Tutorial: Harp-DAAL for High Performance Big Data Machine Learning

Framework & Applications
Harp-DAAL Framework

• Concept • Data Types • APIs
Status of Big Data Tools

- Large volume of input data
- Complex analysis process
- For large-scale Machine Learning tasks
  - Big Data tools have limitations in performance
  - HPC techniques can really help (e.g. MPI, BLAS)
  - Can we do better than simply deploy MPI or BLAS?
High Performance – Apache Big Data Stack

MapReduce
Data Centered, QoS

Classic Parallel Runtimes (MPI)
Efficient and Proven techniques

Expand the Applicability of MapReduce to more classes of Applications

Sequential
Input map Output

Map-Only
Input map Output

MapReduce
Input map reduce

Iterative MapReduce
Input map reduce iterations

MPI and Point-to-Point
Pij
Intel® DAAL is an open-source project that provides:

- **Algorithms Kernels to Users**
  - Batch Mode (Single Node)
  - **Distributed Mode (multi nodes)**
  - Streaming Mode (single node)

- **Data Management & APIs to Developers**
  - Data structure, e.g., Table, Map, etc.
  - HPC Kernels and Tools: MKL, TBB, etc.
  - Hardware Support: Compiler
Harp-DAAL is our effort to bridge Data intensive applications (e.g. Machine Learning with Big training data) with HPC platforms.

- **High Level**: Usability oriented, Python Interface, Well documented and packaged modules
- **Middle Level**: Data oriented, HDFS fault-tolerance, Optimized communication pattern
- **Low Level**: Performance oriented: High performance kernels and advanced hardware platform
Harp is an open-source project developed at Indiana University, it has:

- MPI-like **collective communication** operations that are highly optimized for big data problems.
- Harp has efficient and innovative **computation models** for different machine learning problems.
Why Collective Communications for Big Data Processing?

• Collective Communication and Data Abstractions
  o Optimization of global model synchronization
    o ML algorithms: convergence vs. consistency
    o Model updates can be out of order
  o Hierarchical data abstractions and operations

• Map-Collective Programming Model
  o Extended from MapReduce model to support collective communications
  o BSP parallelism at Inter-node vs. Intra-node levels

• Harp Implementation
  o A plug-in to Hadoop
# Collective Communication Operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Algorithm</th>
<th>Time Complexity&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast</td>
<td>chain</td>
<td>$n^\beta$</td>
</tr>
<tr>
<td></td>
<td>minimum spanning tree</td>
<td>$(\log_2 p)n^\beta$</td>
</tr>
<tr>
<td>reduce</td>
<td>minimum spanning tree</td>
<td>$(\log_2 p)n^\beta$</td>
</tr>
<tr>
<td>allgather</td>
<td>bucket</td>
<td>$pn^\beta$</td>
</tr>
<tr>
<td>allreduce</td>
<td>bi-directional exchange</td>
<td>$(\log_2 p)n^\beta$</td>
</tr>
<tr>
<td>regroup</td>
<td>point-to-point</td>
<td>$n^\beta$</td>
</tr>
<tr>
<td>push &amp; pull</td>
<td>point-to-point plus routing optimization</td>
<td>$n^\beta$</td>
</tr>
<tr>
<td>rotate</td>
<td>exchange data between neighbors on a ring topology</td>
<td>$n^\beta$</td>
</tr>
</tbody>
</table>

<sup>a</sup>Note in “time complexity”, $p$ is the number of processes, $n$ is the number of input data items per worker, $\beta$ is the per data item transmission time, communication startup time is neglected and the time complexity of the “point-to-point” based algorithms are estimated regardless of potential network conflicts.
High Performance

Datasets: 5 million points, 10 thousand centroids, 10 feature dimensions

10 to 20 Intel KNL7250 processors

Harp-DAAL has 15x speedups over Spark MLlib

Datasets: 500K or 1 million data points of feature dimension 300

Running on single KNL 7250 (Harp-DAAL) vs. single K80 GPU (PyTorch)

