

GraphLab: A Distributed Abstraction for Large Scale Machine Learning

Yucheng Low

**June 2013
CMU-ML-13-104**



GraphLab: A Distributed Abstraction for Large Scale Machine Learning

Yucheng Low

June 2013

CMU-ML-13-104

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Carlos Guestrin (Chair)

Guy E. Blelloch

David Andersen

Andrew Y. Ng (Stanford University)

Joseph M. Hellerstein (University of California, Berkeley)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2013 Yucheng Low

This research was sponsored by the Army Research Office under grant number W911NF0710287, the Office of Naval Research under grant numbers N000140710747, N000140810752, N000141010672, the National Science Foundation under grant numbers CNS0721591, IIS0803333, the Air Force Research Laboratory under grant number FA87500910141, and a fellowship from the Intel Science and Technology Center for Cloud Computing.

Abstract

Machine Learning methods have found increasing applicability and relevance to the real world, finding applications in a broad range of fields in robotics, data mining, physics and biology, among many others. However, with the growth of the World Wide Web, and with improvements in data collection technology, real world datasets have been rapidly increasing in size and complexity, necessitating comparable scaling of Machine Learning algorithms.

However, designing and implementing efficient parallel Machine Learning algorithms is challenging. Existing high-level parallel abstractions like MapReduce are insufficiently expressive while low-level tools such as MPI are difficult to use, and leave Machine Learning experts repeatedly solving the same design challenges.

In this thesis, we trace the development of a framework called GraphLab which aims to provide an expressive and efficient high level abstraction to satisfy the needs of a broad range of Machine Learning algorithms. We discuss the initial GraphLab design, including details of a shared memory and distributed memory implementation. Next, we discuss the scaling limitations of GraphLab on real-world power-law graphs and how that informed the design of PowerGraph. By placing restrictions on the abstraction, we are able to improve scalability, demonstrating state of the art performance on a variety of benchmarks. Finally, we end with the WarpGraph abstraction which improves usability of PowerGraph by combining features of GraphLab and PowerGraph to achieve an abstraction that is easy to use, scalable, and fast.

We demonstrate in this thesis, that by designing a domain specific abstraction for Machine Learning, we are able to provide a system that is both easy to use, and provides exceptional scaling and performance on real world datasets.

Acknowledgments

I would like to thank my advisor, Carlos Guestrin, who provided crucial research guidance, without which this thesis would not have been possible. Carlos's constant emphasis and guidance on the importance of writing and presentation skills has been invaluable to my development as a better communicator. Carlos had also put together a wonderful research group of talented people who I would like to thank for their friendship and collaboration.

I began working with Joseph Gonzalez right from the beginning of graduate school and nearly all of my papers are coauthored with him. I thank Joseph for his continuous encouragement and perseverance during difficult times. Our frequent discussions were crucial in the formation of this thesis, and his development skills were essential to bring GraphLab to fruition. I also thank Joseph for his patience; his seek for perfection in writing quality, and his truly amazing ability with PowerPoint animations.

The GraphLab project was developed by a small team consisting of Joseph Gonzalez, Danny Bickson, Aapo Kyrola and Haijie Gu. I collaborated with Joseph for the entirety of the GraphLab project and he has helped shape, develop and implement all versions of the abstraction. Danny Bickson has been instrumental emphasizing the importance of the sparse matrix factorization task and built the open source community around GraphLab. Aapo took a different view of the abstraction, challenging the in-memory requirement of GraphLab, and built GraphChi, demonstrating that high performance computation on disk can be a reality. Haijie developed the graph partitioning heuristics in PowerGraph, which was pivotal for PowerGraph's performance and success. Additionally, I would also like to thank Alex Smola, Joseph Hellerstein, Seth Goldstein and Guy Blelloch for their advice throughout the process of the GraphLab project.

I would also like to thank the Machine Learning Department at Carnegie Mellon University which fostered a wonderful, friendly, collaborative environment for research.

Finally, I would like to thank all family and friends for their support.

Contents

1	Introduction	1
1.1	Design Goals	2
1.1.1	Common Properties	2
1.2	Thesis Statement	3
1.3	Notation	3
2	GraphLab 1	
	<i>Graph As Computation</i>	5
2.1	Graph-Parallel Abstractions	5
2.2	GraphLab Abstraction	7
2.2.1	Data Graph	7
2.2.2	Update Functions	7
2.2.3	The GraphLab Execution Model	8
2.2.4	Ensuring Serializability	9
2.2.5	Sync Operation and Global Values	9
2.2.6	Abstraction Summary	10
2.3	Shared Memory Implementation	11
2.3.1	Engine	11
2.3.2	Schedulers	11
2.3.3	Evaluation	13
2.4	Distributed Implementation	22
2.4.1	The Data Graph	22
2.4.2	Engines	24
2.4.3	Fault Tolerance	26
2.4.4	Evaluation	28
2.5	Conclusion	34
3	PowerGraph	
	<i>Restricting the Abstraction for Performance</i>	35
3.1	GraphLab 1 Limitations	35
3.1.1	Gather vs Scatter	36
3.1.2	Natural Graphs	36
3.1.3	Serializability Frequently Unnecessary	39
3.2	Characterization	39
3.3	PowerGraph Abstraction	40
3.3.1	GAS Vertex-Programs	41

3.3.2	Delta Caching	43
3.3.3	Scheduling	43
3.3.4	Comparison with GraphLab 1 and Pregel	45
3.4	Distributed Graph Placement	45
3.4.1	Balanced p -way Vertex-Cut	47
3.4.2	Greedy Vertex-Cuts	49
3.5	Abstraction Comparison	51
3.5.1	Computation Imbalance	52
3.5.2	Communication Imbalance	53
3.5.3	Runtime Comparison	53
3.6	Implementation and Evaluation	54
3.6.1	Graph Loading and Placement	54
3.6.2	Synchronous Engine (Sync)	54
3.6.3	Asynchronous Engine (Async)	56
3.6.4	Async. Serializable Engine (Async+S)	56
3.6.5	Fault Tolerance	58
3.6.6	Applications	58
3.7	Conclusion	59
4	WarpGraph	
	<i>Fine Grained Data-Parallelism for Usability</i>	61
4.1	PowerGraph Limitations	61
4.1.1	Expressiveness	62
4.1.2	Usability	62
4.2	WarpGraph	65
4.2.1	Update Function	65
4.2.2	Fine Grained Data-Parallel Primitives	66
4.2.3	Expressiveness	69
4.3	Implementation	70
4.4	Evaluation	72
4.4.1	PageRank	72
4.4.2	Graph Coloring	73
4.4.3	Stochastic Gradient Descent	74
4.5	Conclusion	77
5	Related Work	79
5.1	Map-Reduce Abstraction	79
5.1.1	Relation to GraphLab	80
5.2	BSP Abstraction (Pregel)	80
5.2.1	Dynamic Execution	81
5.2.2	Asynchronous Computation	81
5.2.3	Edge Data	81
5.2.4	Generalized Messaging	82
5.2.5	Scatter vs Gather	82
5.2.6	Relation to GraphLab	83
5.3	Generalized SpMV Abstraction	83
5.3.1	Relation to GraphLab	83

5.4	Dataflow Abstraction	83
5.4.1	Relation to GraphLab	84
5.5	Datalog/Bloom	84
6	Conclusion	85
6.1	Thesis Summary	85
6.2	Observations	86
6.3	Future Work	89
6.3.1	Graph Partitioning	89
6.3.2	Incremental Machine Learning	89
6.3.3	Dynamic Execution	89
6.3.4	Fault Tolerance	90
6.3.5	Other Abstractions	90
6.3.6	Parameter Server	91
A	Extended Chandy Misra	92
A.1	Chandy Misra	92
A.2	Hierarchical Chandy Misra	93
B	Algorithm Examples	96
B.1	PageRank	96
B.2	Connected Component	98
B.3	Triangle Counting	100
B.4	Collaborative Filtering	102
B.5	Belief Propagation	106
C	Dataset List	110
	Bibliography	111

Chapter 1

Introduction

The field of Machine Learning (ML) has matured significantly in the last few decades, making it increasingly relevant to the real world. Simultaneously, with the growth of the World Wide Web and with improvements in data collection capabilities, real world dataset sizes have been experiencing exponential growth rates; growing beyond the capability of what most research groups can handle. For instance: Facebook now reaches to over 845 million users: increasing by 237 million in the last year¹; Youtube now observes an average of 48 hours of video uploaded per minute, increasing by a factor of two over the last year². While simple ML methods which only require moments of the data [Chu et al., 2007] can cope with such large datasets with ease, in many cases more sophisticated algorithms can provide increased accuracy and performance. However, such advanced algorithms are also significantly harder to implement and scale. To increase applicability of ML methods to the real world, there is a need for scalable systems that can execute modern ML algorithms on *large* dataset sizes.

Unfortunately, existing high level parallel abstractions such as MapReduce [Dean and Ghemawat, 2004], Dryad [Isard et al., 2007], Pregel [Malewicz et al., 2010] among others, do not provide sufficient expressiveness for large classes of Machine Learning algorithms, requiring excessive programmer effort, or incurring unnecessary inefficiency and overhead. Alternatively, implementing ML algorithms on low level frameworks such as OpenMP [Dagum and Menon, 1998] or MPI [Gropp et al., 1996] can be challenging, requiring the user to address complex issues like race conditions, deadlocks, communication and synchronization.

In this thesis, we propose a computation abstraction called GraphLab based on sparse graph structures, that we find to be widely applicable to a broad range of Machine Learning problems. We track the evolution of the abstraction over the years (renaming it as PowerGraph, and WarpGraph) as we modify it to resolve scalability and usability issues. While the subsequent abstractions have different names, they revolve around the same original GraphLab abstraction blueprint. The thesis is organized as follows:

Section 1.1: Design Goals Describe the computation needs of Machine Learning Algorithms, and how existing computation abstractions fail to address these requirements.

Chapter 2: GraphLab 1 Our first attempt at a unified ML abstraction which addresses these computation needs for the shared and distributed memory setting.

¹<http://www.facebook.com/press/info.php?timeline>

²<http://youtube-global.blogspot.com/2011/05/thanks-youtube-community-for-two-big.html>

Chapter 3: PowerGraph GraphLab was powerful and easy to use, but encounters severe scalability limitations with power-law graphs. PowerGraph introduces restrictions on the GraphLab 1 abstraction to improve efficiency and scalability, achieving unparalleled performance on a variety of benchmarks.

Chapter 4: WarpGraph PowerGraph was fast, but difficult to use in practice due to its restrictions. WarpGraph, combines features of GraphLab 1 and PowerGraph to provide a new abstraction to provide both the ease of use of GraphLab 1, and the performance of PowerGraph.

Chapter 5: Related Work We discuss the similarities and differences between the work in this thesis and other related abstractions.

Chapter 6: Conclusion We conclude and summarize the thesis. We discuss how GraphLab fits in the bigger picture of Large Scale Machine Learning and the future research directions that can follow.

Appendix A: Chandy Misra We describe a hierarchical extension to the Chandy-Misra algorithm [Chandy and Misra, 1984], which we used to provide distributed locking in PowerGraph.

Appendix B: Algorithm Examples We provide five algorithm examples implemented in all three abstractions.

Appendix C: Dataset List A list of all experimental datasets used for testing and benchmarking.

1.1 Design Goals

The goal of this thesis is to design, and implement a computation abstraction which is uniquely designed to target the needs of many Machine Learning algorithms. To do so, it is necessary to understand the common properties that exist in many Machine Learning algorithms and how existing computation abstractions fail to address these properties.

1.1.1 Common Properties

Interdependent Computation: Many of the recent advances in ML have focused on modeling the *dependencies* between data. By modeling data dependencies, we are able to extract more signal from noisy data and build a deeper understanding. For example, modeling the dependencies between similar shoppers in a semi-supervised or transductive learning setting allows us to make better product recommendations than treating shoppers in isolation. Unfortunately, data parallel abstractions like MapReduce [Dean and Ghemawat, 2004] are not generally well suited for the *dependent* computation typically required by more advanced ML algorithms. Although it is often possible to transform algorithms with computational dependencies into MapReduce algorithms, the resulting transformations can introduce inefficiencies.

Asynchronous Iterative Computation: Many important machine learning algorithms iteratively update large collections of parameters. Unlike **synchronous** computation, in which all parameters are updated simultaneously (in parallel) using the previous parameters values, **asynchronous** computation allows individual parameter updates to use the most recently available values for dependent parameters. Asynchronous computation provides many ML algorithms with significant algorithmic benefits. For example, linear systems (common to many ML algorithms) have been theoretically shown in Bertsekas and Tsitsiklis [1989] to converge faster when solved asynchronously. Additionally, there are numerous cases in ML where asynchronous procedures have been empirically shown to significantly outperform synchronous

procedures, even though the theory is not entirely well understood. Such cases include Belief Propagation, Expectation Maximization [Neal and Hinton, 1998] and stochastic optimization [Siapas, 1996, Smola and Narayanamurthy, 2010]. We demonstrate one such case in Fig. 1.1a where asynchronous Belief Propagation Yedidia et al. [2001] can be demonstrated to converge faster and to a better answer than synchronous belief propagation.

Dynamic Computation: 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 [Efron et al., 2004, Elidan et al., 2006]. More recently, Zhang et al. [2011] also demonstrated the effectiveness of prioritized computation for a variety of graph algorithms including PageRank. If we update all parameters equally often, we waste computation recomputing parameters that have effectively converged. Conversely, by focusing early computation on more challenging parameters, we can potentially accelerate convergence. In Fig. 1.1 we demonstrate how dynamic scheduling can accelerate convergence in belief propagation and matrix factorization.

Serializability/Sequential Consistency: By ensuring that all parallel executions have an equivalent sequential execution, serializability eliminates many challenges associated with designing, implementing, and testing parallel ML algorithms. An abstraction which can provide serializable computation as an option, eliminates much of the complexity introduced by concurrency, allowing the ML expert to focus on algorithm and model design. Furthermore, many ML algorithms require serializability for correctness. For instance, Gibbs sampling [Geman and Geman, 1984] requires strict serializability to be statistically valid, and this could be of great importance to non-Machine Learning, statistical applications. However, maintaining serializability incurs performance penalties, and there are empirical and theoretical evidence that some ML algorithms are resilient to race conditions [Agarwal and Duchi, 2011, Bradley et al., 2011, Recht et al., 2011, Siapas, 1996, Smola and Narayanamurthy, 2010]. An effective ML computation abstraction should therefore be able to provide serializability when needed, but provide the option to trade serializability for throughput.

1.2 Thesis Statement

By focusing on the computational needs of Machine Learning we can develop effective abstractions for efficient large-scale distributed Machine Learning.

1.3 Notation

While each chapter will try to reintroduce all required notation, there are some common notation which are used consistently throughout the thesis. We summarize them here for easy referencing.

V	All vertices of a graph.
E	All edges of a graph.
$N[v]$	The set of all neighbors of vertex v .
$D[v]$	The degree of vertex v .

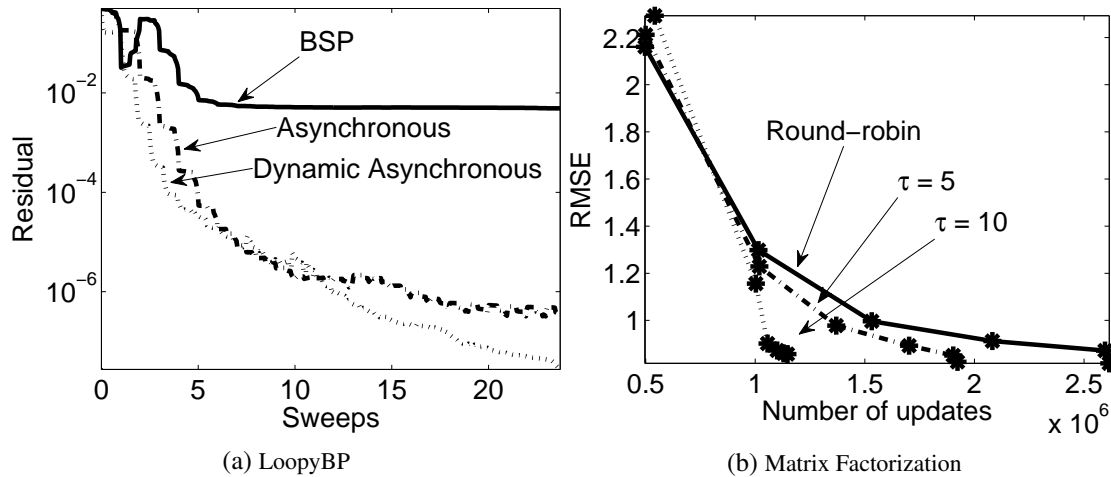


Figure 1.1: **(a)** To compare the efficiency of different computation models we plot the level of convergence of Belief Propagation as a function of time using synchronous (BSP), asynchronous and dynamic schedulings. The difference in convergence is discussed in Gonzalez et al. [2009b]. **(b)** Convergence speed of Matrix factorization using Alternative Least Squares comparing full round-robin updates and using dynamic scheduling with different skipping thresholds (τ). Skipping updates on relatively converged vertices leads to focused computation and as consequence, faster convergence.

- D_v The data stored on a vertex v .
- $D_{u \rightarrow v}$ The data stored on the directed edge $u \rightarrow v$.
- D_{u-v} Refer to either the data located on $u \rightarrow v$ or $v \rightarrow u$ depending on context (See Sec. 2.2.1).
- $D_{* \rightarrow v}$ The set of all data on in-vertices to vertex v . Formally, $\{D_u : (u \rightarrow v) \in E\}$
- $D_{v \rightarrow *}$ The set of all data on out-vertices to vertex v . Formally, $\{D_u : (v \rightarrow u) \in E\}$
- D The set of all data on all edges and vertices of the graph.

Chapter 2

GraphLab 1

Graph As Computation

In this chapter, we begin by defining the first version of the GraphLab abstraction in Sec. 2.2 which comprises of

1. A **data graph** which represents the data and computational dependencies,
2. An **update function** which describe local computation on the graph,
3. **Scheduling primitives** which express the order of computation,
4. A data **consistency model**, which determines the extent to which computation can overlap,
5. And a **Sync mechanism** for aggregating global state.

Next, in Sec. 2.3, we describe an implementation of the GraphLab abstraction in the shared memory setting [Low et al., 2010], and provide an extensive evaluation of the implementation on five Machine Learning tasks, demonstrating the expressiveness and performance of the GraphLab abstraction.

Finally, in Sec. 2.4 we extend the shared memory implementation to the distributed setting [Low et al., 2012], addressing the difficulties of state partitioning, data consistency, and fault tolerance. We evaluate our implementation on three Machine Learning tasks and show that we are able to achieve good distributed performance.

2.1 Graph-Parallel Abstractions

Large-scale graph-structured computation is central to many recent advances in ML. From modeling the coupled interests of friends in a social network to the topical relationship between connected documents, graphs encode dependencies in data. The data dependencies encoded by graphs often lead to computational dependencies in ML algorithms. For example, to estimate the interests of individuals in a social network, an ML algorithm might iteratively optimize parameters for each individual conditioned on the current best parameters for their friends.

Unfortunately, high-level data-parallel frameworks such as Map-Reduce [Dean and Ghemawat, 2004] are not well suited for graph-structured computation. As a consequence some [Asuncion et al., 2009, Gonzalez et al., 2009a, Graf et al., 2005, Nallapati et al., 2007, Newman et al., 2008, Smola and Narayanamurthy, 2010] have adopted low-level tools and techniques to build large-scale distributed systems. This approach blends statistical models and algorithms with distributed system design leads to systems that are specialized and difficult to extend. In order to avoid the challenges of distributed system design, others have tried to adapt their algorithms to high-level data-parallel abstractions such as MapReduce. However, this approach does not exploit the structure of the problem and can lead to inefficient systems.

A **graph-parallel** abstraction consists of a *sparse* graph $G = \{V, E\}$ and a **vertex-program** Q which is executed in parallel on each vertex $v \in V$ and can interact (e.g., through shared-state in GraphLab, or messages in Pregel) with neighboring instances $Q(u)$ where $(u, v) \in E$. In contrast to more general message passing models, graph-parallel abstractions constrain the interaction between vertex-programs to a graph structure enabling the optimization of data-layout and communication.

For instance, Pregel is a bulk synchronous *message passing* abstraction in which all vertex-programs run simultaneously in a sequence of super-steps. Within a **super-step** each program instance $Q(v)$ receives all messages from the previous super-step and sends messages to its neighbors in the next super-step. A barrier is imposed between super-steps to ensure that all program instances finish processing messages from the previous super-step before proceeding to the next. The program terminates when there are no messages remaining and every program has voted to halt. Pregel introduces commutative associative message **combiners** which are user defined functions that merge messages destined to the same vertex.

To ground Pregel in a concrete problem, we will use the PageRank algorithm [Page et al., 1999] as a running example. While PageRank is not a common machine learning algorithm, it is easy to understand and shares many properties with other learning algorithms.

Example 1 (PageRank). *The PageRank algorithm recursively defines the rank of a webpage v :*

$$R(v) = \alpha + (1 - \alpha) \sum_{u \text{ links to } v} R(u) / \text{OutDegree}(u) \quad (2.1.1)$$

in terms of the ranks of those pages that link to v , some probability α of randomly jumping to that page. The PageRank algorithm, simply iterates Eq. (2.1.1) until the individual PageRank values converge (e.g., change by less than some small ϵ). We will let $\alpha = 0.15$ in all examples.

The following is an example of the above PageRank vertex-program implemented in Pregel. The vertex-program receives the single incoming message (after the combiner) which contains the sum of the PageRanks of all in-neighbors. The new PageRank is then computed and sent to its out-neighbors.

```

Message combiner(Message m1, Message m2) :
    return Message(m1.value() + m2.value());

void PregelPageRank(Message msg) :
    float total = msg.value();
    vertex.val = 0.15 + 0.85 * total;
    foreach(nbr in out_neighbors) :
        SendMsg(nbr, vertex.val/num_out_nbrs);

```

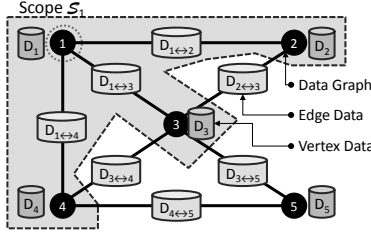


Figure 2.1: In this figure we illustrate the GraphLab **data graph** as well as the scope \mathcal{S}_1 of vertex 1. Each gray cylinder represents a block of user defined data and is associated with a vertex or edge. The **scope** of vertex 1 is illustrated by the region containing vertices $\{1, 2, 3, 4\}$. 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_{1-2}, D_{1-3} , and D_{1-4}).

2.2 GraphLab Abstraction

The GraphLab abstraction consists of three main parts, the data graph, the update function, and the sync operation. The data graph (Sec. 2.2.1) represents user modifiable program state, and stores both the mutable user-defined data and encodes the sparse computational dependencies. The update function (Sec. 2.2.2) represents the user computation and operate on the data graph by transforming data in small overlapping contexts called **scopes**. Finally, the sync operation (Sec. 2.2.5) concurrently maintains global aggregates. Once again, we will use PageRank as a running example.

2.2.1 Data Graph

The GraphLab abstraction stores the program state as a directed 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. Users can associate arbitrary data with each vertex $\{D_v : v \in V\}$ and edge $\{D_{u \rightarrow v} : \{u, v\} \in E\}$ in the graph. While the graph data is mutable, the graph structure is *static* and cannot be changed during execution. We will use $N[v]$ to denote the set of neighbors of vertex v in the graph. Since the GraphLab abstraction does not place special meaning on the edge direction but treats it simply as a property of the edge, we will frequently use D_{u-v} to refer to the contents of an edge $u - v$ when the direction of the edge is unimportant.

Example 2 (PageRank: Ex. 1). *The data graph 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 $\mathbf{R}(v)$, the current estimate of the PageRank, and the edge data $D_{u \rightarrow v}$ stores $w_{u,v}$, the directed weight of the link.*

2.2.2 Update Functions

Computation is encoded in the GraphLab abstraction in the form of 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 adjacent edges as shown in Fig. 2.1.

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

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

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 tasks \mathcal{T} are executed *eventually* by running $f(v, \mathcal{S}_v)$ following the execution semantics described in Sec. 2.2.3.

Rather than adopting a message passing or data flow model as in Isard et al. [2007], Malewicz et al. [2010], 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.

Example 3 (PageRank: Ex. 1). *The update function for PageRank (defined in Alg. 2.1) 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.*

Algorithm 2.1: PageRank Update Function

```

PageRankUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *} \in \mathcal{S}_v$ ) begin
  prev_pagerank  $\leftarrow D_v$ 
  new_pagerank  $\leftarrow 0.15$ 
  // compute the new pagerank by summing over in neighbors
  foreach  $(t \rightarrow v) \in (* \rightarrow v)$  do
    | new_pagerank  $\leftarrow$  new_pagerank  $+ 0.85 \times D_t / \text{OutDegree}(t)$ 
  // store the new pagerank
   $D_v \leftarrow$  new_pagerank
  // If the PageRank changes sufficiently
  if  $| \text{new\_pagerank} - \text{prev\_pagerank} | > \epsilon$  then
    | // Schedule out neighbors to be updated
    | foreach  $(v \rightarrow t) \in (v \rightarrow *)$  do
    | | AddTask( $t$ )

```

2.2.3 The GraphLab Execution Model

The GraphLab execution model, presented in Alg. 2.2, 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** and an initial set of tasks \mathcal{T} to update. 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). Duplicate tasks are ignored. Upon completion, the resulting data graph and global values are returned to the user.

To permit efficient scheduling, the exact behavior of $\text{RemoveNext}(\mathcal{T})$ (Line 1) is undefined and up to the implementation of the GraphLab abstraction. Instead the only guarantee the GraphLab abstraction provides is that RemoveNext removes and returns an update task in \mathcal{T} . The scheduler may accept prioritization hints from the user, but there are no guarantees regarding the effect of the hint on the order of execution. The flexibility in the order in which RemoveNext removes tasks from \mathcal{T} provides the opportunity to balance features with performance constraints.

Algorithm 2.2: GraphLab Execution Model

Input: Data Graph $G = (V, E, D)$

Input: Initial task set $\mathcal{T} = \{(f, v_1), (g, v_2), \dots\}$

while \mathcal{T} is not Empty **do**

```
1    $(f, v) \leftarrow \text{RemoveNext}(\mathcal{T})$ 
2    $(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$ 
3    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$ 
```

Output: Modified Data Graph $G = (V, E, D')$

2.2.4 Ensuring Serializability

An execution of the GraphLab abstraction must be able to enforce **serializability**. A serializable execution of the GraphLab abstraction implies that there exists a corresponding serial schedule of update functions that when executed by Alg. 2.2 produces the same data-graph. By ensuring serializable executions, GraphLab simplifies reasoning about highly-asynchronous dynamic computation in the distributed setting.

A simple method to achieve serializability is to ensure that the scopes of concurrently executing update functions do not overlap. We call this the **full consistency** model (see Fig. 2.2a). However, full consistency limits the potential parallelism since concurrently executing update functions must be at least two vertices apart (see Fig. 2.2b). However, 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. (2.1.1) only requires read access to edges and neighboring vertices. To provide greater parallelism while retaining serializability, GraphLab defines the **edge consistency** model. If the edge consistency model is used (see Fig. 2.2a), 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. 2.2b). Finally, the **vertex consistency** model allows all update functions to be run in parallel, providing only consistent access to the scope's central vertex.

2.2.5 Sync Operation and Global Values

Finally, 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 global convergence estimators. To address this need, the GraphLab abstraction defines global values which may be read by update functions but are written using the **sync operation**. The sync operation, much like the **aggregates** in Pregel, is an associative commutative operation:

$$Z = \mathbf{Finalize} \left(\bigoplus_{v \in V} \mathbf{Map}(\mathcal{S}_v) \right) \quad (2.2.1)$$

defined over all the scopes in the graph. Note that the aggregate procedure includes a **Finalize**(\cdot) operation, which is useful for performing final normalization (a common step in ML algorithms). The sync operation in GraphLab runs continuously in the background to maintain updated estimates of the global value.

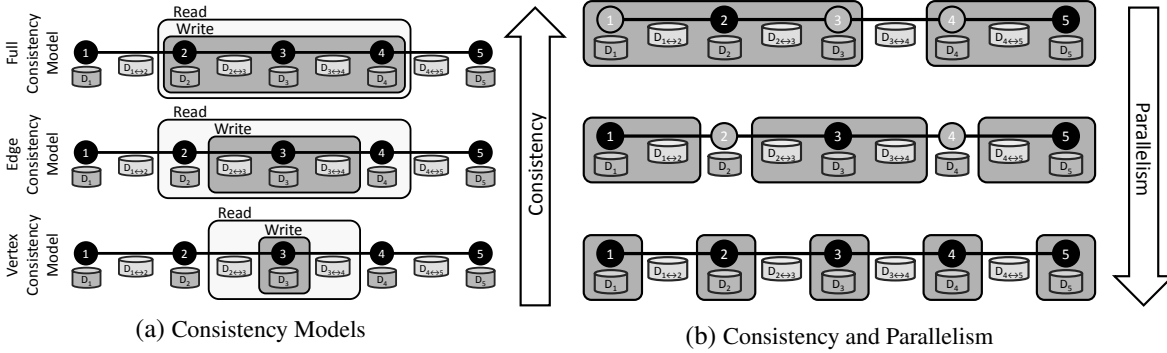


Figure 2.2: To ensure sequential consistency while providing the maximum parallelism, the GraphLab abstraction provides three different consistency models: full, edge, vertex. In figure (a), we illustrate the read and write permissions for an update function executed on the central vertex under each 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 $f(2, S_2)$ and $f(5, S_5)$ simultaneously while ensuring sequential consistency. Under the edge consistency model we are able to run three update functions (i.e., $f(1, S_1)$, $f(3, S_3)$, and $f(5, S_5)$) in parallel. Finally under the vertex consistency model we are able to run update functions on all vertices in parallel.

Use of Synced variables within the update function technically break serializability (and in fact we considered requiring Syncs to be serializable at one point), but the synchronization cost of enforcing serializability of Syncs in the distributed setting was deemed to be excessive.

2.2.6 Abstraction Summary

In summary, a GraphLab program is composed of the following parts:

1. A **data graph** which represents the data and computational dependencies.
2. **Update functions** which describe local computation
3. **Scheduling primitives** which express the order of computation and may depend dynamically on the data.
4. A data **consistency model** (i.e., *Fully Consistent*, *Edge Consistent* or *Vertex Consistent*), which determines the extent to which computation can overlap.
5. A **Sync mechanism** for aggregating global state.

2.3 Shared Memory Implementation

In Low et al. [2010], we implemented GraphLab for the shared memory setting. To summarize the results, we evaluated on five different problems: MRF Parameter Learning, Gibbs Sampling, CoEM, Lasso and Compressed sensing, and we demonstrated that the Shared Memory GraphLab implementation generally achieves good performance, obtaining excellent strong scaling. The ability to choose between different scheduler implementations (see Sec. 2.3.2), as well as the ability to pick between different consistency guarantees permit a degree of adaptivity: allowing the user to pick the optimal scheduler, and the optimal consistency level for his/her application.

To provide a basis for comparison of runtimes, we benchmarked our CoEM implementation to be 15x faster than an equivalent Hadoop implementation while using only only 16 CPUs on a shared memory system, as compared to 95 CPUs on Hadoop.

Finally, the parallel Gibbs Sampler developed here was further extended in Gonzalez et al. [2011] to perform adaptive-block-Gibbs Sampling; once again showing excellent performance and scaling.

2.3.1 Engine

The shared memory implementation of the GraphLab framework was implemented in C++ using PThreads for parallelism. The data consistency models were implemented by associated a standard PThread read-write lock to read vertex, and acquiring the locks under the following locking strategies. These locking strategies exactly represent the different consistency models.

	Central Vertex	Adjacent Vertices
Vertex Consistency	Write-Lock	-
Edge Consistency	Write-Lock	Read-Lock
Full Consistency	Write-Lock	Write-Lock

Locks are acquired in a canonical ordering to prevent dead-locks. A precise description of the locking strategy is provided in Fig. 2.3.

2.3.2 Schedulers

We implemented a collection of parallel schedulers satisfying the simple scheduler contract described in Sec. 2.2.3. We built **FIFO** schedulers which only permit task creation but do not permit task reordering and **prioritized** schedules which permit task reordering at the cost of increased overhead. For both types of task scheduler GraphLab also provide relaxed versions which increase performance at the expense of reduced control:

	Strict Order	Relaxed Order
FIFO	Single Queue	Multi Queue / Partitioned
Prioritized	Priority Queue	Approx. Priority Queue

For Loopy Belief Propagation (BP), different choices of scheduling leads to different BP algorithms. Using the Synchronous scheduler corresponds to the classical implementation of BP and using priority scheduler corresponds to Residual BP [Elidan et al., 2006]. In addition GraphLab provides the **splash**

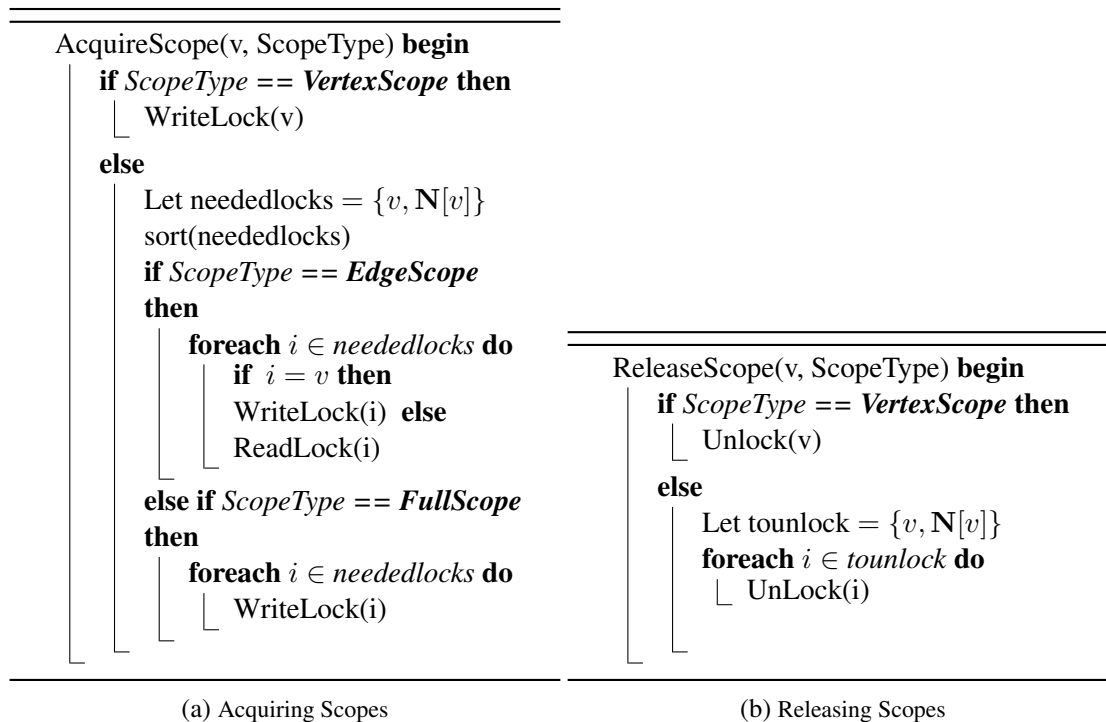


Figure 2.3: **(a)** Procedure to acquire a scope of a particular consistency (vertex, edge or full). **(b)** Procedure to release a scope of a particular consistency.

scheduler based on the loopy BP schedule proposed by Gonzalez et al. [2009b] which executes tasks along spanning trees.

We also implemented a scheduler construction framework called the **set scheduler** which enables users to safely and easily compose custom update schedules. To use the set scheduler the user specifies a sequence of vertex set and update function pairs $((S_1, f_1), (S_2, f_2) \cdots (S_k, f_k))$, where $S_i \subseteq V$ and f_i is an update function. This sequence implies the following execution semantics:

```

for i = 1 .. k do
  | Execute  $f_i$  on all vertices in  $S_i$  in parallel.
  | Wait for all updates to complete

```

The amount of parallelism depends on the size of each set; the procedure is highly sequential if the set sizes are small, and the procedure is highly parallel if the set sizes are large. Executing the schedule in the manner described above can lead to the majority of the processors waiting for a few processors to complete the current set. However, by leveraging the causal data dependencies encoded in the graph structure we are able to construct an **execution plan** which identifies tasks in future sets that can be executed *early* while still producing an equivalent result.

