Relaxed Scheduling for Scalable Belief Propagation

Abstract

The ability to leverage large-scale hardware parallelism has been one of the key enablers of the accelerated recent progress in machine learning. Consequently, there has been considerable effort invested into developing efficient parallel variants of classic machine learning algorithms. However, despite the wealth of knowledge on parallelization, some classic machine learning algorithms often prove hard to parallelize efficiently while maintaining convergence.

In this paper, we focus on efficient parallel algorithms for the key machine learning task of inference on graphical models, in particular on the fundamental belief propagation algorithm. We address the challenge of efficiently parallelizing this classic paradigm by showing how to leverage scalable relaxed schedulers in this context. We present an extensive empirical study, showing that our approach outperforms previous parallel belief propagation implementations both in terms of scalability and in terms of wall-clock convergence time, on a range of practical applications.

1 Introduction

Hardware parallelism has been a key computational enabler for recent advances in machine learning, as it provides a way to reduce the processing time for the ever-increasing quantities of data required for training accurate models. Consequently, there has been considerable effort invested into developing efficient parallel variants of classic machine learning algorithms, e.g. [28, 22, 23, 24, 16].

In this paper, we will focus on efficient parallel algorithms for the fundamental task of *inference on graphical models*. The inference task in graphical models takes the form of *marginalisation*: we are given observations for a subset of the random variables, and the task is to compute the conditional distribution of one or a few variables of interest. The marginalization problem is known to be computationally intractable in general [10, 33, 9], but inexact heuristics are well-studied for practical inference tasks.

One popular heuristic for inference on graphical models is *belief propagation* [27], inspired by the exact dynamic programming algorithm for marginalization on trees. While belief propagation has no general approximation or even convergence guarantees, it has proven empirically successful in inference tasks, in particular in the context of decoding low-density parity check codes [8]. However, it remains poorly understood how to properly parallelize belief propagation.

Parallelizing belief Propagation. To illustrate the challenges of parallelizing belief propagation, we will next give a simplified overview of the belief propagation algorithm, and refer the reader to Section 2 for full details. Belief propagation can be seen as a *message passing* or a *weight update* algorithm. In brief, belief propagation operates over the underlying graph $G = (V, E)$ of the graphical model, maintaining a vector of real numbers called a *message* $\mu_{i\to j}$ for each ordered pair (i, j) corresponding to an edge $\{i, j\} \in E$ (Fig. 1). The core of the algorithm is the *message update rule* which specifies how to update an outgoing message $\mu_{i\to j}$ at node i based on the *other* incoming messages at node i; for the purposes of the present discussion, it is sufficient to view this as black box function f over these other messages, leading to the update rule

$$\mu_{i\to j} \leftarrow f\big(\{\mu_{k\to i} \colon k \in N(i) \setminus \{j\}\}\big)\,. \tag{1}$$

This update rule is applied to messages until the values of messages have converged to a stable solution, at which point the algorithm is said to have terminated.

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

Importantly, the message update rule does not specify *in which order* messages should be updated. The standard solution, called *synchronous belief propagation*, is to update all the message simultaneously. That is, in each global round $t = 1, 2, 3, \ldots$, given message values $\mu^t_{i\to j}$ for all pairs (i, j), the new values $\mu^{t+1}_{i\to j}$ are computed as

$$\mu^{t+1}_{i\to j} \leftarrow f\big(\{\mu^t_{k\to i} \colon k \in N(i) \setminus \{j\}\}\big)$$

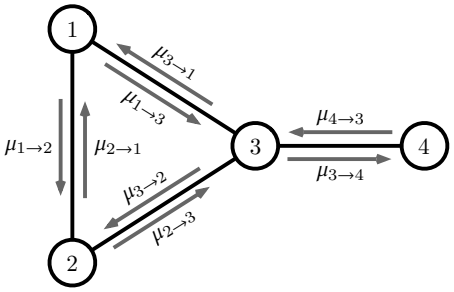


Figure 1: State of the belief propagation algorithm consist of two directed messages for each edge.

However, there is evidence that updating messages *one at a time* leads to faster and more reliable convergence [14]; in particular, various proposed *priority-based schedules*—schedules that try to prioritize message updates that would make 'more progress'—have proven empirically to converge with much fewer message updates than the synchronous schedule [14, 20, 38].

Having to execute updates in a strict priority order poses a challenge for efficient *parallel* implementations of belief propagation: while the synchronous schedule is naturally parallelizable, as all message updates can be done independently, the more efficient priority-based schedules are inherently sequential and thus seem difficult to parallelize. Accordingly, existing work on efficient parallel belief propagation has focused on designing custom schedules that try to import some features from the priority-based schedules while maintaining a degree of parallelism [16, 11].

Our contributions. In this work, we address the challenges of parallel belief propagation by showing how to efficiently parallelize any priority-based schedule for belief propagation. The key idea is that we can *relax* the priority-based schedules by allowing limited out-of-order execution, concretely implemented using a *relaxed scheduler*, as we will explain next.

Consider a belief propagation algorithm that schedules the message updates according to a priority function r by always updating the message $\mu_{i\to j}$ with the highest priority $r(\mu_{i\to j})$ next; this framework captures existing priority-based schedules such as residual belief propagation [14] and its variants [20, 38]. Concretely, an iterative centralized version of this algorithm can be implemented by storing the messages in a priority queue Q, and iterating the following procedure:

(1) Pop the top element for Q to obtain the message $\mu_{i\to j}$ with highest priority $r(\mu_{i\to j})$.

(2) Update message $\mu_{i\to j}$ following (1).

(3) Update the priorities in Q for messages affected by the update.

This template does not easily lend itself to efficient parallelization, as the priority queue Q becomes a contention bottleneck. Previous work, e.g. [16, 11] investigated various heuristics for the parallel scheduling of updates in belief propagation, trading off increased parallelism with additional work in processing messages or even potential loss of convergence.

In this paper, we investigate an alternative approach, replacing the priority queue Q with a *relaxed priority queue (scheduler)* to obtain a efficient parallel version of the above template. A *relaxed scheduler* [3, 1, 2, 5] is similar to a priority queue, but instead of guaranteeing that the *top* element is always returned first, it only guarantees to return *one of the top k elements by priority* , where k is a variable parameter. Relaxed schedulers are popular in the context of parallel graph processing, e.g. [17, 26], and can induce non-trivial trade-offs between the degree of relaxation and the scalability of the underlying implementation, e.g. [1, 5].

For belief propagation, relaxed schedulers induce a *relaxed priority-based scheduling* of the messages, roughly following the original schedule but allowing for message updates to be performed out of order, with guarantees on the maximum degree of priority inversion. We investigate the scalability-convergence trade-off between the *degree of relaxation* in the scheduler, and the *convergence behaviour* of the underlying algorithm, both theoretically and practically. In particular:

– We present a general scheme for parallelizing belief propagation schedules using relaxed schedulers with theoretical guarantees. While relaxed schedulers have been applied in other settings, and relaxed scheduling has been touched upon in belief propagation scheduling [16], no systematic study on relaxed belief propagation scheduling has been performed in prior work.

– We provide a theoretical analysis on the effects of relaxed scheduling for belief propagation on trees. We exhibit both positive results–instance classes where relaxation overhead is negligible– and negative results, i.e., worst-case instances where relaxation can cause significant wasted work.
– We implement and experimentally compare different variants of belief propagation under relaxed scheduling. We identify a new family of relaxed schedulers which consistently matches or outperforms previous proposals. Benchmarks show that this framework gives state-of-the-art parallel scalability on a wide variety of Markov random field models.

2 Preliminaries and related work

2.1 Belief Propagation

We consider marginalization in *pairwise Markov random fields*; one can equivalently consider factor graphs or Bayesian networks [40]. A pairwise Markov random field is defined by a set of random variables $X_1, X_2, \ldots, X_n$, a graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, and a set of *factors*

$$\psi_i \colon D_i \to \mathbb{R}^+ \qquad \text{for } i \in V,$$
$$\psi_{ij} \colon D_i \times D_j \to \mathbb{R}^+ \qquad \text{for } \{i, j\} \in E,$$

where D_i denotes the domain of random variable X_i. The edge factors ψ_{ij} represent the dependencies between the random variables, and the node factors ψ_i represent a priori information about the individual random variables; the Markov random field defines a joint probability distribution on $X = (X_1, X_2, \ldots, X_n)$ as

$$\Pr\big[X = x\big] \propto \prod_i \psi_i(x_i) \prod_{ij} \psi_{ij}(x_i, x_j)\,,$$

where the 'proportional to' notation $\propto$ hides the normalization constant applied to the right-hand side to obtain a probability distribution. The marginalization problem is to compute the probabilities $\Pr[X_i = x]$ for a specified subset of variables; for convenience, we assume that any observations regarding the values of other variables are encoded in the node factor functions ψ_i.

