Splash

User-friendly Programming Interface for Parallelizing Stochastic Algorithms

Yuchen Zhang and Michael Jordan

AMP Lab, UC Berkeley
Batch Algorithm v.s. Stochastic Algorithm

Consider minimizing a loss function $L(w) := \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$.
Batch Algorithm v.s. Stochastic Algorithm

Consider minimizing a loss function \( L(w) := \frac{1}{n} \sum_{i=1}^{n} \ell_i(w) \).

**Gradient Descent:** iteratively update

\[
    w_{t+1} = w_t - \eta_t \nabla L(w_t).
\]
Consider minimizing a loss function $L(w) := \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$.

**Gradient Descent:** iteratively update

$$w_{t+1} = w_t - \eta_t \nabla L(w_t).$$

**Pros:** Easy to parallelize (via Spark).

**Cons:** May need hundreds of iterations to converge.
Batch Algorithm v.s. Stochastic Algorithm

Consider minimizing a loss function $L(w) := \frac{1}{n} \sum_{i=1}^{n} \ell_i(w)$.

**Stochastic Gradient Descent (SGD):** randomly draw $\ell_t$, then

$$w_{t+1} = w_t - \eta_t \nabla \ell_t(w_t).$$
Batch Algorithm v.s. Stochastic Algorithm

Consider minimizing a loss function \( L(w) := \frac{1}{n} \sum_{i=1}^{n} \ell_i(w) \).

**Stochastic Gradient Descent (SGD):** randomly draw \( \ell_t \), then

\[
 w_{t+1} = w_t - \eta_t \nabla \ell_t(w_t).
\]

**Pros:** Much faster convergence.

**Cons:** Sequential algorithm, difficult to parallelize.
More Stochastic Algorithms

Convex Optimization

- Adaptive SGD (Duchi et al.)
- Stochastic Average Gradient Method (Schmidt et al.)
- Stochastic Dual Coordinate Ascent (Shalev-Shwartz and Zhang)
More Stochastic Algorithms

Convex Optimization
- Adaptive SGD (Duchi et al.)
- Stochastic Average Gradient Method (Schmidt et al.)
- Stochastic Dual Coordinate Ascent (Shalev-Shwartz and Zhang)

Probabilistic Model Inference
- Markov chain Monte Carlo and Gibbs sampling
- Expectation propagation (Minka)
- Stochastic variational inference (Hoffman et al.)
More Stochastic Algorithms

Convex Optimization
- Adaptive SGD (Duchi et al.)
- Stochastic Average Gradient Method (Schmidt et al.)
- Stochastic Dual Coordinate Ascent (Shalev-Shwartz and Zhang)

Probabilistic Model Inference
- Markov chain Monte Carlo and Gibbs sampling
- Expectation propagation (Minka)
- Stochastic variational inference (Hoffman et al.)

SGD variants for
- Matrix factorization
- Learning neural networks
- Learning denoising auto-encoder
More Stochastic Algorithms

Convex Optimization
- Adaptive SGD (Duchi et al.)
- Stochastic Average Gradient Method (Schmidt et al.)
- Stochastic Dual Coordinate Ascent (Shalev-Shwartz and Zhang)

Probabilistic Model Inference
- Markov chain Monte Carlo and Gibbs sampling
- Expectation propagation (Minka)
- Stochastic variational inference (Hoffman et al.)

SGD variants for
- Matrix factorization
- Learning neural networks
- Learning denoising auto-encoder

How to parallelize these algorithms?
First Attempt

After processing a subsequence of random samples...

**Single-thread Algorithm**: incremental update $w \leftarrow w + \Delta$. 
First Attempt

After processing a subsequence of random samples...

**Single-thread Algorithm:** incremental update $w \leftarrow w + \Delta$.

**Parallel Algorithm:**
- Thread 1 (on $1/m$ of samples): $w \leftarrow w + \Delta_1$.
- Thread 2 (on $1/m$ of samples): $w \leftarrow w + \Delta_2$.
- . . .
- Thread $m$ (on $1/m$ of samples): $w \leftarrow w + \Delta_m$. 
First Attempt

After processing a subsequence of random samples...

**Single-thread Algorithm:** incremental update $w \leftarrow w + \Delta$.

**Parallel Algorithm:**
- Thread 1 (on $1/m$ of samples): $w \leftarrow w + \Delta_1$.
- Thread 2 (on $1/m$ of samples): $w \leftarrow w + \Delta_2$.
- ... 
- Thread $m$ (on $1/m$ of samples): $w \leftarrow w + \Delta_m$.

