TuX^2: Distributed Graph Computation for Machine Learning

Wencong Xiao[†*], Jilong Xue[*◇], Youshan Miao[*], Zhen Li[†*], Cheng Chen[*],
Ming Wu[*], Wei Li[†], Lidong Zhou[*]
[†]*SKLSDE Lab, Beihang University,* [*]*Microsoft Research,* [◇]*Peking University*

Abstract

TuX^2 is a new distributed graph engine that bridges graph computation and distributed machine learning. TuX^2 inherits the benefits of an elegant graph computation model, efficient graph layout, and balanced parallelism to scale to billion-edge graphs; we extend and optimize it for distributed machine learning to support heterogeneity, a Stale Synchronous Parallel model, and a new *MEGA* (Mini-batch, Exchange, GlobalSync, and Apply) model.

We have developed a set of representative distributed machine learning algorithms in TuX^2, covering both supervised and unsupervised learning. Compared to implementations on distributed machine learning platforms, writing these algorithms in TuX^2 takes only about 25% of the code: Our graph computation model hides the detailed management of data layout, partitioning, and parallelism from developers. Our extensive evaluation of TuX^2, using large data sets with up to 64 billion edges, shows that TuX^2 outperforms state-of-the-art distributed graph engines PowerGraph and PowerLyra by an order of magnitude, while beating two state-of-the-art distributed machine learning systems by at least 48%.

1 Introduction

Distributed graph engines, such as Pregel [30], PowerGraph [17], and PowerLyra [7], embrace a *vertex-program* abstraction to express iterative computation over large-scale graphs. A graph engine effectively encodes an index of the data in a graph structure to expedite graph-traversal-based data access along edges, and supports elegant graph computation models such as Gather-Apply-Scatter (GAS) for ease of programming. A large body of research [7, 18, 22, 24, 32, 33, 36, 39, 46, 47] has been devoted to developing highly scalable and efficient graph engines through data layout, partitioning, scheduling, and balanced parallelism. It has been shown that distributed graph engines can scale to graphs with more than a trillion edges [10, 43, 38] for simple graph algorithms such as PageRank.

Early work on graph engines (e.g., GraphLab [29]) was motivated by machine learning, based on the observation that many machine learning problems can be modeled naturally and efficiently with graphs and solved by iterative convergence algorithms. However, most subsequent work on graph engines adopts a simplistic graph computation model, driven by basic graph benchmarks such as PageRank. The resulting graph engines lack flexibility and other key capabilities for efficient distributed machine learning.

We present TuX^2, a distributed graph engine for machine learning algorithms expressed in a graph model. TuX^2 preserves the benefits of graph computation, whlie also supporting the Stale Synchronous Parallel (SSP) model [20, 11, 42, 13], a heterogeneous data model, and a new *MEGA* (Mini-batch, Exchange, GlobalSync, and Apply) graph model for efficient distributed machine learning. We evaluate the performance of TuX^2 on a distributed cluster of 32 machines (with over 500 physical cores) on both synthetic and real data sets with up to 64 billion edges, using representative distributed machine learning algorithms including Matrix Factorization (MF) [16], Latent Dirichlet Allocation (LDA) [45], and Block Proximal Gradient (BlockPG) [27], covering both supervised and unsupervised learning. The graph model in TuX^2 significantly reduces the amount of code (by 73–83%) that developers need to write for the algorithms, compared to the state-of-the-art distributed machine learning platforms such as Petuum [20, 44] and Parameter Server [26]. It also enables natural graph-based optimizations such as vertex-cut for achieving balanced parallelism. Our evaluation shows that TuX^2 outperforms state-of-the-art graph engines PowerGraph and PowerLyra by more than an order of magnitude, due largely to our heterogeneous MEGA graph model. TuX^2 also beats Petuum and Parameter Server by at least 48%

thanks to a series of graph-based optimizations.

As one of our key contributions, TuX^2 bridges two largely parallel threads of research, graph computation and parameter-server-based distributed machine learning, in a unified model, advancing the state of the art in both. TuX^2 significantly expands the capabilities of graph engines in three key dimensions: data representation and data model, programming model, and execution scheduling. We propose a set of representative machine learning algorithms for evaluating graph engines on machine learning applications, guiding graph engines towards addressing the real challenges of distributed machine learning and thereby becoming more widely used in practice. We have also, through extensive evaluation on real workloads at scale, shown significant benefits in programmability, scalability, and efficiency for graph computation models in distributed machine learning.

The rest of the paper is organized as follows. §2 offers an overview of graph computation and machine learning, highlighting their connections. §3 describes TuX^2's design. §4 presents three machine learning algorithms, detailing how they are expressed and realized in TuX^2; §5 discusses the implementation and evaluation of TuX^2. We discuss related work in §6 and conclude in §7.

2 Graphs for Machine Learning

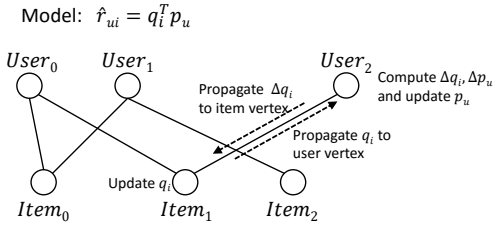
In this section, we highlight the benefits of abstraction into a graph model, show how a large class of machine learning algorithms can be mapped to graph models, and outline why existing graph engines fall short of supporting those algorithms in expressiveness and efficiency.

Graph parallel abstraction. A graph parallel abstraction models data as a graph $G = \{V, E\}$ with V the set of vertices and E the set of edges. A vertex-program P is provided to execute in parallel on each vertex $v \in V$ and interact with neighboring instances $P(u)$, where $(u, v) \in E$. The vertex-program often maintains an application-specific state associated with vertices and with edges, exchanges the state values among neighboring vertices, and computes new values during graph computation. It typically proceeds in iterations and, when a Bulk Synchronous Parallel (BSP) model is used, introduces a synchronization barrier at the end of each iteration. By constraining the interactions among nodes of a vertex-program using a graph model, this abstraction lets the underlying system encode an index of the data as a graph structure to allow fast data access along edges. Many existing state-of-the-art graph engines have adopted this parallel vertex-program approach, though the actual form of vertex-program design might vary. As a representative graph model, the GAS model proposed in PowerGraph [17] defines three phases of a vertex-program: Gather, Apply, and Scatter. For each vertex u, the *gather* phase collects information about neighbor vertices and edges of u through a generalized sum function that is commutative and associative. The result of this phase is then used in the *apply* phase to update u's state. Finally, the *scatter* phase uses u's new state to update its adjacent edges.

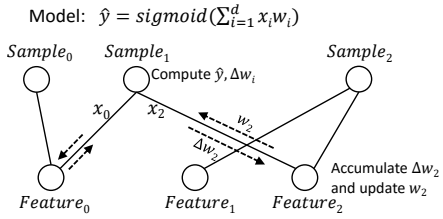
With a graph model like GAS, a graph algorithm can be succinctly expressed in three functions, without having to worry about managing data layout and partitioning, or about scheduling parallel executions on multiple cores and multiple machines. A graph engine can then judiciously optimize data layout for efficient graph data access, partition the data in a way that reduces cross-core or cross-server communication, and achieve balanced parallelism for scaling and efficiency. For example, PowerGraph introduces vertex-cut to achieve balanced partitioning of graph data, resulting in improved scalability even for power-law graphs. In our experience, these optimizations are effective for machine learning algorithms; further, they need only be implemented once per engine rather than redundantly for each algorithm.