Belief propagation is a message-passing algorithm; for each ordered pair (i, j) such that $\{i, j\} \in E$, we maintain a *message* $\mu_{i\to j} \colon D_j \to \mathbb{R}$, and the algorithm iteratively updates these messages until the values (approximately) converge to a fixed point. On Markov random fields, the message update rule gives the new value of message $\mu_{i\to j}$ as a function of the old messages directed to node i by

$$\mu_{i\to j}(x_j) \propto \sum_{x_i \in D_i} \psi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i)\setminus\{j\}} \mu_{k\to i}(x_i)\,, \tag{2}$$

where $N(j)$ denotes the neighbors of node j in the graph G. Once the algorithm has converged, the marginals are estimated as

$$\Pr[X_i = x_i] \propto \psi_i(x_i) \prod_{j \in N(i)} \mu_{j\to i}(x_i)\,.$$

The update rule (2) can be applied in arbitrary order. The standard *synchronous belief propagation* updates all the message simultaneously; in each global round $t = 1, 2, 3, \ldots$, given message values $\mu^t_{i\to j}$ for all pairs $\{i, j\} \in E$, the new values $\mu^{t+1}_{i\to j}$ are computed as

$$\mu^{t+1}_{i\to j}(x_j) \propto \sum_{x_i \in D_i} \psi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i)\setminus\{j\}} \mu^t_{k\to i}(x_i)\,.$$

Asynchronous belief propagation. Starting with Elidan et al. [14], there has been a line of research arguing that *asynchronous* or *iterative* schedules for belief propagation tend to converge more reliably and with fewer message updates that the synchronous schedule. In particular, practical work has focused on schedules that attempt to iteratively perform 'the most useful' update at each step; the most prominent of these algorithms is the *residual belief propagation* of Elidan et al. [14], with other proposals aiming to address its shortcomings in various cases.

Residual belief propagation. Given a current state of messages, let $\mu'_{i\to j}$ denote the message we would obtain by applying the message update rule (2) to message $\mu_{i\to j}$. In residual belief propagation, the priority of a message is given by the *residual* $\mathrm{res}(\mu_{i\to j})$ of a message $\mu_{i\to j}$, defined as

$$\mathrm{res}(\mu_{i\to j}) = \|\mu'_{i\to j} - \mu_{i\to j}\|\,, \tag{3}$$

where $\|\cdot\|$ is an arbitrary norm; in this work, we assume L^2 norm is used unless otherwise specified. That is, the residual of a message corresponds to amount of change that would happen if message $\mu_{i\to j}$ would be updated. Note that this means that residual belief propagation performs *lookahead*, that is, the algorithm precomputes the future updates before applying them to the state of the algorithm. We will explore the performance of this base algorithm, as well as additional variants with *weight decay* [20] and *without lookahead* [38]. (Due to space constraints, we leave their detailed description to Appendix C.)

2.2 Parallel belief propagation

The question of parallelizing belief propagation is not fully understood. The synchronous schedule is trivially parallelizable by performing updates within each round in parallel, but the improved convergence properties of iterative schedules cannot easily be translated to parallel setting. Recent proposals aim to bridge this gap in an ad-hoc manner by designing custom algorithms for specific parallel settings.

Residual splash. *Residual splash* [16] is a vertex-based algorithm inspired by residual BP. Residual splash was designed for MapReduce computation, and it aims to have larger individual tasks while retaining a similar structure to residual BP. Specifically, the algorithm works by defining a priority function over nodes of the Markov random field, and selecting the next node to process in a strict priority order. For the selected node, the algorithm performs a *splash* operation that propagates information within distance *H* in the graph; in practice, this results in threads performing larger individual tasks at once, offsetting the cost of accessing the strict scheduler.

Randomized synchronous belief propagation. Van der Merve et al. [11] proposed a parallelization scheme for belief propagation on GPUs, mixing the structure of synchronous and residual belief propagation. Their algorithm considers all messages at once in global rounds, and performs *filter-and-select* steps. First, it filters out all messages whose residuals are below the convergence threshold. Second, out of the remaining messages, select a p-fraction uniformly at random to update. The fraction p is adjusted on the fly based on the convergence of the algorithm, preferring a low value if the algorithm is converging slowly, and a high value if it is converging fast. This algorithm is well-suited for GPUs, as the filter-and-select steps can be efficiently implemented. However, as shown by our experimental study, this strategy is not efficient on CPUs, on real-world models. Conversely, as discussed in [11], the dynamic priority-based strategy we propose would be hard to implement efficiently on GPUs, due to its irregular structure.

3 Relaxed priority-based belief propagation

In this section, we describe our framework for parallelizing belief propagation schedules via relaxed schedulers. The main idea of the framework follows the description given in the introduction; however, we generalize it to capture schedules that do not use individual messages as elementary tasks, e.g. residual path [16].

3.1 Relaxed scheduling for iterative algorithms

Relaxed schedulers are a basic tool to parallelize iterative algorithms, used in the context of large-scale graph processing [17, 26, 7, 12, 13]. At a high level, such iterative algorithms can be split into *tasks*, each corresponding to a local operation involving, e.g., a vertex and its edges. A standard example is Dijkstra's classic shortest-paths algorithm, where each task updates the distance between a vertex and the source, as well as the distances of the vertex's neighbours. In many algorithms, tasks have a natural priority order—in Dijkstra's, the top task corresponds to the vertex of minimum distance from the source. Many graph algorithms share this structure [37, 2], and can be mapped onto the priority-queue pattern described in the introduction. However, due to contention on the priority queue, implementing this pattern in practice can negate the benefits of parallelism [26].

Relaxed Scheduler Definition. A natural idea is to downgrade the guarantees of the perfect priority queue, to allow for more parallelism. Relaxed schedulers [1] formalize this relaxation as follows. We are given a parameter k, the degree of relaxation of the scheduler. The k-relaxed scheduler is a *sequential* object supporting `Insert (<key, priority>)`, `IncreaseKey (<key, priority>)`, with the usual semantics, and an `ApproxDeleteMin()` operations, ensuring:

(1) **Rank Bound.** `ApproxDeleteMin()` always returns one of the top k elements in priority order.

(2) k**-Fairness.** A *priority inversion* on element `<key, priority>` is the event that `ApproxDeleteMin()` returns a key with a *lower* priority while `<key, priority>` is in the queue. Any element can experience at most k priority inversions before it must be returned.

Relaxed schedulers are quite popular in practice, as several efficient implementations have been proposed and applied [36, 6, 39, 4, 18, 26, 31, 3, 35], with state-of-the-art results in the graph processing domain [26, 17, 19]. A parallel line of work has attempted to provide guarantees on the amount of relaxation in individual schedulers [3, 2, 34], as well as the impact of using relaxed scheduling on existing iterative algorithms [1, 5]. Here, we employ the modeling of relaxed schedulers used in e.g. [2, 5] for graph algorithms, but apply it to inference on graphical models.

3.2 Relaxed priority-based belief propagation

Given a Markov random field, a priority-based schedule for BP is defined by a set of *tasks* $T_1, T_2, \ldots, T_K$, each corresponding to a sequence of edge updates, and a priority function r that assigns a priority $r(T_i)$ to a task based on the current state of the messages as well as possible auxiliary information maintained separately. As discussed in the introduction, a non-relaxed algorithm can store all tasks in a priority queue, iteratively retrieve the highest-priority task, perform all its message updates, and update priorities accordingly.

The relaxed variant works in exactly the same way, but assuming a k*-relaxed* priority scheduler Q_k. More precisely, the following steps are repeated until a fixed convergence criterion is reached:

(1) $T_i \leftarrow Q_k$.`ApproxDeleteMin`() selects a task T_i among the k of highest priority in Q_k.
(2) Perform all message updates specified by the task T_i.
(3) Update the priorities for all tasks.

Note that tasks can be executed multiple times in this framework. In particular, we assume that the priority $r(T_i)$ of a task T_i can only remain the same or increase when other tasks are executed, and the only point where the priority decreases is when the task is actually executed.

3.3 Concurrent implementation

The sequential version of a priority-based schedule for belief propagation can be implemented using a priority queue Q. One could map the sequential pattern directly to a parallel setting, by replacing the sequential priority queue with a linearizable concurrent one. However, this may not be the best option, for two reasons. First, it is challenging to build *scalable* exact priority queues [21]—the data structure is inherently contended, which leads to poor cache behavior and poor performance. Second, in this context, linearizability only gives the illusion of atomicity with respect to task message updates: the data structure only ensures that the *task removal* is atomic, whereas the actual message updates which are part of the task are not usually performed atomically together with the removal.

