MALT: Distributed Data Parallelism for Existing ML Applications

Hao Li*, Asim Kadav, Erik Kruus, Cristian Ungureanu

* University of Maryland-College Park
NEC Laboratories, Princeton
Data, data everywhere...

<table>
<thead>
<tr>
<th>Software</th>
<th>User-generated content</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data generated by</td>
<td>Transactions, website visits, other metadata</td>
<td>facebook, twitter, reviews, emails</td>
</tr>
<tr>
<td>Applications (usually based on ML)</td>
<td>Ad-click/Fraud prediction, Recommendations</td>
<td>Sentiment analysis, Targeted advertising</td>
</tr>
</tbody>
</table>
Timely insights depend on updated models.

**Training**
- Data (such as image, label pairs)
- Usually trained in *real time*
- Expensive to train HD videos

**Surveillance/Safety Driving**
- Usually trained in *real time*
- Expensive to train HD videos

**Advertising (ad prediction, ad bidding)**
- Usually trained *hourly*
- Expensive to train millions of requests

**Knowledge banks (automated answering)**
- Usually trained *daily*
- Expensive to train large corpus

---

*labrador*
Model training challenges

• Large amounts of data to train
  • Explosion in types, speed and scale of data
    • **Types**: Image, time-series, structured, sparse
    • **Speed**: Sensor, Feeds, Financial
    • **Scale**: Amount of data generated growing exponentially
  • Public datasets: Processed splice genomic dataset is 250 GB and data subsampling is unhelpful
  • Private datasets: Google, Baidu perform learning over TBs of data

• Model sizes can be huge
  • Models with billions of parameters do not fit in a single machine
  • E.g.: Image classification, Genome detection

Model accuracy generally improves by using larger models with more data
Properties of ML training workloads

• Fine-grained and Incremental:
  • Small, repeated updates to model vectors

• Asynchronous computation:
  • E.g. Model-model communication, back-propagation

• Approximate output:
  • Stochastic algorithms, exact/strong consistency maybe an overkill

• Need rich developer environment:
  • Require rich set of libraries, tools, graphing abilities
MALT: Machine Learning Toolset

- Run existing ML software in data-parallel fashion

- Efficient shared memory over RDMA writes to communicate model information
  - Communication: Asynchronously push (scatter) model information, gather locally arrived information
  - Network graph: Specify which replicas to send updates
  - Representation: SPARSE/DENSE hints to store model vectors

- MALT integrates with existing C++ and Lua applications
  - Demonstrate fault-tolerance and speedup with SVM, matrix factorization and neural networks
  - Re-uses existing developer tools
Outline

- Introduction
- Background
- MALT Design
- Evaluation
- Conclusion
Distributed Machine Learning

- ML algorithms learn incrementally from data
  - Start with an initial guess of model parameters
  - Compute gradient over a loss fn, and update the model

- Data Parallelism: Train over large data
  - Data split over multiple machines
  - Model replicas train over different parts of data and communicate model information periodically

- Model parallelism: Train over large models
  - Models split over multiple machines
  - A single training iteration spans multiple machines
Stochastic Gradient Descent (SGD)

SGD trains over one (or few) training example at a time

- Every data example processed is an iteration
- Update to the model is gradient
- Number of iterations to compute gradient is batch size
- One pass over the entire data is an epoch
- Acceptable performance over test set after multiple epochs is convergence

Can train wide range of ML methods: k-means, SVM, matrix factorization, neural-networks etc.
Data-parallel SGD: Mini-batching

- Machines train in parallel over a *batch* and exchange model information
  - Iterate over data examples faster (in parallel)
  - May need more passes over data than single SGD (poor convergence)
Approaches to data-parallel SGD

- Hadoop/Map-reduce: A variant of bulk-synchronous parallelism
  - Synchronous averaging of model updates *every epoch* (during reduce)

1) Infrequent communication produces low accuracy models
2) Synchronous training hurts performance due to stragglers
Parameter server