The set scheduler compiles an execution plan by rewriting the execution sequence as a Directed Acyclic Graph (DAG), where each vertex in the DAG represents an update task in the execution sequence and edges represent execution dependencies. Fig. 2.4 provides an example of this process. The DAG imposes a partial ordering over tasks which can be compiled into a parallel execution schedule using the greedy algorithm described by Graham [1966].

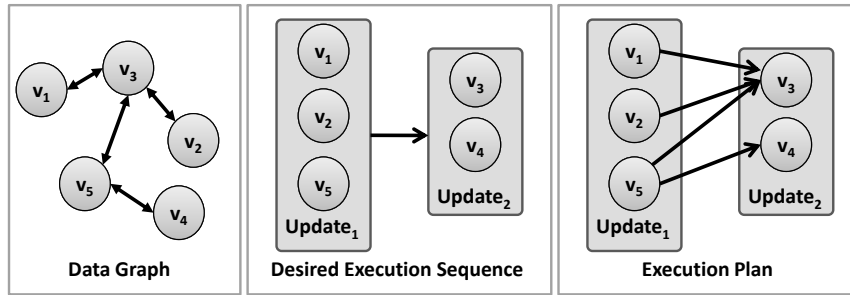


Figure 2.4: A simple example of the set scheduler planning process. Given the data graph, and a desired sequence of execution where v_1, v_2 and v_5 are first run in parallel, then followed by v_3 and v_4 . If the edge consistency model is used, we observe that the execution of v_3 depends on the state of v_1, v_2 and v_5 , but the v_4 only depends on the state of v_5 . The dependencies are encoded in the execution plan on the right. The resulting plan allows v_4 to be immediately executed after v_5 without waiting for v_1 and v_2 .

2.3.3 Evaluation

We evaluated our shared memory implementation on five different problems: MRF Parameter Learning, Gibbs Sampling, CoEM, Lasso and Compressed sensing.

2.3.3.1 Markov Random Field Parameter Learning and Inference

To demonstrate how the various components of the GraphLab framework can be assembled to build a complete ML “pipeline,” we use GraphLab to solve a novel three-dimensional retinal image denoising task. In this task we begin with raw three-dimensional laser density estimates, then use GraphLab to generate composite statistics, learn parameters for a large three-dimensional grid pairwise MRF, and then finally compute expectations for each voxel using Loopy BP. Each of these tasks requires both local iterative computation and global aggregation as well as several different computation schedules.

We begin by using the GraphLab data-graph to build a large (256x64x64) three dimensional MRF in which each vertex corresponds to a voxel in the original image. We connect neighboring voxels in the 6 axis aligned directions. We store the density observations and beliefs in the vertex data and the BP messages in the directed edge data. As global variables, we store three global edge parameters (λ) which determine the smoothing (accomplished using a Laplace similarity potentials) in each dimension. Prior to learning the model parameters, we first use the GraphLab sync mechanism to compute axis-aligned averages as a proxy for “ground-truth” smoothed images along each dimension. We then performed simultaneous learning and inference in GraphLab by using the background sync mechanism (Alg. 2.4) to aggregate inferred model statistics and apply a gradient descent procedure. To the best of our knowledge, this is the first time graphical model parameter learning and BP inference have been done concurrently.

Results: In Fig. 2.5a we plot the speedup of the parameter learning algorithm, executing inference and learning sequentially. We found that the Splash scheduler outperforms other scheduling techniques enabling a factor 15 speedup on 16 cores. We then evaluated simultaneous parameter learning and inference by allowing the sync mechanism to run concurrently with inference (Fig. 2.5b and Fig. 2.5c). By running a background sync at the right frequency, we found that we can further accelerate parameter learning while

Algorithm 2.3: BP update function

```
BPUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *} \in \mathcal{S}_v$ ) begin
  Compute the local belief  $b(x_v)$  using  $\{D_{* \rightarrow v}, D_v\}$ 
  foreach  $(v \rightarrow t) \in (v \rightarrow *)$  do
    Update  $m_{v \rightarrow t}(x_t)$  using  $\{D_{* \rightarrow v}, D_v\}$  and  $\lambda_{\text{axis}(vt)}$ .
    residual  $\leftarrow \|m_{v \rightarrow t}(x_t) - m_{v \rightarrow t}^{\text{old}}(x_t)\|_1$ 
    if residual > Termination Bound then
      AddTask( $t$ , residual)
```

Algorithm 2.4: Parameter Learning Sync

```
Fold(acc, vertex) begin
  Return acc + image statistics on vertex
Apply(acc) begin
  Apply gradient step to  $\lambda$  using acc and return  $\lambda$ 
```

only marginally affecting the learned parameters. In Fig. 2.6a and Fig. 2.6b we plot examples of noisy and denoised cross sections respectively.

2.3.3.2 Gibbs Sampling On Protein Interaction Networks

The Gibbs sampling algorithm is inherently sequential and has frustrated efforts to build asymptotically consistent parallel samplers. However, a standard result in parallel algorithms [Bertsekas and Tsitsiklis, 1989] is that for any fixed length Gauss-Seidel schedule there exists an equivalent parallel execution which can be derived from a coloring of the dependency graph. We can extract this form of parallelism using the GraphLab framework. We first use GraphLab to construct a greedy graph coloring on the MRF and then to execute an exact parallel Gibbs sampler.

Algorithm 2.5: Graph Coloring update function

```
GraphColoringUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *} \in \mathcal{S}_v$ ) begin
  NeighborColors =  $\emptyset$ 
  foreach Neighbor  $t$  do
    Insert  $D_t$  into NeighborColors
   $D_v$  = Lowest Color number not in NeighborColors
  // No rescheduling needed. If run with edge consistency, will
  // guarantee completion in 1 pass
```

We implement the standard greedy graph coloring algorithm in GraphLab by writing an update function which examines the colors of the neighboring vertices of v , and sets v to the first unused color (Alg. 2.5). We use the edge consistency model with the parallel coloring algorithm to ensure that the parallel execution retains the same guarantees as the sequential version. The parallel sampling schedule is then built using the GraphLab set scheduler (Sec. 2.3.2) and the coloring of the MRF. The resulting schedule

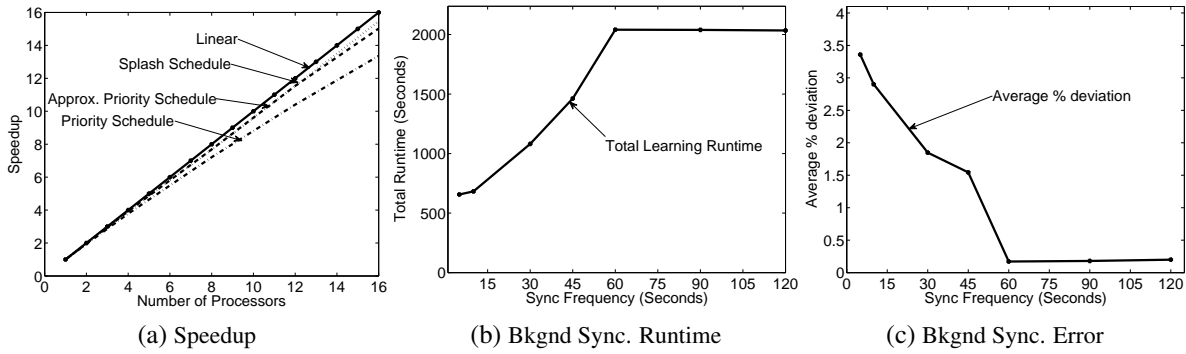


Figure 2.5: *Retinal Scan Denoising* (a) Speedup relative to the best single processor runtime of parameter learning using priority, approx priority, and Splash schedules. (b) The total runtime in seconds of parameter learning and (c) the average percent deviation in learned parameters plotted against the time between gradient steps using the Splash schedule on 16 processors.

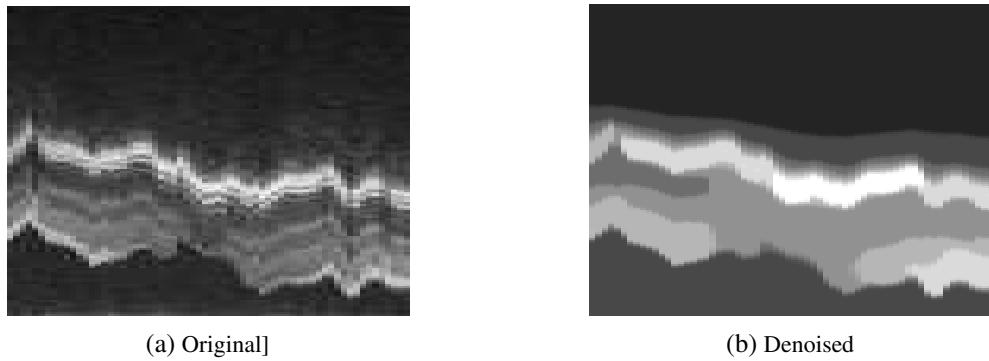


Figure 2.6: *Retinal Scan Denoising* (a,b) A slice of the original noisy image and the corresponding expected pixel values after parameter learning and denoising.

consists of a sequence of vertex sets S_1 to S_C such that S_i contains all the vertices with color i . The *vertex consistency* model is sufficient since the coloring procedure with the set scheduler ensures sequential consistency. The Gibbs Coloring update function is described in Alg. 2.6.

To evaluate the GraphLab parallel Gibbs sampler we consider the task of marginal estimation on a factor graph representing a protein-protein interaction network obtained from Elidan et al. [2006] by generating 10,000 samples. The MRF has roughly 100K edges and 14K vertices. As a baseline for comparison we also ran a GraphLab version of the Splash Loopy BP [Gonzalez et al., 2009b] algorithm.

Results: In Fig. 2.7 we present the speedup and efficiency results for Gibbs sampling and Loopy BP. Using the set schedule in conjunction with the planning optimization enables the Gibbs sampler to achieve a factor of 10 speedup on 16 processors. The execution plan takes 0.05 seconds to compute, an immaterial fraction of the 16 processor running time. Because of the structure of the MRF, a large number of colors (20) is needed and the vertex distribution over colors is heavily skewed. Consequently there is a strong sequential component to running the Gibbs sampler on this model. In contrast the Loopy BP speedup demonstrates considerably better scaling with factor of 15 speedup on 16 processor. The larger cost per BP update in conjunction with the ability to run a fully asynchronous schedule enables Loopy BP to achieve relatively uniform update efficiency compared to Gibbs sampling.

Algorithm 2.6: Gibbs Sampling Update Function

```
GibbsUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *} \in S_v$ ) begin  
  // The sampling distribution  
   $D =$  Uniform Vector over all candidate assignments  
  foreach Neighbor  $t$  do  
    EdgePotential =  $D_{v \rightarrow t}$   
    // Marginalize edge potential over neighboring assignment  
     $D = D . * \text{EdgePotential}[D_t, :]$   
  Normalize( $D$ )  
  // Sample an assignment from the distribution  
   $D_v =$  Sample from  $D$ 
```

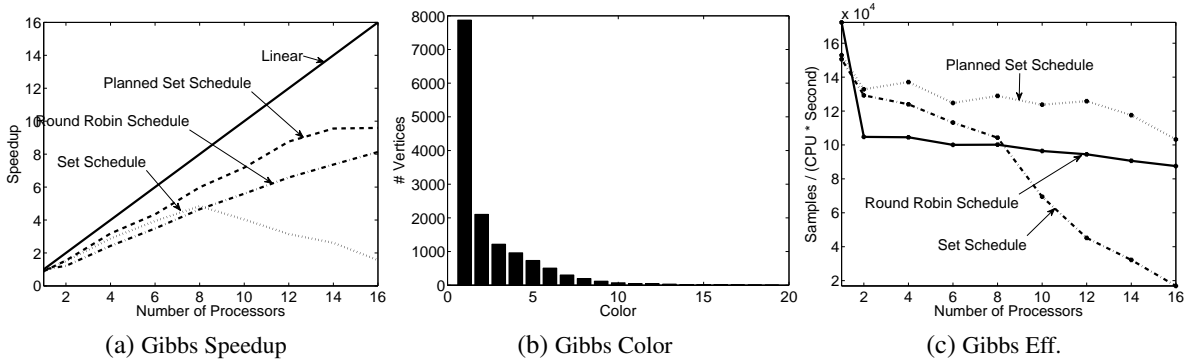


Figure 2.7: *Gibbs Sampling* (a) The speedup of the Gibbs sampler using three different schedules. The *planned set schedule* enables processors to safely execute more than one color simultaneously. The *round robin schedule* executes updates in a fixed order and relies on the edge consistency model to maintain sequential consistency. The regular *set scheduler* does not apply optimization and therefore suffers from substantial synchronization overhead. (b) The distribution of vertices over the 20 colors is strongly skewed resulting in a high sequential set schedule. (c) The sampling rate per processor plotted against the number of processor provides measure of parallel overhead which is substantially reduced by the plan optimization in the set scheduler.

2.3.3.3 Named Entity Recognition with CoEM

To illustrate how GraphLab scales in settings with large structured models we designed and implemented a parallel version of CoEM [Jones, 2005, Nigam and Ghani, 2000], a semi-supervised learning algorithm for named entity recognition (CoEM).

Named Entity Recognition (CoEM) is the task of determining the type (e.g., *Person*, *Place*, or *Thing*) of a **noun-phrase** (e.g., *Obama*, *Chicago*, or *Car*) from its **context** (e.g., “*President _*”, “*lives near _*”, or “*bought a _*”). CoEM is used in many natural language processing applications as well as information retrieval. In this application we obtained a large crawl of the web from the NELL project [Carlson et al., 2010], and we counted the number of occurrences of each noun-phrase in each context. Starting with a small seed set of pre-labeled noun-phrases, the CoEM algorithm labels the remaining noun-phrases and contexts (see Table 2.1) by alternating between estimating the best assignment to each noun-phrase given the types of its contexts and estimating the type of each context given the types of its noun-phrases.

The data graph for the CoEM problem is bipartite with one set of vertices corresponding to noun-phrases

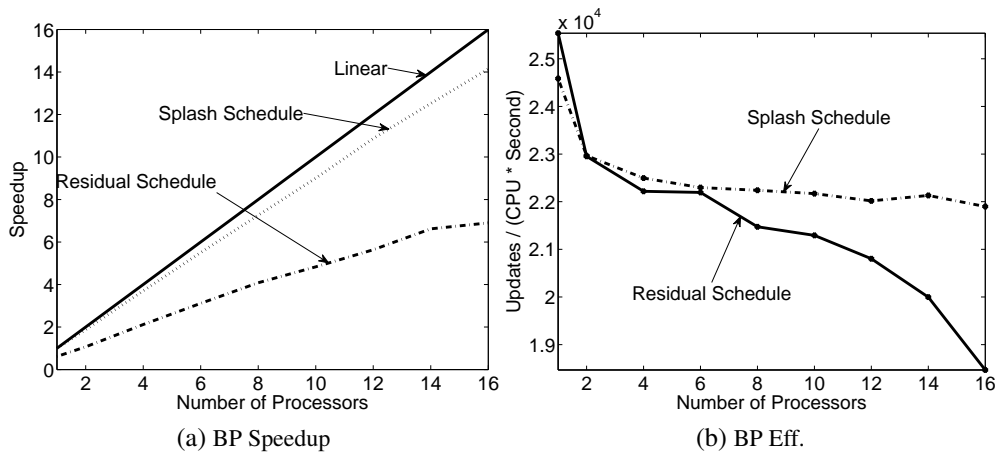


Figure 2.8: *Belief Propagation* (a) The speedup for Loopy BP is improved substantially by the Splash. (b) The efficiency of the GraphLab framework as function of the number of processors.

Food	Religion	City
onion	Catholic	Munich
garlic	Freemasonry	Cape Twn.
noodles	Marxism	Seoul
blueberries	Catholic Chr.	Mexico Cty.
beans	Humanism	Winnipeg

Table 2.1: **CoEM**: Top words for several categories.

and other set corresponding to contexts. There is an edge between a noun-phrase and a context if the noun-phrase occurs in the context. The vertices store the estimated distribution over types and the edges store the number of times the noun-phrase appears in a context. The update function for CoEM recomputes the local belief by taking a weighted average of the adjacent vertex beliefs. The adjacent vertices are rescheduled if the belief changes by more than some predefined threshold (10^{-5}). The algorithm is described in Alg. 2.7.

We experimented with the following two CoEM datasets obtained from web-crawling data.

Name	Classes	Verts.	Edges	1 CPU Runtime
small	1	0.2 mil.	20 mil.	40 min
large	135	2 mil.	200 mil.	8 hours

We plot in Fig. 2.9a and Fig. 2.9b the speedup obtained by the Partitioned Scheduler and the MultiQueue FIFO scheduler, on both small and large datasets respectively. We observe that both schedulers perform similarly and achieve nearly linear scaling. In addition, both schedulers obtain similar belief estimates suggesting that the update schedule may not affect convergence in this application.

With 16 parallel processors, we could complete three full Round-robin iterations on the large dataset in less than 30 minutes. As a comparison, a comparable Hadoop implementation took approximately 7.5 hours to complete the exact same task, executing on an average of 95 cpus. [Personal communication with Justin Betteridge and Tom Mitchell, Mar 12, 2010]. Our large performance gain can be attributed to data persistence in the GraphLab framework. Data persistence allows us to avoid the extensive data copying

Algorithm 2.7: CoEM update function

```
CoEMUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) begin
   $D = [0]$ 
  // Distribution of classes at a vertex is a weighted sum of
  // neighbors
  foreach Neighbor vertex  $t$  do
     $D = D + D_{v \rightarrow t} \times D_t$ 
   $D_v = \text{Normalize}(D)$ 
  // Schedule neighbors if there is substantial change
  foreach Neighbor vertex  $t$  do
    if estimated change to neighbor  $> 10^{-5}$  then
      AddTask( $t$ )
```

and synchronization required by the Hadoop implementation of MapReduce.

Using the flexibility of the GraphLab framework we were able to study the benefits of dynamic (Multi-queue FIFO) scheduling versus a regular round-robin scheduling in CoEM. Fig. 2.9c compares the number of updates required by both schedules to obtain a result of comparable quality on the larger dataset. Here we measure quality by L_1 distance to an estimate of the fixed point x^* , obtained by running a large number of synchronous iterations. For this application we do not find a substantial benefit from dynamic scheduling.

We also investigated how GraphLab scales with problem size. Figure 2.9d shows the maximum speedup on 16 CPUs attained with varying graph sizes, generated by sub-sampling a fraction of vertices from the large dataset. We find that parallel scaling improves with problem size and that even on smaller problems GraphLab is still able to achieve a factor of 12 speedup on 16 cores.

2.3.3.4 Lasso with the Shooting Algorithm

The Lasso [Tibshirani, 1996] is a popular feature selection and shrinkage method for linear regression which minimizes the objective $L(w) = \sum_{j=1}^n (w^T x_j - y_j)^2 + \lambda \|w\|_1$. During the time of writing Low et al. [2010], there did not exist to the best of our knowledge, a parallel algorithm for fitting a Lasso model. In this section we implement two different parallel algorithms for solving the Lasso.

We use GraphLab to implement the Shooting Algorithm [Fu, 1998], a popular Lasso solver, and demonstrate that GraphLab is able to *automatically* obtain parallelism by identifying operations that can execute concurrently while retaining sequential consistency.

The shooting algorithm works by iteratively minimizing the objective with respect to each dimension in w , corresponding to coordinate descent. We can formulate the Shooting Algorithm in the GraphLab framework as a bipartite graph with a vertex for each weight w_i and a vertex for each observation y_j . An edge is created between w_i and y_j with weight $X_{i,j}$ if and only if $X_{i,j}$ is non-zero. We also define an update function (Alg. 2.8) which operates only on the weight vertices, and corresponds exactly to a single minimization step in the shooting algorithm. A round-robin scheduling of Alg. 2.8 on all weight vertices corresponds exactly to the sequential shooting algorithm. We automatically obtain an equivalent parallel

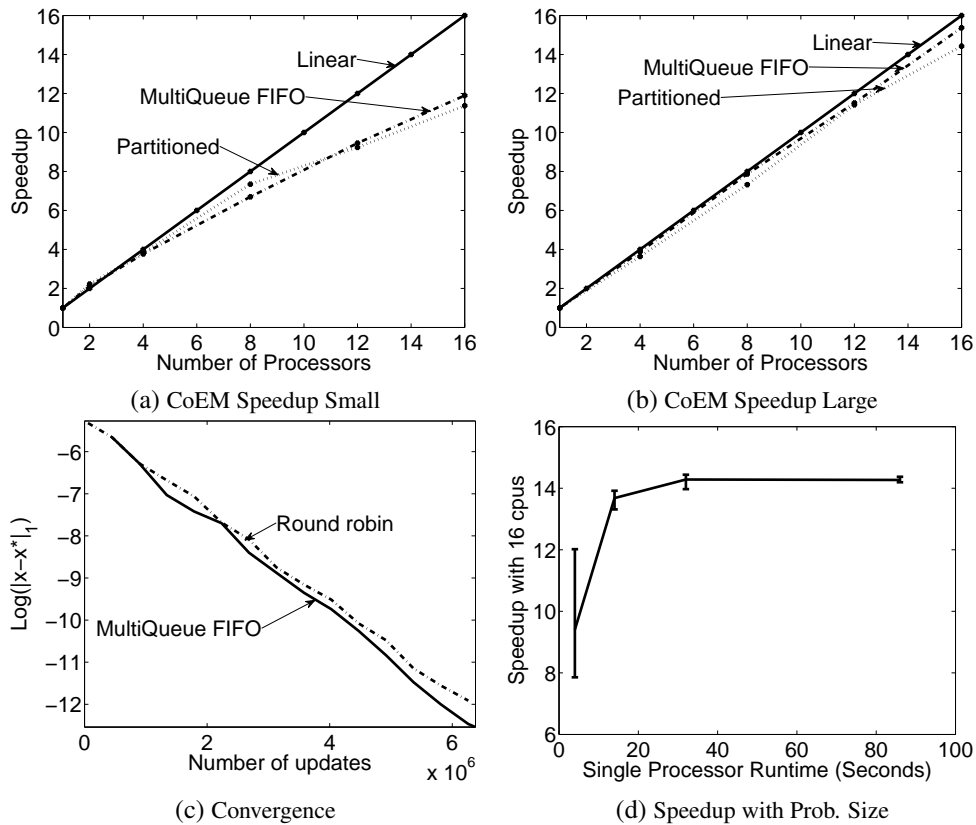


Figure 2.9: *CoEM Results (a,b)* Speedup of MultiQueue FIFO and Partitioned Scheduler on both datasets. Speedup is measured relative to fastest running time on a single cpu. The large dataset achieves better scaling because the update function is slower. (c) Speed of convergence measured in number of updates for MultiQueue FIFO and Round Robin (equivalent to synchronized Jacobi schedule), (d) Speedup achieved with 16 cpus as the graph size is varied.

algorithm by select the full consistency model. Hence, by encoding the shooting algorithm in GraphLab we are able to discover a sequentially consistent *automatic parallelization*.

We evaluate the performance of the GraphLab implementation on a financial data set obtained from Kogan et al. [2009]. The task is to use word counts of a financial report to predict stock volatility of the issuing company for the consequent 12 months. Data set consists of word counts for 30K reports with the related stock volatility metrics.

Results: To demonstrate the scaling properties of the full consistency model, we create two datasets by deleting common words. The sparser dataset contains 209K features and 1.2M non-zero entries, and the denser dataset contains 217K features and 3.5M non-zero entries. The speedup curves are plotted in Fig. 2.10. We observed better scaling (4x at 16 CPUs) on the sparser dataset than on the denser dataset (2x at 16 CPUs). This demonstrates that ensuring full consistency on denser graphs inevitably increases contention resulting in reduced performance.

Additionally, we experimented with relaxing the consistency model, and we discovered that the shooting algorithm still converges under the weakest vertex consistency guarantees; obtaining solutions with only

Algorithm 2.8: Shooting Algorithm

ShootingUpdate($D_{w_i}, D_*, D_{* \rightarrow w_i}, D_{w_i \rightarrow *}$ $\in \mathcal{S}_{w_i}$) **begin**
 Minimize the loss function with respect to w_i
 if w_i changed by $> \epsilon$ **then**
 Revise the residuals on all y 's connected to w_i
 Schedule all w 's connected to neighboring y 's

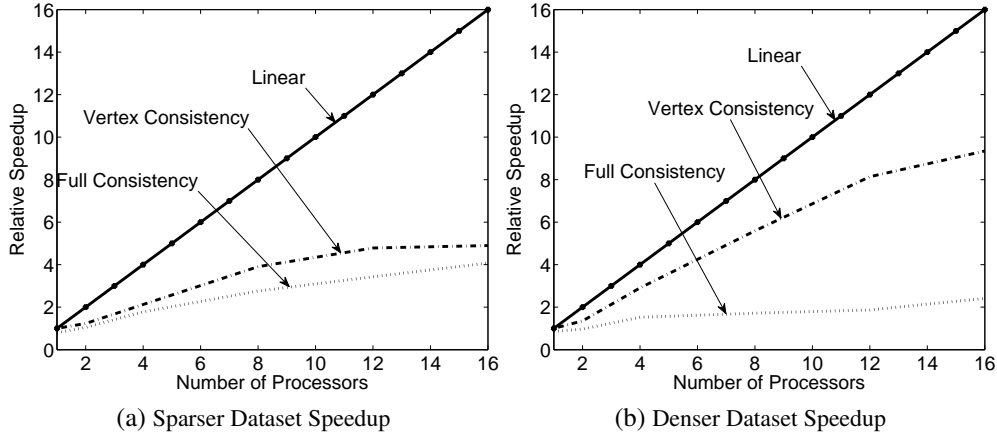


Figure 2.10: *Shooting Algorithm* (a) Speedup on the sparser dataset using *vertex consistency* and *full consistency* relative to the fastest single processor runtime. (b) Speedup on the denser dataset using *vertex consistency* and *full consistency* relative to the fastest single processor runtime.

0.5% higher loss on the same termination criterion. The vertex consistent model is much more parallel and we can achieve significantly better speedup, especially on the denser dataset. After the writing of Low et al. [2010], the Shotgun procedure [Bradley et al., 2011] was developed which provided theoretical justifications for the use of relaxed consistency in the shooting algorithm.

2.3.3.5 Compressed Sensing

To show how GraphLab can be used as a subcomponent of a larger *sequential* algorithm, we implement a variation of the interior point algorithm proposed by Kim et al. [2007] for the purposes of compressed sensing. The aim is to use a sparse linear combination of basis functions to represent the image, while minimizing the reconstruction error. Sparsity is achieved through the use of elastic net regularization.

The interior point method is a double loop algorithm where the sequential outer loop (Alg. 2.10) implements a Newton method while the inner loop computes the Newton step by solving a sparse linear system using GraphLab. We used Gaussian BP (GaBP) as a linear solver [Bickson, 2008] since it has a natural GraphLab representation. The GaBP GraphLab construction follows closely the BP example in Sec. 2.3.3.1, but represents potentials and messages analytically as Gaussian distributions. In addition, the outer loop uses a Sync operation on the data graph to compute the duality gap and to terminate the algorithm when the gap falls below a predefined threshold. Because the graph structure is fixed across iterations, we can leverage data persistency in GraphLab, avoid both costly set up and tear down operations and resume from the converged state of the previous iteration.

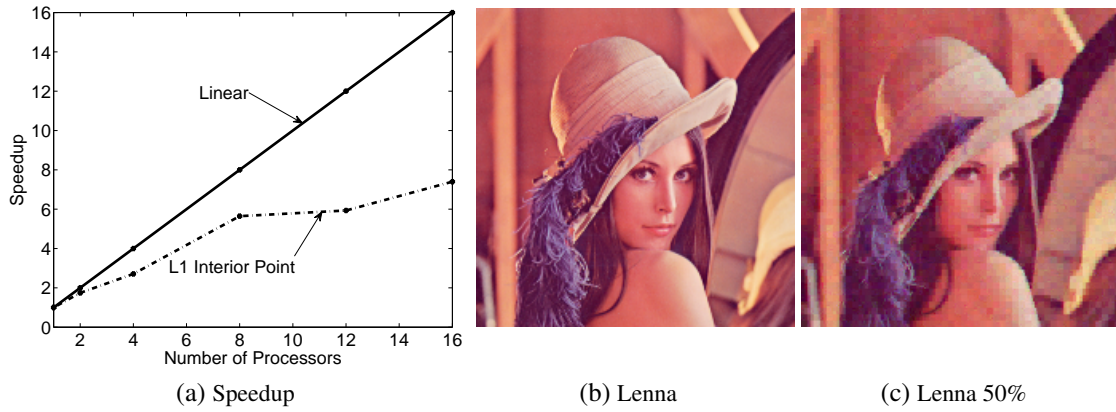


Figure 2.11: (a) Speedup of the Interior Point algorithm on the compressed sensing dataset, (b) Original 256x256 test image with 65,536 pixels, (c) Output of compressed sensing algorithm using 32,768 random projections.

We evaluate the performance of this algorithm on a synthetic compressed sensing dataset constructed by applying a random projection matrix to a wavelet transform of a 256×256 Lenna image (Fig. 2.11). Experimentally, we achieved a factor of 8 speedup using 16 processors using the round-robin scheduling.

Algorithm 2.9: GaBPUpdateFunction

```
GABPUpdateFunction( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in S_v$ ) begin
  Compute Gaussian belief  $b(x_v)$  with  $\{D_{* \rightarrow v} D_v\}$ 
  foreach  $(v \rightarrow t) \in (v \rightarrow *)$  do
    Update  $m_{v \rightarrow t}(x_t)$  using  $\{D_{* \rightarrow v}, D_v\}$ 
```

Algorithm 2.10: Compressed Sensing Outer Loop

```
while  $duality\_gap \geq \epsilon$  do
  Update edge and node data of the data graph.
  Use GraphLab to run GaBP on the graph
  Use Sync to compute duality gap
  Take a newton step
```

2.4 Distributed Implementation

The usefulness of the shared memory GraphLab system, encouraged us to provide greater scalability and performance by extending to the distributed setting [Low et al., 2012]. In this section, we discuss the techniques required to achieve this goal. We focus on in distributed **in-memory** setting, requiring the entire graph and all program state to reside in distributed RAM. Our distributed implementation is written in C++ and extends the previous shared memory GraphLab implementation.

The key challenges in extending the shared memory implementation to the distributed setting are:

State Partitioning How to efficiently load, partition and distribute the data graph across machines. We address this using a two-phase partitioning procedure described in Sec. 2.4.1.

Consistency The GraphLab abstraction specifies three grades of computation consistency: i.e. vertex, edge, and full consistency (Sec. 2.2.4). While consistency is easy to implement in the shared memory setting, achieving consistency in the distributed setting is substantially more difficult. We discuss two solutions in the Engines section Sec. 2.4.2.

Fault Tolerance Snapshotting is a common scheme for achieving fault tolerance in the distributed setting. While “stop the world” snapshots are always feasible, the sparse computation structure of GraphLab permits the use of an asynchronous snapshot procedure which we describe in Sec. 2.4.3.

2.4.1 The Data Graph

Efficiently implementing the data graph in the distributed setting requires balancing computation, communication, and storage. To ensure balanced computation and storage, each machine must hold only a small fraction of the data graph. At the same time we want to minimize the state that must be synchronized across machines or equivalently the number of edges in the data graph that cross between machines. Finally, because the Cloud setting enables the size of the cluster to vary with budget and performance demands, we must be able to quickly load the data-graph on varying sized cloud deployments. To resolve these challenges, we developed a graph representation based on two-phased partitioning which can be efficiently load balanced on arbitrary cluster sizes.

The data graph is initially over-partitioned by an expert, or by using a distributed graph partitioning heuristic (for instance ParMetis [Karypis and Kumar, 1998], or even simple random hashed partitioning) into k parts where k is much greater than the number of machines. Each part (called an atom) is stored as a separate file on a distributed storage system (e.g., HDFS, Amazon S3). The connectivity structure of the k atoms are then represented as a **meta-graph** with k vertices, where each vertex of the **meta-graph** represents an atom, and is weighted by the amount of data it stores, and each edge is weighted by the number of edges crossing the atoms. A simple demonstration of this procedure is shown in Fig. 2.12.

Distributed loading is accomplished by performing a fast balanced partition of the meta-graph over the number of physical machines, assigning graph atoms to machines. This partitioning phase is extremely fast since the meta graph is very small (only hundreds to thousands of atoms). Therefore, while the two-phase partitioning procedure requires a lengthy initial preprocessing phase of overpartitioning the graph, it can adapt the partition to varying cluster sizes easily. In Fig. 2.13, we briefly evaluate the quality of the two-phase partitioning procedure on the web graph from the Google programming competition 2002¹

¹<http://snap.stanford.edu/data/web-Google.html>

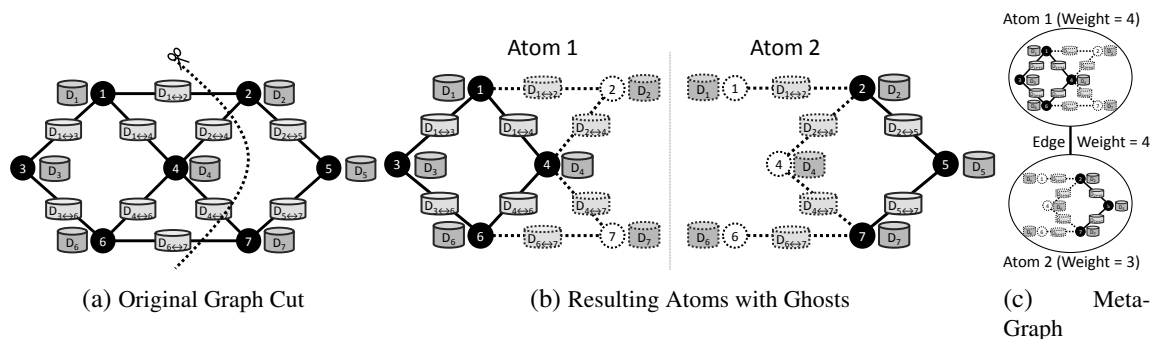


Figure 2.12: *Partitions and Ghosts*. To represent a distributed graph we partition the graph into files each containing fragments of the original graph. In (a) we cut a graph into two (unbalanced) pieces. In (b) we illustrate the ghosting process for the resulting two atoms. The edges, vertices, and data illustrated with broken edges are **ghosts** in the respective atoms. The meta-graph (c) is a weighted graph of two vertices.

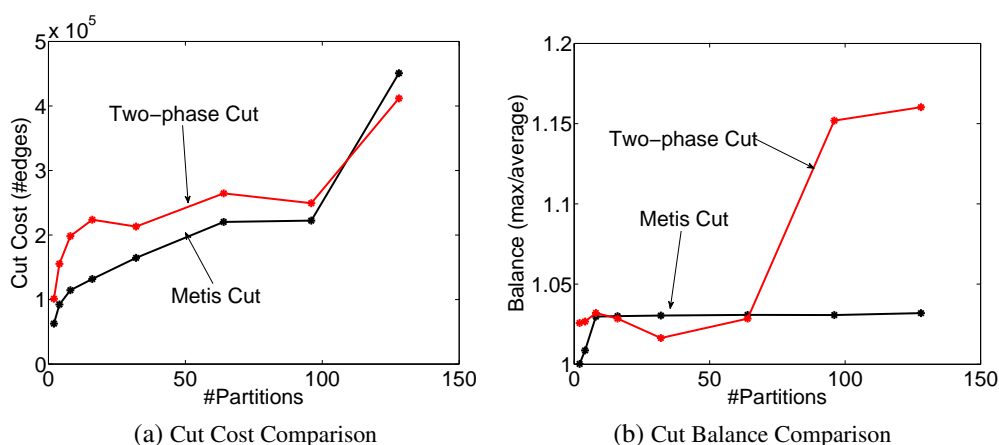


Figure 2.13: *Two Phase Cut Quality*. Comparison of the two-phase cut procedure vs a direct Metis cut on web graph from the Google programming competition 2002. For the two-phase cut, the graph was originally partitioned with Metis into 512 components to form the meta-graph. (a) The number of edges cut by both procedures varying the number of partitions. (b) The balance achieved by both procedures. (size of largest partition / average partition size)

with 875,713 vertices and 5,105,039 edges. The two-phase cut first uses Metis to over-partition the graph into 512 partitions to form a 512-node meta-graph. The meta-graph is then re-partitioned (again using Metis) to obtain the desired number of partitions. We observe that the two-phase cut procedure does not substantially affect the cut cost. Cut balance worsens rapidly beyond 100 partitions, but the quality is still tolerable (largest partition is only 15% larger than average).

