The GraphLab Abstraction

1 A Need for GraphLab in ML

The GraphLab abstraction is the product of several years of research in designing and implementing systems for statistical inference in probabilistic graphical models. Early in our work [12], we discovered that the high-level parallel abstractions popular in the ML community such as MapReduce [2, 13] and parallel BLAS [14] libraries are unable to express statistical inference algorithms efficiently. Our work revealed that an efficient algorithm for graphical model inference should explicitly address the *sparse dependencies* between random variables and adapt to the input data and model parameters.

Guided by this intuition we spent over a year designing and implementing various machine learning algorithms on top of low-level threading primitives and distributed communication frameworks such as OpenMP [15], CILK++ [16] and MPI [1]. Through this process, we discovered the following set of core algorithmic patterns that are common to a wide range of machine learning techniques. Following, we detail our findings and motivate why a new framework is needed (see Table 1).

Sparse Computational Dependencies: Many ML algorithms can be factorized into local **dependent** computations which examine and modify only a small subregion of the entire program state. For example, the conditional distribution of each random variable in a large statistical model typically only depends on a small subset of the remaining variables in the model. This computational sparsity in machine learning arises naturally from the statistical need to reduce model complexity.

Parallel abstractions like MapReduce [2] require algorithms to be transformed into an embarrassingly parallel form where computation is **independent**. Unfortunately, transforming ML algorithms with computational *dependencies* into the embarrassingly parallel form needed for these abstractions is often complicated and can introduce substantial algorithmic inefficiency [17]. Alternatively, data flow abstractions like Dryad [3], permit directed acyclic dependencies, but struggle to represent cyclic dependencies common to iterative ML algorithms. Finally, graph-based messaging abstractions like Pregel [4] provide a more natural representation of computational dependencies but require users to explicitly manage communication between computation units.

Asynchronous Iterative Computation: From simulating complex statistical models, to optimizing parameters, many important machine learning algorithms iterate over local computation kernels. Furthermore, many iterative machine learning algorithms benefit from [18, 19, 12] and in some cases require [20] asynchronous computation. Unlike **synchronous** computation, in which all kernels are computed simultaneously (in parallel) using the previous values for dependent parameters, **asynchronous** computation requires that the local computation kernels use the most recently available values.

Abstractions based on bulk data processing, such as MapReduce [2] and Dryad [3] were not designed for iterative computation. While recent projects like MapReduce Online [9], Spark [11], Twister [21], and Nexus [10] extend MapReduce to the iterative setting, they do not support asynchronous computation. Similarly, parallel graph based abstractions like Pregel [4] and BPGL [5] adopt the Bulk Synchronous Parallel (BSP) model [6] and do not naturally express asynchronous computation.

Sequential Consistency: By ensuring that all parallel executions have an equivalent sequential execution, sequential consistency eliminates many challenges associated with designing, implementing, and testing parallel ML algorithms. In addition, many algorithms converge faster if sequential consistency is ensured, and some even require it for correctness.

However, this view is not shared by all in the ML community. Recently, [22, 23] advocate soft-optimization techniques (e.g., allowing computation to intentionally race), but we argue that such techniques do not apply broadly in ML. Even for the algorithms evaluated in [22, 23], the conditions under which the soft-optimization techniques work are not well understood and may fail in unexpected ways on different datasets.

Indeed, for some machine learning algorithms sequential consistency is strictly required. For instance, Gibbs sampling [24], a popular inference algorithm, requires sequential consistency for statistical correctness, while many other optimization procedures require sequential consistency to converge (Fig. 1 demonstrates that the prediction error rate of one of our example problems is dramatically better when computation is properly asynchronous.). Finally, as [19] demonstrates, the lack of sequential consistency can dramatically increase the time to convergence for stochastic optimization procedures.

