synchronized int get() {  
        return count;  
    }  
    public synchronized void setZero() {  
        count= 0;  
    }  
}
```


PrimeCountExecutor class

· 18



- Kick-off class for the program
- It initializes the Executor service

```
class PrimeCountExecutor {  
    private ExecutorService pool;  
    ...  
    public PrimeCountExecutor () {  
        pool= new ForkJoinPool();  
        Future<?> done= pool.submit(new countPrimesTask( ... ));  
  
        try { done.get(); }  
    }  
}
```

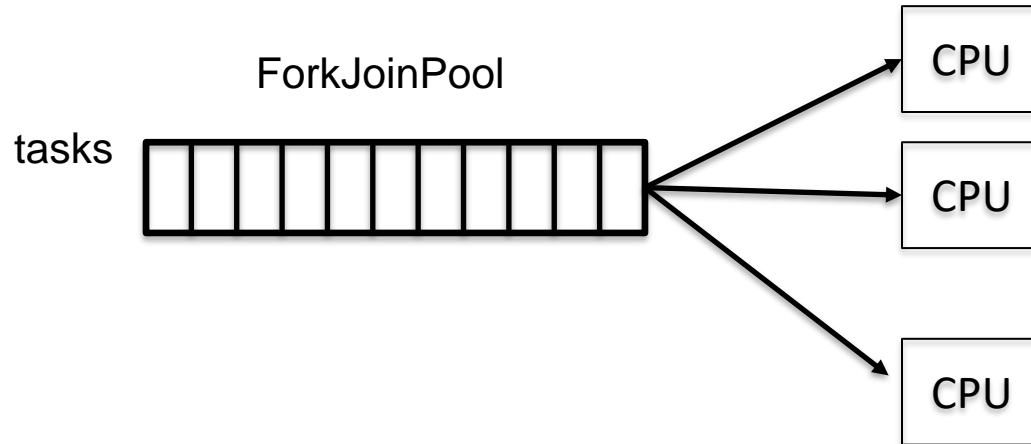
<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

Thread pools



· 19

```
class PrimeCountExecutor {  
    private ExecutorService pool;  
    ...  
    public PrimeCountExecutor () {  
        pool= new ForkJoinPool();  
        ...  
    }  
}
```





```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;
```

```
class PrimeCountExecutor {
    private ExecutorService pool;
    ...
    public PrimeCountExecutor () {
        pool= new ForkJoinPool( n );
        ...
    }
}
```

n specifies the number of CPUs that will be used

default value: [Runtime.availableProcessors\(\)](#)

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html>

Count prime results



Executor

1 120.6 s

2 68.0 s

4 37.7 s

8 32.2 s

16 32.4 s

1/8: 3.7

Threads

1 126.7 s

2 82.4 s

4 47.7 s

8 38.2 s

16 37.2 s

1/8: 3.9

range 2.. 1_000_000



- Executors and Future
- **Scalability, speed-up and loss (of scalability) classification**
Example: QuickSort
- Lock striping
 - A case study with Hash maps

Quicksort

· 23



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

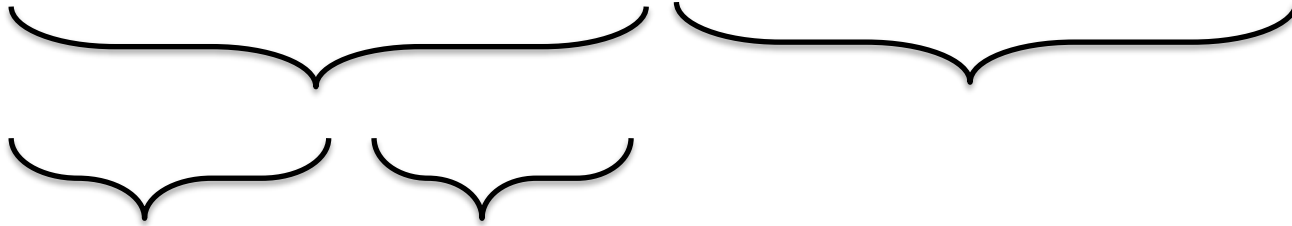
Two parts can be
sorted independently

Distributing work as tasks



· 24

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



...

No further splitting when the sorting problem is smaller than a threshold (similarly to what we did for prime counting)

These tasks may differ in size !!

Quicksort executor (pseudo code)



· 25

