Journal of Machine Learning Research 1 (2009) 1-48 Submitted 4/00; Published 10/00

Parallel Splash Belief Propagation

Joseph Gonzalez JEGONZAL@CS.CMU.EDU
Machine Learning Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA

Yucheng Low YLOW@CS.CMU.EDU
Machine Learning Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA

Carlos Guestrin GUESTRIN@CS.CMU.EDU
Machine Learning Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA

Editor: Other People.

Abstract

As computer architectures transition towards exponentially increasing parallelism we are forced to adopt parallelism at a fundamental level in the design of machine learning algorithms. In this paper we focus on parallel graphical model inference. We demonstrate that the natural, synchronous parallelization of belief propagation is highly inefficient. By bounding the achievable parallel performance in chain graphical models we develop a theoretical understanding of the parallel limitations of belief propagation. We then provide a new parallel belief propagation algorithm which achieves optimal performance. Using several challenging real-world tasks, we empirically evaluate the performance of our algorithm on large cyclic graphical models where we achieve near linear parallel scaling and out perform alternative algorithms.

Keywords: Parallel Algorithms, Graphical Models, Inference, Belief Propagation

1. Introduction

Exponential gains in hardware technology have enabled increasingly sophisticated machine learning techniques to be applied to rapidly growing real-world problems. Physical and economic limitations have forced computer architecture towards parallelism and away from exponential frequency scaling. Meanwhile, increased access to ubiquitous sensing and the web have resulted in an explosion in the size of machine learning tasks. In order to benefit from current and future trends in processor technology we must discover, understand, and exploit the available parallelism in machine learning.

The true potential of parallelism is more than just a linear speedup: it is the ability to automatically access present and future exponential gains in processor technology. While parallel computing is a well developed field with a rich research history, decades of exponential frequency scaling have, until recently, hindered its mass adoption. Around 2003, a wide range of factors including wire delay, clock jitter, and infeasible power and cooling demands (Asanovic et al., 2006), forced processor manufactures away from frequency scaling and towards parallel scaling. Driven by the sustained transistor doubling promised by Moore's law and multi-core inspired market forces, it is widely believed that the parallelism available on a single platform will continue to grow exponentially for the foreseeable future (Asanovic et al., 2006).

As ubiquitous online services powered by large parallel clusters catalog the flood of information we produce every day, we are faced with an explosion in the computational scale of machine learning. The blogosphere and online image and video databases are growing rapidly enabling richer automated learning opportunities but defying our sequential computational abilities. By exposing cluster level parallelism our algorithms will be able to scale with data collection systems.

Ideally, we would like to expose parallelism throughout machine learning by developing only a few core parallel algorithms. Graphical models provide a common language for representing statistical models in machine learning. Inference, the process of calculating marginals and conditionals of probability distributions, is the primary computationally intensive procedure in learning and reasoning with graphical models. Therefore by providing an efficient parallel graphical model inference algorithm, we can expose parallelism in a wide range of machine learning applications.

While there are many popular algorithms for inference in graphical models, one of the most popular is belief propagation. In this work, we develop a theoretical understanding of the available parallelism in the belief propagation algorithm, revealing the hidden sequential structure of the parallel message scheduling. Using our theoretical foundation, we develop Parallel Splash belief propagation a new parallel approximate inference algorithm, which performs asymptotically better than the natural direct parallelization of belief propagation. Furthermore we find that the Splash Belief Propagation algorithm outperforms synchronous, round-robin, wild-fire (Ranganathan et al., 2007), and residual (Elidan et al., 2006) belief propagation even in the single processor setting. We thoroughly evaluate the performance of the Parallel Splash belief propagation algorithm on several challenging real-world tasks using high performance shared and distributed memory systems. More specifically, our key contributions are:

- A Map-Reduce based parallelization of Synchronous belief propagation and an analysis of its parallel scaling and efficiency.
- The τ_ϵ-approximation for characterizing approximate inference and a lower bound on the theoretically achievable parallel running time for belief propagation.
- The ChainSplash algorithm for optimal parallel inference on chain graphical models.
- The Splash operation which generalizes the optimal forward-backward subroutine of the ChainSplash algorithm to arbitrary cyclic graphical models.
- Belief residual prioritization and convergence assessment for improved dynamic scheduling
- The DynamicSplash Operation which uses belief residual scheduling to automatically and dynamically adjust the shape and size of the Splash operation
- The efficient shared and distributed memory Parallel Splash inference algorithms which combine the DynamicSplash procedure with the new belief residual scheduling.
- A simple, effective over-segmentation procedure to improve work balance in the distributed setting.
- An extensive empirical evaluation of our new algorithm and its components in both the shared and distributed memory setting on large scale synthetic and real-world inference problems.

2. Computational Models

Unlike the sequential setting where most processor designs faithfully execute a single random access computational model, the parallel setting has produced a wide variety of parallel architectures. In this work we focus on Multiple Instruction Multiple Data (MIMD) parallel architectures which includes multicore processors and parallel clusters. The MIMD architecture permits each processor to asynchronously execute its own sequence of instruction (MI) and operate on separate data (MD). We exclude Graphics Processing Units (GPUs) which are more typically[1] considered Single Instruction Multiple Data (SIMD) parallel architectures. Furthermore we divide the MIMD architectures into Shared Memory (SM-MIMD) represented by multi-core processors and Distributed Memory (DM-MIMD) represented by large commodity clusters. This section will briefly review the design and unique opportunities and challenges afforded by shared and distributed memory MIMD systems in the context of parallel machine learning.

2.1 Shared Memory

The shared-memory architectures are a natural generalization of the standard sequential processor to the parallel setting. In this work we assume that each processor is identical and has symmetric access to a single shared memory. All inter-processor communication is accomplished through shared memory and is typically very fast (on the order of 10s of clock cycles). Consequently, shared memory architectures are particularly convenient because they do not require partitioning the algorithm state and enable frequent interaction between processors.

The class of parallel architectures that implement symmetric access to shared memory are commonly referred to as symmetric multi-processors (SMP). Most modern SMP systems are composed of one or more multi-core processors. Each multi-core chip is itself composed of several processors typically sharing a local cache. These systems must therefore implement sophisticated mechanisms to ensure cache coherency, limiting the potential for parallel scaling. Nonetheless, moderately sized mutli-core/SMP can be found in most modern computer systems and even in many embedded systems. It is expected that the number of cores in standard SMP systems will continue to increase (Asanovic et al., 2006).

While multi-core systems offer the convenience of shared memory parallelism they introduce challenges in data intensive computing. By dividing the already limited bandwidth to main memory over several processors, multi-core systems quickly become memory bandwidth constrained. Therefore, efficient multi-core algorithms need to carefully manage memory access patterns.

2.2 Distributed Memory and Cluster Computing

Distributed memory architectures are composed of a collection of independent processors each with its own local memory and connected by a relatively slow communication network. Distributed memory algorithms must therefore address the additional costs associated with communication and distributed synchronization. While there are many performance and connectivity models for distributed computing (Bertsekas and Tsitsiklis, 1989), most provide point-to-point routing, a maximum latency which scales with the number of processors, and a limited bandwidth.

While the distributed setting often considers systems with network and processor failure, in this work we assume that the all resources remain available throughout execution and that all mes-

1. Recent trends suggest that future GPUs will increasingly permit MIMD style parallelism

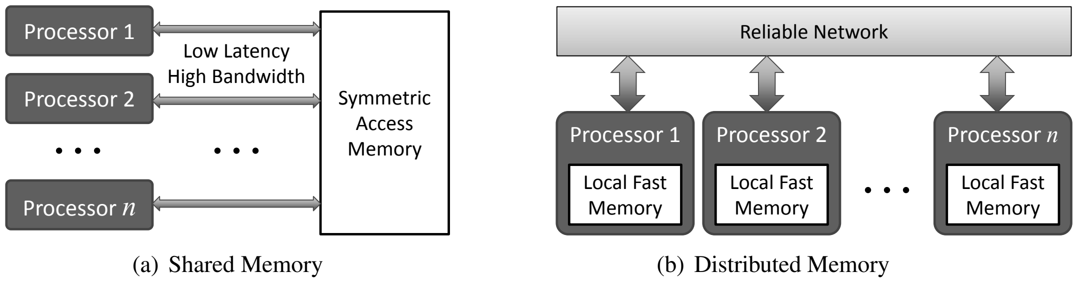


Figure 1: The two common parallel architectures encountered in typical commodity hardware. **(a)** The standard shared memory architectural model consists of a collection of processors that have low latency high bandwidth access to a single uniform address space. All inter-processor communication is accomplished through the shared memory. **(b)** The standard distributed memory architectural model consists of a collection of processors each with their own local memory which are able to communicate by passing messages across a reliable but typically slow communication network.

sages eventually reach their destination. While this assumption is unrealistic on extremely large commodity parallel systems we believe that a relatively simple check-pointing mechanism would be adequate for the algorithms we present and more sophisticated failure recovery mechanisms are beyond the scope of this work.

Computer clusters ranging from several hundreds to thousands of processors are the most common instantiation of the distributed memory model. These system typically employ fast commodity networks with switch hierarchies or highly tuned system specific interconnects. Often distributed memory systems are composed of smaller shared memory systems, typically having several multi-core processors at each node. Here we will use the term node to refer to a single machine on the network and we will use the term processor to refer each processing element. For instance, a network of 8 machines, each with 2 quad-core CPUs, has 8 nodes, 64 processors.

By eliminating the need for cache coherency, the distributed memory model permits the construction of substantially larger systems and also offers several key advantages to highly data intensive algorithms. Because each processor has its own memory and dedicated bus, the distributed memory setting provides linear scaling in memory capacity and memory bandwidth. In heavily data intensive algorithms linear scaling in memory can provide substantial performance gains by eliminating memory bottlenecks and permitting *super-linear* performance scaling.

Because of the latency associated with network communication and the limited network bandwidth, efficient algorithms in the distributed memory model must minimize inter-processor communication. Consequently, efficient distributed memory algorithms must deal with the challenges associated with partitioning both the state and execution in a way that minimizes network congestion while still ensuring that no one processor has disproportionally greater work.

3. Graphical Models

Large probabilistic models are frequently used to represent and reason about uncertainty. Graphical models exploit conditional independence assumptions to provide a compact representation for large probabilistic models. As we will demonstrate, both the graphical structure as well as the associated factors combine to form the basis for the limiting sequential computational component associated with operations on large probability distributions. Here we will briefly discuss the classes of graphical models considered in this work and introduce the necessary notation.

Many important probabilistic models may be represented by factorized distributions of the form:

$$\mathbf{P}\left(x_1, \ldots, x_n \,|\, \theta\right) = \frac{1}{Z(\theta)} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{x_c} \,|\, \theta), \tag{3.1}$$

where $\mathbf{P}\left(x_1, \ldots, x_n\right)$ is the probability mass function for the set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$. Each clique $c \in \mathcal{C}$, is a small subset, $c \subset \{1, \ldots, n\}$, of indicies and the clique set $\mathcal{C}$ is the set of cliques. The factors $\mathcal{F} = \{\psi_c : c \in \mathcal{C}\}$ are un-normalized positive functions, $\psi_c : \mathbf{x_c} \to \mathbb{R}^+$, which map the assignments of subsets of random variables to the positive reals and depend on the parameters θ. We use the bold-faced $\mathbf{x_c}$ to denote an assignment to the subset of variables in the clique c. Since the focus of this work is inference and not learning[2], we will assume that the parameters, represented by θ, are fixed in advance. For notational simplicity we will omit the conditional parameters from the factors. The log-partition function $Z(\theta)$ is the normalizing constant which depends only on the parameters θ and has the value

$$Z(\theta) = \sum_{\mathcal{X}} \prod_{\alpha \in \mathcal{C}} \psi_c(\mathbf{x_c} \,|\, \theta), \tag{3.2}$$

computed by summing over the exponentially many joint assignments. We focus on discrete random variables $X_i \in \{1, \ldots, A_i\} = \mathcal{A}_i$ taking on a finite set of A_i discrete values, even though the algorithms and techniques proposed in later sections may be easily extended to Guassian random variables (Weiss and Freeman, 2001) or other continuous variables (Ihler and McAllester, 2009). For notational convenience we will define $A_{\psi_c} = \prod_{i \in c} A_i$ as the size of the domain of ψ_c.

3.1 Factor Graphs

Factorized distributions of the form Eq. (3.1) may be naturally represented as an undirected bipartite graph $G = (V, E)$ called a factor graph. The vertices $V = \mathcal{X} \cup \mathcal{F}$ correspond to the variables and factors and the edges $E = \{\{\psi_\alpha, X_i\} : i \in \alpha\}$ connect factors with the variables in their domain. To simplify notation, we use $\psi_i, X_j \in V$ to refer to vertices when we wish to distinguish between factors and variables, and $i, j \in V$ otherwise. We define Γ_i as the neighbors of i in the factor graph. In Fig. 2 we illustrate a simple factor graph with 4 variables and 2 factors.

Factor graphs can be used to compactly represent a wide range of common probabilistic models, from Markov Logic Networks (MLNs) (Domingos et al., 2008) for natural language processing to pairwise Markov Random Fields (MRFs) for protein folding (Yanover and Weiss, 2002) and image processing (A. Saxena, 2007; Li, 1995). The clique sizes and connectivity of these graphs varies widely. The factor graphs corresponding to MLNs encode the probability of truth assignments to a logic and can often have large factors and a highly irregular connectivity structure with a few variables present in many of the factors. Meanwhile, the factor graphs corresponding to pairwise Markov Random Fields have clique sizes of at most two (e.g., $\forall c \in \mathcal{C} : |c| \leq 2$). Pairwise MRFs may be represented by an undirected graph where the vertices correspond to variables and their

2. It is important to note that while this focus of this work is graphical model inference and not the learning, inference is a key step in parameter learning.

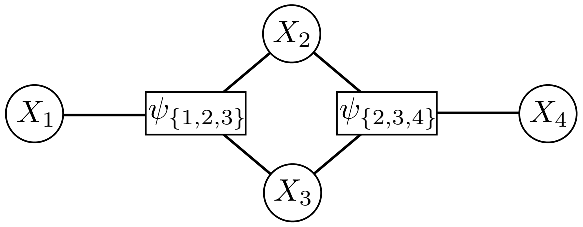


Figure 2: The factor graph corresponding to the factorized distribution $\mathbf{P}(x_1, x_2, x_3, x_4) \propto \psi_{\{1,2,3\}}(x_1, x_2, x_3)\psi_{\{2,3,4\}}(x_2, x_3, x_4)$. The circles represent variables and the squares represent factors.

associated unary factors and edges correspond to the binary factors. Many MRFs are constructed from regular templates which form grids, which are often be planar in the case of image processing, or small three dimensional cliques in the case of protein folding. In addition, Bayesian Networks, which are used in a wide range of applications including Hidden Markov Models, can be naturally transformed into a factor graph by replacing the associated conditional probability tables with factors and connecting all the variables in the domain. Like Markov Logic Networks, the corresponding factor graphs can have highly irregular structures.

3.2 Belief Propagation

Estimating marginal distributions is essential to learning and reasoning about large graphical models. While graphical models provide a compact representation, computing marginals, conditionals, and even the joint probability can often be intractable. Computing exact marginals and conditionals is NP-hard in general (Cooper, 1990) and even computing bounded approximations is known to be NP-hard (Roth, 1993). Nonetheless, there are several approximate inference procedures which often perform well in practice. In this work we focus on Belief Propagation, one of the more popular approximate inference algorithms, which is often believed to be naturally amenable to parallelization (Mendiburu et al., 2007; Sun et al., 2003).

Belief propagation, or the Sum-Product algorithm, was originally proposed by Pearl (1988) and estimates variable and clique marginals by iteratively updating parameters along edges in the factor graph until convergence. The edge parameters in belief propagation are typically referred to as "messages" which are computed in both directions along each edge. The message sent from variable X_i to factor ψ_j along the edge (X_i, ψ_j) is given in Eq. (3.3) and the message sent from factor ψ_j to vertex X_i along the edge (ψ_j, X_i) is given in Eq. (3.4).

$$m_{X_i \to \psi_j}(x_i) \propto \prod_{k \in \Gamma_i \setminus j} m_{k \to i}(x_i) \tag{3.3}$$

$$m_{\psi_j \to X_i}(x_i) \propto \sum_{\mathbf{x_j} \setminus x_i} \psi_j(\mathbf{x_j}) \prod_{k \in \Gamma_j \setminus i} m_{k \to j}(x_k) \tag{3.4}$$

The sum, $\sum_{\mathbf{x_j} \setminus x_i}$, is computer over all assignments to $\mathbf{x_j}$ excluding the variable x_i, and the product, $\prod_{k \in \Gamma_j \setminus i}$, is computed over all neighbors of the vertex ψ_j excluding vertex X_i. The messages are typically normalized to ensure numerical stability. The local variable and factor marginals can be estimated by combining the messages:

$$\begin{aligned} \mathbf{P}(X_i = x_i) &\approx b_{X_i}(x_i) \propto \prod_{j \in \Gamma_i} m_{j \to i}(x_i) \\ \mathbf{P}(\mathbf{X_i} = \mathbf{x_i}) &\approx b_{\mathbf{X_i}}(\mathbf{x_i}) \propto \psi_i(\mathbf{x_i}) \prod_{j \in \Gamma_i} m_{j \to i}(x_j). \end{aligned} \tag{3.5}$$

In acyclic factor graphs, messages are computed sequentially starting from the leaves and computing only messages in the direction of an arbitrarily chosen root, and then the process is reversed computing only the messages in the opposite direction. This message update scheduling is often called the forward-backward scheduling and was shown by Pearl (1988) to yield exact marginals using $O(2|E|)$ message calculations.

Unfortunately, choosing the best schedule on loopy graphs is often difficult and can depend heavily on the factor graph structure and even the model parameters. For simplicity, many applications of BP adopt a **synchronous** schedule in which all messages are *simultaneously* updated using messages from the previous iteration. Otherwise, some type of **asynchronous** schedule is employed, in which messages are updated *sequentially* using the most recent inbound messages. For example, the popular **round-robin** asynchronous schedule, sequentially updates the messages in fixed order which is typically a random permutation over the vertices.

More recent advances by Elidan et al. (2006) and Ranganathan et al. (2007) have focused on **dynamic** asynchronous schedules, in which the message update order is determined as the algorithm proceeds. Other recent work by Wainwright et al. (2001) have focused on tree structured schedules, in which messages are updated along collections of spanning trees. By dynamically adjusting the schedule and by updating along spanning trees, these more recent techniques attempt to indirectly address the schedule dependence on the model parameters and structure. As we will see in this chapter, by varying the BP schedule we can affect the speed, convergence, and parallelism of BP.

Independent of the update schedule, messages are computed until the change in message values between consecutive computations is bounded by small constant $\beta \geq 0$:

$$\max_{(i,j) \in E} \left|\left| m_{i \to j}^{(\text{new})} - m_{i \to j}^{(\text{old})} \right|\right|_1 \leq \beta. \tag{3.6}$$

Belief propagation converges if at some point Eq. (3.6) is achieved. Unfortunately, in cyclic graphs there are limited guarantees that belief propagation will converge (Tatikonda and Jordan, 2002; Ihler et al., 2005; Mooij and Kappen, 2007). To improve convergence in practice, damping of the form given in Eq. (3.7) is often used with $\alpha \in [0, 1)$. In our theoretical analysis, we will assume $\alpha = 0$ however in our experiments we will typically use a damping value $\alpha = 0.3$.

$$m_{\psi_j \to X_i}(x_i)^{(\text{new})} \propto \alpha \times m_{\psi_j \to X_i}(x_i)^{(\text{old})} + (1 - \alpha) \sum_{\mathbf{x_j} \setminus x_i} \psi_j(\mathbf{x_j}) \prod_{k \in \Gamma_j \setminus i} m_{k \to j}(x_k) \tag{3.7}$$

While there are few guarantees for convergence or correctness, belief propagation on cyclic graphs is used extensively with great success as an approximate inference algorithm (McEliece et al., 1998; Sun et al., 2003; Yedidia et al., 2003; Yanover and Weiss, 2002). For a more detailed review of the belief propagation algorithm and it generalizations we recommend reading the tutorial by Yedidia et al. (2003).

3.3 Opportunities for Parallelism in Belief Propagation

Belief propagation offers several opportunities for parallelism. At the graph level, multiple messages can be computed in parallel. At the factor level, individual message calculations (sums and products) can be expressed as matrix operations which can be parallelized relatively easily (See Bertsekas and Tsitsiklis (1989) for details). For typical message sizes where $A_i << n$, graph level parallelism is asymptotically more beneficial than factor level parallelism. Therefore, we will not provide a treatment of message level parallelism in this paper, but concentrate on graph level parallelism. Running time will be measured in terms of the number of message computations, treating individual message updates as atomic unit time operations.

Xia and Prasanna (2008) and Pennock (1998) explored parallel transformations of the graph which go beyond the standard message calculations. While these techniques expose additional parallelism they require low treewidth models and tractable exact inference, which are strong requirements when operating on massive real-world distributions. In this work we restrict our attention to message passing algorithms on the original graph and do not require low treewidth models.