• Central server to merge updates every few iterations
  • Workers send updates asynchronously and receive whole models from the server
  • Central server merges incoming models and returns the latest model
  • Example: Distbelief (NIPS 2012), Parameter Server (OSDI 2014), Project Adam (OSDI 2014)
Peer-to-peer approach (MALT)

- Workers send updates to one another asynchronously
  - Workers communicate every few iterations
  - No separate master/slave code to port applications
  - No central server/manager: simpler fault tolerance
Outline

Introduction

Background

MALT Design

Evaluation

Conclusion
Model replicas train in parallel. Use shared memory to communicate. Distributed file system to load datasets in parallel.
dStorm: **Distributed one-sided remote memory**

- RDMA over infiniband allows high-throughput/low latency networking
  - RDMA over Converged Ethernet (RoCE) support for non-RDMA hardware
- Shared memory abstraction based over RDMA one-sided writes (no reads)

```plaintext
S1.create(size, ALL)  S2.create(size, ALL)  S3.create(size, ALL)
```

**Similar to partitioned global address spaces - Local vs global memory**
scatter() propagates using one-sided RDMA

- Updates propagate based on communication graph

Remote CPU not involved: Writes over RDMA. Per-sender copies do not need to be immediately merged by receiver
`gather()` function merges **locally**

- Takes a user-defined function (UDF) as input such as average

Useful/General abstraction for data-parallel algos: Train and `scatter()` the model vector, `gather()` received updates
VOL: Vector Object Library

• Expose vectors/tensors instead of memory objects

• Provide representation optimizations
  • sparse/dense parameters store as arrays or key-value stores

• Inherits \texttt{scatter()}/\texttt{gather()} calls from dStorm

• Can use vectors/tensors in existing vectors
Propagating updates to everyone

model 1

model 2

model 3

model 4

model 5

model 6
$O(N^2)$ communication rounds for $N$ nodes
In-direct propagation of model updates

Use a uniform random sequence to determine where to send updates to ensure all updates propagate uniformly. Each node sends to fewer than N nodes (such as logN)
O(Nlog(N)) communication rounds for N nodes

- MALT proposes sending models to fewer nodes (log N instead of N)
  - Requires the node graph be connected
  - Use any uniform random sequence

- Reduces processing/network times
  - Network communication time reduces
  - Time to update the model reduces
  - Iteration speed increases but may need more epochs to converge

- Key Idea: Balance communication with computation
  - Send to less/more than log(N) nodes

Trade-off model information recency with savings in network and update processing time
Converting serial algorithms to parallel

Serial SGD

Gradient g;
Parameter w;
for epoch = 1:maxEpochs do
  for i = 1:N do
    g = cal_gradient(data[i]);
    w = w + g;

Data-Parallel SGD

maltGradient g(sparse, ALL);
Parameter w;
for epoch = 1:maxEpochs do
  for i = 1:N/ranks do
    g = cal_gradient(data[i]);
    g.scatter(ALL);
    g.gather(AVG);
    w = w + g;

- `scatter()` performs one-sided RDMA writes to other machines.
- “ALL” signifies communication with all other machines.
- `gather(AVG)` applies average to the received gradients.
- Optional `barrier()` makes the training synchronous.
Consistency guarantees with MALT

• Problem: Asynchronous *scatter/gather* may cause models to diverge significantly

• Problem scenarios:
  • **Torn reads:** Model information may get re-written while being read
  • **Stragglers:** Slow machines send stale model updates
  • **Missed updates:** Sender may overwrite its queue if receiver is slow

• Solution: All incoming model updates carry iteration count in their header and trailer
  • Read header-trailer-header to skip torn reads
  • Slow down current process if the incoming updates are too stale to limit stragglers and missed updates (Bounded Staleness [ATC 2014])
  • Few inconsistencies are OK for model training (Hogwild [NIPS 2011])
    • Use barrier to train in BSP fashion for stricter guarantees
MALT fault tolerance: Remove failed peers

