Memory-efficient inference in dynamic graphical models using multiple cores

Galen Andrew
University of Washington
Department of CSE

Jeff Bilmes
University of Washington
Departments of EE & CSE

Abstract

We introduce the archipelagos algorithm for memory-efficient multi-core inference in dynamic graphical models. By making use of several processors running in parallel, the archipelagos algorithm uses exponentially less memory compared to basic forward-backward message passing algorithms ($\mathcal{O}(\log T)$ compared to $\mathcal{O}(T)$ on sequences of length T) and, under often-satisfied assumptions on the relative speed of passing forward and backward messages, runs no slower. We also describe a simple variant of the algorithm that achieves a factor of two speedup over forward-backward on a single core. Experiments with our implementation of archipelagos for the computation of posterior marginal probabilities in an HMM validate the space/time complexity analysis: using four cores, the required memory on our test problem was reduced from 8 GB to 319 KB (a factor of 25000) relative to forward-backward, but completed in essentially the same time. The archipelagos algorithm applies to any dynamic graphical model, including dynamic Bayesian networks, conditional random fields, and hidden conditional random fields.

1 Introduction

The basic forward-backward message-passing algorithms (FB) for inference in Dynamic Graphical Models (DGMs) (which include hidden Markov models (HMMs) [Ephraim, 2002, Rabiner, 1989, Eddy, 1998], dynamic Bayesian networks (DBNs) [Dean and Kanazawa, 1988], conditional random fields (CRFs) [Lafferty et al., 2001], and hidden conditional random fields (HCRFs) [Gunawardana et al., 2005]) are quite limited. These message passing algorithms typically run in $\mathcal{O}(T)$ time and require $\mathcal{O}(T)$ storage space, where T is the length of the sequence. In applications that have extremely long sequences, as are frequently encountered in bioinformatics, there may not be sufficient RAM to store all of the forward messages for every frame. In applications that have extremely large state spaces, as are frequently encountered in speech recognition and natural language processing, there also may not be sufficient RAM to store all of the forward messages for every frame since there is sufficient memory to store only a very small number of such messages. Forward messages could be paged to a larger store such as a hard disk or even a network, but that would entail enormous overhead, as sustained reading speeds from hard disk are orders of magnitude slower than DRAM [Hennessy et al., 2003].

The Island algorithm (IA) manages to reduce the memory requirements dramatically while increasing the time slightly [Binder et al., 1997, Murphy, 2002]. By repeating some computation, IA gets away with storing only a tiny fraction of the forward messages, so that it uses only $\mathcal{O}(T^{1/d})$ memory while increasing the running time by at most a constant factor d, for any $d \in \{2, 3, \ldots\}$. If d is allowed to vary so that $d = \lceil \log_b T \rceil$ for fixed parameter b, then it requires only $\mathcal{O}(\log T)$ space, at the cost of increasing the time complexity to $\mathcal{O}(T \log T)$. The stored messages are "islands" placed at regular intervals of T/b, giving the algorithm its name.

Another desire in such sequential models is parallelization. Considering any DGM as a ("fat") chain, a simple and naïve parallelization of forward-backward would send messages simultaneously starting from both the left and the right end of the chain, eventually meeting in the middle of the chain. At that point, message propagation proceeds from the middle back

Appearing in Proceedings of the 15^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume XX of JMLR: W&CP XX.

out to the ends. Using two processors, this gives a factor of two speedup. The Residual Splash algorithm of [Gonzalez et al., 2009] performs parallel inference that is exact for chain graphical models and achieves an approximation guarantee for arbitrary cyclic graphical models. Without redundant or approximate computation, however, there is no way to further utilize many threads ($\gg 2$) to exploit modern multi-core and future many-core microprocessors for simultaneous scheduling of DGM inference messages.

