Residual Splash for Optimally Parallelizing Belief Propagation

Joseph E. Gonzalez
Carnegie Mellon University

Yucheng Low
Carnegie Mellon University

Carlos Guestrin
Carnegie Mellon University

Abstract

As computer architectures move towards multicore we must build a theoretical understanding of parallelism in machine learning. In this paper we focus on parallel inference in graphical models. We demonstrate that the natural, fully 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 two 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.

1 INTRODUCTION

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 continue to fuel exponential growth in the size of machine learning tasks. To ensure that graphical model based techniques continue to scale with hardware improvements and match the growth in problem size, we must develop a theoretical understanding of parallelism in graphical models and design new algorithms to leverage that parallelism.

Efficient inference is critical to the effective application of graphical models to large scale real world tasks. Research by [Nallapati et al., 2007, Newman et al., 2008] in statistical text clustering with Latent Dirichlet Allocation (LDA) has produced sophisticated parallel variational inference methods for latent topic models. However, this work does not directly address general inference. Alternatively, [Pennock, 1998] provide a parallel exact inference algorithm and strong asymptotic analysis in settings where many processors are available and exact inference is tractable. However, this work does not consider approximate inference, which is typically necessary for large complex models. Finally, work by [Chu et al., 2006] provides more general insight into the parallelism afforded by the Statistical Query Model (SQM) of [Kearns, 1998]. However, the SQM is already embarrassingly parallel, i.e. having completely independent computational components, and does not efficiently represent many challenging machine learning tasks. We will show that the embarrassingly parallel form of belief propagation, using fully synchronous updates, is actually asymptotically inefficient in the parallel setting.

In this work, we focus on the parallelism exposed by inference algorithms which can be represented by passing messages along edges in a graph. Belief propagation (BP), a popular message passing algorithm, is considered to be naturally parallel and recent work by [Casado et al., 2007, Mendiburu et al., 2007] give basic parallel implementations of BP but provide no theoretical insight into their performance. We develop a theoretical understanding of the limiting sequential component of BP and its relationship to the graph structure, factors, and desired approximation accuracy.

Using our theoretical foundation, we develop a new parallel approximate inference algorithm, ResidualSplash, which performs asymptotically better than embarrassingly parallel BP and achieves the optimal running time in chain graphical models. We evaluate the performance of our algorithm on two challenging real-world tasks. More specifically, our key contributions are:

- A natural Map-Reduce based embarrassingly parallel BP algorithm and an analysis of its parallel scaling.
- The τ_ϵ-approximation for characterizing approximate BP inference, a formal analysis of the parallelism exposed by τ_ϵ-approximate inference, and a lower bound on the theoretically achievable parallel running time for chain graphical models.
- A new parallel approximate inference algorithm, ResidualSplash, which performs optimally on chain

Appearing in Proceedings of the 12^{th} International Confe-rence on Artificial Intelligence and Statistics (AISTATS) 2009, Clearwater Beach, Florida, USA. Volume 5 of JMLR: W&CP 5. Copyright 2009 by the authors.

graphical models and extends to arbitrary cyclic models using an efficient dynamic scheduling.

- An empirical evaluation of our new algorithm on two types of cyclic graphical models, demonstrating that it outperforms other proposed techniques.

2 BELIEF PROPAGATION

Graphical models provide compact representations of large probability distributions. Without loss of generality, we consider only pairwise Markov Random Fields (MRFs), as any graphical model can be transformed into a pairwise MRF (cf. [Koller and Friedman]). Consider the set of n discrete random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ each taking on values $X_i \in A_i$. A pairwise MRF is an undirected graph $G = (V, E)$ where the vertices V correspond to the random variables and the edges E correspond to dependencies between variables encoded by functions (factors) $\{\psi_{i,j} : A_i \times A_j \to \mathbb{R}^+ \mid \{i,j\} \in E\}$. Additionally, each vertex has a corresponding node function, $\{\psi_i : A_i \to \mathbb{R}^+ \mid i \in V\}$. The joint distribution over $\mathcal{X}$ is

$$\mathbf{P}(x_1, \ldots, x_N) \propto \prod_{i \in V} \psi_i(x_i) \prod_{\{i,j\} \in E} \psi_{i,j}(x_i, x_j).$$

The node marginals of graphical models are central to learning and inference. While computing exact marginals is NP-hard in general, [Pearl, 1988] proved that the BP may be used to efficiently compute the exact marginals in acyclic graphical models.