Harp-DAAL achieves 3x to 6x speedups

Datasets: Twitter with 44 million vertices, 2 billion edges

25 Intel Xeon E5 2670

Harp-DAAL has 2x to 5x speedups over state-of-the-art MPI-Fascia solution
Optimization and related issues
- Task level only can't capture the traits of computation
- Model is the key for iterative algorithms. The structure (e.g. vectors, matrix, tree, matrices) and size are critical for performance
- Solver has specific computation and communication pattern

We investigate different computation and communication patterns of important ml algorithms
Parallelization of Machine Learning Applications
Harp Computation Models
Inter-node & Intro-node

(A) Synchronized algorithm
• The latest model

(B) Synchronized algorithm
• The latest model

(C) Synchronized algorithm
• The stale model

(D) Asynchronous algorithm
• The stale model
Computation Model A

- Once a process trains a data item, it locks the related model parameters and prevents other processes from accessing them. When the related model parameters are updated, the process unlocks the parameters. Thus the model parameters used in local computation is always the latest.

Computation Model B

- Each process first takes a part of the shared model and performs training. Afterwards, the model is shifted between processes. Through model rotation, each model parameters are updated by one process at a time so that the model is consistent.

Computation Model C

- Each process first fetches all the model parameters required by local computation. When the local computation is completed, modifications of the local model from all processes are gathered to update the model.

Computation Model D

- Each process independently fetches related model parameters, performs local computation, and returns model modifications. Unlike A, workers are allowed to fetch or update the same model parameters in parallel. In contrast to B and C, there is no synchronization barrier.

Harp Computing Models

Intro-node (Process)
<table>
<thead>
<tr>
<th>(A) Dynamic Scheduler</th>
<th>(B) Static Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>• All computation models can use this</td>
<td>• All computation models can use this</td>
</tr>
<tr>
<td>scheduler.</td>
<td>scheduler.</td>
</tr>
<tr>
<td>• All the inputs are submitted to one</td>
<td>• Each thread has its own input queue</td>
</tr>
<tr>
<td>queue.</td>
<td>and output queue.</td>
</tr>
<tr>
<td>• Threads dynamically fetch inputs</td>
<td>• Each thread can submit inputs to</td>
</tr>
<tr>
<td>from the queue.</td>
<td>another thread.</td>
</tr>
<tr>
<td>• The main thread can retrieve the</td>
<td>• The main thread can retrieve</td>
</tr>
<tr>
<td>outputs from the output queue.</td>
<td>outputs from each task’s output queue.</td>
</tr>
</tbody>
</table>

**Intra-node Schedulers (Thread)**
Schedule Training Data Partitions to Threads

(only Data Partitions in Computation Model A, C, D; Data and/or Model Partitions in B)
Data Conversion

Two ways to store data using Intel® DAAL Java API

- Keep Data on JVM heap
  - no contiguous memory access requirement
  - Small size DirectByteBuffer and parallel copy (OpenMP)
- Keep Data on Native Memory
  - contiguous memory access requirement
  - Large size DirectByteBuffer and bulk copy

Code Optimization Highlights

- Table<Obj>
  - Data on JVM Heap

- NumericTable
  - Data on JVM heap
  - Data on Native Memory

- MicroTable
  - Data on Native Memory
  - A single DirectByteBuffer has a size limits of 2 GB
Data Structures of Harp and Intel® DAAL

Table<Obj> in Harp has a three-level data hierarchy:

- **Table**: consists of partitions
- **Partition**: partition id, container
- **Data container**: wrap up Java objs, primitive arrays

**Data in different partitions, non-contiguous in memory**

Harp Table consists of Partitions:

- **HomogenNumericTable**: data storage at Native side
- **SOANumericTable**: data storage at Java side

**DAAL Table has different types of Data storage**

- **Native Contiguous Memory Space**
- **JVM heap Mem**: non-contiguous

Data in contiguous memory space favors matrix operations with regular memory accesses.
Two Types of Data Conversion

