Improving performance can increase complexity, increasing the likelihood of safety and liveness failures.

First, make your program right, then if performance requirements tell you so make it faster.

Thinking about performance

Performance and scalability

Resources: CPU cycles, memory, network bandwidth, IO bandwidth, database requests, disk space, etc.
When the performance of the activity is limited by availability of a particular resource, we say it is **bounded by that resource**: CPU-bound, for instance. Performance is discussed in the context of the resource.

Using threads always introduces some performance costs compared to the single-threaded approach:

- overhead associated with coordinating between threads (locking, signaling, memory synchronization)
- increased context switching
- thread creation and teardown
- scheduling overhead

Two ways to measure application performance

- **how fast** a given unit of work can be processed
  - service time, latency
  - area of **traditional performance optimizations**: caching, using optimal algorithms
- **how much** work can be performed with a given quantity of computing resources
  - capacity, throughput
  - area of **scalability optimizations**

These two aspects of performance are completely separate, sometimes even at odds with each other

- Scalability is usually of greater concerns for server applications
- For interactive applications lateny aspetcs (traditional performance) tend to be more important

Evaluating performance tradeoffs

Avoid premature optimizations. First make it right, then make it fast — if it is not already fast enough.

Then **use conservative approach** in tuning performance

- What do you mean by faster?
- Under which conditions will this approach be actually faster? Light or heavy load? Large or small data sets? Support the answer with measurements.
- How often are these conditions likely to arise in your situation? Support the answer with measurements.
- Is the code likely to be used in other situations where the conditions may be different?
- What hidden costs are you trading for this improved performance? Is this a good tradeoff?

Why you should take conservative approach in tuning performance:

- The quest for performance is probably the single greatest source of concurrency bugs which are amongh the most difficult bugs to track down and eliminate. Trade carefully.
- Worse, trading something for performance you can get neither.

Tradeoffs:

- one cost for another (service time versus memory consumtion)
- trading cost for safety
- trading cost for readability or maintainability
  - clever or non-obvious code
  - compromising good object-oriented design principles, such as breaking encapsulation
  - risk of error, faster algorithms are more complicated

Measure, never guess. Always support your reasoning about performance with measurements: the intuition of where the performance problem lies is often incorrect.

Amdahl's law

Speedup <= 1 / (F + (1 - F) / N)
where

- **F** in [0; 1] is the fraction of calculation that must be executed serially
- **1 - F** is the fraction of calculation that can be executed in parllel
- **N** is the amount of processors
- **(1 - F) / N** is the time that **1 - F** part of calculation takes on **N** processors

Utilization is the **speedup** divided by the number of resources **N**.

- Perfectly, utilization is 1, meaning **N** added reousrces produce speedup=N, and N/N=1
- If you add N resources and get speedup=P<N then you've actually utilized P/N of added resources, hence the term **utilization**

The serial fraction of computaion must be identified carefully

- It can include time to fetch data being processed from queue/dequeue (less for non-blocking lists)
- It must include some time for result handling. Time to merge the results or for producing a side-effect.

Applying Amdahl's law qualitively

The law quantifies the possible speedup if we can accurately estimate the fraction of execution that is serialized. Also, this law can be used without such measurement.

When evaluating an algorithm, thinking "in the limit" about what would happen with hundreds or thousands of processors can offer some insight into where scaling might appear.

Costs introduced by threads

For threads to offer a performance improvement, the performance benefits of parallelization must outweight the costs introduced by concurrency.

Context switching

If there are more than one runnable threads than CPUs, eventually the OS will preempt one thread to start another. This causes a context switch, which requires

- saving the execution context of the currently running thread
- restoring the execution context of the newly scheduled thread

Why context switches are not free

- Thread scheduling requires manipulating shared data structures in the OS and JVM. This activiy takes CPU time.
- When a new thread is switched in, the data it needs is unlikely to be in the local processor cache, so a context switch causes a flurry of cache misses, threads run a little more slowly when they are first scheduled. (This is why schedulers give threads some minimum time quantum to amortize the cose of the context switch, improving overall throughput.)
- When a thread blocks (waiting for a contended lock, blocking on IO) it is suspended by the JVM and is allowed to be switched out. If the thread blocks frequently, it will be unable to use their full scheduling quantum.

vmstat Unix tool shows the number of context switches and the percentage of time spent in the kernel. High kernel usage indicates heavy scheduling activity, which may be caused by blocking due to IO or lock contention.