In the implementation, each atom file is a simple binary compressed journal of graph generating commands which stores information about the interior of the partition, as well as the **ghosts** of the partition (the set of vertices and edges adjacent to the partition boundary). To load an atom, the machine simply constructs its local portion of the graph by playing back the journal. The ghosts are then used as caches for their true counterparts across the network. Cache coherence is managed using a simple but efficient versioning system [Bernstein and Goodman, 1981], eliminating the transmission of unchanged or constant data (e.g., edge weights).

2.4.2 Engines

We evaluate two engine designs: a low overhead **Chromatic Engine**, which requires a graph coloring, and executes tasks partially asynchronously, and the more expressive **Locking Engine** which is fully asynchronous.

2.4.2.1 Chromatic Engine

A classic technique [Bertsekas and Tsitsiklis, 1989] to achieve a serializable parallel execution of a set of dependent tasks (represented as vertices in a graph) is to construct a vertex coloring which assigns a color to each vertex such that no adjacent vertices share the same color. Given a vertex coloring of the data graph, we can satisfy the *edge consistency model* by executing, *synchronously*, all update tasks associated with vertices of the same color before proceeding to the next color. Other consistency models can be satisfied analogously. While graph coloring is NP-Hard in general, a reasonable quality graph coloring can be constructed quickly using graph coloring heuristics. Furthermore, many ML problems produce graphs with trivial colorings. For example, many optimization problems in ML are naturally expressed as bipartite graphs (two-colorable), while problems based upon templated models [Koller and Friedman, 2009] can be easily colored using the template [Gonzalez et al., 2011].

The chromatic engine operates in synchronous phases, with a distributed barrier at the end of each color. To maximize use of network bandwidth, changes to ghost vertices and edges communicated asynchronously as they are made. Care is made to ensure that all modifications are communicated before moving to the next color.

2.4.2.2 Distributed Locking Engine

While the chromatic engine satisfies the distributed GraphLab abstraction defined in Sec. 2.2, it does not provide sufficient scheduling flexibility for many interesting applications. In addition, it presupposes the availability of a graph coloring, which may not always be readily available. To overcome these limitations, we introduce the distributed locking engine which extends the mutual exclusion technique used in the shared memory engine.

We achieve distributed mutual exclusion by associating a readers-writer lock with each vertex. The different consistency models can then be implemented using different locking protocols. Vertex consistency is achieved by acquiring a write-lock on the central vertex of each requested scope. Edge consistency is achieved by acquiring a write lock on the central vertex, and read locks on adjacent vertices. Finally, full consistency is achieved by acquiring write locks on the central vertex and all adjacent vertices. Deadlocks are avoided by acquiring locks sequentially following a canonical order. We use the ordering induced by machine ID followed by vertex ID ($\text{owner}(v), v$) since this allows all locks on a remote machine to be requested in a single message.

Since the graph is partitioned, we restrict each machine to only run updates on local vertices. The ghost vertices/edges ensure that the update have direct memory access to all information in the scope. Each machine maintains a local scheduler datastructure (like those in Sec. 2.3.2), which maintains an update schedule over only local vertices. Each worker thread on each machine evaluates the loop described in Alg. 2.14a until the scheduler is empty. Termination is evaluated using the distributed consensus algorithm described in Misra [1983].

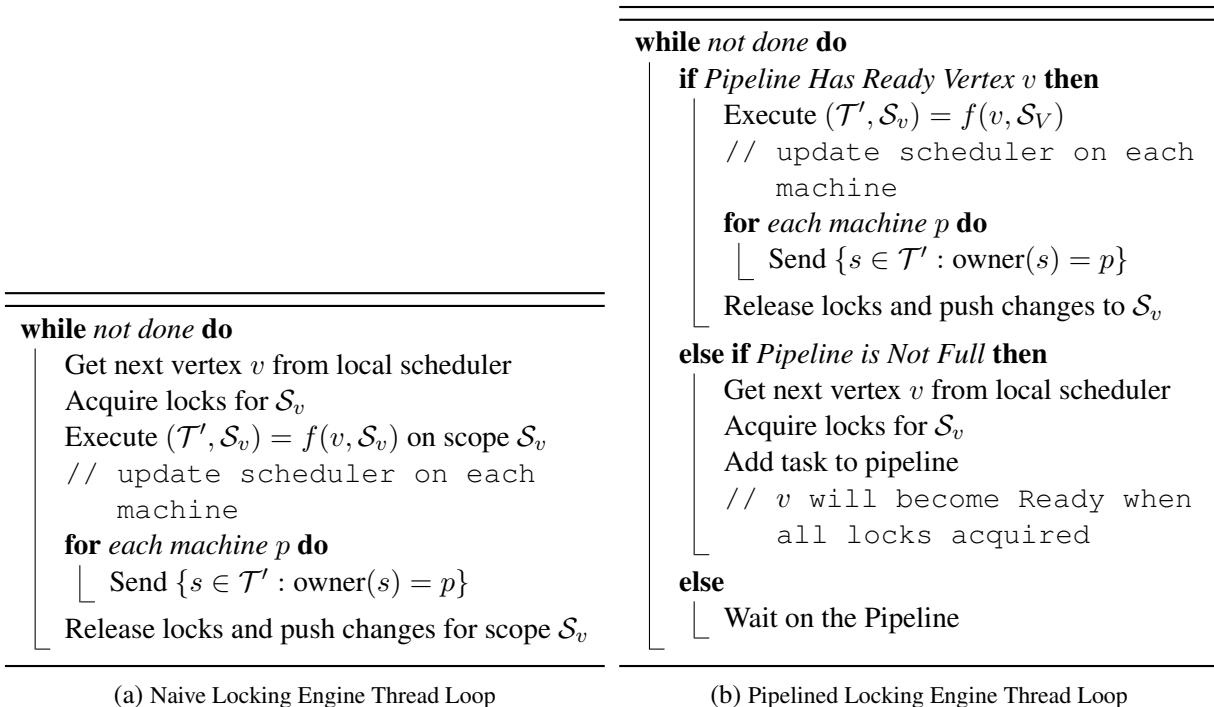


Figure 2.14: The Naive (a), and the Pipelined (b) locking engine procedures.

A naive implementation Alg. 2.14a will perform poorly due to the latency of remote lock acquisition and data synchronization. We therefore rely on several techniques to both reduce latency and hide its effects [Gupta et al., 1991]. First, the ghosting system provides caching capabilities eliminating the need to transmit or wait on data that has not changed remotely. Second, all lock requests and synchronization calls are *pipelined* allowing each machine to request locks and data for many scopes simultaneously and then evaluate the update function only when the scope is ready.

Pipelined Locking and Prefetching: Each machine maintains a pipeline of vertices for which locks have been requested, but have not been fulfilled. Vertices that complete lock acquisition and data synchronization leave the pipeline and are executed by worker threads. The local scheduler ensures that the pipeline is always filled to capacity. An overview of the pipelined locking engine loop is shown in Alg. 2.14b.

To implement the pipelining system, regular readers-writer locks cannot be used since they would halt the pipeline thread on contention. We therefore implemented a non-blocking variation of the readers-writer lock that operates through callbacks. Lock acquisition requests provide a pointer to a callback, that is called once the request is fulfilled. These callbacks are chained into a distributed continuation passing scheme that passes lock requests across machines in sequence. Since lock acquisition follows the total ordering described earlier, deadlock free operation is guaranteed. To further reduce latency, synchronization of locked data is performed immediately as each machine completes its local locks.

Example 4. *To acquire a distributed edge consistent scope on a vertex v owned by machine 2 with ghosts on machines 1 and 5, the system first sends a message to machine 1 to acquire a local edge consistent scope on machine 1 (write-lock on v , read-lock on neighbors). Once the locks are acquired, the message is passed on to machine 2 to again acquire a local edge consistent scope. Finally, the message is sent to machine 5 before returning to the owning machine to signal completion.*

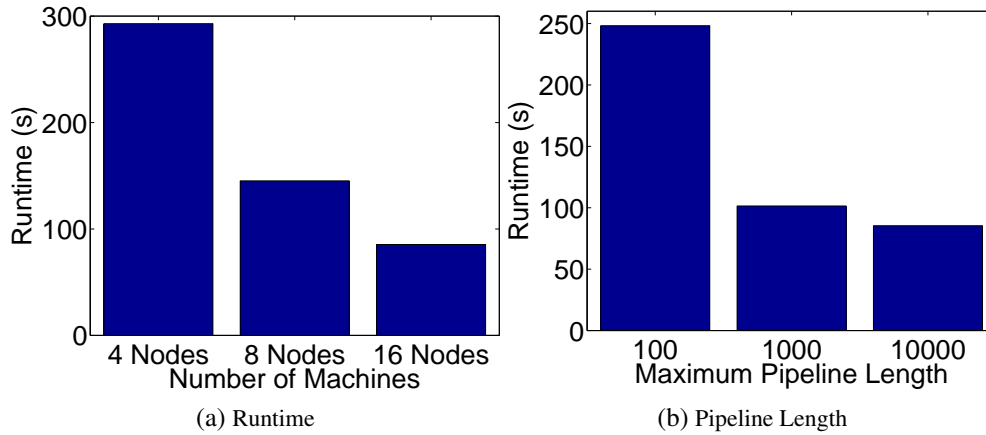


Figure 2.15: **(a)** Plots the runtime of the Distributed Locking Engine on a synthetic loopy belief propagation problem varying the number of machines with pipeline length = 10,000. **(b)** Plots the runtime of the Distributed Locking Engine on the same synthetic problem on 16 machines (128 CPUs), varying the pipeline length. Increasing pipeline length improves performance with diminishing returns.

To evaluate the performance of the distributed pipelining system, we constructed a three-dimensional mesh of $300 \times 300 \times 300 = 27,000,000$ vertices. Each vertex is 26-connected (to immediately adjacent vertices along the axis directions, as well as all diagonals), producing over 375 million edges. The graph is partitioned using Metis [Karypis and Kumar, 1998] into 512 atoms. We interpret the graph as a binary Markov Random Field [Gonzalez et al., 2009b] and evaluate the runtime of 10 iterations of loopy Belief Propagation [Gonzalez et al., 2009b] varying the length of the pipeline from 100 to 10,000, and the number of EC2 cluster compute instance (`cc1.4xlarge`) from 4 machines (32 processors) to 16 machines (128 processors). We observe in Fig. 2.15a that the distributed locking system provides strong, nearly linear, scalability. In Fig. 2.15b we evaluate the efficacy of the pipelining system by increasing the pipeline length. We find that increasing the length from 100 to 1000 leads to a *factor of three* reduction in runtime.

2.4.3 Fault Tolerance

We used distributed snapshots to achieve the first fault tolerant implementation of the GraphLab abstraction. Periodically, the state of the system is *consistently* flushed to disk, and in the event of a failure, the system can be recovered from the last snapshot. We evaluate two strategies to construct consistent snapshots: a synchronous method which freezes all computation while the snapshot is constructed, and an asynchronous method which incrementally constructs a snapshot without interrupting execution.

The synchronous snapshot systems periodically, at user-defined intervals, signals all computation activity to halt and then synchronizes all caches (ghosts) and saves to disk all data which has been modified since the last snapshot.

Synchronous snapshotting, is simple, but eliminates the systems advantage of asynchronous computation; reintroducing the “curse of the last reducer” [Suri and Vassilvitskii, 2011]. We therefore also implemented a fully asynchronous snapshotting algorithm based on the Chandy-Lamport algorithm described in Chandy and Lamport [1985]. It is interesting to note that the Chandy-Lamport snapshotting algorithm can be

implemented easily using little more than regular GraphLab update functions with the edge-consistency model:

Algorithm 2.11: Snapshot Update on vertex v

```

if  $v$  was already snapshotted then
  └ Quit
Save  $D_v$  // Save current vertex
// Save all edges connected to un-snapshotted vertices
foreach  $u \in \mathcal{N}[v]$  do // Loop over neighbors
  └ if  $u$  was not snapshotted then
    └ Save  $D_{u \rightarrow v}$  if edge  $u \rightarrow v$  exists
    └ Save  $D_{v \rightarrow u}$  if edge  $v \rightarrow u$  exists
    └ Reschedule  $u$  for a Snapshot Update
Mark  $v$  as snapshotted

```

Theorem 2.4.1. *The procedure described in Alg. 2.11 guarantees a valid serializable snapshot under the following conditions:*

1. *Edge Consistency is used on all update functions.*
2. *Reschedule completes before the scope is unlocked.*
3. *The Snapshot Update takes priority over all other update functions.*

The proof of correctness follows easily by making the following transformations and showing equivalence to Chandy-Lamport [Chandy and Lamport, 1985].

- Vertices correspond to Machines and edges represent communication channels.
- Modifications to D_v by vertex v correspond to the machine v broadcasting the vertex data to all adjacent machines.
- Modification of either $D_{u \rightarrow v}$ or $D_{v \rightarrow u}$ by vertex v corresponds to machine v sending the new data to machine u .
- The “snapshotted” flag corresponds to the marker in Chandy-Lamport.

Observe that given the conditions stated, there cannot exist a message sent from an un-snapshotted vertex to a snapshotted vertex. This eliminates the need for “channel recording” in the Chandy-Lamport algorithm.

We evaluate the performance of the snapshotting algorithms on the same synthetic mesh problem described on the previous section, running on 16 machines (128 processors). We configure the implementation to issue exactly one snapshot in the middle of the second iteration. In Fig. 2.16a we plot the number of updates completed against time elapsed. The effect of the synchronous snapshot and the asynchronous snapshot can be clearly observed: synchronous snapshots stops execution, while the asynchronous snapshot only slows down execution. Both runs terminate at nearly the same runtime (104s for async. vs 109s for sync.).

However, the benefits of asynchronous snapshotting become more apparent in the **multi-tenancy** setting where variation in system performance due to external load exacerbates the cost of synchronous

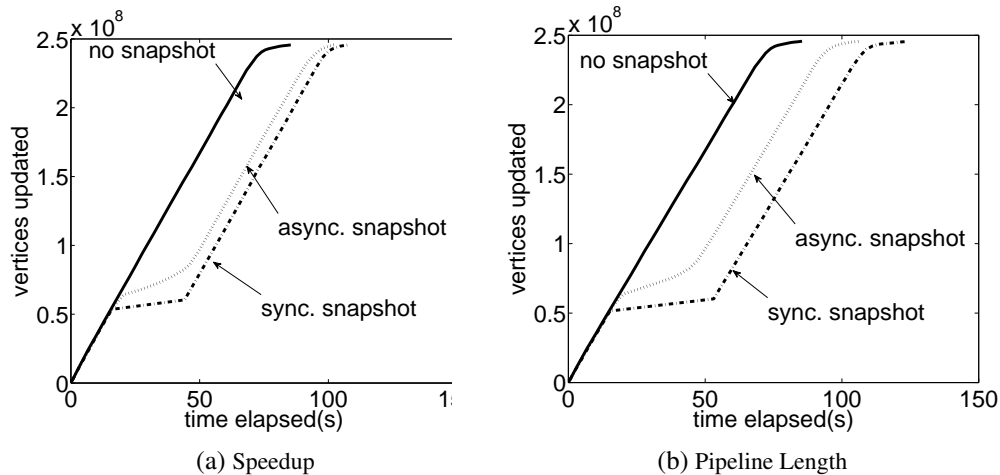


Figure 2.16: **(a)** The number of vertices completed vs time elapsed for 10 iterations comparing asynchronous and synchronous snapshotting. Synchronous snapshotting has the characteristic “flatline” while asynchronous is a smooth curve. 10 iterations with an asynchronous snapshot completed in **104s** while 10 iterations with a synchronous snapshot took **109s**. **(b)** Same as (a) but with a single machine fault injection lasting 15 seconds. runtime with asynchronous snapshotting is **107s** while runtime with synchronous snapshot took **125s**.

operations. We simulate this on Amazon EC2 by halting one of the processes for 15 seconds after snapshotting begins. In figures Fig. 2.16b we plot the resultant curves of number updates completed against time elapsed. Observe that asynchronous snapshotting is minimally affected by the simulated failure, increasing in runtime to 107s, while synchronous snapshotting experiences a maximal impact, increasing in runtime by slightly over 15s to 125s.

2.4.4 Evaluation

Finally, we implement three algorithms in the distributed setting: Netflix Matrix Factorization, A video co-segmentation problem using BP and Gaussian Mixture Models in an Expectation Maximization (EM) setting and the CoEM algorithm for Noun Entity Recognition. The graph sizes are shown in Table 2.2. The video co-segmentation problem uses the Locking Engine and a Priority scheduler, while CoEM and Netflix uses the Chromatic Engine. Both CoEM and the Netflix problem are also evaluated against MPI and Hadoop. Performance results are plotted in Fig. 2.19.

Experiments were performed on Amazon’s Elastic Computing Cloud (EC2) using up to 64 High-Performance Cluster (HPC) instances (cc1.4xlarge) each with dual Intel Xeon X5570 quad-core Nehalem processors and 22 GB of memory and connected by a 10 Gigabit Ethernet network. All timings include

Exp.	#Verts	#Edges	Vertex Data	Edge Data	Update Complexity	Shape	Partition	Engine
Netflix	0.5M	99M	$8d + 13$	16	$O(d^3 + deg.)$	bipartite	random	Chromatic
CoSeg	10.5M	31M	392	80	$O(deg.)$	3D grid	frames	Locking
CoEM	2M	200M	816	4	$O(deg.)$	bipartite	random	Chromatic

Table 2.2: Distributed experiment input sizes. The vertex and edge data are measured in bytes.

data loading and are averaged over three or more runs. On each node, GraphLab spawns eight engine threads (matching the number of cores). Numerous other threads are spawned for background communication.

In Fig. 2.19a we present an aggregate summary of the parallel speedup of GraphLab when run on 4 to 64 HPC machines on all three applications. In all cases, speedup is measured relative to the *four node deployment* since single node experiments were not always feasible due to memory limitations. No snapshots were constructed during the timing experiments.

Our principal findings are:

- On equivalent tasks, GraphLab outperforms Hadoop by 20-60x and performance is comparable to tailored MPI implementations.
- GraphLab’s performance scaling improves with higher computation to communication ratios.
- The GraphLab abstraction more compactly expresses the Netflix, CoEM and Coseg algorithms than MapReduce or MPI.

2.4.4.1 Netflix Movie Recommendation

The Netflix movie recommendation task uses *collaborative filtering* to predict the movie ratings for each user, based on the ratings of similar users. We implemented the **alternating least squares (ALS)** algorithm Zhou et al. [2008], a common algorithm in collaborative filtering. The input to ALS is a sparse users by movies matrix R , containing the movie ratings of each user. The algorithm iteratively computes a low-rank matrix factorization:

$$\begin{array}{c} \text{Movies} \\ \boxed{\begin{array}{c} \text{Users} \\ R \\ \text{Sparse} \end{array}} \approx \begin{array}{c} d \\ \boxed{\text{Users} \\ U} \end{array} \times \begin{array}{c} \text{Movies} \\ d \\ \boxed{V} \end{array} \end{array} \tag{2.4.1}$$

where U and V are rank d matrices. The ALS algorithm alternates between computing the least-squares solution for U and V while holding the other fixed. Both the quality of the approximation and the computational complexity depend on the magnitude of d : higher d produces higher accuracy while increasing computational cost. Collaborative filtering and the ALS algorithm are important tools in ML: an effective solution for ALS can be extended to a broad class of other applications.

While ALS may not seem like a graph algorithm, it can be represented elegantly using the GraphLab abstraction. The *sparse* matrix R defines a bipartite graph connecting each user with the movies they rated. The edge data contains the rating for a movie-user pair. The vertex data for users and movies contains the corresponding row in U and column in V respectively. The GraphLab update function recomputes the d length vector for each vertex by reading the d length vectors on adjacent vertices and then solving a least-squares regression problem to predict the edge values. Since the graph is bipartite and two colorable, and the edge consistency model is sufficient for serializability, the chromatic engine is used.

The Netflix task provides us with an opportunity to quantify the distributed chromatic engine overhead since we are able to directly control the computation-communication ratio by manipulating d : the dimen-

sionality of the approximating matrices in Eq. (2.4.1). In Fig. 2.19b we plot the speedup achieved for varying values of d and the corresponding number of cycles required per update. Extrapolating to obtain the theoretically optimal runtime, we estimated the overhead of Distributed GraphLab at 64 machines (512 CPUs) to be about 12x for $d = 5$ and about 4.9x for $d = 100$. Note that this overhead includes graph loading and communication. This provides us with a measurable objective for future optimizations.

Next, we compare against a Hadoop and an MPI implementation in Fig. 2.19c ($d = 20$ for all cases), using between 4 to 64 machines. The Hadoop implementation is part of the Mahout² project and is widely used. Since fault tolerance was not needed during our experiments, we reduced the Hadoop Distributed Filesystem's (HDFS) replication factor to one. A significant amount of our effort was then spent tuning the Hadoop job parameters to improve performance. However, even so we find that GraphLab performs between **40-60** times faster than Hadoop.

Performance Analysis: While some of the Hadoop inefficiency may be attributed to Java, job scheduling, and various design decisions, GraphLab also leads to a more efficient representation of the underlying algorithm. We can observe that the Map function of a Hadoop ALS implementation, performs no computation and its only purpose is to emit *copies* of the vertex data for every edge in the graph; unnecessarily multiplying the amount of data that need to be tracked.

For example, a user vertex that connects to 100 movies must emit the data on the user vertex 100 times, once for each movie. This results in the generation of a large amount of unnecessary network traffic and unnecessary HDFS writes. This weakness extends beyond the MapReduce abstraction, but also affects the graph message-passing models (such as Pregel) due to the lack of a *scatter* operation that would avoid sending same value multiple times to each machine. Comparatively, the GraphLab update function is simpler as users do not need to explicitly define the flow of information. Synchronization of a modified vertex only requires as much communication as there are ghosts of the vertex. In particular, only machines that require the vertex data for computation will receive it, and each machine receives each modified vertex data at most once, even if the vertex has many neighbors.

Our MPI implementation of ALS is highly optimized, and uses synchronous MPI collective operations for communication. The computation is broken into super-steps that alternate between recomputing the latent user and movies low rank matrices. Between super-steps the new user and movie values are scattered (using `MPI_Alltoall`) to the machines that need them in the next super-step. As a consequence our MPI implementation of ALS is roughly equivalent to an optimized Pregel version of ALS with added support for parallel broadcasts. Surprisingly, GraphLab was able to outperform the MPI implementation. We attribute the performance to the use of background asynchronous communication in GraphLab.

Finally, we can evaluate the effect of enabling dynamic computation. In Fig. 2.17, we plot the test error obtained over time using a dynamic update schedule as compared to a static BSP-style update schedule. This dynamic schedule is easily represented in GraphLab while it is difficult to express using Pregel messaging semantics. We observe that a dynamic schedule converges much faster, reaching a low test error in about half amount of work.

²<https://issues.apache.org/jira/browse/MAHOUT-542>

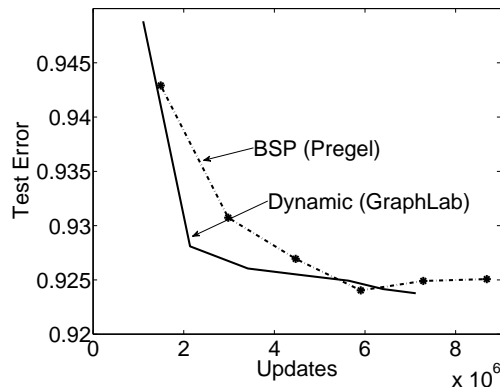


Figure 2.17: Convergence rate of ALS on Netflix data when dynamic computation is used. Dynamic computation can converge to equivalent test error in about half the number of updates.

2.4.4.2 Video Co-segmentation (CoSeg)

Video co-segmentation automatically identifies and clusters spatio-temporal segments of video (Fig. 2.18) that share similar texture and color characteristics. The resulting segmentation (Fig. 2.18) can be used in scene understanding and other computer vision and robotics applications. Previous co-segmentation methods [Batra et al. \[2010\]](#) have focused on processing frames in isolation. Instead, we developed a joint co-segmentation algorithm that processes all frames simultaneously and is able to model temporal stability.



Figure 2.18: (a) **Coseg**: a frame from the original video sequence and the result of running the co-segmentation algorithm. (b) **CoEM**: Top words for several types.

We preprocessed 1,740 frames of high-resolution video by coarsening each frame to a regular grid of 120×50 rectangular **super-pixels**. Each super-pixel stores the color and texture statistics for all the raw pixels in its domain. The CoSeg algorithm predicts the best label (e.g., sky, building, grass, pavement, trees) for each super pixel using **Gaussian Mixture Model (GMM)** in conjunction with **Loopy Belief Propagation (LBP)** [Gonzalez et al. \[2009a\]](#). The GMM estimates the best label given the color and texture statistics in the super-pixel. The algorithm operates by connecting neighboring pixels in time and space into a large three-dimensional grid and uses LBP to smooth the local estimates. We combined the two algorithms to form an Expectation-Maximization algorithm, alternating between LBP to compute the label for each super-pixel given the GMM and then updating the GMM given the labels from LBP.

The GraphLab update function executes the LBP local iterative update. We implement the dynamic update schedule described by [Elidan et al. \[2006\]](#), where updates that are expected to change vertex values

significantly are prioritized. We therefore make use of the locking engine with an approximate priority scheduler. The parameters for the GMM are maintained using the sync operation.

Performance Analysis: In Fig. 2.19a we demonstrate that the locking engine can achieve scalability and performance on the large 10.5 million vertex graph used by this application, resulting in a 10x speedup with 16x more machines.

We conclude that for the video co-segmentation task, Distributed GraphLab provides excellent performance while being the only distributed graph abstraction that allows the use of dynamic prioritized scheduling. In addition, the pipelining system is an effective way to hide latency, and to some extent, a poor partitioning.

2.4.4.3 Named Entity Recognition with CoEM

The Named Entity Recognition task using the CoEM algorithm extends from our shared memory evaluation of the same task in Sec. 2.3.3.3; but in the distributed setting.

We evaluated our Distributed GraphLab implementation against a Hadoop and an MPI implementation in Fig. 2.19d. In addition to the optimizations listed in Sec. 2.4.4.1, our Hadoop implementation required the use of binary marshaling methods to obtain reasonable performance (decreasing runtime by 5x from baseline).

We demonstrate that GraphLab implementation of CoEM was able to obtain a 20-30x speedup over Hadoop. The reason for the performance gap is the same as that for the Netflix evaluation. Since each vertex emits a copy of itself for each edge: in the extremely large CoEM graph, this corresponds to over 100 GB of HDFS writes occurring between the Map and Reduce stage.

Analysis: On the other hand, our MPI implementation was able to outperform Distributed GraphLab by a healthy margin. The CoEM task requires extremely little computation in comparison to the amount of data it touches. The extremely poor computation to communication ratio stresses our communication implementation, that is outperformed by MPI's more efficient communication layer.

We conclude that while Distributed GraphLab is suitable for the CoEM task providing an effective abstraction, further optimizations are needed to improve scalability and to bring performance closer to that of a dedicated MPI implementation.

2.4.4.4 Performance Summary

To summarize the results, the video co-segmentation problem scales very well demonstrating the effectiveness of the Locking Engine. Netflix has moderate scaling, which improves with increased dimensions in the matrix factorization. Netflix also outperforms a Hadoop implementation by **40-60x**, and also outperforms an MPI implementation by a small margin. The ability to outperform MPI can be attributed to the use of background asynchronous communication for synchronization. On the other hand, CoEM has poor scaling and this can be attributed to its light-weight update function but heavy vertex data, which results in it bringing to light several inefficiencies in our communication framework implementation. As

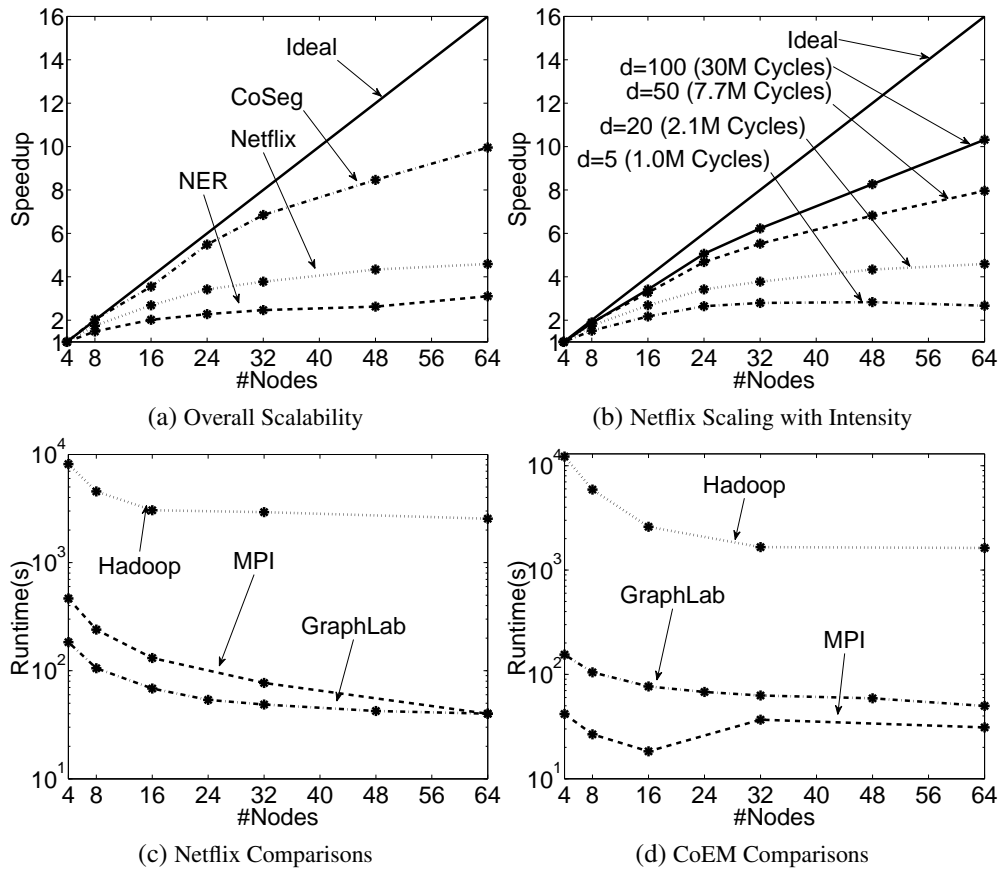


Figure 2.19: **(a)** Scalability of the three example applications with the default input size. CoSeg scales excellently due to very sparse graph and high computational intensity. Netflix with default input size scales moderately while CoEM is hindered by high network utilization. **(b)** Scalability of Netflix when the average number of cycles per update function call is increased. **(c)** Runtime of the Netflix experiment with GraphLab, Hadoop and MPI implementations. Note the logarithmic scale. GraphLab outperforms Hadoop by over 40-60x and is comparable to an MPI implementation. **(d)** Runtime of the CoEM experiment with Distributed GraphLab, Hadoop and MPI implementations. Note the logarithmic scale. GraphLab outperforms Hadoop by about 80x when the number of computing nodes is small, and about 30x when the number of nodes is large. The performance of Distributed GraphLab is comparable to the MPI implementation.

a result, even though we outperform Hadoop by between **30-80x**, we fail to beat an MPI implementation which uses highly optimized, though synchronous, communication.

2.5 Conclusion

The GraphLab 1 abstraction is easy to implement in the shared memory setting requiring little more than standard datastructures and parallel primitives. The distributed implementation however, requires a substantially larger effort since we must address issues of graph placement, data consistency and fault tolerance.

Our key contributions are:

- The GraphLab 1 abstraction comprising of:
 - A **data graph** which represents the data and computational dependencies.
 - **Update functions** which describe local computation
 - A **Sync mechanism** for aggregating global state.
 - A data **consistency model** which determines the extent to which computation can overlap.
- Shared Memory Implementation:
 - A shared memory implementation of the GraphLab 1 abstraction.
 - A collection of concurrent schedulers which allow for dynamic, prioritized computation.
 - Experimental evaluations of parameter learning and inference in graphical models, Gibbs sampling, CoEM, Lasso and compressed sensing on real-world problems.
- Distributed Implementation:
 - A **distributed data graph** format built around a two-stage partitioning scheme which allows for efficient load balancing and distributed ingress on variable-sized cluster deployments.
 - Two GraphLab 1 engines:
 - A **chromatic engine** that is partially synchronous and assumes the existence of a graph coloring, and
 - a **locking engine** that is fully asynchronous, supports general graph structures, and relies upon a graph-based pipelined locking system to hide network latency.
 - An asynchronous snapshot algorithm based on Chandy-Lamport snapshots that can be expressed using regular GraphLab 1 primitives.
 - Evaluation of the distributed implementation on three challenging Machine Learning tasks of CoEM, ALS, and Video Cosegmentation.

The resultant system is highly expressive and easy to use, and permits a rather broad range of algorithms to be developed, and parallelized, with relatively little effort. Danny Bickson has developed, and released as open source, a large number of different matrix factorization libraries implemented in GraphLab 1. Furthermore, the shared memory GraphLab 1 implementation has been instrumental in the development of some new graphical model inference procedures such as parallel Gibbs Samplers [Gonzalez et al., 2011], and kernelized belief propagation [Song et al., 2011].

Chapter 3

PowerGraph

Restricting the Abstraction for Performance

In the previous chapter, we defined the GraphLab 1 abstraction which defined the notion of graph computation and described both a shared memory and a distributed memory implementation of the abstraction. In this chapter, we begin by describing the lessons learned through the development of GraphLab 1 and explore the limitations of the model. We then design the PowerGraph abstraction [Gonzalez et al., 2012] which addresses the fundamental performance limitations of GraphLab 1, allowing our implementation to achieve the next order of magnitude gain in both performance and scalability.

We begin in Sec. 3.1 by enumerating the limitations of GraphLab 1 which block scalability, following which in Sec. 3.2 we describe the **Gather-Apply-Scatter** model which we can use to analyze both GraphLab 1 and Pregel in a common framework.

Next, in Sec. 3.3, we describe the PowerGraph abstraction which factorizes the GraphLab 1 update function into the **Gather**, **Apply** and **Scatter** phases. The new abstraction also permits the use of the vertex-separator model of distributed graph placement, bringing about large reductions in communication and storage requirements (Sec. 3.4). We also introduce streaming graph partitioning heuristics to reduce communication requirements, while ensuring load balancing. Finally, in Sec. 3.6 we describe a distributed implementation of the PowerGraph abstraction, and demonstrate that we are able to achieve large performance gains, achieving state of the art performance, and outperforming published results on some benchmarks.

3.1 GraphLab 1 Limitations

Through the design, implementation and development of a shared memory and distributed version of the GraphLab 1 abstraction in the previous chapter, we were made aware of some problems and limitations of the abstraction; and what is needed to scale GraphLab to even larger problems.

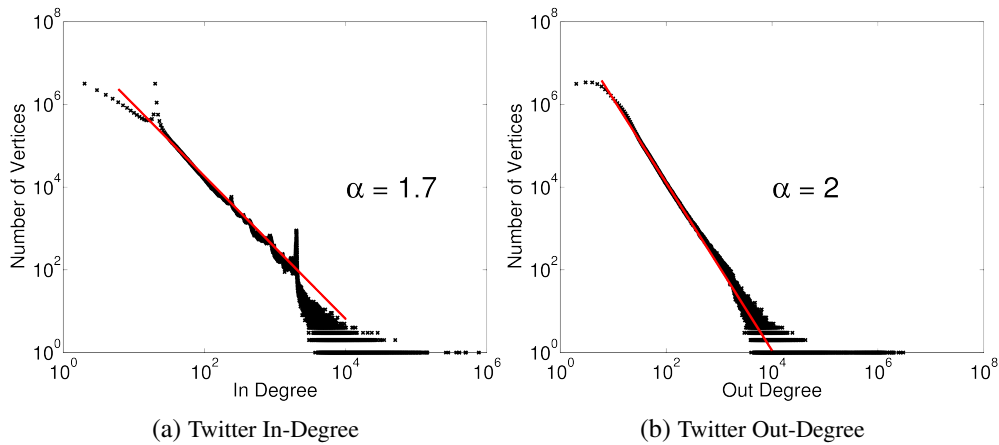


Figure 3.1: The in and out degree distributions of the Twitter follower network plotted in log-log scale.

3.1.1 Gather vs Scatter

GraphLab 1 has very efficient **scatter** capability, where a modification of a single vertex, can be efficiently transmitted to all machines which need it, with no unnecessary replication of data. Pregel however, requires excessive replication of data, requiring modifications of vertex data to be communicated $\#$ degree times (Sec. 2.4.4.1).