```
class QuicksortTask implements Runnable {
    Task p; // low and high boundaries
    ExecutorService pool;

    @Override public void run() { qsort(p, pool, ... ); }

    public static void qsort(Task p, ExecutorService pool, ... ) {
        //split task in two: Low and High as shown on previous slides

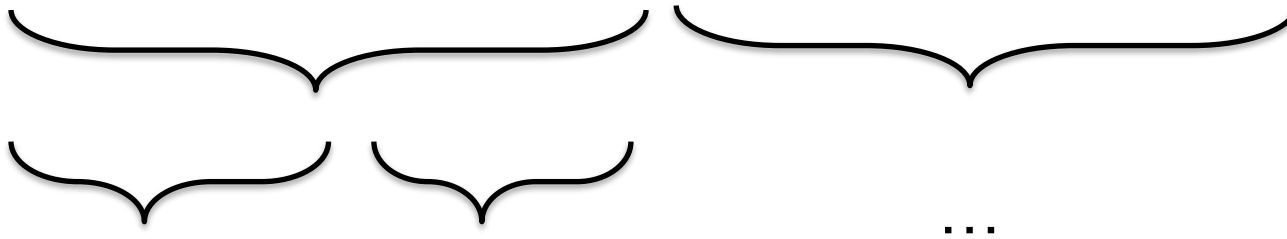
        if (Low.size>= threshold) pool.submit( new QuicksortTask( pLow, pool, ... ))
        else Quicksort(pLow); //sequential sort

        if (High.size>= threshold) pool.submit( new QuicksortTask( pHigh, pool, ... ))
        else Quicksort(pHigh);

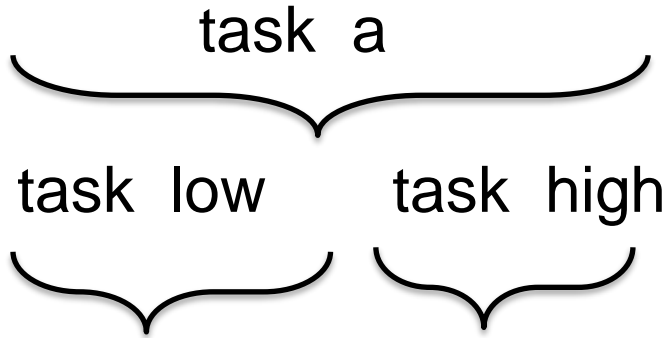
    }
}
```

Code in [QuickSortTask.java](#)

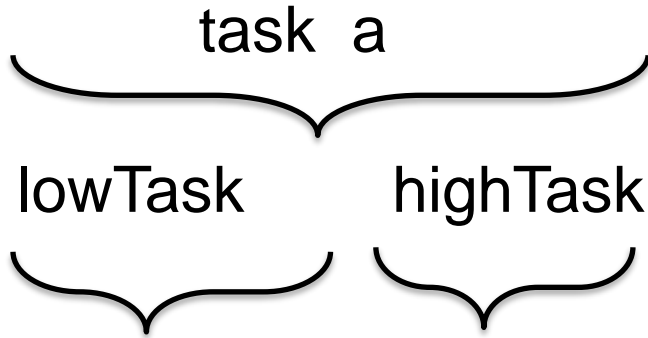
Termination (Quicksort)



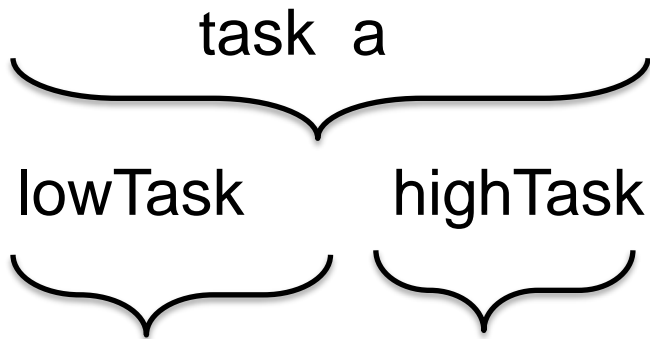
How do we know when all task are done?



3: When can task a finish?



task a can finish when
both task low and task high
have finished

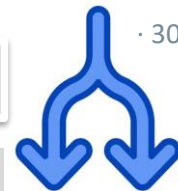


`lowTask.get()` returns when `lowTask` has finished

`highTask.get()` returns when `highTask` has finished

```
@Override
public void run() { // modified Quicksort using Executor tasks
    ...