4. Parallel Synchronous Belief Propagation

Synchronous belief propagation is an inherently parallel algorithm. Given the messages from the previous iteration, all messages in the current iteration can be computed simultaneously and independently without communication. This form of completely independent computation is often deemed embarrassingly parallel[3]. In this section we will introduce, the Bulk Synchronous Parallel belief propagation (BSP-BP) algorithm, which is a natural parallelization of Synchronous belief propagation. We will also show how the BSP-BP algorithm can be represented in the popular Map-Reduce framework for large scale data parallel algorithms.

4.1 Bulk Synchronous Parallel Belief Propagation

A Bulk Synchronous Parallel (BSP) algorithm is an iterative parallel algorithm where each iteration consists of an embarrassingly parallel computation phase followed by a synchronized communication phase. All processors must complete the previous iteration before a processor may enter the next iteration. We define Synchronous BP in the BSP model in Alg. 1 where on each iteration all messages are computed in parallel in the computation phase and then the new messages are exchanged among processors in the communication phase. In the shared memory setting messages are exchanged by keeping entirely separate old and new message sets and swapping after each iteration. In the distributed memory setting messages must be sent over the communication network.

In each iteration of the BSP-BP algorithm (Alg. 1) there are $O(|E|)$ message computations all of which may run in parallel. This fully synchronous form of parallel update is referred to as a Jacobi update in traditional parallel algorithms literature and is often considered ideal as it exposes substantial parallelism. The synchronization phase can be accomplished efficiently in the shared memory setting by swapping address spaces. However, in the distributed setting the synchronization may require substantial communication.

The task of assessing convergence in Alg. 1 requires additional care. In particular, we must identify whether there exists at least one message which has changed by more than β. In the shared

3. Embarrassing parallelism is the form of parallelism that is trivially achievable and which would be "embarrassing" not to exploit.

Algorithm 1: Bulk Synchronous Parallel Belief Propagation BSP-BP

```
$t \leftarrow 0$ ;
while Not Converged do
    // Embarrassingly Parallel Phase
    forall $(u, v) \in E$ do in parallel
        $m_{u \to v}^{(t+1)} \leftarrow$ Update($m^{(t)}$) ;
    // Synchronization Phase
    Exchange $m^{(t+1)}$ among processors ;
    $t \leftarrow t + 1$ ;
```

Algorithm 2: Map Function for Synchronous BP

```
Input  : A vertex $v \in V$ and all inbound messages $\{m_{i \to v}\}_{i \in \Gamma_v}$
Output: Set of outbound messages as key-value pairs $(j, m_{v \to j})$
forall $j \in \Gamma_v$ in the neighbors of $v$ do in parallel
    Compute Message $m_{v \to j}$ using $\{m_{i \to v}\}_{i \in \Gamma_v}$;
    return key-value pair $(j, m_{v \to j})$;
```

memory setting, convergence assessment is easily accomplished by maintaining concurrent-read, concurrent-write access to a global boolean flag. In the distributed memory setting, one does not have the convenience of a cheap globally accessible memory pool and more complex algorithms are needed. We will return to the task of global termination assessment in greater detail in Sec. 9.3.

4.1.1 MAP-REDUCE BELIEF PROPAGATION

The Synchronous BP algorithm may also be expressed in the context of the popular Map-Reduce framework. The Map-Reduce framework, introduced by Dean and Ghemawat, concisely and elegantly represents algorithms that consist of an embarrassingly parallel map phase followed by a reduction phase in which aggregate results are computed. Because Map-Reduce simplifies designing and building large scale parallel algorithms, it has been widely adopted by the data-mining and machine learning community (Chu et al., 2006). The Map-Reduce abstraction was not originally designed for iterative algorithms and so many of the standard implementations incur a costly communication and disk access penalty between iterations.

A Map-Reduce algorithm is specified by a `Map` operation, which is applied to each data atom in parallel, and a `Reduce` operation, which combines the output of the mappers. Synchronous BP can naturally be expressed as an iterative Map-Reduce algorithm where the `Map` operation (defined in Alg. 2) is applied to all vertices and emits destination-message key-value pairs and the `Reduce` operation (defined in Alg. 3) joins messages at their destination vertex and prepares for the next iteration.

4.2 Runtime Analysis

While it is not possible to analyze the running time of parallel Synchronous belief propagation on a general cyclic graphical model, we can analyze the running time on tree graphical models. To aid

```
Algorithm 3: Reduce Function for Synchronous BP
    Input  : The key-value pairs $\{(v, m_{i\to v})\}_{i\in\Gamma_v}$
    Output: The belief $b_v$ for vertex $v$ as well as the $\{(v, m_{i\to v})\}_{i\in\Gamma_v}$ pairs $(j, m_{v\to j})$
    Compute the belief $b_v$ for $v$ using $\{(v, m_{i\to v})\}_{i\in\Gamma_v}$;
    Return $b_v$ and $\{(v, m_{i\to v})\}_{i\in\Gamma_v}$;
```

in our analysis we introduce the concept of awareness. Intuitively, awareness captures the "flow" of message information along edges in the graph. If messages are passed *sequentially* along a chain of vertices starting at vertex i and terminating at vertex j then vertex j is aware of vertex i.

Definition 4.1 (Awareness) *Vertex j is **aware** of vertex i if there exists a path $(v_1, \ldots, v_k)$ from $v_1 = i$ to $v_k = j$ in the factor graph G and a sequence of messages $[m^{(t_1)}_{v_1\to v_2}, \ldots, m^{(t_k)}_{v_{k-1}\to v_k}]$ such that each message was computed after the previous message in the sequence $t_1 < \ldots < t_k$.*

In Theorem 4.1 we use the definition of awareness to bound the running time of Synchronous belief propagation on acyclic graphical models in terms of the longest path.

Theorem 4.1 (Parallel Synchronous BP Running Time) *Given an acyclic factor graph with n vertices, longest path length d, and $p \leq 2(n-1)$ processors, parallel synchronous belief propagation will compute exact marginals in time (as measured in number of vertex updates):*

$$\Theta\left(\frac{nd}{p} + d\right).$$

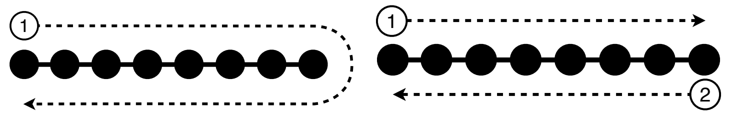
Proof On the k^{th} iteration of parallel synchronous belief propagation, all vertices are **aware** of all reachable vertices at a distance k or less. If the longest path contains d vertices then it will take d iterations of parallel Synchronous belief propagation to make all vertices aware of each other in an acyclic graphical model. Therefore, parallel Synchronous belief propagation will converge in d iterations. Each iteration requires $2(n-1)$ message calculations, which we divide evenly over p processors. Therefore it takes $\lceil 2\,(n-1)\,/p \rceil$ time to complete a single iteration. Thus the total running time is:

$$d\left\lceil \frac{2\,(n-1)}{p} \right\rceil \leq d\left(\frac{2\,(n-1)}{p} + 1\right) \in \Theta\left(\frac{nd}{p} + d\right)$$

■

If we consider the running time given by Theorem 4.1 we see that the $\frac{n}{p}$ term corresponds to the parallelization of each synchronous update while the overall running time is limited by the length of the longest path d which determines the number of synchronous updates. The length of the longest path is therefore the limiting sequential component which cannot be eliminated by scaling the number of processors.

As long as the number of vertices is much greater than the number of processors, the Synchronous algorithm achieves linear parallel scaling and therefore appears to be an optimal parallel algorithm. However, an optimal parallel algorithm should also be *efficient*. That is, the total work done by all processors should be asymptotically equivalent to the work done by a single processor running the *optimal* sequential algorithm. We will now show that the synchronous algorithm can actually be asymptotically inefficient.



(a) Single Sequential (b) Optimal Parallel

Figure 3: **(a)** The optimal forward-backwards message ordering for exact inference on a chain using a single processor. **(b)** The optimal message ordering for exact inference on a chain using two processors. Note that it is not possible to construct a parallel message scheduling which uses more than two processors without introducing additional unnecessary work.

4.2.1 ANALYSIS OF BULK SYNCHRONOUS BELIEF PROPAGATION ON CHAINS

In Theorem 4.1 we showed that the parallel runtime synchronous belief propagation depends on the length of the longest sub-chain. To illustrate the inefficiency of parallel synchronous belief propagation, we analyze the running time on a chain graphical model with n vertices. Chain graphical models act as a theoretical benchmark for parallel belief propagation because they directly capture the limiting sequential structure of belief propagation (through the concept of awareness) and can be seen as a sub-problem in both acyclic and cyclic graphical models.

Chain graphical models are the worst case graphical model structure for parallel belief propagation because they contain only a single long path. The branching structure of trees permits multiple longest paths each with the limiting parallel performance of the chain. In cyclic graphical models, the longest paths in the unrolled computation graph again reduces to the chain graphical model.

It is well known that the forward-backward schedule (Fig. 3(a)) for belief propagation on chains is optimal. The forward-backward schedule, as the name implies, sequentially computes messages $(m_{1\to 2}, \ldots, m_{n-1\to n})$ in the forward direction and then sequentially computes messages $(m_{n\to n-1}, \ldots, m_{2\to 1})$ in the backward direction. The running time of this simple schedule is therefore $\Theta(n)$ or exactly $2(n-1)$ message calculations.

If we run the parallel synchronous belief propagation algorithm using $p = 2(n-1)$ processors on a chain graphical model of length n, we obtain a running time of exactly $n-1$. This means that parallel synchronous belief propagation obtains only a *factor of two* speedup using two processors per edge, almost twice as many processors as the number of vertices. More surprisingly, if we use fewer than $n-1$ processors, the parallel synchronous algorithm will be *slower* than the simple sequential forward-backward algorithm running on a single processor. Finally, If we use any constant number of processors (for example $p = 2$), then the parallel synchronous algorithm will run in *quadratic* time while the sequential single processor algorithm will run in *linear* time.

What is the optimal parallel scheduling for belief propagation on chain graphical models? Recall that in the message update equations (Eq. (3.3) and Eq. (3.4)), the message $m_{i\to j}$ does not depend on the message $m_{j\to i}$. Therefore in the sequential forward-backward algorithm, messages in the forward pass do not interact with messages in the backward pass and therefore we can compute the forward and backward pass simultaneously as shown in Fig. 3(b). Using only $p = 2$ processors, one computing messages *sequentially* in the forward direction and one computing messages sequentially in the backward direction as shown in Fig. 3(b), we obtain a running time of $n-1$ and achieve a *factor of two* speedup using only two processors.

While messages may be computed in any order, information (awareness) is propagated *sequentially*. On every iteration of synchronous belief propagation only a few message computations (in the case of chain graphical models only two message computations) increase awareness while the rest are *wasted*. Unfortunately, in the case of chains there is no parallel scheduling which achieves greater than a factor of two speedup. Furthermore, it is unclear how to generalize the optimal forward-backward scheduling to arbitrary cyclic graphical models. In the next section we characterize approximate inference in *acyclic* graphical models and show how this can reduce the limiting sequential structure and expose greater parallelism.

5. τ_ϵ-Approximate Inference

Intuitively, for a long chain graphical model with weak (nearly uniform) edge potentials, distant vertices are approximately independent. For a particular vertex, an accurate approximation to the belief may often be achieved by running belief propagation on a small subgraph around that vertex. By limiting vertex awareness to its local vicinity, we can reduce the sequential component of belief propagation to the longest path in the subgraph. We define (in Definition 5.1) τ_ϵ as the diameter of the subgraph required to achieve an ϵ level approximation accuracy.

Definition 5.1 **(τ_ϵ-Approximation)** *Given an acyclic*[4] *factor graph, we define a τ-approximate message $\tilde{m}^{(\tau)}_{i\to j}$ as the message from vertex i to vertex j when vertex j is* aware *of all vertices within a distance of at least τ. Let m^* be the vector of all messages at convergence (after running the complete forward-backward algorithm). For a given error, ϵ, we define a τ_ϵ-Approximation as the smallest τ such that:*

$$\max_{\{u,v\}\in E} \left|\left|\tilde{m}^{(\tau)}_{u\to v} - m^*_{u\to v}\right|\right|_1 \leq \epsilon. \tag{5.1}$$

for all messages.

An illustration of Definition 5.1 is given in Fig. 5. Suppose in the sequential forward pass we change the message $m_{3\to 4}$ to a uniform distribution (or equivalently disconnect vertex 3 and 4) and then proceed to compute the rest of the messages from vertex 4 onwards. Assuming $\tau_\epsilon = 3$ for the chosen ϵ, the final approximate belief at b_7 would have less than ϵ error in some metric (here we will use L_1-norm) to the true probability $\mathbf{P}\left(X_{\tau_\epsilon+1}\right)$. Vertices τ_ϵ apart are therefore approximately independent. If τ_ϵ is small, this can dramatically *reduces* the length of the sequential component of the algorithm.

By Definition 5.1, τ_ϵ is the earliest iteration of synchronous belief propagation at which *every message* is less than ϵ away from its value at convergence. Therefore, by stopping synchronous belief propagation after τ iterations we are computing a τ_ϵ approximation for some ϵ. In Sec. 5.2 we relate the conventional stopping condition from Eq. (3.6) to approximation accuracy ϵ.

The concept of message decay in graphical models is not strictly novel. In the setting of online inference in Hidden Markov Models Russell and Norvig (1995) used the notion of Fixed Lag Smoothing to eliminate the need to pass messages along the entire chain. Meanwhile, work by Ihler et al. (2005) has investigated message error decay along chains as it relates to robust inference in distributed settings, numerical precision, and general convergence guarantees for belief propagation

4. It is possible to extend Definition 5.1 to arbitrary cyclic graphs by considering the possibly unbounded computation tree described by Weiss (2000).

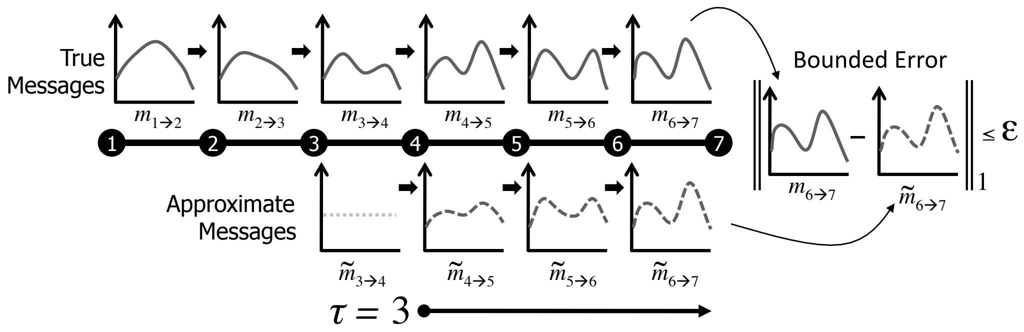


Figure 4: Above the chain graphical model we plot the sequence of messages from vertex 1 to vertex 7. For visual clarity these messages are plotted as continuous distributions. Below the chain we plot the approximate sequence of messages starting at vertex 4 assuming a uniform message is received from vertex 3. We see that the message that finally arrives at vertex 7 is only ϵ away from the original message. Therefore for an ϵ level approximation at vertex 7 messages must only be sent a distance of $\tau = 3$.

in cyclic models. Our use of τ_ϵ is to theoretically characterize the additional parallelism exposed by an ϵ level approximation and to ultimate guide the design of the new optimal parallel belief propagation scheduling.

5.1 Empirical Analysis of τ_ϵ

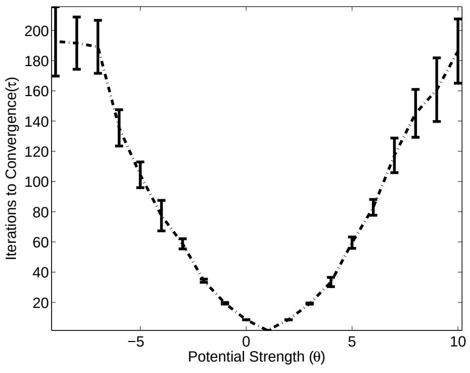
The τ_ϵ structure of a graphical model is a measure of both the underlying chain structure as well as the *strength* of the *factors* along those structures. The strength of a factor refers to its coupling affect on the variables in its domain. A factor that assigns relatively similar energy to all configurations is much weaker than a factor that assigns disproportionately greater energy to a subset of its assignments. Graphical models with factors that tightly couple variables along chains will require a greater τ to achieve the same ϵ level approximation.

To illustrate the dependence τ_ϵ on the coupling strength of the factors we constructed a sequence of binary chain graphical models each with 1000 variables but with varying degrees of attractive and repulsive edge factors. We parameterized the edge factors by

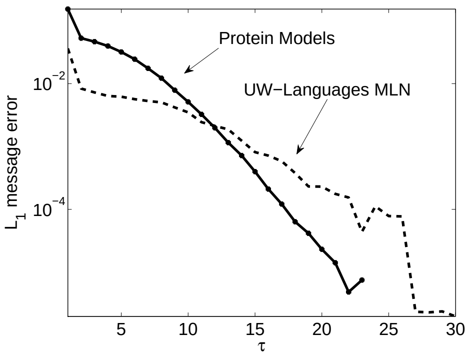
$$\psi_{x_i, x_{i+1}} = \begin{cases} e^{\theta} & x_i = x_{i+1} \\ e^{1} & \text{otherwise} \end{cases}$$

where θ is the "strength" parameter. When $\theta = 1$, every variable is independent and we expect $\tau_\epsilon = 1$ resulting in the shortest runtime. When $\theta < 1$ (corresponding to anti-ferromagnetic potentials) alternating assignments become more favorable. Conversely when $\theta > 1$ (corresponding to ferromagnetic potentials) matching assignments become more favorable. We constructed 16 chains at each potential with vertex potentials ψ_{x_i} chosen randomly from Unif$(0, 1)$. We ran synchronous belief propagation until a fixed $\epsilon = 10^{-5}$ level approximation was achieved and plotted the number of iterations versus the potential strength in Fig. 5(a). We observe that even with relatively strong potentials, the number of iterations, τ_ϵ, is still relatively small compared to the length of the chain.

We can further experimentally characterize the decay of message errors as a function of propagation distance. On several real networks we ran belief propagation to convergence (at $\beta = 10^{-10}$)



(a) τ_ϵ versus θ



(b) τ_ϵ versus θ

Figure 5: In **(a)** the number of iterations synchronous BP required to obtain a $\epsilon = 10^{-5}$ approximation on a 1000 variable chain graphical model is plotted against the strength of the pairwise potential θ. In **(b)** the log average message error is plotted as a function of the walk length on the protein pair-wise markov random fields and the UW-Languages MLN.

and then simulated the effect of replacing an arbitrary message with a uniform distribution and then propagating that *error* along random paths in the graph. In Fig. 5(b) we plot the message error as a function of distance averaged over 1000 random walks on various different factor graphs. Walks were terminated after the error fell below the original $\beta = 10^{-10}$ termination condition. In all cases we see that the affect of the original message error decays rapidly.

5.2 Bounding τ_ϵ Under the Contraction Mapping Assumption

We can represent a single iteration of synchronous belief propagation by a function f_{BP} which maps all the messages $m^{(t)}$ on the t^{th} iteration to all the messages $m^{(t+1)} = f_{\text{BP}}(m^{(t)})$ on the $(t+1)^{\text{th}}$ iteration. The fixed point is then the set of messages $m^* = f_{\text{BP}}(m^*)$ that are invariant under f_{BP}. In addition, we define a max-norm for the message space

$$\left|\left|m^{(t)} - m^{(t+1)}\right|\right|_\infty = \max_{(i,j)\in E} \left|\left|m^{(t)}_{i\to j} - m^{(t+1)}_{i\to j}\right|\right|_1, \tag{5.2}$$

which matches the standard termination condition.

A common method for analyzing fixed point iterations is to show (assume) that the f_{BP} is a contraction mapping and then use the contraction rate to bound the number of iterations for an ϵ level approximation of m^*. If f_{BP} is a max-norm contraction mapping then for the fixed point m^* and $0 \le \alpha < 1$,

$$||f_{\text{BP}}(m) - m^*||_\infty \le \alpha \, ||m - m^*||_\infty \, .$$

Work by Mooij and Kappen (2007) provide sufficient conditions for $f_{\text{BP}}(m)$ to be a contraction mapping under a variety of norms including the max-norm and the L_1-norm and shows that BP on acyclic graphs is guaranteed to be a contraction mapping under a particular spectral norm.

