

GPU-Accelerated Belief Propagation for Program Analysis

Haoyu Feng
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education;
School of Computer Science
Peking University
Beijing, China
haoyufeng@stu.pku.edu.cn

Xin Zhang*
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education;
School of Computer Science
Peking University
Beijing, China
xin@pku.edu.cn

Abstract

Belief Propagation (BP) is a widely used approximate inference algorithm in probabilistic graphical models (PGMs), but it is computationally expensive when applied to large-scale program analysis. Existing GPU-based approaches are unable to support flexible update strategies and have yet to integrate logical constraints with GPU acceleration, leading to challenges in both generality and efficiency.

We present FastLBP, a GPU-accelerated BP framework for program analysis. We propose a unified representation for specifying flexible update strategies required in program analysis, along with a dependency analysis algorithm to enable parallel execution. Furthermore, we implement BP with local structures on GPUs by assigning individual threads to message computations and utilizing a memory-efficient representation. Experiments on SmartFL and Bingo show that FastLBP achieves average speedups of 17.42× and 2.82× over CPU-based approaches on SmartFL and Bingo, respectively, and 6.14× over the GPU-based approach on SmartFL, while preserving accuracy. Moreover, FastLBP supports update strategies that existing GPU-based approaches cannot support, demonstrating its improved generality for real-world program analysis.

CCS Concepts

• **Theory of computation → Program analysis**; • **Mathematics of computing → Loopy belief propagation**; *Factor graphs*; • **Computing methodologies → Massively parallel algorithms**.

Keywords

Static Analysis, Fault Localization, Probabilistic Inference, GPU Acceleration

ACM Reference Format:
Haoyu Feng and Xin Zhang. 2026. GPU-Accelerated Belief Propagation for Program Analysis. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834444

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834444

1 Introduction

Belief Propagation (BP) is a widely used approximate inference algorithm for probabilistic graphical models (PGMs) [24]. In these models, random variables are represented as nodes, and their dependencies are captured by the graph structure. The primary task of inference is to estimate the posterior marginal distributions of these variables given observed evidence. Exact inference is often computationally intractable, making approximate algorithms a more practical alternative. BP is a prominent approximate inference algorithm [22] that operates by iteratively passing messages between neighboring nodes until convergence or for a specified number of iterations. Due to its simplicity and efficiency, BP has been successfully applied in diverse fields including computer vision [9], channel coding [19], protein folding [33], and, notably, program analysis [25].

In recent years, the combination of program analysis and probabilistic reasoning has enabled the application of BP to program analysis [6, 13, 15, 16, 18, 25, 29, 32, 34, 36, 37]. For instance, Bingo [25] converts the Datalog program analysis into a directed acyclic graph, transforms the graph into a Bayesian network by attaching probabilities to logic rules, and applies BP to infer the likelihood of bug reports. SmartFL [32, 34] models program semantics as a factor graph by utilizing information from static analysis results and dynamic execution traces, and employs BP to infer the probabilities of program elements being faulty.

Program analysis on large-scale software often yields large PGMs, leading to high computational costs for BP. Reducing this inference cost is particularly important because probabilistic program analysis relies on BP to infer posterior probabilities conditioned on observations such as user feedback or dynamic execution results. Faster BP therefore reduces the time required to incorporate such observations and obtain posterior estimates.

Graphics Processing Units (GPUs), optimized for massive parallel operations, are a natural choice for reducing the inference cost of BP. This potential has motivated prior GPU-based BP approaches [1, 5, 12, 20, 26, 38].

However, existing GPU-based BP approaches suffer from two key limitations when applied to program analysis. First, they lack support for flexible message update strategies required in program analysis. In the existing BP literature, researchers have devoted considerable attention to update strategies as a means of improving BP performance. The most straightforward scheme is synchronous BP, where all messages are updated simultaneously [22]. In contrast, asynchronous schemes introduce a degree of sequential updating, such as tree-based updates [30] and greedy message selection [2, 7, 11, 20]. While synchronous BP exposes high levels of

parallelism and can approximate the correct posteriors well when it converges, asynchronous BP has been shown, both empirically and theoretically, to converge more reliably when synchronous BP fails to converge [7]. Program analysis applications often require different update strategies depending on usage scenarios, yet existing GPU-based implementations support only limited update strategies. For example, widely used tools such as PGMax [38] support only synchronous BP, restricting their applicability in program analysis.

Second, BP with local structures [31] has not yet been explored in GPU-based implementations, although it is theoretically efficient. In program analysis, probabilistic constraints often exhibit inherent local structures, where many variable assignments share the same probability value. A typical example is the probabilistic Horn constraints in the form of $p : x_1 \wedge \cdots \wedge x_n \rightarrow y$, which is ubiquitous in modeling program semantics. While traditional BP requires an exponential number of operations ($O(n \cdot 2^n)$) to marginalize over n variables in such a constraint, BP with local structures can reduce this complexity to linear time ($O(n)$) by grouping assignments with identical probabilities and extracting common terms. However, existing GPU-based approaches typically rely on dense tabular representations that fail to capture these computational redundancies. Consequently, due to these limitations in generality and algorithmic efficiency, CPU-based approaches [21, 31] still remain the preferred choice for program analysis applications.

We present FastLBP, a GPU-accelerated BP framework designed for program analysis. First, to accommodate diverse update strategies required in program analysis, we propose a unified representation for user-specified update strategies. We further develop a dependency analysis algorithm that identifies groups of messages that can be updated safely in parallel. By partitioning edges into ordered groups, FastLBP preserves the semantics of user-specified update strategies while enabling parallel execution. Second, to address the inefficiency caused by traditional BP, we incorporate BP with local structures into our GPU design. We organize messages and factor configurations in a compact layout to enable efficient memory access. We further design a GPU execution scheme that maps each message computation to a thread, where each thread evaluates a sequence of sub-messages. This design reduces the computational complexity by efficiently implementing BP with local structures.

We conduct extensive experiments to evaluate the efficiency and accuracy of our algorithm. We implement FastLBP in a prototype system and compare it with state-of-the-art approaches [31, 38]. We evaluate FastLBP using two representative approaches: SmartFL, a probabilistic fault localization approach, and Bingo, a Bayesian program analysis approach, to demonstrate the effectiveness of our approach. Experimental results show that: (1) compared to the state-of-the-art CPU-based approach [31], our approach achieves average speedups of 17.42× and 2.82× on SmartFL and Bingo, respectively, while maintaining the accuracy. Compared to the state-of-the-art GPU-based approach [38], our approach achieves an average speedup of 6.14× on SmartFL. (2) FastLBP supports update strategies used in Bingo that are not supported by existing GPU-based approaches, highlighting its advantage in enabling flexible update strategies.

In summary, our contributions are as follows:

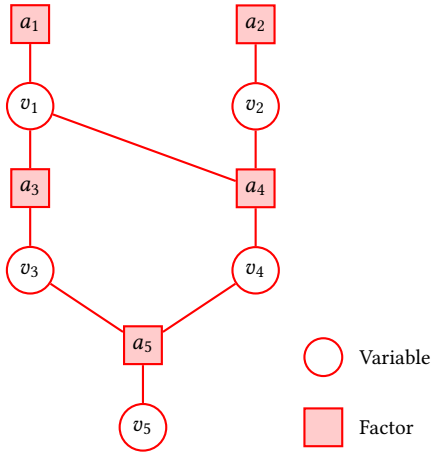


Figure 1: Example of a factor graph.

- We present FastLBP, a GPU-accelerated BP framework for program analysis, enabling flexible update strategies and efficient inference on large-scale PGMs.
- We propose a unified representation for user-specified update strategies and a dependency analysis algorithm for effective parallelization.
- We design a GPU implementation of BP with local structures by assigning each GPU thread to compute a single message and organizing the memory in a compact way.
- We implement a high-performance prototype and conduct extensive experiments to demonstrate the flexibility and efficiency of our approach.

2 Motivating Example

In this section, we illustrate the key ideas of our approach using an example generated by Bingo [25], which performs BP on PGMs derived from analysis rules.

