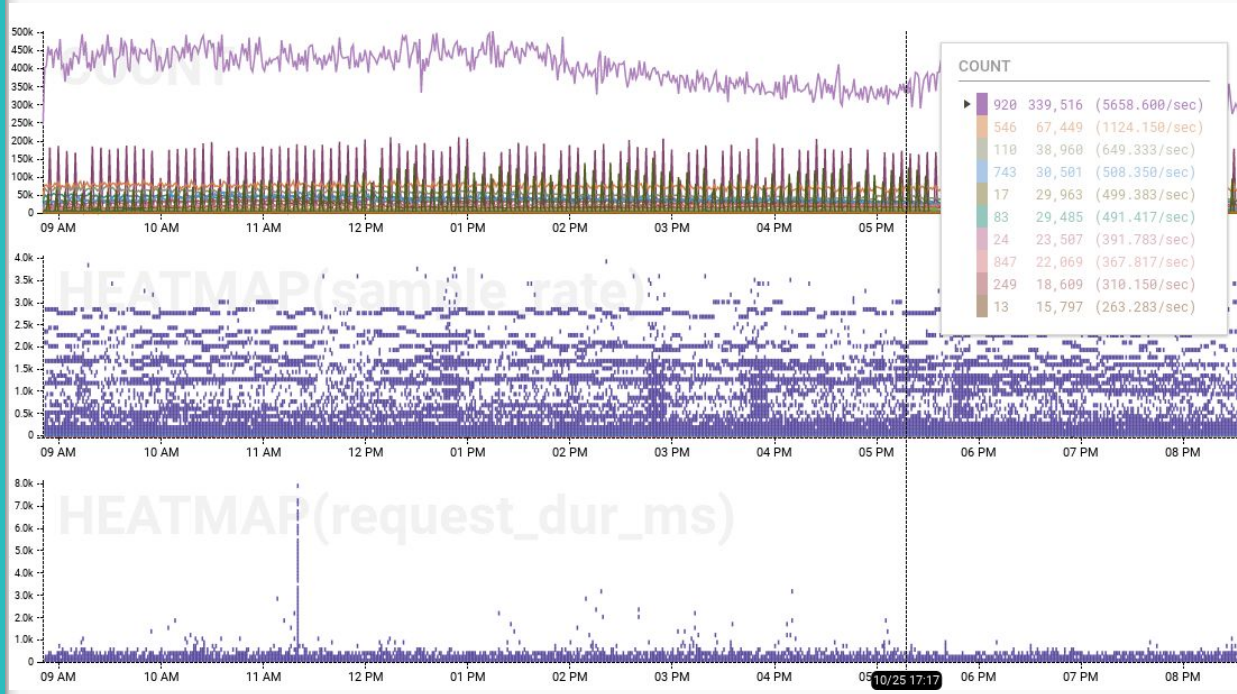
Queueing Theory, In Practice

Performance Modelling in Cloud-Native Territory

Eben Freeman
@_emfree_ | honeycomb.io

Hi, I'm Eben!

currently: building cool stuff at honeycomb.io



(come talk to me afterwards about tracing, events, observability for distributed systems . . .)

Myth: Queueing theory combines the tedium of waiting in lines with the drudgery of abstract math.

Lemma 2 (Serial Fraction). *The serial fraction (2.1) can be expressed in terms of M/M/1//p metrics by the identity [6, 10, 11]:*

$$\sigma = \frac{S}{S+Z} \rightarrow \begin{cases} 0 & \text{as } S \rightarrow 0,\ Z = const., \\ 1 & \text{as } Z \rightarrow 0,\ S = const. \end{cases} \tag{10}$$

If $\sigma = 0$, then there is no communication between processors and the interconnect latency therefore vanishes (maximal execution time). Conversely, if $\sigma = 1$, then the execution time vanishes (maximal communication latency).

Proof. The RTT for a single (unpartitioned) task in Fig. 2 is $T_1 = S + Z$. The RTT for a p equipartitioned subtasks is $T_p = p(S/p) + Z/p$. From definition 1, the corresponding speedup is

$$S_p = \frac{S+Z}{S+Z/p} \tag{11}$$

Equating (11) with (1), we find

$$S = \sigma T_1 \quad \text{and} \quad Z = (1-\sigma)T_1\,. \tag{12}$$

Eliminating T_1 produces

Reality: it’s all about asking questions

What target utilization is appropriate for our service?

If we double available concurrency, will capacity double?

How much speedup do we expect from parallelizing queries?

Is it worth it for us to spend time on performance optimization?

Reality: it’s all about asking questions

Queueing theory gives us a **vocabulary** and a **toolkit** to

- **approximate** software systems with models
- **reason** about their behavior
- **interpret** data we collect
- **understand** our systems better.

In this talk

I. Modelling serial systems

Building and applying a simple model

II. Modelling parallel systems

Load balancing and the Universal Scalability Law

III. Takeaways

Caveat!

Any model is reductive, and worthless without real data!

Production data and experiments are still essential.

Caveat!

Any model is reductive, and worthless without real data!

Production data and experiments are still essential.

But **having** a model is key to *interpreting* that data:

"*Service latency starts increasing measurably at 50% utilization. Is that expected?*"

"*This benchmark uses fixed-size payloads, but our production workloads are variable. Does that matter?*"

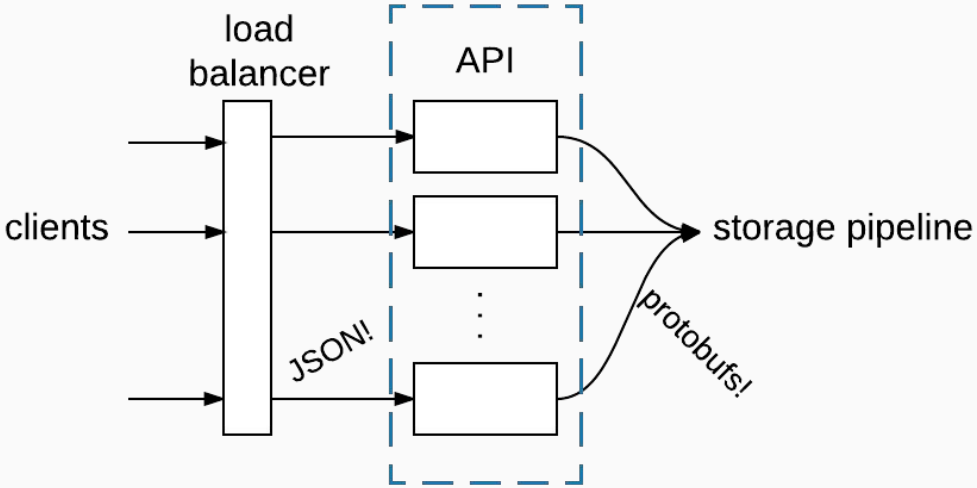
"This change increases throughput, but makes latency less consistent. Is that a good tradeoff for us?"

I. Serial Systems

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency

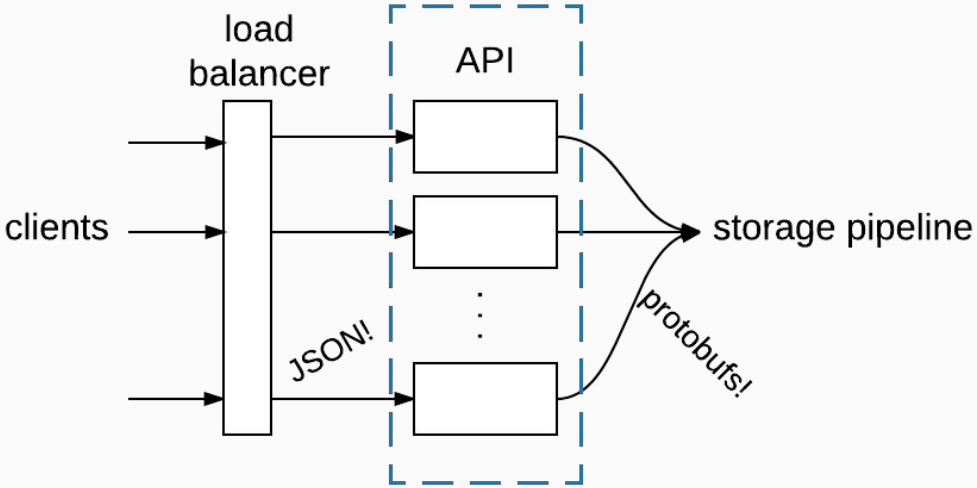


Question: How do we allocate appropriate resources for this service?

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency



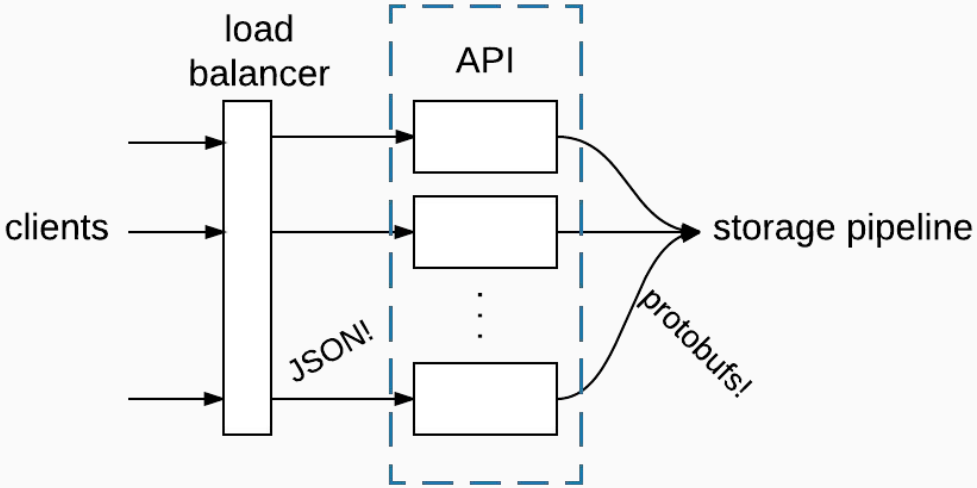
Question: How do we allocate appropriate resources for this service?

- Guesswork

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency



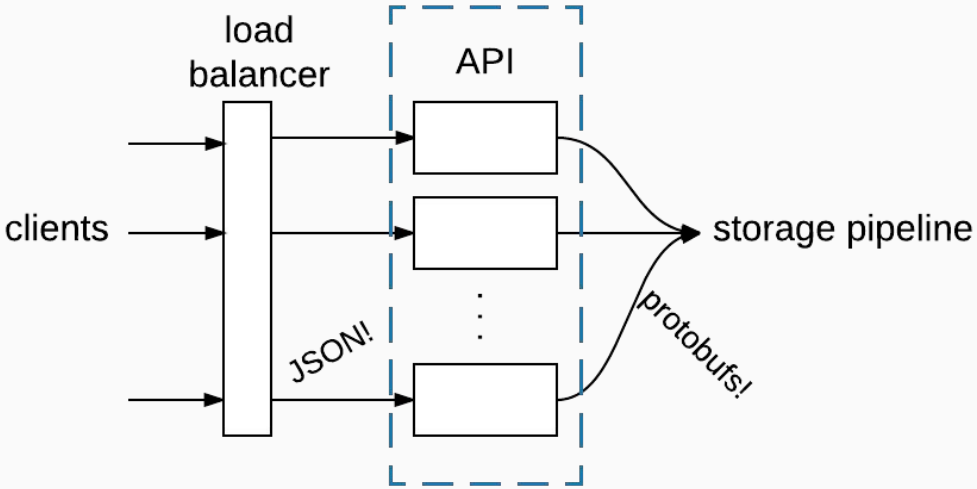
Question: How do we allocate appropriate resources for this service?

- ~~Guesswork~~

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency



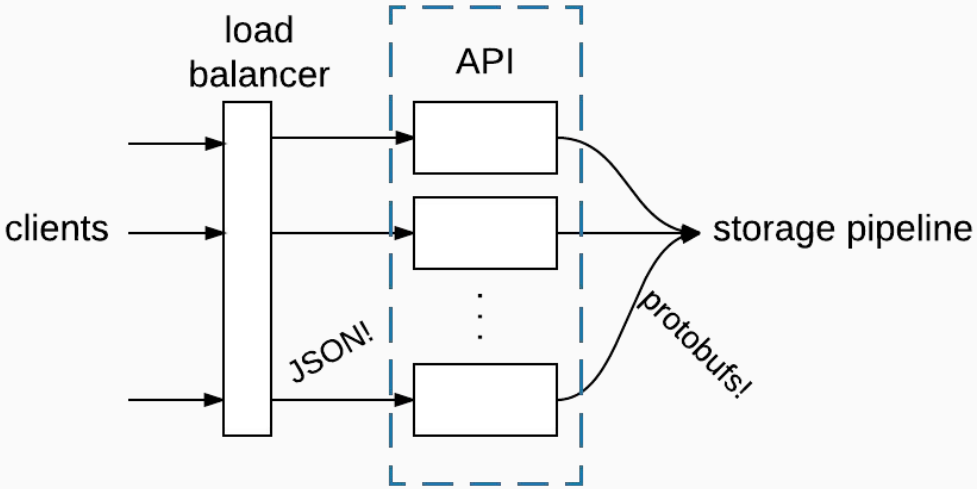
Question: How do we allocate appropriate resources for this service?

- ~~Guesswork~~
- Production-scale load testing

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency



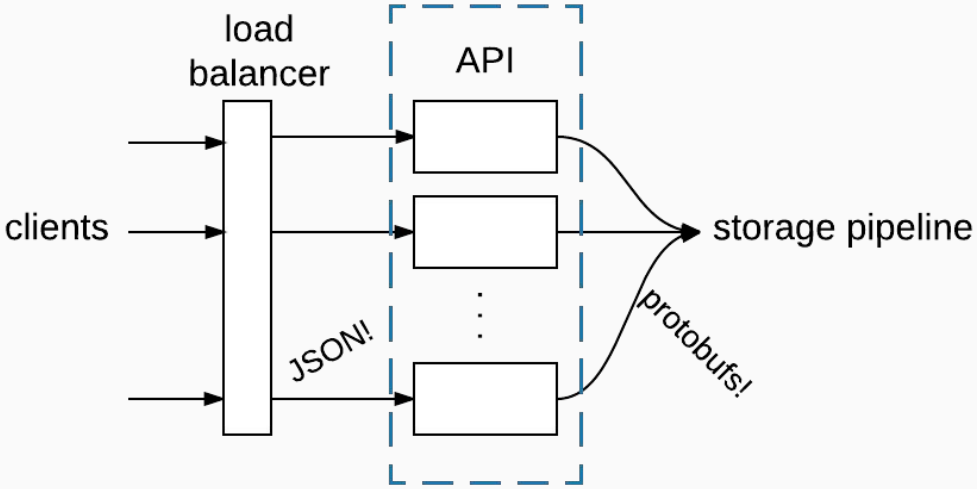
Question: How do we allocate appropriate resources for this service?

- ~~Guesswork~~
- ~~Production-scale load testing~~ (yes! but time-consuming)

A case study

The Honeycomb API service

- Receives data from customers
- Highly concurrent
- Mostly CPU-bound
- Low-latency



Question: How do we allocate appropriate resources for this service?

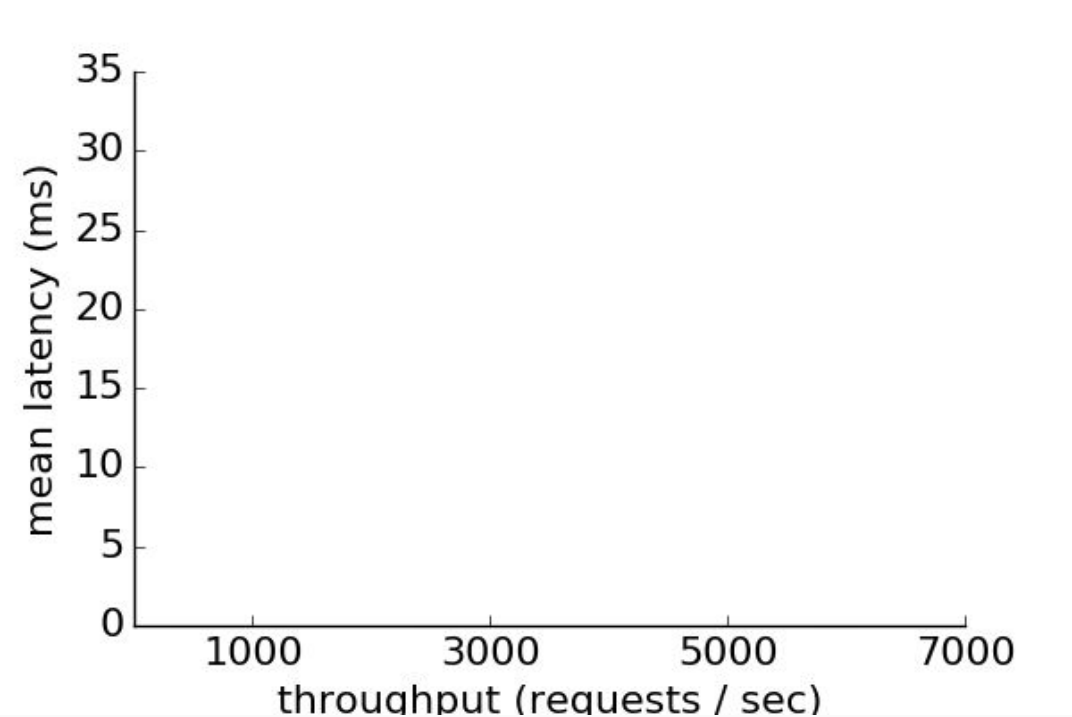
- ~~Guesswork~~
- ~~Production-scale load testing~~ (yes! but time-consuming)
- Small experiments plus modelling

An experiment

Question: What's the maximal single-core throughput of this service?

