HMMlib: A C++ Library for General Hidden Markov Models Exploiting Modern CPUs

Andreas Sand, Christian N.S. Pedersen
and Thomas Mailund
Bioinformatics Research Centre
Aarhus University, Denmark
{*asand, cstorm, mailund*}*@birc.au.dk*

Asbjørn Tølbøl Brask
Actua
Aarhus, Denmark
atbrask@actua.dk

***Abstract*—We present a C++ library for constructing and analyzing general hidden Markov models. The library consists of a number of template classes and generic functions, parameterized with the precision of floating point types and different types of hardware acceleration.**

***Keywords*-HMMlib, Hidden Markov Models, SSE, OpenMP**

I. Introduction

Hidden Markov models (HMMs) is a class of statistical models for sequential data that was introduced to the field of bioinformatics in the early 1990s, and is now one of the most popular methodologies in the field. The popularity stems from a combination of versatility and computational efficiency. HMMs are used for a very wide variety of problems - from gene annotation [LB98], [E98] over alignment [B94], [E95] to phylogenetic analysis [SH04] - and the common algorithms for HMMs scale linearly in sequence length and quadratic in the number of states of the model.

Because of their computational efficiency, HMMs are one of the few widely used statistical methodologies that are feasible to use on genome wide analysis, where sequence length is in the millions or billions of characters. With data sets of this size, however, analysis time is still usually measured in days or weeks, and the models need to be kept relatively simple to keep the number of states sufficiently small.

Improving on the performance of HMM analyses will not only reduce the time necessary for data analysis but also, and, more importantly, make more complex and more sophisticated modelling computationally feasible.

In this paper we present HMMlib: an easy to use C++ library for general hidden Markov models, exploiting modern CPUs with multiple cores and support for the SSE instruction set. We show how we use these technologies to obtain significant speed-ups for the four classical HMM algorithms: The Viterbi, forward, backward and Baum Welch algorithms.

HMMlib is available at http://birc.au.dk/~asand/hmmlib/.

II. Hidden Markov Models

We can formally define an HMM as consisting of [RJ86]:

- An alphabet $\Sigma = \{a_1, a_2, \ldots, a_M\}$,
- A finite set of (hidden) states $Q = \{1, 2, \ldots, K\}$.
- A vector $\pi = (\pi_1, \pi_2, \ldots, \pi_K)$ of initial state probabilities, in which π_i is the probability of the model initially being in state i. As π is a probability distribution on the set of states, it must sum to 1.
- A matrix $T = \{t_{ij}\}_{i,j=1,2,\ldots,K}$ of transition probabilities, in which t_{ij} is the probability of the transition from state i to state j. Each row $t_{i\cdot}$ in T must sum to 1, since each of them composes a probability distribution on the states.
- A matrix $E = \{e_{ij}\}_{i=1,2,\ldots,M}^{j=1,2,\ldots,K}$, where e_{ij} is the probability of state j emitting alphabet symbol a_i. Each column $e_{\cdot j}$ in E must sum to 1.

Now using this definition, a data sequence $s = s_1 s_2 \ldots s_n$ in Σ^* can be generated from an HMM by performing the following procedure:

1) Set $t := 1$;
2) Sample the initial state z_1 according to the probability distribution π;
3) Sample the alphabet symbol s_t from the emission probability distribution $e_{\cdot z_t}$;
4) set $t := t + 1$;
5) if $t \leq n$ then sample the next state z_t from the probability distribution $t_{z_{t-1}\cdot}$ and repeat from step 3; otherwise terminate.

An HMM is parameterized by π, T and E. We will denote these parameters by $\Theta = (\pi, T, E)$, but for brevity we will omit Θ, when it is given by the context.

A. HMM algorithms

Given this definition of an HMM, we now describe the four classical HMM algorithms: the Viterbi algorithm, the forward and backward algorithms and the Baum Welch algorithm. We describe how to run these algorithm using HMMlib in section II-B and III-B.

1) The Viterbi Algorithm: Given an observed sequence $s = s_1, s_2, \ldots, s_n$, the Viterbi algorithm computes the sequence $z^* = z_1, z_2, \ldots, z_n$ of hidden states maximizing $P(s, z^*|\Theta)$.

The probability of a sequence of hidden states $z = z_1 z_2 \ldots z_n$ emitting s may be computed as

$$P(s,z) = \pi(z_1)e_{s_1,z_1} \cdot t_{z_1,z_2}e_{s_2,z_2} \cdot t_{z_2,z_3}e_{s_3,z_3} \cdot t_{z_3,z_4}e_{s_4,z_4} \cdots t_{z_{n-1},z_n}e_{s_n,z_n}.$$

Thus we want to find $z = z^*$, such that this expression is maximized. To avoid enumerating all paths through the model, we make use of a dynamic programming table V of size $n \times K$, in which the jth entry in the ith row is the probability of the most likely path of the sequence $s_1, s_2, \ldots, s_i$ ending in state j:

$$v_{ij} = \max_{z_1,z_2,\ldots,z_{i-1}} P(s_1, s_2, \ldots, s_i, z_1, z_2, \ldots, z_{i-1}, z_i = j).$$

Rewriting this using the Markov property, we get:

$$v_{ij} = e_{s_i,j} \max_k t_{k,j} v_{i-1,k},$$

yielding a straight forward procedure to fill in all entries in V running in time $O(nK^2)$. Computing in $\log$-space to avoid numerical instabilities, caused by repeatedly multiplying small numbers, this becomes

$$\tilde{v}_{ij} = \log e_{s_i,j} + \max_k(\log t_{k,j} + \tilde{v}_{i-1,k}). \tag{1}$$

Once $\tilde{V}$ is filled in, the maximum entry of the last row in $\tilde{V}$ holds $\log(P(s, z^*|\Theta))$, and it is easy to backtrack the computation from this entry to derive z^*.

