MALT: Distributed Data-Parallelism for Existing ML Applications

Hao Li [*†], Asim Kadav[†], Erik Kruus[†], Cristian Ungureanu[†]

NEC Labs, Princeton[†] University of Maryland, College Park[*]

{hli, asim, kruus, cristian} @nec-labs.com

Abstract

Machine learning methods, such as SVM and neural networks, often improve their accuracy by using models with more parameters trained on large numbers of examples. Building such models on a single machine is often impractical because of the large amount of computation required.

We introduce MALT, a machine learning library that integrates with existing machine learning software and provides data parallel machine learning. MALT provides abstractions for fine-grained in-memory updates using one-sided RDMA, limiting data movement costs during incremental model updates. MALT allows machine learning developers to specify the dataflow and apply communication and representation optimizations. Through its general-purpose API, MALT can be used to provide data-parallelism to existing ML applications written in C++ and Lua and based on SVM, matrix factorization and neural networks. In our results, we show MALT provides fault tolerance, network efficiency and speedup to these applications.

1. Introduction

Machine learning (ML) is becoming increasingly popular due to a confluence of factors: an abundance of data produced and captured in digital form [31]; an abundance of compute power and convenient access to it from various devices; and advances in the ML field, making it applicable to an ever growing number of situations [29]. The acceptance and success of ML, from natural language processing to image recognition to others, comes from the increasing accuracy achieved by ML applications. This accuracy is achieved partly through advances in ML algorithms, but also through using *known algorithms* with *larger models* trained on *larger datasets* [29]. Building these models on a single machine is often impractical because of the large amount of computation required, or may even be impossible for very large models such as those in state-of-the-art image recognition.

Existing data-parallel frameworks such as the map-reduce model have proven to be tremendously useful and popular paradigm for large-scale batch computations. However, existing frameworks are a poor fit for long running machine learning tasks. Machine learning algorithms such as gradient descent are iterative, and make multiple iterations to refine the output before converging to an acceptable value. Machine learning tasks have *all* of the following properties:

- *Fine-grained and Incremental*: Machine learning tasks perform repeated model updates over new input data. Most existing processing frameworks lack abstractions to perform iterative computations over small modifications *efficiently*. This is because in existing map-reduce implementations, jobs synchronize using the file-system [24] or maintain in-memory copies of intermediate data [54]. For computations with large number of iterations and small modifications, techniques such as these are suboptimal.
- *Asynchronous*: Machine learning tasks that run in parallel may communicate asynchronously. As an example, models that train in parallel may synchronize model parameters asynchronously. Enforcing determinism in the order of parameter updates can cause unnecessary performance overhead.
- *Approximate*: Machine learning tasks may perform computation stochastically and often an approximation of the trained model is sufficient. Existing general purpose systems rarely provide abstractions for trading off strong guarantees such as consistency or accuracy for performance (reduced job times).
- *Need Rich Developer Environment*: Developing ML applications require a rich set of ML libraries, developer tools and graphing abilities which is often missing in many highly scalable systems. Furthermore, existing ML

[*] Work done as NEC Labs intern.

EuroSys'15, April 21—25, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04.

http://dx.doi.org/10.1145/2741948.2741965

software such as sci-kit [2], Torch [19], RAPID [42], R [4] provide an efficient single-system library that performs well over multiple cores. However, most existing distributed learning tasks force developers to re-write their existing libraries in a new software stack and expose an unfamiliar environment to the developers.

To address these properties, we propose a system called MALT (stands for distributed Machine Learning Toolset), that allows ML developers to run their existing ML software in a distributed fashion. MALT provides an efficient shared memory abstraction that runs existing ML software in parallel and allows them to communicate updates periodically. MALT exports a `scatter-gather` API, that allows pushing model parameters or model parameter updates (called *gradients*) to parallel model replicas. These replicas then process the received values by invoking a user-supplied `gather` function locally. Additionally, the API allows developers to specify the dataflow across these replicas and specify representation optimizations (such as sparseness). MALT communication is designed using one-sided RDMA writes (no reads for faster round-trip times [33]) and provides abstractions for asynchronous model training.

Our data-parallel, peer-to-peer model communication complements the master-slave style parameter server approach [22, 23, 36]. In MALT, parallel model replicas send model updates to one-another instead of a central parameter server. This reduces network costs because the machines only communicate model updates back and forth instead of full models. Furthermore, implementing MALT, does not require writing separate master/slave code or dealing with complex recovery protocols to deal with master failures.

The contributions of this paper are as follows:

- We describe a general data-parallel machine learning framework that provides a simple and flexible API for parallel learning. MALT provides APIs for sending model parameter updates, designing the dataflow of the communication, and making this communication synchronous or asynchronous. Furthermore, MALT abstracts RDMA programming, and deals with system issues like recovering from unresponsive or failed nodes.
- We design a shared memory abstraction for ML workloads, that provides sending updates with no CPU overheads at the receiver, allowing for fully asynchronous model training. Furthermore, we demonstrate a network efficient, parallel learning implementation of MALT where we trade-off model freshness at replicas with faster model training times, for the same final accuracy. In our results, we show that our network efficient implementation achieves up to 1.8X speedup over asynchronous training, for training SVM over a 250 GB Genome classification workload, at 10% of network bandwidth costs.
- MALT provides ML developers data-parallelism in their existing ML software. We demonstrate how MALT can transform existing ML software written in procedural or scripting languages with reasonable developer efforts. This allows developers to incorporate data-parallelism in their applications in a familiar environment and use rich developer tools provided within their ML software. We use MALT to rewrite three applications for data-parallelism: SVM [12], matrix factorization [46] and neural networks [42]. We demonstrate how MALT can be used to gain speedup over a single machine for small datasets and train models over large datasets that span multiple machines efficiently.

We now present background on machine learning.

2. Distributed Machine Learning

Machine learning algorithms generalize from data. Machine learning algorithms train over data to create a *model* representation that can predict outcomes (regression or classification) for new unseen data. More formally, given a training set $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)\}$, the goal of model training is to determine the distribution function f such that $y = f(\mathbf{x}, \mathbf{w})$. The input $\mathbf{x}$ may consist of different features and the model consists of parameters $\mathbf{w}$, representing the weights of individual features to compute y. The process of model training is to estimate the values of model parameters $\mathbf{w}$. During model testing, this model is tested using an unseen set of $\mathbf{x}_t$ to compare against ground truth (already known y_t), to determine the model accuracy. Thus, machine learning algorithms *train* to minimize the *loss*, which represents some function that evaluates the difference between estimated and true values for the test data.

Model training algorithms are iterative, and the algorithm starts with an initial guess of the model parameters and learns incrementally over data, and refines the model every iteration, to converge to a final acceptable value of the model parameters. Model training time can last from minutes to weeks and is often the most time consuming aspect of the learning process. Model training time also hurts model refinement process since longer training times limit the number of times the model configuration parameters (called *hyper-parameters*) can be tuned through re-execution.

Machine learning algorithms can benefit from a scale-out computing platform support in multiple ways: First, these algorithms train on large amounts of data, which improves model accuracy [29]. Second, they can train large models that have hundreds of billions of parameters or require large computation such as very large neural networks for large-scale image classification or genomic applications [16]. Training with more data is done by *data parallelism*, which requires replicating the model over different machines with each model training over a portion of data. The replicas synchronize the model parameters after a fix number of iterations. Training large models requires the model to be split

across multiple machines, and is referred to as *model parallelism*.

The MALT API limits itself to data parallelism because models that learn over vast amounts of training data are more common than models with 100 billion parameters [9]. Furthermore, a single machine can process models of the order of 10 billion parameters in-memory (about 80 GB for dense representation; a server with 128GB DRAM costs about $3000). This is sufficient for most large machine learning models. Second, even though exposing a distributed, replicated, shared array is fairly straight-forward, efficiently providing model parallelism in existing applications is non-trivial. It requires exposing APIs and modifying machine learning algorithms to ensure that the model is split such that the communication costs are limited within each iteration (this is zero for data-parallelism). This is feasible for systems that focus on specific algorithms that are amenable to such splits like convolutional networks [16] or systems that write their own algorithms [36]. MALT's goal is to provide a simple, general purpose API that integrates easily with existing software, with a wide-variety of algorithms with reasonable developer efforts.