- Each replica has a fault monitor
  - Detects local failures (processor exceptions such as divide-by-zero)
  - Detects failed remote writes (timeouts)

- When failure occurs
  - Locally: Terminate local training, other monitors see failed writes
  - Remote: Communicate with other monitors and create a new group
  - Survivor nodes: Re-register queues, re-assign data, and resume training

- Cannot detect byzantine failures (such as corrupt gradients)
Integrating existing ML applications with MALT

**Support Vector Machines (SVM)**
- Application: Various classification applications
- Existing application: Leon Bottou’s SVM SGD
- Datasets: RCV1, PASCAL suite, splice (700M - 250 GB size, 47K - 16.6M parameters)

**Matrix Factorization (Hogwild)**
- Application: Movie recommendation (Netflix)
- Existing application: HogWild (NIPS 2011)
- Datasets: Netflix (1.6 G size, 14.9M parameters)

**Neural networks (NEC RAPID)**
- Application: Ad-click prediction (KDD 2012)
- Cluster: Eight Intel 2.2 Ghz with 64 GB RAM machines connected with Mellanox 56 Gbps infiniband backplane
Speedup using SVM-SGD with RCV1 dataset

![Graph showing speedup over single SGD for a fixed loss](image)

- **goal**: Loss value as achieved by single rank SGD

- **cb size**: Communication batch size - Data examples processed before model communication

**Figure 5.** This figure shows convergence for RCV1 workload with a single machine workload. We find that this benefit of different synchronization methods. We compare the speedup of the systems under test by running them until they reach the same loss value and compare the total time taken. For each of our experiments, we pick the desired final optimization goal as a percentage of the training data and number of iterations (passes) over data per machine. Distributed training requires fewer iterations per machine and number of iterations (passes) over data per machine compared to single-machine SGD with matrix factorization. We show convergence in the forward and the reverse direction. Hence, fully-connected neural networks have dense parameters and there is computational cost (data communication) before model communication and communicated parameters. SVM uses one-shot averaging at the end of every epoch to communicate parameters (for low-latency communication and communicated parameters). We implement MR-SVM algorithms over the MALT library and run it over our infiniBand cluster. MR-SVM is designed for map-reduce (MR-SVM). MR-SVM runs multiple replicas on a single machine. We are unable to saturate the network and CPU with a single cluster. For RCV1 and other smaller workloads, we find that neural network requires limited computation and communication as opposed to gradient and 2) text processing in a non-convex and requires high-dimensional model communication batch size - Data examples processed before model communication

**Figure 6.** We now compare MALT-SVM performance with an existing algorithm designed for map-reduce (MR-SVM). MR-SVM replica, and run multiple replicas on a single machine. We find that gates the model updates to all other machines. We find that processes 5000 examples from the dataset and then propagates the model updates to all other machines. We find that MALT converges faster since it is designed for low latency frameworks such as MALT. MALT provides 3 speedup for certain datasets. In addition, we find that MALT library provides 3 speedup for certain datasets. We implement MR-SVM algorithms over the MALT library and run it over our infiniBand cluster. MR-SVM runs multiple replicas on a single machine. We are unable to saturate the network and CPU with a single cluster. For RCV1 and other smaller workloads, we find that MALT converges faster since it is designed for low latency frameworks such as MALT. MALT converges faster since it is designed for single rank SGD on the PASCAL alpha dataset. This happens because the averaging effect of gradients provides super-linear speedup for low latency frameworks such as MALT. We implement MR-SVM algorithms over the MALT library and run it over our infiniBand cluster. MR-SVM runs multiple replicas on a single machine. We are unable to saturate the network and CPU with a single cluster. For RCV1 and other smaller workloads, we find that MALT converges faster since it is designed for low latency frameworks such as MALT. MALT converges faster since it is designed for single rank SGD on the PASCAL alpha dataset. This happens because the averaging effect of gradients provides super-linear speedup for low latency frameworks such as MALT.
Speedup using RAPID with KDD 2012 dataset