On the other hand, Pregel [Malewicz et al., 2010] has very efficient **gather** capability which allows information going to the same vertex to be “pre-combined”, reducing network communication. GraphLab 1 however, requires the update function to have full access to the entire scope and not just an aggregated value. As a result, even though an aggregated value might be sufficient, GraphLab 1 will perform excessive unnecessary communication.

GraphLab 1 will need fundamental changes to the abstraction to be able to efficiently express both **gather**-like operations and **scatter**-like operations.

3.1.2 Natural Graphs

One of the hallmark properties of natural graphs is their skewed power-law degree distribution [Faloutsos et al., 1999]: most vertices have relatively few neighbors while a few have many neighbors (e.g., celebrities in a social network). Under a power-law degree distribution the probability that a vertex has degree d is given by:

$$\mathbf{P}(d) \propto d^{-\alpha}, \quad (3.1.1)$$

where the exponent α is a positive constant that controls the “skewness” of the degree distribution. Higher α implies that the graph has lower density (ratio of edges to vertices), and that the vast majority of vertices are low degree. As α decreases, the graph density and number of high degree vertices increases. Most natural graphs typically have a power-law constant around $\alpha \approx 2$. For example, Faloutsos et al. [1999] estimated that the inter-domain graph of the Internet has a power-law constant $\alpha \approx 2.2$. One can visualize the skewed power-law degree distribution by plotting the number of vertices with a given degree in log-log scale. In Fig. 3.1, we plot the in and out degree distributions of the Twitter follower network demonstrating the characteristic linear power-law form.

Examining the behavior of the distributed GraphLab 1 implementation on natural graphs demonstrated several fundamental limitations of the GraphLab 1 abstraction.

Engine Performance The good performance of GraphLab 1’s locking engine on regular graphs in the earlier sections is tarnished by its poor performance on power-law graphs. We demonstrate this by generating a domain graph from the Yahoo! Web Graph [Yahoo, Retrieved 2011] producing a power law graph with 25.5M vertices and 355M, as well as a 26-connected $300 \times 300 \times 300$ grid graph with 27,000,000 vertices and 375 million edges (the same synthetic graph evaluated in Sec. 2.4.2.2). The performance results of running 2 iterations of loopy belief propagation on this graph is shown in Fig. 3.2. We observe from Fig. 3.2c that speedup is extremely poor, providing almost no speedup at all as number of machines is increased. To put runtime in perspective we observe that GraphLab 1 took nearly **500s per iteration** using 16 machines, while a regular graph with of the same size took only **10s per iteration**. In other words, a mere change in connectivity structure can result in **50x decrease** in performance even though the total amount of work is held constant.

Excessive Memory Load Loading the entire AltaVista Web Graph of 1.4B vertices and 6.6B edges into memory for PageRank computation on 64 machines required over 51GB of memory per machine for a total memory load of over 3 terabytes, far exceeding the real size of the graph. This is due to the poor partitioning scheme (random partitioning was used), which resulted in an excessive number of ghost vertices and replicated edges. Performance suffers as a result of excessive communication.

3.1.2.1 Challenges Of Natural Graphs

Natural graphs presents a challenge not just to GraphLab 1, but is also a fundamental limitation to all graph-parallel abstractions such as GraphLab 1 and Pregel. To minimize communication requirements and memory load, distributed graph systems rely on the availability of a good graph partitioning.

Where the graph is $G = (V, E)$, a graph bi-partitioning is a partition of the set of vertices V into approximately balanced two disjoint sets B and B' such that the number of edges crossing the partition is minimized. Formally:

$$\begin{aligned}
 B, B' &= \arg \min_{B, B'} |(i, j) \in E \text{ s.t. } i \in B, j \in B'| \\
 \text{subj. to} & \quad B \cup B' = V \text{ and } B \cap B' = \emptyset \text{ and } \frac{\max(|B|, |B'|)}{|V|/2} \leq \gamma
 \end{aligned}$$

where γ is a balance tolerance parameter. Observe that if $\gamma = 1$, the partitions must be exactly balanced, while $\gamma > 1$ permits some slackness. The objective can be analogously generalized to k -way partitions.

Mesh-like graph structures are relatively easily to partition and heuristic partitioners such as Metis or ParMetis [Karypis and Kumar, 1998] provide good results in very little runtime. Power law graphs however, are known empirically to have poor graph partitions [Lang, 2004, Leskovec et al., 2008b]. We have also empirically discovered that many heuristic graph partitioners (Metis [Karypis and Kumar, 1998],

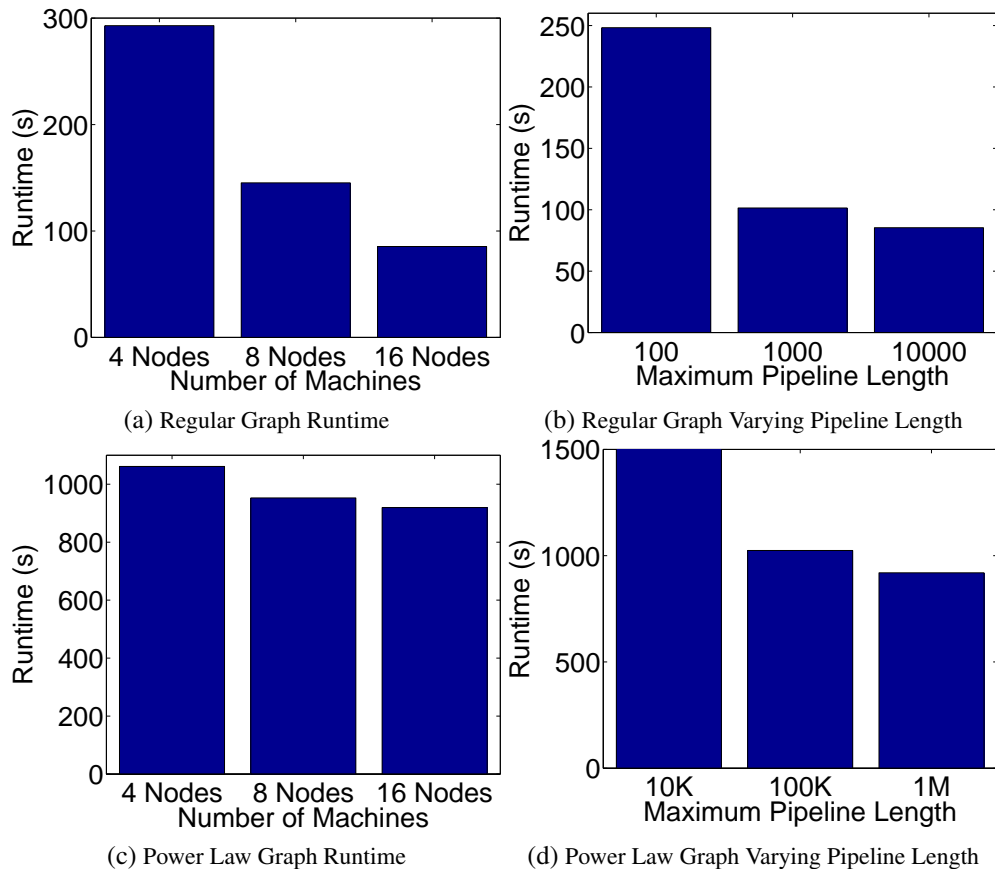


Figure 3.2: **(a)** Plots the runtime of the Distributed Locking Engine on 10 iterations of a loopy belief propagation problem on a regular graph varying the number of machines. Pipeline length was set to 10,000. Nearly linear speedup was observed. **(b)** Plots the runtime of the Distributed Locking Engine on the same synthetic problem on 16 machines (128 CPUs), varying the length of the pipeline. Increased pipeline length results in improved performance, but with diminishing returns. **(c)** Plots the runtime of the Distributed Locking Engine on 2 iterations of loopy belief propagation on a domain graph derived from the Yahoo Web Graph varying the number of machines. Pipeline length was set to 1M. Negligible Speedup is observed. **(d)** Plots the runtime of the Distributed Locking Engine on the same problem on 16 machines (128 CPUs), varying the length of the pipeline. Increased pipeline length results in slightly improved performance.

Scotch [Chevalier and Pellegrini, 2006] and Zoltan [Devine et al., 2006] as of 2010) do not behave well under Power Law graph structures; either crashing, or running out of memory¹.

Both GraphLab 1 and Pregel depend on graph partitioning to minimize communication and ensure work balance. However, in the case of natural graphs both are forced to resort to hash-based (random) partitioning which has extremely poor locality.

This has the following implications:

¹Most graph partitioning libraries use a coarsening procedure to collapse the size of the graph. However, in power law graphs, the coarsening procedure does not effectively decrease the number of edges in the graph [Abou-Rjeili and Karypis, 2006], resulting in high runtimes and memory consumption even on relatively small graphs. Coarsening heuristics that are more well-behaved exist for power-law graphs [Abou-Rjeili and Karypis, 2006], but we were unable to locate an implementation of such procedures as of our evaluation in 2010

Computation: While multiple vertex-programs may execute in parallel, existing graph-parallel abstractions do not parallelize *within* individual vertex-programs. A vertex program which has complexity linear in the number of edges will thus encounter scalability and performance issues which are exacerbated by poor partitioning quality.

Work Balance: The power-law degree distribution can lead to substantial work imbalance in graph parallel abstractions that treat vertices symmetrically. Since the storage, communication, and computation complexity of the vertex-program may be linear in the degree, the running time of vertex-programs can vary widely [Suri and Vassilvitskii, 2011].

Communication: The skewed degree distribution of natural-graphs leads to communication asymmetry and consequently bottlenecks. In addition, high-degree vertices can force messaging abstractions, such as Pregel, to generate and send many identical messages.

Storage: Since graph parallel abstractions must locally store the adjacency information for each vertex, each vertex requires memory linear in its degree. Consequently, high-degree vertices can exceed the memory capacity of a single machine.

3.1.3 Serializability Frequently Unnecessary

Serializability is a useful property to have for the reasons described in Sec. 1.1.1. While many Machine Learning algorithms require serializability for various theoretical guarantees to hold, empirically it has been observed for many algorithms, that serializability is not strictly necessary. In particular, serializability-breaking parallel/distributed implementations of various algorithms (LDA [Smola and Narayana-murthy, 2010], BP [Gonzalez et al., 2009a] and Matrix Factorization) have been empirically demonstrated to converge well.

Breaking serializability breaks theoretical guarantees of many algorithms, but provide large gains in throughput. This is most significant in the distributed setting since we avoid the cost of distributed locking. On the other hand, it has been demonstrated both empirically and theoretically that some algorithms have limits on non-serializable parallelism [Bradley et al., 2011, Recht et al., 2011, Siapas, 1996], either converging slower, or failing to converge when there is a large amount of parallelism. Therefore, it is still desirable to provide an abstraction with a broad set of consistency rules, allowing serializability to be broken when unnecessary, but providing it when required.

3.2 Characterization

While the implementation of ML vertex-programs in GraphLab 1 and Pregel differ in how they collect and disseminate information, they share a common overall structure. To characterize this common structure and differentiate between vertex and edge specific computation we introduce the GAS model of graph computation.

The **GAS** model represents three *conceptual* phases of a vertex-program: **G**ather, **A**pply, and **S**catter. In the **gather** phase, information about adjacent vertices and edges is collected through a generalized sum over the neighborhood of the vertex u on which $Q(u)$ is run:

$$\Sigma \leftarrow \bigoplus_{v \in \mathcal{N}[u]} g(D_u, D_{u-v}, D_v). \quad (3.2.1)$$

where D_u , D_v , and D_{u-v} are the values (program state and meta-data) for vertices u and v and edge (u, v) . The user defined sum \oplus operation must be commutative and associative and can range from a numerical sum to the union of the data on all neighboring vertices and edges.

The resulting value Σ is used in the **apply** phase to update the value of the central vertex:

$$D_u^{\text{new}} \leftarrow a(D_u, \Sigma). \quad (3.2.2)$$

Finally the **scatter** phase uses the new value of the central vertex to update the data on adjacent edges:

$$\forall v \in \mathbf{N}[u] : (D_{u-v}) \leftarrow s(D_u^{\text{new}}, D_{u-v}, D_v). \quad (3.2.3)$$

The fan-in and fan-out of a vertex-program is determined by the corresponding gather and scatter phases. For instance, in PageRank, the gather phase only operates on in-edges and the scatter phase only operates on out-edges. However, for many ML algorithms the graph edges encode ostensibly symmetric relationships, like friendship, in which both the gather and scatter phases touch all edges. In this case the fan-in and fan-out are equal. As we will show in Sec. 3.1.2.1, the ability for graph parallel abstractions to support both high fan-in and fan-out computation is critical for efficient computation on natural graphs.

GraphLab 1 and Pregel express GAS programs in very different ways. In the Pregel abstraction the gather phase is implemented using message combiners and the apply and scatter phases are expressed in the vertex program. Conversely, GraphLab 1 exposes the entire neighborhood to the vertex-program and allows the user to define the gather and apply phases within their program. The GraphLab 1 abstraction implicitly defines the communication aspects of the gather/scatter phases by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices. It is also important to note that GraphLab 1 does not differentiate between edge directions.

3.3 PowerGraph Abstraction

To address the challenges of computation on power-law graphs, we introduce PowerGraph [Gonzalez et al., 2012], a new graph-parallel abstraction that eliminates the degree dependence of the vertex-program by directly exploiting the GAS decomposition to factor vertex-programs over edges. By lifting the Gather and Scatter phases into the abstraction, PowerGraph is able to retain the natural “think-like-a-vertex” philosophy [Malewicz et al., 2010] while distributing the computation of a single vertex-program over the entire cluster.

PowerGraph combines the best features from both Pregel and GraphLab 1. From GraphLab 1, PowerGraph borrows the data-graph and shared-memory view of computation eliminating the need for users to architect the movement of information. From Pregel, PowerGraph borrows the commutative, associative gather concept. PowerGraph supports both the highly-parallel bulk-synchronous Pregel model of computation as well as the computationally efficient asynchronous GraphLab 1 model of computation.

Like GraphLab 1, the state of a PowerGraph program factors according to a **data-graph** with user defined vertex data D_v and edge data $D_{u \rightarrow v}$. Just like the GraphLab 1 abstraction, the PowerGraph abstraction does not place special meaning on the edge direction but treats it simply as a property of the edge. We will therefore frequently use D_{u-v} to refer to the contents of an edge $u - v$ when the direction of the edge is unimportant.


```

interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather( $D_u, D_{u-v}, D_v$ )  $\rightarrow$  Accum
  sum(Accum left, Accum right)  $\rightarrow$  Accum
  apply( $D_u, Accum$ )  $\rightarrow D_u^{new}$ 
  // Run on scatter_nbrs(u)
  scatter( $D_u^{new}, D_{u-v}, D_v$ )  $\rightarrow (D_{u-v}^{new}, Accum)$ 
}

```

Figure 3.3: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

Algorithm 3.1: Vertex-Program Execution Semantics

```

Input: Center vertex  $u$ 
if Cache Disabled then
  // Basic Gather-Apply-Scatter Model
  foreach neighbor  $v$  in gather_nbrs( $u$ ) do
    |  $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$ 
   $D_u \leftarrow \text{apply}(D_u, a_u)$ 
  foreach neighbor  $v$  scatter_nbrs( $u$ ) do
    |  $(D_{u-v}) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$ 
else if Cache Enabled then
  // Faster GAS Model with Delta Caching
  if cached accumulator  $a_u$  is empty then
    | foreach neighbor  $v$  in gather_nbrs( $u$ ) do
      | |  $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{u-v}, D_v))$ 
   $D_u \leftarrow \text{apply}(D_u, a_u)$ 
  foreach neighbor  $v$  scatter_nbrs( $u$ ) do
    |  $(D_{u-v}, \Delta a) \leftarrow \text{scatter}(D_u, D_{u-v}, D_v)$ 
    | if  $a_v$  and  $\Delta a$  are not Empty then  $a_v \leftarrow \text{sum}(a_v, \Delta a)$ 
    | else  $a_v \leftarrow$  Empty

```

The data stored in the data-graph includes both meta-data (e.g., urls and edge weights) as well as computation state (e.g., the PageRank of vertices). In Sec. 3.4 we introduce vertex-cuts which allow PowerGraph to efficiently represent and store power-law graphs in a distributed environment. We now describe the PowerGraph *abstraction* and how it can be used to naturally decompose vertex-programs. Then in Sec. 3.4 through Sec. 3.6 we discuss how to implement the PowerGraph abstraction in a distributed environment.

3.3.1 GAS Vertex-Programs

Computation in the PowerGraph abstraction is encoded as a state-less **vertex-program** which implements the `GASVertexProgram` interface (Fig. 3.3) and therefore explicitly factors into the gather, sum, apply, and scatter functions. Each function is invoked in stages by the PowerGraph engine following the semantics in Alg. 3.1. By factoring the vertex-program, the PowerGraph execution engine can distribute a single vertex-program over multiple machines and move computation to the data.

<pre> // gather_nbrs: IN_NBRS gather(D_u, D_{u-v}, D_v): return D_v.rank/outdeg(v) sum(a, b): return a + b apply(D_u, acc): rnew = 0.15 + 0.85 * acc D_u.delta = rnew - D_u.rank D_u.rank = rnew // scatter_nbrs: OUT_NBRS scatter(D_u, D_{u-v}, D_v): if(D_u.delta >epsilon) Activate(v) </pre>	<pre> // gather_nbrs: ALL_NBRS gather(D_u, D_{u-v}, D_v): return set(D_v) sum(a, b): return union(a, b) apply(D_u, S): D_u = min c where c not in S // scatter_nbrs: ALL_NBRS scatter(D_u, D_{u-v}, D_v): // Nbr changed since gather if(D_u == D_v) Activate(v) </pre>	<pre> // gather_nbrs: ALL_NBRS gather(D_u, D_{u-v}, D_v): return D_v + D_{v-u} sum(a, b): return min(a, b) apply(D_u, new_dist): D_u = new_dist // scatter_nbrs: ALL_NBRS scatter(D_u, D_{u-v}, D_v): // If changed activate nbr if(changed(D_u)) Activate(v) </pre>
--	---	--

(a) PageRank

(b) Greedy Graph Coloring

(c) Single Source Shortest Path (SSSP)

Figure 3.4: The PageRank, graph-coloring, and single source shortest path algorithms implemented in the basic PowerGraph abstraction without Delta Caching.

During the gather phase the `gather` and `sum` functions are used as a *map* and *reduce* to collect information about the neighborhood of the vertex. The `gather` function is invoked in parallel on the edges adjacent to u . The particular set of edges is determined by `gather_nbrs` which can be `none`, `in`, `out`, or `all`. The gather function is passed the data on the adjacent vertex and edge and returns a temporary accumulator (a user defined type). The result is combined using the commutative and associative `sum` operation. The final result a_u of the gather phase is passed to the `apply` phase and cached by PowerGraph.

After the gather phase has completed, the `apply` function takes the final accumulator and computes a new vertex value D_u which is atomically written back to the graph. The size of the accumulator a_u and complexity of the `apply` function play a central role in determining the network and storage efficiency of the PowerGraph abstraction and should be *sub-linear* and ideally constant in the degree.

During the scatter phase, the `scatter` function is invoked in parallel on the edges adjacent to u producing new edge values D_{u-v} which are written back to the data-graph. As with the gather phase, the `scatter_nbrs` determines the particular set of edges on which `scatter` is invoked. The scatter function returns an optional value Δa which is used to dynamically update the cached accumulator a_v for the adjacent vertex (see Sec. 3.3.2).

In Fig. 3.4 we implement the PageRank, greedy graph coloring, and single source shortest path algorithms using the PowerGraph abstraction. In PageRank the `gather` and `sum` functions collect the total value of the adjacent vertices, the `apply` function computes the new PageRank, and the `scatter` function is used to activate adjacent vertex-programs if necessary. In graph coloring the `gather` and `sum` functions collect the set of colors on adjacent vertices, the `apply` function computes a new color, and the `scatter` function activates adjacent vertices if they violate the coloring constraint. Finally in single source shortest path (SSSP), the `gather` and `sum` functions compute the shortest path through each of the neighbors, the `apply` function returns the new distance, and the `scatter` function activates affected neighbors.

```

// gather_nbrs: IN_NBRs
gather( $D_u, D_{u-v}, D_v$ ):
    return  $D_v$ .rank / outdeg( $v$ )
sum( $a, b$ ): return  $a + b$ 
apply( $D_u, acc$ ):
    rnew = 0.15 + 0.85 * acc
     $D_u$ .delta = rnew -  $D_u$ .rank
     $D_u$ .rank = rnew
// scatter_nbrs: OUT_NBRs
scatter( $D_u, D_{u-v}, D_v$ ):
    if ( $|D_u$ .delta| >  $\epsilon$ )
        Activate( $v$ )
    return  $D_u$ .delta / outdeg( $u$ )

```

(a) PageRank With Delta Caching

```

// gather_nbrs: ALL_NBRs
gather( $D_u, D_{u-v}, D_v$ ):
    return  $D_v + D_{v-u}$ 
sum( $a, b$ ):
    return min( $a, b$ )
apply( $D_u, new\_dist$ ):
     $D_u$  = new_dist
// scatter_nbrs: ALL_NBRs
scatter( $D_u, D_{u-v}, D_v$ ):
    // If changed activate nbr
    if (changed( $D_u$ ))
        Activate( $v$ )
    return  $D_u + D_{u-v}$ 

```

(b) SSSP with Delta Caching

Figure 3.5: The PageRank, and single source shortest path algorithms implemented in the PowerGraph abstraction and taking advantage of Delta Caching.

3.3.2 Delta Caching

In many cases a vertex-program will be triggered in response to a change in a *few* of its neighbors. The gather operation is then repeatedly invoked on *all* neighbors, many of which remain unchanged, thereby wasting computation cycles. For many algorithms [Ahmed et al., 2012] it is possible to dynamically maintain the result of the gather phase a_u and skip the gather on subsequent iterations.

The PowerGraph engine maintains a cache of the accumulator a_u from the previous gather phase for each vertex. The scatter function can *optionally* return an additional Δa which is atomically added to the cached accumulator a_v of the neighboring vertex v using the sum function. If Δa is not returned, then the neighbor’s cached a_v is cleared, forcing a complete gather on the subsequent execution of the vertex-program on the vertex v . When executing the vertex-program on v the PowerGraph engine uses the cached a_v if available, bypassing the gather phase.

Intuitively, Δa acts as an additive correction on-top of the previous gather for that edge. More formally, if the accumulator type forms an **abelian group**: has a commutative and associative sum (+) and an *inverse* (−) operation, then we can define

$$\Delta a = \text{gather}(D_u, D_{u-v}^{\text{new}}, D_v^{\text{new}}) - \text{gather}(D_u, D_{u-v}, D_v). \quad (3.3.1)$$

We demonstrate Delta Caching by modify the PageRank and the SSSP algorithms in Fig. 3.5. In the PageRank example (Fig. 3.4) we take advantage of the abelian nature of the PageRank sum operation. In the SSSP example, while the *min* operation is not abelian, the nature of the computation ensures that the distance value on each vertex is always decreasing, thus allowing the delta cache to be updated correctly.

3.3.3 Scheduling

PowerGraph continues to inherit the dynamic scheduling capability of GraphLab 1. The PowerGraph engine maintains a set of active vertices on which to eventually execute the vertex-program. The user

initiates computation by calling `Activate(v)` or `Activate_all()`. The PowerGraph engine then proceeds to execute the vertex-program on the active vertices until none remain. Once a vertex-program completes the scatter phase it becomes inactive until it is reactivated.

Vertices can activate themselves and neighboring vertices. Each function in a vertex-program can only activate vertices visible in the arguments to that function. For example the scatter function invoked on the edge (u, v) can only activate the vertices u and v . This restriction is essential to ensure that activation events are generated on machines on which they can be efficiently processed.

The order in which activated vertices are executed is up to the PowerGraph execution engine. The only guarantee is that all activated vertices are eventually executed. This flexibility in scheduling enables PowerGraph programs to be executed both *synchronously* and *asynchronously*, leading to different trade-offs in algorithm performance, system performance, and determinism.

3.3.3.1 Bulk Synchronous Execution

When run synchronously, the PowerGraph engine executes the gather, apply, and scatter phases in order. Each phase, called a **minor-step**, is run synchronously on all active vertices with a barrier at the end. We define a **super-step** as a complete series of GAS minor-steps. Changes made to the vertex data and edge data are committed at the end of each minor-step and are visible in the subsequent minor-step. Vertices activated in each super-step are executed in the subsequent super-step.

The synchronous execution model ensures a deterministic execution regardless of the number of machines and closely resembles Pregel. However, the frequent barriers and inability to operate on the most recent data can lead to an inefficient distributed execution and slow algorithm convergence. To address these limitations PowerGraph also supports asynchronous execution.

3.3.3.2 Asynchronous Execution

When run asynchronously, the PowerGraph engine executes active vertices as processor and network resources become available. Changes made to the vertex and edge data during the apply and scatter functions are immediately committed to the graph and visible to subsequent computation on neighboring vertices.

By using processor and network resources as they become available and making any changes to the data-graph immediately visible to future computation, an asynchronous execution can more effectively utilize resources and accelerate the convergence of the underlying algorithm. For example, the greedy graph-coloring algorithm in Fig. 3.4 will not converge when executed synchronously but converges quickly when executed asynchronously. The merits of asynchronous computation have been studied extensively in the context of numerical algorithms [Bertsekas and Tsitsiklis, 1989]. In Gonzalez et al. [2009a,b], Low et al. [2012] we demonstrated that asynchronous computation can lead to both theoretical and empirical gains in algorithm and system performance.

Unfortunately, the behavior of the asynchronous execution depends on the number machines and availability of network resources leading to non-determinism that can complicate algorithm design and debugging. Furthermore, for some algorithms, like statistical simulation, the resulting non-determinism, if not carefully controlled, can lead to instability or even divergence [Gonzalez et al., 2011].

To address these challenges, GraphLab 1 automatically enforces **serializability**: every parallel execution of vertex-programs has a corresponding sequential execution. In Chap. 2 we showed that serializability is sufficient to support a wide range of ML algorithms. To achieve serializability, GraphLab 1 prevents adjacent vertex-programs from running concurrently using a fine-grained locking protocol which requires *sequentially* grabbing locks on all neighboring vertices. Furthermore, the locking scheme used by GraphLab 1 is unfair to high degree vertices.

PowerGraph retains the strong serializability guarantees of GraphLab 1 while addressing its limitations. We address the problem of sequential locking by introducing a new *parallel* locking protocol (described in Sec. 3.6.4) which is fair to high degree vertices. In addition, the PowerGraph abstraction exposes substantially more fine grained (edge-level) parallelism allowing the entire cluster to support the execution of individual vertex programs.

3.3.4 Comparison with GraphLab 1 and Pregel

Surprisingly, despite the strong constraints imposed by the PowerGraph abstraction, it is *possible* to emulate both GraphLab 1 and Pregel vertex-programs in PowerGraph. To emulate a GraphLab 1 vertex-program, we use the gather and sum functions to *concatenate* all the data on adjacent vertices and edges and then run the GraphLab 1 program within the apply function.

Similarly, to express a Pregel vertex-program, we use the gather and sum functions to combine the inbound messages (stored as edge data) and *concatenate* the list of neighbors needed to compute the outbound messages. The Pregel vertex-program then runs within the apply function generating the set of messages which are passed as vertex data to the scatter function where they are written back to the edges.

In order to address the challenges of natural graphs, the PowerGraph abstraction requires the size of the accumulator and the complexity of the apply function to be sub-linear in the degree. However, directly executing GraphLab 1 and Pregel vertex-programs within the apply function leads the size of the accumulator and the complexity of the apply function to be *linear* in the degree eliminating many of the benefits on natural graphs.

3.4 Distributed Graph Placement

The PowerGraph abstraction relies on the distributed data-graph to store the computation state and encode the interaction between vertex-programs. The placement of the data-graph structure and data plays a central role in minimizing communication and ensuring work balance.

A common approach to placing a graph on a cluster of p machines is to construct a balanced p -way **edge-cut** (e.g., Fig. 3.6a) in which vertices are evenly assigned to machines and the number of edges spanning machines is minimized. Unfortunately, the tools [Chevalier and Pellegrini, 2006, Karypis and Kumar, 1998] for constructing balanced edge-cuts perform poorly [Abou-Rjeili and Karypis, 2006, Lang, 2004, Leskovec et al., 2008a] on power-law graphs. When the graph is difficult to partition, both GraphLab 1 and Pregel resort to hashed (random) vertex placement. While fast and easy to implement, hashed vertex placement cuts most of the edges:

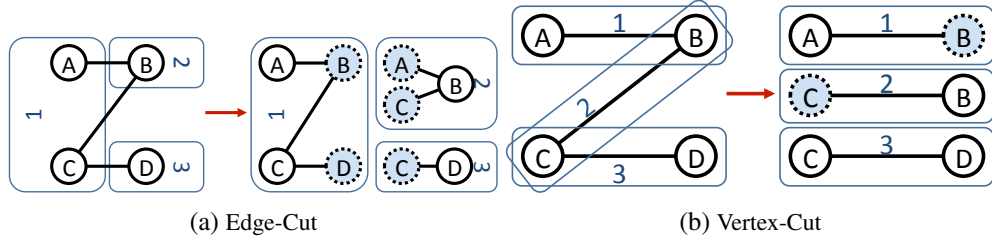


Figure 3.6: (a) An edge-cut and (b) vertex-cut of a graph into three parts. Shaded vertices are ghosts and mirrors respectively.

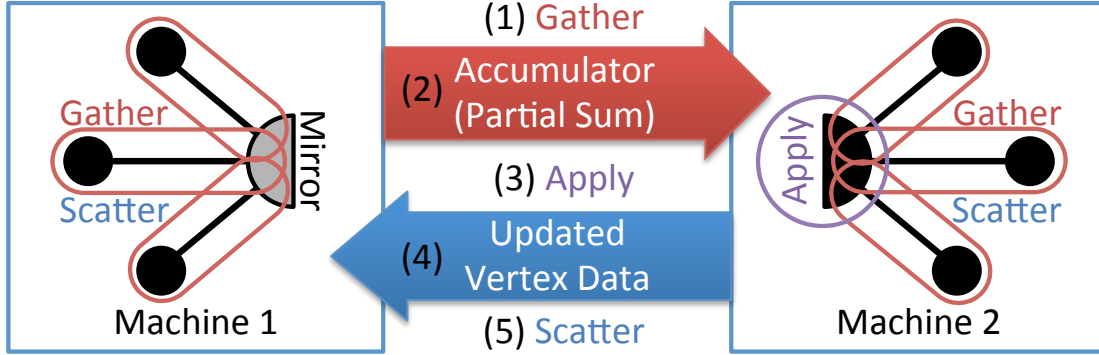


Figure 3.7: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulators is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.

Theorem 3.4.1. *If vertices are randomly assigned to p machines then the expected fraction of edges cut is:*

$$\mathbf{E} \left[\frac{|Edges\ Cut|}{|E|} \right] = 1 - \frac{1}{p}. \quad (3.4.1)$$

For a power-law graph with exponent α , the expected number of edges cut per-vertex is:

$$\mathbf{E} \left[\frac{|Edges\ Cut|}{|V|} \right] = \left(1 - \frac{1}{p} \right) \mathbf{E} [D[v]] = \left(1 - \frac{1}{p} \right) \frac{\mathbf{h}_{|V|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha)}, \quad (3.4.2)$$

where the $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.

Proof. An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. \square

Every cut edge contributes to storage and network overhead since both machines maintain a copy of the adjacency information and in some cases [Gregor and Lumsdaine, 2005], a **ghost** (local copy) of the vertex and edge data. For example in Fig. 3.6a we construct a three-way edge-cut of a four vertex graph resulting in five ghost vertices and all edge data being replicated. Any changes to vertex and edge data associated with a cut edge must be synchronized across the network. For example, using just two machines, a random cut will cut roughly *half* the edges, requiring $|E|/2$ communication.

3.4.1 Balanced p -way Vertex-Cut

By factoring the vertex program along the edges in the graph, The PowerGraph abstraction allows a single vertex-program to span multiple machines. In Fig. 3.7 a single high degree vertex program has been split across two machines with the gather and scatter functions running in parallel on each machine and accumulator and vertex data being exchanged across the network.

Because the PowerGraph abstraction allows a single vertex-program to span multiple machines, we can improve work balance and reduce communication and storage overhead by evenly *assigning edges* to machines and allowing *vertices to span machines*. Each machine only stores the edge information for the edges assigned to that machine, evenly distributing the massive amounts of edge data. Since each edge is stored exactly once, changes to edge data do not need to be communicated. However, changes to vertex must be copied to all the machines it spans, thus the storage and network overhead depend on the number of machines spanned by each vertex.

We minimize storage and network overhead by limiting the number of machines spanned by each vertex. A balanced p -way **vertex-cut** formalizes this objective by assigning each *edge* $e \in E$ to a machine $A(e) \in \{1, \dots, p\}$. Each vertex then spans the set of machines $A(v) \subseteq \{1, \dots, p\}$ that contain its adjacent edges. We define the balanced vertex-cut objective:

$$\min_A \frac{1}{|V|} \sum_{v \in V} |A(v)| \quad (3.4.3)$$

$$\text{s.t.} \quad \max_m |\{e \in E \mid A(e) = m\}|, < \lambda \frac{|E|}{p} \quad (3.4.4)$$

where the imbalance factor $\lambda \geq 1$ is a small constant. We use the term **replicas** of a vertex v to denote the $|A(v)|$ copies of the vertex v : each machine in $A(v)$ has a replica of v . Because changes to vertex data are communicated to all replicas, the communication overhead is also given by $|A(v)|$. The objective (Eq. (3.4.3)) therefore minimizes the average number of replicas in the graph and as a consequence the total storage and communication requirements of the PowerGraph engine.

For each vertex v with multiple replicas, one of the replicas is *randomly* nominated as the **master** which maintains the master version of the vertex data. All remaining replicas of v are then **mirrors** and maintain a local cached *read only* copy of the vertex data. (e.g., Fig. 3.6b). For instance, in Fig. 3.6b we construct a three-way vertex-cut of a graph yielding only 2 mirrors. Any changes to the vertex data (e.g., the Apply function) must be made to the master which is then immediately replicated to all mirrors.

Vertex-cuts address the major issues associated with edge-cuts in power-law graphs. Percolation theory [Albert et al., 2000] suggests that power-law graphs have good vertex-cuts. Intuitively, by cutting a small fraction of the very high degree vertices we can quickly shatter a graph. Furthermore, because the balance constraint (Eq. (3.4.4)) ensures that edges are uniformly distributed over machines, we naturally achieve improved work balance even in the presence of very high-degree vertices.

The vertex-cut approach to distributed graph placement is related to work by Catalyurek and Aykanat [1996], Devine et al. [2006] in hypergraph partitioning. In particular, a vertex-cut problem can be cast as a hypergraph-cut problem by converting each edge to a vertex, and each vertex to a hyper-edge. However, since existing hypergraph partitioning can be too time intensive to use effectively on extremely large graphs, we instead consider the streaming setting where each edge of the graph is accessed exactly once. The streaming graph partitioning setting has been explored in Stanton and Kliot [2011] for the streaming edge-cut problem but did not consider the vertex-cut problem.

The simplest method to construct a vertex cut is to randomly assign edges to machines. Random (hashed) edge placement is fully data-parallel, achieves nearly perfect balance on large graphs, and can be applied in the streaming setting. In the following theorem, we relate the expected normalized replication factor (Eq. (3.4.3)) to the number of machines and the power-law constant α .

Theorem 3.4.2 (Randomized Vertex Cuts). *A random vertex-cut on p machines has an expected replication:*

$$\mathbf{E} \left[\frac{1}{|V|} \sum_{v \in V} |A(v)| \right] = \frac{p}{|V|} \sum_{v \in V} \left(1 - \left(1 - \frac{1}{p} \right)^{\mathbf{D}[v]} \right). \quad (3.4.5)$$

where $\mathbf{D}[v]$ denotes the degree of vertex v . For a power-law graph the expected replication (Fig. 3.8a) is determined entirely by the power-law constant α :

$$\mathbf{E} \left[\frac{1}{|V|} \sum_{v \in V} |A(v)| \right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)} \sum_{d=1}^{|V|-1} \left(\frac{p-1}{p} \right)^d d^{-\alpha}, \quad (3.4.6)$$

where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution.

