Messaging for Large-Scale Distributed Computation with Factor Graphs

by

Vinayak Ramesh

S.B., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Signature redacted

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2018

Signature redacted

Certified by . . . .
Devavrat Shah
Professor, Department of EECS
Thesis Supervisor
May 18, 2018

Signature redacted

Accepted by . . . .
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Messaging for Large-Scale Distributed Computation with Factor Graphs

by

Vinayak Ramesh

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present a language for generic computation using *Factor Graphs*, a computationally convenient data structure abstraction that has been popularly utilized for efficient inference in the framework of probabilistic graphical models cf. [22, 15, 30].

We show that message passing over Factor Graphs is Turing-complete. As an important contribution of this work, we show that a *Factor Graph* can be realized using any Publisher-Subscriber (PubSub) infrastructure. The resulting computational framework has multiple desirable properties.

We utilize different benchmark problems to demonstrate these properties of expressibility, ease of use, and performance, of our Factor Graph Computing framework: (a) Integer Optimization for hard problems, (b) Page-Rank, and (c) Singular Value Decomposition (SVD). We implement Factor Graph Computing on top of two dfferent PubSub systems: Redis's out-of-the-box PubSub and a PubSub that we have built on top of the Ligra graph processing system[25]. Both of these offer single machine PubSub implementations. We find that our *single* machine implementation is comparable to (a) state-of-the-art commercial optimization solvers [17] for challenge optimization benchmarks [18], (b) native Ligra [25] for large scale PageRank, and (c) a hardware optimized implementation over 68 machine cluster of Apache Spark for computing SVD [11]. In addition, we present a new algorithm for Integer Optimization problems using Belief Propagation, which is of independent interest.

Our framework using *Factor Graphs* brings computation next to data: this removes the communication bottleneck present in modern distributed computation infrastructures.

Thesis Supervisor: Devavrat Shah
Title: Professor, Department of EECS

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Devavrat Shah for taking me on as an M.Eng student, and for the continuous guidance, support, and encouragement, he has provided over the last two years. His door was always open whenever I needed help with my research or writing, and his enthusiasm for problem solving was contagious. I could not have imagined having a better advisor and mentor for this work.

A big thanks to MIT and the staff who work tirelessly to make this unique educational experience here possible - first as an undergrad, and then as a graduate student. I could not have imagined finding a better home to pursue my intellectual curiosity.

I would also like to acknowledge my family, without whom none of this would have been possible. Their support continues to drive me, and I would not have been able to be where I am today without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this work, we propose a language for computation using *Factor Graphs* [30]. A factor graph is a bipartite graph which has two types of nodes: *variable* nodes and *factor* nodes. Figure 1-1 shows an example of a factor graph. Formally, factor graphs were introduced in [29, 13, 12] in the context of decoding for graph-based codes and more generally for inference in probabilistic graphical models (also see [22, 15, 33, 9]). Distinct data and distinct computation functions are associated with each variable node and factor node; in addition, each edge also has data, called messages, associated along both of its directions. Variable and factor computation functions update the data associated with edges, when the appropriate node *receives* a message. Computation happens through an iterative *message-passing* process between nodes of the factor graph. When a message along a particular edge direction is updated

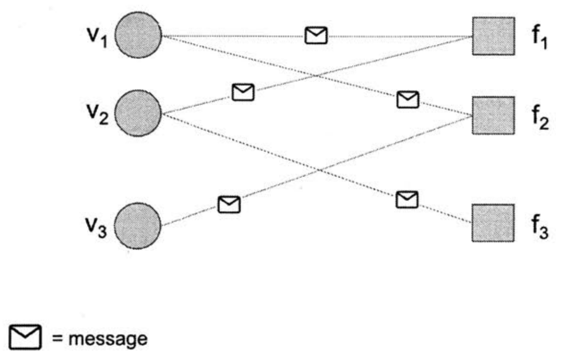

Figure 1-1: An example of a Factor Graph.

by a node, it triggers the opposite end-point of the edge to re-compute data along all of its out-going edges using the appropriate node's computation function, which further triggers updates. Computation can be implemented in either an asynchronous or synchronous manner.

The main contributions of this work are the following:

1. Factor graphs are computationally expressible (Turing complete)
2. They can be implemented efficiently using Publisher-Subscriber (PubSub) infrastructure

The main components of a Publisher-Subscriber system are *publishers*, *subscribers* and *channels*. Communication happens between publishers and subscribers through channels, and PubSub infrastructure has been built to perform this communication efficiently. This communication works as follows: Each publisher can *publish* an event (or message) to a channel, which in turn broadcasts that message to all subscribers of that particular channel. Upon receiving a message from its subscribed channel, some computation can be triggered at a subscriber. For a factor graph, each direction of each edge can be considered a channel. One of the end nodes (variable or factor) can be considered a publisher which publishes to the edge's *channel*, and the other end node (factor or variable) is a subscriber subscribed to that edge channel. Upon receiving a message from its (publisher) neighbor, the subscriber node is activated to run its node computation function. Factor graphs are designed to have extremely light-weight messages traversing along its edges and PubSub systems are optimized for communicating these light-weight messages at a very high bandwidth, making it ideal for implementing factor graph based computation.

To answer the natural question of what factor graphs can compute, we establish that our factor graph "language", based on message-passing between nodes, is Turing complete. This means that, unlike prior work, our language is not restrictive in its computational power.

To understand the performance of factor graph computation, we consider implementations of factor graphs using two PubSub systems: an out of the box (unopti-

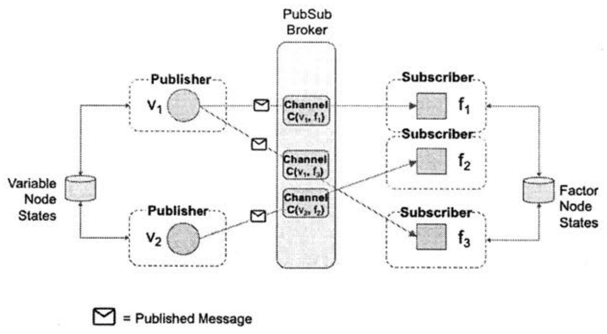

Figure 1-2: Diagram of a PubSub system of a Factor Graph

mized) asynchronous PubSub system using Redis, with computation functions written in Python, and a PubSub system that we built on top of the Ligra framework, written in C.

With these two implementations, but the same "program", we study three benchmark problems: (a) Integer Optimization to establish that our system can solve extremely challenging computational problems; (b) Computing PageRank for large graph as these have been the benchmarks used historically by such frameworks, including Ligra; and (c) Computing Singular Value Decomposition (SVD) of a large matrix as it is the work-horse of linear algebraic computations that are prevalent in scientific computation and modern machine learning.

We show that for *easy* optimization problems, our framework works as well as popular commercial optimization solvers cf. Gurobi [17]. For benchmark *challenge* binary optimization problems [18], our framework manages to converge and find an integer solution while Gurobi seems to struggle.

For the PageRank benchmark, we utilize the Orkut network dataset from [21]. We compare the performance of our implementation with that of native Ligra. Our implementation using Ligra-based PubSub takes similar time as that of Ligra suggesting that our framework is no worse than Ligra for simple PageRank-like problems that are canonical questions for a Ligra-like language.

Finally, we compare the Singular Value Decomposition (SVD) computation for a very large synthetic matrix with 51M non-zeroes in a 1Mx1M square matrix. Our framework using Ligra-PubSub obtains similar performance as the Spark benchmark

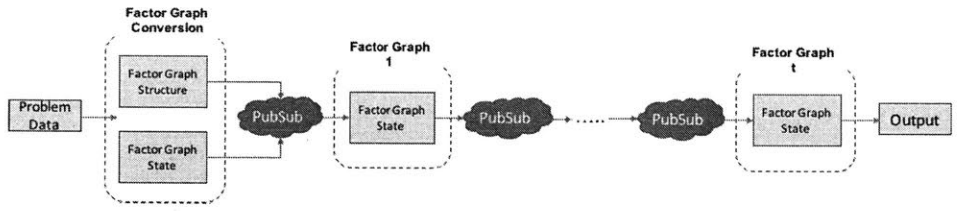

Figure 1-3: Flow of data through Factor Graph Compute system using PubSub.

[11], however, using a single machine compared to 68 executors utilized in [11] along with hardware acceleration. In terms of overall resource utilization, this is >13x improvement (and this ignores all the overhead involved in setting up a Spark cluster, etc.).

This work has two distinguishing advantages over prior work. One, our language based on message passing over factor graphs is Turing complete. That being said, the factor graph view of problems is not necessarily intuitive. However, it is possible to design "factor graph converters" for many problems, including: optimization, linear algebraic operations, and loss function minimization in machine learning (i.e. stochastic gradient descent) as shown in Section 2.2. Second, the language (and hence program) is decoupled from its implementation, which is based on PubSub. By utilizing the underlying PubSub, the same computation can run in different environments without changing a line of "application" code, whether it is single-machine environment, like that based on Redis or Ligra, or a multi-machine environment, such as Apache Kafka [19]. This is particularly helpful property for transitioning from prototyping to production environments.

Chapter 2

Computation Framework

We now present our factor graph based language for computation. First, we establish that this language is Turing complete. We then discuss how many common problems, including graphical model inference, PageRank computation, matrix multiplication and singular value decomposition, as well as loss-function minimization for model learning in machine learning, can be represented in this language in a natural manner.

2.1 Factor Graph Computing

Factor Graphs were initially introduced as an abstraction for computationally efficient inference over probabilistic graphical models [22]. They have been shown to be a universal representation for any probabilistic graphical model. Generally, such a factor graph has been used to derive heuristic algorithms like Belief Propagation [34]. However, by viewing the factor graph in generality rather than restricting to belief propagation like algorithms only, we will argue that it can lead to a Turing complete language.

2.1.1 Language of Factor Graph

We now describe the language of Factor Graph Computing [30].

Factor Graph: A factor graph $\mathcal{G} = (V, F, E)$ is a bipartite graph where $V = \{v_1, \ldots, v_n\}$

is the set of variable nodes, $F = \{f_1, \dots, f_m\}$ is the set of factor nodes, and $E \subset V \times F$ denote the set of undirected edges between variable nodes V and factor nodes F. We shall use $\mathbb{Q}$ to denote rational numbers.

States and Messages: Each variable node $v \in V$ has finite dimensional state $x_v \in \mathbb{Q}^{p_v}$ associated with it for some $p_v \geq 1$; each factor node $f \in F$ has finite dimensional state $y_f \in \mathbb{Q}^{p_f}$ associated with it for some $p_f \geq 1$.

Each edge $(v, f) \in E$ with $v \in V, f \in F$ has two messages associated with it, one for each direction: message $m_{v \to f} \in \mathbb{Q}^{b_v}$ is from variable node $v \in V$ to factor node $f \in F$ and message $m_{f \to v} \in \mathbb{Q}^{b_f}$ in the opposite direction.

Each variable node $v \in V$ has message-update function $\textsc{UpdateVar}_v : \mathbb{Q}^{a_v} \times F \to \mathbb{Q}^{b_v}$ associated with it; each factor node $f \in F$ has message-update function $\textsc{UpdateFac}_f : \mathbb{Q}^{a_f} \times V \to \mathbb{Q}^{b_f}$ associated with it. Here a_v, b_v for any $v \in V$ and a_f, b_f for any $f \in F$ are such that

$$a_v = p_v + \sum_{g \in \mathcal{N}(v)} b_g, \quad a_f = p_f + \sum_{u \in \mathcal{N}(f)} b_u.$$

where $\mathcal{N}(v) = \{g : (v, g) \in E\}$ represents neighbors of v and $\mathcal{N}(f) = \{u : (u, f) \in E\}$ represents incoming neighbors of f in the factor graph $\mathcal{G}$ with respect to E.

Computation Dynamics: During the execution of message passing, a message can be dynamically updated using other messages or states from nodes. Messages are the only truly dynamic data structures that need "communication" along edges E of a factor graph. States associated with variable or factor nodes are allowed to be updated only as an "external" input - that is, states can not be modified by messages.

Messages are updated as follows: Precisely, for any $(f, v) \in E$ with $f \in F, v \in V$, the message $m_{f \to v}$ is updated as

$$m_{f \to v} = \textsc{UpdateFac}_f(y_f; m_{u \to f}, u \in \mathcal{N}(f); v); \tag{2.1}$$

and for any $(v, f) \in E$ with $v \in V, f \in F$, the message $m_{v \to f}$ is updated as

$$m_{v \to f} = \textsc{UpdateVar}_v(x_v; m_{g \to v}, g \in \mathcal{N}(v); f). \tag{2.2}$$

In the above, by explicitly having v as an argument in $\textsc{UpdateFac}_f$ and f as an argument in $\textsc{UpdateVar}_v$, we are allowing for flexibility to have different update functions for different edges incident on each factor and variable node.

Modes of Computation: There are two modes of computation: *asynchronous* and *synchronous*. In the *asynchronous* mode of computation, the message update along edge $(v, f) \in E$ for $v \in V, f \in F$ from $v \to f$ is triggered when any of the messages $m_{g \to v}$ for $g \in N(v)$ is updated; similarly, the message update along edge $(v, f) \in E$ for $v \in V, f \in F$ from $f \to v$ is triggered when any of the messages $m_{u \to f}$ for $u \in N(f)$ is updated.

In the *synchronous* mode of the computation, in each time step, all messages from variable nodes to factor nodes are updated simultaneously and then messages from factor nodes to variable nodes are updated simultaneously.

Output: The output of the computation is viewed as the value associated with the messages. Specifically, the value of a pre-designated subset of messages can be viewed as the value of computation output at any given instance.

2.1.2 Turing Completeness of Factor Graph Message Passing

To establish the expressibility of the language of Factor Graph as a Turing complete language, we argue that a recursive neural network architecture can be simulated within the framework of Factor Graph [30].

It has been established that recursive neural network is Turing complete [28]. Therefore, we will be able to conclude that the language of Factor Graph is Turing complete. The proof given in [30] is as follows:

We start by defining a recursive neural network as considered in [28]. In [28], the authors defined a *processor net* as a non-linear dynamical system with an external input. Specifically, let $t \geq 1$ denote discrete time. Let $x(t) \in \mathbb{Q}^d$ denote the finite-dimensional state of the dynamical system at time t with rational values. Let $u(t) \in \{0,1\}^p$ denote p-dimensional external binary input at each t. The state is updated as

$$x(t+1) = \sigma\big(Ax(t) + bu(t) + c\big), \tag{2.3}$$

where $A \in \mathbb{Q}^{d\times d}$, $b \in \mathbb{Q}^{d\times p}$ and $c \in \mathbb{Q}^d$ are system parameters; with notation $\sigma(q_1, \ldots, q_d) = (\sigma(q_1), \ldots, \sigma(q_d))$ for $q_1, \ldots, q_d \in \mathbb{Q}$; and the *sigmoid* function $\sigma : \mathbb{R} \to [0,1]$ is defined as

$$\sigma(q) = \begin{cases} 0 & \text{if } q < 0 \\ q & \text{if } q \in [0,1] \\ 1 & \text{if } q > 1. \end{cases} \tag{2.4}$$

Theorem 1. *Any processor net can be simulated using factor graph computing in synchronous mode.*

Proof. To establish Theorem 1, we need to show a factor graph representation for any *processor net.* Such a factor graph is shown in Figure 2-1. Specifically, given a processor net as described by (2.3), we simulate it using a factor graph which has two variable nodes $V = \{v_1, v_2\}$ and a factor node $F = \{f\}$. The state associated with node v_1 is $x_{v_1} = 0$, while the state associated with v_2 is the external input of the processor net, $x_{v_2} = u(t)$. The state associated with factor node f, $y_f = 0$ as well.