In this paper, we show how the redundant computation of the IA (which achieves the improved memory behavior) can be handled using parallel computation. Specifically, we show how the IA's repeated computation can be performed simultaneously on d threads running on multiple cores without additional memory overhead. The communication and scheduling overhead of the algorithm is quite small, and under often-satisfied assumptions on the relative speed of performing the forward and backward recursions, the algorithm runs as fast as standard forward-backward inference, while using only a logarithmic amount of memory.

We also describe a simple modification to the algorithm that brings an additional factor of two speedup (at the cost of using twice as many processors and memory), thereby achieving the same factor of two speedup mentioned above but with only logarithmic memory. We note that the new points of achievability for the time-space tradeoff in dynamic graphical models apply both to exact inference and also to approximate inference methods such as beam pruning, sequential Monte Carlo, or any DGM algorithm that passes approximate messages. Because multiple threads create and store groups of islands at different locations, we call our resulting new algorithm the "archipelagos algorithm" (AA).

In the next section, we describe the island and archipelagos algorithms in more detail. We then present results of experiments with our implementation of AA for the computation of posterior marginal probabilities in an HMM demonstrating the excellent empirical memory and time complexity of AA with respect to FB and IA.

2 Forward-Backward, Island, and Archipelagos

Consider a DGM and a forward-backward message passing inference algorithm. Such an algorithm could be used, say, to compute posterior marginals, or find the most likely hidden sequence, or to compute the k-best (most probable) sequences. The total size of the message passed at each time slice (i.e., the state space) is denoted n_m, and the total size of the set of observed variables at each time slice is n_o, over a sequence of length T. We are interested particularly in applications where either $n_m \gg n_o$ as commonly occurs in speech and language processing, or where T is very large (e.g., hundreds of millions) as commonly occurs in bioinformatics applications, or both. Under these conditions, the memory required to exactly store all forward messages is $\mathcal{O}(n_m T)$ which can easily be too large to fit even in todays' machines with large main memory.

For concreteness, we can consider standard forward-backward inference for smoothing in an HMM. Let $Y_t \in \{1 \dots n_Y\}$ be a random variable representing the hidden state at time t, and $X_t \in \{1 \dots n_X\}$ be the observation at time t, for $t \in \{0, \dots, T-1\}$ (note, the zero offset). The parameters of the HMM model are $p(i) \equiv \Pr(Y_0 = i)$ for all i, $q_i(j) \equiv \Pr(Y_{t+1} = j | Y_t = i)$ for all i and j, and $e_i(k) \equiv \Pr(X_t = k | Y_t = i)$, for all i and k. For simplicity, we assume the HMM is time-homogeneous, so that the transition and output probability tables do not depend on t (although this assumption is not required for either IA or AA to work). Given a sequence of observations $\hat{x}_{0:T-1}$, we will describe things in terms of the "smoothing" problem: that is, to compute the posterior marginal probabilities $\Pr(Y_t = i | X_{0:T-1} = \hat{x}_{0:T-1})$ for all t and i. We note that an almost identical analysis can be used to describe the problem of computing the k-best assignments to $Y_1, \dots, Y_T$ for any $k \geq 1$ but we do not include this in the paper.

2.1 Forward-backward

The forward-backward algorithm for computing all posterior marginals proceeds as follows. First, for $t = 0 \dots T-1$, we compute the *forward probabilities*

$$\alpha_t(i) = \Pr(Y_t = i, X_{0:t} = \hat{x}_{0:t})$$

according to $\alpha_0(i) = p(i) e_i(\hat{x}_0)$ and the recursion

$$\alpha_{t+1}(j) = e_j(\hat{x}_{t+1}) \sum_i \alpha_t(i) q_i(j). \tag{1}$$

Then, for $t = T-1 \dots 0$, we compute the *backward probabilities*

$$\beta_t(i) = \Pr(X_{t+1:T-1} = \hat{x}_{t+1:T-1} | Y_t = i)$$

using $\beta_{T-1}(i) = 1$ and

$$\beta_{t-1}(i) = \sum_j q_i(j) e_j(\hat{x}_t) \beta_t(j). \tag{2}$$

Finally, the desired posterior marginals are

$$\Pr(Y_t = i | X_{0:T-1} = \hat{x}_{0:T-1}) = \frac{\alpha_t(i)\beta_t(i)}{\sum_i \alpha_t(i)\beta_t(i)}. \tag{3}$$

The algorithm requires $\mathcal{O}(Tn_Y^2)$ operations for both the forward and backward passes. The β_t vectors can be updated in place, but all α_t vectors must be stored concurrently, so $\mathcal{O}(Tn_Y)$ memory is required.