Figure 6. This figure shows speedup by iterations over single machine workload. We find that

**MALT provides speedup over single process SGD**
Speedup with the Halton scheme

Splice-site, modelavg, cb=5000, ranks=8

![Graph showing loss over time for different methods]

- **Goal**: 0.01245
- **BSP**: Bulk-Synchronous Processing
- **ASYNC**: Asynchronous 6X
- **ASYNC Halton**: Asynchronous with the Halton network 11X

**Indirect propagation of model improves performance**
Data transferred over the network for different designs

Figure 13. This figure shows the data sent by MALT, MALT-Halton, and the parameter server for the webspam workload. MALT sends and receives gradients while parameter server sends gradients but needs to receive whole models. To summarize, we find that MALT provides sending gradients (instead of sending the model) that saves network costs. Furthermore, MALT-Halton is network efficient and achieves speedup over MALT-all.

Network saturation tests: We perform infiniBand network throughput tests, and measure the time to scatter updates in MALT-all case with the SVM workload. In the synchronous case, we find that all ranks operate in a log step fashion, and during the scatter phase, all machines send models at the line rate (about 5 GB/s). Specifically, for the webspam workload, we see about 5.1 GB/s (about 40 Gb/s) during scatter. In the asynchronous case, to saturate the network, we run multiple replicas on every machine. When running three ranks on every machine, we find that each machine sends model updates at 4.2 GB/s (about 33 Gb/s) for the webspam dataset. These tests demonstrate that using a large bandwidth network is beneficial for training models with large number of parameters. Furthermore, using network-efficient techniques such as MALT-Halton can improve performance.

6.3 Developer Effort
We evaluate the ease of implementing parallel learning in MALT by adding support to the four applications listed in Table 3. For each application we show the amount of code we modified as well as the number of new lines added. In Section 4, we described the specific changes required. The new code adds support for creating MALT objects, to scatter-gather the model updates. In comparison, implementing a whole new algorithm takes hundreds of lines new code assuming underlying data parsing and arithmetic libraries are provided by the processing framework. On average, we moved 87 lines of code and added 106 lines, representing about 15% of overall code.

6.4 Fault Tolerance
We evaluate the time required for convergence when a node fails. When the MALT fault monitor in a specific node receives a time-out from a failed node, it removes that node from send/receive lists. We run MALT-SVM over ten ranks on eight nodes to train over the PASCAL-DNA dataset. We inject faults on MALT jobs on one of the machines and observe recovery and subsequent convergence. We inject the faults through an external script and also inject programmer errors such as divide by zero. We find that in each case, MALT fault monitors detected the unreachable failed mode, triggered a recovery process to synchronize with the remaining nodes and continued to train. We also observe that subsequent group operations only execute on the surviving nodes. Finally, we verify that the models converge to an acceptable accuracy in each of the failed cases. We also find that local fault monitors were able to trap processor exceptions and terminate the local training replica. We note that MALT cannot detect corruption of scalar values or Byzantine failures. Figure 14 shows one instance of failure recovery, and the time to converge is proportional to the number of remaining nodes in the cluster.

7. Related work
Our work draws inspiration from past work on data-parallel processing, ML specific platforms, ML optimizations and RDMA based key-value stores. Existing data-parallel frameworks: Batch processing systems based on map-reduce perform poorly for ma...
Conclusions

• MALT integrates with existing ML software to provide data-parallel learning
  • General purpose `scatter() / gather()` API to send model updates using one-sided communication
  • Mechanisms for network/representation optimizations
  • Supports applications written in C++ and Lua

• MALT provides speedup and can process large datasets
  • More results on speedup, network efficiency, consistency models, fault tolerance, and developer efforts in the paper