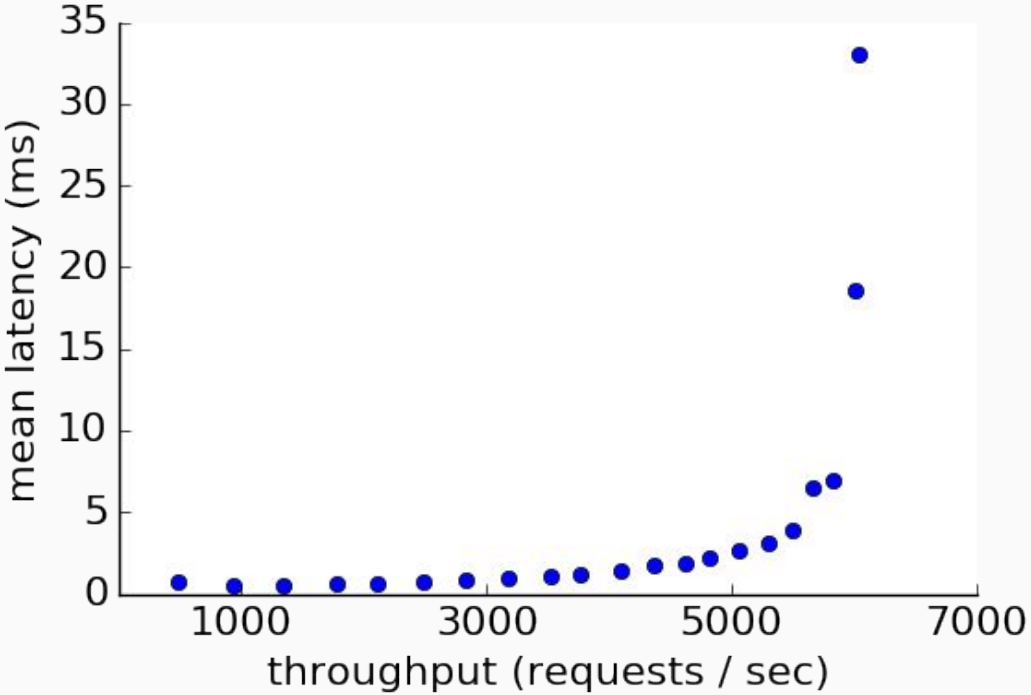
- Simulate requests arriving uniformly at random
- Measure latency at different levels of throughput

An experiment

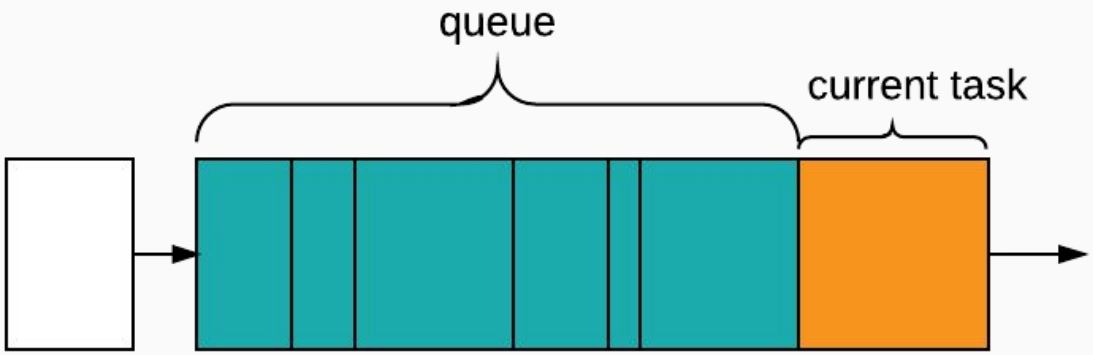


Our question

Can we find a model that predicts this behavior?



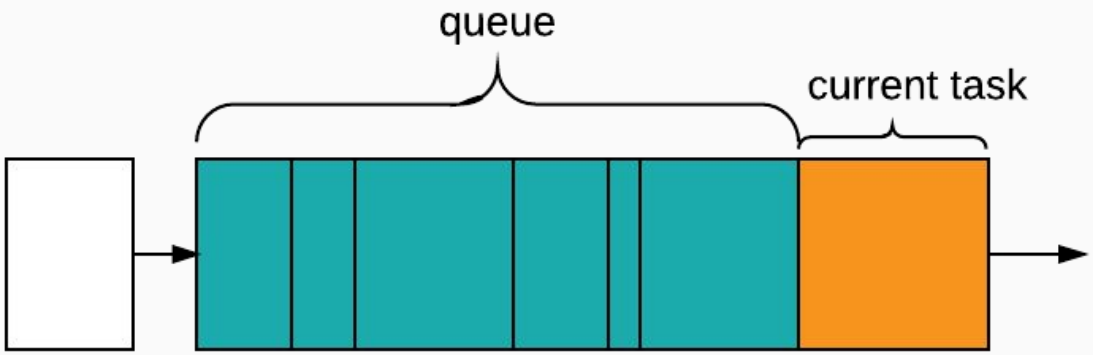
A single-queue / single-server model



Step 1: identify the question

- The busier the server is, the longer tasks have to wait before being completed.
- *How much* longer as a function of throughput?

A single-queue / single-server model



Step 2: identify assumptions about our system

- Tasks arrive *independently and randomly* at an average rate λ.
- The server takes a constant time S, the *service time*, to process each task.
- The server processes one task at a time.

Building a model

Step 3: gnarly math

Here are the full balance equations:

State	Balance equation	Simplified equation
$0:$	$\pi_0\lambda = \pi_1\mu$	$\Rightarrow \pi_1 = \frac{\lambda}{\mu}\pi_0$
$1:$	$\pi_1(\lambda+\mu) = \pi_0\lambda + \pi_2\mu$	$\Rightarrow \pi_2 = \left(\frac{\lambda}{\mu}\right)^2 \pi_0$
$2:$	$\pi_2(\lambda+\mu) = \pi_1\lambda + \pi_3\mu$	$\Rightarrow \pi_3 = \left(\frac{\lambda}{\mu}\right)^3 \pi_0$
$i:$	$\pi_i(\lambda+\mu) = \pi_{i-1}\lambda + \pi_{i+1}\mu$	$\Rightarrow ???$

We guess that

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0.$$

We check that this is correct by substituting back into the balance equation for state i, as follows:

$$\left(\frac{\lambda}{\mu}\right)^i \pi_0(\lambda+\mu) = \left(\frac{\lambda}{\mu}\right)^{i-1} \pi_0\lambda + \left(\frac{\lambda}{\mu}\right)^{i+1} \pi_0\mu$$

$$\frac{\lambda^{i+1}}{\mu^i} + \frac{\lambda^i}{\mu^{i-1}} \stackrel{\surd}{=} \frac{\lambda^i}{\mu^{i-1}} + \frac{\lambda^{i+1}}{\mu^i}$$

Next we determine π_0 such that the equation $\sum_{i=0}^{\infty} \pi_i = 1$ is satisfied:

$$\sum_{i=0}^{\infty} \pi_i = 1$$

$$\Rightarrow \sum_{i=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^i \pi_0 = 1$$

$$\Rightarrow \pi_0 \left(\frac{1}{1-\frac{\lambda}{\mu}}\right) = 1$$

$$\Rightarrow \pi_0 = 1 - \frac{\lambda}{\mu}$$

Therefore, substituting back into the equation for π_i we obtain

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \left(1 - \frac{\lambda}{\mu}\right) = \rho^i(1-\rho)$$

$$\pi_0 = 1 - \frac{\lambda}{\mu} = 1 - \rho$$

where $\rho = \lambda/\mu$ is the server utilization. It should make sense that π_0, the probability that the system is idle, equals $1 - \rho$.

Observe that the condition $\rho < 1$ must be met if the system is to be stable in the sense that the number of customers in the system does not grow without bound. For this condition to be true, we must have $\lambda < \mu$.

The mean number of customers in the system can be derived by conditioning on the state:

Building a model

~~Step 3: gnarly math~~

Here are the full balance equations:

State	Balance equation	Simplified equation
$0:$	$\pi_0\lambda = \pi_1\mu$	$\Rightarrow \pi_1 = \frac{\lambda}{\mu}\pi_0$
$1:$	$\pi_1(\lambda+\mu) = \pi_0\lambda + \pi_2\mu$	$\Rightarrow \pi_2 = \left(\frac{\lambda}{\mu}\right)^2 \pi_0$
$2:$	$\pi_2(\lambda+\mu) = \pi_1\lambda + \pi_3\mu$	$\Rightarrow \pi_3 = \left(\frac{\lambda}{\mu}\right)^3 \pi_0$
$i:$	$\pi_i(\lambda+\mu) = \pi_{i-1}\lambda + \pi_{i+1}\mu$	$\Rightarrow$???

We guess that

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0.$$

We check that this is correct by substituting back into the balance equation for state i, as follows:

$$\left(\frac{\lambda}{\mu}\right)^i \pi_0(\lambda+\mu) = \left(\frac{\lambda}{\mu}\right)^{i-1} \pi_0\lambda + \left(\frac{\lambda}{\mu}\right)^{i+1} \pi_0\mu$$

$$\frac{\lambda^{i+1}}{\mu^i} + \frac{\lambda^i}{\mu^{i-1}} \stackrel{\surd}{=} \frac{\lambda^i}{\mu^{i-1}} + \frac{\lambda^{i+1}}{\mu^i}$$

Next we determine π_0 such that the equation $\sum_{i=0}^{\infty} \pi_i = 1$ is satisfied:

$$\sum_{i=0}^{\infty} \pi_i = 1$$

$$\Rightarrow \sum_{i=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^i \pi_0 = 1$$

$$\Rightarrow \pi_0 \left(\frac{1}{1-\frac{\lambda}{\mu}}\right) = 1$$

$$\Rightarrow \pi_0 = 1 - \frac{\lambda}{\mu}$$

Therefore, substituting back into the equation for π_i we obtain

$$\pi_i = \left(\frac{\lambda}{\mu}\right)^i \left(1 - \frac{\lambda}{\mu}\right) = \rho^i(1-\rho)$$

$$\pi_0 = 1 - \frac{\lambda}{\mu} = 1 - \rho$$

where $\rho = \lambda/\mu$ is the server utilization. It should make sense that π_0, the probability that the system is idle, equals $1 - \rho$.

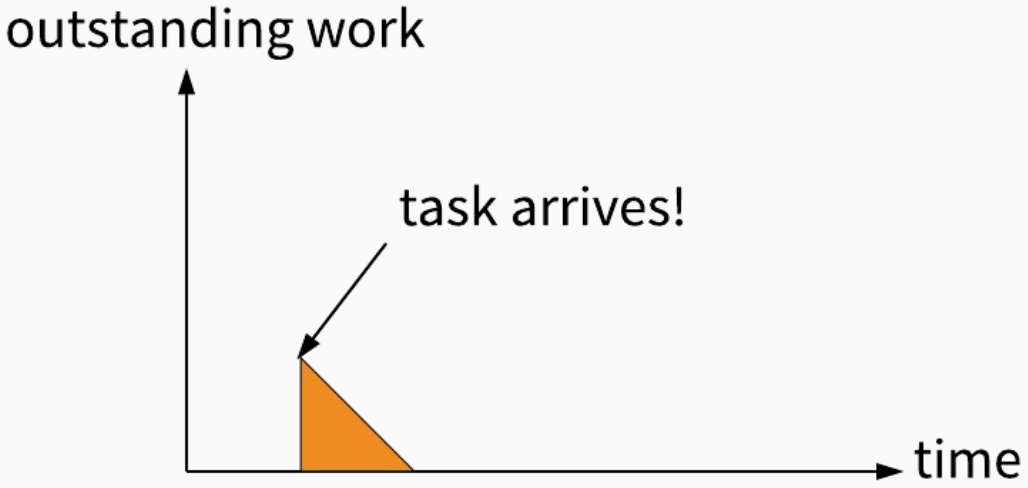
Observe that the condition $\rho < 1$ must be met if the system is to be stable in the sense that the number of customers in the system does not grow without bound. For this condition to be true, we must have $\lambda < \mu$.

The mean number of customers in the system can be derived by conditioning on the state:

Building a model

Step 3: ~~gnarly math~~ draw a picture of the system over time!

At any given time, how much unfinished work is at the server?

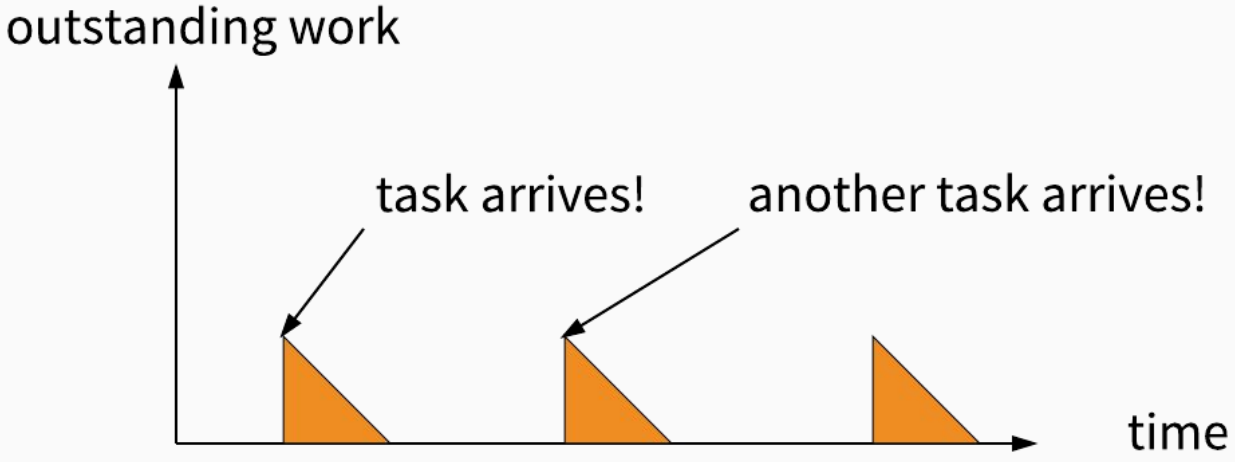


Building a model

Step 3: ~~gnarly math~~ draw a picture of the system over time!

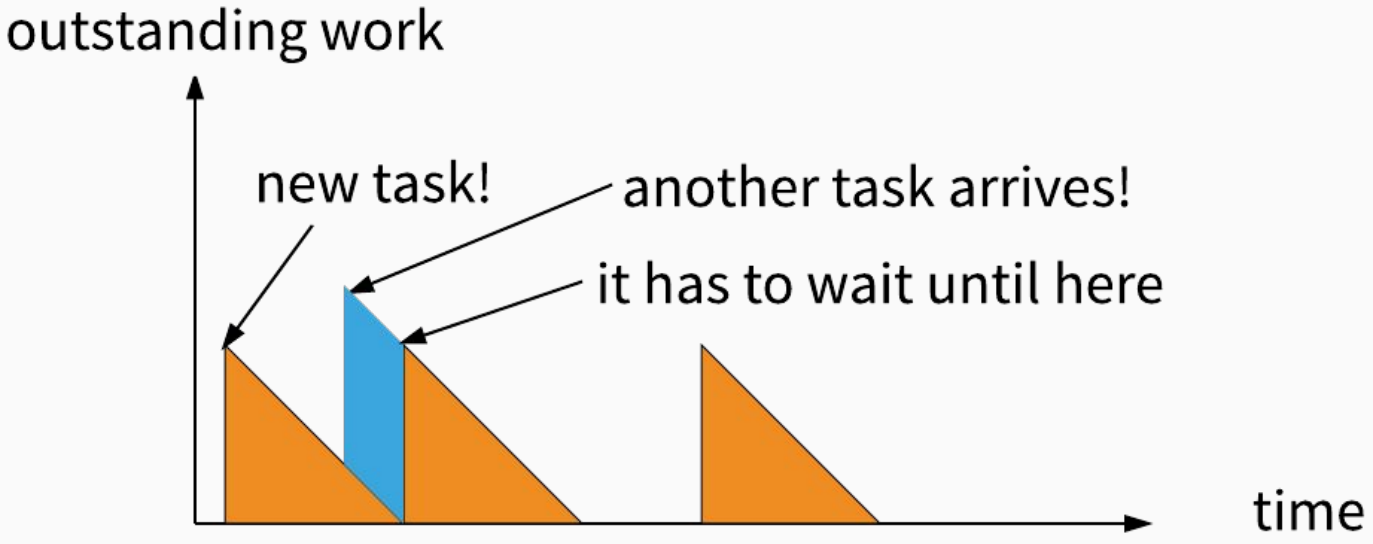
At any given time, how much unfinished work is at the server?

If throughput is low, tasks almost never have to queue: they can be served immediately.



Building a model

But as throughput increases, tasks may have to wait!



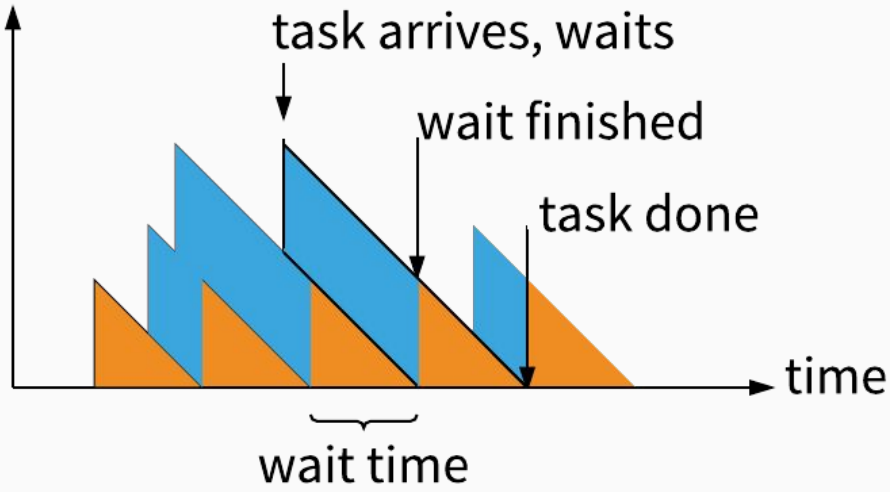
Building a model

But as throughput increases, tasks may have to wait!

Remember, we care about **average wait time**.

Two ways to find average wait time in this graph:

1. Average **width** of blue parallelograms.



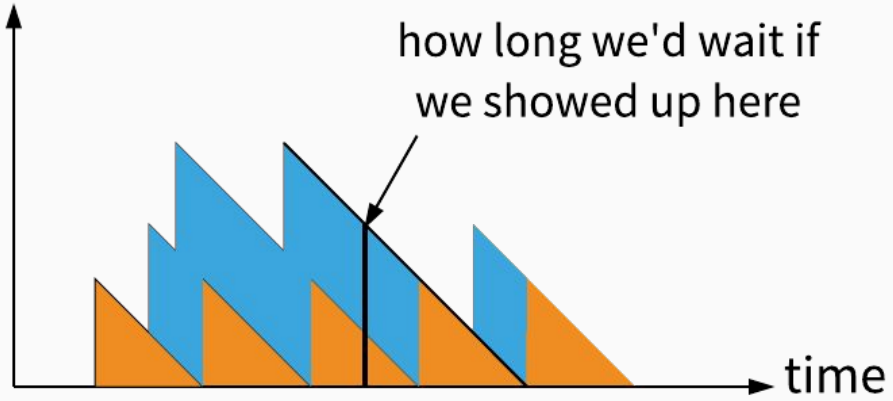
Building a model

But as throughput increases, tasks may have to wait!

Remember, we care about **average wait time**.

Two ways to find average wait time in this graph:

1. Average **width** of blue parallelograms.
2. Average **height** of graph.



Building a model

But as throughput increases, tasks may have to wait!