Proof. By linearity of expectation:

$$\mathbf{E} \left[\frac{1}{|V|} \sum_{v \in V} |A(v)| \right] = \frac{1}{|V|} \sum_{v \in V} \mathbf{E} [|A(v)|], \quad (3.4.7)$$

The expected replication $\mathbf{E} [|A(v)|]$ of a single vertex v can be computed by considering the process of randomly assigning the $\mathbf{D}[v]$ edges adjacent to v . Let the indicator X_i denote the event that vertex v has at least one of its edges on machine i . The expectation $\mathbf{E} [X_i]$ is then:

$$\mathbf{E} [X_i] = 1 - \mathbf{P}(v \text{ has no edges on machine } i) \quad (3.4.8)$$

$$= 1 - \left(1 - \frac{1}{p} \right)^{\mathbf{D}[v]}, \quad (3.4.9)$$

The expected replication factor for vertex v is then:

$$\mathbf{E} [|A(v)|] = \sum_{i=1}^p \mathbf{E} [X_i] = p \left(1 - \left(1 - \frac{1}{p} \right)^{\mathbf{D}[v]} \right). \quad (3.4.10)$$

Treating $\mathbf{D}[v]$ as a Zipf random variable:

$$\mathbf{E} \left[\frac{1}{|V|} \sum_{v \in V} |A(v)| \right] = \frac{p}{|V|} \sum_{v \in V} \left(1 - \mathbf{E} \left[\left(\frac{p-1}{p} \right)^{\mathbf{D}[v]} \right] \right), \quad (3.4.11)$$

and taking the expectation under $\mathbf{P}(d) = d^{-\alpha} / \mathbf{h}_{|V|}(\alpha)$:

$$\mathbf{E} \left[\left(1 - \frac{1}{p} \right)^{\mathbf{D}[v]} \right] = \frac{1}{\mathbf{h}_{|V|}(\alpha)} \sum_{d=1}^{|V|-1} \left(1 - \frac{1}{p} \right)^d d^{-\alpha}. \quad (3.4.12)$$

□

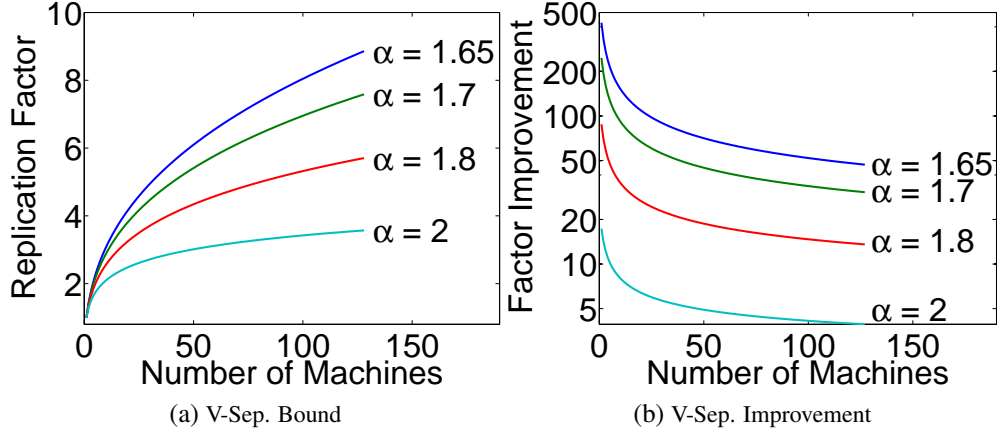


Figure 3.8: **(a)** Expected replication factor for different power-law constants. **(b)** The ratio of the expected communication and storage cost of random edge cuts to random vertex cuts as a function of the number machines. This graph assumes that edge data and vertex data are the same size.

While lower α values (more high-degree vertices) imply a higher replication factor (Fig. 3.8a) the effective gains of vertex-cuts relative to edge cuts (Fig. 3.8b) actually *increase* with lower α . In Fig. 3.8b we plot the ratio of the expected costs (comm. and storage) of random edge-cuts (Eq. (3.4.2)) to the expected costs of random vertex-cuts (Eq. (3.4.6)) demonstrating order of magnitude gains.

Finally, the vertex cut model is also highly effective for regular graphs since in the event that a good edge-cut can be found it can be converted to a better vertex cut:

Theorem 3.4.3. *For a given an edge-cut with g ghosts, **any** vertex cut along the same partition boundary has strictly fewer than g mirrors.*

Proof of Theorem 3.4.3. Consider the two-way edge cut which cuts the set of edges $E' \in E$ and let V' be the set of vertices in E' . The total number of ghosts induced by this edge partition is therefore $|V'|$. If we then select and delete arbitrary vertices from V' along with their adjacent edges until no edges remain, then the set of deleted vertices corresponds to a vertex-cut in the original graph. Since at most $|V'| - 1$ vertices may be deleted, there can be at most $|V'| - 1$ mirrors. This can be generalized easily to k -way cuts. \square

3.4.2 Greedy Vertex-Cuts

We can improve upon the randomly constructed vertex-cut by de-randomizing the edge-placement process. The resulting algorithm is a sequential greedy heuristic which places the next edge on the machine that minimizes the conditional expected replication factor. To construct the de-randomization we consider the task of placing the $i + 1$ edge after having placed the previous i edges. Using the conditional expectation we define the objective:

$$\arg \min_k \mathbf{E} \left[\sum_{v \in V} |A(v)| \mid A_i, A(e_{i+1}) = k \right], \quad (3.4.13)$$

where A_i is the assignment for the previous i edges. Using Theorem 3.4.2 to evaluate Eq. (3.4.13) we obtain the following edge placement rules for the edge (u, v) :

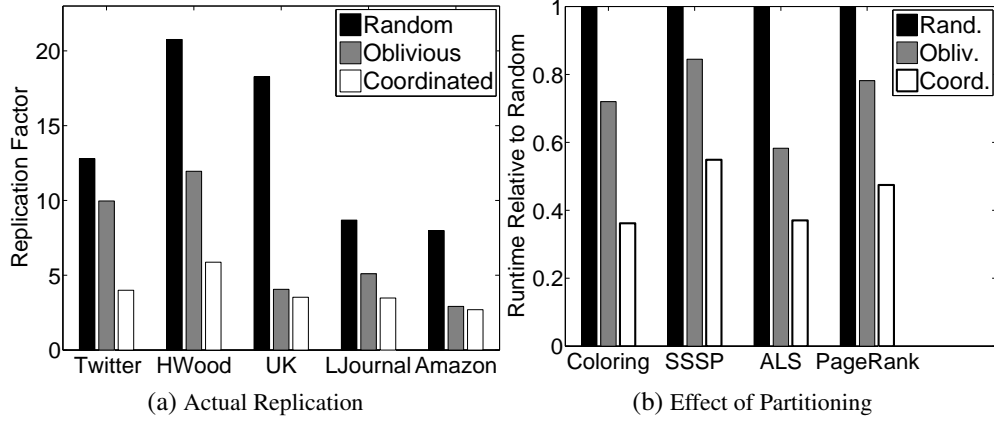


Figure 3.9: (a) The actual replication factor on 32 machines. (b) The effect of partitioning on runtime.

Graph	$ V $	$ E $	α	# Edges
Altavista [Yahoo, Retrieved 2011]	1.4B	6.6B	1.8	641,383,778
Twitter [Kwak et al., 2010]	42M	1.5B	1.9	245,040,680
UK [Bordino et al., 2008]	132.8M	5.5B	2.0	102,838,432
Amazon [Boldi and Vigna, 2004, Boldi et al., 2011]	0.7M	5.2M	2.1	57,134,471
LiveJournal [Chierichetti et al., 2009]	5.4M	79M	2.2	35,001,696
Hollywood [Boldi and Vigna, 2004, Boldi et al., 2011]	2.2M	229M		

(a) Real world graphs

(b) Synthetic Graphs

Table 3.1: (a) A collection of Real world graphs. (b) Randomly constructed ten-million vertex power-law graphs with varying α . Smaller α produces denser graphs.

Case 1: If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

Case 2: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.

Because the greedy-heuristic is a de-randomization it is guaranteed to obtain an expected replication factor that is no worse than random placement and in practice can be much better. Unlike the randomized algorithm, which is embarrassingly parallel and easily distributed, the greedy algorithm requires coordination between machines. We consider two distributed implementations:

Coordinated: maintains the values of $A_i(v)$ in a distributed table. Then each machine runs the greedy heuristic and periodically updates the distributed table. Local caching is used to reduce communication at the expense of accuracy in the estimate of $A_i(v)$.

Oblivious: runs the greedy heuristic independently on each machine. Each machine maintains its own estimate of A_i with no additional communication.

In Fig. 3.10a, we compare the replication factor of both heuristics against random vertex cuts on the Twitter follower network. We plot the replication factor as a function of the number of machines (EC2 instances described in Sec. 3.6) and find that random vertex cuts match the predicted replication given in

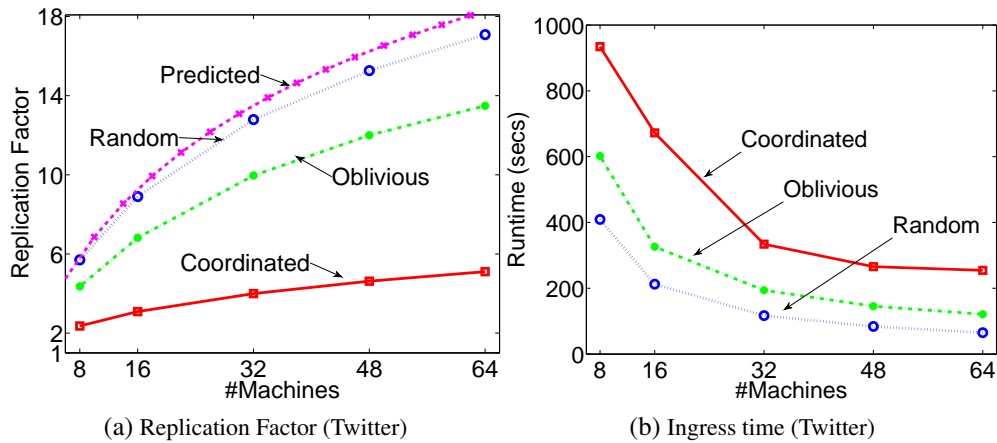


Figure 3.10: (a,b) Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

Theorem 3.4.2. Furthermore, the greedy heuristics substantially improve upon random placement with an order of magnitude reduction in the replication factor, and therefore communication and storage costs. For a fixed number of machines ($p = 32$), we evaluated (Fig. 3.9a) the replication factor of the two heuristics on five real-world graphs (Table 3.1a). In all cases the greedy heuristics out-perform random placement, while doubling the load time (Fig. 3.10b). The Oblivious heuristic achieves compromise by obtaining a relatively low replication factor while only slightly increasing runtime.

3.5 Abstraction Comparison

In this section, we experimentally characterize the dependence on α and the relationship between fan-in and fan-out by using the Pregel, GraphLab 1, and PowerGraph abstractions to run PageRank on five synthetically constructed power-law graphs. Each graph has ten-million vertices and an α ranging from 1.8 to 2.2. The graphs were constructed by randomly sampling the out-degree of each vertex from a Zipf distribution and then adding out-edges such that the in-degree of each vertex is nearly identical. We then inverted each graph to obtain the corresponding power-law fan-in graph. The density of each power-law graph is determined by α and therefore each graph has a different number of edges (see Table 3.1b).

We used the GraphLab 1 C++ implementation from Chap. 2 and added instrumentation to track network usage. As of the writing of the PowerGraph paper [Gonzalez et al., 2012], public implementations of Pregel (e.g., Giraph) were unable to handle even our smaller synthetic problems due to memory limitations. Consequently, we used Piccolo [Power and Li, 2010] as a proxy implementation of Pregel since Piccolo naturally expresses the Pregel abstraction and provides an efficient C++ implementation with dynamic load-balancing. Finally, we used our implementation of PowerGraph described in Sec. 3.6.

All experiments in this section are evaluated on an eight node Linux cluster. Each node consists of two quad-core Intel Xeon E5620 processors with 32 GB of RAM and is connected via 1-GigE Ethernet. All systems were compiled with GCC 4.4. GraphLab 1 and Piccolo used random edge-cuts while PowerGraph used random vertex-cuts.. Results are averaged over 20 iterations.

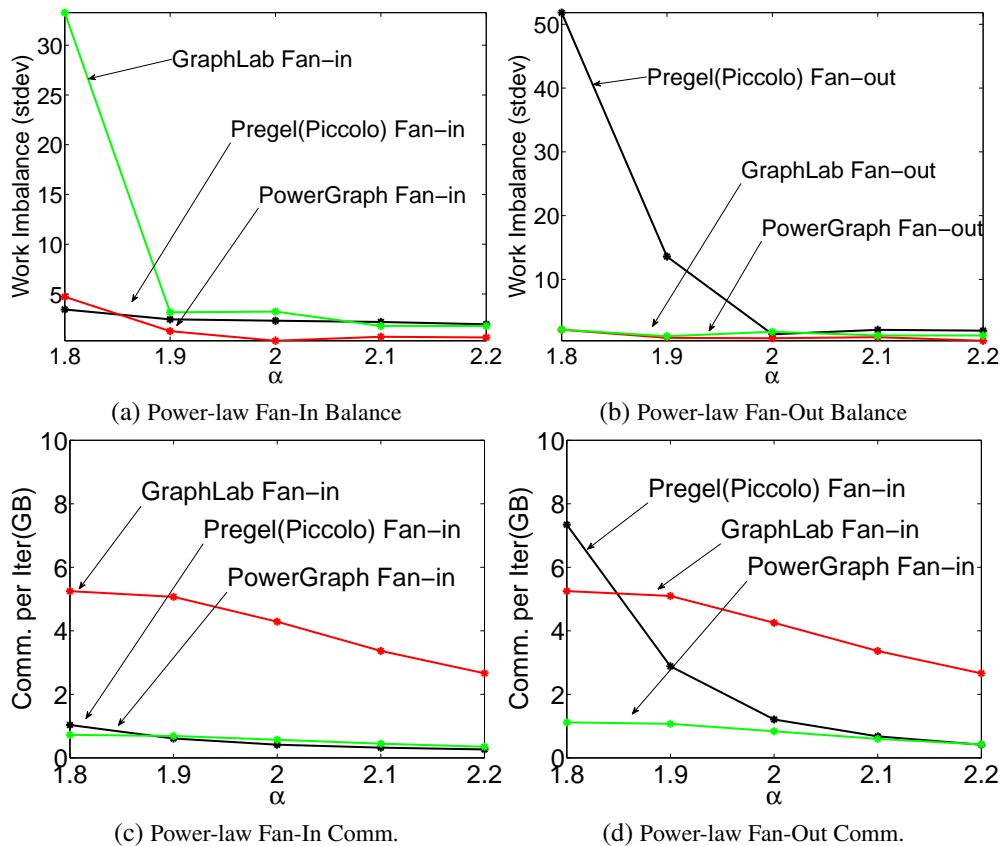


Figure 3.11: **Synthetic Experiments: Work Imbalance and Communication.** (a, b) Standard deviation of worker computation time across 8 distributed workers for each abstraction on power-law fan-in and fan-out graphs. (b, c) Bytes communicated per iteration for each abstraction on power-law fan-in and fan-out graphs.

3.5.1 Computation Imbalance

The *sequential* component of the PageRank vertex-program is proportional to the out-degree in the Pregel abstraction and the in-degree in the GraphLab 1 abstraction. PowerGraph eliminates this sequential dependence by distributing the computation of *individual* vertex-programs over multiple machines. Therefore we expect highly-skewed (low α) power-law graphs to increase work imbalance under the Pregel (fan-in) and GraphLab 1 (fan-out) abstractions but not under the PowerGraph abstraction, which evenly distributed high-degree vertex-programs. To evaluate this hypothesis we ran eight “workers” per system (64 total workers) and recorded the vertex-program time on each worker.

In Fig. 3.11a and Fig. 3.11b we plot the *standard deviation* of worker per-iteration runtimes, a measure of work imbalance, for power-law fan-in and fan-out graphs respectively. Higher standard deviation implies greater imbalance. We observe that lower α increases work imbalance for GraphLab 1 on fan-in, and Pregel on fan-out. On the other hand, the PowerGraph abstraction is unaffected in either edge direction.

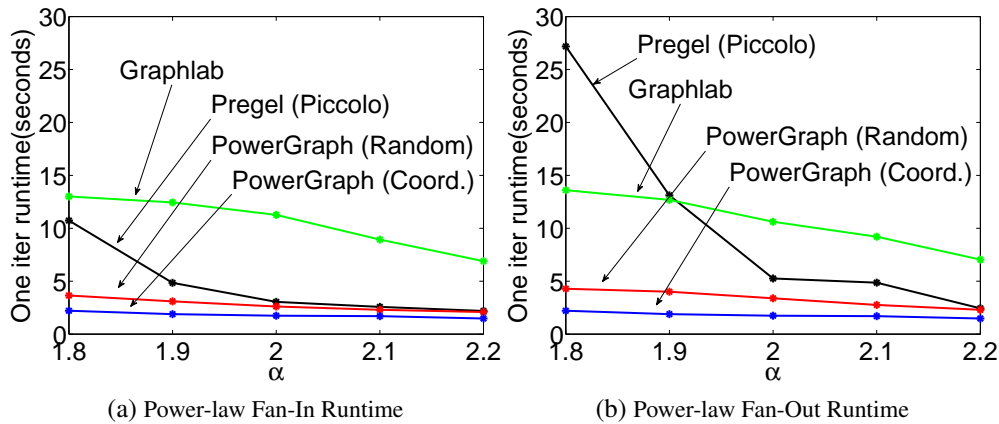


Figure 3.12: **Synthetic Experiments Runtime.** (a, b) Per iteration runtime of each abstraction on synthetic power-law graphs.

3.5.2 Communication Imbalance

Because GraphLab 1 and Pregel use edge-cuts, their communication volume is proportional to the number of ghosts: the replicated vertex and edge data along the partition boundary. If one message is sent per edge, Pregel’s combiners ensure that exactly one network message is transmitted for each ghost. Similarly, at the end of each iteration GraphLab 1 synchronizes each ghost and thus the communication volume is also proportional to the number of ghosts. PowerGraph on the other hand uses vertex-cuts and only synchronizes mirrors after each iteration. The communication volume of a complete iteration is therefore proportional to the number of mirrors induced by the vertex-cut. As a consequence we expect that PowerGraph will reduce communication volume.

In Fig. 3.11c and Fig. 3.11d we plot the bytes communicated per iteration for all three systems under power-law fan-in and fan-out graphs. Because Pregel only sends messages along out-edges, Pregel communicates more on power-law fan-out graphs than on power-law fan-in graphs.

On the other hand, GraphLab 1 and PowerGraph’s communication volume is invariant to power-law fan-in and fan-out since neither considers edge direction during data-synchronization. However, PowerGraph communicates significantly less than GraphLab 1 which is a direct result of the efficacy of vertex cuts. Finally, PowerGraph’s total communication increases only marginally on the denser graphs and is the lowest overall.

3.5.3 Runtime Comparison

PowerGraph significantly out-performs GraphLab 1 and Pregel on low α graphs. In Fig. 3.12a and Fig. 3.12b we plot the per iteration runtime for each abstraction. In both cases the overall runtime performance closely matches the communication overhead (Fig. 3.11c and Fig. 3.11d) while the computation imbalance (Fig. 3.11a and Fig. 3.11b) appears to have little effect. The limited effect of imbalance is due to the relatively lightweight nature of the PageRank computation and we expect more complex algorithms (e.g., statistical inference) to be more susceptible to imbalance. However, when greedy (coordinated) partitioning is used we see an additional 25% to 50% improvement in runtime.

3.6 Implementation and Evaluation

In this section, we describe and evaluate our implementation of the PowerGraph² abstraction. All experiments are performed on a 64 node cluster of Amazon EC2 `c4.4xlarge` Linux instances. Each instance has two quad core Intel Xeon X5570 processors with 23GB of RAM, and is connected via 10 GigE Ethernet. PowerGraph was written in C++ and compiled with GCC 4.5.

We implemented three variations of the PowerGraph abstraction. To demonstrate their relative implementation complexity, we provide the line counts, excluding common support code:

Bulk Synchronous (Sync): A fully synchronous implementation of PowerGraph as described in Sec. 3.3.3.1. [600 lines]

Asynchronous (Async): An asynchronous implementation of PowerGraph which allows arbitrary interleaving of vertex-programs Sec. 3.3.3.2. [900 lines]

Asynchronous Serializable (Async+S): An asynchronous implementation of PowerGraph which guarantees serializability of *all* vertex-programs (equivalent to “edge consistency” in GraphLab 1). [1600 lines]

In all cases the system is entirely symmetric with no single coordinating instance or scheduler. Each instances is given the list of other machines and start by reading a unique subset of the graph data files from HDFS. TCP connections are opened with other machines as needed to build the distributed graph and run the engine.

3.6.1 Graph Loading and Placement

The graph structure and data are loaded from a collection of text files stored in a distributed file-system (HDFS) by all instances in parallel. Each machine loads a separate subset of files (determined by hashing) and applies one of the three distributed graph partitioning algorithms to place the data *as it is loaded*. As a consequence partitioning is accomplished in parallel and data is immediately placed in its final location. Unless specified, all experiments were performed using the oblivious algorithm. Once computation is complete, the final vertex and edge data are saved back to the distributed file-system in parallel.

In Fig. 3.9b, we evaluate the performance of a collection of algorithms varying the partitioning procedure. Our simple partitioning heuristics are able to improve performance significantly across *all algorithms*, decreasing runtime and memory utilization. Furthermore, the runtime scales *linearly* with the replication factor: halving the replication factor approximately halves runtime.

3.6.2 Synchronous Engine (Sync)

Our synchronous implementation closely follows the description in Sec. 3.3.3.1. Each machine runs a single multi-threaded instance to maximally utilize the multi-core architecture. We rely on background communication to achieve computation/communication interleaving. The synchronous engine’s fully deterministic execution makes it easy to reason about programmatically and minimizes effort needed for tuning and performance optimizations.

²PowerGraph is now part of the GraphLab2 open source project and can be obtained from <http://graphlab.org>.

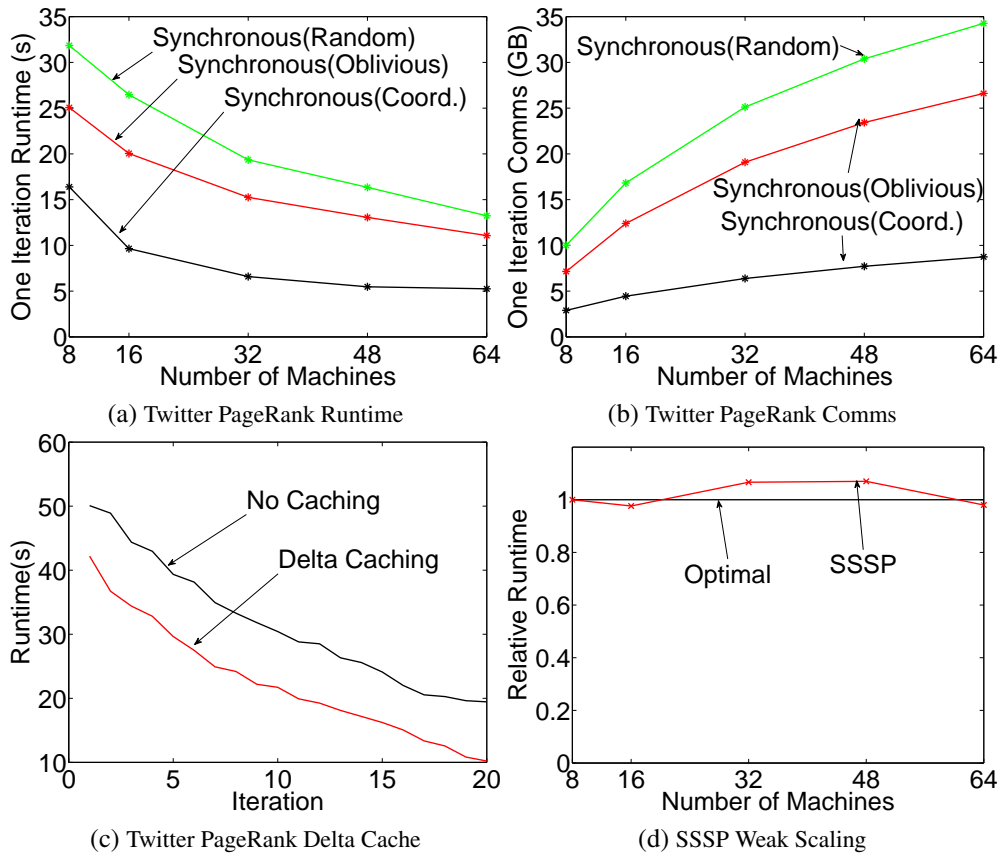


Figure 3.13: **Synchronous Experiments** (a,b) Synchronous PageRank Scaling on Twitter graph. (c) The PageRank per iteration runtime on the Twitter graph with and without delta caching. (d) Weak scaling of SSSP on synthetic graphs.

In Fig. 3.13a and Fig. 3.13b we plot the runtime and total communication of one iteration of PageRank on the Twitter follower network for each partitioning method. To provide a point of comparison (Table 3.2), the Spark [Zaharia et al., 2010] framework computes one iteration of PageRank on the same graph in 97.4s on a 50 node-100 core cluster [Stanton and Kliot, 2011]. PowerGraph is therefore between 3-8x faster than Spark on a comparable number of cores. On the full cluster of 512 cores, we can compute one iteration in 3.6s (later optimizations halves this to 1.8s).

The greedy partitioning heuristics improves both performance and scalability of the engine at the cost of increased load-time. The load time for random, oblivious, and coordinated placement were 59, 105, and 239 seconds respectively. While greedy partitioning heuristics increased load-time by up to a factor of four, they still improve overall runtime if more than 20 iterations of PageRank are performed. In Fig. 3.13c we plot the runtime of each iteration of PageRank on the Twitter follower network. Delta caching improves performance by avoiding unnecessary gather computation, decreasing total runtime by 45%. Finally, in Fig. 3.13d we evaluate weak-scaling: ability to scale while keeping the problem size per processor constant. We run SSSP (Fig. 3.4) on synthetic power-law graphs ($\alpha = 2$), with ten-million vertices per machine. Our implementation demonstrates nearly optimal weak-scaling and requires only 65s to solve a 6.4B edge graph.

3.6.3 Asynchronous Engine (Async)

We implemented the asynchronous PowerGraph execution model (Sec. 3.3.3.2) using a simple state machine for each vertex which can be either: `INACTIVE`, `GATHER`, `APPLY` or `SCATTER`. Once activated, a vertex enters the gathering state and is placed in a *local* scheduler which assigns cores to active vertices allowing many vertex-programs to run simultaneously thereby hiding communication latency. While arbitrary interleaving of vertex programs is permitted, we avoid data races by ensuring that individual gather, apply, and scatter calls have exclusive access to their arguments.

We evaluate the performance of the Async engine by running PageRank on the Twitter follower network. In Fig. 3.14a, we plot throughput (number of vertex-program operations per second) against the number of machines. Throughput increases moderately with both the number of machines as well as improved partitioning. We evaluate the gains associated with delta caching (Sec. 3.3.2) by measuring throughput as a function of time (Fig. 3.14b) with caching enabled and with caching disabled. Caching allows the algorithm to converge faster with fewer operations. Surprisingly, when caching is disabled, the throughput increases over time. Further analysis reveals that the computation gradually focuses on high-degree vertices, increasing the computation/communication ratio.

We evaluate the graph coloring vertex-program (Fig. 3.4) which cannot be run synchronously since all vertices would change to the same color on every iteration. Graph coloring is a proxy for many ML algorithms [Gonzalez et al., 2011]. In Fig. 3.14c we evaluate weak-scaling on synthetic power-law graphs ($\alpha = 2$) with five-million vertices per machine and find that the Async engine performs nearly optimally. The slight increase in runtime may be attributed to an increase in the number of colors due to increasing graph size.

3.6.4 Async. Serializable Engine (Async+S)

The Async engine is useful for a broad range of tasks, providing high throughput and performance. However, unlike the synchronous engine, the asynchronous engine is difficult to reason about programmatically. We therefore extended the Async engine to enforce serializability.

The Async+S engine ensures serializability by preventing adjacent vertex-programs from running simultaneously. Ensuring serializability for graph-parallel computation is equivalent to solving the dining philosophers problem where each vertex is a philosopher, and each edge is a fork. GraphLab 1 (Sec. 2.3.1 and Sec. 2.4.2.2) implements Dijkstra’s solution [Dijkstra, 1971] where forks are acquired *sequentially* according to a total ordering. Instead, we implement the Chandy-Misra solution [Chandy and Misra, 1984] which acquires all forks simultaneously, permitting a high degree of parallelism. We extend the Chandy-Misra solution to the vertex-cut setting by enabling each vertex replica to request only forks for local edges and using a simple consensus protocol to establish when all replicas have succeeded. The details of the algorithm are described in Appendix A

We evaluate the scalability and computational efficiency of the Async+S engine on the graph coloring task. We observe in Fig. 3.14c that the amount of achieved parallelism does not increase linearly with the number of vertices. Because the density (i.e., contention) of power-law graphs increases super-linearly with the number of vertices, we do not expect the amount of serializable parallelism to increase linearly.

In Fig. 3.14d, we plot the proportion of edges that satisfy the coloring condition (both vertices have different colors) for both the Async and the Async+S engines. While the Async engine quickly satisfies

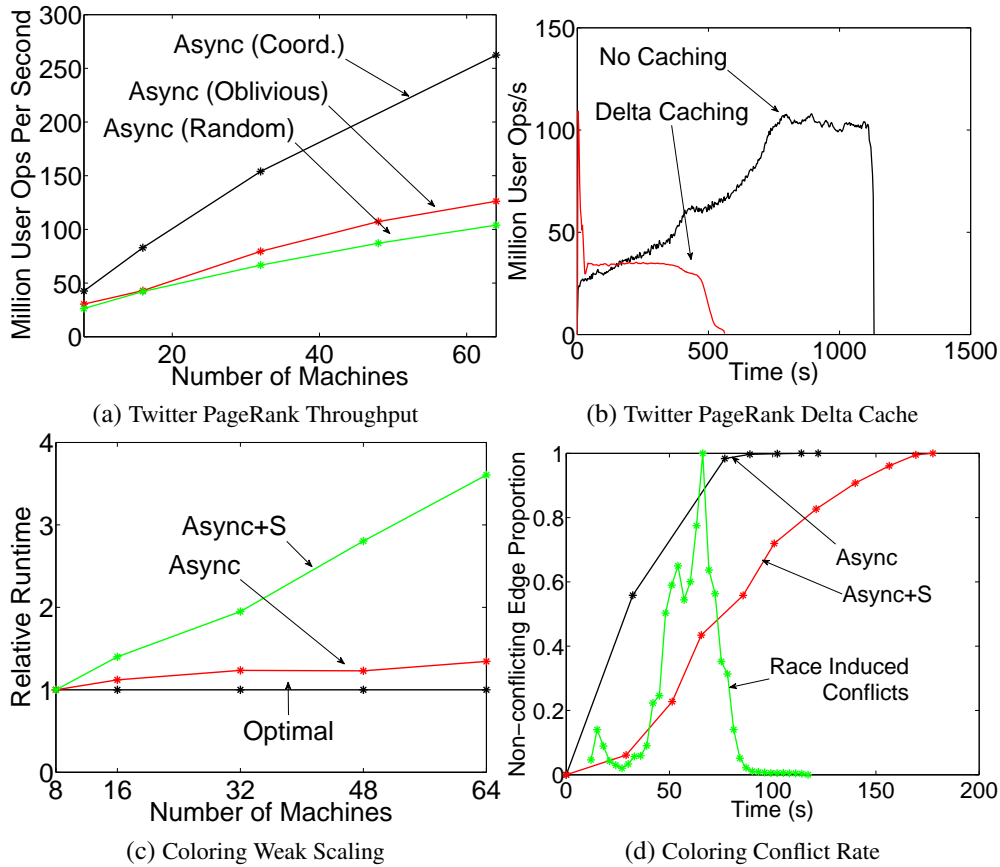


Figure 3.14: **Asynchronous Experiments** (a) Number of user operations (gather/apply/scatter) issued per second by Dynamic PageRank as # machines is increased. (b) Total number of user ops with and without caching plotted against time. (c) Weak scaling of the graph coloring task using the Async engine and the Async+S engine (d) Proportion of non-conflicting edges across time on a 8 machine, 40M vertex instance of the problem. The green line is the rate of conflicting edges introduced by the lack of consistency (peak 236K edges per second) in the Async engine. When the Async+S engine is used no conflicting edges are ever introduced.

the coloring condition for most edges, the remaining 1% take 34% of the runtime. We attribute this behavior to frequent races on tightly connected vertices. Alternatively, the Async+S engine performs more uniformly. If we examine the total number of user operations we find that the Async engine does more than *twice* the work of the Async+S engine.

Finally, we evaluate the Async and the Async+S engines on a popular machine learning algorithm: Alternating Least Squares (ALS). The ALS algorithm has a number of variations which allow it to be used in a wide range of applications including user personalization [Zhou et al., 2008] and document semantic analysis [Hofmann, 1999]. We apply ALS to the Wikipedia term-document graph consisting of 11M vertices and 315M edges to extract a mixture of topics representation for each document and term. The number of topics d is a free parameter that determines the computational complexity $O(d^3)$ of each vertex-program. In Fig. 3.15a, we plot the ALS throughput on the Async engine and the Async+S engine. While the throughput of the Async engine is greater, the gap between engines shrinks as d increases and computation dominates the consistency overhead. To demonstrate the importance of serializability, we plot in Fig. 3.15b the training error, a measure of solution quality, for both engines. We observe that while the Async engine has greater throughput, the Async+S engine *converges* faster.

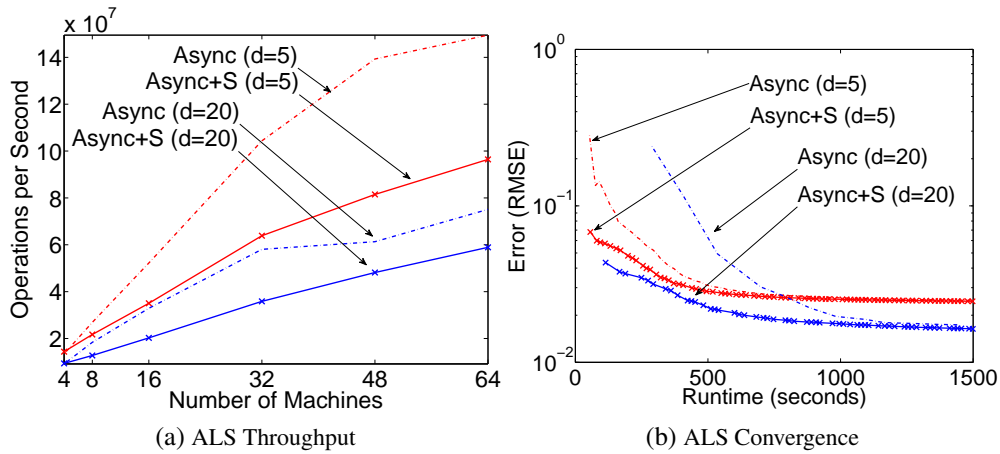


Figure 3.15: **(a)** The throughput of ALS measured in millions of User Operations per second. **(b)** Training error (lower is better) as a function of running time for ALS application.

The complexity of the Async+S engine is justified by the necessity for serializability in many applications (e.g., ALS). Furthermore, serializability adds predictability to the nondeterministic asynchronous execution. For example, even graph coloring may not terminate on dense graphs unless serializability is ensured.

3.6.5 Fault Tolerance

Like GraphLab 1 and Pregel, PowerGraph achieves fault-tolerance by saving a snapshot of the data-graph. The synchronous PowerGraph engine constructs the snapshot between super-steps and the asynchronous engine suspends execution to construct the snapshot. An asynchronous snapshot using GraphLab 1’s snapshot algorithm [Low et al., 2012] can also be implemented. The checkpoint overhead, typically a few seconds for the largest graphs we considered, is small relative to the running time of each application.

3.6.6 Applications

In Table 3.2 we provide comparisons of the PowerGraph system with published results on similar data for PageRank, Triangle Counting [Suri and Vassilvitskii, 2011], and collapsed Gibbs sampling for the LDA model [Smola and Narayanamurthy, 2010]. The PowerGraph implementations of PageRank are one to two orders of magnitude faster than published results. For LDA, the state-of-the-art solution is a heavily optimized system designed for this specific task by Smola and Narayanamurthy [2010]. In contrast, PowerGraph is able to achieve comparable performance using only 200 lines of user code.

The *3 orders of magnitude* performance difference between PowerGraph and Hadoop for the Triangle Counting task is due to an abstraction mismatch between Hadoop and the algorithm used in Suri and Vassilvitskii [2011]. Specifically, it can be showed that the Hadoop implementation described requires $O(\text{degree}^2)$ communication per vertex, while the PowerGraph implementation only requires $O(\text{replication} \times \text{degree})$. The need to write intermediate results to disk exacerbates the gap. This of course does not preclude the existence of more efficient Hadoop based triangle counting algorithms.

