Performance and scalability

Ferd van Odenhoven
Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 7, 2014

Table of Contents

Fontys Hogescholen

Performance and scalability

The reason to use threads (and all its complexity) is improved performance

- Improve processor utilisation
- improve responsiveness

Here we explore techniques for analyzing, monitoring and improving the performance of concurrent programs

- ⚠ These techniques may increase complexity even more, with potentially more safety or liveness failures
- Worse even is that some "techniques" are counterproductive
- Safety should always come first: first make it **right**, then maybe make it **fast**.

Fontys Hogescholen

Do more with less

Improving performance means doing more work with less. Dependent on the application this less can be found in

- CPU cycles
- memory
- network bandwidth
- database requests
- I/O bandwidth
- disk space
- or any number of other resources

When the activity is limited by the availability of a particular resource, we say it is *bound* by that resource: CPU-bound, bandwidth-bound, database-bound etc.

Fontys Hogescholen

Cost and profit of concurrency

Multiple threads introduce extra costs: coordinating (locking, signaling, memory synchronization), context switching, thread creation and teardown.

When threading is employed effectively the costs are made up by the greater throughput, responsiveness or capacity. ⚠ A poorly designed concurrent application may perform worse then a comparable sequential one.

We want to achieve 2 things:

1. use processing power more effectively, with useful work
2. enable more use of the available resources once they become available again

Fontys Hogescholen

How Fast versus *How Much*

Performance can be measured in a number of ways: **service time**, **latency**, **throughput**, **efficiency**, **scalability** or **capacity**

Some of these metrics say **how fast**, others **how much**

Scalability

describes the ability to improve throughput or capacity by adding computing resources (CPU's, memory, storage, bandwidth)

Scalability depends on the *bound*ness of the application.

Traditional performance optimization uses big-O notation and works towards *do the same work with less effort*. In scalability you want to get more work done when applying more resources.

Fontys Hogescholen

How Fast versus *How Much* cont'd

How much and how fast are not the same, in fact sometimes contradict each other. Often the total work is increased by subdividing the work into better scaling subtasks.

Many single threaded performance improvement tricks are bad for scalability (hot field example in 11.4.4).

The three-tier application model is a prime example here: an monolithical solution would outperform it on a small system, but then this monolith would not scale at all.

For server applications *how much* is of greater concern then *how fast*.

Fontys Hogescholen

Engineering is makeing tradeoffs

To be able to assess optimal performance, you need to know about the metrics of the application, like the typical size of the data set, but also what to optimize: average-case time, worst-case time or predictability.

Avoid premature optimization

First make it right, then make it fast, *if* it is not already fast enough.

The cost in this case may not only be thread-safity hazards, but also things like code readability and thus maintainability.

Fontys Hogescholen

Which is faster?

Before considering to make things faster, answer these questions:

- What do you mean by *faster*
- What are the conditions for this faster: heavy or light load, large or small data sets, is there any measurement on these speeds?
- What is the likeliness that such conditions will occur for your application?
- How likely is it that these conditions are not met?
- What are the hidden costs (in development and maintainance time and risks). Is that a good tradeoff?

Fontys Hogescholen

Measure before you tweek

The quest for performance is probably the single greatest source of concurrency bugs.

Start measuring the actual performance. Use profilers to track down your bottlenecks.

Never guess about your performance bottlenecks. Most likely you are wrong.

Fontys Hogescholen

Farmers wisdom

Any farmer will tell you that a lot of hands are a great help during harvest season, but of little use during growing season: Some things can be parallelized, and some can't.

Concurrent programs have a lot in common with farming: a mix of parallel and serial portions.

Amdahls law describes how much a program can theoretically be speed up: If F is the serial fraction and N is the number of CPU's then then upper boundary of the speedup is

$$\text{Speedup} \leq \frac{1}{F + \frac{(1-F)}{N}}$$

The utilization is determined by the serialization required by the application. Very often in task dispensing and result collection.