**JavaBulkCopy:**
Dataflow: Harp Table<Obj> -----
Java primitive array ---- DirectByteBuffer ---- NumericTable (DAAL)
Pros: Simplicity in implementation
Cons: high demand of DirectByteBuffer size

**NativeDiscreteCopy:**
Dataflow: Harp Table<Obj> ----
DAAL Java API (SOANumericTable) ---- DirectByteBuffer ---- DAAL native memory
Pros: Efficiency in parallel data copy
Cons: Hard to implement at low-level kernels
Harp-DAAL Framework

- Concept
- Data Types
- APIs
### Data Types

#### Arrays & Objects

<table>
<thead>
<tr>
<th>Primitive Arrays</th>
<th>Serializable Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArray, ShortArray, IntArray, FloatArray, LongArray, DoubleArray</td>
<td>Writable</td>
</tr>
</tbody>
</table>

#### Partitions & Tables

<table>
<thead>
<tr>
<th>Partition</th>
<th>Table</th>
<th>Key-value Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>An array/object with partition ID</td>
<td>The container to organize partitions</td>
<td>Automatic partitioning based on keys</td>
</tr>
</tbody>
</table>
Table<DoubleArray> table = new Table<>(0, new DoubleArrPlus());
for (int i = 0; i < numPartitions; i++) {
    DoubleArray array = DoubleArray.create(size, false);
    table.addPartition(new Partition<>(i, array));
}

• To create a table, an ID and a combiner is required.
• Table ID is user defined. Default ID is allowed.
• Combiner can combine partitions with the same ID in the table.
• A partition contains a partition ID and an object <? extends Simple>. 

Partition & Table
Users are required to implement `getNumWriteBytes`, `write` & `read` methods to direct the serialization / deserialization.

- The `create` method utilizes the pool based management to fetch an cached allocation. After using, the array/object can be released back to the pool or freed to GC.
- The boolean flag is used to indicate if the allocation size can be padded to a size in order to increase the chance for reuse.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected void setup(Context context)</td>
<td>The interface invoked before running the task, used for fetching job</td>
</tr>
<tr>
<td></td>
<td>configurations to the Map task.</td>
</tr>
<tr>
<td>protected void mapCollective(KeyValReader reader, Context context)</td>
<td>The main interface to process key-value pairs</td>
</tr>
<tr>
<td></td>
<td>KeyValReader is used to read all the key-value pairs to the task</td>
</tr>
</tbody>
</table>
Various Table type: Homogeneous, Structure of Array (SOA), etc.

Data store at C++ Side or Java Side

Easy-to-use function to retrieve table data or meta-data

// get a block of rows from daal table
public DoubleBuffer getBlockOfRows(long vectorIndex, long vectorNum, DoubleBuffer buf) {
    return ((HomogenNumericTableImpl)tableImpl).getBlockOfRows(vectorIndex, vectorNum, buf);
}

// get row number of a daal table
public long getNumberOfRows() {
    return tableImpl.getNumberOfRows();
}
Harp-DAAL Framework

- Concept
- Data Types
- APIs
## Harp APIs

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Collective</th>
<th>Event Drive</th>
<th>Algorithm</th>
<th>Data converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>• DynamicScheduler</td>
<td>• broadcast</td>
<td>• getEvent</td>
<td>• TrainingBatch</td>
<td>• HomogenTableHarpMap</td>
</tr>
<tr>
<td>• StaticScheduler</td>
<td>• reduce</td>
<td>• waitEvent</td>
<td>• TrainingDistributed</td>
<td>• HomogenTableHarpTable</td>
</tr>
<tr>
<td></td>
<td>• allgather</td>
<td>• sendEvent</td>
<td>• Prediction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• allreduce</td>
<td></td>
<td>• PredictionDistributed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• regroup</td>
<td></td>
<td>• Input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• pull</td>
<td></td>
<td>• Parameter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• push</td>
<td></td>
<td>• PartialResult</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• rotate</td>
<td></td>
<td>• Result</td>
<td></td>
</tr>
</tbody>
</table>
public <P extends Simple> boolean broadcast(String contextName, String operationName, Table<P> table, int bcastWorkerID, boolean useMSTBcast)

