2011 IEEE International Parallel & Distributed Processing Symposium

Algorithms for a parallel implementation of Hidden Markov Models with a small state space

Jesper Nielsen, Andreas Sand
Bioinformatics Research Centre
Aarhus University
Aarhus, Denmark
Email: {jn,asand}@birc.au.dk

***Abstract*—Two of the most important algorithms for Hidden Markov Models are the forward and the Viterbi algorithms. We show how formulating these using linear algebra naturally lends itself to parallelization. Although the obtained algorithms are slow for Hidden Markov Models with large state spaces, they require very little communication between processors, and are fast in practice on models with a small state space.**

We have tested our implementation against two other implementations on artificial data and observe a speed-up of roughly a factor of 5 for the forward algorithm and more than 6 for the Viterbi algorithm. We also tested our algorithm in the Coalescent Hidden Markov Model framework, where it gave a significant speed-up.

***Keywords*-Hidden Markov Model; parallelization; parallel reduction; parredhmmlib**

I. Introduction

Hidden Markov models (HMMs) are a class of statistical models for sequential data with an underlying hidden structure. They were first introduced to the field of bioinformatics in the early 1990s [1], and have since then been used in a wide variety of applications - for example gene annotation [2], [3], protein structure modeling [4], sequence alignment [5], [6] and phylogenetic analysis [7], [8]. Because of their computational efficiency, HMMs are one of the few widely used statistical methodologies that are feasible for genome wide analysis, where sequence lengths are in the millions or billions of characters. With data sets of this size, however, analysis time is still often measured in days or weeks. Improving on the performance of HMM analysis is therefore important to keep up with the quickly growing amount of biological sequence data to be analyzed.

In previous work [9] we have parallelized algorithms for HMM analysis to increase performance, by distributing the computations for each state among the available processors. This works well if the number of states is large, but for HMMs with a small number of states, the synchronization overhead makes this approach inefficient.

In this paper we present an alternative formulation of the forward algorithm and the Viterbi algorithm, parallelizing the workload across the observed sequence, instead of across the state space. This makes it feasible to give each processor a greater chunk of work and reduces communication overhead between the processors to a minimum. We thereby get a very efficient parallelization for HMMs with a small number of states. The algorithms have been implemented in a C++ library, *parredhmmlib*, that is freely available at http://www.birc.au.dk/~asand/parredhmmlib. Our implementation has been tested on artificial data, and in the Coalescent Hidden Markov Model (CoalHMM) framework [8].

II. Methods

An HMM is a probability distribution over a sequence $O = O_1 O_2 \dots O_T \in V^*$, where $V = \{V_1, V_2, \dots, V_M\}$ is an alphabet. We can formally define an HMM as consisting of [1]:

- A finite set of (hidden) states $S = \{S_1, S_2, \dots, S_N\}$. At any time t, the HMM will be in any of these states, $q_t = S_i$.
- A vector $\pi = (\pi_1, \pi_2, \dots, \pi_N)$ of initial state probabilities, in which $\pi_i = P(q_1 = S_i)$ is the probability of the model initially being in state i.
- A matrix $A = \{a_{ij}\}_{i,j=1,2,\dots,N}$ of transition probabilities, in which $a_{ij} = P(q_t = S_j | q_{t-1} = S_i)$ is the probability of the transition from state S_i to state S_j.
- A matrix $B = \{b_i(j)\}_{i=1,2,\dots,N}^{j \in V}$, where $b_i(j) = P(O_t = V_j | q_t = S_i)$ is the probability of state S_i emitting alphabet symbol V_j.

Now using this definition, a data sequence O of length T can be generated from an HMM by performing the following procedure:

1) Set $t := 1$;
2) Sample the initial state q_1 according to the probability distribution π;
3) Sample the alphabet symbol O_t from the emission probability distribution $b_{q_t}(\cdot)$;
4) set $t := t + 1$;
5) if $t \leq T$ then sample the next state q_t from the probability distribution $a_{q_{t-1}\cdot}$ and repeat from step 3; otherwise terminate.

An HMM is parameterized by π, A and B, which we will denote by $\lambda = (\pi, A, B)$.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.181

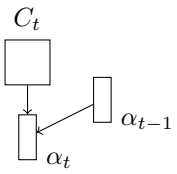


Figure 1. The information flow in the forward algorithm. α_t is computed only from α_{t-1} and C_t.