By designing an abstraction which enforces sequen-

	Computation Model	Sparse Depend.	Async. Comp.	Iterative	Prioritized Ordering	Sequentially Consistent [a]	Distributed
MPI[1]	Messaging	Yes	Yes	Yes	N/A [b]	N/A [b]	Yes
MapReduce[2]	Par. data-flow	No	No	extensions[c]	N/A	N/A	Yes
Dryad[3]	Par. data-flow	Yes	No	extensions[d]	N/A	N/A	Yes
Pregel[4]/BPGL[5]	GraphBSP[6]	Yes	No	Yes	N/A	N/A	Yes
Piccolo[7]	Distr. map[f]	N/A [f]	Yes	Yes	No	accumulators	Yes
Pearce et.al.[8]	Graph Visitor	Yes	Yes	Yes	Yes	No	No
GraphLab	**GraphLab**	Yes	Yes	Yes	Yes[e]	Yes	Yes [g]

Table 1: **Comparison chart of parallel abstractions:** Detailed comparison against each of the abstractions are in the text (Sec. 1). **(a)** Here we refer to Sequential Consistency with respect to asynchronous computation. See Sec. 1 for details. This property is therefore relevant only for abstractions which support asynchronous computation. **(b)** MPI-2 does not define a data model and is a lower level abstraction than others listed. **(c)** Iterative extension for MapReduce are proposed [9, 10, 11]. **(d)** [10] proposes an iterative extension for Dryad. **(e)** The GraphLab abstraction allows for flexible scheduling mechanisms (our implementation provides FIFO and priority ordering). **(f)** Piccolo computes using user-defined kernels with random access to a distributed key-value store. It does not model data dependencies. **(g)** To be released.

tially consistent computation, we eliminate much of the complexity introduced by parallelism, allowing the ML expert to focus on algorithm design and correctness of numerical computations. Debugging mathematical code in a parallel program which has random errors caused by non-deterministic ordering of concurrent computation is particularly unproductive.

The discussion of sequential consistency is relevant only to frameworks which support asynchronous computation. Piccolo [7] provides a limited amount of consistency by combining simultaneous writes using accumulation functions. However, this only protects against single write races, but does not ensure sequential consistency in general. The parallel asynchronous graph traversal abstraction by Pearce et. al. [8] does not support any form of consistency, and thus is not suitable for a large class of ML algorithms.

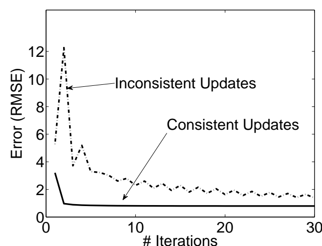


Figure 1: Convergence plot of Alternating Least Squares (Sec. **??**) comparing prediction error when running sequentially consistent asynchronous iterations vs inconsistent asynchronous iterations over a five node distributed cluster. Consistent iterations converge rapidly to a lower error while inconsistent iterations oscillate and converge slowly.

Prioritized Ordering: In many ML algorithms, iterative computation converges asymmetrically. For example, in parameter optimization, often a large number of parameters will quickly converge after only a few iterations, while the remaining parameters will converge slowly over many iterations [25, 26]. If we update all parameters equally often, we could waste substantial computation recomputing parameters that have effectively converged. Conversely, by focusing early computation on more challenging parameters first, we can potentially reduce computation.

Adaptive prioritization can be used to focus iterative computation where it is needed. The only existing framework to support this is the parallel graph framework by Pearce et. al. [8]. The framework is based on the visitor-pattern and prioritizes the ordering of visits to vertices. GraphLab however, allows the user to define *arbitrary ordering* of computation, and our implementation supports efficient FIFO and priority-based scheduling.

Rapid Development: Machine learning is a rapidly evolving field with new algorithms and data-sets appearing weekly. In many cases these algorithms are not yet well characterized and both the computational and statistical properties are under active investigation. Large-scale parallel machine learning systems must be able to adapt quickly to changes in the data and models in order to facilitate rapid prototyping, experimental analysis, and model tuning. To achieve these goals, an effective high-level parallel abstraction must hide the challenges of parallel algorithm design, including race conditions, deadlock, state-partitioning, and communication.

2 The GraphLab Abstraction

Using the ideas from the previous section, we extracted a single coherent computational pattern: *asynchronous parallel computation on graphs* with a *sequential* model of computation. This pattern is both sufficiently expressive to encode a wide range of ML algorithms, and sufficiently restrictive to enable efficient parallel implementations.

The GraphLab abstraction consists of three main parts, the data graph, the update function, and the sync operation. The data graph (Sec. 2.1) represents user modifiable program state, and both stores the mutable user-defined

data and encodes the sparse computational dependencies. The update functions (Sec. 2.2) represent the factorized user computation and operate on the data graph by transforming data in small overlapping contexts called scopes. Finally, the sync operation (Sec. 2.3) is used to maintain global aggregate statistics of the data graph.

