Distributed inference with declarative overlay networks

Stanislav Funiak
Ashima Atul
Kuang Chen
Joseph M. Hellerstein
Carlos Guestrin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-135.html

October 19, 2008

Acknowledgement

We would like to thank Tyson Condie for his help with P2.

Distributed inference with declarative overlay networks

Stanislav Funiak, Carlos Guestrin
Carnegie Mellon University
{sfuniak, guestrin}cs.cmu.edu

Ashima Atul, Kuang Chen, Joseph Hellerstein
University of California, Berkeley
{ashima, kuangc, hellerstein}cs.berkeley.edu

Abstract

There is a growing trend towards systems in multiple distributed locations that generate massive amounts of data. Distributed inference algorithms enable these systems to combine noisy observations without communicating them to a central location. However, the design and implementation of distributed algorithms is very challenging. We propose a combination of overlays and declarative programming to simplify the design of distributed inference algorithms. We demonstrate the effectiveness of our approach on a number of inference algorithms and evaluate the implementation on a suite of datasets in a real network deployment.

1 INTRODUCTION

There is a growing trend towards systems in multiple distributed locations that generate massive amounts of data. A key problem in these systems is probabilistic inference. To perform this task, the nodes in a physical network need to collaborate, in order to estimate desired quantities from noisy observations. For example, routers in a network may combine similarity measurements of IP's sending patterns to identify which IP addresses send spam. Distributed inference algorithms eliminate the centralized point of failure, distribute the computation across several nodes, and avoid the need to share sensitive data. (Kearns, Tan, & Wortman, 2008)

Distributed inference is a challenging task. First, it is difficult to design distributed algorithms: nodes need to coordinate to distribute the computation across the network, and can only access a portion of the model at a time. The algorithms need to be network-aware: in real settings, link qualities change over time, nodes can enter and leave the network, and interference can disconnect parts of the network. The second major challenge is that programming distributed systems is itself difficult. In conventional implementations, high-level algorithm description needs to be translated into a series of low-level communication protocols, coded in low-level languages, such as C++. Such process is very error-prone and difficult to debug. Addressing these challenges is key to designing robust and efficient distributed inference algorithms.

In this paper, we propose a novel formulation that uses a combination of **overlays** and **declarative programming** (Loo & Hellerstein, 2007) to simplify the design of inference algorithms. An overlay is a distributed algorithm that creates and maintains a global data structure on a physical network. Examples of overlays include distributed hash tables, DeBruijn graphs, and data aggregation structures. Overlays are often described with a set of constraints on an underlying graph structure; these constraints can be effectively expressed in declarative programming languages, such as Overlog (Loo & Hellerstein, 2007). The combination of overlays and declarative programming allows the algorithm designer operate on a higher level of abstraction that rises above low-level message passing protocols.

There have been a few distributed inference algorithms that utilize overlays. For example, Schmidt and Aberer (2006) proposed to use distributed hash tables to perform content-based addressing in loopy belief propagation. Paskin, Guestrin, and Mcfadden (2005) described a robust architecture for distributed junction tree inference in sensor networks. However, each of these approaches were a point solution and, to our knowledge, have never been reproduced. Furthermore, it is difficult to adapt existing algorithms to a new communication pattern. We demonstrate the effectiveness of our approach on a number of algorithms. We show that it is simple to implement existing algorithms, such as loopy belief propagation and junction tree inference. Furthermore, we show how the conver-

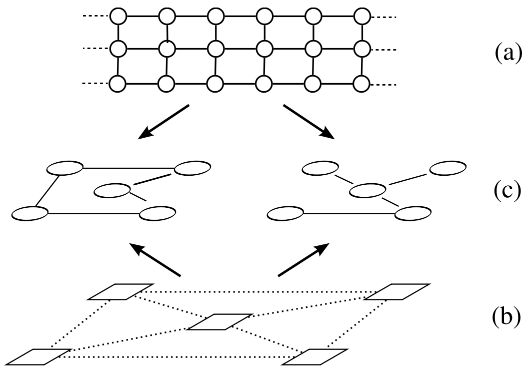

Figure 1: Overlays in distributed inference: (a) a graphical model, (b) physical network, (c) a set of overlays that aid in the computation.

gence of existing inference algorithms can be improved using a data aggregation overlay. We complement existing randomized schemes (Elidan, Mcgraw, & Koller, 2006) with an analysis of their convergence.

We evaluate our implementations on a suite of applications that include standard benchmark examples (Elidan et al., 2006) as well as applications more specific to distributed systems. We evaluate the implementation on Emulab (White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, & Joglekar, 2002), a large testbed that simulates realistic network conditions, such as communication delays. We demonstrate that our implementations achieve similar performance as the corresponding low-level distributed algorithms and are comparable to idealized centralized algorithms that does not perform global coordination.