If the contraction rate α is known, and we desire an ϵ approximation of the fixed point, τ_ϵ is the smallest value such that $\alpha^{\tau_\epsilon} ||m_0 - m^*||_\infty \leq \epsilon$. This is satisfied by setting

$$\tau_\epsilon \leq \left\lceil \frac{\log(2/\epsilon)}{\log(1/\alpha)} \right\rceil . \tag{5.3}$$

Finally, in Eq. (5.4) we observe that the convergence criterion, $||m - f(m)||_\infty$ defined in Eq. (3.6), is a constant factor upper bound on the distance between m and the fixed point m^*. If we desire an ϵ approximation, it is sufficient to set the convergence criterion $\beta \leq \epsilon(1-\alpha)$.

$$\begin{aligned}
||m - m^*||_\infty &= ||m - f_{\mathrm{BP}}(m) + f_{\mathrm{BP}}(m) - m^*||_\infty \\
&\leq ||m - f_{\mathrm{BP}}(m)||_\infty + ||f_{\mathrm{BP}}(m) - m^*||_\infty \\
&\leq ||m - f_{\mathrm{BP}}(m)||_\infty + \alpha\, ||m - m^*||_\infty \\
||m - m^*||_\infty &\leq \frac{1}{1-\alpha} ||m - f_{\mathrm{BP}}(m)||_\infty \qquad (5.4)
\end{aligned}$$

In practice, the contraction rate α is likely to be unknown and f_{BP} will not be a contraction mapping on all graphical models. Furthermore, it may be difficult to determine τ_ϵ without first running the inference algorithm. Ultimately, our results *only* rely on τ_ϵ as a theoretical tool for comparing inference algorithms and understanding parallel convergence behavior. In practice, rather than constructing a β for a particular α and ϵ, we advocate the more pragmatic method of picking the smallest possible value that resources permit.

5.3 Constructing a τ_ϵ-Approximation with Parallel Belief Propagation

In practice we are interested in obtaining an τ_ϵ approximation for all vertices. We can now reanalyze our runtime bound in Sec. 4 under the τ_ϵ-Approximation setting. Suppose we desire an ϵ level approximation for *all* messages in an acyclic graphical model. As discussed earlier, a τ_ϵ-approximation can be achieved in τ_ϵ iterations of synchronous belief propagation leading to the runtime bound for parallel synchronous belief on acyclic graphical models given in Prop. 5.1. In particular τ_ϵ replaces the role of the maximum path length yielding an improved running time over the bound given in Theorem 4.1.

Proposition 5.1 (Parallel Synchronous BP Running Time for a τ_ϵ-Approximation) *Given an acyclic graphical model with n vertices a τ_ϵ-approximation is obtained by running Synchronous BP with p processors ($p \leq n$) in running time*

$$\Theta\left(\frac{n\tau_\epsilon}{p} + \tau_\epsilon\right).$$

However, even with the reduced runtime afforded by the τ_ϵ-approximation, we can show that on a simple chain graphical model, the performance of Synchronous BP is still far from optimal when computing a τ_ϵ approximation. Here we derive a lower bound for the running time of τ_ϵ approximate belief propagation on a chain graphical model.

Theorem 5.2 *For an arbitrary chain graph with n vertices and p processors, the running time of a τ_ϵ-approximation is lower bounded by:*

$$\Omega\left(\frac{n}{p} + \tau_\epsilon\right).$$

Proof The messages sent in opposite directions are independent and the amount of work in each direction is symmetric. Therefore, we can reduce the problem to computing a τ_ϵ-approximation in one direction (X_1 to X_n) using $p/2$ processors. Furthermore, to achieve a τ_ϵ-approximation, we need exactly $n - \tau_\epsilon$ vertices from $\{X_{\tau_\epsilon+1}, \ldots, X_n\}$ to be τ_ϵ left-aware (i.e., for all $i > \tau_\epsilon$, X_i is aware of $X_{i-\tau_\epsilon}$). By definition when $n - \tau_\epsilon$ vertices first become τ_ϵ left-aware the remaining (first) τ_ϵ vertices are maximally left-aware.

Let each processor compute a set of $k \geq \tau_\epsilon$ consecutive message updates in sequence (e.g., $[m^{(1)}_{1\to2}, m^{(2)}_{2\to3}, \ldots, m^{(k)}_{k-1\to k}]$). Notice that it is never beneficial to skip a message or compute messages out of order on a single processor since doing so cannot increase the number of vertices made left-aware. After the first τ_ϵ updates each additional message computation make exactly one more vertex left-aware. Therefore after k message computations each processor can make at most $k - \tau_\epsilon + 1$ vertices left-aware. Requiring all $p/2$ processors to act simultaneously, we observe that pre-emption will only decrease the number of vertices made τ_ϵ left-aware.

We then want to find a lower bound on k such that the number of vertices made left-aware after k iterations greater than the minimum number of vertices that must be made left-aware. Hence we obtain the following inequality:

$$\begin{aligned} n - \tau_\epsilon &\leq \frac{p}{2}(k - \tau_\epsilon + 1) \\ k &\geq \frac{2n}{p} + \tau_\epsilon\left(1 - \frac{2}{p}\right) - 1 \end{aligned} \qquad (5.5)$$

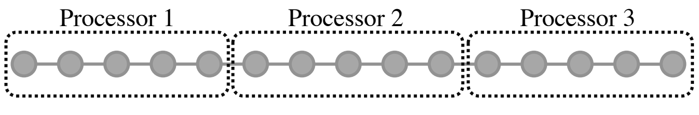
relating required amount of work and the maximum amount of work done on the k^{th} iteration. For $p > 2$, Eq. (5.5) provides the desired asymptotic result. ■

We observe from Prop. 5.1, that Synchronous BP has a runtime multiplicative in τ_ϵ and n whereas the optimal bounds are only additive. To illustrate the size of this gap, consider a chain of length n, with $\tau_\epsilon = \sqrt{n}$. The Synchronous parallel belief propagation running time is $O\left(n^{3/2}/p\right)$, while the optimal running time is $O\left(n/p + \sqrt{n}\right) = O\left(n/p\right)$. In the next section we present an algorithm that achieves this lower bound on chains and also generalizes to arbitrary cyclic graphical models.

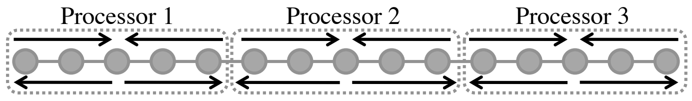
6. Optimal Parallel Scheduling on Chains

In this section we exploit the local τ_ϵ structure by composing locally optimal forward-backward sequential schedules to develop the ChainSplash Parallel Belief Propagation algorithm and achieve the optimal running time for a τ_ϵ approximation on a chain graphical model. In the subsequent sections we generalize the ChainSplash algorithm to arbitrary cyclic graphical models to obtain our new Parallel Splash Belief Propagation algorithm.

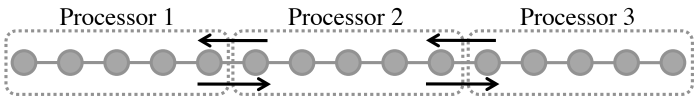
The ChainSplash algorithm (Alg. 4 begins by dividing the chain evenly among the p processors as shown in Fig. 6(a). Then, in parallel, each processor runs the sequential forward-backward algorithm on its sub-chain as shown in Fig. 6(b). Each processor exchanges messages along the boundaries and then repeats the procedure. In Theorem 6.1 we show that the ChainSplash algorithm achieves the optimal lower bound for a τ_ϵ-approximation in a chain graphical model. Notice, the algorithm does not require knowledge of τ_ϵ and instead relies on the convergence criterion to ensure an accurate approximation.



(a) Chain Partitioning



(b) Parallel Forward-Backwards



(c) Message Exchange

Figure 6: An illustration of the steps in the ChainSplash algorithm. **(a)** The chain is partitioned evenly among the processors. **(b)** In parallel each processor runs the forward-backward sequential scheduling. The forward sweep updates messages towards the center vertex and the backwards sweep updates messages towards the boundary. This slightly abnormal forward-backward sweep will be more natural when we generalize to arbitrary cyclic graphs in the Parallel Splash Algorithm. **(c)** Messages that cross the processor boundaries are exchanged and steps **(b)** and **(c)** are repeated until convergence.

Algorithm 4: ChainSplash Belief Propagation Algorithm

```
Input : Chain graphical model $(V, E)$ with $n$ vertices
// Partition the chain over the $p$ processors
forall $i \in \{1, \ldots, p\}$ do in parallel
    $B_i \leftarrow \{x_{\lceil (i-1)n/p \rceil}, \ldots, x_{\lceil in/p \rceil - 1}\}$
while Not Converged do
    // Run Forward-Backward Belief Propagation on each Block
    forall $i \in \{1, \ldots, p\}$ do in parallel
        Run Sequential Forward-Backward on $B_i$
    // Exchange Messages
    forall $i \in \{1, \ldots, p\}$ do in parallel
        Exchange messages with $B_{i-1}$ and $B_{i+1}$
```

Theorem 6.1 (ChainSplash Optimality) *Given a chain graph with n vertices and $p \leq n$ processors, the ChainSplash belief propagation algorithm, achieves a τ_ϵ level approximation for all vertices in time*

$$O\left(\frac{n}{p} + \tau_\epsilon\right)$$

Proof As with the lower bound proof we will consider messages in only one direction from x_i to x_{i+1}. After each local execution of the forward-backward algorithm all vertices become left-aware of all vertices within each processor block. After messages are exchanged the left most vertex becomes left-aware of the right most vertex on the preceding processor. By transitivity of left-awareness, after k iterations, all vertices become left-aware of $(k-1)n/p$ vertices. We require all vertices to be made τ_ϵ left-aware and so solving for k-iterations we obtain:

$$\begin{aligned}\tau_\epsilon &= (k-1)\frac{n}{p} \\ k &= \frac{\tau_\epsilon p}{n} + 1\end{aligned}$$

Each iteration is executed in parallel in time n/p so the total running time is then:

$$\begin{aligned}\frac{n}{p}k &= \frac{n}{p}\left(\frac{\tau_\epsilon p}{n} + 1\right) \\ &= \frac{n}{p} + \tau_\epsilon\end{aligned}$$

■

7. The Splash Belief Propagation Algorithm

Our new Splash belief propagation algorithm generalizes the ChainSplash algorithm to arbitrary cyclic graphical models. The Splash algorithm is composed of two core components, the Splash routine which generalizes the forward-backward scheduling to arbitrary cyclic graphical models, and a dynamic Splash scheduling which ultimately determines the shape, size, and location of Splashes. We will first present the Parallel Splash algorithm as a sequential algorithm, introducing the Splash operation, belief scheduling, and how they are combined to produce a simple single processor Splash algorithm. Then in subsequent sections we describe the parallel structure of this algorithm and how it can efficiently be mapped to shared and distributed memory systems.

7.1 The Splash Operation

Our new algorithm is built around the Splash operation (Alg. 5 and Fig. 7) which generalizes the forward backward scheduling illustrated in Fig. 3(a) from chains to arbitrary cyclic graphical models. The Splash operation constructs a small tree, which we call a Splash, and then executes a local forward-backward message scheduling on that tree, sequentially updating all messages originating from vertices within the tree. By scheduling message calculations along a local tree, we ensure that all message calculations increase awareness with respect to the local tree and that the algorithm is locally optimal on tree structures.

The inputs to the Splash operation are the root vertex v and the size of the Splash, W, which is the parameter bounding the overall work associated with executing the Splash. A Splash begins by constructing a local spanning tree rooted at v, adding vertices in breadth first search order such that the total amount of work in the Splash does not exceed the limit set by W. We define the work associated with each vertex u (which could be a variable or factor) as:

$$w_u = |\Gamma_u| \times A_u + \sum_{v \in \Gamma_u} A_v, \tag{7.1}$$

where $|\Gamma_u| \times A_u$ represents the work required to recompute all outbound messages and $\sum_{v \in \Gamma_u} A_v$ is the work required to update the beliefs of all the neighboring vertices. We update the beliefs

Algorithm 5: Splash(v, W)

```
   Input  : vertex $v$
   Input  : maximum splash size $W$
   // Construct the breadth first search ordering with $W$ message
      computations and rooted at $v$.
   fifo ← [] // FiFo Spanning Tree Queue
   $\sigma \leftarrow (v)$ // Initial Splash ordering is the root $v$
   AccumW ← $\sum_{w \in \Gamma_v} \mathcal{A}_w + |\Gamma_v| * \mathcal{A}_v$ // Total work in the Splash
   visited ← $\{v\}$ // Set of visited vertices
   fifo.Enqueue($\Gamma_v$)
   while fifo is not empty do
       $u$ ← fifo.Dequeue()
       UWork ← $\sum_{w \in \Gamma_u} \mathcal{A}_w + |\Gamma_u| * \mathcal{A}_u$
       // If adding $u$ does not cause me to exceed the work limit
       if AccumW + UWork ≤ $W$ then
           AccumW ← AccumW + UWork
           Add $u$ to the end of the ordering $\sigma$
           foreach neighbors $w \in \Gamma_u$ do
               if $w$ is not in visited then
                   fifo.Enqueue($w$) // Add to boundary of spanning tree
                   visited ← visited ∪ $\{w\}$ // Mark Visited
   // Make Root Aware of Leaves
 1 foreach $i \in$ ReverseOrder($\sigma$) do
       SendMessages($i$)
   // Make Leaves Aware of Root
 2 foreach $i \in \sigma$ do
       SendMessages($i$)
```

of neighboring vertices as part of the scheduling described in Sec. 7.2. Recall that when u is a factor vertex A_u represents the size of the factor (i.e., the size of the domain which is exponential in the degree). The work w_u defined in Eq. (7.1) is proportional to the running time of invoking `SendMessages(u)`. In Tab. 1 we compute the work associated with each vertex shown in Fig. 7(a).

The local spanning tree rooted at vertex F in Fig. 7 is depicted by the shaded rectangle which grows outwards in the sequence of figures Fig. 7(b) through Fig. 7(e). The maximum splash size is set to $W = 170$. In Fig. 7(b) the Splash contains only vertex F (the root) and has total accumulated work of $w = 30$. The vertices A, E, and F are added in Fig. 7(c) expanding the breadth first search without exceeding $W = 170$. The effect of $W = 170$ is seen in Fig. 7(d) vertices B and K are added but vertex G is *skipped* because including G would cause the Splash to exceed $W = 170$. The maximum splash size is achieve in Fig. 7(e) when vertex C is added and no other vertices may be added without exceeding $W = 170$. The final splash ordering is $\sigma = [F, E, A, J, B, K, C]$.

Using the *reverse* of the breadth first search ordering constructed in the first phase, the `SendMessages` operation is sequentially invoked on each vertex in the spanning tree (Line 1) starting at the leaves, generalizing the forward sweep. The function `SendMessages(`v`)` updates all outbound messages from vertex v using the most recent inbound messages. In Fig. 7(f), `SendMessages(`C`)` is invoked on vertex C causing the messages $m_{C\to B}$, $m_{C\to G}$, and $m_{C\to D}$ to be recomputed. Finally, messages are computed in the original σ ordering starting at the root and invoking `SendMessages` sequentially on each vertex, completing the backwards sweep. Hence, with the exception of the root vertex v, all messages originating from each vertex in σ are computed twice, once on the forwards sweep in Line 1 and once in the backwards sweep in Line 2.

Vertex	Assignments (A_i)	Degree $(\lvert\Gamma_i\rvert)$	Neighbor Costs $\left(\sum_{w\in\Gamma_i}\lvert\Gamma_w\rvert\right)$	Work (w_i)
A	2	3	$2^3+2^4+2^2$	34
B	2^2	2	$2+2$	12
C	2	3	$2^2+2^4+2^2$	30
D	2^2	2	$2+2$	12
E	2	1	2^3	10
F	2^3	3	$2+2+2$	30
G	2^4	4	$2+2+2+2$	72
H	2	2	2^2+2^2	12

Table 1: The work associated with each vertex in Fig. 7(a) is computed using Eq. (7.1). We have omitted vertices I, J, K, and L sinc they are equivalent to vertices D, A, B, and C respectively.

We can recover the ChainSplash algorithm by repeatedly executing p parallel splashes of size $W = wn/p$ (where w is the work of updating a single vertex) placed evenly along the chain. We therefore achieve the runtime lower bound for τ_ϵ approximation by using the Splash operation. The remaining challenge is determine how to place splashes in arbitrary cyclic graphical models. In the next section, we will introduce our new belief residual scheduling, the second core element of the parallel Splash algorithm.

7.2 Belief Residual Scheduling

To schedule Splash operations, we extend the residual heuristic introduced by Elidan et al. (2006) which prioritizes message updates based on their residual, greedily trying to achieve the convergence criterion given in Eq. (3.6). The residual of a message is defined as the difference between its current value and its value when it was last used (i.e., $\left|\left|m^{\text{next}}_{i\to u} - m^{\text{last}}_{i\to u}\right|\right|_1$). The residual heuristic greedily receives the message with highest residual first and then updates the dependent message residuals by temporarily computing their new values. Elidan et al. (2006) proposed the residual heuristic as a greedy method to minimize a global contraction. Alternatively, one can interpret the heuristic as a method to greedily achieve the termination condition by removing the message that is currently failing the convergence test. In addition the residual heuristic ensures that messages which have "converged" are not repeatedly updated.

The original presentation of the residual heuristic in Elidan et al. (2006) focuses on scheduling the reception of individual messages. However, in the process of receiving a new message, all outbound messages from vertex must be recomputed. The message updates in Elidan et al. (2006) recompute temporary versions of all outbound messages using only current version of the highest

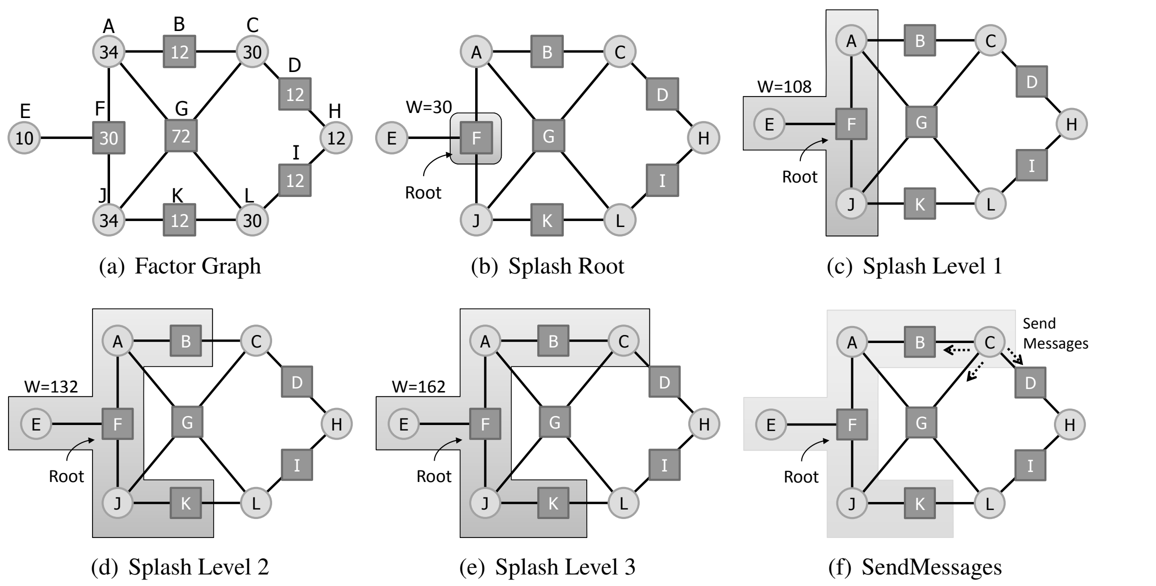


Figure 7: A splash of splash size $W = 170$ is grown starting at vertex F. The Splash spanning tree is represented by the shaded region. **(a)** The initial factor graph is labeled with the vertex work (w_i) associated with each vertex. **(b)** The Splash begins rooted at vertex F with accumulated work $w = 30$. **(c)** The neighbors of F are added to the Splash and the accumulated work increases to $w = 108$. **(d)** The Splash expands further to include vertex B and K but does not include vertex G because doing so would exceed the maximum splash size of $W = 170$. **(e)** The splash expand once more to include vertex C but can no longer expand without exceeding the maximum splash size. The final Splash ordering is $\sigma = [F, E, A, J, B, K, C]$. **(f)** The SendMessages operation is invoked on vertex C causing the messages $m_{C\to B}$, $m_{C\to G}$, and $m_{C\to D}$ to be recomputed.

residual message and older versions of all the remaining messages. Consequently, a single high degree vertex could be forced to recompute all outbound messages repeatedly until it has received each inbound message.

In **?** we defined a scheduling over vertices and not messages. The priority (residual) of a vertex is the maximum of the residuals of the incoming messages.

$$r_u = \max_{i\in\Gamma_u} \left|\left|m_{i\to u}^{\text{next}} - m_{i\to u}^{\text{last}}\right|\right|_1 \tag{7.2}$$

Intuitively, vertex residuals capture the amount of new information available to a vertex. Recomputing outbound messages from a vertex with unchanged inbound messages results in a wasted update. Once the `SendMessages` operation is applied to a vertex, its residual is set to zero and its neighbors residuals are updated. Vertex scheduling has the advantage over message residuals of using all of the most recent information when updating a message.