2.1 Support for Update Strategies

BP is a message-passing algorithm widely used for performing inference on factor graphs. Figure 1 shows a subgraph of a factor graph generated by Bingo. A factor graph is a bipartite graph consisting of two types of nodes, variables and factors. For example, the variables are $V = \{v_1, v_2, v_3, v_4, v_5\}$ and the factors are $A = \{a_1, a_2, a_3, a_4, a_5\}$ in Figure 1. BP operates by iteratively passing messages along edges between variables and factors. A factor-to-variable message is computed depending on incoming variable-to-factor messages, while a variable-to-factor message is computed based on incoming factor-to-variable messages. For each message update, the incoming messages are those from neighboring nodes except the target node. For example, the message from a_4 to v_1, denoted as $\mu_{a_4 \rightarrow v_1}$, depends on messages from v_2 and v_4 to a_4, denoted as $\mu_{v_2 \rightarrow a_4}$ and $\mu_{v_4 \rightarrow a_4}$. Moreover, $\mu_{v_2 \rightarrow a_4}$ depends on $\mu_{a_2 \rightarrow v_2}$, and $\mu_{v_4 \rightarrow a_4}$ depends on $\mu_{a_5 \rightarrow v_4}$. We treat variable-to-factor message computation as an intermediate step and therefore consider only dependencies among factor-to-variable messages. Before updating a factor-to-variable message, we first compute all of its required variable-to-factor

messages using the corresponding incoming factor-to-variable messages. Consequently, in the example above, $\mu_{a_4 \to v_1}$ depends on $\mu_{a_2 \to v_2}$ and $\mu_{a_5 \to v_4}$.

Update strategies play a critical role in the performance and convergence behavior of BP. They define how message updates are scheduled. For instance, synchronous BP requires all messages to be updated in parallel, whereas asynchronous BP enforces a degree of sequentialism in message updates. Although synchronous BP is well suited for parallel execution, asynchronous BP has been shown to converge more reliably [7], which is important for many program analysis applications. For example, Bingo ranks alarms according to the inferred probabilities. Asynchronous BP often produces more accurate probabilities than synchronous BP, thereby reducing the effort required for further inspection of bug reports. At the same time, synchronous BP remains useful because it runs faster and produces good results when it converges.

However, existing approaches are unable to support flexible update strategies. One line of GPU-based work designs update strategies for specific domains, such as computer vision [12] and channel code decoding [27]. PGMax [38] is a Python package for running BP efficiently on both CPUs and GPUs and supports arbitrary graphs, but it only supports synchronous BP. Another line of work studies the convergence behavior of different update strategies in sequential and parallel scenarios [2, 7, 11, 20], but lacks a general mechanism to capture user-specified dependencies among message updates. Our work differs from these works by providing a systematic way to capture update dependencies, thereby enabling a wider range of update strategies for GPU parallel computation.

We illustrate a complex update strategy that is not supported by existing GPU-based approaches. To balance convergence stability and computational efficiency, Bingo adopts a round-robin asynchronous update strategy, where messages are updated sequentially in a predefined order during each iteration. When updating a message, the most recently updated neighboring messages are used. Although this sequential schedule improves convergence behavior, it introduces strict data dependencies between message updates. These dependencies arise from both the user-specified order and the message update rules of BP. As a result, updating multiple messages simultaneously may violate the semantics of the asynchronous schedule.

Consider the factor graph in Figure 1. Suppose the factor-to-variable messages are updated in the following order during an iteration:

$$\mu_{a_1 \to v_1},\ \mu_{a_2 \to v_2},\ \mu_{a_3 \to v_1},\ \mu_{a_3 \to v_3},\ \mu_{a_4 \to v_1},$$
$$\mu_{a_4 \to v_2},\ \mu_{a_4 \to v_4},\ \mu_{a_5 \to v_3},\ \mu_{a_5 \to v_4},\ \mu_{a_5 \to v_5}.$$

Let $\mu^{i}_{a \to v}$ denote the message from factor a to variable v at iteration i. In this example, $\mu^{i+1}_{a_4 \to v_1}$ depends on $\mu^{i+1}_{a_2 \to v_2}$ because the latter is updated earlier in the schedule, and on $\mu^{i}_{a_5 \to v_4}$ because that message has not yet been updated. Such ordering constraints create dependencies that limit naive parallel execution. However, not all messages are mutually dependent. Two messages can be safely updated in parallel if neither requires the updated value of the other. In this example, $\mu^{i+1}_{a_2 \to v_2}$ is independent of $\mu^{i+1}_{a_1 \to v_1}$, and $\mu^{i+1}_{a_3 \to v_1}$ is independent of both. Therefore, these messages can be updated concurrently without violating the asynchronous semantics. However, existing GPU-based approaches fail to capture such dependencies.

To enable parallel execution while preserving the semantics of different user-specified update strategies, our approach first provides a unified representation of update strategies, namely a preordered set $(E, \lesssim)$ over the edge set E of the factor graph. Intuitively, specifying $e_1 \lesssim e_2$ means that, in a single iteration, the update of the message on edge e_1 must happen no later than that on edge e_2. We then perform dependency analysis that captures both the update strategy constraints and BP computation dependencies. Based on this analysis, messages can be partitioned into ordered groups that can be updated in parallel.

In this example, our approach represents the update strategy as a total order induced by the preorder according to the updating sequence:

$$(a_1, v_1) \prec (a_2, v_2) \prec (a_3, v_1) \prec (a_3, v_3) \prec (a_4, v_1) \prec$$
$$(a_4, v_2) \prec (a_4, v_4) \prec (a_5, v_3) \prec (a_5, v_4) \prec (a_5, v_5).$$

Our approach then performs a topological traversal on the Hasse diagram of this order and partitions the messages into groups. Specifically, for $(a, v) \prec (a', v')$, our approach ensures that if $\mu_{a' \to v'}$ depends on $\mu_{a \to v}$ in the BP computation, then $\mu_{a \to v}$ must be computed strictly earlier than $\mu_{a' \to v'}$; otherwise, the two updates can be performed in parallel. First, we can update $\mu_{a_1 \to v_1}$, $\mu_{a_2 \to v_2}$, and $\mu_{a_3 \to v_1}$ in parallel, as we discussed earlier in this section. Next, we place $\mu_{a_3 \to v_3}$ into a new group because it depends on $\mu_{a_1 \to v_1}$. We then add $\mu_{a_4 \to v_1}$, $\mu_{a_4 \to v_2}$, and $\mu_{a_4 \to v_4}$ to the same group one by one, because these four messages are mutually independent. But we place $\mu_{a_5 \to v_3}$ into another group because it depends on $\mu_{a_4 \to v_4}$. Finally, $\mu_{a_5 \to v_4}$ and $\mu_{a_5 \to v_5}$ can be placed in the same group as $\mu_{a_5 \to v_3}$. As a result, the messages can be partitioned into three groups:

$$\{\mu_{a_1 \to v_1},\ \mu_{a_2 \to v_2},\ \mu_{a_3 \to v_1}\},$$
$$\{\mu_{a_3 \to v_3},\ \mu_{a_4 \to v_1},\ \mu_{a_4 \to v_2},\ \mu_{a_4 \to v_4}\},$$
$$\{\mu_{a_5 \to v_3},\ \mu_{a_5 \to v_4},\ \mu_{a_5 \to v_5}\}.$$

Updating each group in parallel, while processing the groups sequentially, preserves the semantics of the original asynchronous schedule and exposes substantial parallelism.

2.2 Implementing BP with Local Structures

Many probabilistic constraints arising in program analysis exhibit local structures [31]. Local structures occur when multiple assignments of variables share the same probability value in a factor. Instead of representing the factor as a full table over all assignments, these repeated patterns can be grouped and computed more efficiently. A common example in program analysis is the probabilistic Horn constraint $p : x_1 \wedge x_2 \wedge \cdots \wedge x_n \to y$. Such a constraint states that if all premises hold, the conclusion holds with probability p, and 0 otherwise. When the constraint is encoded as a factor in a factor graph, the resulting factor is represented by a factor function f_a, which maps each joint assignment $(x_1, x_2, \ldots, x_n, y)$ to a probability value. Many such assignments of the variables share identical probabilities, constituting the local structure in f_a. Given f_a, traditional BP computes a factor-to-variable message by enumerating all assignments of neighboring variables, resulting in

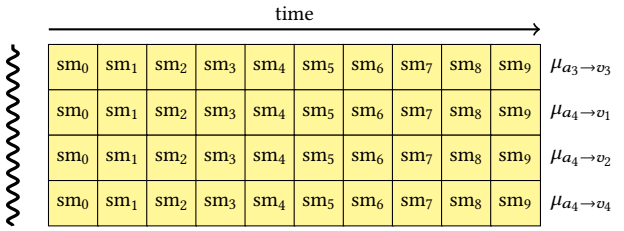


Figure 2: The computation process of our approach. Each row represents one GPU thread computing the message shown on the right. Within each row, $\mathrm{sm}_0, \ldots, \mathrm{sm}_9$ denote sub-message computations executed sequentially from left to right.

an exponential complexity of $O(n \cdot 2^n)$. BP with local structures exploits the repeated patterns in the factor to group assignments with identical probabilities, reducing the computation to linear time $O(n)$.