Memory synchronization

Sources:

- 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, stall execution pipelines.
- Indirect performance consequences because they inhibit other compiler optimizations, for examples, many operations can not be reordered with memory barriers.
- Synchronization creates traffic on the shared memory bus which has a limited bandwidth and is shared across all processors. Threads start competing for synchronization bandwidth, all threads using synchronization will suffer.

It is important to distinguish between **contended** and **uncontended** synchronizations, because **synchronized** mechanism is optimized for the **uncontended** case (**volatile** is always **unconteded**). Focues optimization efforts on areas where lock contention actually occurs.

- The performance cost of a "fast-path" uncontended synchronization ranges from 50 to 250 clock cycles, which is rarely significant in overall application performance, and the alternative involves compromising safety.
- JVMs can reduce the cost of incidental syncrhonization by optimizaing away locking that can be proven to never contend. For examples, when a lock object is accessible only to the current thread.
- JVMs can use escape analysis to identify when a local object reference is never published to the heap. All lock acquisitions can be eliminated.
- Compilers can perform lock coarsening, the merging of adjacent **synchronized** blocks using the same lock. This may also enable more optimizations inside the larger block of code.

Blocking

When locking is **contended**, the losing thread must block.

Blocking can be implemented via spin-waiting or by suspending the blocked thread through the OS. Spin-waiting is preferable for short waits and suspension is preferable for long waits. Some JVMs choose adaptively but most suspend a thread.

Suspending a thread entails two additional context switches:

- the blocked thread is switched out before its quantum is expired
- swithced back when the lock or resource is available
- blocking due to the lock contention: when a thread releases the lock it must then ask the OS to resume the blocked thread

Reducing lock contention

Lock contention causes both **serialization** and **context switches**

- **Serialization** hurts **scalability**
- **Context switches** hurt **performance**

The corollary of Little's law, a result from queueing theory, says **"the average number of customers in a stable system is equal to their average arrival rate multiplied by their average time in the system".**

Hence 3 ways to reduce lock contention:

- Reduce the duration for which locks are held
- Reduce the frequency with which locks are requested
- Replace exclusive locks with coordination mechanisms that permit greater concurrency

First. Narrowing lock scope

Hold locks as briefly as possible:

- move code that does not require the lock out of **synchronized** block
- especially for expensive operations and potentially blocking operations such as I/O.

Second. Reducing lock granularity

Reducing the time the lock is held — have threads to ask for it less often

- **lock splittiing**
- **lock striping**

The disadvantage is that the risk of deadlock increases.

Splitting locks:

- offers greatest possibility for improvement when the lock experiences moderate but not heavy contention, might actually turn them into mostly uncontended locks
- for locks experiencing little contention this yields little improvement in performance or throughput, but might increase the load threshold at which performance starts to degrade due to contention

Splitting a heavily contended lock into two is likely to result in two heavily contended locks. Lock splitting can be extended to **lock striping** — partitioning locking on a variable-sized set of independent objects. (Implementation of **ConcurrentHashMap** uses an array of 16 locks, each of which guards 1/16 of the hash buckets).

The disadvantage of **striping locks** is that locking collection for exclusive access is more difficult and costly than with a single lock, all locks must be acquired.

A program that would benefit from lock splittin or striping necessarily exhibits contention for a lock more often than for the data guarded by that lock. If a lock guards two independent vairables X and Y, and thread A watns to access X while B wants to access Y, then the two threads are not contending for any data, even though they are contending for a lock.

Lock granularity can not be reduced when there are variables that are required for every operation.