Additionally, there are two key flaws with using message residuals for termination and scheduling. First, using message residuals as the termination criterion may lead to nonuniform convergence in beliefs. Second, high degree vertices may often have at least one new message with high residual however the resulting contribution to the belief and outbound messages may be minimal leading to over scheduling of high degree vertices.

For a vertex of degree d, ϵ changes to individual messages can compound, resulting in up to $d\epsilon$ change in beliefs. In Appendix A.1 we consider the case of a binary random variable in which all d inbound messages change from uniform $[\frac{1}{2}, \frac{1}{2}]$ to $[\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon]$. We show that while the maximum L_1 message residual is bounded by 2ϵ the L_1 change in the corresponding belief grows linearly with d. Consequently, high degree vertices may have poorly estimated beliefs even when the overall message residual is small.

Conversely, a large $1 - \epsilon$ residual in a single message may result in a small ϵ change in belief at high degree vertices. In Appendix A.2 we consider a binary random variable with d inbound messages in which all messages are initial $[1 - \epsilon, \epsilon]$. We show that by making a large change from $[1 - \epsilon, \epsilon]$ to $[\epsilon, 1 - \epsilon]$ in a single message we can actually obtain a change in belief bounded by $o\epsilon/2^{d-3}$. Consequently, high degree vertices are likely to be over-scheduled when using a message based residual scheduling. The resulting loss in performance is compounded by the large cost of updating a high degree vertex.

7.2.1 BELIEF RESIDUAL SCHEDULING

The goal of belief propagation is to estimate the belief for each variable. However, message scheduling and convergence tests use the change in messages rather than beliefs. We showed that small message residuals do not imply small changes in beliefs and conversely large message residuals do not imply large changes in belief. Here we define a belief residual which addresses the problems associated with the message-centric approach and enable improved scheduling and termination assessment.

A natural definition of the belief residuals analogous to the message residuals defined in Eq. (7.2) is

$$r_j = \left|\left|b_i^{\text{new}} - b_i^{\text{old}}\right|\right|_1 \tag{7.3}$$

where b_i^{old} is the belief at vertex i the last time vertex i was updated. Unfortunately, Eq. (7.3) has a surprising flaw that admits premature convergence on acyclic graphical models even when the termination condition

$$\max_{i \in V} \left|\left|b_i^{\text{new}} - b_i^{\text{old}}\right|\right|_1 \leq \beta \tag{7.4}$$

is set to $\beta = 0$. In Appendix A.3 we construct a simple ordering on a chain graphical model which will terminate prematurely using the termination condition given in Eq. (7.4) even with $\beta = 0$.

An alternative formulation of the belief residual which does not suffer from premature convergence and offers additional computational advantages is given by:

$$r_i^{(t)} \leftarrow r_i^{(t-1)} + \left|\left|b_i^{(t)} - b_i^{(t-1)}\right|\right|_1 \tag{7.5}$$

$$b_i^{(t)}(x_i) \propto \frac{b_i^{(t-1)}(x_i) m_{i \to j}^{(t)}(x_i)}{m_{i \to j}^{(t-1)}(x_i)}. \tag{7.6}$$

$b_i^{(t-1)}$ is the belief after incorporating the previous message and $b_i^{(t)}$ is the belief after incorporating the new message. Intuitively, the belief residual defined in Eq. (7.5) measures the cumulative effect of all message updates on the belief. As each new message arrives, the belief can be efficiently recomputed using Eq. (7.6). Since with each message change we can quickly update the local belief using Eq. (7.6) and corresponding belief residual (Eq. (7.5)), we do not need to retain

Algorithm 6: The Sequential Splash Algorithm

```
Input   : Constants $W, \beta$
$Q \leftarrow$ InitializeQueue($Q$)
Set All Residuals to $\infty$
while TopResidual(Q) $> \beta$ do
    $v \leftarrow$ Top($Q$)
    Splash($Q$, $v$, $W$)
```

the old messages and beliefs reducing the storage requirements and the overhead associated with scheduling.

The belief residual may also be used as the convergence condition

$$\max_{i \in V} r_i \leq \beta \tag{7.7}$$

by terminating when all vertices have a belief residual below a fixed threshold $\beta >= 0$. Since Eq. (7.5) satisfies the triangle inequality, it is an upper bound on the total change in belief ensuring that the algorithm does not change while their are beliefs that have changed by more than β. Because Eq. (7.5) accumulates the change in belief with each new message, it will not lead to premature termination scenario encountered in the more naive belief residual definition (Eq. (7.3)).

When used as a scheduling priority, the belief residual prevents starvation. Since the belief residual of a vertex can only increase as its neighbors are repeatedly updated, all vertices with belief residuals above the termination threshold are eventually updated. In addition, Belief Residuals address over-scheduling in high degree vertices since a large change in a single inbound message that does not contribute to a significant change in the belief will not significantly change the belief residual.

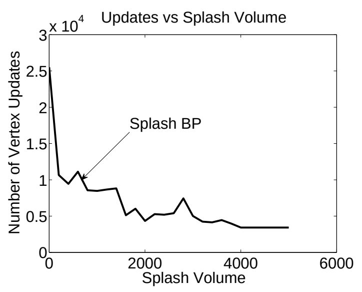
7.3 Sequential Splash Algorithm

By combining the Splash operation with the residual scheduling heuristic we obtain the Sequential Splash algorithm given in Alg. 6. The sequential Splash algorithm maintains a shared belief residual priority queue over vertices. The queue is initialized in random order with the priorities of all vertices set to infinity[5]. This ensures that every vertex is updated at least once.

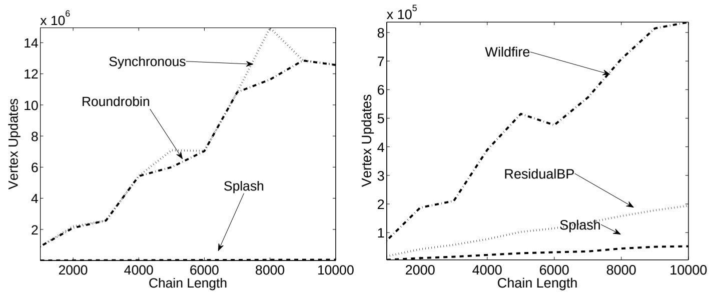
The Splash operation is applied to the vertex at the top of the queue. During the Splash operation the priorities of vertices that receive new messages are increased using the belief residual definition. Immediately after updating a vertex its belief residual is reset to zero. Due to the additional costs associated with maintaining the belief residuals, the work associated with each vertex includes not only the cost of recomputing all outbound messages but also the cost of updating the belief residuals (recomputing the beliefs) of its neighbors.

To demonstrate the performance gains of the Sequential Splash algorithm on strongly sequential models we constructed a set of synthetic chain graphical models and evaluated the running time on these models for a fixed convergence criterion while scaling the size of the splash in Fig. 8(a) and while scaling the size of the chain in Fig. 8(b). As the size of the Splash expands (Fig. 8(a)) the

5. The IEEE Inf value is sufficient.



(a) Runtime versus Splash Size



(b) Runtime versus Chain Size

Figure 8: In this figure, we plot the number of vertex updates needed to run a randomly generated chain graphical model to convergence. Run-time to convergence is measured in vertex updates rather than wall clock time to ensure a fair *algorithmic* comparison and eliminate hardware and implementation effects which appear at the extremely short run-times encountered on these simple models. **(a)** The number of vertex updates made by Sequential Splash BP, fixing the chain length to 1000, and varying the splash size. **(b)** The number of vertex updates made by various BP algorithms varying the length of the chain. Two plots are used to improve readability since the Splash algorithm is an order of magnitude faster than the Synchronous and Round-Robin algorithms. Note the difference in the scale of the Y axis. The Splash algorithm curve is the same for both plots.

total number of updates on the chain decreases reflecting the optimality of the underlying forward-backward structure of the Splash. In Fig. 8(b) we compare the running time of Splash with Synchronous, Round-Robin, Residual, and Wild-fire belief propagation as we increase the size of the model. The conventional Synchronous and Round-Robin algorithms are an order of magnitude slower than Wild-fire, ResidualBP, and Splash and scale poorly forcing separate comparisons for readability. Nonetheless, in all cases the Splash algorithm (with splash size $W = 500$) is substantially faster and demonstrates better scaling with increasing model size.

7.4 The Parallel Splash Algorithm

We construct the Parallel Splash belief propagation algorithm from the Sequential Splash algorithm by executing multiple Splashes in parallel. The abstract Parallel Splash algorithm is given in Alg. 7. Notice that the Parallel Splash algorithm only differs from the sequential Splash algorithm in Line 1 in which p processors are set to run the sequential Splash algorithm all drawing from the same shared queue.

While we do not require that parallel Splashes contain disjoint sets of vertices, we do require that each Splash has a unique root which is achieved through the shared scheduling queue and atomic `Push` and `Pop` operations. To prevent redundant message update when Splashes overlap, if multiple processors simultaneously call `SendMessages(`i`)` all but one return immediately ensuring a single update.

To achieve maximum parallel performance the Parallel Splash algorithm relies on an efficient parallel scheduling queue to minimize processor locking and sequentialization when `Push`, `Pop`, and `UpdatePriority` are invoked. While there is considerable work from the parallel computing community on efficient parallel queue data structures, we find that in the shared memory setting basic locking queues provide sufficient performance. In the distributed Splash algorithm discussed in

Algorithm 7: Parallel Splash Belief Propagation Algorithm

```
  Input : Constants $W, \beta$
  $Q \leftarrow$ InitializeQueue($Q$)
  Set All Residuals to $\infty$
1 forall processors do in parallel
      while TopResidual(Q) $> \beta$ do
          $v \leftarrow$ Pop($Q$) // Atomic
          Splash($Q$, $v$, $W$)
          $Q$.Push(($v$, Residual($v$))) // Atomic
```

Sec. 9 we employ a distributed scheduling queue in a work balanced manner to eliminate processor contention.

7.5 Chain Optimality of Parallel Splash

We now show that, in expectation, the Splash algorithm achieves the optimal running time from Theorem 5.2 for chain graphical models. We begin by relating the Splash operation to the vertex residuals.

Lemma 7.1 (Splash Residuals) *Immediately after the Splash operation is applied to an acyclic graph all vertices interior to the Splash have zero belief residual.*

Proof The proof follows naturally from the convergence of BP on acyclic graphs. The Splash operation runs BP to convergence on the subgraph contained within the Splash. As a consequence all messages along edges in the subgraph will (at least temporarily) have zero residual. ■

After a Splash is completed, the residuals associated with vertices interior to the Splash are propagated to the exterior vertices along the boundary of the Splash. Repeated application of the Splash operation will continue to move the boundary residual leading to Lemma 7.2.

Lemma 7.2 (Basic Convergence) *Given a chain graph where only one vertex has nonzero residual, the Splash algorithm with Splash size W will run in $O(\tau_\epsilon + W)$ time.*

Proof When the first Splash, originating from the vertex with nonzero residual is finished, the interior of the Splash will have zero residual as stated in 7.1, and only the boundary of the Splash will have non-zero residual. Because all other vertices initially had zero residual and messages in opposite directions do not interact, each subsequent Splash will originate from the boundary of the region already covered by the previous Splash operations. By definition the convergence criterion is achieved after the high residual messages at the originating vertex propagate a distance τ_ϵ. However, because the Splash size is fixed, the Splash operation may propagate messages an additional W vertices. ■

If we set the initial residuals to ensure that the first p parallel Splashes are uniformly spaced, Splash obtains the optimal lower bound.

Theorem 7.3 (Splash Chain Optimality) *Given a chain graph with n vertices and $p \leq n$ processors we can apply the Splash algorithm with the Splash size set to $W = n/p$ and uniformly spaced initial Splashes to obtain a τ_ϵ-approximation in expected running time*

$$O\left(\frac{n}{p} + \tau_\epsilon\right)$$

Proof We set every n/p vertex $\left\{X_{n/2p}, X_{3n/2p}, X_{5n/2p}, \ldots\right\}$ to have slightly higher residual than all other vertices forcing the first p Splash operations to start on these vertices. Since the height of each splash is also $W = n/p$, all vertices will be visited in the first p splashes. Specifically, we note that at each Splash only produces 2 vertices of non-zero residual (see Lemma 7.1). Therefore there are at most $O(p)$ vertices of non-zero residual left after the first p Splashes.

To obtain an upper bound, we consider the runtime obtained if we compute independently, each τ_ϵ subtree rooted at a vertex of non-zero residual. This is an upper bound as we observe that if a single Splash overlaps more than one vertex of non-zero residual, progress is made simultaneously on more than one subtree and the running time can only be decreased.

From Lemma 7.1, we see that the total number of updates needed including the initial $O(p)$ Splash operations is $O(p(\tau_\epsilon + W)) + O(n) = O(n + p\tau_\epsilon)$. Since work is evenly distributed, each processor performs $O(n/p + \tau_\epsilon)$ updates. ■

In practice, when the graph structure is not a simple chain graph, it may be difficult to evenly space Splash operations. By randomly placing the initial Splash operations we can obtain a factor $\log(p)$ approximation in expectation.

Corollary 7.4 (Splash with Random Initialization) *If all residuals are initialized to a random value greater than the maximum residual, the total expected running time is at most* $O(\log(p)(n/p + \tau_\epsilon))$.

Proof Partition the chain graph into p blocks of size n/p. If a Splash originates in a block then it will update all vertices interior to the block. The expected time to Splash (collect) all p blocks is upper bounded[6] by the coupon collectors problem. Therefore, at most $O(p\log(p))$ Splash operations (rather than the p Splash operations used in Theorem 9.1) are required in expectation to update each vertex at least once. Using the same method as in Theorem 9.1, we observe that the running time is $O(\log(p)(n/p + \tau_\epsilon))$. ■

7.6 Dynamic Splashes with Belief Residuals

A weakness of the Splash belief propagation algorithm is that it requires tuning of the Splash size parameter which affects the overall performance. If the Splash size is too large than the algorithm will be forced to recompute messages that have already converged. Alternatively, if the Splash size is set too small then we lose optimality on local sequential structures. To address this weakness in the Splash algorithm, we introduce Dynamic Splashes which substantially improve performance in practice and eliminate the need to tune the Splash size parameter.

6. Since candidate vertices are removed between parallel rounds, there should be much fewer collection steps than the analogous coupon collectors problem.

The key idea is that we can use the belief residuals to automatically adapt the size and shape of the Splash procedure as the algorithm proceeds. In particular we modify the initial breadth first search phase of the Splash operation, to exclude vertices with belief residuals below the termination condition. This ensures that we do not recompute messages that have already "converged." and more importantly allows the Splash operation to *adapt* to the local convergence patterns in the factor graph. As the algorithm approaches convergence, removing low belief residual vertices rapidly disconnects the graph forcing the breadth first search to terminate early and causing the size of each Splash to shrink reducing wasted updates. As a consequence, the algorithm is less sensitive to the Splash size W. Instead we can fix Splash size to be a relatively large fraction of the graph (i.e., n/p) and let pruning automatically decrease the Splash size as needed. The Splash procedure with the "pruning" modification is called the DynamicSplash procedure. A complete description of DynamicSplash is provided in Alg. 8.

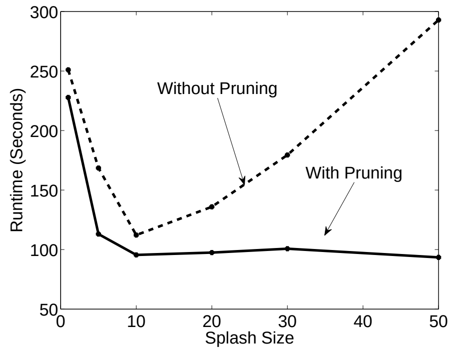
We plot in Fig. 9(a) the running time of the Parallel Splash algorithm with different Splash sizes W and with and without Splash pruning. For relatively small Splash sizes, Splash pruning has little effect but the overall performance of the algorithm is decreased. With Splash pruning disabled there is clear optimal Splash size where as with Splash pruning enabled increasing the size of the Splash beyond the optimal does not reduce the performance. We have also plotted in Fig. 9(c) examples of the Splashes at various phases of the algorithm on the classic image denoising task. Again we see with pruning enabled the Splashes start relatively large and uniform but near convergence they are relatively small and have adapted to local shape of the remaining non-converged regions in the model.

8. Splash Belief Propagation in Shared Memory

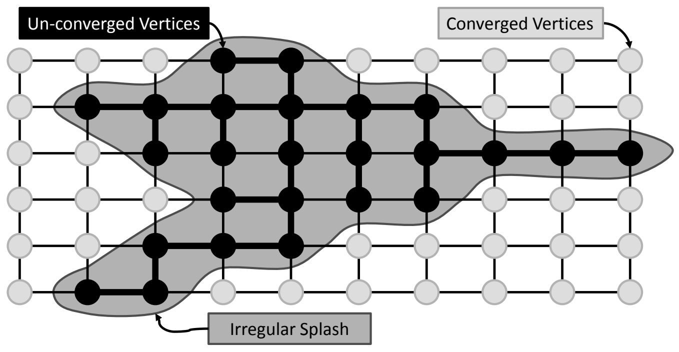
The abstract parallelization of the parallel Splash algorithm presented in the previous section (Alg. 7) can be easily mapped to the shared memory setting described in Sec. 2.1. Because the shared memory setting ensures approximately uniform access latency to all memory it is not necessary assign ownership to messages or factors. However, because multiple processors can read and modify the same message, we must ensure that messages and beliefs are manipulated in a consistent manner. Here we describe a simple locking scheme to ensure message consistency.

A single mutex is associated with each vertex in the factor graph. The mutex must be acquired, to read or update the belief associated with the vertex or any of the inbound messages to that vertex. The details of the locking mechanism for the `SendMessages` routine are described in Alg. 9. For each outbound message, the mutex is acquired and the cavity distribution is constructed from the belief by dividing out the dependence on the corresponding inbound message. The mutex is then released and any additional marginalization is accomplished on the cavity distribution. The mutex on the receiving vertex is again grabbed and the new belief is computed by subtracting out the old message value and adding back the new message value. Finally, the inbound message is updated and the mutex is released. This locking protocol ensures that only one lock is held at a time and that whenever a lock is grabbed the state of the inbound messages and belief are consistent.

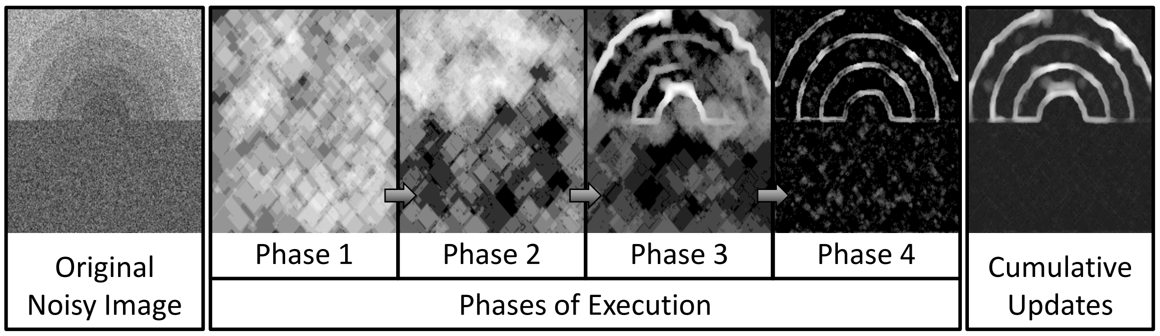
Ensuring exclusive access to each vertex can be accomplished by introducing an addition try-lock at each vertex. When a processor invokes `SendMessages` on vertex it first attempts to grab the vertex try-lock. If the processor fails, it immediately returns skipping the local update. We find that in practice this can improve performance when there are high degree vertices that may participate in multiple Splashes simultaneously.