2 THE DISTRIBUTED INFERENCE PROBLEM

2.1 Distributed inference

One method to approximate the marginals $\{p(y_i; \mathbf{x})\}$ is loopy belief propagation (Murphy, Weiss, & Jordan, 1999) [1]. Loopy belief propagation is an iterative method that can be viewed as passing messages on the underlying Markov network. A message from variable s to variable t is computed as

$$\mu_{s,t}(y_t) \leftarrow \sum_{y_s} \psi_s \times \psi_{s,t} \times \prod_{r \in N_G(s) \setminus t} \mu_{r,s}(y_s), \qquad (1)$$

where the product is taken over all neighbors r of variable s in the Markov network, other than t. At convergence, the node marginals can be approximated as

$$p(y_s) \approx \frac{1}{Z_i}\, \psi_s(y_s; x_s) \times \prod_{r \in N_G(s)} \mu_{r,s}(y_s).$$

While conceptually, we could collect all the features $\mathbf{x}$ to a centralized location, a distributed version of the inference algorithm (1) has several advantages: it eliminates the centralized point of failure, and distribute the computation across several nodes.

In general, each network node is assigned a portion of the probabilistic model, and the nodes collaborate to compute marginal distribution over one or more variables. We assume a networking model where each node can communicate to a subset of other nodes, but the communication costs between nodes can vary.

2.2 Challenges

Distributed inference is a challenging task. Distributed algorithms are difficult to design. In the context of distributed inference, this difficulty arises due to the following three facts: The algorithms need to use a **decentralized representation** of the probabilistic model and perform **global coordination** to distribute the computation and the parts of the model across several nodes. The algorithms need to be **network-aware**, responding to changes in the link quality and perform robustly in the presence of communication delays.

A core challenge in distributed algorithms is that programming distributed systems is itself difficult. Conventionally, the high-level, compact description of these algorithms needs to be translated manually into a set of low-level communication protocols, executed in a low-level programming language. Such translation is often time-consuming and error-prone, and results in programs that are very difficult to debug. In this paper, we examine how declarative languages simplify the implementation of distributed inference algorithms with faster convergence and lower bandwidth consumption.

3 DECLARATIVE SPECIFICATION OF OVERLAYS

Several distributed datastructures have been proposed in recent literature, including distributed hash tables, approximate hypercubes, and DeBruijn graphs. The datastructures are robust to changes in the network topology, responding to nodes entering and leaving the network. Our approach is to describe the distributed datastructure and the inference algorithm logic with a declarative programming language. In this paper,

[1]We discuss other inference methods in the later parts of the paper.

BP1: $(s,t) \in \mathsf{update}(n) \;\vdash$
$\quad s \in \mathsf{variables}(n),\ (s,t) \in \mathsf{edges}(n).$

BP2: $(s,t,\prod \mu_{r,s}) \in \mathsf{incoming}(n) \;\vdash$
$\quad (s,t) \in \mathsf{update}(n),\ \mu_{r,s} \in \mathsf{msg}(n),\ r \neq s.$

BP3: $\mu_{s,t} \in \mathsf{msg}(m) \;\vdash$
$\quad (s,t,\phi) \in \mathsf{incoming}(n),\ \psi_s, \psi_{st} \in \mathsf{pot}(n),$
$\quad (t,m) \in \mathsf{locations}(n),$
$\quad \mu_{s,t} = \sum_{y_s} \phi \times \psi_s \times \psi_{st}.$

Figure 2: Loopy belief propagation's message computation (1) in a declarative-style program. The definition of periodic updates was omitted for brevity.

we focus on **P2** `http://p2.cs.berkeley.edu`. P2 takes specifications in a declarative query language, and uses database query optimization techniques to compile them into dataflow programs that resemble a mixture of traditional relational query plans and network routers.

To illustrate the declarative style of programming distributed inference algorithms, we begin with a simple example that implements loopy belief propagation (1) in the case where the variables are partitioned across the nodes of the network, and the nodes have a global knowledge of this assignment. This distributed algorithm is very simple: nodes communicate exactly along the edges of Markov random network, and do not perform any global coordination. We will consider more interesting examples in the following sections.

The corresponding program is shown in Figure 2. The program is written in Overlog, which is an extension of the traditional recursive query language Datalog, enhanced with aggregation functions and periodic events. Overlog programs consist of a set of declarative **rules**. The right-hand-side of the rule represents a conjunctive predicate over relations in a database, and the left-hand side represents the deduction from that predicate. For example, the rule BP1 can be read as "**if** there is a variable s in the relation of variables stored at node n, and **if** there is a tuple (s,t) in the relation of the MRF edges stored at node n, **then** there is an (s,t) tuple in the relation of updates to be performed at that node." Overlog provides constructs for periodically updating the derived facts. These constructs resolve the apparent circular definition of the msg relation and simplify the definition of iterative algorithms like loopy belief propagation. Comparing the program in Figure 2 with the update equation (1), we see that there is almost a one-to-one correspondence between the algorithm description and its distributed implementation.

