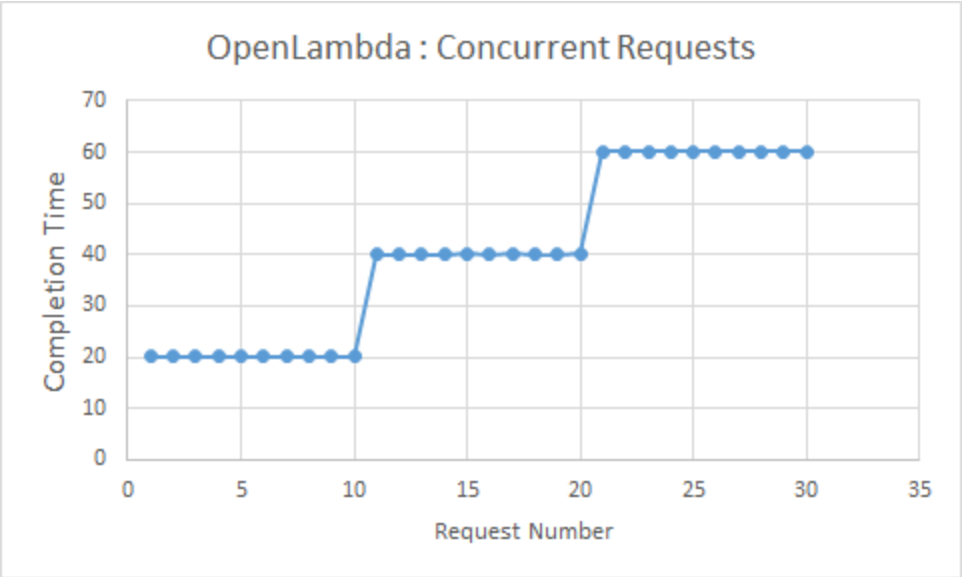
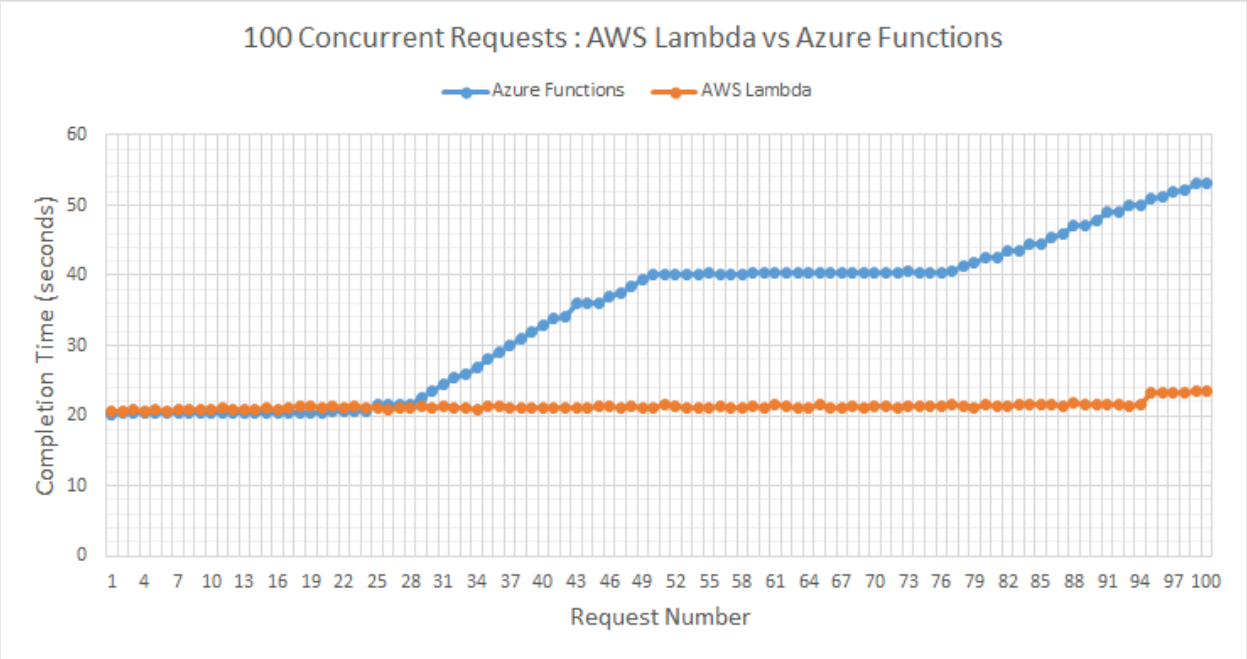
Writing Scalable Applications using Lambdas

Vishakha Dhelia, Neha Mittal, Adithya Bhat

Initial Comparison of FaaS Offerings:

- We wrote a simple application that first idles (sleeps) for 20 seconds, and then echoes the input back.
- This was used to compare the burst scaling of AWS Lambda, Azure Functions, and OpenLambda (single node setup).
- We found that AWS Lambda scales the best, while Azure Functions takes a bit longer to start up more Lambda instances.
- Open Lambda seems to be able to operate a fixed number of Lambda instances at a time.
- Further, we found that OpenLambda's HTTP Server seems to timeout with a 504 if it doesn't receive a response from the Lambda in 60 seconds, which should ideally be a configurable parameter. AWS has a similar limitation in that it's API Gateway, through which Lambdas can be triggered, has a 30 second timeout.



