



Ray - Heikki Pulli

1

Authors aim to create distributed cluster-based AI/ML framework Ray. Ray framework is supposed to be quite performant and dynamic in any architecture and with multiple different devices, ie. CPUs and GPUs. It's supposed to do all the required steps of creating AI/ML applications, simulation, training and serving of the applications.

2



a)



```
cloud-and-edge-computing on  master at  minikube (default)
→ lscpu
Architecture:           x86_64
  CPU op-mode(s):       32-bit, 64-bit
  Address sizes:        39 bits physical, 48 bits virtual
  Byte Order:           Little Endian
CPU(s):                 8
  On-line CPU(s) list:  0-7
Vendor ID:              GenuineIntel
  Model name:           Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
    CPU family:         6
    Model:              60
  Thread(s) per core:   2
  Core(s) per socket:   4
  Socket(s):            1
  Stepping:             3
  CPU(s) scaling MHz:   85%
  CPU max MHz:          3900,0000
  CPU min MHz:          800,0000
```



Intel website of the CPU




<https://www.intel.com/content/www/us/en/products/sku/75123/intel-core-i74770k-processor-8m-cache-up-to-3-90-ghz/specifications.html>

b)

```
cloud-and-edge-computing on  master [?] at  minikube (default)
→ python week6/thread_test.py
Min time: 6.031990051269531e-05s
Max time: 0.0002415180206298828s
Avg time: 6.842136383056641e-05s
STD: 1.3042354032493471e-05s

cloud-and-edge-computing on  master [?] at  minikube (default)
→ python week6/thread_test.py
Min time: 6.0558319091796875e-05s
Max time: 0.0002300739288330078s
Avg time: 7.050871849060059e-05s
STD: 1.4577393854124811e-05s

cloud-and-edge-computing on  master [?] at  minikube (default)
→ python week6/thread_test.py
Min time: 5.888938903808594e-05s
Max time: 0.0002143383026123047s
Avg time: 6.678366661071777e-05s
STD: 1.076813145323223e-05s

cloud-and-edge-computing on  master [?] at  minikube (default)
→ 
```

Source code

https://github.com/HegePI/cloud-and-edge-computing/blob/master/week6/thread_test.py

c)

```

cloud-and-edge-computing on  master [!?] at  minikube (default)
→ python week6/function_test.py
Min time: 6.67572021484375e-06s
Max time: 5.316734313964844e-05s
Avg time: 7.525920867919922e-06s
STD: 2.3226380663609663e-06s

cloud-and-edge-computing on  master [!?] at  minikube (default)
→ python week6/function_test.py
Min time: 6.4373016357421875e-06s
Max time: 3.600120544433594e-05s
Avg time: 7.068872451782227e-06s
STD: 1.5163458930575482e-06s

cloud-and-edge-computing on  master [!?] at  minikube (default)
→ python week6/function_test.py
Min time: 6.4373016357421875e-06s
Max time: 4.1961669921875e-05s
Avg time: 7.326364517211914e-06s
STD: 2.0866946359321237e-06s

cloud-and-edge-computing on  master [!?] at  minikube (default)
→

```

Source code

https://github.com/HegePI/cloud-and-edge-computing/blob/master/week6/function_test.py

d)

All the stats are smaller when using worker function that calls the other function. Differences are small on this scale, but when running millions of tasks/s the difference is much bigger. Based on this, if the goal is to create a performant AI/ML platform the goal is to use less threading and more function invocation.

3

Idempotence means that no matter how many times code is run the results stays the same on every run. It is important for fault tolerance, because if the function does the same thing every time it is much more robust and less likely break on different inputs.

<https://en.wikipedia.org/wiki/Idempotence>

4

By saying that GCS is pub-sub key-value store implies that every component in the system basically stores their values in the GCS, effectively making them stateless. This kind of design also simplifies architecture significantly and also makes it more fault tolerant.

5

6

Lineage re-execution means using stored object data in the GCS to build the node object again. Because nodes are stateless thanks to GCS, which stores all the information about the node and its execution, it is possible to use that data and recreate node to the state before it crashed.

7

a)

X-axis represents object sizes that are written/copied into the GCS.

b)

Left (Blue) y-axis represents the amount of IOPS, input-output operations per seconds, in range of [0, 20000] in the GCS. Right y-axis represents the throughput of GCS ranging from [0, 16]GB/s.

c)

Bar plots report throughput on different amount of threads used in the test. Threads used in tests are 1,2,4,8,16. Testing is also done along x-axis (increasing object size) and averaged over the test runs.

d)

Blue line represents the amount of IOPS operations through increasing object size. IOPS is higher for smaller objects, because if the object is small, GCS uses only 1 thread to copy the object, but if object is larger than 0.5mb it uses 8 threads to copy.

Red line represents GCSs object throughput through increasing object size. Same rules apply to this as to the blue line, with smaller objects less threads are used. Because of this throughput increases when more threads are used in the writing/copying of the objects to GCS.

e)

GCS handles carefully how many threads it uses. Looking at the bar plots 8 threads are more performant than 16 threads so thats why thread cap is set 8 threads when writing/copying larger objects. With smaller objects thread count is set to 1 when object size is smaller than 0.5mb because 1 thread is more performant than 8 threads. The main take away is that increasing the thread count doesn't always mean that operations are done more quickly in the system.

8

All-reduce is an operation done to all working nodes/processes wich results to them all having the same state/values afterwards.

All-reduce wikipedia

https://en.wikipedia.org/wiki/Collective_operation#All-Reduce

An example of an all-reduce operation is for example calculating the sum of all the worker nodes and then setting all worker nodes values to that sum.

example of sum all-reduce input: node1: [1,2], node2: [2,3], node3: [3,4]

output: node1: [15], node2: [15], node3: [15]

9

The key strength of Ray is GCS, which allows all the worker nodes to be stateless. Data is most important part of machine learning, and if data is lost all the processing afterwards is useless. When/If node crashes, data isn't lost and node can be restarted using the data in the GCS.

Other key strength of Ray is its scalability. Training process with large amounts of data can be time consuming but Ray allows scaling the training.

10

11

In figure 8 authors only report how well Ray performs with tasks, but they don't elaborate any situations where Ray's performance would drop. Hard to believe that Ray would be a perfect solution for AI/ML workload management, but authors give that assumption.

In figure 10a the y-axis is skewed to make the line straight.

In all the figures it seems that authors only show how great Ray is in certain tasks, but they don't tell any situations where it might not work properly other than the case of not using GCS flush.

12

As a cloud provider an important thing for providing Ray as a service for customers would be allocating enough resources for multiple Ray instances. Also giving the customers possibility to finetune their resource needs would be important, and not just letting Ray to request all the possible resources to its self.

Also providing customers a UI to interact with their Ray instance would be important. Using the UI would be possible to define what kind of workloads would be running in the instance and monitoring the results and how well the Ray instance performs.

To automate this workflow a CLI tool would be also important. CLI tool would have access to the Ray instance and it would be possible to start, stop and report results workloads of the workloads. This would be automated when CLI tool would be part of some workflow tool, ie. github actions or travis.