• MALT uses RDMA support to reduce model propagation costs
  • Additional primitives such as `fetch_and_add()` may further reduce model processing costs in software
Thanks

Questions?
Extra slides
Other approaches to parallel SGD

• **GPUs**
  • Orthogonal approach to MALT, MALT segments can be created over GPUs
  • Excellent for matrix operations, smaller sized models
  • However, a single GPU memory is limited to 6-10 GB
    • Communication costs dominate for larger models
  • Small caches: Require regular memory access for good performance
    • Techniques like sub-sampling/dropout perform poorly
    • Hard to implement convolution (need matrix expansion for good performance)

• **MPI/Other global address space approaches**
  • Offer a low level API (e.g. perform remote memory management)
  • A system like MALT can be built over MPI
## Evaluation setup

<table>
<thead>
<tr>
<th>Application</th>
<th>Model</th>
<th>Dataset/Size</th>
<th># training items</th>
<th># testing items</th>
<th># parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Classification</td>
<td>SVM</td>
<td>RCV1/480M</td>
<td>781K</td>
<td>23K</td>
<td>47K</td>
</tr>
<tr>
<td>Image Classification</td>
<td></td>
<td>Alpha/1G</td>
<td>250K</td>
<td>250K</td>
<td>500</td>
</tr>
<tr>
<td>DNA Classification</td>
<td></td>
<td>DNA/10G</td>
<td>23M</td>
<td>250K</td>
<td>800</td>
</tr>
<tr>
<td>Webspam detection</td>
<td></td>
<td>webspam/10G</td>
<td>250K</td>
<td>100K</td>
<td>16.6M</td>
</tr>
<tr>
<td>Genome classification</td>
<td></td>
<td>splice-site/250G</td>
<td>10M</td>
<td>111K</td>
<td>11M</td>
</tr>
<tr>
<td>Collaborative filtering</td>
<td>Matrix factorization</td>
<td>netflix/1.6G</td>
<td>100M</td>
<td>2.8M</td>
<td>14.9M</td>
</tr>
<tr>
<td>Click Prediction</td>
<td>Neural networks</td>
<td>KDD2012/3.1G</td>
<td>150M</td>
<td>100K</td>
<td>12.8M</td>
</tr>
</tbody>
</table>

- Eight Intel 8-core, 2.2 GHz Ivy-Bridge, with 64 GB
  - All machines connected via Mellanox 56 Gbps infiniband
Speedup using NMF with netflix dataset

![Graph showing speedup using NMF with netflix dataset](image)

- Median speedup over a single machine despite the additional costs of distributed computing for SVM and neural networks.
- We also find that the convergence of distributed algorithms is faster.
- In the case of the big data workload, distributed computing provides a larger speedup over the single machine.
- Comparing the performance on the big data workload, using distributed computing provides a larger speedup over the single machine.

### Figure 7

This figure shows the time spent for different distributed synchronous training steps for 20 ranks. We find that our distributed Hogwild implementation avoids conflicts and achieves convergence in about 574 seconds.

### Figure 8

This figure shows iterations vs test RMSE (Root Mean Square Error) for matrix factorization with the Netflix dataset. We now evaluate the time spent by MALT in different configurations do not wait while parameter server may send gradients but need to receive full model parameters. Furthermore, the parameter server has wait times for model-averaging even in the asynchronous case because workers need to wait for model parameters to arrive from the server before proceeding to the next iteration. Hence, as the workers need to wait for the stragglers to avoid incorporating stale updates. For small parameter values, convergence can be reduced by using a better function than the Halton function. A clever partitioning algorithm also finds that the convergence of distributed algorithms is faster.

