Machine Learning with HPC: Optimizing for Big Data, Accuracy, and Response Time

Ole J. Mengshoel, Aniruddha Basak, and Tong Yu, ECE Department, CMU

Intel® HPC Developer Conference 2017
High-Performance Computing (HPC) Trends

- CPUs are continuing to evolve, in particular multi-core processors are becoming the norm.
  - Intel Xeon processors with broadwell architecture
- Graphics processing units (GPUs): Promise to continue to dramatically speed up the performance of many-core processing in the cloud and on the mobile device.
- MapReduce, Hadoop, and Spark: Easy-to-use frameworks and platforms for embarrassingly parallel computation:
  - [http://hadoop.apache.org](http://hadoop.apache.org)
  - [http://spark.apache.org](http://spark.apache.org)
- Speed-up performance of processing in the cloud – integration with analytics, machine learning, and visualization software.
High-Performance Computing in ML

● The ambition and scale of AI and ML projects are dramatically increasing
  ○ HPC is required to meet such AI and ML needs

● HPC use case in this talk:
  ○ Large Scale Graph Mining for Anomaly Detection using Commute time distances
    ■ Large matrix computations

● Dimensions in focus in this talk:
  ○ Big Data: we handle graphs with $O(1\text{TB})$ size
  ○ Accuracy: approximation method - small loss in accuracy
  ○ Response Time: a few hours using cluster with 200 computers
Large Scale Graph Mining

Joint work with Kamalika Das

https://www.csee.umbc.edu/~kdas1/
Spectral Graph Mining

• Many real world data-mining applications require relational analysis:
  – Spectral clustering
  – Graph anomaly detection
  – Creating low dimensional embedding of a graph

• A variety of graph analysis hinges on:
  – Spectral decomposition of a graph, or
  – Random walk-based distance measure
Spectral Graph Mining: Examples

Commute-Time Distance: A Graph Metric

- Commute time distance between two vertices $s$ and $t$ is the expected time taken by the natural random walk starting in vertex $s$ to travel to vertex $t$ and back to $s$.
- The commute-time distance metric captures the structural properties of a graph.
Commute Time Distance for Large Graphs

- Laplacian, Random Projection
- Euclidian distances

Graph Adjacency Matrix → Approximate Eigen-Embedding → Commute Time distances

• Exact computation takes $O(n^3)$ where $n$ is number of vertices in the graph
• For large $n$, approximating is the traditional approach
Approximation of Commute Time Distance (CTD)

• For each random vector, solve a Symmetric Diagonally Dominant (SDD) system involving the graph laplacian $L$
• Solving SDD systems for large graphs is compute intensive
  ○ Hard to tackle general graphs with different topologies, in particular dense graphs

Possible remedy:

• Near-linear time approximate SDD solvers*
• Only centralized implementations exist
  • What to do for large dense graphs that do not fit in memory?

Our SparkCTD: CTD for Large Dense Graphs

• Distributed SDD solver using matrix chain product
  – Apache Spark for distributed operation
  – Utilizes matrix inverse identity
    \[(I - A)^{-1} = 1 + A + A^2 + \cdots = \prod_{k \geq 0} (I + A^{2^k}) = \prod_{k=0}^{a} (I + A^{2^k}) + \epsilon\]

• Distributed implementation of commute-time distances using only one chain computation,
  – Chain length: logarithmic in the number of terms in the sum \(1 + A + A^2 + \cdots + A^n\)

• Other optimizations in pre-processing

SparkCAD: Our Distributed CAD Variant

Distributed CAD (SparkCAD)

Distributed CTD (SparkCTD)

Distributed SDD Solver

Matrix Chain Multiplication

Matrix Multiplication
Distributed Block Matrix Multiplication

\[ C = A \times B \]

- BlockMatrix is a Spark RDD: \(((\text{row}_\text{id}, \text{col}_\text{id}), M)\)
- Computing each product block in \( C \) requires multiple blocks from \( A \) and \( B \)

\[ C[i][j] = \text{sum}(A[i][k] \times B[k][j]) \text{ for } k = 0 \ldots n \]
Distributed Matrix Multiplication: Spark Implementation

- Spark manages memory and cores
  - Several distributed operations are lazy
- Block matrix multiplication: heavy mixing of blocks between $A$ and $B$
- Spark’s BlockMatrix-multiply attempts complete parallel computation
  - Many copies of $A$ and $B$ created
  - Huge shuffle of data: memory and disk intensive

Proposed approach:

- Instead of shuffling data across workers, write $A$ and $B$ to distributed file system
- Compute each block of $C$ by reading all the necessary blocks
Pleides Supercomputing Center*

- Intel Xeon processors: E5-2680v4 (Broadwell), E5-2680v3 (Haswell), E5-2680v2 (Ivy Bridge), and E5-2670 (Sandy Bridge)
- We use Broadwell nodes: 28 cores, 128GB
- Distributed storage: "nobackup", total of 40 petabytes (PB) of disk space, serving thousands of cores
  - Lustre-based filesystems managed under Lustre software version 2.x.