Node marginals are computed in BP by iteratively "sending" (computing) messages in both directions along edges in the graph. The message sent from vertex i to vertex j along $\{i,j\} \in E$ is the function

$$m_{i \to j}(x_j) \propto \sum_{x_i \in A_i} \psi_{i,j}(x_i, x_j) \psi_i(x_i) \prod_{k \in \Gamma_i \setminus j} m_{k \to i}(x_i),$$

which encodes the "belief" variable X_i has about the value of X_j. Here, $\Gamma_i \subseteq V$ is the set of neighbors of vertex i. Messages are typically initialized to uniform distributions and normalized to ensure numerical stability.

We can represent each individual message as a vector in the vector space $(\mathbb{R}^+)^{|A_j|}$ and all messages jointly as a vector in $\mathcal{M} = \otimes_{i=1}^{n} A_i$, the cartesian product. For $m \in \mathcal{M}$ denote i^{th} message by m_i. The BP algorithm can then be expressed as the iterative application of some function $f_{\text{BP}} : \mathcal{M} \to \mathcal{M}$ such that $m^{(t)} = f_{\text{BP}}(m^{(t-1)})$ and $m^{(t)}, m^{(t-1)} \in \mathcal{M}$. We define $m^* \in \mathcal{M}$ as a fixed-point $m^* = f_{\text{BP}}(m^*)$.

In **synchronous BP**, all vertices *simultaneously* compute their outbound messages at every iteration using the messages from the previous iteration, i.e., $m^{(t)} = (f_{\text{BP}}(m^{(t-1)})_1, \ldots, f_{\text{BP}}(m^{(t-1)})_{2|E|})$. In **asynchronous BP**, messages are updated *sequentially* using the most recent messages, i.e., $m^{(t)} = (m_1^{(t-1)}, \ldots, f_{\text{BP}}(m^{(t-1)})_k, \ldots m_{2|E|}^{(t-1)})$, for some message m_k. In both synchronous and asynchronous BP, messages are sent until some convergence criterion is reached. For a small constant $\beta \geq 0$, we use the convergence criterion

$$\max_{i,j \in V} \left| \left| m_{i \to j}^{(\text{new})}(x_j) - m_{i \to j}^{(\text{old})}(x_j) \right| \right|_1 \leq \beta. \qquad (2.1)$$

The estimates of the marginal distributions are then

$$\mathbf{P}(X_i = x_i) \approx b_i(x_i) \propto \psi_i(x_i) \prod_{k \in \Gamma_i} m_{k \to i}(x_i).$$

While BP is guaranteed to converge to the exact marginals in acyclic graphs, there are few guarantees for convergence or correctness in general graphs. Nonetheless, BP on "loopy" graphs (often referred to as Loopy BP) 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, Yanover et al., 2007].

We introduce the concept of **awareness** to capture the underlying *sequential* structure of BP inference. Intuitively, awareness captures the "flow" of message information along edges in a 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 2.1 (Awareness). *Vertex j is **aware** of vertex i if there exists a chain of connected edges $\{\{i, v_1\}, \{v_1, v_2\}, \ldots, \{v_m, j\}\} \subseteq E$, and a sequence of messages $[m_{i \to v_1}^{(t_1)}, m_{v_1 \to v_2}^{(t_2)}, \ldots, m_{v_m \to j}^{(t_m)}]$ such that each message was computed using the previous message in the sequence $t_1 < \ldots < t_m$.*

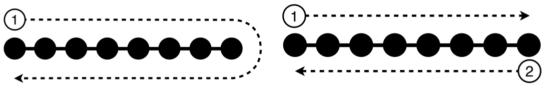
The definition of awareness leads to a few useful properties. On the k^{th} iteration of synchronous BP, every vertex is **aware** of all reachable vertices at a distance k or less. In an acyclic MRF, if a vertex is aware of all reachable vertices, then its current belief is exact. Consequently, synchronous BP on an acyclic MRF of diameter d converges to the exact marginals in d iterations. In the next section we use awareness to expose inefficiency in synchronous belief propagation and identify the parallel limits.

3 PARALLEL SYNCHRONOUS BP

Belief propagation presents several opportunities for parallelism. The individual message sums and products can be expressed as parallel matrix operations. However, for the typical message sizes $|A_i| << n$ graph level parallelism between message updates is likely to be asymptotically more beneficial. As a consequence, we define running time in terms of the number of message computations, treating individual message updates as atomic unit time operations.

