Parallel Belief Propagation for Image Denoising
15-418 Final Project Report
{vidhana, rbrigden}
May 8, 2018

1 Summary
We implemented parallel loopy belief propagation (LBP) for large graphical models using OpenMP on a multi-core CPU. Given the acceleration achieved by two of our three parallel implementations of LBP algorithms, we were able to run an unsupervised image denoising task with an 12 million variable loopy Ising model. The algorithms inherently are NP Hard and difficult to scale. We were able to achieve a 7x speedup (Carnegie Data) over the sequential baseline without using any locks.

2 Background
Graphical models provide common language for representing statistical models in machine learning. Belief propagation (graphical model inference) efficiently computes the marginal distribution over unobserved variables given the observed variables of a model. Belief Propagation in graphs is a NP-Hard inference problem. We try to see if we can do any better by using approximation based algorithms instead and scaling it for a multi-core system.

2.1 Loopy Belief Propagation
The loopy belief propagation algorithm applies to computing the marginals in undirected graphical models known as Markov Random Fields (MRFs). Unlike directed graphical models in which an edge represents a conditional dependency of one variable on another, an edge in an MRF has no directional relationship and specifies an ambiguous relationship. The celebrated Hammersley-Clifford theorem demonstrates that the joint probability of an MRF can be factored as a product of a potential function $\Psi$ applied to all maximal cliques in the graph – that is for an MRF with a set of maximal cliques $C$,

$$ P(X_1 = x_1, \ldots, X_n = x_n | \theta) = \frac{1}{Z(\theta)} \prod_{A \in C} \psi_A(x_A | \theta) $$
where \( Z \) is a normalizing function of the parameters. The belief propagation algorithm uses the observed values of some subset of the model’s variables to marginalize the values of the unobserved variables. Without being too pedantic about the specific derivation of the algorithm, consider a clarifying example of a loopy model (simply a graph with loops) with 4 variables and 4 edges. We will use this same representative example to discuss some details of our parallel implementation.

Say we wish to compute the marginal of \( a \), then we need to do the following. First set up the joint (size of all maximal cliques in this graph is 2).

\[
P(a, b, c, d) \propto \psi_{a,b}(a, b) \psi_{a,c}(a, c) \psi_{c,d}(c, d) \psi_{b,d}(b, d)
\]

Now marginalize over all the variables except \( a \).

\[
P(a) \propto \sum_b \sum_c \sum_d \psi_{a,b}(a, b) \psi_{a,c}(a, c) \psi_{c,d}(c, d) \psi_{b,d}(b, d)
\]

Notice that we can “push” these sums through some of the potentials if they do not depend on any variable in that potential.

\[
P(a) \propto \sum_b \psi_{a,b}(a, b) \sum_c \psi_{a,c}(a, c) \sum_d \psi_{c,d}(c, d) \psi_{b,d}(b, d)
\]

From here we see that two of these summation terms are going to leave the term in terms of a through marginalization. We create a useful abstraction for these kind of terms that pass information directly between adjacent nodes and call them \textit{messages}. Breaking the computation down into messages allows us to simply consider communication between adjacent nodes, making the problem amenable to parallelism. A message from a variable (node) \( i \) to \( j \) is denoted as \( m_{i \rightarrow j}(j) \)

\[
P(a) \propto m_{b \rightarrow a}(a) \ m_{c \rightarrow a}(a) \ \sum_d \psi_{c,d}(c, d) \psi_{b,d}(b, d)
\]

Notice above that we can’t formulate that last term as a message. It turns out that if this graph was a tree structure then we could run BP and get the exact
marginal values. Not all is lost – in loopy BP it turns out that we can simply pass a function of a subset of the incoming messages to a node to its neighbor nodes, updating their belief incrementally with these received messages. However, the algorithm is only guaranteed to converge to a fixed point if it converges at all (it can diverge if certain conditions are not met). Based on this, we define an iterative update for both the message values

\[ m_{i \rightarrow j}(t+1) = \sum_{x_i} \psi(x_i, x_j) \prod_{k \in N(i) \setminus j} m_{i \rightarrow k}(t) \]

where \( N(i) \) denotes the set of \( i \)'s neighbors. The belief at a node \( i \) (which is the same as the marginal estimate at convergence) is simply the product of its incoming messages!

\[ P(x_i)^{t+1} \propto \prod_{j \in N(i)} m_{j \rightarrow i}(t) \]

With this result, we note that we can calculate each node’s incoming and outgoing messages until some state of convergence is reached across all messages in the system. We investigate applying parallelism in this manner and show that the we can achieve a good speedup over a serial baseline. However, we also explore other variants of that simple algorithm that are less naturally amenable to parallelism, but exhibit better properties of convergence.

