Java Concurrency in Practice

Chapter-11
Performance and Scalability

Upcode Software
Engineer Team

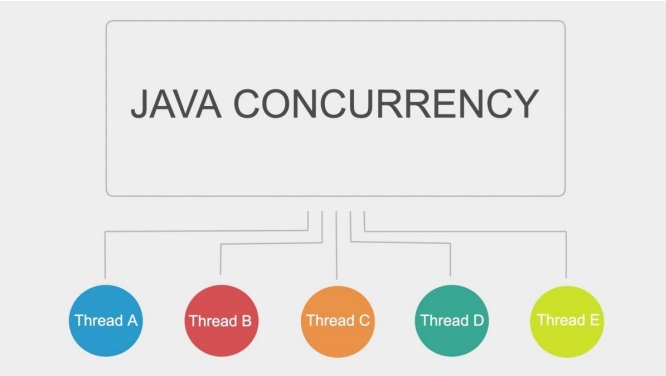
-2023-

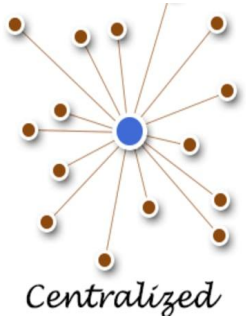
CONTENT

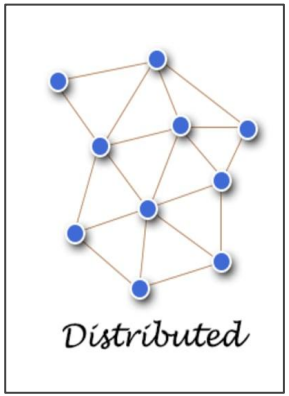
1. Concurrency Models and Distributed System Similarities
2. Thinking about performance
3. Amdahl’s law
4. Cost introduced by threads
5. Reducing lock contention
6. Solution
7. Reference

Concurrency Models and Distributed System (1/n)

- In **a concurrent system** different threads communicate with each other.
- In **a distributed system** different processes communicate with each other (possibly on different computers).
- **Threads and processes** are quite similar to each other in nature. That is why the different concurrency models often look similar to different distributed system architectures.





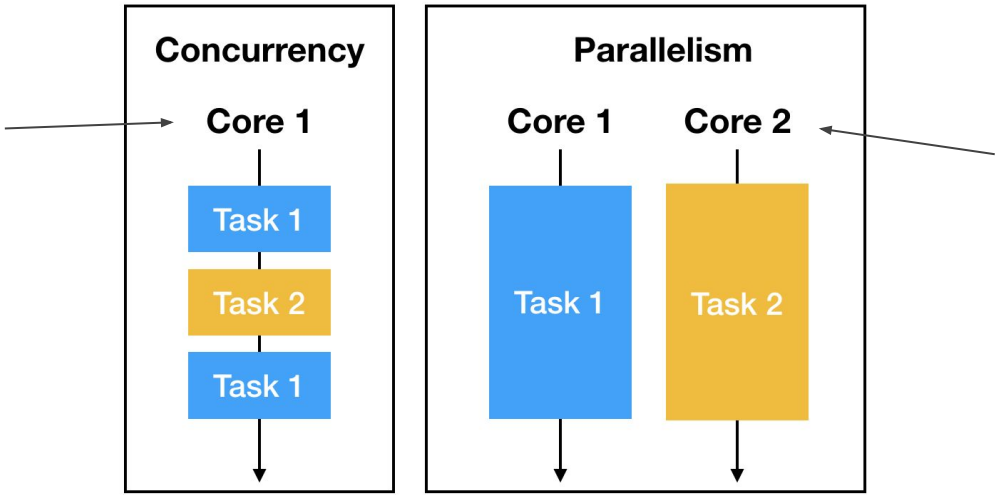


Concurrency Models and Distributed System (2/n)

- **Distributed systems** have the extra challenge that the network may fail, or a remote computer or process is down etc.
- a **concurrent system** running on a big server may experience similar problems if a CPU fails, a network card fails, a disk fails etc. The probability of failure may be lower, but it can theoretically still happen.