The message update function associated with factor node f is given as:

$$\textsc{UpdateFac}_f(y_f; m_{v_1\to f}, m_{v_2\to f}; v) \tag{2.5}$$

$$= \begin{cases} \sigma\big(Am_{v_1\to f} + bm_{v_2\to f} + c\big) & \text{if } v = v_1, \\ \text{null} & \text{if } v = v_2. \end{cases} \tag{2.6}$$

The message update function associated with variable nodes v_1 and v_2 are given as:

$$\text{UPDATEVAR}_{v_1}(x_{v_1}; m_{f\to v_1}; f) = m_{f\to v_1}, \tag{2.7}$$

$$\text{UPDATEVAR}_{v_2}(x_{v_2}; m_{f\to v_2}; f) = x_{v_2}. \tag{2.8}$$

As shown in Figure 2-1, we initialize messages as

$$m_{v_1\to f} = x(1) \tag{2.9}$$

$$m_{v_2\to f} = u(1) \tag{2.10}$$

$$m_{f\to v_1} = x(1) \tag{2.11}$$

$$m_{f\to v_2} = \text{null}. \tag{2.12}$$

Here $x(1), u(1)$ are the initial state of the processor net and the external input, respectively. Let the factor graph computation be done in synchronous mode with the state of v_2 being updated in time t to be the external input $u(t)$. It can be easily checked that the $m_{f\to v_1}$ at the end of time instance t is precisely $x(t+1)$ of the processor net, which completes the proof. $\square$

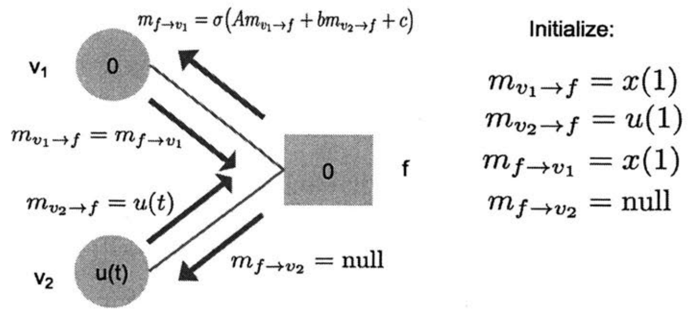
$m_{f\to v_1} = \sigma(Am_{v_1\to f} + bm_{v_2\to f} + c)$
Initialize:
v1
0
$m_{v_1\to f} = x(1)$
$m_{v_2\to f} = u(1)$
$m_{f\to v_1} = x(1)$
$m_{f\to v_2} = \text{null}$
$m_{v_1\to f} = m_{f\to v_1}$
0
f
$m_{v_2\to f} = u(t)$
$m_{f\to v_2} = \text{null}$
v2
u(t)

Figure 2-1: Factor Graph that simulates a σ-process net (a la recursive neural network).[30]

2.2 Applications

We now show how our factor graph language can be used to represent a variety of computations, just by specifying the appropriate dynamics: initial messages and variable and factor update functions. We provide examples of inference over graphical models, PageRank, Linear Algebra, and loss function minimization for model learning in machine learning via Stochastic Gradient Descent algorithm[30].

2.2.1 Inference Over Graphical Models

Consider a collection of n random variables, represented as $\vec{X} = (X_1, \dots, X_n) \in \Sigma^n$. Let $\mathbb{P}_{\vec{X}} : \Sigma^n \to [0,1]$ represent their joint distribution. As long as $\mathbb{P}_{\vec{X}}(\vec{\sigma}) > 0$ for any $\vec{\sigma} = (\sigma_1, \dots, \sigma_n) \in \Sigma^n$, there exists a probabilistic factor graph representation for $\mathbb{P}_{\vec{X}}$. Specifically, there exists a bipartite factor graph $\mathcal{G} = (V, F, E)$ with $V = \{v_1, \dots, v_n\}$ corresponding to n variables, $F = \{f_1, \dots, f_m\}$ corresponding to factors and $E \subset V \times F$. With each factor node f_i, is an associated factor function $g_i : \Sigma^{|\mathcal{N}(f_i)|} \to \mathbb{R}_+$ such that for any $\vec{\sigma} \in \Sigma^n$,

$$\mathbb{P}_{\vec{X}}(\vec{\sigma}) \propto \prod_{i=1}^{m} g_i\big(\sigma_j : v_j \in \mathcal{N}(f_i)\big) \tag{2.13}$$

$$= \frac{1}{Z} \prod_{i=1}^{m} g_i\big(\sigma_j : v_j \in \mathcal{N}(f_i)\big), \tag{2.14}$$

where $Z = \sum_{\vec{\sigma} \in \Sigma^n} \prod_{i=1}^{m} g_i\big(\sigma_j : v_j \in \mathcal{N}(f_i)\big)$ is the normalization constant.

The two inference tasks we are interested in are: (a) Computing the marginal distribution: the marginalization of X_i for each $i \leq n$ given (2.13) (b) Finding the Maximum A Posteriori (MAP) assignment: the $\vec{\sigma} \in \Sigma^n$ with maximal probability The factor graph representation of our joint probability distribution is useful to design computationally efficient *heuristics* known as *Belief Propagation* (BP) algorithms. The variation of BP for Marginalization is known as *sum-product* and BP for MAP is known as *max-product.* These algorithms naturally fit the Factor Graph Computation framework.

In both sum-product and max-product, the underlying graph for Factor Graph Compute is the same as the probabilistic factor graph. The states associated with variables $v \in V$ are null and the states associated with factors $f \in F$ are the factor functions g_f. For each $(v, f) \in E$, let $m_{v \to f}, m_{f \to v} \in \mathbb{Q}^{|\Sigma|}$. Without loss of generality, let $\Sigma = \{1, \ldots, k\}$. Then, we can view $m_{v \to f}, m_{f \to v}$ as k dimensional vector with $m_{v \to f}[j]$ (resp. $m_{f \to v}[j]$) representing the jth component of the message, $1 \leq j \leq k$.

In the sum-product algorithm, messages are iteratively updated as follows: for any $(v, f) \in E$,

$$m_{v \to f}[j] = \prod_{f' \in \mathcal{N}(v) \setminus \{f\}} m_{f' \to v}[j], \tag{2.15}$$

$$\begin{aligned} m_{f \to v}[j] = \sum_{\sigma_{v'} \in \Sigma : v' \in \mathcal{N}(f) \setminus v} & g_f\big(\sigma_v = j; \sigma_{v'}, v' \in \mathcal{N}(f) \setminus v\big) \times \\ & \prod_{v' \in \mathcal{N}(f) \setminus v} m_{v' \to f}[\sigma_{v'}]. \end{aligned} \tag{2.16}$$

For max-product, the (2.15) remains the same but (2.16) changes as

$$\begin{aligned} m_{f \to v}[j] = \max_{\sigma_{v'} \in \Sigma : v' \in \mathcal{N}(f) \setminus v} & g_f\big(\sigma_v = j; \sigma_{v'}, v' \in \mathcal{N}(f) \setminus v\big) \times \\ & \prod_{v' \in \mathcal{N}(f) \setminus v} m_{v' \to f}[\sigma_{v'}]. \end{aligned} \tag{2.17}$$

It can be easily checked that (2.15), (2.16) and (2.17) lead to defining appropriate UPDATEVAR and UPDATEFAC functions, and hence they can be viewed as instances of Factor Graph Compute.

2.2.2 PageRank

We show that we can compute PageRank using Factor Graph Compute. Let $G = (V_G, E_G)$ be a graph with vertices $V_G = \{1, \ldots, n\}$ and directed edges $E_G \subset V_G \times V_G$. Let $\mathcal{N}_G^{\text{in}}(i) = \{j \in V_G : (j, i) \in E_G\}$ be the set of incoming neighbors of i and $\mathcal{N}_G^{\text{out}}(i) = \{j \in V_G : (i, j) \in E_G\}$ be the set of outgoing neighbors of i.

The *PageRank* of node $i \in V_G$, denoted as $PR(i)$, is defined as

$$PR(i) = \frac{1-\alpha}{n} + \alpha \sum_{j \in \mathcal{N}^{\text{in}}(i)} \frac{PR(j)}{|\mathcal{N}^{\text{out}}(j)|}, \tag{2.18}$$

Here, $\alpha \in (0,1)$ is the *dampening* factor. PageRank can be computed by a simple iterative algorithm where we first set $PR(i) = 1/n$ for all $i \in V_G$ and then iteratively apply (2.18). After several iterations this procedure will converge to the correct PageRank value for all nodes.

We now show how to implement this iterative algorithm via Factor Graph Compute. We define the associated factor graph for computation over our initial graph G, as follows. Let $\mathcal{G} = (V, F, E)$ with

$$\begin{aligned} V &= \{v_i,\ i \in V_G\}, \\ F &= \{f_i,\ i \in V_G\}, \\ E &= \{(v_j, f_i),\ (j,i) \in E_G\} \cup \{(v_i, f_i)\}. \end{aligned}$$

The state of variable node $v_i \in V$, x_{v_i}, is set to $|\mathcal{N}^{\text{out}}(i)|$. The state associated with factor nodes $f_i \in F$ is set to null. We initialize messages as follows. For any $(v_j, f_i) \in E$, with $j \neq i$,

$$m_{v_j \to f_i} = 0, \quad m_{f_i \to v_j} = 0. \tag{2.19}$$

For any $(v_i, f_i) \in E$

$$m_{v_i \to f_i} = 0, \quad m_{f_i \to v_i} = \frac{1}{n}. \tag{2.20}$$

The message updates dynamics can be written as follows. For any $(v_j, f_i) \in E$ with

$j \neq i$,

$$m_{v_j \to f_i} = \frac{m_{f_j \to v_j}}{x_{v_j}}, \tag{2.21}$$

$$m_{f_i \to v_j} = 0. \tag{2.22}$$

And for (v_i, f_i),

$$m_{v_i \to f_i} = 0, \tag{2.23}$$

$$m_{f_i \to v_i} = \frac{1-\alpha}{n} + \alpha \sum_{v_j \in \mathcal{N}(f_i) \setminus \{v_i\}} m_{v_j \to f_i}. \tag{2.24}$$

It can be seen that (2.21)-(2.24) lead to the appropriate UPDATEVAR and UPDATEFAC definitions. The value of $PR(i)$ at any time (after some number of iterations) is just $m_{f_i \to v_i}$ for all $i \in V_G$.

2.2.3 Machine Learning and (Stochastic) Gradient Descent

Optimization, and in particular, Stochastic Gradient Descent has become a cornerstone of supervised learning. This works by observing data (y_i, x_i), $1 \leq i \leq N$ where x_i are *features* and y_i are *targets* of interest. The goal of this procedure is to learn a model that helps predict an unknown target y, given some features x. This can be viewed as solving an optimization problem of the following form:

$$\mathsf{minimize} \ \frac{1}{n} \sum_{i=1}^{n} L(y_i, x_i; \theta) + R(\theta) \quad \mathsf{over} \quad \theta \in \mathbb{R}^d. \tag{2.25}$$

$\theta \in \mathbb{R}^d$ represents the model parameter and $L(y, x; \theta)$ represents the loss-function parameterized by θ which captures the "loss" or error in predicting target y using features x, for a specific value of θ. The model regularizer $R(\theta)$ imposes a penalty for complex models, where complexity is a function of θ. The gradient descent algorithm for (2.25) can be written as follows: components of θ are iteratively updated as

$$\theta_k \leftarrow \theta_k - \alpha \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\partial L}{\partial \theta_k}(y_i, x_i; \theta) + \frac{\partial R}{\partial \theta_k}(\theta) \right), \tag{2.26}$$

for $1 \leq k \leq d$.

We show how this algorithm can be implemented in the Factor Graph Compute framework in a straight forward manner.

Define a factor graph $\mathcal{G} = (V, F, E)$ with $V = \{v_1, \ldots, v_d\}$ where v_k corresponds to θ_k for $1 \leq k \leq d$, $F = \{f_1, \ldots, f_n, f_{n+1}\}$ where f_i corresponds to data point (y_i, x_i) for $1 \leq i \leq n$ and f_{n+1} corresponds to regularization, and all possible edges between V and F are present, i.e. $E = V \times F$. The state associated with variable nodes is *null*, state associated with factor nodes $f_i, 1 \leq i \leq n$ is the data observation (y_i, x_i), state associated with f_{n+1} is *null*.

All messages are initially set to 0. The associated message update dynamics for gradient descent are as follows:

$$m_{v_k \to f_i} = \frac{1}{n} \left(\sum_{j=1}^{n} m_{f_j \to v_k} \right) + m_{f_{n+1} \to v_k}, \tag{2.27}$$

$$m_{f_i \to v_k} = \begin{cases} m_{v_k \to f_i} - \alpha \frac{\partial L}{\partial \theta_k}\left(y_i, x_i; [m_{v_\ell \to f_i}]_{1 \leq \ell \leq d}\right), & 1 \leq i \leq n \\ -\alpha \frac{\partial R}{\partial \theta_k}\left([m_{v_\ell \to f_i}]_{1 \leq \ell \leq d}\right), & i = n+1 \end{cases} \tag{2.28}$$

for all $1 \leq k \leq d$, $1 \leq i \leq n+1$. In 2.27, $m_{v_k \to f_i}$ for all i is identical and represents the value of kth component of model parameter.

These messages, which correspond to the model parameter, are the output of the algorithm. It can be checked that (2.27) and (2.28) provide the appropriate UPDATEVAR and UPDATEFAC functions, thus showing gradient descent can implemented using Factor Graph Compute. For gradient descent, the above message updates have to be performed synchronously. If the message updates from factors are coming in at random and / or asynchronously, the algorithm becomes an implementation of *stochastic gradient descent*.

2.2.4 Linear Algebra

Linear algebra is central to most modern scientific computing. The key operation in Linear Algebra is matrix-vector multiplication. We argue that it fits into the Factor Graph Compute model in a seamless manner. To that end, let $\vec{b} \in \mathbb{R}^n$ be a vector, $\mathbf{A} \in \mathbb{R}^{n \times n}$ be an $n \times n$ matrix and our interest is in $\mathbf{A}\vec{b}$. We define the factor graph $\mathcal{G} = (V, F, E)$ as

$$V = \{v_i,\ 1 \le i \le n\}, \quad F = \{f_j,\ 1 \le j \le n\}, \quad E = V \times F. \tag{2.29}$$

The variable nodes correspond to components of $\vec{b}$, the factor nodes correspond to columns of $\mathbf{A}$. The state associated with all variable nodes is *null* and state associated with factor node f_i is the ith column of $\mathbf{A}$, which is n-dimensional vector $a_{\cdot i} = [A_{ki}]_{1 \le k \le n}$. The messages are initialized as follows: for all $1 \le i, j \le n$,

$$m_{v_i \to f_j} = \begin{cases} b_i & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{2.30}$$

$$m_{f_j \to v_i} = 0. \tag{2.31}$$

The following message updates need to happen only *once*, in a synchronous manner. First we send messages from factor nodes to variable nodes and then variable nodes to factor nodes. The precise update dynamics are as follows: for all $1 \le i, j \le n$,

$$m_{f_j \to v_i} = m_{v_j \to f_j} a_{\cdot j}^T e_i \tag{2.32}$$

$$m_{v_i \to f_j} = \begin{cases} \sum_\ell m_{f_\ell \to v_i} & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases} \tag{2.33}$$

In the above, $e_i = [0 \ldots 1 \ldots 0]$ is the vector with one 1 in ith component. Observe that the message $m_{v_i \to f_i}$ is the ith component of vector $\mathbf{A}\vec{b}$. Repeatedly iterating the above leads to a power-iteration like algorithm which can become the key step for performing Singular Value Decomposition.

Chapter 3

Publisher Subscriber (PubSub)

3.1 Background

Publish Subscribe (PubSub) [32] is a widely used software pattern for communicating "events" between different entities, at scale. A PubSub system consists of *Publishers* and *Subscribers* who *do not* need to know about each others' existence and communicate by publishing / subscribing to pre-defined *Channels*. This decoupling of senders (Publishers) and receivers (Subscribers) through the interface of communication channels has made it a particularly suitable infrastructure for a variety of applications.

PubSub systems have found many applications such as in gaming, in building chat systems, asynchronous event notifications [7], stream ingestion [16], and log processing [19]. Different PubSub systems come with different guarantees and properties. These include exactly-once, at-most once, or at-least once delivery and processing [16].

PubSub systems have the advantage that neither Publishers nor Subscribers need to know the communication topology. That is, messages are broadcast without Publishers knowing the message destination. Decoupling Publishers and Subscribers allows for scaling each independently of the other. The PubSub pattern is similar to message queues. Unlike message queues, PubSub messages are broadcast and done so immediately and asynchronously. There is no queuing of messages, and messages can be processed by multiple Subscribers at once.