### 2.2 Ising Models for Image Denoising

We consider a similar but much larger and practical version of the square loop model in Figure 1. An Ising model is a loopy MRF composed of binary random variables in which the nodes are arranged in a grid and edges are placed between adjacent nodes. These are referred to as the latent variable nodes \( x \). Additionally, each of the latent variable nodes have a single edge connecting them to another independent node that we call an observable node \( y \). For an \( N \times N \) grid of latent variables in an Ising model, there are a corresponding set of \( N \times N \) observable nodes.

Because the maximal clique size of the Ising model is the same as the square graph, 2, we define potential functions on all edges. The domain assumption
of the Ising model is that adjacent latent variables encourage each other to maintain the same “polarity” and penalize them otherwise. Furthermore, every latent variable is highly encouraged to maintain polarity with its respective observable node. In order to obtain these useful properties, we specify the two potential functions of the Ising model

$$\psi(x_i, x_j) = \begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}$$

$$\psi(x_i, y_i) = \begin{bmatrix} 1.0 & 0.1 \\ 0.1 & 1.0 \end{bmatrix}$$

For these properties, the Ising model is very well suited for the task of image denoising. We take each observable node $y$ to correspond to a pixel in the noisy observed image and the pixels of the “true state” image with noise removed to correspond to the latent nodes $x$. If we marginalize each $x_i$ and take the most likely belief for each to be our estimate of each true pixel value, then we can reconstruct the image with these estimates as our overall “denoised” image.

As demonstrated previously, we can efficiently compute the marginals of a large set of random variables in an MRF using belief propagation. Given that the Ising model is a loopy graph on the order of $N^2$ nodes, where $N^2$ is the number of pixels in the image, it requires a significant number of message passing iterations before convergence. As such parallel acceleration is a very needed and, as we have discussed and will demonstrate, amenable to this application.

2.3 Loopy BP Algorithms We Implemented

The critical operation in all forms of loopy belief propagation is the task of reading and writing messages between adjacent nodes.

2.3.1 Batch (Synchronous)

This is an inherently parallel approach in which all the messages that every node needs to generate can be computed simultaneously. One of the key advantages of parallelizing this algorithm is that the message generation for every node is independent of other nodes and therefore there is no communication or mutual exclusion needed between different threads.

This is because the Bulk Synchronous approach requires that two copies of all messages is maintained at all times - one copy to read as input and another copy to store the resultant outputs. Both the copies are swapped at the end of every iteration.

2.3.2 Round-Robin (Asynchronous)

In asynchronous scheduling, we use the same copy of messages to both read the inputs and write the resultant messages (instead of maintaining two separate copies). Thus there is in-place updates to the messages which multiple processors could be reading from. In the round robin algorithm, we define a fixed
but random permutation (using Fisher Yates Algorithm) of all the vertices and these are then updated in that order repeatedly till convergence. Because of in-place updates, it is possible that readers get inconsistent message because another writer is currently updating that message. Therefore this algorithm requires locks to be placed on messages before reading or writing. When it is a node’s turn to generate the messages for its neighbors, it acquires locks all the slots it would use to fill up the new messages. For this algorithm, we used the atomic operation of \texttt{sync\_fetch\_and\_add} to increment a variable and return its old value (to be used to index into the permutation we discussed). Thus each processor atomically reads in the shared index variable and increments it. Once the processor reads in the value from the shared counter, it executes the algorithm on the node corresponding to that index in the permutation.

2.3.3 Wildfire (Asynchronous)

This algorithm is a further optimization over the round-robin algorithm discussed in the previous section. If one of the regions has converged while the other region still needs more iterations, the iterations over the first regions waste unnecessary cycles generating messages that are only slightly different than their previous ones. The wildfire algorithm addresses this problem by skipping the vertices that have converged.

In this algorithm, we introduce the notion of a residual which is an approximate measure of the amount of new information that has not yet been sent by a node. If we skip the vertices which have converged with respect to this notion of residual, then we have the wildfire algorithm.

3 Approach

We did not start with any existing code and wrote the algorithm based on the equations in [4] and the algorithm sketches for BSP, Round-Robin, and Wildfire in [2]. So the initial part of the project for us was to get a good handle on the domain knowledge of probabilistic graphical models.

All these algorithms work by each node in the graph estimating a probability density over possible labels for all its immediate neighbors and then sending this 'belief' to each neighbor. The calculation of the belief is different for every (source, destination) pair of vertices.