The Multiqueue. For this reason, in our framework, we use a *relaxed* priority scheduler, i.e. a scalable approximate priority queue called the Multiqueue [32, 3]. As the name suggests, the Multiqueue is composed of m independent *exact* priority queues. To `Insert` an element, a thread picks one of the exact priority queues uniformly at random, and inserts into it. To perform `ApproxDeleteMin()`, the thread picks *two* of the priority queues uniformly at random, and removes the *higher priority* element among their two top elements. Although very simple, this strategy has been shown to have strong probabilistic rank and fairness guarantees:

Theorem 1 ([3, 1])**.** *A Multiqueue formed of $m \geq 3$ priority queues ensures the rank and fairness guarantees with parameter $k = O(m \log m)$, with high probability.*

Our Implementation. For our purposes, we assume that each thread i has one or a few local concurrent priority queues, used to store pointers to BP-specific tasks (e.g. messages), which are prioritized by an algorithm-specific function, e.g. the residual values for residual BP. We store additional metadata as required by the algorithm and the graphical model. (In our experiments, we use binary heaps for these priority queues, protected by locks.) To process a new task, the thread selects two among all the priority queues uniformly at random, and accesses the task/message from the queue whose top element has higher priority. The task is marked as *in-process* so it cannot be processed concurrently by some other thread. The thread then proceeds to perform the metadata updates required by the underlying variant of belief propagation, e.g., updating the message and the priorities of messages from the corresponding node. The termination condition, e.g., the magnitude of the largest update, is checked periodically.

4 Dynamics of relaxed belief propagation on trees

As we will see in Section 5, the relaxed priority-based belief propagation schedules yield fast converge times on a wide variety of Markov random fields; specifically, the number of message updates is roughly the same as for the non-relaxed version, while the running times are lower. The complementary theoretical question we examine here is whether we can give analytical bounds how much extra work—in terms of additional message updates—the relaxation incurs in the worst-case.

Unfortunately, the dynamics of even synchronous belief propagation are poorly understood on general graphs, and no priority-based algorithms provide general guarantees on the convergence time. As such, we can only hope to gain some limited understanding on why relaxation retains the fast convergence properties of the exact priority-based schedules. Here, we present some theoretical evidence suggesting that as long as a schedule does not impose long dependency chains in the sequence of updates, relaxation incurs low overhead, but also that simple (non-loopy) worst-case instances exist.

Analytical model. For analysis of the relaxed priority-based belief propagation, we consider the formal model introduced by [5, 2] to analyze performance of iterative algorithms under relaxed schedulers. Specifically, we model a relaxed scheduler Q_k as a data structure which stores pairs corresponding to tasks and their priorities, with the operational semantics given in Section 3. In particular, there exists a parameter k such that each `ApproxDeleteMin` returns one of the k highest priority tasks in Q_k, and if a task T becomes the highest priority task in Q_k at some point during the execution, then one of the next k `ApproxDeleteMin` operations returns T. (By [3, 1], our practical implementation will satisfy these conditions with parameter $k = O(p \log p)$ w.h.p., where p is the number of concurrent threads.) Other than satisfying these properties, we assume that the behavior of Q_k can be adversarial, or randomized.

We model the behavior of relaxed priority-based belief propagation by investigating the number of message updates needed for convergence when the algorithm is executed *sequentially* using a relaxed scheduler Q_k satisfying the above constraints. This analysis reduces to a sequential game between the algorithm, which queries Q_k for tasks/messages, and the scheduler, which returns messages in possibly arbitrary fashion. One may think of the relaxed sequential execution as a form of linearization for the actual parallel execution—reference [1] formalizes this intuition. Please see the discussion at the end of this section for a practical interpretation.

Relaxed BP on trees. We now consider the behavior of relaxed residual belief propagation schedules on *trees with a single source*. The setting is similar to the analysis of residual splash of Gonzalez et al. [16].Specifically, we assume that the Markov random field and the initialization of the algorithm satisfies (1) The graph $G = (V, E)$ is a tree with a specified root r; and (2) The factors of the Markov random field and the initial messages are such that the residuals are zero for all messages other than the outgoing messages from the root, i.e., $\mathrm{res}(\mu_{i \to j}) = 0$ if $i \neq r$.

These conditions mean that residual belief propagation will start from the root, and propagate the messages down the trees until propagation reaches all leaves. In particular, residual belief propagation without relaxation will perform $n - 1$ message updates before convergence, updating each message away from root once. While this setting is restrictive, it does model practical instances where the MRF has tree-like structure, such as LDPC codes (see Section 5).

To characterize the dynamics on relaxed residual belief propagation on trees with a single source, we observe that the algorithm can make two types of message updates:

- Updating a message with zero residual, in which case nothing happens (*a wasted update*). This happens if the scheduler relaxes past the range of messages with non-zero residual.
- Updating a message $\mu_{i \to j}$ with non-zero residual, in which case the residual of $\mu_{i \to j}$ goes down to zero, and the messages $\mu_{j \to k}$ for the children k of j may change their residuals to non-zero values (*a useful update*).

It follows that each edge will get updated only once with non-zero residual. At any point of time during the execution of the algorithm, we say that the *frontier* is the set of messages with non-zero residual, and use $F(t)$ to denote the size of the frontier at time step t.

To see how the size of the frontier relates to the number message updates in relaxed residual belief propagation, observe that after a useful update, we have one of the following cases:

- If $F(t) \geq k$, then the next `ApproxDeleteMin`() operation to Q_k will give an edge with non-zero residual, resulting in a useful update.

– If $F(t) < k$, then in the worst case we need k `ApproxDeleteMin()` operations until we perform a useful update.

Our main analytic result bounds the worst-case work incurred by relaxation in two concrete cases.

Lemma 2. *Assume a k-relaxed scheduler Q_k for belief propagation in the tree model defined above. The total number of updates performed by relaxed residual BP can be bounded as follows:*

– ***Good case: uniform expansion.*** *If the tree model has identical and non-deterministic edge factors ψ_{ij} with $\psi_{ij}(x_i, x_j) \neq 0$ for all $\{i, j\}$, then the total number of updates performed by relaxed residual BP is $n + O(Hk^2)$.*
– ***Bad case: long paths.*** *There exists a tree instance with height $H = o(n)$ and an adversarial scheduler where relaxed residual belief propagation performs $\Omega(kn)$ message updates.*

Discussion. The full analysis and illustrations are provided in Appendix A and B. To interpret the results, first note that, in practice, the relaxation factor is in the order of p, the number of threads, and that H is usually small (e.g., logarithmic) w.r.t. the total number of baseline updates n. Thus, in the good case, the $O(k^2H)$ overhead can be seen as negligible: as p iterations occur in parallel, the average time-to-completion should be $n/p + O(kH)$, which suggests almost perfect parallel speedup. At the same time, our worst-case instances shows that relaxed residual BP is not a "silver bullet:" there exist tree instances where it may lead to $\Omega(kn)$ message updates, i.e. asymptotically no speedup. We discuss how to alter the priority function to mitigate these worst-case instances on trees in the Appendix. The next section shows experimentally that such worst-case instances are unlikely.

5 Evaluation

We now empirically test the performance of the relaxed priority-based algorithms, comparing it against prior work. For the experiments, we have implemented multiple priority-based algorithms and instantiated them with both exact and relaxed priority schedulers.

Priority-based algorithms. We implemented several variants of sequential belief propagation, among which synchronous (round-robin), residual, weight decay, and residual without lookahead. These variants were briefly described in Section 2, and are specified in detail in Appendix C. For residual splash, we implemented two variants. The first is the standard splash algorithm, as given in [17]. The second is our own optimized version we refer to as *smart splash*, which only updates messages along breadth-first-search edges during a splash operation. This variant has similar convergence as the baseline residual splash algorithm, but performs fewer message updates and should be more efficient. We include the following instantiations of the algorithms in the benchmarks and compare them against the sequential residual algorithm. Please see Section 3.3 or Appendix C for implementation details.

Prior work algorithms. We ran many possible concurrent variants for the baseline algorithms, and chose the four best to we compare against our relaxed versions. For the full results, please see Appendix E. First, we choose the parallel version of the standard synchronous belief propagation (Synch). We omit some synchronous algorithms such as the randomized synchronous belief propagation of Van der Merve et al. [11] since they perform consistently worse (Appendix E.2.4). We also include the exact residual algorithms implemented via the coarse-grained priority queue, which maintains exact priority order. Specifically, here we implemented standard the residual BP algorithm (Coarse-G) and the splash algorithm of [17] (Splash) with the best value of H, which we found to be 10.

We also include the randomized version of splash algorithm (Random Splash) proposed in the journal version of the paper [16] with $H = 2$, which performed best. This algorithm uses a similar idea of relaxation, but, crucially, instead of a Multiqueue scheduler, they implement a naive relaxed queue where threads randomly insert and delete into p exact priority queues. While this distinction may seem small, it is known [3] that this variant does not implement a k-relaxed scheduler for any k, as its relaxation factor grows (diverges) as more and more operations are performed, and therefore corresponds to picking tasks at random to perform. As we will see in our experimental analysis, this does result in a significant difference between the number of additional (wasted) steps performed relative to a relaxed priority scheduler. Finally, we note that we are the first to implement this algorithm.