Algorithm 1: Multicore `MapReduceBP` Algorithm

```
  while Not Converged do
1     Swap($m^{(new)}$,$m^{(old)}$);
      forall $i \to j \in E$ do in parallel
2         $b_{i\backslash j}(x_i) \leftarrow \psi_i(x_i) \prod_{k \in \Gamma_i \backslash \{j\}} m^{\text{old}}_{k \to i}(x_i)$ ;
3         $m^{\text{new}}_{i \to j}(x_j) \leftarrow \sum_{x_i \in A_i} \psi_{i,j}(x_i, x_j) b_{i\backslash j}(x_i)$;
      // Finished an Iteration
```



(a) Single Sequential (b) Optimal Parallel

Figure 1: **(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.

The parallel nature of the synchronous message updates suggests that BP is an embarrassingly parallel algorithm. Specifically, given the messages from the previous iteration, each new message can be computed completely independently and in any order. Additionally, message updates are idempotent; repeated calculations of the new messages will not change the result.

Map-Reduce, introduced by [Dean and Ghemawat, 2008], is a popular framework for encoding embarrassingly parallel algorithms that are associative and idempotent. In the Map-Reduce framework, an algorithm is specified by a `Map` operation, which transforms the input, and a `Reduce` operation, which combines the results of the map operation.

We can naturally encode synchronous BP as an iterative Map-Reduce algorithm (Alg. 1) where the `Map` operation is applied to all vertices and emits destination-message key-value pairs and the `Reduce` operation joins messages at their destination vertex. In a parallel shared memory setting the `Reduce` operation is accomplished by swapping old and new message sets.

The messages are updated in parallel (Line 2 and Line 3 of Alg. 1) without locking, and all updated messages are written directly to shared memory. Only Line 1 of Alg. 1 requires synchronization to ensure all processors have consistent new and old message sets.

While MapReduceBP may appear to be an ideal parallel algorithm, we show that it can perform asymptotically slower than running asynchronous BP on a single processor. We begin by bounding the parallel running time of MapReduceBP.

Theorem 3.1 (Map-Reduce Exact Inference in Trees). *Given an acyclic MRF with n vertices, diameter d, and $p \leq 2(n-1)$ processors (Mappers), MapReduceBP will compute exact marginals in time $\Theta(\max(nd/p, d))$.*

Proof of Theorem 3.1. Each iteration of MapReduceBP, splits $2(n-1)$ message updates over p processors and therefore takes $\lceil 2(n-1)/p \rceil \leq 2(n-1)/p + 1$ time to complete. Because MapReduceBP is a strict parallelization of synchronous BP, it will converge in d iterations. □

To illustrate how poorly MapReduceBP performs, we analyze the running time on a chain graphical model with n variables. We use chain graphical models as a theoretical benchmark for parallel BP because they directly capture the limiting sequential structure of awareness and are a subproblem in both acyclic and cyclic graphical models.

The length of the chain and minimum running time, $n-1$, is achieved using $p = 2(n-1)$ processors. However, the optimal single processor asynchronous scheduling for a chain graph (Fig. 1(a)) is achieved by passing messages forward $(m_{1\to2}, \ldots, m_{n-1\to n})$ and then backward $(m_{n\to n-1}, \ldots, m_{2\to1})$, with total running time $2(n-1)$. Therefore we only reduce the running time by half using $p = 2(n-1)$ processors (2 processors per edge)!

Surprisingly, if we use fewer than $p = n-1$ processors, the parallel MapReduceBP will perform *slower* than the asynchronous algorithm using only a single processor. If we use any constant number of processors ($p = 2$) in parallel, than the MapReduceBP algorithm will run in *quadratic* time while the sequential single processor version will run in *linear* time. Alternatively, using two processor ($p = 2$) and computing the forward and backward asynchronous message schedule in parallel (Fig. 1(b)), we achieve the same running time $n-1$ as MapReduceBP using $p = 2(n-1)$.

While messages may be computed in any order, **awareness** is propagated *sequentially*. On every iteration of MapReduceBP only a few message computations increase awareness while the rest are wasted. 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.