Concurrency Models and Distributed System (3/n)

- **Concurrency** means executing multiple tasks at the same time but not necessarily simultaneously.
- **Parallelism** does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using the multi-core infrastructure of CPU, by assigning **one core to each task or sub-task.**

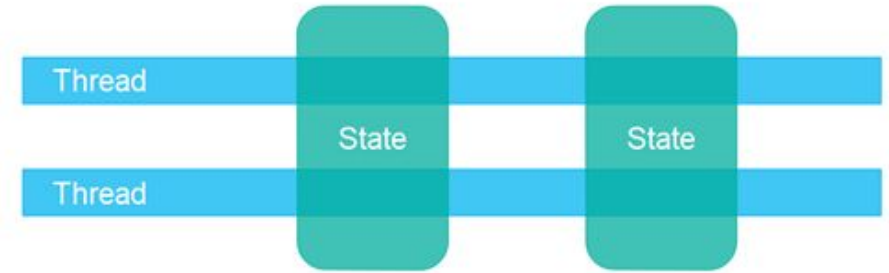


Concurrency Models and Distributed System (4/n)

S.NO	Concurrency	Parallelism
1.	Concurrency is the task of running and managing the multiple computations at the same time.	While parallelism is the task of running multiple computations simultaneously.
2.	Concurrency is achieved through the interleaving operation of processes on the central processing unit(CPU) or in other words by the context switching.	While it is achieved by through multiple central processing units(CPUs).
3.	Concurrency can be done by using a single processing unit.	While this can't be done by using a single processing unit. it needs multiple processing units.
4.	Concurrency increases the amount of work finished at a time.	While it improves the throughput and computational speed of the system.
5.	Concurrency deals lot of things simultaneously.	While it do lot of things simultaneously.
6.	Concurrency is the non-deterministic control flow approach.	While it is deterministic control flow approach.
7.	In concurrency debugging is very hard.	While in this debugging is also hard but simple than concurrency.

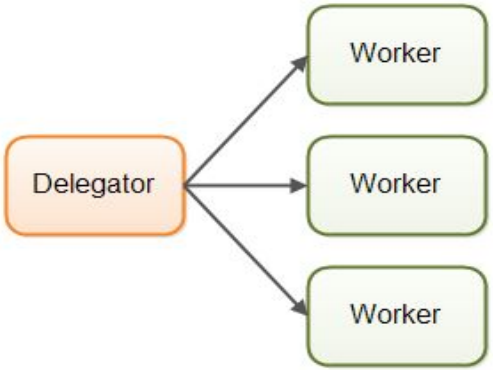
Shared State vs. Separate State

- **Shared state** means that the different threads in the system will share some state among them. By *state* is meant some data, typically one or more objects or similar. When threads share state, **problems like race conditions and deadlock etc**. may occur. It depends on how the threads use and access the shared objects, of course.
- **Separate state** means that the different threads in the system do not share any state among them. In case the different threads need to communicate, they do so either by exchanging immutable objects among them, or by sending copies of objects (or data) among them. **Thus, when no two threads write to the same object (data / state), you can avoid most of the common concurrency problems**.



