Scaling Machine Learning with the Parameter Server

Alex Smola with Mu Li, Li Zhou, Dave Andersen, Junwoo Park, Aaron Li, Amr Ahmed, Vanja Josifovski, Bor-Yiing Su, Eugene Shekita, James Long

http://www.istc-cc.cmu.edu/
Background
The Challenge

- **Scale**
  - 100s Terabytes of data
  - 1000s of computers
  - 100 Billions of parameters
- **Reality**
  - Faulty machines
  - Shared cluster
- **Performance**
  - Front end serving machines
  - Real time response
• Many models have $O(1)$ blocks of $O(n)$ terms (LDA, logistic regression, recommender systems)
• More terms than what fits into RAM (personalized CTR, large inventory, action space)
• Local model typically fits into RAM
• Data needs many disks for distribution
• Decouple data processing from aggregation

• Optimize for the 80% of all ML problems
• Clients have local view of parameters
• P2P is infeasible since $O(n^2)$ connections
• Synchronize with parameter server
  • Reconciliation protocol
    average parameters, lock variables
  • Synchronization schedule
    asynchronous, synchronous, episodic
  • Load distribution algorithm
    uniform distribution, fault tolerance, recovery

Smola & Narayanamurthy, 2010, VLDB
Gonzalez et al., 2012, WSDM
Shervashidze et al., 2013, WWW
Communication pattern

- **client**
- **server**

- **client** syncs to many masters
- **master** serves many clients

**put(keys, values, clock)**, **get(keys, values, clock)**
Architecture

High-performance and multi-threaded linear algebra operations are provided between parameters and local training data. There are two challenges. One is flexible and efficient communication between workers and servers. A natural thought is viewing it as a distributed key-value system. The standard API that setting and getting a key, however, is potentially inefficient. Because both key and value are often basic types such as integers and floats, the overhead associated with sending a single key-value pair would be large. Our insight comes from the observation that a large portion of machine learning algorithms represent parameters as mathematical objects, such as vectors, matrices, or tensors. On a logical time (or an iteration), typically a part of the object is updated. For example, a segment of a vector, or a row of a matrix. From the key-value system perspective, it is equivalent to synchronizing a range of keys each time. This batched communication pattern could reduce the overhead and make it easy to do optimization. Furthermore, it allows us to build an efficient vector clock that supports the flexible consistency requirements of machine learning tasks.

The other challenge comes from the fault tolerances. We implemented the system and did awesome experiments. We briefly compare the parameter server with other general-purpose machine learning systems, more details will be provided in Section 6. GraphLab is ....

Furthermore, parameter server is highly efficient. Figure 1 compares the largest experiments each system performed. Distbelief (DNN), VW (LR), Yahoo! LDA (LDA), Graphlab (LDA), Naiad (LR), REFF (LR), Petuum (Lasso), MLbase (LR), Parameter server (Sparse LR) are listed. Figure 2 shows the architecture of parameter server.
Key layout & recovery
Caching
  
  - Store many (key, value) pairs
  - Linear scaling in clients & servers
  - Automatic key distribution

memcached
  
  - (key, value) servers
  - client access library distributes access patterns
  - randomized O(n) bandwidth
  - aggregate O(n) bandwidth
  - load balancing via hashing
  - no versioned writes / vector clocks
  - very expensive to iterate over all keys for a given server

\[ m(key, M) = \arg\min_{m' \in M} h(key, m') \]
Virtual servers
• loadbalancing
• multithreading

DHT
• contiguous key range for clients
• easy bulk sync
• easy insertion of servers

Replication
• Machines hold replicas
• Easy fallback
• Easy insertion / repair
Keys arranged in a DHT

- Virtual servers
  - loadbalancing
  - multithreading
- DHT
  - contiguous key range for clients
  - easy bulk sync
  - easy insertion of servers
- Replication
  - Machines hold replicas
  - Easy fallback
  - Easy insertion / repair

Yes, we screwed up before!
And everyone copied us!
Key layout

servers
1
2
3
4
5
6

A B C D E

original

replica
Key layout

servers: 1 2 3 4 5 6

- A: original
- B
- C
- D
- E

replica: 4 5 6
Key layout

Servers

1

2

4

5

6

A

B

C

D

E

Original

Copy
• Precopy server content to new candidate (3)
• After precopy ended, send log
• For k virtual servers this causes $O(k^{-2})$ delay
• Consistency using vector clocks
Communication
Message Compression

- Convergence speed depends on communication efficiency
  - Sending (key,value) pairs is inefficient
    Send only values (cache key list) instead
  - Sending small gradients is inefficient
    Send only sufficiently large ones instead
  - Updating near-optimal values is inefficient
    Send only large violators of KKT conditions
  - Filter data before sending
• **Scheduling**
  have controller decide when to send
  (this requires very smart controller)