Relaxed algorithms. We compare the above algorithms against the algorithms we propose, i.e. relaxed versions of residual belief propagation (Relaxed Residual), weight decay belief propagation (Weight-Decay), residual without lookahead (Priority) and smart splash (Relaxed Smart Splash)

Input	Prior Work				Relaxed			
	Synch	Coarse-G	Splash (10)	Random Splash (2)	Residual	Weight-Decay	Priority	Smart Splash (2)
Tree	2.538x	0.265x	1.648x	2.252x	1.391x	1.282x	1.239x	2.121x
Ising	3.009x	0.801x	5.393x	11.731x	6.720x	6.276x	5.759x	14.175x
Potts	—	0.624x	1.041x	11.855x	7.454x	5.978x	5.850x	15.235x
LDPC	17.735x	1.166x	—	5.150x	13.393x	5.615x	—	10.519x

Table 1: Algorithm speedups with respect to the sequential residual algorithm. Higher is better.

Input	Prior Work				Relaxed			
	Synch	Coarse-G	Splash (10)	Random Splash (2)	Residual	Weight-Decay	Priority	Smart Splash (2)
Tree	48.000x	1.003x	16.442x	8.344x	1.020x	1.012x	3.657x	2.565x
Ising	45.006x	1.003x	9.266x	5.787x	1.058x	1.068x	1.816x	1.878
Potts	—	1.006x	9.005x	5.983x	1.068x	1.053x	1.791x	1.891x
LDPC	4.404x	1.003x	—	4.089x	1.007x	0.883x	—	0.973x

Table 2: Total updates relative to the sequential residual algorithm at 70 threads. Lower is better.

with the best value $H = 2$. For all these algorithms, the scheduler is a Multiqueue with 4 priority queues per thread, as discussed in Section 3, which we found to work best (although other values behave similarly).

Methodology. We run our experiments on four MRFs of moderate size: a binary tree of size 10^7, an Ising model [14, 20] of size $10^3 \times 10^3$, a Potts [38] of size $10^3 \times 10^3$ and the decoding of $(3, 6)$-LDPC code [30] of size $3 \cdot 10^5$. More information about tasks and running times given in Appendix D.

For each pair of algorithm and model, we run each experiment five times, and average the execution time and the number of performed updates on the messages. We executed on a 4-socket Intel Xeon Gold 6150 2.7 GHz server with four sockets, each with 18 cores, and 512GB of RAM. The code is written in Java; we use Java 11.0.5 and OpenJDK VM 11.0.5. Our code is available at https://cutt.ly/neurips20208924. Our code is fairly well optimized—in sequential executions it outperforms the C++-based open-source framework of libDAI [25] by more than 10x, and by more that 100x with multi-threading. The sizes of the inputs described in the previous paragraph are chosen such that their execution is fairly long while the data still can fit into RAM.

We run the baseline algorithm on one process since it is sequential, while all other algorithms are concurrent and, thus, are executed using 70 threads. The execution times (speedups) relative to the sequential baseline are presented in Table 1. Each cell of the table shows how much faster the corresponding algorithm works in comparison to the sequential residual one, i.e., higher is better; "—" means that the execution did not converge within a reasonable limit of time.

Results. See Table 1 for the speedups versus the baseline, on 70 threads. (For ablation studies, see Appendix E.) On trees, the fastest algorithm is, predictably, the synchronous one, since on tree-like models with small diameter D it performs only approximately $O(D)$ times more updates in comparison to the sequential baseline, while being almost perfectly parallelizable. So, it works well on perfect binary tree as our Tree model, but works much worse on the chain graphs. On Ising and Potts models, the best algorithm is Relaxed Smart Splash (RSS) with $H = 2$. The algorithm closest to it is Random Splash with $H = 2$, which is $20 - 30\%$ slower. For LDPC decoding, which is a tree-like model, the best-performing is again the Synchronous algorithm. We note the good performance of the relaxed residual algorithm, as well as of RSS, and the relatively poor performance of Random Splash, due to high numbers of wasted updates. Examining Table 2, we notice in general the relatively low number of wasted updates for relaxed algorithms. In summary, the choice of algorithm can depend on the model; however, one may choose Relaxed Smart Splash since it performs well on all our models.

6 Discussion

We have investigated the use of relaxed schedulers in the context of the classic belief propagation algorithm for inference on graphical model, and have shown that this approach leads to an efficient family of algorithms, which improve upon the previous state-of-the-art non-relaxed parallelization approaches in our experiments. Overall, our relaxed implementations, either Relaxed Residual or Relaxed Smart Splash, have state-of-the-art performance in multithreaded regimes, making them a good generic choice for any belief propagation task.

For future work, we highlight two possible directions. First is to extend our theoretical analysis to cover more types of instances; however, as we have seen, the structure of belief propagation schedules can be quite complicated, and the challenge is the figure out a proper framework for more general analysis. Second possible direction is extending our empirical study to a massively-parallel, multi-machine setting.

Broader impact

As this work is focused on speeding up existing inference techniques and does not focus on a specific application, the main benefit is enabling belief propagation applications to process data sets more efficiently, or enable use of larger data sets. We do not expect direct negative societal consequences to follow from our work, though we note that as with all heuristic machine learning techniques, there is an inherent risk of misinterpreting the results or ignoring biases in the data if proper care is not taken in application of the methods. However, such risks exist regardless of our work.

References

[1] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 133–142, New York, NY, USA, 2018. ACM.

[2] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 377–386, New York, NY, USA, 2018. ACM.

[3] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 283–292, New York, NY, USA, 2017. ACM.

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, USA, 2015. ACM.

[5] Dan Alistarh, Giorgi Nadiradze, and Nikita Koval. Efficiency guarantees for parallel incremental algorithms under relaxed schedulers. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 145–154, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. CAFÉ: Scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 475–488, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 467–478. ACM, 2016.

[8] Andres I Vila Casado, Miguel Griot, and Richard D Wesel. Informed dynamic scheduling for belief-propagation decoding of LDPC codes. In *2007 IEEE International Conference on Communications*, pages 932–937. IEEE, 2007.

[9] Gregory F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42(2):393–405, 1990.

[10] Paul Dagum and Michael Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60(1):141–153, 1993.

[11] Mark Van der Merwe, Vinu Joseph, and Ganesh Gopalakrishnan. Message scheduling for performant, many-core belief propagation. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC 2019)*, 2019.

[12] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 293–304, New York, NY, USA, 2017. ACM.

[13] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *30th on Symposium on Parallelism in Algorithms and Architectures (SPAA 2018)*, pages 393–404, 2018.

[14] Gal Elidan, Ian McGraw, and Daphne Koller. Residual belief propagation: informed scheduling for asynchronous message passing. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006)*, pages 165–173, 2006.

[15] Robert G. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[16] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (UAI 2009)*, volume 5, pages 177–184, 2009.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 17–30, 2012.

[18] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013.

[19] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. Unlocking ordered parallelism with the swarm architecture. *IEEE Micro*, 36(3):105–117, 2016.

[20] Christian Knoll, Michael Rath, Sebastian Tschiatschek, and Franz Pernkopf. Message scheduling methods for belief propagation. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2015)*, pages 295–310, 2015.

[21] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. In *European Conference on Parallel Processing*, pages 209–221. Springer, 2015.

[22] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX conference on Operating Systems Design and Implementation (OSDI'14)*, page 583–598, 2014.

[23] Ji Liu and Stephen J Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

[24] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrölä, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. In *26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, page 340–349, 2010.

[25] Joris M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, August 2010.

[26] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[27] Judea Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the Second AAAI Conference on Artificial Intelligence (AAAI 1982)*, page 133–136. AAAI Press, 1982.

[28] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[29] Thomas J. Richardson and Rüdiger L. Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on information theory*, 47(2):599–618, 2001.

[30] Thomas J. Richardson and Rüdiger L. Urbanke. *Modern coding theory*. Cambridge university press, 2008.

[31] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: MultiQueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.

[32] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 80–82, 2015.

[33] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996.

[34] Adones Rukundo, Aras Atalar, and Philippas Tsigas. Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[35] Konstantinos Sagonas and Kjell Winblad. A contention adapting approach to concurrent ordered sets. *Journal of Parallel and Distributed Computing*, 2017.

[36] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.

[37] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 152–163, New York, NY, USA, 2013. ACM.

[38] Charles Sutton and Andrew McCallum. Improved dynamic schedules for belief propagation. In *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007)*, pages 376–383, 2007.

[39] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-LSM relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*, pages 277–278, 2015.

[40] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. In *Exploring Artificial Intelligence in the New Millenium, chapter 8*, pages 239—269. Morgan Kaufmann, 2003.

A Analysis

Lemma 2. *Assume a k-relaxed scheduler Q_k for belief propagation in the tree model defined above. The total number of updates performed by relaxed residual BP can be bounded as follows:*

- ***Good case: uniform expansion.*** *If the tree model has identical and non-deterministic edge factors ψ_{ij} with $\psi_{ij}(x_i, x_j) \neq 0$ for all $\{i, j\}$, then the total number of updates performed by relaxed residual BP is $n + O(Hk^2)$.*
- ***Bad case: long paths.*** *There exists a tree instance with height $H = o(n)$ and an adversarial scheduler where relaxed residual belief propagation performs $\Omega(kn)$ message updates.*

Proof. **Good case: uniform expansion.** As the first case, we consider the tree model in the case where the edge factors ψ_{ij} are identical for all edges and not *deterministic*, i.e. $\psi_{ij}(x_i, x_j) \neq 0$ for all $\{i, j\}$. Let us say that the *level* of a message $\mu_{i\to j}$ is ℓ if the distance from i to the root r is ℓ. The conditions we imposed our Markov random field, together with the update rule (2), imply that the residuals of the messages are decreasing in the level ℓ of the message, and all messages on level ℓ will have the same residual when they are in the frontier. This means that residual schedule will prefer updating messages on lower levels first.