To leverage this advantage on GPUs, we extend the message-level parallelization of BP to support BP with local structures. We assign each GPU thread to compute a single factor-to-variable message. In BP with local structures, a factor-to-variable message is computed as the sum of several sub-messages. Each sub-message corresponds to a group of assignments that share the same probability value. It is either zero or computed as a constant multiplied by the product of incoming variable-to-factor messages. For instance, for the constraints generated by Bingo, the variables are binary (domain $\{0, 1\}$). Each message has two values, one for each possible assignment of the corresponding variable. According to the local structures in Bingo, each value is computed as the sum of five sub-messages, requiring ten sub-message computations for the entire message. Figure 2 illustrates the computation process for the second message group from the running example in Section 2.1:

$$\{\mu_{a_3\to v_3},\ \mu_{a_4\to v_1},\ \mu_{a_4\to v_2},\ \mu_{a_4\to v_4}\}.$$

Another problem is how to represent factor graphs efficiently, as those generated by program analysis applications are typically large and sparse. To achieve memory-efficient storage and efficient access, we adopt the Compressed Sparse Row (CSR) [28] format to represent messages. In CSR, a sparse matrix is stored as a flat row-major array. To compute a factor-to-variable message $\mu_{a\to v}$, we require all incoming variable-to-factor messages to factor a. Therefore, the variable-to-factor message matrix is organized such that rows are indexed by factor identifiers, grouping messages associated with the same factor contiguously. Conversely, to compute a variable-to-factor message $\mu_{v\to a}$, we require all incoming factor-to-variable messages to variable v, and thus the factor-to-variable message matrix is indexed by variable identifiers. For example, the first row (corresponding to v_1) contains $\{\mu_{a_1\to v_1}, \mu_{a_3\to v_1}, \mu_{a_4\to v_1}\}$, and the second row (corresponding to v_2) contains $\{\mu_{a_2\to v_2}, \mu_{a_4\to v_2}\}$. Additionally, factor configurations are stored in a similar manner to variable-to-factor messages. Unlike standard BP, which enumerates over joint assignments of variables, BP with local structures computes sub-messages based on an indicator specifying whether a variable is fixed. Instead of using a dense tabular representation, we store these configurations in a flat array. By providing each thread with the corresponding edge, factor, and variable identifiers, we can efficiently access the required messages and factor configurations. Figure 3 illustrates the memory layout of the factor graph in Figure 1 and how $\mu_{a_4\to v_1}$ accesses the required messages and factor configurations.

3 Preliminaries

In this section, we introduce the background of our approach, including formal definitions of factor graphs and belief propagation, as well as a brief overview of GPU programming.

3.1 Factor Graph

A factor graph is a special type of PGM. Common PGMs such as Bayesian networks and Markov random fields can be converted into factor graphs, enabling a unified approach to inference. A factor graph is a bipartite graph representing the factorization of a joint probability distribution. Given a set of random variables $\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$, suppose their joint probability distribution can be factorized into a product of functions:

$$P(\mathbf{X}) = \frac{1}{\mathcal{Z}} \prod_{a\in A} f_a(\mathbf{X}_a),$$

where $\mathbf{X}_a \subseteq \mathbf{X}$ is a subset of $\mathbf{X}$, and $\mathcal{Z}$ is the normalization constant. The corresponding factor graph $G = (V, A, E)$ consists of variable nodes $V = \{v_1, v_2, \ldots, v_n\}$, factor nodes $A = \{a_1, a_2, \ldots, a_m\}$, and edges $E \subseteq V \times A$. Each node v_i corresponds to the random variable X_i (or X_{v_i} for notational convenience) and each node a_j corresponds to the function f_{a_j}. An edge $(v_i, a_j) \in E$ exists if and only if $X_i \in \mathbf{X}_{a_j}$.

3.2 Belief Propagation

Belief Propagation (BP) is a message-passing algorithm for performing inference on factor graphs. BP iteratively exchanges messages between variables and factors along the edges of a factor graph and computes marginal distributions of variables with these messages. For each edge $(v, a) \in E$, two types of messages are defined, a message $\mu_{v\to a}$ from v to a, and a message $\mu_{a\to v}$ from a to v. Both types of messages, $\mu_{v\to a}, \mu_{a\to v} : \mathrm{Range}(X_v) \to \mathbb{R}$, are real-valued functions whose domain is the set of values that can be taken by the random variable associated with v.

The message sent from a variable v to a neighboring factor a is defined as the product of the incoming factor-to-variable messages:

$$\mu_{v\to a}(x_v) = \prod_{a^*\in N_V(v)\setminus\{a\}} \mu_{a^*\to v}(x_v), \tag{1}$$

where $N_V(v)$ denotes the set of neighboring factors of v.

The message sent from a factor a to a neighboring variable v is defined as the product of the factor function with the incoming variable-to-factor messages:

$$\mu_{a\to v}(x_v) = \sum_{\mathbf{x}_a\setminus x_v} f_a(\mathbf{x}_a) \prod_{v^*\in N_A(a)\setminus\{v\}} \mu_{v^*\to a}(x_{v^*}), \tag{2}$$

where $N_A(a)$ denotes the set of neighboring variables of a, and $\sum_{\mathbf{x}_a\setminus x_v}$ denotes summation over all possible values of $\mathbf{x}_a$ while keeping x_v fixed.

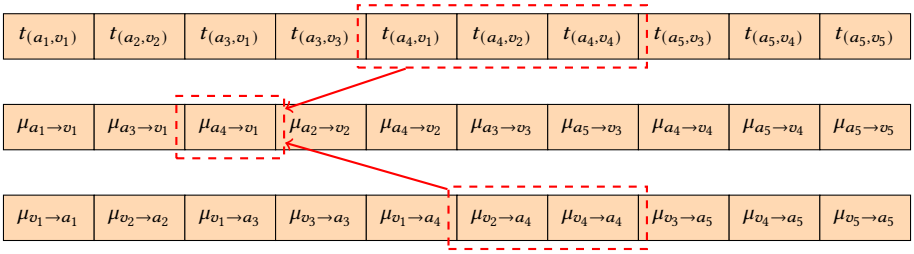


Figure 3: Example of the memory layout and access. Factor configurations for factor a and variable v are denoted as $t_{(a,v)}$.

After convergence or a certain number of iterations, the marginal probability of a variable v is computed as

$$P(X_v = x_v) = \frac{\prod_{a\in N_V(v)} \mu_{a\to v}(x_v)}{\sum_{x_v} \prod_{a\in N_V(v)} \mu_{a\to v}(x_v)}. \tag{3}$$

If the factor graph is a tree, BP yields exact marginal distributions after a finite number of iterations. For graphs with cycles, BP is not guaranteed to converge but often provides good approximate solutions in practice.

3.3 BP with Local Structures

The main computational bottleneck of BP lies in the factor-to-variable message $\mu_{a\to v}$ in Equation (2), which requires enumeration over an exponential number of factor values. Local structures in a constraint can be exploited to simplify this computation by grouping assignments that share the same factor value. The computation of $\mu_{a\to v}$ is then reduced to the summation of multiple sub-messages based on the local structure of the factor function. Formally,

$$\mu_{a\to v}(x_v) = \sum_{k=1}^{l} \sum_{\mathbf{x}_a \setminus x_v} t_k(\mathbf{x}_a) \prod_{v^*\in N_A(a)\setminus\{v\}} \mu_{v^*\to a}(x_{v^*}), \tag{4}$$

where each t_k specifies a weight together with indicators that determine whether a variable is fixed to a particular value or left unrestricted. If a variable is unrestricted, its incoming message is summed over all possible values, as shown in lines 9-10 of Algorithm 1. According to Equation (4), each message is computed using Algorithm 1, where v is the target variable index, x_v is the target value, $p[l]$ denotes the weights associated with the factor, $b = |N_A(a)|$ is the number of variables adjacent to a, and l is the number of sub-messages. The indicator arrays are $indicator[b][l]$, and $\mu[b][r]$ stores messages from $N_A(a)$ to a. As shown in Algorithm 1, the complexity is reduced from $O(b \cdot r^b)$ to $O(l \cdot b \cdot r)$, where b can be large.