Machine learning on graphs. Machine learning is widely used in web search, recommendation systems, document analysis, and computational advertising. These algorithms learn models by training on data samples consisting of features. The goal of machine learning can often be expressed via an *objective function* with *parameters* that represent a model. This objective function captures the properties of the learned model, such as the error it incurs when predicting the probability that a user will click on an advertisement given that user's search query. The learning algorithm typically minimizes the objective function to obtain the model. It starts from an initial model and then iteratively refines the model by processing the training data, possibly multiple times.

Many machine learning problems can be modeled naturally and efficiently with graphs and solved by iterative convergence algorithms. For example, the Matrix Factorization (MF) algorithm [16], often used in recommendation systems, can be modeled as a computation on a bipartite user-item graph where each vertex corresponds to a user or an item and each edge corresponds to a user's rating of an item. As another example, a topic-modeling algorithm like LDA performs operations on a document-word graph where documents and words are vertices. If a document contains a word, there is an edge between them; the data on that edge are the topics of the word in the document. For many machine learning algorithms described as computations on a sparse matrix, the computation can often be easily transformed to operations on a graph representation of the sparse matrix. For example,



(a) Matrix Factorization



(b) Logistic Regression

Figure 1: Examples of machine learning on graphs.

in Logistic Regression (LR) the parameters of the model are maintained in a weight vector with each element being the weight of the corresponding feature. Each training sample is a sparse feature vector with each element being the value of a specific feature. The entire set of training samples can be treated as a sparse matrix with one dimension being the samples and the other being the features. If a sample i contains a value for feature j, the element (i, j) of the matrix is the value. Therefore, the data can also be modeled as a graph with samples and features being vertices. Weights are the data associated with feature vertices, and the feature values in each training sample are the data on edges. Figure 1 illustrates how MF and LR are modeled by graphs.

Gaps. Even though these machine learning algorithms can be cast in graph models, we observe gaps in current graph engines that preclude supporting them naturally and efficiently. These gaps involve data models, programming models, and execution scheduling.

Data models: The standard graph model assumes a homogeneous set of vertices, but the graphs that model machine learning problems often naturally have different types of vertices playing distinct roles (e.g., user vertices and item vertices). A heterogeneity-aware data model and layout is critical to performance.

Programming models: For machine learning computations, an iteration of a graph computation might involve multiple rounds of propagations between different types of vertices, rather than a simple series of GAS phases. The standard GAS model is unable to express such computation patterns efficiently. This is the case for LR, where the data (weights) of the feature vertices are first propagated to sample vertices to compute the objective function, with the gradients propagated back to feature vertices to update the weights. Implementing this process in GAS would unnecessarily require two consecutive GAS phases, with two barriers.

Execution scheduling: Machine learning frameworks have been shown to benefit from the Stale Synchronous Parallel (SSP) model, a relaxed consistency model with bounded staleness to improve parallelism. This is because machine learning algorithms typically describe the process to converge to a "good" solution according to an objective function and the convergence process itself is robust to variations and slack that can be leveraged to improve efficiency and parallelism. The mini-batch is another important scheduling concept, often used in stochastic gradient descent (SGD), where a small batch of samples are processed together to improve efficiency at the expense of slower convergence with respect to the number of iterations. Mini-batch size is an important parameter for those algorithms and needs to be tuned to find the best balance. Graph engines typically operate on individual vertices [29], or define an "iteration" or a batch on the entire graph [30], while mini-batches offer the additional flexibility to be in between.

TuX^2 therefore supports and optimizes for heterogeneity in the data model, advocates a new graph model that allows flexible composition of stages, and supports SSP and mini-batches in execution scheduling.

3 TuX^2 Design

TuX^2 is designed to preserve the benefits of graph engines while extending their data models, programming models, and scheduling approaches in service to distributed machine learning.

TuX^2 uses the *vertex-cut* approach, in which the edge set of a (high-degree) vertex can be split into multiple partitions, each maintaining a replica of the vertex. One of these replicas is designated the *master*; it maintains the master version of the vertex's data. All remaining replicas are called *mirrors*, and each maintains a local cached copy. We adopt vertex-cut because it is proven effective in handling power-law graphs and it connects naturally to the parameter-server model [26, 11]: The master versions of all vertices' data can be treated as the (distributed) global state stored in a parameter server. In each partition, TuX^2 maintains vertices and edges in separate arrays. Edges in the edge array are grouped by source vertex. Each vertex has an index giving the offset of its edge-set in the edge array. Each edge contains information such as the id of the partition containing the destination vertex and the index of that vertex in the corresponding vertex array. This graph data structure is optimized for traversal and outperforms vertex indexing using a lookup table.

Each partition is managed by a process that logically plays both a worker role, to enumerate vertices in the partition and propagate vertex data along edges, and a server role, to synchronize states between mirror vertices and their corresponding masters. Inside a process, TuX^2 uses multiple threads for parallelization and assigns both the server and worker roles of a partition to the same thread. Each thread is then responsible for enumerating a subset of mirror vertices for local computation and maintaining the states of a subset of master vertices in the partition owned by the process. Figure 2 shows how data are partitioned, stored, and assigned to execution roles in TuX^2.

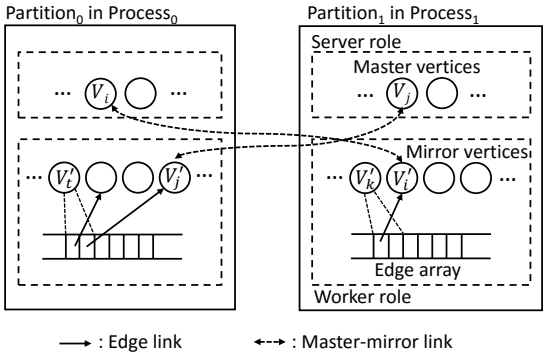


Figure 2: Graph placement and execution roles in TuX^2

3.1 Heterogeneous Data Layout

While traditional graph engines simply assume a homogeneous graph, TuX^2 supports heterogeneity in multiple dimensions of data layout, including vertex type and partitioning approach; it even supports heterogeneity between master and mirror vertex data types. Support for heterogeneity translates into significant performance gains (40%) in our evaluation (§5.2).

We highlight optimizations on bipartite graphs because many machine learning problems map naturally to bipartite graphs with two disjoint sets of vertices, e.g., users and items in MF, features and samples in LR, and so on. The two sets of vertices therefore often have different properties. For example, in the case of LR, only feature vertices contain a weight field and only sample vertices contain a target label field. And, in variants of LR like BlockPG [27], feature vertices also maintain extra history information. TuX^2 therefore allows users to define different vertex types, and places different types of vertex in separate arrays. This leads to compact data representation, thereby improving data locality during computation. Furthermore, different vertex types may have vastly different degrees. For example, in a user-item graph, item vertices can have links to thousands of users but user vertices typically only link to tens of items. TuX^2 uses bipartite-graph aware partitioning algorithms proposed in PowerLyra [7] and BiGraph [8] so that only high-degree vertices have mirror versions.