4 DISTRIBUTED NORMALIZATION

A concern with synchronous iterative algorithms, such as loopy belief propagation, is that they update messages indiscriminately, rather than focusing on the important messages that need to be updated to obtain fast convergence. In the distributed setting, where bandwidth and power consumption are often the limiting factors, updating the messages indiscriminately can be especially costly. The paper by Elidan et al. (2006) proposed an effective centralized solution that updates messages in the order, given by a lower-bound on the difference between the current message and the fixed point. In this section, we describe a simple randomized approximation of this algorithm. Our algorithm is similar in spirit to (Schiff, Antonelli, Dimakis, Chu, & Wainwright, 2007), but uses an overlay to compute an implicit normalization constant in the algorithm, and we derive approximation bound that relates our solution to the one in (Elidan et al., 2006).

The message passing update (1) can be viewed as an operator $f_{s,t}$ that takes a set of messages $\boldsymbol{\mu} = \{\mu_{q,r} : (q,r) \in E\}$ and computes a new set of messages $\boldsymbol{\mu}' = \left\{\mu'_{q,r}\right\}$ that differ exactly on the message $\mu_{s,t}$. When $f_{s,t}$ is a contraction operator, that is,

$$\left\| f_{s,t}(\boldsymbol{\mu}) - \boldsymbol{\mu}^* \right\| \leq \alpha \left\| \boldsymbol{\mu} - \boldsymbol{\mu}^* \right\|$$

for some message norm $\|\cdot\|$ and a constant $\alpha < 1$, it can be shown that the **residual** $r_{s,t} \triangleq \|\boldsymbol{\mu} - f_{s,t}(\boldsymbol{\mu})\|$ is a lower bound on the distance between $\boldsymbol{\mu}$ and the fixed point $\boldsymbol{\mu}^*$. Elidan et al. (2006) use this intuition to formulate a greedy heuristic that updates messages $\mu_{s,t}$ in the order, given by the residual $r_{s,t}$. To compute the order of updates efficiently, the algorithm maintains a priority queue over the residuals and updates the queue in constant time (for Markov networks with bounded degree).

The problem with the above approach is that it requires a global priority queue and blocks the computation of the nodes while the node with the leading residual performs the update. Instead, we wish to obtain a local algorithm that attains a similar performance, without maintaining a globally controlled priority queue.

A simple strategy, proposed by (Schiff et al., 2007) is to delay messages with smaller residuals. This strategy can be implemented by performing a sequence of independent Bernoulli trials. At each iteration, the message $\mu_{s,t}$' is transmitted with probability given by

$$p_{s,t} = \left\| \mu'_{s,t} - \mu_{s,t} \right\|^{\rho} \tag{2}$$

for a suitably chosen constant $\rho \geq 0$ (here, $\mu_{s,t}$ is the last transmitted message). Effectively, the messages with larger residuals $r_{s,t} = \|\mu'_{s,t} - \mu_{s,t}\|$ will be transmitted more often.[2]

The algorithm, as described so far, has a significant drawback: as the messages get closer to the fixed point, the transmission probabilities $\{p_{s,t}\}$ decrease throughout the network. Therefore, the algorithm will eventually stop making progress and will *never* converge. In order to ensure convergence, we need to multiply the update probability in (2) with a suitably chosen normalization term λ. This term is not known a priori; rather, it is periodically estimated from the current set of residuals $\{r_{s,t}\}$. The nodes can employ a number of data aggregation techniques to estimate the sum $\sum r^{\rho}_{s,t}$. For example, the aggregate can be computed by forming spanning tree over the network nodes and computing the sum recursively using a dynamic programming algorithm. As we demonstrate in Section 7, the resulting algorithm offers substantial improvement over the naive implementation in Figure 2.

We conclude this section by relating the proposed randomized algorithm to the solution (Elidan et al., 2006). Let $K_{s,t}$ denote the number of iterations before the message is sent. The key observation is that, as the parameter ρ increases, the ordered statistics $\min_{s,t} K_{s,t}$, converges to the maximum residual.