    //Waiting for longest running subtask to finish
    try {
        lowTask.get();
        highTask.get();
    } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
}
```



```
@Override
public void run() { // modified Quicksort using Executor tasks
    int a= low;  int b= high;
    Future<?> lowTask= null;    Future<?> highTask= null;

    if (a < b) { ... // split array in two independent part

        if ((j-a)>= threshold)
            lowTask= pool.submit(new QuickSortTask(arr, a, j, pool, threshold));
        else // all remaining work done without starting more tasks
            SearchAndSort.qsort(arr, a, j);

        if ((b-i)>= threshold)
            highTask= pool.submit(new QuickSortTask(arr, i, b, pool, threshold));
        else // all remaining work done without starting more tasks
            SearchAndSort.qsort(arr, i, b);
    }

    //Waiting for longest running subtask to finish
    try {
        if (lowTask != null ) lowTaskF.get();
        if (highTask != null) highTaskF.get();
    } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
}
```

Benchmarking Quicksort

“Measure, don’t guess”

Goetz p. 224

· 31



We use Mark8Setup to measure runtime

```
private static void runSize(ExecutorService pool, int pSize, int threshold, int n) {
    final int[] intArray= fillIntArray(pSize);
    Benchmark.Mark8Setup("Quicksort Executor", String.format("%2d", n),
        new Benchmarkable() {

        public void setup() {
            shuffle(intArray);
        }

        public double applyAsDouble(int i) {
            Future<?> done= pool.submit(new QuickSortTask(intArray, 0, pSize-1, pool, threshold));
            PoolFinish(done);
            //testSorted(intArray); //only needed while testing
            return 0.0;
        }
    }
    );
}
```

See [benchmarkingNotes](#)
in week8

Code in [PoolSortingBenchmarkable.java](#)

Sorting results



Does not scale perfectly

Executor

1 8.5 s

2 4.8 s

4 2.6 s

8 2.2 s

16 2.2 s

Speed-up

1/8: 3.9

Threads

1 11.2 s

2 6.4 s

4 3.8 s

8 3.2 s

16 3.5 s

Speed-up

1/8: 3.9

Sorting 100_000 numbers

What limits scalability?



33

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

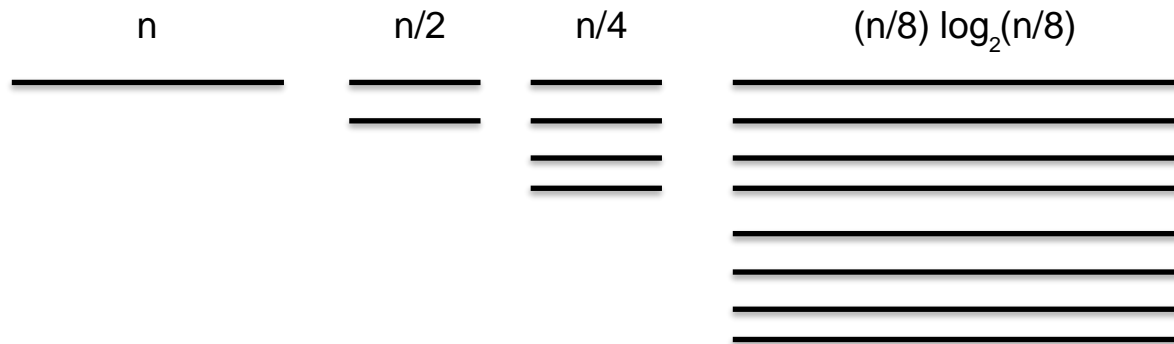
Amdahl's law (Goetz 11.2)

F = sequential fraction of problem e.g. $4/5 = 0.8$

N = number of threads (people) e.g. 30

$$\text{Speed-up} \leq 1/(F+(1-F)/N) = 1/(0.8+0.2/30) = 1.24$$

Maximum speed-up Quicksort



Best case

$$R(n) = n + n/2 + n/4 + (n/8)\log_2(n/8)$$

$$1,75n + (n/8)\log_2(n/8)$$

Amdahl: Starvation loss

$$\text{Max speed-up} = \frac{n \log_2(n)}{1,75n + (n/8)\log_2(n/8)}$$

$$n = 100.000$$

$$\text{Max speed-up} \sim 4.8$$

Loss of scalability (much more than Amdahl)



- Starvation loss (Amdahl)
 - Minimize the time that the task pool is emptyQuickSort
- Separation loss (best threshold)
 - Find a good threshold to distribute workload evenlyPrime count
- Saturation loss (locking common data structure)
 - Minimize high thread contention in the problem
- Braking loss
 - Stop all tasks as soon as the problem is solvedString search

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

The `ExecutorService` can be shut down.

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

