Assignment 3

Adithya Bhat, Aparna Subramanian

ML on Spark vs TensorFLow

Spark :

- Iterative training is not a natural fit. RDD lineage chains will become extremely long.
    - Checkpointing alleviates this, but is again a costly activity
- Hence, error recovery can be potentially a bottleneck
- Experimentation is restricted, due to the rigidity of RDDs
+ Performance is much better
+ Integration with the Spark Ecosystem, other parts of the Apache Ecosystem
    + Easier to tie into an entire application
+ Targeted for general computation, MLLib provides ML specific support
+ Core programming abstraction : RDDs
    + RDDs themselves are distributed across the cluster

TensorFlow :

- Low level operations/API
- Practical usage downside : in development, in case of errors, sometimes freezes without error output
- Performance worse than alternatives
+ Iterative execution supported
+ Variety of devices supported
    + Kernels allow the same high-level code to work optimally across different devices
+ Greater flexibility
    + Subgraph execution
    + Conditional execution
    + Looping
+ Same code can work optimally on different machines thanks to different kernels
+ Good documentation / TensorBoard is useful
+ Designed specifically with ML in mind
+ The core programming abstraction is a Tensor
    + Optimizations for distribution, such as gather, Sparse Tensors in developers control
    + Tensors themselves are not distributed.

Bottleneck Analysis

Measurements

- Network bandwidth : 1.8 Gbps between vm-1 and vm-2
- Network upload bandwidth max ever observed for vm-1 : 2.63Gbps
- Network RTT : order of 1ms
- Size per record read : 300 Bytes
- I used dstat to gather data as 30 second delayed snapshots, (tabulated below), but this wasn't my only source of analysis.
    - Issues - usually one core is used for a TensorFlow job.
    - Dstat reports utilization across cores, however.
    - Hence top, that shows per core utilization, is more useful.
    - That said, dstat numbers also support the pattern observed.
- The utilization of various resources followed a periodic pattern, depending on the code running
    - E.g. testing - only one machine - no n/w utilization, maxed out CPU utilization
    - Training - max n/w utilization
- Hence, i used top and iftop to view the utilization live during execution.

Synchronous

V1 - Dense Tensors

- Took 7-8 seconds per iteration
- Issue : Model is a tensor of 33 million, each a float
- Additionally, the local gradient passed back also has the same shape
- It looks like one TF operation is assigned to one 'device' (CPU core in this case)
- This core sees high utilization during the multiplication and other processing operations.
    - Used 'top' and 'iftop' to monitor utilization patterns within a run
- Network bandwidth is the primary bottleneck, CPU utilization is high during certain operations

V2 - Sparse Tensors, Gather, etc

- Down to 3-4 seconds per iteration
- Changes : using gather to selectively transfer the W (model) tensor

- Potentially, network latency could be a bottleneck
    - However, RTT = order of 1-2ms, hence probably not the case
- Potentially, disk bandwidth could be a bottleneck
    - However, each file has 2M entries, and has size ~600MB
    - Ie, one entry has ~300 Bytes, sequential read
    - Hence, disk read is not likely to be the bottleneck
- CPU and memory utilization are well below 100%
- Based on iperf, and iftop analysis, bandwidth is the order of 1.7-1.8 Gbps
- This bandwidth is often reached on vm-1 on synchronous execution
- Further, the iteration time is upper-bound by the slowest executor
- Hence, it appears that network bandwidth is still the bottleneck.

Asynchronous

V2 - Sparse Tensors, Gather, etc

- No cooperation necessary between the different tasks
- CPU utilization higher than synchronous
- Please refer to the disk and network latency explanations under synchronous too.
- On vm-1, where the model tensor is, 1-1.5s / iteration
    - High CPU utilization
- On other vms, 3-3.5 s / iteration
    - CPU utilization not as high as vm-1
    - Network transfer of model and gradient
    - Some contention possible on updating w (model).
- NOTE : even if the sessions and operations are on another machine/task, if the process is kicked off from vm-1, there is some CPU utilization on vm-1, of the order of 5-20%.
    - Perhaps coordination is still run from this machine.
    - Hence, i'm using pdsh to scp and kick off the task from the different vms
    - This improved speed of execution
- Overall, i see roughly 100 iterations per minute (1.5 iterations per second)

Data

- Remember that this is CPU utilization across cores, and out program doesn't have much scope for parallelism, and hence usually runs only on one core.
- Testing is done only on vm-1, where the model is local. Hence, for those brief periods (20-ish seconds), network utilization is zero, and cpu utilization is the bottleneck.