(a) Automatic Splash Size Tuning



(b) Irregular Splash Shape



(c) Execution of the Splash Algorithm on the Denoising Task

Figure 9: **(a)** The running time of the Splash algorithm using various different Splash sizes, W with and without pruning. By enabling pruning and setting the Splash size sufficiently large we are able to obtain the optimal Splash size automatically. **(b)** Splash pruning permits the construction of irregular spanning trees that can adapt to the local convergence structure. The vertices with high belief residual,shown in black, are included in the splash while vertices with belief residual below the termination threshold, shown in gray are excluded. **(c)** The execution of the Splash algorithm on the denoising task illustrates the advantages of Splash pruning. The cumulative vertex updates are plotted with brighter regions being updates more often than darker regions. The execution is divided into 4 distinct phases. Initially, large regular (rectangular) Splashes are evenly spread over the entire model. As the algorithm proceeds the Splashes become smaller and more irregular focusing on the *challenging* regions of the model.

```
Algorithm 8: DynamicSplash($Q$, $v$, $W$)
  Input   : scheduling queue $Q$
  Input   : vertex $v$
  Input   : maximum splash size $W$
  // Construct the breadth first search ordering with $W$ message
     computations and rooted at $v$.
  fifo ← [] // FiFo Spanning Tree Queue
  $\sigma \leftarrow (v)$ // Initial Splash ordering is the root $v$
  AccumW ← $\sum_{w \in \Gamma_v} \mathcal{A}_w + |\Gamma_v| * \mathcal{A}_v$ // Total work in the Splash
  visited ← $\{v\}$ // Set of visited vertices
  fifo.Enqueue($\Gamma_v$)
  while fifo is not empty do
      $u$ ← fifo.Dequeue()
      UWork ← $\sum_{w \in \Gamma_u} \mathcal{A}_w + |\Gamma_u| * \mathcal{A}_u$
      // If adding $u$ does not cause me to exceed the work limit
      if AccumW + UWork $\leq W$ then
          AccumW ← AccumW + UWork
          Add $u$ to the end of the ordering $\sigma$
          foreach neighbors $w \in \Gamma_u$ do
              if Belief Residual of $w$ is greater than $\beta$ and is not in visited then
                  fifo.Enqueue($w$) // Add to boundary of spanning tree
                  visited ← visited ∪ $\{w\}$ // Mark Visited
  // Make Root Aware of Leaves
  foreach $u \in$ ReverseOrder($\sigma$) do
      SendMessages($u$)
      Q.SetPriority(u, 0) // Set Belief residual to zero
      foreach $w \in \Gamma_u$ do
          Q.UpdatePriority(w) // Recompute belief residual
  // Make Leaves Aware of Root
  foreach $i \in \sigma$ do
      SendMessages($i$)
      Q.SetPriority(u, 0) // Set Belief residual to zero
      foreach $w \in \Gamma_u$ do
          Q.UpdatePriority(w) // Recompute belief residual
```

Algorithm 9: Locking SendMessages

```
Input   : vertex $u$
foreach $v \in \Gamma_u$ do
    // Construct the cavity distribution
    lock $u$
        $m'_{v \to u} \leftarrow b_u / m_{v \to u}$ ;
    Marginalize $m'_{v \to u}$ if necessary ;
    // Send the message to $v$ updating the belief at $v$
    lock $v$
        $b'_v \leftarrow b_v / m_{u \to v} m'_{u \to v}$ ;
        $m_{u \to v} \leftarrow m'_{u \to v}$ ;
        $r_\Delta \leftarrow ||b'_v - b_v||_1$ ;
        $b_v \leftarrow b'_v$
```

When the message computations are fairly simple, the priority queue becomes the central synchronizing bottleneck. An efficient implementation of a parallel priority queue is therefore the key to performance and scalability. There are numerous parallel priority queue algorithms in the literature (Driscoll et al., 1988; Crupi et al., 1996; Parberry, 1995; Sanders, 1998). Many parallel priority queues require sophisticated fine grained locking mechanisms while others employ binning strategies with constraints on the priority distribution.

Because the residual priorities are a heuristic, we find that relaxing the strict ordering requirement can substantially improve performance without the need for complex locking mechanisms or constrains on priority distributions. By constructing a separate coarse locking priority queue for each processor and then randomly assigning vertices to each queue we can reduce the queue collision rate while maintaining reasonable load balance. While processors draw from their own queue they must update the priorities of vertices stored in the queues owned by other processors. Therefore, a locking mechanism is still required. If necessary, contention can be further reduced by creating multiple queues per processor.

Ensuring that the maximum amount of *productive* computation for each access to memory is critical when many cores share the same memory bus and cache. Updating all messages emanating from a vertex in `SendMessages`, maximizes the productive work for each message read. The sequential Splash operation ensures that all interior messages are received soon after they are sent and before being evicted from cache. Profiling experiments indicate that Splash algorithm reduces cache misses over synchronous BP algorithm and residual belief propagation (Splash with $W = 1$).

9. Distributed Splash Belief Propagation

In this section, we address the challenges associated with distributing the state of the Splash algorithm over p processors in the distributed memory setting. In contrast to the shared memory setting where each processor has symmetric fast access to the entire program state, in the distributed memory setting each process can only directly access its local memory and must pass messages to communicate with other processors. Therefore efficient distributed parallel algorithms are forced to explicitly distribute the program state.

While the distributed memory setting may appear more limited, it offers the potential for linear scaling in memory capacity and memory bandwidth. To fully realize the linear scaling in memory capacity and bandwidth we must partition the program state in a way that places only a fraction $1/p$ of the global program state on each processor. Achieving this memory balancing objective while partitioning the program state in a way that ensures efficient parallel computation is the central challenge in the design of scalable distributed parallel algorithms.

The distributed Splash algorithm consists of two phases. In the first phase (described in Sec. 9.1), the factor graph and messages are partitioned over the processors. In the second phase, each processor executes the sequential splash algorithm on its piece of the factor graph exchanging messages across processors as necessary. We now described the details of each phase and then present the complete distributed algorithm.

9.1 Partitioning the Factor Graph and Messages

We begin by partitioning the factor graph and messages. To maximize throughput and hide network latency, we must minimize inter-processor communication and ensure that the data needed for message computations are locally available. We define a partitioning of the factor graph over p processors as a set $\mathcal{B} = \{B_1, ..., B_p\}$ of disjoint sets of vertices $B_k \subseteq V$ such that $\cup_{k=1}^{p} B_k = V$. Given a partitioning $\mathcal{B}$ we assign all the factor data associated with $\psi_i \in B_k$ to the k^{th} processor. Similarly, for all (both factor and variable) vertices $i \in B_k$ we store the associated belief and all inbound messages on the processor k. Each vertex update is therefore a local procedure. For instance, if vertex i is updated, the processor owning vertex i can read factors and all incoming messages without communication. To maintain the locality invariant, after new outgoing messages are computed, they are transmitted to the processors owning the destination vertices.

By requiring that each processor manage the factors, beliefs, and all inbound belief propagation messages, we define the storage, computation, and communication responsibilities for each processor under a particular partitioning. Ultimately, we want to minimize communication and ensure balanced storage and computation, therefore, we can frame the minimum communication load balancing objective in terms of a graph partitioning. In particular, to evenly distribute work over the processors while minimizing network communication we must obtain a balanced minimum cut. We formally define the graph partitioning problem as:

$$\min_{\mathcal{B}} \quad \sum_{B \in \mathcal{B}} \sum_{(i \in B, j \notin B) \in E} (U_i + U_j) w_{ij} \tag{9.1}$$

$$\text{subj. to:} \quad \forall B \in \mathcal{B} \quad \sum_{i \in B} U_i w_i \leq \frac{\gamma}{p} \sum_{v \in v} U_v w_v \tag{9.2}$$

where U_i is the number of times `SendMessages` is invoked on vertex i, w_{ij} is the communication cost of the edge between vertex i and vertex j, w_i is the vertex work defined in Eq. (7.1), and $\gamma \geq 1$ is the balance coefficient. The objective in Eq. (9.1) minimizes communication while for small γ, the constraint in Eq. (9.2) ensures work balance. We define the communication cost as:

$$w_{ij} = \min(A_i, A_j) + C_{\text{comm}} \tag{9.3}$$

the size of the message plus some fixed network packet cost C_{comm}. In practice we set $C_{\text{comm}} = 1$ since it is constant for all processors and we do additional message bundling in our implementation.

Unfortunately, obtaining an optimal partitioning or near optimal partitioning is NP-Hard in general and the best approximation algorithms are generally slow. Fortunately, there are several very fast heuristic approaches which typically produce reasonable partitions in time $O(|E|)$ linear in the number of edges. Here we use the collection of multilevel graph partitioning algorithms in the METIS (Karypis and Kumar, 1998) graph partitioning library. These algorithms, iteratively coarsen the underlying graph, apply high quality partitioning techniques to the small coarsened graph, and then iteratively refine the coarsened graph to obtain a high quality partitioning of the original graph. While there are no theoretical guarantees, these algorithms have been shown to perform well in practice and are commonly used for load balancing in parallel computing.

9.1.1 UPDATE COUNTS U_i

Unfortunately, due to dynamic scheduling, the update counts U_i for each vertex depend on the evidence, graph structure, and progress towards convergence, and are not known before running the Splash algorithm. In practice we find that the Splash algorithm updates vertices in a non-uniform manner; a key property of the dynamic scheduling, which enables more frequent updates of slower converging beliefs.

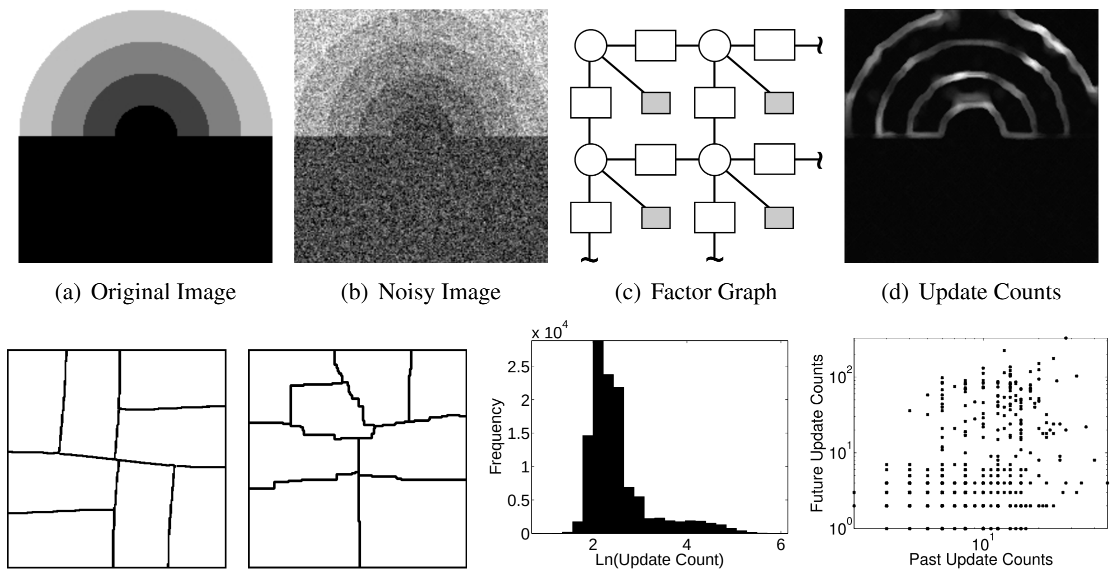
To illustrate the difficulty involved in estimating the update counts for each vertex, we again use the synthetic denoising task. The input, shown in Fig. 10(a), is a grayscale image with independent Gaussian noise $\mathrm{N}\left(0, \sigma^2\right)$ added to each pixel. The factor graph (Fig. 10(c)) corresponds to the pairwise grid Markov Random Field constructed by introducing a latent random variable for each pixel and connecting neighboring variables by factors that encode a similarity preference. We also introduce single variable factors which represent the noisy pixel evidence. The synthetic image was constructed to have a nonuniform update pattern (Fig. 10(d)) by making the top half more irregular than the bottom half. The distribution of vertex update frequencies (Fig. 10(g)) for the denoising task is nonuniform with a few vertices being updated orders of magnitude more frequently than the rest. The update patterns is temporally inconsistent frustrating attempts to estimate future update counts using past behavior (Fig. 10(h)).

9.1.2 UNINFORMED PARTITIONING

Given the previous considerations, one might conclude, as we did, that sophisticated dynamic graph partitioning is necessary to effectively balance computation and minimize communication. Surprisingly, we find that an uninformed cut obtained by setting the number of updates to a constant (i.e., $U_i = 1$) yields a partitioning with comparable communication cost and work balance as those obtained when using the true update counts. We designate $\hat{\mathcal{B}}$ as the partitioning that optimizes the objectives in Eq. (9.1) and Eq. (9.2) where we have assumed $U_i = 1$. In Tab. 2 we construct uninformed $p = 120$ partitionings of several factor graphs and report the communication cost and balance defined as:

$$\begin{aligned}\text{Rel. Com. Cost} &= \frac{\sum_{B\in\hat{\mathcal{B}}}\sum_{(u\in B, v\notin B)\in E} w_{uv}}{\sum_{B\in\mathcal{B}^*}\sum_{(u\in B, v\notin B)\in E} w_{uv}} \\ \text{Rel. Work Balance} &= \frac{p}{\sum_{v\in V} w_v}\max_{B\in\hat{\mathcal{B}}}\sum_{v\in B} w_v\end{aligned}$$

relative to the ideal cut $\mathcal{B}^*$ obtained using the true update counts U_i. We find that uninformed cuts have lower communication costs at the expense of increased imbalance. This discrepancy arises



(a) Original Image (b) Noisy Image (c) Factor Graph (d) Update Counts

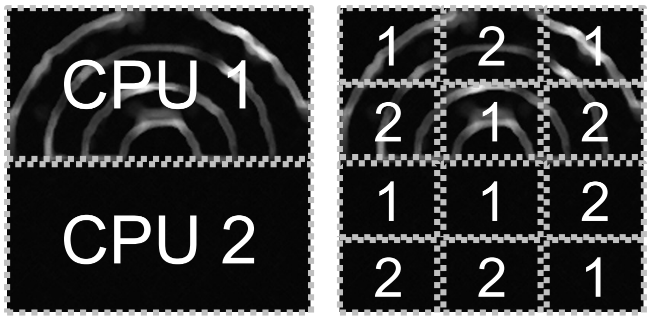
(e) Uninformed Part. (f) Informed Part. (g) Update Distribution (h) Update Prediction

Figure 10: We use the synthetic denoising task (also used in Fig. 9) to illustrate the difficult in estimating the update patterns of Splash belief propagation. **(a)** The original synthetic sunset image which was specifically created to have an irregular update pattern. **(b)** The synthetic image with Gaussian noise added. **(c)** Factor graph model for the true values for the underlying pixels. The latent random variable for each pixel is represented by the circles and the observed values are encoded in the unary factors drawn as shaded rectangles. **(d)** The update frequencies of each variable plotted in log intensity scale with brighter regions updated more frequently. **(e)** Uniformed $U_i = 1$ balanced partitioning. **(f)** The informed partitioning using the true update frequencies after running Splash. Notice that the informed partitioning assigns fewer vertices per processors on the top half of the image to compensate for the greater update frequency. **(g)** The distribution of vertex update counts for an entire execution. **(h)** Update counts from first the half of the execution plotted against update counts from the second half of the execution. This is consistent with phases of execution described in Fig. 9(c).

from the need to satisfy the balance requirement with respect to the true U_i at the expense of a higher communication cost.

Graph	Rel. Com. Cost	Rel. Work Balance
denoise	0.980	3.44
uw-systems	0.877	1.837
uw-languages	1.114	2.213
cora-1	1.039	1.801

Table 2: Comparison of uninformed and informed partitionings with respect to communication cut cost and work balance. While the communication costs of the uniformed partitionings are comparable to those of the informed partitionings, the work imbalance of the uninformed cut is typically greater.



(a) Denoise Uniformed Cut (b) Denoise Over-Partitioning

Figure 11: Over-partitioning can help improve work balance by more uniformly distributing the graph over the various processors. **(a)** A two processor uninformed partitioning of the denoising factor graph can lead to one processor (CPU1) being assigned most of the work. **(b)** Over-partitioning by a factor of 6 can improve the overall work balance by assigning regions from the top and bottom of the denoisining image to both processors.

9.1.3 OVER-PARTITIONING

Because uninformed partitions tend to have reduced communication cost and greater work imbalance relative to informed partitions, we propose over-partitioning to improve the overall work balance with a small increase in communication cost. When partitioning the graph with an uninformed cut a frequently updated subgraph may be placed within a single partition (Fig. 11(a)). To lessen the chance of such an event, we can over-partition the graph (Fig. 11(b)) into $k \times p$ balanced partitions and then randomly redistribute the partitions to the original p processors. By partitioning the graph more finely and randomly assigning regions to different processor, we more evenly distribute nonuniform update patterns improving the overall work balance. However, over-partitioning also increases the number of edges crossing the cut and therefore the communication cost. By over-partitioning in the denoise task we are able to improve the work balance (shown in Fig. 12(a)) at a small expense to the communication cost (shown in Fig. 12(b)). Over-partitioning also helps satisfy the balanced memory requirements objective by more evenly distributing the global memory footprint over the individual processors.

Choosing the optimal over-partitioning factor k is challenging and depends heavily on hardware, graph structure, and even factors. In situations where the Slash algorithm may be run repeatedly, standard search techniques may be used. We find that in practice a small factor, e.g., $k = 5$ is typically sufficient. When using a recursive bisection style partitioning algorithm where the true work split at each step is an unknown random variable, we can provide a theoretical bound on the ideal size of k. If at each split the work is divided into two parts of proportion X and $1 - X$ where $\mathbf{E}[X] = \frac{1}{2}$ and $\mathbf{Var}[X] = \sigma^2$ ($\sigma \leq \frac{1}{2}$]), Sanders (1994) shows that we can obtain work balance with high probability if we select k at least $\Omega\left(p^{\left(\log\left(\frac{1}{\sigma+1/2}\right)\right)^{-1}}\right)$.

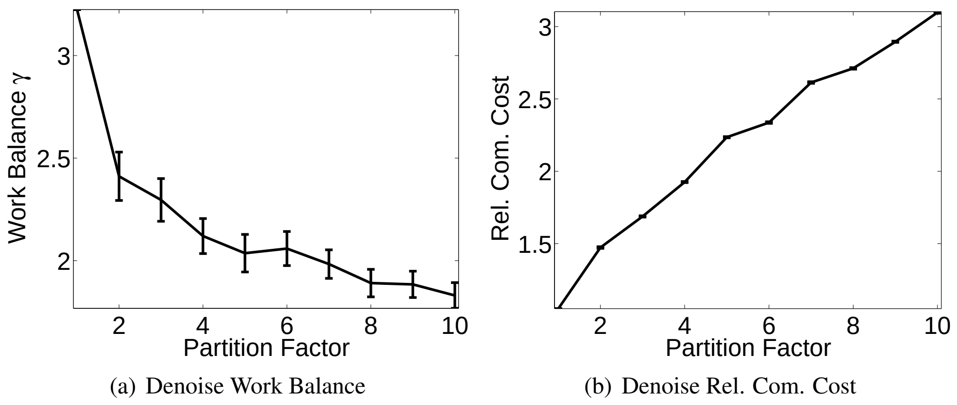


Figure 12: The effect of over-partitioning on the work balance and communication cost. For all points 30 trials with different random assignments are used and 95% confidence intervals are plotted. **(a)** The ratio of the size of the partition containing the most work, relative to the ideal size (smaller is better). **(b)** The communication cost relative to the informed partitioning (smaller is better).

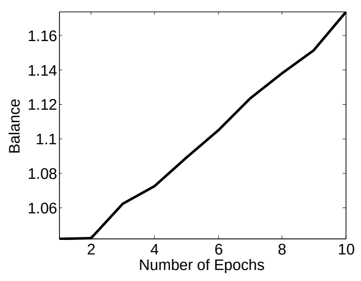
9.1.4 INCREMENTAL REPARTITIONING

One might consider occasionally repartitioning the factor graph to improve balance. For example, we could divide the execution into T phases and then repartition the graph at the beginning of each phase based on the update patterns from the previous phase. To assess the potential performance gains from incremental repartitioning we conducted a retrospective analysis. We executed the sequential Splash algorithm to convergence on the denoising model and then divided the complete execution into T phases for varying values of T. We then partitioned each phase using the true update counts for that phase and estimate the work imbalance and number of transmitted messages. While the true update counts would not be known in practice, they provide an upper bound on the performance gains of the best possible predictor.

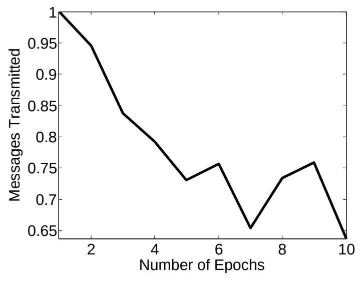
In Fig. 13 we plot the performance of both optimal phased partitioning and phased partitioning in which future update counts are predicted from previous update counts. In both cases we consider phased partitioning for $T \in \{1, \ldots, 10\}$ phases on a simulated 16 processor system. In Fig. 13(a) we see that by repartitioning more often we actually slightly increase the average imbalance over the epochs. We attribute the slight increased imbalance to the increasingly irregular local update counts relative to the connectivity structure (the regular grid). However, as seen in Fig. 13(b) the number of messages transmitted over the network (i.e., the communication cost) relative to the optimal single phase partitioning actually decreases by roughly 35%. Alternatively, using the update counts from the previous phase to partition the next phase confers a substantial decrease in overall balance (Fig. 13(c)) and a dramatic increase in the communication costs (Fig. 13(d)). We therefore conclude that only in situations where update counts can be accurately predicted and network communication is the limiting performance can incremental repartitioning lead to improved performance.