2) The Forward Algorithm: We now describe the forward algorithm for solving the following problem: Given an observed sequence $s = s_1, s_2, \ldots, s_n$, compute the probability $P(s|\Theta)$ of the model with parameters Θ generating s. For this purpose, we will define another dynamic programming table $F = \{f_{ij}\}_{i=1,2,\ldots,n}^{j=1,2,\ldots,K}$, in which the jth entry of the ith row is the probability of the model generating $s_1, s_2, \ldots, s_i$ and ending up in state j. Thus we have

$$\begin{aligned} f_{ij} &= P(s_1, s_2, \ldots, s_i, z_i = j) \\ &= e_{s_i,j} \sum_l f_{i-1,l} t_{l,j}, \end{aligned}$$

which gives us a straight forward procedure to fill in the forward dynamic programming table. The probability $P(s|\Theta)$ of s being generated by the model, can now be computed as the sum of the last row in F.

To avoid numerical instabilities, we define a normalized version of F:

$$\hat{f}_{ij} = \frac{f_{ij}}{P(z_1, s_2, \ldots, s_i)} = P(z_i = j|s_1, s_2, \ldots, s_i).$$

To compute these probabilities we introduce scaling factors $c_k = P(s_k|s_1, s_2, \ldots, s_{k-1})^{-1}$, such that

$$p(s_1, s_2, \ldots, s_k)^{-1} = \prod_{m=1}^{k} c_m.$$

This finally gives us the recursion to fill $\hat{F}$:

$$\hat{f_{ij}} = c_i e_{s_i,j} \sum_l \hat{f}_{i-1,l} t_{l,j}. \tag{2}$$

To use this recursion, we need to compute and store c_i in the ith iteration. This is easilly done, since it is the coefficient normalizing $\sum_{j=1}^{K}(e_{s_i,j} \sum_l \hat{f}_{i-1,l} t_{l,j})$. To get $P(s|\Theta)$ we may now compute the reciprocal of the product of all scaling factors. However to avoid nummerical instabilities we will compute the log-likelihood instead:

$$\tilde{P}(s|\Theta) = -\sum_{i=1}^{n} \log c_i. \tag{3}$$

3) Finding the Posterior Decoding using the Forward and Backword Algorithms: The posterior decoding of an observed sequence of data is an alternative to the output of the Viterbi Algorithm. The Viterbi Algorithm computes the most likely sequence of underlying hidden states, but we are often more interested in knowing the probability with which the ith observation was emitted by the jth state. This is the *a posteriori* probability, and to compute it we note that

$$\begin{aligned} P(z_i = j|s) &= \frac{P(s, z_i = j)}{P(s)} \\ &= \frac{P(s_1, \ldots, s_i, z_i = j)P(s_{i+1}, \ldots, s_n|z_i = j)}{P(s)}. \end{aligned}$$

We recognize $P(s_1, s_2, \ldots, s_i, z_i = j)$ as f_{ij} and $P(s)$ as the sum of the last row in F. Thus all we need to compute is $P(s_{i+1}, s_{i+2}, \ldots, s_n|z_i = j)$. To do this we define the backward dynamic programming table $B = \{b_{ij}\}_{i=1,2,\ldots,n}^{j=1,2,\ldots,K}$, in which the jth entry of the ith row is the probability of the model generating $s_{i+1}, s_{i+2}, \ldots, s_n$ when starting in state j: $P(s_{i+1}, s_{i+2}, \ldots, s_n|z_i = j)$. Rewriting this, using the Markov property, we get the following recursion for computing B in a bottom-up traversal:

$$b_{ij} = \sum_l e_{s_{i+1},l} t_{jl} b_{i+1,l}.$$

And using the scaling factors from the forward algorithm to avoid numerical instability, we arrive at the scaled backward algorithm:

$$\hat{b}_{ij} = c_{i+1} \sum_l e_{s_{i+1},l} t_{jl} \hat{b}_{i+1,l}. \tag{4}$$

Finally to compute the posterior decoding, we first run the forward algorithm to fill $\hat{F}$, saving the scaling factors, then run the backward algorithm to fill $\hat{B}$, and finally we compute

$$P(z_i = j|s) = \frac{\hat{f}_{ij}\hat{b}_{ij}}{c_i}. \tag{5}$$

4) The Baum Welch Algorithm: We now turn to an algorithm for the third canonical problem of HMMs: Given an observed sequence of data s, find $\Theta = (\pi, T, E)$, such that $P(s|\Theta)$ is maximized. Given the expression for the a posteriori probabilities $P(z_i = j|s)$ in (5), we may estimate the initial probabilities $\pi = (\pi_1, \pi_2, \ldots, \pi_K)$ by

$$\pi_j = \frac{\hat{f}_{1,j}\hat{b}_{1,j}}{c_1}. \tag{6}$$

To estimate the probabilities in T and E, we first define $\xi_{ij}^t = P(z_t = i, z_{t+1} = j|s)$ to be the probability of being in state i at time t and state j at time $t+1$ given the observed sequence s. I.e.