2.2 Island algorithm

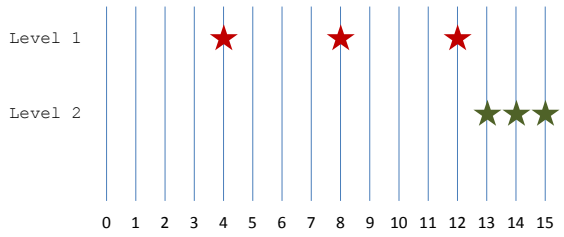
The idea of the island algorithm [Binder et al., 1997] is to discard most of the forward messages α_t, produced by the recursion (1), storing them only at $b-1$ regularly-spaced intervals. Then the algorithm is recursively applied to each of the b segments between the stored vectors until it bottoms out in standard forward recursion over much shorter sequences after $d = \log_b T$ applications. After producing the islands from left to right, the intervals are processed recursively from right to left, so that the first contiguous section to have all forward messages completed is at the end of the sequence. As each completed section is produced, the backward recursion can be applied and the posteriors computed. When the posteriors have been computed, the memory used to store the section can be overwritten to store the forward messages of the next section to the left. As with FB, the posteriors are produced from the end to the beginning of the sequence so backward probabilities need never be stored.

The time and space complexity of IA depends on the choice of d and b. Since each α_t (except for the islands) must be recomputed d times, IA performs d times as much work in the forward pass as FB. However, it requires storing only bd forward messages α_t at a time. If we choose a fixed b and set $d = \log_b T$, then IA requires $\mathcal{O}(T \log T)$ time and $\mathcal{O}(\log T)$ space. This is a log factor slower than FB, but uses exponentially less memory. Alternatively, we could fix d and set $b = T^{1/d}$, so that it uses $\mathcal{O}(T)$ time and $\mathcal{O}(T^{1/d})$ space: only a constant factor of d slower than FB, but polynomially less memory. For example, if $d = 2$, IA takes no more than twice as long as FB, but uses $2/\sqrt{T}$ as much memory.[1]

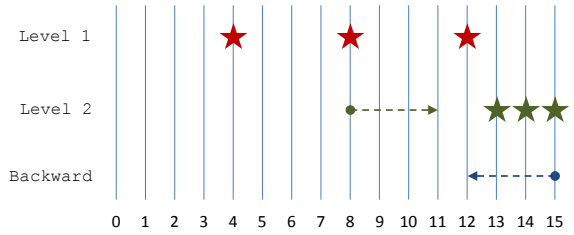
2.3 Archipelagos algorithm

Our proposed algorithm uses d compute threads (potentially running on separate cores/processors) to perform the work of the island algorithm with depth d. In order to explain the operation of AA, consider the moment during a run of the island algorithm when the first contiguous section of forward messages has been produced, at the end of the sequence, so that the backward probabilities can begin to be produced. The

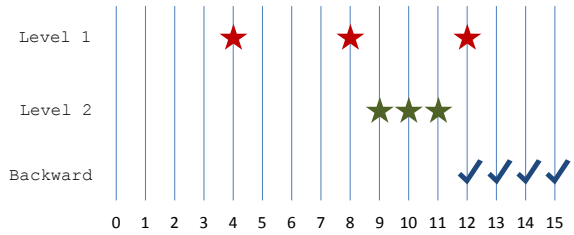
[1] We say "no more than twice as long" because the forward recursion (1) is slowed down by a factor of two while the backward recursion (2) and posterior computation (3) are unchanged.