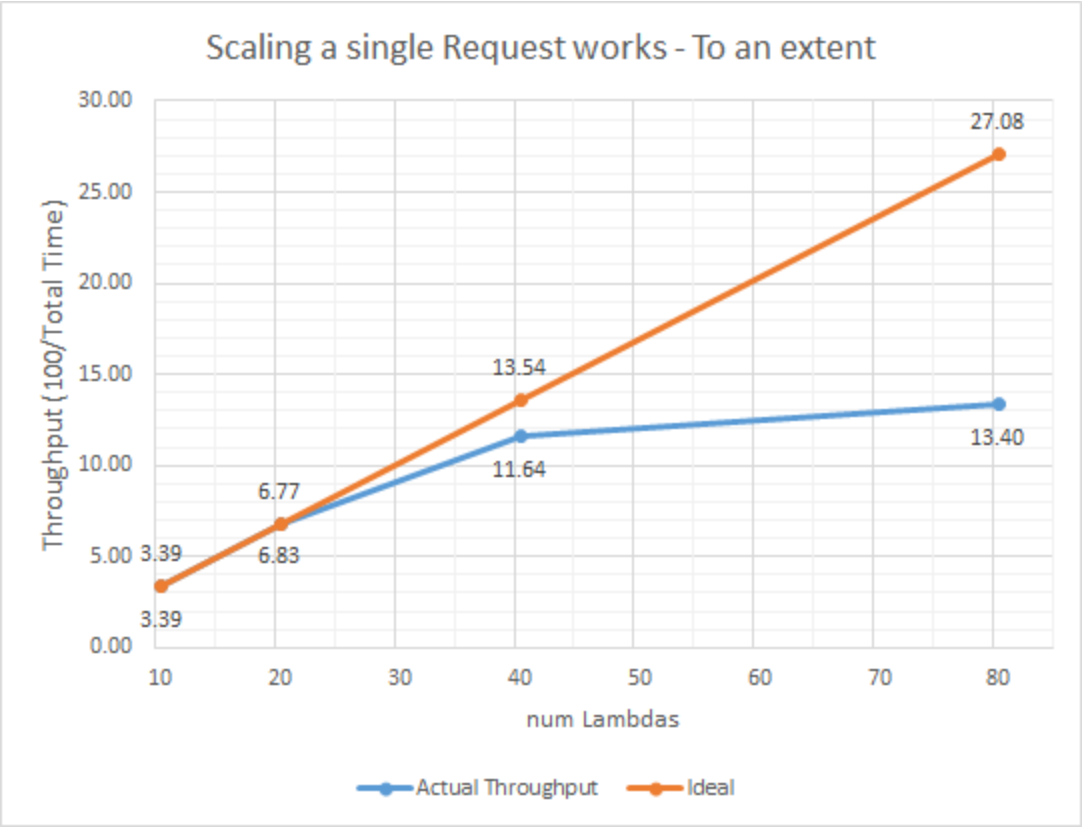


Note : This was after several runs of this size, which is why AWS Lambda services all requests (starts all nodes) near - simultaneously, and has no stragglers.

Sort, Max, and Other Applications :

- An initial idea we had on using Lambdas was an utility library for basic functions, that could **smartly decide between local execution and distributed execution**, based on the input size and computing resources available.
- Such a library could transparently increase the input size that can be handled by a regular service/system without having to use specialized big data systems, or parallel computing frameworks.
- We tried out this approach on AWS with distributed sort and distributed max performed on Lambdas, with a final ‘merge’ step at the coordinator / caller.
- However, there are various limitations of the FaaS offerings that make **applications that have to exchange considerable amounts of data during Lambda invocations** inefficient.
- For e.g., it makes sense to sort in parallel only at very high input sizes, due to the network overhead. However, there is a maximum size limit to Response messages from AWS Lambda (6MB), which means we have an upper bound on how many numbers can be processed and returned by one Lambda invocation. This forces us to use a large number of Lambda invocations, wherein stragglers affect the completion time.
- This might not be the case if data transfer happens via an in-cluster data store.