With datasets getting larger, there has been a recent focus to investigate online algorithms that can process data-sets incrementally such as the *gradient descent* family of algorithms. Gradient descent algorithms compute the gradient of a loss function over the entire set of training examples. This gradient is used to update model parameters to minimize the loss function [11]. Stochastic Gradient Descent (SGD) is a variant of the above algorithm that trains over one single example at time. With each example, the parameter vector is updated until the loss function yields an acceptable (low) value. SGD and its variants are preferred algorithms to train over large data-sets because it can process large training datasets in batches. Furthermore, gradient descent can be used for a wide-range of algorithms such as regression, k-means, SVM, matrix- factorization and neural networks [13, 17].

In data-parallel learning, model replicas train over multiple machines. Each replica trains over a subset of data. There are several ways in which the individual model parameters can be synchronized. We describe three such methods. First, models may train independently and synchronize parameters when all parallel models finish training by exhausting their training data [56]. These methods are commonly used to train over Hadoop where communication costs are prohibitive. In addition, while these models may train quickly because of limited communication between replicas, they may require more passes over training data (each pass over training data is called an *epoch*) for acceptable convergence. Furthermore, for non-convex problems, this method may not converge, since the parallel replicas may be trapped in a different local minimas, and averaging these diverging models may return a model with low accuracy.

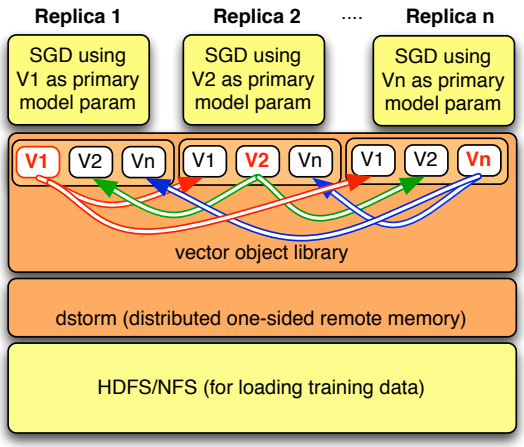

Figure 1. MALT architecture. Existing applications run with modified gradient descent algorithms that receive model update (V) from replicas training on different data. Model vectors are created using a Vector Object Library that allows creation of shared objects. Each replica `scatters` its model update after every (or every few) iteration and `gathers` all received updates before the next iteration.

The second method is the *parameter server* approach [23, 36]. Here, individual models send their updates to a central parameter server (or a group of parameter servers) and receive an updated model from them. A third method is the *peer-to-peer* approach (used in MALT), where parameters from model replicas train in parallel and are mixed every (or every few) iteration [30]. The last two methods achieve good convergence, even when the parameters are communicated asynchronously [30, 37]. With MALT, we perform asynchronous parameter mixing with multiple parallel instances of model replicas. This design allows developers to write code once, that runs everywhere on parallel replicas (no separate code for parameter server and client). This design also simplifies fault tolerance – a failed replica is removed from the parameter mixing step and its data is redistributed to other replicas. Finally, instead of performing simple gradient descent, MALT can be used to implement averaging of gradients from its peers, which provides speedup in convergence for certain workloads [13, 52].

The goal of our work is to provide distributed machine learning over existing ML systems. MALT exposes an asynchronous parameter mixing API that can be integrated into existing ML applications to provide data-parallel learning. Furthermore, this API is general enough to incorporate different communication and representation choices as desired by the machine learning developer. MALT provides peer-to-peer learning by interleaving gradient (changes to parameters) updates with parameter values to limit network costs. In the next section, we describe MALT design.

3. MALT Architecture

Figure 1 describes MALT architecture. Model replicas train in parallel on different cores (or sets of cores) across different nodes using existing ML libraries. ML libraries use the MALT vector library to create model parameters or gradients (updates to parameters) that need to be synchronized across machines. These vectors communicate over DiSTributed One-sided Remote Memory or *dstorm*. Furthermore, like other data-parallel frameworks, MALT loads data in model-replicas from a distributed file-system such as NFS or HDFS. Developers use the MALT API to shard input data across replicas and send/receive gradients. Furthermore, developers can also specify the dataflow across replicas and make their algorithms fully asynchronous. We now describe the shared memory design, the MALT API that allows developers access to shared memory, fault tolerance and network communication mechanisms that allow developers to balance communication and computation.

3.1 Abstractions for Shared Memory with `dstorm`

Machine learning models train in parallel over sharded data and periodically share model updates after few iterations. The parallel replicas may do so synchronously (referred to as the bulk-synchronous processing [48]). However, this causes the training to proceed at the speed of the slowest machine in that iteration. Relaxing the synchronous requirement speeds up model training but may affect the accuracy of the generated model. Since model weights are approximate, applications developers and researchers pick a point in this trade-off space (accuracy vs speed) depending on their application and system guarantees [21, 46]. Furthermore, this accuracy can be improved by training for multiple epochs or increasing the amount of data at for training each model replica.

The original map-reduce design communicates results over GFS/HDFS. However, using disk for communication, results in poor performance especially for machine learning applications which may communicate as often as every iteration. Spark [54] provides immutable objects (RDDs) for an efficient in-memory representation across machines. Spark provides fault tolerance using lineage of RDDs as they are transformed across operations. However, this enforces determinism in the order of operations. As a result, the immutability and determinism makes it less suitable for fine-grained, asynchronous operations [49, 54]. Furthermore, machine learning applications may contain multiple updates to large sparse matrices or may need to propagate model updates asynchronously across machines and need first-class support for fine-grained and asynchronous operations.

MALT's design provides efficient mechanisms to transmit model updates. There has been a recent trend of wide availability for cheap and fast infiniBand hardware and they are being explored for applications beyond HPC environments [25, 39]. RDMA over infiniBand allows low latency networking of the order of 1–3 micro-seconds by using user-space networking libraries and by re-implementing a portion of the network stack in hardware. Furthermore, the RDMA protocol does not interrupt the remote host CPU while accessing remote memory. RDMA is also available over Ethernet with the newer RDMA over Converged Ethernet (RoCE) NICs that have comparable performance to infiniBand. InfiniBand NICs are priced competitively with 10G NICs, costing around $500 for 40 Gbps NICs and 800$ for 56 Gbps NICs (as of mid 2014). Finally, writes are faster than reads since they incur lower round-trip times [33]. MALT uses one-sided RDMA writes to propagate model updates across replicas.

We build `dstorm` (dstorm stands for DiSTributed One-sided Remote Memory) to facilitate efficient shared memory for ML workloads. In MALT, every machine can create shared memory abstractions called *segments* via a `dstorm` object. Each `dstorm` segment is created by supplying the object size and a directed dataflow graph. To facilitate one-sided writes, when a `dstorm` segment is created, the nodes in the dataflow synchronously create `dstorm` segments. `dstorm` registers a portion of memory on every node with the infiniBand interface to facilitate one-sided RDMA operations. When a `dstorm` segment is transmitted by the sender, it appears at all its receivers (as described by the dataflow), without interrupting any of the receiver's CPU. We call this operation as *scatter*. Hence, a `dstorm` segment allocates space (a receive queue) in multiples of the object size, for every sender in every machine to facilitate the `scatter` operation. We use per-sender receive queues to avoid invoking the receiver CPU for resolving any write-write conflicts arising from multiple incoming model updates from different senders. Hence, our design uses extra space with the per-sender receive queues to facilitate lockless model propagation using one-sided RDMA. Both these mechanisms, the one sided RDMA and per-sender receive queues ensure that the `scatter` operation does not invoke the receive-side CPUs.

Once the received objects arrive in local per-sender receive queues, they can be read with a local *gather* operation. The `gather` function uses a user-defined function (UDF), such as an average, to collect the incoming updates. We also use queues on the sender side, allowing senders to perform writes asynchronously. Additionally, the sender-side queues maintain a back-pressure in the network to avoid congestion [47].

The receiver does not know when its per-sender receive queues get filled unless the receiver is actively polling and consuming these items. When the receive queue is full, the default behavior of `dstorm` is to over-write previously sent items in the queue. We discuss the consistency behavior after we describe the vector abstraction to create shared vectors or tensors (multi-dimensional vectors) over the `dstorm` object.