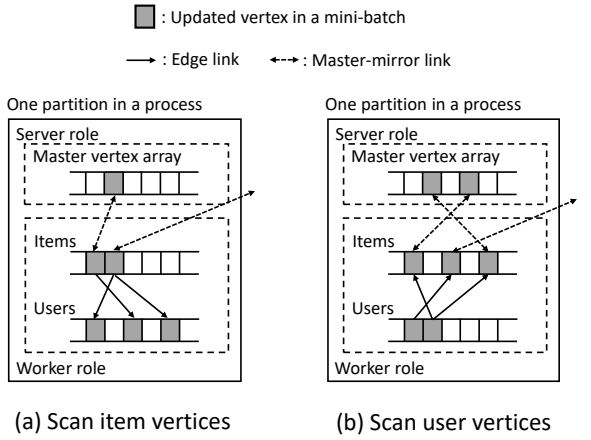


Figure 3: Example showing how separate vertex arrays are used for an MF bipartite graph. Edge arrays are omitted for conciseness.

In a bipartite graph, TuX^2 can enumerate all edges by scanning only vertices of one type. The choice of which type to enumerate sometimes has significant performance implications. Scanning the vertices with mirrors in a mini-batch tends to lead to a more efficient synchronization step as long as TuX^2 can identify the set of mirrors that have updates to synchronize with their masters, because these vertices are placed contiguously in an array. In contrast, if TuX^2 scans vertices without mirrors in a mini-batch, the mirrors that get updated for the other vertex type during the scan will be scattered and thus more expensive to locate. TuX^2 therefore allows users to specify which set of vertices to enumerate during the computation.

Figure 3 illustrates how TuX^2 organizes vertex data for a bipartite graph, using MF on a user-item graph as an example. Because user vertices have much smaller degree in general, only item vertices are split by vertex-cut partitioning. Therefore, a master vertex array in the server role contains only item vertices, and the worker role only manages user vertices. This way, there are no mirror replicas of user vertices and no distributed synchronization is needed. In the worker role, the mirrors of item and user vertices are stored in two separate arrays.

The figure also shows the benefit of scanning item vertices in a mini-batch. As shown in Figure 3(a), this leads to updated mirror vertices being located contiguously in an item vertex array. TuX^2 can therefore easily identify them for master-mirror synchronization by simply rescanning the corresponding range of that array. In contrast, scanning user vertices in a mini-batch would require an extra index structure to identify the mirror up-

dates. This is because they are scattered in an item vertex array as shown in Figure 3(b). Such an index structure would introduce extra overhead.

Another type of heterogeneity comes from different computations performed on master and mirror replicas of vertices, which may require different data structures for synchronization efficiency. For example, the BlockPG algorithm accesses and updates weights of a block of features in a mini-batch, while the objective function computed at sample vertices might depend on weights of features not in this block. This leads to auxiliary feature vertex attributes on mirrors, to record the historical deltas of feature weights to compute the value of the objective function incrementally. However, this delta attribute is not needed on masters, and hence does not need to be exchanged during synchronization. Similarly, a master vertex also maintains some extra attributes that are not needed on mirrors. TuX^2 therefore allows users to define different data structures for the master and mirror replicas of the same vertex.

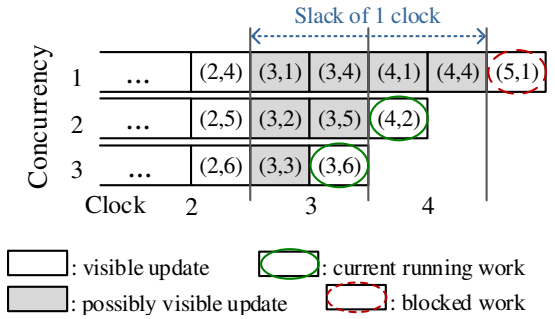


Figure 4: SSP with bounded staleness. A block labeled (i, j) indicates a task with id j in clock i.

3.2 Scheduling with SSP

TuX^2 supports the Stale Synchronous Parallel (SSP) model [11] with bounded staleness and mini-batches. SSP is based on the notion of *work-per-clock*, where a clock corresponds to an iteration over a mini-batch executed by a set of concurrent tasks. Iterative batch processing can be considered as a special case in which each iteration uses all input data. SSP introduces an explicit *slack* parameter, which specifies in clocks how stale a task's view of the globally shared state can be. The slack thus dictates how far ahead of the slowest task any task may progress. With a slack of s, a task at clock t is guaranteed to see all updates from clocks 1 to $t-s-1$, and it may see the updates from clocks $t-s$ to $t-1$. Figure 4 illustrates an SSP execution with a slack of 1.

TuX^2 executes each iteration on a mini-batch with a specified size. Each worker first chooses a set of vertices or edges as the current mini-batch to execute on. After the execution on the mini-batch finishes, TuX^2 acquires another set of vertices or edges for the next mini-batch, often by continuing to enumerate contiguous segments of vertex or edge arrays. TuX^2 supports SSP in the mini-batch granularity. It tracks the progress of each mini-batch iteration to enable computation of clocks. A worker considers clock t completed if the corresponding mini-batch is completed on all workers (including synchronizations between masters and mirrors) and if the resulting update has been applied to and reflected in the state. A worker can execute a task at clock t only if it knows that all clocks up to $t-s-1$ have completed, where s is the allowed slack.

3.3 MEGA Model in TuX^2

TuX^2 introduces a new stage-based *MEGA* model, where each stage is a computation on a set of vertices and their edges in a graph. Each stage has user-defined functions (UDF) to be applied on the vertices or edges accessed during it. TuX^2 supports four types of stage: Mini-batch, Exchange, GlobalSync, and Apply (hence the name MEGA); it allows users to construct an arbitrary sequence of stages. The engine is responsible for scheduling parallel executions of the UDFs on multiple cores and/or machines in each stage.

The MEGA model preserves the simplicity of the GAS model, while introducing additional flexibility to address deficiencies of the GAS model in supporting machine learning algorithms. For example, in algorithms such as MF and LDA, processing an edge involves updating both vertices. This requires two GAS phases, but can be accomplished in one Exchange phase in our model. For LR, the vertex data propagations in both directions should be followed by an *Apply* phase, but no *Scatter* phases are necessary; this can be avoided in the MEGA model because MEGA allows an arbitrary sequence of stages. We elaborate on the different types of stages next.