$$\xi_{ij}^t = \frac{P(z_t = i, z_{t+1} = j, s)}{P(s)} = \hat{f}_{ti} t_{ij} e_{s_{t+1},j} \hat{b}_{t+1,j}.$$

Given $\hat{F}$ and $\hat{B}$, ξ_{ij}^t can be computed in $O(1)$ time for any i, j and t. Using this and (5) we estimate T and E as follows:

$$t_{ij} = \frac{\sum_{t=1}^{n} \xi_{ij}^t}{\sum_{t=1}^{n} P(z_t = i|s)} \tag{7}$$

$$e_{ij} = \frac{\sum_{t \in \{k=1,2,\ldots,n | s_k = a_i\}} P(z_t = j|s)}{\sum_{t=1}^{n} P(z_t = j|s)} \tag{8}$$

B. The basics of HMMlib

HMMlib is an easy to use C++ library for constructing and analyzing general hidden Markov Models. To suit different needs, it supports both single and double precision floating-point numbers. As we shall see in Section V this is a trade-off between numerical precision and computation time. The implementation of HMMlib uses template meta programming to support these two types of floating-point numbers, and thus the cost of having this choice is payed for at compile-time only.

The basis of HMMlib is composed by the three classes `HMM`, `HMMMatrix` and `HMMVector`. While the `HMM` class encapsulates the representation of an HMM, `HMMMatrix` objects are used to represent the matrices E, T, F etc. To represent vectors, we use `HMMVector` objects, that are just `HMMMatrix`es with a single row.

In order to run the HMM algorithms described in section II-A, we must first construct an `HMM` object. This is parameterized by π, T and E and thus the constructor of the `HMM` class takes an `HMMVector` object and two `HMMMatrix` objects as parameters, representing π, T and E, respectively. Thus these must be constructed and initialized before constructing the `HMM` object. To construct an `HMMMatrix` of size $n \times m$ containing `doubles`, we write:

```
HMMMatrix<double> matrix(n, m);
```

All entries of `matrix` are now set to 0.0. To change the i,jth entry in `matrix`, we use the indexing operator; e.g:

```
matrix(i, j) = 0.5;
```

Using the `HMMVector` class is very similar. To construct an `HMM` object, we first construct the π `HMMVector`, the T `HMMMatrix` and the E `HMMMatrix`. To make sure, the memory is managed in an efficient way, we use the smart pointer implementation, provided by the Boost libraries, `shared_ptr`, to store these:

```
shared_ptr<HMMVector<double> >
  pi_ptr(new HMMVector<double>(K));
shared_ptr<HMMMatrix<double> >
  T_ptr(new HMMMatrix<double>(K,K));
shared_ptr<HMMMatrix<double> >
  E_ptr(new HMMMatrix<double>(M,K));
...
// initialize pi, T and E
...
HMM<double> hmm(pi_ptr,
                T_ptr,
                E_ptr);
```

Now given an `HMM` object `hmm` and an observed sequence of data $s = s_1 s_2 \ldots s_n$, to run e.g. the forward algorithm, we first construct an `HMMVector` object of length n to retrieve the scaling factors c_i and an `HMMMatrix` object of size

```
int K = 2, M = 2, n = 4;

shared_ptr<HMMVector<float> >
  pi_ptr(new HMMVector<float>(K));
shared_ptr<HMMMatrix<float> >
  T_ptr(new HMMMatrix<float>(K,K));
shared_ptr<HMMMatrix<float> >
  E_ptr(new HMMMatrix<float>(M,K));

HMMVector<float> &pi = *pi_ptr;
HMMMatrix<float> &T = *T_ptr;
HMMMatrix<float> &E = *E_ptr;

// initial state probabilities
pi(0) = 0.2; pi(1) = 0.8;

// transitions from state 0
T(0,0) = 0.1; T(0,1) = 0.9;
// transitions from state 1
T(1,0) = 0.9; T(1,1) = 0.1;

// emissions from state 0
E(0,0) = 0.25; E(1,0) = 0.75;
// emissions from state 1
E(0,1) = 0.75; E(1,1) = 0.25;

HMM<float> hmm(pi_ptr, T_ptr, E_ptr);

sequence obs(n);
obs[0] = 0;
obs[1] = 1;
obs[2] = 0;
obs[3] = 1;

sequence hiddenseq(n);
hmm.viterbi(obs, hiddenseq);

HMMMatrix<float> F(n, K);
HMMVector<float> scales(n);
hmm.forward(obs, scales, F);
```

Figure 1: Using HMMlib.

```
float log_likelihood = hmm.likelihood(scales);

HMMMatrix<float> B(n, K);
hmm.backward(obs, scales, B);

HMMMatrix<float> pd(n, K);
hmm.posterior_decoding(obs, F, B, scales, pd);

shared_ptr<HMMVector<float>>
  new_pi_ptr(new HMMVector<float>(K));
shared_ptr<HMMMatrix<float>>
  new_T_ptr(new HMMMatrix<float>(K,K));
shared_ptr<HMMMatrix<float>>
  new_E_ptr(new HMMMatrix<float>(M,K));
hmm.baum_welch(obs,
               F,
               B,
               scales,
               *new_pi_ptr,
               *new_T_ptr,
               *new_E_ptr);

HMM<float> new_hmm(new_pi_ptr,
                   new_T_ptr,
                   new_E_ptr);
```

Figure 1: continued

$n \times K$ to retrieve the forward probabilities, and execute the forward algorithm to fill in these tables as follows:

```
HMMVector<double> scales(n);
HMMMatrix<double> F(n, K);
hmm.forward(obs, scales, F);
```

where `obs` is a standard C++ vector containing the observed sequence as a sequence of numbers $s_i \in \{0, 1, \ldots, M\}$.