Fontys Hogescholen

Little's law

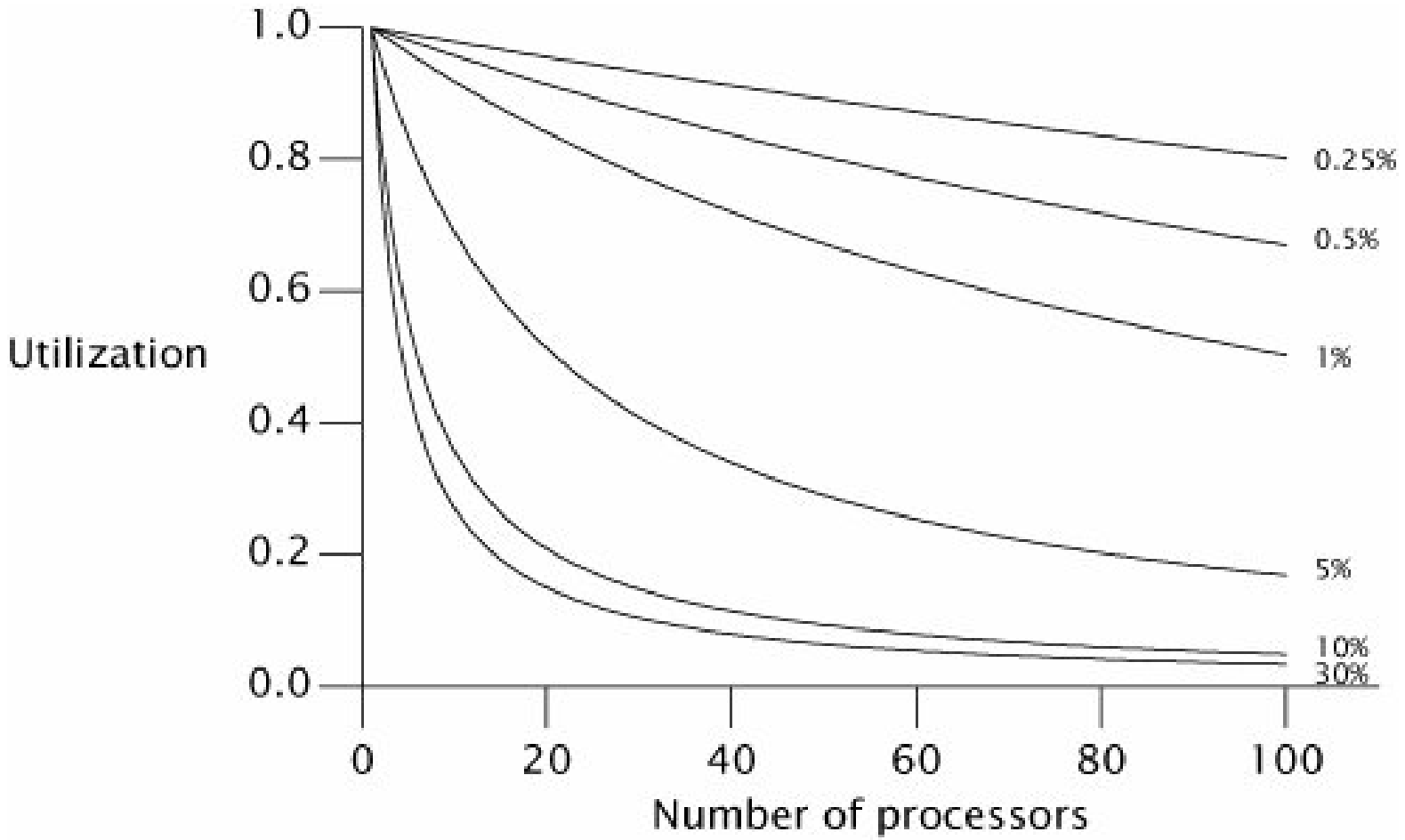
There is another remarkable law called Little's law which tells us something about the average number of active tasks.
If the rate of task arrival is λ and the task execution time is expressed as W then the average tasks that are busy L and thus would have need for a thread is