Exchange: This stage enumerates edges of a set of vertices, taking a UDF with the following signature:

$$Exchange(D_u, a_u, D_{(u,v)}, a_{(u,v)}, D_v, a_v, \tau)$$

`Exchange()` is performed on each enumerated edge. D_u and D_v are the data on vertices u and v, respectively. $D_{(u,v)}$ is the data associated with the edge (u,v). a_u, a_v, and $a_{(u,v)}$ are the corresponding accumulated deltas of the vertex and edge data, and τ is a user-defined shared context associated with each worker thread and maintained during the execution of the entire computation. All these parameters are allowed to be updated in this UDF. Users can use it to generate new accumulated deltas for vertices and edges, or to update their states directly. Given the vertex-cut graph placement, `Exchange()` may only update the mirror version data

(i.e., the local states) of the vertices. Users can also use τ to compute and store some algorithm-specific non-graph context data, which may be shared through global aggregation. By default, vertices not specified for enumeration are protected by vertex-level locks, but TuX^2 also allows users to implement their own lock-free semantics for some applications [14, 21, 37]. This stage is more flexible than the Gather/Scatter phases in the GAS model in that it does not imply or enforce a direction of vertex data propagation along an edge, and it can update the states of both vertices in the same UDF. It thereby improves efficiency for algorithms such as LDA and MF.

Apply: This stage enumerates a set of vertices and synchronizes their master and mirror versions. For each vertex, the master accumulates deltas from the mirrors, invokes $Apply(D_u, a_u, \tau)$ to update its global state, then updates the states on the mirrors. To support heterogeneity between master and mirror, TuX^2 allows users to define a base class `VertexDataSync` for the global state of a vertex that needs to be synchronized; masters and mirrors can define different subclasses, each inheriting from the base class, to include other information. The engine synchronizes only the data in `VertexDataSync` between master and mirror vertices.

GlobalSync: This stage is responsible for synchronizing the contexts τ across worker threads and/or aggregating the data across a set of vertices. There are three UDFs associated with this stage:

$$\tau^{i+1} \leftarrow Aggregate(D_v, \tau^i)$$

$$\tau^l \leftarrow Combine(\tau^i, \tau^j)$$

$$\tau^{i+1} \leftarrow Apply(\tau^i)$$

`Aggregate()` aggregates data across vertices into worker context τ. `Combine()` aggregates context τ across workers into a special worker, which maintains multiple versions of context τ for different clocks to support SSP. `Apply()` finalizes the globally aggregated τ (e.g., for re-scaling). After the execution of `Apply()`, the final aggregated τ is synchronized back to all workers. If the `Aggregate()` function is not provided, this stage will aggregate and synchronize the contexts τ only across workers.

Mini-Batch: This is a composite stage containing a sequence of other stages; it defines the stages to be executed iteratively for each mini-batch. `MiniBatch` defines the mini-batch size in terms of the number of vertices or edges to enumerate in each mini-batch, and, in the case of bipartite graphs, which type of vertex to enumerate (see examples in §4).

```
void StageSequenceBuilder(ExecStages){
    ExecStages.Add(ExchangeStage);
    ExecStages.Add(ApplyStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(a) MF stage sequence for a batch

```
void StageSequenceBuilder(ExecStages){
    val mbStage = new MiniBatchStage;
    mbStage.SetBatchSize(1000, asEdge);
    mbStage.Add(ExchangeStage);
    mbStage.Add(ApplyStage);

    ExecStages.Add(mbStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(b) MF stage sequence for a mini-batch

```
//ExchangeStage::
Exchange(v_user, v_item, edge,
         a_user, a_item, context){
    val pred = PredictRating(v_user,v_item);
    val loss = pred - edge.rating;
    context.loss += loss^2;
    (a_user,a_item) +=
        Gradient(loss,v_user,v_item);
}

//ApplyStage::
Apply(ver, accum, ctx){
    //Apply accumulated gradient
    ver.data += accum;
}

//GlobalSyncStage::
Combine(ctx1, ctx2){
    ctx.loss = ctx1.loss + ctx2.loss;
    return ctx;
}
```

(c) MF UDFs for each stage

Figure 5: Programming MF with the MEGA model

4 ML Algorithms on TuX^2

In this section, we detail how three machine learning algorithms are expressed and implemented in TuX^2.

Matrix Factorization (MF). MF, commonly used in recommendation systems, aims to decompose an adjacency matrix $M_{|U|\times|I|}$, where U is the set of users, I is the set of items, and the entry (u,i) is user u's rating on item i, into two matrices L and R, making M approximately equal to $L \times R$. TuX^2 models training data as a bipartite graph with users and items being vertices and user-item ratings being edges, and solves MF using SGD [16].

Figure 5 illustrates how MF is implemented in the MEGA model. `StageSequenceBuilder()` builds the stage sequence for each MF iteration. For MF in batch mode (Figure 5a), an iteration is composed of `ExchangeStage`, `ApplyStage`, and `GlobalSyncStage`. `Exchange()` (Figure 5c) computes the gradients of the loss function given a user and an item, and accumulates the gradients into `a_user` and `a_item`, respectively. In `Apply()`, the accumu-

```
//ExchangeStage::
Exchange(v_doc, v_word, edge,
         a_doc, a_word, context){
    val old_topic = edge.topic
    val new_topic = GibbsSampling(context,
                        v_doc, v_word)
    edge.topic = new_topic;

    //topic accumulator
    a_doc[old_topic]--;
    a_doc[new_topic]++;
    a_word[old_topic]--;
    a_word[new_topic]++;
    //update topic summary
    context.topic_sum[old_topic]--;
    context.topic_sum[new_topic]++;
}

//ApplyStage::
Apply(ver, accum, ctx){
    //Apply accumulated topic changes
    ver.topics += accum;
}

//GlobalSyncStage::
Combine(ctx1, ctx2){
    ctx.topic_sum
        = ctx1.topic_sum + ctx2.topic_sum;
    return ctx;
}
```

Figure 6: Programming LDA with the MEGA model

lated gradient is used to update the data of a vertex (a user or an item). `Combine()` sums the losses to evaluate convergence. For the mini-batch version (Figure 5b), only `ExchangeStage` and `ApplyStage` are performed per mini-batch, while `GlobalSyncStage` is conducted per iteration. The mini-batch size is set as the number of edges because each edge with its connected user and item vertices forms a training sample.

Latent Dirichlet Allocation (LDA). When applied to topic modeling, LDA trains on a set of documents to learn document-topic and word-topic distributions and thereby learn how to deduce any document's topics. TuX^2 implements SparseLDA [45], a widely used algorithm for large-scale distributed LDA training. In our graph model, vertices represent documents and words, while each edge between a document and a word means the document contains the word.

LDA's stage sequence is the same as MF's. Figure 6 shows the UDFs of the stages. Each edge is initialized with a randomly assigned topic. Each vertex (document or word) maintains a vector to track its distribution of all topics. The topic distribution of a vertex is the topic summary of the edges it connects. We also have a global topic summary maintained in a shared context. The computation iterates over the graph following the stage sequence until convergence. `Exchange()` performs Gibbs sampling [19] on each edge to compute a new topic for the edge. The new edge topic also changes the topic distributions of the vertices, as well as the topic

```
void StageSequenceBuilder(ExecStages){
    val mbStage = new MiniBatchStage;
    mbStage.SetBatchSize(1000,asVertex,
                         "feature");
    mbStage.Add(ExchangeStage0);
    mbStage.Add(ApplyStage);
    mbStage.Add(ExchangeStage1);

    ExecStages.Add(mbStage);
    ExecStages.Add(GlobalSyncStage);
}
```

(a) BlockPG stage sequence

```
//ExchangeStage0::
Exchange0(v_feature, v_sample, edge,
          a_feature, a_sample, ctx){
    (a_feature.g, a_feature.u) +=
        FeatureGradient(v_feature, v_sample)
}

//ExchangeStage1::
Exchange1(v_feature, v_sample, edge,
          a_feature, a_sample, ctx){
    v_sample.dual *=
        SampleDual(v_feature, v_sample)
}

//ApplyStage::
Apply(v_feature, a_feature, ctx){
    v_feature.weight +=
        SolveProximal(v_feature,a_feature,ctx);
}

//GlobalSyncStage::
Aggregate(ver, ctx){
    ctx.obj += CalcObj(ver);
}
Combine(ctx1, ctx2){
    ctx.obj = ctx1.obj + ctx2.obj;
    return ctx;
}
```

(b) BlockPG UDFs for stages

Figure 7: Programming BlockPG with the MEGA model

summary in the shared context; these changes are accumulated. `Apply()` applies the aggregated topic changes for each vertex. `Combine()` synchronizes the global topic summary among all workers.

