6.824 2018 Lecture 14: Parameter Server Case Study

Scaling Distributed Machine Learning with the Parameter Server
Li, Andersen, Park, Smola, Ahmed, Josifovski, Long, Shekita, Su
OSDI 2014

Today: distributed machine learning
  Case study: Parameter Server
  Why are we reading this paper?
    influential design
    relaxed consistency
    different type of computation

Machine learning primer
  models are function approximators
    true function is unknown, so we learn an approximation from data
    ex: - f(user profile) -> likelihood of ad click
        - f(picture) -> likelihood picture contains a cat
        - f(words in document) -> topics/search terms for document
    function is typically very high-dimensional
  two phases: training and inference
    during training, expose model to many examples of data
      supervised: correct answer is known, check model response against it
      unsupervised: correct answer not known, measure model quality differently
    during inference, apply trained model to get predictions for unseen data
    parameter server is about making the training phase efficient

Features & parameters
  features: properties of input data that may provide signal to algorithm
    paper does not discuss how they are chosen; whole ML subfield of its own
  ex: computer vision algorithm can detect presence of whisker-like shapes
    if present, provides strong signal (= important param) that picture is a cat
    blue sky, by contrast, provides no signal (= unimportant parameter)
    we'd like the system to learn that whiskers are more important than blue sky
      i.e., "whisker presence" parameter should converge to high weight
  parameters are compact: a single floating point or integer number (weight)
    but there are many of them (millions/billions)
    ex: terms in ad => parameters for likelihood of click
      many unique combinations, some common ("used car") some not ("OSDI 2014")
    form a giant vector of numbers
  training iterates thousands of times to incrementally tune the parameter values
    popular algorithm: gradient descent, which generates "gradient updates"
    train, compute changes, apply update, train again
    iterations are very short (sub-second or a few seconds, esp. with GPUs/TPUs)

Distributed architecture for training
  need many workers
    because training data are too large for one machine
    for parallel speedup
    parameters may not fit on a single machine either
  all workers need access to parameters
  plan: distribute parameters + training data over multiple machines
    partition training data, run in parallel on partitions
    only interaction is via parameter updates
  each worker can access *any* parameter, but typically needs only small subset
    cf. Figure 3
    determined by the training data partitioning
  how do we update the parameters?

Strawman 1: broadcast parameter changes between workers
  at end of training iteration, workers exchange their parameter changes
  then apply them once all have received all changes
  similar to MapReduce-style shuffle
  Q: what are possible problems?
  A: 1) all-to-all broadcast exchange => ridiculous network traffic
     2) need to wait for all other workers before proceeding => idle time

Strawman 2: single coordinator collects and distributes updates
  at end of training iteration, workers send their changes to coordinator
  coordinator collects, aggregates, and sends aggregated updates to workers
    workers modify their local parameters
  Q: what are possible problems?
  A: 1) single coordinator gets congested with updates
     2) single coordinator is single point of failure
     3) what if a worker needs to read parameters it doesn't have?

Strawman 3: use Spark
  parameters for each iteration are an RDD (params_i)
  run worker-side computation in .map() transformation
    then do a reduceByKey() to shuffle parameter updates and apply
  Q: what are possible problems?
  A: 1) very inefficient! need to wait for *every* worker to finish iteration
     2) what if a worker need to read parameters it doesn't have? normal
        straight partitioning with narrow dependency doesn't apply

Parameter Server operation
  [diagram: parameter servers, workers; RM, task scheduler, training data]
  start off: initialize parameters at PS, push to workers
  on each iteration: assign training tasks to worker
    worker computes parameter updates
    worker potentially completes more training tasks for the same iteration
    worker pushes parameter updates to the responsible parameter servers
    parameter servers update parameters via user-defined function
      possibly aggregating parameter changes from multiple workers
    parameter servers replicate changes
      then ack to worker
    once done, worker pulls new parameter values
  key-value interface
    parameters often abstracted as big vector w[0, ..., z] for z parameters
      may actually store differently (e.g., hash table)
    in PS, each logical vector position stores (key, value)
      can index by key
      ex: (feature ID, weight)
    but can still treat the (key, value) parameters as a vector
      e.g., do to vector addition
  what are the bottleneck resources?
    worker: CPU (training compute > parameter update compute)
    parameter server: network (talks to many workers)