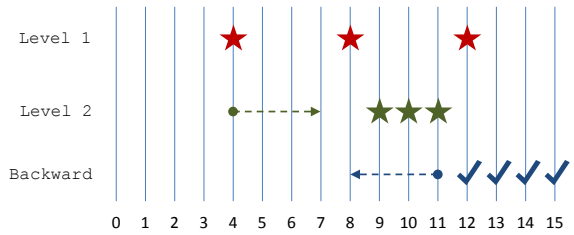
(a) Just before starting backward thread



(b) Two threads simultaneously running

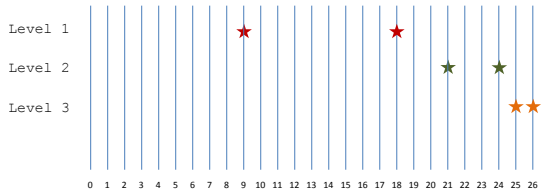


(c) Rendezvous point

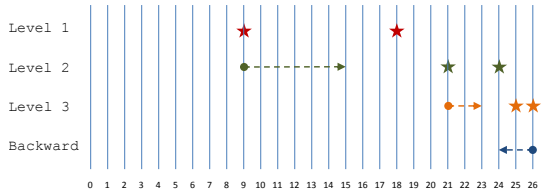


(d) Both threads running again

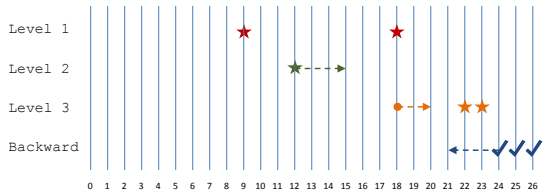
Figure 1: Illustration of archipelagos algorithm with $T = 16$, $b = 4$ and $d = 2$. In 1(a), the level 1 thread has finished, and the level 2 thread produced its first group of forward messages. In 1(b), the level 2 thread and the backward thread run concurrently. In 1(c), the level 2 thread and the backward thread have completed their sections (one having waited for the other before continuing). Finally, in 1(d), they both begin to process the next section.



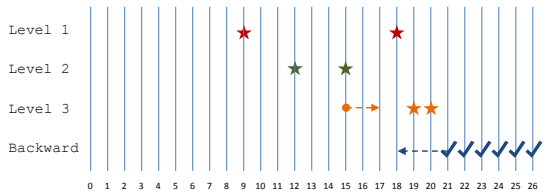
(a) Just before starting backward thread



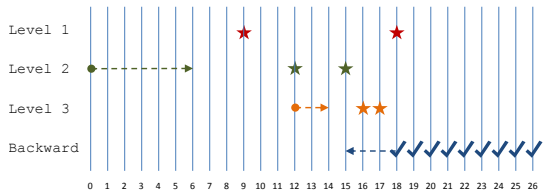
(b) Three threads running



(c) First group finished



(d) Level 2 thread finished



(e) Backward finishes level 2 section

Figure 2: Illustration of archipelagos algorithm with $T = 27$, $b = 3$ and $d = 3$. In the following, $\mathcal{L}_i$ refers to the forward thread at level i, and $\mathcal{B}$ to the backward thread. In 2(a), $\mathcal{L}_1$ has finished, $\mathcal{L}_2$ has finished its first group of islands, and $\mathcal{L}_3$ produced its first group of forward messages. In 2(b), all three threads run concurrently. In 2(c), $\mathcal{L}_3$ and $\mathcal{B}$ have completed their sections (one having waited for the other before continuing), while $\mathcal{L}_2$ continues (probably having produced its first island at about this time). In 2(d), $\mathcal{L}_2$ has finished and waits. $\mathcal{L}_3$ consumes the last island produced by $\mathcal{L}_2$, and $\mathcal{B}$ processes one more group. Finally, in 2(e), $\mathcal{B}$ reaches the beginning of its level 2 section, so $\mathcal{L}_2$ starts to work on the immediately preceding one.