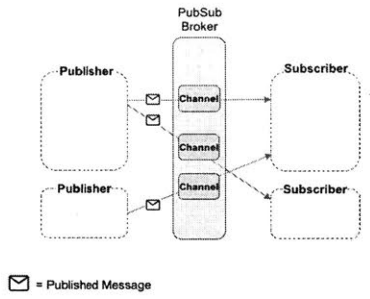

Figure 3-1: A diagram illustrating a pubsub system

There are several implementations of PubSub available including Apache Kafka [19], Redis [23], Akka [3], and zeroMQ [5]. Large scale and commercial implementations of PubSub systems include Google PubSub [16], Amazon Simple Notification Services [6], and Apache ActiveMQ [8].

The main components of a PubSub system are Publishers, Subscribers, Channels, Subscriptions, Events, and a Broker. Subscribers first *subscribe* to channels of interest. These *subscriptions* can be maintained by the Broker or by the Subscribers themselves. Publishers can *publish*, some data, which we call an event, to any channel, with the intention of the event being delivered to the subscribers of that particular channel. When an event is published to a channel, it is routed to the relevant subscribers by the Broker, which is typically a type of message switch. Given an incoming event, the Broker *relays* the event data over the appropriate channel, then *triggers* the relevant subscribers. Finally, the *triggered* subscribers run their associated callback functions with the *relayed* event data as input.

PubSub can be run synchronously or asynchronously. In a synchronous PubSub system, all publishers publish simultaneously, and all relevant subscribers are triggered simultaneously by the Broker. In an asynchronous PubSub system, publishers can publish at different times, and triggering of Subscribers and running of callback functions does not need to happen at the same time.

3.2 Formalism of PubSub

Formally, we define a PubSub system $\Pi = (S, P, C, \mathcal{S}, B, \mathcal{E}(t))$ as consisting of a set of subscribers S, a set of publishers P, a set of channels C, a set of subscriptions $\mathcal{S}$, a Broker B, and a set of events $\mathcal{E}(t)$ which varies with time t.

Let $P = \{p_1, \ldots, p_m\}$ be the set of publishers, $C = \{c_1, \ldots, c_L\}$ be the set of channels, and $S = \{s_1, \ldots, s_n\}$ be the set of subscribers. Let $\textsc{Sub}(c) \subset S$ be the set of subscribers that are subscribed to a given channel $c \in C$. A publisher $p \in P$ can publish to *any* channel $c \in C$. The primary dynamics in PubSub consists of a publisher, say $p \in P$ publishing a "message" to a channel, say $c \in C$, which in turn gets relayed to all the subscribers $s \in \textsc{Sub}(c)$ which in turn triggers *call-back* function f_s associated with the subscriber $s \in S$ with input as the "message" relayed as well as information about the publisher and channel.

To execute the above dynamics, implementation of a PubSub system usually involves a *Broker* that manages various state information as well as execution of operations accordingly. We will not go into detail about how the broker is usually implemented, however make some high level remarks. To begin with, the broker manages registration of publishers, channels, subscribers and subscription of subscribers to channels, and provides an interface to alter this state.

The broker manages the set of "events" which need to be communicated at each point of time. These are primarily messages published by publishers but have not yet been relayed to their subscribers. The broker executes these events by first relaying the message to appropriate subscribers depending on the subscription topology, then executing the call back functions at each subscriber that received new message and updating the "event queue" upon successful completion. The broker provides interface for publisher to publish to message to a channel, i.e. create an event.

From an end user's perspective, the interfaces (or function calls) that PubSub exposes are as follows:

- Registration
  - PUBLISHER(publisher id): register publisher id as publisher.

  - $\textsc{Channel}(\mathsf{channel\ id})$: register $\mathsf{channel\ id}$ as channel.
  - $\textsc{Subscriber}(\mathsf{subscriber\ id}, \mathsf{callback\ function})$: register $\mathsf{subscriber\ id}$ as subscriber with associated $\mathsf{callback\ function}$.
  - $\textsc{Subscribe}(\mathsf{channel\ id}, \mathsf{subscriber\ id})$: subscribe $\mathsf{subscriber\ id}$ to $\mathsf{channel\ id}$, i.e
$$\textsc{Sub}(\mathsf{channel\ id}) = \textsc{Sub}(\mathsf{channel\ id}) \cup \{\mathsf{subscriber\ id}\}.$$

- Publish
  - $\textsc{Publish}(\mathsf{publisher\ id}, \mathsf{channel\ id}, \mathsf{message})$: publish $\mathsf{message}$ on channel $\mathsf{channel\ id}$ from publisher $\mathsf{publisher\ id}$.

3.2.1 Subscribers

Each subscriber $s \in S$ has a set of subscriptions $C_s \subset C$, a *trigger status* $T_s \in \{0, 1\}$, *subscribe* function $\textsc{Subscribe}_s$, and *callback function* f_s.

The set of subscriptions C_s is just the set of channels to which subscriber s is subscribed. The triggered status T_s is 1 if subscriber s has been *triggered* and 0 otherwise. The function $\textsc{Subscribe}$ adds a specified channel to the subscriptions of s:

$$C_s \cup \{c\} = \textsc{Subscribe}_s(c) \tag{3.1}$$

The set of all subscriptions $\mathcal{S}$ is $\{C_s | \ \forall s \in S\}$.

The callback function for subscriber s, f_s, is application dependent and is run after the subscriber s has been *triggered*, that is, when $T_s = 1$. After running f_s, the *triggered status* of s is reset, that is, T_s is set to 0.

3.2.2 Publishers

Each publisher $p \in P$ has publish function $\textsc{Publish}$ associated with it . An *event* is denoted as $e_{p \to c}$, where p denotes the publisher of the event, and c denotes the channel to which the event is meant to be published. Typically, a publisher can publish to any channel. At a particular time t, the PUBLISH function updates the entire set of PubSub events, $\mathcal{E}(t)$ with $e_{p\to c}$ as follows:

$$\text{PUBLISH}(e_{p\to c}) = \mathcal{E}(t) \cup \{e_{p\to c}\} \tag{3.2}$$

The interface for PUBLISH is given in Interface 1. The method PUBLISH takes in a channel c, and event e. The details of publishing are dependent on the implementation of PubSub.

Algorithm 1 PUBLISHERINTERFACE

```
procedure PUBLISH($c, e$)
```

3.2.3 Broker

The Broker B has $\mathcal{S}$, the set of all subscriptions, $\mathcal{E}(t)$, the set of events published to all channels at time t and $S_T(t) \subset S$, the set of all triggered subscribers at time t. The Broker B also has functions RELAY, TRIGGER, and RUN.

The function RELAY checks $\mathcal{E}(t)$ at each time t, then calls TRIGGER to update the trigger status of subscribers with incoming events on their subscribed channels, and finally, facilitates running the callback function f_s on incoming events for each triggered subscriber via the function RUN.

The function TRIGGER updates the set of triggered subscribers S_T by using the set of subscriptions $\mathcal{S}$ to determine the set of channels which have incoming messages: $C_{in} = \{c|\ e_{p\to c} \in \mathcal{E}(T)\}$, and setting the trigger status T_s for every subscriber $s \in S$ to 1 if s is subscribed to $c \in C_{in}$, that is, if $c \in C_{in} \cap C_s$, and $T_s = 0$ otherwise.

The function RUN applies the callback function f_s to events published to triggered subscribers $s \in S_T$.

We give an implementation of RELAY in BROKERINTERFACE (Interface 2). The functions TRIGGER and RUN are given as:

$$S_T(t+1) = \textsc{Trigger}\left(\mathcal{E}(t+1), S_T(t), \mathcal{S}\right) \tag{3.3}$$

$$f_s(e_{p\to c}) = \textsc{Run}(\mathcal{E}(t+1), S_T(t+1), \mathcal{S}), \tag{3.4}$$

$$\forall s \in S_T(t+1), c \in C_{in} \cap C_s \tag{3.5}$$

Algorithm 2 BROKERINTERFACE

```
procedure RELAY(stopCondition)
    while $stopCondition \neq True$ do
        $S_T \leftarrow \textsc{Trigger}(\mathcal{E}, S_T \mathcal{S})$
        RUN($S_T$)
        iter++
    return
procedure TRIGGER($\mathcal{E}, S_T, \mathcal{S}$)
    for all $e_{p\to c} \in \mathcal{E}$ do
        $subs \leftarrow \mathcal{S}[c]$
        for all $s \in subs$ do
            $S_T[s] \leftarrow 1$
procedure RUN($S_T$)
    for all $s \in S_T$ do
        $f_s(e_{p\to c})$
```

3.2.4 PubSub Dynamics

PubSub dynamics begin when some publisher $p \in P$ publishes an event $e_{p\to c}$ to some channel c. Publishing an event updates the set of all events $\mathcal{E}(t)$ at time t. The Broker B subsequently runs RELAY on the $\mathcal{E}(t)$, which then *triggers* Subscribers with incoming events, and finally *runs* the callback function for each triggered Subscriber, on the incoming event(s).

The dynamics of PubSub can be summarized as follows. At any time t, for all $p \in P$, and any channel $c \in C$, if event $e_{p\to c}$ is published:

$$\mathcal{E}(t+1) = \textsc{Publish}\left(\mathcal{E}(t), e_{p\rightarrow c}\right) \tag{3.6}$$

$$S_T(t+1) = \textsc{Trigger}\left(\mathcal{E}(t+1), S_T(t), \mathcal{S}\right) \tag{3.7}$$

$$f_s(e_{p\rightarrow c}) = \textsc{Run}(\mathcal{E}(t+1), S_T(t+1), \mathcal{S}), \tag{3.8}$$

$$\forall s \in S_T(t+1), c \in C_{in} \cap C_s \tag{3.9}$$

Chapter 4

Factor Graph Computing With PubSub

4.1 Formal Mapping

For *programming* in the Factor Graph Compute framework, as an end user, one simply needs to worry about the *Registration* process. This involves defining the factor graphs and associated update functions. The implementation of Factor Graph Computing using PubSub then executes the computation associated with it.

We show how Factor Graphs and their message passing dynamics can be realized via an underlying PubSub system. We then discuss the programming interface for Factor Graphs implemented on top of a PubSub system. We emphasize the simplicity of the programming interface for expressing a wide variety of computations given by Factor Graphs and the scalability of computation achieved by utilizing PubSub. We introduce generic interfaces for various aspects of our system which are independent of any particular implementation of PubSub.

Let $\mathcal{G} = (V, F, E)$ be a Factor Graph, and let $\Pi = (S, P, C, \mathcal{S}, \mathcal{E}(t), B)$ be a PubSub system. Since each edge in $(v, f) \in E$, with $v \in V, f \in F$ has two messages associated with it, $m_{v \to f}$ and $m_{f \to v}$, it is convenient to work with directed edges. We denote the set of *directed* edges as $\tilde{E}$. That is, for every edge $(v, f) \in E$, we have edges $(v, f), (f, v) \in \tilde{E}$.

4.1.1 Factor Graph Structure with PubSub

In order to realize a Factor Graph, we first encode its structure using PubSub Subscribers and channels. First, we let each node $i \in V \cup F$ in $\mathcal{G}$ be a Subscriber $s_i \in S$, and each *directed* edge $(i, j) \in \tilde{E}$ be a channel $c_{ij} \in C$. This gives us, for our PubSub system, $S = V \cup F$ and $C = \tilde{E}$.

4.1.2 Factor Graph Dynamics with PubSub

Factor Graph dynamics involve propagating messages between neighboring nodes and computing new outgoing messages at each node. When a node i sends a message $m_{i \to j}$ to its neighbor j, node j is *triggered* and runs UPDATEVAR or UPDATEFAC on the incoming message.

To capture Factor Graph dynamics in PubSub, we let Factor Graph messages be PubSub events. In order to be notified of events (message) from its neighbors, node i must be *subscribed* to all channels c_{ji} for all $j \in \mathcal{N}(i)$. The set of subscriptions for any particular node i is then $C_{s_i} = \{c_{ji} | \ j \in \mathcal{N}(i)\}$.

PubSub dynamics are induced by publishers *publishing* events to particular channels, and the Broker subsequently relaying the published events and triggering subscribers to run their callback functions.

In order for a node i to propagate a message $m_{i \to j}$ to its neighbor j using PubSub, it must publish $m_{i \to j}$ to any channel subscribed to by s_j.

Thus, having node i publish message $m_{i \to j}$ to channel c_{ij} as event $e_{p_i \to c_{ij}}$, enables propagation of message m_{ij} from node i to node j.

In order to propagate outgoing messages, each node must be also be a Publisher, giving us $P = S = V \cup F$.

Though in general, Publishers can *publish* events to any channel $c \in C$, in the case of Factor Graphs, a particular node i only needs to publish to channels representing its outgoing edges, c_{ij} where $j \in \mathcal{N}(i)$. We can restrict the set of channels any node i can publish to, as the subset $C_{p_i} = \{c_{ij} | \ j \in \mathcal{N}(i)\}$.

After receiving an incoming message $m_{i \to j}$, node j must compute outgoing messages using UPDATEVAR or UPDATEFAC. Since $m_{i \to j}$ is just the event $e_{p_i \to c_{ij}}$, by setting the callback function f_{s_j} associated with s_j to UPDATEVAR or UPDATEFAC, the Broker will trigger node j to compute its outgoing message once $m_{i \to j}$ is published to channel c_{ij}.

4.1.3 Factor Graph Node Updating

Factor Graph computation is accomplished by each node locally running UPDATEVAR or UPDATEFAC on incoming messages. In this section, we specify the programming implementation and interface for realizing the update function at a particular node. The realization of this is given by MESSAGEPASS in Interface 3.

Since the dynamics of message passing, triggering of nodes, and running of callback functions (MESSAGEPASS) for each node is transparently handled by PubSub, only Interface 3 needs to be separately implemented in order to specify how to enable Factor Graph computation at each node.

First, we define the state s_i of a node i as the set of incoming messages from all neighbors of i, the node type, and the node factor function, if relevant.

We realize sets of messages as the collection of pairs:

$$\{(e_{ji}, m_{j \to i}) | \ \forall j \in \mathcal{N}(i)\} \tag{4.1}$$

where e_{ji} is some identifier of edge (j, i). Similarly, The incoming message m_{in} from node j is realized as the pair $(e_{ji}, m_{j \to i})$ and outgoing message m_{out} is realized as the collection of pairs $\{(e_{ij}, m_{i \to j}) | \ \forall j \in \mathcal{N}(i)\}$.

For a particular node i, the function MESSAGEPASS performs three main steps:

1. First, the function UPDATESTATE updates the message value of the pair in s_i which has edge ID corresponding to the edge ID of m_{in} with the message value of m_{in}, and returns the updated state.

2. Next, the function COMPUTEOUTGOING applies UPDATEVAR or UPDATEFAC to the updated state and returns the outgoing messages m_{out}

3. Finally, the function PROPAGATEMESSAGES, sends the messages in m_{out} to the appropriate neighbors of i via PubSub. For each pair $(e_{ij}, m_{i\to j}) \in m_{out}$ the node i publishes the event $e_{p_i \to c_{ij}} = (e_{ij}, m_{i\to j})$.

In general, the details of FETCHSTATE and PERSISTSTATE in STATEINTERFACE will depend on the underlying data store used for managing and persisting state s_i.

The details of UPDATEVAR and UPDATEFAC are application dependent. In Section 4.1.5 a consistent programming interface across applications is provided.

Algorithm 3 NODEINTERFACE

```
procedure MESSAGEPASS($m_{in}, i$)
    $s_i \leftarrow$ UPDATESTATE($m_{in}, i$)
    $m_{out} \leftarrow$ COMPUTEOUTGOINGMESSAGES($s_i$)
    PROPAGATEMESSAGES($m_{out}, i$)
    return
procedure UPDATESTATE($m_{in}$)
    $s_i \leftarrow$ STATE.FETCHSTATE($i$)
    for all $(e_{ki}, m_{k\to i}) \in s_i$ do
        if $e_{ki} == e_{ji}$ then
            $m_{k\to i} \leftarrow m_{ji}$
    $s_i \leftarrow$ STATE.PERSISTSTATE($i, s_i$)
    return $s_i$
procedure COMPUTEOUTGOINGMESSAGES($s_i$)
    if $s_i.type == fac$ then
        $m_{out}$ = UPDATEFAC($s_i.messages, s_i.factor\_function$)
    if $s_i.type == var$ then
        $m_{out}$ = UPDATEVAR($s_i.messages$)
    return $m_{out}$
procedure PROPAGATEMESSAGES($m_{out}$)
    for all $(e_{ij}, m_{i\to j}) \in m_{out}$ do
        PUBLISHER.PUBLISH($c_{ij}, (e_{ij}, m_{i\to j})$)
    return
```

Algorithm 4 STATEINTERFACE

```
procedure FETCHSTATE($i$)
procedure PERSISTSTATE($i$)
```

4.1.4 Factor Graph System

First, Factor Graph computation requires: a PubSub system $\Pi = (S, P, C, \mathcal{S}, \mathcal{E}(t), B)$, a memory store M for handling Factor Graph state on top of which StateInterface is implemented, and a Factor Graph specification. For example, this could be a file $\mathcal{F}_E$ for representing factor graph structure: an adjacency table or adjacency list representing each directed edge in $\tilde{E}$ with edge weights as initial messages, and another file $\mathcal{F}_{V \cup F}$ specifying state for each node in $V \cup F$: node type and factor function (if relevant).