- Netflix, all, ASYNC, cb=1000, ranks=2
- For data pre-processing that partitions non-conflicting (time for convergence)
- MALT configurations do not wait while parameter server may send gradients but need to receive full model parameters. The slaves in the parameter server may send gradients but need to receive full model parameters over the network. The slaves in the parameter server may send gradients but need to receive full model parameters over the network.
Halton sequence: $n \rightarrow n/2, n/4, 3n/4, 5n/8, 7n/8$
Speedup with different consistency models

- **BSP**: Bulk-synchronous parallelism (training with barriers)
- **ASYNC**: Fully asynchronous training
- **SSP**: Stale synchronous parallelism (limit stragglers by stalling fore-runners)

### Splice-site, all, modelavg, cb=5000, ranks=8

- **goal 0.01245**
- **BSP**
- **ASYNC 6X**
- **SSP 7.2X**

**Benefit with asynchronous training**
Developer efforts to make code data-parallel

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset</th>
<th>LOC Modified</th>
<th>LOC Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>RCV1</td>
<td>105</td>
<td>107</td>
</tr>
<tr>
<td>Matrix Factorization</td>
<td>Netflix</td>
<td>76</td>
<td>82</td>
</tr>
<tr>
<td>Neural Network</td>
<td>KDD 2012</td>
<td>82</td>
<td>130</td>
</tr>
</tbody>
</table>

On an average, about 15% of lines modified/added
Fault tolerance

Figure 13. This figure shows the data sent by MALT, MALT Halton and the parameter server for the webspam workload. MALT sends and receives gradients while parameter server sends gradients but needs to receive whole models. To summarize, we find that MALT provides sending gradients (instead of sending the model) that saves network costs. Furthermore, MALT Halton is network efficient and achieves speedup over MALT all.

Network saturation tests: We perform infiniBand network throughput tests, and measure the time to scatter updates in MALT all case with the SVM workload. In the synchronous case, we find that all ranks operate in a log step fashion, and during the scatter phase, all machines send models at the line rate (about 5 GB/s). Specifically, for the webspam workload, we see about 5.1 GB/s (about 40 Gb/s) during scatter. In the asynchronous case, to saturate the network, we run multiple replicas on every machine. When running three ranks on every machine, we find that each machine sends model updates at 4.2 GB/s (about 33 Gb/s) for the webspam dataset. These tests demonstrate that using a large bandwidth network is beneficial for training models with large number of parameters. Furthermore, using network-efficient techniques such as MALT Halton can improve performance.

6.3 Developer Effort
We evaluate the ease of implementing parallel learning in MALT by adding support to the four applications listed in Table 3. For each application we show the amount of code we modified as well as the number of new lines added. In Section 4, we described the specific changes required. The new code adds support for creating MALT objects, to scatter-gather the model updates. In comparison, implementing a whole new algorithm takes hundreds of lines new code assuming underlying data parsing and arithmetic libraries are provided by the processing framework. On average, we moved 87 lines of code and added 106 lines, representing about 15% of overall code.

6.4 Fault Tolerance
We evaluate the time required for convergence when a node fails. When the MALT fault monitor in a specific node receives a time-out from a failed node, it removes that node from send/receive lists. We run MALT-SVM over ten ranks on eight nodes to train over the PASCAL-DNA dataset. We inject faults on MALT jobs on one of the machines and observe recovery and subsequent convergence. We inject the faults through an external script and also inject programmer errors such as divide by zero. We find that in each case, MALT fault monitors detected the unreachable failed mode, triggered a recovery process to synchronize with the remaining nodes and continued to train. We also observe that subsequent group operations only execute on the surviving nodes. Finally, we verify that the models converge to an acceptable accuracy in each of the failed cases. We also find that local fault monitors were able to trap processor exceptions and terminate the local training replica. We note that MALT cannot detect corruption of scalar values or Byzantine failures. Figure 14 shows one instance of failure recovery, and the time to converge is proportional to the number of remaining nodes in the cluster.

7. Related work
Our work draws inspiration from past work on data-parallel processing, ML specific platforms, ML optimizations and RDMA based key-value stores.

Existing data-parallel frameworks: Batch processing systems based on map-reduce perform poorly for ma...