Figure 1 shows a full example using HMMlib, in which both the Viterbi, forward, backward, Baum Welch and posterior decoding algorithms are executed. The example also shows how to use `float` as the template parameter instead of `double`.

III. Exploiting SSE instructions

The Streaming SIMD Extension (SSE) of the `x86` instruction set adds a number of SIMD (single instruction, multiple data) instructions. These instructions are executed on multiple data points at the same time, and are therefore useful in problems, where the same operation must be performed multiple times on a large amount of data. The SSE instructions use eight 128-bit registers (sixteen on a 64-bit CPU), each capable of holding e.g. four 32-bit single-precision floating point numbers or two 64 bit double-precision floating point numbers. To use these instructions, we use the Intel SSE intrinsics supported by most C/C++ compilers [I07]. They define a way to use the instruction set, using functions in C/C++. E.g. to add two tuples of four single-precision floating point numbers each, we may do as follows:

```
__m128 x = _mm_set_ps(1.2f, 2.3f, 4.3f, 1.4f);
__m128 y = _mm_set_ps(5.6f, 6.7f, 2.8f, 1.9f);
__m128 z = _mm_add_ps(x,y);
```

Equivalently the last line could be written, using the overloaded addition operator, as

```
__m128 z = x + y;
```

Similarly we could add two pairs of double-precision floating point numbers using the functions `_mm_set_pd` and `_mm_add_pd` on tuples of the type `__m128d`. In order to use the right type of operators depending on the type of floating point numbers used, we use a trait class, `SSEOperatorTraits`, that selects the right SSE operations:

```
template<>
struct SSEOperatorTraits<float, __m128> {
  ...
  static void set_all(__m128 &dest,
                      float &val) {
    dest = _mm_set1_ps(val);
  }

  static void sum(__m128 &c) {
    c = _mm_hadd_ps(c, c);
    c = _mm_hadd_ps(c, c);
  }
  ...
};

template<>
struct SSEOperatorTraits<double, __m128d> {
 ...
 static void set_all(__m128d &dest,
                     double &val) {
    dest = _mm_set1_pd(val);
  }

  static void sum(__m128d &c) {
    c = _mm_hadd_pd(c, c);
  }
  ...
};
```

When data is moved from memory into the SSE registers, the data must be 16-byte aligned to ensure maximum performance. A chunk of 128 bits (16 bytes) of data is 16-byte aligned if the address to the chunk in memory is a divisible by 16. Thus to make sure that reads and writes in `HMMMatrix`s and `HMMVector`s are efficient, we must ensure that each row in each matrix/vector starts at

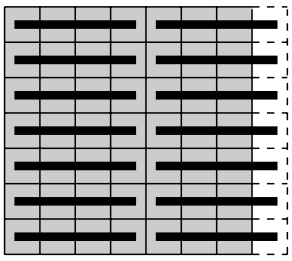


Figure 2: 16-byte aligned matrix with 7 rows and 7 columns. Each row has been padded with one extra column.

an address divisible by 16. This is done by padding extra (empty) columns to the matrix/vector, such that the number of bytes in each row becomes divisible by 16. Thus if a matrix/vector contains `floats`, the number of columns must be divisible by 4, as illustrated in Figure 2, and if it contains `doubles`, the number of rows must be divisible by 2. This is ensured, by using a traits class for allocating and deallocating `HMMMatrixs` and `HMMVectors`, utilizing the `_mm_malloc` and `_mm_free` intrinsics.

A. HMM algorithms using SSE instructions

We now describe how we optimize each of the algorithms in Section II-A using the SSE instruction set, obtaining a significant speed-up for each algorithm.

1) The forward, backward and Viterbi algorithms: These algorithms are very similar in their execution, and they can therefore be optimized using the same techniques. We will therefore only describe the optimization of the forward algorithm in detail.

The forward algorithm works by filling each row of $\hat{F}$ one at a time from the top down. For each entry, $\hat{f}_{ij}$, $\sum_l f_{i-1,l} t_{l,j}$ must be computed. However since all matrices are stored memory as described above, we cannot iterate the columns of T using SSE instructions. We therefore transpose T and iterate the rows of T' instead, computing $\sum_l f_{i-1,l} t'_{j,l}$ for each entry, $\hat{f}_{ij}$. Since the terms in this sum are independent from each other, and both $\hat{F}$ and T' are stored in memory as described above, this computation, may be optimized using SSE instructions, computing 2 or 4 terms of the sum at the same time, and adding these together:

```
sse_float_type p_sum;
sse_op_traits::set_all(p_sum, (float_type) 0.0);
for(int c = 0; c < no_chunks; ++c) {
  p_sum += F.get_chunk(i-1, c) *
           T_t.get_chunk(j, c);
}
```

where `sse_op_traits` is an instance of the `SSEOperatorTraits` traits class. Now `p_sum` ends up being a tuple of 2 or 4 floating-point numbers, thus to get the final result, we must sum these using the sum method of the `SSEOperatorTraits` traits class. When the computation of row i is finished, it may be scaled in the same way, using a chunk in which all entries are set to the value of the ith scaling factor. The essential part of the implementation of the Forward Algorithm now looks as follows:

```
for(int i = 1; i < n; ++i) {
  scales(i) = (float_type) 0.0;
  x = obs[i];
  for(int j = 0; j < K; ++j) {
    sse_float_type p_sum;
    sse_op_traits::set_all(p_sum, (float_type)
        0.0);
    for(int c = 0; c < no_chunks; ++c) {
      p_sum += F.get_chunk(i-1, c) *
               T_t.get_chunk(j, c);
    }
    sse_op_traits::sum(p_sum);
    float_type tmp;
    sse_op_traits::store(tmp, p_sum);
    tmp *= E(x,j);
    F(i, j) = tmp;
    scales(i) += tmp;
  }

  scales(i) = 1.0 / scales(i);
  sse_op_traits::set_all(scale, scales(i));
  for(int c = 0; c < no_chunks; ++c) {
    F.get_chunk(i,c) *= scale;
  }
}
```