3.2 Vector Object Library: Programming Dstorm for Machine Learning

We build a vector object library (VOL) over `dstorm` that allows creating vector objects over shared memory. The goal of VOL is to 1) expose a vector abstraction instead of shared memory abstraction (`dstorm`) and 2) to provide communication and representation optimizations. ML developers can specify gradients or parameters as a VOL vector (or tensor) and specify its representation (sparse or dense). They also specify a dataflow graph describing how the updates should be propagated in the cluster which is used to create the underlying `dstorm` segment.

Hence, creating a vector in turn creates a `dstorm` segment that allows this vector to be propagated to all machines as described in the dataflow graph. This dataflow describes which machines may send updates to one another (in the simplest case, everyone may send their updates to everyone). Hence, an edge in the graph from node A to nodes B and C implies that when node A pushes a model update, it is received by nodes B and node C. As different machines compute model updates, they *scatter* these updates to other remote nodes without acquiring any locks or invoking any operations at the receiver. However, if a machine sends too many updates before the previous ones are consumed, the previous updates are over-written.

VOL inherits `scatter` and `gather` calls from `dstorm` to send the vector to remote machine and gather all the received updates (from local memory). Developers can also specify where to send the model updates within `scatter` calls. This provides fine-grained access to dataflow to the developers, allowing greater flexibility [40]. Table 1 describes the VOL API. In Section 4, we describe how this API can be used to easily convert serial ML algorithms to data-parallel.

Consistency guarantees: We now describe the consistency guarantees that MALT provides when transmitting model updates to other replicas. With machine learning applications, which are stochastic in nature, model updates maybe be over-written or updated locklessly without affecting overall accuracy of the model output significantly. For example, Hogwild demonstrates that asynchronous, lockless model updates lead to models that ultimately converge to acceptable accuracy [46]. Hence, MALT need not provide strict consistency guarantees when sending model updates over infiniBand (ex. as in key-value stores [25]). However, since MALT is a general-purpose API, it provides mechanisms to deal with following inconsistency issues:

1. *Torn reads*: When a model replica sends a model update to another model replica, the sender may overwrite the model update while the receiver is reading it in the case where the replicas operate asynchronously and the receive queue is full. MALT provides an additional `atomic gather` which reads the shared memory in an atomic fashion.
2. *Stale replicas*: Model updates carry an `iteration count` information in the header. When a receiver realizes that a specific model update is arriving too slowly, the receiver may stall its operations until the sender catches up. This design is similar to the bounded-staleness approach explored by recent work [21].

If stricter guarantees are required, the model replicas can train synchronously in *bulk-synchronous* fashion and use the `barrier` construct to do so. The `barrier` construct is a conventional barrier which waits for all model replicas to arrive at a specific point in the training process.

3.3 Fault tolerance

MALT has a straightforward model for fault tolerance. The training data is present on all machines in a distributed file system. The model replicas train in parallel and perform one-sided writes to all peers in the communication. A fault monitor on every node examines the return values of asynchronous writes to sender-side queues. If the fault monitor observes failed writes, it performs a synchronous health check of the cluster with other monitors on other nodes. A node is considered dead if the node is corrupt (the shared memory or the queue has failed) and the remote fault monitor reports this, or if the node is unreachable by *any* of the other healthy node's fault monitor. Furthermore, to detect the failure cases that do not result in a machine or a process crash, local fault monitors can detect processor exceptions such as divide by zero, stack corruption, invalid instructions and segmentation faults and terminate the local training process.

In case of a failure, the working fault monitors create a group of survivor nodes to ensure that all future group operations such as `barrier`, skip the failed nodes. The RDMA interface is re-registered (with old memory descriptors) and the queues are re-built. This is to avoid a zombie situation where a dead node may come back and attempt to write to one of the previously registered queues. Finally, the send and receive lists of all model replicas are rebuilt to skip the failed nodes and the training is resumed. Since the send and receive lists are re-built, it is possible to re-run any MALT configuration on a smaller number of nodes. If there is a network partition, training resumes on both clusters independently. However, it is possible to halt the training if the partition results in a cluster with very few nodes.

After recovery, if an acceptable loss value is not achieved, the training continues on the survivor replicas with additional training examples until the models converge. This causes a slowdown in the training process proportional to the missing machines apart from a short delay to synchronize and perform recovery (of the order of seconds). MALT only provides fail-stop fault tolerance, i.e. it can only handles failures where a fault monitor detects corruption or is unresponsive because of the MALT process being killed or a machine failure or a network failure. MALT cannot handle

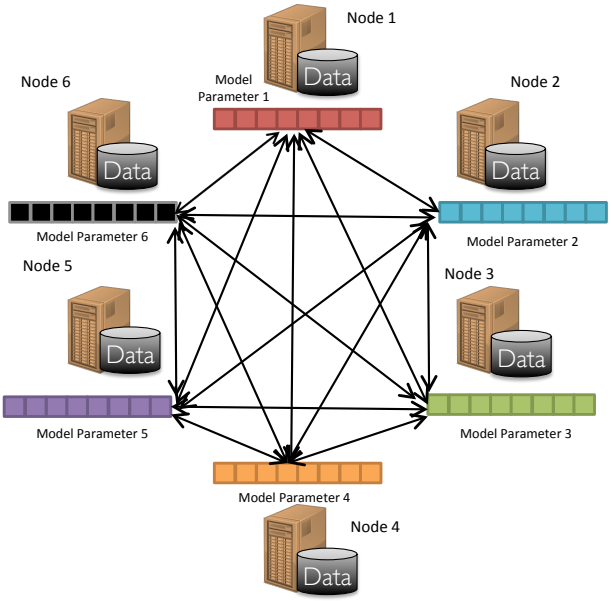

Figure 2. All-reduce exchange of model updates. All arrows indicate bi-directional communication. As number of nodes (N) grow, total number of updates transmitted increases O(N^2).

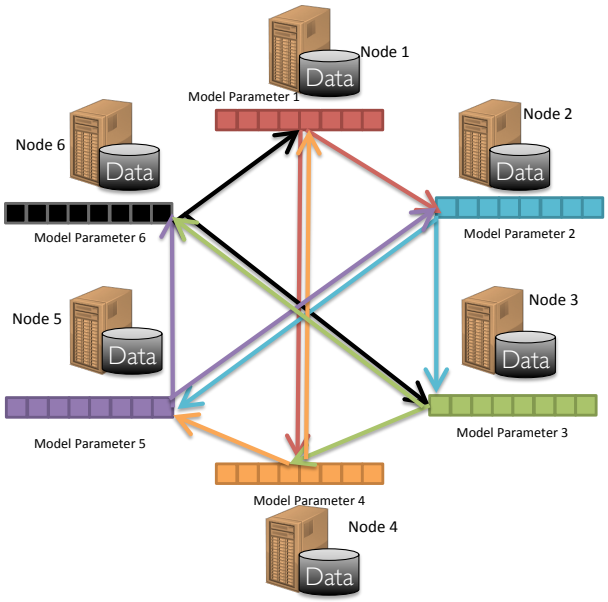

Figure 3. Halton-sequence exchange of model updates ($N = 6$). Each ith machine sends updates to $\log(N)$ (2 for $N = 6$) nodes. (to $N/2 + i$ and $N/4 + i$). As number of nodes N increases, the outbound nodes follows Halton sequence ($N/2, N/4, 3N/4, N/8, 3N/8..$). All arrows are uni-direction. As number of nodes(N) grow, total number of updates transmitted increases $O(N \log N)$.

Byzantine failures such as when a machine sends corrupt gradients or software corruption of scalar values that cannot be detected by local fault monitors.

MALT can afford a simple fault tolerance model because it only provides data parallelism and does not split the model across multiple machines. Furthermore, the model training is stochastic and does not depend on whether the training examples are processed in a specific order, or the training examples are processed more than once, or whether all the training examples have been processed, as long as the model achieves an acceptable accuracy. Furthermore, MALT implements peer-to-peer learning and does not have a central master. As a result, it does not need complex protocols like Paxos [15] to recover from master failures.

3.4 Communication Efficiency in MALT

MALT's flexible API can model different training configurations such as the parameter server [36], mini-batching [20, 38] and peer-to-peer parameter mixing [30].

When MALT is trained using the peer-to-peer approach, each machine can sends its update to all the other machines to ensure that each model receives the most recent updates. We refer to this configuration as MALT_{all}. As the number of nodes (N) increases, the gradient communication overhead in MALT_{all} increases $O(N^2)$ times, in a naïve all-reduce implementation. Efficient all-reduce primitives such as the butterfly [14] or tree style all-reduce [7], reduce the communication cost by propagating the model updates in a tree style. However, this increases the latency by a factor of the height of the tree. Furthermore, if the intermediate nodes are affected by stragglers or failures, an efficient all-reduce makes recovery complex.