3.4 GPU Programming

We introduce GPU programming using Compute Unified Device Architecture (CUDA) as a representative example. Modern GPUs consist of multiple streaming multiprocessors (SMs), each of which contains many compute units for massive parallel execution. The functions executed on GPUs are referred to as kernels. A kernel is launched to run multiple threads in parallel. These threads are grouped into thread blocks, while these blocks together form a grid. Each thread block is scheduled to run on a single SM. Within a thread block, threads are divided into groups of 32 threads known as warps. Threads within a warp execute instructions in a Single-Instruction Multiple-Threads (SIMT) paradigm.

Algorithm 1 Message Calculation

Require: $v, x_v, p[l], indicator[b][l], \mu[b][r]$
Ensure: $message$
1: $message \leftarrow 0$
2: **for** $t \leftarrow 1$ to l **do**
3: **if** $indicator[v][t] \neq x_v \wedge indicator[v][t] \neq -1$ **then**
4: **continue**
5: **else**
6: $sm \leftarrow p[t]$
7: **for** $j \leftarrow 1$ to b, $j \neq v$ **do**
8: $c \leftarrow indicator[j][t]$
9: **if** $c = -1$ **then**
10: $sm \leftarrow sm \times \sum_{k=1}^{r} \mu[j][k]$
11: **else**
12: $sm \leftarrow sm \times \mu[j][c]$
13: **end if**
14: **end for**
15: $message \leftarrow message + sm$
16: **end if**
17: **end for**
18: **return** $message$

4 FastLBP Framework

In this section, we first describe the overall workflow of FastLBP (Section 4.1). We then present two key designs of FastLBP: (1) the update strategy representation with the dependency analysis algorithm (Section 4.2), and (2) the implementation of BP with local structures on GPUs (Section 4.3).

4.1 Overall Workflow

Figure 4 shows the heterogeneous workflow of FastLBP. Given a factor graph derived from probabilistic program analysis and a user-defined update strategy, FastLBP ultimately produces marginal probabilities of variables as results. The workflow consists of the following four major components.

Initialization. FastLBP first takes as input a factor graph, typically generated by probabilistic program analysis frameworks such

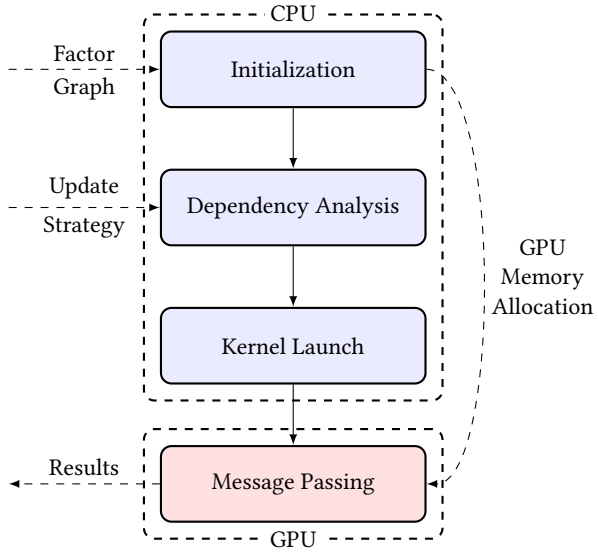


Figure 4: Workflow of FastLBP.

as Bingo or SmartFL. Based on the graph structure and factor functions, FastLBP allocates GPU memory for messages using a CSR-based layout. This representation enables compact storage and efficient access for sparse message structures, avoiding the prohibitive memory cost of dense adjacency matrices on GPUs.

Dependency Analysis. FastLBP requires the user to specify an update strategy. FastLBP represents the update strategy as a preordered set $(E, \lesssim)$ over graph edges. This abstraction uniformly captures different update strategies. It then analyzes the dependencies implied by this preordered set and the BP computation process. The result is a partitioning of edges into ordered groups, each representing a set of factor-to-variable messages that can be updated simultaneously.

Kernel Launch. Based on the results of dependency analysis, FastLBP organizes message updates into batches corresponding to the computed edge groups. For each group, a GPU kernel is launched, where each thread is assigned to compute a single message. The required factor, variable, and edge identifiers are passed to each thread to enable efficient data access. This batching strategy reduces kernel launch overhead and ensures that each kernel exposes sufficient parallelism for efficient GPU execution.

Message Passing. Finally, FastLBP performs iterative message passing on the GPU. In each iteration, kernels are launched following the dependency order, and messages are updated accordingly using the message-level parallel implementation. The process continues until convergence or a user-specified number of iterations, after which marginal probabilities are computed and returned to the analysis framework.

4.2 Representation and Dependency Analysis

In this section, we first briefly introduce the representation of update strategies, then formalize its definition and semantics, and finally present the dependency analysis algorithm.

Table 1: Update Strategies

Strategy Name	Description
PARALL	Update all messages in parallel
SEQFIX	Sequentially update messages in a fixed order
TOPO	Update messages according to topological order
...	...

4.2.1 Update Strategies. An update strategy in FastLBP is represented as a preordered set $(E, \lesssim)$ over graph edges. Table 1 describes several typical update strategies. Users can specify the strategy name, and FastLBP encodes it together with the edge set into a unified intermediate representation, i.e., a preordered set. Additionally, users may define a custom preordered set directly as the update strategy. In practice, this is typically a small adapter that maps message edges to their order in the desired schedule.

4.2.2 Representation Formalization. To perform dependency analysis in a uniform manner, we introduce a unified representation of update dependencies. Our key insight is that update strategies can be uniformly characterized as preorders over the edge set of a factor graph. This abstraction allows different scheduling semantics to be handled within a single formal framework.

We first characterize the dependencies induced by different update strategies. The problem lies in whether a message depends on messages from the current iteration or the previous iteration. We focus on factor-to-variable updates, since they are directly used to compute marginal probabilities. We assume that each factor-to-variable message is updated once per iteration. When computing a factor-to-variable message, all incoming variable-to-factor messages must first be computed. These variable-to-factor messages, in turn, depend on the incoming factor-to-variable messages. By treating the computation of variable-to-factor messages as an intermediate step, the essential dependency lies in whether the required factor-to-variable messages come from the current or the previous iteration. Formally, by substituting Equation (1) into Equation (2) and introducing superscripts to indicate iteration indices, we obtain:

$$\mu_{a\to v}^{i+1}(x_v) = \sum_{\mathbf{x}_a \setminus x_v} f_a(\mathbf{x}_a) \prod_{v^* \in N_A(a)\setminus\{v\}} \prod_{a^* \in N_V(v^*)\setminus\{a\}} \mu_{a^*\to v^*}^{i+\delta((a,v),(a^*,v^*))}(x_{v^*}), \quad (5)$$

where $\delta : E \times E \to \{0, 1\}$ indicates whether a dependency comes from the current iteration ($\delta = 1$) or the previous iteration ($\delta = 0$).

This expression shows that the update of a message on edge (a, v) depends only on a specific subset of edges determined by the graph structure. To make this explicit, we define the edge dependency neighborhood.

Definition 4.1 (Edge Dependency Neighborhood). Define the edge dependency neighborhood function $N_E : E \to \mathcal{P}(E)$ as

$$N_E((a, v)) \stackrel{\text{def}}{=} \bigcup_{v^* \in N_A(a)\setminus\{v\}} \bigcup_{a^* \in N_V(v^*)\setminus\{a\}} \{(a^*, v^*)\}.$$

The key to defining a legitimate update strategy is specifying a reasonable δ. We now give the definition of intermediate representation used in FastLBP for update strategies, along with the corresponding definition of δ.

Definition 4.2 (Intermediate Representation of Update Strategies). The intermediate representation of an update strategy is defined as a preordered set $(E, \lesssim)$ on the edge set E of a factor graph, such that for all $e_1 \in E$ and for all $e_2 \in N_E(e_1)$, either $e_1 \lesssim e_2$ or $e_2 \lesssim e_1$ holds. Then, given $(E, \lesssim)$, for all $e_1, e_2 \in E$,

$$\delta(e_1, e_2) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if } e_2 \prec e_1 \text{ and } e_2 \in N_E(e_1), \\ 0, & \text{otherwise,} \end{cases}$$

where $\prec$ is the strict partial order induced by $\lesssim$. The message update is performed according to Equation (5).