Block Proximal Gradient (BlockPG). BlockPG [26, 27, 28] is a state-of-the-art logistic regression algorithm. It is modeled as a bipartite graph with features and samples as vertices and an edge between a feature and a sample vertex indicating that the sample contains the feature.

Figure 7b shows the pseudocode of BlockPG's UDFs. BlockPG randomly divides features into blocks and enumerates each block as a mini-batch. Each mini-batch involves two `Exchange*()` stages with an `Apply()` stage in between. `Exchange0()` calculates and accumulates for each edge both the gradient and the diagonal part of the second derivative for the corresponding feature vertex. `Apply()` then synchronizes the vertices' accumulated values into the master feature vertices to update their weights using a proximal operator [34]. Then, `Exchange1()` uses the new weights of features

to compute new states of samples. Note that BlockPG does not need `Apply()` for sample vertices. TuX^2 therefore optimizes `Apply()` by setting the mini-batch size in terms of the number of feature vertices (as shown in Figure 7a) and partitioning the graph to not cut the sample vertices. Also, only the weight value needs to be synchronized between master and mirror feature vertices. TuX^2 allows a master vertex to maintain private data that does not get synchronized to mirrors, e.g., information for the proximal operator.

5 Implementation and Evaluation

We implemented TuX^2 in about 12,000 lines of C++ code. It can be built and deployed on both Linux and Windows clusters with each machine running one TuX^2 process. The system is entirely symmetric: All the processes participating in the computation are peers executing the same binary. TuX^2 takes graph data in a collection of text files as input. Each process picks a separate subset of those files and performs bipartite-graph-aware algorithms [7, 8] to partition the graph in a distributed way. Each partition is assigned to, and stored locally with, a process. The data in each partition are placed as they are loaded and used in computation. For interprocess communication, TuX^2 uses a network library that supports both RDMA and TCP.

In the rest of this section, we present detailed evaluation results to support our design choices and to demonstrate the benefits of supporting machine learning on graph engines. We compare TuX^2 with state-of-the-art graph systems PowerGraph [17, 4] and PowerLyra [7] and ML systems Petuum [20, 3] and Parameter Server [26, 2].

Experimental setup. We conduct most of our experiments on a commodity cluster with 32 servers. Each server is equipped with dual 2.6 GHz Intel Xeon E5-2650 processors (16 physical cores), 256 GB of memory, and a Mellanox ConnectX-3 InfiniBand NIC with 54 Gbps bandwidth. TuX^2 uses RDMA by default, but uses TCP when comparing to other systems for fairness.

To evaluate TuX^2, we have fully implemented the MF, LDA, and BlockPG algorithms introduced in §4, setting the feature dimension of MF to 50 and the topic count of LDA to 100 in all experiments. The algorithms are selected to be representative and cover a spectrum, ranging from computation-bound (e.g., LDA) to communication-bound (e.g., BlockPG). Table 1 lists the datasets that we use for evaluation. NewsData and AdsData are two real datasets used in production by Microsoft for news and advertisement. Netflix [6] is the largest public dataset that is available for MF. We also generate a larger synthesized dataset, which is the default dataset for MF experiments. All performance numbers in our experiments are calculated by averaging over 100 iterations; in all cases we observed very little variation.

Dataset name	# of users/ docs/samples	# of items/ words/features	# of edges
NewsData (LDA)	7.3 M	418.4 K	1.4 B
AdsData (BlockPG)	924.8 M	209.3 M	64.9 B
Netflix (MF)	480.2 K	17.8 K	100.5 M
Synthesized (MF)	30 M	1 M	6.3 B

Table 1: Datasets (K: thousand, M: million, B: billion).

Algorithm	ML systems	TuX^2	LOC reduction
MF (Petuum)	> 300	50	83%
LDA (Petuum)	> 950	252	73%
BlockPG (PS)	> 350	79	77%

Table 2: Algorithm implementations in lines of code, ML systems vs. TuX^2. (PS: Parameter Server)

5.1 Programmability

By providing a high-level MEGA graph model, TuX^2 makes it significantly easier to write distributed machine learning algorithms, relieving developers from handling the details of data organization, enumeration, partitioning, parallelism, and thread management. As one indication, Table 2 shows the significant reduction (73–83%) in lines of code (LOC) to implement the three algorithms in TuX^2, compared to the C++ implementations of the same algorithms on Petuum or Parameter Server. The lines of code are comparable to those written in the GAS model.

5.2 Managing ML Data as a Graph

Data layout. Data layout matters greatly in the performance of machine learning algorithms. Figure 8 compares the performance of BlockPG, MF, and LDA with two different layouts: one an array-based graph data layout in TuX^2 and the other a hash-table-based lay-

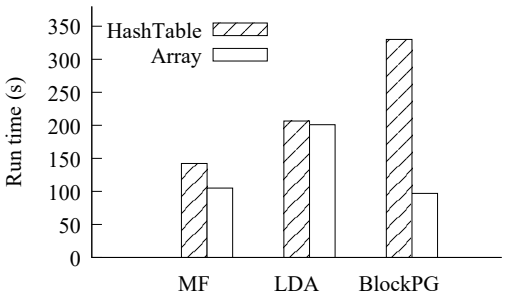


Figure 8: Effect of data layout (32 servers)

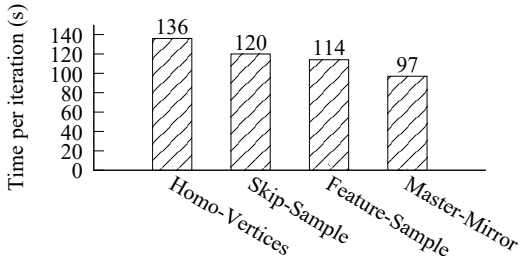


Figure 9: Effect of heterogeneity (BlockPG, 32 servers)

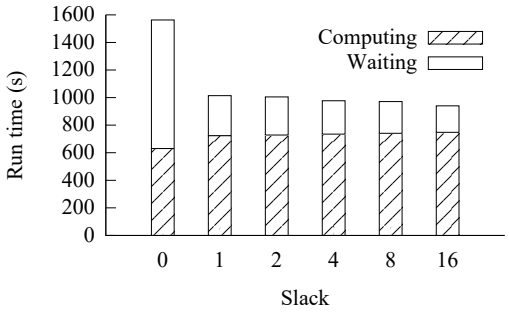


Figure 10: Run time, with breakdown, to converge to the same point under different slack (MF, 32 servers)