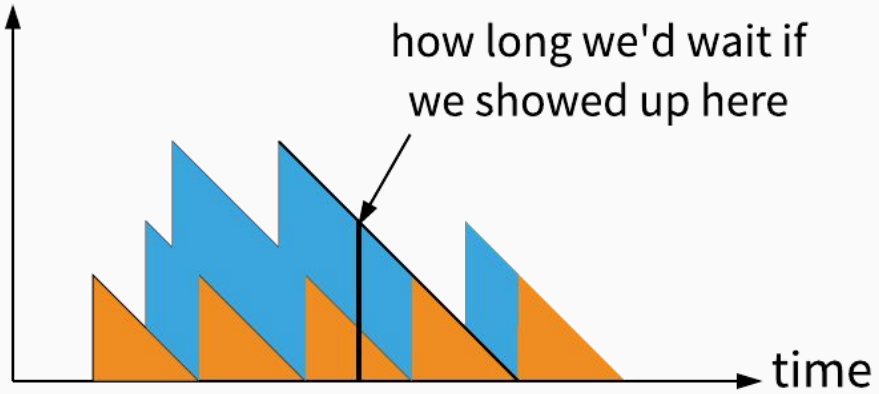
Remember, we care about **average wait time**.

Two ways to find average wait time in this graph:

1. Average **width** of blue parallelograms.
2. Average **height** of graph.

Idea: relate them using **area** under graph,

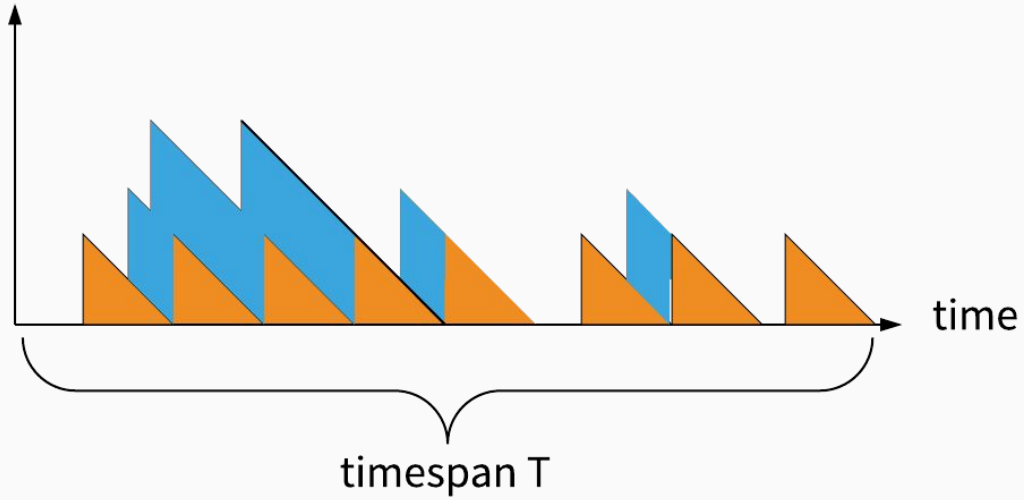
then solve for wait time!



Building a model

Over a long time interval T:

```
(area under graph) = (width) * (avg height of graph)
                   = T * (avg wait time)
                   = T * W
```

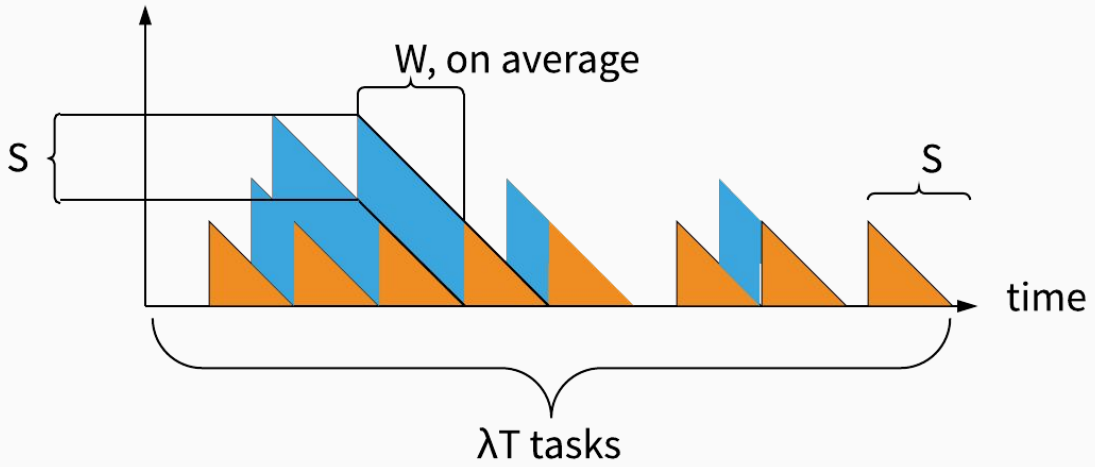


Building a model

For **each** task, there's:

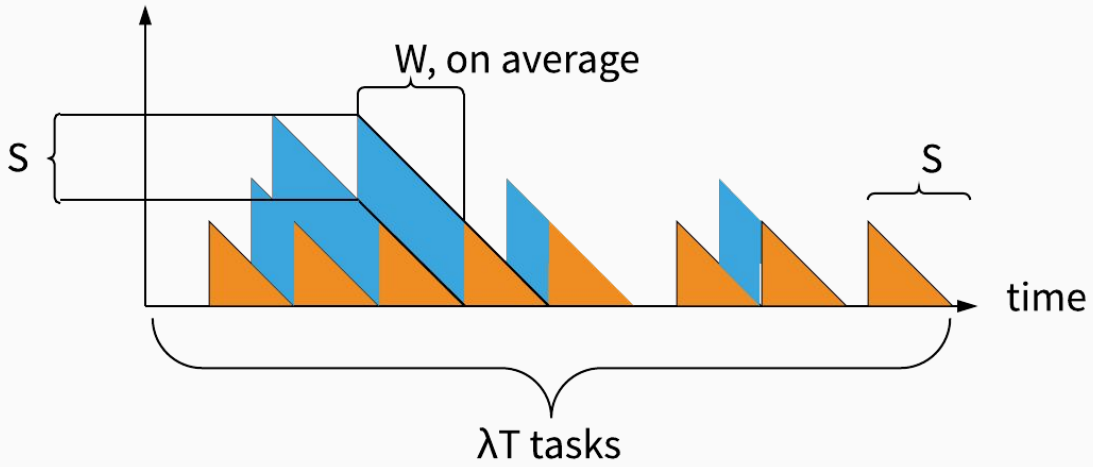
- one triangle
- one parallelogram (might have width 0).

```
(area under graph)
  = (number of tasks) * [(triangle area) + (avg parallelogram area)]
```



Building a model

```
(area under graph)
  = (number of tasks) * [(triangle area) + (avg parallelogram area)]
```

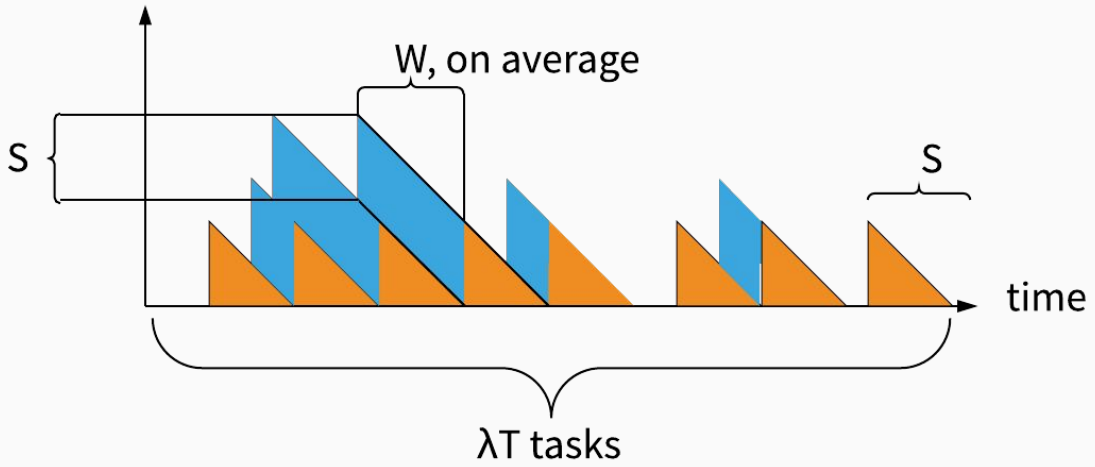


Building a model

(area under graph)

= (number of tasks) * [(triangle area) + (avg parallelogram area)]

= (number of tasks) * [S^2 / 2 + S * W]



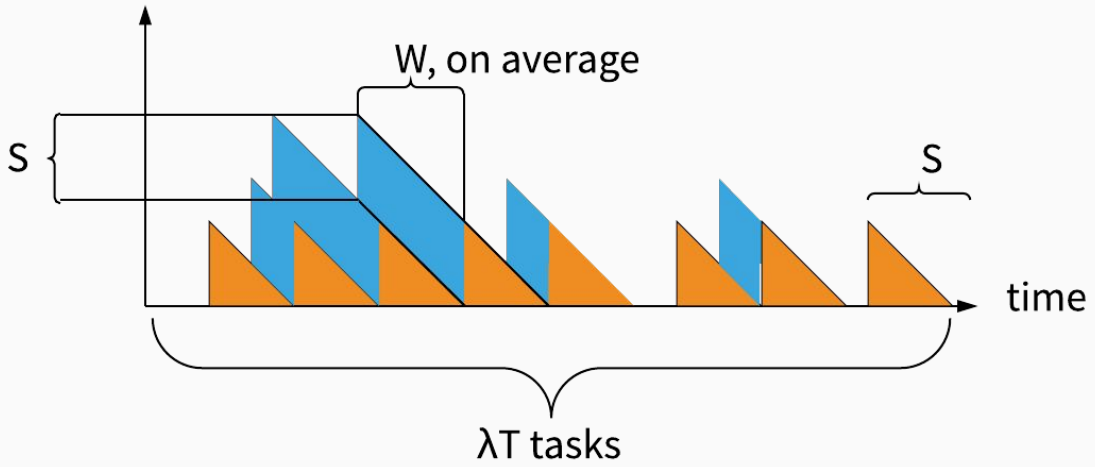
Building a model

(area under graph)

= (number of tasks) * [(triangle area) + (avg parallelogram area)]

= (number of tasks) * [S^2 / 2 + S * W]

= (arrival rate * timespan) * [S^2 / 2 + S * W]



Building a model

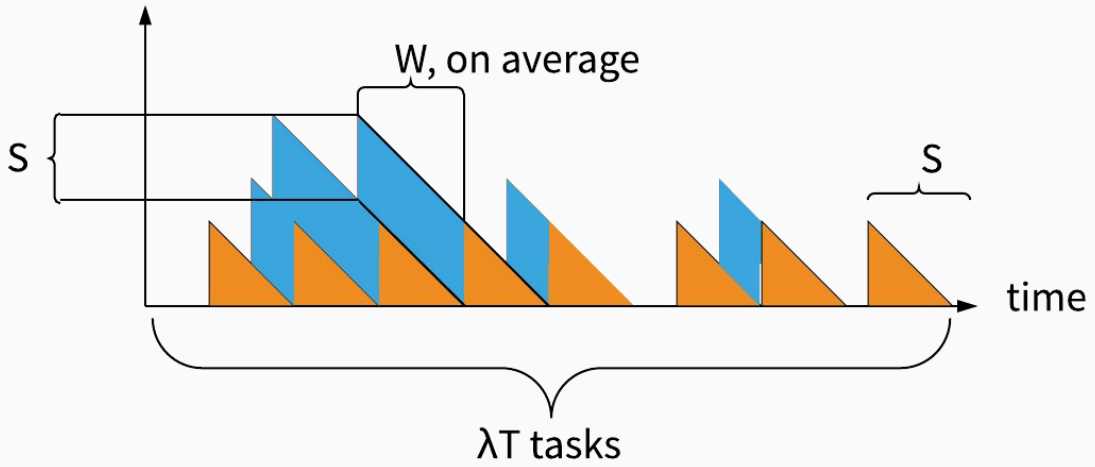
(area under graph)

= (number of tasks) * [(triangle area) + (avg parallelogram area)]

= (number of tasks) * [S^2 / 2 + S * W]

= (arrival rate * timespan) * [S^2 / 2 + S * W]

= λT * (S^2 / 2 + S * W)



Building a model

(area under graph)

= (number of tasks) * [(triangle area) + (avg parallelogram area)]

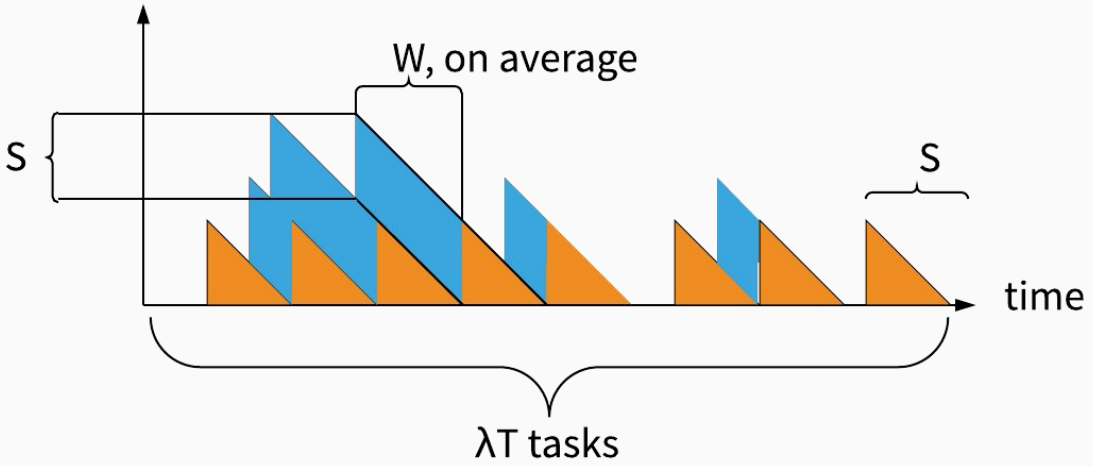
= (number of tasks) * [S^2 / 2 + S * W]

= (arrival rate * timespan) * [S^2 / 2 + S * W]

= λT * (S^2 / 2 + S * W)

Before, we had:

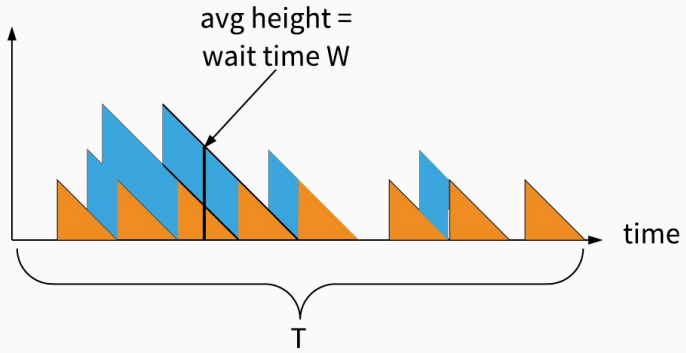
(area under graph) = T * W



Building a model

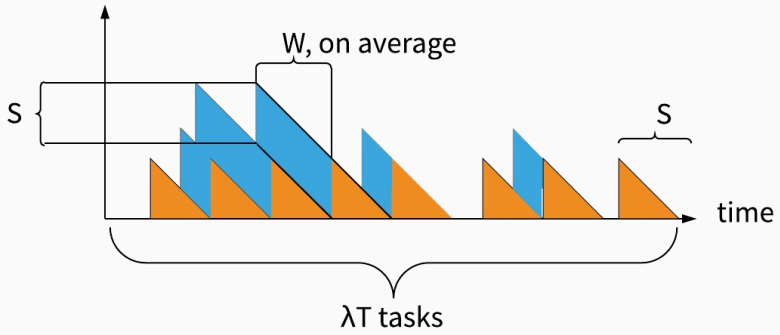
So:

```
   (area under graph)
=   T * W
=   λT * (S * W + S² / 2)
```



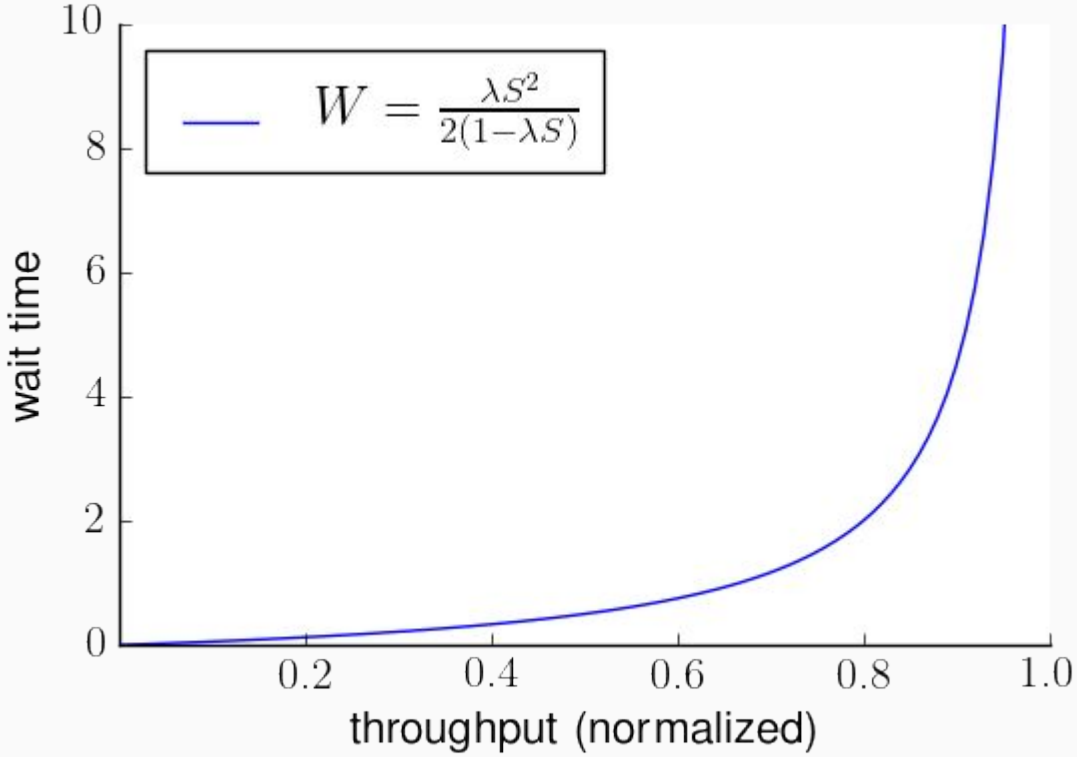
Solving for W:

$$W = \frac{\lambda S^2}{2(1 - \lambda S)}$$



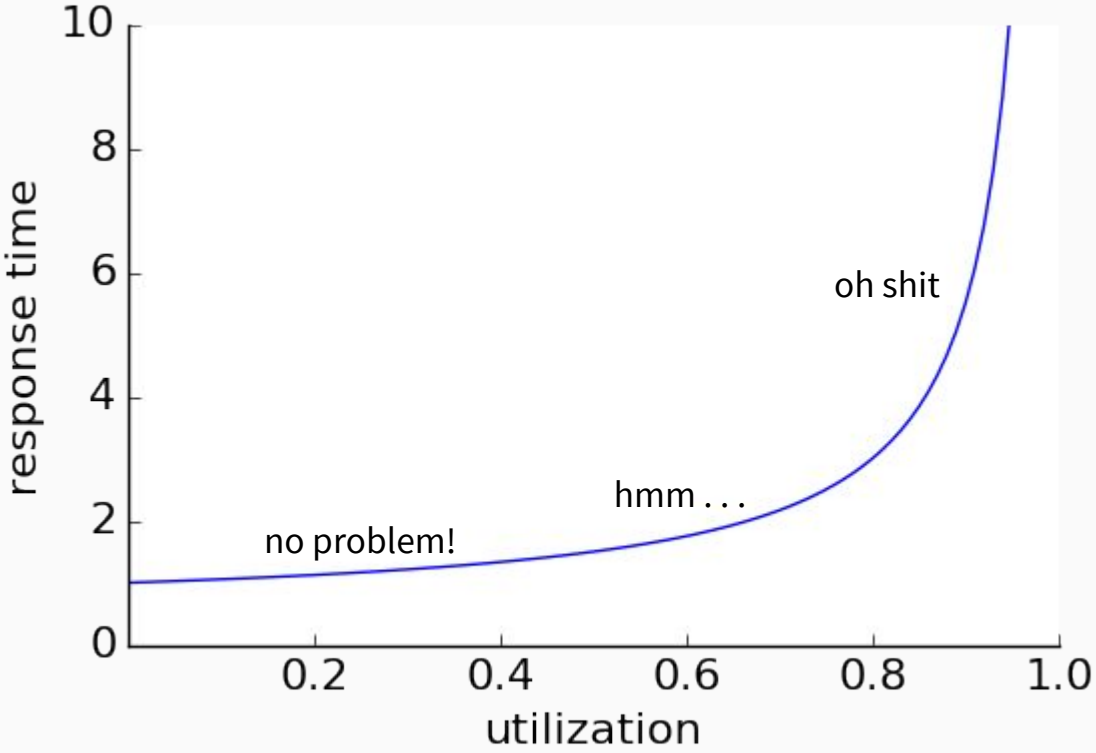
Building a model

As the server becomes saturated, wait time grows without bound!