```
pool.shutdown();
```

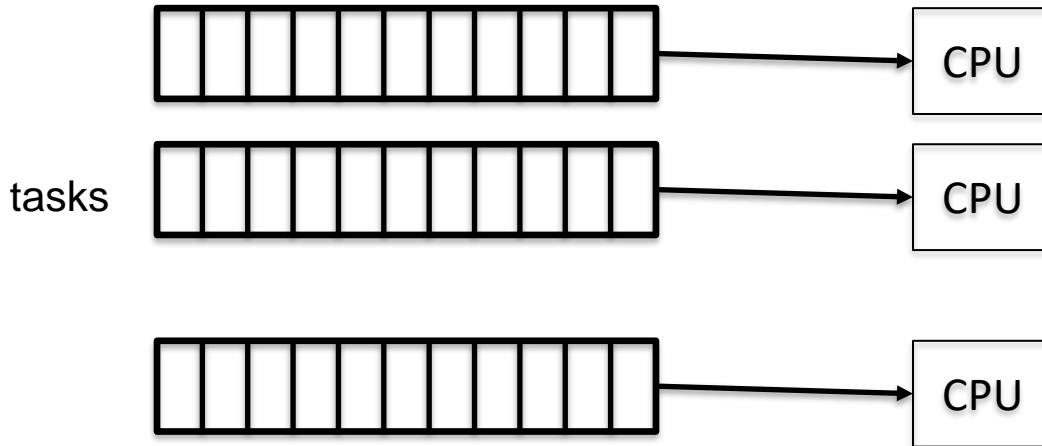
The challenge is often when to shut down

After shutdown the pool cannot be reused,
but you may assign it a new value (of type **`ExecutorService`**)

Thread pools



```
class PrimeCountExecutor {  
    private ExecutorService pool;  
    ...  
    public PrimeCountExecutor () {  
        pool= newFixedThreadPool(3);  
        ...  
    }  
}
```



<https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html>

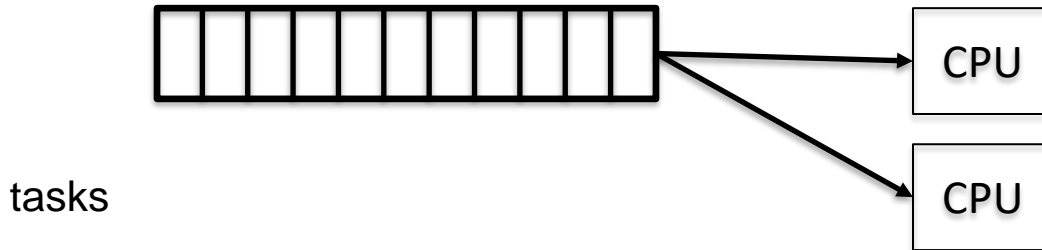
Thread pools



· 38

```
class PrimeCountExecutor {  
    private ExecutorService pool;  
    ...  
    public PrimeCountExecutor () {  
        pool= newWorkStealingPool(x);  
        ...  
    }  
}
```

Quite a few more
types of ExecutorService





Both are used to specify the code of a task.

- Runnable cannot return a result
 - Overrides `run()`
- Callable returns a result (via a Future)
 - Overrides `call()`

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

Both are used to spawn a task

- `pool.execute` does not return a result
may complicate determining when to finish
- `pool.submit` returns a result (via a Future)

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html>

- Executors and Future
- Scalability, speed-up and loss (of scalability) classification
Example: QuickSort
- **Lock striping**
 - **A case study with Hash maps**

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 thread-safe/concurrent implementations

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

Example: synchronizedCollection

· 43



```
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>(); // Collection
        a.add("A"); ...