3.7 Conclusion

The need to reason about large-scale graph-structured data has driven the development of new graph-parallel abstractions such as GraphLab 1 and Pregel. However graphs derived from real-world phenomena often exhibit power-law degree distributions, which are difficult to partition and can lead to work imbalance and substantially increased communication and storage.

To address these challenges, we introduced the PowerGraph abstraction which exploits the Gather-Apply-Scatter model of computation to factor vertex-programs over edges, splitting high-degree vertices and exposing greater parallelism in natural graphs. We then introduced vertex-cuts and a collection of fast greedy heuristics to substantially reduce the storage and communication costs of large distributed power-law graphs. We theoretically related the power-law constant to the communication and storage requirements of the PowerGraph system and empirically evaluate our analysis by comparing against GraphLab 1 and Pregel. Finally, we evaluate the PowerGraph system on several large-scale problems using a 64 node EC2 cluster and demonstrating the scalability and efficiency and in many cases order of magnitude gains over published results.

The PowerGraph abstraction is highly expressive, capable of representing a large number of graph algorithms, matrix factorization algorithms, graphical model inference methods, etc. The GraphLab 2.1 release at <http://graphlab.org> includes our distributed implementation of over 24 different algorithms spread over 7 toolkits of Clustering, Computer Vision, Collaborative Filtering, Graph Analytics, Graphical Models, Linear Solvers and Topic Modeling, and has been downloaded thousands of times.

Finally, Kyrola et al. [2012] presents GraphChi: an efficient single-machine disk-based implementation of the GraphLab 1 abstraction. Impressively, it is able to significantly out-perform large Hadoop deployments on many graph problems while using only a single machine: performing one iteration of PageRank on the Twitter Graph in only 158s (PowerGraph: 1.8s). The techniques described in GraphChi can be used to add out-of-core storage to PowerGraph.

Our key contributions are:

- An analysis of the challenges of power-law graphs in distributed graph computation and the limitations of existing graph parallel abstractions (Sec. 3.1.2.1).
- The PowerGraph abstraction (Sec. 3.3) which factors individual vertex-programs.
- A delta caching procedure which allows computation state to be dynamically maintained (Sec. 3.3.2).
- A new fast approach to data layout for power-law graphs in distributed environments (Sec. 3.4).
- An theoretical characterization of network and storage (Theorem 3.4.2, Theorem 3.4.3).
- A high-performance open-source implementation of the PowerGraph abstraction (Sec. 3.6).
- A comprehensive evaluation of three implementations of PowerGraph on a large EC2 deployment using real-world ML applications (Sec. 3.5 and 3.6).

//

PageRank	Runtime	$ V $	$ E $	System
Hadoop [Kang et al., 2009]	198s	–	1.1B	50x8
Spark [Zaharia et al., 2010]	97.4s	40M	1.5B	50x2
Twister [Ekanayake et al., 2010]	36s	50M	1.4B	64x4
<i>PowerGraph (Sync)</i>	3.6s	40M	1.5B	64x8
<i>PowerGraph (Sync)</i>	7s	1.4B	6.6B	64x16
<i>PowerGraph (Sync)*</i>	1.8s	40M	1.5B	64x16

LDA	Tok/sec	Topics	System
<i>Smola et al.</i> [Smola and Narayanamurthy, 2010]	150M	1000	100x8
<i>PowerGraph (Async)</i>	110M	1000	64x16
<i>PowerGraph (Async)*</i>	110M	1000	64x16

Triangle Count	Runtime	$ V $	$ E $	System
Hadoop [Suri and Vassilvitskii, 2011]	423m	40M	1.4B	1636x?
<i>PowerGraph (Sync)</i>	1.5m	40M	1.4B	64x16
<i>PowerGraph (Sync)*</i>	15s	40M	1.4B	64x16

Table 3.2: Relative performance of PageRank, LDA and triangle counting on similar graphs. PageRank runtime is measured per iteration. Both PageRank and triangle counting were run on the Twitter follower network and LDA was run on Wikipedia. PageRank was also run on the Altavista web graph of 1.4B vertices, 6.6B edges. The systems are reported as number of nodes by number of cores. The *Powergraph* numbers were the original results reported in Gonzalez et al. [2012]. The *PowerGraph** numbers are numbers obtained in April 2013 after the implementation of additional optimizations.

Chapter 4

WarpGraph

Fine Grained Data-Parallelism for Usability

In the previous chapter, we presented the PowerGraph abstraction which restricts the abstraction to the strict **Gather-Apply-Scatter** model. And in doing so, exposed many optimization opportunities, providing large performance gains, and raising the bar for some common benchmarks. In this chapter, we address the usability and expressiveness issues of PowerGraph through the use of fine grained abstractions.

In Sec. 4.1 we begin by describing the limitations of the PowerGraph abstraction and show how PowerGraph is difficult to use in practice. To resolve the usability issues with PowerGraph, we introduce the WarpGraph abstraction in Sec. 4.2 which relaxes the restrictions in PowerGraph through the use of fine-grained data-parallel primitives. Through several examples, we show that the WarpGraph abstraction is easier to use and results in more readable code.

Finally, in Sec. 4.3 we describe an efficient distributed, asynchronous implementation of the WarpGraph abstraction which makes use of user-mode context switching and coroutines to hide communication latency. We evaluate the implementation on three challenging tasks and show that WarpGraph significantly out-performs the PowerGraph asynchronous engine, and can achieve nearly half the throughput rate of the PowerGraph synchronous engine. The performance of the WarpGraph implementation demonstrates that efficient distributed asynchronous computation is feasible even on commodity hardware.

4.1 PowerGraph Limitations

While the PowerGraph abstraction appears to be highly expressive, it has two key limitations. Firstly, the PowerGraph abstraction is insufficiently expressive for complex vertex programs. And secondly, the PowerGraph abstraction is difficult to work with in practice. We further elaborate on these two limitations.

4.1.1 Expressiveness

Not all vertex program computations can be expressed as a single GAS operation. There are situations where multiple Gather/Scatter operations are required. Since the Gather operation is simply an abstract sum over all neighboring information, it can compute only a **decomposable** function over the neighborhood.

We demonstrate this with an example of a computation which cannot be expressed as a single GAS operation. We consider the vertex program which computes the entropy of a discrete probability distribution over its adjacent vertices, where the probability of selecting an vertex is proportionate to the weight of the vertex. We can write this easily as a GraphLab 1 vertex program:

```
NeighborhoodEntropy( $D_v, D_*, D_{*\rightarrow v}, D_{v\rightarrow*} \in \mathcal{S}_v$ ) begin
  // Compute the normalizer
   $s = \sum_{u \in \mathbf{N}[v]} D_u$ 
  // Compute the entropy
  entropy =  $\sum_{u \in \mathbf{N}[v]} (D_u/s) \log(D_u/s)$ 
end
```

However, the above NeighborhoodEntropy function, cannot be implemented in PowerGraph as a single GAS vertex program since it requires two consecutive, non-separable aggregations over the neighborhood; two Gather operations are required. While a simple workaround is possible by using the Gather to accumulate the complete list of neighboring values, this comes at a large increase in communication requirements which eliminates the performance benefit of the PowerGraph abstraction. Alternatively, we can take advantage of the dynamic scheduling mechanism to connect consecutive vertex program executions: after a Gather is performed to compute the normalizer, we store the normalizer in the vertex state and schedule the current vertex again to compute the entropy. This procedure essentially emulates a state machine on each vertex, and incurs substantial complexity.

The need for multiple Gather operations show up in some Machine Learning algorithms such as Restricted Boltzmann Machines for matrix factorization [Salakhutdinov et al., 2007] and the SVD++ algorithm [Koren, 2008], making them challenging to implement in PowerGraph.

4.1.2 Usability

While the PowerGraph abstraction is abstractly simple, it can be difficult to reason about since it slices up the vertex program execution finely into four functions: `gather`, `sum`, `apply` and `scatter`. While a GraphLab 1 update function can be easily understood by a novice user, understanding a PowerGraph vertex program first requires the user to have deep understanding of the hidden dataflow between the four functions of the vertex program.

We can compare equivalent implementations of a dynamic version of the PageRank algorithm in both GraphLab 1 and PowerGraph, where a vertex schedules its out-neighbors to be executed only if its current PageRank value changes by more than some tolerance (10^{-5}). We list the GraphLab 1 implementation in Alg. 4.1 and the PowerGraph implementation in Alg. 4.2. We observe that the GraphLab 1 implementation is easy to understand since the PageRank computation is simply described as an explicit for-loop over the neighborhood, allowing a even user who is not knowledgeable about the GraphLab 1 engine to recognize

```

PageRankUpdate ( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) {
  prev_pagerank =  $D_v$ .rank

  // compute the sum, iterate over all in neighbors
  new_pagerank = 0.15
  ForEach( $u \in \text{InNeighbors}(v)$ ) {
    new_pagerank = new_pagerank + 0.85  $\times D_u$ .rank / OutDegree( $u$ )
  }
   $D_v$ .rank = new_pagerank

  // Reschedule if PageRank changes sufficiently
  if(|new_pagerank - prev_pagerank| >  $\epsilon$ ) {
    ForEach( $u \in \text{OutNeighbors}(v)$ ) {
      Schedule( $u$ )
    }
  }
}

```

Figure 4.1: Dynamic PageRank Graphlab 1 Update Function.

the computation. On the other hand, the PowerGraph implementation in Alg. 4.2 can be difficult to follow since the `gather` function is implicitly executed over the neighborhood, and the `sum` function describes an implicit aggregation operation. As a result, a user must have good familiarity with the PowerGraph engine to understand the behavior of the vertex program.

```

// gather_nbrs: in neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_u$ .rank / OutDegree( $u$ )
}
sum( $a$ ,  $b$ ) {
    return  $a + b$ 
}

apply( $D_v$ , acc) {
    // acc contains the result of the gather. Compute new PageRank
    new_pagerank = 0.15 + 0.85 × acc
    // store the change in the vertex to pass to the scatter
     $D_v$ .lastchange = new_pagerank -  $D_v$ .rank

     $D_v$ .rank = new_pagerank
}

// scatter_nbrs: out neighbors
scatter( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    // Reschedule if PageRank changes sufficiently
    if(| $D_v$ .delta|>ε) {
        Schedule( $u$ )
    }
}

```

Figure 4.2: Dynamic PageRank PowerGraph vertex program.

4.2 WarpGraph

MapReduce [Dean and Ghemawat, 2004], Pregel [Malewicz et al., 2010], Dryad/Linq [Isard et al., 2007], Spark [Zaharia et al., 2010] and many other common parallel computation abstractions are **coarse-grained** data-parallel abstractions, where operations and transformations are defined to operate on large collections of data simultaneously.

On the other **fine-grained** task parallel abstractions, more commonly seen in shared memory multithreading architectures such as QThreads [Wheeler et al., 2008], Cilk [Randall, 1998] and Intel Threading Building Blocks [Intel, 2013], or distributed shared memory architectures like Grappa [Nelson et al., 2011], allow the creation of large numbers of distinct parallel tasks where each could operate on very small units of data.

The PowerGraph abstraction is a hybrid abstraction, having properties of both task-parallel and data-parallel execution models. While PowerGraph is a fine-grained task-parallel abstraction where a dynamic scheduler generates tasks which are executed by vertex programs, the restrictive Gather-Apply-Scatter structure of the vertex program is a data parallel Map-Reduce-Transform operation, but *restricted to the neighborhood of the vertex*.

To resolve the issues with PowerGraph described in Sec. 4.1, we designed WarpGraph, which generalizes the data-parallel property of PowerGraph and expose a complete fine-grained data-parallel abstraction within each vertex program. And as a result, improving usability of the abstraction. The WarpGraph abstraction retains the same dynamic scheduling model from PowerGraph, and only modifies the vertex program specification.

4.2.1 Update Function

Just like in GraphLab 1, in WarpGraph the user implements the vertex program as a simple stateless procedure called an **update function**. However, unlike GraphLab 1, the update function has direct memory access to *only the data on the central vertex*. The update function must instead use local data-parallel aggregate operations to access data on adjacent vertices and edges.

We demonstrate WarpGraph by describing a WarpGraph implementation of the dynamic PageRank example from Sec. 4.1.1. The Dynamic PageRank computation can be decomposed into two data-parallel pieces. First, the new PageRank value is computed via a weighted sum across the neighborhood. Following which, if the new value of the PageRank changes sufficiently, we schedule all neighbors along out edges for execution. To implement the Dynamic PageRank procedure in WarpGraph simply requires the user to explicitly specify these two data-parallel components. The first component is a MapReduce over in-edges, and the second is a broadcast operation over out-edges; WarpGraph provides primitives for both operations.

In Alg. 4.3, we provide the C++ code for the dynamic PageRank algorithm as implemented in WarpGraph. The update function (line 1) calls a MapReduce operation on line 7 which aggregates across the in edges of the current vertex. The Map operation *pagerank_map* (line 14), is evaluated on each neighboring vertex connected via an in-edge, and the return values are combined using the *pagerank_combine* function (line 18). The new pagerank value is computed using the result of the MapReduce operation, and if the new PageRank values changes sufficiently, a *broadcast_signal* call (line 11) is used to schedule neighboring vertices along out edges for execution.

```

PageRankUpdate( $D_v$ ) {
  prev_pagerank =  $D_v$ .rank
  // compute a map reduce over the neighborhood selecting only the in edges
  new_pagerank = 0.15 + 0.85 *
    map_reduce_neighbors(pagerank_map, pagerank_combine, IN_EDGES)
   $D_v$ .rank = new_pagerank

  if (|new_pagerank - prev_pagerank| >  $\epsilon$ ) {
    broadcast(OUT_EDGES)
  }
}

// helper functions
pagerank_map( $D_u, D_{u-v}, D_v$ ) {
  return  $D_u$ .rank / OutDegree( $u$ )
}

pagerank_combine( $a, b$ ) {
  return  $a + b$ 
}

```

Figure 4.3: Dynamic PageRank WarpGraph Update Function.

Comparing the WarpGraph PageRank implementation against the GraphLab 1 (Alg. 4.1) and PowerGraph (Alg. 4.2) versions, we observe that the WarpGraph implementation is as short as the GraphLab 1 update function, and just as easy to understand. Further note that while we implemented *pagerank_map* and *pagerank_combine* as separate functions, under certain languages with functional capabilities, these can be implemented as inline lambdas, further simplifying the code.

The WarpGraph implementation is able to both maintain the simplicity of the GraphLab 1 update function, and like PowerGraph, is able to explicitly expose the available parallelism available in the neighborhood aggregate operations. WarpGraph therefore combines the benefits of both GraphLab 1 and PowerGraph.

4.2.2 Fine Grained Data-Parallel Primitives

The PageRank example demonstrated two primitives: MapReduce which computes an aggregate over the neighborhood, and Broadcast which schedules a set of neighbors for execution. In this section, we formally define these two primitives as well a few other useful primitives which add richness to the abstraction and expand the space of algorithms we can describe efficiently.

Where the update function is executed on vertex u , the following local primitives are available:

MapReduce Equivalent to the Gather operation. Performs an aggregation over the neighborhood.

Transform Equivalent to the Scatter operation. Modifies adjacent edges.

Broadcast Schedules a subset of neighbors depending on a user-defined selector operation.

DHTGather and DHTScatter Reads and writes values from a Distributed Hash Table (DHT).

We now describe the exact behavior of these primitives in detail. We use the notation D_u to identify the data on vertex u , and use the notation D_{u-v} to identify the data on the edge (u, v) .

4.2.2.1 MapReduce

The **MapReduce** primitive as demonstrated in the PageRank example in Sec. 4.2.1 allows the update function to compute an aggregate over the neighborhood of the vertex, and provides the same functionality as the Gather operation in PowerGraph. The **MapReduce** operation takes three arguments: a map function, a combiner, and the set of edges to aggregate over.

$$\text{result} = \text{MapReduce}(\text{MapFunction}, \text{CombineFunction}, \text{EdgeSet})$$

$$\text{MapFunction} : (D_u, D_{u-v}, D_v) \rightarrow T$$

$$\text{CombineFunction} : (T \text{ left}, T \text{ right}) \rightarrow T$$

$$\text{EdgeSet} \in \{\text{IN_EDGES}, \text{OUT_EDGES}, \text{ALL_EDGES}\}$$

The MapReduce operation is semantically equivalent to:

MapReduce(MapFunction, CombineFunction, EdgeSet) **begin**

```

first = True
acc ← Empty
foreach neighbor u in EdgeSet do
  mapresult ← MapFunction( $D_u, D_{u-v}, D_v$ )
  if first then
    first = False
    acc ← mapresult
  else
    acc ← CombineFunction(acc, mapresult)
return acc

```

An overload is provided for the simpler case where MapFunction only needs the value of of the neighboring vertex. In which case, the MapFunction has the simpler form:

$$\text{MapFunction} : (D_v) \rightarrow T$$

4.2.2.2 Transform

The **Transform** primitive provides the same functionality as the Scatter operation in PowerGraph, and allows the update function to make modifications to the edges adjacent to the current vertex. The **Transform** operation takes two arguments: a transform function, and the set of edges to transform.

$$\text{Transform}(\text{TransformFunction}, \text{EdgeSet})$$

$$\text{TransformFunction} : (D_u, D_{u-v}, D_v) \rightarrow D_{u-v}$$

$$\text{EdgeSet} \in \{\text{IN_EDGES}, \text{OUT_EDGES}, \text{ALL_EDGES}\}$$

The transform operation is semantically equivalent to:

```

Transform(TransformFunction, EdgeSet) begin
  | foreach neighbor u in EdgeSet do
  | |  $D_{u-v} \leftarrow \text{TransformFunction}(D_u, D_{u-v}, D_v)$ 

```

4.2.2.3 Broadcast

The **Broadcast** primitive allows the update function to selectively choose a subset of adjacent vertices to schedule. It takes two arguments, a selection function, and the set of edges to evaluate the selection.

Broadcast(SelectionFunction, EdgeSet)

SelectionFunction : $(D_u, D_{u-v}, D_v) \rightarrow \text{True/False}$

EdgeSet $\in \{\text{IN_EDGES}, \text{OUT_EDGES}, \text{ALL_EDGES}\}$

The **Broadcast** primitive is semantically equivalent to:

```

Broadcast(SelectionFunction, EdgeSet) begin
  | foreach neighbor u in EdgeSet do
  | | if SelectionFunction( $D_u, D_{u-v}, D_v$ ) then
  | | | Signal( $v$ )

```

An overload is provided for the simpler case where the selection function always evaluates to True, in which case **Broadcast** only takes one argument: the EdgeSet.

4.2.2.4 DHTGather and DHTScatter

Just as the GraphLab 1 abstraction provides a **Sync** operation (Sec. 2.2.5) which maintains global aggregates, we provide in the WarpEngine, access to a Distributed Hash Table (DHT) where global information can be stored and maintained. An update function can read and write information to the DHT using the DHTGather and DHTScatter functions.

hashmap = **DHTGather**(requested_keys)
DHTScatter(hashmap, CombineFunction)

CombineFunction : $(\text{Any left}, \text{Any right}) \rightarrow \text{Any}$

DHTGather will obtain a subset of the Distributed Hash Table containing only the keys in `requested_keys`. **DHTScatter** performs the reverse operation. Given a hash map mapping keys to values, the **DHTScatter**

operation will use `combine_function` to merge the value in the hash map and the corresponding value in the Distributed Hash Table.

```

DHTGather(requested_keys) begin
  H ← Empty Hash Map
  foreach  $key \in requested\_keys$  do
    | H[key] = DHT[key]
  return H

DHTScatter(hashmap, CombineFunction) begin
  foreach  $(key, value) \in hashmap$  do
    | DHT[key] ← CombineFunction(DHT[key], value)
  return H

```

4.2.3 Expressiveness

Every PowerGraph vertex program can be directly translated to a WarpGraph update function by exploiting the direct correspondence between Gather and MapReduce, and between Scatter and Transform. However, the WarpGraph update function has simpler semantics, and is easier to work with as demonstrated by the dynamic PageRank example in Alg. 4.3.

We demonstrate this once again for the Graph Coloring task, where the objective is to assign a color to each vertex of a graph such that adjacent vertices have different colors. The WarpGraph implementation (Alg. 4.5) follows the PowerGraph implementation closely: a neighborhood MapReduce call is first used to gather the unique set of colors in the neighborhood, a unique color is selected, then a broadcast operation is used to activate neighbors with conflicting colors. Once again, we observe that the code is simple and intuitive.

The WarpGraph update function can also express vertex programs which require multiple gather operations. For instance, the NeighborhoodEntropy program described in Sec. 4.1.1 can now be written as a WarpGraph function:

```

NeighborhoodEntropy( $D_v$ ) {
  sum = map_reduce_neighbors( $fn(x) \Rightarrow x$ , // Map Function
                              $fn(x, y) \Rightarrow x + y$ , // Combine Function
                             ALL_EDGES)
  entropy = map_reduce_neighbors( $fn(x) \Rightarrow (x/sum) \log(x/sum)$ , // Map Function
                                  $fn(x, y) \Rightarrow x + y$ , // Combine Function
                                 ALL_EDGES)
}

```

More generally, WarpGraph can also express update functions with an arbitrary number of gather/scatter operations, or even gather/scatter operations which execute conditionally. We list in Fig. 4.4 some new update function patterns that are easy to express in WarpGraph, but are difficult to express in PowerGraph.

Finally, the DHT functions allow for global aggregates of the computation state to be maintained. For

```

MultipleGather ( $D_v$ ) {
    // multiple gather operations
    map_reduce_neighbors(map_op, combine_op, ALL_EDGES)
    map_reduce_neighbors(map_op2, combine_op2, ALL_EDGES)
}

ScatterThenGather ( $D_v$ ) {
    // arbitrary reordering of scatter and gather operations
    edge_transform(transform_op, ALL_EDGES)
    map_reduce_neighbors(map_op, combine_op, ALL_EDGES)
}

ConditionalGather ( $D_v$ ) {
    // conditional evaluation of gather or scatter operations
    if ( $D_v = 1$ ) {
        map_reduce_neighbors(map_op, combine_op, ALL_EDGES)
    } else {
        map_reduce_neighbors(map_op2, combine_op2, ALL_EDGES)
    }
}

```

Figure 4.4: WarpGraph Update Function patterns that are difficult to express in PowerGraph

instance, we can extend the Dynamic PageRank algorithm to maintain a DHT entry containing the sum of the PageRank of all vertices by pushing changes to the pagerank value to a DHT entry. The resultant algorithm only requires a few additional lines of code to the update function (Alg. 4.6).

4.3 Implementation

Our implementation of WarpGraph extends the PowerGraph implementation from the previous chapter. We use the same vertex separator distributed graph representation from PowerGraph, as it provides substantial benefit in reducing communication requirements.

The WarpGraph execution engine implementation shares the same latency hiding strategy as PowerGraph’s Asynchronous engine (Sec. 3.6.3): to hide the latency of distributed evaluation of the local data-parallel primitives, it is necessary to run a large number of update functions at once.

However, unlike in PowerGraph where the communication phases of a vertex program are well defined by the boundaries between the Gather/Apply/Scatter functions allowing the PowerGraph Asynchronous engine to be implemented using a simple state machine, WarpGraph provides no such luxury. An update function could issue a local MapReduce at any stage in the function evaluation.

The solution is to massively over-thread: by creating thousands of threads, each running an update function. Threads which are waiting on communication can then be descheduled, while the much smaller number of real processors continue execution of other threads. This design is inspired by Grappa [Nelson et al., 2011] which implements an efficient distributed global address space system using the same technique to trade latency for throughput.

However, regular PThread stacks are too large (a few MB in practice) to support a large number of threads and the kernel context switch is too slow to maintain the rapid switching needed to manage a large num-

```

WCCUpdate( $D_v$ ) {
    neighborhood_colors = map_reduce_neighbors(color_map, color_union, ALL_EDGES)

    // find the smallest color not described in the neighborhood
    size_t neighborhoodsize = neighborhood_colors.size()
    for (color_type curcolor = 0; curcolor < neighborhoodsize + 1; ++curcolor) {
        if (curcolor  $\notin$  neighborhood_colors) {
            vertex.data() = curcolor;
            break;
        }
    }
    broadcast(schedule_neighbors_if_different, ALL_EDGES);
}

// helper functions
color_map( $D_u, D_{u-v}, D_v$ ) {
    return { $D_u$ }
}

color_union(a, b) {
    return a  $\cup$  b
}

schedule_neighbors_if_different( $D_u, D_{u-v}, D_v$ ) {
    if ( $D_u = D_v$ ) Schedule( $u$ )
}

```

Figure 4.5: WarpGraph implementation of Graph Coloring.

ber of threads efficiently. To resolve this issue we implemented a user mode coroutine library (also called fibers, cooperative threading, or M:N threading), which allow us to both control the size of the thread stacks, and using the Boost Context context-switch library [Kowalke, 2013], perform the context-switch entirely in user-mode. Our coroutine implementation is simple requiring only 400 lines of code, but achieves a context switch rate of over 8M context switches per second on a 2.8Ghz Intel i7 930 processor.

The engine is then implemented using the coroutine library to support a large number of concurrently executing update functions. In practice, we find that about 3,000 threads are sufficient to maintain throughput on 10-Gigabit Ethernet networks (Sec. 4.4.1); increasing the number of the threads beyond 3,000 do not provide performance gains¹.

Consistency is managed in the same manner as in PowerGraph Async engine (Sec. 3.6.3), allowing update functions may interleave arbitrarily. Only local data races are prevented by ensuring that individual vertex/edge data accesses are locked. A serializable execution engine can also be implemented using the same technique as the PowerGraph Async+S engine (Sec. 3.6.4) through the extension to the Chandy Misra distributed locking protocol (Appendix A).

Finally, while the WarpGraph abstraction does not specify consistency requirements on the DHT implementation, our implementation is strongly consistent on individual key accesses (**DHTGather** always reads the most recent values, and **DHTScatter** writes immediately), making use of the fast context switch

¹The name of the system is inspired by this design. *Warp threads* are the *set of threads held in tension on a loom*.

```

// Combiner Function : Adds the pushed value to the actual DHT value
// -----
dht_combiner(old_value, pushed_value) {
    return old_value + pushed_value;
}

// In the Update Function: Push the change in value to the key
// -----
dht_scatter_values[PAGERANK_SUM_KEY] = newval -  $D_u$ 
dht_scatter(dht_scatter_values, dht_combiner);

```

Figure 4.6: Dynamic PageRank WarpGraph Update Function which maintains a sum of PageRank in a DHT key. Modified from Alg. 4.3

to hide DHT access latency. This design allows the DHT to behave like a Distributed Shared Memory system. Alternate or future implementations may use a weaker consistency model which provides better performance.

4.4 Evaluation

We evaluate the performance of our WarpGraph implementation in comparison to our PowerGraph implementation from the previous chapter. In order to stress the engine implementation, we focus on tasks with relatively light-weight vertex programs, thus making the performance entirely depend on the engine overhead. We therefore evaluate the system on the PageRank, Graph Coloring, and Stochastic Gradient Descent tasks.

All experiments are performed on a 32-node cluster of cc2.8xlarge instances on Amazon EC2 instances. Each instance has two 8-core Intel Xeon E5-2670 processors with 60.5GB of RAM, and is connected via 10 Gigabit Ethernet. Both PowerGraph and WarpGraph are written in C++ and compiled with GCC 4.5. The coordinated graph partitioning strategy (Sec. 3.4.2) is used for all experiments.

4.4.1 PageRank

We evaluate the WarpGraph implementation (Alg. 4.3) of Dynamic PageRank against the Synchronous PowerGraph implementation (Alg. 4.2). We use the Twitter follower network (42M vertices and 1.5B edges) from Kwak et al. [2010] for all experiments. While the PowerGraph PageRank algorithm can be run asynchronously, we find that the Asynchronous PowerGraph implementation required over **16x** the runtime of the synchronous execution, making extensive evaluation unfeasible. We therefore omit Asynchronous PowerGraph from the rest of the PageRank evaluation.

We first evaluate the effect of varying the number of threads created. Intuitively, a certain minimum number of threads is needed to hide the effect of communication latency, beyond which there is no benefit. In Fig. 4.7a, we plot the runtime of WarpGraph running on 32 machines, varying the number of threads created per machine. We observe that as the number of threads increase from 500, runtime improves dramatically as we successfully hide communication latency. However, once the number of threads exceed

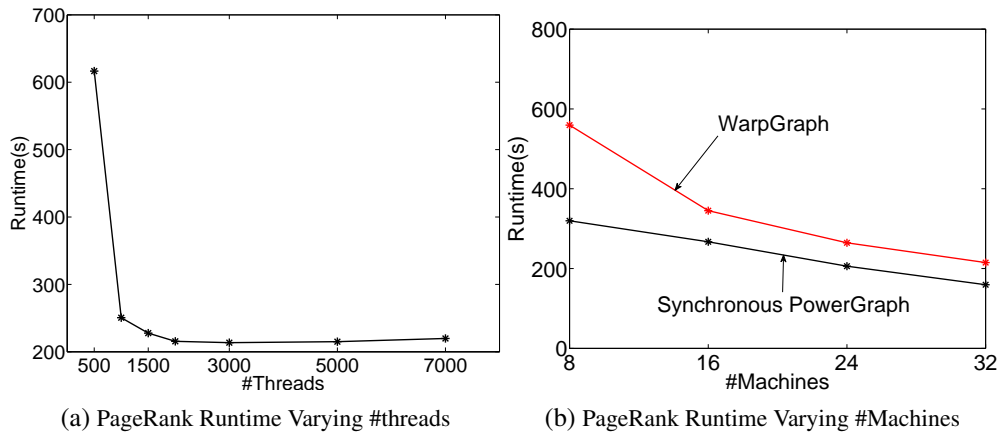


Figure 4.7: **(a)** The runtime for PageRank on Twitter follower graph using WarpGraph varying the number of threads created per machine. 32 machines are used, and the termination tolerance is set to 10^{-3} . **(b)** The runtime for PageRank on the Twitter follower graph using WarpGraph and the Synchronous PowerGraph engine varying the number of machines. The termination tolerance is set to 10^{-3} , and 5,000 threads are used for WarpGraph. The Asynchronous PowerGraph runtime required over 16x the runtime of Synchronous PowerGraph engine and is therefore not plotted.

3000, the runtime flattens, and any further increase in the number of threads has almost no effect on the runtime.

We next evaluate the amount of time required to achieve convergence at tolerances ranging from 10^{-1} to 10^{-5} . In Fig. 4.8a and Fig. 4.8b we plot the amount of time, and the number of updates needed to achieve a certain convergence accuracy using both WarpGraph and Synchronous PowerGraph. We observe that while WarpGraph appears to be consistently slower than Synchronous PowerGraph by about 50s, the number of updates required to achieve convergence is significantly lower, issuing 45% less computation to achieve 10^{-5} accuracy. While it has been suggested that asynchronous computation can converge faster than synchronous computation [Bertsekas and Tsitsiklis, 1989], such results can be hard to realize on commodity distributed systems due to the larger overhead of asynchronous communication. In this experiment, we demonstrate that with the right engine design, an asynchronous system can be competitive with a synchronous system. Furthermore, the relative immaturity of the WarpGraph implementation provides numerous optimization opportunities and we believe that it is possible to close the performance gap, or even out-perform synchronous PowerGraph on this task.

Finally, in Fig. 4.7b, we plot the runtime of the PageRank using WarpGraph and Synchronous PowerGraph varying the number of machines. We observe that both implementations exhibit positive scaling; decreasing runtime as the number of machines increase. Since the PageRank task is extremely communication bound, perfect speedup is difficult to achieve. Nonetheless, the WarpGraph implementation decreases its runtime by nearly **3x** with **4x** the number of machines.

4.4.2 Graph Coloring

We next evaluate the performance of the Graph Coloring task on both WarpGraph and PowerGraph. This task *requires* asynchronous computation and will not converge if run synchronously. We therefore used the Asynchronous engine from the PowerGraph implementation. In Fig. 4.9a we plot the time taken to

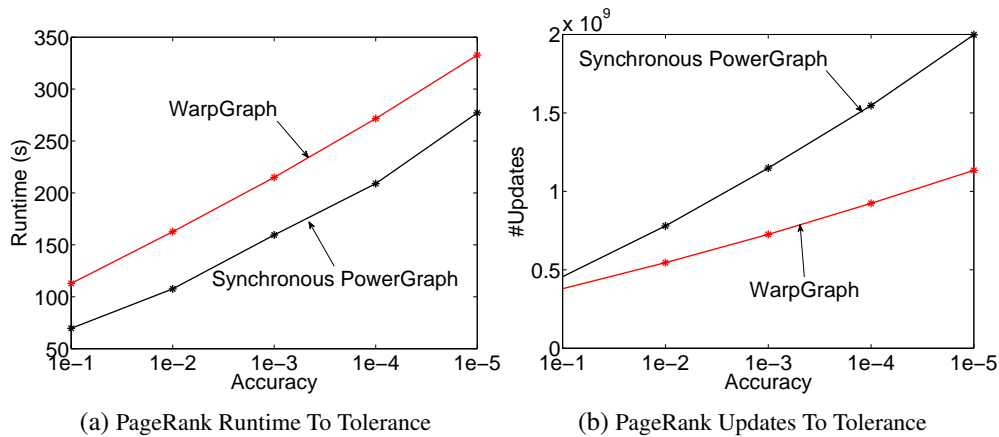


Figure 4.8: The (a) *runtime* and (b) the *number of updates* to achieve a certain convergence tolerance for PageRank on the Twitter follower graph, using WarpGraph and the Synchronous PowerGraph engine. 32 machines are used, and 5,000 threads are used for WarpGraph. The Asynchronous PowerGraph runtime required over 16x the runtime of Synchronous PowerGraph engine and is therefore not plotted.

color the Twitter follower graph of 42M vertices and 1.5B edges using both WarpGraph and PowerGraph on 32 EC2 machines. We observe that WarpGraph is substantially faster than PowerGraph, requiring less than half the amount of time to converge to a solution.

Since both WarpGraph and PowerGraph are run without edge consistency, the total number of updates required to achieve a valid coloring could be greater than the total number of vertices in the graph. For instance, simultaneously running vertex programs on adjacent vertices could both choose the same color and as a result *both* be rescheduled for computation by the broadcast/scatter call. In Fig. 4.9b, we plot the number of vertex updates required by the WarpGraph and by PowerGraph, as well as the optimal number of updates (the number of vertices in the graph). We observe that WarpGraph required about 20% less updates than PowerGraph, and is quite close to optimality.

We may attribute the large decrease in the number of updates to the *quality of asynchrony*. To reduce the chances of situations described above where adjacent vertices are modified simultaneously, it is important to complete each individual vertex program as quickly as possible. We therefore added performance counters to both PowerGraph and WarpGraph to track the amount of time each vertex program takes to complete. We find that each vertex program in Asynchronous PowerGraph requires an average of 200ms to complete, while the update function in WarpGraph completes in less than 10ms, or over **20x** faster resulting in improved asynchronous behavior.

4.4.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a common convex optimization procedure which is highly popular due to its simplicity and effectiveness on a broad range of problems. In this section, we focus on the Sparse-SGD setting where the number of parameters are extremely large, but each datapoint is sparse and only affects a small number of parameters. Such settings are commonly realized in many tasks such as matrix factorization [Koren et al., 2009] and text classification [Lewis et al., 2004].

While the original SGD procedure is defined only for the sequential case, there is a large amount of recent work in parallelizing and distributing the algorithm. In Recht et al. [2011], the authors described an

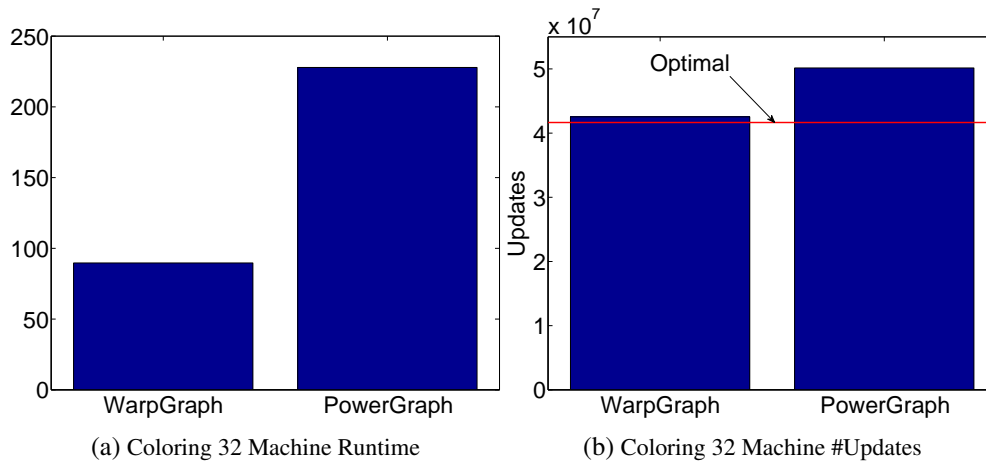


Figure 4.9: The (a) *runtime* and (b) the *number of updates* of the Graph Coloring task on 32 machines using WarpGraph and Asynchronous PowerGraph.