- Common optimizations such as caching frequently computed values can introduce "hot fields" that limit scalability
- Another area where scalability and raw performance come at odds with each other

For example, implementing the size() method of a collection as a cached value may improve performance but makes it much harder to improve scalability. **ConcurrentHashMap** avoids this problem by storing a counter for each stripe.

Third. Alternatives to exclusive locks

Forego the use of exclusive locks in favor of a more concurrency-friendly means of managing shared state:

- **concurrent collections**
- **read-write locks**, can offer greater concurrency for read-mostly data structures
- **immutable objects**, can eliminate the need of locking for read-only objects
- **atomic variables**, offer means of reducing the cost of updating "hot fields" by providing very fine-grained (and scalable) atomic operations, and are implemented using low-level concurrency primitives (compare-and-swap) provided by most modern prcoesssors

Monitoring CPU utilization

When testing for scalability the goal is to keep processors fully utilized. Asymmetric utilization indicates that most of the computation is going on in a small set of threads. Several likely causes:

- **Insufficient load**. It may be that the application tested is not subjected to enough load. Increase the load.
- **I/O** bound. Determine whether applicaiton is disk-bound using iostat, or whether it is bandwidth-limited by monitoring traffic levels.
- **Externally bound**. If your application depends on external services such as a database or web service, the problem may not be in your code.
- **Lock contention**. Use profiling tools or sampling to figure out how much lock contention your application is experiencing. Heavily contended locks will almost always have at least one thread waiting to acquire it and will frequently appear in thread dumps in the appropriate stack frame as "waiting to lock monitor…".
- If CPU utilization is high and there are always runnable threads waiting for CPUs, your application would probably benefit from more processors.

Avoid object pooling

Object pooling has been shown to be a performance loss for all but the most expensive objects in **single-threaded programs**

- It's a loss in terms of CPU time, memory allocation and garbage collection would work faster
- The challenge of setting pool size incorrectly: too small (pooling has no effect), too large (puts pressure on garbage collector and retains memory that could be useful)
- The risk that an object will not be properly reset to its newly allocated state, introducing subtle bugs
- The risk that an object will return the object to the pool but will continue to use it
- Makes more work for generational GC by encouraging a pattern of old-to-young references.

In concurrent applciations

- When thread allocate new objects, very few inter-thread coordination is required, allocators typically used thread-local allocation blocks to eliminate synchronization on heap data structures. But requesting an object from a pool requires some synchronization, introducing the possibilty that a thread will block, which is hundreds times more expensive than allocating.
- Pooling has its very limited uses. Even then it could become a scalability issue.

Reducing context switch overhead

One source of blocking in server applications is generating log messages in the course of processing internals. Approaches:

- When you have something to log, just write it out right then and there
- Perform loggin in a dedicated background thread, but has some design implications
  - **Interruptions** — what happens if thread blocked in a loggin operation is interrupted?
  - **Service guarantees** — does the logger guarantee that a successfully logged message will be written prior to services shut down?
  - **Saturation policy** — what happens when the producer log messages faster then the logger thread can handle them?
  - **Service lifecycle** — how do we shut down the logger and how do we communicate the service state to producers?

Performance and scalability complications of two approaches

- Service time for a logging operation includes
  - Computations accosicated with the I/O stream classes
  - If I/O blocks, it includes the duration for which the thread is blocked
  - The OS will deschedule the blocked thread until the I/O completes or probably a little longer
  - When the I/O is finished other threads will still use their quanta and other threads will be scheduled ahead of us
  - If multiple threads are logging simultaneously, there may be a contention for the output stream lock, the result is the same as with blocking I/O

- Inline logging involves I/O and locking which can lead to increased context switching and increased service times.

Increasing service time is undesireable for 2 reasons

- Service time affects quality of service: someone waits longer for a result
- More lock contention, and contended lock aqcuisition means more context switches which yields lower overall throughput

If we move blocking to another thread:

- We shorten the mean service time for request processing
- Threads need only to queue a message which is less likely to contend then logging I/O and contending for the output stream, and request thread is less likely to blocked and incur context switching in the middle of processing requests