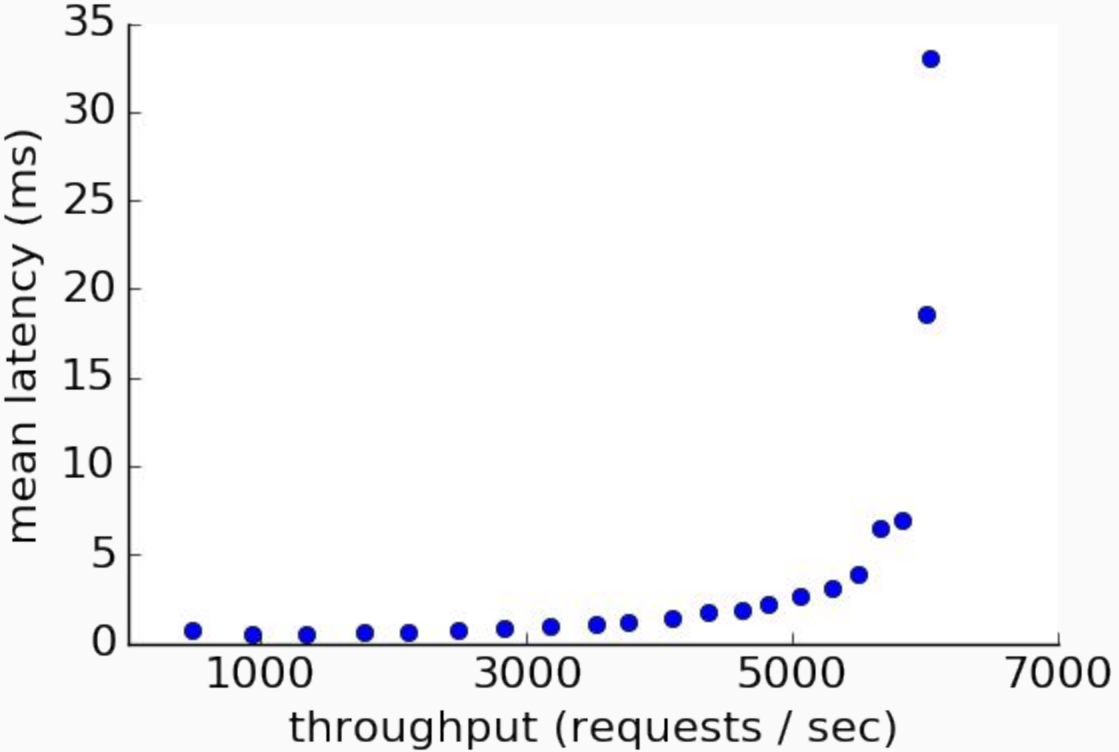
The model as general heuristic

As operators, we can roughly identify three utilization regimes:



Returning to our data

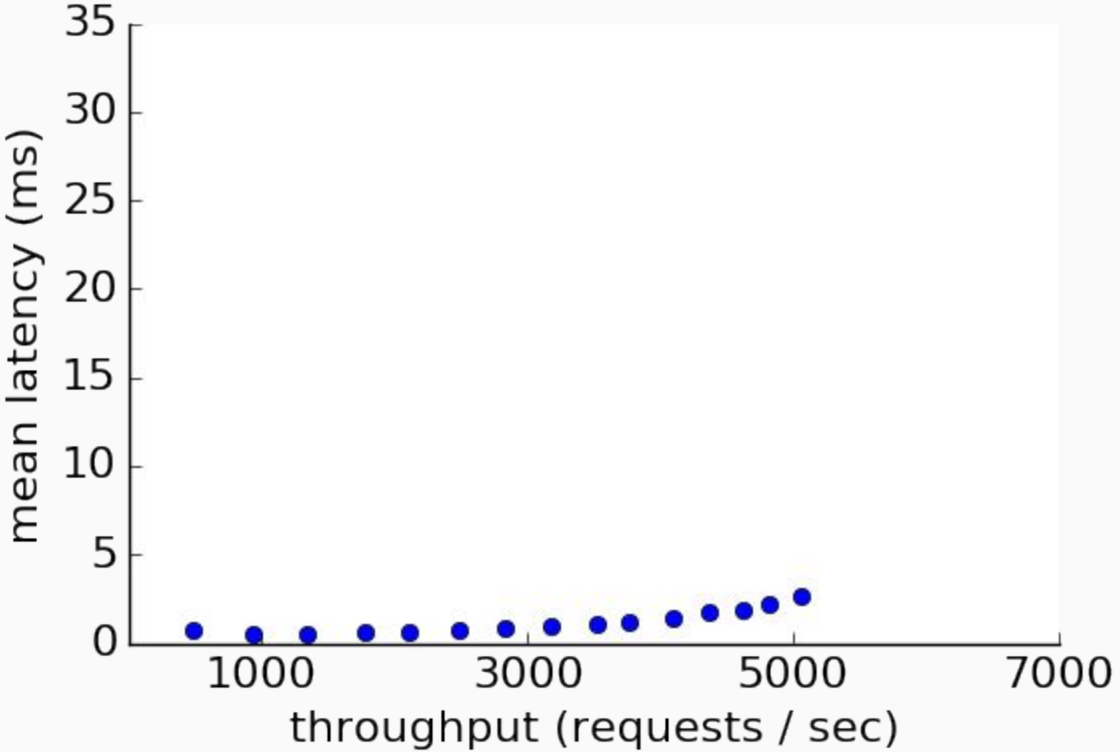
Does this model apply in practice?



Returning to our data

Does this model apply in practice?

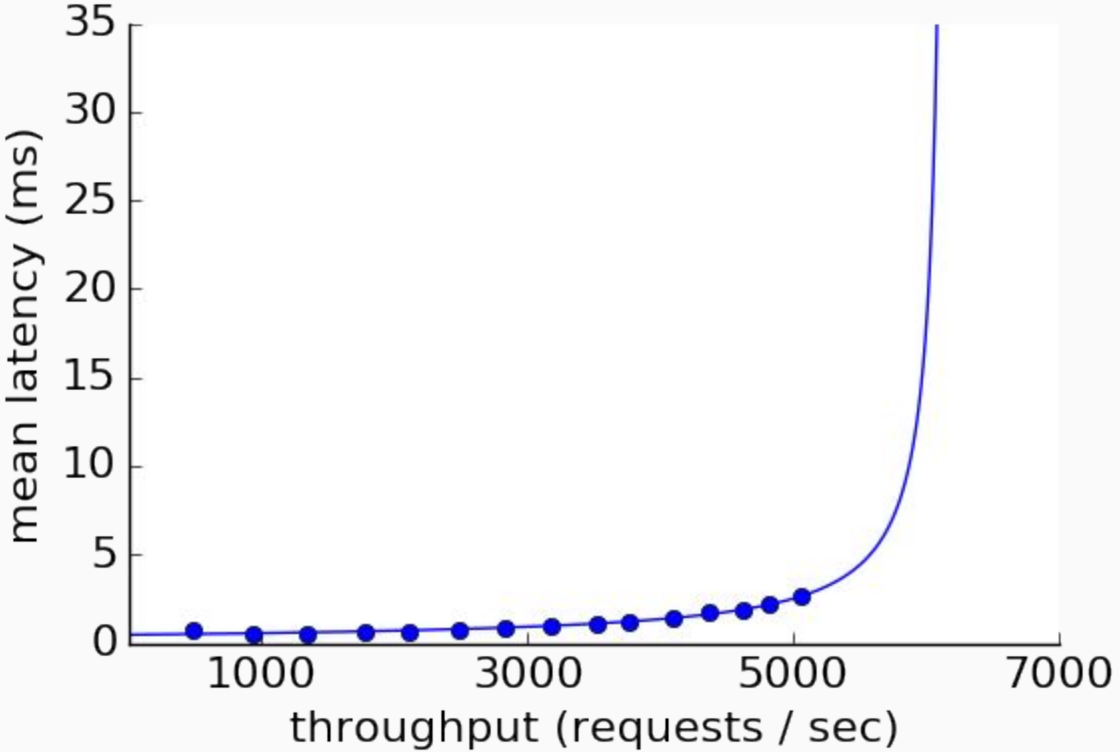
1. Choose subset of data



Returning to our data

Does this model apply in practice?

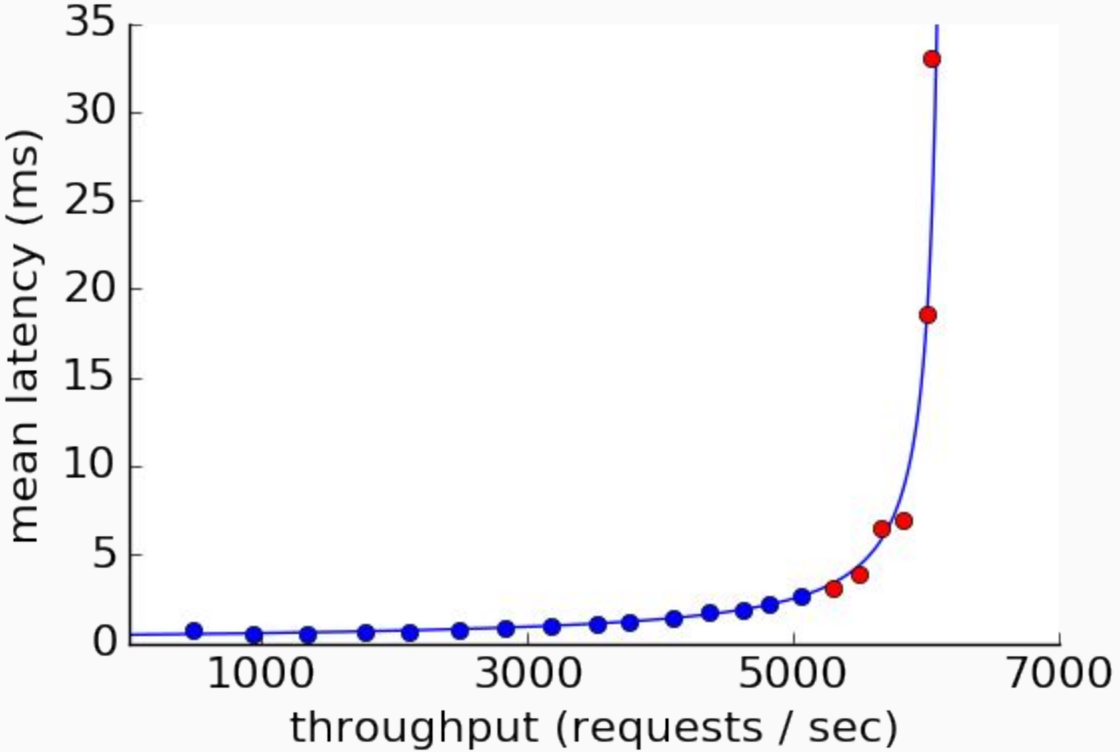
1. Choose subset of data
2. Fit model (R, Numpy, …)



Returning to our data

Does this model apply in practice?

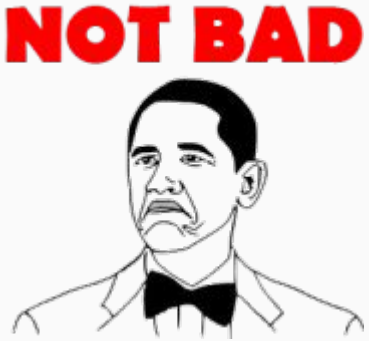
1. Choose subset of data
2. Fit model (R, Numpy, …)
3. Compare

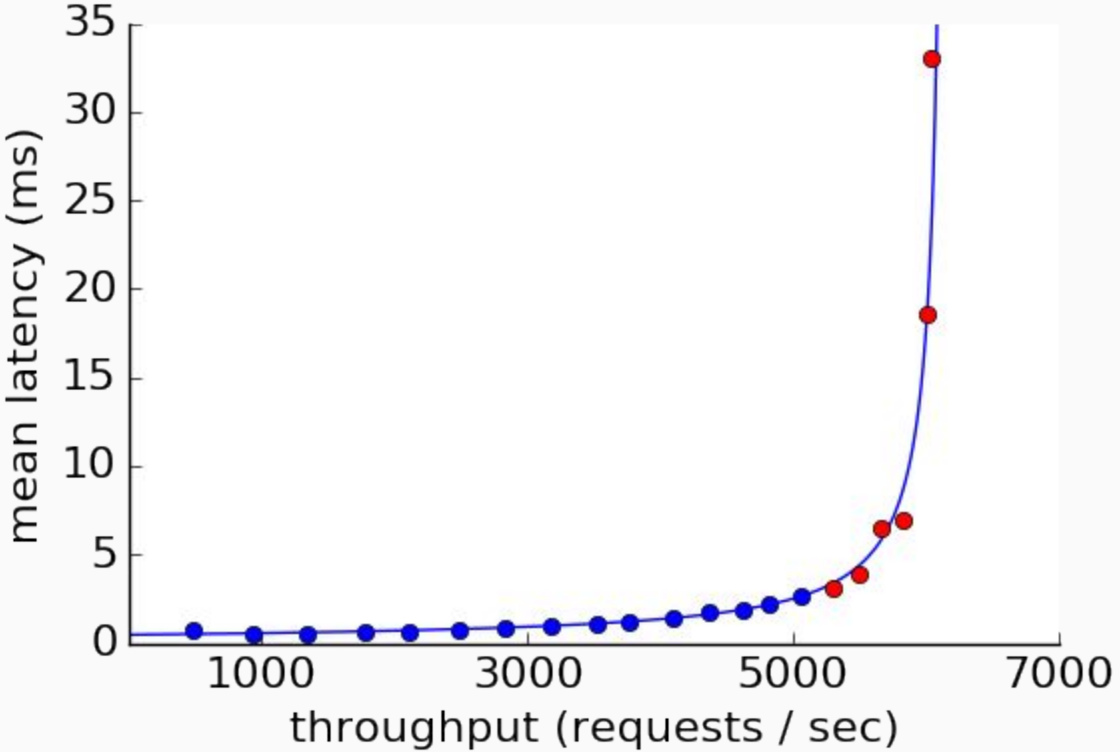


Returning to our data

Does this model apply in practice?

1. Choose subset of data
2. Fit model (R, Numpy, …)
3. Compare





Lessons from the single-server queueing model

1. In this type of system, improving service time helps a lot!

Lessons from the single-server queueing model

1. In this type of system, improving service time helps a lot!

Thought experiment:

1. cut the service time S in half
2. Double the throughput λ

$$W = \frac{\lambda S^2}{2(1 - \lambda S)}$$

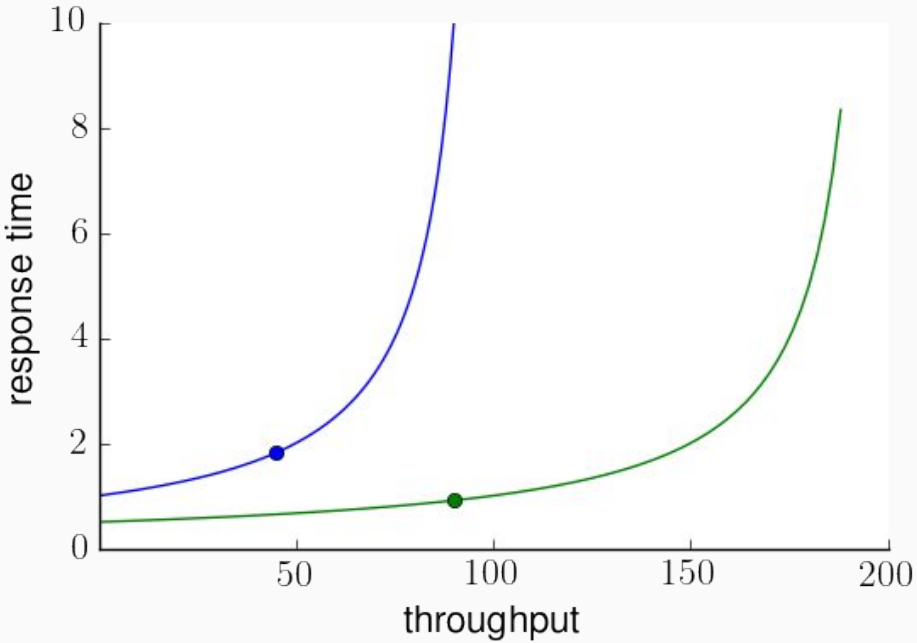
λS^2 ← now twice as small

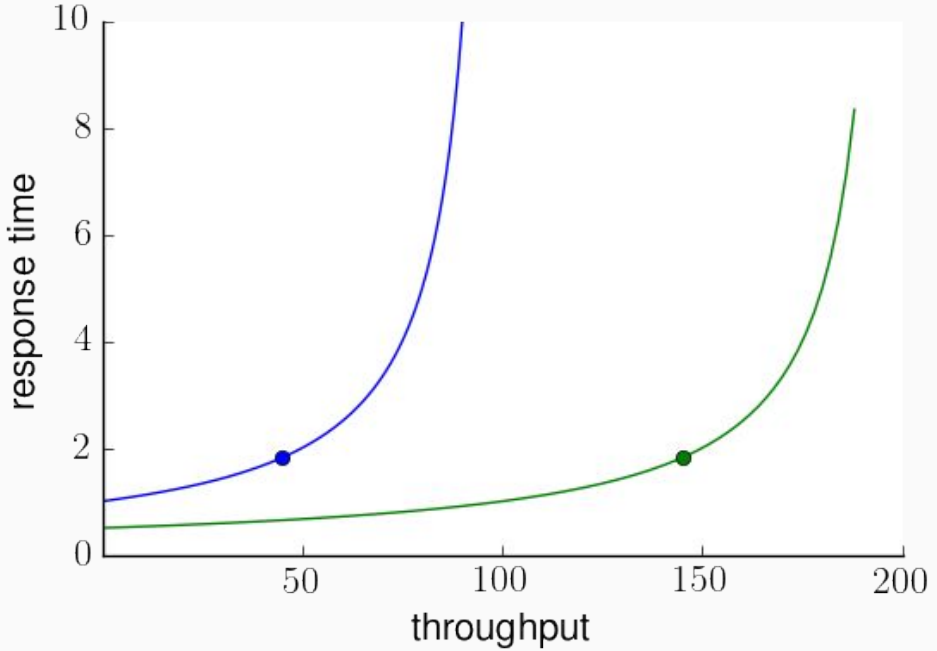
$2(1 - \lambda S)$ ← stays the same

Wait time still improves, even after you double throughput!