In MALT, we propose an efficient mechanism to propagate model updates, what we refer to as *indirect* propagation of model updates. A developer may use the MALT API to send model updates to either all N nodes or fewer nodes k, $(1 \leq k \leq N)$. MALT facilitates choosing a value k such that a MALT replica (i) disseminates the updates across all the nodes eventually; (ii) optimizes specific goals of the system such as freshness, and balanced communication/computation ratio in the cluster. By eventually, we mean that over a period of time all the nodes receive model updates from every other node directly or indirectly via an intermediate node. However, when choosing a value k, less than N, the developer needs to ensure that the communication graph of all nodes *is connected*.

Hence, instead of performing an all-reduce, MALT limits the reduce operation to a subset of the connected nodes. However, naïvely or randomly selecting what nodes to send updates to may either leave out certain nodes from receiving updates from specific nodes (a partitioned graph of nodes) or may propagate updates that may be too stale (a weakly connected node graph). This may adversely affect the convergence in parallel learning models. We now describe how MALT can selectively distribute model updates to ensure low communication costs and uniform dissemination of model updates.

MALT provides a pre-existing dataflow that sends fewer model updates and ensures that all the models send/receive model updates in a uniform fashion. To do so, every node picks a node in a uniform fashion to ensure that the updates are distributed across all nodes. For example, if every node propagates its updates to k nodes ($k < N$), we pick the k node IDs based on a uniform random sequence such as the Halton sequence [1] that generates successive points that create a k-node graph with good information dispersal properties. We further propose that each node only send updates to $\log(N)$ nodes and maintain a $\log(N)$ sized node list. This node list contains the nodes to send updates to, generated using the Halton sequence. Hence, if we mark the individual nodes in training cluster as $1, ..., N$, Node 1 sends its updates to $N/2, N/4, 3N/4, N/8, 3N/8, 5N/8, ...$ and so on (the Halton sequence with base 2). Hence, in this scheme, the total updates sent in every iteration is only $O(N \log N)$. We refer to this configuration as $\text{MALT}_{\text{Halton}}$. The $\text{MALT}_{\text{Halton}}$ scheme ensures that the updates are sent uniformly across the range of nodes. Figures 2 and 3 show the all-to-all and Halton communication schemes. In case of a failure with $\text{MALT}_{\text{Halton}}$, the failed node is removed and the send/receive lists are rebuilt.

Using MALT's network-efficient parallel model training results in faster model training times. This happens because 1) The amount of data transmitted is reduced. 2) The amount of time to compute average of gradients is reduced since the gradient is received from fewer nodes. 3) In a synchronized implementation, this design reduces the number of incoming updates that each node needs to wait for, before going on to the next iteration. Furthermore, our solution reduces the need for high bandwidth interfaces, reducing costs and freeing up the network for other applications. In Section 6, we compare the network costs of indirect updates ($\text{MALT}_{\text{Halton}}$) with MALT_{all} and the parameter server.

Instead of having each node communicate with $\log(N)$ other nodes, developers can program MALT to communicate with higher (or lower) number of nodes. The key idea is to balance the communication (sending updates) with computation (computing gradients, applying received gradients). Hence, MALT accepts a dataflow graph as an input while creating vectors for the model parameters. However, the graph of nodes needs to be connected otherwise the individual model updates from a node may not propagate to remaining nodes, and the models may diverge significantly from one another.

4. Programming Interface

The goal of MALT is to provide data-parallelism to any existing machine learning software or algorithm. Given the MALT library and a list of machines, developers launch multiple replicas of their existing software that perform data-parallel learning.

MALT exposes an API as shown in Table 1. This API can be used to create (and port existing) ML applications for data-parallelism. To do so, the developer creates a parameter or a gradient object using MALT API. The `dense` object is stored as a `float` array and the `sparse` object is stored as key-value pairs.

Figure 4 shows a serial SGD algorithm (Algorithm 1) and a parallel SGD written using MALT (Algorithm 2). In the serial algorithm, the training algorithm goes over entire data and for each training sample, it calculates the associated gradient value. It then updates the model parameters, based on this gradient value.

In order to perform this training in a data-parallel fashion, this algorithm can be re-written using MALT API (as shown in Algorithm 2). The programmer specifies the representation (sparse vs dense) and the dataflow (`ALL` – which represents all machines communicate model updates to one-another, `HALTON` – which represents the network efficient API from previous section or the developer may specify an *arbitrary* graph – which represents the dataflow graph). When a job is launched using MALT it runs this code on each machine. Each machine creates a gradient vector object using the MALT API, with the required representation properties (sparse vs dense), and creates communication queues with other machines based on the dataflow specified, and creates receiving queues for incoming gradients.

Algorithm 1 Serial SGD

```
1: procedure SERIALSGD
2:     Gradient g;
3:     Parameter w;
4:     for epoch = 1 : maxEpochs do
5:         for i = 1 : maxData do
6:             g = cal_gradient(data[i]);
7:             w = w + g;
8:     return w
```

Algorithm 2 Data-Parallel SGD with MALT

```
1: procedure PARALLELSGD
2:     maltGradient g(SPARSE, ALL);
3:     Parameter w;
4:     for epoch = 1 : maxEpochs do
5:         for i = 1 : maxData/totalMachines do
6:             g = cal_gradient(data[i]);
7:             g.scatter(ALL);
8:             g.gather(AVG);
9:             w = w + g;
10:    return w
```

Figure 4: Data-parallel machine learning using MALT. The serial code (in Algorithm 1) is converted to data-parallel using MALT. All machines run the above code (in Algorithm 2). Instead of average, user may specify a function to combine incoming gradients/parameters. Optionally, `g.barrier()` may be used to run the algorithm in a synchronous fashion.

MALT API call	Purpose of the call
`g = createVector(Type)`	Creates a globally accessible shared model parameter or gradient (model update) vector. Type signifies sparse or dense.
`g.scatter (Dataflow Graph optional)`	Send model (or just model updates) to machines as described in graph (default sends to all machines).
`g.gather (func)`	Apply user-defined function `func` (like average) over model updates that have arrived (locally) and return a result.
`g.barrier ()`	Distributed barrier operation to force synchronization.
`load_data (f)`	Shard and load data from HDFS/NFS from `file f`.

Table 1. MALT interface. `g.scatter()` performs one-sided RDMA writes of gradient `g` to other machines. `g.gather()`, a local operation, applies average to the received gradients. `g.barrier()` makes the algorithm synchronous

Each machine trains over a subset of training data and computes the gradient value for each example. After training over each example (or bunch of examples), this gradient value is sent using the one-sided RDMA operation. The algorithm then computes an average of the received gradients using the `gather` function. Instead of an average, one can specify a user-defined function (UDF) to compute the resulting gradient from all incoming gradients. This is useful for algorithms where a simple averaging may not work, such as SVM may require an additional re-scaling function apart from performing an average over the incoming parameters. The training finishes when all machines in the cluster finish training over local examples. The final parameter value $\mathbf{w}$ is identical across all machines in the synchronous, all-all case. In other cases, $\mathbf{w}$ may differ slightly across machines but is within an acceptable loss value. In such cases, the parameters from any machines may be used as the final model or an additional reduce can be performed over $\mathbf{w}$ to obtain final parameter values.

For more complex algorithms, such as neural networks, which require synchronizing parameters at every layer of neural network, each layer of parameters is represented using a separate `maltGradient` and can have its own dataflow, representation and synchronous/asynchronous behavior.

Finally, it may be difficult to use the `maltGradient` allocation for certain legacy applications that use their own data-structures for parameters or gradients. For such opaque representations, where MALT cannot perform optimizations such as sparseness, developers directly use `dstorm`. `dstorm` provides low-level shared memory access with `scatter` and `gather` operations, allows managing the dataflow and controlling the synchronization. Furthermore, the opaque data-structures need to provide a serialization/de-serialization methods to copy-in/out from `dstorm`. Developers can also implement model-parallelism by carefully sharding their model parameters over multiple `dstorm` objects.

4.1 Applications

We use the MALT API to make the following algorithms data-parallel. Currently, MALT allows programmers to extend or write programs in C++ and Lua.