Aggregate parallel updates $w \leftarrow w + \Delta_1 + \cdots + \Delta_m$. 
First Attempt

After processing a subsequence of random samples...

**Single-thread Algorithm:** incremental update $w \leftarrow w + \Delta$.

**Parallel Algorithm:**
- Thread 1 (on $1/m$ of samples): $w \leftarrow w + \Delta_1$.
- Thread 2 (on $1/m$ of samples): $w \leftarrow w + \Delta_2$.
- ... 
- Thread $m$ (on $1/m$ of samples): $w \leftarrow w + \Delta_m$.

Aggregate parallel updates $w \leftarrow w + \Delta_1 + \cdots + \Delta_m$.

Doesn’t work for SGD!
Conflicts in Parallel Updates

**Reason of failure**: $\Delta_1, \ldots, \Delta_m$ simultaneously manipulate the same variable $w$, causing **conflicts** in parallel updates.
Conflicts in Parallel Updates

**Reason of failure:** $\Delta_1, \ldots, \Delta_m$ simultaneously manipulate the same variable $w$, causing conflicts in parallel updates.

**How to resolve conflicts**
Conflicts in Parallel Updates

**Reason of failure:** \( \Delta_1, \ldots, \Delta_m \) simultaneously manipulate the same variable \( w \), causing *conflicts* in parallel updates.

**How to resolve conflicts**

1. Frequent communication between threads:
   - **Pros:** general approach to resolving conflict.
   - **Cons:** inter-node (asynchronous) communication is expensive!
Conflicts in Parallel Updates

**Reason of failure**: $\Delta_1, \ldots, \Delta_m$ simultaneously manipulate the same variable $w$, causing conflicts in parallel updates.

**How to resolve conflicts**

1. **Frequent communication between threads:**
   - **Pros**: general approach to resolving conflict.
   - **Cons**: inter-node (asynchronous) communication is expensive!

2. **Carefully partition the data to avoid threads simultaneously manipulating the same variable:**
   - **Pros**: doesn’t need frequent communication.
   - **Cons**: need problem-specific partitioning schemes; only works for a subset of problems.
**Splash: A Principle Solution**

**Splash** is

- A *programming interface* for developing stochastic algorithms
- An *execution engine* for running stochastic algorithm on distributed systems.
**Splash: A Principle Solution**

**Splash** is

- A **programming interface** for developing stochastic algorithms
- An **execution engine** for running stochastic algorithm on distributed systems.

Features of **Splash** include:

- **Easy Programming**: User develop **single-thread algorithms** via Splash: no communication protocol, no conflict management, no data partitioning, no hyper-parameters tuning.
**Splash: A Principle Solution**

Splash is

- A *programming interface* for developing stochastic algorithms
- An *execution engine* for running stochastic algorithms on distributed systems.

Features of **Splash** include:

- **Easy Programming**: User develop single-thread algorithms via Splash: no communication protocol, no conflict management, no data partitioning, no hyper-parameters tuning.
- **Fast Performance**: Splash adopts novel strategy for automatic parallelization with infrequent communication. Communication is no longer a performance bottleneck.
Splash: A Principle Solution

Splash is

- A programming interface for developing stochastic algorithms
- An execution engine for running stochastic algorithm on distributed systems.

Features of Splash include:

- **Easy Programming**: User develop single-thread algorithms via Splash: no communication protocol, no conflict management, no data partitioning, no hyper-parameters tuning.
- **Fast Performance**: Splash adopts novel strategy for automatic parallelization with infrequent communication. Communication is no longer a performance bottleneck.
- **Integration with Spark**: taking RDD as input and returning RDD as output. Work with KeystoneML, MLlib and other data analysis tools on Spark.
Programming Interface
Programming with Splash

Splash users implement the following function:

```python
def process(sample: Any, weight: Int, var: VariableSet):
    /*implement stochastic algorithm*/
```

where

- **sample** — a random sample from the dataset.
- **weight** — observe the sample duplicated by **weight** times.
- **var** — set of all shared variables.
Example: SGD for Linear Regression

**Goal:** find \( w^* = \arg \min_w \frac{1}{n} \sum_{i=1}^{n} (wx_i - y_i)^2 \).

**SGD update:** randomly draw \((x_i, y_i)\), then \( w \leftarrow w - \eta \nabla_w (wx_i - y_i)^2 \).
Example: SGD for Linear Regression