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

Thread-safe
(but no locking !!!!)

Goetz p. 80

Example: synchronizedCollection

· 44



```
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);
        ...
    }
}
```

4: Is this version thread-safe?

Goetz p. 80

Example: synchronizedCollection

· 45



```
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);
        ...
    }
}
```

No

(because there is not a
happens-before relation
between the writes and reads
in the two methods)

Goetz p. 80

It is very important to note that for a program p :

p only accesses thread-safe classes

\Rightarrow

p is a thread-safe program

Making the synchronized ArrayList thread safe

· 47



```
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() {
        ...
    }
}
```

But then the advantage of the
synchronized collections is lost !!

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 is "synchronized"

What can we do?

Reduce locking !!

Example: A huge HashMap

Key value pairs: $\langle k_1, v_1 \rangle$, $\langle k_2, v_2 \rangle$, ...

```
class HashMap<K,V> {  
    ... // data structure  
    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) { ... }  
    ...  
}
```

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



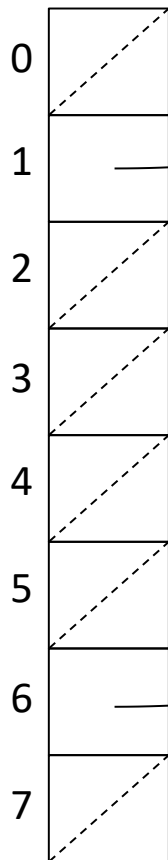
How to make it thread-safe?
(without making all the methods synchronized)

HashMap implementation

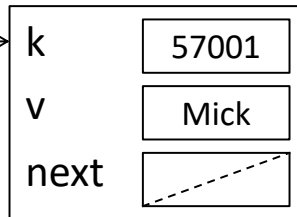
35



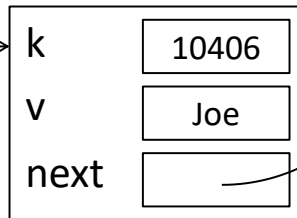
buckets



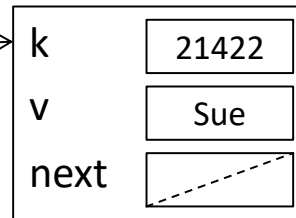
ItemNode



ItemNode



ItemNode



Example **get(10406)**

key k is 10406

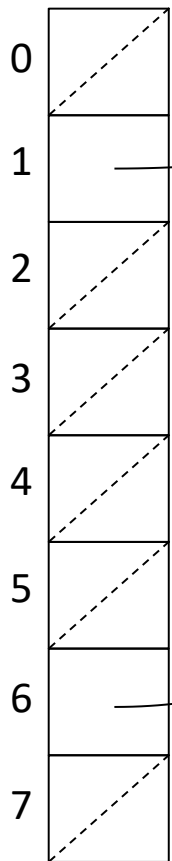
k.hashCode() is 6

HaspMap put

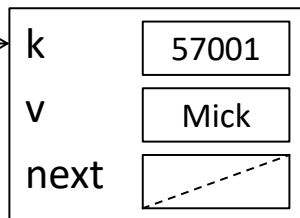
· 51



buckets



ItemNode

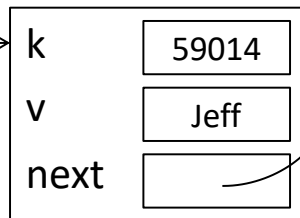


put(59014,"Jeff")