- **Simple** – the interface for `ByteArray`, `ShortArray`, `IntArray`, `FloatArray`, `LongArray` and `DoubleArray` and `Writable`
- **contextName** – user defined name to separate operations indifferent groups
- **operationName** – user defined name to separate operations
- **Table<P> table** – the data structure to broadcast/receive data
- **bcastWorkerID** – the worker to broadcast data
- **useMSTBcast** – default broadcast algorithm is pipelining, set this option to true to enable minimum spanning tree algorithm
public <P extends Simple> boolean reduce(String contextName, String operationName, Table<P> table, int reduceWorkerID)

- **contextName** - user defined name to separate operations indifferent groups
- **operationName** - user defined name to separate operations
- **table** - the data structure to broadcast/receive data
- **reduceWorkerID** - the worker ID to receive the reduced data
public <P extends Simple, PT extends Partitioner> boolean regroup(String contextName, String operationName, Table<P> table, PT partitioner)

- **contextName** – user defined name to separate operations indifferent groups
- **operationName** – user defined name to separate operations
- **table** – the data structure to broadcast/receive data
- **partitioner** – tells which partition to go to which worker for regrouping, e.g. `new Partitioner(numWorkers)`
public <P extends Simple> boolean allgather(String contextName, String operationName, Table<P> table)

- **contextName** – user defined name to separate operations indifferent groups
- **operationName** – user defined name to separate operations
- **table** – the data structure to broadcast/receive data
Normally…

- public <P extends Simple> boolean allreduce(String contextName, String operationName, Table<P> table)

An Alternative Way (if the dataset is large…)

- regroup("main", "regroup", table, new Partitioner(getNumWorkers()));
- allgather("main", "allgather", table);
public <P extends Simple, PT extends Partitioner> boolean push(String contextName, String operationName, Table<P> localTable, Table<P> globalTable, PT partitioner)

- Send the partitions from localTable to globalTable based on the partition ID matching
- **localTable** - contains temporary local partitions
- **globalTable** - is viewed as a distributed dataset where each partition ID is unique across processes
- **partitioner** – if some local partitions is not shown in the globalTable, a partitioner can be used to decide where partitions with this partition ID go

public <P extends Simple> boolean pull(String contextName, String operationName, Table<P> localTable, Table<P> globalTable, boolean useBcast)

- Retrieve the partitions from globalTable to localTable based on partition ID matching
- **useBcast** – if broadcasting is used when a partition is required to send to all the processes.

**Push & Pull**
rotate(String contextName, String operationName, Table<P> globalTable, Int2IntMap rotateMap)

- **contextName** - user defined name to separate operations indifferent groups
- **operationName** - user defined name to separate operations
- **globalTable** -- a distributed dataset where each partition ID is unique across processes
- **rotateMap** – the mapping between source worker and target worker
public class DynamicScheduler<I, O, T extends Task<I, O>> - class declaration

public DynamicScheduler(List<T> tasks) - constructor

public synchronized void submit(I input) – submit an input

public synchronized void start() – start the threads

public synchronized void pause() – pause the threads after the current inputs are processed

public synchronized void pauseNow() - pause the threads immediately

public synchronized void cleanInputQueue() – clean the input queue when the execution is paused or stopped

public synchronized void stop() – join the threads

public synchronized boolean hasOutput() – check if there is an output in the output queue

public synchronized O waitForOutput() – fetch an output from the output queue
public class StaticScheduler<I, O, T extends Task<I, O>> - class declaration

public StaticScheduler(List<T> tasks) - constructor

public synchronized void submit(int taskID, I input) – submit an input to a task

public synchronized void start() – start threads

public synchronized void pause() – pause the threads

public synchronized void cleanInputQueue() – clean the input queue

public synchronized void stop() – stop the threads

public O waitForOutput(int taskID) – wait for the output from a task

public boolean hasOutput(int taskID) – check if a task has an output

StaticScheduler
Public **TrainingBatch**(DaalContext context, Class<? extends Number> cls, TrainingMethod method)