$$L = \lambda \times W$$

following Little's law.
See `http://en.wikipedia.org/wiki/Little%27s_law`

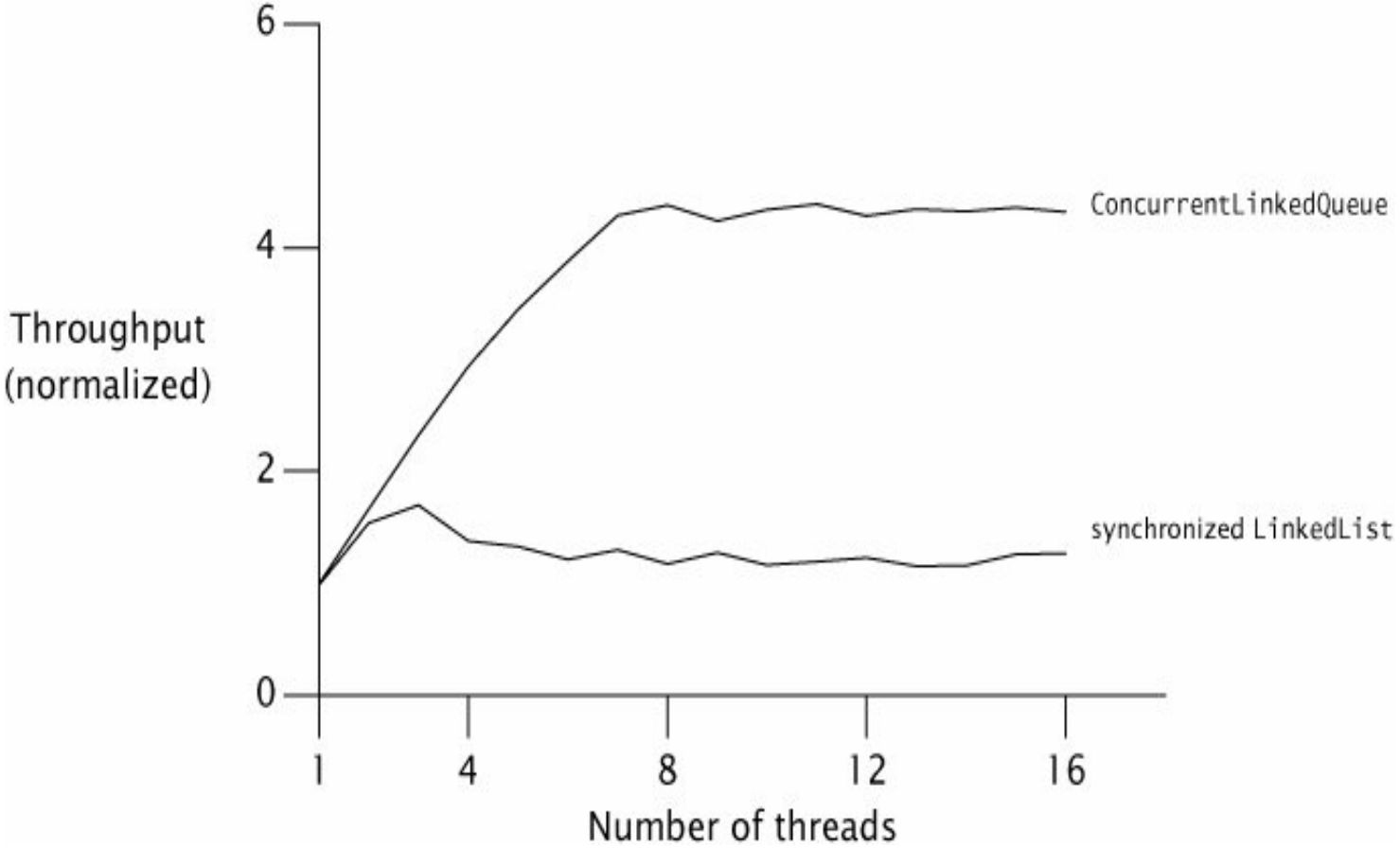
Fontys Hogescholen

Amdahls graph



Queue implementations

Two Queue implementions: Synchronized LinkedList versus ConcurrentLinkedQueue.



Task queue as serialization source

```
public class WorkerThread extends Thread {
    private final BlockingQueue<Runnable> queue;

    public WorkerThread(BlockingQueue<Runnable> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();
            } catch (InterruptedException e) {
                break; /* Allow thread to exit */
            }
        }
    }
}
```

There is always serialization

All concurrent applications have some sources of serialization;
if you think yours has not, think again.