5 VARIABLE LOCALIZATION

As mentioned in the introduction, one of the challenges present in distributed systems is the lack of global knowledge. In particular, when sending a message in loopy belief propagation, a node may not know a priori where the target variable t is located (in the program in Figure 2, this knowledge was captured by the location relation). The key observation here is that the variable location can be looked up using a distributed hash table, such as Chord (Stoica, Morris, Karger, Kaashoek, & Balakrishnan, 2001). This observation was made by (Schmidt & Aberer, 2006); here, we briefly discuss this approach, and refer the reader to related work for a more detailed description of the overlay.

In a distributed hash table (DHT), each node is associated with a key, and nodes form a graph based on their keys. For example, in Chord, the nodes form a ring; the keys of the nodes along the ring form an ordered set of ranges. The object is then placed on a node, whose key immediately precedes the key of the object. In our case, each object consists of the variable identifier and the address of the node that carries the variable. Upon start, each node registers its local variables in the DHT. Before a message $\mu_{s,t}$ is sent, the node looks up the location of variable t in the DHT. The overlay and the variable lookup is very simple to implement in a declarative framework: the implementation of Chord in Overlog is only 250 lines long, compared to thousands lines of C++ for the original version.

[2]Here, it is assumed that the residual $r_{s,t} \leq 1$.

6 DISTRIBUTED TRIANGULATION

In the previous two sections, we saw how overlays can be constructed bottom-up and used as subroutines to simplify the design of distributed loopy belief propagation algorithms. The overlays depended entirely on the network connectivity; for example, the DHT was constructed independently of the variables stored in it. In this section, we examine the **distributed triangulation** problem (Paskin et al., 2005), where the structure of the overlay is determined by both the network connectivity and the probabilistic model. We briefly review the distributed architecture (Paskin et al., 2005), and show that the architecture can be naturally expressed in a declarative framework.

The architecture (Paskin et al., 2005) computes the triangulation as follows. Each node n begins with a set of **local** variables L_n; this is the union of arguments in node n's factors. The network nodes form a **network junction tree**, that is, a spanning tree over the network communication graph such that each node is associated with a clique $C_n \supseteq L_n$, and the cliques $\{C_n\}$ are **triangulated**. Specifically, for each pair of nodes m, n:

$$X \in C_m, X \in C_n \implies X \in C_k \qquad (3)$$

for all nodes k on the (unique) path between m and n.

The distributed algorithm (Paskin et al., 2005) consists of four layers:

1. **Spanning tree formation**: The nodes form a spanning tree. The spanning tree responds to changes in network connectivity and node failures.

2. **Junction tree formation**: The nodes compute a set of minimal cliques $\{C_n\}$ that satisfy the running intersection property (3).

3. **Optimization**: The architecture performs local search in the space of spanning trees in order to minimize the communication and computational complexity.

4. **Inference**: The inference messages are computed using standard equations. For example, for the sum–product algorithm, we compute the message

C: $i \in \mathsf{clique}(n) \ \vdash$
$(m,i),(k,i) \in \mathsf{reachable}(n),\ m \neq k \ \vee$
$i \in \mathsf{local}(n).$

R1: $(n,i) \in \mathsf{reachable}(m) \ \vdash$
$i \in \mathsf{local}(n),\ m \in \mathsf{nbr}(n).$

R2: $(n,i) \in \mathsf{reachable}(m) \ \vdash$
$(k,i) \in \mathsf{reachable}(n),\ m \in \mathsf{nbr}(n),\ m \neq k.$

Figure 3: Implementing distributed triangulation in Overlog. The statement $(k,i) \in \mathsf{reachable}(n)$ represents the fact that the variable i is in the subtree rooted at node n, towards k.

$\mu_{m,n}$ as

$$\mu_{m,n}(\mathbf{x}_{S_{m,n}}) \triangleq \sum_{\mathbf{x}_{C_n - S_{m,n}}} \psi_m(\mathbf{x}_{C_m}) \prod_{k \in N_T(m) \setminus n} \mu_{k,m}(\mathbf{x}_{S_{k,m}}),$$

where $\psi_m(\mathbf{x}_{C_m})$ is the product of potentials assigned to node m, $S_{m,n} \triangleq C_m \cap C_n$ is the **separator** between two nodes in the junction tree, and $N_T(m)$ are node m's neighbors in the tree.

These layers are executed in parallel: for example, if an edge in the spanning tree is added or removed, the junction tree layer updates the clique at each node, and the inference layer recomputes the inference messages.

In the original TinyOS implementation of Paskin et al. (2005), the three layers had to explicitly handle changes in the underlying architecture, listening to local events indicating the change, such as edge addition and deletion. Such complex interleaving of the inference and networking layers can be naturally expressed in a declarative language. For example, the clique at node n of the network junction tree is computed as

$$C_n = L_n \bigcup \left\{ i : i \in R_{n,m} \cap R_{n,k}, m \neq k \right\},$$

where $R_{n,m}$ is the set of all variables in the subtree rooted at node n, towards node m. This equation directly maps to rule C in the declarative implementation in Figure 3. The reachable variables relation $R_{m,n}$ is naturally expressed recursively with R1 (the base case) and R2 (the recursive case). The P2 runtime responds to any changes to the preconditions, and recomputes the reachable sets and cliques as necessary.

