Cache-Oblivious Parallel SIMD Viterbi for sequence search in HMMER

Miguel Ferreira
Instituto Superior Técnico
Av. Rovisco Pais, 1049-001 Lisboa - Portugal
Email: miguel.ferreira@ist.utl.pt

***Abstract*—HMMER is a commonly used bioinformatics tool that makes use of Hidden Markov Models (HMMs) to analyze and process biological sequences. One of its main homology engines is based on Viterbi decoding, which was already highly parallelized and optimized using Farrar's striped pattern with Intel's SSE2 instruction set extension [1].**

A new vectorization of the Viterbi decoding algorithm is proposed, based on a SSE2 inter-task parallelism scheme, similar to the DNA alignment algorithm proposed by Rognes [2]. Besides the adopted alternative vectorization approach, the proposed algorithm introduces a new partitioning of the Markov model that allows a significantly more efficient exploitation of the cache locality. Such optimization, together with an improved loading of the emission scores, allows the achievement of a constant processing throughput, regardless of the innermost-cache size and of the dimension of the considered model.

The proposed vectorized optimization of the Viterbi decoding algorithm was extensively evaluated and compared with HMMER3 decoder, proving to be a rather competitive alternative implementation. Being always faster than the already highly optimized HMMER's ViterbiFilter implementation, the proposed Cache-Oblivious Parallel SIMD Viterbi (COPS) implementation provides a constant throughput and proved to offer a processing speedup as high as 2, depending on the considered model size.

A second parallelization level using multi-threading with an intra-task wave-front pattern was also explored. This approach showed some reasonable good results, although the speedup is clearly sub-linear, and the scalability suffers with an increasing number of threads (mainly due to the synchronization and communication overhead). With a small number of threads however, the multi-threaded wave-front pattern achieved a substantial speedup, e.g., 6.5-fold for 8 threads. It is thus an interesting parallelization avenue to exploit in some cases, such as applications with very large models that are processed only a few times.

I. Background

A. Sequence Alignment Algorithms

One of the most widely used alignment algorithms for sequence homology search is the Smith-Waterman algorithm [3]. It computes an optimal local alignment and the respective similarity score between the most conserved regions of the two sequences, with a complexity proportional to $O(N^2)$. The algorithm is based on a Dynamic Programming approach that considers three possible sources of variations: insertions, deletions, and mutations. To ensure that a local alignment is found, the computed scores have a minimum value of 0, indicating a restart in the alignment. To circumvent the computational complexity of the Smith-Waterman and similar algorithms, alternative heuristic methods (like BLAST [4]) were developed. However, their lower complexity is obtained by sacrificing the resulting sensibility and accuracy.

An effective way to speed up these dynamic programming alignment algorithms is through parallelization. One of

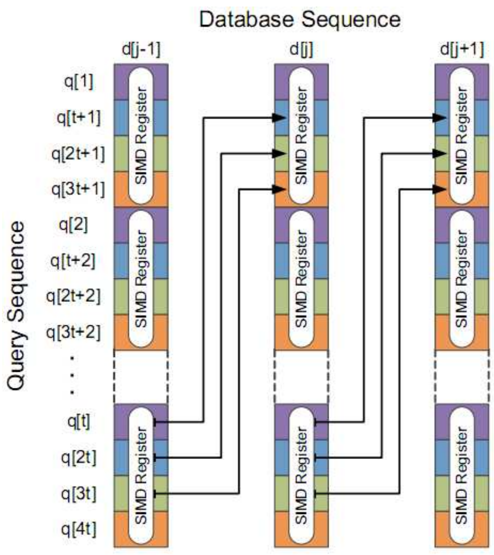

Fig. 1. Illustration of the decomposition pattern used by Farrar [1].