Also note that the Ising model [Fig2] converts the image into 2 sets of node - the unshaded ones and the shaded ones. The shaded ones are the noisy pixel observations of the image before it is denoised. The unshaded ones are the true pixel values which we want to infer from the algorithm based on the messages they receive from their neighbors.

The initial challenge was to figure out how to lay the data in a shared memory setting which for algorithms that involves so much of message passing between nodes of a graph. From the beginning, we were very conscious about writing the serial algorithm code such that it is very amenable for converting into a parallel
implementation. So we took inspiration from the GraphRat Assignment. After reading the graph (a 2D set of pixel values in our case), we create a ‘neighbors’ array and ‘neighbor_sizes’ array similar to the one we used in the assignment. These arrays help us determine who are the neighbors of any node in the entire graph. We also create a ‘write_array’ which is an array of messages and whose size matches the neighbors array. If node A has node B and node C as neighbors and node A happens to be the first node in the graph [Fig1], then the 0th, 1st and 2nd indexes of our ‘neighbors’ array would be A, B, C [Fig3]. Also the 1st index of the ‘write_array’ would be the slot in which A would fill in the message it wants to send to B and the 2nd index would be the slot in which A would fill in the message it wants to send to C.

This approach is very different than the traditional interpretation of message passing algorithms, which tend to rely on graph structures built using pointers between adjacent nodes. Such a method did not guarantee us good spatial locality and we decided to adapt the traditional algorithm to structures and operations that are more amenable to parallelism, such as setting up the message tables to respect message sending patterns (only need to read and write to neighbors) and combining the read step into the write step.

To update their beliefs (and to later calculate the messages for their own neighbors), B and C would need to read their incoming messages first. We create a look-up table such that each node knows exactly which indices in the ‘write_array’ to read (For example B should read the message that A generated for it from index 1 of the ‘write_array’).

Figure 3 and Figure 4 describe the difference between our synchronous implementation and the asynchronous ones. Note the need for locking makes the asynchronous algorithms much harder to parallelize effectively.

We wrote our serial and parallel BP implementations in C using the constructs provided by OpenMP for parallelization. We used a personal desktop computer with a 20-core Xeon CPU for all of our experiments. This was our choice for the experiments as we didn’t want to use latedays or the ghc machines due to heavy load on these systems and the unpredictability in the way different users utilize these machines.

We divided the ‘neighbors’ array such that each thread gets equal contiguous portions of the ‘neighbors’ and ‘write_array’ to work on. This was a reasonable strategy because all the nodes have almost equal amount of work (as each node in the Ising model have almost the same number of neighbors (min 1, max 5)).

We had a whole plethora of correctness and performance bugs that we came across in our experiments. They have been described in much detailed in section 3.4 (Analysis of Limitations).

3.1 Results

We successfully implemented a few parallel belief propagation algorithms and demonstrated notable performance gains over sequential baselines. We found that for practical applications like image denoising on large images (with $2574 \times 2574$ pixels and more than 12 million nodes), the 9x speedup that we measured
with the parallel Wildfire algorithm means the difference between someone waiting 13 minutes (a typical time for the sequential baseline) and a minute and a half.

### 3.2 Experimental Setup

We wrote a number of different scripts in Python that helped us set up our experiments. One would let us generate binary image data add noise to it as well as tell us the distance between a denoised image and the ground truth by counting the number of incorrect pixels. In order to run the denoising on standard PNG image files, we wrote a Python script to pipe a binary PNG image over STDOUT in a format. Our BP implementation in C++ would read in this line by line and initialize our data structures correctly. Once the inference was completed, the image estimate was then written back to disk by
piping the binary representations through STDOUT to a Python script that did the writing.

Ultimately, we focused our experiments on 3 different graphs. The first graph was an 8 node toy Ising model that we hand crafted for debugging and testing purposes. The second graph was a $256 \times 256$ binary image of rice grains ($256 \cdot 256 \cdot 2 = 131,072$ nodes) that proved to be a good benchmark task. To push the algorithms further and explore how different workloads affected the performance of the BP algorithms, we generated a noisy $2574 \times 2574$ pixel binary portrait of Andrew Carnegie using independent Bernoulli noise and benchmarked on that denoising task. From this point on, we will refer to the rice image denoising problem as ‘Rice’ and the Andrew Carnegie portrait denoising as ‘Carnegie’.