process is illustrated in Figure 1, using $T = 16$, $b = 4$ and $d = 2$. At this moment we have top-level islands at $t \in \{4, 8, 12\}$, and the contiguous section of forward messages for $t = 12 \ldots 15$. Next, in Figure 1(b), AA begins to compute the backward probabilities and posteriors on a separate thread, and the thread that produced the forward messages begins to produce the next needed contiguous section, for $t = 8 \ldots 11$. The two threads continue until both the forward thread finishes producing frames 8–11, and the backward thread finishes using frames 12–15 (one or the other may block to wait for this rendezvous point), shown in Figure 1(c). Islands 12–15 are no longer needed, and the memory can be recycled. Then the forward thread can begin producing frames 4–7, while the backward thread works on 8–11 (Figure 1(d)).

If $d > 2$ the process is similar, except we then use d simultaneous compute threads. We use a dedicated thread for each level $l \in \{1 \ldots d\}$, plus one that does the backward recursion. Note that the level 1 thread proceeds continuously from 0 to $\frac{b-1}{b}T$ and then finishes. Its work does not overlap with that of the backward thread, so there are at most d concurrently working threads, and exactly d cores suffice. Each thread at level $l \in \{2 \ldots d-1\}$ consumes islands from the adjacent lower numbered level $(l-1)$ and produces islands spaced at intervals of T/b^{l-1}. The thread at level d produces contiguous sub-sections of forward messages to be consumed by the backward thread. Once each thread finishes producing forward messages in a given segment, it can wait until its consumer is ready to use those messages, then begin to produce messages in the immediately preceding segment. The process is illustrated in Figure 2 using $T = 27$, $b = 3$ and $d = 3$.

2.4 Space/Runtime analysis of AA

The memory requirements of AA are very similar to IA. As described, AA requires a factor of two more islands to be stored than IA, because each thread holds one section of islands that is being consumed while producing another section of the same size. With finer-grained locking the islands could potentially be recycled exactly as they are consumed, which would bring the memory requirements down to exactly that of IA. This would typically not be worth the synchronization overhead since the memory requirements of IA are already so small, but an additional factor of two is available as an option to AA when memory resources are very tight or the underlying models are very big.

Running on a real system with a finite number of available cores, we cannot simply hold b fixed and set $d = \log_b T$ for arbitrarily long sequences to achieve true asymptotic $\mathcal{O}(\log T)$ space. But if b is moderately large (like 10), extremely long sequences could

still be handled (e.g., up to $T = 10^{16}$ on a 16-core system). In any case, the analysis of IA in the fixed-d scenario still applies to achieve asymptotic $\mathcal{O}(T^{1/d})$ space complexity on a d-core system. In practice, one would choose d depending on whether memory or CPU is the limiting factor and use $b = T^{1/d}$ so as to use the minimal amount of memory given d.

Whether AA runs as fast as FB depends on the relative speed of forward and backward message passing. Assume that each iteration of forward message passing takes a constant amount τ_f of time, while a backward iteration takes τ_b time. We consider two scenarios depending on whether $\tau_f \leq \tau_b$.

In smoothing over unstructured DGMs like HMMs or CRFs, it can be reasonable to assume that $\tau_f \leq \tau_b$. For example, in HMM smoothing, evaluating (1) requires the same amount of work as (2), and the backward pass also has to compute the marginals using (3). In this case, we show that AA runs in approximately the same time as FB. Both algorithms begin by running T iterations of forward message passing in $T\tau_f$ time, after which the backward pass begins. We claim that if $\tau_f \leq \tau_b$, the backward thread in AA never has to block, and therefore the time required is just the same as FB.