The optimized implementation of the Viterbi Algorithm is very similar, since the only difference is that we must compute $\max_k(\tilde{v}_{i-1,k}) + \log t_{k,j}$ for each entry in $\tilde{v}_{ij}$ rather than the sum above. Finally in the Backward algorithm $\sum_l e_{s_{i+1},l} t_{jl} \hat{b}_{i+1,l}$ must be computed for each entry in $\hat{B}$. This can be done in the exact same way, since E is also stored, such that each row is 16-byte aligned.

2) Posterior Decoding: To compute the table of a posteriori probabilities for the posterior decoding, $\hat{f}_{ij}\hat{b}_{ij}/c_i$ must be computed for each entry. Since both $\hat{F}$ and $\hat{B}$ are 16-byte aligned in memory, we can compute each row of the table in chunks of size 2 or 4 in the following way:

```
for (int i = 0; i < n; ++i) {
  sse_float_type scale;
  sse_op_traits::set_all(scale, (float_type)
      (1.0/scales(i)));
  for (int c = 0; c < no_chunks; ++c) {
    post.get_chunk(i, c) = F.get_chunk(i, c) *
                           B.get_chunk(i, c) *
                           scale;
  }
}
```

We precompute c_i^{-1} in each iteration of the outer loop, and compute $\hat{f}_{ij}\hat{b}_{ij}c_i^{-1}$ rather than computing $\hat{f}_{ij}\hat{b}_{ij}/c_i$, because the SSE instruction for division is much slower than the instruction for multiplying chunks.

3) The Baum Welch Algorithm: As described in Section II-A4 the vector of initial state probabilities, may easily be updated by setting $\pi_j = \hat{a}_{1,j}\hat{b}_{1,j}/c_1$ for all j. Using the same technique as above, this is expressed as operations on chunks in the following way:

```
sse_float_type scale;
sse_op_traits::set_all(scale, (float_type) (1.0/
    scales(0)));
for (int c = 0; c < no_chunks; ++c) {
  pi_counts.get_chunk(c) = (F.get_chunk(0,c) *
                            B.get_chunk(0,c)) *
                            scale;
}
```

Now, to update T and E, we note that for each observed symbol s_i, we can add one term to all of the K^2 sums (7) composing the new transition matrix, and we can add one term to each of the K sums (8) composing the row corresponding to s_i in the new emission matrix. Since each row in both $\hat{F}$, $\hat{B}$, the new and the old transition and

emission matrices are 16-byte aligned in memory, these computations may be performed by adding terms to each row in the new transition matrix and to the row corresponding to s_i in chunks of size 2 or 4. We do this in the following way:

```
for (int i = 1; i < n; ++i) {
  x = obs[i];
  for(int j = 0; j < K; ++j) {
    sse_float_type prev_f;
    sse_op_traits::set_all(prev_f, F(i-1, j));
    for(int c = 0; c < no_chunks; ++c) {
      T_counts.get_chunk(j, c) +=
            prev_f * T.get_chunk(j,c) *
            E.get_chunk(x,c) * B.get_chunk(i,c);
    }
  }

  sse_op_traits::set_all(scale, (float_type)
      (1.0/scales(i)));
  for(int c = 0; c < no_chunks; ++c) {
    E_counts.get_chunk(x,chunk) +=
            F.get_chunk(i, c) *
            B.get_chunk(i, c) *
            scale;
  }
}
```

B. Running the optimized and non-optimized algorithms

Running the algorithms in HMMlib as described in Section II-B, will automatically cause the SSE optimized algorithms to be executed. Thus Figure 1 shows an example of how to use the optimized algorithms. If we want to use the non-optimized algorithms, we simply use a second template parameter for each of the classes `HMM`, `HMMMatrix` and `HMMVector`. E.g.:

```
HMMVector<float, float> vector =
        new HMMVector<float, float>(n);
HMMMatrix<float, float> matrix =
        new HMMMatrix<float, float>(n,K);
```

The second template parameter is per default linked to the SSE equivalent of the first, if it is not specified.

IV. Parallelizing using OpenMP

As described in the previous section, filling in e.g. the forward dynamic programming table is done one row at a time, setting $\hat{f_{ij}} = c_i e_{s_i,j} \sum_l \hat{f}_{i-1,l} t_{l,j}$. Noting that the entries of each row in $\hat{F}$ depends on no entries in $\hat{F}$ but the entries in the previous row, we see that each entry in the ith row of $\hat{F}$ may be computed in parallel. We will do this using the OpenMP API for shared memory multi-threading [OMP08].