9.2 Distributing the Priority Queue

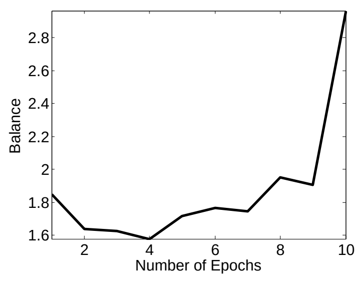
The Splash algorithm relies on a shared global priority queue. However, in the cluster computing setting, a centralized ordering is inefficient. Instead, in our approach, each processor constructs



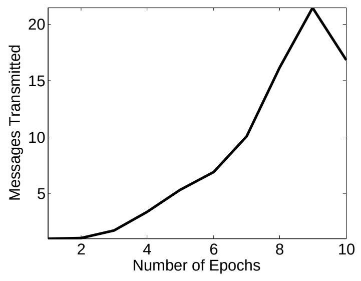
(a) Optimal Balance



(b) Optimal Comm.



(c) Naive Balance



(d) Naive Comm.

Figure 13: Here we plot the balance and communication costs of incremental repartitioning against the number of epochs (repartitioning phases) in a single execution. The balance is measured as the work ratio between the smallest and largest work block. We present the messages transmitted measured relative to the optimal performance for a single phase rather than raw message counts. **(a,b)** The balance and communication costs for optimal phased partitioning where the update counts in each phase are known in advance. **(c,d)** The balance and communication costs for naive phased partitioning where the update counts in each phase are assumed to be the update counts observed in the previous phase.

a local priority queue and iteratively applies the Splash operation to the top element in its local queue. At any point in the distributed execution one of the processor is always applying the Splash operation to the globally highest residual vertex. Unfortunately, the remaining $p-1$ highest vertices are not guaranteed to be at the top of the remaining queues and so we do not recover the original shared memory scheduling. However, any processor with vertices that have not yet converged, must eventually update those vertices and therefore can always make progress by updating the vertex at the top of its local queue. In Sec. 9.6 we show that the collection of local queues is sufficient to retain the original optimality properties of the Splash algorithm.

9.3 Distributed Termination

In the absence of a common synchronized state it is difficult to assess whether a convergence condition has been globally achieved. The task of assessing convergence is complicated by the fact that

Algorithm 10: Distributed Termination Algorithm

```
Output: Converged
Data: Global Variables: LastToken, NetMessagesSent, NetMessagesReceived
// Infinite Loop
while True do
    if I have the Token then
        // Check for the global termination
        if Token == LastToken AND Token.NumSent == Token.NumReceived then
            return Converged = True
        else
            // Update the Token with this node's contributions
            // Reset the network message counters
            Token.NumSent ← Token.NumSent + NetMessagesSent
            Token.NumReceived ← Token.NumReceived + NetMessagesReceived
            NetMessagesSent ← 0
            NetMessagesReceived ← 0
            LastToken ← Token // Remember the last Token we saw
            Transmit(Token, NextNode) // Send the updated token to the
                next node in the ring

    Sleep and Wait For Any Incoming Network Message.

    if Network Message is not a Token then
        // This is not a Token but a BP message.  Wake up and
            process it
        return Converged = False
```

when a node locally converges, it could be “woken up” at any time by an incoming message which may causes its top residual to exceed the termination threshold.

Fortunately, this is a well studied task known as the “distributed termination problem” (Matocha and Camp; Mattern, 1987). We implement a variation of the algorithm described in Misra (1983). The algorithm is described in Alg. 10 and proceeds as follows.

A ring is defined over all the nodes and a token comprising of 2 integer values, **NumSent** and **NumReceived**, is passed in one direction around the ring. The integer values represent the total number of network messages sent and received throughout the execution of the program across all machines. When the node holding onto the token converges, the node will add the number of network messages it has sent and received since the last time it has seen the token, to **NumSent** and **NumReceived** respectively. The node will then forward the token to the next node in the ring. Global termination is achieved when the token completes a loop around the ring without changing, and **NumSent** equals **NumReceived**

9.4 Flap Prevention

We say that a node is "flapping" if it switches rapidly between performing computation, and waiting inside the Distribution Termination algorithm. Such "flapping" behavior may be observed as the algorithm approaches termination. Nodes which have locally converged will still receive BP messages from the nodes which have not converged. These new messages may cause the top residual to exceed the termination threshold, resulting in a very brief flurry of computation which is immediately followed by the node reporting local convergence.

Such behavior is undesirable as the time spent waiting is wasted, and instead could be spent performing useful computation. A plausible solution would be to avoid waiting, but to perform computation continuously. This solution however, results in an ill-defined global termination procedure as any network message could increase the top residual above the termination threshold.

We solve the problem using a hysteresis procedure. In addition to the standard termination threshold β, we also define a "lower threshold" on each node called β_0 where $\beta_0 \leq \beta$. The Splash algorithm will try to achieve local convergence using β_0 as the termination threshold, but the node will only resume computation if incoming messages cause the top residual to exceed β.

Instead of requiring the user to provide β_0, we provide an adaptive procedure to find β_0. We initially set β_0 to be equal to β. When the node receives a message which causes the top residual to exceed β, we decrease β_0 by a constant factor: $\beta_0 \leftarrow \beta_0/c$ where $c > 1$. The Splash algorithm then proceeds as usual, but reporting local convergence only when the top residual falls below β_0. The choice of c is important as small values of c will result in excessive flapping, while large values of c will result in increased runtime due to unnecessarily low termination thresholds. We set $c = 2$ in our distributed experiments.

9.5 The Distributed Splash Algorithm

We now present the distributed Splash algorithm (Alg. 11) which can be divided into two phases, setup and inference. In the setup phase, in Line 1 we over-segment the input factor graph into kp pieces using the METIS partitioning algorithm. Note that this could be accomplished in parallel using ParMETIS, however our implementation uses the sequential version for simplicity. Then in Line 2 we randomly assign k pieces to each of the p processors. In parallel each processor collects its factors and variables (Line 3). On Line 4 the priorities of each variable and factor are set to infinity to ensure that every vertex is updated at least once.

On Line 5 we evaluate the top residual with respect to the β convergence criterion and check for termination in the token ring. On Line 6, the `DynamicSplash` operation is applied to v. In the distributed setting Splash construction procedure does not cross partitions. Therefore each Splash is confined to the vertices on that processor. After completing the Splash all external messages from other processors are incorporated (Line 7). Any beliefs that changed during the Splash or after receiving external messages are promoted in the priority queue on Line 9. On Line 10, the external messages are transmitted across the network. The process repeats until termination at which point all beliefs are sent to the originating processor.

Empirically, we find that accumulating external messages and transmitting only once every 5-10 loops tends to increase performance by substantially decreasing network overhead. Accumulating messages may however adversely affect convergence on some graphical models. To ensure convergence in our experiments, we transmit on every iteration of the loop.

Algorithm 11: The Distributed Splash Algorithm

```
   // The graph partitioning phase ========================>
   // Construct factor $k$ over-partitioning for $p$ processors
1  $\mathcal{B}_{\text{temp}} \leftarrow$ OverSegment($G, p, k$);
   // Randomly assign over-partitioning to the original
      processors
2  $\mathcal{B} \leftarrow$ RandomAssign($\mathcal{B}_{temp}, p$);
   // The distributed inference phase =====================>
   forall $b \in \mathcal{B}$ do in parallel
       // Collect the variables and factors associated with this
          partition
3      Collect($\mathcal{F}_b, \mathcal{X}_b$) ;
       // Initialize the local priority queue
4      Initialize($Q$);
       // Loop until TokenRing signifies convergence
5      while TokenRing($Q, \beta$) do
           $v \leftarrow$ Top($Q$) ;
6          DynamicSplash($v$, $W_{\max}$, $\beta$);
7          RecvExternalMsgs();
           // Update priorities affected by the Splash and newly
              received messages
8          foreach $u \in$ local changed vertices do
9              Promote($Q$, $||\Delta b_v||_1$);
10         SendExternalMsgs();
```

9.6 Preserving Splash Chain Optimality

We now show that Splash retains the optimality in the distributed setting.

Theorem 9.1 (Splash Chain Optimality) *Given a chain graph with $n = n$ vertices and $p \leq n$ processes, the distributed Splash algorithm with no over-segmentation, using a graph partitioning algorithm which returns connected partitions, and with work Splash size at least $2\sum_{v \in V} w_v/p$ will obtain a τ_ϵ-approximation in expected running time $O\left(\frac{n}{p} + \tau_\epsilon\right)$.*

Proof [Proof of Theorem 9.1] We assume that the chain graph is optimally sliced into p connected pieces of n/p vertices each. Since every vertex has at most 2 neighbors, the partition has at most $2n/p$ work. A Splash anywhere within each partition will therefore cover the entire partition, performing the complete "forward-backward" update schedule.

Because we send and receive all external messages after every splash, after $\lceil \frac{\tau_\epsilon}{n/p} \rceil$ iterations, every vertex will have received messages from vertices a distance of at least τ_ϵ away. The runtime will therefore be:

$$\frac{2n}{p} \times \left\lceil \frac{\tau_\epsilon}{n/p} \right\rceil \leq \frac{2n}{p} + 2\tau_\epsilon$$

Since each processor only send 2 external messages per iteration (one from each end of the partition), communication therefore only adds a constant to the total runtime. ■

10. EXPERIMENTS

We evaluated the Splash belief propagation algorithm in the sequential, multi-core (shared memory), and cluster (distributed) settings on a wide variety of graphical models. We compare our results against several popular sequential belief propagation algorithms and their "natural" parallelizations (see Appendix B for details about these algorithms). In this section we briefly describe the graphical models used to assess the Splash algorithm and then present performance results for the sequential and parallel settings.

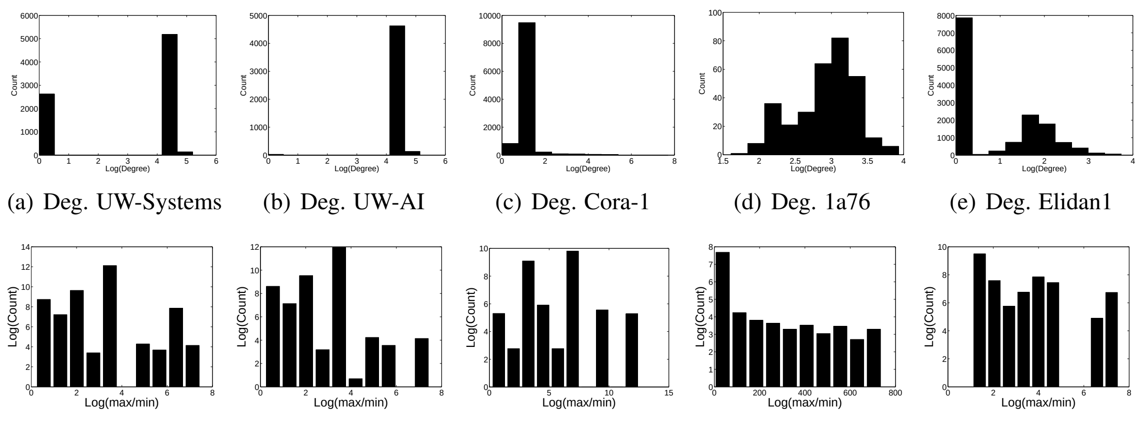
10.1 Experimental Setup

We obtained 7 large Markov Logic Networks (MLNs) from Domingos et al. (2008), 276 protein side chain models from Yanover et al., 8 protein-protein interaction networks Elidan et al. (2006), and 249 graphical models of varying structures from the UAI (A. Darwiche and Otten, 2008). In addition we constructed several variations of the synthetic denoising task described in Sec. **??**. In Tab. 3 and Fig. 10.1 we summarize the properties of a few representative instances.

Name	Type	Var. Type	$\|\mathcal{X}\|$	$\|\mathcal{F}\|$	$\|E\|$
uw-systems	MLN	$\{0,1\}$	7,951	209,843	417,447
uw-ai	MLN	$\{0,1\}$	4,790	175,607	351,596
cora-1	MLN	$\{0,1\}$	10,843	28,041	55,848
protein-1a76	Irregular MRF	$\{1,\ldots,80\}$	315	3,161	6,322
elidan1	MRF	$\{0,1\}$	14,306	21,841	57,659

Table 3: We evaluated the Splash belief propagation algorithm on a wide range of graphical models. These networks drawn from several different domains have varying sparsity structure, variable arity, and size. The columns $|\mathcal{X}|$, $|\mathcal{F}|$, and $|E|$ correspond to the number of variables, factors, and edges respective.

The Markov Logic Networks (MLNs), provided by (Domingos et al., 2008), represent a probabilistic extension to first-order logic obtained by attaching weights to logical clauses. We used *Alchemy* to compile several MLNs into factor graph form. We constructed MLNs from the UW-CSE relational data-set (Domingos, 2009) and present results for the two largest MLNS, *uw-systems* and *uw-ai*. The factor graphs derived from *uwcsedata*, are considerably more challenging than many of the factor graphs and we found that only our Splash algorithm consistently converges for these models. As a consequence we are unable to use the *uwcsedata* models to compare scaling results against the baseline algorithms. Therefore we also constructed MLNs from the *cora* entity resolution data set. These smaller MLNs are more amenable to traditional belief propagation algorithms permitting an effective baseline comparison. With highly irregular degree distributions (Fig. 10.1) and many variables that participate in a large number of factors, these models test effective scheduling methods and in particular the need for belief residuals. While lifted inference (Singla and Domingos, 2008)



(a) Deg. UW-Systems (b) Deg. UW-AI (c) Deg. Cora-1 (d) Deg. 1a76 (e) Deg. Elidan1

(f) Dyn. Rng. UW-Sys. (g) Dyn. Rng. UW-AI (h) Dyn. Rng. Cora-1 (i) Dyn. Rng. 1a76 (j) Dyn. Rng. Elidan1

Figure 14: The distributions of the variable degree and factor dynamic range help characterize the connectivity structure of a network. In **(a)**, **(b)**, **(c)**, **(d)**, and **(e)** we plot the empirical distribution of the $\log(\text{degree})$ of the variables. Models like the large Markov Logic Networks (**(a)** and **(b)**) have irregular degree distributions with many sparsely connected variables and many very densely connected variables. Alternatively the protein side-chain factor graphs have a more uniform degree distribution and therefore connectivity structure. In **(f)**, **(g)**, **(h)**, **(i)**, and **(j)** we plot the distribution of the dynamic range (**?**) given by $\max_{\mathbf{x_u},\mathbf{x_v}} \log\left(\frac{\psi_u(\mathbf{x_u})}{\psi_v(\mathbf{x_v})}\right)$. Higher dynamic ranges imply more deterministic potentials and therefore stronger coupling between the variables.

is often used on MLNs, here we used standard inference on the full model to retain consistency across all experiments.

The pairwise Markov Random Fields provided by Yanover et al. are derived from the protein side-chain prediction task which can be framed as finding the energy minimizing joint assignment to a pairwise MRF. The models vary in complexity with up to 700 variables and angle discretizations ranging from 2 to 80. Due to the high dynamic range in the node and edge potentials (see Fig. 14(i)), log-space message calculations were required. In addition to the side-chain MRF structures and potentials, the true side-chain configurations, (obtained through x-ray crystallography) were provided by (Yanover et al.) and are used to assess the accuracy of the MAP estimates. The protein MRFs test parallel inference techniques in settings where variables are highly connected and share strong interactions.

We also considered protein-protein interaction networks provided by Elidan et al. (2006). These networks consist of approximately 30,000 binary hidden variables with structures induced from a relational Markov network (Taskar et al., 2004) which defines a set of template potentials. These networks contain node factors derived from the noisy observations and trinary factors which encode interactions between co-localized proteins. We adopt the setup of Elidan et al. (2006) using 8 networks with the same structure but different parameters. While Elidan et al. (2006) found these networks, which have many cycles and relatively strong potentials, to be challenging for conventional belief propagation type algorithm.

We evaluated the average accuracy of the belief estimates on the UAI 2008 Probabilistic Inference Evaluation data set (A. Darwiche and Otten, 2008). Ground truth data was obtained using the exact inference software ACE 2.0 (Huang et al., 2006). A benchmark set of 249 models were chosen based on what we managed to get ACE 2.0 to solve in reasonable time.

10.1.1 IMPLEMENTATION

We implemented all algorithms in C++ using PThreads for shared memory parallelism and MPI (MPICH2) for message passing in the distributed setting. All algorithms used the same core scheduling, convergence assessment, and message computation code and differed only in the update scheduling and overall algorithm structure as described earlier and in Appendix B. To ensure numerical stability and convergence, log-space message calculations and 0.3 damping were used. All code was compiled using GCC 4.3.2 and run on 64Bit Linux systems. We have released the code for all the implemented inference algorithms along with user-friendly Matlab wrappers for sequential and shared memory inference [7] at **?**. All timing experiments were conducted in isolation without any external load on the machines or network resources. All sequential and shared memory experiments were conducted on Quad-Core AMD Opteron 2.7GHz (2384) processors with shared 6MB L3 cache.

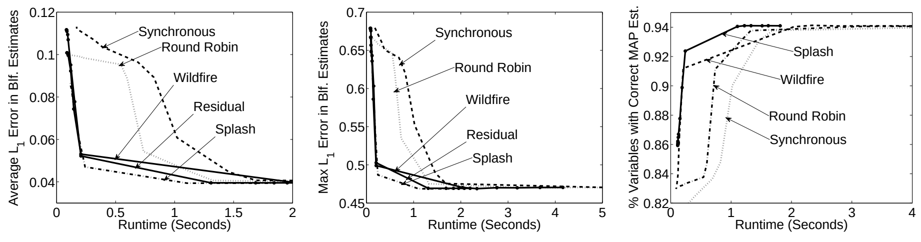
Distributed experiments were conducted on cluster composed of 13 blades (nodes) each with 2 Intel Xeon E5345 2.33GHz Quad core processors. Each processor is connected to the memory on each node by a single memory controller. The nodes are connected via a fast Gigabit ethernet switch. We invoked the weighted *kmetis* partitioning routine from the METIS software library for graph partitioning and partitions were computed in under 10 seconds.

10.2 Sequential Setting

The running time, computational efficiency, and accuracy of the sequential Splash algorithm were evaluated in the single processor setting. In Fig. 15(a), Fig. 15(b), and Fig. 15(c) we plot the average belief accuracy, worst case belief accuracy, and map prediction accuracy against the runtime on a subset of the UAI 2008 Probabilistic Inference Evaluation data set (A. Darwiche and Otten, 2008). We ran each belief propagation algorithm to $\beta = 10^{-5}$ convergence and recorded the runtime in seconds and the marginal estimates for all variables. We compared against the exact marginals obtained using Ace 2.0 (Huang et al., 2006). In all cases the Splash algorithm obtained the most accurate belief estimates in the shortest time. The other baseline belief propagation algorithms follow a consistent pattern with wildfire and residual belief propagation (dynamical scheduled algorithms) consistently outperforming round-robin and synchronous belief propagation (fixed schedules). We also assessed accuracy on the protein side chain prediction task. Here we find that all belief propagation algorithms achieve roughly the same prediction accuracy of 73% for χ_1 and χ_2 angles which is consistent with the results of Yanover et al..

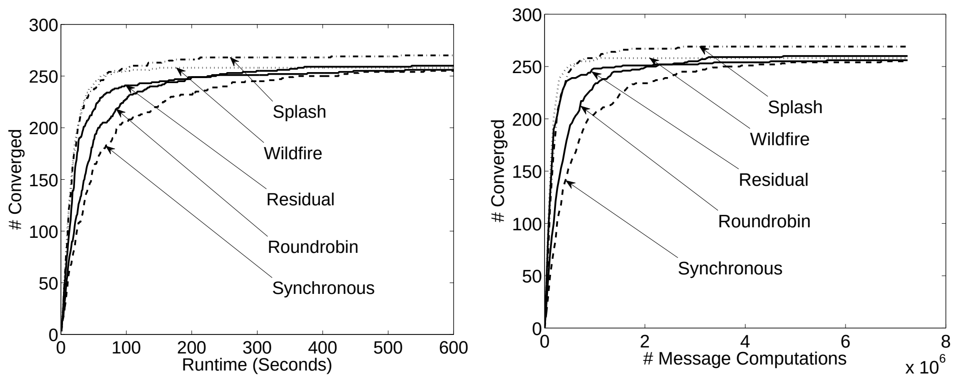
We assessed the convergence of the Splash belief propagation algorithm using several different metrics. In Fig. 16(a) we plot the number of protein networks that have converged ($\beta = 10^{-5}$) against the run-time. Here we find that not only does Splash belief propagation converge faster than other belief propagation algorithms, it also converges more often. In Fig. 16(b) we plot the number of protein networks that have converged ($\beta = 10^{-5}$) against the number of message computations. Again, we see that Splash belief propagation converges using less work than other belief propagation algorithms. Surprisingly, when we extend this analysis to the 5 UW-CSE graphs Splash belief

7. Matlab wrappers for distributed memory inference are substantially more challenging to write because of the additional system requirements imposed by MPI.