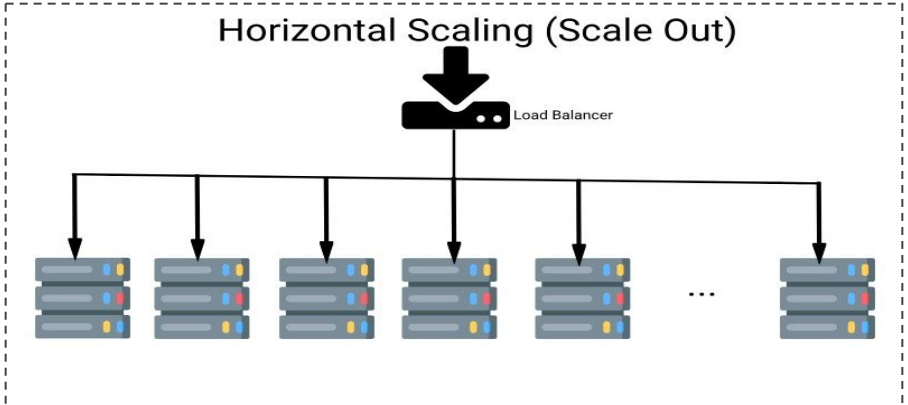
Parallel Workers

- The first concurrency model is what I call the *parallel workers* model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel workers concurrency model:
- The parallel workers concurrency model can be designed to use both shared state or separate state, meaning the workers either has access to some shared state (shared objects or data), or they have no shared stat



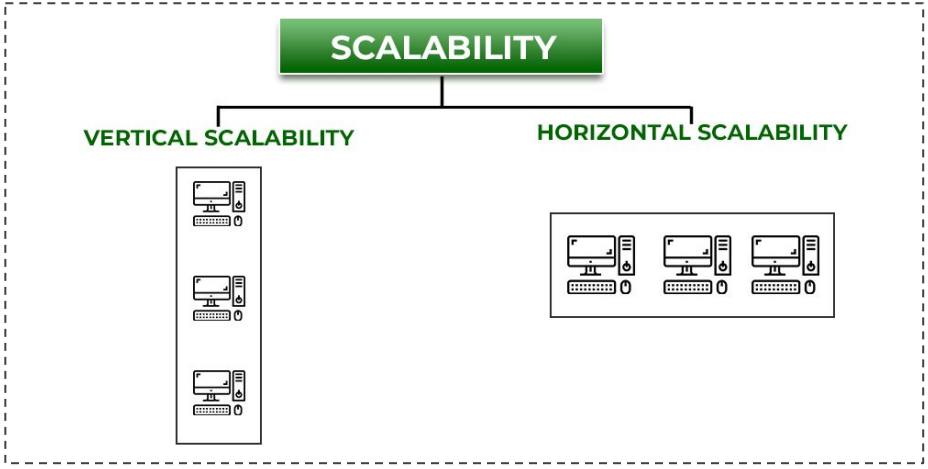
Thinking about performance

- While the goal may be to improve performance overall, using **multiple threads** always introduces **some performance costs compared to the single-threaded approach.**
- These include the overhead associated with coordinating between threads (locking, signaling, and memory synchronization), ***increased context switching,thread creation and teardown, and scheduling overhead.***
- When threading is employed effectively, these costs are more than made up for by **greater throughput, responsiveness, or capacity.**



Performance versus scalability

- **Scalability** describes the ability to **improve throughput** or capacity when additional computing resources (such as **additional CPUs, memory, storage, or I/O bandwidth**) are added.
- When tuning for performance, the goal is usually to do the same work with less effort, such as by reusing previously computed results **through caching or replacing an O(n^2) algorithm with an O(n log n)** one.



Amdahl’s law (1/n)

- **Amdahl’s law** describes how much a program can theoretically be sped up by additional computing resources, based on the proportion of parallelizable and serial components.
- If F is the fraction of the calculation that must be executed serially, then **Amdahl’s law** says that on a machine with N processors, we can achieve a speedup of at most:

$$S_p \leq \frac{1}{(1-f)+\frac{f}{p}}$$

  - **S** is the expected speedup of the entire job execution;
  - **p** denotes the speedup of the job that benefits from enhanced system resources;
  - **f** represents the fraction of execution time that the upgraded resource component initially occupied.

Amdahl's law (2/n)

- If you are using n processors, your **Speedupn is:**

$$Speedup_n = \frac{T_1}{T_n}$$

- where **T1 is the execution time** on one core and **Tn is the execution time on n cores**. Note that Speedupn should be > 1.