4 τ_ϵ-APPROXIMATE INFERENCE

It was shown by [Ihler et al., 2005] that message errors decay along paths. Intuitively, for a long chain graph with weak edge potentials, distant vertices are approximately independent. For a particular vertex, an accurate approximation may often be achieved by considering only the small subgraph around that vertex. By limiting vertex awareness to its local vicinity, we can reduce the sequential component of BP to the longest path in the subgraph.

For a given graphical model we define, in Definition 4.1, the size of the largest subgraph required to achieve a given approximation accuracy. Intuitively, τ_ϵ is the earliest iteration of MapReduceBP at which every message is less than ϵ away from its value at convergence.

Definition 4.1 (τ_ϵ-Approximation)**.** *Given an acyclic*[1] *MRF, 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 first* aware *of all vertices within a distance of τ. For a given error, ϵ, we define a τ_ϵ-Approximation as the smallest τ such that for the fixed point $m^* = f_{BP}(m^*)$*

$$\max_{\{u,v\}\in E} \left|\left|\tilde{m}^{(\tau)}_{u\to v} - m^*_{u\to v}\right|\right|_1 \leq \epsilon \qquad (4.1)$$

Under the condition that f_{BP} is a contraction mapping[2] we can directly relate the τ_ϵ in Definition 4.1 to the actual BP termination condition. If f_{BP} is a max-norm contraction mapping then for the fixed point m^* and $0 \leq \alpha < 1$,

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

Here the L_∞-norm is defined in terms of the maximum of the individual message norms,

$$\left|\left|m^{(i)} - m^{(j)}\right|\right|_\infty = \max_{\{u,v\}\in E} \left|\left|m^{(i)}_{u\to v} - m^{(j)}_{u\to v}\right|\right|_1$$

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

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

Finally, in Eq. (4.2) we observe that the convergence criterion, $||m - f(m)||_\infty$ defined in Eq. (2.1), 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 $\beta \leq \epsilon(1 - \alpha)$.

$$\begin{aligned}
||m - m^*||_\infty &= ||m - f_{\text{BP}}(m) + f_{\text{BP}}(m) - m^*||_\infty \\
&\leq ||m - f_{\text{BP}}(m)||_\infty + ||f_{\text{BP}}(m) - m^*||_\infty \\
&\leq ||m - f_{\text{BP}}(m)||_\infty + \alpha \, ||m - m^*||_\infty \\
||m - m^*||_\infty &\leq \frac{1}{1-\alpha} \, ||m - f_{\text{BP}}(m)||_\infty \qquad (4.2)
\end{aligned}$$

It is important to note that in practice α is likely to be unknown or the MRF may not satisfy the contraction mapping conditions. 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.

Returning to MapReduceBP, we see that the running time improves as we replace d, the diameter of the graph, with τ_ϵ to achieve the desired τ_ϵ approximation.

Theorem 4.1 (τ_ϵ-Approximation MapReduce Running Time)**.** *Given an acyclic MRF with n vertices a τ_ϵ-approximation is obtained by running MapReduceBP with p processors ($p \leq n$) in running time $\Theta\left(n\tau_\epsilon/p\right)$.*

Proof of Theorem 4.1. We know that each iteration will take $\lceil 2(n-1)/p \rceil$ time to complete. From Definition 4.1 it trivially follows that a τ_ϵ-approximation is obtained in exactly τ_ϵ iterations. □

However, even with this reduction in runtime, we show that on a simple chain graph, the performance of MapReduceBP is still far from optimal.

Theorem 4.2 (τ_ϵ-Approximate Parallel Lower Bound)**.** *For an arbitrary chain graph with n vertices and p processors, a τ_ϵ-approximation cannot in general be computed with a running time less than $\Omega\left(n/p + \tau_\epsilon\right)$.*

Proof of Theorem 4.2. 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 $X_{i-\tau_\epsilon}$).

Let each processor compute a set of $k \geq \tau_\epsilon$ message updates in sequence (e.g., $[m^{(1)}_{1\to 2}, m^{(2)}_{2\to 3}, \ldots, m^{(k)}_{k-1\to k}]$). Because after the first τ_ϵ updates all additional updates make exactly one more vertex left-aware, 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 obtain the following inequality:

$$\begin{aligned}
n - \tau_\epsilon &\leq \frac{p}{2}\left(k + \tau_\epsilon - 1\right) \\
k &\geq \frac{2n}{p} + \tau_\epsilon\left(1 - \frac{2}{p}\right) - 1 \qquad (4.3)
\end{aligned}$$

relating required amount of work and the maximum amount of work done on the k^{th} iteration. For $p > 2$, Eq. (4.3) provides the desired asymptotic result. □

We observe from Theorem 4.1, that MapReduceBP 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 MapReduceBP 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 optimal bound.