3.2.1 Performance Metrics

In our problem we specifically optimize each parallel BP algorithm for speedup over both a synchronous sequential baseline and a sequential baseline for the respective algorithm (eg. speedup of Wildfire parallel over Wildfire sequential). A very important consideration that needs to be taken into account is the rate of convergence of improves for both of the asynchronous BP algorithms. That is, although Round Robin and Wildfire both have more computational overhead and contention issues with maintaining locks over messages and beliefs, they reduce the amount of computation overall. For Wildfire, we were able to optimize out some of this overhead and ease some of the contention issues to the point where that improved convergence boost made the algorithm faster than the synchronous parallel implementation. For Round Robin, the overhead and contention issues encountered offset its benefit of decreased iterations for convergence.

3.3 Experiments

We compare the speedups of the the three parallel loopy BP algorithms with respect to a sequential baseline on both the Rice and Carnegie problems.
3.3.1 Sensitivity to Problem Size

We initially hypothesized that these parallel BP algorithms, specifically the asynchronous variety, would achieve better speedups on larger problems. We considered this because of the nature of the locking of messages and beliefs required in the asynchronous case. As the number of nodes increase in the problem and the number of simultaneous workers remains fixed, the likelihood that two workers may be writing to/reading from the same neighbor’s message significantly decreases. We confirmed this empirically as our parallel BP achieved greater speedups on the larger Carnegie problem. This improved scaling with problem size works out in the Ising model because edges are only between spatial neighbors (trivial spatial locality), though this is not guaranteed for arbitrary loopy graphical models.

3.4 Analysis of Limitations

Using our target 20-core CPU, we were only able to achieve a speedup of 7x on the larger Carnegie problem, which is less than half of an ideal linear speedup. The issues we face in gaining a speedup differed depending on the algorithm. The parallel synchronous algorithm did not have many dependencies, but we noted issues with the locality of message writes. The overhead of locking messages and beliefs on writes for the asynchronous parallel BP algorithms, as well as the contention over atomically fetching and incrementing the shared work assignment counter $i$, significantly limiting performance. These issues arose from a dependency that the asynchronous algorithms needed to have an execu-
Figure 7: Speedups of parallel loopy BP algorithms over their own respective sequential baselines on the Rice problem.

1. Most data structures are ordered by the vertices of the graph and their spatial relation. In the image denoising graphs, this that the latent variable vertexes were arranged in row major order and all of the observable variables were placed at the end of the array. In retrospect this memory ordering did not align with the access pattern of message passing belief propagation, in which new message updates (writes) receive messages (reads) from adjacent edges. For example, a node attempting to read a message from its adjacent observable variable and be required to read and write to a different cache line.

2. We realized that OpenMP would give good results only when used correctly. It is very easy to make mistakes such as declaring variables, which are private to each loop, outside the omp pragma causing unnecessary sharing of this variable between different iterations of for loop. We started seeing speedup only when we fixed all these bugs that were not so obvious to us in the very beginning.

3. The dynamic scheduling of threads in OpenMP works by picking up 1 iteration of the loop at a time by a thread whereas the static scheduling works.
by assigning a continuous range of iterations to each loop. In our experiments the dynamic schedule performed far worse than the static schedule. This maybe because the use of the internal work queue to give an iteration to the each thread was more overhead than the work accomplished in 1 iteration.

4. We realized that OpenMP abstraction was difficult to work in a 'while' loop setting. For every node, we want to test for convergence before doing a fresh round of the entire message generation and transmission of messages from this node to its neighbors. We had to re-think our 'round robins' and 'wildfire' (both asynchronous) algorithms in terms of 'for loops' to fit into the OpenMP paradigm. This was one of the major implementation challenges to get right.

5. Using locks in the asynchronous versions of our loopy belief algorithms was as The algorithms are very tricky to get right. There are various nuances such that different convergence criteria (global convergence, local convergence based on beliefs) for different algorithms that can make the algorithm really hard to implement. Furthermore there are hyperparameters like the threshold at which we declare correctness (or convergence). Also we figured out that if nodes update their beliefs based on all the incoming messages directly, then the denoising would be very patchy. Rather the nodes should take a weighted average of the newly computed belief and their old beliefs before updating the new belief (essentially a smoothing parameter). This parameter essentially controls the runtime and the number of iterations it takes for all the three algorithms to converge. If not tuned correctly, the algorithm starts to diverge instead
of converging.

4 References

1. Parallel Splash Belief Propagation by Gonzalez et. al.
2. Scaling up Machine Learning by Bekkerman et al (Chapter 10)

5 Code

Our project code can be found in the following repository. [https://github.com/rbrigden/418-fp](https://github.com/rbrigden/418-fp)

6 Division of Work

Equal work was performed by both project members.