And your Speedup Efficiencyn is:

$$Efficiency_n = \frac{Speedup_n}{n}$$

which could be as high as 1., but probably never will be

Amdahl's law (3/n)

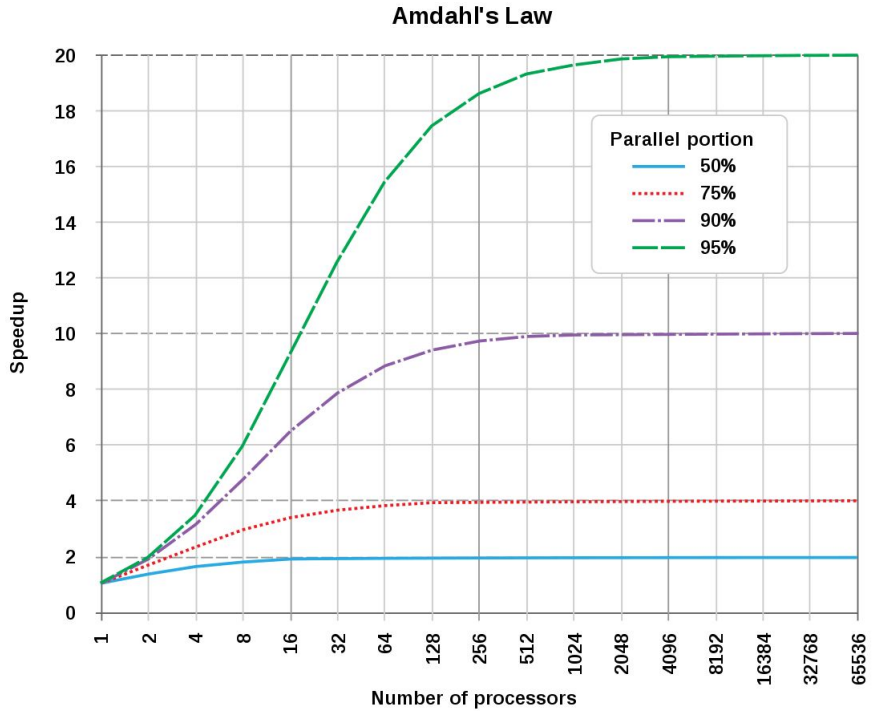
- **If you put in n processors, you should get n times Speedup (and 100% Speedup Efficiency)**, right? Wrong!
- There are always some fraction of the total operation that is inherently sequential and cannot be parallelized no matter what you do. **This includes reading data, setting up calculations, control logic, storing results, etc**
- If you think of all the operations that a program needs **to do as being divided between a fraction that is parallelizable and a fraction** that isn't (i.e., is stuck at being sequential), then Amdahl's Law says:

$$Speedup_n = \frac{T_1}{T_n} = \frac{1}{\frac{F_{parallel}}{n} + F_{sequential}} = \frac{1}{\frac{F_{parallel}}{n} + (1 - F_{parallel})}$$

This fraction can be reduced by deploying multiple processors.

This fraction can't.

Amdahl’s law (4/n)



Amdahl's law (5/n)

- To see how serialization can be hidden in the structure of an application, we can compare throughput as threads are added and infer differences in **serialization based on observed differences in scalability.**
- The curves in Figure 11.2 compare throughput for two thread-safe Queue implementations: **a LinkedList wrapped with synchronizedList, and a ConcurrentLinkedQueue**.