A. The parredForward algorithm

One of the traditional algorithms for Hidden Markov Models is the forward algorithm. The forward algorithm computes the likelihood of seeing our data, given our model, $P(O|\lambda)$. First define

$$\alpha_t(i) = P(O_1, O_2, ..., O_t, q_t = S_i|\lambda).$$

If we can compute these α_ts efficiently, we can compute $P(O|\lambda) = \sum_i \alpha_T(i)$. Let α_t be the vector of the $\alpha_t(i)$s

$$\alpha_t = \begin{bmatrix} \alpha_t(1) \\ \alpha_t(2) \\ ... \\ \alpha_t(N) \end{bmatrix},$$

B_t be a diagonal matrix of the emission probabilities at time t

$$B_t = \begin{bmatrix} b_1(O_t) & & & \\ & b_2(O_t) & & \\ & & ... & \\ & & & b_N(O_t) \end{bmatrix},$$

and

$$C_t = \begin{cases} B_1\pi & \text{if } t = 1 \\ B_t A^T & \text{otherwise.} \end{cases}$$

We can compute α_t using only C_t and the previous α_{t-1}

$$\alpha_t = C_t\alpha_{t-1} = C_tC_{t-1}\cdots C_2C_1,$$

as shown in Figure 1. Now, the classical implementation of the forward algorithm computes α_T as

$$(C_T(C_{T-1}\cdots(C_4(C_3(C_2C_1)))\cdots)).$$

Figure 2 shows how the α_t vectors are computed one at a time along the sequence, each one being derived from C_t and α_{t-1}. If more than one processor is available, one can attempt to parallelize the computation by computing the entries of α_t in parallel, which is the approach taken in HMMlib [9]. However the processors have to synchronize after each α_t is computed, and if N is small the computation of α_t contains very little work, thus the time spent on synchronization will dominate. We propose to compute the matrix product using parallel reduction. The idea of reduction is to take advantage of the fact that matrix multiplication is associative, thus the terms can be grouped arbitrarily. For example the terms could be grouped into a binary tree

$$(\cdots(C_TC_{T-1})\cdots((C_4C_3)(C_2C_1))\cdots).$$

Figure 3 illustrates how the final α_T can be computed by parallel reduction.

Note that not all the α_ts are computed, most being replaced by matrices, and that each matrix multiplication requires a synchronization to wait for its source data.

The traditional algorithm requires T matrix-vector multiplications, giving a workload of $\mathcal{O}\left(N^2T\right)$. In contrast the above algorithm makes use of matrix-matrix multiplications which are somewhat slower. If we assume, for simplicity, that we use the naive $\mathcal{O}\left(N^3\right)$-time matrix multiplication the workload becomes $\mathcal{O}\left(N^3T\right)$, thus it will have more actual work to do, but will be able to do a lot of it in parallel, and may actually be faster. If we assume we have one processor dedicated to each matrix multiplication, we get an execution time of $\mathcal{O}\left(N^3 \log T\right)$ on our new algorithm, which is better than the traditional $\mathcal{O}\left(N^2T\right)$ for small N and large T.

1) Numerical stability: All our matrices contain probabilities, which are between 0 and 1. This means that our products will tend toward zero exponentially fast. Normally the values will be stored in an IEEE 754 floating-point format. These formats have a limited precision, and if the above was implemented naively the results would quickly underflow and be rounded to zero.

If we can make do with $\log(P(O|\lambda))$ instead of $P(O|\lambda)$, we can prevent this underflow by continuously rescaling our matrices, like the way the columns are rescaled in the traditional forward algorithm [1]. We introduce a scaling constant c_i for every matrix multiplication, setting it to the sum of all entries in the resulting matrix. Each c_i is used two times: First we divide each entry in the resulting matrix by it, to keep the values from underflowing, and next we use it to restore the correct result at the end of our computations.

Assume we have l matrix multiplications and α_C is the resulting matrix, scaled to sum to one. Then

$$\alpha_T = \left(\prod_{k=1}^{l} c_k\right)\alpha_C,$$

and we can compute the final likelihood as

$$\begin{aligned} P(O|\lambda) &= \sum_i \alpha_T(i) \\ &= \sum_i \left(\prod_{k=1}^{l} c_k\right)\alpha_C(i) \\ &= \left(\prod_{k=1}^{l} c_k\right)\sum_i \alpha_C(i) \\ &= \prod_{k=1}^{l} c_k, \end{aligned}$$