5 THE RESIDUALSPLASH ALGORITHM

We now give a general parallel inference algorithm that achieves the lower bound for chain graphical models and generalizes to arbitrary cyclic graphical models. Our optimal algorithm is built around the Splash operation (Alg. 2)

[1] It is possible to extend Definition 4.1 to arbitrary cyclic graphs by considering the possibly unbounded computation tree described by Weiss [2000].

[2] Mooij and Kappen [2007] provide sufficient conditions for $f_{\text{BP}}(m)$ to be a contraction mapping under a variety of norms including a variation of the max-morm.

Algorithm 2: Splash (v, h)

```
input: vertex $v$
bfs_order ← ConstructBFSOrdering($v$,$h$)
// Make Root Aware of Leaves
foreach $i \in$ ReverseOrder(bfs_order) do
    SendMessages($i$)
// Make Leaves Aware of Root
foreach $i \in$ bfs_order do
    SendMessages($i$)
```

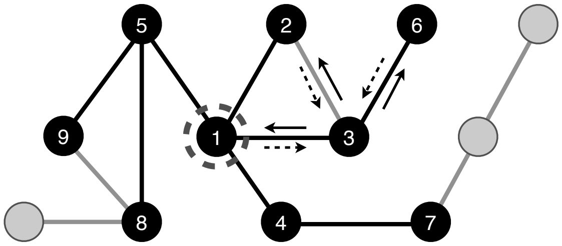


Figure 2: A splash of size $h = 2$ starting on vertex 1, encircled by a dotted ring. The dark vertices and edges represent the BFS tree rooted at vertex 1. The SendMessages operation is invoked on the sequence of vertices $(9, 8, \ldots, 2, 1, 2, \ldots, 8, 9)$. In this figure SendMessages is being applied to vertex 3 where the dotted arrows represent the received messages and the solid arrows represent the newly updated messages.

Algorithm 3: ResidualSplash Algorithm

```
input: Constants $h, \beta$
$Q \leftarrow$ InitializeQueue($Q$)
Set All Residuals to $\infty$
forall processors do in parallel
    while TopResidual(Q) $> \beta$ do
        $v \leftarrow$ Pop($Q$)
        Splash($v$, $h$)
        foreach $v \in$ Vertices Affected By the Splash
        do
            Update($Q$, Residual($v$))
        Push($Q$, ($v$, Residual($v$)))
```

which is a natural generalization of the optimal sequential ordering described in Fig. 1(a) to arbitrary cyclic graphs described in Fig. 2.

The Splash operation grows a shallow breadth first search tree of height h around a vertex v using ConstructBFSOrdering(v, h) and then sequentially sends messages from the leaves to the root and then back to the leaves. The function SendMessages(i) updates all outbound messages from vertex i using the most recent inbound messages. Each individual message update is done atomically with a unique read-write lock for each message.

To schedule Splash operations, we extend the residual heuristic introduced by Elidan et al. [2006] which prioritizes message updates based on their residual, greedily optimizing the convergence criterion given in Eq. (2.1). 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 proposed by Elidan et al. [2006] updates the message with highest residual first and then updates the dependent message residuals by temporarily computing their new values.

Here we define a scheduling over vertices and not messages. The priority (residual) of a vertex is the maximum of the residuals of the incoming messages.

$$r_u = \sup_{i \in \Gamma_u} \left|\left|m^{\text{next}}_{i\to u} - m^{\text{last}}_{i\to u}\right|\right|_1 \qquad (5.1)$$

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 neighboring vertices residuals are updated. Vertex scheduling has the advantage over message residuals of using the most recent information when updating a message.

Using the vertex residual defined in Eq. (5.1) as a scheduling priority, we give the parallel ResidualSplash algorithm (Alg. 3) which maintains a shared residual priority queue over vertices. We initialize the queue by setting the priority of all vertices to an arbitrary value greater than the maximum vertex residual. This ensures that each vertex is updated at least once before convergence. Then, each processor removes the top vertex from the queue and applies the Splash operation.