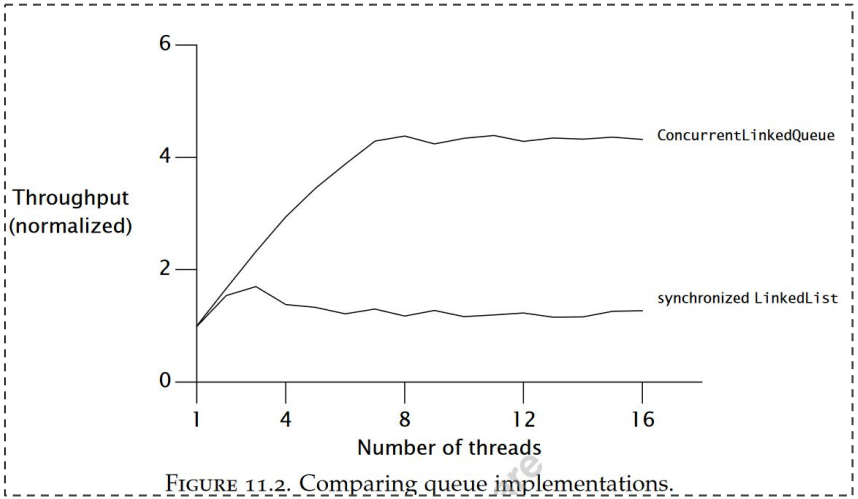


FIGURE 11.2. Comparing queue implementations.

Amdahl's law (6/n)

- Amdahl's law quantifies the possible speedup when more computing resources are available, if we can accurately estimate the fraction of execution that is serialized.
- Although measuring serialization directly can be difficult, Amdahl's law can still be useful without such measurement
- Algorithms that seem scalable on a four-way system may have hidden scalability bottlenecks that have just not yet been encountered.

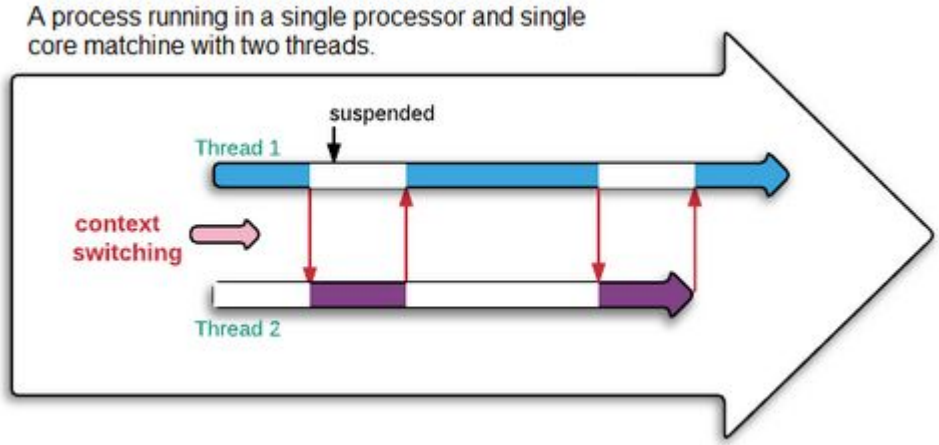
$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Serial part of job =
1 (100%) - Parallel part

Parallel part is divided up by N workers

Cost introduced by threads - Context switching (1/n)

- Single-threaded programs incur neither scheduling nor synchronization overhead, and need not use locks to preserve the consistency of data structures.
- Scheduling and interthread coordination have performance costs; for threads to offer a performance improvement, the performance benefits of parallelization must outweigh the costs introduced by concurrency.



Cost introduced by threads - Context switching (2/n)

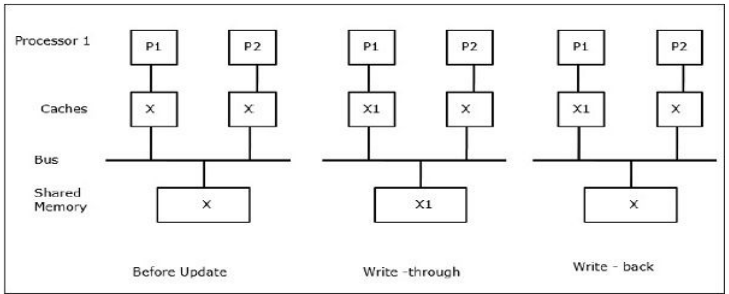
- IF there are **more runnable threads than CPU**s, eventually the OS will preempt one thread so that another can use the CPU.
- **This causes a context switch**, which requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread.
- **Context switches are not free;** thread scheduling requires manipulating shared data structures in the OS and JVM.
- **The OS and JVM use the same CPUs your program does**; more CPU time spent in **JVM and OS code means less is available for your program**.

```
synchronized (new Object()) {
    // do something
}
```