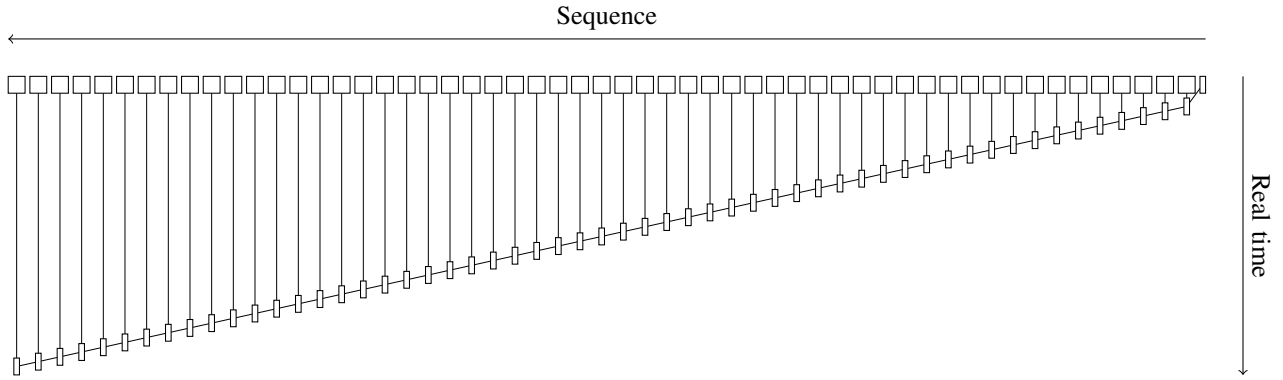


Figure 2. The traditional forward algorithm, as described by Rabiner [1]. The rectangles represent matrices and vectors. The black lines denote dependencies. The top row is the C_i matrices.

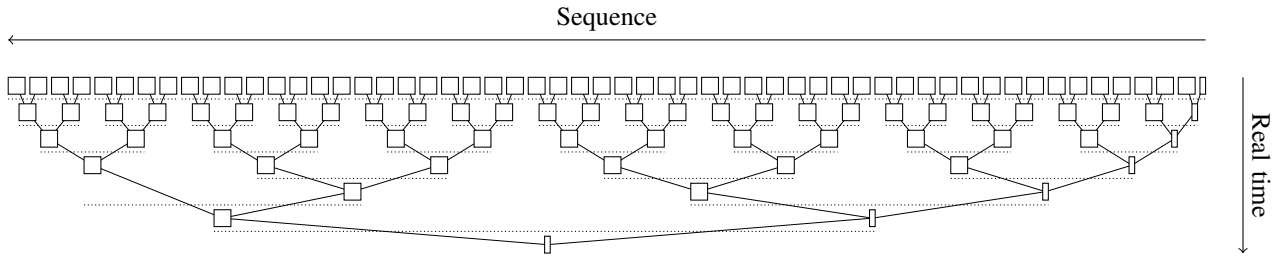


Figure 3. Using parallel reduction on the forward algorithm. The rectangles represent matrices and vectors. The black lines denote dependencies, while the horizontal dotted ones denote synchronization. The top row is the C_i matrices.

since α_C sums to one. Taking the logarithm of this, we get

$$\log(P(O|\lambda)) = \log\left(\prod_{k=1}^{l} c_k\right) = \sum_{k=1}^{l} \log(c_k).$$

2) Practical implementation: The above assumption that we have one processor dedicated to each multiplication is generally not true. In our implementation we assume the number of processors in a machine to be on the order of 10, and the sequence length T to be on the order of 10^6 or greater. We have made a number of changes to the above algorithm to make it simple and fast on a real computer.

To distribute the workload over the available processors we split it into a number of blocks. Each block will be processed completely independently and the result of that computation is a matrix representing that block. When all blocks have been processed one processor multiplies the result matrices to get the final result α_T.

Notice that C_1 is a vector, and not a matrix. The result of multiplying this with another C_i is another vector. This is important because matrix-vector multiplication is faster than matrix-matrix multiplication, which means that the first block, containing C_1, will be processed faster than the others, and that the result is a vector. Once we know the resulting vector from the first block, we can use that to also compute the second block quickly, and so on and so forth. We use this observation by having one processor processing the blocks from the beginning and continuously using the results between them, while the remaining processors consume the blocks from the other end, to retain as much work as possible for the fast algorithm. Notice that our algorithm reduces to the traditional forward algorithm on these first blocks. Figure 4 shows our implementation: In this case the sequence has been split into 11 blocks. The first processor $p = 0$ simply runs the traditional forward algorithm on the blocks, starting from the right, while the remaining processors $p = 1, 2, 3$, in parallel, consumes three blocks at a time from the left. After all the blocks have been processed the threads are joined, and α_T is found by multiplying the final vector from $p = 0$ with the resulting matrices from the other processors.

The algorithm the processor $p = 0$ executes has an asymptotic running time of $\mathcal{O}\left(N^2 T_1\right)$, while the algorithm executed by the remaining processors has a running time of $\mathcal{O}\left(N^3 T_2\right)$. If we assume that the difference in running time is exactly a factor of N, we expect $T_1 = \frac{TN}{N+P-1}$ and $T_2 = \frac{T}{N+P-1}$. This gives us an asymptotic running time for our algorithm of $\mathcal{O}\left(N^3 \frac{T}{N+P-1}\right)$, which is a factor $1 + \frac{P-1}{N}$ better than the traditional algorithm.

There can be only M different C_i matrices, besides C_1.