Intuitively, specifying that $e_2 \lesssim e_1$ in the preorder means that, during one iteration, the message on edge e_1 cannot be updated before the message on edge e_2. Specifically, $e_2 \prec e_1$ means the message on e_2 must be updated before the message on e_1 if e_1 depends on e_2 ($\delta(e_1, e_2) = 1$).

The following Theorem 4.3 shows that the function δ defined in Definition 4.2 is legitimate.

Theorem 4.3 (Legality of Update Strategy Representation). *If the function δ is given by Definition 4.2, then there does not exist a sequence of edges $e_{j_1}, e_{j_2}, \ldots, e_{j_p} \in E$ (with $p > 1$) such that*

$$e_{j_1} = e_{j_p} \quad \text{and} \quad \forall\, 1 \le i < p,\ \delta(e_{j_i}, e_{j_{i+1}}) = 1.$$

Proof. We proceed by contradiction. Suppose such a sequence exists.

When $p = 2$, we have $e_{j_1} = e_{j_2}$ and $\delta(e_{j_1}, e_{j_2}) = 1$. However, by Definition 4.2, $\delta(e_{j_1}, e_{j_2}) = 1$ implies $e_{j_2} \prec e_{j_1}$. Since $e_{j_1} = e_{j_2}$, this implies $e_{j_1} \prec e_{j_1}$, which contradicts the irreflexivity of the strict partial order $\prec$.

For $p > 2$, by Definition 4.2, $\delta(e_{j_i}, e_{j_{i+1}}) = 1$ for all $1 \le i < p$ implies $e_{j_i} \prec e_{j_{i+1}}$. Thus, by transitivity of $\prec$, it follows that $e_{j_1} \prec e_{j_p}$. But since $e_{j_1} = e_{j_p}$, this yields $e_{j_1} \prec e_{j_1}$, again contradicting the irreflexivity of $\prec$.

Therefore, no such cyclic sequence can exist, completing the proof. □

Theorem 4.3 shows that the update strategy defined by Definition 4.2 contains no cyclic dependencies. That is, within a single iteration, starting from any message, although it may depend on other messages computed in the current iteration, and those messages in turn may depend on yet others from the same iteration, this chain of dependencies must eventually terminate at messages whose values are taken from the previous iteration. This termination is guaranteed by the finiteness of the edge set E. Therefore, the current iteration's results can always be computed starting from the messages of the previous iteration, which confirms the well-definedness of Definition 4.2.

Definition 4.2 captures a sufficiently broad subset of valid update strategies. For example, PARALL can be represented by a preorder in which all edges are in the same equivalence class, SEQFIX corresponds to a total order over all edges, and TOPO is expressed as a topological order.

Algorithm 2 Dependency Analysis

Require: A preordered set $(E, \lesssim)$ represented as a Hasse diagram, where $E = \{e_1, \ldots, e_N\}$
Ensure: A sequence of edge sets $S = [s_1, \ldots, s_k]$

```
1:  e_1, ..., e_N ← TopologicalSort(E, ≲)
2:  S ← ∅, NextSet ← ∅
3:  for i ← 1 to N do
4:      for e in NextSet do
5:          if e ∈ N_E(e_i) and e ≺ e_i then
6:              S.append(NextSet)
7:              NextSet ← ∅
8:              break
9:          end if
10:     end for
11:     NextSet ← NextSet ∪ {e_i}
12: end for
13: S.append(NextSet)
14: return S
```

4.2.3 Dependency Analysis Algorithm. We now present the dependency analysis algorithm based on the intermediate representation of the update strategy.

In Algorithm 2, TopologicalSort on line 1 refers to a topological sort performed on the Hasse diagram of the partial order on the set of equivalence classes. Edges in the same equivalence class can be in any order. After the topological sort, the algorithm greedily groups the edges into batches. When processing the edge e_i, if it depends on any edge already in $NextSet$, then the current $NextSet$ is output as a batch, and a new batch is started. Finally, the algorithm outputs a sequence of edge sets S. Updating factor-to-variable messages according to this sequence yields the expected result under the specified update strategy. The overall time complexity of the dependency analysis is $O(|E|^2)$ in the worst case, as the algorithm performs a quadratic scan in the number of edges. Since this analysis is executed only once prior to message iterations, its cost is negligible compared to the overall inference procedure.

Theorem 4.4 establishes the correctness of the algorithm, guaranteeing that all message dependencies for a given batch are correctly sourced, either from previously computed batches in the current iteration or from the previous iteration, as required.

Theorem 4.4 (Correctness). *The dependency analysis algorithm is correct in the following sense: for all $s_j \in \{s_1, \ldots, s_k\}$, for all $e \in s_j$, and for all $e' \in N_E(e)$,*

(1) if $e' \prec e$, then $e' \in \bigcup_{1 \le l < j} s_l$.
(2) if $e \lesssim e'$, then $e' \in \bigcup_{j \le l \le k} s_l$.

Proof. We prove the theorem in two parts:

- Since the batches are constructed according to a topological order (line 1), if $e' \prec e$, then $e' \in \bigcup_{1 \le l \le j} s_l$. Moreover, because $e' \in N_E(e)$ and $e' \prec e$, lines 5–9 ensure that $e' \notin s_j$. Combining both facts, we conclude that $e' \in \bigcup_{1 \le l < j} s_l$.
- If $e \lesssim e'$, by the topological sort (line 1), e' cannot appear in any batch preceding s_j. Therefore, $e' \in \bigcup_{j \le l \le k} s_l$.

□

4.3 Implementation of BP with Local Structures

To leverage the computational benefits of BP with local structures on GPUs, we develop a message-level parallel implementation that maps batches of message updates to GPU execution.

4.3.1 Message-Level Parallelization. We adopt a message-level parallelization strategy, where each GPU thread is responsible for computing a single message on an edge. For each edge $(a, v) \in E$, a thread computes either a variable-to-factor message $\mu_{v\to a}$ or a factor-to-variable message $\mu_{a\to v}$. To compute a batch of factor-to-variable messages obtained from dependency analysis, we organize the computation into two phases. This separation avoids read-after-write dependencies and ensures that all messages can be computed fully in parallel based on the precomputed messages. In the first phase, we compute all required variable-to-factor messages $\mu_{v\to a}$ in a kernel launch. According to Equation (1), each $\mu_{v\to a}$ is computed as the product of incoming factor-to-variable messages and can be computed independently. We assign one thread per edge to compute these messages in parallel. In the second phase, we compute the factor-to-variable messages $\mu_{a\to v}$ using BP with local structures. As shown in Algorithm 1, each message is evaluated as the sum of multiple sub-messages. Each thread iterates over the sub-messages, checks the indicator conditions, and accumulates the results accordingly. This design exposes massive parallelism across edges while maintaining good data locality within each thread.

4.3.2 Memory Organization. Efficient memory organization is critical for achieving high performance on GPUs, especially for large-scale factor graphs generated by program analysis. In FastLBP, we adopt a memory layout that ensures both compact storage and efficient memory access. We store all messages in global GPU memory using a CSR format. Specifically, we maintain two separate CSR arrays for variable-to-factor messages and factor-to-variable messages, respectively. Each CSR representation consists of two key auxiliary arrays for indexing. One is the row offset array that represents the starting position of each row. The other is the column index array that contains the column indices of the corresponding value. To facilitate efficient access patterns, we align the CSR rows with the computation direction. When computing a factor-to-variable message $\mu_{a\to v}$, all required variable-to-factor messages associated with factor a are stored contiguously by using factor identifiers as row indices. Conversely, when computing $\mu_{v\to a}$, we organize messages using variable identifiers as row indices. This layout allows each thread to access all necessary inputs via sequential memory reads, reducing indexing overhead.

For BP with local structures, instead of storing factor functions as dense tables, we store them as a flattened array of sub-message indicators to effectively support BP with local structures. We represent the indicator array as $indicator[N][L]$, where $N = |E|$ is the number of edges and $L = \max_{a\in A} l_a$ is the maximum number of terms among all factors. For efficient GPU storage and access, we organize this array in a CSR-like format using factor identifiers as row indices, so that all indicators associated with the same factor are stored contiguously. For each edge $e = (a, v)$, $indicator[e]$ is an integer array of length L. For $i \le l_a$, $indicator[e][i]$ follows the original semantics in Section 3.3. That is, $indicator[e][i] = c$ if v is constrained to value c in the i-th term, and $indicator[e][i] = -1$ otherwise. For $i > l_a$, we pad $indicator[e][i]$ with a sentinel value (e.g., -2) indicating an invalid term. This representation avoids storing factor tables of exponential size and enables each thread to perform BP with local structures, significantly reducing memory footprint and improving computational efficiency.