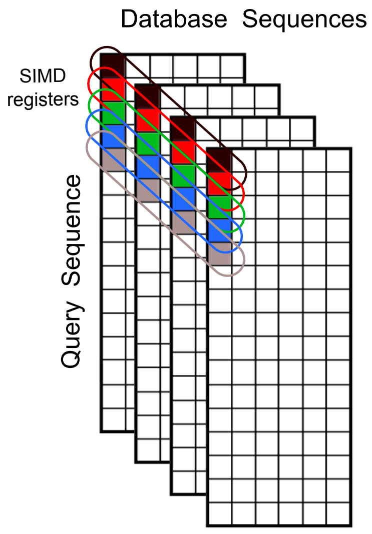

Fig. 2. Illustration of the decomposition pattern used by Rognes in the SWIPE tool [2].

the most innovative parallelization methods was proposed by Farrar [1], by exploiting vector processing techniques using Intel's SSE2 instruction set (see Figure 1). Farrar envisioned a striped data decomposition scheme, in which each vector contains cells from the same column distancing some fixed K from each other, in order to vectorize the inter-row data dependencies (deletions). The greatest problem of this method is concerned with the dependencies across 'segment sections' (continuous sections). In Farrar's algorithm, these dependencies are processed later, in a second inner loop (*Lazy-F loop*).

A different method was devised by Rognes, in his Swipe tool [2], which achieved a better performance than Farrar's. Rognes' method makes use of using vector processing (based on SSE2) to exploit an inter-task parallelism scheme (i.e., running multiple alignment tasks in parallel), by using a lock-step processing model (see Figure 2). Each vector is loaded with N different sequences, one in each vector element (or channel), and the algorithm aligns them concurrently against a target sequence, by using the N vector channels to hold the independent computed values. The drawbacks of this strategy are concerned with its restrictive application domain and limitations, deriving from the fact that the N alignments proceed on step from beginning to end. Any divergence on the program flow carries a high performance penalty, either as stoppage time or as wasted computing potential (e.g., empty padded cells).

Other authors have focused on the use of more specialized hardware architectures, such as GPUs [5], ASICs [6], or on scaling the algorithm onto a multi-node grid, usually by dividing the sequence database in blocks and searching each block independently.

B. Markov Models and Viterbi Decoding

Alternatively to searching with a single query sequence, several applications have adopted a consensus previously built from a family of similar sequences. This consensus structure is usually known as a 'consensus profile', and provides a more flexible way to identify homologs of a certain family, by highlighting the family's common features and by downplaying the divergences between the family's sequences.

A common method to conduct the profile homology search rests on a well-known machine learning technique: Hidden Markov Models. As an example, HMMs may be constructed to model the probabilistic structure of a group of sequences, such as a family of proteins. Such resulting HMM is then used to search within a sequence database, by computing the probability of that sequence being generated by the model. HMMs may also be used to find distant homologs, by iteratively building

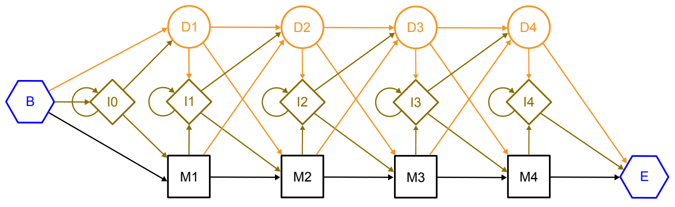

Fig. 3. Krogh and Haussler' HMM [7] for optimal global alignment.

and refining a model that describes them (such as in the SAM tool [8]).

In 1994, Krogh and Haussler [7] developed a straightforward and generalized Hidden Markov Model profile for homology searches, that emulates the results of a optimal alignment algorithm. The model is composed by three different types of states, respectively for matches/mismatches (M), insertions (I) and deletions (D), with transitions between the three types of states (see Figure 3).

The most important algorithms to process HMMs are the Forward algorithm, which gives the full probability for all possible model state paths; and Viterbi's algorithm, used to compute the most likely sequence of model states for the generation of that sequence. The complete path of states that is extracted by the application of Viterbi's procedure thus corresponds to an optimal alignment of the considered sequence against the profiled model.