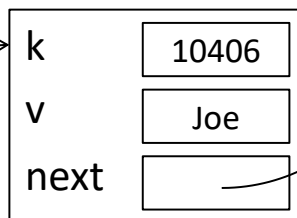
key k is 59014

k.hashCode() is 6

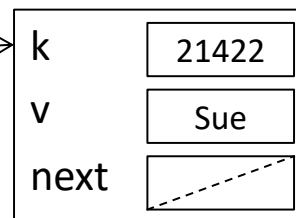
ItemNode



ItemNode



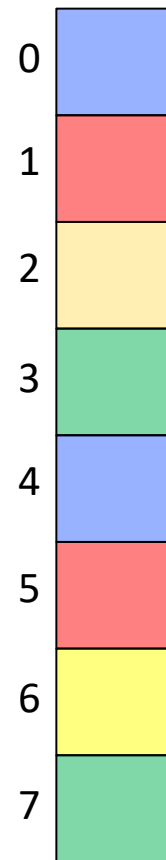
ItemNode



Improving scalability – Lock striping

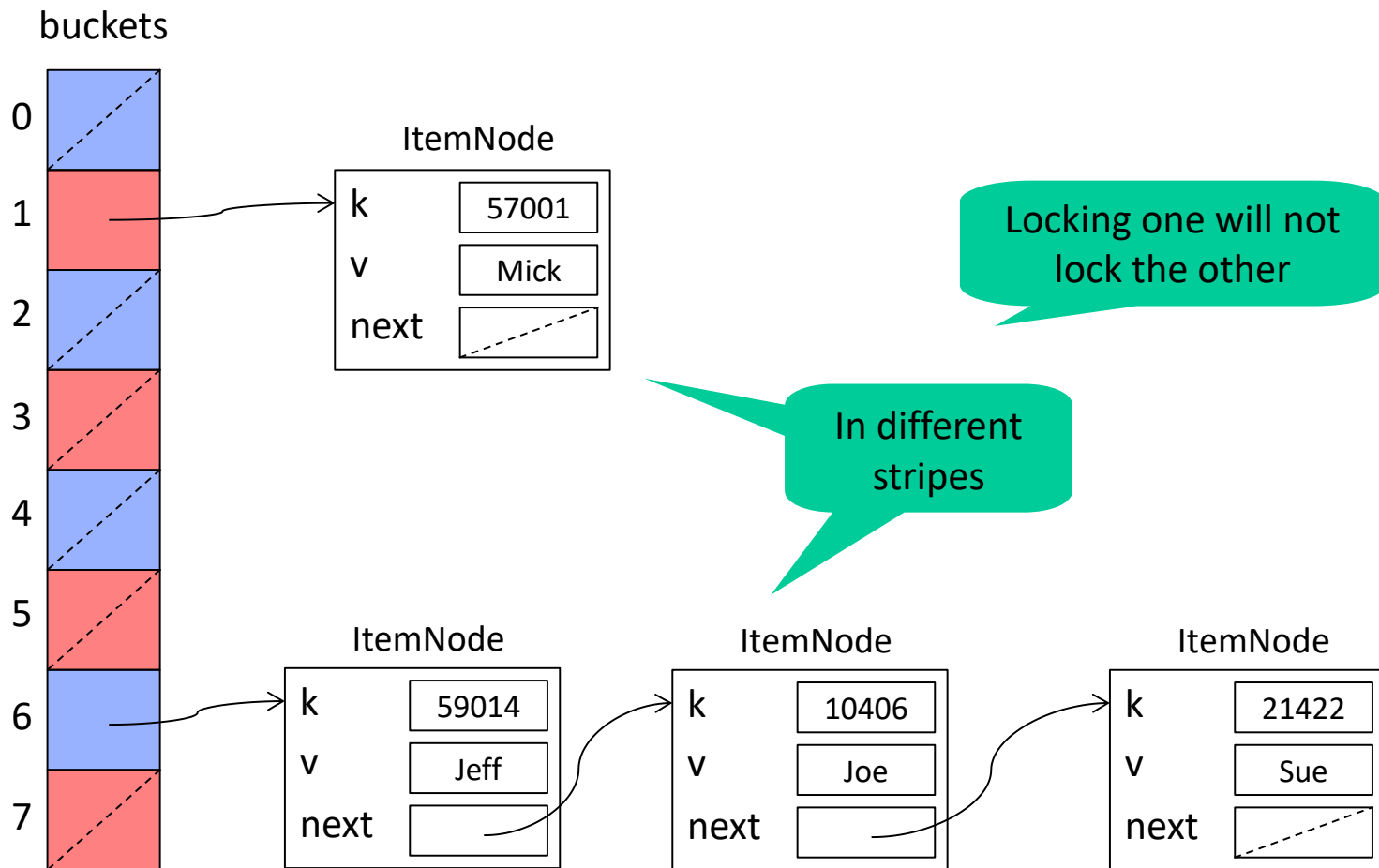


- Guarding the table with a single lock works
 - ... but does not scale well (actually **very** badly)
- Idea: Each bucket could have its own lock
- In practice
 - use fewer, to illustrate we use 4, locks
 - guard every 4th bucket with the same lock
 - locks[0] guards bucket 0, 4, 8, ... (stripe 0)
 - 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