We now show that, in expectation, the ResidualSplash algorithm achieves the optimal running time from Theorem 4.2 for chain graphical models. We begin by relating the Splash operation to the vertex residuals.

Lemma 5.1 (Splash Residuals)**.** *Immediately after the Splash operation is applied to an acyclic graph all vertices interior to the Splash have zero residual.*

Proof of Lemma 5.1. 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, 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 5.2.

Lemma 5.2 (Basic Convergence)**.** *Given a chain graph where only one vertex has nonzero residual, the ResidualSplash algorithm with Splash size h will run in $O(\tau_\epsilon + h)$ 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 5.1, 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 h vertices. □

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

Theorem 5.3 (Splash Chain Optimality)**.** *Given a chain graph with n vertices and $p \leq n$ processors we can apply the ResidualSplash algorithm with the Splash size set to $h = n/p$ and uniformly spaced initial Splashes to obtain a τ_ϵ-approximation in expected running time $O\left(n/p + \tau_\epsilon\right)$.*

Proof of Theorem 5.3. 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 start on these vertices. Since the height of each splash is also $h = n/p$, 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 5.1). Therefore there are at most $O\left(p\right)$ 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 5.1, we see that the total number of updates needed including the initial $O\left(p\right)$ Splash operations is $O\left(p(\tau_\epsilon + h)\right) + O\left(n\right) = O\left(n + p\tau_\epsilon\right)$. Since work is evenly distributed, each processor performs $O\left(n/p + \tau_\epsilon\right)$ 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 5.4 (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 $O\left(\log(p)(n/p + \tau_\epsilon)\right)$.*

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[3] by the coupon collectors problem. Therefore, at most $O\left(p \log(p)\right)$ Splash operations (rather than the p Splash operations used in Theorem 5.3) are required in expectation to update each vertex at least once. Using the same method as in Theorem 5.3, we observe that the running time is $O\left(\log(p)(n/p + \tau_\epsilon)\right)$. □

In practice, by augmenting the Splash operation to prune branches of the BFS when vertices with residuals less than the termination threshold are reached, we were able to further improve performance. We then set τ to be relatively large and allow the pruning heuristic to regulate the sizes of each Splash. We note that Theorem 5.3 still holds under this heuristic.

5.1 MEMORY LOCKS AND SYNCHRONIZATION

All processors in the ResidualSplash algorithm share a single message set and priority queue. Separate synchronization locks for each message ensure exclusive read and exclusive write message operations. Because there are often many more messages than processors, message contention is rare. Additionally, we minimize the impact of synchronization on the shared priority queue by representing it internally as a collection of priority queues with separate locks for each queue.

5.2 MEMORY EFFICIENCY

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 the ResidualSplash algorithm reduces cache misses over the basic residual BP algorithm (ResidualSplash with $h = 1$).

6 EXPERIMENTS

Protein side chain prediction is an important sub-task of the protein folding problem which can be framed as finding the energy minimizing joint assignment to a pairwise MRF. We used 276 protein side chain models provided by [Yanover et al., 2007]. The models vary in complexity with up to 700 variables and angle discretizations ranging from 2 to 79. Due to the high dynamic range in the node and edge potentials, log-space message calculations were required. In addition, message damping ($\alpha = 0.4$) was used to improve convergence. We obtained the true angles used in [Yanover et al., 2007] to assess the quality of our MAP estimates and we report the prediction accuracy. The protein MRFs test parallel inference techniques in settings where variables are

[3] In practice, are removed between parallel round, resulting in fewer collection steps then the coupon collectors problem.

highly connected and share strong interactions. The volume of each Splash was initially set to be the size of the graph and using the pruning heuristic quickly diminished.

To evaluate ResidualSplash on large regular graphical models, we created the popup video task in which we extended the monocular single image depth reconstruction work of [A. Saxena, 2007] to videos. We constructed a three-dimensional grid-graph representing the depth of each pixel and imposing continuity within frames *and* across time. We used Laplacian inter-frame potentials with the scale parameter determined experimentally. We discretized the depths to 40, 80, and 120 levels to test the effect of message sizes on parallel performance. In our experiments, we used 50 frames and sliced each frame into a 107 × 86 pixel grid resulting in a regular cube structured MRF with $50 \times 107 \times 86 = 460100$ vertices. The popup video task tests inference in the setting where connectivity is relatively low and variable count is relatively high. The volume of each Splash was initially set to of 1000.