The recurrence functions of the two algorithms are similar, with Viterbi's using a maximum operation while the Forward uses sum. The equations for Viterbi's algorithm are presented below:

$$V_j^M(i) = log\frac{e_{Mj}(x_i)}{q_{xi}} + Max\begin{cases} V_{j-1}^M(i-1) + log\ a_{M_{j-1}M_j} \\ V_{j-1}^I(i-1) + log\ a_{I_{j-1}M_j} \\ V_{j-1}^D(i-1) + log\ a_{D_{j-1}M_j} \end{cases}$$

$$V_j^I(i) = log\frac{e_{Ij}(x_i)}{q_{xi}} + Max\begin{cases} V_j^M(i-1) + log\ a_{M_jI_j} \\ V_j^I(i-1) + log\ a_{I_jI_j} \\ V_j^D(i-1) + log\ a_{D_jI_j} \end{cases}$$

$$V_j^D(i) = Max\begin{cases} V_{j-1}^M(i) + log\ a_{M_{j-1}D_j} \\ V_{j-1}^I(i) + log\ a_{I_{j-1}D_j} \\ V_{j-1}^D(i) + log\ a_{D_{j-1}D_j} \end{cases}$$

The emission probabilities are computed in odds-ratio form, giving the probability of the considered model against a standard random model. In order to avoid underflows resulting from the repeated products, the involved computations use logarithmic scores ('log-odds'). This conversion also replaces the products with sums, which further simplifies the calculations. For the Forward algorithm, the logarithmic transformation is significantly more complicated.

C. HMMER

HMMER [9] is a commonly used tool that uses Hidden Markov Models to do homology search. The original version of HMMER relied on a model architecture similar to Krogh-Haussler's model. The current version, HMMER 3.1b1 [10], employs the 'Plan 7' model architecture, presented in Figure 4. The core of this architecture is similar to Krogh-Haussler's, but Plan 7 has no D $\rightarrow$ I or I $\rightarrow$ D transitions, which simplifies the algorithm. Also, some special-states are added in the beginning and in the end, in order to allow for arbitrary restarts (thus making it a local alignment) and multiple repeats (multihit alignment). These special states can be parameterized to control the desired form of alignment, such as unihit or multihit, global or local.

This latest HMMER version also introduced a processing pipeline, which uses a combination of incremental filters. Each incremental filter is more accurate, restrictive and expensive than the previous one. All of these filters have been parallelized by vectorization, by using Farrar's striped processing pattern [1]. The ViterbiFilter, in particular, has been parallelized with 16-bit integer scores. Accordingly, the present work proposes a new parallelization strategy based on Rognes' processing pattern [2], with a novel strategy to improve cache efficiency.

II. Cache-Oblivious SIMD Viterbi with Inter-Sequence Parallelism

In this section, the proposed *COPS* (Cache-Oblivious Parallel SIMD Viterbi) approach is presented, representing an optimization of the Viterbi algorithm filter in local unihit mode (i.e., the mode corresponding to the original Smith-Waterman algorithm). Global alignment is not currently supported in the latest version of HMMER, since the E-value optimizations in the Forward and Viterbi filters are only valid for local alignment. Furthermore, HMMER3 considers the Insert states with identical emission probabilities as the background distribution, thus canceling its effect in the algorithm.

The presented implementation was developed on top of the HMMER suite as a standalone tool. A full integration into the HMMER pipeline was deemed unsuitable, since the pipeline was designed to execute a single sequence search at a time, while the proposed approach exploits inter-sequence parallelism, i.e., it concurrently processes in the SIMD SSE2 vectors.

A. Rognes strategy applied to Viterbi decoding

The alignment strategy proposed by Rognes [2] can be equally applied to the Viterbi decoding problem. In the implementation that is now proposed, 128-bit SIMD vectors, composed by eight 16-bit integer scores, are computed to simulaneously process eight different sequences.

The algorithm then proceeds to compute the recursive Viterbi relations, by using three auxiliary arrays to hold the previous values for the Match (M), Insert (I) and Delete (D) states. After each loop over the normal states, the special states (E and C) are updated. Since the proposed implementation does not support multhit alignments, the J transitions were removed from the original model.

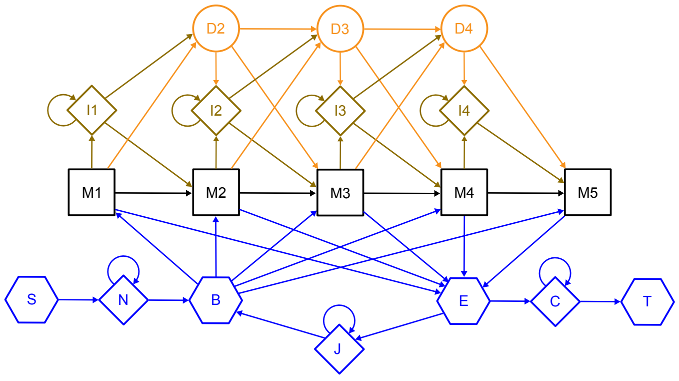

Fig. 4. Complete model used by HMMER3, which allows for multiple hits (multihit mode) in either local or global alignment modes.