Algorithm 4.1: WarpGraph SGD

SGDUpdateFunction() begin

```

for  $(x, y) \in \text{FractionOfDataPoints}$  do
  weights = DHTGather(Features in x)
  gradient = ComputeGradient(weights, x, y)
  DHTScatter(gradient)

```

algorithm called Hogwild, and showed that in the shared memory setting a naive parallelization of SGD is effective due to SGD’s tolerance to race conditions. In the distributed setting, several different models have been proposed, ranging from MapReduce-able models which average multiple descents [Zinkevich et al., 2010], to more complex procedures which synchronize either parameters or gradients more frequently in smaller batches [Cipar et al., 2013, Dekel et al., 2012].

In this experiment, we propose a simpler alternative: to use the strongly consistent DHT as distributed shared memory, and to create a large number of threads on each machine to hide the memory access latency. This allows us to emulate the Hogwild shared-memory-parallel SGD procedure ([Recht et al., 2011]) in the distributed setting.

The key advantage of the distributed shared memory model of SGD, is its simplicity: the user does not have to manage communication, but can instead focus on the computation. To implement this procedure in WarpGraph simply requires the user to create a graph of disconnected vertices (as many as the desired number of threads), and implement an update function which iterates over a subset of data points, reading and writing from the DHT as needed². The algorithm is described in pseudo-code in Alg. 4.1.

We evaluate our SGD implementation using a synthetic binary logistic regression dataset of $k = 1\text{M}$ parameters. A ground truth parameter vector $w \in \mathbb{R}^k$ is generated by sampling each parameter uniformly from the range $[-1, 1]$. Sparse data points $x_i \in \mathbb{R}^k$ with 50 features are randomly generated, and the class is determined by sampling from the true logistic classification probabilities. A step size of $0.2/\sqrt{t}$ was

²To support such patterns in a simpler manner, the WarpGraph implementation provides the ability to launch custom user-defined coroutines which are not associated with the graph, making it unnecessary to create the graph of disconnected vertices.

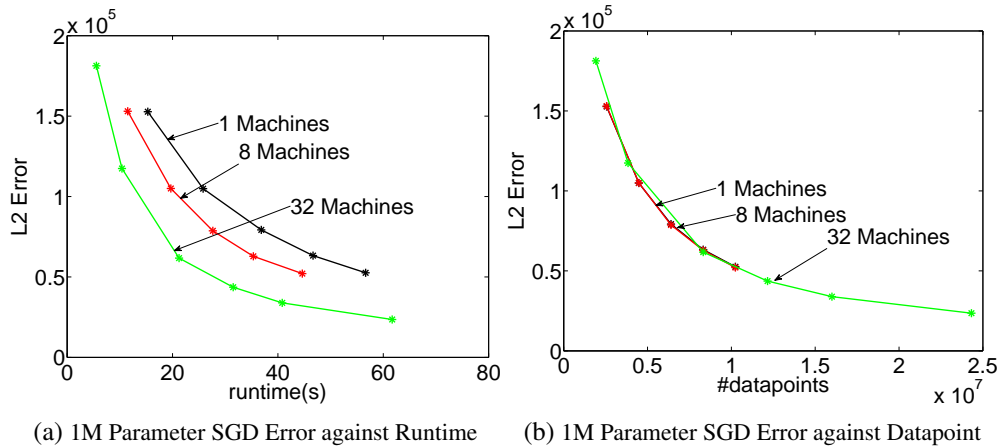


Figure 4.10: *1M parameter experiment*: (a) The L2 error between the learned parameters and the true parameters as a function of time, varying the number of machines. (b) The L2 error between the learned parameters and the true parameters as a function of number of data points encountered varying the number of machines.

used with the timestep t incrementing every 640,000 datapoints. We vary the number of machines from 1 to 32, evaluating the rate of convergence. In the single machine case since latency hiding is unnecessary, over-threading is not used. However, when more than one machine is used, each machine creates 10,000 threads to hide latency.

We plot in Fig. 4.10a, the L2 difference between the learned parameter vector and the ground truth parameter vector over time, varying the number of machines. We observe that with increased number of machines, we can achieve faster convergence. However, more surprisingly, when we plot the error against the total number of datapoints encountered (Fig. 4.10b), we find that the distributed SGD is also *data-efficient*. In other words, there is no difference in the final error between running the same set of datapoints on 1 machine, or on 32 machines.

To validate our intuition that the data-efficiency is due to the sparsity of the datapoints, we benchmarked the system again using a smaller parameter space of 100K, plotting the results in Fig. 4.11a and Fig. 4.11b. With a smaller parameter space, collisions where the same parameter is accessed and updated simultaneously by multiple threads are also more frequent, resulting in decreased data-efficiency. While at 40s, all cluster configurations were able to obtain nearly the same error, the 32 machine configuration required nearly 50% more datapoints to do so.

These experiments also demonstrate a performance bottleneck in the current WarpGraph DHT implementation. While we are able to demonstrate performance gains with increased number of machines, the effective speedup in number of datapoints per second from 1 machine to 32 machines is only a factor of 2. Initial profiling suggests that this is due to the DHTGather operation requiring an average of 250ms to complete. We believe this is due to the poor locking strategy used internally by the DHT and future optimizations should be able to improve both the throughput and the latency of the DHT implementation.

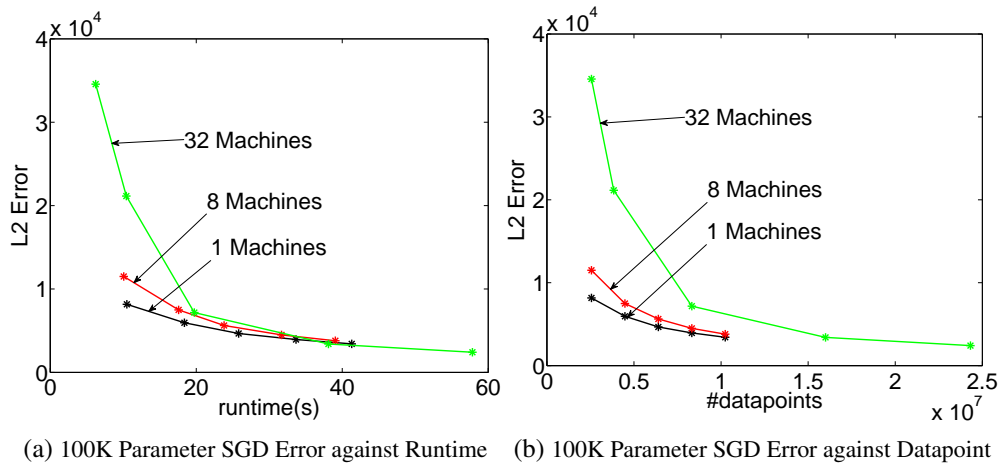


Figure 4.11: *100K parameter experiment*: (a) The L2 error between the learned parameters and the true parameters as a function of time, varying the number of machines. (b) The L2 error between the learned parameters and the true parameters as a function of number of data points encountered varying the number of machines.

4.5 Conclusion

The PowerGraph abstraction was designed specifically to maximize distributed performance and achieved it by restricting the abstraction to the inflexible **Gather-Apply-Scatter** vertex program, decreasing the ease of use of the system.

To improve usability, we introduce the WarpGraph abstraction which returns to the more intuitive update function model of GraphLab 1, but include **fine-grained data parallel primitives** such as “MapReduce over neighbors” which expose parallelism for high degree vertices (thus satisfying the PowerGraph design goals), and are much easier to understand. We also demonstrate that the WarpGraph abstraction can be implemented efficiently in the distributed setting by using multi-threading as a latency hiding technique; out-performing the asynchronous PowerGraph implementation substantially. Initial findings by toolkit developers also concur that the WarpGraph abstraction is both easier to understand, and easier to use.

A drawback of the WarpGraph abstraction, is that a synchronous execution is not feasible since that will entail starting as many threads as there are vertices in the graph. However, in our performance evaluation we demonstrated that our WarpGraph implementation was able to achieve nearly half the vertex program throughput as the highly optimized Synchronous PowerGraph implementation, making the need for a synchronous WarpGraph implementation less critical. Furthermore, a synchronous execution of WarpGraph can be emulated easily by creating two versions of every vertex/edge data, and implementing the update function to read from one version, but write to the other.

Our key contributions are:

- The WarpGraph abstraction (Sec. 4.2) which uses fine-grained data-parallel primitives to improve usability of PowerGraph while retaining PowerGraph’s primary performance characteristics.
- A distributed asynchronous implementation of the WarpGraph abstraction which uses multi-threading and coroutines to hide the latency of the distributed fine-grained data-parallel operations. (Sec. 4.3).

- An evaluation of the WarpGraph implementation, demonstrating that our WarpGraph design substantially out-perform the Asynchronous PowerGraph implementation, and can approach the performance of the Synchronous PowerGraph implementation. (Sec. 4.4)

Chapter 5

Related Work

There are several historical, or existing frameworks which can be used for parallel ML. We address the differences between GraphLab and these other abstractions here.

5.1 Map-Reduce Abstraction

A program implemented in the MapReduce framework consists of a `Map` operation and a `Reduce` operation. The `Map` operation is a function which is applied independently and in parallel to each datum (e.g., webpage) in a large data set (e.g., computing the word-count). The `Reduce` operation is an aggregation function which combines the `Map` outputs (e.g., computing the total word count). MapReduce performs optimally only when the algorithm is *embarrassingly parallel* and can be decomposed into a large number of independent computations. The MapReduce framework expresses the class of ML algorithms which fit the Statistical-Query model [Chu et al., 2007] as well as problems where feature extraction dominates the run-time.

The MapReduce abstraction fails when there are *computational dependencies* in the data. For example, MapReduce can be used to extract features from a massive collection of images but cannot represent computation that depends on small overlapping subsets of images. This limitation makes it difficult to represent algorithms that operate on structured models. As a consequence, when confronted with large scale problems, we often abandon rich structured models in favor of simpler methods that are amenable to the MapReduce abstraction.

Furthermore, many ML algorithms *iteratively* transform parameters during both learning and inference. For example, algorithms like Belief Propagation (BP), EM, gradient descent, and even Gibbs sampling, iteratively refine a set of parameters until some termination condition is achieved. While the MapReduce abstraction can be invoked iteratively, it does not provide a mechanism to directly encode iterative computation. As a consequence, it is not possible to express sophisticated scheduling, automatically assess termination, or even leverage basic data persistence.

The popular implementations of the MapReduce abstraction are targeted at large data-center applications and therefore optimized to address node-failure and disk-centric parallelism. The overhead associated with the fault-tolerant, disk-centric approach is unnecessarily costly when applied to the typical cluster and multi-core settings encountered in ML research. Nonetheless, MapReduce is used in small clusters

and even multi-core settings [Chu et al., 2007]. The GraphLab implementations do not address fault-tolerance or parallel disk access and instead assumes that processors do not fail and all data is stored in memory. While this provides a massive performance advantage, it limits the implementation to dataset sizes which can fit in distributed memory. The GraphLab abstraction however, does not prescribe nor require the in-memory model, and [Kyrola et al., 2012] has demonstrated that it is possible to implement GraphLab on disk without much loss in performance.

5.1.1 Relation to GraphLab

MapReduce may be used to execute one synchronous iteration of GraphLab 1 by mapping over every vertex and edge, and computing the entire update function in the Reducer. Iterative computation can then be defined through the use of either explicit MapReduce calls, or iterative MapReduce methods [Bu et al., 2010, Ekanayake et al., 2010]. Dynamic execution and asynchronous execution however, are difficult to translate.

PowerGraph’s Gather-Apply-Scatter model factorizes the computation over edges, and uses the vertex-separator partitioning to minimize communication. A similar strategy can also be used to optimize the MapReduce execution. Given an edge-list graph representation where each row of the input file contains the edge data on the edge as well as the vertex data of both endpoints, the Gather operation can be emulated through the use of Combiners. This allows MapReduce to achieve the communication reduction observed by PowerGraph. Finally, the Apply phase can be executed on the Reducers, and the Scatter phase can be executed with another MapReduce pass.

The WarpGraph model however, due to its generality is difficult to emulate in MapReduce. While WarpGraph update functions which perform the same sequence of operation on all vertices can be compiled into a fixed sequence of MapReduce operations, other cases such as conditional gathers (Fig. 4.4) where the choice of aggregation operation is data-dependent, are difficult to emulate.

The key performance limitation with translating GraphLab programs to MapReduce is due to the requirement that every Mapper is *independent*. To execute graph algorithms in MapReduce, thus leads to unnecessary replication of information, which in some cases can lead to an asymptotic increase in communication and storage requirements (Sec. 3.6.6).

In the reverse direction, MapReduce can be emulated efficiently in all GraphLab abstractions by treating the data graph as a dataflow graph. By constructing a fully connected bipartite graph with one set of vertices representing Mappers, the other set of vertices representing Reducers, and using the edges for communication, we can achieve a communication efficient emulation of MapReduce. The limitation however, lies in the GraphLab implementations which require all data to fit in memory.

5.2 BSP Abstraction (Pregel)

The question of how the GraphLab abstractions compare with the Pregel [Malewicz et al., 2010] abstraction is often raised since both systems emerged at nearly the same time as solutions for large graph computation¹. Due to the similarities in design goals between GraphLab and Pregel, we provide here a more exhaustive analysis of the abstraction differences.

¹A description of the Pregel abstraction is provided in Sec. 2.1.

5.2.1 Dynamic Execution

While Pregel also supports dynamic execution in that only vertices which receive messages will be executed; there is a crucial difference between how dynamic execution behaves in a Message Passing abstraction such as Pregel, and a (restricted) shared memory abstraction such as GraphLab.

In Pregel, when a vertex is executed, it is unable to read its neighboring vertex/edge data, and instead must rely upon its neighbors sending it the necessary state. For instance, the dynamic PageRank algorithm described in Alg. 2.1 cannot be easily implemented within Pregel; while you can signal a vertex to execute in the next round via a message, the signaled vertex is unable to recompute its own value since it cannot read its neighbors information.

5.2.2 Asynchronous Computation

The Gibbs Sampling [Geman and Geman, 1984] process is conceptually simple, each vertex in the graph represents a single variable in a probability distribution. Computation then proceeds in the sequential case by picking a vertex, and drawing a sampling from the distribution of the vertex condition on the values of its neighboring vertices. Unfortunately in the parallel case, performing this update on all vertices simultaneously will not converge to a meaningful answer [Gonzalez et al., 2011, Siapas, 1996]. Asynchronous computation with edge consistency is necessary to guarantee convergence; though empirically in many settings weaker consistency is sufficient.

All versions of GraphLab provide asynchronous computation capabilities, with optionally enforced edge consistency; implemented either through locking (Sec. 2.3.1, Sec. 2.4.2.2, Sec. 3.6.4, Sec. 4.3), or specific engine choices (Sec. 2.4.2.1).

On the other hand, Gibbs Sampling is difficult to express in Pregel since any given vertex cannot unilaterally choose to read the values on neighboring vertices. The asynchronous computation necessary is also difficult to express. While the chromatic engine could conceptually be implemented in the Pregel, a large amount of additional coordination is necessary.

5.2.3 Edge Data

The GraphLab abstractions include the notion of persistent and mutable edge data, which is useful in many settings:

1. To contain the “messages” in message-passing graphical model inference procedures such as belief propagation. (Sec. 2.3.3.1, Sec. 2.4.4.2)
2. In matrix factorization for collaborative filtering to store the current predicted rating for a user-movie pair. (Sec. 2.4.4.1)
3. In some graph algorithms such as triangle counting to maintain the number of triangles passing through each edge.
4. In Latent Dirichlet Allocation [Blei et al., 2003] where a graph is constructed between documents and words, to contain the sampled topics for each word occurrence.

Without edge data, Pregel must store the edge state on the vertices of the graph which incurs unnecessary replication in the common event that both ends of the edge needs access to the edge data.

5.2.4 Generalized Messaging

Pregel, while frequently described as being restricted to communication along edges of a graph, does technically permit messaging of arbitrary vertex programs from any other vertex program. The “graph” is therefore largely a user construct, and the abstraction permits a more general programming model similar to generalized BSP message passing [Skillicorn et al., 1997].

GraphLab however, enforces all communication patterns between vertex programs to be strictly limited to the graph structure. The only deviations permitted are through the Sync operation (Sec. 2.2.5) in GraphLab and PowerGraph, and the DHT (Sec. 4.2.2.4) in WarpGraph, both of which provide no guarantees of consistency nor recency of data.

5.2.5 Scatter vs Gather

GraphLab provides an efficient distributed representation of algorithms where update functions needs data from adjacent vertices. For instance, consider the alternative least squares (ALS) problem for Netflix matrix factorization. ALS in GraphLab involves simply representing the graph as a bipartite graph between users and movies. For each movie a user rents, an edge between the user and the corresponding movie is creating containing the user rating for the movie. ALS then proceeds in an asynchronous fashion: for each user, update the user vector to better predict the user ratings while holding all the movie vectors fixed. This is then repeated for each movie and the whole process loops until convergence is obtained.

To implement ALS in Hadoop, the Map-function does essentially no computational work and instead only emits *copies* of the vertex data for every edge in the graph. For example, a user vertex which connects to 100 movies must emit the data on the user vertex 100 times, once for each movie. This results in the generation of a large amount of unnecessary network traffic and unnecessary HDFS writes. This weakness extends beyond the MapReduce abstraction, but also affects the graph message-passing models (such as Pregel) due to the lack of a defined **scatter** operation which would avoid sending same value multiple times to each machine. Comparatively, the GraphLab abstractions are simpler as users do not need to explicitly define the flow of information. Synchronization of a modified vertex will only communicate the modifications to machines which require the vertex data for computation.

On the other hand, most other frameworks have very efficient **gather** operations. Piccolo [Power and Li, 2010], Pregel [Malewicz et al., 2010] provide efficient combiner operations which allow messages going to the same vertex to be “pre-combined” on the source machine, thus minimizing network communication. While GraphLab lacks such a gather operation, PowerGraph and WarpGraph resolves this limitation by explicitly specifying a gather operation.

PowerGraph and WarpGraph are, as far as the author is aware, the only high level abstraction which efficiently represents *both gather and scatter* capabilities.

5.2.6 Relation to GraphLab

GraphLab 1 can be emulated in Pregel by simply having each vertex broadcast its value to all neighbors providing every vertex with the complete scope information in the next super-step. PowerGraph's **gather** operation can be emulated through the use of message combiners, and the **scatter** operation can be emulated simply through the use of messages. However as mentioned in the subsections above, the scatter emulation is inefficient. Furthermore, Pregel cannot express the same dynamic execution (Sec. 5.2.1) and asynchronous (Sec. 5.2.2) execution semantics as GraphLab or PowerGraph.

Finally, just like MapReduce, while a restricted class of WarpGraph update functions which perform the same sequence of operation on all vertices can be compiled into Pregel, other cases such as conditional gathers (Fig. 4.4) where the choice of aggregation operation is data-dependent, are difficult to emulate; possibly requiring large state machines on each vertex.

Pregel however, can be emulated efficiently in all GraphLab abstractions by using the graph edges to store communicated messages. Furthermore, the PowerGraph and WarpGraph abstractions can further optimize Pregel programs by providing an efficient broadcast primitive.

5.3 Generalized SpMV Abstraction

The generalized sparse matrix vector product (SpMV) abstraction [Buluç and Gilbert, 2011, Kang et al., 2009] expresses computation as multiplication between large sparse matrices and dense vectors where the data types used in the matrix and vector entries are user-defined with arbitrary multiply and sum operations. This can be seen as a constrained version of the PowerGraph's Gather operation. Just like the BSP Abstraction, the SpMV abstraction cannot effectively expose dynamic, asynchronous computation, and only efficiently defines **gather** operations but not **scatter** operations.

5.3.1 Relation to GraphLab

The SpMV abstractions lack a notion of mutable edge data, thus can make GraphLab programs which require edge data difficult to express. Furthermore, the current SpMV abstractions lack both dynamic and asynchronous computation capabilities. On the other hand, since the SpMV abstraction is a special case of PowerGraph's Gather operation, it can be emulated in PowerGraph efficiently.

5.4 Dataflow Abstraction

In the Dataflow abstraction, parallel computation is represented as a directed acyclic graph with data flowing along edges between vertices. Vertices correspond to functions which receive information on inbound edges and output results to outbound edges. Implementations of this abstraction include Dryad [Isard et al., 2007], Pig Latin [Olston et al., 2008] and Spark [Zaharia et al., 2010].

While the DAG abstraction permits rich computational dependencies, it can only express synchronous iterative computation. Dynamic computation or asynchronous computation cannot be expressed.

5.4.1 Relation to GraphLab

The generality of the dataflow abstractions can permit efficient implementations of the GraphLab and PowerGraph execution models [Xin et al., 2013]. However, WarpGraph relies on fine-grained aggregation primitives and only a restricted subset of update functions can potentially be compiled into a coarse dataflow execution. Update functions which require conditional gathers (Fig. 4.4) where the choice of aggregation operation is data-dependent, are difficult to emulate.

Dataflow abstractions can be emulated efficiently in GraphLab by using the dataflow graph as the data graph, and using edges for communication. However the GraphLab abstractions do not, natively, provide the advantages of the dataflow abstractions such as fault-tolerance, streaming computation, and the use of the datasets larger than memory requirements. Building such capabilities into the GraphLab abstractions require much greater complexity.

5.5 Datalog/Bloom

Datalog is a powerful declarative logic programming language that has been extended by Bloom [Alvaro et al., 2011] to distributed environments, leading to a rich abstraction for distributed asynchronous systems. More recently, Datalog/Bloom has been shown to be useful for understanding certain classes of asynchronous fixed point computation [Conway et al., 2012], as well as incremental computation [McSherry et al., 2013].

Datalog/Bloom is unlike MapReduce, Pregel and other dataflow abstractions in that Datalog does not prescribe a specific execution strategy. The Datalog/Bloom programmer instead simply specifies relationships between *facts*, and it is up to the implementation/query optimizer to figure out the most efficient execution strategy. As a result, Datalog/Bloom has an interesting similarity with GraphLab where the user simply defines a vertex program describing the evolution of graph data, then different engine implementations/scheduler strategies may be used to execute the program to convergence. Understanding this relationship has great potential for interesting future work. Some intriguing possibilities include embedding Datalog/Bloom as a high-level domain specific language for GraphLab, or using some of GraphLab's large-graph partitioning strategies to accelerate program execution in Datalog/Bloom.

Chapter 6

Conclusion

6.1 Thesis Summary

As the people gets more interconnected through the Internet, and as data collection capabilities increase, we enter a new era of Big Data where companies collect and store more information than they can use. As a result, Machine Learning methods become increasingly important as a means of extracting and summarizing useful information from large quantities of data. Google, with its PageRank algorithm (and other proprietary ad-targeting algorithms), as well as Netflix with the success of the Netflix competition, have demonstrated the usefulness and importance of Machine Learning for tech companies.

To apply Machine Learning methods to massive datasets as frequently encountered in the real world, use of parallelism and large scale distributed computation is necessary. However, since programming for distributed systems is highly challenging, the use of high level abstractions such as MapReduce, Dryad and Pregel are often used, which hide much of the complexity. In this thesis, we identified that such high level abstractions, while suitable for a broad range of data processing tasks, do not fit the needs of Machine Learning algorithms. Specifically, in Sec. 1.1.1 we identified four key aspects:

Interdependent Computation Many ML algorithms do not treat data points as independent quantities, but can exploit relationships between data to extract deep and interesting signals.

Asynchronous Iterative Computation ML algorithms are iterative in nature, executing the same sequence of computation repeatedly. Furthermore many algorithms are asynchronous in nature; requiring the most recent parameter values for each computation.

Dynamic Computation Parameters may not converge uniformly in iterative ML algorithms. Focusing computation on difficult regions of the computation can provide performance gains.

Serializability/Sequential Consistency A serializable execution is beneficial for debugging, and certain algorithms such as Gibbs Sampling require it for correctness. However, as we suggested in Sec. 3.1.3, newer empirical results have suggested that ML algorithms may be tolerant to serializability-breaking execution.

Existing computation abstractions such as MapReduce and Pregel are unable to express the Dynamic, Asynchronous, Iterative computation common to many ML algorithms. While in some cases it is possible

to coerce an algorithm into a mismatched abstraction, a large efficiency penalty must be paid. As such, we propose a new abstraction called GraphLab, designed to explicitly expose these properties.

In Chap. 2, we defined the GraphLab 1 abstraction, which consists of a **data graph** where arbitrary data may be stored on vertices and edges; an **update function/vertex program** which describes local computation on vertices; and a **scheduler** which determines the order of update functions executed. Additionally, a **consistency model** is used to determine the extent at which computation may overlap and hence *race*. Finally, a **Sync** mechanism is used to provide global state when necessary. We implemented the abstraction in both the shared memory and distributed memory setting and demonstrated the expressiveness, usefulness and performance that can be obtained through the use of a domain specific abstraction.

While the distributed system built in Chap. 2 works well for regular graphs of over 100 million vertices and edges, it fails on real-world graphs with power-law degree distributions containing billions of vertices and edges. To scale to such problems, in Chap. 3 we introduced PowerGraph which restricts the general update function model in GraphLab 1 to the **GAS: Gather-Apply-Scatter** structure. The GAS model of the vertex program allowed individual vertex program to be parallelized, improving performance on high degree vertices; and also allowed the use of the vertex-separator model for distributed graph representation, bringing about large reductions in communication and storage requirements. We also introduced new streaming graph partitioning heuristics to improve graph partitioning quality. Our distributed implementation of PowerGraph achieved state of the art performance, and in some cases, improved over them by orders of magnitude. Our open source project at <http://graphlab.org> implements over 24 different algorithms and has been downloaded thousands of times.

PowerGraph sacrificed usability for performance; the behavior of the PowerGraph abstraction is difficult to understand, and the Gather-Apply-Scatter restriction makes certain computations difficult to represent, if not impossible. In Chap. 4, we introduce the WarpGraph abstraction to address the usability issue. Through the use of fine-grained data parallel abstractions which allow update functions to call aggregate operations over its neighborhood, we combine the power of PowerGraph, with the simplicity of GraphLab 1. Our implementation relied on the use of cooperative threading: to use thousands of threads to hide communication latency. We demonstrated that the WarpGraph abstraction is not just strictly more expressive than PowerGraph, but is also easier to use and understand. Our WarpGraph implementation outperforms the asynchronous implementations of PowerGraph and is competitive with the synchronous PowerGraph implementation on certain tasks, demonstrating the feasibility of asynchronous computation in the distributed setting.

We summarize the three abstractions we developed in Table 6.1.

6.2 Observations

We learned several key lessons through the development of this thesis.

Importance of Dynamic Computation and Computation Sparsity

As data set sizes increase, batch Machine Learning methods which require multiple complete passes over the entire dataset become impractical. the use of dynamic computation which focuses computation on difficult regions of the problem is necessary to maintain performance. Furthermore in the real-world, dataset sizes are not static, but grow incrementally. The ability to focus computation on only the affected

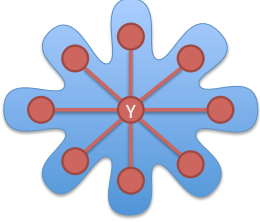

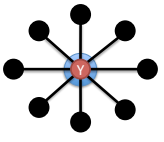
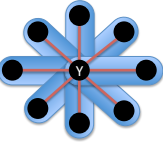
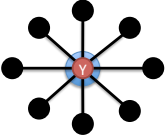

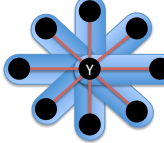
<p>GraphLab 1</p>		<p>Exposes entire scope of a vertex to an update function.</p>
<p>PowerGraph</p>	<div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>Gather</p>  <p>Parallel Sum $\downarrow + \downarrow + \dots + \downarrow \rightarrow \Delta$</p> </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>Apply</p>  <p>Apply the accumulated value to center vertex</p> </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>Scatter</p>  <p>Modify adjacent edges.</p> </div> </div>	<p>Restricts the GraphLab 1 abstraction for performance by factorizing the update function into three phases of Gather, Apply and Scatter.</p>
<p>WarpGraph</p>	<div style="display: flex; align-items: center;">  <div style="margin-left: 20px;"> <p>API</p> <div style="display: flex; justify-content: space-around;"> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>MapReduce Neighbors</p>  <p>Parallel Sum $\downarrow + \downarrow + \dots + \downarrow$</p> </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>Transform Neighbors</p>  <p>Modify adjacent edges.</p> </div> </div> </div> </div>	<p>Improves expressiveness by restricts update function to just the central vertex, but providing API functions to perform aggregate operations over the neighborhood.</p>

Table 6.1: Summary of the three abstractions developed in this thesis

parts of the model will allow the Machine Learning algorithm to achieve real-time performance; something which is increasingly relevant in the Web 2.0 era.

However, not all Machine Learning methods are amenable to dynamic computation. GraphLab focuses on a setting where all computation dependencies in the algorithm can be represented sparsely; in other words, to update a single parameter of the model only requires accessing a small amount of additional data. Through our experience developing this thesis, we find that the notion of *sparse computational dependencies* facilitates the development of dynamic methods. We observe that to develop a dynamic GraphLab algorithm simply requires a vertex-local convergence heuristic to be defined. i.e. If the current vertex value changes by greater than some ϵ , schedule all neighbors. Residual Belief Propagation [Elidan et al., 2006] and Wildfire Belief Propagation [Ranganathan et al., 2007] are examples of dynamic algorithms which are defined in this manner, and achieved substantial performance improvements over their original non-dynamic versions.

Abstraction Mismatch incurs Large Performance Penalties

The growth in popularity of Hadoop has driven many Machine Learning researchers to design new algorithms targeted specifically for the MapReduce architecture. However, we believe that designing algorithms while constraining the choice of abstraction, do not lead to well performing algorithms. As we observed in Sec. 3.6.6, an abstraction mismatch can incur unbearably large performance penalties.

Instead, we propose the use of the alternate strategy: to understand the needs of the Machine Learning task in mind, then seek for an abstraction which best matches those needs. As this thesis demonstrates, the use of the right abstraction for your algorithm can provide massive performance gains.

The GraphLab thesis grew out of a similar process. After working on parallel and distributed Belief Propagation (BP) methods [Gonzalez et al., 2009a,b], we isolated the core computation abstraction needs of BP and discovered that it is sufficiently general to be used for a broad range of ML algorithms. The outcome is a high level domain specific abstraction which can be applied to many ML tasks, and does not sacrifice performance.

Every Few Orders of Magnitude of Scaling Requires a Redesign

This statement is a useful design principle to consider when scaling new algorithms, implementations of ML procedures, or computation abstractions to larger real-world datasets. System designs and algorithms which work on smaller datasets may completely fail when operating on datasets an order of magnitude larger; requiring a fundamental rethink of the entire system.

We learned this lesson when attempting to scale GraphLab to real world graphs of billions of vertices and edges. We discovered that the vertex-ghosting procedure does not scale and resulted in excessive replication; and the difficulty of partitioning such large graphs exacerbated the situation. To resolve this issue required a fundamental redesign of the GraphLab computation model which resulted in the PowerGraph abstraction.

6.3 Future Work

Through the development of this thesis, we encountered several interesting research questions.

6.3.1 Graph Partitioning

Extensive research has been done on computation of high quality edge separators on graphs, but comparatively little work exist for vertex separators. While the streaming partitioning process described in Chap. 3 performed reasonably well, more research into improved streaming vertex separator algorithms will help improve performance of the GraphLab platform dramatically.

6.3.2 Incremental Machine Learning

We would like to consider the situation where GraphLab is used to maintain convergence of an optimal solution to some optimization procedure while the input graph is under continuous modification. For instance in the Netflix matrix factorization problem, we could use GraphLab to maintain a solution to the factorization while new users, new movies, or new user-movie rental pairs are continuously inserted.

The setting bears resemblance to online machine learning [Blum, 1996], but a key difference is that the algorithm is provided access to all existing data and not just one data element at a time. The setting bears much closer resemblance to **incremental optimization** as defined in Hartline [2008] (and hence we give it the same name) though Hartline [2008] analyzed the setting only for particular discrete optimization problems such as max-flow and matching. However, key differences still exist in that we allow arbitrary changes to be made to existing data including modifications of existing data points, and removal of data. Still more related are **warm-start** optimization techniques such as in Yildirim and Wright [2002] and Gondzio and Grothey [2008], where an optimization problem is solved using the solution of a similar or perturbed problem for initialization.

Such a learning system can be built into GraphLab easily by permitting GraphLab to load graphs from external databases. GraphLab is first used to provide a solution to some initial graph. Following which, as data is added or modified, GraphLab's dynamic scheduling capabilities are used to update the minimal set of vertices to maintain convergence. The key idea is therefore to **warm-start** a solution to a modified problem, using a solution to the original problem.

A theoretical understanding of how solutions to optimization problems move as data is modified, as well an understanding of how dynamic scheduling behaves in these problems could lead to efficient methods to solve with high accuracy, extremely large optimization problems.

6.3.3 Dynamic Execution

As mentioned above in Sec. 6.2, GraphLab provides dynamic execution capabilities which will allow the algorithm to focus computation on the hardest parts of the problem. The potential performance benefits that can be gained from the use of dynamic execution is very large.

However, most Machine Learning algorithms are still defined in the strict fully iterative sense: where the same sequence of operations are executed every round. Additional research has to be done in this area to understand how to take advantage of dynamic scheduling in Machine Learning algorithms.

6.3.4 Fault Tolerance

We have empirically observed that some iterative Machine Learning algorithms such as ALS, appear to be strongly fault tolerant by nature; that upon machine failure and losing a fraction of the learned parameters, simply resetting the lost parameters to reasonable values and resuming the algorithm is sufficient. Only a few additional iterations are needed to bring the error rate to pre-machine-failure values.

We believe that this behavior is related to the warm-start property in many optimization procedures and therefore ties in strongly with the Incremental Machine Learning research direction described above. A better understanding of the inherent fault tolerance properties of Machine Learning algorithms will provide great benefit to the use of Machine Learning on massive datasets in very large cluster deployments of thousands of machines.

6.3.5 Other Abstractions

GraphLab is designed to operate well on a reasonably broad family of Machine Learning algorithms; covering a space what we categorize as **Sparse Iterative** methods. However, the breadth of Machine Learning far exceeds the domain of tasks GraphLab is designed for. GraphLab is not particularly suitable for:

Dense Iterative Unlike Sparse Iterative methods, Dense Iterative methods involve the manipulation of large dense matrices. Kernel methods, even including many low-rank versions fall into this category.

Iterative ParFor This is a generalized form of iterative MapReduce, allowing arbitrary read-only memory sharing between parallel threads. This describes classes of optimization procedures such as LBFGS, Batch Gradient Descent, FISTA, etc. AllReduce [all] is an example of an abstraction for this category of problems.

Online Learning This describes methods based on streaming data which encounter each datapoint exactly once. SGD and Bandit algorithms are examples of algorithms in this category. This category also include streaming and sketching procedures such as the well known Count-Min Sketch for approximate statistics [Cormode and Muthukrishnan, 2005].

Instance Based Learning This describes procedures such as K-NN, and top-K-inner product which have a simple learning process, but have a complex test time query process.

New models, new abstractions and new algorithms are needed to scale these classes of problems to the Big Data setting. A promising new abstraction which emerged in recent years is the Parameter Server which we will describe in greater detail.

6.3.6 Parameter Server

The Parameter Server model is a form of a weak consistency distributed shared memory architecture. A Distributed Hash Table is used to maintain global parameter information, and machines read and write to the Distributed Hash Table (DHT) directly in parallel. To obtain performance, several techniques are used including the use of local caching, or requiring that the hash table values form Abelian Group.

First explored in Smola and Narayanamurthy [2010] and generalized in Ahmed et al. [2012], the parameter server model has found applications in Gibbs Sampling [Ahmed et al., 2011, Low et al., 2011, Smola and Narayanamurthy, 2010], Stochastic Gradient Descent (Sec. 4.4.3), and could potentially be applied for other optimization procedures as well.