Any execution of an OpenMP application begins with a single master thread. As the application executes, the master thread may *fork* at parallel regions, marked by the programmer, to create new threads. Each of these threads run concurrently to perform their job. As the OpenMP memory model is a shared memory model, most of the data is shared by all threads, but special constructs allow the programmer to declare variables as private to each thread, such that race conditions can be avoided. OpenMP consists of two basic constructs: pragmas and runtime routines. The OpenMP pragmas are used to specify regions in the code that may be executed in parallel, while the OpenMP runtime routines are primarily used to set and retrieve information about the runtime environment, such as retrieving the maximal number of threads, set the number of running threads or control thread scheduling.

To make a multi-threaded version of the algorithms described in Section II-A, we will use the OpenMP *parallel for* pragma. This is a work sharing directive, causing the master thread to distribute the iterations of a for loop, in which all iterations are completely independent of each other, over the threads (one of which is the master thread itself). The sum of an array may now be computed as follows:

```
double sum;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < 10000; i += 10) {
  sum += a[i]
}
```

Here `reduction(+:sum)` is used to avoid race conditions: it causes each thread to have its own private copy of `sum`, initialized to the identity element of the reduction operator (in this case +). When all threads have finished their part of the iterations, the private copies are added together and the result is assigned to the original sum variable. When the parallel region has finished, the master thread continues executing on it own, until it encounters the next parallel region.

Using the OpenMP parallel for pragma, the SSE optimized forward algorithm from III-A1 may be enhanced as follows:

```
for(int i = 1; i < n; ++i) {
  x = obsseq[i];
  float_type scale_i = (float_type) 0.0;
  #pragma omp parallel for reduction(+:scale_i)
  for(int j = 0; j < K; ++j) {
    sse_float_type p_sum;
    sse_op_traits::set_all(p_sum, (float_type)
        0.0);
    for(int c = 0; c < no_chunks; ++c) {
      p_sum += F.get_chunk(i-1, c) *
               T_t.get_chunk(j, c);
    }
    sse_op_traits::sum(p_sum);
    float_type tmp;
    sse_op_traits::store(tmp, p_sum);
    tmp *= E(x,j);
    F(i, j) = tmp;
    scale_i += tmp;
  }

  scales(i) = 1.0 / scale_i;
  sse_op_traits::set_all(scale, scales(i));
  for(int c = 0; c < no_chunks; ++c) {
    F.get_chunk(i,c) *= scale;
  }
}
```

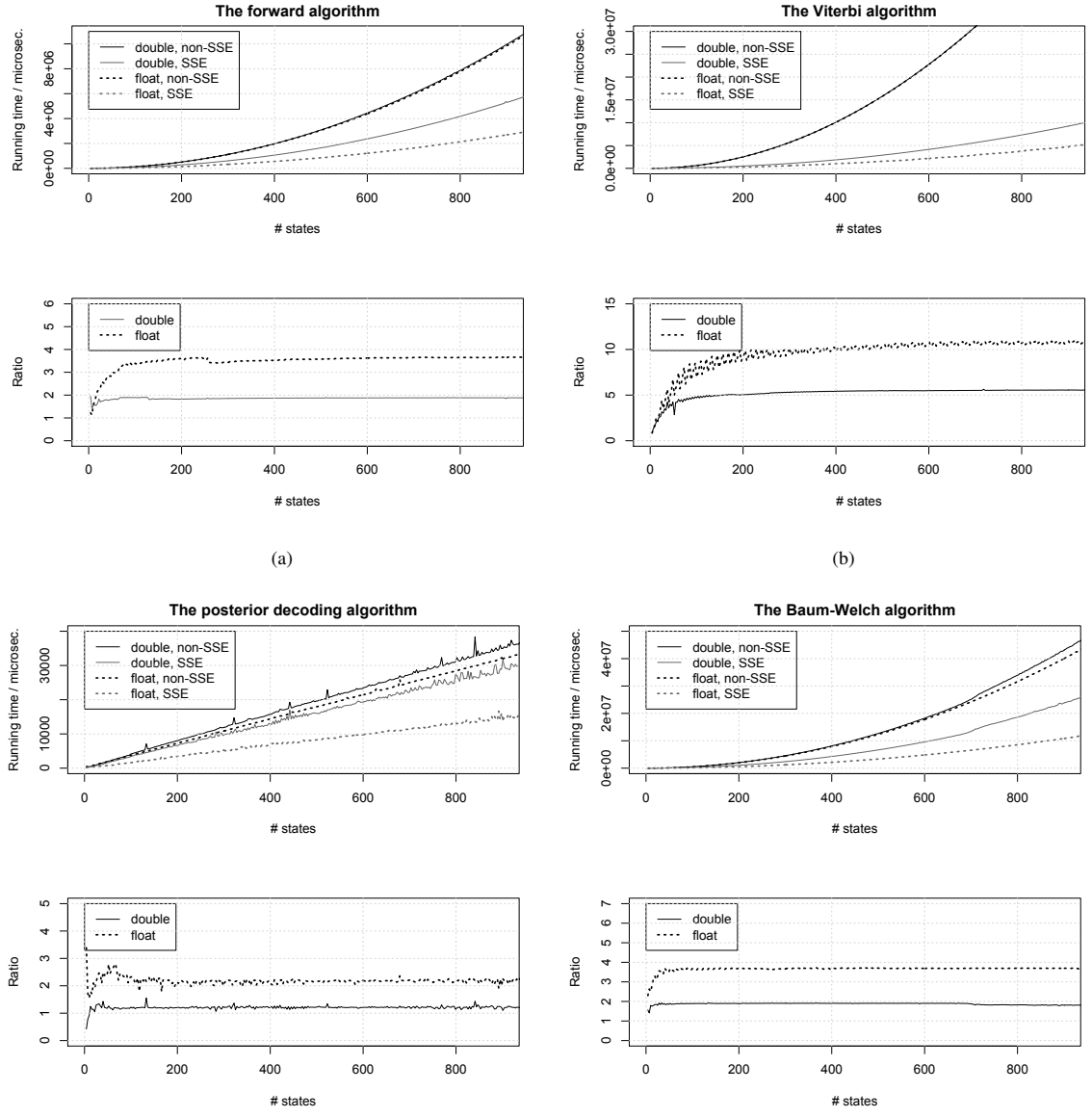


Figure 3: Experiments showing the speed-ups obtained by using SSE instructions.