7 EXPERIMENTAL RESULTS

In this section, we present the experimental results of running our algorithm on several datasets. In addition to the sensor calibration, presented in Section 7.2, we also tested our algorithms on a suite of challenging problems used in (Elidan et al., 2006) that are not specific to the distributed inference domain. All experiments were performed on the Emulab testbed (White et al., 2002), which simulate realistic network conditions, including packet delays and network partitions.

7.1 Belief propagation: Synthetic data

In the first set of experiments, we evaluate the convergence of different methods, presented in this paper, as a function of the number of updates and the bandwidth of the algorithm used.

We generated a set of random Ising grid models, where each edge has the potential $f(x_i, x_j) = \exp(\beta_{i,j})$ when $x_i = x_j$ and $f(x_i, x_j) = \exp(-\beta_{i,j})$ when $x_i \neq x_j$. To make the inference task more challenging, we set some of the potentials to be attractive while other repulsive, and sampled the $\beta_{i,j}$ uniformly in the range $[-C; C]$ for two choices of parameter C. Similar models have been employed for testing loopy belief propagation algorithms in the past.

Figure 4 shows the residual as a function of the number of updates performed by the algorithm. For comparison, we also include the convergence of two centralized algorithms: an algorithm that updates the messages in a round-robin order (labeled as `asynchronous`) and the residual belief propagation algorithm (Elidan et al., 2006). We see that the distributed algorithms perform as well and, in some cases better than the centralized algorithms they seek to emulate. This behavior can attributed to a hypothesis that the residual belief propagation is perhaps too greedy in selecting which messages to update next; in this case, adding randomness to the algorithm helps. Clearly, we need a global coordination step is missing from the algorithm; the convergence of the algorithm effectively stops after some time.

Figure 5 shows the convergence of the distributed algorithms as a function of bandwidth consumed. We see that the global coordination, required by the exponential scheme, increases the communication complexity of the solution only mildly.

7.2 Junction tree inference

We evaluate junction tree inference on the sensor calibration dataset (Paskin & Guestrin, 2004). In our Overlog implementation [3], the link quality information is provided externally, rather than being estimated explicitly from the configuration messages. The paper (Paskin et al., 2005) described two ways to optimize the junction tree; we implemented the simpler heuris-

[3]The overlog code is listed in appendix A

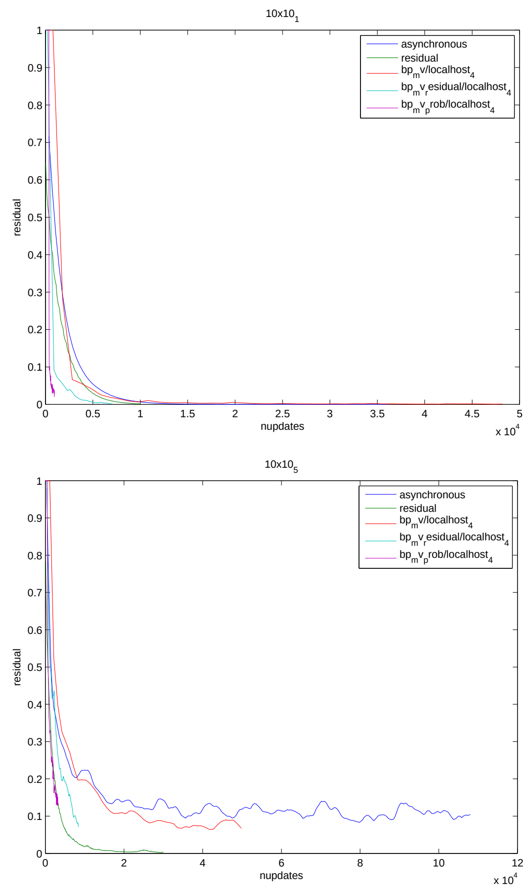

Figure 4: The convergence (residual) of four loopy belief propagation methods we implemented for (a) $C = 1$ and (b) $C = 5$. The curves labeled `asynchronous` and `residual` refer to the corresponding centralized algorithms.

