Queueing Theory in Practice

Link to video

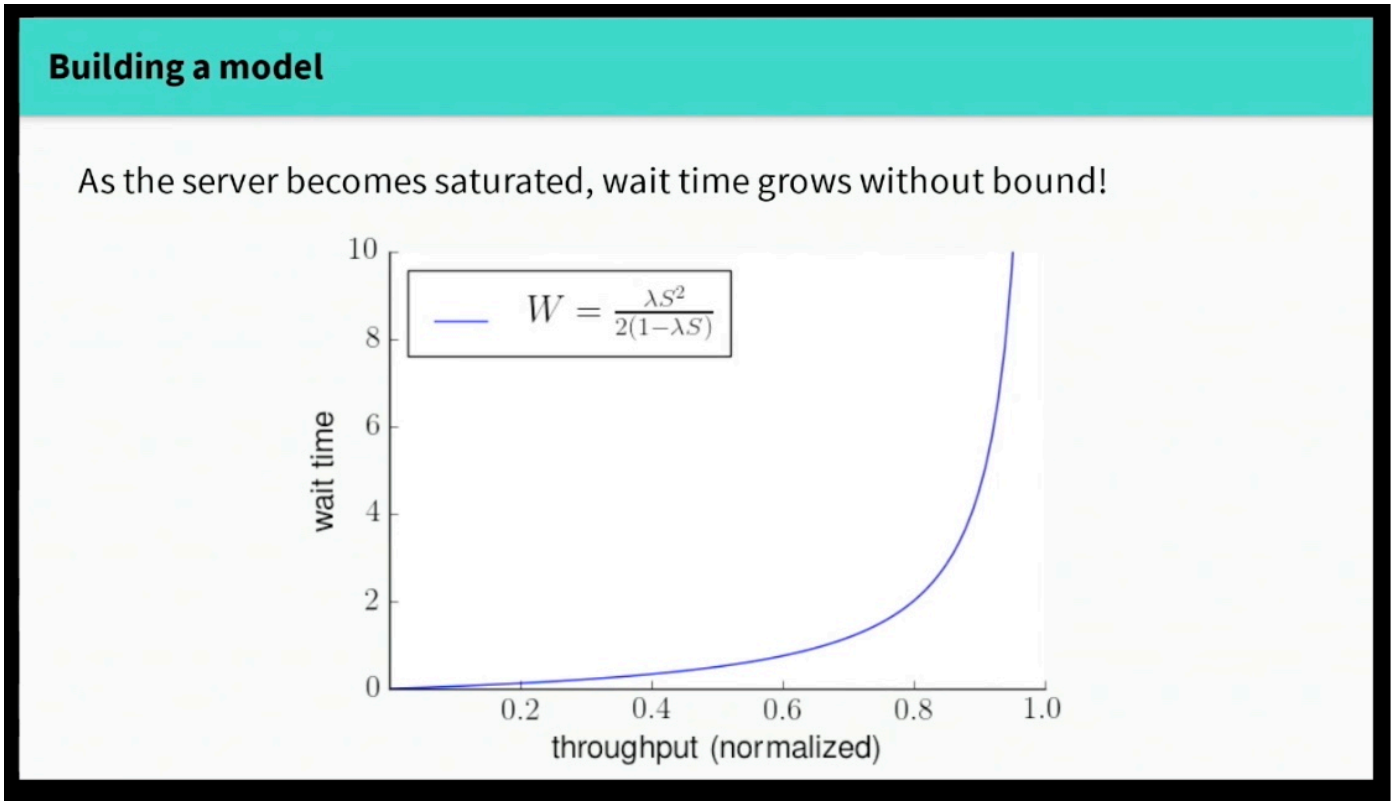
Serial Systems

Take a concurrent, CPU-bound, low latency API. How do we allocate resources to this properly?

Take a **single-queue / single-queue** model. Say:

- Tasks arrive independently and randomly at average rate `R`.
- Server takes time `S` to process each task.
- Server processes each task one at a time.