4.3.3 Kernel Launch. To execute message updates efficiently on GPUs, we organize computation into batches derived from dependency analysis. Each batch consists of messages that can be computed independently without synchronization. For each batch, we launch a GPU kernel where each thread is assigned a single message computation. Each thread is provided with the corresponding factor, variable, and edge identifier. The edge identifier is computed using the CSR representation and determines the location for storing the output message. These identifiers allow each thread to efficiently locate input messages and write results without contention. Batching not only reduces kernel launch overhead but also improves efficiency by exposing massive parallelism.

5 Experimental Evaluation

In this section, we aim to answer the following research questions:

RQ1. How effective is FastLBP in improving the efficiency of program analysis?

RQ2. How important is supporting flexible update strategies for program analysis?

We describe our experimental setup in Section 5.1, and answer the above questions in Section 5.2 and Section 5.3, respectively.

5.1 Experimental Setup

We implement FastLBP in C++ and CUDA, reusing the frontend of libDAI [21] for reading factor graphs. We conduct all experiments on a Linux machine with 2.40 GHz Intel Xeon processors, 256 GB RAM, and a single NVIDIA GeForce RTX 4090 GPU.

5.1.1 Program Analysis Applications. We evaluate FastLBP on two probabilistic program analysis applications, SmartFL [34] and Bingo [25], which target fault localization and alarm prioritization, respectively.

SmartFL performs probabilistic fault localization. It constructs a factor graph from static analysis information and dynamic execution traces, and uses BP to infer the posterior probability that each program element is faulty. Based on these probabilities, SmartFL ranks suspicious program elements to help users inspect likely faulty locations.

Bingo prioritizes alarms produced by Datalog program analysis. In our evaluation, the underlying analysis is a flow- and context-sensitive datarace analysis [23]. Bingo constructs a PGM that captures the probabilistic dependencies among analysis facts and rules, and uses BP to estimate which alarms are likely true. It repeatedly presents the highest-ranked alarm to the user for inspection and updates the posterior probabilities based on the feedback about whether the inspected alarm corresponds to a true bug.

5.1.2 Benchmarks. We use the same benchmarks as in the previous work [31], which we describe below.

For SmartFL, we use the Defects4J benchmark suite [14]. After filtering invalid cases where the factor graphs cannot be constructed

by SmartFL, we obtain 194 factor graphs. On average, each benchmark instance contains about 102K lines of code, and the generated factor graph contains approximately 258K variables and 812K edges.

For Bingo, we use the benchmark suite consisting of eight Java programs collected from previous work [8, 35] and the DaCapo benchmark suite [3]. The average program size is about 81K lines of code. The corresponding factor graphs generated by Bingo contain on average 62K variables and 159K edges.

5.1.3 Baselines. We compare FastLBP against two baselines: Wu et al. [31] and PGMax [38]. Wu et al. [31] implemented their approach on SmartFL and libDAI, evaluating it on SmartFL and Bingo, respectively. Both implementations are single-threaded. We adopt these implementations as the state-of-the-art CPU-based baselines. PGMax [38] is a Python/JAX [4] package that implements BP for discrete factor graphs with arbitrary shapes. It only supports synchronous BP with a fixed number of iterations. PGMax can run on GPUs since JAX supports GPU acceleration. We adopt PGMax as the state-of-the-art GPU-based baseline in our evaluation.

5.2 Efficiency

5.2.1 Efficiency on SmartFL. To evaluate the efficiency of FastLBP, we compare it with Wu et al. [31] and PGMax [38] on SmartFL. Since the original setting of SmartFL adopts synchronous BP, all approaches use synchronous BP in this experiment to ensure a fair comparison. The number of message-passing iterations is set to 500, at which point BP has converged on most factor graphs (190/194).

Overall, FastLBP significantly outperforms both baselines. Figure 5 shows the inference time against the number of variables in the factor graphs. Compared to Wu et al. [31], FastLBP achieves a 17.42× speedup on average (geometric mean). Since Wu et al. [31] already uses BP with local structures, the speedup primarily comes from GPU parallel execution. The advantage becomes more pronounced as the size of the factor graph increases. For small graphs (around 10^2 to 10^3 variables), the performance differences between them are relatively small due to the limited amount of parallelism available. However, as the number of variables grows beyond 10^4, FastLBP exhibits significantly lower inference time than Wu et al. [31].

FastLBP also outperforms PGMax, achieving an average speedup of 6.14×. This improvement highlights the benefit of FastLBP's efficient implementation of BP with local structures, which PGMax does not support. The speedup does not increase monotonically with the number of variables because variable count is not the only factor that determines GPU performance. One reason is that PGMax incurs JAX compilation overhead, whose relative impact is greater on smaller graphs. Another reason is that graph size affects the GPU utilization of the two approaches differently. Moreover, FastLBP's benefit from exploiting local structures depends more directly on factor degree than on the total number of variables. Therefore, FastLBP may benefit more from local structures on a smaller graph with high-degree factors than on a larger graph dominated by low-degree factors. We note that `Chart-15` is not shown in Figure 5 because PGMax runs out of memory on this factor graph. This happens because PGMax adopts a tabular representation of factors, while `Chart-15` contains factors connected to many variables.

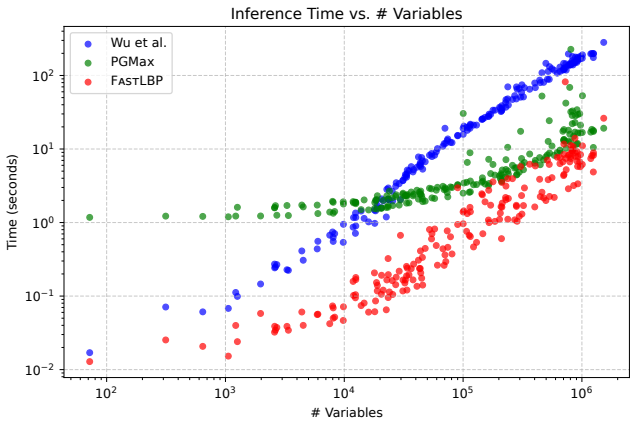


Figure 5: Comparison of Inference Time on SmartFL. Each point corresponds to a test program.

We also check the correctness of FastLBP on SmartFL. As shown in Theorem 4.4, FastLBP is mathematically equivalent to sequential implementations, but minor floating-point errors might still occur due to implementation details. We quantify the differences in marginal probabilities using relative error. Across all test cases, FastLBP has relative errors less than 10^{-8} compared to Wu et al. [31], confirming its correctness.

5.2.2 Efficiency on Bingo. Since Bingo performs BP before each round of user feedback, efficient inference is crucial for identifying true alarms quickly. We compare FastLBP with Wu et al. [31] on Bingo. We adopt the asynchronous BP introduced in Section 2.1, which is also the update strategy originally used by Bingo. Empirically, BP does not converge in most factor graphs generated by Bingo, making asynchronous update strategy more suitable for this workload. We do not compare PGMax in this experiment because it only supports synchronous BP. We follow the original stopping criterion, limiting the number of message-passing iterations to between 500 and 1,000 when BP does not converge.