**Goal:** find $w^* = \arg \min_w \frac{1}{n} \sum_{i=1}^{n} (wx_i - y_i)^2$.

**SGD update:** randomly draw $(x_i, y_i)$, then $w \leftarrow w - \eta \nabla_w (wx_i - y_i)^2$.

**Splash implementation:**

```python
def process(sample: Any, weight: Int, var: VariableSet):
    val stepsize = var.get("eta") * weight
    val gradient = sample.x * (var.get("w") * sample.x - sample.y)
    var.add("w", - stepsize * gradient)
```

**Supported operations:** get, add, multiply, delayedAdd.

AMP Lab
Splash
April 2015 10 / 27
Example: SGD for Linear Regression

**Goal:** find \( w^* = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} (wx_i - y_i)^2 \).

**SGD update:** randomly draw \((x_i, y_i)\), then \( w \leftarrow w - \eta \nabla_w (wx_i - y_i)^2 \).

**Splash implementation:**

```scala
def process(sample: Any, weight: Int, var: VariableSet){
    val stepsize = var.get("eta") * weight
    val gradient = sample.x * (var.get("w") * sample.x - sample.y)
    var.add("w", - stepsize * gradient)
}
```

**Supported operations:** get, add, multiply, delayedAdd.
Get Operations

Get the value of the variable (Double or Array[Double]).

- `get(key)` returns var[key]
- `getArray(key)` returns varArray[key]
- `getArrayElement(key, index)` returns varArray[key][index]
- `getArrayElements(key, indices)` returns varArray[key][indices]

Array-based operations are more efficient than element-wise operations, because the key-value retrieval is executed only once for operating an array.
Add Operations

Add a quantity $\delta$ to the variable.

- **add(key, delta):** $\text{var}[\text{key}] += \text{delta}$
- **addArray(key, deltaArray):** $\text{varArray}[\text{key}] += \text{deltaArray}$
- **addArrayElement(key, index, delta):** $\text{varArray}[\text{key}][\text{index}] += \text{delta}$
- **addArrayElements(key, indices, deltaArrayElements):** $\text{varArray}[\text{key}][\text{indices}] += \text{deltaArrayElements}$
Multiply Operations

Multiply a quantity $\gamma$ to the variable $v$.

- $\text{multiply(key, gamma)}$: var[key] *= gamma
- $\text{multiplyArray(key, gamma)}$: varArray[key] *= gamma

We have optimized the implementation so that the time complexity of $\text{multiplyArray}$ is $O(1)$, independent of the array dimension.

Example: SGD with sparse features and $\ell_2$-norm regularization.

$$w \leftarrow (1 - \lambda) \cdot w \quad \text{(1)} \quad \text{multiply operation}$$

$$w \leftarrow w - \eta \nabla f(w) \quad \text{(2)} \quad \text{addArrayElements operation}$$

Time complexity of (1) = $O(1)$; Time complexity of (2) = $\text{nnz}(\nabla f(w))$. 
Multiply Operations

Multiply a quantity $\gamma$ to the variable $v$.

- multiply(key, gamma): $\text{var}[\text{key}] *= \gamma$
- multiplyArray(key, gamma): $\text{varArray}[\text{key}] *= \gamma$

We have optimized the implementation so that the time complexity of multiplyArray is $O(1)$, independent of the array dimension.

Example: SGD with sparse features and $\ell_2$-norm regularization.

$$w \leftarrow (1 - \lambda) \times w \quad \text{(multiply operation)} \quad (1)$$
$$w \leftarrow w - \eta \nabla f(w) \quad \text{(addArrayElements operation)} \quad (2)$$

Time complexity of (1) = $O(1)$; Time complexity of (2) = $\text{nnz}(\nabla f(w))$. 
Delayed Add Operations

Add a quantity \( \delta \) to the variable \( v \). The operation is not executed until the next time the same sample is processed by the system.