While the notion of distributed shared memory is not new, the novelty in the parameter server has been to specialize the weak consistency model for the purposes of Machine Learning. For instance, [Cipar et al., 2013] defined cache staleness to be dependent on the iteration counter of the ML algorithm. Other possible staleness conditions are conceivable: based on the frequency of modification of the parameter or the amount of change in the parameter. Or even in Sec. 4.4.3 which does not use a cache but instead emulates massive parallelism to hide the latency of distributed memory.

The parameter server model is convenient since it is highly general, being functionally similar to distributed shared memory. However, more theoretical and empirical work has to be done to understand the space of algorithms for which the parameter server model is useful; and among the many possible variations of the parameter server, which is the best.

Appendix A

Extended Chandy Misra

The Dining Philosopher’s problem provides a convenient analogy for the locking procedure needed for GraphLab’s “edge consistency” model (Sec. 2.2.4). Let each edge in the graph represent a fork (we ignore edge direction), and let each vertex represent a philosopher. Since Each philosopher must pick up all forks on all adjacent edges before he can eat, therefore adjacent philosophers cannot eat simultaneously; satisfying the requirements of Edge Consistency. As a result, any solution to the Dining Philosopher’s problem, also describes a solution for enforcing edge consistency.

The locking procedure used by the shared memory and distributed memory implementation of GraphLab in Chap. 2, corresponds exactly to the Dijkstra solution described in Dijkstra [1971]. However, for large graphs with high degree vertices, the requirement to acquire locks sequentially, limits performance dramatically. Furthermore, the overhead of “fair mutex” datastructures, where a queue has to be maintained per mutex increases memory load substantially in the distributed setting.

Instead, for PowerGraph (Chap. 3), we consider an alternate method based on the the Chandy Misra solution [Chandy and Misra, 1984].

A.1 Chandy Misra

We will begin by describing the classical Chandy Misra solution to the Dining Philosopher’s problem.

Each Philosopher cycles through the following 3 states.

Thinking Philosopher is inactive. After some time, he may get hungry.

Hungry Philosopher wants to eat and is trying to acquire all adjacent forks. Once all forks are acquired, he begins Eating.

Eating Philosopher has all forks and is eating. After some finite amount of time, he stops eating and goes back to Thinking.

Each edge represents a fork, and thus can only be used by the two philosophers on its endpoints. The fork is always owned by one of its adjacent philosophers and must be in one of two states:

Clean The philosopher currently owning the fork keeps the fork even if others request for it. The fork becomes after the philosopher eats.

Dirty The philosopher currently owning the fork gives up the fork if others request for it. The fork is cleaned when it changes hands.

At initialization all forks are initially dirty, and are owned by the philosopher with the lower ID.

The Chandy Misra algorithm is described by the following pseudocode.

When philosopher v becomes hungry:

Send request for each unowned fork adjacent to v

When philosopher v receives request for fork f :

```
if  $v$  is Eating or  $f$  is clean
    %  $v$  is using the fork. Don't give it up.
    Store request
elseif  $v$  is Thinking
    %  $v$  is not using the fork. Give it up.
    Clean Fork  $f$  and give up fork
elseif  $v$  is Hungry and  $f$  is dirty
    %  $v$  wants the fork, but the neighbor has priority since fork is dirty
    % Give it up, but also request for it so I get it back
    Clean Fork  $f$  and give up fork
    Send request for fork  $f$ 
end
```

When philosopher v has all needed forks and is hungry:

```
%  $v$  got all the forks. He can eat.
Change state of  $v$  to Eating
```

When philosopher v Stops Eating:

```
% Finished eating, so all forks are dirty.
Change state of  $v$  to Thinking.
Set all forks to be Dirty.
% Give forks up to everyone who asked for it.
for each stored request:
    Clean the requested fork and give it up
end
```

Surprisingly, the above procedure will never deadlock and is fair if the initialization of forks is done correctly. We refer to the original paper [Chandy and Misra, 1984] for the proof.

A.2 Hierarchical Chandy Misra

While the Chandy Misra algorithm highly natural for the distributed setting since the algorithm is described in terms of “messages” between philosophers, it only works for the distributed edge-separator setting where the partitioning is defined by placing philosophers on machines.

In the vertex-separator setting (Chap. 3), the partitioning is defined by placing edges (forks) on machines, while vertices (philosophers) may span multiple machines. Modifications have to be made to the algorithm to support this case.

Each machine instantiates a local graph describing its assigned partition of the graph. In a vertex-separator each vertex may span multiple machines; we use the term **replicas** to denote the instances of each vertex. We designate one arbitrary replica of each vertex as the **master**.

For vertex v to start eating requires all forks adjacent to each replica of v to be acquired. To describe this coordination, we introduce one additional state to each replica.

Hors_Doeuvre If a replica of vertex v is in this state, it means that the replica has locally acquired all forks. However, other replicas of v may not have acquired all their forks.

Intuitively, the extension is simple. When a vertex/philosopher becomes hungry, all replicas of the vertex also becomes hungry. However, when the replicas acquire all forks locally, they do not start eating, but only start on appetizers (the Hors_Doeuvre state). In the meantime, the master of the vertex counts the number of replicas in the Hors_Doeuvre state, and only tells everyone to start eating when all replicas are in Hors_Doeuvre. A little additional care is needed since a replica in Hors_Doeuvre, may be holding dirty forks and thus neighboring vertices are allowed to take forks from it.

Each master vertex v has two variables $counter(v) = 0$ and $cancel_sent(v) = false$. $replica(v)$ denotes the set of replicas of v . $M(v)$ denotes the master of vertex v .

Master v becomes hungry:

```
% broadcast to all replicas.
broadcast make\_hungry(q) to each q in replicas(v)
```

Master v stops eating:

```
% broadcast to all replicas.
counter(v) = 0
broadcast stop\_eating(q) to each q in replicas(v)
```

Replica v receive make_hungry(v):

```
Send request for each unowned fork adjacent to v
```

When Replica v Obtains all Forks:

```
Switch v to Hors_Doeuvre
% Inform master that that we are ready
Send ready(v) message to M(v)
```

When Replica v receives request for fork f :

```
% Protocol is the same as the original Chandy Misra
% But we need to also handle the Hors_Doeuvre state
if v is Eating or f is clean
    % v is using the fork. Don't give it up.
    Store request
elseif v is Thinking
    % v is not using the fork. Give it up.
    Clean Fork and give up fork
elseif (v is Hungry and f is dirty)
    % v wants the fork, but the neighbor has priority since fork is dirty
    % Give it up, but also request for it so I get it back
    Clean Fork f and give up fork
    Send request for f
elseif v is Hors_Doeuvre and f is dirty
    % Tell the master that this replica is trying to leave Hors_Doeuvre
    store the request
    if cancel_sent(v) == false
```

```

    send cancel(M(v), v) message to M(v)
    cancelsent(v) = true
end
end

```

When Master v receives ready(v):

```

% Count the number of replicas which are ready
counter(v) = counter(v) + 1
if counter(v) = #replicas(v)
    % If all replicas are ready, we switch to eating
    Broadcast start_eating(q) to each q in replicas(v)
end

```

When Master v receives cancel(v, q):

```

% If we are not yet eating, and
% if a replica is leaving Hors Douevre, we cancel
if counter(v) > 0 && counter(v) < |R(v)|
    counter(v) = counter(v) - 1
    Send cancel_accept(q) to q
end

```

When Replica v receives start_eating(v):

```

% Master has counted the right number of Hors Douevre states.
% We are now in the eating state.
cancelsent(v) = false
Change state of v to Eating

```

When Replica v receives cancel_accept(v):

```

for requested fork f:
    if fork f is dirty
        clean f and give it up clearing the request
    end
end
cancelsent(v) = false
Change state of v to Hungry

```

When Replica v receives stop_eating(v) :

```

Set state of v to Thinking.
% Finished eating, so all forks are dirty.
Set all forks to be Dirty.
% Give forks up to everyone who asked for it.
for each request fork f:
    clean f and give it up clearing the request
end

```

Appendix B

Algorithm Examples

In this appendix, we provide pseudo-code examples of five algorithms as implemented in all GraphLab, PowerGraph and WarpGraph abstractions. The five algorithms are PageRank, Connected Component, Triangle Counting, Collaborative Filtering and Belief Propagation, and are presented in order of increasing complexity. In these examples, we do not discuss or implement optimizations, but simply focus on expressing the core kernel of the computation.

B.1 PageRank

The PageRank algorithm computes a PageRank value for each vertex in a directed graph. The PageRank value is computed by iteratively computing for each vertex in the graph,

$$\mathbf{R}(v) = \alpha + (1 - \alpha) \sum_{u \text{ links to } v} \mathbf{R}(u) / \mathbf{OutDegree}(u) \quad (\text{B.1.1})$$

in terms of the ranks of those pages that link to v , some probability α of randomly jumping to that page. The PageRank algorithm iterates the equation above until the individual PageRank values converge (change by less than some small ϵ). We will let $\alpha = 0.15$ in this example.

GraphLab

```
PageRankUpdate ( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) {
  prev_pagerank =  $D_v$ .rank

  // compute the sum, iterate over all in neighbors
  new_pagerank = 0.15
  ForEach( $u \in \text{InNeighbors}(v)$ ) {
    new_pagerank = new_pagerank + 0.85  $\times D_u$ .rank / OutDegree( $u$ )
  }
   $D_v$ .rank = new_pagerank

  // Reschedule if PageRank changes sufficiently
  if(|new_pagerank - prev_pagerank| >  $\epsilon$ ) {
    ForEach( $u \in \text{OutNeighbors}(v)$ ) {
      Schedule( $u$ )
    }
  }
}
```


PowerGraph

```
// gather_nbrs: in neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_u$ .rank / OutDegree( $u$ )
}

sum( $a$ ,  $b$ ) {
    return  $a + b$ 
}

apply( $D_v$ , acc) {
    // acc contains the result of the gather. Compute new PageRank
    new_pagerank = 0.15 + 0.85 × acc
    // store the change in the vertex to pass to the scatter
     $D_v$ .lastchange = new_pagerank -  $D_v$ .rank

     $D_v$ .rank = new_pagerank
}

// scatter_nbrs: out neighbors
scatter( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    // Reschedule if PageRank changes sufficiently
    if(| $D_v$ .delta|>ε) {
        Schedule( $u$ )
    }
}
```

WarpGraph

```
PageRankUpdate( $D_v$ ) {
    prev_pagerank =  $D_v$ .rank
    // compute a map reduce over the neighborhood selecting only the in edges
    new_pagerank = 0.15 + 0.85 *
        map_reduce_neighbors(pagerank_map, pagerank_combine, IN_EDGES)
     $D_v$ .rank = new_pagerank

    if (|new_pagerank - prev_pagerank| > ε) {
        broadcast(OUT_EDGES)
    }
}

// helper functions
pagerank_map( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_u$ .rank / OutDegree( $u$ )
}

pagerank_combine( $a$ ,  $b$ ) {
    return  $a + b$ 
}
```

B.2 Connected Component

In this task, we solve the Weakly Connected Component (WCC) problem using the GraphLab abstractions. The algorithm we will implement is not asymptotically optimal, but is quite efficient in practice for relatively low diameter graphs. The algorithm begins by initializing each vertex to a unique component ID. Each vertex then takes on the maximum value of its neighbors' component IDs:

$$\mathbf{ComponentID}(v) = \max_{u \in \mathbf{N}[v]} \mathbf{ComponentID}(u)$$

This repeats until all vertices stop changing. At convergence, all vertices with the same component ID are in the same component.

GraphLab

```
WCCUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) {  
  // compute the largest ID among me and my neighbors  
  newid =  $D_v$ .componentid  
  ForEach( $u \in \mathbf{N}[v]$ ) {  
    newid = max(newid,  $D_u$ .componentid)  
  }  
  // If my component changes, schedule neighbors  
  if (newid !=  $D_v$ .componentid) {  
     $D_v$ .componentid = newid  
    ForEach( $u \in \mathbf{N}[v]$ ) {  
      Schedule( $u$ )  
    }  
  }  
}
```

PowerGraph

```
// Gather computes the maximum component ID among neighbors
// gather_nbrs: all neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_u$ .componentid
}

sum(a, b) {
    return max(a, b)
}

apply( $D_v$ , acc) {
    newid = max(newid,  $D_v$ .componentid)
    // remember if I changed
    if (newid !=  $D_v$ .componentid) {
         $D_v$ .changed = 1
    }
    else {
         $D_v$ .changed = 0
    }
     $D_v$ .componentid = newid
}

// scatter_nbrs: all neighbors
scatter( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    // If my component changes, schedule neighbors
    if( $D_v$ .changed) {
        Schedule( $u$ )
    }
}
}
```

WarpGraph

```
WCCUpdate( $D_v$ ) {
    newid = max( $D_v$ .componentid,
               map_reduce_neighbors(wcc_map, wcc_combine, ALL_EDGES)

    if (newid !=  $D_v$ .componentid) {
         $D_v$ .componentid = newid
        broadcast(ALL_EDGES)
    }
}

// helper functions
wcc_map( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_u$ .componentid
}

wcc_combine(a, b) {
    return max(a, b)
}
```

B.3 Triangle Counting

In this task, we count the total number of triangles in an undirected graph. We implement the edge-iterator algorithm [Schank and Wagner, 2005] which iterates over every edge of the graph computing the intersection size of the adjacency list of both incident vertices.

```
count = 0
ForEach((i, j) ∈ E) {
  count = count + |N[i] ∩ N[j]|
}
// every triangle will be counted 3 times.
count = count / 3
```

To implement this procedure in the GraphLab abstractions require us to first list the neighborhood for each vertex, then compute the set intersection over all edges. We represent the undirected input graph as a directed data graph in GraphLab by arbitrarily directing each edge. At the end of the GraphLab computation, each edge will contain a count of the number of triangles passing through the edge. An additional sum over all edges will complete the count.

GraphLab

```
// two update functions are needed.
// First Update: Store the neighborhood list on each vertex
// Second Update: compute the intersections on each edge
StoreAdjUpdate(Dv, D*, D*→v, Dv→* ∈ Sv) {
  Dv.neighbors = AllNeighbors(v)
}
TriangleCountUpdate(Dv, D*, D*→v, Dv→* ∈ Sv) {
  // pick only the in-neighbors to ensure that every edge
  // is only processed once.
  ForEach(u ∈ InNeighbors(v)) {
    Du-v.count = Dv.neighbors ∩ Du.neighbors
  }
}
```

PowerGraph

```
// Only one vertex is needed using PowerGraph.
// On gather, we get the neighborhood list on each vertex
// On scatter, we compute the intersections on each edge
// gather_nbrs: all neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return {u} // a set of 1 element
}
sum(a, b) {
    return a  $\cup$  b // to combine gathers we take set unions
}

apply( $D_v$ , acc) {
     $D_v$ .neighbors = acc
}

// scatter_nbrs: in neighbors
// pick only the in-neighbors to ensure that every edge
// is only processed once.
scatter( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
     $D_{u-v}$ .count =  $D_v$ .neighbors  $\cap$   $D_u$ .neighbors
}
```

WarpGraph

```
// two update functions are needed.
// First Update: Store the neighborhood list on each vertex
// Second Update: compute the intersections on each edge
StoreAdjUpdate( $D_v$ ) {
     $D_v$ .neighbors = map_reduce_neighbors(neighborhood_map,
                                        neighborhood_combine, ALL_EDGES)
}
TriangleCountUpdate( $D_v$ ) {
    edge_transform(triangle_count_transform, IN_EDGES)
}

// helper functions
neighborhood_map( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return {u}
}

neighborhood_combine(a, b) {
    return a  $\cup$  b
}

triangle_count_transform( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
     $D_{u-v}$ .count =  $D_v$ .neighbors  $\cap$   $D_u$ .neighbors
}
```

B.4 Collaborative Filtering

In this example, we consider the task of collaborative filtering using Alternating Least Squares (ALS) matrix factorization. In this task, a bipartite graph is provided connecting users and products where an edge connects a user to a product if the user has rated the product. We let $r_{u,p}$ denote the rating user u assigns to product p . The goal of the task is to estimate a latent feature vector $f \in \mathbb{R}^d$ for each user and product such that:

$$r_{u,p} \approx f_u^T f_p$$

Given the rating information, we can solve for all latent feature vectors by minimizing the least squares error:

$$L(f) = \sum_{(u,p) \in E} (r_{u,p} - f_u^T f_p)^2$$
$$f = \arg \min_f L(f).$$

While the above loss function is not convex in f , we can still optimize the function by performing block coordinate descent on each individual user/product latent feature vector. Holding all other feature vectors constant, a single user's latent feature vector can be solved in closed form:

$$f_u = (X^T X)^+ X^T y$$

where y is a vector of ratings made by user u , and X is the matrix formed by stacking the feature vectors of the products rated by user u . $(X^T X)^+$ denotes the pseudo-inverse of the matrix $X^T X$.

The update equation for the product feature vectors are symmetric:

$$f_p = (X^T X)^+ X^T y$$

where y is a vector of ratings of product p , and X is the matrix formed by stacking the feature vectors of the users who rated product p .

This coordinate descent procedure repeats iteratively over every user and product until convergence is reached. We define convergence as every feature vector changing by less than ϵ^1 . Furthermore, since the computation is symmetric for both users and products, the implementations below need not differentiate between users and products.

Each edge on the data graph has a **rating** field:

D_{u-v} .**rating** The rating between vertex u and vertex v . By construction, one of u and v must represent a user, and the other, a product.

And each vertex on the data graph has a **feature** field:

D_v .**feature** A feature vector of length d initialized to arbitrary values.

¹As far as we are aware, there is no work on dynamic scheduling procedures for ALS. We use a simplistic convergence criterion here based simply upon the amount of change in the feature vector, but more complex criteria can be defined.

GraphLab

```
ALSUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) {
  // loop through neighbors, computing  $X^T X$  and  $Xy$ 

   $XTX =$  zero matrix of size  $d \times d$ 
   $Xy =$  zero vector of size  $d \times 1$ 

  // iterate over all neighbors computing  $XTX$  and  $Xy$ 
  ForEach( $u \in \mathbf{N}[v]$ ) {
    // Where  $\otimes$  denotes the outer product
     $XTX = XTX + D_u.feature \otimes D_u.feature$ 
     $Xy = Xy + D_u.feature \times D_{u-v}.rating$ 
  }
  // solve the linear system  $(XTX) f = (Xy)$  returning  $f$ 
   $new\_feature = solve(XTX, Xy)$ 
   $change\_in\_feature = |D_v.feature - new\_feature|_1$ 
   $D_v.feature = new\_feature$ 

  // Schedule Neighbors if my value changes by a large amount
  if ( $change\_in\_feature > \epsilon$ ) {
    ForEach( $u \in \mathbf{N}[v]$ ) {
      Schedule( $u$ )
    }
  }
}
```

PowerGraph

```
// gather computes  $X^T X$  and  $Xy$ 
// gather_nbrs: All neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
   $XTX = XTX + D_u.feature \otimes D_u.feature$ 
   $Xy = Xy + D_u.feature \times D_{u-v}.rating$ 
  // return a pair of values
  return pair( $XTX$ ,  $Xy$ )
}
sum((a1,b1), (a2,b2)) {
  // a1 and a2 corresponds to the contributions to  $X^T X$ 
  // b1 and b2 corresponds to the contributions to  $Xy$ 
  return pair(a1 + a2, b1 + b2)
}

apply( $D_v$ , acc) {
  // acc is the result of the gather
   $XTX = acc.first$ 
   $Xy = acc.second$ 
  // solve the linear system  $(XTX) f = (Xy)$  returning f
   $new\_feature = solve(XTX, Xy)$ 
   $D_v.change\_in\_feature = |D_v.feature - new\_feature|_1$ 
   $D_v.feature = new\_feature$ 
}

// scatter computes new messages on each edge
// scatter_nbrs: all neighbors
scatter( $D_u, D_{u-v}, D_v$ ) {
  // Schedule Neighbors if my value changes by a large amount
  if ( $D_v.change\_in\_feature > \epsilon$ ) {
    Schedule(u)
  }
}
}
```


WarpGraph

```
ALSUpdate( $D_v$ ) {
  (XTX, Xy) = map_reduce_neighbors(als_map, als_combine, ALL_EDGES)

  // solve the linear system (XTX) f = (Xy) returning f
  new_feature = solve(XTX, Xy)

  change_in_feature = | $D_v$ .feature - new_feature|1
   $D_v$ .feature = new_feature

  if (change_in_feature >  $\epsilon$ ) {
    broadcast(ALL_EDGES)
  }
}

// helper functions
als_map( $D_u, D_{u-v}, D_v$ ) {
  XTX = XTX +  $D_u$ .feature  $\otimes$   $D_u$ .feature
  Xy = Xy +  $D_u$ .feature  $\times$   $D_{u-v}$ .rating
  // return a pair of values
  return pair(XTX, Xy)
}

als_combine((a1,b1), (a2,b2)) {
  // a1 and a2 corresponds to the contributions to  $X^T X$ 
  // b1 and b2 corresponds to the contributions to  $Xy$ 
  return pair(a1 + a2, b1 + b2)
}
```

B.5 Belief Propagation

The Belief Propagation algorithm [Yedidia et al., 2001] is a commonly used approximate graphical model inference procedure. While the technique can be applied to more general models, we consider the simplified case of discrete pairwise Markov Random Fields (MRF) in this example.

Given a discrete probability distribution of the form

$$P(x_1, x_2, \dots, x_n) \propto \prod_{v \in V} \phi_v(x_v) \prod_{(u,v) \in E} \phi_{u,v}(x_u, x_v),$$

The belief propagation algorithm recursively defines a message $m_{u,v}(x_v)$ in each edge direction as well as a belief on each vertex $b_v(x_v)$.

$$b_v(x_v) \propto \phi_v(x_v) \prod_{u \in \mathbf{N}[v]} m_{u,v}(x_v)$$

$$m_{u,v}(x_v) \propto \sum_{x_u} b_u(x_u) * \phi_{u,v}(x_u, x_v) / m_{v,u}(x_u).$$

The above equations are then iteratively recomputed until all beliefs converge (norm change by less than ϵ). At convergence, the belief on each vertex is the estimated marginal distribution of the variable.

Since the Markov Random Field is undirected, we represent this in the directed data graph by arbitrarily directing each edge. Each edge (u, v) therefore contains two messages, $m_{u,v}$ and $m_{v,u}$, one in each edge direction.

Each edge on the data graph therefore has the following fields:

D_{u-v} .**potential** The potential $\phi_{u,v}$ on the edge

D_{u-v} .**message(v)** The message $m_{u,v}$ going towards v

D_{u-v} .**message(u)** The message $m_{v,u}$ going towards u

And each vertex on the data graph has the following fields:

D_v .**potential** The potential ϕ_v on the vertex

D_v .**belief** The estimated belief of the vertex b_v

GraphLab

```
BPUpdate( $D_v, D_*, D_{* \rightarrow v}, D_{v \rightarrow *}$   $\in \mathcal{S}_v$ ) {
  // compute the new belief, iterate over all neighbors
  // selecting the message entering the current vertex
  new_belief =  $D_v$ .potential
  ForEach( $u \in \mathbf{N}[v]$ ) {
    new_belief = new_belief .*  $D_{u-v}$ .message( $v$ ) // element-wise multiplication
  }
  new_belief = normalize(new_belief)
  change_in_belief =  $|D_v$ .belief - new_belief|1
   $D_v$ .belief = new_belief

  // compute new messages on each edge
  ForEach( $u \in \mathbf{N}[v]$ ) {
    new_message = 1 // vector of all ones
    ForEach(assignment  $x_u$ ) {
      ForEach(assignment  $x_v$ ) {
        new_message( $x_u$ ) +=  $D_v$ .belief( $x_v$ )  $\times$   $D_{u-v}$ .potential( $x_u, x_v$ )
          /  $D_{u-v}$ .message( $v$ )( $x_v$ )
      }
    }
    new_message = normalize(new_message)
     $D_{u-v}$ .message( $u$ ) = new_message
    if (change_in_belief >  $\epsilon$ ) {
      Schedule( $u$ )
    }
  }
}
```

PowerGraph

```
// gather computes the product of messages for the belief
// gather_nbrs: All neighbors
gather( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    return  $D_{u-v}$ .message(v)
}
sum(a, b) {
    return a .* b // element-wise multiplication
}

apply( $D_v$ , acc) {
    new_belief =  $D_v$ .potential .* acc
    new_belief = normalize(new_belief)
     $D_v$ .change_in_belief = | $D_v$ .belief - new_belief|1
     $D_v$ .belief = new_belief
}

// scatter computes new messages on each edge
// scatter_nbrs: all neighbors
scatter( $D_u$ ,  $D_{u-v}$ ,  $D_v$ ) {
    new_message = 1 // vector of all ones
    ForEach(assignment  $x_u$ ) {
        ForEach(assignment  $x_v$ ) {
            new_message( $x_u$ ) +=  $D_v$ .belief( $x_v$ ) ×  $D_{u-v}$ .potential( $x_u, x_v$ )
                               /  $D_{u-v}$ .message(v)( $x_v$ )
        }
    }
    new_message = normalize(new_message)
     $D_{u-v}$ .message(u) = new_message
    if ( $D_v$ .change_in_belief >  $\epsilon$ ) {
        Schedule(u)
    }
}
```

WarpGraph

```
BPUpdate( $D_v$ ) {
  new_belief =  $D_v$ .potential .* map_reduce_neighbors(belief_map,
                                                    belief_combine, ALL_EDGES)
  new_belief = normalize(new_belief)

  change_in_belief = | $D_v$ .belief - new_belief|1
   $D_v$ .belief = new_belief

  edge_transform(compute_new_message, ALL_EDGES);
  if (change_in_belief >  $\epsilon$ ) {
    broadcast(ALL_EDGES)
  }
}

// helper functions
belief_map( $D_u, D_{u-v}, D_v$ ) {
  return  $D_{u-v}$ .message(v)
}

belief_combine(a, b) {
  return a .* b // element-wise multiplication
}

compute_new_message( $D_u, D_{u-v}, D_v$ ) {
  new_message = 1 // vector of all ones
  ForEach(assignment  $x_u$ ) {
    ForEach(assignment  $x_v$ ) {
      new_message( $x_u$ ) +=  $D_v$ .belief( $x_v$ )  $\times$   $D_{u-v}$ .potential( $x_u, x_v$ )
                        /  $D_{u-v}$ .message(v)( $x_v$ )
    }
  }
  new_message = normalize(new_message)
   $D_{u-v}$ .message(u) = new_message
}
```

Appendix C

Dataset List

Dataset	Statistics	Origin	Real Tasks	Synthetic Tasks
Retina Image	256x64x64 grid 6-connected			Belief Propagation (Chap. 2)
Protein Interaction Network	14K vertices 100K edges	Elidan et al. [2006]	Gibbs Sampling (Chap. 2)	
CoEM Small	0.2M vertices 20M edges	Jones [2005]	Entity Recognition (Chap. 2)	
CoEM Large	2M vertices 200M edges	Jones [2005]	Entity Recognition (Chap. 2, Chap. 3)	
Financial Reports and Stock Volatility	30K companies 217K features	Kogan et al. [2009]	Lasso (Chap. 2)	
Lena Image	256x256 pixels			Compressed Sensing (Chap. 2)
Netflix	0.5M vertices 99M edges	Bennett and Lanning [2007]	Matrix Factorization (Chap. 2, Chap. 3)	
1,740 Frame Video	10.5M vertices 31M edges			Cosegmentation (Chap. 2)
Altavista Webgraph	1.4B vertices 6.6B edges	Yahoo [Retrieved 2011]	PageRank (Chap. 3)	
Altavista Domain Graph	25.5M vertices 355M edges	Yahoo [Retrieved 2011]	PageRank (Chap. 3)	
Synthetic Grid	300x300x300 grid 26-connected			Belief Propagation, Async. Snapshotting (Chap. 2)
Twitter Follower Graph	42M vertices 1.5B edges	Kwak et al. [2010]	PageRank (Chap. 3, Chap. 4) Triangle Counting (Chap. 3)	Graph Coloring (Chap. 4)
Wikipedia Term-Doc graph	11M vertices 315M edges	Wikipedia	LDA (Chap. 3)	Matrix Factorization (Chap. 3)
$\alpha = 2$ Synthetic Graphs				Graph Coloring (Chap. 3)
Sparser Synthetic Logistic Regression	1M parameters 50 features per point			SGD (Chap. 4)
Denser Synthetic Logistic Regression	100K parameters 50 features per point			SGD (Chap. 4)

Table C.1: Summary of all Datasets used in this thesis. Synthetic Tasks are tasks created for the purpose of benchmarking. Datasets are ordered by their approximate order of appearance in the thesis.

Bibliography

- Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDS*, 2006.
- A. Agarwal and J. C. Duchi. Distributed Delayed Stochastic Optimization. *ArXiv e-prints*, April 2011.
- Amr Ahmed, Yucheng Low, Mohamed Aly, Vanja Josifovski, and Alexander J. Smola. Scalable distributed inference of dynamic user interests for behavioral targeting. In *KDD*, pages 114–122, 2011.
- Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shравan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.
- Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. In *Nature*, volume 406, pages 378–482, 2000.
- Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.
- Arthur Asuncion, Padhraic Smyth, and Max Welling. Asynchronous distributed learning of topic models. In *NIPS*, pages 81–88, 2009.
- Dhruv Batra, Adarsh Kowdle, Devi Parikh, Jiebo Luo, and Tsuhan Chen. iCoseg: Interactive co-segmentation with intelligent scribble guidance. In *CVPR*, pages 3169–3176, 2010.
- James Bennett and Stan Lanning. The Netflix prize. In *Proceedings of KDD Cup and Workshop*, 2007.
- Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.
- Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.
- Danny Bickson. *Gaussian Belief Propagation: Theory and Application*. PhD thesis, The Hebrew University of Jerusalem, 2008.
- David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- Avrim Blum. On-line algorithms in machine learning. In *In Proceedings of the Workshop on On-Line Algorithms, Dagstuhl*, pages 306–325, 1996.
- Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.
- Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

- Ilaria Bordino, Paolo Boldi, Debora Donato, Massimo Santini, and Sebastiano Vigna. Temporal evolution of the UK Web. In *ICDM Workshops*, pages 909–918, 2008.
- Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l_1 -regularized loss minimization. In *ICML*, 2011.
- Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.
- Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.
- Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.
- Umit Catalyurek and Cevdet Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *IRREGULAR*, pages 75–86, 1996.
- K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
- K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, October 1984.
- C. Chevalier and F. Pellegrini. PT-SCOTCH: A tool for efficient parallel graph ordering. In *PMAA*, 2006.
- Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.
- Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2007.
- James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, and Garth Gibson. Solving the straggler problem with bounded staleness. In *HotOS*. USENIX Association, 2013.
- Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. *SoCC*, pages 1:1–1:14, 2012.
- Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Algorithms*, 55(1):58–75, April 2005.
- Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. ISSN 1070-9924.
- Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.
- Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, 2012.
- Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Umit V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *IPDPS*, 2006.
- Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
- Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. Least angle regression. *Annals of Statistics*, 32:407–451, 2004.
- Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818. ACM, 2010.

- Gal Elidan, Ian McGraw, and Daphne Koller. Residual Belief Propagation: Informed scheduling for asynchronous message passing. In *UAI*, pages 165–173, 2006.
- Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.
- Wenjiang J. Fu. Penalized regressions: The bridge versus the lasso. *Journal of Computational and Graphical Statistics*, 7(3):397–416, 1998.
- Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. In *PAMI*, 1984.
- Jacek Gondzio and Andreas Grothey. A new unblocking technique to warmstart interior point methods based on sensitivity analysis. *SIAM Journal on Optimization*, 19(3):1184–1210, 2008.
- Joseph Gonzalez, Yucheng Low, Carlos Guestrin, and David R. O’Hallaron. Distributed parallel inference on large factor graphs. In *UAI*, pages 203–212, 2009a.
- Joseph E. Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, volume 5, pages 177–184, 2009b.
- Joseph E. Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *AISTATS*, volume 15, pages 324–332, 2011.
- Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, October 2012.
- Hans Peter Graf, Eric Cosatto, Léon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In *NIPS*, 2005.
- Ronald L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal (BSTJ)*, 45:1563–1581, 1966.
- Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *POOSC*, 2005.
- William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. ISSN 0167-8191.
- Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Computer Architectures*, 1991.
- Jeffrey Robert Karplus Hartline. *Incremental optimization*. PhD thesis, Cornell University, Ithaca, NY, USA, 2008. AAI3295840.
- Thomas Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57, 1999.
- Intel. Intel threading building blocks, 2013. URL <http://threadingbuildingblocks.org/>.
- Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 41(3), 2007.
- Rosie Jones. *Learning to Extract Entities from Labeled and Unlabeled Text*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 2005.
- U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238, 2009.
- George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

- Seung-Jean Kim, Kwangmoo Koh, Michael Lustig, Stephen Boyd, and Dimitry Gorinevsky. An interior-point method for large-scale ℓ_1 -regularized least squares. *Selected Topics in Signal Processing, IEEE Journal of*, 1(4):606–617, 2007.
- Shimon Kogan, Dimitry Levin, Bryan R. Routledge, Jacob S. Sagi, and Noah A. Smith. Predicting risk from financial reports with regression. In *NAACL*, pages 272–280, 2009.
- Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
- Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD*, pages 426–434, 2008.
- Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.
- Oliver Kowalke. Boost context, 2013. URL http://www.boost.org/doc/libs/1_53_0/libs/context/doc/html/index.html.
- Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.
- Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, October 2012.
- Kevin Lang. Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs, Nov. 2004.
- Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2008a.
- Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008b.
- David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5:361–397, 2004.
- Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.
- Yucheng Low, Deepak Agarwal, and Alexander J. Smola. Multiple domain user personalization. In *KDD*, pages 123–131, 2011.
- Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.
- 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 *SIGMOD*, pages 135–146, 2010.
- Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR*, 2013.
- Jayadev Misra. Detecting termination of distributed computations using markers. In *SIGOPS*, 1983.
- Ramesh Nallapati, William Cohen, and John Lafferty. Parallelized variational EM for latent Dirichlet allocation: An experimental evaluation of speed and scalability. In *ICDMW*, 2007.
- Radford M. Neal and Geoffrey E. Hinton. A view of the EM algorithm that justifies incremental, sparse,

- and other variants. *Learning in graphical models*, 89:355–368, 1998.
- Jacob Nelson, Brandon Myers, A. H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching large graphs with commodity processors. In *HotPar*, 2011.
- David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, pages 1081–1088, 2008.
- Kamal Nigam and Rayid Ghani. Analyzing the effectiveness and applicability of co-training. In *CIKM*, pages 86–93, 2000.
- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.
- Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.
- Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.
- Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.
- Ananth Ranganathan, Michael Kaess, and Frank Dellaert. Loopy sam. In *IJCAI*, pages 2191–2196, 2007.
- Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.
- Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *ICML*, pages 791–798, 2007.
- Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs. Technical report, Universität Karlsruhe, Fakultät für Informatik, 2005.
- Athanassios G. Siapas. *Criticality and parallelism in combinatorial optimization*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, Jan. 1996.
- David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
- Alexander J. Smola and Shравan Narayanamurthy. An Architecture for Parallel Topic Models. In *VLDB*, 2010.
- Le Song, Arthur Gretton, Danny Bickson, Yucheng Low, and Carlos Guestrin. Kernel belief propagation. In *AISTATS*, volume 15, pages 707–715, 2011.
- Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. Technical Report MSR-TR-2011-121, Microsoft Research, November 2011.
- Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.
- Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.
- Kyle Wheeler, Richard Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *IPDPS*, pages 1–8, 2008.
- Reynold Xin, Joseph Gonzalez, and Michael Franklin. Graphx: A resilient distributed graph system on spark. In *GRADES SIGMOD Workshop*, 2013.

- Yahoo. Yahoo! Altavista web graph, Retrieved 2011. URL <http://webscope.sandbox.yahoo.com/>.
- J.S. Yedidia, W.T. Freeman, and Y. Weiss. Bethe free energy, kikuchi approximations, and belief propagation algorithms. Technical report, Mitsubishi Electric Research Laboratories, 2001.
- E. Alper Yildirim and Stephen J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization*, 12:782–810, March 2002.
- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.
- Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: a distributed framework for prioritized iterative computations. In *SOCC*, 2011.
- Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pages 337–348, 2008.
- Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.



**MACHINE LEARNING
DEPARTMENT**

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon.

Carnegie Mellon University does not discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex, handicap or disability, age, sexual orientation, gender identity, religion, creed, ancestry, belief, veteran status, or genetic information. Furthermore, Carnegie Mellon University does not discriminate and if required not to discriminate in violation of federal, state, or local laws or executive orders.

Inquiries concerning the application of and compliance with this statement should be directed to the vice president for campus affairs, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone, 412-268-2056