Now consider the progress of the relaxed residual belief propagation on this tree; let H denote the height of the tree. Now assume that all messages on levels $0, 1, \ldots, \ell - 1$ have had a useful update, and consider how many wasted updates we can make in the worst case before all messages on level ℓ have been processed. Let f denote the number of message on level ℓ still in the frontier:

- While $f \geq k$, there are at least k messages of level ℓ on the frontier. Since they have the highest residual out of the messages in the frontier, each update is a useful update of a message on level ℓ.
- When $f < k$, there can be updates that do not hit messages on level ℓ, which can possibly be wasted updates. However, the highest-priority messages are still from level ℓ, so every kth update will hit a message on level ℓ by the guarantees of the scheduler. Thus, in $(k-1)f = O(k^2)$ updates, all remaining messages on level ℓ have been processed.

Since there can be at most $n - 1$ useful updates, and the number of levels is $H - 1$, the total number of updates performed by relaxed residual belief propagation is $n + O(Hk^2)$.

Bad case: long paths. A simple example where relaxed residual belief propagation performs poorly is a path. That is, if our underlying tree is a path of length n with a root at one end, then relaxed residual belief propagation can perform $\Omega(kn)$ message updates in the worst case. However, the path has height $H = n$, so one might ask if there is a general upper bound of form $n + O(Hk^2)$ on trees without restricting the edge factors as in our previous example.

Unfortunately, turns out that without the restrictions above, we can construct examples of trees with height $H = o(n)$ where relaxed residual belief propagation still performs $\Omega(kn)$ message updates (see Figure 2 for an illustration):

(1) Start with a path of length $\sqrt{n}$, with a root at one end.
(2) Attach a new path of length $\sqrt{n}$ to each vertex.
(3) For each remaining degree-2 node in the graph, attach a single new node to it.

This construction results in a 3-regular rooted tree with $\Theta(n)$ nodes and depth $H = O(\sqrt{n})$. Finally, we choose the edge factors so that residuals on the side paths are larger than the residuals on the main path, so residual belief propagation will prefer following the side paths first.

One can now observe that under the adversarial model for the relaxed scheduler, the adversary can select the execution of the relaxed scheduler so that the frontier size never exceeds 4. That is, adversary forces the algorithm to process the graph one side path at time, wasting $k - 1$ steps between each useful update.

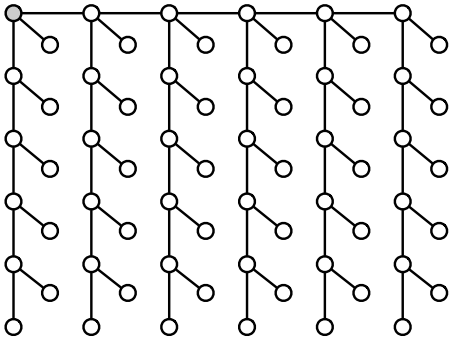


Figure 2: Example of the tree where relaxed residual belief propagation performs poorly.

Finally, we note that the same construction can be generalized to obtain instances with similar relaxation overhead and diameter $O(n^{1/c})$ for larger constants $c < k$, by simply working with paths of length $n^{1/c}$ and repeating the path attachment step c times.

□

Remark 3. As suggested by the above examples, one might consider changing the priority function to preferentially select messages closer to the source. This can lead to improved work guarantees for the relaxed schedule. Indeed, we discuss one concrete example in Section B, where we show how to relax the optimal schedule on trees. However, it is not straightforward to construct such priority functions so that they also make sense on general graphs, which can have non-monotonic potentials and cycles.

B Optimal schedule on trees

On trees, the belief propagation gives exact marginals under any schedule that updates each edge infinitely often. However, there is an optimal schedule that updates each message exactly once, requiring $O(n)$ message updates [27]. Assume the tree has a fixed root v:

(1) In the first phase, all messages towards the root are updated starting from the leaves; each message is updated only after all its predecessors have been updated.

(2) In the second phase, all messages away from the root are update starting from the root.

This schedule can be modeled in the priority-based scheduling framework as follows:

(1) Initially, the outgoing messages at leaf nodes have priority n, and all other messages have priority 0.

(2) When message is updated with non-zero priority, its priority is changed to 0.

(3) Once all messages $\mu_{k\to i}$ for $k \in N(i) \setminus \{j\}$ have been updated once with non-zero priority, the message $\mu_{i\to j}$ changes to priority to minimum of update priorities of the incoming edges minus one.

This priority function can clearly be implemented by keeping a constant amount of extra information per message. When the above schedule is executed with an exact scheduler, the algorithm will update each message once with non-zero priority before considering any messages with zero priority, and by following the analysis of [27], one can see that the algorithm has converged at that point.

Similarly, in the relaxed version of the schedule, the algorithm has converged once all messages have been updated once with non-zero priority. In addition, some messages may be updated multiple times with priority 0; we call these *wasted* updates, and the updates done while the message has non-zero priority *useful* updates.

Claim 4. *The relaxed version of the optimal schedule on trees performs $O(n + k^2H)$ message updates, where H is the height of the tree.*

Proof. For the purposes of analysis, assign messages into buckets $B_1, B_2 \ldots, B_n$ so that bucket B_ℓ contains the messages that will have their useful update done with priority ℓ. One can observe that the update priority of message $\mu_{i\to j}$ is the $n - d$, where d is the maximum distance from node i to a leaf using a path that does not cross edge $\{i, j\}$. Since this is bounded by the diameter of the tree, there are at most $2H$ non-empty buckets.

Assume that all messages in buckets $B_n, B_{n-1}, \ldots, B_{\ell+1}$ have been already had a useful update. We now show that in there can be at most k^2 wasted updates before all messages in B_ℓ have had a useful update. Since all earlier buckets have been processed, all messages in B_ℓ have either already had a useful update, or have priority ℓ. Let b be the number of messages remaining in bucket B_ℓ:

– While $b \geq k$, there are at least k messages with priority ℓ, so each update is a useful update of a message in B_ℓ

– When $b < k$, there can be wasted updates. However, since buckets $B_n, B_{n-1}, \ldots, B_{\ell+1}$ have had all useful updates, the top elements in the schedule will be from bucket B_ℓ, and

thus by the guarantees of the scheduler, there can be at most $k-1$ wasted updates before the top element is processed. Thus, in $b(k-1) = O(k^2)$ updates, all remaining messages of B_ℓ will have their useful update.

By an inductive argument, all non-empty buckets have been processed after $O(k^2 H)$ wasted updates, so the total number of updates is $O(n + k^2 H)$. □

C Algorithms

C.1 Asynchronous belief propagation

Starting with Elidan et al. [14], there has been a line of research arguing that *asynchronous* or *iterative* schedules for belief propagation tend to converge more reliably and with fewer message updates that the synchronous schedule. In particular, the practical work has focused on developing schedules that attempt to iteratively perform 'the most useful' update at each step; the most prominent of these algorithms is the *residual belief propagation* of Elidan et al. [14], with other proposals aiming to address the shortcomings of residual belief propagation in various cases.

Residual belief propagation. Given a current state of messages, let $\mu'_{i\to j}$ denote the message we would obtain by applying the message update rule (2) to message $\mu_{i\to j}$. In residual belief propagation, the priority of a message is given by the *residual* $\mathrm{res}(\mu_{i\to j})$ of a message $\mu_{i\to j}$, defined as

$$\mathrm{res}(\mu_{i\to j}) = \|\mu'_{i\to j} - \mu_{i\to j}\|\,, \tag{4}$$

where $\|\cdot\|$ is an arbitrary norm; in this work, we assume L^2 norm is used unless otherwise specified. That is, the residual of a message corresponds to amount of change that would happen if message $\mu_{i\to j}$ would be updated. Note that this means that residual belief propagation performs *lookahead*, that is, the algorithm precomputes the future updates before applying them to the state of the algorithm.

Weight decay belief propagation. *Weight decay belief propagation* of [20] is a variant of residual belief propagation that penalizes message priorities for repeated updates. That is, let $m(\mu_{i\to j})$ denote how many times message $\mu_{i\to j}$ has been updated by the algorithm, and let $\mathrm{res}(\mu_{i\to j})$ denote the residual of a message as above. The priority function of weight decay belief propagation is

$$r(\mu_{i\to j}) = \frac{\mathrm{res}(\mu_{i\to j})}{m(\mu_{i\to j})}\,.$$

The motivation behind this weight decay scheme is that empirical observations suggest that one possible failure mode of residual belief propagation is getting stuck in cycles with large residuals; the weight decay prioritizes other edges in cases where this happens.