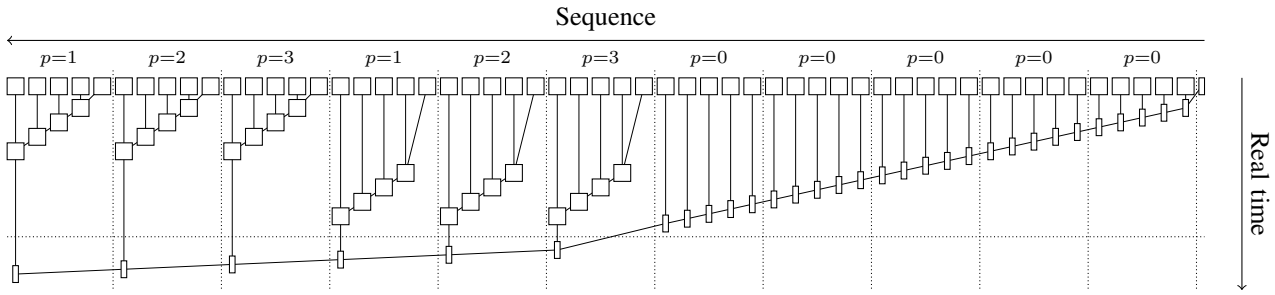


Figure 4. Our parredForward implementation, as run on a 4-way parallel system. The rectangles represent matrices and vectors. The black lines denote dependencies, the horizontal dotted ones denote synchronization, and the vertical dotted ones show blocks. For each block p identifies the processor computing it. The top row is the C_i matrices.

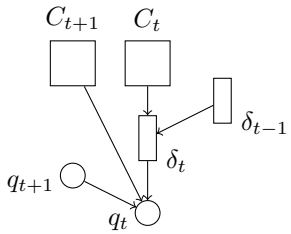


Figure 5. The information flow in the viterbi algorithm. δ_t is computed from δ_{t-1} and C_t similarly to the forward algorithm. When backtracking q_t depends on q_{t+1} and the data that gave rise to that: C_{t+1} and δ_t.

We assume M is small compared to T, and precompute all the different possible C_i matrices.

B. The parredViterbi algorithm

Another classical algorithm, as important as the forward algorithm is the Viterbi algorithm. The Viterbi algorithm finds the most likely state sequence $Q = q_1, q_2, ..., q_T$ given the observed data. This time define

$$\delta_t(i) = \max_{q_1,q_2,\cdots q_{t-1}} P\left(q_1, q_2, ..., q_t = i, O_1, O_2, ..., O_t|\lambda\right).$$

Remember that matrix multiplication is defined as:

$$(P \times Q)_{ij} = \sum_k P_{ik} Q_{kj}.$$

Similarly we define

$$(P \times^m Q)_{ij} = \max_k \{P_{ik} Q_{kj}\}.$$

Note that this new operator is associative. We can now compute

$$\delta_t = C_t \times^m \delta_{t-1} = C_t \times^m C_{t-1} \times^m ... \times^m C_2 \times^m C_1.$$

The entry in δ_T containing the maximal value will correspond to the final state q_T in Q, and the value of the entry will be the likelihood of Q. The rest of Q can be found by backtracking as sketched in Figure 5.

Traditionally the δs and Q would be computed linearly, but we can reduce it in parallel in exactly the same way as α_T was in the forward algorithm. Figure 6 illustrates how the δ_t vectors are traditionally computed exactly like the α_t vectors in the forward algorithm, and how Q also is found by a simple scan. Before we show the above formally, we will define some more notation:

$$D_{k:l} = C_l \times^m C_{l-1} \times^m ... \times^m C_{k+1} \times^m C_k,$$

for $1 \leq k \leq l \leq T$, and note that

$$\delta_t = D_{1:t} \qquad \text{and} \qquad D_{m+1:l} \times^m D_{k:m} = D_{k:l}.$$

Assume we have found some δ_t and its corresponding q_t. This is enough information to find all q_{t-1}, ..., q_1. δ_t must have been computed by some computation

$$\begin{aligned}\delta_t &= D_{1:t} \\ &= D_{k+1:t} \times^m D_{1:k} \\ &= D_{k+1:t} \times^m \delta_k,\end{aligned}$$

for some k and this allows us to find q_k as the entry in δ_k that would give rise to q_t,

$$q_k = \operatorname*{argmax}_j \left\{(D_{k+1:t})_{q_t j}\ \delta_k(j)\right\}.$$

The values q_{k-1}, ..., q_1 can be found by recursion. For the values q_{t-1}, ..., q_{k+1} note that $D_{k+1:t}$ must also be some product $D_{l+1:t} \times^m D_{k+1:l}$. Thus

$$\begin{aligned}\delta_t &= D_{k+1:t} \times^m \delta_k \\ &= D_{l+1:t} \times^m D_{k+1:l} \times^m \delta_k \\ &= D_{l+1:t} \times^m \delta_l.\end{aligned}$$

Using the above q_l can be found, and the entire range q_{t-1}, ..., q_{k+1} can be found by recursion. Since we started out showing how to find q_T we can find all of Q.