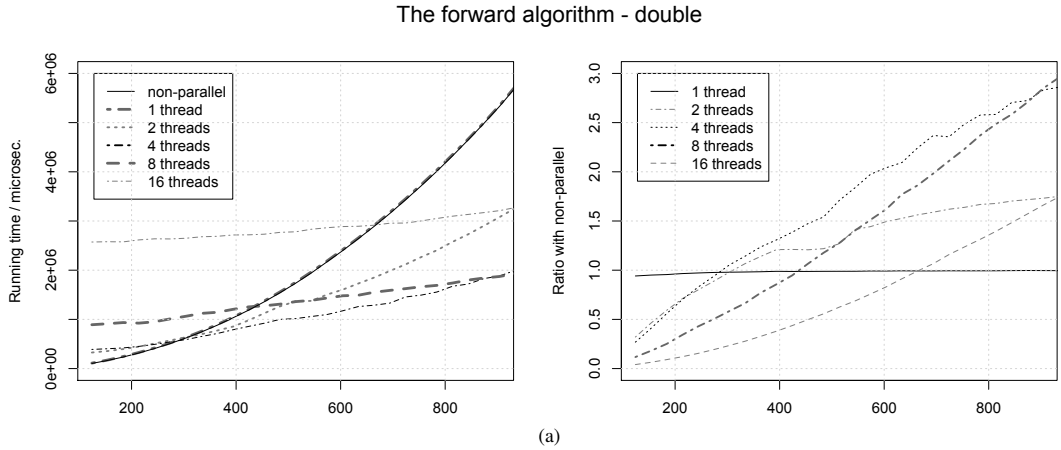
Each of the other algorithms may be parallelized using the same technique, filling each entry in each row in parallel, except that we do not need to use the `reduction` construct, since nothing is to be reduced.

V. Experiments

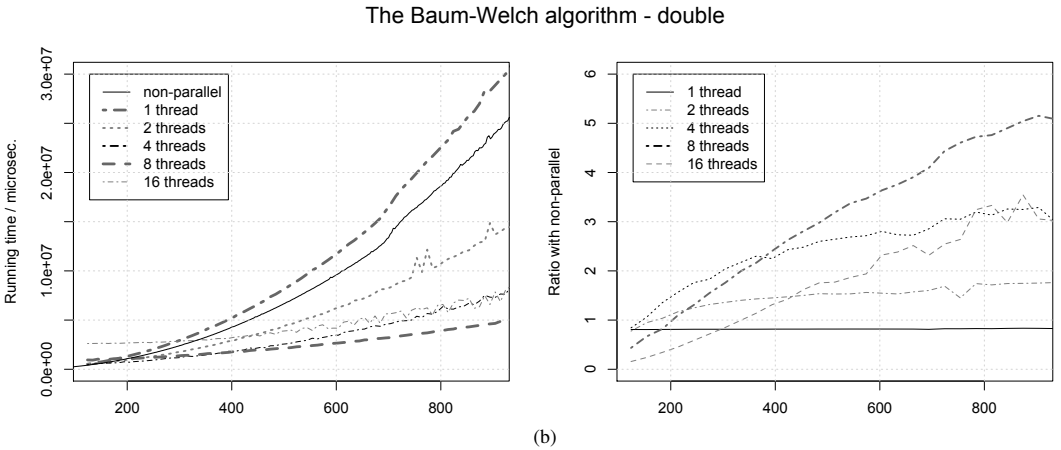
A. SSE optimization

All experiments in this section has been performed on a MacPro with two Intel quad-core Xeon processors (256kB L2 cache, 8MB L3 cache) running at 2.26GHz and 8GB main memory. This gives a total of 8 cores, but as each core supports hyper-threading, we get up to 16 virtual cores.

Figure 3 shows the speed-ups obtained by utilizing the SSE instruction set. All experiments have been performed with random observed sequences of length $n = 10000$ on differing randomly generated general HMMs with an alphabet of size $M = 16$ and different numbers of states, $K = 4, 7, 10, \ldots, 1024$. The generated transition matrices



(a)



(b)

Figure 4: Experiments showing the speed-ups obtained for the forward and Baum-Welch algorithms by parallelizing it using OpenMP.

are dense with random positive probabilities on each transition, and each state may emit every alphabet symbol with a random positive probability.

We note that we get the expected speed-ups, for models with more than 100 states, in the forward and Baum-Welch algorithms: close to at factor 2 when using `doubles` and close to a factor 4 when using `floats`. The plot for the backward algorithm is almost identical to the plot for the forward algorithm, and it is therefore not presented here.

In the experiments with the Viterbi algorithm we see a more significant speed-up than expected: more than a factor 5 when using `doubles` and more than a factor 10 when using `floats`. This speed-up is primarily because we are executing less instructions and greater data transfers when using SSE instructions. But this cannot explain the entire speed-up. One additional explanation could be, that using the SSE intrinsics `_mm_max_ps` and `_mm_max_pd` is much faster than using the standard `max` function in `C`, which is typically implemented using an if statement, leading to an expensive branching.