Public **TrainingDistributed**(DaalContext context, Class<? extends Number> cls, TrainingMethod method)

public **Prediction**(DaalContext context, Class<? extends Number> cls, PredictMethod method)

Public **input.set** (InputId, InputData) – set up input data tagged with input ID

Public **parameter.set** (paraValue) – set up parameter values for algorithm

Public **compute**() – computation of an algorithm

Public **communicate**() -- inter-mapper communication

Public **Result.get** (ResultId) – fetch result after computation

Algorithm
Hands-on

• Examples • Python
Explore Algorithms

Single Value Decomposition
PCA

Alternating least squares (ALS)

Latent Dirichlet Allocation

MatrixFactorization-SGD

SubGraph Counting
<table>
<thead>
<tr>
<th>Learning Tasks</th>
<th>Algorithms</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>Naive Bayes, Neural Network</td>
<td>Naive Bayes is not iterative</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iterative algorithm</td>
</tr>
<tr>
<td>Regression</td>
<td>Linear Regression, Ridge Regression</td>
<td>All solved by normal equation, not iterative algorithms</td>
</tr>
<tr>
<td>Clustering</td>
<td>K-means</td>
<td>Iterative algorithm</td>
</tr>
<tr>
<td>Recommender System</td>
<td>Matrix Factorization(SGD), ALS</td>
<td>Iterative algorithm</td>
</tr>
<tr>
<td>Dimension Deduction</td>
<td>SVD, PCA, QR</td>
<td>Linear algebra kernel</td>
</tr>
<tr>
<td>Statistics</td>
<td>Moments, Covariance</td>
<td>Statistics</td>
</tr>
</tbody>
</table>
Solutions to Big Data Problems
Example: K-means Clustering

The Allreduce Computation Model

Worker → Model → Worker

When the model size is small
When the model size is large but can still be held in each machine’s memory
When the model size cannot be held in each machine’s memory

- broadcast
- allreduce
- reduce
- regroup
- allgather
- push & pull
- rotate
Computation models for K-means

Harp-DAAL-Kmeans

- Inter-node: Allreduce, Easy to implement, efficient when model data is not large

- Intra-node: Shared Memory, matrix-matrix operations, xGemm: aggregate vector-vector distance computation into matrix-matrix multiplication, higher computation intensity (BLAS-3)

1: procedure
2: Given \((x^1, x^2, \ldots, x^m), \forall i, x^i \in \mathbb{R}^n\)
3: Initialize centroids randomly: \(\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n\)
4: Repeat until convergence
5: \(\forall i, c^i := \text{argmin}_j \|x^i - \mu_j\|^2\)
6: \(\forall j, \mu_j := \frac{\sum_{i=1}^{m} 1\{c^i = j\} x^i}{\sum_{i=1}^{m} 1\{c^i = j\}}\)
7: End Repeat
8: end procedure

\[
C \leftarrow \alpha \text{op}(A) \text{op}(B) + \beta C
\]
Computation models for MF-SGD

- Inter-node: Rotation
- Intra-node: Asynchronous

Rotation: Efficient when the mode data is large, good scalability

Asynchronous: Random access to model data Good for thread-level workload balance.

\[ R \in \mathbb{R}^{m \times n}, P \in \mathbb{R}^{k \times m}, \text{and } Q \in \mathbb{R}^{k \times n} \]

while true do
  select randomly a point \( r_{ij} \) from \( I \)
  \[ e_{ij} = r_{ij} - p_i^T q_j \]
  \[ p_i \leftarrow p_i + \gamma (e_{ij} q_j - \lambda p_i) \]
  \[ q_j \leftarrow q_j + \gamma (e_{ij} p_i - \lambda q_j) \]
  if \( P, Q \) converged then
    Exit While loop
  end if
end while
end procedure
Computation Models for ALS