6.1 IMPLEMENTATION

We implemented optimized versions of ResidualSplash, MapReduceBP and several other base line algorithms in C++ using PThreads. We have publicly released the code as well as Matlab wrappers at [SelectLab, 2009]. To ensure a fair comparison, all three algorithms used the same convergence, update, and support code and only differed in what order messages were updated or whether they were updated synchronously. For all experiments, the Sum-Product variant of belief propagation was used and the termination condition was set with $\epsilon = 10^{-5}$. Experiments were compiled using GCC 4.3.2 and tested using 64Bit Linux with dual Quad-Core AMD Opteron 2.7GHz (2384) processors. Each quad cores has a shared 6MB L3 cache. Profiling experiments were performed with the OProfile profiler.

6.2 RESULTS

In Fig. 3(a,b) and Fig. 4(a,b) we compare the running time, in seconds, and corresponding speedup of the ResidualSplash, Residual BP, and MapReduceBP algorithms on both tasks. The speedup represents the ratio of the p processor running time, of each algorithm over the running times of the fastest single processor algorithm.

In both tasks the ResidualSplash algorithm significantly outperformes the MapReduceBP algorithm, obtaining a single core running time only slightly higher that the 8 core running time for MapReduce. Furthermore in both tasks the ResidualSplash algorithm achieved near-linear to super-linear performance scaling. When compared against the Residual algorithm, ResidualSplash scaled comparably on the protein side-chain task but significantly outperformed Residual on the popup-video task. The popup-video MRF is less tightly connected resulting in a smaller τ_ϵ relative to the diameter leading to an improved ResidualSplash performance. Furthermore the greater connectivity and state size of the protein side-chain prediction task increases running time of the `SendMessages` operation improving cache efficiency and reducing bus contention for both algorithms.

We evaluated the role of bus contention by varying the discretization in the popup video task. While increasing the discretization has a linear effect on the overall amount of memory required, it has a quadratic effect on the work done in the message updates. In Fig. 3(d) we see that increasing the discretization decrease the cache miss rate implying that the cache is used more efficiently. Finally, Fig. 3(c) demonstrates that increasing the discretization leads to an increase in the parallel performance suggesting that the popup video task is bus limited.

In Fig. 4(c) we report the accuracy of the MAP estimates for the protein sidechain prediction task using all three algorithms. The prediction scores in Fig. 4(c) are comparable to those obtained by [Yanover et al., 2007], and do not fluctuate significantly as a function of the number of cores.

7 CONCLUSION

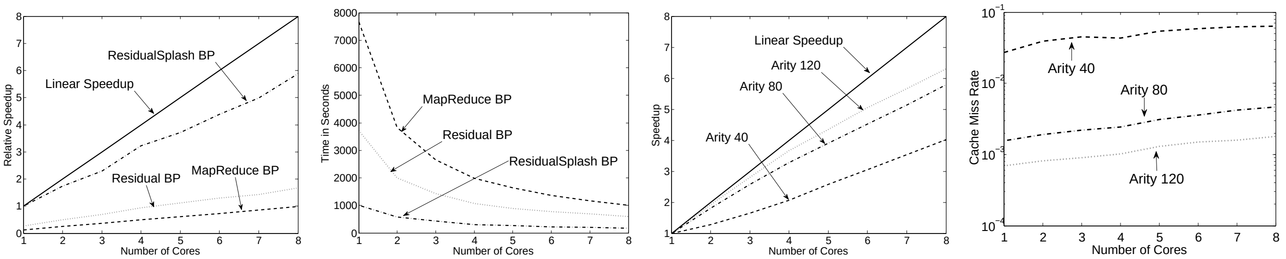
In this paper, we considered the problem of exploiting parallelism in belief propagation (BP). We showed that the natural synchronous parallelization of BP, MapReduceBP, performs asymptotically worse (quadratic in the number of messages) than a single processor sequential implementation (linear in the number of messages) on chain models.

We introduced the concept of τ_ϵ-approximate inference in acyclic graphical models and showed that the available parallelism depends on the graph structure, potentials, and desired level of approximation. In the τ_ϵ-approximate setting, we derived a lower bound on the parallel running time of message passing inference in chain graphical models and showed that the MapReduceBP algorithm does not achieve this lower bound.

