We thank the reviewers for their comments, and take the opportunity to answer their questions below.

Positioning relative to MCMC and VI (R2). This is a very interesting question, and in fact one of our long-term goals is to perform a rigorous comparison of different inference solutions at scale, using modern parallelization techniques. Our focus and current experimental data pertain to BP on CPU-centric platforms, and so we will refrain from definitive statements regarding other methods on specialized hardware.

However, what we can say with good confidence is that BP is still the go-to solution for specific applications such as LDPC decoding, and that significant work has been invested in speeding up such applications, see e.g. [8] and references within. Our work should definitely be relevant to such applications, and generally to CPU-centric inference; conversely, we believe dynamic scheduling ideas can be extended to other modern inference methods on CPUs.

Writing and notation (R3, R4). We acknowledge this point, and we will give more prominence to the experimental evaluation in the next revision. @R3: Thank you for the pointers on notation, which we will address.

Additional related work (R3). Thank you for this additional related work, which we will cite and discuss in detail. The first reference you noted focuses on distribution costs in the multi-node setting (e.g. partitioning the input graphs to avoid cross-node communication), and is therefore less relevant in terms of direct comparison. However, the second reference is relevant: we therefore implemented the algorithm in our framework, and present the results below. We take the $0.1|V|$ best vertices according to the Splash metric, and update the messages from these vertices, as suggested in this reference. The results in terms of speedup and number of updates relative to the baseline residual BP on one thread are presented below, where non-baseline algorithms are executed on 70 threads ("—" denotes failure to converge):

Speedup relative to residual BP on 1 thread:

Model	Relaxed Residual	Synchronous	Yin & Gao
Tree	1.391x	2.538x	1.692x
Ising	6.720x	3.009x	3.311x
Potts	7.454x	—	—
LDPC	13.393x	17.735x	3.044x

Number of updates relative to residual BP on 1 thread:

Model	Relaxed Residual	Synchronous	Yin & Gao
Tree	1.007x	48.000x	5.110x
Ising	1.058x	45.006x	3.996x
Potts	1.063x	—	—
LDPC	1.020x	4.404x	1.668x

As can be seen from an examination of the results, the algorithm of [2] has lower performance relative to relaxed residual on most inputs, due to the consistently higher number of updates.

Accuracy of inference (R3, R4). This is a good point, and we will discuss accuracy of inference more explicitly. In short, we use the LDPC model to test the accuracy of inference. In our LDPC instances, the correct marginals match the original codeword, and synchronous BP provably recovers these with high probability. All algorithms that converged recovered the correct codeword, indicating good accuracy for all schedules.

As suggested by R3, we also compared the marginal estimates of the algorithms as in ref. [2] of R3. We computed the average and maximum differences of point-wise marginal estimates between the baseline residual BP and all other algorithms. The average difference is less than 10^{-5} for all algorithms on LDPC and Ising and within $2 \cdot 10^{-3}$ on Potts. The maximum difference is less than $4 \cdot 10^{-4}$ for LDPC and Ising, i.e., all algorithms appear to converge to the same marginals. However, the maximum difference for Potts can be very high, but the residual algorithms seem to converge to the same answer; this is likely due to the fact that it seems Potts does not always converge properly.

Model selection and model variety (R4). We chose the models and parameters following prior work, to provide reasonable benchmarks and comparison to prior work. The experiments include two model size regimes, with the more detailed scaling studies in the appendix using a smaller model size. Across our experiments, we found that behavior on models generated with different random seeds is very similar. Indeed, as our test models are fairly large and drawn from random ensembles, it is expected that they follow the average behavior of the ensemble as a whole.

Implementation details on convergence detection and locking (R4). The convergence thresholds are detailed in Appendix D. The convergence condition is checked only periodically, as detailed in E.2.2, and has minimal (amortized) cost, included in the running times.

Tasks that get marked as in-process are withdrawn from the multiqueue, so not other thread can take them. Each thread takes locks on all messages affected by the task before processing it, in a fixed predetermined order, in order to avoid deadlocks. Thus, if two threads want to update the same message concurrently, the second one has to wait for the first to finish. We did not specifically measure the amount of time threads need to wait in this manner; however, we tested *unfair* versions of the algorithms, where threads can update messages without taking locks (leading to possibly inconsistent updates). The timing results were virtually identical, suggesting that overlaps are very unlikely in practice.