We now present the GraphLab abstraction in greater detail. To make these ideas more concrete, we will use the PageRank algorithm [27] as a running example. While PageRank is not a common machine learning algorithm, it is easy to understand and shares many properties common to machine learning algorithms.

Example 2.1 (PageRank)**.** *The PageRank algorithm recursively defines the rank of a webpage v:*

$$\boldsymbol{R}(v) = \frac{\alpha}{n} + (1-\alpha) \sum_{u \text{ links to } v} w_{u,v} \times \boldsymbol{R}(u) \qquad (2.1)$$

in terms of the ranks of those pages that link to v and the weight w of the link as well as some probability α of randomly jumping to that page. The PageRank algorithm, simply iterates Eq. (2.1) until the individual PageRank values converge (i.e., change by less than some small ϵ).

2.1 Data Graph

The GraphLab abstraction stores the program state as an undirected graph called the **data graph**. The data graph $G = (V, E, D)$ is a container which manages the user defined data D. Here we use the term "data" broadly to refer to model parameters, algorithmic state, and even statistical data. The user can associate arbitrary data with each vertex $\{D_v : v \in V\}$ and edge $\{D_{u\leftrightarrow v} : \{u, v\} \in E\}$ in the graph. Since some machine learning applications require directed edge data (e.g., weights on directed links in a web-graph) we provide the ability to store and retrieve data associated with directed edges. While the graph data is mutable, the graph structure is *static*[1] and cannot be changed during execution.

Example (PageRank: Ex. 2.1)**.** *The data graph for PageRank is directly obtained from the web graph, where each vertex corresponds to a web page and each edge represents a link. The vertex data D_v stores $\boldsymbol{R}(v)$, the current estimate of the PageRank, and the edge data $D_{u\rightarrow v}$ stores $w_{u,v}$, the directed weight of the link.*

The data graph is convenient for representing the state of a wide range of machine learning algorithms. For example, many statistical models are efficiently represented by undirected graphs [28] called Markov Random Fields (MRF). The data graph is derived directly from the MRF, with each vertex representing a random variable. In this case the vertex data and edge data may store the local parameters that we are interested in learning.

[1] Although we find that fixed structures are sufficient for most ML algorithms, we are currently exploring the use of dynamic graphs.

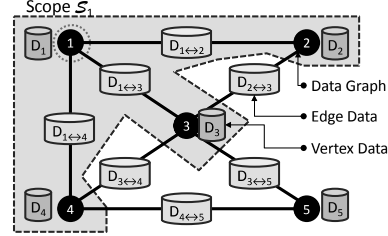


Figure 2: In this figure we illustrate the GraphLab **data graph** as well as the scope $\mathcal{S}_1$ of vertex 1. Each of the gray cylinders represent a block of user defined data and is associated with a vertex or edge. The **scope** of vertex 1 is illustrated by the region containing vertices $\{1, 2, 3, 4\}$. An **update function** applied to vertex 1 is able to read and modify all the data in $\mathcal{S}_1$ (vertex data D_1, D_2, D_3, and D_4 and edge data $D_{1\leftrightarrow 2}$, $D_{1\leftrightarrow 3}$, and $D_{1\leftrightarrow 4}$).

2.2 Update Functions

Computation is encoded in the GraphLab abstraction via user defined update functions. An **update function** is a stateless procedure which modifies the data within the **scope** of a vertex and schedules the future execution of other update functions. The scope of vertex v (denoted by $\mathcal{S}_v$) is the data stored in v, as well as the data stored in all adjacent vertices and edges as shown in Fig. 2.2.

A GraphLab update function takes as an input a vertex v and its scope $\mathcal{S}_v$ and returns the new version of the scope as well as a set of tasks $\mathcal{T}$ which encodes future task executions.

$$\textbf{Update} : (v, \mathcal{S}_v) \rightarrow (\mathcal{S}_v, \mathcal{T})$$

After executing an update function the modified scope data in $\mathcal{S}_v$ is written back to the data graph. Each **task** in the set of tasks $\mathcal{T}$, is a tuple (f, v) consisting of an update function f and a vertex v. All returned task $\mathcal{T}$ are executed *eventually* by running $f(v, \mathcal{S}_v)$ following the execution semantics described in Sec. 2.4.