*https://www.nas.nasa.gov/hecc/resources/pleiades.html
Spark Architecture: Pleiades and NAS
Our Matrix Multiplication via Spark: 1\textsuperscript{st} Step

- Write $A$ and $B$ matrices to nobackup
- Each block is written as a mat file with block-id as filename

![Diagram showing the process of writing matrices $A$ and $B$ to nodackup](image-url)
Our Matrix Multiplication via Spark: 2\textsuperscript{nd} Step

Compute each block of the product

- Read one row from $A$ and one column from $B$
- Compute dot product

Read $A_{0,0}, A_{0,1}, \ldots, A_{0,p}$

Read $B_{0,0}, B_{1,0}, \ldots, B_{p,0}$
Tuning IO Performance: Lustre File System (LFS)

- Pleiades compute environment share several “nobackup” filesystems
- The nobackup filesystems are Lustre-based filesystems managed under Lustre software version 2.x.
- We tune ‘Stripe count’ to optimize parallel read-write performance
  - Matrix blocks are read and written in parallel
- Intel Lustre Software will be very useful in improving performance
From Multiplication to SparkCAD

- Chain product computation: series of distributed multiplications
  \[(I - A)^{-1} = 1 + A + A^2 + \cdots = \prod_{k \geq 0} (I + A^{2^k}) = \prod_{k=0}^{a} (I + A^{2^k}) + \epsilon\]

- Commute-time distances: repeated use of the computed chain product

- Anomaly detection (SparkCAD):
  - Compute the pairwise distances for both graphs
  - Take difference and sort edges by anomaly score
Experimental Setup

• Two different datasets
  a. *Synthetic*: Fully connected graphs from random data
  b. *Stack Overflow*: Sparse graphs from Stack Overflow data

• The same SparkCAD algorithm and implementation works for both sparse and dense graphs
  a. Difference from previous work’s focus on sparse case

• For sparse graphs, we use larger blocks
Accuracy of Distributed CAD

- CAD accuracy: depends on accuracy of commute-time distances
  - Comparison on a small synthetic graph
  - Ground truth: eigen-decomposition, Matgraph
  - Approximate commute-time distances
  - SparkCTD: our implementation for distributed commute-time distances

<table>
<thead>
<tr>
<th></th>
<th>Ground Truth</th>
<th>Koutis Code, SPL</th>
<th>SparkCTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Relative Error</td>
<td>0.0%</td>
<td>3.16%</td>
<td>3.72%</td>
</tr>
</tbody>
</table>

1. Ed Scheinerman  
2. Yiannis Koutis  
Scalability with Increasing Graph Size

- Graph type: synthetic, fully connected
- For a fixed compute cluster and block-size, we increase the size of the graph
- Number of tasks increases quadratically
Scalability with Increasing Compute Cluster-Size

- Graph type: synthetic, fully connected
- After the number of executors (available cores) exceed number of parallel tasks, no significant improvement is observed

- Matrix size = 100K, block size = 5K
- Parallel tasks = 400
- Executors/worker = 12
- 34 workers, thus 408 executors
Distributed CAD: Stack Overflow Network

• Dataset: From Stack Overflow Web message board
  – See https://stackoverflow.com/
  – https://snap.stanford.edu/data/sx-stackoverflow.html

• Experimental Setup
  – Edge weight $A(u,v) = \text{all interactions between users } u \text{ and } v$
  – Undirected graph: $\tilde{A}(u,v) = A(u,v) + A(v,u)$

• Problem: Find anomalous users based on their interactions with other users

• Size: 2.6 Million users, O(10 Million) edges
Stack Overflow Anomaly: Bridge Node Jason

Gordon Linoff: 516,248
pskink: 12,654
Sahil Mittal: 17,812
Stack Overflow Anomaly: Local Change
Node Stephen

- No activity from Stephen

James McNellis: 253,419 (Microsoft)

mousey: 3,516 (google)

neuromancer: 13,480

Carnegie Mellon University Silicon Valley
Conclusions

- Commute-time based Anomaly Detection (CAD): algorithm for commute-time distance - captures the structural properties of a graph
  - Unfortunately, cannot handle massive networks
- Distributed CAD, SparkCAD: handles massive networks via Spark
- Demonstration on network generated from Stack Overflow data:
  - Big data, with some loss of accuracy, within a few hours of computation time
Backup Slides
Approximate Commute-time Distances