For $l > 1$, let $\delta_l \equiv T/b^{l-1}$ be length of a level-l section. We use $\mathcal{L}_l$ to denote the (forward) thread at level l, and $\mathcal{B}$ to denote the backward thread. We first show that for all l, $\mathcal{L}_{l+1}$ never has to block waiting for $\mathcal{L}_l$ to produce islands. The statement is clearly true for $l = 1$ because $\mathcal{L}_1$ completes all of its islands for the entire run before $\mathcal{L}_2$ begins. Now suppose (for an inductive argument) that it is true for all $l \leq k$ for some k. When the $\mathcal{B}$ reaches the beginning of a level-l section, $\mathcal{L}_l$ has the islands it needs to begin processing, by hypothesis. It takes $(\delta_l - \delta_{l+1})\tau_f$ time to produce its last island. $\mathcal{L}_{l+1}$ will require that island when $\mathcal{B}$ reaches the first level-$(l+1)$ subsection. Before that, $\mathcal{L}_{l+1}$ has to operate on $(b-1)$ level-$(l+1)$ subsections, which also takes at least $(\delta_l - \delta_{l+1})\tau_f$ time (it could take more time, if $\mathcal{L}_{l+1}$ has to block waiting for the backward thread to catch up). Therefore $\mathcal{L}_l$ finishes first, and $\mathcal{L}_{l+1}$ will not need to wait for it. Note that this part of the argument did not rely on the assumption that $\tau_f \leq \tau_b$.

By the preceding argument, whenever $\mathcal{B}$ is ready to work on a level-d section, $\mathcal{L}_d$ can start producing forward messages for the preceding section immediately. It requires $(T/b^{d-1} - 1)\tau_f$ time to complete, while the backward thread needs $(T/b^{d-1})\tau_b$ time. Therefore the forward thread finishes first and the backward thread does not need to block.

Now consider if $\tau_f > \tau_b$. This may be the case if the forward pass has to perform expensive memory allocations that the backward pass may presume to exist, if the forward pass needs to compile factors of a factored DGM into tables, or in Viterbi decoding where the backward pass essentially only has to follow a backtrace. In this case, $\mathcal{B}$ finishes sections at each level before the next level-d section is ready for it. However, it is still the case that the forward threads never have to block to wait for their parents, and in particular $\mathcal{L}_d$ can work continuously because it needn't wait for either $\mathcal{L}_{d-1}$ or $\mathcal{B}$. It starts after $(T - \delta_d)\tau_f$ time and finishes $T\tau_f$ time after that. So the total time is no more than $2T\tau_f$, a factor of two more than FB.

2.5 Double-ended Archipelagos

A simple modification to AA provides an additional factor of two speedup at the cost of using twice as many cores. We have described message passing algorithms on DGMs as going forward first, storing the forward messages, and then aggregating them during the backward pass, but one could just as well first go backward and then forward. The double-ended variant of AA uses two cores to pass messages from both ends of the chain simultaneously towards the chain center. When the two processes meet in the middle of the DGM, they can exchange messages and then begin the multi-core stage of AA in opposite directions, using d cores each. Using this modification, AA is about twice as fast, and based on the analysis in the previous section, it is clear that double-ended AA is no slower than FB on a single core, even when $\tau_f > \tau_b$.

2.6 Structured State Spaces

We have described AA when Y_t was assumed to be an unstructured monolithic integer random variable. For many applications, however, it is important to be able to factor the state space into a structured model. Such a property is true in (DBNs) [Dean and Kanazawa, 1988, Ghahramani, 1998] and more generally dynamic graphical models (DGMs) [Bilmes, 2010] such as CRFs with a structured state space. In this case, each state variable Y_t consists of a set components $Y_t = (Y_t^1, Y_t^2, \ldots, Y_t^m)$ and factors associating two successive state variables may themselves factor so that $\phi(Y_t, Y_{t+1}) = \prod_f \phi_f(Y_{t+t_{f_a}}^{f_1}, Y_{t+t_{f_b}}^{f_2})$ for appropriate values of f_1, f_2 and $t_{f_a}, t_{f_b} \in \{0, 1\}$.

Both the IA and AA can easily be applied in this context by first turning the structured model into a temporal junction tree. To make the description simple, we describe this process using a particular example.