The scores were discretized using a simple scale operation, with an additional bias (as was used in HMMER's implementation) and saturated arithmetic. Likewise the '-2.0 nat approximation' used by HMMER, the N $\rightarrow$ N and C $\rightarrow$ C transitions were set to zero, and a -2.0 score offset was added in the end. This value approximates the cumulative contribution of N $\rightarrow$ N and C $\rightarrow$ C transitions which, for a large L, is given by $\log \frac{L}{L+2}$. The B contributions thus become constant, since they only depend on the N values (which are constant) and on the J values (which are zero in unihit modes).

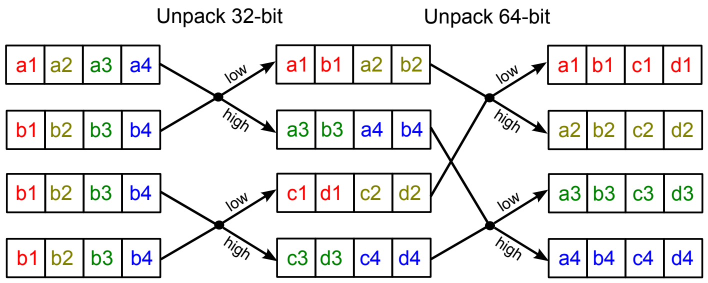

Fig. 5. Emission scores pre-processing using SSE unpacks, according to Rognes' Swipe tool.

An important step that is necessary in an inter-sequence SIMD implementation of Viterbi decoding is the pre-loading and arrangement of the per-residue emission scores. However, the emission scores depend on the searched sequences and they cannot be predicted, pre-computed and memorized before knowing those sequences. Also, each new batch of 8 sequences to search requires the loading of new emission scores.

Rognes' solution to this problem can also be adapted to Viterbi Decoding and consists on loading the emission scores for 8 different residues from the 8 sequences under processing, each from its own emission scores' array, before starting the inner loop of the model. The scores are then transposed from the original continuous pattern into a convenient striped pattern, by using the unpack and shuffle SSE operations. An illustration of the implemented processing pattern can be seen in Figure 5.

B. Inline pre-processing of scores

Rognes' method of pre-loading the emission scores before each inner loop iteration (i.e., iteration over the model states) suffers from a considerable handicap: it needs an additional re-write to memory. Accordingly, an alternative approach is herein proposed, illustrated in Listing 1 (lines 16-37). Instead of doing the transposition of all the emission scores for each tuple of residues in the outer loop of the algorithm (loop over sequence residues), the transposition was moved inwards to the inner loop, and subsequently unrolled for 8 iterations. Hence, each iteration pre-loads exactly 8 emission values from each of the 8 continuous arrays, which are subsequently transposed into 8 temporary striped SSE2 vectors and used in the computation of the next 8 model states. These 8 SSE2 register transpositions are inlined in each round of 8 inner loop iterations, and thus the inner loop must be unrolled into the

8 state-triplets that are processed by each loop iteration. As a result, the emission scores can be kept in close memory, thus improving the memory and cache efficiency. Furthermore, we avoid the re-writing of memory in the pre-loading phase, which constrats with Rognes' solution.

The vectorization approach that was adopted by this method requires the number of model states to be a multiple of 8 (for 8 channels of 16-bit integers), to support the 8-state loop step. In order to easily deal with this restriction, the model should be padded with dummy states up to a 8-state barrier. The dummy states carry dummy scores, set to $-\infty$, so that they have a null influence on the final results. These few extra dummy states have a negligible effect on the overall performance. According to the conducted evaluations (further detailed in the latest sections of this manuscript), the optimization of the loading procedure of the inlined scores lead to an execution time roughly 30% faster than the pre-loading method used by Rognes' tool.