Residual without lookahead. Another variant of residual belief propagation is the lookahead-avoiding belief propagation of [38]. As the name implies, this algorithm does not perform the exact residual computation using (4), but instead approximates the residuals indirectly, with the aim of reducing the computational cost of priority updates.

Informally, the basic idea is that for each message $\mu_{i\to j}$, we track the amount other incoming messages at node i have changed since the last update of $\mu_{i\to j}$, and use this to define the priority of updating $\mu_{i\to j}$. The actual approximation in the algorithm uses a slightly different notion of residual from (4), so we refer to [38] for full details.

C.2 Parallel belief propagation

As discussed above, the question of parallelizing belief propagation is fairly poorly understood. The synchronous schedule is trivially parallelizable by performing updates within each round in parallel, but the improved converge properties of the iterative schedules cannot easily be translated to parallel setting. There have been recent proposals that aim to bridge this gap in an ad-hoc manner by designing custom algorithms for specific parallel computation settings.

Residual splash. The *residual splash* belief propagation [16] is a vertex-based algorithm inspired by residual belief propagation. The residual splash algorithm was initially designed for MapReduce computation, and it aims to have larger individual tasks while retaining a similar structure to residual belief propagation.

Specifically, the residual splash algorithm works by defining a priority function over nodes of the Markov random field, and selecting the next node to process in a strict priority order. For the selected node, the algorithm performs a *splash* operation that propagates information within distance *H* in the graph; in practice, this results in threads performing larger individual tasks at once, offsetting the cost of accessing the strict scheduler.

In detail, the priority of for nodes is given by the *node residual*, defined as

$$\mathrm{res}(i) = \max_{j \in N(i)} \mathrm{res}(\mu_{j \to i}) \,.$$

Given a *depth parameter* H, the splash operation at node i is defined by following sequence of message updates:

(1) Construct a BFS tree T of depth H rooted at node i.

(2) In the reverse BFS order on T—starting from leaves—process all nodes in T, updating all outgoing messages for each node processed.

(3) Repeat the previous step in BFS order, i.e., starting from the root.

In other words, this process gather all available information at radius H from the selected node, and propagates it to all nodes within the radius.

Randomized synchronous belief propagation. Van der Merve et al. [11] proposed a parallelization scheme for belief propagation on GPUs, mixing the structure of synchronous and residual belief propagation. Their algorithm considers all messages at once in global rounds, and performs the following filter-and-select steps before computing the message updates:

(1) Filter out all messages whose residuals are below the convergence threshold.

(2) Out of the remaining messages, select a p fraction of messages uniformly at random to update.

Alternatively, the process can perform the algorithm on per-node basis, using node residuals as in the residual splash algorithm.

The fraction p is adjusted on the fly based on the convergence of the algorithm, preferring a low value if the algorithm is converging slowly, and a high value if it is converging fast. Concretely, the selection scheme for p used by [11] is to set $p = 1$ if the number of messages above the convergence threshold decreased by at least 10% in the last round, and set it to a smaller fixed value otherwise.

We note that the randomized synchronous algorithm is particularly well suited for GPU use, as the filter-and select steps can be efficiently implemented on GPUs. However, as shown by our experimental study, this strategy is not efficient on a subset of real-world models, when ported to CPU. Conversely, as discussed by the authors of [11], the dynamic priority-based strategy we propose would be hard to implement efficiently on GPUs, due to its irregular structure.

D Models

We run our experiments on four Markov random fields models.

Trees. As a simple base case, we consider a simple tree model similar to the analytical setting in Section 4. The underlying graph is a full binary tree on 10 millions vertices, and the other parameters are set up as follows:

- All variables are binary, i.e. the domain is $\{0, 1\}$ for each variable.
- Vertex factors are $(0.1, 0.9)$ for the root and $(0.5, 0.5)$ for all other vertices.
- Edge factors are $\psi_{ij}(x, y) = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$ for all edges.

As discussed in Section 4, these choices create a setup where the belief propagation has to propagate information from the root to all other nodes. Thus, under an optimal schedule, the total number of performed updates is be equal to $10^7 - 1$. Since we know that all algorithms will converge on this model, we run the algorithms until exact convergence.

Ising and Potts models. Ising and Potts models are Markov random fields defined over an $n \times n$ grid graph, arising from applications in statistical physics. Both of Ising [14, 20] and Potts [38] models were used in prior work as test case, and in general they offer a class of good test instances, as they both exhibit complex cyclic propagations and are easy to generate.

For the parameters of the models, we mostly follow prior work in the setup. As the underlying graph, we use a $10^3 \times 10^3$ grid graph to get instances where the effects of parallelization are clearly visible. For the Ising model, we select the factors similarly to [14, 20]:

- The variable domain is $\{-1, 1\}$ for all variables.
- Vertex factors are $\psi_i(x) = e^{\beta_i x}$.
- Edge factors are $\psi_{ij}(x, y) = e^{\alpha_{ij} xy}$.
- The parameters α_{ij} and β_i are chosen uniformly at random from $[-1, 1]$.

For the Potts model, we select the factors following [38]:

- The variable domain is $\{0, 1\}$ for all variables.
- Vertex factors are $\psi_i(x) = \begin{cases} e^{\beta_i}, & x = 1 \\ 1, & x = 0 \end{cases}$.
- Edge factors are $\psi_{ij}(x, y) = \begin{cases} e^{\alpha_{ij}}, & x = y \\ 1, & x \neq y \end{cases}$.
- The parameters α_{ij} and β_i are chosen uniformly at random from $[-2.5, 2.5]$.

For both Ising and Potts models, we set the convergence threshold to 10^{-5}. That is, we terminate algorithm once all task have priority below this threshold.

LDPC codes. Finally, we generate Markov random fields corresponding to the $(3, 6)$-LDPC (*low density parity check code* [15]) decoding. LDPC decoding is one of the more successful application of belief propagation. We consider a simple version of LDPC decoding task where convergence guarantees exist [29]. However, we stress that coding theory is its own extensive research area, and far more optimized codes and decoding algorithms exist in practice—we simply use LDPC decoding to observe the comparative scaling behavior of our implementations on instances where synchronous belief propagation is guaranteed to converge. For a more detailed background on LDPC decoding and other aspects of coding theory, refer e.g. to the book [30].

More precisely, we consider $(3, 6)$-LDPC decoding over a binary symmetric channels. Informally, a $(3, 6)$-LDPC code is a $(3, 6)$-regular bipartite graph, where each degree 3 node corresponds to a binary *variable* and each degree 6 node corresponds to a *constraint* of form $x_{i_1} + x_{i_2} + \ldots + x_{i_6} = 0$ over the neighboring variables $x_{i_1}, x_{i_2}, \ldots, x_{i_6}$. Each sequence of variables that satisfies the all the constraints is *codeword* of the code. The basic setup is then that we send a codeword over a *channel* that flips each bit with probability ε, and the receiver will run belief propagation and use results of marginalization to infer the original codeword.

For our experiments, we build a $(3, 6)$-LDPC instance with $300\,000$ variable nodes and $150\,000$ constraint nodes by selecting a random $(3, 6)$-regular bipartite graph, and initialize the node factors corresponding to the all-zero codeword sent over binary symmetric channel with error probability $\varepsilon = 0.07$. Under these conditions, belief propagation is guaranteed to correctly decode the instance with high probability [29]; indeed, all the algorithms that converged decoded the codeword correctly in our experiments. The codeword length was again selected to get roughly comparable baseline running times as for the other instances.

Concretely, we get Markov random field where the underlying graph is a random bipartite graph with $450\,000$ nodes. For each variable node i, let $x_i \in \{0, 1\}$ be the 'transmitted' value of the variable, randomly generated to be 1 with probability ε and 0 otherwise. The factors have the following structure:

- The domains of variable nodes are binary domains $\{0, 1\}$. For the constraint nodes, the domain is $\{0, 1\}^6$—different bit masks of length 6.
- The node factors for variable nodes are

$$\psi_i(y) = \begin{cases} 1 - \epsilon, & y = x_i \\ \epsilon, & y \neq x_i. \end{cases}$$

Input	Residual	Prior Work						Relaxed				
		Synch	CG	S 2	S 10	RS 2	RS 10	Residual	WD	Priority	RSS 2	RSS 10
Tree	1.30 min	2.538x	0.265x	0.608x	1.648x	2.252x	2.241x	1.391x	1.282x	1.239x	2.121x	2.110x
Ising	2.76 min	3.009x	0.801x	0.609x	5.393x	11.731x	13.512x	6.720x	6.276x	5.759x	14.175x	10.337x
Potts	3.02 min	—	0.624x	0.484x	1.041x	11.855x	12.854x	7.454x	5.978x	5.850x	15.235x	11.091x
LDPC	4.62 min	17.735x	1.166x	—	—	5.150x	—	13.393x	5.615x	—	10.519x	—

Table 3: Algorithm speedups with respect to the sequential residual algorithm. Higher is better.