We now describe how computation runs in the entire system for a Factor Graph $\mathcal{G} = (V, F, E)$ implemented on top of a PubSub system.

1. **Initialize PubSub**: Register each edge $(i, j) \in \mathcal{F}_E$ as channel $c_{ij} \in C$, and subscribe each node to its relevant channels. Start the PubSub Broker B.

2. **Load Factor Graph State**: Initialize the state for each node according to entries of $\mathcal{F}_{V \cup F}$. Details will depend on the memory store used for storing and managing state.

3. **Initialize Computation**: Computation begins by having a subset of nodes (such as all variable nodes) publish their initial messages along all neighboring edges, at which point the PubSub Broker starts delivering messages and triggers neighboring nodes to perform their respective computations via MessagePass.

4. **Run Computation**: At each time t, if a particular node i wants to propagate a message m_{ij} over a particular edge (i, j), it publishes the event $e_{p_i \to c_{ij}} = (e_{ij}, m_{ij})$ to the channel c_{ij}. The broker subsequently relays the message to all subscribers of channel c_{ij} (in this case, just node j), triggers node j (the only subscriber of channel c_{ij}), and runs the callback function associated with node j with m as the input, which is just the update function associated with node j.

5. **End Computation**: Finally, computation concludes once all nodes have met some sort of stop criteria. This stop criteria is implementation and application dependent.

6. **Read Result**: The end result of the computation can be read by inspecting the collection of messages in the state of each variable node.

During each time step of the computation, for an event $e_{p_i \to c_{ij}}$ published by node i to channel c_{ij} at time t, the Broker runs RELAY, updating as:

$$\begin{aligned}
\mathcal{E}(t+1) &= \textsc{Publish}\left(\mathcal{E}(t), e_{p_i \to c_{ij}}\right) && (4.2)\\
&= \mathcal{E}(t) \cup \{e_{p_i \to c_{ij}} = (e_{ij}, m_{i \to j})\} && (4.3)\\
S_T(t+1) &= \textsc{Trigger}\left(\mathcal{E}(t+1), S_T(t), \mathcal{S}\right) && (4.4)\\
&= S_T \cup \{s_j\} && (4.5)\\
f_{s_i}(e_{p_i \to c_{ij}}, i) &= \textsc{Run}(\mathcal{E}(t+1), S_T(t+1), \mathcal{S}) && (4.6)\\
&= \textsc{MessagePass}((e_{ij}, m_{i \to j}), i) && (4.7)
\end{aligned}$$

4.1.5 Update Function Interface

Factor Graphs allow us to implement a variety of different computations by simply specifying initial messages and the appropriate UPDATEVAR and UPDATEFAC functions.

In Interfaces 5 and 6, we provide a simple interface for the UPDATEVAR and UPDATEFAC functions. This interface breaks each node's update function into two stages: UPDATENODE and UPDATEEDGE.

The UPDATENODE function for a particular node i computes an aggregate state based on taking all incoming messages from neighbors of i. It takes in a set of edge ID, message pairs M, and returns an aggregate NODESTATE (scalar, vector, etc) depending on the problem.

The function UPDATEEDGE is used to compute the particular message update along an edge, based on the aggregate NODESTATE returned by UPDATENODE. For a particular edge ID e_{ij}, the function UPDATEEDGE takes NODESTATE, N, and the edge, message pair $(e_{ij}, m_{i \to j}) \in M$, to compute the new message.

In the case of factor nodes, the functions UPDATENODEFAC and UPDATEEDGEFAC also take the corresponding factor functions into account when computing new messages.

Each application would have to implement the appropriate UPDATENODE and UPDATEEDGE functions. These functions are independent of Factor Graph structure and dynamics implementation, meaning that they can be ported across Factor Graphs, PubSub systems, and environments, without needing to be changed.

As an example, when considering the case of Sum-Product Belief Propagation, the function UPDATENODEVAR would compute the product of all incoming messages as the node state, and the function UPDATEEDGEVAR would divide the node state (product of all incoming edges) by the incoming message along that particular edge. This gives us the desired outgoing message along edge (i, j) from variable node i to factor node j:

$$\begin{aligned} m_{i\to j} &= \frac{\prod_{n\in\mathcal{N}(i)} m_{n\to i}}{m_{j\to i}} && (4.8) \\ &= \prod_{n\in\mathcal{N}(i)\setminus j} m_{n\to i} && (4.9) \end{aligned}$$

Algorithm 5 UPDATEVARINTERFACE

```
procedure UPDATEVAR(E)
    $n \leftarrow$ UPDATENODEVAR($M$)
    $e \leftarrow []$
    for all $e_c \in M$ do
        $e[e_c] \leftarrow$ UPDATEEDGEVAR($n, M$)
    return $e$
procedure UPDATENODEVAR(M)
procedure UPDATEEDGEVAR(n, m)
```

4.2 Heuristics for Scaling

The Factor Graph Computing implementation using PubSub described above can be made efficient as graph size grows using simple heuristics described next.

Algorithm 6 UPDATEFACINTERFACE

```
procedure UPDATEFAC(M, F)
    $n \leftarrow$ UPDATENODEFAC($M, F$)
    $e \leftarrow []$
    for all $e_c \in M$ do
        $e[e_c] \leftarrow$ UPDATEEDGEFAC($n, M, F[e_c]$)
    return $e$
procedure UPDATENODEFAC(M, F)
procedure UPDATEEDGEFAC(n, m, f)
```

4.2.1 Nodes As Channels

For factor graphs which are *dense*, i.e. the number of edges are much larger than number of nodes ($|E| \gg |V| + |F|$), it makes sense to consider each node (in $V \cup F$) as a channel. Each message update $m_{v \to f}, m_{f \to v}$ is then published to the channel corresponding to the receiving node rather than the corresponding edge. This will help scale PubSub infrastructure better with respect to the number of channels.

We can reduce the total number of PubSub channels by utilizing nodes as channels, rather than edges. In this case, when a publisher p_i, corresponding to a node i in a Factor Graph publishes some event $e_{p_i \to c}$, we let $c = c_i$ rather than $c_{ij}, j \in \mathcal{N}(i)$. Now, each neighbor of i: $j \in \mathcal{N}(i)$ subscribes to channel c_i. Now the set of channels C is exactly the set of nodes in our Factor Graph: $V \cup F$, and the set of subscriptions is the set of edges E of the Factor Graph. Now, events will be sets of edge ID, message pairs: $e_{p_i \to c_i} = \{(e_{ij}, m_{i \to j}) | \, j \in \mathcal{N}(i)\}$. As before, Subscribers can match the relevant edge ID with edge IDs in their state and filter out any messages which are irrelevant and meant for other neighbors node i.

4.2.2 Partitioning

The key characteristic of Factor Graph Compute is that the compute (and data storage) associated with each node is very light. Therefore, it can make sense to "group" a collection of variable nodes and factor nodes as "partitions" in the factor graph so that the resulting "partitioned" factor graph is smaller than the original factor graph, while keeping the size of each partition manageable. This partitioning can be simply done in a greedy manner or can be done cleverly by utilizing graph partitioning methods that minimize the number of edges crossing or minimize the weighted cut size between partitions. In our experiments, we utilize a greedy method to obtain non-trivial performance speedup.

Partitioned Factor Graph Structure

As stated previously, from a Factor Graph $\mathcal{G} = (V, F, E)$ we can in turn build a *Partitioned* Factor Graph, $\mathcal{G}^p = (V^p, F^p, E^p)$. Each variable partition $v^p \in V^p$, and each factor partition $f^p \in F^p$ are just sets of variable and factor nodes, respectively.

Edges between partitions exist if any node in a particular partition has an edge with a node in another partition of our original Factor Graph. Specifically, let v^p be a variable node partition and f^p be a factor node partition. If the edge (v, f) exists, for variable node $v \in v^p$ and factor node $f \in f^p$, then there exists an edge $(v^p, f^p) \in E^p$ between partitions v^p, f^p in $\mathcal{G}^p$.

From our definition of Factor Graph, every variable node v maintains state x_v and every factor node f maintains state y_f. For $\mathcal{G}^p$, each variable partition $v^p \in V^p$ maintains state $x_v^p = \bigcup_{v \in v^p} x_v$ and each factor partition $f^p \in F^p$ maintains state $y_f^p = \bigcup_{f \in f^p} y_f$. The state for each partition can also be represented as EdgeCollections.

For implementation purposes, in order to properly update state, the state of a partition i, s_i needs to capture the relationships between nodes, edges, and respective messages. Within a particular partition, these relationships can be thought of as a tree, shown in Figure 4-1. When an incoming message is received by a partition, only the set of subtrees which contain edges with updated messages, denoted as s_i^δ, is required for computing outgoing messages.

Partitioned Factor Graph Dynamics

The programming interface for *Partitioned* Factor Graphs is given by PartitionInterface in Interface 7. This interface is quite similar to that of NodeInterface except that in the function MessagePass, outgoing messages are computed using s_i^δ rather than s_i.

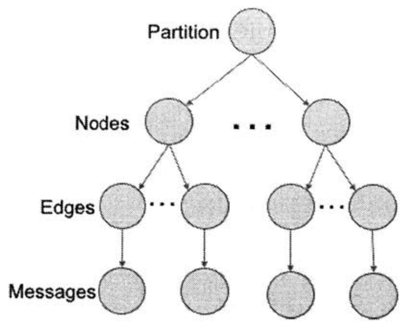

Figure 4-1: Partition Tree

The Partition version of UPDATESTATE takes incoming messages m_{in} and a partition identifier i and outputs s_i^δ. The interface for handling partition state, PARTITIONSTATEINTERFACE, given in Interface 8 is implementation specific, and must handle computing which nodes in the partition need to be updated and returning the appropriate s_i^δ

The function UPDATESTATE for Partitions applies the node version of UPDATESTATE to each node subtree in s_i^δ and returns a set of outgoing messages, m_{out}.

Like in the unpartitioned version, the implementations of PARTITIONSTATEINTERFACE and PUBLISHERINTERFACE depend on the underlying data store and PubSub system used. As the compute, state, and PubSub components are independent of each other, scaling can be done in a similar manner as with unpartitioned Factor Graphs.

Algorithm 7 PARTITIONINTERFACE

```
procedure MESSAGEPASS($m_{in}, i$)
    $s_i^\delta \leftarrow$ UPDATESTATE($m_{in}, i$)
    $m_{out} \leftarrow$ COMPUTEOUTGOINGMESSAGES($s_i^\delta, i$)
    PROPAGATEMESSAGES($m_{out}, i$)
    return
procedure UPDATESTATE($m_{in}, i$)
    $s_i^\delta \leftarrow$ $PartitionState$.UPDATESTATEMESSAGES($i, m_{in}$)
    return $s_i^\delta$
procedure COMPUTEOUTGOINGMESSAGES($s_i^\delta, i$)
    $m_{out} \leftarrow [\,]$
    for all $n \in s_i^\delta$ do
        $m_{out}[n] \leftarrow$ $Node$.COMPUTEOUTGOINGMESSAGES($s_i^\delta[n], n$)
    return $m_{out}$
procedure PROPAGATEMESSAGES($m_{out}, i$)
    $Publisher$.PUBLISH($i, m_{out}$)
    return
```

Algorithm 8 PARTITIONSTATEINTERFACE

```
procedure UPDATESTATEMESSAGES($i, m_{in}$)
```

Chapter 5

Implementation

Using the generic interfaces presented in the previous section, Factor Graphs can be implemented on top of different PubSub systems based on application requirements.

We present a synchronous implementation of Factor Graphs on top of a custom PubSub system we implement using the Ligra graph processing framework, and an asynchronous implementation of Factor Graphs on top of Redis, a popular in-memory store with PubSub capabilities. We choose Ligra as it is a specialized, single-machine graph processing system, and we choose Redis as it is an off-the-shelf, widely-used in memory storage system which can be used on one machine or easily extended to operate over several machines.

We demonstrate how to implement the necessary Interfaces in both Ligra and Redis and utilize the same UPDATEVAR and UPDATEFAC code across implementations.

Implementing Factor Graphs on top of these systems enables them to perform a variety of computations they were not initially meant for: such as optimization, linear algebra, and graph processing.

5.1 Synchronous Factor Graph Computation

5.1.1 Ligra

Ligra [25] is a specialized system for single machine, shared memory, large scale graph processing, written in C. Ligra was built for high performance implementations of traditional iterative graph algorithms such as Breadth-First Search, PageRank, and Connected Components. Ligra computations are run in parallel, using threads, and each iteration of the computation over the input graph happens synchronously. That is, the next iteration over the graph only starts once the current iteration is finished.

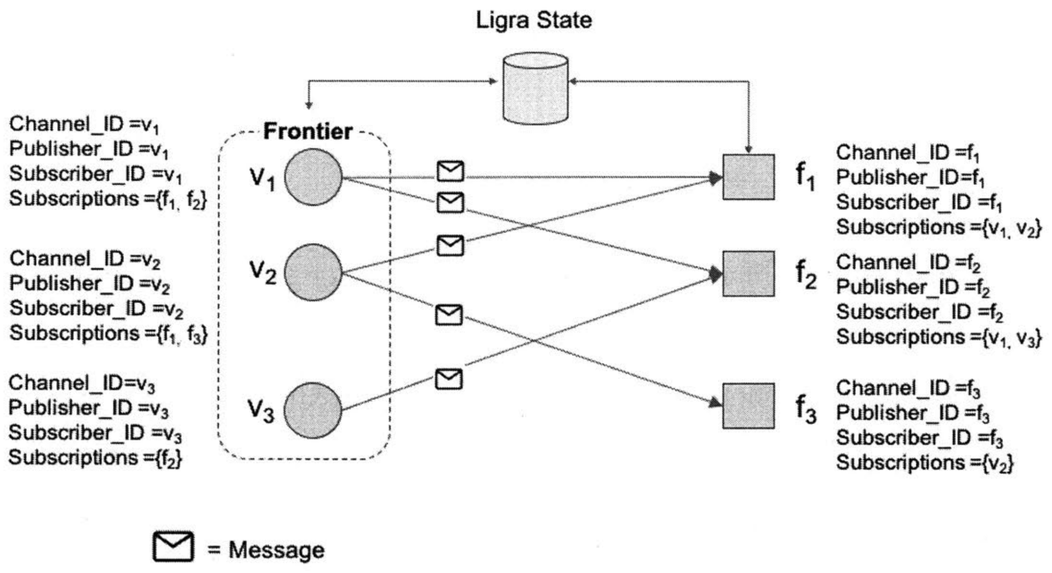

Figure 5-1: Illustration of Factor Graph Compute using PubSub based on Ligra. The implementation of PubSub using Ligra effectively rotates between variables nodes and factor nodes as the frontier; above, variable nodes are depicted as the frontier.

5.1.2 PubSub Messaging in Ligra

Ligra runs computations in synchronous iterations on specified subsets of nodes in the graph. In the case of PubSub, this means that published events are relayed to Subscribers by the Broker and each triggered Subscriber finishes running its callback function, before any new messages can be published and relayed in the next iteration.

The main PubSub BROKER.RELAY method is implemented as a loop in Ligra. Each iteration of BROKER.RELAY first runs BROKER.TRIGGER to determine triggered subscribers and then BROKER.RUN to run callback functions of the triggered subscribers.

The set of triggered subscribers, S_T is represented as a Ligra FRONTIER. A Ligra FRONTIER is a subset of nodes in a graph on which to perform computations via the Ligra method VERTEXMAP.

Since each node in the Ligra graph is a subscriber, the subscriber callback functions can be run in parallel on the current FRONTIER by Ligra's VERTEXMAP method.