So using some simple geometry with the above variables, you can derive an equation for service latency vs. throughput that maps freakishly well onto actual service latency tests:



Given that equation, you can see based on time `S` to process each task that improving your service's performance per task will lead to easily the best performance improvement in latency as throughput increases!

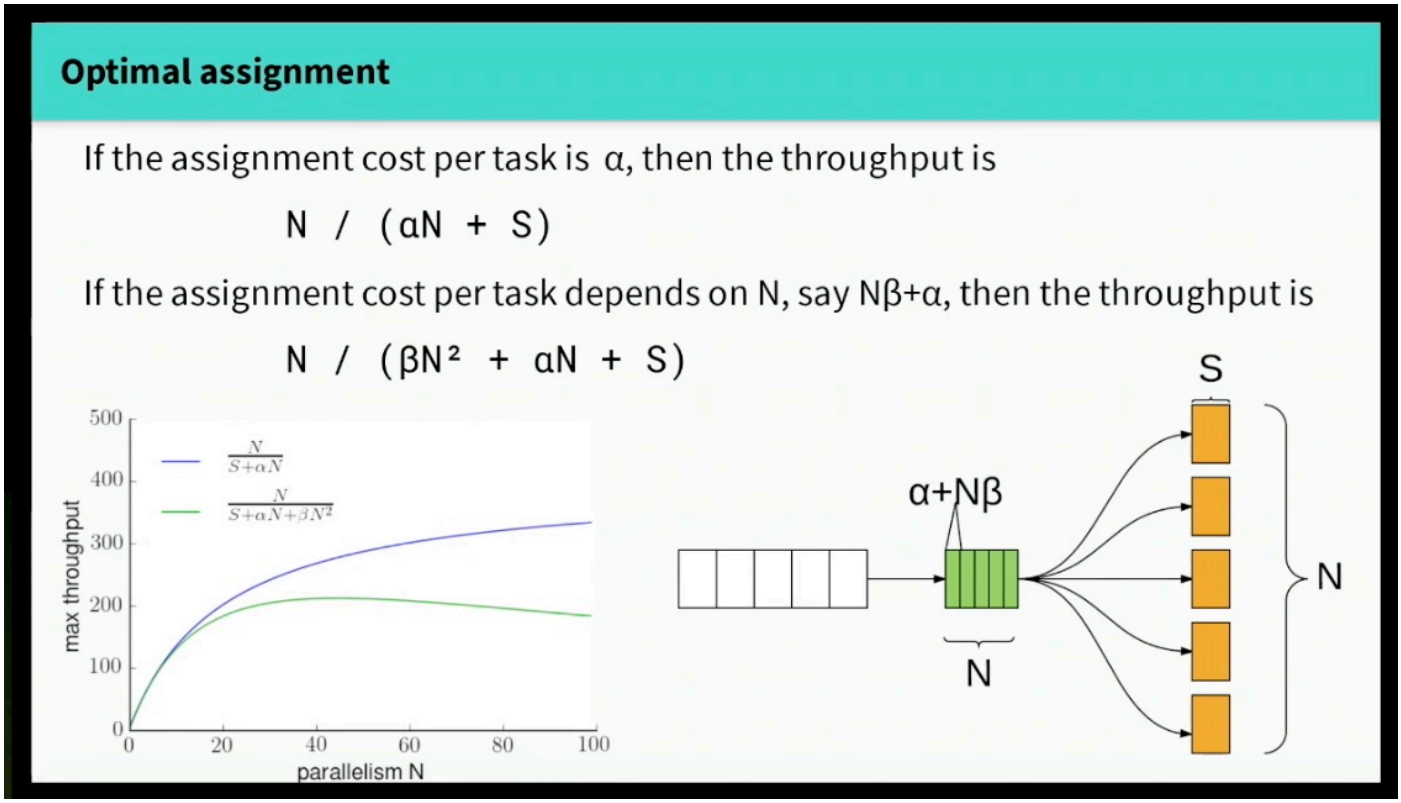
This relies, of course, on a constant time `S` and low variance in arrival rate. We should control these knobs through batching, concurrency control, back-pressure control, fast preemptions, etc.