Input	Residual	Prior Work						Relaxed				
		Synch	CG	S 2	S 10	RS 2	RS 10	Residual	WD	Priority	RSS 2	RSS 10
Tree	10M	48.000x	1.003x	8.658x	16.442x	8.344x	15.197x	1.020x	1.012x	3.657x	2.565x	5.027x
Ising	25.3M	45.006x	1.003x	5.719x	9.266x	5.787x	10.232x	1.058x	1.068x	1.816x	1.878x	6.147x
Potts	30M	—	1.006x	5.903x	9.005x	5.983x	9.109x	1.068x	1.053x	1.791x	1.891x	6.328x
LDPC	7.23M	4.404x	1.003x	—	—	4.089x	—	1.007x	0.883x	—	0.973x	—

Table 4: Total updates relative to the sequential residual algorithm at 70 threads. Lower is better.

For the constraint nodes, the node factor $\psi_c(y)$ is equal to the number of ones in $y \in \{0,1\}^6$ modulo 2; this effectively penalizes any value that does not satisfy the constraint.

– Edge factors $\psi_{ic}(x, y)$ is one if the corresponding bit in the $y \in \{0,1\}^6$ equals $x \in \{0,1\}$, and is zero otherwise.

For the LDPC instances, we set the convergence threshold to 10^{-2} to ensure fast convergence; this approximates the behavior of actual LDPC decoders.

E Experiments

In this section, we provide an additional study on the evaluation of the algorithms. At first, we give the extended results of running the algorithms on the moderate size inputs chosen in the main body of the paper. Unfortunately, due to the reasonably high running time it was impossible to make enough points for the plots to reason about the general effect of the parallelization. Thus, we execute the algorithms on a little bit smaller inputs.

E.1 Moderate size inputs

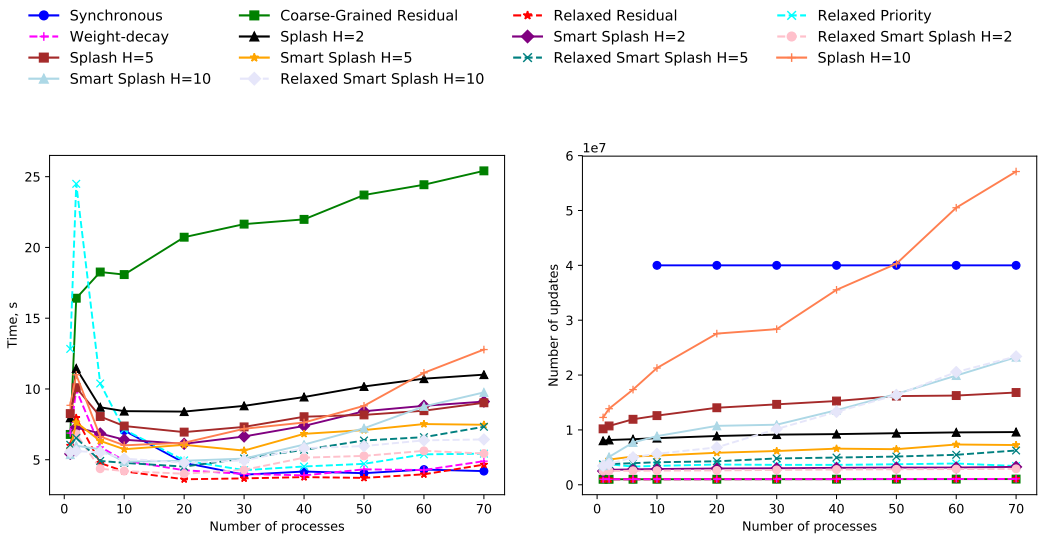
To present all the executed algorithms in the table, we shrink the abbreviations a little bit: Coarse-Grained now becomes CG, Splash becomes S, Random Splash becomes RS, Relaxed Residual rests Residual, Weight-Decay becomes WD, Relaxed Priority becomes Priority, and, finally, Relaxed Smart Splash becomes RSS. Table 3 contains the execution times (speedups) of the algorithms relative to the sequential baseline. Table 4 contains the number of updates performed by the algorithms in compare to the number of updates performed by the sequential baseline. The results do not differ much from the ones presented in the main body of the paper. The only notable thing is that Random Splash with $H = 10$ it performs better on Ising and Potts model than Random Splash with $H = 2$. However, we chose Random Splash with $H = 2$ as the best one, since Random Splash with $H = 10$ does not finish on LDPC input. Nevertheless, `Relaxed Smart Splash` outperforms Random Splash with both settings of H.

E.2 Small size inputs

In this subsection, we decrease the size of the inputs. Now, Tree model maintains a tree of size 10^6, Ising and Potts models are built on top of 300×300 grid graph, and, finally, LDPC model is set up with $30\,000$ length of the input vector. In general, we simply reduce the sizes of the models by approximately 10.

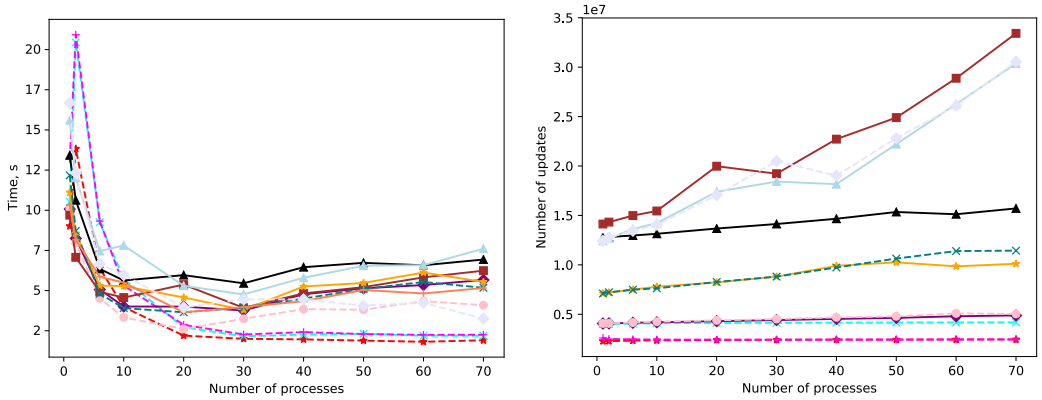
E.2.1 Scaling

How to read the plots. There are two types of plots per each model: the first shows the execution time of the algorithms, while the other one shows the number of updates performed. On the x axis we have the number of threads the algorithms were run on, while on the y axis we have: the time in seconds (for time plots) and the number of updates (for update plots). The dashed lines on the plots correspond to the algorithms that use a relaxed scheduler, while the others use either no concurrent scheduler, or an exact priority queue.



(a) Execution time (b) Number of updates

Figure 3: The results of the evaluation of the algorithms on the Tree model



(a) Execution time (b) Number of updates

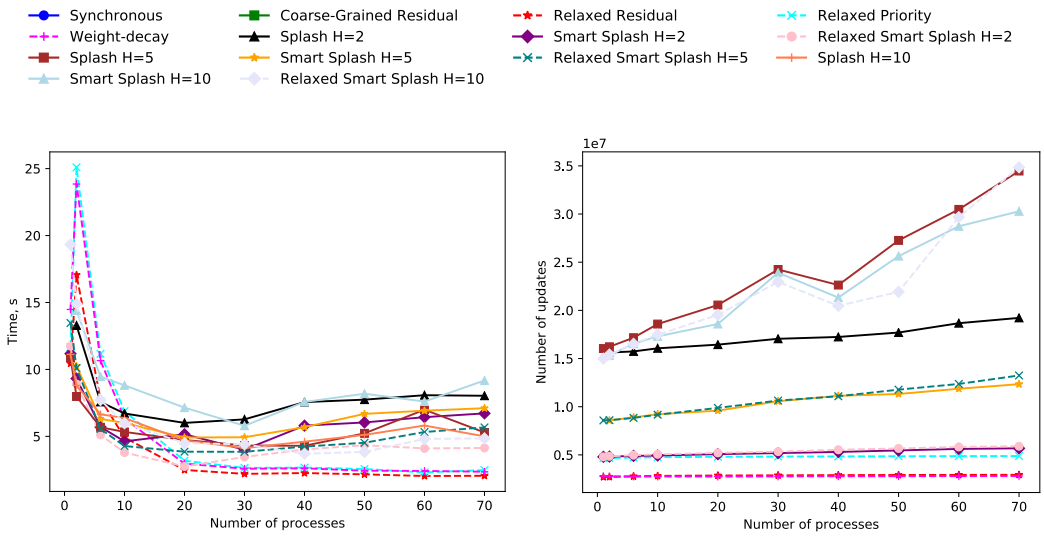
Figure 4: The results of the evaluation of the algorithms on Ising model

Whenever we have omitted algorithms from the plots or display incomplete data, this indicates poor performance for that algorithm on the metric displayed on the graph: either the algorithm did not converge or the values exceed the limit of the plot.