In addition, we pass any application dependent variables to the BROKER.RELAY function, so that the Broker and Subscribers have access to them. In particular, for the case of Factor Graph, main memory arrays for Factor Functions and State are passed into the BROKER.RELAY method so that individual subscribers have access. In addition, creating these arrays ahead of time improves performance by avoiding arrays having to be creation of new arrays during each subscriber iteration.

5.1.3 Factor Graph Structure in Ligra PubSub

Since Ligra is already highly optimized to handle very large graphs while processing each node and edge individually, we do not introduce any additional partitioning in our implementation and have each directed edge as its individual channel.

Recall that for some particular factor graph $\mathcal{G} = (V, F, E)$, and directed edges $\tilde{E}$, we want a PubSub system $\Pi_{\text{LIGRA}} = (S, P, C, \mathcal{E}(t), B)$ with:

$$S = P = V \cup F \tag{5.1}$$

$$C = \tilde{E} \tag{5.2}$$

Each node $i \in V \cup F$ of $\mathcal{G}$ is both a Publisher and Subscriber. Each directed edge $(i, j) \in \tilde{E}$ represents a channel. A node i is subscribed to channel c_{ij} if $j \in \mathcal{N}(i)$.

Ligra keeps track of directed graph structure and provides interfaces to access edges, nodes, and edge weights. We utilize Ligra itself to store PubSub subscriptions via graph edges and we track events via edge weights. That is, the event $e_{p_i \to c_{ij}} \in \mathcal{E}(t)$ is stored in Ligra as the current weight of edge (i, j). Publishing is accomplished by using standard Ligra interfaces to write events as edge weights.

5.1.4 Factor Graph Computation in Ligra

Implementing full Factor Graph computation in Ligra first requires a PubSub system, a memory store M, and a Factor Graph specification. The Ligra system itself is used as PubSub, Factor Graph structure specification is represented as a file in the adjacency graph format [1], $\mathcal{F}_E$ with edge weights as initial messages. Factor Graph state specification is a seperate file $\mathcal{F}_{V \cup F}$ with node IDs mapped to node type and factor function, if relevant.

Factor Graph state involves keeping track of incoming messages, node type, and factor function for each node. Messages, which are PubSub events, are already stored in Ligra as edge weights. Factor Functions are stored in Ligra as a C array, M_{factor}. For a particular node i, its node type is inferred by whether or not the entry $M_{factor}[i]$ exists - a node is a factor node if it exists and variable node otherwise.

Below, we detail the steps of Factor Graph computation in Ligra.

1. **Initialize PubSub**: PubSub is initialized in Ligra by first creating a Ligra GRAPH from $\mathcal{F}_E$. Since the Ligra GRAPH structure encodes vertices and their edges, no additional work needs to be done in order to specify PubSub subscriptions.

2. **Load Factor Graph State**: Initialize the array M_{factor} from $\mathcal{F}_{V \cup F}$

3. **Initialize Computation**: The subset of initially triggered subscribers, $S_T(0)$, which will kick off PubSub are specified as the initial Ligra FRONTIER. This is typically the set of variable nodes V for Factor Graph computation. PubSub computation begins by passing the initial FRONTIER to the BROKER.RELAY method which begins a loop in Ligra for iterative computation.

4. **Run Computation**: As the set of triggered subscribers is already specified ($S_T(0)$), the function BROKER.TRIGGER can be skipped in the initial iteration. Otherwise, the function BROKER.TRIGGER updates the FRONTIER with the latest set of triggered nodes. The function BROKER.RUN runs callback functions for each triggered subscriber (node) in parallel by using Ligra's VERTEXMAP function on the current FRONTIER. For Factor Graph Computation, the method MESSAGEPASS and the initial FRONTIER are passed into VERTEXMAP to compute and publish outgoing messages for each node in the FRONTIER.

5. **End Computation**: Subsequent iterations are run by continuing BROKER.RELAY until the stop condition is reached (i.e. after a certain number of iterations have been completed), which ends the computation.

6. **Read Result**: The result of the computation can be read from the final state of variable node entries by inspecting the out weights of edges of variable nodes.

Synchronous Factor Graph computation using a shared memory Broker like Ligra is well-suited and optimized for single machine computation, when all data (the entire Factor Graph) can fit in memory. However, when data is too big to fit in memory, computation must happen across multiple machines. In order to handle this case, we present an asynchronous implementation of Factor Graph computation in the next section, which is suitable for multi-machine environments.

5.1.5 Broker Interface in Ligra

The PubSub Broker RELAY method is implemented as a loop in Ligra, which runs for a specified number of iterations. Before beginning the loop, the FRONTIER is initialized to an initial subset of subscribers $S_T(0)$. For Factor Graph computation, this is usually the set of variable nodes. When an initial FRONTIER is provided in the first iteration of RELAY, the function TRIGGER does not need to be run, but is run for every iteration after.

The OUTBOX keeps track of which publishers have published messages, and the INBOX keeps track of which subscribers are to be triggered.

The function TRIGGER first determines the set of subscribers which are to be triggered, through the method INCOMINGMESSAGES. This message is implemented via Ligra's EDGEMAP interface, which, in parallel, applies the function CHECKFORINCOMING over every directed edge in our graph.

Given a directed edge (i, j), CHECKFORINCOMING sets INBOX$[i]$ to INBOX$[i]$ `OR` OUTBOX$[j]$. If INBOX$[i]$ is set to 1, then node i has incoming messages from its subscribed channels, and will subsequently run its callback function.

Next, the OUTBOX is reset, that is, every entry is set to 0, indicating there are no outstanding published messages,

Finally, the method TRIGGER updates the FRONTIER based on the updated values of INBOX. Since, in our PubSub system, we have $P = S$, that is, every subscriber is publisher, we SWAP the values of INBOX and OUTBOX. This is because every subscriber will subsequently have a message to publish.

Algorithm 9 BROKERINTERFACELIGRA

```
procedure RELAY(maxIters, inbox, outbox, Frontier)
    $iter \leftarrow 0$
    while $iter < maxIters$ do
        if $iter > 0$ then
            $Frontier \leftarrow$ TRIGGER($Frontier, inbox, outbox$)
        RUN($Frontier$)
        iter++
    return
procedure TRIGGER(i,j, inbox, outbox)
    EDGEMAP($Frontier$, INCOMINGMESSAGES($inbox, outbox$))
    $Frontier \leftarrow Frontier(inbox)$
    VERTEXMAP($Frontier$, PRVERTEXRESET($outbox$))
    SWAP($outbox, inbox$)
    return $Frontier$
procedure RUN(Frontier)
    VERTEXMAP($Frontier, G, Subscriber.listen$)
procedure INCOMINGMESSAGES(i,j, inbox, outbox)
    $inbox[i] \leftarrow inbox[i]$ OR $outbox[j]$
```

5.1.6 Subscriber Interface in Ligra

Graphs in Ligra consist of directed edges, so every undirected edge (i, j) in our factor graph, is stored twice in Ligra, as (i, j) and (j, i) (once for each direction). As part of its GRAPH structure, Ligra provides interfaces for accessing the degree of each node as well as its (directed) neighbors and weights.

A message from a node i to node j is stored in Ligra as the weight of edge (i, j), and can be accessed using Ligra's GETWEIGHT and SETWEIGHT functions.

The function RUNCALLBACKFUNCTION runs the supplied callback function f on the incoming messages m_{in} and the current vertex V_i. In our implementation, we pass the function MESSAGEPASS as the callback function.

5.1.7 Publisher Interface in Ligra

Each published event $e_{p_i \to c_{ij}}$ is stored as the *weight* of edge (i, j). In Ligra, this is implemented by setting the OUTWEIGHT of node N_i with respect to neighbor j. Edge weights in Ligra can be accesed and set through a Ligra VERTEX interface. This is done by the Ligra VERTEX methods GETOUTWEIGHT and SETOUTWEIGHT. The method PUBLISH thus takes the Ligra VERTEX associated with the publisher, channel (edge) to publish on, event (message) to publish and updates the appropriate OUTWEIGHT.

Algorithm 10 PUBLISHERINTERFACELIGRA

```
procedure PUBLISH($j, e_{p_i \to c_{ij}}, V_i$)
    $V_i$.SETOUTWEIGHT($j, e_{p_i \to c_{ij}}$)
    return
```

5.1.8 Factor Graph State Interface in Ligra

Since the Ligra implementation of PubSub is synchronous, the set of published messages for a particular node (incoming edge weights) is exactly its state (incoming messages from neighbors). Thus, no extra work is needed to implement STATEINTERFACE, as getting and setting edge weights is enough to access state.

Since factor functions are static and do not change over the course of a computation, factor functions are kept in main memory using array M_{factor} in Ligra. The factor function of node i is stored as the array entry $M_{factor}[i]$. If a particular node is a factor node, it contains an entry in M_{factor}, and if a node is a variable node, it does not contain an entry in M_{factor} array. For factor functions which can be represented as vectors, each factor node entry of the M_{factor} is a pointer to an array containing the factor function entries. Otherwise, if the factor function can be represented as a scalar, each entry corresponding to a factor node contains the factor function scalar itself.

In order to improve performance, an array M_{state} is pre-initialized before computation starts. For a particular node i, its entry in M_{state} is a pointer to an array. This avoids having to create arrays to hold all incoming messages each iteration, when running MESSAGEPASS.

Persisting state (in memory) is not needed, as this is done by Ligra when writing edge weights via publishing. The STATEINTERFACE for Ligra is given in Interface 11.

Algorithm 11 STATEINTERFACELIGRA

```
procedure FETCHSTATE($V_i$, $M_{state}$)
    $s_i \leftarrow M_{state}[i]$
    for all $j \in$ GETINDEGREE($V[i]$) do
        $s_i[j] \leftarrow$ GETINWEIGHT($V_i$, $j$)
    return $s_i$
```

5.1.9 Node Update Functions in Ligra

Each subscriber implements NODE.MESSAGEPASS as its callback function. All methods in Interface 3 are implemented in C.

Computation of message updates by UPDATEVAR and UPDATEFAC is accomplished by implementing Algorithm 16 in Cython [10]. Cython enables writing Python code which runs as C code and can be imported into C programs.

This allows us to write our UPDATEVAR and UPDATEFAC functions once and port it across Python and C implementations without worrying about details of the underlying PubSub system.

The main memory arrays M_{state} and M_{factor} are accessible by MESSAGEPASS in Ligra as they are passed to the underlying SUBSCRIBER via the VERTEXMAP method used in BROKER.RUN.

5.2 Asynchronous Factor Graph Computation

5.2.1 Redis

In addition to our Ligra implementation of PubSub, we also present an implementation in Redis [23], a popular in-memory key-value store which also has out-of-the-box PubSub capabilities.

While Ligra is meant for single machine graph processing and handles in-memory storage and compute, Redis has no compute capabilities but its storage and PubSub capabilities can be utilized on a single machine or extended to work with multiple machines over a network.

Unlike with Ligra, our implementation over Redis is completely asynchronous.

Our Redis implementation is highly unoptimized as compared with our Ligra implementation (as Redis is a general purpose system, not necessarily meant for fast compute). However, we emphasize Redis's ease of implementation, setup, and ability to scale to multiple machines. All interfaces are implemented in Python.

We are able to port our same UPDATEVAR and UPDATEFAC code from the Ligra implementation, as it implemented in Cython for use with Redis PubSub. Callback functions are implemented in Python, and use the Redis-Py [2] library to handle interfacing with Redis.

5.2.2 PubSub Messaging in Redis

Redis provides its own PubSub interface. Redis is its own Broker which runs as its own process. Through its PubSub interface, the Redis PubSub Broker handles relaying events over channels and specifying and storing subscriptions, assigning Subscriber callback functions, and publishing messages.

Triggering is handled by Subscribers, rather than by the Broker. Each Subscriber runs as its own process which periodically checks the Redis PubSub Broker for relevant published events to see whether it has been triggered. If a Subscriber has been triggered, it runs its assigned callback function on the relevant event, which is relayed by the Redis Broker.

Redis only publishes events which are strings. In our Python implementation, all outgoing events $e_{p \to c}$ are first encoded as strings strings before being published. This is given in PUBLISHERINTERFACEREDIS in Interface 12.

On a single machine, we utilize Unix sockets for PubSub communication with the Redis Broker. For multi-machine implementations, it is possible to utilize TCP connections for communication with the Redis Broker.

Algorithm 12 PUBLISHERINTERFACEREDIS

```
procedure PUBLISH($c, e_{p \to c}$)
    REDISPUBLISH($c$, PICKLE($e_{p \to c}$))
    return
```

5.2.3 Factor Graph Structure in Redis PubSub

Our asynchronous Factor Graph is implemented in Redis and Python, which unlike Ligra, are not optimized for dealing with large graphs operations. In order to reduce complexity, our Redis implementation utilizes node partitioning, so each Subscriber is a Partition Node, and the total number of Partitions corresponds to the total number of processors or compute units on our system.

In order to minimize the number of subscriptions which Redis needs to track, we use Partition Nodes as channels, rather than edges. Each subscription now becomes an edge between Partition Nodes.

In the case that a Partition Node Subscriber receives a message which does not contain updates for any of its edges. The function UPDATESTATE would return an empty set as s_i^δ if that is the case and no further upating would be necessary.

5.2.4 Factor Graph Computation in Redis

We implement asynchronous Factor Graph computation using Redis as the PubSub system, with one process per Subscriber (Partition Node). The state for each Partition Node is represented as a Python object and stored in process memory.

Like our synchronous implementation, Factor Graph structure can be specified as a file in the adjacency graph format [1], $\mathcal{F}_E$ with edge weights as initial messages. Factor Graph state specification is a seperate file $\mathcal{F}_{V \cup F}$ with node IDs mapped to node type and factor function, if relevant. In addition, Partition Nodes are specified in a separate file $\mathcal{F}^P_{V \cup F}$ where each line contains a Partition ID mapped to IDs of nodes contained within that partition. Partition edges are specified in a file $\mathcal{F}^P_E$ which has a line for each edge between partitions. The identifier (Partition ID) for each partition is stored in the name of the process associated with the particular Partition.

The Subscriber callback function which implements PARTITIONINTERFACE is written in Python. We give both Python and Cython implementations of the function COMPUTEOUTGOINGMESSAGES for Partitions and for Nodes.

The set of outgoing messages m_{out} and incoming messages m_{in} are represented as NumPy Arrays [31]. Each edge ID, message pair $(e_{ij}, m_{i \to j}) \in m_{out}, m_{in}$ is represented as a row in the NumPy Array.

Since MESSAGEPASS is the callback function for each Partition Node, the underlying Subscriber is relayed m_{out} via the Redis Broker.

Publishing via the PROPAGATEMESSAGES is handled by Redis's PUBLISH command.

Below, we detail the process of Factor Graph computation in Redis:

1. **Initialize PubSub**: PubSub is initialized by starting one subscriber per partition ID by reading $\mathcal{F}^P_{V \cup F}$. Specifying subscriptions is done by reading in $\mathcal{F}^P_E$ and handled by the Redis PubSub interface. Each Subscriber and the Broker are started as their own processes by Redis.

2. **Load Factor Graph State**: State for each partition is initialized by creating a Python object for state for each partition. Details of how state is stored as

in-memory Python objects is given in the next section.

3. **Initialize Computation**: Computation is initialized by first having each variable partition publish its initial message via Redis PubSub interface, after which the Redis PubSub Broker takes care of relaying subsequent events.

4. **Run Computation**: Each subscriber periodically checks the Redis Broker to see whether it has been triggered. If a particular subscriber (partition) has been triggered, it runs its callback function MESSAGEPASS, with m_{in} supplied from the Redis Broker. As each partition node subscriber is its own process, computation is run in parallel across all partitions. Unlike the Ligra implementation, computation is asynchronous - subscribers are triggered as they are ready, rather than waiting for a global notion of iteration to complete.

5. **End Computation**: Computation ends when each Partition node has reached its stop condition (i.e. after a certain number of iterations have been completed for a particular node). The status of stop condition (such as a counter) can be stored in the Python object representing partition state.

6. **Read Result**: The result of the computation can be read from the final state of variable node for each variable partition.

This asynchronous implementation of Factor Graph computation is well-suited for multi-machine computations. When data (Factor Graph) is too large to fit in memory on a single machine, Partitions can be distributed across several machines. The advantage of PubSub is that each machine can operate individually and does not need to be aware of the global topology. A Broker such as Redis is well-suited for coordinate communication across multiple machines.