We do not see a speed-up as significant as we had hoped in the experiments with the posterior decoding algorithm. This is most likely due to the lower time complexity of the posterior decoding algorithm, $O(Kn)$, compared to the time complexity of $O(K^2n)$ for the other algorithms. Thus a minor fraction of the time is used in the optimized code (the actual algorithm) when executing the posterior decoding in HMMlib than when executing the other algorithms (a greater fraction of the time is used to set up data structures etc). But nevertheless we do see a significant decrease in the running time: close to a factor 1.2 when using `doubles` and a factor 2.2 when using `floats`.

B. OpenMP optimization

To test the OpenMP optimization, we have executed each of the algorithms using random sequences of length

10000 on HMMs with an alphabet of size $M = 16$ and different number of states, $K = 124, 134, \ldots, 1024$, using 1, 2, 4, 8 or 16 threads. The transition and emission matrices have been generated as described in the previous subsection. We present the numbers obtained when using double precision floating-point numbers here. The numbers for single precision floating-point numbers are similar. The experiments with the forward and Baum-Welch algorithms are summarized in Figure 4. All experiments was performed with SSE instructions enabled, and the running times for the non-parallelized implementations using SSE instructions are included; these are the lines labeled *non-parallel*, while the lines labeled *1 thread* show the running times for the parallelized implementations using only one thread.

In Figure 4a we see that the parallel version of the forward algorithm is just as fast as the non-parallel version, when running it using a single thread. Running it using two threads causes a significant speed-up for models with more than 174 states, getting closer to the theoretical limit of a factor 2 as the number of states grows. Using 4 threads we get an even more significant decrease in the running time, and using 8 threads we get a marginally greater speed-up for very big models. Using 16 threads yields a slow-down compared to using 8 threads for all model sizes. This is because of the overhead of synchronizing threads. Using 8 threads we have obtained a speed-up of the forward algorithm greater than a factor 5 (10 when using `floats`) for big models, compared to the non-optimized version (not using SSE instructions).

In the experiments with the multi-threaded version of the Baum-Welch algorithm, we see an even more impressive decrease in the running time: The multi-threaded version is faster for all sizes of the state space, and we get a speed-up greater than a factor 1.5 for models with more than 400 states, when running it using two threads, and a speed-up close to a factor 3 for models with more than 600 states, when running it using 4 threads. For models with more than 400 states we get the greatest speed-up, when running it with 8 threads: close to a factor 5 for models with more 800 states. Even though the parallel version of the algorithm is slower than the non-parallel version, when running the parallel version using only one thread, the non-parallel version is significantly outperformed by the parallel version, when using more than one thread. Using 8 threads we have obtained a speed up for the Baum-Welch algorithm greater than a factor 9 (18 when using `floats`) for big models, compared to the non-optimized version (not using SSE instructions).

The experiments with the backward and Viterbi algorithms show approximately the same speed-ups as seen in Figure 4a. They are therefore not presented here. Finally the experiments with the posterior decoding algorithm show the maximum speed-up of a factor 2 for models of all sizes, when using 8 threads.

VI. Conclusion and future work

In this paper we have presented a C++ library for constructing and analyzing general hidden Markov models. The library consists of a number of template classes and generic functions, parameterized with the precision of floating-point types and different types of hardware acceleration based on utilizing SSE instructions and multiple cores. We achieve significant speed-ups for all of the classical HMM algorithms except posterior decoding, where the speed-up is less than expected for reasons which must be studied further. In the future it would also be interesting to address the sparsity of the HMM models to see how this affect the running times.

References

[B09] Brask, A.T. Optimering af generelle skjulte Markovmodeller på multikerne CPUer og GPUer. Master Thesis, Department of Computer Science, Aarhus University, 2009.

[B06] Christopher, M. Bishop. Pattern Recognition and Machine Learning. Cambridge: Springer; 2006.

[E95] Eddy, S. Multiple alignment using hidden Markov models. Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology. 1995; 3:114-120.

[E96] Eddy, S.R. Hidden Markov models. Current opinion in structural biology. 1996;6(3):361-5.

[E98] Eddy, S.R. Profile hidden Markov models. Bioinformatics. 1998;14(9):755.

[F10] Fog, A. Optimizing software in C ++. 2010:1–160. Available at: http://www.agner.org/optimize/optimizing_cpp.pdf.

[R89] Rabiner, L.R. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. Proceedings of the IEEE. 1989;77(2):257–286.

[RJ86] Rabiner, L.R., Juang BH. An introduction to hidden Markov models. IEEE ASSp Magazine. 1986;3:4–16.

[I07] Intel® SSE4 Programming Reference. 2007.

[LB98] Lukashin, A.V., Borodovsky M. GeneMark.hmm: new solutions for gene finding. Nucleic acids research. 1998;26(4):1107-15.

[B94] Baldi, P., Chauvin, Y., Hunkapiller, T., McClure, M.A. Hidden Markov models of biological primary sequence information. Proceedings of the National Academy of Sciences of the United States of America. 1994;91(3):1059-63.

[SH04] Siepel, A., Haussler, D. Computational identification of evolutionarily conserved exons. Proceedings of the eighth annual international conference on Research in computational molecular biology. 2004:186.

[OMP08] OpenMP Application Program Interface: version 3.0, OpenMP Architecture Review Board; 2008.