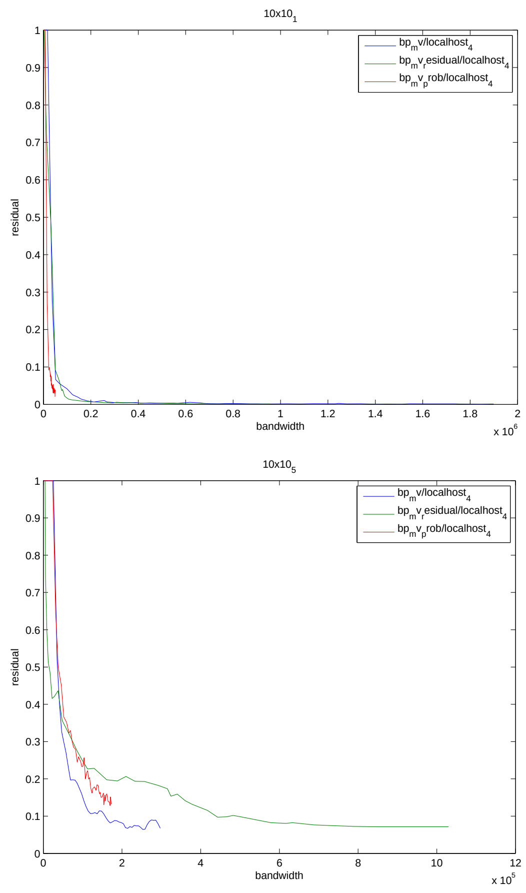

Figure 5: The convergence (residual) of four loopy bp methods as a function of the bandwidth used. (a) C=1, (b) C=5.

tic advocated that assigns each edge of the communication network a score based on the link quality and number of shared local variables. This heuristic giving preference to strong links with many shared variables often performs very well in practice.

Figure 6 shows the quality of the junction tree (measured in terms of the expected cost of performing inference on the tree), as a function of the bandwidth used in maintaining the data structure.

Table 1 illustrates the conciseness of the implementation compared to hand-coded Lisp implementation of Paskin et al. (2005).

Protocol	Lisp	Overlog
spanning tree	848 lines (9452)	274 lines (2589)
triangulation	457 lines (5812)	105 lines (1092)
sum–product	574 lines (6040)	131 lines (1144)

Table 1: The number of lines and the program size (in gzip-bytes) of different modules in Overlog and Lisp.

8 DISCUSSION

We propose a novel combination of network overlays and declarative programming to simplify the design and implementation of distributed inference algorithms. We demonstrate the effectiveness of our approach on a number of distributed inference algorithms

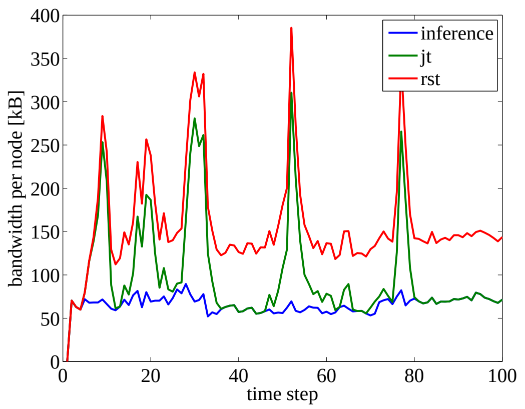

Figure 6: An example run for a small network with 10 nodes. Partition occurrs at time 25, and the communication is restored at time 40

and complement existing randomized schemes with a new analysis of their convergence. We demonstrate our results on simulated and real data, evaluated on a real network deployment.

The proposed approach offers fundamentally new opportunities. It enables algorithms that, in addition to local coordination, have a global coordination component. The global coordination can be performed without forcing the algorithm designer to think about low-level networking. The declarative specification of the algorithm is very transparent and is easier to debug. In the process of writing this paper, we have found multiple bugs in existing distributed algorithms, and we were able to correct these errors very quickly. We believe that, by simplifying the design and implementation of the algorithms, it will be easier to evaluate new algorithms on real systems, in realistic conditions.

Acknowledgements

We would like to thank Tyson Condie for his help with P2.

References

Elidan, G., Mcgraw, I., & Koller, D. (2006). Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI)*, Boston, Massachussetts.

Kearns, M., Tan, J., & Wortman, J. (2008). Privacy-preserving belief propagation and sampling. In Platt, J. C., Koller, D., Singer, Y., & Roweis, S. (Eds.), *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA.

Loo, B. T., & Hellerstein, J. (2007). Declarative routing: Extensible routing with declarative queries. *SIGCOMM 2005*.

Murphy, K. P., Weiss, Y., & Jordan, M. I. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pp. 467–475.

Paskin, M., Guestrin, C., & Mcfadden, J. (2005). A robust architecture for distributed inference in sensor networks.. pp. 55–62.

Paskin, M. A., & Guestrin, C. E. (2004). Robust probabilistic inference in distributed systems. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 436–445. AUAI Press.