4.1.1 Support Vector Machines

We explore distributed stochastic gradient descent algorithms over linear and convex problems using Support Vector Machines(SVM). We use Leon Bottou's SVM-SGD [12]. Each machine calculates the partial gradient and sends it to other machines. Each machine averages the received gradients and updates its model weight vector ($\mathbf{w}$) locally.

4.1.2 Matrix Factorization

Matrix factorization involves partitioning a large matrix into its two smaller matrices. This is useful for data composed of two sets of objects, and their interactions needs to be quantified. As an example, movie ratings data contains interactions between users and movies. By understanding their interactions and calculating the underlying features for every user, one can determine how a user may rate an unseen movie. To scale better, large-scale matrix factorization is not exact, and algorithms approximate the factorizations [27]. SGD gives good performance for matrix factorizations on a single machine [34], and we perform matrix factorization using SGD across multiple machines. We implement Hogwild [46] and extend it from a multi-core implementation to a multi-node using MALT. With Hogwild, the gather function is a `replace` operation that overwrites parameters.

4.1.3 Neural Networks

We train neural networks for text learning. The computation in a neural network occurs over multiple layers forming a network. The training happens in forward and backward passes. In the forward pass, the input samples are processed at each layer and fed forward into the network, finally returning a predicted result at the end of the network. The difference in the ground truth and this predicted result is used in the *back-propagation* phase to update model weights using the gradient descent algorithm. Parallel training over neural networks is more difficult than SVM for two reasons. First, a data-parallel neural network requires synchronizing parameters for every layer. Second, finding the model weights for neural networks is a non-convex problem. Hence, just sending the gradients is not sufficient as the parallel model replicas maybe stuck in different local minimas. Hence, gradient synchronization needs to be interleaved with whole

model synchronization. We use RAPID [42], and extend its neural-network library with MALT. RAPID is similar in architecture to Torch [19], and provides a C++ library with Lua front-end for scripting. MALT exports its calls with Lua bindings and integrates with RAPID.

5. Implementation

MALT is implemented as a library, and is provided as a package to SVM-SGD, Hogwild and RAPID [42], allowing developers to use and extend MALT. `dstorm` is implemented over GASPI [8], that allows programming shared memory over infiniBand. GASPI exposes shared memory segments and supports one-sided RDMA operations. `dstorm` implements object creation, scatter, gather and other operations. `dstorm` hides all GASPI memory management from the user and provides APIs for object creation, scatter/gather and dataflow. GASPI is similar to MPI, and MALT can be implemented over MPI. However, GASPI has superior performance to certain MPI implementations [28].

We implement the vector object library over `dstorm` that provides vector abstractions, and provides other APIs for loading data, sparse and dense representations. Overall, MALT library is only 2366 LOC. To integrate with Lua, we have written Lua bindings (in Lua and C++) consisting of 1722 LOC. In Section 6.3, we evaluate the costs of integrating individual applications to MALT.

6. Evaluation

We evaluate MALT along the following criteria:

1. *Speedup with MALT*: What is the speedup provided by using MALT ?
2. *Network Optimizations*: How do the MALT network optimizations benefit training time?
3. *Developer Effort*: What is the developer effort required to port existing ML applications?
4. *Fault Resilience*: How does MALT behave in the presence of failures?

We use MALT to modify SVM-SGD, Hogwild (matrix factorization) and RAPID (neural networks). Table 2 lists the application and the datasets used. We perform all experiments on a 8 machine research cluster connected via an infiniBand backplane. We run multiple processes, across these 8 machines, and we refer to each process as a *rank* (from the HPC terminology). We run multiple ranks on each machine, especially for models with less than 1M parameters, where a single model replica is unable to saturate the network and CPU. Each machine has an Intel Xeon 8-core, 2.2 GHz Ivy-Bridge processor with support for SSE 4.2/AVX instructions, and 64 GB DDR3 DRAM. Each machine is connected via a Mellanox Connect-V3 56 Gbps infiniBand card and all machines are connected using a Mellanox managed-switch with copper cables. Our 56 Gbps infiniBand network architecture provides a peak throughput of slightly over 40 Gbps after accounting for the bit-encoding overhead for reliable transmission. All machines share storage using a 10 TB NFS partition that we use for loading input data. Each process loads a portion of data depending on the number of processes. For all our experiments, we randomize the input data and assign random subsets to each node. All reported times do not account the initial one-time cost for the loading the data-sets in memory. All times are reported in seconds.

Application	Model	Dataset	Training	Testing	Params
Document classification	SVM	RCV1	781K	23K	47,152
Image classification	SVM	Alpha	250K	250K	500
DNA	SVM	DNA	23M	250K	800
Webspam detection	SVM	Webspam	250K	100K	16.6M
Genome detection	SVM	Splice-site	10M	111K	11M
Collaborative filtering	MF	Netflix	100M	2.8M	14.9M
CTR prediction	SSI	KDD12	150M	100K	12.8M

Table 2. MALT applications and dataset properties.

We perform our experiments on applications as described in Table 2. We perform data-parallel learning for SVM, matrix factorization and neural networks. We use small and large datasets. The small datasets have well understood convergence behavior and allow us to verify correctness. Furthermore, they allow us to measure the speedup over single machine performance. The large datasets help us evaluate scalability of our design. To evaluate SVM, we use RCV1, PASCAL suite (alpha, webspam, DNA) and splice-site datasets [6]. The compressed training set sizes are RCV1 – 333 MB (477 MB uncompressed), PASCAL alpha – 651 MB (1 GB uncompressed), webspam – 2.6 GB (10 GB uncompressed), DNA 2.5 GB (10 GB uncompressed) and splice-site – 110 GB (250 GB uncompressed). The splice-site dataset does not fit in a single machine and requires majority of the dataset for an accurate model. For matrix factorization, we use the Netflix dataset (1.6 GB uncompressed). For neural networks, we perform click-through rate (CTR) prediction based on the Tencent data released with KDD Cup 2012 challenge [3]. The neural network is a three-layer fully-connected neural network that performs supervised-semantic indexing (SSI) [10]. The SSI model trains on 3.1 GB of processed training data. We report performance compared to a single process SGD except for the splice-site dataset which cannot run in one machine. The baseline for splice-site dataset is bulk-synchronous processing over MALT.

6.1 Speedup

In this section, we compare the speedup of different datasets over a single machine and existing methods. We also evaluate the time spent across different processing tasks and the

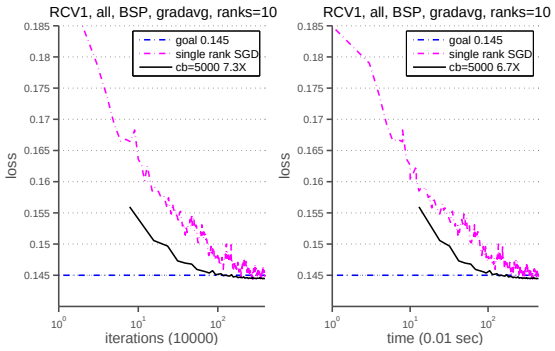

Figure 4. This figure shows convergence for RCV1 workload for MALT$_{all}$ with a single machine workload. We find that MALT$_{all}$ converges quicker to achieve the desired accuracy.

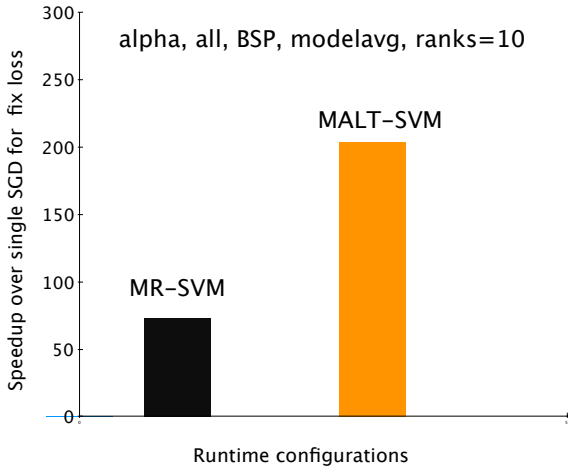

Figure 5. This figure shows speedup by iterations with PASCAL alpha workload for MALT$_{all}$ SVM with MR-SVM. MR-SVM algorithm is implemented using the MALT library over infiniBand. We achieve super-linear speedup for some workloads because of the averaging effect from parallel replicas [52].

benefit of different synchronization methods. We compare speedup of the systems under test by running them until they reach the same loss value and compare the total time and number of iterations (passes) over data per machine. Distributed training requires fewer iterations per machine since examples are processed in parallel. For each of our experiments, we pick the desired final optimization goal as achieved by running a single-rank SGD [6, 12]. Figure 4 compares the speedup of MALT$_{all}$ with a single machine for the RCV1 dataset [12], for a *communication batch size* or *cb size* of 5000. By *cb size* of 5000, we mean that every model processes 5000 examples from the dataset and then propagates the model updates to all other machines. We find that MALT$_{all}$ provides a speedup with good convergence. By 10 *ranks*, we mean 10 processes, that span our eight machine cluster. For RCV1 and other smaller workloads, we find that we are unable to saturate the network and CPU with a single replica, and run multiple replicas on a single machine.