Parallel Systems

How about the performance of a **fleet of servers**?

If each server handles `T` tasks, can `N` servers handle `N * T` tasks? Depends on how we assign tasks - round robin, random, least busy server?

Well, least busy would be nice, but you need to coordinate with your servers (via some routing layer) to figure out what the least busy server is. It takes time to interrogate servers for their load: `N / (aN + S)` is the throughput of concurrent servers, where `N` is number of nodes, `a` is the coordination cost, and `S` is still the time to process a task. It gets worse if the time to coordinate is related to the number of nodes `N`



So how can we find the balance between coordination at low parallelism versus high parallelism? Coordination helps for low parallelism, but not for high!

Two ideas:

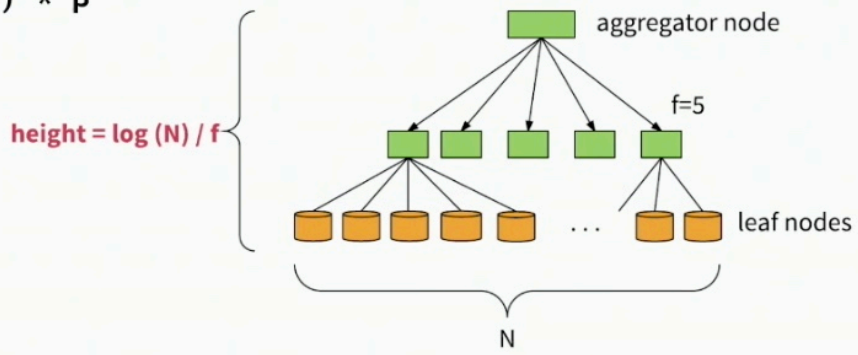
1. Approximate assignment — pick two random servers and choose best. Constant coordination time with scale.
2. Iterative partitioning — remember the Universal Scalability Law applies to *all* parallel processes. Process time per node decreases with scale, but coordination time increases. Think about Spark – what if we had a secondary aggregation layer to keep fanout constant per aggregation node, and do aggregation on those intermediate nodes?

Beating the beta factor

Idea: multi-level query fanout

add intermediate aggregators, make fanout a constant *f*

```
T(total) = S / N + (height of tree) * f * β
         = S / N + log(N) / f * f * β
         = S / N + log(N) * β
```

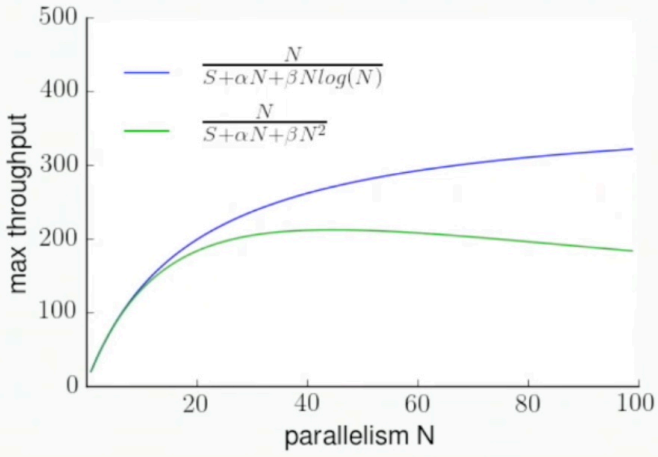


Beating the beta factor

before: `T(total) = S / N + N * β`

now: `T(total) = S / N + log(N) * β`

Result: better scaling!



Woo! Better! Chief takeaway: coordination has costs, but we can be clever by modelling and looking for better big-O performance.