Synchronous

Synchronous	**vm-1**	**vm-2**	**vm-3**	**vm-4**	**vm-5**
avg_idle_cpu (%)	73.9946	93.8766	95.7151	94.7432	94.4365
avg_mem_used GB	11.5221	2.2254	1.4805	1.9703	2.1774
avg_mem_free GB	7.4683	17.0797	17.8120	17.3103	17.1183
avg nw_rec MBps	1.3900	40.6234	40.6697	40.6633	40.6706
avg nw_send MBps	156.1833	0.0789	0.6525	0.5889	0.0717
avg disk_read MBps	0.0134	0.0058	0.0058	0.0059	0.0059
avg disk_write MBps	0.0218	0.0047	0.0048	0.0046	0.0047

- CPU utilization is low, memory is not under strain.
- Note that since the straggler limits the operation, **average** bandwidth is less than bandwidth cap.
- Point to note is that this is bursty.. Please refer to the analysis above. Stragglers, etc.

Asynchronous

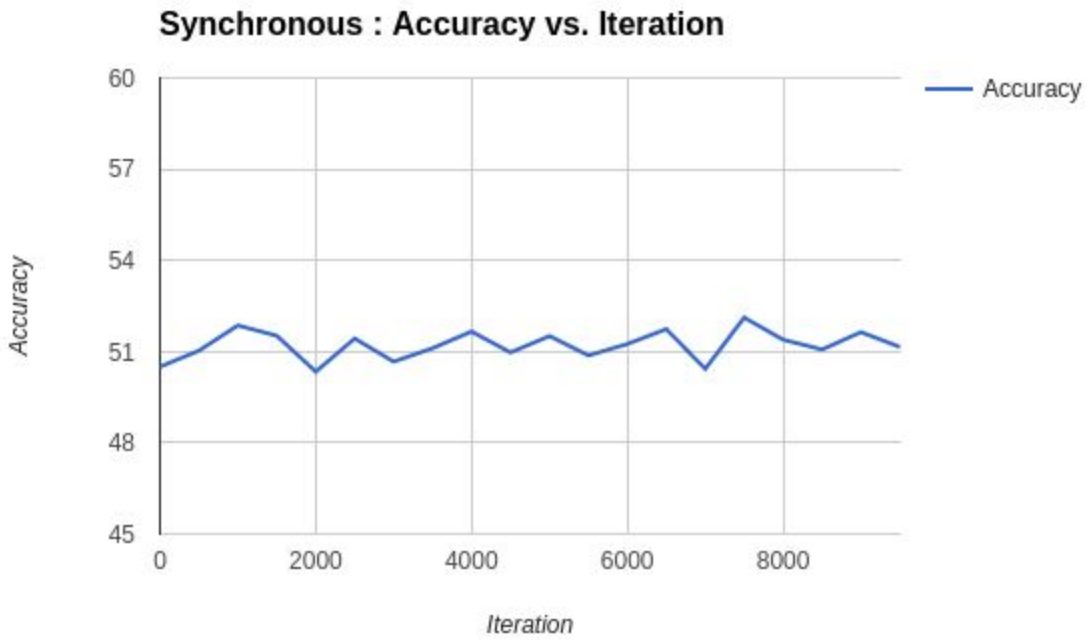
Asynchronous	**vm-1**	**vm-2**	**vm-3**	**vm-4**	**vm-5**
avg_idle_cpu (%)	46.2315	77.3645	77.3645	78.9952	79.6714
avg_mem_used GB	15.6286	2.6168	1.8753	2.3535	2.5317
avg_mem_free GB	3.7590	16.8108	17.5355	17.0552	16.8795
avg nw_rec MBps	0.5336	78.7882	77.4975	69.8809	71.0476
avg nw_send MBps	288.1872	0.1453	0.1324	0.0883	0.1605
avg disk_read MBps	0.0082	0.0037	0.0042	0.0045	0.0046
avg disk_write MBps	0.0169	0.0020	0.0022	0.0020	0.0021

- CPU utilization is higher
- Note that there is no straggler issue here.
- Average n/w bandwidth = 288MBps = 2.304Gbps appears to be near or at the maximum transmission bandwidth possible (2.5Gbps)