LISTING 11.2. Synchronization that has no effect. *Don't do this.*

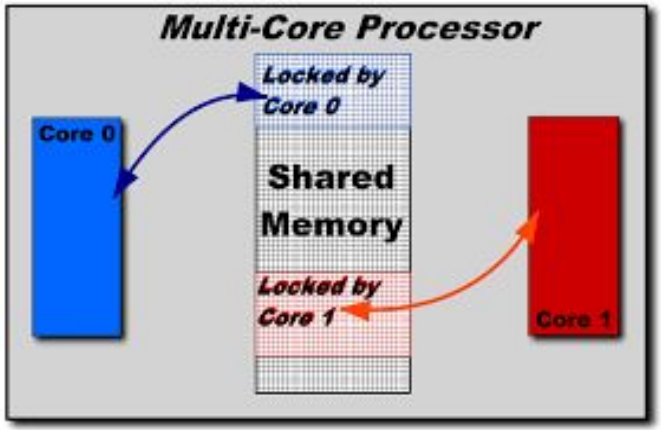
Cost introduced by threads - Memory synchronization(1/n)

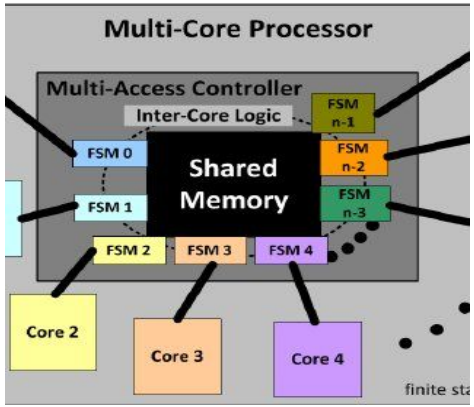
- The actual cost of **context switching** varies across platforms, but a good rule of thumb is that **a context switch costs the equivalent of 5,000 to 10,000 clock cycles**, or several microseconds on most current processors.
- **The visibility guarantees provided** by synchronized and volatile may entail using special instructions called memory barriers that can **flush or invalidate caches, flush hardware write buffers, and stall execution pipeline**s.
- The synchronized mechanism is optimized for the uncontended case (**volatile** is always uncontended), and at this writing, the performance cost of a “fast-path” uncontended synchronization ranges from **20 to 250 clock cycles for most systems**

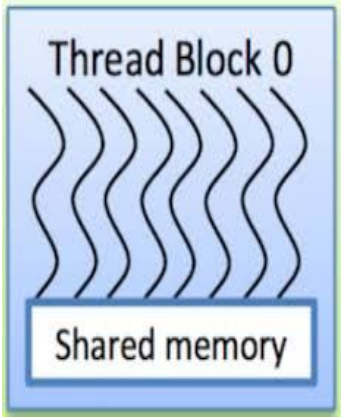


Cost introduced by threads - Memory synchronization (2/n)

- While this is certainly not zero, the effect of needed, uncontended synchronization is rarely significant in overall application performance,
- and the alternative involves compromising safety and potentially signing yourself (or your successor) up for some very painful bug hunting later.







Reducing lock granularity (1/n)

Уменьшение детализации блокировки

- This can be accomplished by **lock splitting and lock striping**, which involve using separate locks to guard multiple independent state variables previously guarded by a single lock.
- These techniques **reduce the granularity at which locking occurs, potentially allowing greater scalability**—but using more locks also i**ncreases the risk of deadlock.**

```
@ThreadSafe
public class ServerStatus {
    @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);
    }
}
```

LISTING 11.6. Candidate for lock splitting.