Schiff, J., Antonelli, D., Dimakis, A. G., Chu, D., & Wainwright, M. J. (2007). Robust message-passing for statistical inference in sensor networks. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pp. 109–118.

Schmidt, R., & Aberer, K. (2006). Efficient peer-to-peer belief propagation. In *OTM Conferences (1)*, pp. 516–532.

Stoica, I., Morris, R., Karger, D., Kaashoek, F. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, *31*(4), 149–160.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., & Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 255–270, Boston, MA. USENIX Association.

A Junction Tree Inference

Here we provide the full Overlog specification for junction tree inference. The rules for setting the running intersection property are described in Figure 7 while the inference rules are present in Figure 8. The Overlog for spanning tree (used by junction tree) is in Figure 9 & 10. For a detailed explanation of the distributed spanning tree algorithm please refer (Paskin et al., 2005). All the base relations in the Overlogs have been highlighted in italic font. The three layers: spanning tree, running intersection property (junction tree) and inference work together and get updated in case of node/link failures and additions.

```
jtUpdate(@Node, Time) :-
        periodic(@Node, E, JT_EPOCH),
        Time := f_timerElapsed().

jtNbrUpdate(@Node, Nbr) :-
        jtUpdate(@Node),
        edge(@Node, Nbr).

reachvars(@Node, Node, Vars) :-
        localVars(@Node, Vars).

reachvarsIncoming(@Node, Node, Vars) :-
        reachvars(@Node, Nbr, Vars),
        Node == Nbr.

/* Union of incoming reachvar message.  Localvars are included*/
reachvars(@Target, Node, a_UNION<Vars>) :-
        jtNbrUpdate(@Node, Target),
        reachvarsIncoming(@Node, Nbr, Vars),
        Nbr != Target.

/* Incoming message.  The message is deleted when the edge gets deleted.  */
reachvarsIncoming(@Node, Nbr, Vars) :-
        reachvars(@Node, Nbr, Vars),
        edge(@Node, Nbr).

/* Compute the clique at each node */
cliqueRIP(@Node, a_UNION<Vars>) :-
        jtUpdate(@Node),
        reachvarsIncoming(@Node, Nbr1, Vars1),
        reachvarsIncoming(@Node, Nbr2, Vars2),
        Nbr1 != Nbr2,
        Vars := Vars1 & Vars2.

clique(@Node, Vars) :-
        cliqueRIP(@Node, RipVars),
        localVars(@Node, LocalVars),
        Vars := RipVars | LocalVars.

/* Compute the separator.  */
separator(@Node, Nbr, Vars) :-
        reachvarsIncoming(@Node, Nbr, NbrVars),
        clique(@Node, MyVars),
        Node != Nbr,
        Vars := MyVars & NbrVars.
```

Figure 7: *Overlog for Running Intersection Property.*

```
incoming(@Node, Node, Factor) :-
        localFactor(@Node, Factor).

/***********************************
One round of updates
***********************************/

jtinfUpdate(@Node, Time) :-
        periodic(@Node, E, JTINF_EPOCH),
        started(@Node),
        Time := f_timerElapsed(),
        /* Each node must have a local factor.  */
        incoming(@Node, Node, _).

nbrUpdate(@Node, Nbr) :-
        jtinfUpdate(@Node, _),
        edge(@Node, Nbr).

/* Update the separator set.  */
separator_set
separatorSet(@Node, Nbr, a_mkList<Var>) :-
        nbrUpdate(@Node, Nbr),
        separator(@Node, Nbr, Var).

/* Calculate the factors whose product forms the messge.  */
msgFactors(@Node, TargetNbr, a_mkList<F>) :-
        nbrUpdate(@Node, TargetNbr),
        /* includes the local factor */
        incoming(@Node, Nbr, F),
        Nbr != TargetNbr.

/* Compute the message.  */
message(@Nbr, Node, NewF) :-
        msgFactors(@Node, Nbr, MessageFactors),
        separatorSet(@Node, Nbr, Retain),
        FProd := f_product(MessageFactors),
        NewF := f_marginal(FProd, Retain).

/* The incoming message.  Message is deleted on edge deletion*/
incoming(@Node, Nbr, Factor) :-
        message(@Node, Nbr, Factor),
        edge(@Node, Nbr).

/********************************
Calculate beliefs
********************************/
beliefFactors(@Node, a_mkList<Factor>) :-
        jtinfUpdate(@Node, Time),
        incoming(@Node, _, Factor).

belief(@Node, Factor) :-
        beliefFactors(@Node, BeliefFactors),
        Factor := f_product(BeliefFactors).

factorCount(@Node, Count) :-
        beliefFactors(@Node, Factors),
        Count := f_size(Factors).
```