(a) Average L_1 Error (b) Maximum L_1 Error (c) MAP Accuracy

Figure 15: We assessed the accuracy of Splash algorithm using the exact inference challenge networks from **?** as well as the protein side chain prediction networks obtained from Yanover et al.. In **(a)** and **(b)** we plot the average and max L_1 error in the belief estimates for all variables as a function of the running time. In **(c)** we plot the prediction accuracy of the MAP estimates as a function of the running time. In all cases we find that Splash belief propagation achieves the greatest accuracy in the least time.

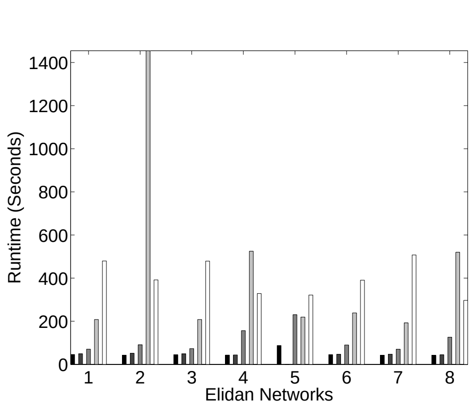


(a) Protein Convergence vs. Time (b) Protein Convergence vs. Work

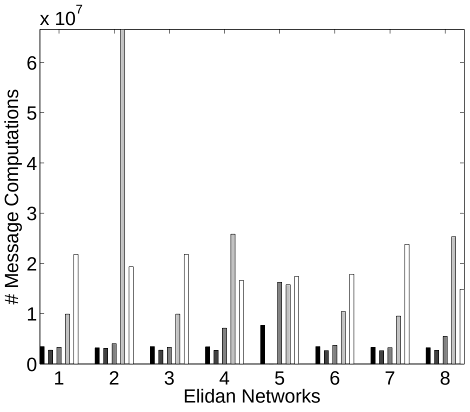
Figure 16: The Splash algorithm demonstrates faster and more consistent convergence than other baseline algorithms on a single processor. In the number of converged ($\beta = 10^{-5}$) networks (out of 276) is plotted against the runtime **(a)** and number of message calculations **(b)**.

propagation only fails to converge ($\beta = 10^{-5}$) on one, *uw-ai*, while the popular baseline algorithms fail to converge on *all* of the UW-CSE MLNs.

In Fig. 17(a) and Fig. 17(b) we directly compare the running time and work of the sequential Splash algorithm against other common scheduling strategies on the challenging *elidan* protein-protein interaction networks. The results are presented for each of the 8 networks separately with bars in the order *Splash*, *Residual*, *Wildfire*, *Round-Robin*, and *Synchronous*. All algorithms converged on all networks except residual belief propagation which failed to converge on the *elidan5* network. In all cases the Splash algorithm achieves the shortest running time and is the in the bottom two in total message computations. Since the Splash algorithm favors faster message computations (lower degree vertices), it is possible for the Splash algorithm is able to achieve a shorter runtime even while doing slightly more message computations.



(a) Runtime Elidan Networks



(b) Work Elidan Networks

Figure 17: We assessed runtime and convergence on the *elidan* protein-protein interaction networks. In **(a)** and **(b)** we plot the runtime in seconds and the work in message computations for each algorithm on each network. Each bar represents a different algorithm in the order *Splash*, *Residual*, *Wildfire*, *Round-Robin*, and *Synchronous* from left (darkest) to right (lightest).

10.3 Shared Memory Parallel Setting

In the shared memory setting we focus our analysis on computational efficiency and speedup. The accuracy and convergence behavior in the shared memory parallel setting are equivalent to the single processor setting. We present runtime, speedup, work, and efficiency results as a function of the number of cores on the 1a76 (Fig. 18) and Elidan-1 (Fig. 19) protein networks as well as the Cora-1(Fig. 20), UW-AI (Fig. 21), and UW-Systems (Fig. 22) MLNs. While all algorithms converged on the 1a76, Elidan-1, Cora-1 networks, only Residual and Splash converged on the UW-Languages network and only Splash converged on the UW-Systems and UW-AI networks.

The runtime, shown in sub-figure **(a)** of Fig. 18 through Fig. 22, is measured in seconds of elapsed wall clock time before convergence. An ideal run-time curve for p processors is proportional $1/p$. As discussed earlier it is important that the initial runtime $p = 1$ be as low as possible. On all of the models we find that the Splash algorithm achieved a runtime that was strictly less than the standard belief propagation algorithms. We also find that the popular static scheduling algorithms, round-robin and synchronous belief propagation, are consistently slower than the less common dynamic scheduling algorithms, Residual, Widlfire, and Splash.

The speedup, shown in sub-figure **(b)**, is measured relative to the fastest single processor algorithm. By measuring the speedup relative to the fastest single processor algorithm we ensure that highly parallel inefficiencies do not appear as optimal scaling. An algorithm with highly parallel inefficiency will demonstrate ideal scaling when measured relative to itself but poor scaling when measured relative to the best (most efficient) single processor algorithm. As a consequence of the relative-to-best speedup definition, inefficient algorithms may exhibit a speedup less than 1. We find that the Splash algorithm scales better and achieves near linear speedup on all of the models. Furthermore, we again see a consistent pattern in which the dynamic scheduling algorithms dramatically outperform the static scheduling algorithms. The inefficiency in the static scheduling algorithms (synchronous and round robin) is so dramatic that the parallel variants seldom achieve more than a factor of 2 speedup using 16 processors.

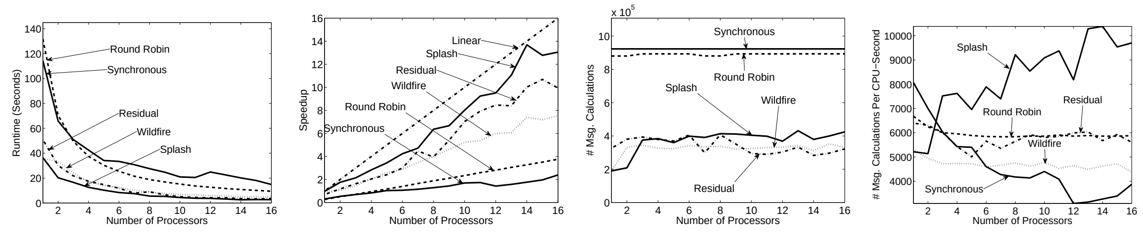
We measured work, plotted in sub-figure **(c)**, in terms of the number of message calculations before convergence. The total work, which is a measure of algorithm efficiency, should be as small as possible and not depend on the number of processors. We find that the Splash algorithm generally does the least work and that the number of processors has minimal impact on the total work done. However, surprisingly, we found on several of the Cora MLNs, the Wildfire algorithm actually does slightly less work than the Splash and Residual algorithms.

Finally, we assessed computation efficiency, shown in sub-figure **(d)**, by computing the number of message calculations per processor-second. The computational efficiency is a measure of the raw throughput of message calculations during inference. Ideally, the efficiency should remain as high as possible. The computational efficiency, captures the cost of building spanning trees in the Splash algorithm or frequently updating residuals in the residual algorithm. The computational efficiency also captures concurrency costs in-curred at spin-locks and barriers. In all cases we see that the Splash algorithm is considerably more computationally efficient. While it is tempting to conclude that the implementation of the Splash algorithm is more optimized, all algorithms used the same message computation, scheduling, and support code and only differ in the core algorithmic structure. Hence it is surprising that the Splash algorithm, even with the extra overhead associated with generating the spanning tree in each Splash operations. However, by reducing the frequency of queue operations and the resulting lock contention, by increasing cache efficiency through message reuse in the forward and backward pass, and by favoring lower work high residual vertices in each spanning tree, the Splash algorithm is able to update more messages per processor-second.

10.4 Distributed Parallel Setting

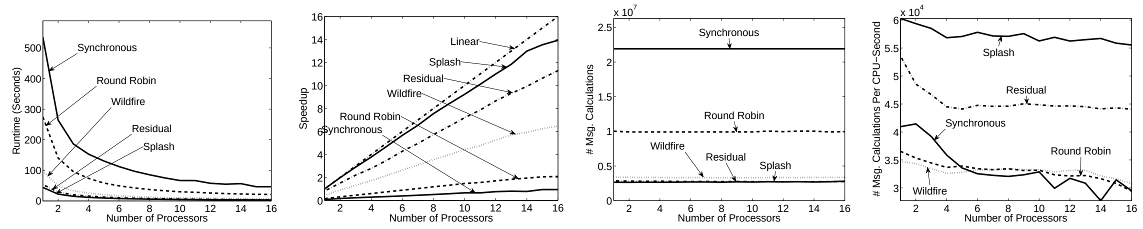
To assess the performance of the Splash algorithm in the distributed setting we focus our attention on larger models. While it is possible to run any of the distributed belief propagation algorithms on the smaller models, the cost of starting a distributed execution dominates the already sub-second multicore runtime. Moreover, it is unreasonable to expect to see a speedup on a large 104 processor distributed system when the single processor runtime is less than a few seconds. To adequately challenge the distributed algorithms we consider the Elidan protein networks (with a stricter termination bound $\beta = 10^{-10}$) and the larger UW-CSE MLNs (with the termination bound $\beta = 10^{-5}$ from the previous section). All the baseline algorithms converged on the Elidan protein networks except synchronous at 1 processor which ran beyond the 10,000 second cutoff. Consistent with the multicore results, only the Splash algorithm converges on the larger UW-CSE MLNs.

Since all the baseline algorithms converged on the smaller Elidan networks we use these networks for algorithm comparison. Because the single processor runtime on these networks is only a few minutes we analyze performance from 1 to 40 processors in increments of 5 rather than using the entire 104 processor system. As discussed in the experimental setup, we consider partition factors of 2, 5, and 10 for each algorithm and present the results using the best partition factors for each of the respective algorithms (which is typically 5). In Fig. 23(a) and Fig. 23(b) we present the standard runtime and speedup curves which show that the Splash algorithm achieves the best runtime and speedup performance with a maximum speedup of rough 23 at 40 processors. While the speedup does not scale linearly we achieve a runtime of 6.4 seconds at 40 processors making it difficult to justify extending to an additional 80 processors. We also plot the amount of work



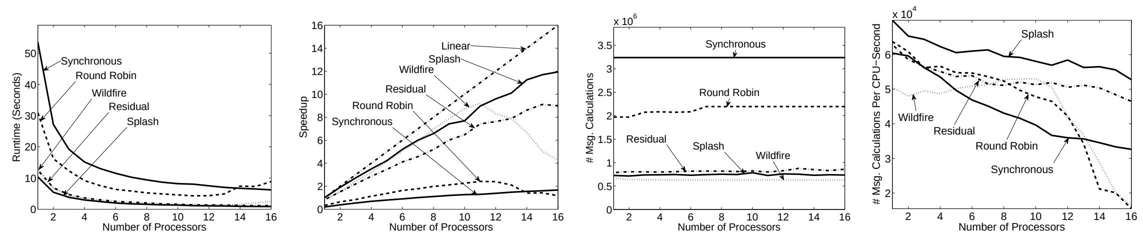
(a) Runtime (b) Speedup (c) Work (d) Comp. Efficiency

Figure 18: Shared memory results for the 1a76 Protein Network



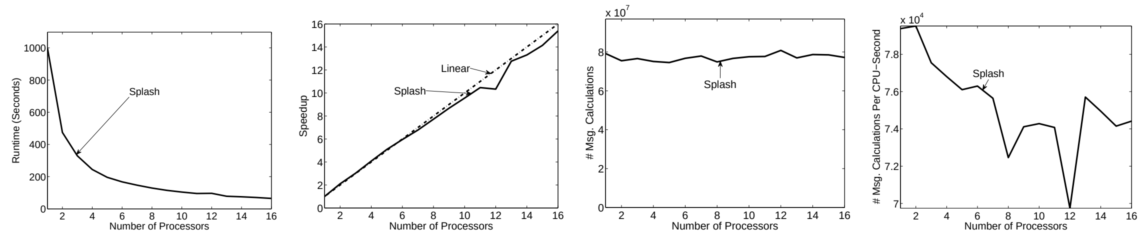
(a) Runtime (b) Speedup (c) Work (d) Comp. Efficiency

Figure 19: Shared memory results for the Elidan-1 Protein Network



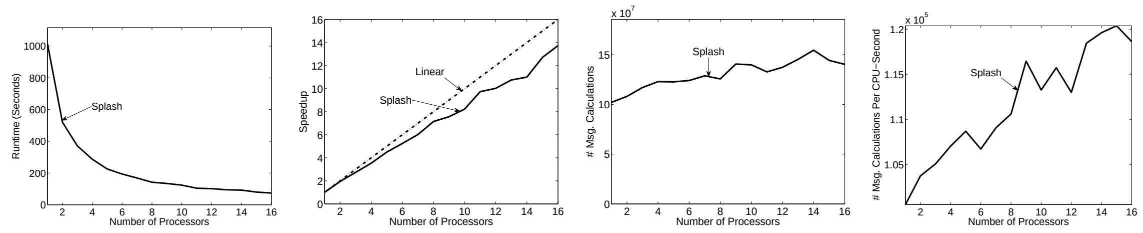
(a) Runtime (b) Speedup (c) Work (d) Comp. Efficiency

Figure 20: Shared memory results for the Cora-1 MLN



(a) Runtime (b) Speedup (c) Work (d) Comp. Efficiency

Figure 21: Shared memory results for the UW-AI MLN



(a) Runtime (b) Speedup (c) Work (d) Comp. Efficiency

Figure 22: Shared memory results for the UW-Systems MLN

Fig. 23(c) as a function of the number of processors and find that despite the message delays and distributed scheduling, the algorithmic efficiency remains relatively constant with the Splash and Wildfire algorithm performing optimally.

In Fig. 23(d) we plot the total amount of network traffic measured in bytes as a function of the number of processors and we find that the Splash algorithm performs the minimal amount of network communication. Furthermore the amount of network communication scales linearly with the number of processors. We also find in Fig. 23(e) that the Splash algorithm is able to more fully utilize the cluster by executing more message calculations per processor-second and doing so consistently as the number of processor scales. Finally, we compare the algorithms in terms of their communication efficiency measured in total bytes sent per processor-second. The total bytes sent per processor-second, plotted in Fig. 23(f) is a measure of the network bandwidth used per machine. Again we find that the Splash requires the minimal network bandwidth and that the network bandwidth grows sub-linearly with the number of processors. At peak load we use less than a megabyte per second per machine.

The larger UW-CSE MLNs provide the best demonstration of the effectiveness of the Splash algorithm. None of the other inference algorithms converge on these models, and the models are sufficiently large to justify the use of a cluster. In Fig. 24 and Fig. 25, we provide plots for the UW-Systems and the UW-AI MLNS respectively.

The UW-Systems model is larger, and we could demonstrate a linear to super-linear speedup up to about 65 processors. At 65 processors, the runtime is about 23.6 seconds with a partition factor of 10. The speedup curves start to flatten out beyond that. The plot in Fig. 24(b) justify the claim that over-partitioning can increase performance through improved load-balancing. Over-partitioning however, does lead to increased network communication as seen in Fig. 24(d) and Fig. 24(f). At 65 processors, over 100 MB of network messages were communicated in 23 seconds.

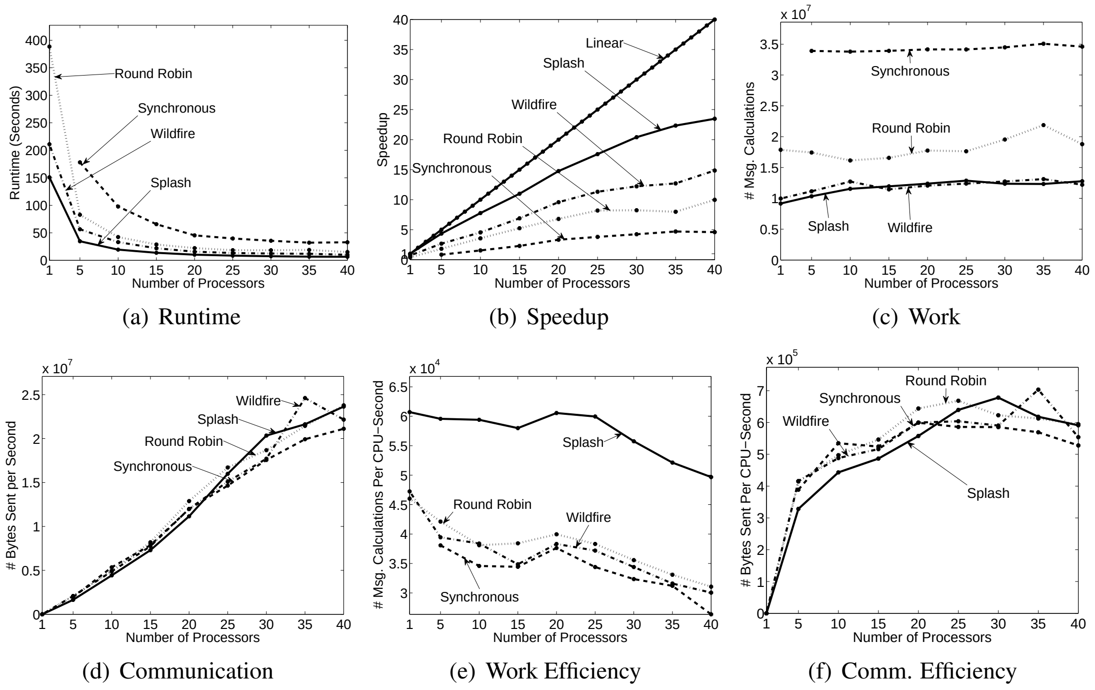
The UW-AI model in Fig. 25 has similar results. However, as the model is smaller, we notice that the speedup curves start to flatten out earlier, demonstrating linear speedup only up to about 39 processors. Once again, at 65 processors, the runtime is only 22.7 seconds with a partition factor of 10.

11. Conclusion

In this work we identified, characterized, and addressed the challenges to the design and implementation of fast efficient parallel belief propagation in the shared and distributed memory settings. We identified the underlying sequential structure to message passing inference. Through a worst-case asymptotic runtime analysis we characterized the parallel runtime of the popular synchronous belief propagation algorithm revealing a substantial (quadratic) inefficiency. We then presented the optimal parallel forward-backward schedule which achieves optimal performance.

We identified the role of approximation in exposing greater parallelism in belief propagation. Using the notion of a τ_ϵ-approximation, we characterized the role of message approximations in the optimal parallel running time of belief propagation. We showed that the naturally parallel synchronous belief propagation algorithm is far from the optimal lower bound even in the τ_ϵ-approximation setting. We then presented simple parallel ChainSplash algorithm which is built around small local forward-backward schedules and achieves the optimal lower bound.