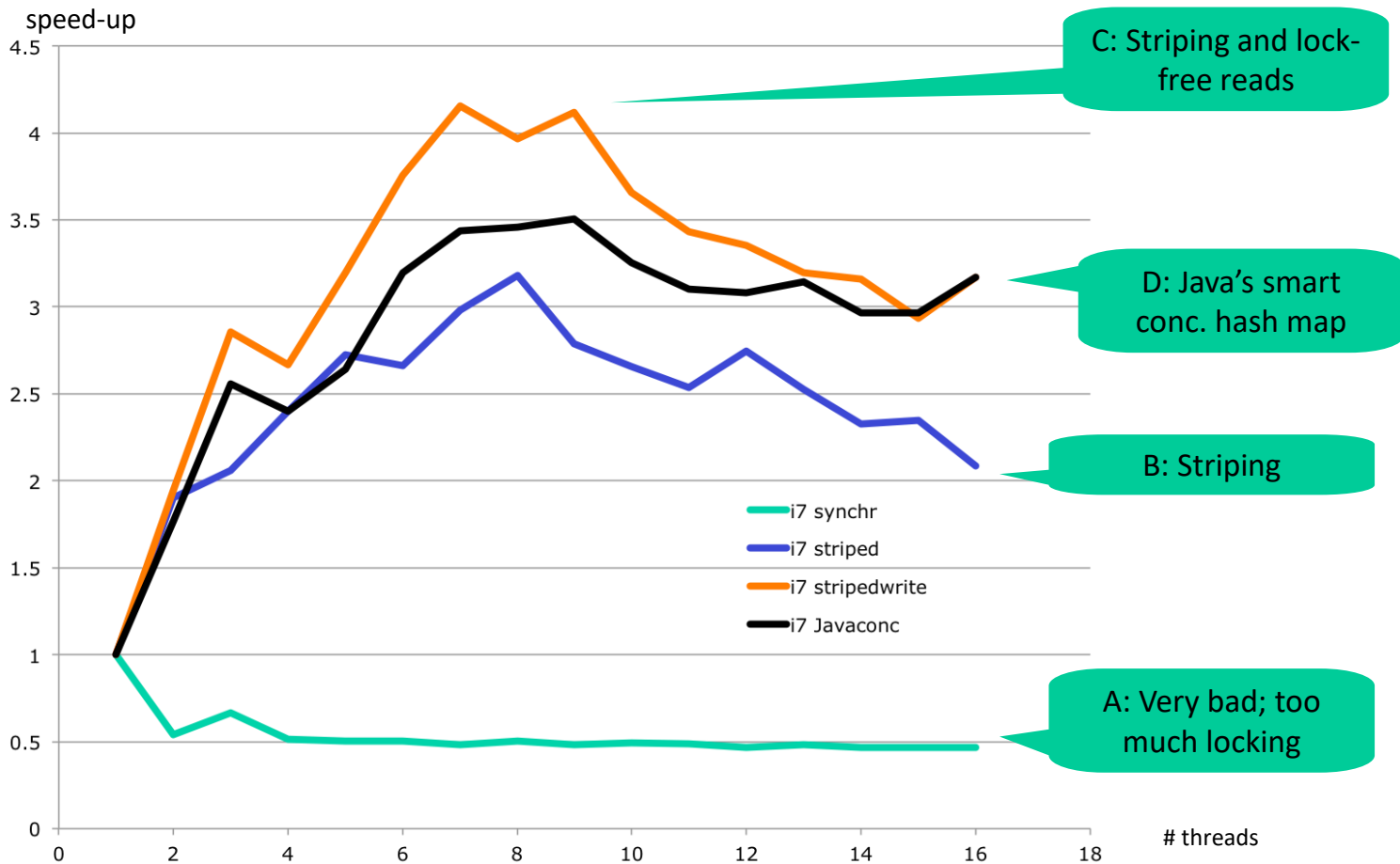
· 53



Reducing locking



· 54



A web-shop, Facebook, ...

We must give up thread safety,

but still maintain some sort of consistency