Figure 8: *Overlog for Junction Tree Inference.*

```
/* Update each node's own root pulse time.  */
pulse(@Node, MYID, Time) :-
        started(@Node),
        periodic(@Node, E, ROUTING_EPOCH),
        Time := f_timerElapsed().

/* Insert the default pulse for a root we have never heard.  */
pulse(@Node, RootId, Pulse) :-
        config(@Node, _, _, RootId, Pulse),
        notin pulse(@Node, RootId,_).

/* Update the pulse for the current parent.  */
pulse(@Node, RootId, Pulse) :-
        config(@Node, Nbr, _, RootId, Pulse),
        parent(@Node, Nbr).

/* Update the root for the current parent.  */
root(@Node, RootId) :-
        config(@Node, Nbr, _, RootId, Pulse),
        parent(@Node, Nbr).

/*********************************************************************************
Update the internal state and configurations by selecting the new best parent.
*********************************************************************************/

/* Run a single update.  */
updateparent(@Node) :-
        periodic(@Node, E, ROUTING_EPOCH).

/* The cost of changing a parent.  */
newparent(@Node, a_max<InvCostParent>) :-
        updateparent(@Node),
        config(@Node, Nbr, NbrParent, NbrRootId, NbrPulse),
        pulse(@Node, NbrRootId, OldPulse),
        parent(@Node, OldParent),
        root(@Node, OldRootId),
        NbrRootId <= OldRootId, /* Neighbor has a better root */
        Nbr != OldParent, /* Neighbor is not already my parent */
        NbrPulse > OldPulse, /* Neighbor is not my descendant.  */
        NbrParent != Node, /* Neighbor is not my child:  avoid cycles */
        link(@Node, Nbr, _, PReceive),
        Cost := 1.0 / PReceive + ROUTING_SWITCH_COST,
        InvCostParent := f_cons(1.0 - Cost, Nbr).

/* The cost of keeping the parent */
newOldparent(@Node, NewParent, a_max<InvCostParent>) :-
        newparent(@Node, NewParent),
        config(@Node, Nbr, NbrParent, NbrRootId, NbrPulse),
        parent(@Node, Nbr),
        NbrRootId < MYID,
        NbrParent != Node,
        link(@Node, Nbr, _, PReceive),
        Cost := 1.0 / PReceive,
        InvCostParent := f_cons(1.0 - Cost, Nbr).
```

Figure 9: *Spanning Tree Overlog.*

```
/* Local node is the root since no parent has a lower root.  */
bestparent(@Node, Node) :-
        newOldparent(@Node, NewParent, OldParent),
        f_size(NewParent) == 0,
        f_size(OldParent) == 0.

/* Select the best parent.  */
bestparent(@Node, Parent) :-
        newOldparent(@Node, NewParent, OldParent),
        CostParent := f_max(NewParent, OldParent),
        f_size(CostParent) > 0,
        Parent := f_last(CostParent).

/* Additional info about the parent.  */
bestParentInfo(@Node, Parent, RootId, Pulse) :-
        bestparent(@Node, Parent),
        config(@Node, Parent, _, RootId, Pulse).

bestParentInfo(@Node, Node, MYID, Pulse) :-
        bestparent(@Node, Parent),
        Parent == Node,
        pulse(@Node, MYID, Pulse).

/* Update the parent, the root, and the pulse.  */
parent(@Node, Parent) :-
        bestParentInfo(@Node, Parent, _, _).

root(@Node, RootId) :-
        bestParentInfo(@Node, _, RootId, _).

pulse(@Node, RootId, Pulse) :-
        bestParentInfo(@Node, _, RootId, Pulse).

/* Send a configuration message describing Node's state to all neighbors.  */
config(@Nbr, Node, Parent, RootId, Pulse) :-
        bestParentInfo(@Node, Parent, RootId, Pulse),
        linkEnabled(@Node, Nbr).

configBroadcast(@Base, Node, Parent, RootId, Pulse) :-
        bestParentInfo(@Node, Parent, RootId, Pulse),
        Base := BASE_ADDR.

/* Establish bidirectional edges, used by upper levels.  */
edge(@Node, Parent) :-
        parent(@Node, Parent).

edge(@Parent, Node) :-
        parent(@Node, Parent).

/*********************************************************************
Maintain a relation of active links.
*********************************************************************/
config_inserted(@Node, Nbr) :-
        config(@Node, Nbr, Parent, RootId, Pulse).

/* Update the link age.  */
link(@Node, Nbr, PSend, PReceive, Time) :-
        config_inserted(@Node, Nbr),
        link(@Node, Nbr, PSend, PReceive, _),
        Time := f_timerElapsed().
```

Figure 10: *Spanning Tree Overlog.*