- \( \text{delayedAdd(key, delta)} \): \( \text{var}[\text{key}] \) += delta
- \( \text{delayedAddArray(key, deltaArray)} \): \( \text{varArray}[\text{key}] \) += deltaArray
- \( \text{delayedAddArrayElement(key, index, delta)} \):
  \( \text{varArray}[\text{key}][\text{index}] \) += delta

Example: Collapsed Gibbs Sampling for LDA – update the word-topic counter

\[ n_{wk} \leftarrow n_{wk} + \text{weight} \text{(add operation)} \quad (3) \]
\[ n_{wk} \leftarrow n_{wk} - \text{weight} \text{(delayed add operation)} \quad (4) \]

(3) executed instantly; (4) will be executed at the next time before a new topic is sampled for the same word.
Delayed Add Operations

Add a quantity $\delta$ to the variable $v$. The operation is not executed until the next time the same sample is processed by the system.

- `delayedAdd(key, delta): var[key] += delta`
- `delayedAddArray(key, deltaArray): varArray[key] += deltaArray`
- `delayedAddArrayElement(key, index, delta): varArray[key][index] += delta`

Example: Collapsed Gibbs Sampling for LDA – update the word-topic counter $n_{wk}$ when topic $k$ is assigned to word $w$.

\[
\begin{align*}
  n_{wk} &\leftarrow n_{wk} + weight \quad \text{(add operation)} \quad (3) \\
  n_{wk} &\leftarrow n_{wk} - weight \quad \text{(delayed add operation)} \quad (4)
\end{align*}
\]

(3) executed instantly; (4) will be executed at the next time before a new topic is sampled for the same word.
Running Stochastic Algorithm

Three simple steps:

1. Convert RDD dataset to **Parametrized RDD**:

   ```scala
   val paramRdd = new ParametrizedRDD(rdd)
   ```
Running Stochastic Algorithm

Three simple steps:

1. Convert RDD dataset to **Parametrized RDD**:
   
   ```scala
   val paramRdd = new ParametrizedRDD(rdd)
   ```

2. Set a function that implements the algorithm:
   
   ```scala
   paramRdd.setProcessFunction(process)
   ```
Running Stochastic Algorithm

Three simple steps:

1. Convert RDD dataset to Parametrized RDD:
   ```scala
   val paramRdd = new ParametrizedRDD(rdd)
   ```

2. Set a function that implements the algorithm:
   ```scala
   paramRdd.setProcessFunction(process)
   ```

3. Start running:
   ```scala
   paramRdd.run()
   ```
Execution Engine
How does **Splash** work?

In each iteration, the execution engine does:

1. Propose candidate degrees of parallelism $m_1, \ldots, m_k$ such that $\sum_{i}^{k} m_i = m := \text{(\# of cores)}$. For each $i \in [k]$, collect $m_i$ cores and do:

   - Each core gets a sub-sequence of samples (by default $m_i$ of the full data). They process the samples sequentially using the process function. Every sample is weighted by $m_i$.
   - Combine the updates of all $m_i$ cores to get the global update. There are different strategies for combining different types of updates. For add operations, the updates are averaged.
   - If $k > 1$, then select the best $m_i$ by a parallel cross-validation procedure.
   - Broadcast the best update to all machines to apply this update. Then proceed to the next iteration. (degree of parallelism doesn't decrease)
How does **Splash** work?

In each iteration, the execution engine does:

1. Propose candidate degrees of parallelism $m_1, \ldots, m_k$ such that $\sum_i^k m_i = m := \text{(\# of cores)}$. For each $i \in [k]$, collect $m_i$ cores and do:
   1. Each core gets a sub-sequence of samples (by default $\frac{1}{m}$ of the full data). They process the samples sequentially using the process function. Every sample is weighted by $m_i$. 

2. Combine the updates of all $m_i$ cores to get the global update. There are different strategies for combining different types of updates. For add operations, the updates are averaged.

3. If $k > 1$, then select the best $m_i$ by a parallel cross-validation procedure.

4. Broadcast the best update to all machines to apply this update. Then proceed to the next iteration. (degree of parallelism doesn't decrease)
How does **Splash** work?

In each iteration, the execution engine does:

1. Propose candidate degrees of parallelism $m_1, \ldots, m_k$ such that $\sum_i^k m_i = m := (\# \text{ of cores})$. For each $i \in [k]$, collect $m_i$ cores and do:
   1. Each core gets a sub-sequence of samples (by default $\frac{1}{m}$ of the full data). They process the samples sequentially using the `process` function. Every sample is weighted by $m_i$.
   2. Combine the updates of all $m_i$ cores to get the global update. There are different strategies for combining different types of updates. For add operations, the updates are averaged.

If $k > 1$, then select the best $m_i$ by a parallel cross-validation procedure.

