# Summary of “TUX$^2$: Distributed Graph Computation for Machine Learning”

Peter Paquet (pmpaquet), Wenting Tan (wentingt)

## Problem and Motivation:

Many of the machine learning tasks in industry today have a graphical structure, such as recommendation systems, topic models, and click prediction. Naturally, one might assume that a distributed graph engine would be an excellent fit to work on such problems. Such engines exhibit many advantages, such as simple GAS-based (Gather, Apply, Scatter) programming model, certain graph-aware optimizations, and enhanced scalability to trillion-edge graphs. Unfortunately, there exist many gaps between advancements in graph engines and helpful features for solving machine learning problems, such as supporting heterogeneous datatypes, supporting Mini-Batch updating of the model, and supporting flexible consistency models.

## Hypothesis:

TUX$^2$ is a distributed graph engine for machine learning algorithms that are expressed in a graphical model. TUX$^2$ consolidates graph and machine learning research into one system using a unified model. TUX$^2$ extends the graph model for machine learning by utilizing Stale Synchronous Parallel (SSP) scheduling, supporting a heterogeneous data model, and the new MEGA (Mini-Batch, Exchange, GlobalSync, and Apply) graph model. As a result, TUX$^2$ outperforms many graph and machine learning systems running machine learning algorithms.

## Solution Overview:

As previously mentioned, TUX$^2$ is designed to preserve the benefits of graph engines while extending their data models, programing models, and scheduling approaches to distributed machine learning.

**System Architecture:**

1. TUX$^2$ uses the Vertex-Cut approach; The edge set of a high-degree (i.e. many connections) vertex can be split into several partitions, with each partition maintaining a replica of the vertex. One replica is denoted the *master* and all other replicas are denoted *mirrors*.
2. Not only does the vertex-cut approach efficiently process power-law (many real-world networks follow power-law degree distributions) graphs, but it also emulates the parameter-server model; The master versions of all vertices’ data can be treated as the global state stored in a parameter server.
3. This graphical data structure is optimized for traversal and outperforms vertex indexing using a lookup table.

**Data Model:**

1. In contrast to traditional graph engines, TUX$^2$ does not assume a homogeneous set of vertices. TUX$^2$ supports heterogeneity in vertex type, partitioning approach, and even heterogeneity between master and mirror vertex data types.
2. This is critical to performance, as many machine learning problems map to bipartite graphs with two disjoint sets of vertices.
3. TUX$^2$ enables users to define multiple vertex types, which are placed in separate arrays. This results in compact data representation and improved data locality.
4. Further, TUX$^2$ can take advantage of the bipartite graph representation so that only high-degree vertices have mirrors. Because TUX$^2$ can scan vertices by type, TUX$^2$ can scan only vertices with mirrors in a Mini-Batch to efficiently synchronize updates with the master replicas.
5. TUX$^2$ allows users to define different data structures for the master and mirror replicas of vertices, which aids in synchronization efficiency. For example, during computation there may be auxiliary feature vertex attributes store locally in the mirror replicas that are not needed by the master replicas. Also, the master replicas may need to store some attributes that are not needed by the mirror replicas.

**Scheduling:**

1. TUX$^2$ supports the Stale Synchronous Parallel (SSP) scheduling model with bounded staleness (defined by the *slack* parameter) and mini-batch granularity. The *slack* parameter denotes how stale a task's view of the globally shared state can be. SSP is based on the idea of *work-per-clock,* where a clock specifies an iteration over a mini-batch executed by a set of concurrent tasks.
2. The SSP model enables TUX$^2$ to support mini-batch updating of the model, which aids performance.

**Programming Model:**

1. TUX$^2$ uses the new stage-based *MEGA* model, where each stage is a computation on a set of vertices and their corresponding edges. Each stage applies a user-defined function (UDF) to a specified set of vertices or edges. The four types of stages in the model are Mini-Batch, Exchange, GlobalSync, and Apply. The MEGA model maintains the simplicity of the GAS model while adding additional flexibility to better suit machine learning algorithms.
2. The Exchange stage enumerates edges of a set of vertices. Users can use it to generate new accumulated deltas for vertices and edges or to update their states directly. This stage is more flexible than the Gather/Scatter phases in the GAS model because it does not enforce a direction of vertex data flow along edges, and it can update the states of both vertices and edges within the same UDF.
3. The Apply stage enumerates a set of vertices and synchronizes their master and mirror versions. Because the user can define different data structures for master and mirror replicas, TUX$^2$ allows users to define a base class *VertexDataSync* for the global state of a vertex, and then define different subclasses for the master and mirror replicas.

4. The GlobalSync stage synchronizes contexts across worker threads and/or consolidates data across a set of vertices. This stage has three UDFs. *Aggregate()* aggregates data across vertices into a worker context; *Combine()* aggregates the context across workers into a special worker, which maintains different versions of the context for different SSP clocks; *Apply()* finalizes the globally aggregated context.
5. The Mini-Batch is composed of a sequence of other stages. It defines the stages to be executed iteratively for each mini-batch. Users define which vertices, edges, or vertex-types to enumerate (i.e. the mini-batch size).

## Limitations and Possible Improvements:

1. The optimal value for the *slack* parameter in SSP varies by algorithm, along with the nature of the relationship between the *slack* parameter and the convergence time. Users are required to tune the parameter for optimal performance.
2. Users are also required to tune the mini-batch size parameter for optimal performance.
3. There could perhaps be some default definition of *VertexDataSync* base class and subclasses for heterogeneous master and mirror data structures.
4. TUX$^2$ could, perhaps, have some method to efficiently identify the set of mirrors that have updates that need to be synchronized with their master replicas so that the users are not required to specify the set of vertices to enumerate.

## Summary of Class Discussion

1. The most macroscopic question was asked early in the lecture. The question concerned whether using a graph engine was the “right way” to solve machine learning problems, especially compared to alternatives such as TensorFlow. Part of the conversation steered towards the user’s perspective. From this vantage point, it doesn’t really matter; the user only cares about which environment delivers the correct answer using the least amount of time (or any other constraints like CPU/memory usage). In this sense, it doesn’t matter what happens “under the hood” from the user’s perspective if performance is suitable. The conversation then drifted back towards the “right way” for processing such problems. In short, there is no “best” way (that we know of). Aside from performance, when designing a system, it is critical to evaluate the trade-off between allowing flexibility without over-burdening the user with tunable parameters.
2. Replicas of high-degree vertices are partitioned so that they are disjoint sets of edges.
3. There was a question regarding the underlying file system for storing and accessing graph information. Unfortunately, the paper did not elaborate on this issue.
4. SSP allows the user to find a balance between fully synchronous and fully asynchronous scheduling. There is no one optimal level for the slack parameter, and the user must tune it for optimal performance. There are cases where SSP can deliver better performance than fully asynchronous scheduling due to long convergence time.
5. There was a question regarding fault tolerance and straggle mitigation in SSP.

6. There was discussion elaborating on how Exchange in the MEGA model can eliminate unnecessary phases in the GAS execution model. Further, the MEGA model allows more flexibility and allows updating of both of an edge’s vertices in the same UDF.
7. There was discussion on if the system is Graph partitioned or Data partitioned. It cannot be both, as that would prevent data from flowing to different sections of the graph.
8. To the best of the presenter’s knowledge, TUX$^2$ is not open-source.