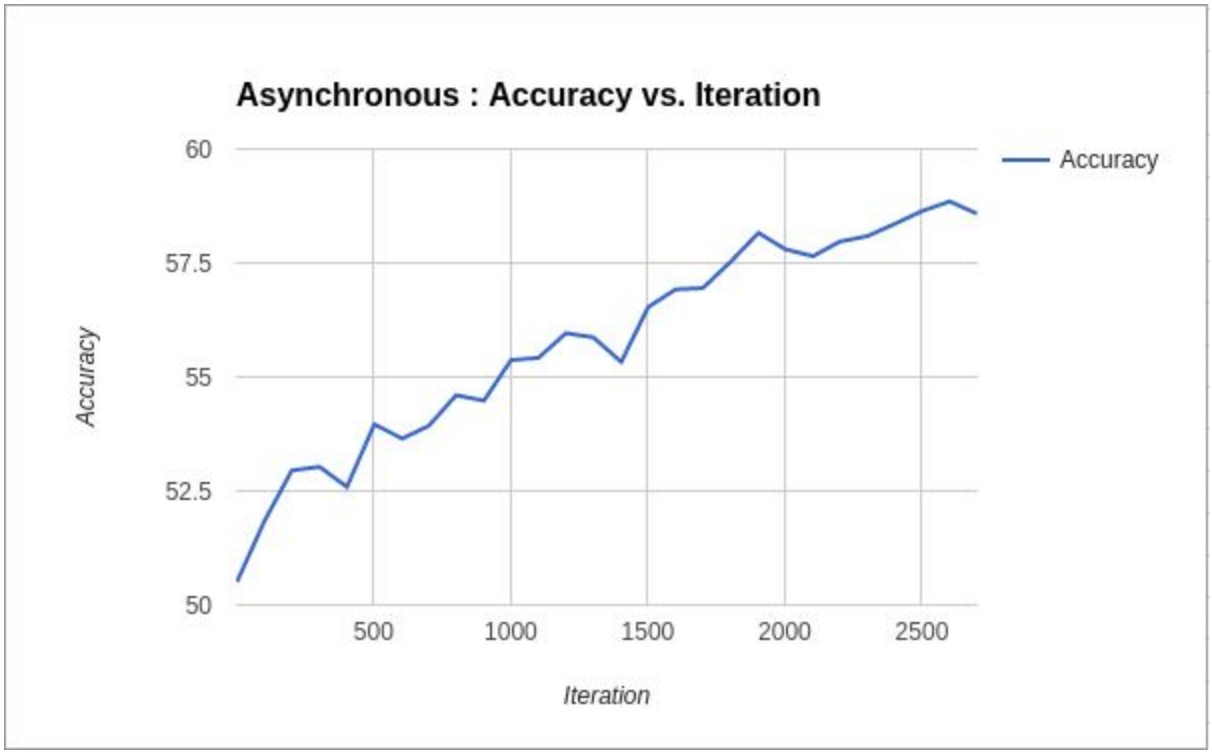
Graphs

- Note : I initialized with values between -10 and +10, which is very different from the all-ones model / random between -1 and +1 used by most people. Further, due to bug fixes, didn't have the time to run for many iterations…
    - For synchronous, reporting values with a much tinier eta (multiplied twice)
- I'm plotting accuracy, which is (num_correct / num_tested)

Synchronous



Asynchronous



README

- Before running any application, please do source ~/gitRepository/TensorFlow/uploaded-scripts/tfdefs.sh

BigMatrixMultiplication.py:

Running :

- Start cluster
- python bigmatrixmultiplication.py

Discussion :

- Changes on top of exampleMatmulSingle.py
  - Break up into individual block sub-matrices
  - Compute intermediate traces performing matmul on individual blocks
  - Aggregate the traces.
- Key Idea 1 :
  - Distribute the individual blocks over the different machines
  - Distribute the intermediate trace computations
  - Finally aggregate and add on one device
- Key idea 2 :
  - Intermediate trace computation involves block matrix multiplication
  - Optimize placement such that both the matrices to be multiplied are placed on the same machine
  - `with tf.device("/job:worker/task:%d" % int((i+j)%5)):`

SGD:

Running:

- Start cluster
- python synchronoussgd.py
- ./launch_asyncsgd.sh

Discussion:

- Please also see the discussion on bottlenecks in the pdf.
- Model (w) tensor is on vm-1 (task 0)
- Input files distributed across all VMs (lesser on vm-5)
    - Each vm processes its input
- Testing done only on vm-1
    - Reason - co-location with model to speed it up
    - No network usage.
- Initially, passed the dense w tensor to each task for processing each input
    - This is a 33M long float tensor
    - Huge bandwidth utilization, this is the bottleneck
- Optimization
    - Using tf.gather(input_example.indices())
    - Key idea : not all the fields are set in the input example
    - Hence, gather only the necessary indices
    - Similarly, send only the indices to be updated back to vm-1 for updating the model
    - Improvement in run time

Synchronous:

- Updates are coordinated centrally by vm-1
    - In one session.run, all the tasks process one (batch) of the inputs and computes the local gradient
    - Local gradients are aggregated and added to the W vector on vm-1

Asynchronous:

- W is initialized on vm-1
- Each task/vm runs its own session, updates W independently
- Also, we are keeping a shared **count** variable on vm-1 to keep track of how many training examples have been processed totally.
- All the different tasks to run on different machines can be launched from the same vm-1

    - Thanks to the ‘with tf.device()’ option for task placement
    - However, I noticed that the each of individual tasks still utilize a non-insignificant amount of CPU on vm-1 (5-20% each)
    - This appears odd, since only coordination, if that, should be done on this vm, core updating logic is executed on the individual vms
- Hence, i’m scp-ing and executing the individual clients remotely
    - Saw a slight performance benefit in doing this