- Inter-node: Allreduce

\[ C \leftarrow \alpha A A^T + \beta C \]
\[ A \leftarrow \alpha x x^T + A \]

Intra-node: Shared Memory, Matrix operations
\text{xSyrk}: symmetric rank-k update

```
procedure
  Load \( R, R^T \)
  Initialize \( X, Y \)
  repeat
    for \( i = 1, 2, \ldots, n \) do
      \[ V_i = Y_i, R^T(i, I_i) \]
      \[ A_i = Y_i, Y_i^T + \lambda n_{x_i} E \]
      \[ x_i = A_i^{-1} V_i \]
    end for
    for \( j = 1, 2, \ldots, m \) do
      \[ U_j = X_{I_j}, R(I_j, j) \]
      \[ B_j = X_{I_j} X_{I_j}^T + \lambda n_{m_j} E \]
      \[ y_j = B_j^{-1} U_j \]
    end for
  until convergence
end procedure
```
Com-Friendster
- 65 million nodes
- 5 billion edges

Count number of template u12-2
4.358118772743884×10^{37}

Count time of template u12-2
15 minutes

SNAP network repository
http://snap.stanford.edu/netinf/
1. **Data**
   31G graph data in adjacent list format

2. **Process templates**
   Recursively decompose tree-like templates

3. **Counting**
   Color Coding
   Dynamic programming

4. **Evaluate**
   Accuracy
Computation models for Subgraph Counting

- On-demand decoupled steps
- Pipelined by computation
- Each step partial collective communication within group

**Algorithm 1** The sequential color coding algorithm.

1. **Input:** Graph $G = (V, E)$, a template $T = (V_T, E_T)$, and parameters $\epsilon, \delta$.
2. **Output:** A $(1 \pm \epsilon)$-approximation to $\eta(T, G)$ with probability at least $1 - \delta$.
3. $N = O\left(\frac{\epsilon^3 \log(1/\delta)}{\epsilon^2}\right)$
4. for $j = 1$ to $N$ do
5. For each $v \in V_G$, pick a color $c(v) \in S = \{1, \ldots, k\}$ uniformly at random, where $k = |V_T|$.
6. Pick a root $\rho(T)$ for $T$ arbitrarily.
7. Partition $T$ into subtrees recursively to form $T$.
8. For each $v \in V, T_i \in T$ with root $\rho_i = \rho(T_i)$, and subset $S_i \subseteq S$, with $|S_i| = |T_i|$, we compute:

$$c(v, T_i, S_i) = \sum_u \sum_{v} c(v, T'_i, S'_i) \cdot c(u, T''_i, S''_i),$$

(1)

where $T'_i$ is partitioned into trees $T'_i$ and $T''_i$ in $T$.
9. Compute $C^{(j)}$, the number of colorful embeddings of $T$ in $G$ for the $j$th coloring as

$$C^{(j)} = \frac{1}{q} \frac{k^5}{k!} \sum_{v \in V_G} c(v, T(\rho), S),$$

(2)

where $q$ denotes the number of vertices $\rho' \in V_T$ such that $T$ is isomorphic to itself when $\rho$ is mapped to $\rho'$.
10. **end for**
11. Partition the $N$ estimates $C^{(1)}, \ldots, C^{(N)}$ into $t = O(\log(1/\delta))$ sets of equal size. Let $Z_j$ be the average of set $j$. Output the median of $Z_1, \ldots, Z_t$. 