Baseline algorithm:
• Uses near-linear time SDD solver
• Requires the matrices to fit in memory

\begin{align*}
\text{Input} & : G, E, \epsilon_{RP} \\
\text{Output:} & \epsilon_{RP} \text{ close approximations, } d_{ij}, \text{ of commute time distances between } (i, j) \text{ for all } (i, j) \in E \\
1 & L = D^G - G \\
2 & \text{Compute edge-vertex incidence matrices } B \text{ and } W \text{ from } G \\
3 & \text{Initialize: } C = \{d_{ij} \leftarrow 0 | (i, j) \in E\} \\
4 & n \leftarrow \text{Number of nodes in } G \\
5 & k_{RP} = \lceil \log(n/\epsilon_{RP}) \rceil \\
6 & \text{For } j = 1 \text{ to } k_{RP} \\
7 & \text{Create random vector } q \\
8 & y = W^{1/2} Bq \\
9 & \text{Solve for } z \text{ in } Lz = y \\
10 & \forall (i, j) \in E, \text{ do:} \\
11 & d_{ij} \leftarrow d_{ij} + (z_i - z_j)^2 \\
12 & \text{return } C
\end{align*}

QoS-Aware Scheduling for Inference in Deep Neural Networks

Joint work with Zhou Fang and Rajesh K. Gupta

QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks. In Proceedings of the 26th ACM International Conference on Information and Knowledge Management (CIKM), 2017
Background

- DNN has been widely applied in different areas, such as computer vision and natural language processing.
- High overhead of computation and memory makes DNN deployment on the client-end a challenging task (e.g., limited resource).
- Executing DNNs on cloud-hosted GPU servers provides an attractive alternative.
- This work studies scheduling of DNN inference queries on servers that require sub-second delay.
To train a model achieving high accuracy without severe delay in this real-time system, we propose scheduling mechanism to select the most suitable neural network model and batch size.
Simple Scheduling by Heuristic

The heuristic scheduler always selects the smallest batch size ($B$) that satisfies the requirement on throughput

$$B \geq \left[ \alpha \cdot (D_{B,M,G} \cdot R_t) + (1 - \alpha) \cdot N_t \right] \cdot \beta \cdot \xi_G,$$

where $R_t$ is the query arrival rate at time $t$, and $\xi_G \in (0, 1]$ is the portion of queries allocated to GPU $G$. The term $D_{B,M,G} \cdot R_t$ represents the number of incoming queries and $N_t$ is the total number of buffered queries. $\alpha \in [0, 1]$ and $\beta > 0$ are tuning parameters.
Advanced Scheduling by Deep Reinforcement Learning

An action selects a batch size $B$ and a model type $M$,

The reward is defined as the average quality of the latest $W$ query responses.
Advanced Scheduling by Deep Reinforcement Learning

State:

- **Histogram of query deadlines**: We use a histogram of query deadlines as the feature representing urgency of buffered queries.
- **Arriving rate**: The state contains the query arrival rate \( R_t \) to estimate the number of incoming queries in future.
- **GPU type**: We use a binary vector to represent the GPU type: an element \( e \) of the vector is 1 if the GPU matches the type that \( e \) is associated with, otherwise \( e \) is 0.
- **Availability of GPUs**: We represent availability as the expected rest computing time of each device \( (t_{end} - t_{now}) \), where \( t_{end} \) is estimated using the measured mean computing time.
Fixed scheduling interval exerts high computation overhead.

To resolve this problem, we compute the next action to take using the policy DNN model only when there are free GPUs. It leads to a large reduction of the scheduling overhead when incoming workloads keep all GPUs busy.
Dataset

**SogouQ**: The query stream is generated from a web query trace provided by the SogouQ query log dataset. The queries per second (QPS) over a day is plotted as following.

This work uses YOLO [10], a model for real-time object detection. A query is a YOLO object detection task that uses either the full or the tiny model: the full model achieves a mean average precision (mAP) of 63.4% on the PASCAL VOC dataset [4] and the mAP is 57.1% for the less complex tiny model.
Experiment Setting

Single GPU:
- We measure delay on High Frequency Intel Xeon E5-2686v4 Processors and NVIDIA K80 GPUs.
- The bin size of query buffer is $\Delta t = 200\text{ms}$.
- The arrival rate of queries ($R_t$) is measured in a sliding window of 5s.
- The heuristic schedule uses the parameters $\alpha = 0.5$ and $\kappa = 0.5$ that give the highest mean response quality on the training data.
- The policy model of the RL scheduler has 1 fully connected hidden layer with 64 neurons.
- The reward is measured using the qualities of last $W = 1000$ queries.