Lessons from the single-server queueing model

1. **In this type of system, improving service time helps a lot!**





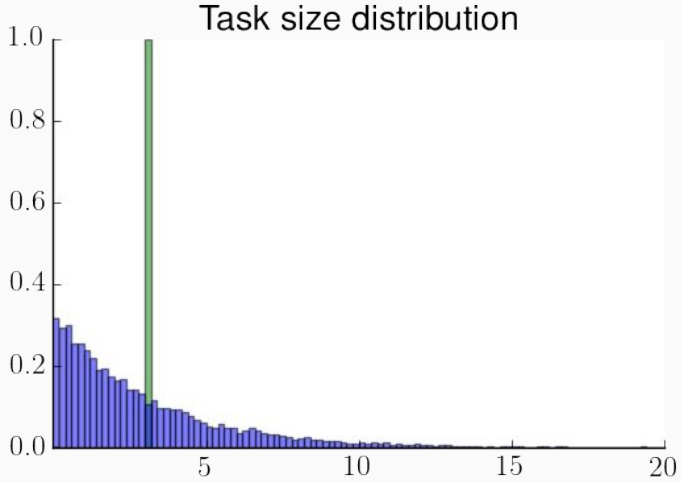
Lessons from the single-server queueing model

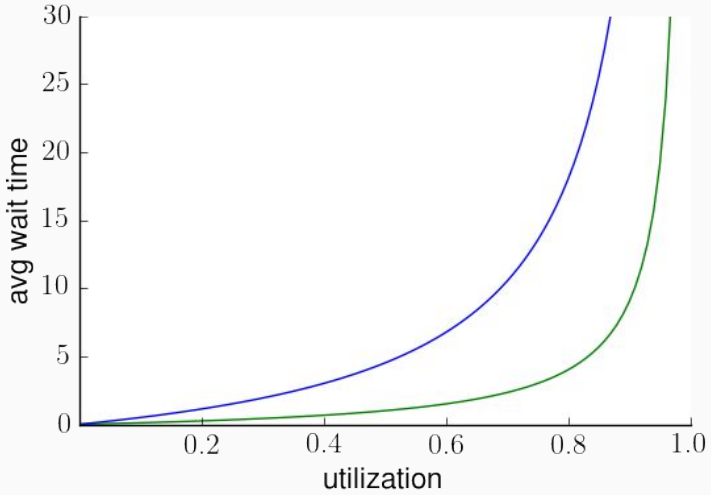
2. Variability is bad!

If we have uniform tasks at perfectly uniform intervals, there's never any queueing.

The slowdown we see is entirely due to variability in arrivals.

If job sizes are variable too, things get even worse.





Lessons from the single-server queueing model

2. Variability is bad!

If we have uniform tasks at perfectly uniform intervals, there's never any queueing.

The slowdown we see is entirely due to variability in arrivals.

If job sizes are variable too, things get even worse.

As system designers, it behooves us to measure and minimize variability:

- batching
- fast preemption or timeouts
- client backpressure
- concurrency control

But wait a minute!

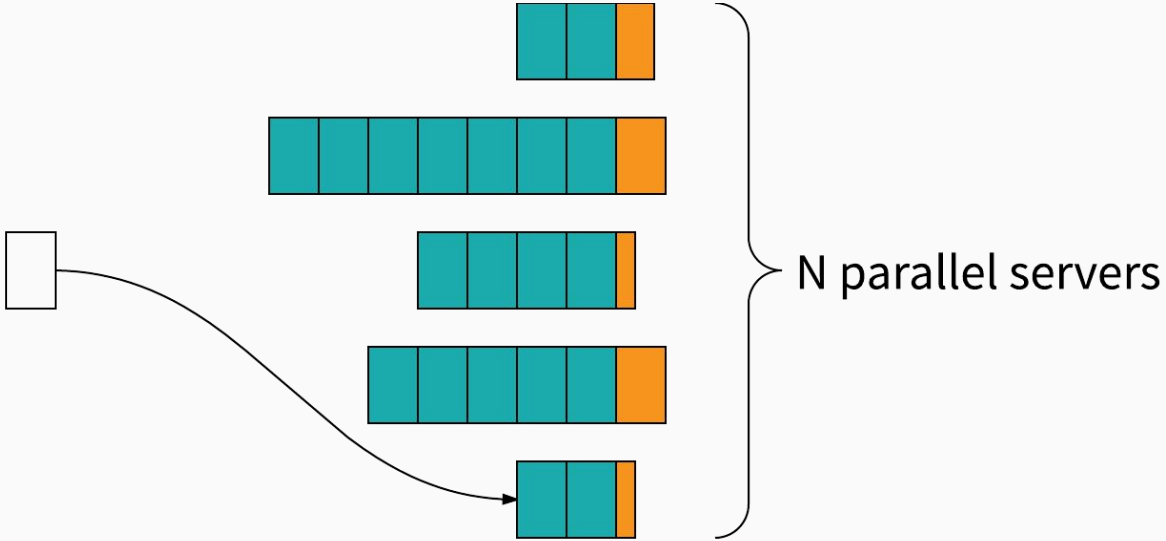
We don’t have one server, we have lots and lots!

What can we say about the performance of a *fleet* of servers?

II. Parallel Systems

Mo servers mo problems

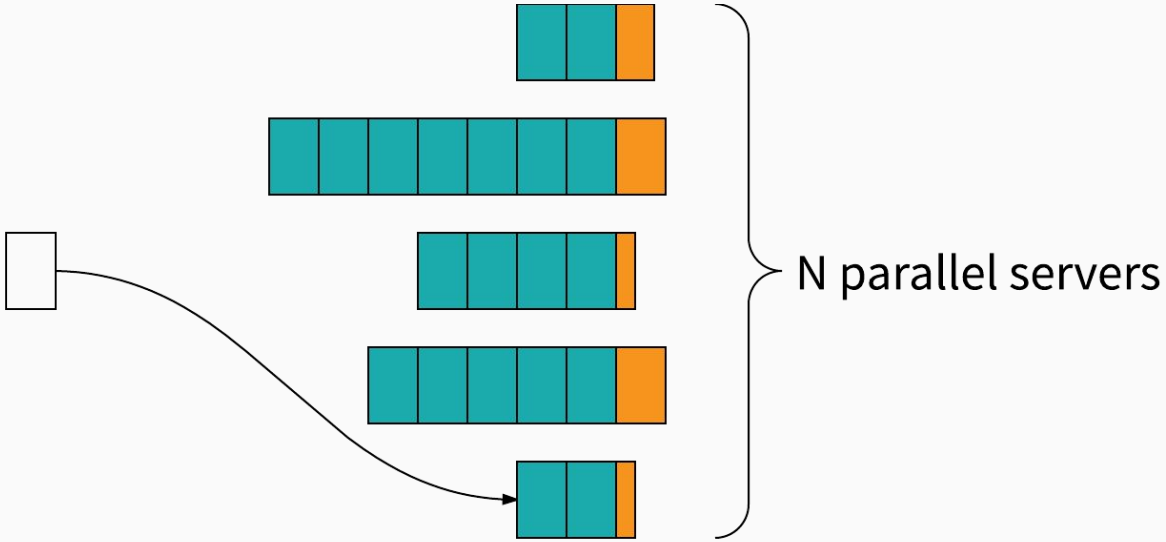
If we know that *one* server can handle *T* requests per second with some latency SLA, do we need *N* servers to handle *N* * *T* requests per second?



Mo servers mo problems

Well, it depends on how we assign incoming tasks!

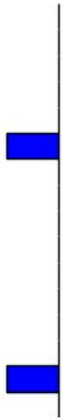
- to the least busy server
- randomly
- round-robin
- some other way

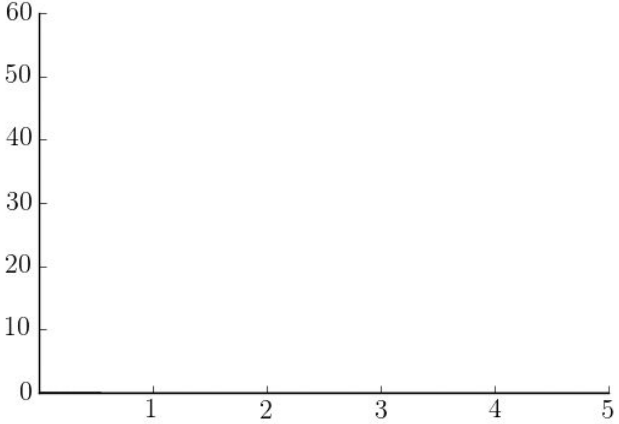


Instantaneous queue lengths

Cumulative latency distribution

random assignment

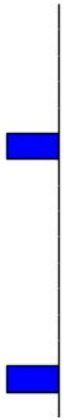


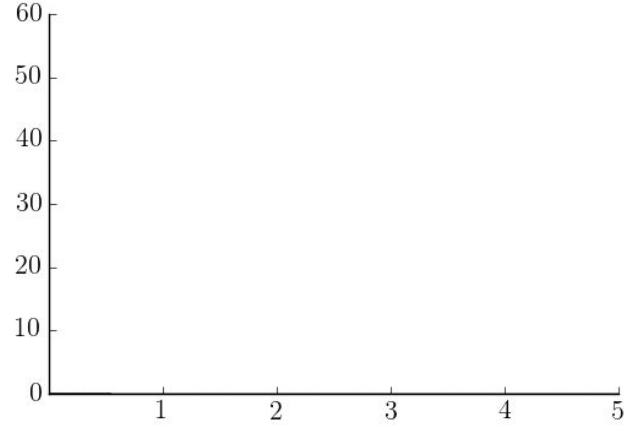


Instantaneous queue lengths

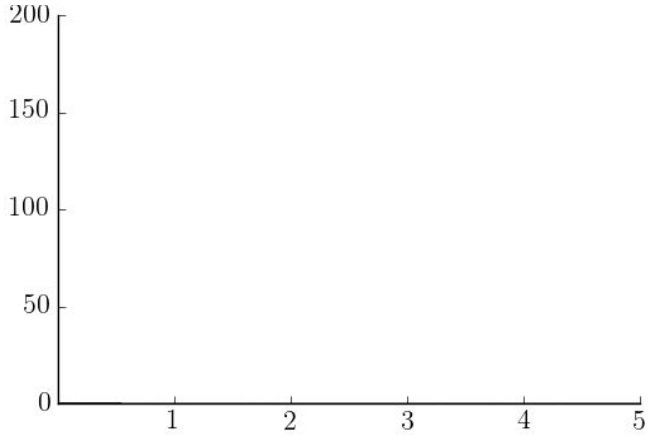
Cumulative latency distribution

random assignment





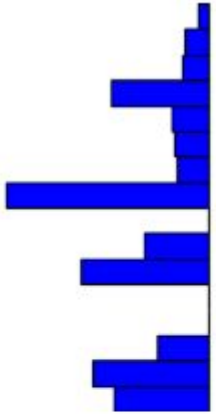
optimal assignment
(always choose the least busy server)

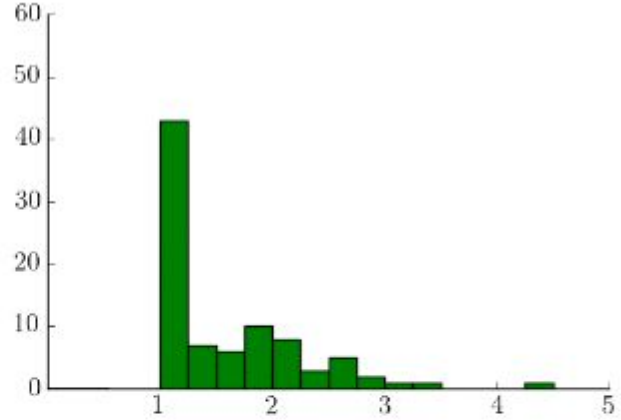


Instantaneous queue lengths

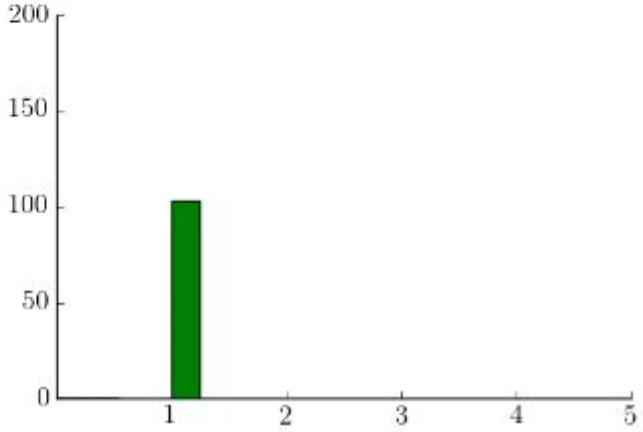
Cumulative latency distribution

random assignment





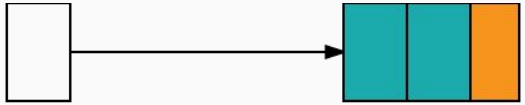
optimal assignment
(always choose the least busy server)



Optimal assignment

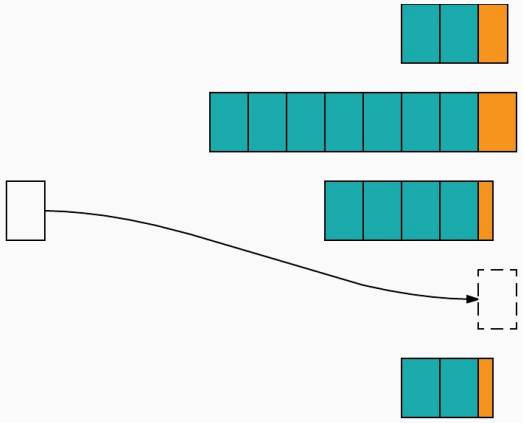
Given 1 server at utilization ρ (say ρ=60%):

$$P(\text{queueing}) = P(\text{server is busy}) = \rho$$



Given N servers at utilization ρ:

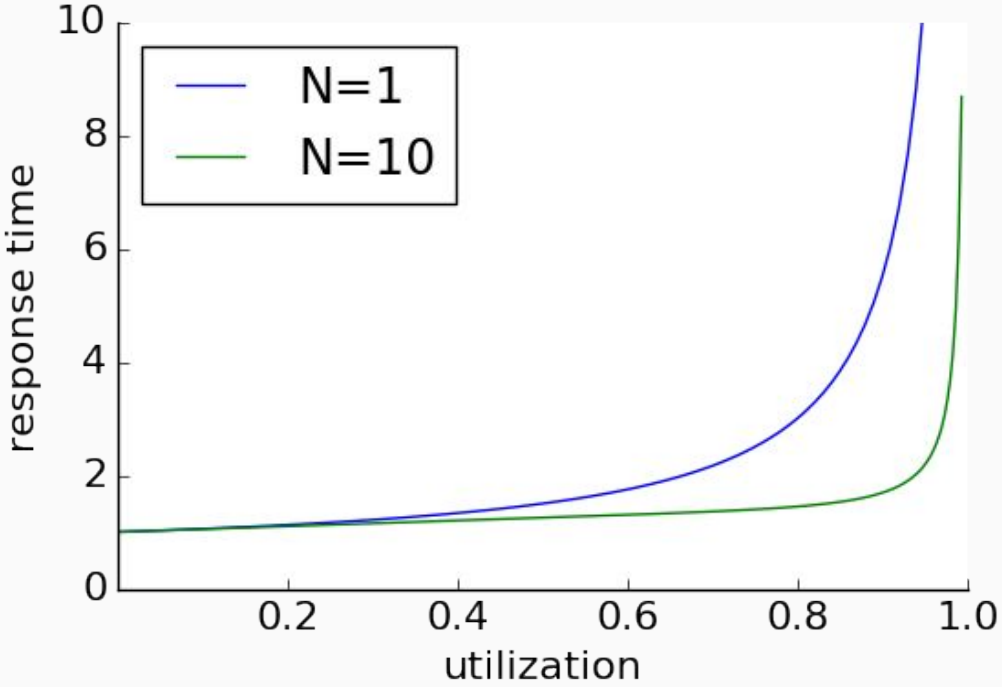
$$P(\text{queueing}) = P(\textit{all}\ \text{servers are busy}) < \rho$$



Optimal assignment

If we have many servers, higher utilization gives us the same queueing probability.

To serve N times more traffic, we won't need N times more servers.



Optimal assignment

There’s just one problem:

We’re assuming optimal assignment of tasks to servers.