Figure 3 shows an example DGM, where instead of a single Y_t at each time, we have the four-element vector (A_t, B_t, C_t, D_t). Note that only the hidden variables

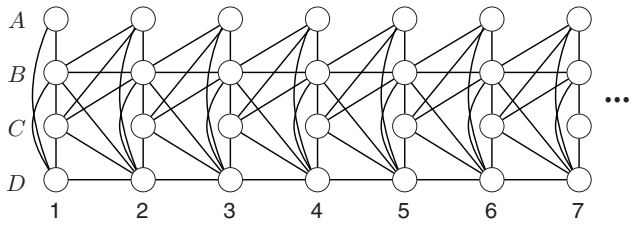


Figure 3: Example structured state space where at each frame there are four random variables (A_t, B_t, C_t, D_t) and the interaction relationships between random variables are as indicated in the graph.

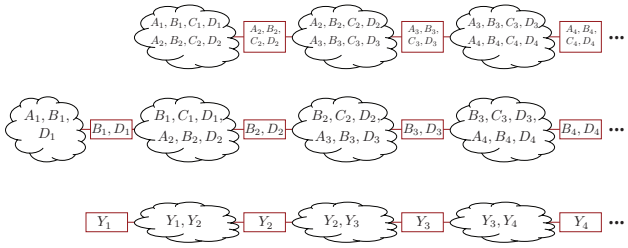


Figure 4: Top: A naive temporal junction tree corresponding to Figure 3. Middle: an improved temporal junction tree with reduced per-frame memory requirements. Bottom: The analogous unstructured temporal junction tree corresponding to an HMM.

are shown — e.g., if the model came from a DBN, then any coupling via observations with shared parents is already represented by undirected edges.

A naive temporal junction tree can be produced for this model by grouping together two successive frames of variables (i.e., $A_t, B_t, C_t, D_t, A_{t+1}, B_{t+1}, C_{t+1}, D_{t+1}$) into one large clique at time t, and where two successive cliques have their intersection, the four variables (A_t, B_t, C_t, D_t), as a separator. Assuming variable A_t has n_A values (and similarly for B, C, and D), the memory size for one separator would be $n_A n_B n_C n_D$, which in general is exponential in the number of variables in the separator. Thus, the overall memory cost for standard inference would be $O(T n_A n_B n_C n_D)$. This junction tree for this case is shown at the top of Figure 4.

The relationship, however, between the four random variables at frame t and those of the successor frame is not that of single factor, say $\phi(A_t, B_t, C_t, D_t, A_{t+1}, B_{t+1}, C_{t+1}, D_{t+1})$, directly relating all variables to each other. Rather, the advantage of structured models is that more nuanced relationships may be expressed between individual elements of the vector, as shown in Figure 3. An alternative strategy, therefore, that considerably reduces memory requirements has fewer variables both in each clique and each separator. This junction tree is shown in the middle of Figure 4. This temporal junction tree uses six (rather than eight) variables in each clique and only two (rather than four) variables in each separator. The per-frame memory requirements, therefore, are reduced to $n_B n_D$. See [Bilmes, 2010] for more details on how to find such junction trees automatically.

Once such a junction tree is defined, then either IA or the AA may be defined using junction tree messages, analogously to the way IA and AA were previously defined using the HMM messages described in Section 2.1. Indeed, in the HMM, the implicit temporal junction tree uses two successive variables Y_t, Y_{t+1} for each clique and one variable Y_t for each separator (see the bottom of Figure 4). The only difference between the top two figures of Figure 4 and the bottom figure is that the cliques and separators consist of vectors of variables rather than scalars, but the forward and backward message definitions between successive cliques have exactly the same form. Therefore, all of the preceding analysis for both IA and AA, including (and especially) the consideration of the various relationships between τ_f and τ_b, applies in the structured case as well.