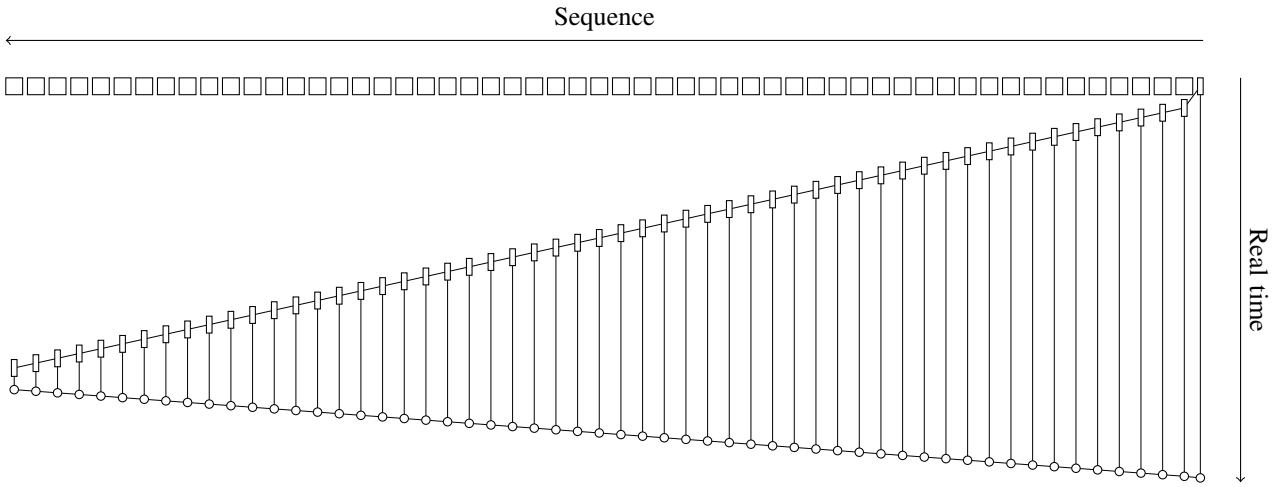


Figure 6. The traditional viterbi algorithm, as described by Rabiner [1]. The rectangles represent matrices and vectors, and the circles the q_t states. The black lines denote dependencies. The top row is the C_i matrices – to minimize clutter most dependencies on these are left out.

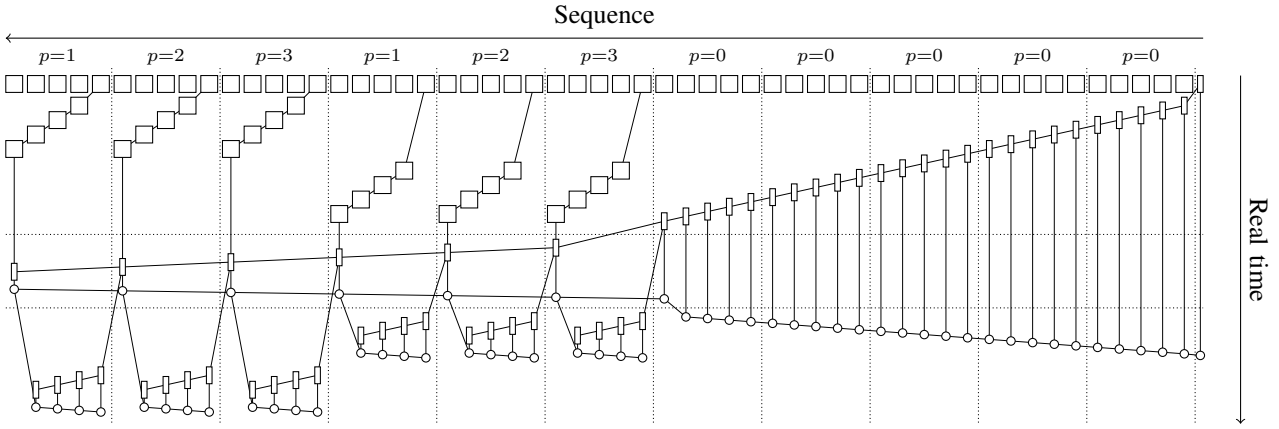


Figure 7. Our parredViterbi implementation, as run on a 4-way parallel system. The rectangles represent matrices and vectors, and the circles the q_t states. The black lines denote dependencies, the horizontal dotted ones denote synchronization, and the vertical dotted ones show blocks. For each block p identifies the processor computing it. The top row is the C_i matrices – to minimize clutter most dependencies on these are left out.

1) Practical implementation: Note that any $D_{1:i}$ will be a vector, while $D_{i:j}$ is a matrix, that takes up N times as much space, for $i > 1$. For a large T this could use a lot of memory. To conserve memory we store only one $D_{1:t_1}$, $D_{t_1+1:t_2}$, ..., $D_{t_n+1:T}$, from each block. From these we compute $D_{1:t_1}$, $D_{1:t_2}$, ..., $D_{1:T}$, and between these we fill out linearly, such that the majority of the matrices we store are of the form $D_{1:i}$ and we do not use significantly more memory than a traditional implementation.