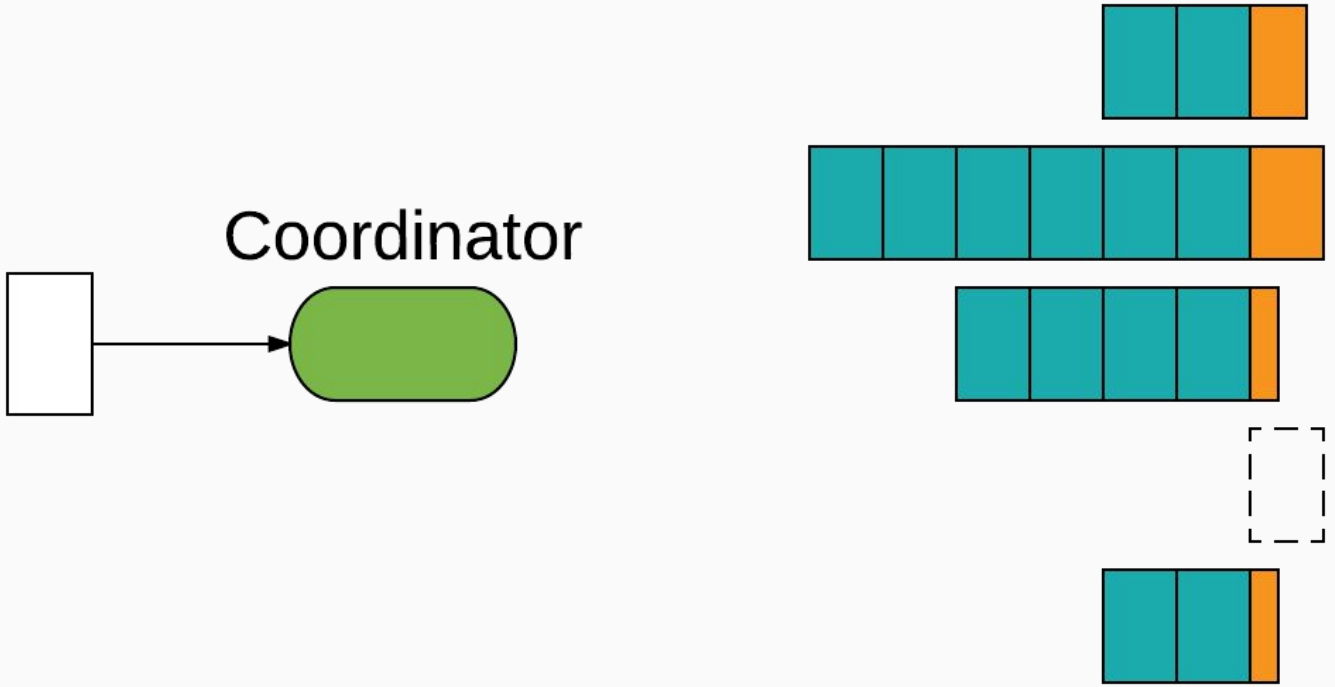
Optimal assignment is a coordination problem.

In real life, coordination is expensive.

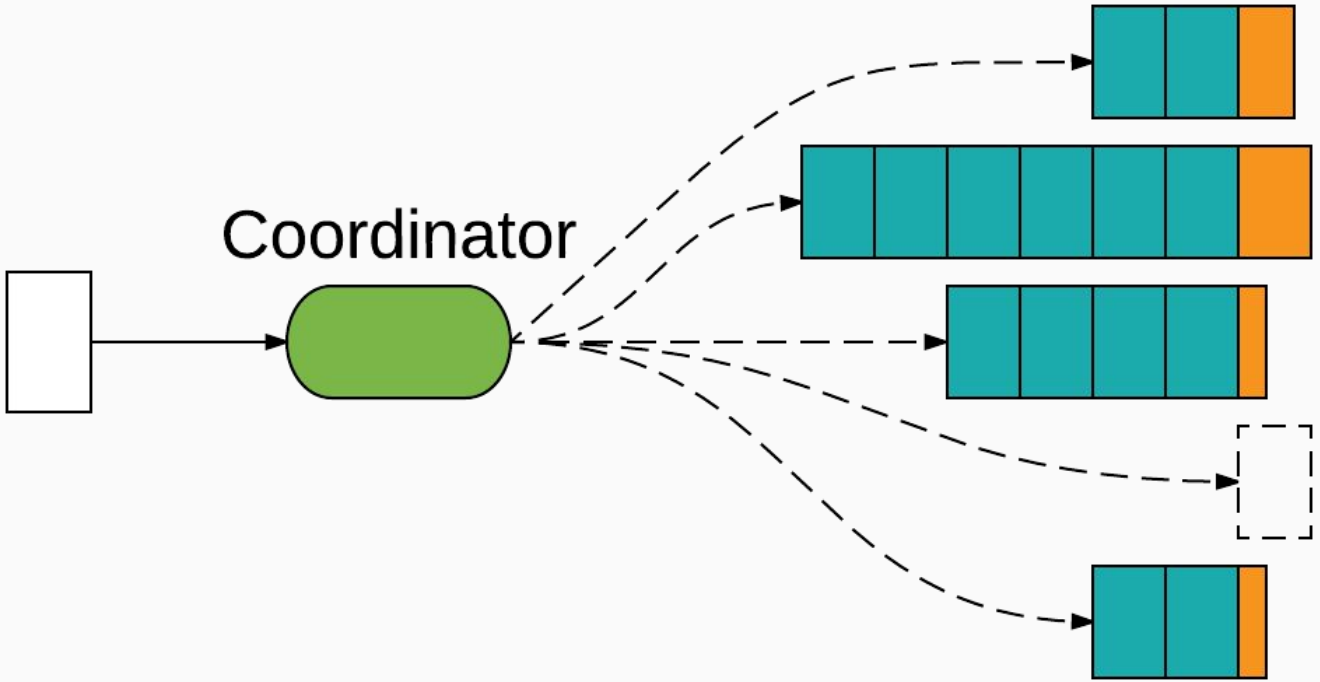
Optimal assignment

We need some coordination mechanism!

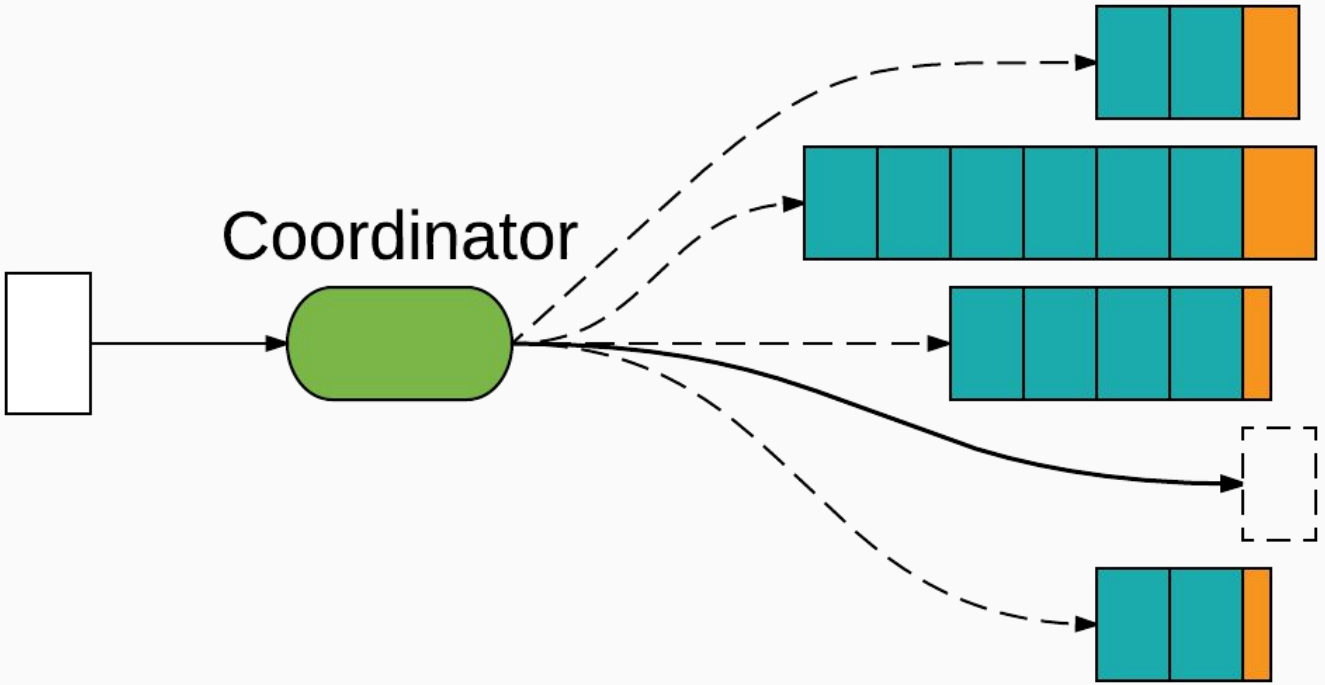
(a load balancer, cluster scheduler, etc.)



Optimal assignment



Optimal assignment



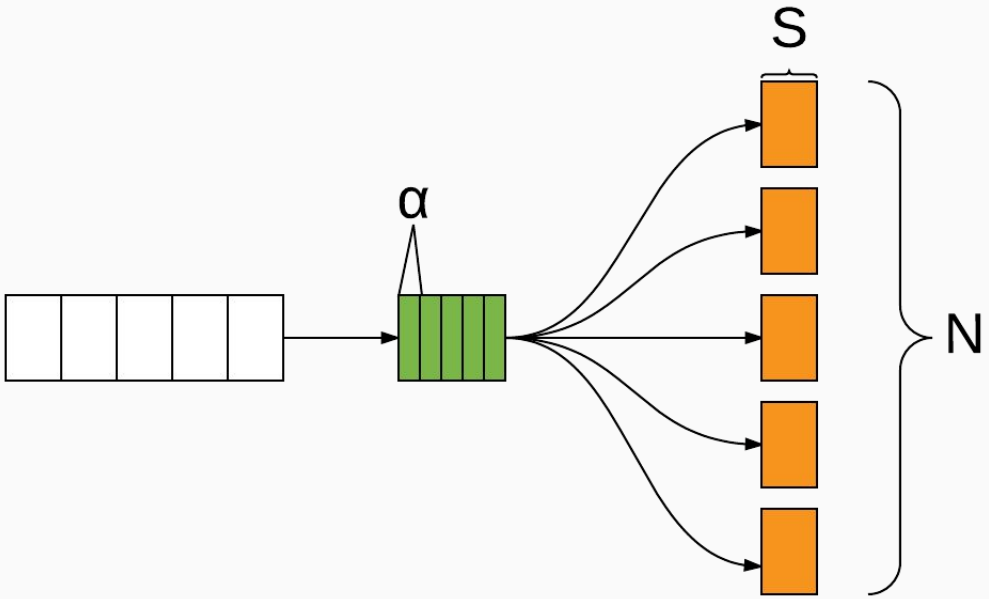
Optimal assignment

If the assignment cost per task is α, then the time to process N tasks in parallel is

$$\alpha N + S$$

And the throughput is

$$N / (\alpha N + S)$$



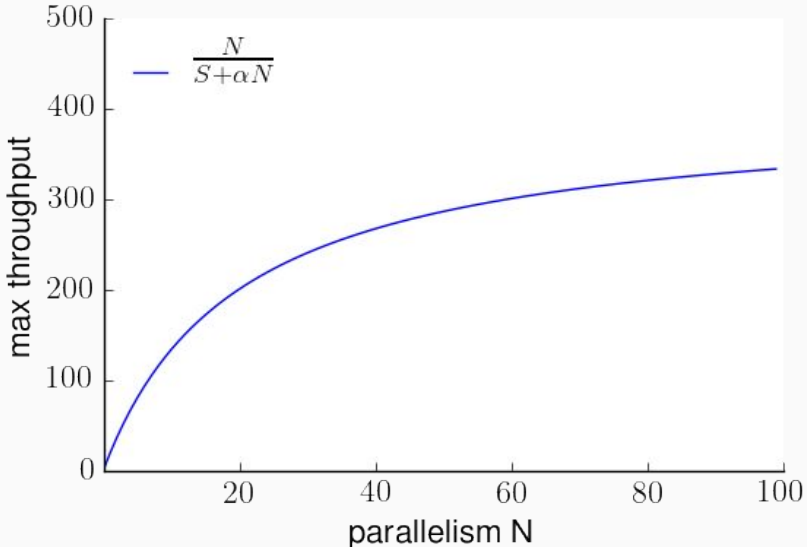
Optimal assignment

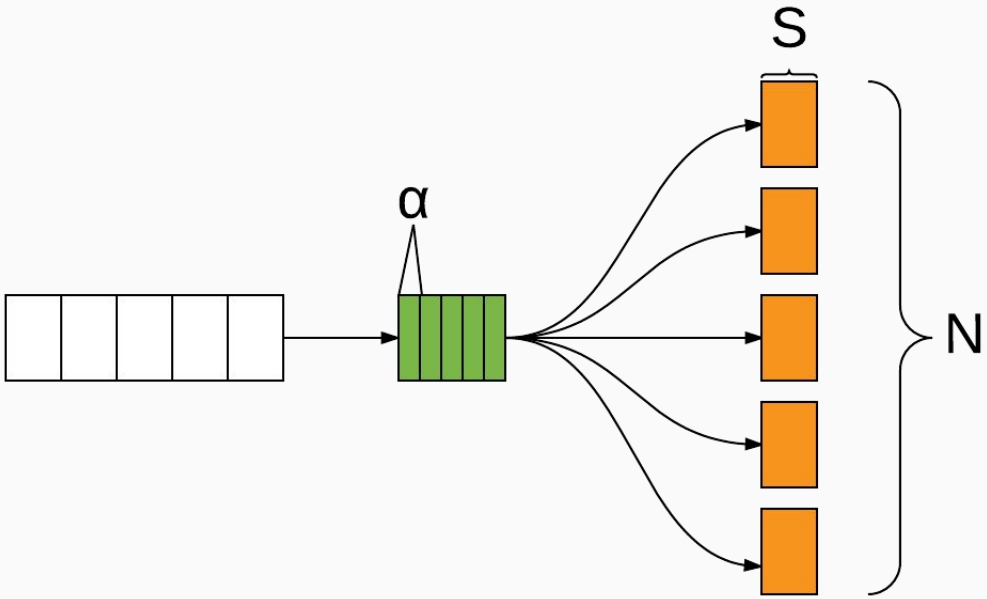
If the assignment cost per task is α, then the time to process N tasks in parallel is

$$\alpha N + S$$

And the throughput is

$$N / (\alpha N + S)$$





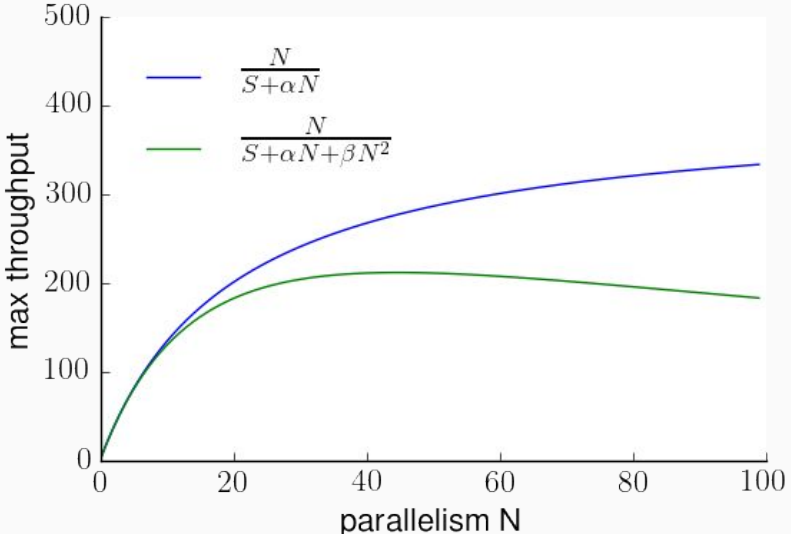
Optimal assignment

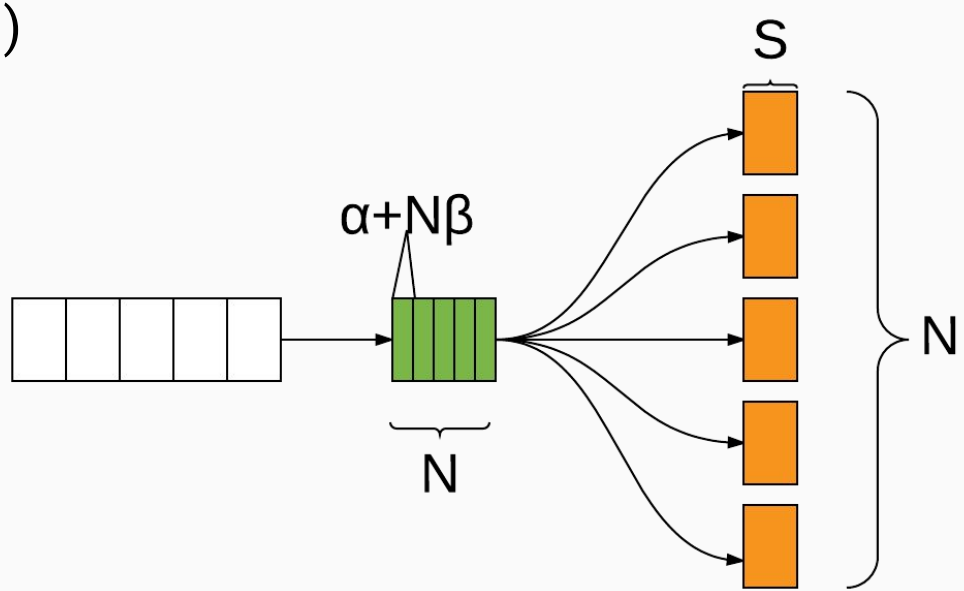
If the assignment cost per task is α, then the throughput is

$$N / (\alpha N + S)$$

If the assignment cost per task depends on N, say Nβ+α, then the throughput is

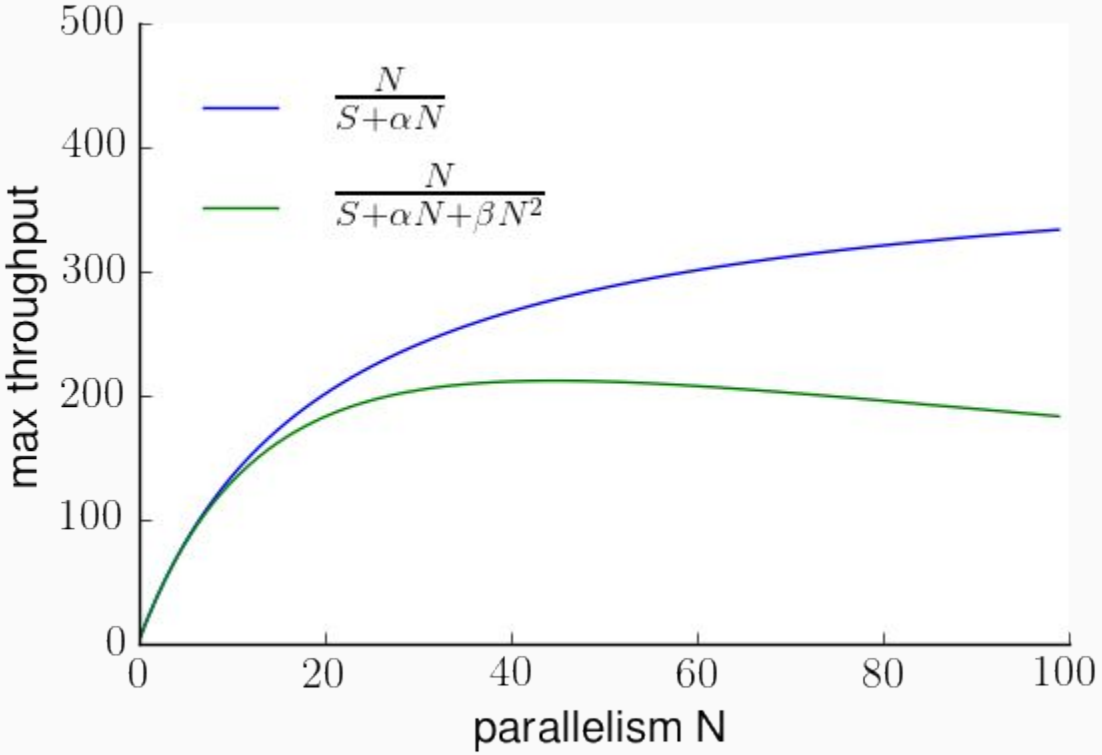
$$N / (\beta N^2 + \alpha N + S)$$





The Universal Scalability Law

This is one example of the Universal Scalability Law in action.