We now compare MALT-SVM performance with an existing algorithm designed for map-reduce (MR-SVM). MR-SVM algorithm is based on common Hadoop implementations and communicates gradients after every partition-epoch [56]. We implement MR-SVM algorithms over the MALT library and run it over our infiniBand cluster. MR-SVM uses one-shot averaging at the end of every epoch to communicate parameters (*cb size* = 25K). MALT is designed for low-latency communication and communicated parameters more often (*cb size* = 1K). Figure 5 shows speedup by iterations for the PASCAL alpha workload for MR-SVM (implemented over MALT) and MALT$_{all}$. We find that both the workloads achieve super-linear speedup over a single machine SGD on the PASCAL alpha dataset. This happens because the averaging effect of gradients provides super-linear speedup for certain datasets [52]. In addition, we find that MALT provides 3× speedup (by iterations, about 1.5× by time) over MR-SVM. MALT converges faster since it is designed over low latency communication, and sends gradients more frequently. This result shows that existing algorithms designed for map-reduce may not provide the most optimal speedup for low latency frameworks such as MALT.

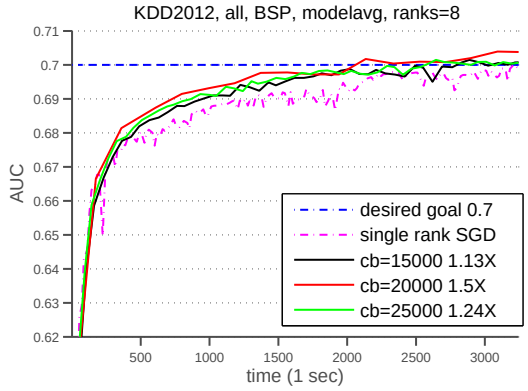

Figure 6. This figure shows the AUC (Area Under Curve) vs time (in seconds) for a three layer neural network for text learning (click prediction).

Figure 6 shows the speedup with time for convergence for ad-click prediction implemented using a fully connected, three layer neural network. This three layer network needs to synchronize parameters at each layer. Furthermore, these networks have dense parameters and there is computation in the forward and the reverse direction. Hence, fully-connected neural networks are harder to scale than convolution networks [23]. We show the speedup by using MALT$_{all}$ to train over KDD-2012 data on 8 processes over single machine. We obtain up to 1.5× speedup with 8 ranks. The speedup is limited as compared to SVM because 1) SSI is non-convex and requires high-dimensional model communication as opposed to gradient and 2) text processing in a neural network requires limited computation and communication costs dominate.

Figure 7 shows speedup by iterations over single machine SGD with matrix factorization. We show convergence for two different learning rate strategies – *byiter*, where we start with a learning rate and decrease every certain num-

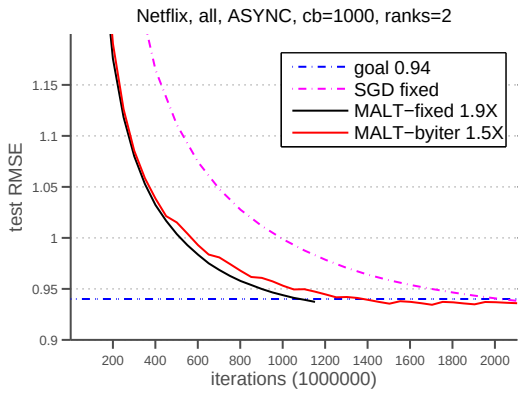

Figure 7. This figure shows iterations vs test RMSE (Root Mean Square Error) for matrix factorization with the Netflix dataset. We train asynchronously over two machines, and use `replace` as the `gather` function.

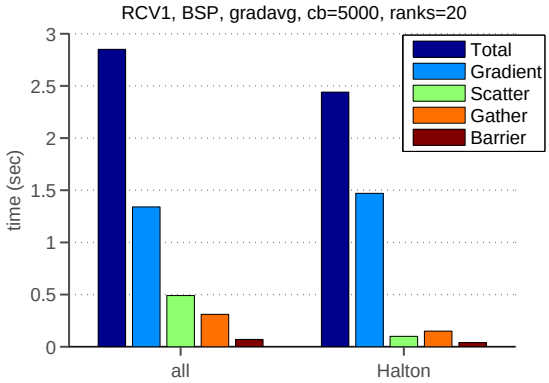

Figure 8. This figure shows the time consumed by different steps in distributed SVM for RCV1 workload for synchronous training.

ber of iterations and *fixed*, where we keep a fixed learning rate during the entire training process [53]. Each rank processes part of the training data and communicate updates asynchronously across different machines, where updates refers to changed rows and columns of the factorized matrix. We use `replace` as the `gather()` function that emulates Hogwild [46] in a distributed environment. We show iterations and error rate for different communication batch sizes. We find that our distributed Hogwild implementation generates lots of network traffic for small batch sizes (*cb size* < 500) and hurts performance. For larger batch sizes, there is lots of wasted work over similar ($user$, $movie$) since the Hogwild approach overwrites parameter values across ranks. To avoid wasted work, we sort the input data by $movie$ and split across ranks (for all runs including baseline), to avoid conflicts and achieve convergence in about 574 seconds. We find that MALT performs one pass over the data (epoch) in 26 seconds. For one epoch, Sparkler takes 96 seconds and Spark takes 594 seconds, for 25 ranks, over non-infiniBand hardware (the paper does not report the total time for convergence) [35]. A clever partitioning algorithm for data pre-processing that partitions non-conflicting ($user$, $movie$) pairs across different ranks may provide additional performance gains [44].

We now evaluate the time spent by MALT in different training steps such as calculating the gradient, sending updates, etc. Figure 8 shows the time spent for different distributed synchronous training steps for 20 ranks. We find that MALT balances computation with communication and nodes spend most of their time computing gradients and pushing them (as opposed to blocking). In async workloads, MALT configurations do not wait while parameter server clients need to wait for updated models to arrive after sending their gradients as shown in Figure 9. Figure 9 also shows the efficiency of using gradient updates instead of sending whole models over the network. The slaves in the 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 the workers need to wait for model parameters to arrive from the server before proceeding to the next iteration. Hence, it suffers from higher network costs for high-dimensional models such as the webspam workload.

Figure 10 shows the speedup with MALT bulk synchronous (BSP), asynchronous (ASP) and bounded staleness (SSP) models for the splice-site dataset over 8 machines. This training dataset consists of 10M examples (250 GB) and does not fit in a single machine. Each machine loads a portion of the training data (about 30 GB). SSP was proposed using the parameter server approach [21] and we implement it for MALT. Our implementation of bounded staleness (SSP) only merges the updates if they are within a specific range of iteration counts of its neighbor. Furthermore, if a specific rank lags too far behind, other ranks stall their training for a fixed time, waiting for the straggler to catch up. Our ASP implementation skips merging of updates from the stragglers to avoid incorporating stale updates. For the splice-site dataset, we find that SSP converges to the desired value first, followed by ASP and BSP.

To summarize, we find that MALT achieves reasonable a speedup over a single machine despite the additional costs of distributed computing for SVM and neural networks. Furthermore, MALT can process large datasets and models with large parameter values in a reasonable amount of time. We also find that the convergence of distributed algorithms is less smooth when compared to a single rank SGD. This variance can be reduced by using a better `gather` function than simple averaging [38, 55].

