Asynchronous Parallel Stochastic Gradient Descent

- A Numeric Core for Scalable Distributed Machine Learning Algorithms

J. Keuper and F.-J. Pfreundt

Competence Center High Performance Computing
Fraunhofer ITWM, Kaiserslautern, Germany
Training Machine Learning Models

**Formulation of the training problem:**

- Set of observations (Training Samples) \( X = \{x_0, \ldots, x_m\} \) with \( x_i \in \mathbb{R}^n \)

- [Supervised Learning] Set of (semantic) labels \( Y = \{y_0, \ldots, y_m\}, y_i \in \mathbb{R} \).

- Loss function to determine quality the learned model,
  - Writing \( x_j(w) \) or \((x_j, y_j)(w)\) for the loss of given samples
  - and model state \( w \)

- → Optimization problem, minimizing the loss
  - Straight forward gradient descent optimization
  - Pitfalls: sparse, high dimensional target space \( \rightarrow \) Overfitting problem
Optimization Algorithms for ML

**Simple Method BATCH-Optimization**

- Run over ALL samples
- Compute average gradient of loss-function
- Make an update step in gradient direction

```
Algorithm 1 BATCH optimization with samples \( X = \{x_0, \ldots, x_m\} \), iterations \( T \) and states \( w \)
1: for all \( t = 0 \ldots T \) do
2: \hspace{1em} Init \( w_{t+1} = 0 \)
3: \hspace{1em} update \( w_{t+1} = w_t - \epsilon \sum_{(x_j \in X)} \partial w x_j (w_t) \)
4: \hspace{1em} w_{t+1} = w_{t+1}/|X|
```

- Computationally expensive
- Scales very poor in the number of samples
Optimization Algorithms for ML

**Simple Method BATCH-Optimization**

- Run over ALL samples
- Compute average gradient of loss-function
- Make an update step in gradient direction

```
Algorithm 1 BATCH optimization with samples \( X = \{x_0, \ldots, x_m\} \), iterations \( T \) and states \( w \)
1: for all \( t = 0 \ldots T \) do
2: \quad Init \( w_{t+1} = 0 \)
3: \quad update \( w_{t+1} = w_t - \epsilon \sum_{(x_j \in X)} \partial_w x_j(w_t) \)
4: \quad w_{t+1} = w_{t+1}/|X|
```

- Computationally expensive
- Scales very poor in the number of samples

**Stochastic Gradient Descent**

- Online algorithm
- Randomized
- Update after EACH sample

```
Algorithm 2 SGD with samples \( X = \{x_0, \ldots, x_m\} \), iterations \( T \), steps size \( \epsilon \) and states \( w \)
Require: \( \epsilon > 0 \)
1: for all \( t = 0 \ldots T \) do
2: \quad draw \( j \in \{1 \ldots m\} \) uniformly at random
3: \quad update \( w_{t+1} = w_t - \epsilon \partial_w x_j(w_t) \)
4: return \( w_T \)
```
Optimization Algorithms for ML

**Simple Method BATCH-Optimization**

- Run over ALL samples
- Compute average gradient of loss-function
- Make an update step in gradient direction

```
Algorithm 1 BATCH optimization with samples \( X = \{x_0, \ldots, x_m\} \), iterations \( T \) and states \( w \)

1: for all \( t = 0 \ldots T \) do
2: \( \text{Init} \ w_{t+1} = 0 \)
3: \( \text{update} \ w_{t+1} = w_t - \epsilon \sum_{(x_j \in X)} \partial_w x_j(w_t) \)
4: \( w_{t+1} = w_{t+1}/|X| \)
```