Tree model. As one can observe on the time plot (Figure 3a), the three algorithms with the best scaling on the tree instance are the synchronous belief propagation, relaxed residual and the weight-decay algorithm. For the relaxed algorithms, this mirrors our theoretical analysis from Section 4: as can be seen from Figure 3b, the relaxation incurs very low overhead in terms of additional updates, while the overhead from parallelization is also low. By contrast, the exact residual belief propagation performs exactly the minimum number of updates needed, but scales very badly due to the contention on the priority queue.

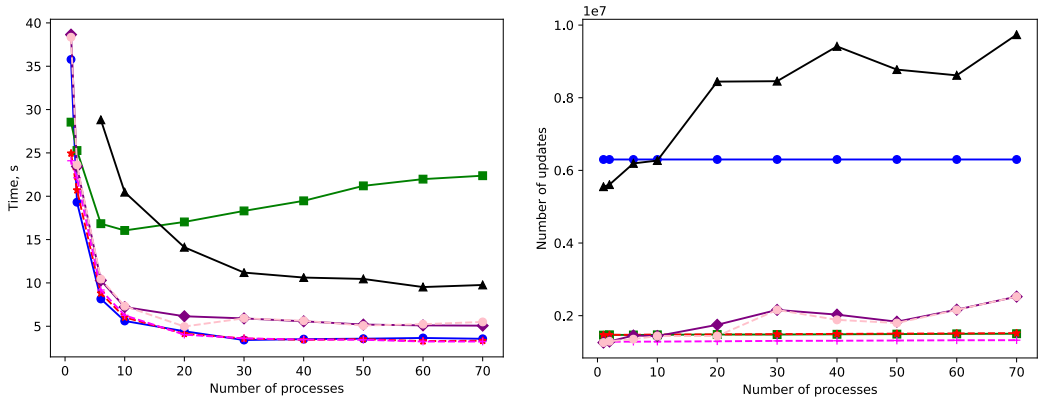
We note that on the tree instance, the synchronous belief propagation also scales very well when parallelized. The amount of work can be split evenly between the threads, and only $O(\log n)$ synchronous rounds are required for convergence.

Ising and Potts model. Ising and Potts models represent more challenging instances with lots of cycles, and are generally thought to be more representative of hard general graph instances for belief propagation. As can be seen in Figures 4a and 5a, relaxed algorithms perform consistently well on these instances, with relaxed residual belief propagation giving consistently the fastest convergence. These are followed by the exact splash algorithms, which generally perform slightly worse; however,



(a) Execution time (b) Number of updates

Figure 5: The results of the evaluation of the algorithms on Potts model



(a) Execution time (b) Number of updates

Figure 6: The results of the evaluation of the algorithms on decoding LDPC code

the scaling seems to be somewhat sensitive to the choice of the parameter H. Both the synchronous and exact residual belief propagation are omitted, as the former did not consistently converge, and the latter was very slow.

An interesting insight is that the exact variants of splash and smart splash do not converge at all in single-threaded executions for some values of the parameter H, but always converge on two and more threads. Similarly, synchronous belief propagation, which has a fixed schedule, does not converge. By contrast, relaxed smart splash converged under all parameter values. We conjecture that this is due to the phenomenon observed by [20]: exact priority-based algorithms may get stuck in non-convergent cyclic schedules, and injecting randomness into the schedule may help the algorithm to 'escape' these situations. In particular, relaxation to the priority queue, i.e., sometimes executing low-priority items, can provide a such source of randomness. Similarly, an increase in the number of threads leads to the relaxation of the algorithm even for exact schedulers, as several messages are processed in parallel, not only the best one. Thus, we empirically observe that the randomness in the relaxation might help belief propagation to avoid bad cyclic schedules and, therefore, converge.

LDPC model. There are five algorithms that perform similarly (Figure 6a): synchronous belief propagation, relaxed residual belief propagation, the weight decay algorithm, relaxed smart splash with $H = 2$ and, finally, smart splash with $H = 2$. The other algorithms did not converge within our five minutes time limit per experiment.

		Message updates			
	Threads	Tree	Ising	Potts	LDPC
Exact	1	1000000	2279000	2700000	1464000
Relaxed	1	+0.14%	+0.11%	-0.01%	+0.55%
	2	+0.26%	+0.24%	+0.37%	+0.57%
	6	+0.56%	+2.50%	+2.70%	+0.64%
	10	+0.92%	+3.71%	+4.45%	+1.05%
	20	+2.08%	+5.27%	+5.87%	+1.41%
	30	+2.90%	+6.10%	+6.56%	+1.87%
	40	+3.48%	+6.52%	+7.39%	+2.35%
	50	+5.04%	+6.83%	+7.92%	+2.83%
	60	+4.96%	+7.39%	+8.28%	+3.20%
	70	+5.74%	+7.71%	+8.53%	+3.70%

Table 5: Number of additional message updates performed by relaxed residual belief propagation compared to exact residual belief propagation.

We note that synchronous belief propagation performs very well on this instance. This is not surprising, as standard belief propagation is known to perform well in LDPC decoding. Generally speaking, the necessary propagation chains seem to be very short on LDPC instances, and the synchronous algorithm parallelizes well in such cases.

E.2.2 The effects of relaxation

In Table 5, we measure how many more updates the relaxed residual algorithm needs to perform in comparison to the number of updates performed by the standard sequential residual algorithm, denoted as "baseline". We count the total number of updates only approximately: we check the convergence condition only after every 1000 iterations.

The left column indicates whether it is a baseline algorithm or the number of threads for relaxed residual belief propagation. The other columns present the numbers for each model we consider. Each cell contains the corresponding number of updates and how many more updates the relaxed version of the algorithm executed (percentage).

On one process, relaxed residual performs more updates than the baseline does, except in the case of the Potts model. It is expected since our algorithm uses relaxed Multiqueue instead of the strict priority queue. Moreover, as expected the overhead on the number of updates in comparison to the baseline increases with the number of threads. This is again due to the relaxation of the priority queue–recall that we allocate $4\times$ more queues than threads. Interestingly, this overhead is limited even on 70 threads—its maximum value is 9% maximum. This explains the good performance of our algorithm: we reduce the contention by relaxing accesses to the priority queue, while at the same time the total number of updates does not increase significantly.

E.2.3 Relaxed versus Non-Relaxed Algorithms

In Table 6, we analyze the speedups obtained by the relaxed residual algorithm relative to the best-performing non-relaxed alternative across models and thread counts. We notice that our algorithm outperforms the alternatives in most of the cases, often by a large margin—the highest speedup is of $2.85\times$, whereas the highest slow-down is of $0.47x$. Both occur on the Potts model, which is generally the most difficult instance in our tests. Overall, the combination of our relaxed scheduling framework combined with the standard residual belief propagation is clearly the algorithm of choice at high thread counts, where it consistently outperforms the alternatives; on the other hand, relaxed residual also performs reasonably well on a single thread, making it a consistently good choice all across the board.

E.2.4 Random Synchronous Algorithm

In Table 7, we present the execution time of random synchronous algorithm on 70 threads (Random Synch 70) with different values of $lowP = 0.1, 0.4$ and 0.7, where the parameter $lowP$ controls the

	Speedup			
Threads	Tree	Ising	Potts	LDPC
1	0.89x	**1.08x**	**1.04x**	**1.14x**
2	0.75x	0.51x	0.47x	**1.13x**
6	**1.20x**	0.77x	0.73x	**1.17x**
10	**1.16x**	**1.01x**	0.94x	**1.20x**
20	**1.36x**	**1.66x**	**1.89x**	**1.49x**
30	**1.38x**	**1.88x**	**1.82x**	**1.65x**
40	**1.61x**	**2.21x**	**1.90x**	**1.62x**
50	**1.91x**	**2.67x**	**2.36x**	**1.48x**
60	**1.89x**	**2.66x**	**2.85x**	**1.55x**
70	**1.61x**	**2.71x**	**2.44x**	**1.52x**

Table 6: Speedup of relaxed residual belief propagation versus the best non-relaxed alternative on different thread counts. We note that overhead of parallelization can overcome the benefits on small thread counts, as seen in the scaling experiments.

random selection fraction p in steps where the algorithm is converging slowly (see Section C.2). We compare it with the execution time of two baselines: Synchronous algorithm on 70 threads (Synch 70) and Relaxed Residual on one process (RR 1). Cells with '—' indicate executions that either take more than five minutes to run or simply do not converge.

To summarize, we did not include the execution time of random synchronous algorithm in the scaling plots since it exceeds the execution time of one of the baselines in all cases.

		Running time (s)			
Algorithm		Tree	Ising	Potts	LDPC
Synch 70		4.088	—	—	3.504
RR 1		5.579	9.012	10.583	25.663
Random Synch 70	$lowP = 0.1$	37.052	62.629	—	28.543
	$lowP = 0.4$	8.420	20.396	—	7.269
	$lowP = 0.7$	6.306	12.581	—	4.791

Table 7: Randomized synchronous algorithm versus baselines.