3 Experiments

We implemented AA to compute the sum of the posterior marginal probabilities of an HMM. We compare the memory usage and runtime of IA and AA for $d = 1 \ldots 4$ on problems with large state space, long sequences, or both. In all experiments, the size of the observations is $n_X = 2$. The parameters are randomly generated (dense), as are the observations X_t. Experiments were run on an Intel Core i7-920 processor with 12GB RAM. The Core i7-920 has four cores and 8 MB L2 cache. For each problem, each setting of d and each algorithm, we perform ten runs and report the minimum wall-clock time to complete inference (not including generation of the HMM parameters and data). Results are summarized in tables 1, 2, and 3.

Not surprisingly, IA requires approximately a constant amount more time per level. AA on the other hand uses almost exactly the same time regardless of the number of levels in every scenario, as predicted for HMM smoothing, where the forward recursion is faster than backward. Note that $d = 1$ corresponds to standard FB in both columns, so we are observing that AA runs in exactly the same amount of time as basic FB. Interestingly, AA with $d \in \{2, 3\}$ was always slightly faster than $d = 1$, which we attribute to the fact that $d = 1$ must fill in all 8GB of forward messages, so it

d	IA mem	AA mem	IA time	AA time
1	8000	8000	296.9	297.5
2	12.2	16.8	403.5	297.2
3	0.52	0.86	493.6	298.4
4	0.182	0.319	624.6	301.1

Table 1: Results of experiments with $n_Y = 100$, $n_X = 2$ and $T = 10^7$. Memory is in MB, time in seconds.

d	IA mem	AA mem	IA time	AA time
1	800	800	249.4	249.5
2	5.06	7.59	360.2	248.4
3	1.18	2.52	466.0	248.2
4	0.55	1.01	572.1	252.3

Table 2: Results of experiments with $n_Y = 1000$, $n_X = 2$ and $T = 100000$. Memory is in MB, time in seconds.

d	IA mem	AA mem	IA time	AA time
1	80.0	80.0	284.5	284.5
2	5.12	7.68	463.5	282.7
3	2.40	4.00	679.1	284.6
4	1.92	3.36	922.2	287.4

Table 3: Results of experiments with $n_Y = 10000$, $n_X = 2$ and $T = 1000$. Memory is in MB, time in seconds.

experiences more cache misses. Note that all model parameters plus several islands' worth of forward messages require under 1MB, so they fit comfortably in this machine's 8MB L2 cache.

4 Conclusions

The usual reason for parallelizing an algorithm is to accelerate it, where the gold standard is to achieve an n-fold speedup via the use of n processors. In this paper we have shown how to use multiple cores not to accelerate an algorithm, but to make it use less memory, and the reduction in the amount of required memory is not linear, but exponential in the number of processors. As compute cores become cheaper to produce and the size of problems we wish to tackle grows ever larger, we believe that techniques like the archipelagos algorithm will become more and more necessary.

5 Acknowledgments

This research was supported by NSF grant IIS-0535100. The opinions expressed in this work are those of the authors and do not necessarily reflect the views of the funding agency.

References

Jeff Bilmes. Dynamic graphical models. *IEEE Signal Processing Magazine*, 27(6):29–42, November 2010. doi: 10.1109/MSP.2010.938078.

J. Binder, K. Murphy, and S. Russell. Space-efficient inference in dynamic probabilistic networks. *Int'l, Joint Conf. on Artificial Intelligence*, 1997.

T. Dean and K. Kanazawa. Probabilistic temporal reasoning. *AAAI*, pages 524–528, 1988.

S.R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755, 1998.

Y. Ephraim. Hidden markov processes. *IEEE Trans. Info. Theory*, 48(6):1518–1569, June 2002.

Z. Ghahramani. *Lecture Notes in Artificial Intelligence*, chapter Learning Dynamic Bayesian Networks, pages 168–197. Springer-Verlag, 1998.

Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *In Artificial Intelligence and Statistics (AISTATS)*, Clearwater Beach, Florida, April 2009.

A. Gunawardana, M. Mahajan, A. Acero, and J.C. Platt. Hidden conditional random fields for phone classification. In *Proc. Interspeech*, volume 2, page 1. Citeseer, 2005.

J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA, 2001.

K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, U.C. Berkeley, Dept. of EECS, CS Division, 2002.

L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.