Figure 6 shows the number of true alarms inspected over the accumulated inference time for each benchmark. On most benchmarks, FastLBP identifies all true alarms in significantly less time. The only exception is `weblech`, where the factor graph is relatively small (with 696 variables). Under the constraints of the asynchronous update strategy, FastLBP cannot achieve high parallelism on such small graphs. The benefits of GPU parallelism do not outweigh the overhead introduced by dependency analysis and memory transfers. As the graph size increases, the advantage of FastLBP becomes increasingly evident. On `hedc`, FastLBP is slightly faster than Wu et al. [31]. On the remaining benchmarks (`luindex`, `jspider`, `avrora`, `xalan`, `sunflow`, and `ftp`), FastLBP runs several times faster. Although FastLBP is marginally slower on small graphs, this has negligible practical impact because the total BP time on such graphs is inherently small (less than 3 seconds on `weblech`). Thus, the performance difference does not significantly affect user experience. In contrast, users are primarily concerned with execution time on large-scale graphs, where the total inference time can extend to

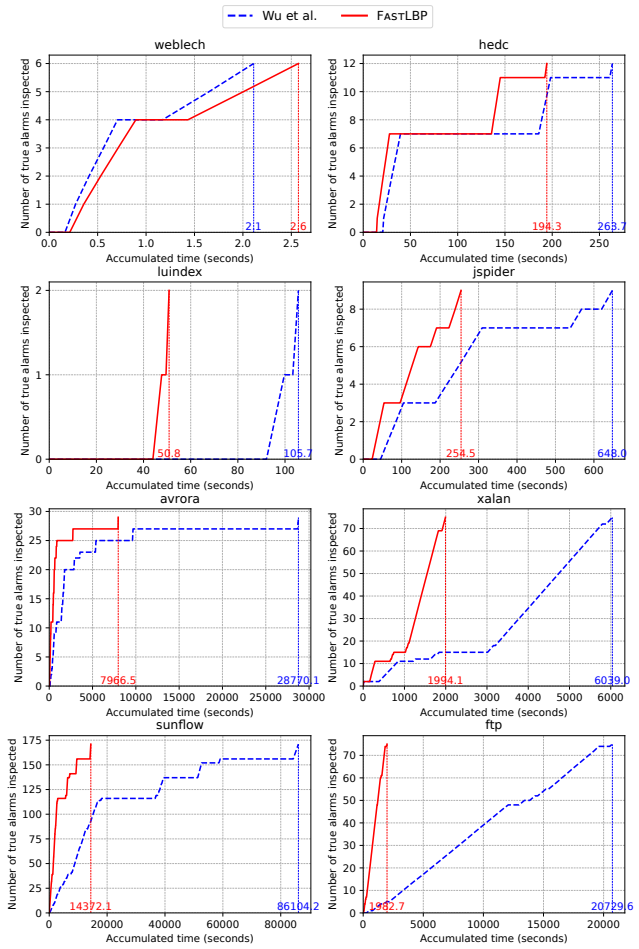


Figure 6: Comparison of Inference Time on Bingo.

several hours or even exceed one day. It is precisely in these large-scale scenarios that FastLBP excels, significantly accelerating BP and substantially reducing user waiting time.

FastLBP achieves an average speedup of 2.82× in total inference time compared to Wu et al. [31]. The relatively moderate overall speedup is mainly due to two reasons. First, the asynchronous update strategy required by Bingo limits the amount of parallelism that can be exploited by GPU execution. Second, several benchmarks contain relatively small factor graphs. When excluding benchmarks where Wu et al. [31] finishes within 15 minutes (i.e., `weblech`, `hedc`, `luindex`, and `jspider`), the average speedup increases to 5.12×, with a maximum speedup of 10.46×. These results demonstrate that FastLBP effectively supports asynchronous update strategies required by program analysis applications.

5.3 The Importance of Supporting Flexible Update Strategies

To investigate the importance of supporting flexible update strategies in program analysis, we evaluate how different BP update strategies affect the quality of inference results on Bingo. In particular, we compare asynchronous BP used by FastLBP and Wu et al. [31] with synchronous BP implemented in PGMax. We do not evaluate on SmartFL because synchronous BP already converges reliably on SmartFL. We use the default stopping criterion of Bingo for FastLBP and Wu et al. [31], which stops at 500 to 1,000 iterations based on the convergence rate. Since PGMax runs BP for a fixed number of iterations, we set the number of iterations to 1,000 to match the upper bound of Bingo's stopping criterion.

Following Bingo, we employ three metrics to quantify the quality of inference results: inversion count, Rank-100%-T, and Rank-90%-T. An inversion is defined as a pair of alarms where a false alarm is ranked higher than a true alarm in the user's inspection sequence. Formally, let $l_1, \ldots, l_n \in \{0, 1\}$ denote the ground-truth labels of the alarms provided by the user. Then, the inversion count is calculated as: $\text{Inversion}(l_1, \ldots, l_n) = \sum_{i=1}^{n}(1 - l_i)\sum_{j=i+1}^{n} l_j$. Rank-100%-T and Rank-90%-T represent the numbers of interactions to identify all and 90% of the true alarms. Lower values indicate that users can prioritize and inspect true alarms more rapidly. Consequently, higher accuracy in BP results correlates with improved alarm ranking and reduced inspection effort.

Table 2 shows the results of the three approaches. Compared with Wu et al. [31], FastLBP shows only negligible differences in all three metrics. Theoretically, FastLBP should produce identical results to the sequential implementation under the same update strategy. The small discrepancies observed in practice are primarily due to floating-point errors between the two implementations. This leads to differences in the marginal probabilities, thus affecting the final alarm rankings. These results confirm that FastLBP preserves the inference quality of the sequential implementation while enabling GPU acceleration.

In contrast, PGMax exhibits significantly worse results across all metrics. The inversion count, Rank-100%-T, and Rank-90%-T increase by 256.14%, 186.68%, and 128.48%, respectively. This degradation significantly affects user experience because users must inspect many more false alarms before discovering true bugs. The main reason is that PGMax only supports synchronous BP. In program analysis frameworks such as Bingo, BP often fails to converge and tends to exhibit more oscillating behaviors under synchronous BP than asynchronous BP.

These results highlight the importance of supporting flexible update strategies for program analysis. In particular, asynchronous BP is essential for workloads where synchronous BP leads to unstable inference results. By supporting both synchronous and asynchronous update strategies, FastLBP can adapt to different workloads while maintaining both efficiency and accuracy.

6 Related Work

Our work is related to research on (1) GPU acceleration for BP, (2) parallel BP on CPUs, and (3) probabilistic program analysis using BP. We discuss the related work below.

GPU Acceleration for BP. GPU acceleration of BP has been explored in several domain-specific applications, such as computer vision [5, 12, 17] and channel coding [1, 26, 27]. These approaches design GPU implementations tailored to the characteristics of PGMs in specific domains. In contrast, our work focuses on accelerating

Table 2: Results of FastLBP and PGMax compared to Wu et al. [31]. PGMax runs out of memory (OOM) on sunflow. Average Change represents the geometric mean of the ratios, minus one.

Program	Inversion			Rank-100%-T			Rank-90%-T		
	Wu et al. [31]	FastLBP	PGMax	Wu et al. [31]	FastLBP	PGMax	Wu et al. [31]	FastLBP	PGMax
weblech	18	18	19	11	11	11	11	11	11
hedc	347	347	1109	87	87	152	65	65	151
luindex	29	29	120	17	17	62	17	17	62
jspider	72	64	398	26	20	196	26	20	196
avrora	3,187	3,048	5,625	900	885	926	293	302	381
xalan	3,110	3,890	6,306	126	141	868	117	128	122
sunflow	25,200	25,497	OOM	829	825	OOM	595	566	OOM
ftp	570	606	14,999	92	95	430	81	81	304
Average Change	-	1.69% ↑	256.14% ↑	-	1.73% ↓	186.68% ↑	-	2.37% ↓	128.48% ↑

BP in the context of program analysis, where the resulting PGMs are significantly larger in scale and more complex in graph structure. Beyond domain-specific applications, Zhou et al. [38] implement BP using JAX [4], enabling GPU acceleration for general PGMs. Compared to their work, our approach supports more sophisticated update strategies and a more efficient BP algorithm. Randomized Belief Propagation [20] randomly selects a subset of messages to update in each iteration, effectively balancing convergence behavior and parallel efficiency. In contrast, our approach supports a broader range of update strategies and can be easily adapted to incorporate their update strategy.

Parallel BP on CPUs. To improve convergence, prior work on CPU-based parallel BP has focused on designing specific update strategies – frontier-based update strategies [2, 11]. They design update strategies that use the concept of residuals [7] and update the most influential messages during each iteration to balance parallelism and convergence behavior. While effective on CPUs, these approaches incur significant overhead on GPUs [20], limiting their applicability in GPU settings. In comparison, we propose a framework that enables flexible update strategies through a unified representation and dependency analysis, capturing diverse user-specified dependencies. Our approach can also be naturally extended to CPU execution. We do not directly compare with CPU-based parallel BP approaches because our target workloads expose massive fine-grained parallelism that is better matched to GPU execution. Moreover, the per-update scheduling overhead of frontier-based approaches can become prohibitive for large factor graphs.