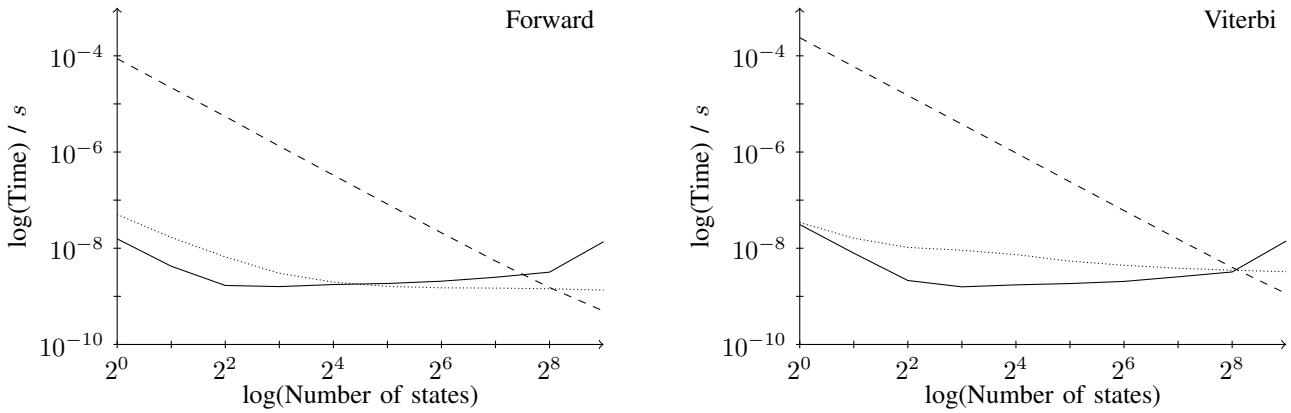
Figure 7 depicts our implementation of parredViterbi: The upper half of the figure is similar to the figure for parredForward (Figure 4), and indeed δ_T is found like α_T was. q_T can be found directly from δ_T, and the last state in each preceding block can be found by backtracking through the vectors just computed and the result matrices that gave rise to them. Once these states are found we start another parallel phase: $p = 0$ simply backtracks through the blocks it processed in the first phase, and $p = 1, 2, 3$ executes the traditional Viterbi algorithm on each of the remaining blocks. For $p = 1, 2, 3$ the initial vector for the Viterbi algorithm is based on the result vector δ_t from the preceding block, and the backtracking is started from the state found in the single threaded phase.

Also note that since we only do multiplication and maximum of scalars, and no addition, numerical stability is much easier to handle – simply do all computations directly in logarithmic space.

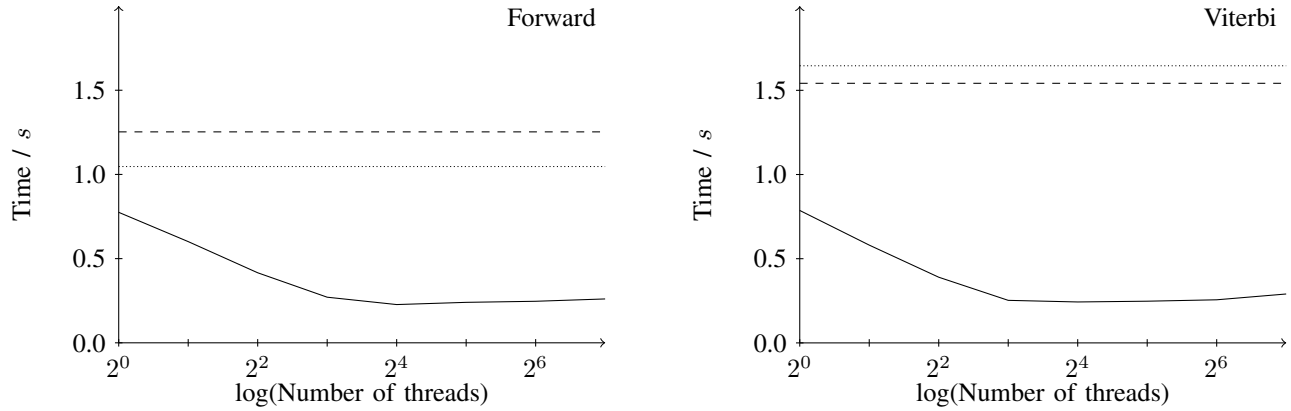
III. Results

We have implemented the above algorithms in the *parredhmmlib* package. The package is written in C++ and

A) Time per transition $\left(\frac{\text{time}}{N^2}\right)$.



B) Number of threads versus time.



C) Sequence length versus time.