Range optimizations
  PS applies operations (push/pull/update) on key ranges, not single parameters
  why? big efficiency win due to batching!
    amortizes overheads for small updates
    e.g., syscalls, packet headers, interrupts
    think what would happen if we sent updates for individual parameters
      (feature ID, weight) parameter has 16 bytes
      IP headers: at least 20 bytes, TCP headers: dito
      40 bytes of header for 16 bytes of data -- 2.5x overhead
      syscall: ~2us, packet transmission: ~5ns at 10 Gbit/s -- 400x overhead
    so send whole ranges at a time to improve throughput
  further improvements:
    skip unchanged parameters
    skip keys with value zero in data for range
      can also use threshold to drop unimportant updates
    combine keys with same value

API
  not super clear in the paper!
  push(range, dest) / pull(range, dest) to send updates and pull parameters
  programmer implements WorkerIterate and ServerIterate methods (cf. Alg. 1)
    can push/pull multiple times and in response to control flow
  each WorkerIterate invocation is a "task"
    runs asynchronously -- doesn't block when pull()/push() invoked
    instead, run another task and come back to it when the RPC returns
  programmer can declare explicit dependencies between tasks
    details unclear

Consistent hashing
  need no single lookup directory to find parameter locations
    unlike earlier PS iteration, which used memcached
  parameter location determined by hashing key: H(k) = point on circle
  ranges on circle assigned to servers by hashing server identifier, i.e., H'(S)
    domains of H and H' must be the same (but hash function must not)
    each server owns keys between its hash point and the next
  well-known technique (will see again in Chord, Dynamo)

Fault tolerance
  what if a worker crashes?
    restart on another machine; load training data, pull parameters, continue
    or just drop it -- will only lose a small part of training data set
      usually doesn't affect outcome much, or training may take a little longer
  what if a parameter server crashes?
    lose all parameters stored there

  replicate parameters on multiple servers
    use consistent hashing ring to decide where
    each server stores neighboring counter-clockwise key ranges
    on update, first update replicas, then reply to worker
      no performance issue because workers are async, meanwhile do other tasks
  on failure, neighboring backup takes over key range
    has already replicated the parameters
    workers now talk to the new master

Relaxed consistency
  many ML algorithms tolerate somewhat stale parameters in training
    intuition: if parameters only change a little, not too bad to use old ones
      won't go drastically wrong (e.g., cat likelihood 85% instead of 91%)
    still converges to a decent model
    though may take longer (more training iterations due to higher error)
  trade-off: wait (& sit idle) vs. continue working with stale parameters
    up to the user to specify (via task dependencies)

Vector clocks
  need a mechanism to synchronize
    when strong consistency is required
    even with relaxed consistency: some workers may by very slow
      want to avoid others running ahead and some parameters getting very stale
  i.e., workers need to be aware of how far along others and the servers are
  vector clocks for each key
    each "clock" indicates where *each* other machine is at for that key
  [example diagram: vector with time entries for N nodes]
  only really works because PS uses ranges!
    vector clock is O(N) size, N = 1,000+
    overhead would be ridiculous if PS stored a vector clock for every key
    but range allows amortizing it
    PS always updates vector clock for a whole range at a time

Performance
  scale: 10s of billions of parameters, 10-100k cores (Fig. 1)
  very little idle time (Fig. 10)
  network traffic optimizations help, particularly at servers (Fig. 11)
  relaxed consistency helps up to a point (Fig. 13)

Real-world use
  TensorFlow, other learning frameworks
    high performance: two PS easily saturate 5-10 GBit/s
  influential design, though APIs vary
  discussion
    ML is an example of an application where inconsistency is often okay
    allows for different and potentially more efficient designs

References
 * PS implementation in TensorFlow: https://www.tensorflow.org/deploy/distributed