Heterogeneous GPU Cluster:
- To evaluate the schedulers for more servers, we use a cluster of 5 K80 GPUs.
- The policy DNN model of the RL agent has the same hidden layer as the model for one GPU.
Evaluation Metrics

1) Quality of a query response, defined as a function of the end time of query processing \( t_{end} \), the deadline \( (t_{ddl}^{s}, t_{ddl}^{h}) \) and the DNN model \( (M) \), is computed as:

\[
Q(t_{end}, t_{ddl}^{s}, t_{ddl}^{h}, M) = \kappa_{M} \cdot V(t_{end}, t_{ddl}^{s}, t_{ddl}^{h}),
\]

where \( \kappa_{M} \in (0, 1] \) is a factor to penalize less accurate models, and \( V \) is a time-utility function (TUF) to represent the influence of delay on response quality.

2) \( P_{tiny} \) : the percentage of queries that use the tiny model and have lower inference accuracy.
Baseline Methods

● A baseline scheduler (denoted as fixed) is used for performance comparison. It always batches all buffered queries to run on a GPU, with 32 as the maximal batch size.

● Simple scheduling by heuristic
Experimental Results: Single GPU

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>12-15hr</th>
<th>15-18hr</th>
<th>18-21hr</th>
<th>21-24hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>20.2</td>
<td>33.8</td>
<td>25.0</td>
<td>25.4</td>
</tr>
<tr>
<td>Heuristic</td>
<td>5.5</td>
<td>16.3</td>
<td>9.2</td>
<td>11.7</td>
</tr>
<tr>
<td>RL</td>
<td>2.0</td>
<td>8.2</td>
<td>4.1</td>
<td>6.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>12-15hr</th>
<th>15-18hr</th>
<th>18-21hr</th>
<th>21-24hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>0.883 (0.204)</td>
<td>0.818 (0.235)</td>
<td>0.860 (0.218)</td>
<td>0.863 (0.218)</td>
</tr>
<tr>
<td>Heuristic</td>
<td>0.968 (0.117)</td>
<td>0.915 (0.185)</td>
<td>0.950 (0.146)</td>
<td>0.939 (0.161)</td>
</tr>
<tr>
<td>RL</td>
<td>0.978 (0.084)</td>
<td>0.931 (0.149)</td>
<td>0.961 (0.114)</td>
<td>0.949 (0.132)</td>
</tr>
</tbody>
</table>

The fixed scheduler has the lowest response quality, because it does not adapt the batch size and relies upon using the tiny model to avoid missing deadlines. The heuristic scheduler effectively improves the mean quality, and the RL scheduler achieves better performance.
Experimental Results: CDF

It illustrates that compared with the heuristic approach, the RL approach has larger response delays, whereas it achieves higher quality by using the tiny model for less times (as in Figure 6b).
Experimental Results: Multi-GPUs

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>12-15hr</th>
<th>15-18hr</th>
<th>18-21hr</th>
<th>21-24hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>32.6</td>
<td>43.5</td>
<td>36.5</td>
<td>34.4</td>
</tr>
<tr>
<td>Heuristic</td>
<td>9.6</td>
<td>22.8</td>
<td>14.4</td>
<td>16.4</td>
</tr>
<tr>
<td>RL</td>
<td>8.2</td>
<td>20.6</td>
<td>12.8</td>
<td>14.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>12-15hr</th>
<th>15-18hr</th>
<th>18-21hr</th>
<th>21-24hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>0.820 (0.231)</td>
<td>0.767 (0.242)</td>
<td>0.801 (0.237)</td>
<td>0.813 (0.234)</td>
</tr>
<tr>
<td>Heuristic</td>
<td>0.948 (0.148)</td>
<td>0.883 (0.210)</td>
<td>0.925 (0.176)</td>
<td>0.916 (0.186)</td>
</tr>
<tr>
<td>RL</td>
<td>0.955 (0.139)</td>
<td>0.895 (0.202)</td>
<td>0.933 (0.168)</td>
<td>0.922 (0.179)</td>
</tr>
</tbody>
</table>

We find that similar to the single GPU case, the RL scheduler delivers higher response quality by using the tiny YOLO model less frequently.
Conclusion and Future Work

1. We present a QoS-aware scheduling approach for deep learning applications, by reinforcement learning. It can effectively select the most suitable neural network model and batch size to optimize the accuracy of deep learning model and reduce the response time of the system.
Visualization of Scheduling Actions

The scheduling actions performed by the heuristic and RL schedulers in a duration of 10min are visualized.