Fontys Hogescholen

For concurrency to be useful, it must pay off

To get any benefit from your efforts, the benefits of parallelization must outweigh its costs.

Fontys Hogescholen

The costs of context switching

If you only have one thread, it will almost never be scheduled out.

If there are more runnable threads then CPU's then the OS and VM will preemt threads so that others can use the processor.

Context switching involves manipulating *shared* OS an JVM data structures. More time spent switching context means less work is done.

Context switched almost invariably leads to cache misses.

A program that needs to do a lot of blocking (I/O waiting for locks or condition variables) will incur more context switching than a CPU bound program.

Much time spent in kernel mode often points to heavy scheduling activity, either by blocking for I/O or contended locks.

Fontys Hogescholen

Memory synchronization

The cost of `volatile` and `synchronized`.

It is important to make a distinction between **contended** (lock is owned by other thread) and **uncontended** (lock is free) synchronization. The synchronization mechanism is optimized for the uncontended case (volatile is always uncontended).

Uncontended synchronization is not free, but the trend is making it faster in modern JVM's.

A modern JVM can reduce the costs even more, by optimizing locking away if it knows a lock can never be contended.

Fontys Hogescholen

Removal of certainly uncontended locks

```
public String getStoogeNames() {
    List<String> stoogesL = new Vector<String>();
    stoogesL.add("Moe");
    stoogesL.add("Larry");
    stoogesL.add("Curly");
    return stoogesL.toString();
}
```

In the above code the (synchronized) methods of Vector use the same lock, but since stooges is local, the whole thing is stack-confined en thus threadlocal: all synchronization can be dropped.

Fontys Hogescholen

Current and future costs of uncontended locking

Do not worry too much about the costs of (mostly) uncontended synchronization. The basic mechanism is quite fast and JVM's can perform optimizations that further reduce or even eliminate the cost. Instead, concentrate on reducing areas where contention is most likely.

Fontys Hogescholen

The cost of synchronization and blocking

Bypassing the cache for `volatile` and reads and writes from synchronized blocks also leads to contention on the shared memory bus. (Processors must wait for one another; Bus bandwidth is also a limiting factor).

Uncontended synchronization can be handled completely within the JVM. Contended synchronization may involve the OS which adds to its costs.

If the blocking is implemented by suspending the losing thread (instead of a spin-lock) the OS is involved, leading to two context switches: it is switched out before the end of its quantum and then switched back in again when the resource becomes available.

The winning thread also has a cost: it must tell the JVM/OS to resume the blocked threads.

Fontys Hogescholen

Threat to scalability

The principal threat to scalability in concurrent applications is the exclusive resource lock.

Three ways to reduce lock contention

1. Reduce the duration in which the locks are held;
2. Reduce the frequency with which locks are requested;
3. Replace exclusive locks with coordinating mechanisms that allow greater concurrency.

Fontys Hogescholen

Narrow the scope of a lock

In the code below, the reason for synchronized is access to the Map. The surrounding code is not *critical*. So it gets a 

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches(String name,
                                                    String regexp) {
        String key = "users." + name + ".location";
```

```
        String location = attributes.get(key);
```

```
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Fontys Hogescholen

Better

```
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
```

```
        synchronized (this) {
            location = attributes.get(key);
        }
```

```
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Best would be: replace the HashMap with a synchronized map or even ConcurrentHashMap, delegating thread safety to those components.

Shrinking until you fall in

Take care that when shrinking the synchronized blocks, operations that must occur atomically still do so. Do not make the blocks to small to maintain invariants that are involved.

The example is where multiple variables participate in an invariant. Any invariant affecting code must be atomically with respect to that invariant.

Reducing lock granularity

If a lock is heavily contended, but the synchronized blocks access independent state variables (those that *do not* participate in an invariant, then the locks may be split or striped.

As an example think of one lock for the whole application: everything would contend for that single lock. This lock would be heavily contended.

If you isolate independent parts, those parts may use independent locks, reducing the likeliness that threads contend over the same locks.

Fontys Hogescholen

Lock splitting candidate