- Computationally expensive
- Scales very poor in the number of samples

**Stochastic Gradient Descent**

- Online algorithm
- Randomized
- Update after EACH sample

```
Algorithm 2 SGD with samples \( X = \{x_0, \ldots, x_m\} \), iterations \( T \), steps size \( \epsilon \) and states \( w \)

Require: \( \epsilon > 0 \)
1: for all \( t = 0 \ldots T \) do
2: \( \text{draw} \ j \in \{1 \ldots m\} \text{ uniformly at random} \)
3: \( \text{update} \ w_{t+1} \leftarrow w_t - \epsilon \partial_w x_j(w_t) \)
4: \( \text{return} \ w_T \)
```

- Much faster
- Better ML results
- But: intrinsically sequential!
Distributed Optimization Algorithms for ML

Map-reduce scheme possible for basically all ML algorithms:

Distributed Optimization Algorithms for ML

Map-reduce scheme possible for basically all ML algorithms:


BUT

Map-reduce only works for BATCH-solvers
Parallel Optimization Algorithms for ML

Hogwild!


- Parallel SGD for shared memory systems
- Basic Idea:
  - Write asynchronous updates to shared memory (without any save-guards)
  - NO mutual exclusion / locking what so ever → Data races + race conditions
  - NO theoretical guarantees on converges
  - Only constraint: sparsity (time and/or space) to reduce probability of races

- BUT:
  - Works very well in practice → fast, stable, ...

- Why does it work? → Robust nature of ML algorithms + Cache hierarchy
Parallel SGD on Distributed Systems


• Prove of convergence for distributed SGD with on one final Reduce step.

→ Synchronization at the very end

• Only condition: constant step size

\[
\text{Algorithm 3 SimuParallel\text{SGD} with samples } X = \{x_0, \ldots, x_m\}, \text{ iterations } T, \text{ steps size } \epsilon, \text{ number of threads } n \text{ and states } u
\]

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Require: } \epsilon > 0, n > 1
\State 1: define \( H = \left[ \frac{n}{m} \right] \)
\State 2: randomly partition \( X \), giving \( H \) samples to each node
\State 3: for all \( i \in \{1, \ldots, n\} \) \text{ parallel } do
\State 4: randomly shuffle samples on node \( i \)
\State 5: init \( w^0_i = 0 \)
\State 6: for all \( t = 0 \ldots T \) do
\State 7: get the \( i \)th sample on the \( i \)th node an compute
\State 8: update \( w^{t+1}_i = w^t_i - \epsilon \Delta_t(w^t_i) \)
\State 9: aggregate \( v = \frac{1}{n} \sum_{i=1}^{n} w^t_i \)
\State 10: return \( v \)
\end{algorithmic}
\end{algorithm}
Distributed Asynchronous Stochastic Gradient Descent

**ASGD has become very hot topic – especially in the light of Deep Learning (DL)**

- DL currently mostly on GPUs (Hogwild SGD + derivates)
- Recent cluster based DL optimizations by Google, Microsoft, ...
  - Hogwild scheme with parameter servers and message passing updates
Distributed Asynchronous Stochastic Gradient Descent

**ASGD has become very hot topic - especially in the light of Deep Learning (DL)**

- DL currently mostly on GPUs (Hogwild SGD + derivates)

- Recent cluster based DL optimizations by Google, Microsoft, ...
  - Hogwild scheme with parameter servers and message passing updates

**Our ASGD approach (in an nutshell)**

- Asynchronous *communication* via RDMA
  - Partitioned Global Address Space model
  - Using the GASPI protocol implemented by our GPI2.0
  - Host to Host communication - NO central parameter server

- Hogwild like update scheme, several extensions
  - Mini-BATCH updates
  - Update *states – not gradients*
  - Parzen-Window function selecting external updates
Asynchronous Communication

synchronous

Asynchronous single-sided
Distributed parallel ASGD: Our Algorithm

- Cluster Setup
  - Nodes with several CPUs

- Each thread operates independently in parallel
Distributed parallel ASGD: Our Algorithm