Rather than adopting a message passing or data flow model as in [4, 3], GraphLab allows the user defined update functions complete freedom to read and modify any of the data on adjacent vertices and edges. This simplifies user code and eliminates the need for the users to reason about the movement of data. By controlling what tasks are added to the task set, GraphLab update functions can efficiently express adaptive computation. For example, an update function may choose to reschedule its neighbors only when it has made a substantial change to its local data.

The update function mechanism allows for *asynchronous computation* on the *sparse dependencies* defined by the data graph. Since the data graph permits the expression of general cyclic dependencies, *iterative computation* can be represented easily.

Algorithm 1: PageRank update function

```
Input: Vertex data R(v) from S_v
Input: Edge data {w_{u,v} : u ∈ N[v]} from S_v
Input: Neighbor vertex data {R(u) : u ∈ N[v]} from S_v
R_old(v) ← R(v)              // Save old PageRank
R(v) ← α/n
foreach u ∈ N[v] do      // Loop over neighbors
    R(v) ← R(v) + (1 − α) ∗ w_{u,v} ∗ R(u)
// If the PageRank changes sufficiently
if |R(v) − R_old(v)| > ε then
    // Schedule neighbors to be updated
    return {(PageRankFun, u) : u ∈ N[v]}
Output: Modified scope S_v with new R(v)
```

Algorithm 2: GraphLab Execution Model

```
Input: Data Graph G = (V, E, D)
Input: Initial task set T = {(f, v_1), (g, v_2), ...}
Input: Initial set of syncs:
        (Name, Fold, Merge, Finalize, acc(0), τ)
while T is not Empty do
1   (f, v) ← RemoveNext(T)
2   (T′, S_v) ← f(v, S_v)
3   T ← T ∪ T′
    Run all Sync operations which are ready
Output: Modified Data Graph G = (V, E, D′)
Output: Result of Sync operations
```

Many algorithms in machine learning can be expressed as simple update functions. For example, probabilistic inference algorithms like Gibbs sampling [24], belief propagation [29], expectation propagation [30] and mean field variational inference [31] can all be expressed using update functions which read the current assignments to the parameter estimates on neighboring vertices and edges and then apply sampling or optimization techniques to update parameters on the local vertex.

Example (PageRank: Ex. 2.1)**.** *The update function for PageRank (defined in Alg. 1) computes a weighted sum of the current ranks of neighboring vertices and assigns it as the rank of the current vertex. The algorithm is adaptive: neighbors are listed for update only if the value of current vertex changes more than a predefined threshold.*

2.3 Sync Operation

In many ML algorithms it is necessary to maintain global statistics describing data stored in the data graph. For example, many statistical inference algorithms require tracking of global convergence estimators. Alternatively, parameter estimation algorithms often compute global averages or even gradients to tune model parameters. To address these situations, the GraphLab abstraction expresses global computation through the **sync operation**, which aggregates data across all vertices in the graph in a manner analogous to MapReduce. The results of the sync operation are stored globally and may be accessed by all update functions. Because GraphLab is designed to express iterative computation, the sync operation runs repeated at fixed user determined intervals to ensure that the global estimators remain fresh.

The sync operation is defined as a tuple ($\texttt{Key}$, **Fold**, **Merge**, **Finalize**, $\mathrm{acc}(0), \tau$) consisting of a unique key, three user defined functions, an initial accumulator value, and an integer defining the interval between sync operations. The sync operation uses the **Fold** and **Merge** functions to perform a *Global Synchronous Reduce* where **Fold** aggregates vertex data and **Merge** combines intermediate **Fold** results. The **Finalize** function performs a transformation on the final value and stores the result. The `Key` can then be used by update functions to access the most recent result of the sync operation. The sync operation runs periodically, approximately every τ update function calls[2].