Beating the beta factor

Making scale-invariant design decisions is hard:

- at low parallelism, coordination makes latency more predictable.
- at high parallelism, coordination degrades throughput.

Beating the beta factor

Making scale-invariant design decisions is hard:

- at low parallelism, coordination makes latency more predictable.
- at high parallelism, coordination degrades throughput.

Can we **find strategies** to balance the two?

Beating the beta factor

Idea 1: Approximate optimal assignment

The Power of Two Choices in Randomized Load Balancing

Michael Mitzenmacher, *Member*, *IEEE*

Abstract—We consider the following natural model: Customers arrive as a Poisson stream of rate λn, $\lambda < 1$, at a collection of n servers. Each customer chooses some constant d servers independently and uniformly at random from the n servers and waits for service at the one with the fewest customers. Customers are served according to the first-in first-out (FIFO) protocol and the service time for a customer is exponentially distributed with mean 1. We call this problem the *supermarket model*. We wish to know how the system behaves and in particular we are interested in the effect that the parameter d has on expected time a customer spends in the system in equilibrium. Our approach uses a limiting, deterministic model representing the behavior as $n \to \infty$ to approximate the behavior of finite systems. The analysis of the deterministic model is interesting in its own right. Along with a theoretical justification of this approach, we provide simulations that demonstrate that the method accurately predicts system behavior, even for relatively small systems. Our analysis provides surprising implications: Having $d = 2$ choices leads to exponential improvements in the expected time a customer spends in the system over $d = 1$, whereas having $d = 3$ choices is only a constant factor better than $d = 2$. We discuss the possible implications for system design.

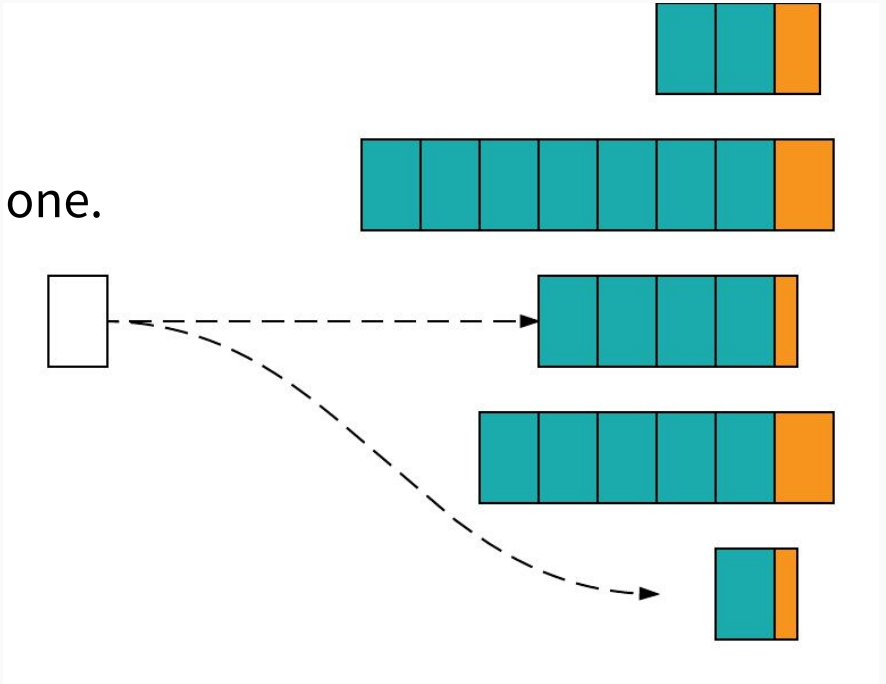
Index Terms—Load balancing, queuing theory, distributed systems, limiting systems, choices.

Beating the beta factor

Randomized approximation

Idea:

- finding best of N servers is expensive
- choosing one randomly is bad
- pick 2 at random and then use the better one.

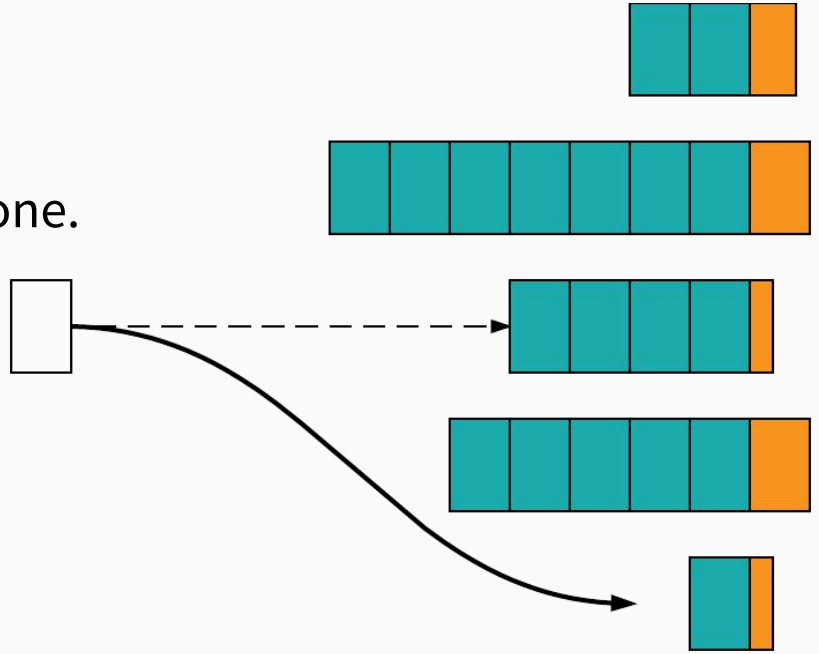


Beating the beta factor

Randomized approximation

Idea:

- finding best of N servers is expensive
- choosing one randomly is bad
- pick 2 at random and then use the better one.



Beating the beta factor

Randomized approximation

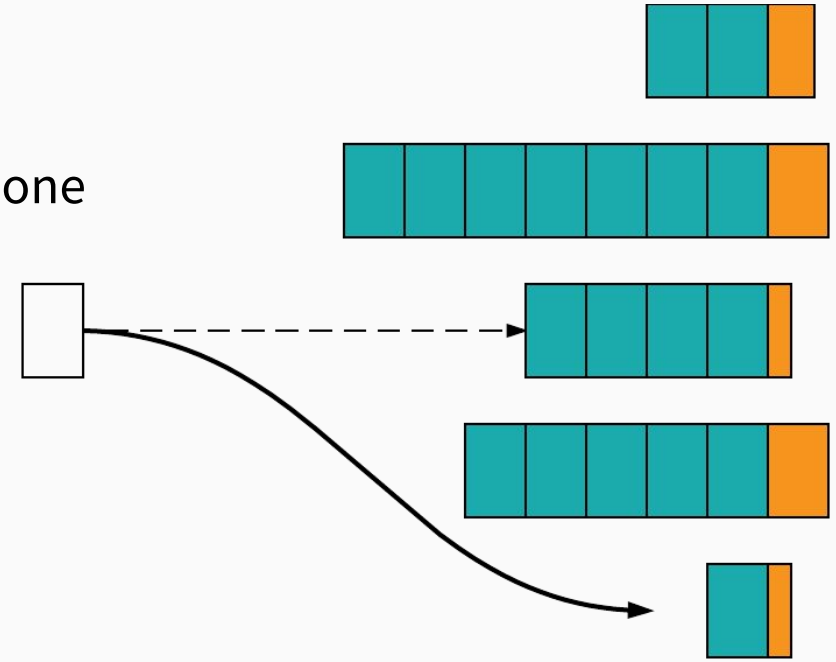
Idea:

- finding best of N servers is expensive
- choosing one randomly is bad
- pick 2 at random and then use the better one

Wins:

- constant overhead for all N
- improves instantaneous max load

  from `O(log N)` to `O(log log N)`

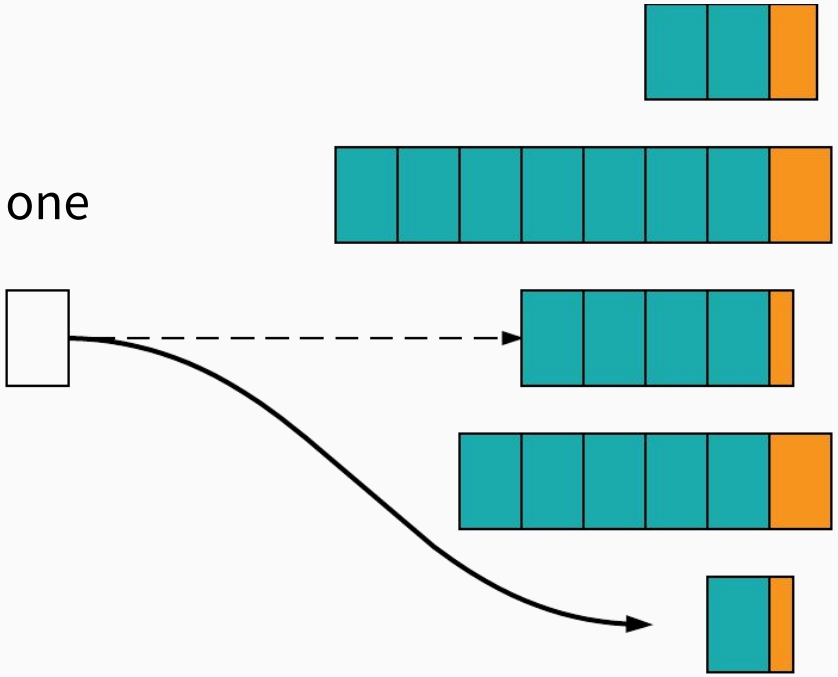


Beating the beta factor

Randomized approximation

Idea:

- finding best of N servers is expensive
- choosing one randomly is bad
- pick 2 at random and then use the better one



Wins:

- constant overhead for all N
- improves instantaneous max load

  from `O(log N)` to `O(log log N)`

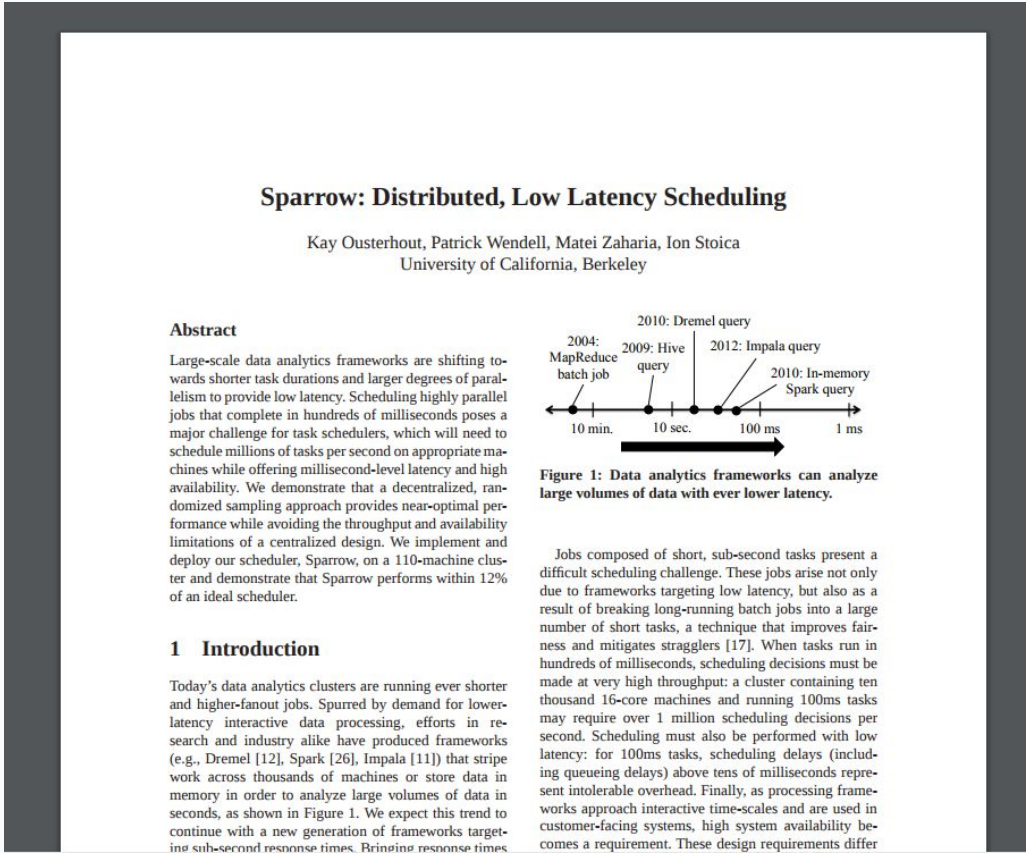
  which is baaaasically `O(1)`

Beating the beta factor

Sparrow:

(and Hashicorp's Nomad)

- distributed, stateless scheduling
- low-latency scheduling for lots of short tasks
- uses two-random-choices (plus optimizations)



Sparrow: Distributed, Low Latency Scheduling

Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica
University of California, Berkeley

Abstract

Large-scale data analytics frameworks are shifting towards shorter task durations and larger degrees of parallelism to provide low latency. Scheduling highly parallel jobs that complete in hundreds of milliseconds poses a major challenge for task schedulers, which will need to schedule millions of tasks per second on appropriate machines while offering millisecond-level latency and high availability. We demonstrate that a decentralized, randomized sampling approach provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. We implement and deploy our scheduler, Sparrow, on a 110-machine cluster and demonstrate that Sparrow performs within 12% of an ideal scheduler.

1 Introduction

Today's data analytics clusters are running ever shorter and higher-fanout jobs. Spurred by demand for lower-latency interactive data processing, efforts in research and industry alike have produced frameworks (e.g., Dremel [12], Spark [26], Impala [11]) that stripe work across thousands of machines or store data in memory in order to analyze large volumes of data in seconds, as shown in Figure 1. We expect this trend to continue with a new generation of frameworks targeting sub-second response times. Bringing response times

Figure 1: Data analytics frameworks can analyze large volumes of data with ever lower latency.

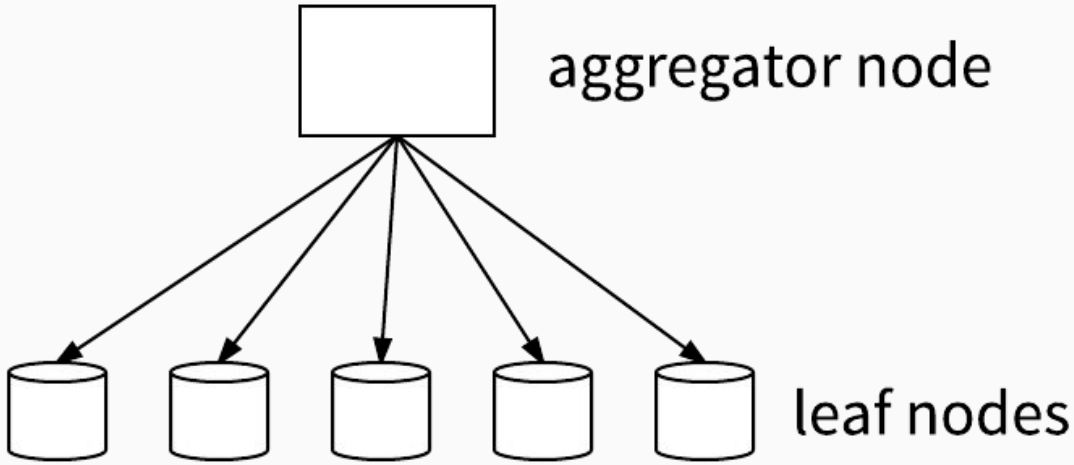
Jobs composed of short, sub-second tasks present a difficult scheduling challenge. These jobs arise not only due to frameworks targeting low latency, but also as a result of breaking long-running batch jobs into a large number of short tasks, a technique that improves fairness and mitigates stragglers [17]. When tasks run in hundreds of milliseconds, scheduling decisions must be made at very high throughput: a cluster containing ten thousand 16-core machines and running 100ms tasks may require over 1 million scheduling decisions per second. Scheduling must also be performed with low latency: for 100ms tasks, scheduling delays (including queueing delays) above tens of milliseconds represent intolerable overhead. Finally, as processing frameworks approach interactive time-scales and are used in customer-facing systems, high system availability becomes a requirement. These design requirements differ

Beating the beta factor

Idea 2: Iterative partitioning

The Universal Scalability Law applies not just to task assignment, but to *any* parallel process!

Example: Facebook's Scuba (and Honeycomb): fast distributed queries over columnar data.



Beating the beta factor

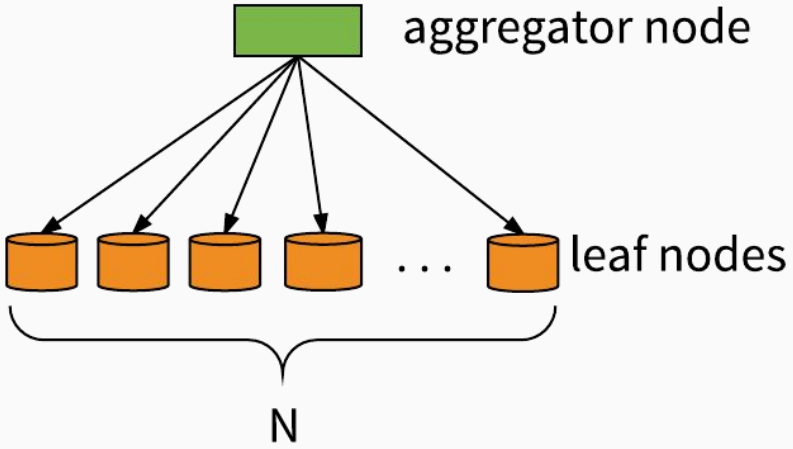
Iterative partitioning

The Universal Scalability Law applies not just to task assignment, but to *any* parallel process!