```
@ThreadSafe
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) {
        users.remove(u);
    }

    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
```

Fontys Hogescholen

Lock striping

One step further then splitting a lock in two, is splitting it up into a larger set, with the intention that these extra locks are less contended. This is called **striping**.

The example `StripedMap` is similar in design as the ConcurrentHashMap, so you get the idea.

Fontys Hogescholen

Avoid hot fields

Hot fields are areas that are heavily contended

Commonly, caching is one way to achieve optimisation.

These caches themselves could become hot field liabilities if not used properly.

Keeping a count to have a quick answer for `size()` and `isEmpty()` works well in the single threaded case, but could become a contended data area in a concurrent setting.

Fontys Hogescholen

Reduce lock exclusivity

Another way to prevent the effects of lock contention is to abstain from using exclusive locks and instead use more concurrent friendly ways of managing state. Examples are using

- concurrent collection,
- read-write locks,
- immutable objects and
- atomic variables.

ReadWriteLock provides a multiple-reader, single-writer lock. For mostly-read data structure ReadWriteLock can offer greater concurrency then exclusive locking; for read-only access, using immutability (perhaps with a wrapper) can eleminate the need for a lock completely.

Fontys Hogescholen

Use atomic variables

Atomic variables can help reducing costs of updating hot fields such as statistics counters, sequence generators or the reference to the first node in a linked data structure.

Atomic variable classes exhibit fine-grained atomic operations on integers or object references and are implemented blocking free, using such instructions as **compare-and-swap**

Note that reducing the number of hot fields helps more, because atomic updates do not completely reduce the cost of updates.

Fontys Hogescholen

Monitoring CPU Utilization

In general, to improve throughput, keeping the CPU fully utilized is the main optimization goal.

If the hot-ness of the CPU's is unbalanced, you need to find possibilities to increase parallelism. Such asymmetry indicates that computation is taking place in a small set of threads. Your application will not fully utilize additional processors.

Fontys Hogescholen

Underutilization causes

Insufficient Load The application may not receive sufficient load. The clients might running at full capacity.

I/O-Bound One can be monitor if the application is disk or network bound with appropriate monitoring tools.

Externally bound The bottleneck may be in the database or web service, not in your own code. Use profiling or database administration tools to determine the time spent in waiting for answers .