• **Filtering**
  have algorithm decide when to shut up
  • Gradient (only send large gradients)
  • KKT (only send variables violating KKT)
  • Randomized (sparse random vectors)
  • Quantization (reduce accuracy)
We show the convergence results in Figure 8. As can be seen, baseline-B outperforms baseline-A, because the block proximal gradient method converges faster than L-BFGS on this dataset. Parameter server further improves baseline-B even by using the same algorithms, because of the relaxed consistency model parameter server adopted. The KKT filter significantly reduced the network traffic. It skipped 93.4% of gradients should be sent, which are shown in Figure 9. The bounded delay consistency allow to start updating the next block without waiting the data communication finished in previous blocks. With $\tau = 4$, it affects the convergence speed little, but further hide the communication cost. The benefit of relaxed consistency model can be clearer seen in Figure 10, which shows the time decomposition of a worker node. As can be seen, System-A has around 32% idle time, while this number goes to 53% for system-B due to the barrier placed in each block. However, the parameter server reduces this cost under 2%. But also note that parameter server uses more computational time than system-B. The reason are two-fold. On one hand, system-B optimizes its gradient calculating on this dataset by careful data transformation. On the other hand, the asynchronous updates of parameter server needs more iterations to achieve the same objective value as system-B. However, due to the significant gain on reducing communication cost, parameter server reduces the total time into half.

### 6 Related Works

There exist several general purpose distributed machine learning systems. Mahout \[6\], based on Hadoop \[1\] and MLI \[28\], based on Spark \[30\], adopt the iterative MapReduce \[15\] framework. While Spark is substantially superior to Hadoop MapReduce due to its preservation of state and optimized execution strategy, both of these approaches use a synchronous iterative communication pattern. This makes them vulnerable to nonuniform performance distributions for iterative machine learning algorithms, i.e. machines that might happen to be slow at any given time. To overcome this limitation, distributed GraphLab \[22\] asynchronously schedules communication using a graph abstraction. It, however, lacks the elastic scalability of the map/reduce-based frameworks, and relies on coarse-grained snapshots for recovery. Moreover, global variables synchronization is not a first-class primitive. Of course, beyond these general frameworks, numerous systems have been developed that target specific applications, such as \[3,14,25,23,29,12,16\].

We found that many inference problems have a rather restricted structure in terms of their parametrization where considerable gains can be made by exploiting this design. For instance, generalized linear models typically use a single massive parameter vector, or topic models use an array of sparse vectors. In general, many relevant large-scale graphical models consist largely of a small number of plates, thus allowing for a repeated structure of a small number of components which are shared between observations and machines. This offers considerable efficiencies by performing these operations in bulk and by specializing synchronization primitives for the specific datatypes.
Message Aggregation on Server

Algorithm 1

Set range \( R \) of node \( i \) into \( t \):

Require: \( S_1, \ldots, S_n \) are the existing ranges

1: for \( S_2 \in \{S_i: S_i \notin R \} \) do
2: if \( S \not\subset R \) then
3: \( vc_i(S) \) \( \in t \)
4: else
5: \( a_{\max}(S_0, R_0) \) and \( b_{\min}(S_1, R_1) \)
6: split \( S \) into \( [S_0, a), [a, b), [b, S_1) \)
7: \( vc_i([a, b)) \) \( \in t \)
8: end if
9: end for

4.2 Message

Messages carry the data communicated between nodes. It consists of a list of key-value pairs and the timestamp:

\[ vc_i(R) \], \((k_1, v_1), \ldots, (k_p, v_p)\]

The keys may be a subset of all available keys within range \( R \). For the missing keys, we assign them the same timestamp but with 0 or unchanged values.

There are several ways to reduce the size of a message. First of all, the vector clock can only have the sender's time. For example, when a worker pushes data to a server, the worker doesn't necessarily need to send others timestamps except for itself. Secondly, the keys a node sending to another may be unchanged if the same range will be communicated again. For example, when a node pushes keys to a server, it may pull the same keys from the server. Many machine learning algorithms also iterate on the same training data with keys fixed on each iteration. Then it is desirable for the receiver to cache the keys. So that if the sender will send the same keys again, it only needs to send a signature of this key list.

Thirdly, even if a subset of keys will be sent again, which may due to the user-defined filter, we can still make use of the cached keys of the receiver by padding 0 in the according value. Then we compress the values. There are several compression algorithms such as Snappy and Zlib which are fast on both compression and decompression, and also efficient to remove 0s.

4.3 Consistent Hashing

The basic idea comes from distributed hash tables [10, 26], where both key-value pairs and server nodes are inserted into the hash ring. Each node manages the key segment starting with its insertion point to the next point by server nodes, as shown in Figure 6.

In the example shown in Figure 6, the server nodes manage segments of the same color. Different to performing key discovery and routing as [18], we use a consistent hashing for assignment and we store the mapping from key segments to nodes in a server manager, which backups the data by Paxos [19], as implemented in Zookeeper. Note that, to facilitate load-balancing, a physical server node contains several virtual server nodes, so they are inserted multiple times into the ring.