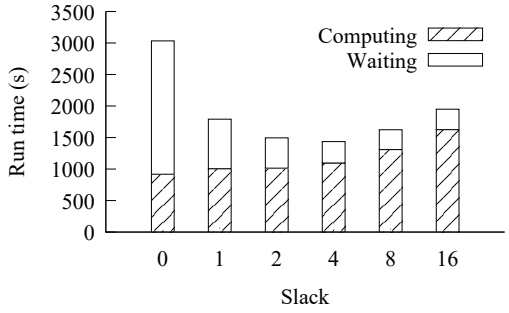


Figure 11: Run time, with breakdown ,to converge to the same point under different slack (BlockPG, 32 servers)

out often used in parameter-server-based systems (but implemented in TuX^2 for comparison). The y-axis is the average running time of one iteration for BlockPG, and of 10 iterations for MF and LDA to show the numbers on a similar scale. These results show that the graph layout improves performance by up to 2.4× over the hash-table-based layout. We observe smaller improvement in LDA because LDA involves more CPU-intensive floating-point computation, making data access contribute a smaller portion of overall run time.

Heterogeneity. Supporting heterogeneity in TuX^2 is critical to the performance of machine learning algorithms. We evaluate the benefits of supporting different dimensions of vertex heterogeneity using BlockPG on 32 servers. As shown in Figure 9, if we model all vertices as the same vertex type, each iteration takes 136 s (Homo-Vertices in the figure). For BlockPG, because only feature vertices have mirrors, we can specify to enumerate only feature vertices in each mini-batch and not track whether sample vertices are updated because they do not have mirrors to synchronize (see §3.1). This setup leads to a reduction of 16 s per iteration (Skip-Sample in the figure). Next, if we define different vertex data types for features and samples for a more compact representation, each iteration can save an additional 6 s (Feature-Sample in the figure). Finally, as discussed in §3.3, we can allow masters and mirrors to have different types and can indicate which data need to be synchronized. Doing this makes each iteration take only 97 s (Master-Mirror in the figure), a total performance improvement of 40% over the original homogeneous setting.

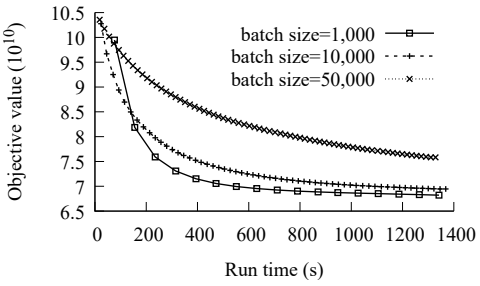
5.3 Extensions for Machine Learning

SSP slack and mini-batch size can be configured in TuX^2 to tune algorithm convergence.

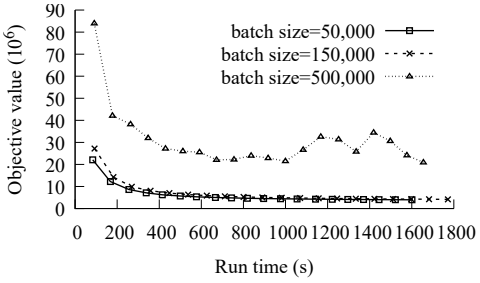
Stale Synchronous Parallel. TuX^2 supports the configuration of slack as a staleness bound for SSP, to allow users to tune the parameter for desirable convergence. The effect of slack varies by algorithm. For MF, as shown in Figure 10, the overall convergence accelerates as the slack increases. The breakdown confirms that increasing slack reduces waiting time, while increasing computing time only slightly, indicating that it takes about the same (or a slightly larger) number of iterations to reach the same convergence point. For BlockPG, however, as shown in Figure 11, computing time increases significantly as slack increases to 8 and 16, indicating that it is taking many more iterations for BlockPG to converge when slack is larger. A slack value of 4 is the optimal point in terms of overall execution time.

Mini-Batch. Mini-batch size is another important parameter that TuX^2 lets users tune, since it also affects convergence, as we show in this experiment for MF and BlockPG. (LDA is inherently a batch algorithm and does not support mini-batches.) Figure 12 shows the convergence (to objective values) over time with slack set to 16 on 32 servers. We show each iteration as a point on the corresponding curve to demonstrate the effect of mini-batch size on the execution time of each iteration.

For MF, as shown in Figure 12a, we see that convergence with a smaller mini-batch size (e.g., 1,000) is much faster than that with a larger one (e.g., 10,000). However, a smaller mini-batch size could introduce more frequent communication, slowing down the computation in each iteration significantly, as confirmed by more



(a) MF, 32 servers



(b) BlockPG, 32 servers

Figure 12: Convergence with varying mini-batch size

sparse iteration points on the curve for mini-batch size 1,000. This is why we also observe that convergence with mini-batch size 1,000 is worse than that with 10,000 during the first 180 s. Similar results can be observed for BlockPG in Figure 12b. For BlockPG, an improper batch size could even make it non-convergent, as is the case when batch size is 500,000.

5.4 System Performance

TuX^2 vs. PowerGraph and PowerLyra. We first compare TuX^2 with PowerGraph and its successor PowerLyra, which support a GAS-based MF implementation. Because PowerGraph and PowerLyra do not support SSP or mini-batch, for fairness we configure TuX^2 to use a batched MF implementation with no slack. We run MF on the Netflix dataset, and Figure 13 shows the performance comparison of TuX^2, PowerGraph, and PowerLyra with different numbers of servers (each with 16 threads).

The figure shows that, consistent with the results reported in PowerLyra [7], PowerLyra outperforms PowerGraph in the multi-server cases (by 1.6x) due to a better partitioning algorithm that leads to a lower vertex replication factor. TuX^2 outperforms both PowerGraph and PowerLyra by more than an order of magnitude. The huge performance gap is largely due to our flexible MEGA model. Specifically, in PowerGraph and PowerLyra, the computation per iteration for MF is composed of two GAS phases, one for updating the user vertices and the other for the item vertices. This introduces significant synchronization overhead and some unnecessary stages due to the constraints of the GAS model. In contrast, TuX^2 needs only an `ExchangeStage` and an `ApplyStage` for each iteration in the MEGA model. Our detailed profiling on one iteration in the 8-server experiment further shows that, while the Exchange phase (which calculates the gradients) in TuX^2 takes only 0.5 s, the corresponding Gather phase takes 1.6 s in the GAS model. The difference is mainly due to TuX^2's heterogeneous data layout. Furthermore, the extra phases (i.e., the two Scatter phases) needed in the GAS model take an additional 7.4 s.