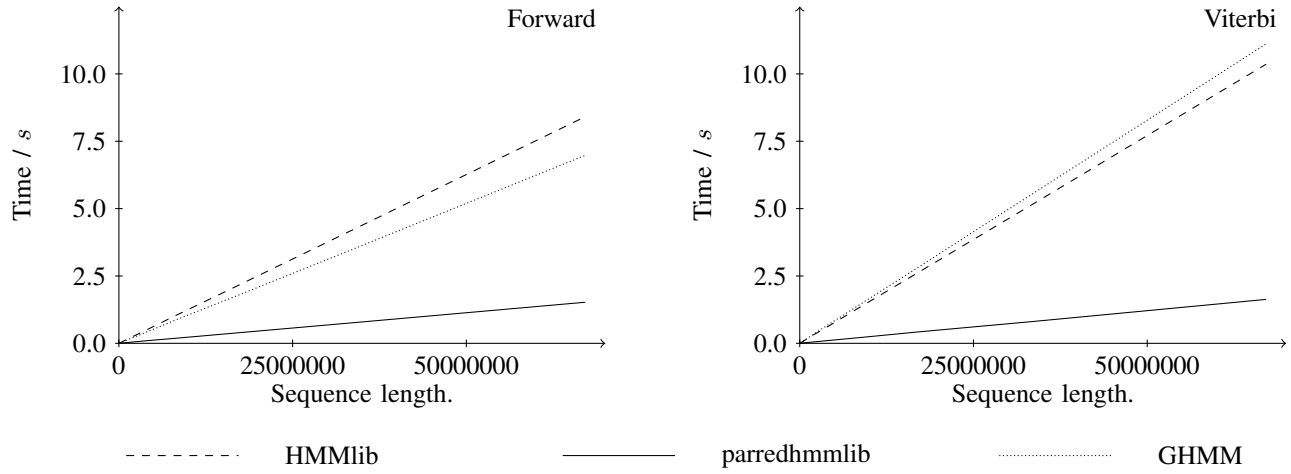


Figure 8. Results from our experiments. In A) we tested running time versus state space size, $M = 10$, $T = 366210$, parredhmmlib use 16 threads, HMMlib use 8 threads, with SIMD enabled and GHMM is inherintly single threaded. B) shows results from our experiment testing running time as a function of number of threads used, with $N = 4$, $M = 10$, $T = 10000000$ and HMMlib using 1 thread with SIMD disabled. Finally C) shows results from an experiment testing running time as a function of sequence length, using $N = 4$, $M = 10$, parredhmmlib with 16 threads, HMMlib and GHMM both running single threaded and HMMlib having SIMD disabled.

Python bindings are provided. The package is available from http://www.birc.au.dk/~asand/parredhmmlib. We have compared our implementation to GHMM [10], and our previous work HMMlib [9]. HMMlib is an implementation that takes advantage of all the features of a modern computer, such as SIMD instruction and multiple cores. The individual features of HMMlib can be turned on or off by the user, and we recommend only enabling these features for HMMs with large state spaces. Comparison with HMMlib is especially interesting because HMMlib also use parallelization, but does so at the level of each time-step instead of across them as presented in this paper. GHMM is a straightforward, but general implementation of the classical HMM algorithms [1], that does not use any kind of parallelization. Our experiments were run on a Mac-Pro with two Intel quad-core Xeon processors (256kB L2 cache, 8MB L3 cache) running at 2.26GHz and 8GB main memory. In total there are eight hyper-threaded cores allowing us to execute up to 16 processes in parallel.

We did three experiments on artificially generated HMMs. All the plotted points are averages over 10 runs. In the first experiment we varied the state space of the HMM, while keeping the sequence length T constant at 366210 and the alphabet size M at 10. We set HMMlib up for running on HMMs with large state spaces to demonstrate what happens when that kind of parallelization is applied to HMMs with small state spaces. The relative short sequence length T was chosen because HMMlib stores the entire forward table in memory, and as N grows this can take up a very large amount of memory. As explained earlier we expect parredhmmlib to be fast for small states spaces, and HMMlib to be fast for large state spaces. GHMM is not optimized for neither small nor large state spaces, and thus we do not expect it to be able to compete with parredhmmlib or HMMlib in those cases. Figure 8A shows our experimental results, in which the algorithms to a large extend behave as expected. The slowdown of parredhmmlib around 2^8 states may be because that is the point where the 8MB of L3 cache of the processor is exceeded.

We have also tested how the number of threads influence the running time of our algorithms, shown in Figure 8B. We ran the parredForward and parredViterbi on HMMs with 4 states and an alphabet of size 10 and sequences of length 10^7, varying the number of threads from one to 128. To compare our algorithms to HMMlib, we set HMMlib up to run as fast as possible on the models with small state spaces that we use. That is SIMD optimizations and parallelization was turned off. Curiously we see that our implementations are actually fastest even for one thread. This is probably because we precomputed the C_i matrices, while HMMlib and GHMM do not. We also note that both parredForward and parredViterbi run fastest with 16 threads, which again is what we expected. Using all eight cores of the machine the parredForward gains a speed-up of a factor of 2.9, and using hyper-threading and running on 16 cores a speed-up of a factor of 3.4. For parredViterbi the numbers are 3.1 and 3.2 respectively.