Probabilistic Program Analysis Using BP. Recent years have seen increasing interest in applying probabilistic reasoning to program analysis. In particular, Bayesian program analysis transforms static analysis derivation graphs into PGMs and reasons about the likelihood that a reported alarm corresponds to a real bug. Prior work in this area can be classified in four directions. The first line of work focuses on incorporating diverse forms of posterior information to refine the alarm ranking. Posterior information includes user feedback on the alarms [25], differences between the derivations of alarms [13], test execution results [6], and informal information in the program [16]. The second line explores how to refine Bayesian networks to mitigate false generalization of posterior information. Kim et al. [15] proposed an approach to learn new rules and probabilities. Zhang et al. [36] and Shi et al. [29] aimed at learning to select a suitable abstraction. In the third line, Lin et al. [18] studied alarm selection during interactive Bayesian program analysis and proposed an approach that balances exploration and exploitation instead of always selecting the alarm with the highest posterior probability. The last line applies probabilistic program analysis to augment other software engineering tasks, such as fault localization [32, 34] and fuzzing [37]. These approaches typically model program semantics as PGMs and perform BP for marginal inference. However, the resulting models are often large and complex, making BP a major performance bottleneck. Our work can accelerate these systems by providing an efficient and accurate inference library.

7 Conclusion

We present FastLBP, a GPU-accelerated belief propagation framework for program analysis. By introducing a unified representation for update strategies and a dependency analysis algorithm, FastLBP enables parallel execution for flexible update strategies. Additionally, our memory-efficient GPU implementation of BP with local structures effectively handles logical constraints in program analysis. Our evaluation on SmartFL and Bingo demonstrates that FastLBP significantly outperforms both CPU-based and GPU-based approaches in efficiency without sacrificing accuracy.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was partly supported by the National Natural Science Foundation of China (NSFC, Grant No. W2411051).

Data Availability Statement

The artifact for this paper is available at Zenodo [10]. It includes source code, scripts, and data required to reproduce the results presented in Section 5.

References

[1] Kiran Kumar Abburi. 2011. A Scalable LDPC Decoder on GPU. In *2011 24th International Conference on VLSI Design*. 183–188. doi:10.1109/VLSID.2011.44

[2] Vitalii Aksenov, Dan Alistarh, and Janne H. Korhonen. 2020. Scalable Belief Propagation via Relaxed Scheduling. In *Advances in Neural Information Processing*

Systems, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 22361–22372. https://proceedings.neurips.cc/paper_files/paper/2020/file/fdb2c3bab9d0701c4a050a4d8d782c7f-Paper.pdf

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[5] Alan Brunton, Chang Shu, and Gerhard Roth. 2006. Belief Propagation on the GPU for Stereo Vision. In *Proceedings of the The 3rd Canadian Conference on Computer and Robot Vision (CRV '06)*. IEEE Computer Society, USA, 76. doi:10.1109/CRV.2006.19

[6] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1154–1165. doi:10.1145/3468264.3468626

[7] Gal Elidan, Ian McGraw, and Daphne Koller. 2006. Residual belief Propagation: informed scheduling for asynchronous message passing. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence* (Cambridge, MA, USA) *(UAI'06)*. AUAI Press, Arlington, Virginia, USA, 165–173. https://dl.acm.org/doi/10.5555/3020419.3020440

[8] Mahdi Eslamimehr and Jens Palsberg. 2014. Race directed scheduling of concurrent programs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 301–314. doi:10.1145/2555243.2555263

[9] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 2006. Efficient Belief Propagation for Early Vision. *Int. J. Comput. Vision* 70, 1 (Oct. 2006), 41–54. doi:10.1007/s11263-006-7899-4

[10] Haoyu Feng and Xin Zhang. 2026. *GPU-Accelerated Belief Propagation for Program Analysis (Artifact)*. doi:10.5281/zenodo.21188278

[11] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. 2009. Residual Splash for Optimally Parallelizing Belief Propagation. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 5)*, David van Dyk and Max Welling (Eds.). PMLR, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 177–184. https://proceedings.mlr.press/v5/gonzalez09a.html

[12] Scott Grauer-Gray, Chandra Kambhamettu, and Kannappan Palaniappan. 2008. GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In *2008 IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS 2008)*. 1–4. doi:10.1109/PRRS.2008.4783169

[13] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential Bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 561–575. doi:10.1145/3314221.3314616

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[15] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning probabilistic models for static analysis alarms. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1282–1293. doi:10.1145/3510003.3510098

[16] Tianchi Li and Xin Zhang. 2025. Combining Formal and Informal Information in Bayesian Program Analysis via Soft Evidences. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 143 (April 2025), 28 pages. doi:10.1145/3720508

[17] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and Homer H. Chen. 2009. Hardware-efficient belief propagation. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 80–87. doi:10.1109/CVPR.2009.5206819

[18] Haoran Lin, Zhenyu Yan, and Xin Zhang. 2026. Beer: Interactive Alarm Resolution in Bayesian Program Analysis via Exploration-Exploitation. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 106 (April 2026), 27 pages. doi:10.1145/3798214

[19] David JC MacKay and Radford M Neal. 1996. Near Shannon limit performance of low density parity check codes. *Electronics letters* 32, 18 (1996), 1645–1646. doi:10.1049/el:19961141

[20] Mark Van der Merwe, Vinu Joseph, and Ganesh Gopalakrishnan. 2019. Message Scheduling for Performant, Many-Core Belief Propagation. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2019.8916366

[21] Joris M. Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11, 74 (2010), 2169–2173. http://jmlr.org/papers/v11/mooij10a.html

[22] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy belief propagation for approximate inference: an empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence* (Stockholm, Sweden) *(UAI'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 467–475. https://dl.acm.org/doi/10.5555/2073796.2073849

[23] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. *SIGPLAN Not.* 41, 6 (June 2006), 308–319. doi:10.1145/1133255.1134018

[24] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. doi:10.1016/C2009-0-27609-4

[25] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 722–735. doi:10.1145/3192366.3192417

[26] Bharath Kumar Reddy L. and Nitin Chandrachoodan. 2012. A GPU implementation of belief propagation decoder for polar codes. In *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*. 1272–1276. doi:10.1109/ACSSC.2012.6489228

[27] David L. Romero and Nicholas B. Chang. 2012. Sequential decoding of non-binary LDPC codes on graphics processing units. In *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*. 1267–1271. doi:10.1109/ACSSC.2012.6489227

[28] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718003

[29] Yuanfeng Shi, Yifan Zhang, and Xin Zhang. 2025. On Abstraction Refinement for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 388 (Oct. 2025), 27 pages. doi:10.1145/3763166

[30] M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. 2006. Tree-based reparameterization framework for analysis of sum-product and related algorithms. *IEEE Trans. Inf. Theor.* 49, 5 (Sept. 2006), 1120–1146. doi:10.1109/TIT.2003.810642

[31] Yiqian Wu, Yifan Chen, Yingfei Xiong, and Xin Zhang. 2025. Belief Propagation with Local Structure and Its Applications in Program Analysis. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1566–1577. doi:10.1109/ASE63991.2025.00132

[32] Yiqian Wu, Yujie Liu, Yi Yin, Muhan Zeng, Zhentao Ye, Xin Zhang, Yingfei Xiong, and Lu Zhang. 2025. SmartFL: Semantics Based Probabilistic Fault Localization. *IEEE Transactions on Software Engineering* 51, 7 (2025), 2161–2180. doi:10.1109/TSE.2025.3574487

[33] Chen Yanover and Yair Weiss. 2002. Approximate Inference and Protein-Folding. In *Advances in Neural Information Processing Systems*, S. Becker, S. Thrun, and K. Obermayer (Eds.), Vol. 15. MIT Press. https://proceedings.neurips.cc/paper_files/paper/2002/file/100d9f30ca54b18d14821dc88fea0631-Paper.pdf

[34] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 958–969. doi:10.1145/3510003.3510073

[35] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 57 (Oct. 2017), 30 pages. doi:10.1145/3133881

[36] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 128 (April 2024), 29 pages. doi:10.1145/3649845

[37] Yifan Zhang and Xin Zhang. 2026. Fuzzing Guided by Bayesian Program Analysis. *Proc. ACM Program. Lang.* 10, POPL, Article 17 (Jan. 2026), 31 pages. doi:10.1145/3776659

[38] Guangyao Zhou, Antoine Dedieu, Nishanth Kumar, Wolfgang Lehrach, Shrinu Kushagra, Dileep George, and Miguel Lázaro-Gredilla. 2024. PGMax: Factor Graphs for Discrete Probabilistic Graphical Models and Loopy Belief Propagation in JAX. *Journal of Machine Learning Research* 25, 371 (2024), 1–25. http://jmlr.org/papers/v25/23-1010.html

Received 2026-03-27; accepted 2026-06-18