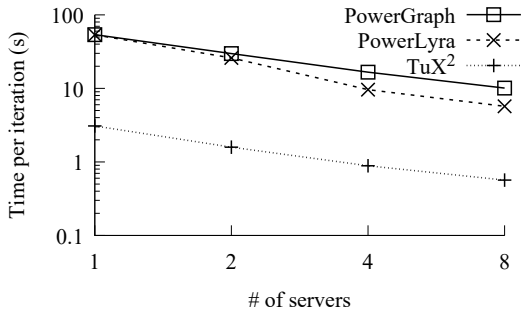
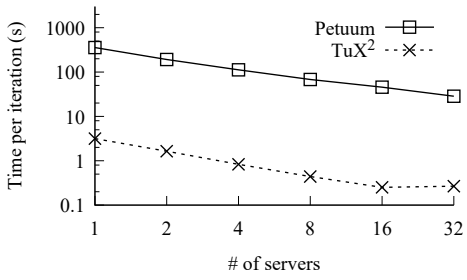


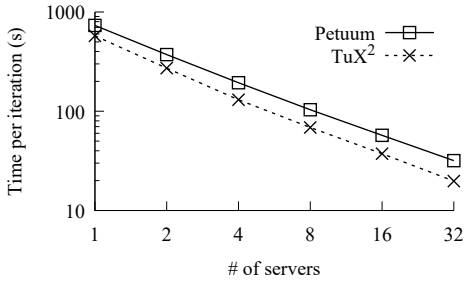
Figure 13: TuX^2 vs. PowerGraph/PowerLyra (MF, Netflix, log scale)

TuX^2 vs. machine learning systems. We compare TuX^2 with two state-of-the-art distributed machine learning systems: Petuum and Parameter Server (PS). We compare with Petuum using MF and LDA and we compare with PS using BlockPG. We have validated the results of our experiments to confirm that the algorithms in TuX^2 are the same as those in Petuum and in PS, respectively. We set slack to 0 as it produces a deterministic result every iteration, leading to the same convergence curve. We use time per iteration as our metric for comparison because the convergence per iteration is the same in this configuration. We evaluate on other configurations (not shown due to space constraints) and the results are similar. Compared with these systems, TuX^2, as a graph engine, inherits a series of graph-related optimizations for machine learning, such as efficient graph layout and balanced parallelism from vertex-cut partitioning. The following experiments evaluate these benefits of TuX^2.

Petuum: We compare Petuum and TuX^2 using MF and LDA because these two algorithms have been implemented in both Petuum and TuX^2. All the experiments are conducted on 32 servers with 16 threads per server. Figures 14a and 14b show the average execution time per

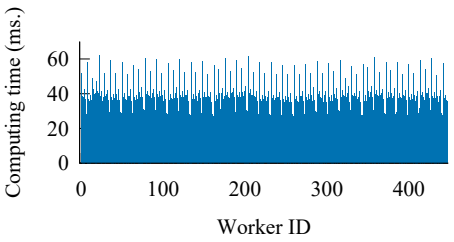


(a) MF, Netflix

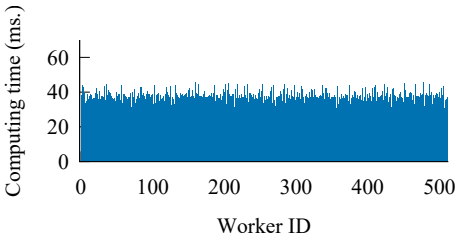


(b) LDA, NewsData

Figure 14: TuX^2 vs. Petuum (log scale)



(a) Imbalance in ParameterServer, 32 servers



(b) Balance in TuX^2, 32 servers

Figure 15: Mini-batch time across workers (BlockPG)

iteration of MF and LDA in Petuum and TuX^2 with different numbers of servers.

For MF, TuX^2 outperforms Petuum by two orders of magnitude, due to two main reasons. First, Petuum's distributed shared memory table, implemented in a multi-layer hash structure, introduces significant overhead, even compared with our hash-table baseline used in §5.2. Petuum also does fine-grained row-level version tracking, causing a staleness check to be triggered for every read/write operation. In contrast, TuX^2 uses a worker-level staleness check when an iteration/mini-batch starts, as described in §3.2. Second, in Petuum, both user data and item data contain model parameters and are stored in the parameter server. Updating either type involves communication with the parameter server. It is worth pointing out that this is not a fundamental problem with Petuum's design and can be fixed by moving user data off the parameter server. (Based on our communication with the Petuum authors, this issue has already been fixed in the new version [42], but the fixed version is not yet publicly available.) TuX^2 partitions the bipartite graph in such a way that only item vertices have mirrors, making the updates on user vertices efficient without unnecessary communication. This is a natural configuration in TuX^2 that users can enable effortlessly, easily avoiding the problem we observe in this version of Petuum. Note that TuX^2 does not scale well from 16 to 32 servers for MF. This is because Netflix data is small when divided among 32 servers $\times$ 16 threads (only around 3 MB per thread), so communication cost starts to dominate, limiting further scaling.

For LDA, the graph layout benefit is smaller compared to that for MF. Figure 14b shows that TuX^2 outperforms Petuum in LDA by 27% (1 server) to 61% (32 servers). This is consistent with the layout experiment in §5.2, which was also affected by the CPU-intensive nature of the floating-point computation in LDA.

Parameter Server (PS): We compare PS with TuX^2 using BlockPG, as it is implemented in both systems. We set mini-batch size to 300,000 for both. For PS, based on our experimentation on different thread-pool configurations, we find the best configuration uses 14 worker threads and 2 server threads per machine. We therefore use this configuration in our experiments. Due to the large data size (64B edges) involved, the experiment is performed only on 32 servers. When operating on the AdsData dataset, BlockPG takes 125 s on average per iteration on TuX^2, compared to 186 s on PS, which is 48% longer.

Unlike with Petuum, data layout is not the main reason that TuX^2 outperforms PS: PS carefully customizes its data structure for BlockPG, which is largely on par with TuX^2's general graph layout. Handling the imbalance caused by data skew (e.g., where some features exist in a large number of samples) makes the most difference in this case. Figure 15a shows the execution time of one representative mini-batch for all worker threads

in PS. A few threads are shown to work longer than the others, forcing those others to wait at synchronization points. In contrast, TuX^2 employs vertex-cut even for threads inside the same process, a built-in feature of the graph engine, to alleviate imbalance. Figure 15b shows that TuX^2 achieves balanced execution for all threads. While SSP slack could also help alleviate the effect of imbalance, it usually leads to slower convergence. Our vertex-cut optimization does not affect convergence and is strictly better.

6 Related Work

TuX^2 builds upon a large body of research on iterative graph computation and distributed machine learning systems. Pregel [30] proposes the vertex-program model, which has been adopted and extended in subsequent work, such as GraphLab [29] and PowerGraph [17]. TuX^2 uses the vertex-cut model proposed in PowerGraph and applies it also to partitioning within the process for balanced thread parallelism. It also incorporates bipartite-graph-specific partitioning schemes proposed in PowerLyra [7] and BiGraph [8] with further optimizations of computation. By connecting graph models to machine learning, our work makes advances in graph computation relevant to machine learning. This includes optimizations on graph layout, sequential data access, and secondary storage (e.g., GraphChi [24], Grace [36], XStream [39], Chaos [38], and FlashGraph [47]), distributed shared memory and RDMA (e.g., Grappa [32] and GraM [43]), and NUMA-awareness, scheduling, and load balancing (e.g., Galois [33], Mizan [22], and Polymer [46]).