6.2 Network Optimizations

We now evaluate the benefit of our network optimizations. Figure 11 shows the model convergence graph for MALT_{all} and $\text{MALT}_{\text{Halton}}$ to reach the required loss value for the RCV1 dataset. We find that $\text{MALT}_{\text{Halton}}$ converges more slowly than MALT_{all}, in terms of convergence per iteration. However, the overall time to converge is less because: First, parallel nodes (N) only communicate with fewer ($\log(N)$)

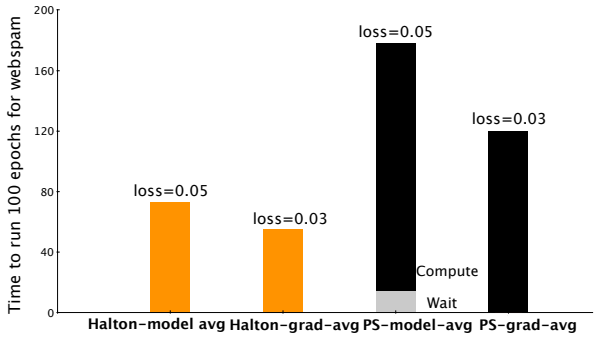

Figure 9. This figure compares MALT$_{\text{Halton}}$ with parameter server (PS) for distributed SVM for webspam workload for asynchronous training, with achieved loss values for 20 ranks.

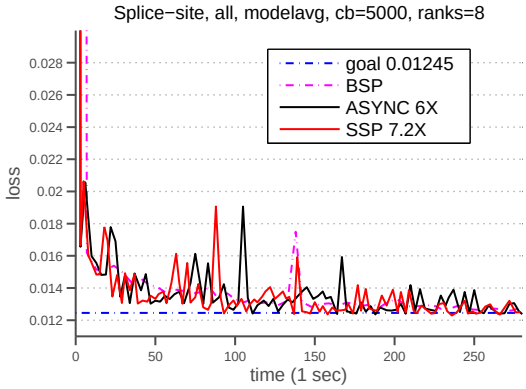

Figure 10. This figure shows the convergence for bulk-synchronous (BSP), asynchronous processing (ASP) and bounded staleness processing (SSP) for splice-site workload.

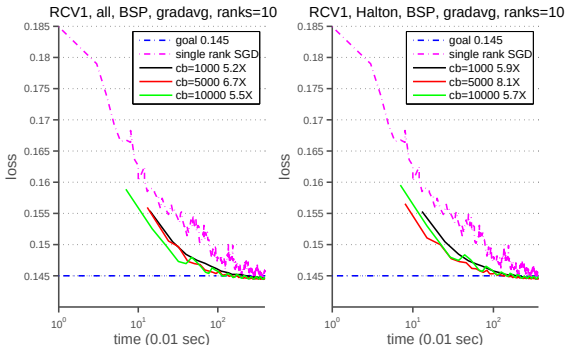

Figure 11. This figure shows convergence (loss vs time in seconds) for RCV1 dataset for MALT$_{\text{all}}$ (left) and MALT$_{\text{Halton}}$ (right) for different communication batch sizes. We find that MALT$_{\text{Halton}}$ converges faster than MALT$_{\text{all}}$.

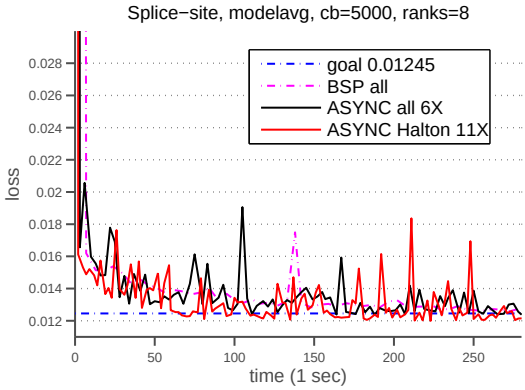

Figure 12. This figure shows convergence (loss vs time in seconds) for splice-site dataset for MALT$_{\text{all}}$ and MALT$_{\text{Halton}}$. We find that MALT$_{\text{Halton}}$ converges faster than MALT$_{\text{all}}$.

machines. Second, each node performs model averaging of fewer $(\log(N))$ incoming models. Hence, even though MALT$_{\text{Halton}}$ may require more iterations than MALT$_{\text{all}}$, the overall time required for every iteration is less, and *overall convergence time to reach the desired accuracy is less*. Finally, since MALT$_{\text{Halton}}$ spreads out its updates across all nodes, that aids faster convergence.

Figure 12 shows the model convergence for the splice-site dataset and the speedup over BSP-all in reaching the desired goal with 8 nodes. From the figure, we see that MALT$_{\text{Halton}}$ converges faster than MALT$_{\text{all}}$. Furthermore, we find that that until the model converges to the desired goal, each node in MALT$_{\text{all}}$ sends out 370 GB of updates for every machine, while MALT$_{\text{Halton}}$ only sends 34 GB of data for every machine. As the number of nodes increase, the logarithmic fan-out of MALT$_{\text{Halton}}$ should result in lower amounts of data transferred and faster convergence.

MALT $_{\text{Halton}}$ trades-off freshness of updates at peer replicas with savings in network communication time. For workloads where the model is dense and network communication costs are small compared to the update costs, MALT$_{\text{all}}$ configuration may provide similar or better results over MALT $_{\text{Halton}}$. For example, for the SSI workload, which is a fully connected neural network, we only see a $1.1\times$ speedup for MALT$_{\text{Halton}}$ over MALT$_{\text{all}}$. However, as the number of nodes and model sizes increase, the cost of communication begins to dominate, and using MALT$_{\text{Halton}}$ is beneficial.

Figure 13 shows the data sent by MALT$_{\text{all}}$, MALT$_{\text{Halton}}$, and the parameter server over the entire network, for the webspam workload. We find that MALT$_{\text{Halton}}$ is the most network efficient. Webspam is a high-dimensional workload. MALT$_{\text{Halton}}$ only sends updates to $\log(N)$ nodes. The parameter server sends gradients but needs to receive the whole model from the central server. We note that other optimizations such as compression, and other filters can further reduce the network costs as noted in [36]. Furthermore, when the parameter server is replicated for high-availability, there is more network traffic for additional N (asynchronous) messages for $N-way$ chain replication of the parameters.

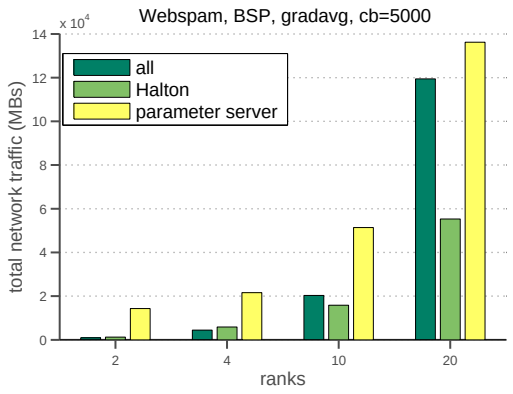

Figure 13. This figure shows the data sent by $MALT_{all}$, $MALT_{Halton}$ and the parameter server for the webspam workload. MALT sends and receives gradients while parameter server sends gradients but needs to receive whole models.

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 [6] 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.

Application	Dataset	MALT annotations	
		LOC Modified	LOC Added
SVM	RCV1	105	107
Matrix Factorization	Netflix	76	82
SSI	KDD2012	82	130

Table 3. Developer effort for converting serial applications to data-parallel with MALT.

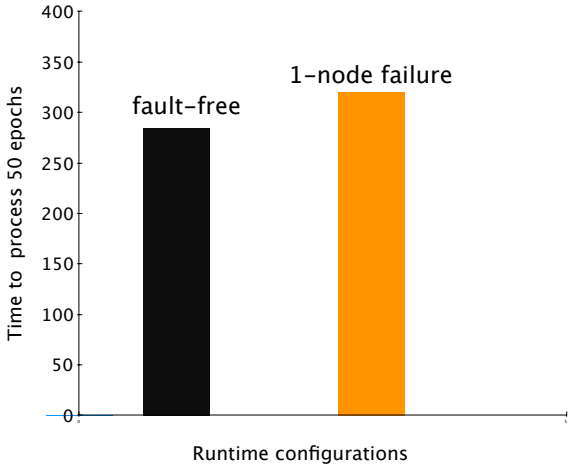

Figure 14. This figure shows the time taken to converge for DNA dataset with 10 nodes in fault-free and a single process failure case. We find that MALT is able to recover from the failure and train the model with desired accuracy.