4.4 Replica and Consistency

![Figure 7: Replica and Consistency](image-url)
• Datatypes are eigen3 native
  • Dense vectors
  • Sparse vectors
• Push(Header flag)
• Pull(Header flag)
  Flag may specify
  • Value or delta update
  • key range
  • recipient (all server, all clients, particular node)

Shared pointer. No copy on queue (by default)!
Consistency models

(a) Sequential

(b) Eventual

(c) Bounded delay

via task processing engine on client/controller
Vector Clocks for Ranges

- Keep track of when we received an update from a client / server.
- For $c$ clients this means $O(c)$ metadata. This is impossible to store per key (Dynamo).
- Very cheap and feasible for ranges.
- When inconsistent ranges, split segments
  
- This is infrequent + defragmentation.
Experiments
• Implementation on Parameter Server

\[
\min_{w \in \mathbb{R}^p} \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle x_i, w \rangle)) + \lambda \|w\|_1
\]

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Consistency</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>System-A</td>
<td>L-BFGS</td>
<td>Sequential</td>
<td>10,000</td>
</tr>
<tr>
<td>System-B</td>
<td>Block PG</td>
<td>Sequential</td>
<td>30,000</td>
</tr>
<tr>
<td>Parameter Server</td>
<td>Block PG</td>
<td>Bounded Delay KKT Filter</td>
<td>300</td>
</tr>
</tbody>
</table>
• System A and B are production systems at a very large internet company ...
We show the convergence results in Figure 8. As can be seen, baseline-B outperforms baseline-A, because block proximal gradient method converges faster than L-BFGS on this dataset. Parameter server further improves baseline-B even by using the same algorithms, because of the relaxed consistency model parameter server adopted. The KKT filter significantly reduced the network traffic. It skipped 93.4% of gradients should be sent, which are shown in Figure 9. The bounded delay consistency allow to start updating the next block without waiting the data communication finished in previous blocks. With $\tau = 4$, it affects the convergence speed little, but further hide the communication cost.

The benefit of relaxed consistency model can be clearer seen in Figure 10, which shows the time decomposition of a worker node. As can be seen, System-A has around 32% idle time, while this number goes to 53% for system-B due to the barrier placed in each block. However, the parameter server reduces this cost under 2%. But also note that parameter server uses more computational time than system-B. The reason are two-fold. On one hand, system-B optimizes its gradient calculating on this dataset by careful data transformation. On the other hand, the asynchronous updates of parameter server needs more iterations to achieve the same objective value as system-B. However, due to the significant gain on reducing communication cost, parameter server reduces the total time into half.

### Related Works

There exist several general purpose distributed machine learning systems. Mahout [6] and MLI [28], based on Spark [30], adopt the iterative MapReduce [15] framework. While Spark is substantially superior to Hadoop MapReduce due to its preservation of state and optimized execution strategy, both of these approaches use a synchronous iterative communication pattern. This makes them vulnerable to nonuniform performance distributions for iterative machine learning algorithms, i.e. machines that might happen to be slow at any given time. To overcome this limitation, distributed GraphLab [22] asynchronously schedules communication using a graph abstraction. It, however, lacks the elastic scalability of the map/reduce-based frameworks, and relies on coarse-grained snapshots for recovery. Moreover, global variables synchronization is not a first-class primitive. Of course, beyond these general frameworks, numerous systems have been developed that target specific applications, such as [3, 14, 25, 23, 29, 12, 16]. We found that many inference problems have a rather restricted structure in terms of their parametrization where considerable gains can be made by exploiting this design. For instance, generalized linear models typically use a single massive parameter vector, or topic models use an array of sparse vectors. In general, many relevant large-scale graphical models consist largely of a small number of plates, thus allowing for a repeated structure of a small number of components which are shared between observations and machines. This offers considerable efficiencies by performing these operations in bulk and by specializing synchronization primitives for the specific datatypes.
Distributed CountMin Sketch

- Clients only act as data preprocessors
- Shard keys over servers for balancing
- Replication between machines on DHT
- Servers perform simple updates

- **15 servers, 40GBit network (dedicated)**
  \[ M[h(k, j), j] \leftarrow M[h(k, j), j] + v \text{ for all } j \in \{1, \ldots d\} \]

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak inserts per second</td>
<td>1.3 billion</td>
</tr>
<tr>
<td>Average inserts per second</td>
<td>1.1 billion</td>
</tr>
<tr>
<td>Peak network bandwidth per machine</td>
<td>4.37 GBit/s</td>
</tr>
<tr>
<td>Time to recover a failed node</td>
<td>0.8 second</td>
</tr>
</tbody>
</table>
• For 1000 iterations do
  • For each document do
    • For each word in the document do
      • Resample topic for the word
      • Update local (document, topic) table
      • Generate local update message
    • Update local table
    • Lock local (word,topic) table
    • Update local (word,topic) table
    • Unlock local (word,topic) table
  • Synchronize local and global tables
4B documents, 1M tokens, 60k cores, 2k topics
Palo Verde, AZ
3 Gigawatt (4 million people)
Largest nuclear reactor in the USA
Palo Verde, AZ
3 Gigawatt (4 million people)
Largest nuclear reactor in the USA

1 machine = 10 cores
1 core = 50 watt
consumption of 3 Megawatt
See our paper at OSDI 2014

parameterserver.org

blog.smola.org @smolix