Finally we have tested how the sequence length L affects the execution time. As above we have set the number of states N to four and the alphabet size M to 10. The running time is expected to be linear in the sequence length, which also is what we see in Figure 8C. For the forward algorithm we are 5.5 times faster than HMMlib and 4.6 times faster than GHMM. For the Viterbi algorithm those numbers are 6.4 and 6.8 respectively.

We have also merged our method into the CoalHMM framework, where the existing implementation is based on HMMlib. CoalHMM is a framework that uses an HMM parameterized by coalescent theory, to infer changing genealogy along an alignment of DNA sequences [8]. The hidden states represent different genealogies, and the probability of change in genealogy, between two neighboring loci, is computed based on the probability of coalescence and recombination events. The observations are the columns of the alignment. We have benchmarked our method by running the model on an alignment of chromosome 22 of a Bornean and a Sumatran Orangutan [11], [12]. The HMM used in this experiment had $N = 10$ states and the length of the alignment was $T = 35 \cdot 10^6$. The analysis using the old implementation took 2.95 hours while the implementation using parredForward took 1.94 hours, giving a speed-up of a factor of 1.52. The relatively meager speedup can be explained by the CoalHMM framework having a quite significant overhead, especially the derivation of the transition matrices from the coalescent takes a long time.

IV. Conclusion

We have demonstrated how Hidden Markov Models with small state spaces can be parallelized. Although the obtained speed-up is not proportional to the number of processors, our approach actually does provide a significant improvement, as opposed to previous methods that were counter-productive for such HMMs. Speeding up processing of HMMs with a small state space is highly relevant because many HMMs are handcrafted and have small state spaces.

One important aspect of our method is that it requires very little communication between processors, making it a candidate for use on general purpose graphical processing units, or distribution over a network. Other future work would include applying our method to the Baum-Welch parameter estimation and the posterior decoding algorithms.

Acknowledgment

We thank Thomas Mailund and Anders Egerup Halager for help with, and data for the CoalHMM experiments.

REFERENCES

[1] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," pp. 267–296, 1990.

[2] A. V. Lukashin and M. Borodovsky, "GeneMark.hmm: new solutions for gene finding," *Nucleic Acids research*, vol. 26, no. 4, pp. 1107–1115, 1998.

[3] S. R. Eddy, "Profile hidden Markov models," *Bioinformatics*, vol. 14, no. 9, 1998.

[4] A. Krogh, B. E. Larsson, G. Von Heijne, and E. L. L. Sonnhammer, "Predicting transmembrane protein topology with a hidden markov model: application to complete genomes," *Journal of Molecular Biology*, vol. 305, no. 3, pp. 567–580, 2001.

[5] P. Baldi, Y. Chauvin, T. Hunkapiller, and M. A. McClure, "Hidden Markov models of biological primary sequence information," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 91, no. 3, pp. 1059–1063, 1994.

[6] S. R. Eddy, "Multiple alignment using hidden Markov models," in *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, vol. 3, 1995, pp. 114–120.

[7] A. Siepel and D. Haussler, "Computational identification of evolutionarily conserved exons," in *Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*. ACM, 2004, p. 186.

[8] J. Y. Dutheil, G. Ganapathy, A. Hobolth, T. Mailund, M. K. Uyenoyama, and M. H. Schierup, "Ancestral Population Genomics: The Coalescent Hidden Markov Model Approach," *Genetics*, vol. 183, pp. 259–274, 2009.

[9] A. Sand, A. T. Brask, C. N. S. Pedersen, and T. Mailund, "HMMlib: A C++ Library for General Hidden Markov Models Exploiting Modern CPUs," in *Proceedings of the Second International Conference on High Performance Computational Systems Biology*, 2010.

[10] A. Schliep, B. Georgi, W. Rungsarityotin, I. Costa, and A. Schonhuth, "The general hidden markov model library: Analyzing systems with unobservable states," *Forschung und wissenschaftliches Rechnen: Beiträge zum Heinz-Billing-Preis 2004*, pp. 121–135, 2005.

[11] D. P. Locke *et al.*, "Comparative and demographic analysis of orang-utan genomes," *Nature*, vol. 469, no. 7331, pp. 529–533, Jan. 2011.

[12] T. Mailund, J. Y. Dutheil, A. Hobolth, G. Lunter, and M. H. Schierup, "Estimating Divergence Time and Ancestral Effective Population Size of Bornean and Sumatran Orangutan Subspecies using a Coalescent Hidden Markov Model," 2011.