5.2.5 Factor Graph State

We use Python objects to represent the state of each partition. Python objects are stored in process-memory. This guarantees that only the current process can access and modify its state. The function FETCHSTATE returns the Python object for the state of the appropriate partition.

The State object stores the partition's *tree* in two collections, E and N. The collection E captures relationships from edges to nodes and messages. The collection N stores relationships between nodes and edges. The state of a partition can be given as $s_i = E \cup N$. We define $N^\delta \subset N$ as the subset of nodes with updated edge messages. The partition state to update can be given as $s_i^\delta = E \cup N^\delta$. We give two implementations of the State object: one in pure Python, and one which is optimized to be used for Cython implementations of COMPUTEOUTGOINGMESSAGES functions.

Algorithm 13 REDISPARTITIONSTATEINTERFACE

```
procedure UPDATESTATEMESSAGES($i, m_{in}$)
    $s_i \leftarrow$ FETCHSTATE($i$)
    $N^\delta \leftarrow [\,]$
    for all $e, m \in m_{in}$ do
        if Cython is True then
            $e \leftarrow$ CONVERTIDTOROW($s_i, e$)
        $updated \leftarrow$ UPDATEEDGEMESSAGELEAF($s_i, e, m$)
        if updated is True then
            $N^\delta[n] \leftarrow$ FETCHEDGENODETOUPDATE($s_i, e$)
    $s_i^\delta \leftarrow (N_i^\delta, s_i.E)$
    return $s_i^\delta$
procedure UPDATEEDGEMESSAGELEAF($s_i, e_{id}, m$)
    if $s_i.E[e_{id}] \neq \emptyset$ then
        $s_i.E[e_{id}][m] \leftarrow m$
        return True
    else
        return False
procedure FETCHEDGENODETOUPDATE($s_i, e_{id}$)
    $n_{id} \leftarrow s_i.E[e_{id}][n_{id}]$
    $n \leftarrow s_i.N[n_{id}]$
    return n
procedure CONVERTIDTOROW($s_i, e_{id}$)
    $e_{row} \leftarrow s_i.C[e_{id}]$
    return $e_{row}$
```

5.2.6 Python

In a pure Python implementation of State, both collections E and N can be represented as Python dictionaries.

The dictionary E maps edge IDs to its parent node's ID and its latest incoming message. This enables fast lookups for checking which nodes in the partition have new incoming messages in FETCHEDGENODETOUPDATE and fast updates for updating the incoming message for a particular edge in UPDATEEDGEMESSAGELEAF. Entries in E can be thought of as representing the mappings $e_{id} \mapsto (n_{id}, m)$.

The dictionary N maps the ID of each node in the partition to a collection of its edge IDs, which can be represented as a Python set or list. Entries in N can be thought of as representing the mappsings $n_{id} \mapsto \{e_{ids}\}$.

The function UPDATEEDGEMESSAGELEAF updates the message of a particular edge in E, given the new incoming message m, and the ID of the edge to update, e.

The function FETCHEDGENODETOUPDATE first retrieves n_{id}, the ID of the node in the partition associated with edge with ID e_{id} and then retrieves the collection of edge IDs associated with n_{id}, which is returned at added to N^{δ}.

The final state returned, s_i^{δ} consists of both N^{δ} and E.

5.2.7 Cython

For our Cython implementation of COMPUTEOUTGOINGMESSAGES, the collections E and N for a particular partition are represented as Numpy arrays, since Cython has out of the box support for Cython. Rather than indexing E and N by the ID of the respective edge or node, we index them by their row number in E or N.

Each row of the the Numpy array for E represents a particular edge e, consisting of the edge ID, the row in N of the node in the partition associated with e, and the latest incoming message m. The collection E can be thought of containing the mappings $e_{row} \mapsto (e_{id}, n_{row}, m)$.

Each row of the Numpy array for N represents a node in the partition, and each column represents an edge in the partition.

A particular entry $N_{ij} \in N$ is 1 if edge j belongs to node i, and 0 otherwise. Edges are stored as columns in the same order they appear in E, meaning the jth column of N corresponds to the jth row of E. Entries in N can be thought of as representing the mapping $n_{row} \mapsto \{e_{rows}\}$.

Numpy arrays with rows of variable length are represented as Numpy arrays of Python lists. Passing Python objects into Cython functions slows down the implementation, so the Numpy array N is maintained as a rectangular array of integers. The array N is typically sparse, and can be represented as a sparse array. Sparse arrays can be represented in COO or CSR format, and for easy access by Cython, can be broken into three separate Numpy arrays consisting of entries, non-zero rows, and non-zero columns. Since entries in N are 0 or 1, only the arrays of non-zero rows and non-zero columns are needed.

Since incoming and outgoing messages are indexed by edge IDs, each State object in this implementation also includes a Python dictionary C which maps edge IDs to edge rows in the function ConvertIDToRow.

Chapter 6

Experiments

In this section, we describe experiments performed to validate properties of the Factor Graph Compute framework in practice. To establish that the framework is versatile and can solve a variety of different problems, we present experiments for three important classes problems: (a) Integer Optimization, (b) PageRank, and (c) Singular Value Decomposition.

First, using the Factor Graph Compute framework, we show that a simple variant of the belief propagation algorithm can solve optimization problems. Our implementation performs competitively with respect to a state-of-the-art commercial solver for a range of optimization problems including easy, hard and challenging.

Next, we show that for PageRank computations, the Factor Graph Compute framework with Ligra as the underlying PubSub system performs comparably with respect to Ligra[25, 27, 26]. This suggests that our framework does not add much inefficiency on top of the underlying PubSub. It should be noted that Ligra has benchmarked excellent performance for PageRank computation against a variety of other solutions [25, 27, 26].

Finally, we find that our single machine implementation of Factor Graph Compute using Ligra PubSub does >13x better for computing Singular Value Decomposition of a large matrix compared to that of Spark, with respect to similar size matrix reported in the literature. We could not produce this benchmark and hence we are using Spark's performance for data point reported in the literature for similar matrix.

Our experiments utilize the following datasets: two (hard and challenge) binary optimization problems from benchmark datasets [18], the popular Orkut graph to compute PageRank and a very large sparse matrix to test scalability of our framework by computing its singular value decomposition.

6.1 Optimization

Here we describe our optimization algorithm in the Factor Graph Compute framework. The algorithm is an adaptation of the belief propagation algorithm for inference for graphical models. Consider an optimization problem in the canonical form:

$$\max \sum_{i=1}^{n} w_i x_i \quad \text{over} \quad \vec{x} \in \mathbb{R}^n, \vec{b} \in \mathbb{R}^m, \mathbf{A} \in \mathbb{R}^{m\times n} \tag{6.1}$$

$$\text{subject to} \qquad \mathbf{A}\vec{x} \leq \vec{b} \tag{6.2}$$

This is equivalent to finding Maximum A Posteriori (MAP) with respect to distribution

$$p(\vec{x}) = \frac{1}{Z} e^{\sum_{i=1}^{n} w_i x_i} \prod_{j=1}^{m} \mathbf{1}_{\{\vec{a_j}^\intercal \vec{x} \leq b_j\}} \tag{6.3}$$

This suggests that we can use the max-product belief propagation algorithm, implemented in our Factor Graph Compute framework, for solving the associated optimization problem.

To evaluate the performance of the algorithm, we focus on binary optimization problems. The belief propagation algorithm is a *heuristic* for binary optimization and *not* an exact solution, with the exception of a few special cases (tree-structured factor graphs) [13]. To account for this, we utilize various modifications to the original belief propagation algorithm: *Variable Node Decimation* to speed up convergence, and *Sampling* to speed up execution.

6.1.1 Variable Node Decimation

In practice, *loopy* belief propagation is not guaranteed to converge. In order to both guarantee and speed up convergence, we implement a *Decimation* procedure. Decimation is a procedure to *fix* the beliefs of particular variables while running belief propagation, to avoid cycling of values. At a high level, our decimation algorithm works as follows:

Following the execution of the belief propagation algorithm for the binary MAP problem, the messages at each variable node can be interpreted as a relative "score" for the value of each variable (its probability of taking on a value of 1 vs 0) in the optimal solution. In the context of belief propagation, this is simply the product of messages coming to the node from all of its associated factor nodes. In addition, each variable node is assigned a "proposed value" which is just its highest probably value (1 or 0), where the probability is given from incoming messages as above. We only "decimate" or "fix" values for those variable nodes with a "proposed value" of 1; effectively we treat assignment of value 0 to a variable as the default or natural.

In each iteration of the decimation procedure, we consider the top K variables (where K is a pre-specified parameter, usually a small value), sorted in descending order of their probability of being 1.

Intuitively, if a node believes that it should be 1 with probability close to 1, it might make sense to "fix" its value as 1, subject to its constraints.

As constraints are encoded by factor nodes, we use them to decide whether to fix a particular variable node value. That is, once we choose the top K variables as above, each factor node with which it interacts casts a "vote" as to whether a particular node should be fixed. Each factor node votes on its neighbor variables in a greedy, iterative manner. A factor node sorts all of its variable neighbors by their probability of being 1. It then sequentially checks if setting them to 1 will violate the constraint or not. If it violates the constraint, then the factor node votes against that particular neighboring variable node being set to 1; else it votes for the neighboring variable node being set to 1.

If a variable node, amongst the top K chosen, receives unanimous positive votes from all neighboring factors, it will set its value to 1 and it will be "decimated" (or fixed) for the duration of the computation. The decimated variable nodes effectively stop participating in the algorithm.

Decimation Algorithm

Our decimation procedure is run after every variable node has processed t iterations. After decimation is run over all variable nodes, the message passing procedure continues. The decimation algorithm is given in 14:

Algorithm 14 BP-DECIMATE

```
procedure DECIMATENODES(V, F, n)
    $V_s \leftarrow []$
    $V_{f_s} \leftarrow []$
    $V_d \leftarrow []$
    for all $v \in V$ do
        $V_s[v] \leftarrow$ ASSIGNSCORE($v$)
    for all $f \in F$ do
        $V_{f_s}[f] \leftarrow$ ASSIGNFACTORSCORE($V_s, f$)
    $d \leftarrow 0$
    for all $v \in V$ do
        if SORT($V_{f_s}$)[$v$] $= 1$ and $d < n$ then
            $V_d[d] \leftarrow V_s[v]$
            $d \leftarrow d + 1$
    for all $v_d \in V_d$ do DECIMATENODE($v_d$)
```

During decimation, each variable node i is assigned a score by the function ASSIGNSCORE. In the case of binary programming, the score is just the probability that a particular variable node takes on a value of 1. This is computed using node i's incoming messages as the normalized value of $\prod_{j \in \mathcal{N}(i)} m_{j \to i}(1)$, which corresponds to node i's estimated marginal distribution thus far. Each node also keeps track of its `decimated marginal`, which is just node i's highest probability value.

The function ASSIGNFACTORSCORE is applied to every factor node. For every factor node j, this function checks whether the `decimated marginal` values of the neighbors of j violate the factor function constraint. All `decimated marginal` values of neighbors of j are sorted, in descending order, by their score. For each neighbor, i, of j, Each factor node computes whether its factor function is violated by its variable neighbor's current decimated marginal. Each edge (i, j) is assigned a factor score of $+1$ if the decimated marginal of node i satisfies the factor function constraint of node j and -1 otherwise.

Variables with decimated marginals which do not violate any factor functions are eligible for decimation.

Next, variable nodes are sorted in decreasing order by score. The top num_to_decimate variable nodes whose decimated marginal does not violate any factor neighbors, are chosen to be decimated (set to their decimated marginal), and can no longer change for the remainder of running Max-Product Belief Propagation.

In our PubSub implementation, we implement the decimation procedure as a separate subscriber, subscribed to all partition channels. Once each node has updated its value t times, the decimation procedure is run before continuing with Max-Product Belief Propagation. Edges are represented as Python dictionaries which keep track of the current score, decimated marginal, factor scores, and decimation status values. All values of decimation status are initialized as False at the beginning. values of score, decimated marginal, and factor scores are reset for undecimated variables after each round of decimation.

6.1.2 Speeding Up By Sampling

In order to give a practical implementation of the factor update function for Max-Product Belief Propagation, we introduce a modified algorithm BP-SAMPLE, based on sampling.

Recall the message update equation (2.17) from factor node to variable node. For binary optimization, generating each factor to variable message requires solving the following maximization problem: let $n_f = |\mathcal{N}(f)|$ be the degree of factor node f;

probability $w_k \in [0,1]$ for $1 \le k \le n_f$; and function $g : \{0,1\}^{n_f} \to \mathbb{R}_+$.

$$\begin{aligned} \textsf{maximize} \quad & g(\sigma_1, \ldots, \sigma_{n_f}) \prod_{k=1}^{n_f} w_k^{\sigma_k}(1-w_k)^{1-\sigma_k} \\ \textsf{over} \quad & \sigma_k \in \{0,1\},\ 1 \le k \le n_f. \end{aligned} \tag{6.4}$$

Exactly Solving (6.4) requires enumerating over 2^{n_f} values. Since this computation quickly becomes unwieldy as n_f becomes large, we can utilize a simple approximation that is effective for large n_f: sample $r \ll 2^{n_f}$ binary strings from $\{0,1\}^{n_f}$ from the following joint probability distribution where the probability of a particular binary vector $(\sigma_k)_{1 \le k \le n_f} \in \{0,1\}^{n_f}$ is given by

$$\mathbb{P}(\sigma_1, \ldots, \sigma_{n_f}) = \prod_{k=1}^{n_f} w_k^{\sigma_k}(1-w_k)^{1-\sigma_k} \tag{6.5}$$

Let $\mathbf{Y} \subset \{0,1\}^{n_f}$ be this random subset. Then we simply maximize over this sampled set, i.e.

$$\begin{aligned} \textsf{maximize} \quad & g(\sigma_1, \ldots, \sigma_{n_f}) \prod_{k=1}^{n_f} w_k^{\sigma_k}(1-w_k)^{1-\sigma_k} \\ \textsf{over} \quad & (\sigma_k)_{1 \le k \le n_f} \in \mathbf{Y}. \end{aligned} \tag{6.6}$$

We can interpret each message $m_{v_k \to f}, v_k \in \mathcal{N}(f)$ as a probability distribution over $\{0,1\}$, which is just the probability distribution of σ_k. In particular, we have $m_{v_k \to f}[0] = \mathbb{P}(\sigma_k = 0)$ and $m_{v_k \to f}[1] = \mathbb{P}(\sigma_k = 1)$. From this, we get each term of the form $w_k^{\sigma_k}(1-w_k)^{1-\sigma_k}$ in 6.5. Thus, we can compute 6.6 by just sampling values from the appropriate incoming messages $m_{v_k \to f}$.

Our algorithm based on this method essentially computes 6.6 as

$$\begin{aligned} & \frac{1}{|\mathbf{Y}|} \max_{\tilde{\mathbf{y}}_j \in \mathbf{Y}} \{ g(\tilde{\mathbf{y}}_j) \textsc{Freq}(\tilde{\mathbf{y}}_j) \} \\ = & \frac{1}{|\mathbf{Y}|} \max_{\substack{g(\tilde{\mathbf{y}}_j)=1 \\ \forall \tilde{\mathbf{y}}_j \in \mathbf{Y}}} \{ \textsc{Freq}(\tilde{\mathbf{y}}_j) \} \end{aligned} \tag{6.7}$$

where $\mathbf{Y}$ is the set of samples from the joint probability distribution given by $\prod_{v_k \in \mathcal{N}(f)} m_{v_k \to f}$ and $g(\tilde{\mathbf{y}}_j)$ is the factor function evaluated at sample $\tilde{\mathbf{y}}_j \in \{0,1\}^{n_f}$.

BP-Sample Algorithm

We now describe our algorithm for computing Max Product belief propagation factor functions based on sampling, in Algorithm 15. First, the function GENERATESAMPLES, takes in a set of incoming messages, M and number of samples to generate, n, and then generates n samples from the joint distribution over M. This method returns an array of of n samples, where each sample is a vector of length $|M|$. In the case of binary programming, each sample is a vector in $\{0,1\}^{|M|}$.