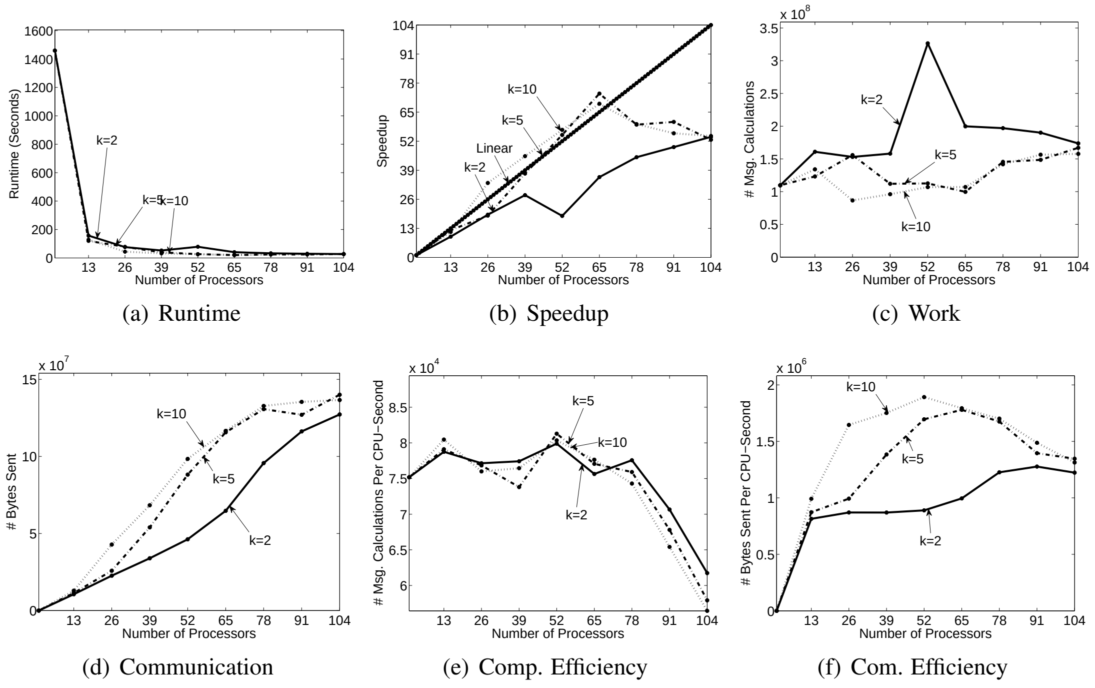
We extended the chain specific Splash operation in the ChainSplash algorithm to arbitrary cyclic graphical models by generalizing the local forward backward scheduling to a forward-backward



(a) Runtime (b) Speedup (c) Work

(d) Communication (e) Work Efficiency (f) Comm. Efficiency

Figure 23: Distributed Parallel Results for the Elidan1 Protein Network

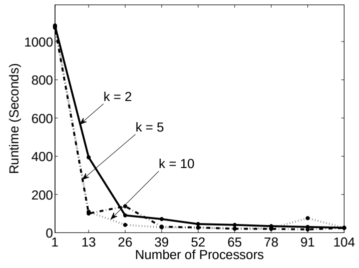


(a) Runtime (b) Speedup (c) Work

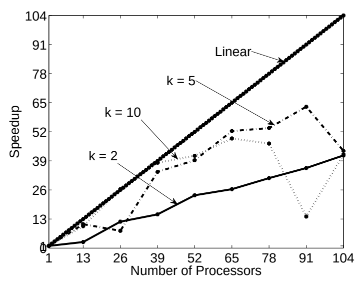
(d) Communication (e) Comp. Efficiency (f) Com. Efficiency

Figure 24: Distributed Parallel Results for the UW-Systems MLN

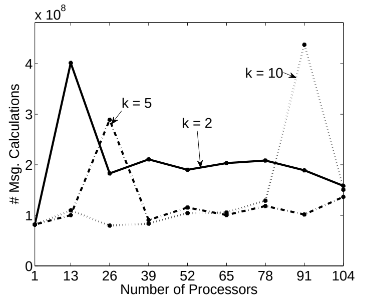
scheduling on specially constructed local spanning-trees. We developed dynamic belief based scheduling technique to select Splash locations intelligently adapt the size and shape of each Splash. We then demonstrated that the resulting Splash algorithm out-performs all existing schedules in



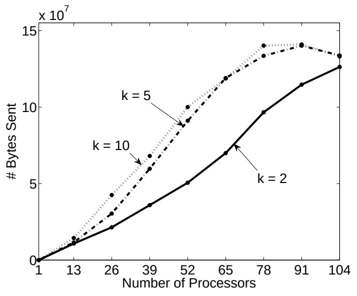
(a) Runtime



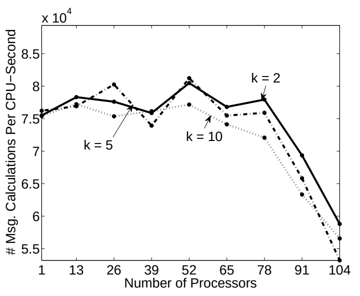
(b) Speedup



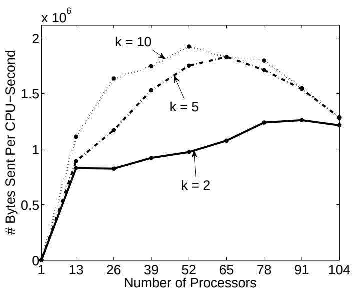
(c) Work



(d) Communication



(e) Comp. Efficiency



(f) Com. Efficiency

Figure 25: Distributed Parallel Results for the UW-AI MLN

the sequential and shared memory setting. Finally we presented how graph partitioning and over-partitioning can be used to adapt the Splash algorithm to the distributed setting. Again in the distributed setting we demonstrated that the Splash algorithm outperforms all existing techniques.

Acknowledgments

This work is supported by ONR Young Investigator Program grant N00014-08-1-0752, the ARO under MURI W911NF0810242, DARPA IPTO FA8750-09-1-0141, the NSF under grants NeTS-NOSS and CNS-0625518 and Joseph Gonzalez is supported by the AT&T Labs Fellowship. We thank Intel Research for cluster time.

Appendix A. Belief Residuals

A.1 Message Residuals May Underestimate Changes in Beliefs

For a vertex of degree d, ϵ changes to individual messages can compound, resulting in up to $d\epsilon$ change in beliefs. We demonstrate this behavior by considering a variable X_i with $d = |\Gamma_i|$ incoming messages $\{m_1, \ldots, m_d\}$. Suppose all the incoming messages are changed to $\{m'_1, \ldots, m'_d\}$ such that the resulting residual is less than 2ϵ (i.e., $\forall k : |m'_k - m_k|_1 \leq 2\epsilon$); and that the messages have converged using the convergence criterion in Eq. (3.6) (i.e., $2\epsilon \leq \beta$). However, the effective change in belief depends linearly on the degree, and therefore can be far from convergence.

Assume $\{m_1, \ldots, m_d\}$ are binary uniform messages. Then the belief at that variable is also uniform (i.e., $b_i = [\frac{1}{2}, \frac{1}{2}]$). If we then perturb the messages $m'_k(0) = \frac{1}{2} - \epsilon$ and $m'_k(1) = \frac{1}{2} + \epsilon$. the new belief is:

$$b_i'(0) = \frac{\left(\frac{1}{2} - \epsilon\right)^d}{\left(\frac{1}{2} + \epsilon\right)^d + \left(\frac{1}{2} - \epsilon\right)^d}.$$

The L_1 change of the belief due to the compounded ϵ change in each message is then:

$$\left\|b_i'(0) - b_i(0)\right\|_1 = \frac{1}{2} - \frac{\left(\frac{1}{2} - \epsilon\right)^d}{\left(\frac{1}{2} + \epsilon\right)^d + \left(\frac{1}{2} - \epsilon\right)^d}.$$

A 2nd order Taylor expansion around $\epsilon = 0$ obtains:

$$\left\|b_i'(0) - b_i(0)\right\|_1 \approx d\epsilon + O(\epsilon^3).$$

Therefore, the change in belief varies linearly in the degree of the vertex enabling small ϵ message residuals to translate into large $d\epsilon$ belief residuals.

A.2 Message Residuals May Overestimate Changes in Beliefs

A large $1 - \epsilon$ residual in a single message may result in a small ϵ change in belief at a high degree vertex. For instance, if we consider the set $\{m_1, \ldots, m_d\}$ of binary messages with value $[1 - \epsilon, \epsilon]$ then the resulting belief at that variable would be

$$b_i(0) = \frac{(1-\epsilon)^d}{(1-\epsilon)^d + \epsilon^d}.$$

If we then change m_1 to $[\epsilon, 1 - \epsilon]$ then the resulting belief is

$$b_i'(0) = \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)}.$$

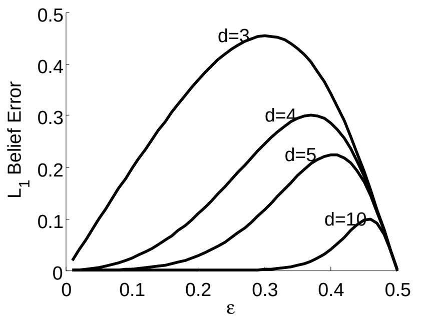
Assuming that $0 < \epsilon \leq \frac{1}{4}$ and $d \geq 3$,

$$\begin{aligned}
\frac{1}{2}\left\|b_i' - b_i\right\|_1 &= \frac{(1-\epsilon)^d}{(1-\epsilon)^d + \epsilon^d} - \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)} \\
&\leq 1 - \frac{(1-\epsilon)^{d-1}\epsilon}{(1-\epsilon)^{d-1}\epsilon + \epsilon^{d-1}(1-\epsilon)} \\
&= 1 - \frac{(1-\epsilon)^{d-2}}{(1-\epsilon)^{d-2} + \epsilon^{d-2}} \\
&= \frac{\epsilon^{d-2}}{(1-\epsilon)^{d-2} + \epsilon^{d-2}}
\end{aligned}$$

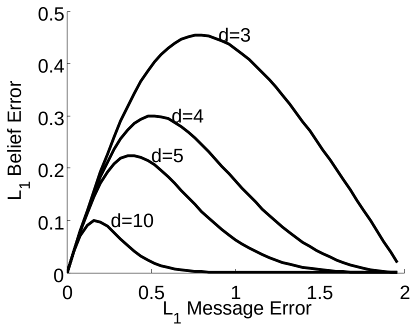
We can bound the denominator to obtain:

$$\frac{1}{2}\left\|b_i' - b_i\right\|_1 \leq \frac{\epsilon^{d-2}}{(1/2)^{d-3}} = (2\epsilon)^{d-3}\epsilon \leq \frac{\epsilon}{2^{d-3}}$$

To demonstrate this graphically, we plot in Fig. 26(a), the L_1 error in the belief $|b_i' - b_i|_1$, varying the value of ϵ. In Fig. 26(b), we plot the L_1 error in the beliefs against the L_1 change in m_1. The curves look nearly identical (but flipped) since the L_1 change is exactly $2 - 4\epsilon$ (recall that m_1 was changed from $[1 - \epsilon, \epsilon]$ to $[\epsilon, 1 - \epsilon]$).



(a) Belief Error against ϵ



(b) Belief Error against Message Error

Figure 26: (a): Given a vertex of degree d with all incoming messages equal to $[1-\epsilon, \epsilon]$. This graph plots the L1 change of belief on a vertex of degree d caused by changing one message to $[\epsilon, 1-\epsilon]$. (b): Similar to (a), but plots on the X-axis the L1 change of the message, which corresponds to exactly $2-4\epsilon$.

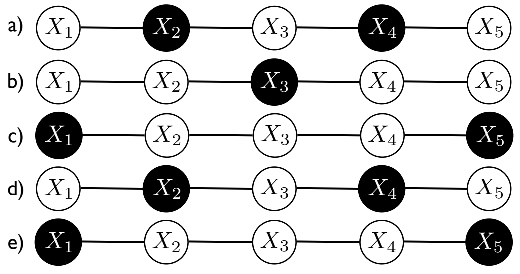
A.3 Failure Case of Naive Belief Residuals

Without loss of generality we consider a chain MRF of 5 vertices with the binary factors $\psi_{X_i,X_{i+1}}(x_i, x_{i+1}) = I[x_i = x_{i+1}]$ and unary factors:

$$\psi_{X_1} = \left[\tfrac{1}{9}, 9\right] \quad \psi_{X_2} = \left[\tfrac{9}{10}, \tfrac{1}{10}\right] \quad \psi_{X_3} = \left[\tfrac{1}{2}, \tfrac{1}{2}\right]$$

$$\psi_{X_4} = \left[\tfrac{1}{10}, \tfrac{9}{10}\right] \quad \psi_{X_5} = \left[9, \tfrac{1}{9}\right]$$

We begin by initalizing all vertex residuals to infinity, and all messages to uniform distributions. Then we perform the following update sequence marked in black:



After stage (b), $m_{2\to3} = \psi_{X_2}$ and $m_{4\to3} = \psi_{X_4}$. Since $b_{X_3} \propto (m_{2\to3} \times m_{4\to3}) \propto [1, 1]$, the belief at X_3 X_3 will have uniform belief. Since X_3 is just updated, it will have a residual of 0.

After stage (d), $m_{2\to3} \propto (\psi_{X_1} \times \psi_{X_2}) \propto \psi_{X_4}$ and $m_{4\to3} \propto (\psi_{X_5} \times \psi_{X_4}) \propto \psi_{X_2}$. These wo messages therefore have swapped values since stage (b).

Since $b_{X_3} \propto (m_{2\to3} \times m_{4\to3})$, the belief at X_3 will not be changed and will therefore continue to have uniform belief and zero residual. At this point X_2 and X_4 also have zero residual since they were just updated. After stage (e), the residuals at X_1 and X_5 at set to 0. However, the residuals on X_2 and X_4 remain zero since messages $m_{1\to2}$ and $m_{5\to4}$ will not change since state (c). All variables therefore now have a residual of 0.

By Eq. (7.3) with $\beta = 0$ we have converged prematurely since no sequence of messages connects X_1 and X_5. The use of the naive belief residual in Eq. (7.3) will therefore converge to an erroneous solution.

Appendix B. Natural Parallelizations of Common Belief Propagation Algorithms

In order to evaluate the parallel Splash algorithm against a strong baseline of competitive parallel belief propagation algorithms, we developed natural parallelizations of several popular sequential belief propagation schedulings. In this section we described both the shared and distributed parallelizations of round-robin, wild-fire, and residual belief propagation.

B.1 Parallel Round-Robin Belief Propagation

The round-robin belief propagation scheduling consists of a fixed ordering σ in which vertices are updated. When possible the ordering σ is constructed using domain knowledge. For our purposes the ordering σ is determined randomly. In the shared-memory version of the parallel round-robin belief propagation algorithm (Alg. 12), p processors simultaneously update blocks of p consecutive vertices from σ. Termination is assessed using the maximum belief residual. In the distributed memory version of parallel round-robin, the over-partitioning procedure described in Sec. 9.1 is first used to partitioning the graph and then a local sequential round-robin is executed separately on each processor. Termination in the distributed setting is assessed again using the belief residuals along with the `TokenRing` procedure described in Sec. 9.3.

Algorithm 12: Shared Memory Parallel Round-Robin Algorithm

```
$\sigma \leftarrow$ Random permutation on $\{1, \dots, |V|\}$
$i \leftarrow 1$
while Not Converged do
    // Update the next $p$ vertices in parallel
    forall $v \in \{\sigma(i), \dots, \sigma(i + p \mod |V|)\}$ do in parallel
        SendMessages($v$)
    $i \leftarrow i + p + 1 \mod |V|$
```

B.2 Parallel Wildfire Belief Propagation

We construct a parallel version of the Wildfire algorithm original introduced by Ranganathan et al. (2007) by augmenting the round-robin algorithm to skip vertices with belief residual below the termination threshold β. The shared-memory and distributed versions of the parallel Wildfire belief propagation algorithm are given in Alg. 14 and Alg. 15 respectively.

B.3 Parallel Residual Belief Propagation

We use a slightly modified version of residual belief propagation original proposed by Elidan et al. (2006). Instead of scheduling messages we schedule vertices using the belief residuals introduced in Sec. 7.2. Therefore, we simply run the parallel and distributed Splash algorithms with the splash

Algorithm 13: Distributed Round-Robin Algorithm

```
// Over-segmentation is used to construct a partitioning
$\mathcal{B} \leftarrow$ over-partitioning of the graph ;
// Enter the distributed phase
forall Processors $b \in \mathcal{B}$ do in parallel
    // Collect the vertices of the factor graph associated with
       this processor.
    Collect($\mathcal{F}_b, \mathcal{X}_b$);
    // Construct a random ordering over vertices in this
       partitioning
    $\sigma \leftarrow$ Random permutation on $b$ ;
    $i \leftarrow 1$ ;
    while TokenRing($\beta$) do
        SendMessages($\sigma(i)$) ;
        $i \leftarrow i + 1 \mod |b|$ ;
        RecvExternalMsgs();
        SendExternalMsgs();
```

Algorithm 14: Shared Memory Parallel Wildfire Algorithm

```
$\sigma \leftarrow$ Random permutation on $\{1, \ldots, |V|\}$
$i \leftarrow 1$
while Not Converged do
    // Update the next $p$ vertices in parallel
    forall $v \in \{\sigma(i), \ldots, \sigma(i + p \mod |V|)\}$ do in parallel
        if Belief Residual of $v$ is greater than $\beta$ then
            SendMessages($v$)
    $i \leftarrow i + p + 1 \mod |V|$
```

size $W = 1$ set to 1. This forces all Splashes to contain only the root and eliminates the spanning tree construction.

Algorithm 15: Distributed Wildfire Algorithm

```
// Over-segmentation is used to construct a partitioning
$\mathcal{B} \leftarrow$ over-partitioning of the graph ;
// Enter the distributed phase
forall Processors $b \in \mathcal{B}$ do in parallel
    // Collect the vertices of the factor graph associated with
       this processor.
    Collect($\mathcal{F}_b, \mathcal{X}_b$);
    // Construct a random ordering over vertices in this
       partitioning
    $\sigma \leftarrow$ Random permutation on $b$ ;
    $i \leftarrow 1$ ;
    while TokenRing($\beta$) do
        if Belief Residual of $v$ is greater than $\beta$ then
            SendMessages($v$) ;
        $i \leftarrow i + 1 \mod |b|$ ;
        RecvExternalMsgs();
        SendExternalMsgs();
```

References

A. Choi V. Gogate A. Darwiche, R. Dechter and L. Otten. Uai'08 workshop: Evaluating and disseminating probabilistic reasoning systems, 2008. http://graphmod.ics.uci.edu/uai08/.

A.Y. Ng A. Saxena, S.H. Chung. 3-d depth reconstruction from a single still image. In *International Journal of Computer Vision (IJCV)*, 2007.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

C.T. Chu, S.K. Kim, Y.A. Lin, Y. Yu, G.R. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

Crupi, Das, and Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *ICPP: 25th International Conference on Parallel Processing*, 1996.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, (1).

P. Domingos. Uw-cse mlns, 2009. URL `alchemy.cs.washington.edu/mlns/uw-cse`.

P. Domingos, S. Kok, D. Lowd, H. F. Poon, M. Richardson, P. Singla, M. Sumner, and J. Wang. Markov logic: A unifying language for structural and statistical pattern recognition. In *SSPR*, 2008.

James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31:1343–1354, 1988.

G. Elidan, I. Mcgraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *UAI*, 2006.

Jinbo Huang, Mark Chavira, and Adnan Darwiche. Solving MAP exactly by searching on compiled arithmetic circuits. In *AAAI*. AAAI Press, 2006.

Alexander Ihler and David McAllester. Particle belief propagation. In D. van Dyk and M. Welling, editors, *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS) 2009*, pages 256–263, Clearwater Beach, Florida, 2009. JMLR: W&CP 5.

A.T. Ihler, J.W. Fischer III, and A.S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *J. Mach. Learn. Res.*, 6, 2005.

G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1), 1998.

S. Z. Li. *Markov random field modeling in computer vision*. Springer-Verlag, London, UK, 1995. ISBN 4-431-70145-1.

J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *SYSTSOFT*.

F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 1987.

R.J. McEliece, D.J.C. MacKay, and J.F. Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. *J-SAC*, 1998.

A. Mendiburu, R. Santana, J.A. Lozano, and E. Bengoetxea. A parallel framework for loopy belief propagation. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, 2007.

J. Misra. Detecting termination of distributed computations using markers. In *SIGOPS*, 1983.

J.M. Mooij and H.J. Kappen. Sufficient conditions for convergence of the Sum-Product algorithm. *ITIT*, 2007.

Ian Parberry. Load sharing with parallel priority queues. *J. Comput. Syst. Sci*, 50(1):64–73, 1995.

J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. 1988.

David M. Pennock. Logarithmic time parallel bayesian inference. In *Proc. 14th Conf. Uncertainty in Artificial Intelligence*, 1998.

Ananth Ranganathan, Michael Kaess, and Frank Dellaert. Loopy sam. In *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2191–2196, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

D. Roth. On the hardness of approximate reasoning. In *ijcai93*, pages 613–618, 1993.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

P. Sanders. Randomized static load balancing for tree-shaped computations. In *Workshop on Parallel Processing*, 1994.

Peter Sanders. Randomized priority queues for fast parallel access. *J. Parallel Distrib. Comput*, 49 (1):86–97, 1998.

Parag Singla and Pedro Domingos. Lifted first-order belief propagation. In Dieter Fox and Carla P. Gomes, editors, *AAAI*. AAAI Press, 2008.

J. Sun, N.N. Zheng, and H.Y. Shum. Stereo matching using belief propagation. *ITPAM*, 2003.

Ben Taskar, Vassil Chatalbashev, and Daphne Koller. Learning associative markov networks. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 102, New York, NY, USA, 2004. ACM. ISBN 1-58113-828-5. doi: http://doi.acm.org/10.1145/1015330.1015444.

Sekhar Tatikonda and Michael I. Jordan. Loopy belief propogation and gibbs measures. In Adnan Darwiche and Nir Friedman, editors, *UAI*, pages 493–500. Morgan Kaufmann, 2002. ISBN 1-55860-897-4.

M. Wainwright, T. Jaakkola, and A.S. Willsky. Tree-based reparameterization for approximate estimation on graphs with cycles. In *NIPS*, 2001.

Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Comput.*, 2000.

Yair Weiss and William T. Freeman. Correctness of belief propagation in gaussian graphical models of arbitrary topology. *Neural Computation*, 13(10):2173–2200, 2001.

Yinglong Xia and Viktor K. Prasanna. Junction tree decomposition for parallel exact inference. In *IPDPS*, pages 1–12. IEEE, 2008. URL `http://dx.doi.org/10.1109/IPDPS.2008.4536315`.

C. Yanover and Y. Weiss. Approximate inference and protein folding. In *NIPS*, 2002.

C. Yanover, O. Schueler-Furman, and Y. Weiss. Minimizing and learning energy functions for side-chain prediction. *J Comput Biol*.

J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In *Exploring artificial intelligence in the new millennium*, 2003.