Example (PageRank: Ex. 2.1)**.** *We can compute the second most popular page on the web by defining the following sync operation:*

$$\begin{aligned} \textbf{\textit{Fold}} &: \mathit{fld}(acc, v, D_v) := \mathit{TopTwo}(acc \cup \boldsymbol{R}(v)) \\ \textbf{\textit{Merge}} &: \mathit{mrg}(acc, acc') := \mathit{TopTwo}(acc \cup acc') \\ \textbf{\textit{Finalize}} &: \mathit{fin}(acc) := acc[2] \end{aligned}$$

*Where the accumulator taking on the initial value as the empty array $acc(0) = \emptyset$. The function "TopTwo(X)" returns the two pages with the highest pagerank in the set X. After the global reduction, the acc array will contain the top two pages and $acc[2]$ in **Finalize** extracts the second entry. We may want to update the global estimate every $\tau = |V|$ vertex updates.*

2.4 The GraphLab Execution Model

The GraphLab execution model, presented in Alg. 2, follows a simple single loop semantics. The input to the GraphLab abstraction consists of the data graph $G = (V, E, D)$, an update function **Update**, an initial set of tasks $\mathcal{T}$ to update, and any sync operations. While there are tasks remaining in $\mathcal{T}$, the algorithm removes (Line 1) and executes (Line 2) tasks, adding any new tasks back into $\mathcal{T}$ (Line 3). The appropriate sync operations are executed whenever necessary. Upon completion, the resulting data graph and synced values are returned to the user.

The exact behavior of $\texttt{RemoveNext}(\mathcal{T})$ (Line 1) is up to the implementation of the GraphLab abstraction. The only guarantee the GraphLab abstraction provides is

[2] The resolution of the synchronization interval is left up to the implementation since in some architectures a precise synchronization interval may be difficult to maintain.

that RemoveNext removes and returns an update task in $\mathcal{T}$. The flexibility in the order in which RemoveNext removes tasks from $\mathcal{T}$ provides the opportunity to balance features with performance constraints. For example, by restricting task execution to a fixed order, it is possible to optimize memory layout. Conversely, by supporting *prioritized ordering* it is possible to implement more advanced ML algorithms at the expense run-time overhead.

The GraphLab abstraction presents a rich *sequential model* that is automatically translated into a *parallel execution* by allowing multiple processors to remove and execute update tasks simultaneously. To retain the same *sequential* execution semantics we must ensure that overlapping computation is not run simultaneously. However, the extent to which computation can *safely* overlap depends on the user defined update function. In the next section we introduce several **consistency models** that allow the runtime to optimize the parallel execution while maintaining consistent computation.

2.5 Sequential Consistency Models

A parallel implementation of GraphLab must guarantee sequential consistency [32] over update tasks and sync operations. We define sequential consistency in the context of the GraphLab abstraction as:

Definition 2.1 (GraphLab Sequential Consistency)**.** *For every parallel execution of the GraphLab abstraction, there exists a sequential ordering on all executed update tasks and sync operations which produces the same data graph and synced global values.*

A simple method to achieve sequential consistency among update functions is to ensure that the scopes of concurrently executing update functions do not overlap. We refer to this as the **full consistency** model (see Fig. 3(a)). Full consistency limits the potential parallelism since concurrently executing update functions must be at least two vertices apart (see Fig. 3(b)). Even in moderately dense data graphs, the amount of available parallelism could be low. Depending on the actual computation performed within the update function, additional relaxations can be safely made to obtain more parallelism without sacrificing sequential consistency.

We observed that for many machine learning algorithms, the update functions do not need full read/write access to all of the data within the scope. For instance, the PageRank update in Eq. (2.1) only requires read access to edges and neighboring vertices. To provide greater parallelism while retaining sequential consistency, we introduced the **edge consistency** model. If the edge consistency model is used (see Fig. 3(a)), then each update function has exclusive read-write access to its vertex and adjacent edges but read only access to adjacent vertices. This increases parallelism by allowing update functions with slightly overlapping scopes to safely run in parallel (see Fig. 3(b)).

Finally, for many machine learning algorithms there is often some initial data pre-processing which only requires read access to adjacent edges and write access to the central vertex. For these algorithms, we introduced the weakest **vertex consistency** model (see Fig. 3(a)). This model has the highest parallelism but only permits fully independent (Map) operations on vertex data.

While sequential consistency is essential when designing, implementing, and debugging complex ML algorithms, an adventurous user [23] may want to relax the theoretical consistency constraints. Thus, we allow users to choose a weaker consistency model at their own risk.

References

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[4] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODC '09*, page 6. ACM, 2009.

[5] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *POOSC*, 2005.

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

[7] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI '10*, pages 1–14, 2010.

[8] R. Pearce, M. Gokhale, and N.M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SuperComputing '10*, pages 1–11, 2010.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI'10*, pages 21–21, 2010.