Reducing lock granularity (2/n)

- As a thought experiment, imagine what would happen if there was only one lock for **the entire application instead of a separate lock for each object**. Then execution of **all synchronized blocks, regardless of their lock, would be serialized.**
- If **a lock guards** more than one independent state variable, you may be able to **improve scalability by splitting it into multiple locks** that each guard different variables. This results in each lock being requested less often.
- Instead of **guarding both users and queries with the ServerStatus lock**, we can instead guard each with **a separate lock**, as shown in Listing 11.7. **After splitting the lock, each new finer-grained lock will see less locking traffic** than the original coarse lock would have.

Reducing lock granularity (3/n)

```
@ThreadSafe
public class ServerStatus {
    @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);
        }
    }
    // remove methods similarly refactored to use split locks
}
```

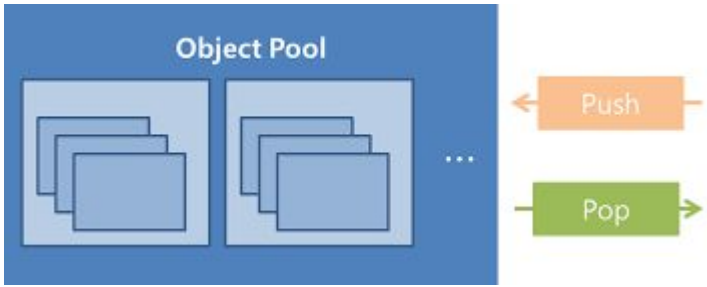
LISTING 11.7. ServerStatus refactored to use split locks.

Reducing lock granularity (4/n)

- **Splitting locks** that are experiencing little contention yields little net improvement in performance or throughput, although it might increase the load threshold at which performance starts to degrade due to contention.
- **Splitting locks** experiencing moderate convention might actually turn them into **mostly uncontended locks**, which is t**he most desirable outcome for both performance and scalability.**

Reducing lock granularity (5/n)-Just say no to object pooling

- In early JVM versions, object allocation and garbage collection were slow, 13 but their performance has improved substantially since then.
- In fact, allocation in **Java is now faster than malloc is in C:** the common code path for new Object in HotSpot 1.4.x and 5.0 is approximately ten machine instructions.
- To work around “slow” object lifecycles, many developers turned to object pooling, where objects are recycled instead of being garbage collected and allocated anew when needed.
- Even taking into account its reduced garbage collection overhead, object pooling has been shown to be a performance loss14 for all but the most expensive objects (and a serious loss for light- and medium-weight objects) in single-threaded programs



Reducing lock granularity (6/n)