TuX^2's design is influenced by parameter-server-based distributed machine learning, which was initially proposed and evolved to scale specific machine learning applications such as LDA [40, 5] and deep learning [15]. Petuum [13, 20, 42, 44] and Parameter Server [26] move towards general platforms, incorporate flexible consistency models, and improve scalability and efficiency. Petuum and its subsequent work on STRADS [23, 25] further propose to incorporate optimizations such as model parallelism, uneven convergence, and error tolerance. Many of these can be integrated into a graph engine like TuX^2, allowing users to benefit from both graph and machine learning optimizations, a future direction that we plan to explore further. We also see a trend where some of the design and benefits in graph systems have found their way into these machine learning systems (e.g., optimized layout in Parameter Server's BlockPG implementation and a high-level graph-model-like abstraction in STRADS), further supporting our theme of the convergence of the two. Parameter servers have also been proposed to support deep learning [9, 12, 15] and have been enhanced with GPU-specific optimizations in GeePS [12].

There is a large body of work on general distributed big-data computing platforms, including for example Mahout [1] on Hadoop and MLI [41] on Spark for machine learning on MapReduce-type frameworks. Piccolo [35] enables parallel in-memory computation on shared distributed, mutable state in a parameter-server-like interface. Another interesting research direction, pursued in GraphX [18] for example, explores how to support graph computation using a general data-flow engine. Naiad [31] introduces a new data-parallel dataflow model specifically for low-latency streaming and cyclic computations, which has also been shown to express graph computation and machine learning. Both have built known graph models, such as GAS, on top of their dataflow abstractions, while TuX^2 proposes a new graph model with important extensions for machine learning algorithms.

7 Conclusion

Through TuX^2, we advocate the convergence of graph computation and distributed machine learning. TuX^2 represents a critical step in this direction by showing not only the feasibility, but also the potential, of such convergence. We accomplish this by introducing important machine learning concepts to graph computation; defining a new, flexible graph model to express machine learning algorithms efficiently; and demonstrating the benefits through extensive evaluation on representative machine learning algorithms. Going forward, we hope that TuX^2 will provide a common foundation for further research in both graph computation and distributed machine learning, allowing more machine learning algorithms and optimizations to be expressed and implemented easily and efficiently at scale.

Acknowledgments

We thank our shepherd Adam Wierman and the anonymous reviewers for their valuable comments and suggestions. We are grateful to our colleague Jay Lorch, who carefully went through the paper and helped improve the quality of writing greatly. Jilong Xue was partially supported by NSF of China (No. 61472009).

References

[1] Apache foundation. mahout project. `http://mahout.apache.org`.

[2] ParameterServer. `https://github.com/dmlc/parameter_server`.

[3] Petuum v0.93. https://github.com/petuum/bosen/tree/release_0.93.

[4] PowerGraph v2.2. https://github.com/dato-code/PowerGraph.

[5] AHMED, A., ALY, M., GONZALEZ, J., NARAYANAMURTHY, S., AND SMOLA, A. J. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining* (2012), WSDM'12, ACM.

[6] BENNETT, J., AND LANNING, S. The Netflix Prize. In *Proceedings of KDD cup and workshop* (2007), vol. 2007.

[7] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys'15, ACM.

[8] CHEN, R., SHI, J., ZANG, B., AND GUAN, H. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014), APSys'14, ACM.

[9] CHILIMBI, T., SUZUE, Y., APACIBLE, J., AND KALYANARAMAN, K. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[10] CHING, A., EDUNOV, S., KABILJO, M., LOGOTHETIS, D., AND MUTHUKRISHNAN, S. One trillion edges: Graph processing at Facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015).

[11] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference* (2014), USENIX ATC'14, USENIX.

[12] CUI, H., ZHANG, H., GANGER, G. R., GIBBONS, P. B., AND XING, E. P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys'16, ACM.

[13] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G., AND XING, E. P. High-performance distributed ML at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015), AAAI'15, AAAI Press.

[14] DE SA, C. M., ZHANG, C., OLUKOTUN, K., RÉ, C., AND RÉ, C. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in Neural Information Processing Systems 28* (2015), NIPS'15, Curran Associates, Inc.

[15] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., AURELIO RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., AND NG, A. Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, NIPS'12. Curran Associates, Inc., 2012.

[16] GEMULLA, R., NIJKAMP, E., HAAS, P. J., AND SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2011), KDD'11, ACM.

[17] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX.

[18] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[19] GRIFFITHS, T. L., AND STEYVERS, M. Finding scientific topics. *Proceedings of the National Academy of Sciences 101*, suppl 1 (2004).

[20] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, NIPS'13. Curran Associates, Inc., 2013.

[21] JOHNSON, M., SAUNDERSON, J., AND WILLSKY, A. Analyzing Hogwild parallel Gaussian Gibbs sampling. In *Advances in Neural Information Processing Systems 26*, NIPS'13. Curran Associates, Inc., 2013.

[22] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys'13, ACM.

[23] KIM, J. K., HO, Q., LEE, S., ZHENG, X., DAI, W., GIBSON, G. A., AND XING, E. P. STRADS: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys'16, ACM.

[24] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX.

[25] LEE, S., KIM, J. K., ZHENG, X., HO, Q., GIBSON, G. A., AND XING, E. P. On model parallelization and scheduling strategies for distributed machine learning. In *Advances in Neural Information Processing Systems 27* (2014), NIPS'14, Curran Associates, Inc.

[26] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX.

[27] LI, M., ANDERSEN, D. G., AND SMOLA, A. J. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning* (2013).

[28] LI, M., ANDERSEN, D. G., SMOLA, A. J., AND YU, K. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems 27* (2014), NIPS'14, Curran Associates, Inc.

[29] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012).

[30] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD'10, ACM.

[31] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[32] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference* (2015), USENIX ATC'15, USENIX.

[33] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[34] PARIKH, N., AND BOYD, S. Proximal algorithms. *Found. Trends Optim. 1*, 3 (Jan. 2014).

[35] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX.

[36] PRABHAKARAN, V., WU, M., WENG, X., MCSHERRY, F., ZHOU, L., AND HARADASAN, M. Managing large graphs on multi-cores with graph awareness. In *2012 USENIX Annual Technical Conference* (2012), USENIX ATC'12, USENIX.

[37] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, NIPS'11. Curran Associates, Inc., 2011.

[38] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, ACM.

[39] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM.

[40] SMOLA, A., AND NARAYANAMURTHY, S. An architecture for parallel topic models. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010).

[41] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAM, J., PAN, X., GONZALEZ, J., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. MLI: An API for Distributed Machine Learning. In *2013 IEEE 13th International Conference on Data Mining* (2013), ICDM'13, IEEE.

[42] WEI, J., DAI, W., QIAO, A., HO, Q., CUI, H., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), SoCC'15, ACM.

[43] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), SoCC'15, ACM.

[44] XING, E. P., HO, Q., DAI, W., KIM, J.-K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), KDD'15, ACM.

[45] YAO, L., MIMNO, D., AND MCCALLUM, A. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009), KDD'09, ACM.

[46] ZHANG, K., CHEN, R., AND CHEN, H. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), PPoPP'15, ACM.

[47] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX.