MALT: Distributed Data-Parallelism for Existing ML Applications

Hao Li[*], **Asim Kadav, Erik Kruus**
NEC Laboratories America
{asim, kruus@nec-labs.com}

Abstract

We introduce MALT, a machine learning library that integrates with existing machine learning software and provides peer-to-peer 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. In our results, we find that MALT provides fault tolerance, network efficiency and speedup to SVM, matrix factorization and neural networks.

1 Introduction

Existing map-reduce frameworks have proven to be tremendously useful and popular paradigm for large-scale batch computations. However, these frameworks can be a poor fit for long running ML tasks. ML algorithms such as gradient descent are iterative, and make multiple iterations to refine the output before converging to an acceptable value. ML tasks have the following properties:

- *Fine-grained and Incremental*: ML 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 [8] or maintain in-memory copies of intermediate data [15]. For computations with large number of iterations and small modifications, these techniques can be sub-optimal.
- *Asynchronous and Approximate*: ML 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. Furthermore, ML tasks may perform computation stochastically and often an approximation of the trained model is sufficient.

To address these properties, we propose a system called MALT (stands for distributed Machine Learning Toolset), that runs existing ML software over a cluster [11]. 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 (*gradients*) to parallel model replicas. These replicas then process the received values by invoking a user-supplied `gather` function *locally*. MALT communication is designed using one-sided RDMA writes (no reads for faster round-trip times [10]) and provides abstractions for asynchronous model training. Furthermore, MALT abstracts RDMA programming, and deals with system issues like recovering from unresponsive or failed nodes.

[*] Work done as a NEC Labs intern; now at U. Maryland

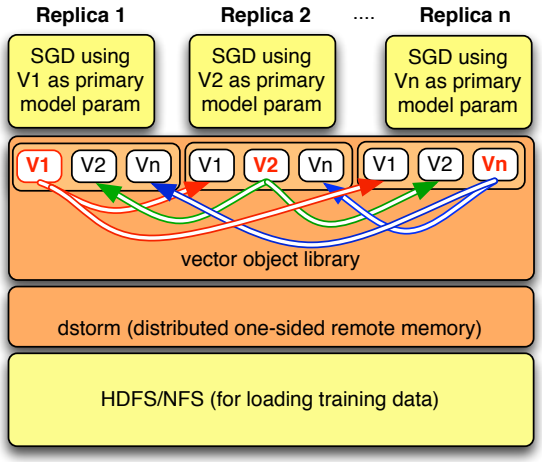

Figure 1: **MALT architecture. Existing applications run with modified gradient descent algorithms that receive model update (V) from replicas training on different data. Model vectors are created using a Vector Object Library that allows creation of shared objects. Each replica `scatters` its model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

Our data-parallel, peer-to-peer model communication complements the master-slave style parameter server approach [6, 7, 12]. 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. We demonstrate that 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.

2 MALT Design

In MALT, model replicas train in parallel on different cores across different nodes using existing ML libraries. ML libraries use the MALT vector library to create model parameters or gradients that need to be synchronized across machines. These vectors communicate over MALT's shared memory abstraction over infiniBand. Furthermore, MALT loads data in model-replicas from a distributed file-system such as NFS or HDFS. Each machines reads a portion of randomized data. This requirement may be relaxed with learning algorithms that can train over non-random data [14].

Abstractions for Shared Memory with `dstorm` 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 [9, 13]. 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. Finally, writes are faster than reads since they incur lower round-trip times [10] and MALT exclusively uses writes to implement its shared memory architecture.

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 gradient flow 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

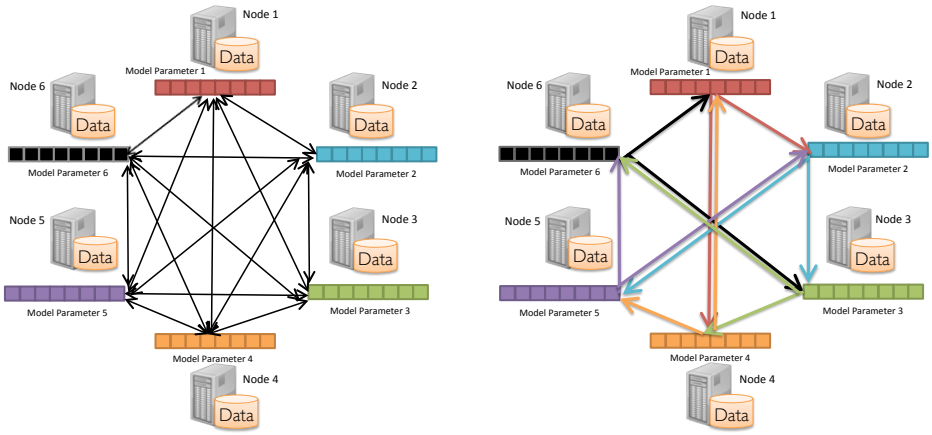

Figure 2: **P2P communication protocols: All-all (left): As the number of nodes (N) grow, the total number of updates transmitted increases $O(N^2)$. Halton-sequence exchange of model updates for $N = 6$ (Right). Each ith machine sends updates to $\log(N)$ (2 for $N = 6$) nodes (to $N/2 + i$ and $N/4 + i$). As number of nodes(N) grow, total number of updates transmitted increases $O(N \log N)$.**

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 and each machine can compute gradients asynchronously.

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. Finally, we build a vector library over dstorm to expose a vector abstraction over shared memory and to provide additional communication or representation optimizations (such as compression).

Communication Efficiency in MALT : When MALT is trained using the peer-to-peer approach, each machine can send 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 [5] or tree style all-reduce [3] may reduce the communication cost. However, this increases the latency by a factor of the height of the tree. Furthermore, this makes recovery complex if the intermediate nodes are affected by stragglers or failures.

In MALT, we propose *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 the developer needs to ensure that the communication graph of all nodes is *strongly connected*. Naïvely or randomly selecting what nodes to send updates to may either result in a partitioned graph of nodes or may propagate updates that may be too stale. This can adversely affect the convergence in parallel learning models.

In MALT we provide a pre-set sequence to ensure uniform gradient propagation. For every node, that 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. 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 $\text{MALT}_{\text{Halton}}$. Figures 2 shows the all-to-all and Halton communication schemes.

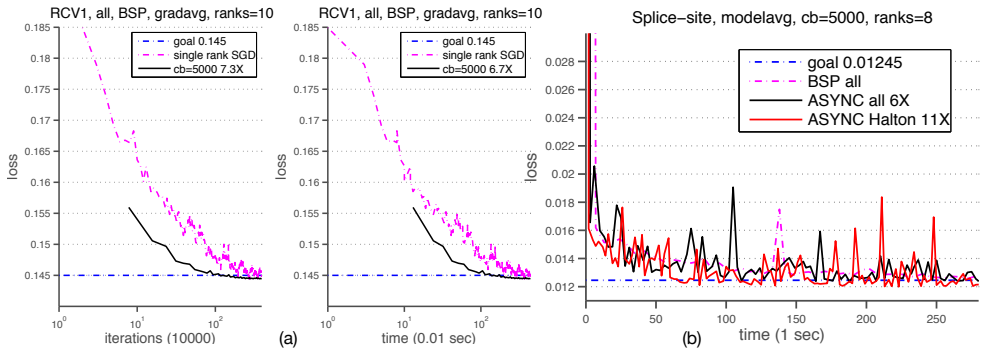

Figure 3: **This figure shows convergence for RCV1 workload for MALT$_{all}$ with a single machine workload. We find that MALT$_{all}$ converges quicker to achieve the desired accuracy.**

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. The key idea with MALT$_{Halton}$ is to balance the communication (sending updates) with computation (computing gradients, applying received gradients). However, the node-graph needs to be strongly connected otherwise the individual model updates from a node may not propagate to remaining nodes, and the models may diverge from one another.

3 Evaluation

We use MALT to modify SVM-SGD, Hogwild (matrix factorization) and RAPID (neural networks). We perform all experiments on 8 machine cluster. Each machine has an Intel Xeon 2.2Ghz CPU and 64 GB DDR3 DRAM, connected to one another via a Mellanox Connect-V3 56 Gbps infiniBand. To evaluate SVM, we use RCV1, the PASCAL suite (alpha, webspam, DNA) and splice-site datasets [2]. The compressed training set sizes are RCV1 – 333 MB (477 MB uncompressed), and splice-site – 110 GB (250 GB uncompressed). Please refer to [11] for detailed results on SVM-PASCAL, matrix factorization and neural networks and additional evaluation for developer efforts, and performance of different consistency protocols.

Speedup In this section, we compare the speedup of different datasets over a single machine and existing 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 over data per machine. For each of our experiments, we pick the desired final optimization goal as achieved by running a single-rank SGD [4]. Figure 3 (a) compares the speedup of MALT$_{all}$ for 10 ranks with a single-rank SGD for the RCV1 dataset, for a *communication batch size* or *cb size* of 5000. By *cb size* of 5000, we mean that every model processes 5000 examples and then propagates the model updates to other machines. By 10 *ranks*, we mean 10 processes, that span our eight machine cluster. For RCV1, we are unable to saturate the network and CPU with a single replica, and run multiple replicas on each machine. We obtain about 6.7X speedup over a single rank performance using MALT.

Network Optimizations Figure 3 (b) shows the model convergence for the splice-site dataset and the speedup over (Bulk Synchronous Parallelism) BSP-all in reaching the desired goal with 8 nodes. From the figure, we see that MALT$_{Halton}$ converges faster than MALT$_{all}$. Furthermore, we find that until convergence, each node in MALT$_{all}$ sends out 370 GB of updates, while MALT$_{Halton}$ only sends 34 GB of data for each machine. As the number of nodes increase, the logarithmic fan-out of MALT$_{Halton}$ should result in lower amounts of data sent and faster convergence. MALT $_{Halton}$ 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$_{all}$ configuration may provide similar or better results over MALT $_{Halton}$.

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

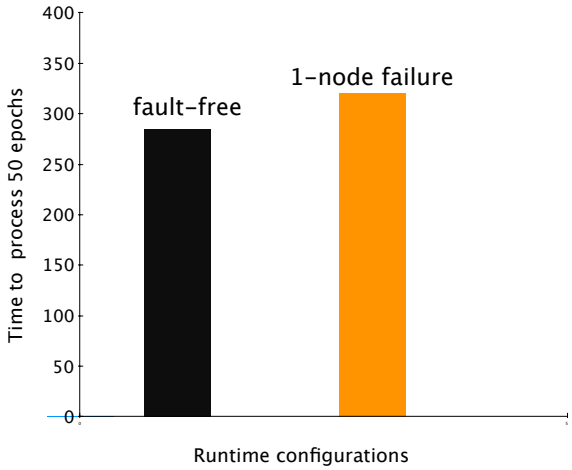

Figure 4: **This figure shows the time taken to converge for DNA dataset with 10 nodes in fault-free and a single process failure case. We find that MALT is able to recover from the failure and train the model.**

node from send/receive lists. We run MALT-SVM over ten ranks on eight nodes to train over the PASCAL-DNA [2] 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. 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 4 shows one instance of failure recovery, and the time to converge is proportional to the number of remaining nodes.

4 Lessons and Conclusions

To summarize, given a list of machines and MALT library, we demonstrate that one can parallelize ML algorithms, control the gradient-flow and synchrony. We briefly discuss lessons on deploying MALT on our clusters:

Cross-rack reduce: When performing a reduce operation across racks, an all-reduce primitive can be extremely slow. We find that using partial reduce with $\text{MALT}_{\text{Halton}}$, that is aware of the underlying rack topolgy can reduce these wait times.

Consistency: Synchronous training is implemented using the `barrier` construct where workers may spend considerable time waiting. Furthermore, `barrier` gives no information if the recipient has seen or processed the gradient. `barrier` is also inefficient for partial reduce ($\text{MALT}_{\text{Halton}}$). For these reasons, we provide the `notify-ack` mechanism, where each receiver acknowledges processing a gradient to the receivers. This gives stricter guarantees than `barrier` and can improve performance in some cases.

Data Loading: We find that loading data into memory consumes a significant portion of job times. For training tasks with significant CPU times, such as processing image data through deep networks, having a separate data loader that loads and sends data to various workers over infiniBand can be helpful. This allows overlapping data loading with model training, and removes the initial data load wait times.

RDMA support: MALT uses one-sided RDMA primitives that reduces network processing costs and transmission overhead using hardware support. 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.

References

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

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

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

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

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

[6] 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.

[7] 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.

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

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

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

[11] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. MALT: Distributed Data-Parallelism for Existing ML Applications. In *Proceedings of the 10th European conference on Computer systems*. ACM, 2015.

[12] 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.

[13] 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.

[14] C. Mu, A. Kadav, E. Kruus, D. Goldfarb, and M. R. Min. Random Walk Distributed Dual Averaging Method For Decentralized Consensus Optimization. In *Proceedings of the NIPS Optimization Workshop*, 2015.

[15] 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.