(a) Step 1  
(b) Step 2  
(c) Step 3  
(d) Step 4
Performance

- Datasets: Twitter with 44 million vertices, 2 billion edges
- 25 Intel® Xeon E5 2670
- Harp-DAAL has 2x to 5x speedups over state-of-the-art MPI-Fascia solution
Hands On

Algorithms

K-means

Naive Bayes

MF-SGD

Python Interface to Harp and Harp-Daal

Harp-DAAL Programming Interface with Java
<table>
<thead>
<tr>
<th>Learning Tasks</th>
<th>Algorithms</th>
<th>Dataset</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering</td>
<td>K-means</td>
<td>15scene</td>
<td>Fifteen Scene Categories</td>
</tr>
<tr>
<td>Regression</td>
<td>Linear Regression</td>
<td>Ifw</td>
<td>Labeled Faces in the Wild (LFW) people</td>
</tr>
<tr>
<td>Classification</td>
<td>Naive Bayes</td>
<td>20news</td>
<td>20 newsgroups</td>
</tr>
<tr>
<td></td>
<td>Neural Network</td>
<td>mnist8m</td>
<td>handwritten digits</td>
</tr>
<tr>
<td>Recommender System</td>
<td>Matrix Factorization (SGD)</td>
<td>movielens</td>
<td>Rating data sets from the MovieLens web site</td>
</tr>
</tbody>
</table>
General Machine Learning Pipeline

1. Define the problem, (binary or multiclass, classification or regression, evaluation metric...)
2. Dataset preparation, (data collection, data munging, cleaning, split, normalization, ...)
3. Feature engineering, (feature selection, dimension reduction, ...)
4. Training: select model and hyper-parameter tuning
5. Output the best models with optimized hyper-parameters
Problem

It's non-trivial to investigate the large scale image collections which contain billions of images.

Image clustering algorithms, such as K-Means, face performance and scalability issues when running on large datasets.
This is a dataset that consists of fifteen natural scene categories that expands on the thirteen category dataset released by Fei Fei Li. The two new categories are industrial and store.

Use the label as ground truth to evaluate the performance of K-Means.
Solution

1. **Data**
   - 15Scene, 4485 images, 15 categories

2. **Features**
   - Extract 1000 sbow features with the imagenet feature extraction tool ILSVRC2010_devkit-1.0.

3. **Train**
   - K-Means, euclidean distance on sbow features, K=15

4. **Evaluate**
   - Cluster quality metrics, such as homo(homogeneity score), ARI(adjusted Rand index) etc.
K-means clustering on the image dataset (PCA-reduced data)
Centroids are marked with white cross

Result

result: sample images of the result clusters, one row for each cluster
• Identification of **Apache Big Data Software Stack** and integration with **High Performance Computing Stack** to give **HPC-ABDS**
  - ABDS (Many Big Data applications/algorithms need HPC for performance)
  - HPC (needs software model productivity/sustainability)

• Identification of **4 computation models** for machine learning applications
  - Locking, Rotation, Allreduce, Asynchronous

• **HPC-ABDS**: High performance **Hadoop** (with **Harp-DAAL** enhancement) on Intel® Xeon an Xeon Phi™ architectures
Source codes became available on Github in February, 2017.

- Harp-DAAL follows the same standard of DAAL's original codes

- Twelve Applications
  - Harp-DAAL Kmeans
  - Harp-DAAL MF-SGD
  - Harp-DAAL MF-ALS
  - Harp-DAAL SVD
  - Harp-DAAL PCA
  - Harp-DAAL Neural Networks
  - Harp-DAAL Naïve Bayes
  - Harp-DAAL Linear Regression
  - Harp-DAAL Ridge Regression
  - Harp-DAAL QR Decomposition
  - Harp-DAAL Low Order Moments
  - Harp-DAAL Covariance
Welcome to HPCDC Tutorial at SC 2017
Interactive Website (https://dexterrules.github.io/SC-Demo-17/SC-Demo.html)

Harp-DAAL

is a high performance framework with the fastest machine learning algorithms on Intel's Xeon and Xeon Phi architectures.

Collaborators

See how it works  Performance  Explore algorithms  Hands on