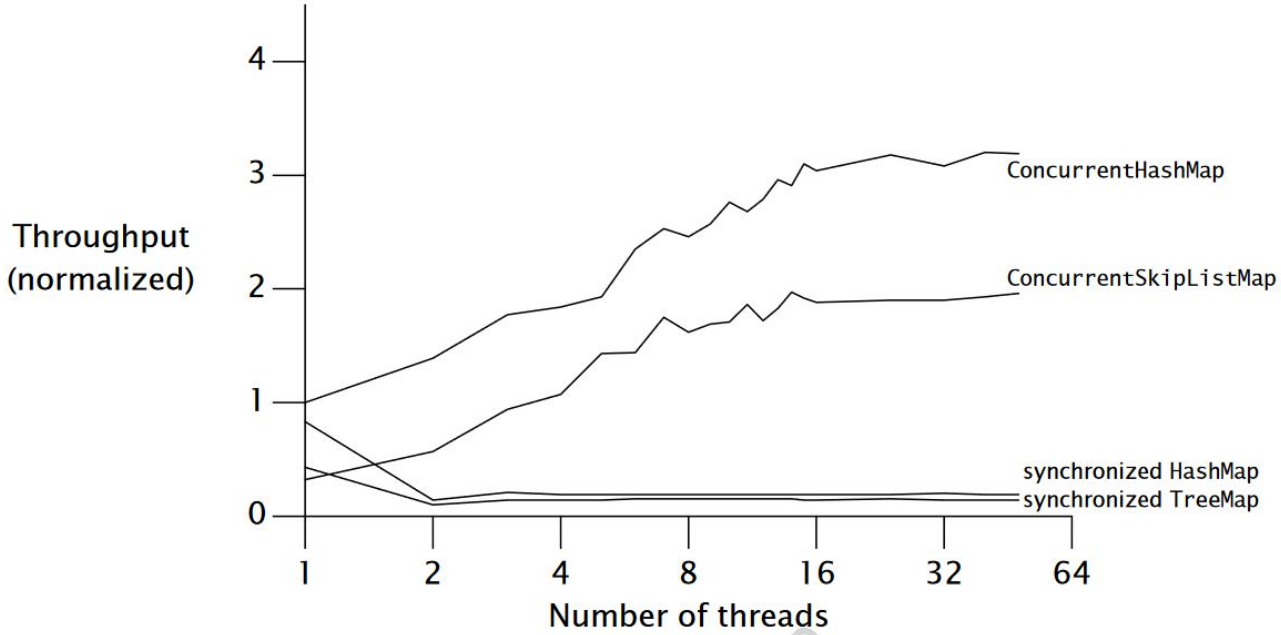
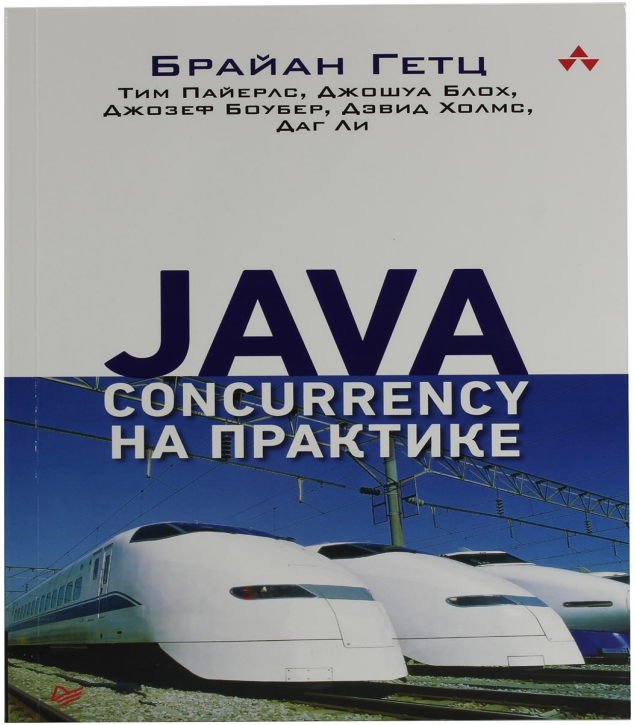


FIGURE 11.3. Comparing scalability of Map implementations.

Summary

- the most common reasons to use **threads** is to exploit **multiple processors**, in discussing the performance of concurrent applications, **we are usually more concerned with throughput or scalability** than we are with raw service time.
- **Amdahl's law** tells us that **the scalability of an application is driven by the proportion of code that must be executed serially.**
- Since **the primary source of serialization** in Java programs is the exclusive resource **lock**, **scalability** can often be improved by **spending less time holding locks** either by reducing **lock granularity, reducing the duration** for which **locks are held, or replacing exclusive locks with nonexclusive or nonblocking alternatives.**

Resources


Брайан Гетц
Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли
JAVA
CONCURRENCY
НА ПРАКТИКЕ

Reference

1. Java Concurrency book.
2. Concurrency-vs-parallelism
3. Java-concurrency
4. amdahls-law-and-its-proof
5. Admahls law
6. CUDA

Thank you!

Presented by

Hamdamboy Urunov

(hamdamboy.urunov@gmail.com)