**Algorithm 5 ASGD** \( (X = \{x_0, \ldots, x_m\}, T, \epsilon, w_0, b) \)

**Require:** \( \epsilon > 0, n > 1 \)

1: define \( H = \lfloor \frac{m}{n} \rfloor \)
2: randomly **partition** \( X \), giving \( H \) samples to each node
3: for all \( i \in \{1, \ldots, n\} \) **parallel** do
4: randomly **shuffle** samples on node \( i \)
5: init \( w_0^i = 0 \)
6: for all \( t = 0 \ldots T \) do
7: draw mini-batch \( M \leftarrow b \) samples from \( X \)
8: update \( w_{t+1}^i \leftarrow w_t^i - \epsilon \Delta_M(w_{t+1}^i) \)
9: send \( w_{t+1}^i \) to random node \( \neq i \)
10: return \( w_T^1 \)

\[
\Delta_M(w_{t+1}^i) = \left[ w_t^i - \frac{1}{2} \left( w_t^i + w_t^j \right) \right] \delta(i, j) + \Delta_M(w_{t+1}^i)
\]
\[ \delta(i,j) := \begin{cases} 
1 & \text{if } \|w_t^i - \epsilon \Delta w_t^i) - w_t^j\| < \|w_t^i - w_t^j\|^2 \\
0 & \text{otherwise}
\end{cases} \]

\[ \Delta_M(w_{t+1}^i) = \left[w_t^i - \frac{1}{2} \left(w_t^i + w_t^j\right)\right] \delta(i,j) + \Delta_M(w_{t+1}^i) \]
ASGD: Evaluation

Experimental setup I

- Simple K-Means Algorithm
  - Easy to implement → no hidden optimization possible
  - Widely used
  - Artificial data for the cluster problem easy to produce and to control

- HPC Cluster Setup
  - FDR Infiniband interconnect
  - 16 CPUs / node (Xeon)
  - BeeGFS parallel file system
  - GPI2.0 asynchronous RDMA communication
Results I: Strong Scaling

K-Means: \( n=10, k=10 \)
\(~1\text{TB Train Data}\)
Results I: Convergence

K-Means: \( n=10, k=100 \)

\(~1\text{TB Train Data}\)
Results I: Effect of Asynchronous Communication

Message Impact: error rate for # iterations

Message Impact: time needed to reach error rate
ASGD: Evaluation

Experimental setup II

- Deep Learning: training CNNs with modified CAFFE [https://github.com/BVLC/caffe]
  - New parallel input layer
    - parallel file read via BeeGFS
    - data completely in memory
  - New ASGD Solver
    - GPI communication on SGD update
    - GPI startup extensions to caffe cmd-line tool

- HPC Cluster Setup
  - 40GbE interconnect
  - 8 CPUs / node (Atom) – currently only 1 thread per node
  - BeeGFS parallel file system
  - GPI2.0 asynchronous RDMA communication
Results II: training CNNs

**MNIST benchmark**
60000 train images
10000 test images

**Parameter Space:**
Layers: 10
# Params: ~500K
~4MB
Results II: training CNNs

Parallel data input:
Split data into n parts of size 1/n.
Results II: training CNNs

Parallel data input:
Split data into 1000 parts
Train on 1000 nodes.
Results II: training CNNs

**ImageNet benchmark [ILSVRC2012]**
- ~1.2 M train images
- 100 K test images
- ~20 K classes

**Parameter Space:**
- Layers: 25
- # Params: ~61 M
- ~500 MB

NVIDIA Titan (~2700 cores): ~1s/Iter → ~6 Days till convergence
Xeon (single core): ~20s/Iter
ATOM (single core): ~60s/Iter

OUTLOOK
Discussion, Outlook and Advertisement

- Upcoming ASGD DeepLearning Paper
  - Multi-Threaded Nodes
  - Full evaluation on large scale problems
  - Compare to other approaches (Dogwild, ...)
  - Caffe branch release

- New GPI2.0 Release 1.2
  - supporting GASPI over Ethernet
  - Supporting GPU to GPU RDMA communication

- → www.gpi-site.com