Algorithm 15 BP-SAMPLE

```
procedure GENERATESAMPLES(M, n)
    S ← []
    for all $m \in M$ do
        $S[m] \leftarrow$ SAMPLES($n, m, \{0,1\}$)
procedure BPFACTORUPDATEFROMSAMPLES(S, f)
    $S_c \leftarrow$ UNIQUEWITHCOUNTS($S$)
    $f_{max} \leftarrow 0$
    for all $(s,c) \in S_c$ do
        $f_c \leftarrow f(s) * c$
        if $f_c > f_{max}$ then
            $f_{max} \leftarrow f_c$
    return $f_{max}$
```

The function BPFACTORUPDATEFROMSAMPLES, takes an array of samples, S, and a particular factor function f and computes 6.7. First, the function UNIQUEWITHCOUNTS takes in S and outputs a set S_c of pairs (s,c) where each s is a unique sample from S and c is the count of the number of times the unique sample s occured in S. Then, for each $(s,c) \in S_c$, $f(s)$FREQ(s) is computed by evaluating the factor function f at s and multiplying by c. The maximum value, corresponding to the frequency of the mode, so far, is kept track of as f_{max} and $\max_{s \in S}\{f(s)\text{FREQ}(s)\}$ is returned after termination. This value can be normalized before being returned.

BP-Sample Algorithm Performance

For an instance of an optimization problem, we provide the comparison of speed up achieved by this approximation as well as the approximation error introduced in the message values by the algorithm. Specifically, Figure 6-1 presents the speedup achieved through the sampling approximation and the approximation error.

In Figure 6-1 (upper left hand side) we show the Log of the speedup by using the sampled version of Belief Propagation compared to the original algorithm, as the number of factor node neighbors increasesfrom 1 to 20 for binary variables. In optimization, this corresponds to the number of variables in a particular constraint. We note that there is exponential speedup as the number of neighbors increases. In Figure 6-1 (upper right hand side) we show the normalized ℓ_2-norm of the error by using the sampling algorithm versus the original algorithm. This is the ℓ_2-norm of the difference between answer from the sampling algorithm and the original algorithm, divided by $\sqrt(2)$ which is the maximum ℓ_2 norm error. Factor Function constraints were of the form $\sum_{i=1}^{n} x_i \leq r$ where n is the number of factor node neighbors/constraint variables and $r \in \{1, \dots, n\}$ is an integer selected uniformly at random. The upper bound for the error in Figure 6-1 is about 0.4, while most values of the error is close to 0. The lower the value of r (i.e. close to 1) the harder it is to sample relevant events as the number of neighbors/variables goes up. The upper bound on the error just corresponds to not sampling rare events for lower values of r and the sampled message being returned as $[0.5, 0.5]$. The speedup by using this method is exponential and we later show that using this method still results in high quality solutions.

BP-Sample Interface

We now give the programming interface for implementing Sampled Max-Product Belief Propagation. For the interface, we only need to define the appropriate UPDATE functions for our variable and factor nodes:

For a variable node $v_i \in V$, the function UPDATENODEVAR takes a set of incoming messages M, a weight w and computes a new outgoing message $m_{v_i \to f}(n) =$

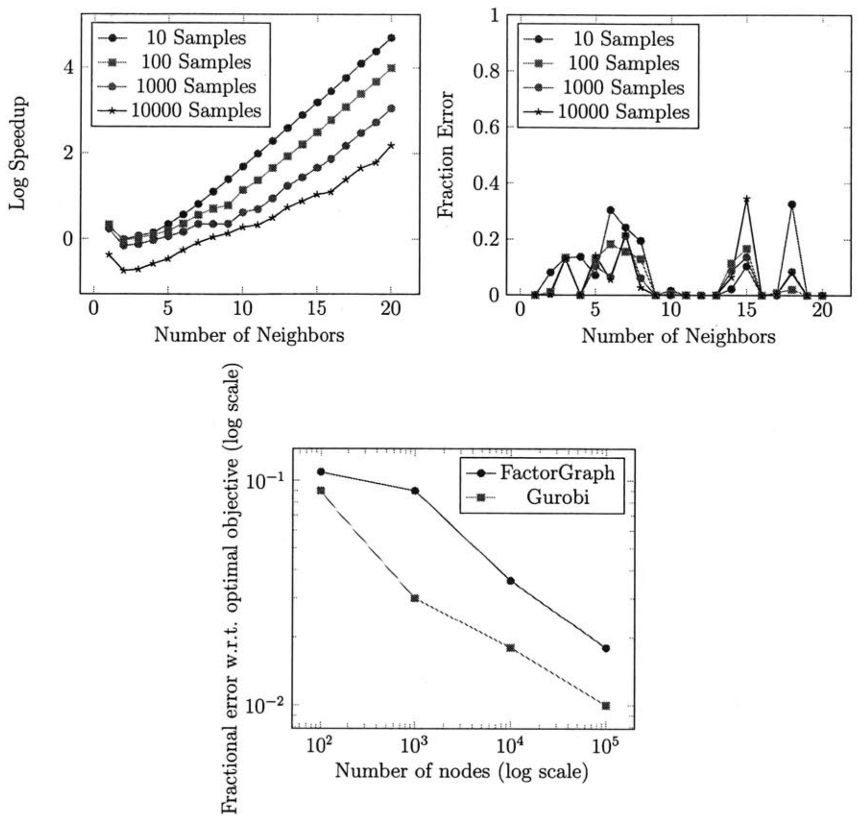

Figure 6-1: Top left: The log speedup from the sampling algorithm. Top Right: The fraction error from the sampling algorithm. Bottom: The fraction error relative to the optimal objective in the limit (0.55) for MWIS

$e^{w_i x_i} \prod_{m \in M} m$ where $m_{v_i \to f} \in \mathbb{Z}^n$ and $n \in \mathbb{Z}_n$. The input weight w is the weight w_i associated with variable x_i in the optimization objective. The vector $\mathbf{W}$ is $e^{w_i x_i}$ evaluated over all values of x_i, component-wise, where x_i is just the domain over which messages are defined. In the case of binary programming, for some particular w:

$$\mathbf{W} = \exp(w * \textsc{Domain}(M)) \tag{6.8}$$

$$= \exp(w * [0, 1]) \tag{6.9}$$

$$= [1, \exp(w)] \tag{6.10}$$

The function UPDATEEDGEVAR divides out a particular message, computing, for

Algorithm 16 SAMPLEDBPINTERFACE

```
procedure UPDATENODEVAR($M, w$)
    $\mathbf{W} \leftarrow \exp(w * \text{DOMAIN}(M))$
    $n \leftarrow \mathbf{W} \prod_{m \in M} m$
    return $n$
procedure UPDATEEDGEVAR($n, m$)
    $e \leftarrow \text{NORMALIZE}\left(\frac{n}{m}\right)$
    return $e$
procedure UPDATENODEFAC($M, s$)
    $n \leftarrow \text{GENERATESAMPLES}(M, s)$
    return $n$
procedure UPDATEEDGEFAC($n, m, m_i, f$)
    $e \leftarrow []$
    $S_{m_i} \leftarrow n \setminus \{n[m_i]\}$
    for all $i \in m$ do
        $e[i] \leftarrow \text{BPFACTORUPDATEFROMSAMPLES}(S_{m_i}, f_{m_i})$
    return NORMALIZE($e$)
```

variable node $v \in V$, and factor node $f \in \mathcal{N}(v)$:

$$m_{v \to f} = \frac{e^{w_v x_v} \prod_{g \in \mathcal{N}(v)} m_{g \to v}}{m_{f \to v}} \tag{6.11}$$

$$= e^{w_v x_v} \prod_{g \in \mathcal{N}(v) \setminus \{f\}} m_{g \to v} \tag{6.12}$$

For computing new factor node messages, we use the BP-SAMPLE algorithm. First, for a particular factor node $f \in F$, the function UPDATENODEFAC takes an array of incoming messages M, and number of samples to generate, s for the sampling algorithm. Samples are generated from the joint probability distribution given by the product of the incoming messages in M by the GENERATESAMPLES function. The set of all samples is used as the node state.

For computing a new outgoing message to variable node $m_{f \to v}, v \in \mathcal{N}(f)$, the function UPDATEEDGEFAC takes the previously generated set of samples as the node state, the incoming message m corresponding to $m_{v \to f}$, the index, m_i of m with respect to M, and the factor function, f. The set of samples S_{m_i} is the set of samples given in the node state, n, without the samples associated with the current edge.

6.1.3 Benchmarks

We evaluate the performance of our algorithm (and Factor Graph Compute framework) over several binary optimization problems as benchmarks: both synthetically generated problem instances (random instances of maximum weight independent set) as well "real world" challenge benchmarks of varying difficulty, from the well known optimization benchmark library [18]. We compare our algorithm's performance with Gurobi [17], a state-of-art commercial solver. The standard optimization problem format is **MPS** (Mathematical Programming System), which we then convert into respective factor graphs in a straightforward manner. All optimization experiments are run on a 32-core machine with 110GB of main memory, and using our asynchronous Redis implementation of PubSub.

In our Redis implementation, we utilize 30 partitions - 15 variable node partitions, and 15 factor node partitions, with 1 subscriber process per partition.

In addition, for each problem, we set particular decimation threshold D. We decimate the variable solutions every time all nodes have performed D iterations. The decimation function is implemented as a separate subscriber, which is subscribed to all variable node channels. The Decimation subscriber keeps track of the number of nodes which have completed D iterations. Each time a variable node partition, P_i^v publishes a message, the Decimation subscriber increments the iteration count of P_i^v. All iteration counts are stored as keys in Redis.

Once all nodes have completed D iterations, the variable node performs the decimation procedure, after which message passing continues

All subscriber callback functions and update functions are implemented in Python. Each problem has different decimation parameters which are specified with results of each problem.

Random Instances of Maximum Weight Independent Set.

Consider a graph $G = (V, E)$ and set of positive weights $W = \{w_i\}$ assigned to each node $i \in V$. The maximum weight independent set problem [24] can be expressed as the following binary programming problem:

$$\max \sum_{i \in V} w_i x_i \quad \text{over} \quad x_i \in \{0,1\}\ \forall i, \tag{6.13}$$

$$\text{subject to } x_i + x_j \leq 1, \forall (i,j) \in E. \tag{6.14}$$

In order to benchmark the accuracy of our framework, we utilize the following fact to generate benchmarks for MWIS over synthetic graphs: Let $G(n,p)$ be an Erdos-Renyi random graph of size n nodes, edge probability $p = \frac{2e}{n}$, and weights w_i sampled from an exponential distribution with mean $\lambda = 1$. Let $x^* = (x_i^*)$ be an optimal solution to the MWIS problem. Then, it is shown in [14] that

$$\lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} w_i x_i^* \approx 0.55 \tag{6.15}$$

We evaluate our algorithm's performance for n = 100, 1000, 10000 and 100000. In Figure 6-1 (log-log scale) we plot the fractional error with respect to the optimal limit per (6.15) as problem size n changes for both our algorithm as well as for Gurobi. As can be seen, both our solver and the Gurobi solver have small and comparable error with respect to asymptotic limit. This seems to suggest that for "simpler" instances of binary optimization, our solver does indeed find (near) optimal solutions.

Challenging Benchmark Optimization Problems

We test our framework on optimization problems provided by [18], a standard repository for integer programming problems and for benchmarking associated solvers. Optimization problems from [18] are stored in the .mps file format, which we convert from to our factor graph representation. We pick the following problems:

- `p6b.mps` - Maximum Independent Set
- `f2000.mps` - Pseudoboolean Optimization with unknown integer feasibility and objective

The problem `p6b.mps` is a Maximum Independent Set problem, formulated as a binary program, with 462 variables and 5,852 constraints. This problem was formulated in [18] (see p6b) and describes the maximum independent set problem over a component of the 2048-node graph. This problem is known to be feasible with optimal objective (maximum independent set size) of *63*. This is classified as a *hard* problem in [18] which means that it could not be solved within one hour using a commercial MIP solver. For this problem, our solver achieve a feasible solution with an objective value of 52, which is not too far from optimal objective. We note that the Gurobi solver finds several solutions, including the optimal value of 63, but most of its subsequent time is spent in trying to figure out whether this is an optimal solution or not.

The problem `f2000.mps` is a pseudo-boolean satisfaction (SAT) problem, formulated as a binary program, with 4,000 variables and 10,500 constraints. This problem is given in [18] (see f2000) and is currently an open problem - the integer objective and feasibility are unknown. The optimal objective for the linear programming relaxation of this problem is given as *1331*. This problem is classified by [18] as an *open* problem, meaning no commercial MIP solver is currently able to give an integer objective. The goal here is to find a (close to) feasible solution with maximal objective value. Our solver achieves an objective value of 1938, with $\approx 6\%$ violated constraints. In contrast, for this problem, Gurobi struggles to converge to an integer objective – it finds the relaxed linear objective, and like the previous problem, spends subsequent iterations verifying whether it is the best (relaxed) solution.

For both problems, we employ a decimation threshold of $D = 200$ iterations and decimate 10 nodes every time.

6.2 PageRank

As explained in Section 2.2.2, PageRank of a graph can be computed using the Factor Graph Compute framework in a straightforward manner. Given a graph with N nodes, we initialize all message as $1/N$.

To evaluate performance of our PageRank algorithm, we utilize the Orkut network

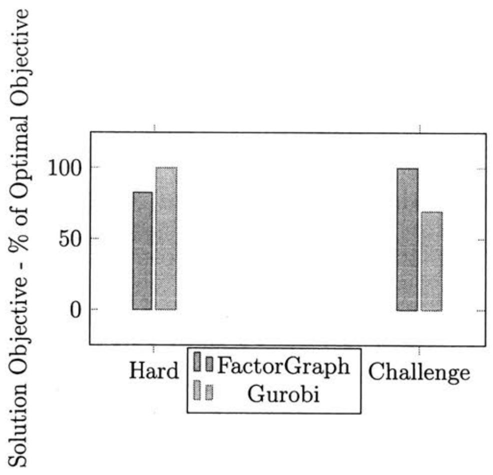

Figure 6-2: Gurobi and Factor Graph objectives for hard and challenge optimization problems

dataset from [21]. Orkut is an online social network - nodes represent users and edges represent "friendship" relationships between users. The network dataset consists of 3 million nodes and 117 million undirected edges. The factor graph of the Orkut network consists of 120 million nodes and 468 million (directed) edges. We compare our performance with respect to Ligra [25] which seems to one of the best performant in the recent literature on this question set.

For our Factor Graph Compute implementation of PageRank, 2 iterations is equivalent to 1 iteration of PageRank over the original graph. The first iteration consists of variable nodes publishing their messages to factor nodes, and the second iteration involves factor nodes publishing their messages back to variable nodes, upon which the experiment terminates. We use our implementation of UPDATEVAR and UPDATEFAC function in *Cython* and we utilize Ligra based PubSub to make it compatible with Ligra implementation.

Benchmarks are run with 32 threads and compiled with g++ using CilkPlus [4] to enable parallelism. We conduct benchmarks using both our implementation in Cython on top of Ligra and a PageRank implementation written in native C for Ligra.