Example: Facebook's Scuba (and Honeycomb): fast distributed queries over columnar data.

1. **Leaf nodes** read data from disk, compute partial results
2. **Aggregator node** merges partial results

Question: What level of fanout is optimal?



Beating the beta factor

Iterative partitioning

The Universal Scalability Law applies not just to task assignment, but to *any* parallel process!

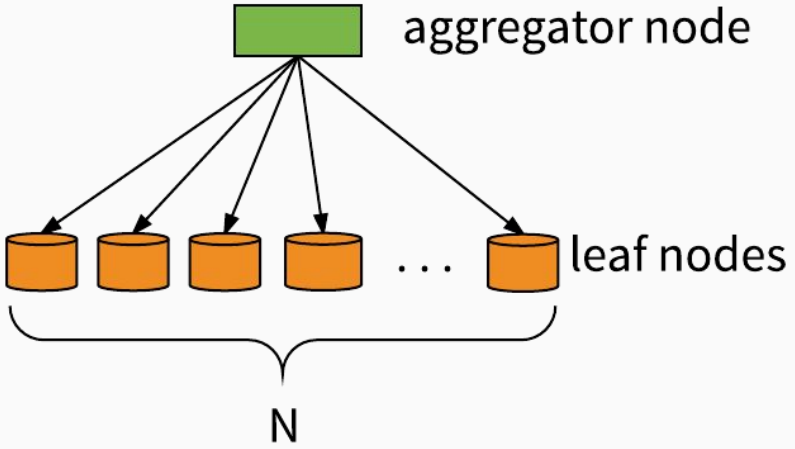
Example: Facebook's Scuba (and Honeycomb): fast distributed queries over columnar data.

1. **Scan time** is proportional to (`1 / fanout`):

   `T(scan) = S / N`

2. **Aggregation time** is proportional to number of partial results

   `T(agg) = N * β`



Beating the beta factor

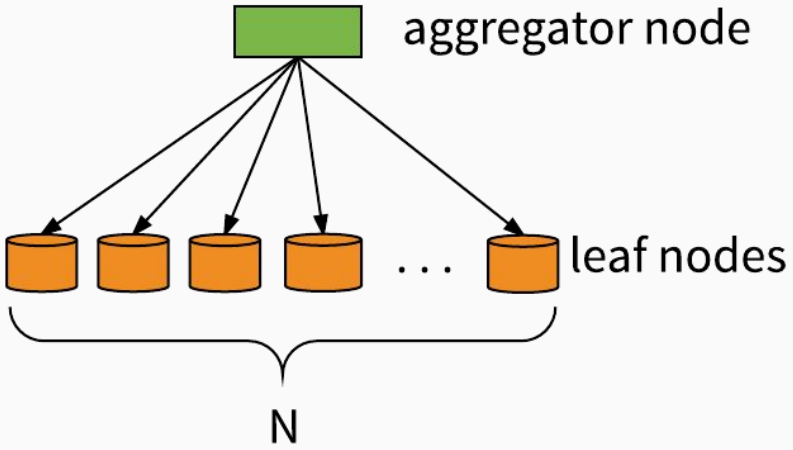
```
T(scan)  = S / N
T(agg)   = N * β
T(total) = N * β + S / N
```

(gets better as N grows)

(gets worse as N grows)

(at first gets better, then gets worse)

$$\text{throughput} \sim 1 / T(\text{total}) = N / (\beta * N^2 + S)$$



Beating the beta factor

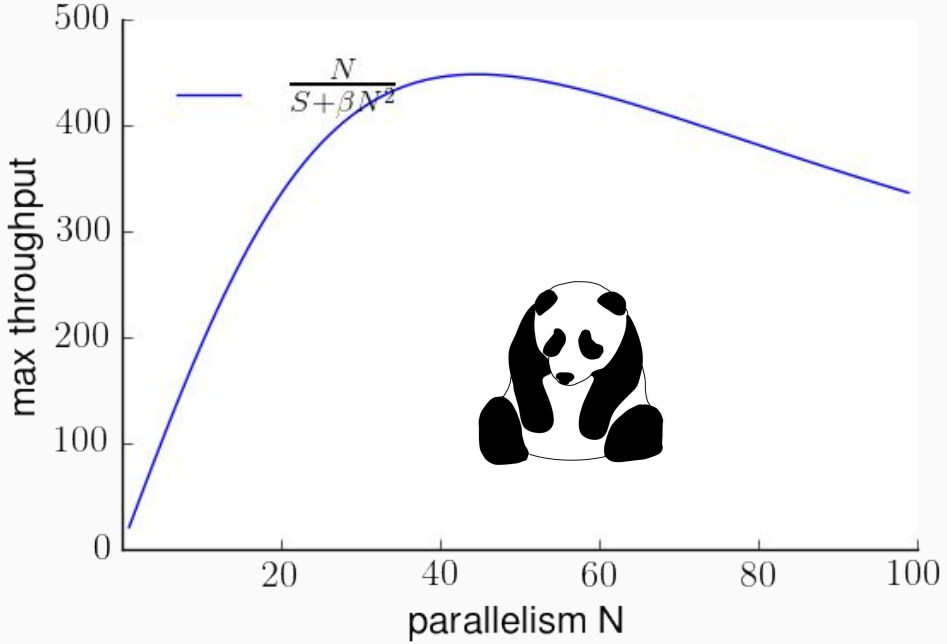
```
T(scan)  = S / N
T(agg)   = N * β
T(total) = N * β + S / N
```

(gets better as N grows)

(gets worse as N grows)

(at first gets better, then gets worse)

```
throughput ~ 1 / T(total)
           = N / (β * N² + S)
```

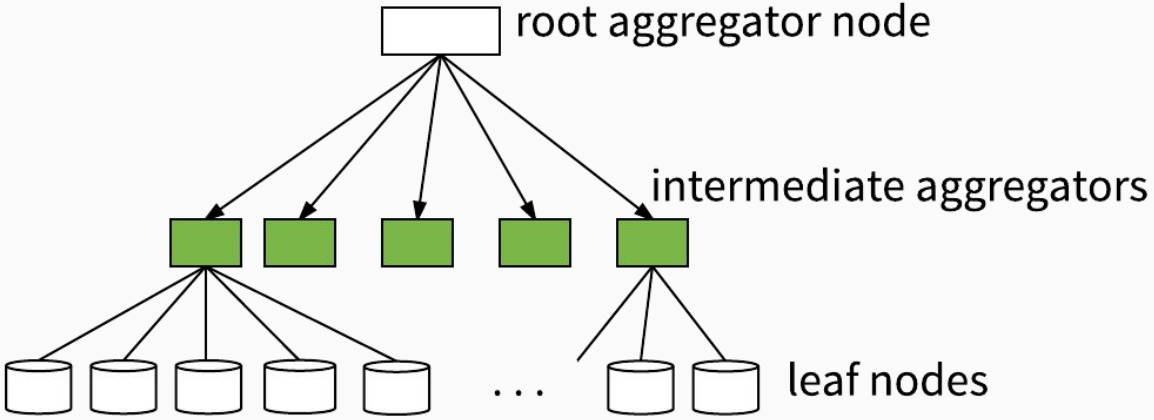


Beating the beta factor

Idea: multi-level query fanout

Throughput gets worse for large fanout, so:

- make fanout at each node a **constant** *f*
- add intermediate aggregators

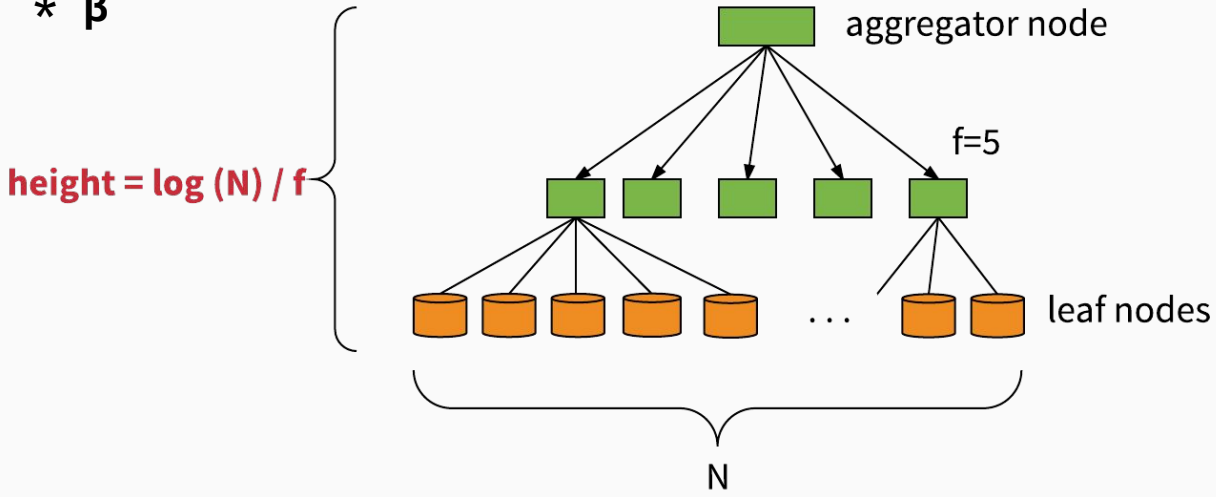


Beating the beta factor

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
```

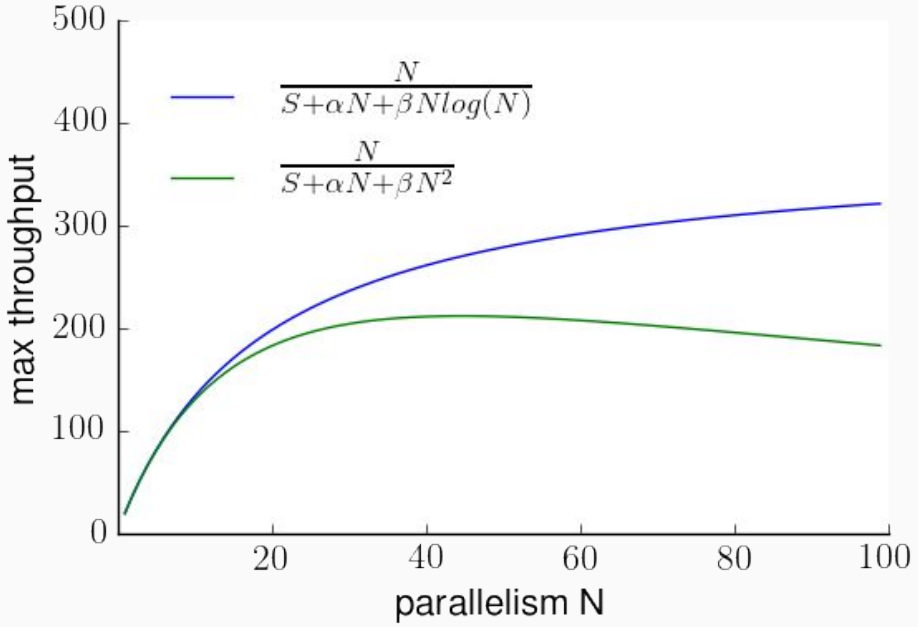


Beating the beta factor

before: `T(total) = S / N + N * β`

now: `T(total) = S / N + log(N) * β`

Result: better scaling!



Beating the beta factor

Lessons:

Making scale-invariant design decisions is hard:

- at low parallelism, coordination makes latency more predictable.
- at high parallelism, coordination degrades throughput.

But, smart compromises produce pretty good results!

- randomized choice: approximates best assignment cheaply
- iterative parallelization: amortizes aggregation / coordination cost
- USL helps *quantify* the effect of these choices!

III. In Conclusion

Queueing theory:
not so bad!

Lessons

Model building isn’t magic!

- State goals and assumptions

  Do we care most about throughput, or consistent latency?

  How is concurrency managed?

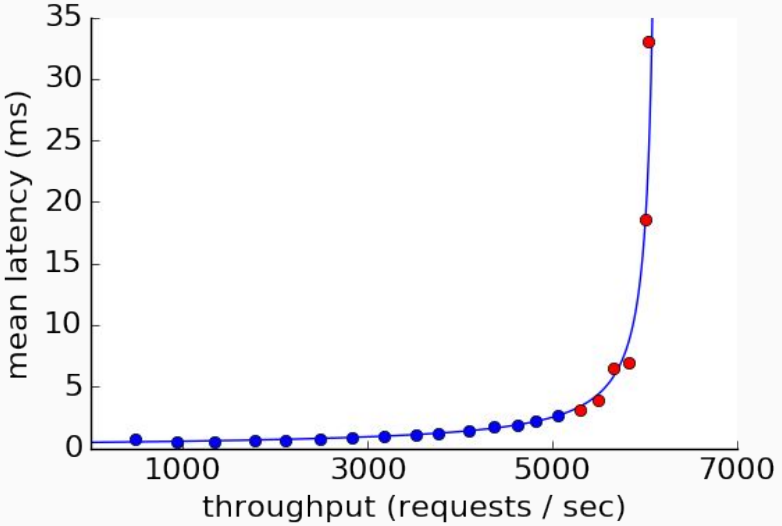
  Are task sizes variable, or constant?

- Don’t be afraid!

  Not just scary math

  Draw a picture

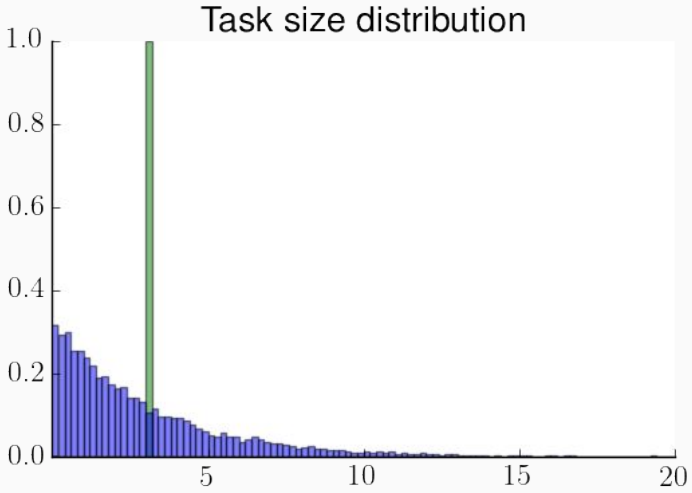
  Write a simulation

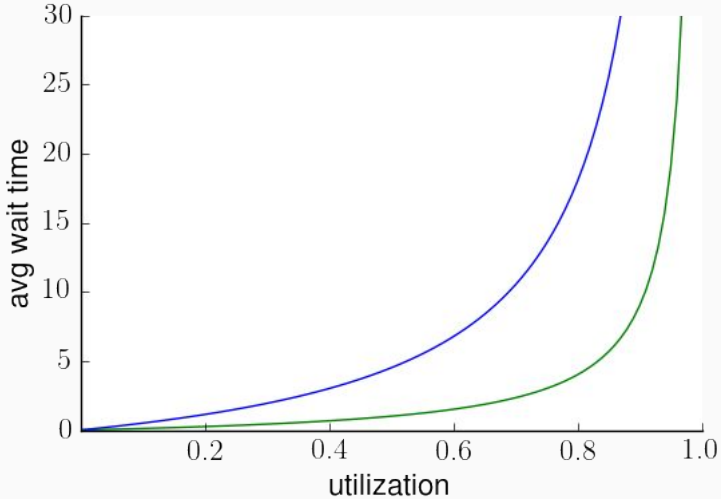


Lessons

Modelling latency versus throughput

- Measure and minimize variability
- Beware unbounded queues
- The best way to have more capacity is to do less work

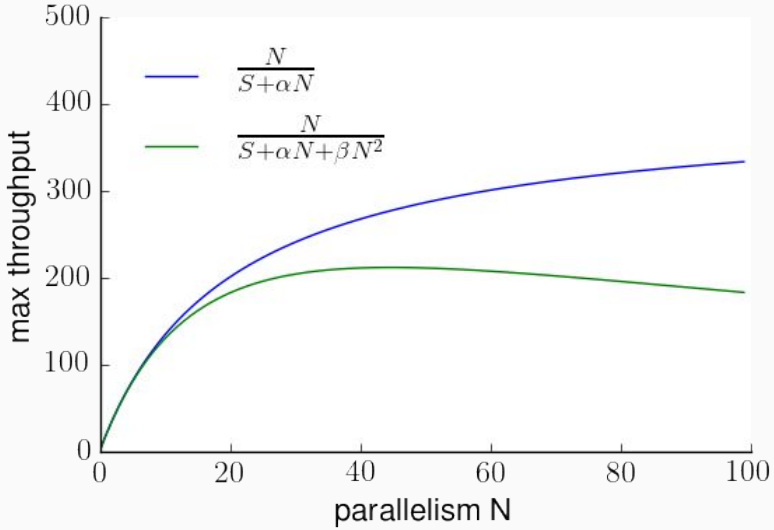


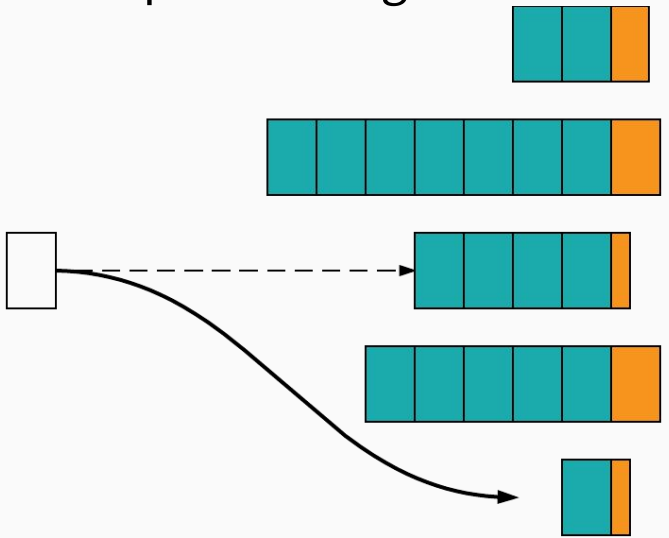


Lessons

Modelling Scalability

- Coordination is expensive
- Express its costs with the Universal Scalability Law
- Consider randomized approximation and iterative partitioning





Thank you!

@_emfree_
honeycomb.io

Special thanks to Rachel Perkins, Emily Nakashima, Rachel Fong and Kavya Joshi!

References

Performance Modeling and Design of Computer Systems: Queueing Theory in Action, Mor Harchol-Balter

A General Theory of Computational Scalability Based on Rational Functions, Neil J. Gunther

The Power of Two Choices in Randomized Load Balancing, Michael David Mitzenbacher

Sparrow: Distributed, Low Latency Scheduling, Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica

Scuba: Diving Into Data at Facebook, Lior Abraham et. al.