Lock contention Profilers can tell how much lock contention you experience and which locks are hot. By randomly triggering thread dumps, these can help find contended locks. Waiting threads will show "waiting for lock on monitor ...". Mostly uncontended locks will hardly show up. Heavily contended locks most likely will.

Fontys Hogescholen

Room for improvement?

If your CPU's are sufficiently hot, you may want to find out if adding CPU's will help: is the application scalable.

Reconfiguring the system may be in order such as changing the number of threads in the pool.

If CPU utilization is high and there is always a number of threads waiting for the CPU, your application might benefit from throwing more CPU's at it.

Fontys Hogescholen

Do not pool objects

Pooling objects[1] was popular in times when memory management (allocation and garbage collection) was less mature.

Nowadays allocation is faster then `malloc` in C. The common code path for `new Object()` is about 10 machine instructions.

Pooling (reusing) object in concurrent application only makes things worse:

- A pool is a shared data structure; `new Object()`s are not shared;
- Potential problem of improper state of the reused object.
- Blocking is 100 times more costly then new's allocation.

[1] This slide is about *Object pooling*, not Thread Pooling. Don't mix things up.

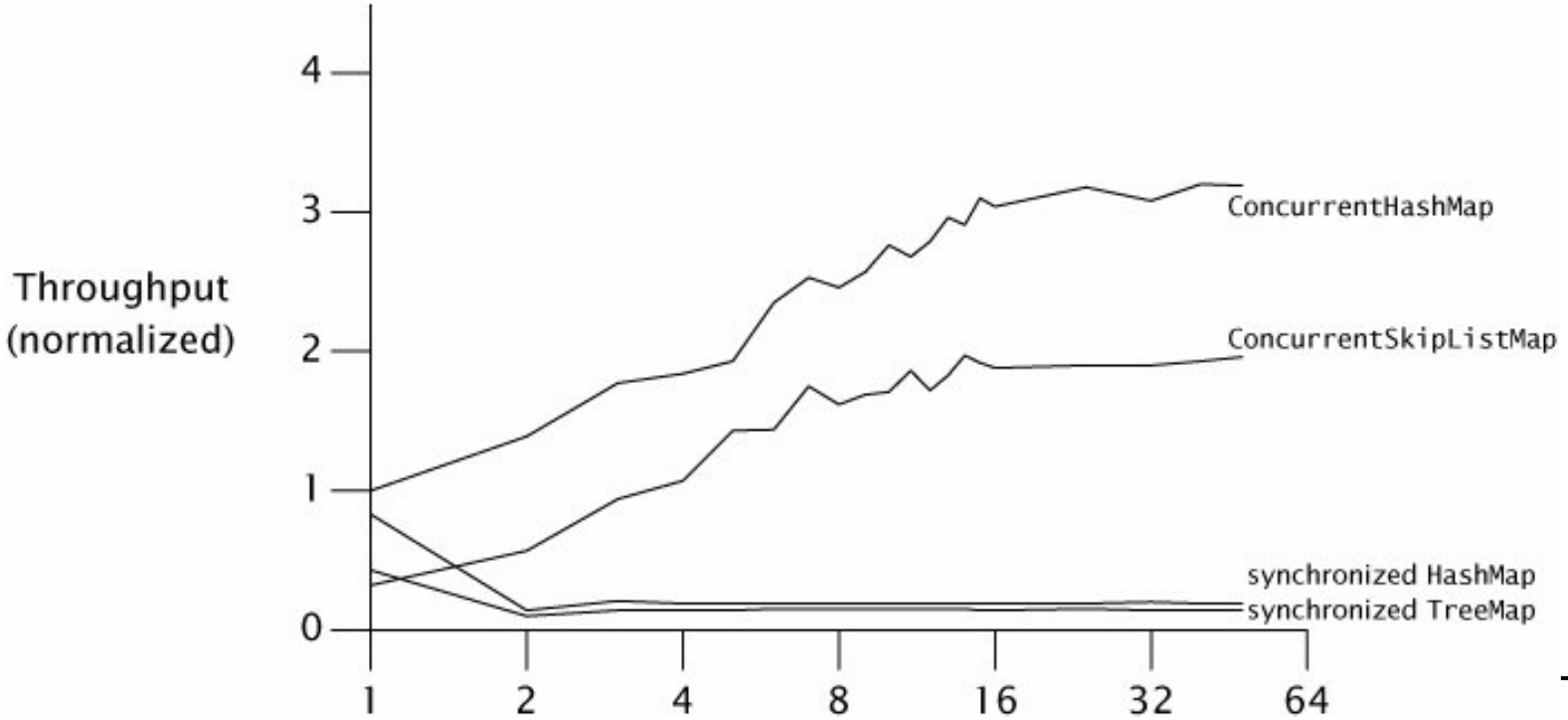
Fontys Hogescholen

Cost of allocation

Allocating objects is usually cheaper then synchronization.

Comparing map performance

Traditional maps share one lock for the entire map. In the figure two synchronized maps and two Concurrent maps are compared.



The behaviour of the synchronized versions is typical for applications whose scalability is limited by lock contention.

If something makes tired, do it less often

Prevent contention or make contention less likely by holding less locks or holding them for shorter times. This also reduces the number of context switches

In the logging example: contention on the queue is less likely than blocking for I/O, which mostly involves the OS anyway.

By moving the I/O to one thread

we not only eleminate the change of contention for the output stream.

Having *all* running to the water pool and *all* putting water on the fire will produce contention at both ends.

Fontys Hogescholen

Summary

- Multiprocessors is nowadays the most common reason to use threads,
- this implies that we are concerned with throughput or scalability,
- raw service time comes second to that
- Amdahls law states that scalability is determined by the proportion of code that shares data, i.e. that must be executed serially
- Serialisation is determined largely by the use of resource locks,
- scalability will improve by reducing the lock holding time:
  - reduce the time holding the lock
  - reduce lock granularity
  - replace exclusive locks by non-exclusive locks or non blocking alternatives

Fontys Hogescholen