6.2.1 Implementation

Algorithm 17 implements the PageRank UPDATEVAR and UPDATEFAC functions over a factor graph $\mathcal{G} = (V, F, E)$. This algorithm implements a slightly different variant from that described in section 2.2.2, which we describe now. One can see this implementation and that given in 2.2.2 are ultimately equivalent. In this implementation, our Factor Graph for PageRank $\mathcal{G} = (V, F, E)$, of an undirected graph $G = (V_G, E_G)$ is as follows:

$$V = \{v_i,\ i \in V_G\}, \tag{6.16}$$

$$F = \{f_{ij},\ (i, j) \in E_G\}, \tag{6.17}$$

$$E = \{(v_j, f_{ij}),\ v_j, v_i \in V, f_{ij} \in F, i \in \mathcal{N}_G(j)\} \tag{6.18}$$

Essentially, all nodes in the original graph G become variable nodes of the factor graph $\mathcal{G}$, and all edges in G become factor nodes in $\mathcal{G}$. The notation $\mathcal{N}_G(v_j)$ just means the neighborhood of node $i \in V_G$. We initialize messages as follows:

$$m_{v \to f \in \mathcal{N}(v)} = n^{-1} \tag{6.19}$$

$$m_{f \to v \in \mathcal{N}(f)} = \text{null} \tag{6.20}$$

We specify state for each node as follows:

$$x_v = [m_{f_1 \to v}, \ldots, m_{f_k \to v}]^T \in \mathbb{Q}^k \tag{6.21}$$

$$y_f = [m_{v_1 \to f}, \ldots, m_{v_l \to f}]^T \in \mathbb{Q}^l \tag{6.22}$$

$$k = |\mathcal{N}^-(v)|, l = |\mathcal{N}^-(f)| \tag{6.23}$$

We specify computation dynamics as follows:

$$\begin{aligned}
m_{v\to f} &= \textsc{UpdateVar}_v(x_v; m_{g\to v}, g \in \mathcal{N}^-(v); f) \\
&= \frac{1-d}{n} + d\left(m_{g\to v} + \sum_{k\in\mathcal{N}^-(v)\setminus\{g\}} x_v[k]\right) \\
m_{f\to v} &= \textsc{UpdateFac}_f(y_f; m_{u\to f}, u \in \mathcal{N}^-(f); v) \\
&= \begin{cases} \frac{m_{u\to f}}{deg(u)} & \text{if } u \neq v \\ \text{null} & \text{if } u = v \end{cases}
\end{aligned} \tag{6.24}$$

6.2.2 Interface

Algorithm 17 PAGERANKINTERFACE

```
procedure UPDATENODEVAR($M, d, N$)
    $n \leftarrow \frac{1-d}{N} + d\sum_{m\in M} m$
    return $n$
procedure UPDATEEDGEVAR($n, m$)
    $e \leftarrow n$
    return $e$
procedure UPDATENODEFAC($M, f$)
    $n \leftarrow \sum_{m\in M} m * f[m]^{-1}$
    return $n$
procedure UPDATEEDGEFAC($n, m, m_i, f$)
    $e \leftarrow n - m * f[m_i]^{-1}$
    return $e$
```

In Algorithm 17, the function UPDATENODEVAR computes the PageRank of a particular node $v \in V$, as in 6.24, given an array of incoming messages M, the damping factor d, and total number of nodes N. Node v sends its newly computed PageRank value (node state n) to all neighbors, so the function UPDATEEDGEVAR returns n which is exactly the PageRank of v. The functions UPDATENODEFAC and UPDATEEDGEFAC correspond to dividing incoming PageRank values by the degree of the appropriate node, as in equation 6.24. For a particular factor node $f \in F$, the factor function of f contains the degrees of the neighbors of f. The function UPDATEN-

ODEFAC for f computes the node state

$$n = \sum_{m \in M} mf[m]^{-1} \tag{6.25}$$

$$= \sum_{g \in \mathcal{N}(f)} \frac{1}{deg(g)} PR(g) \tag{6.26}$$

The function UPDATEEDGEFAC for a particular edge (f, v) computes

$$m_{f \to v} = n - mf[m_i]^{-1} \tag{6.27}$$

$$= \sum_{g \in \mathcal{N}(f)} \frac{1}{deg(g)} PR(g) - \frac{1}{deg(v)} PR(v) \tag{6.28}$$

$$= \frac{1}{deg(g)} PR(g), g \neq v \tag{6.29}$$

Since all factor nodes have 2 neighbors, giving us the desired value for the new message $m_{f \to v}$ to send along edge (f, v). The damping factor d and number of nodes in the graph, N are given at the start. Factor functions are stored as arrays and represent the degrees of the appropriate nodes in the original graph. All messages are initialized to $\frac{1}{N}$.

6.2.3 Benchmarks

In Table 6.1, we show the total time to run a single of iteration of PageRank in our framework, as well as using the Ligra framework [25]. Our runtimes for PageRank over the Orkut factor graph, implemented using Ligra PubSub, is similar to that of running PageRank over the Orkut factor graph, using Ligra's PageRank implementation. For two iterations of Ligra PubSub, which corresponds to a single iteration of PageRank, we achieve a running time of 4.62 seconds. Two iterations of Ligra PageRank over the factor graph runs in 3.5 seconds. This demonstrates that while Ligra itself is optimized for graph algorithms, performing similar computations by using it as the PubSub for Factor Graph Compute does not slow down the system by much, while providing a large gain in expressibility and flexibility to solve a variety of different problems.

Implementation	Nodes	Directed Edges	Iterations	Time (seconds)	Parallelism
Factor Graph Compute	120,257,524	468,739,654	2	4.62	32 threads
Ligra	120,257,524	468,739,654	2	3.5	32 threads

Table 6.1: Ligra and Factor Graph Compute (w/Ligra PubSub) have comparable PageRank performance

6.3 Singular Value Decomposition

PageRank is a special instance of Singular Value Decomposition (SVD). Implementing SVD involves matrix-vector multiplication, which, as we showed earlier in Section 2.2.4, fits the Factor Graph Compute framework in a straightforward manner.

6.3.1 Implementation

Using matrix-vector multiplication, it is straightforward to implement algorithms such as Singular Value Decomposition (SVD).

We consider the special case of computing the SVD of a square, positive semi-definite matrix **A**. Computing the singular values of **A** can be done using a power iteration. This involves a series of matrix multiplies of **A**, for some number of iterations, n, and an initial random vector $\vec{x}$. Each iteration gives a new vector $\vec{x}_n = \mathbf{A}\vec{x}_{n-1}/||\mathbf{A}\vec{x}_{n-1}||$.

This can also be thought of as continually multiplying the initial vector $\vec{x}$ by **A** a number of times, with a normalization step after each iteration. Computing the result of $\mathbf{A}^n\vec{x}$ corresponds to running n iterations of the matrix-vector mutliplication algorithm (with normalization), given in algorithm 18.

This can be achieved by using matrix-vector as described in Section 2.2.4 along with another sub-routine that computes the norm of a vector (this is just addition and can be viewed as a factor node connected to all variable nodes in our factor graph).

6.3.2 Interface

The interface for SVD is very similar to PageRank. Rather than initializing all nodes to $\frac{1}{|V|}$, we initialize our nodes to a random value $x_i \in \mathbb{R}$, which corresponds to picking a random vector for the power iteration.

Algorithm 18 SVDINTERFACE

```
procedure UPDATENODEVAR(M)
    $n \leftarrow \sum_{m \in M} m$
    return $n$
procedure UPDATEEDGEVAR(n, m)
    $e \leftarrow n$
    return $e$
procedure UPDATENODEFAC(M, f)
    $n \leftarrow 0$
    for all $m \in M$ do
        $n \leftarrow n + m * f[m]$
    if $||M|| == 1$ then
        $n \leftarrow 2 * n$
    return $n$
procedure UPDATEEDGEFAC($n, m, m_i, f$)
    $e \leftarrow n - m * f[m_i]$
    return $e$
```

Since SVD is implemented as a power iteration of matrix-vector multiplies, it implements the UPDATEVAR and UPDATEFAC functions for Matrix-Vector Multiplication given in Section 2.2.4.

For a particular variable node i, the function UPDATENODEVARABLE takes a collection of its incoming messages, M and returns its sums as the node state. The function UPDATEEDGEVARIABLE returns the node state previously computed as the outgoing messages along all edges of i.

For a particular factor node f, with variable node neighbors v_i, v_j, the outgoing message along an edge (f, v_i), after applying UPDATEFAC, should be $a_{ij}m_j$ where m_j is the incoming message from node v_j.

The factor function for f is an array `factor_function`, containing entries a_{ij}, a_{ji} from our input matrix. The function UPDATENODEFACTOR computes as the node state, n_f, $n_f = a_{ij}m_j + a_{ji}m_i$, where m_i, m_j.

The function UPDATEEDGEFACTOR for an edge of f, (f, v_i) computes $n_f - a_{ji}m_i = a_{ij}m_j$ as the message to send along edge (f, v_i). The case where $j = i$ corresponds to a diagonal element a_{ii} of our input matrix. In this case, `factor_function` only contains a single element, a_{ii}, and the function UPDATEEDGEFACTOR just returns the computed node state $n_f = a_{ii}m_i$ as the outgoing message along edge (f, v_i).

PageRank computation is a special instance of Singular Value Decomposition as it tries to compute leading eigenvector of a specific matrix. As discussed in Section 2.2.4, matrix-vector multiplication fits the Factor Graph Compute framework in a straightforward manner. We will utilize this basic operation to compute Singular Value Decomposition (SVD) of a matrix.

To that end, it is sufficient to consider a square, symmetric matrix $\mathbf{A}$ since computing singular vectors of matrix M boil down to computing eigenvectors of MM^T or M^TM. In particular, we shall employ power-iteration (ideally, it's variants such as Lanczos method [20] but for simplicity as well as fair comparison with results reported in literature [11] we shall consider power-iteration only). Here, starting with a random initial vector $\vec{x}_0$, the goal is to iteratively compute

$$\vec{x}_n = \frac{1}{\|\mathbf{A}\vec{x}_{n-1}\|}\mathbf{A}\vec{x}_{n-1}. \tag{6.30}$$

This can be achieved by using matrix-vector multiplication sub-routine as in Section 2.2.4 along with another sub-routine that can compute norm of a vector (which is simply addition, can be viewed as a 1-factor graph with all components of vector as variables connected with it). That is, such an iteration can be implemented in our framework in a straightforward manner.

6.3.3 Benchmarks

We ran our experiments for SVD using Ligra as our PubSub system. We generated a synthetic sparse square symmetric matrix with dimensions $1M \times 1M$ and $51M$ non-zeros (out of 10^{12} possible entries). The associated factor graph for this sparse matrix has 26,499,350 nodes and 203,994,406 (directed) edges. All computations are run with 32 threads and Ligra compiled with g++ using CilkPlus [4] to enable parallelism. One power-iteration of SVD corresponds to two iterations (one variable and one factor) in the Factor Graph Compute framework.

As reported in the Table 6.2, it takes 0.98 seconds of wall-clock time to compute one iteration of power-iteration for such a matrix.

We compare our SVD experiments on a sparse square matrix against the Spark SVD benchmark reported in [11]. The Spark benchmark correspond to a "tall and skinny" with $51M$ non-zeroes. Computing the singular value decomposition (along the skinny side) results in a much smaller square matrix. The Spark benchmark was run using 68 executors in the Spark environment with a wall-clock time of 0.2 seconds. Thus, the total time over all 68 executors is $0.2 * 68 = 13.6$ seconds. The reported experiment for Spark utilized 8GB of memory per executor and hence total of 544GB of memory.

In contrast, our SVD experiment was run on a single 110GB memory machine with 32 threads. Comparing our total compute time, which is 0.98 seconds on a single machine, with that of 13.6 seconds for Spark, suggests an improvement of $>$ 13x.

Table 6.2: SVD Experiments

Implementation	Dataset	Time (seconds)	Parallelism
Factor Graph Compute	SPARSE51NZ-FACTORGRAPH	0.9815	32 threads
Spark	TALLANDSKINNYSPARSE51NZ	0.2	68 executors

6.4 Scaling

We consider the scaling properties as we increase parallelism of our system. We run a single iteration of PageRank on our asynchronous Factor Graph Compute implementation which uses Redis as PubSub. This corresponds to every variable node publishing its initial message, all factor nodes computing outgoing messages, variable nodes receiving all messages from their factor neighbors and computing outgoing

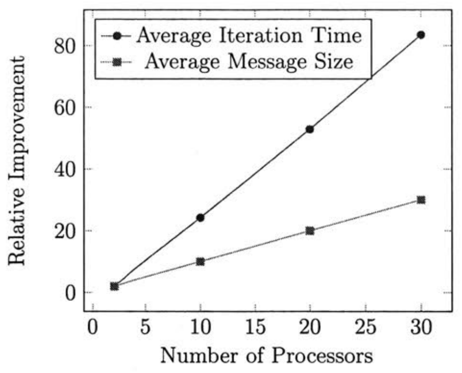

Figure 6-3: Performance metrics scale linearly with the number of processors

messages. Our implementation uses a *partitioned* factor graph with the same number of variable and factor partitions. We assign nodes to their respective partitions uniformly at random, which corresponds to the simplest or worst-case partitioning scheme. All benchmarks are run on the Amazon Product Co-Purchasing Network dataset [21] which consists of 403,394 nodes and 3,387,388 (directed) edges. The corresponding undirected factor graph consists of 2,846,802 nodes - 403,394 variable nodes and 2,443,408 factor nodes, and 4,886,816 edges. We consider the following metrics: average iteration time per partition, average message size (number of messages published per partition)

We run our benchmarks on a single machine with 32 processors and scale the number of processors from 2 to 30, with 1 partition for each processor. In Figure 6-3 we see that the average iteration time, and average message size improve as the number of processors (amount of parallelism) increases. The average iteration time drops from 10 seconds to about 0.24 seconds, which is about a 42x speedup. The average message size drops from 4,886,816 messages to 325,787.7333 which is about a 15x improvement.

Figure 6-3 confirms that the system performance scales linearly with the available resources which is the best one can expect. This also hints at the possibility that the same "program" can run on a single machine PubSub or can scale gracefully when run on distributed PubSub.

Chapter 7

Conclusions

We introduce a computational framework based on factor graphs implemented using PubSub infrastructure. This framework is expressive (Turing complete), allows for a unified "programming" interface across different environments, and scales seamlessly. We showed our framework provides excellent performance, and is comparable to the state of the art, for binary optimization, graph, and Linear Algebra problems. However, though our language based on Factor Graphs is Turing complete, the factor graph view of a given problem is not intuitive. It is not generally straightforward to convert problems into their factor graph representation. Making this framework "user friendly" requires building transformers to convert problems from the standard view to the factor graph view. This is feasible for a large class of problems including those discussed in Section 2.2.

Bibliography

[1] https://www.cs.cmu.edu/ pbbs/benchmarks/graphIO.html.

[2] https://github.com/andymccurdy/redis-py.

[3] Akka concurrency framework. https://akka.io/.

[4] CilkPlus. https://www.cilkplus.org/.

[5] ZeroMQ. http://zeromq.org/.

[6] Amazon Web Services. Amazon simple notification service (sns). https://aws.amazon.com/sns/.

[7] Amazon Web Services. What is pub/sub messaging? https://aws.amazon.com/pub-sub-messaging/.

[8] Apache. ActiveMQ. http://activemq.apache.org/.

[9] Mohsen Bayati, Devavrat Shah, and Mayank Sharma. Max-product for maximum weight matching: Convergence, correctness, and lp duality. *IEEE Transactions on Information Theory*, 54(3):1241–1251, 2008.

[10] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[11] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Aaron Staple, and Matei Zaharia. Matrix computations and optimization in apache spark. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 31–38, New York, NY, USA, 2016. ACM.

[12] G David Forney. Codes on graphs: Normal realizations. *IEEE Transactions on Information Theory*, 47(2):520–548, 2001.

[13] Brendan J Frey, Frank R Kschischang, Hans-Andrea Loeliger, and Niclas Wiberg. Factor graphs and algorithms. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, volume 35, pages 666–680. UNIVERSITY OF ILLINOIS, 1997.

[14] David Gamarnik, Tomasz Nowicki, and Grzegorz Swirszcz. Maximum weight independent sets and matchings in sparse random graphs. exact results using the local weak convergence method. *Random Structures and Algorithms*, 28(1):76–106, 2006.

[15] David Gamarnik, Devavrat Shah, and Yehua Wei. Belief propagation for min-cost network flow: Convergence and correctness. *Operations Research*, 60(2):410–428, 2012.

[16] Google Cloud Platform. Cloud pub/sub. https://cloud.google.com/pubsub/.

[17] Gurobi Optimization, Inc. Gurobi optimizer reference manual. http://www.gurobi.com, 2016.

[18] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[19] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[20] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45:255–282, 1950.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[23] Redis Labs. Redis. https://redis.io.

[24] Sujay Sanghavi, Devavrat Shah, and Alan S Willsky. Message passing for maximum weight independent set. *IEEE Transactions on Information Theory*, 55(11):4822–4834, 2009.

[25] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[26] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Data Compression Conference (DCC), 2015*, pages 403–412. IEEE, 2015.

[27] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. Parallel local graph clustering. *Proceedings of the VLDB Endowment*, 9(12):1041–1052, 2016.

[28] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[29] R Tanner. A recursive approach to low complexity codes. *IEEE Transactions on information theory*, 27(5):533–547, 1981.

[30] V.Ramesh and D. Shah. Computing With PubSub. submitted.

[31] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[32] Wikipedia contributors. Publish-subscribe pattern — Wikipedia, the free encyclopedia. https://en.wikipedia.org/.

[33] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Generalized belief propagation. In *Advances in neural information processing systems*, pages 689–695, 2001.

[34] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.