Broadcast the best update to all machines to apply this update. Then proceed to the next iteration. (degree of parallelism doesn’t decrease)
How does **Splash** work?

In each iteration, the execution engine does:

1. Propose candidate degrees of parallelism $m_1, \ldots, m_k$ such that 
   $\sum_i^k m_i = m := (\# \text{ of cores})$. For each $i \in [k]$, collect $m_i$ cores and do:
   1. Each core gets a sub-sequence of samples (by default $\frac{1}{m}$ of the full data). They process the samples sequentially using the process function. Every sample is weighted by $m_i$.
   2. Combine the updates of all $m_i$ cores to get the global update. There are different strategies for combining different types of updates. For add operations, the updates are averaged.

2. If $k > 1$, then select the best $m_i$ by a parallel cross-validation procedure.
How does Splash work?

In each iteration, the execution engine does:

1. Propose candidate degrees of parallelism $m_1, \ldots, m_k$ such that $\sum_i^k m_i = m := (# \text{ of cores})$. For each $i \in [k]$, collect $m_i$ cores and do:
   1. Each core gets a sub-sequence of samples (by default $\frac{1}{m}$ of the full data). They process the samples sequentially using the process function. Every sample is weighted by $m_i$.
   2. Combine the updates of all $m_i$ cores to get the global update. There are different strategies for combining different types of updates. For add operations, the updates are averaged.

2. If $k > 1$, then select the best $m_i$ by a parallel cross-validation procedure.

3. Broadcast the best update to all machines to apply this update. Then proceed to the next iteration. (degree of parallelism doesn’t decrease)
Example: Reweighting for SGD

(a) Optimal solution
(b) Solution with full update
(c) Local solutions with unit-weight update
Example: Reweighting for SGD

(a) Optimal solution
(b) Solution with full update
(c) Local solutions with unit-weight update
(d) Average local solutions in (c)
(e) Aggregate local solutions in (c)

(29,8)
Example: Reweighting for SGD

(a) Optimal solution
(b) Solution with full update
(c) Local solutions with unit-weight update
(d) Average local solutions in (c)
(e) Aggregate local solutions in (c)
(f) Local solutions with weighted update

(29,8)
Example: Reweighting for SGD

(a) Optimal solution
(b) Solution with full update
(c) Local solutions with unit-weight update
(d) Average local solutions in (c)
(e) Aggregate local solutions in (c)
(f) Local solutions with weighted update
(g) Average local solutions in (f)
Experiments
Experiment Setups

- **System**: Amazon EC2 cluster with 8 workers. Each worker has 8 Intel Xeon E5-2665 cores and 30 GBs of memory and was connected to a commodity 1GB network.

- **Algorithms**: SGD for logistic regression; mini-batch SGD for collaborative filtering; Gibbs Sampling for topic modelling.

- **Datasets**:
  - MNIST 8M (LR): 8 million samples, 7,840 parameters.
  - Netflix (CF): 100 million samples, 65 million parameters.
  - NYTimes (LDA): 100 million samples, 200 million parameters.

- **Baseline methods**: single-thread stochastic algorithm; MLlib (the official machine learning library for Spark).
Splash converges to a good solution in a few seconds, while other methods take hundreds of seconds.

Splash is 10x - 25x faster than single-thread SGD.

Splash is 15x - 30x faster than parallelized L-BFGS.
Splash is 36x faster than parallelized Alternating Least Square (ALS).
Splash converges to a better solution than ALS (the problem is non-convex).
Splash is 3x - 6x faster than parallelized Variational Inference (VI).
Splash converges to a better solution than VI.
Waiting time is 16%, 21%, 26% of the computation time.

Communication time is 6%, 39% and 103% of the computation time.
Machine Learning Package
Stochastic Machine Learning Library on Splash

- **Goal:**
  - Ease of use: call with one line of code.
  - Integration: easy to build a data analytics pipeline.

- **Algorithms:**
  - Stochastic gradient descent.
  - Stochastic matrix factorization.
  - Gibbs sampling for LDA.

- Will implement more algorithms in the future...
Summary

- **Splash** is a general-purpose programming interface for developing stochastic algorithms.
- **Splash** is also an execution engine for automatic parallelizing stochastic algorithms.
- Reweighting is the key to achieve fast performance without scarifying communication efficiency.
- We observe good empirical performance and we have theoretical guarantees for SGD.
- **Splash** is online at [http://zhangyuc.github.io/splash/](http://zhangyuc.github.io/splash/).