We then introduced the ResidualSplash algorithm which achieves the lower bound and generalizes to arbitrary pairwise Markov Random Fields. Using the protein side chain prediction task and a novel popup video task, we tested ResidualSplash and found that it outperforms MapReduceBP both in running time and parallel speedup.

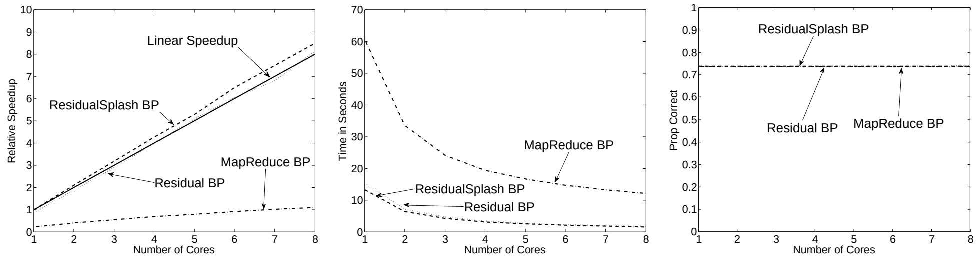
7.1 ACKNOWLEDGEMENTS

This work is supported by ONR Young Investigator Program grant N00014-08-1-0752, the ARO under MURI W911NF0810242, the NSF under grants NeTS-NOSS and CNS-0625518 and Joseph Gonzalez was also supported by the AT&T Labs Fellowship. We would like to thank David O‘Hallaron and Jason Campbell for their guidance and Intel Research for cluster time.



(a) Video Speedup (b) Video Running Time (c) Video Speedup by $|A_i|$ (d) Video Cache Miss

Figure 3: **(a)** Speedup relative to the fastest single core algorithm on popup video task. **(b)** Running times of all algorithms on the popup video task. **(c)** Self speedup attained with different levels of depth discretization (i.e., $|A_i| \in \{40, 80, 120\}$). **(d)** Average cache miss rate plotted on a logarithmic axis for the popup video task.



(a) Protein Speedup (b) Protein Running Time (c) Protein Score

Figure 4: **(a)** Xpeedup relative to the fastest single core algorithm on protein task. **(b)** Running times of all algorithms on protein task. **(c)** Average accuracy of MAP estimates on the protein task.

References

R. Nallapati, W. Cohen, and J. Lafferty. Parallelized variational EM for latent Dirichlet allocation: An experimental evaluation of speed and scalability. In *ICDMW '07: Proceedings of the Seventh IEEE International Conference on Data Mining Workshops*, pages 349–354, 2007.

D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent dirichlet allocation. In *NIPS*, pages 1081–1088. 2008.

D. M. Pennock. Logarithmic time parallel bayesian inference. In *Proc. 14th Conf. Uncertainty in Artificial Intelligence*, pages 431–438. Morgan Kaufmann, 1998.

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

M. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, 1998.

A. I. Vila Casado, M. Griot, and R.D. Wesel. Informed dynamic scheduling for belief-propagation decoding of LDPC codes. *CoRR*, abs/cs/0702111, 2007.

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*, pages 2843–2850, 2007.

D. Koller and N. Friedman. Probabilistic graphical models.

J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. 1988. ISBN 0-934613-73-7.

R.J. McEliece, D.J.C. MacKay, and J.F. Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. *Selected Areas in Communications, IEEE Journal on*, 16(2):140–152, Feb 1998.

J. Sun, N.N. Zheng, and H.Y. Shum. Stereo matching using belief propagation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(7):787–800, July 2003.

J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. pages 239–269, 2003.

C. Yanover and Y. Weiss. Approximate inference and protein folding. In *NIPS*, pages 84–86. MIT Press, 2002.

C. Yanover, O. Schueler-Furman, and Y. Weiss. Minimizing and learning energy functions for side-chain prediction. pages 381–395. 2007.

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

A.T. Ihler, J.W. Fischer III, and A.S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *J. Mach. Learn. Res.*, 6:905–936, 2005.

Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Comput.*, 12(1):1–41, 2000.

J.M. Mooij and H.J. Kappen. Sufficient conditions for convergence of the Sum-Product algorithm. *Information Theory, IEEE Transactions on*, 53(12):4422–4437, Dec. 2007.

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

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.

SelectLab. ResidualSplash Pairwise MRF code, 2009. URL `http://www.select.cs.cmu.edu/code`.