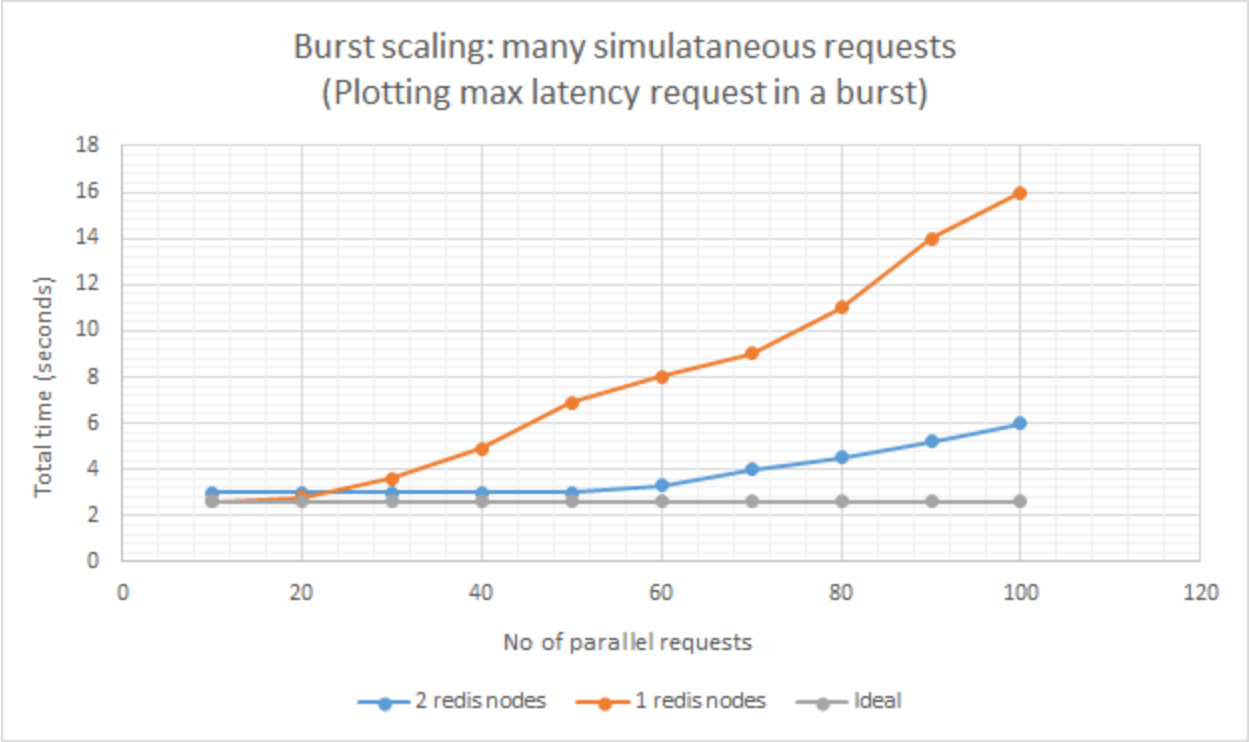
Scaling a Single ‘Nearby People’ Request :



Ideal : Doubling the number of lambdas should double the throughput/rate/speed.

- In our baseline Nearby People application that **iteratively** reads from Redis, the operation that takes the most time is the Redis read over the network; i.e., the **latency of a redis read over the network** is the bottleneck.
- Hence, we use data-parallel processing to improve the performance.
- Data-Parallel scaling with Lambdas diverges from ideal scaling starting at 30-40 Lambdas.
- Root Cause : each lambda bombards our Redis data store with thousands of requests per second, and hence we appear to hit the **throughput limit of a single Redis node**.
- NOTE : This could either be the **processing** limit of this Redis instance, or the **network** throughput limit of the machine instance it is hosted on.

Performance with burst scaling: ‘Nearby Friends’



When we sent burst requests to the “Nearby Friends” application for a user with (~600 friends) we saw the per request latency increase with increase in number of requests.
For a user with ~10 friends latency of access was nearly increase in no. of parallel requests.
Root cause: each lambda bombards our Redis data store with thousands of requests per second, and hence we appear to hit the **throughput limit of a single Redis node**.
Solution : Increase the no. of Redis nodes and observe: with 2 Redis nodes , we see perfect scalability upto 60 parallel requests and the curve is closer to ideal. This shows that we were hitting the throughput limit of Redis. Application should be able to adaptively change the number of redis nodes used.

Unintended side effects of selecting a memory configuration :

Memory configuration	Memory Ratio to 128MB	Redis operation Speed Up (Ratio to 128MB) Over the network	Sort Operation Speed Up (Ratio to 128MB) Happens Locally
1024	8	10.36	1.61
512	4	4.84	1.28
128	1	1.00	1.00

- For Lambda invocations, AWS specifications state that CPU performance varies relative to memory configured. However, nothing is mentioned about network latency / throughput.
- During profiling we, noticed that there is a huge performance difference based upon the memory chosen.
- To test this, we wrote an application that **iteratively**
    - reads data from Redis, and
    - generates and sorts an integer array.
- We believe that generating and sorting a list of 1000 integers is an CPU bound task.
- We then measure the time taken for each of these operations over a few thousand iterations, and take the average time taken per 1000 operations.
- We also observe that **memory** is not the bottleneck on the **lambda** node, and that the **redis** node does not have a **CPU or memory** bottleneck.
- We find that the CPU bound sort operation’s speed-up is not as significant as the over-the-network Redis operation.
- This leads us to **hypothesize** that configuring the memory also results in near-proportional network performance. If CPU speed-up was the cause of the Redis operation speed-up, then it is our belief that the sort operation should have seen a corresponding speed-up.