[10] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *HotCloud*, pages 19–19, 2009.

[11] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud '10*. USENIX Association, 2010.

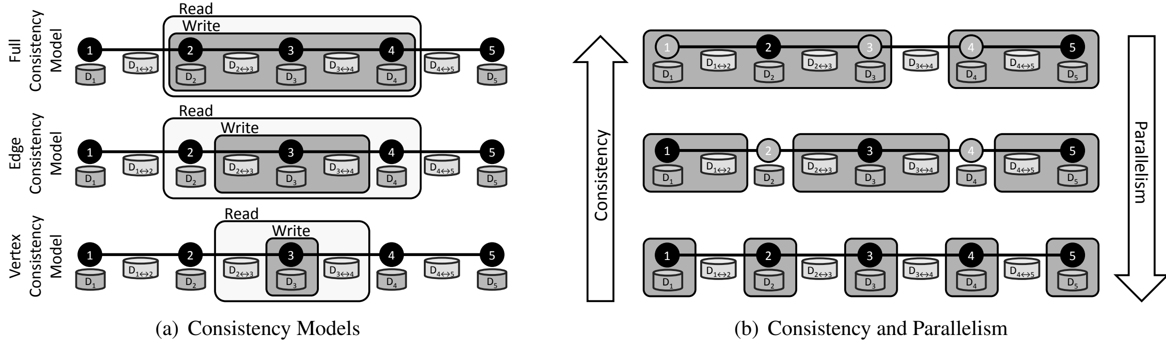
[12] **Anonymous**. Distributed parallel inference on large factor graphs. In *UAI*.

[13] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *NIPS*, pages 281–288. MIT Press, 2006.

[14] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. A proposal for a set of parallel basic linear algebra subprograms. *PARA*, 1996.

[15] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[16] C.E. Leiserson. The Cilk++ concurrency platform. In *DAC*. IEEE, 2009.



(a) Consistency Models

(b) Consistency and Parallelism

Figure 3: To ensure sequential consistency while providing the maximum parallelism, the GraphLab abstraction provides three different consistency models: full, edge, vertex. In figure **(a)**, we illustrate the read and write permissions for an update function executed on the central vertex under each of the consistency models. Under the **full consistency model** the update function has complete read write access to its entire scope. Under the slightly weaker **edge consistency model** the update function has only read access to adjacent vertices. Finally, **vertex consistency model** only provides write access to the local vertex data. The vertex consistency model is ideal for independent computation like feature processing. In figure **(b)** We illustrate the trade-off between consistency and parallelism. The dark rectangles denote the write-locked regions which cannot overlap. Update functions are executed on the dark vertices in parallel. Under the full consistency model we are only able to run two update functions $f(2, \mathcal{S}_2)$ and $f(5, \mathcal{S}_5)$ simultaneously while ensuring sequential consistency. Under the edge consistency model we are able to run three update functions (i.e., $f(1, \mathcal{S}_1)$, $f(3, \mathcal{S}_3)$, and $f(5, \mathcal{S}_5)$) in parallel. Finally under the vertex consistency model we are able to run update functions on all vertices in parallel.

[17] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, 2009.

[18] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., USA, 1989.

[19] W. G. Macready, A. G. Siapas, and S. A. Kauffman. Criticality and parallelism in combinatorial optimization. *Science*, 271:271–56, 1995.

[20] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *AISTATS*, 2011.

[21] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*. ACM, 2010.

[22] M. Jiayuan, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS*, pages 1 –12, May 2009.

[23] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP '11*, pages 35–46. ACM, 2011.

[24] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. In *PAMI*, 1984.

[25] G. Elidan, I. McGraw, and D. Koller. Residual Belief Propagation: Informed scheduling for asynchronous message passing. In *UAI '06*, pages 165–173, 2006.

[26] B Efron, T Hastie, I M Johnstone, and Robert Tibshirani. Least angle regression. *Annals of Statistics*, 32:407–451, 2004.

[27] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[28] D Koller and N Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[29] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[30] T. P. Minka. *A family of algorithms for approximate bayesian inference*. PhD thesis, MIT, 2001. AAI0803033.

[31] E. P. Xing, M. I. Jordan, and S. Russell. A Generalized Mean Field Algorithm for Variational Inference in Exponential Families. In *UAI '03*, pages 583–559, 2003.

[32] L Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.