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 [24, 51] perform poorly for ma-

chine learning jobs because the iterative nature of these algorithms require frequent communication using disks. Furthermore, the data-flow model provided by map-reduce is restrictive, and limits the flexibility of expressing communication across tasks. Spark [54] provides an efficient in-memory representation to synchronize data across iterations. Spark provides copy-on-write, in-memory structures that improve performance for batch workloads. It also provides fault tolerance using lineage (re-computation) that enforces determinism. However, this can be less efficient for certain machine learning algorithms that make fine-grained and asynchronous updates. Dryad [32] and CIEL [41] provide a more flexible data-parallel communication API to write any arbitrary data flow but share data across tasks through disks. MPI [26] provides a low-level message passing constructs, and a system similar to MALT can be built over MPI. Picollo [45] provides a distributed, master-slave key-value store, and resolves writes conflicts using user-defined functions. Picollo provides strong fault tolerance, consistency and determinism guarantees that can be relaxed in MALT for performance. MALT is completely asynchronous, and allows senders to transmit data with one-sided write operations. Furthermore, unlike Piccolo, MALT provides a communication API that allows the programmer to control where the updates may reside and how they may propagate.

ML frameworks: DistBelief [23] Project Adam [16], and the parameter server [36], use a master-client style communication with the parameter server, that complements MALT. MALT is a general purpose API, designed to train models in a peer-to-peer fashion. MALT's peer-to-peer style simplifies fault-tolerance, and only requires writing code once that executes over all machines. Furthermore, most existing parallel learning frameworks require a re-write of applications and libraries. Vowpal Wabbit [5] provides data-parallel learning, where individual model replicas train in parallel and average the gradients using the LBFGS algorithm. However, it runs over Hadoop and lacks efficient shared memory semantics that MALT provides. Presto [49], has a similar goal to ours, to provide a rich developer environment for parallel learning. It provides a parallel R, since R is a common data analysis tool. Presto provides a large distributed array abstraction to shard data and removes scalability bottlenecks in R implementation. The MALT library can be used to parallelize many existing learning frameworks. There are many GPU based frameworks [18]. However, GPU speedups are limited for datasets exceeding its memory sizes ($<$10 GB). Furthermore, training multiple-GPUs over the network incurs significant communication costs. However, there has been recent work on improving GPU-GPU communication using infiniBand [50].

ML optimizations: We now discuss prior work to optimize distributed machine learning to reduce synchronization and improve convergence. Many researchers have explored improving stochastic gradient descent over distributed systems by providing *mini-batching* [20]. This reduces the amount of communication but reduces convergence speed [30]. HogWild [46] provides a single shared parameter vector and allows parallel workers to update model parameters without any locking (in a single machine). However, this method only works when updates are sparse and there is limited overlap. HogWild can also generate significant network traffic when adopted in a distributed setting. Bounded-staleness [21] limits stale updates from stragglers by slowing down the forerunners. Iterative parameter mixing (used by MALT) has been shown to provide high-accuracy models over map-reduce [30]. Optimistic Concurrency Control [43] uses database style coordination free model updates for distributed machine learning. MALT also provides coordination free updates by allocating a per-receiver queue at sender and avoids invoking the remote CPU by using one-sided RDMA.

RDMA systems: MALT uses one-sided RDMA writes to communicate gradients for every batch and performs no remote reads. Recently, infiniband hardware has been used to build transactions on objects in shared address space in FARM [25] and client-server based key-value stores with Pilaf [39]. Pilaf provides consistency using checksums, while FARM provides consistency by ordering DMA writes. MALT provides a finite per-sender queue at the receiver to avoid write-write conflicts. Older gradients maybe overwritten by a fast sender and the receiver averages (or computes any other user defined function) the incoming model updates for conflict resolution.

8. Conclusion

Existing map-reduce frameworks are optimized for batch processing systems and ill-suited for tasks that are iterative, fine-grained and asynchronous. Recent scalable ML platforms force developers to learn a new programming environment and rewrite their ML software. The goal of MALT is to efficiently provide data-parallelism to existing ML software. Given a list of machines and MALT library, we demonstrate that one can program ML algorithms, control the data-flow and synchrony. We provide MALT library interface for procedural (C++) and scripting (Lua) languages and demonstrate data-parallel benefits with SVM, matrix factorization and neural networks. MALT uses one-sided RDMA primitives that reduces network processing costs and transmission overhead. The new generation of RDMA protocols provide additional opportunities for optimizations. Primitives such as `fetch_and_add` can be used to perform gradient averaging in hardware and further decrease the model training costs in software.

Acknowledgments

We would like to thank our shepherd Derek Murray and the anonymous reviewers for the useful feedback. We also thank Igor Durdanovic for helping us port RAPID to MALT and Hans-Peter Graf for his support and encouragement.

References

[1] Halton sequence. en.wikipedia.org/wiki/Halton_sequence.

[2] Machine Learning in Python. http://scikit-learn.org/.

[3] Tencent 2012 KDD Cup. https://www.kddcup2012.org.

[4] The R Project for Statistical Computing. www.r-project.org/.

[5] Vowpal Wabbit. http://hunch.net/ vw/.

[6] PASCAL Large Scale Learning Challenge. http://largescale.ml.tu-berlin.de, 2009.

[7] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *JMLR*, 2014.

[8] T. Alrutz, J. Backhaus, T. Brandes, V. End, T. Gerhold, A. Geiger, D. Grünewald, V. Heuveline, J. Jägersküpper, A. Knüpfer, et al. Gaspi–a partitioned global address space programming interface. In *Facing the Multicore-Challenge III*, pages 135–136. Springer, 2013.

[9] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[10] B. Bai, J. Weston, D. Grangier, R. Collobert, K. Sadamasa, Y. Qi, O. Chapelle, and K. Weinberger. Supervised semantic indexing. In *ACM CIKM*, 2009.

[11] C. M. Bishop et al. *Pattern Recognition and Machine Learning*. Springer New York, 2006.

[12] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Springer COMPSTAT*, 2010.

[13] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[14] J. Canny and H. Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *SDM*, 2013.

[15] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *IEEE PODC*, 2007.

[16] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX OSDI*, 2014.

[17] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *USENIX HotOS*, 2013.

[18] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and A. Ng. Deep learning with COTS HPC systems. In *ACM ICML*, 2013.

[19] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[20] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *NIPS*, 2011.

[21] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[22] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[23] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: fast remote memory. In *USENIX NSDI*, 2014.

[26] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[27] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *ACM KDD*, 2011.

[28] GPI2: Programming Next Generation Supercomputers. Benchmarks.

[29] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[30] K. B. Hall, S. Gilpin, and G. Mann. Mapreduce/bigtable for distributed optimization. In *NIPS LCCC Workshop*, 2010.

[31] M. Hilbert and P. López. The worlds technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[32] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM EuroSys*, 2007.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*. ACM, 2014.

[34] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[35] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *ACM EDBT*, 2013.

[36] M. Li, D. Andersen, A. Smola, J. Park, A. Ahmed, V. Josifovski, J. Long, E. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, 2014.

[37] M. Li, D. G. Andersen, and A. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[38] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *ACM KDD*, 2014.

[39] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *ACM SOSP*, 2013.

[41] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *USENIX NSDI*, 2011.

[42] NEC Laboratories America. MiLDE: Machine Learning Development Environment. `http://www.nec-labs.com/research-departments/machine-learning/machine-learning-software/Milde`.

[43] X. Pan, J. E. Gonzalez, S. Jegelka, T. Broderick, and M. Jordan. Optimistic concurrency control for distributed unsupervised learning. In *NIPS*, 2013.

[44] F. Petroni and L. Querzoni. GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. In *ACM RecSys*, 2014.

[45] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX OSDI*, 2010.

[46] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[47] J. R. Santos, Y. Turner, and G. Janakiraman. End-to-end congestion control for infiniband. In *IEEE INFOCOM*, 2003.

[48] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[49] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *ACM EuroSys*, 2013.

[50] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development*, 26(3-4):257–266, 2011.

[51] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2009.

[52] W. Xu. Towards optimal one pass large scale learning with averaged stochastic gradient descent. *arXiv preprint arXiv:1107.2490*, 2011.

[53] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. In *ACM VLDB*, 2014.

[54] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.

[55] S. Zhang, A. Choromanska, and Y. LeCun. Deep learning with Elastic Averaging SGD. *arXiv preprint arXiv:1412.6651*, 2014.

[56] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.