C. Model Partitioning

One common problem that is often observed in these algorithms is concerned with the degradation of the cache efficiency when the score arrays exceed the capacity of the innermost-level caches (i.e.; L1D), leading to a spike in the number of cache misses and causing a substantial reduction of the overall performance. The same performance drops also happens in HMMER's Farrar-based ViterbiFilter: it only happens in larger models. The presented approach exceeds the capacity of the innermost cache sooner, since 8 times more transition scores and 8-fold larger dynamic programming arrays are required in the inner loop (see Table I).

TABLE I. ESTIMATES OF THE AMOUNT OF MEMORY (BYTES) USED BY THE CORE INNER LOOP. M IS THE MODEL'S LENGTH.

Data Structure	COPS	HMMER
Mmx, Dmx, Imx	$3 \times M \times 16$	$3 \times M \times 2$
Transition Scores	$8 \times M \times 16$	$8 \times M \times 2$
Emission scores	$M \times 16$	$M \times 2$
Auxiliary Emission array	24×16	–
~20 aux. variables	20×16	20×16
Total	$192 \times M + 700$	$24 \times M + 320$
Total minus Em. scores	$176 \times M + 700$	$22 \times M + 320$
Max. M to fill 32KB L1D	$\frac{32768-700}{192} \approx 167$	$\frac{32768-320}{24} \approx 1350$

In the preliminary evaluations that were conducted, the computation performance and the number of L1D cache misses for the COPS tool and the ViterbiFilter program of HMMER were accurately measured, by using models of varying lengths. The theoretical estimates suggested a *critical point* of full L1D utilization when using models of size ~1470 for ViterbiFilter and ~167 for ViterbiStream, when considering commercial off-the-shelf Intel processors, with 32KB L1 data caches. These points coincide rather well with the observed spikes in cache misses and drops of performance, which are strongly correlated in the results (see Figure 6).

To solve this cache efficiency problem, it was devised a *loop-tiling* (a.k.a. *strip-mining*) strategy based on a partitioning of the model states, in order to limit the amount of memory required by the core loop. The M, I and D model states are

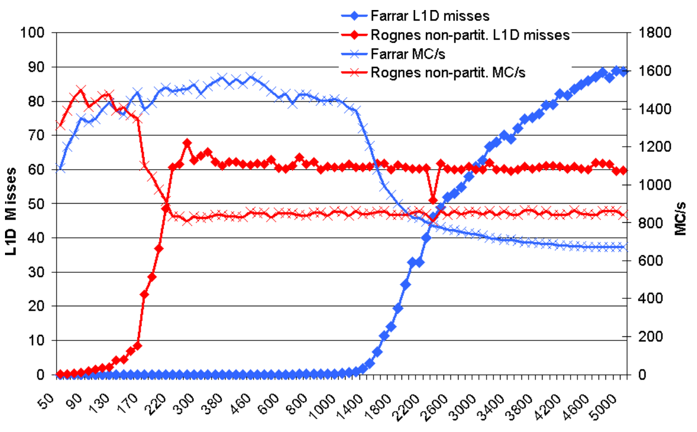

Fig. 6. Cache profiling results of HMMER's Farrar-based ViterbiFilter and Rognes-based COPS, on an Intel i7 Sandy Bridge with 32KB of L1D cache. Measured in millions of cells per second (MC/s) and billions of L1 data cache misses (L1D_PEND_MISS in Oprofile [11]).

split in blocks of *Maximum Partition Length (MP)* states, and the blocks are iterated over in a new outermost-loop. With this new and shorter inner loop, it is now possible to obtain an optimal cache use in the main loops — the outer loop iterates over the 8 sequences, while the inner loop iterates over a single partition of model states. The code transformation is illustrated in Table II. The resulting code is shown in Listing 1.

TABLE II. COMPARISON OF THE INNER LOOP CODE, BEFORE AND AFTER PARTITIONING THE MODEL

Original code, non-partitioned

```
▷ Loop through the sequence symbols
for $i \leftarrow 1$ to $L$ do
  ...
  ▷ Loop through the model states
  for $i \leftarrow 0$ to $M - 1$ do
    ▷ Core Viterbi code
  ...
  end for
  ▷ Update the special states
  ...
end for
```

Strip-mined code, partitioned

```
▷ Loop through the partitions
for $i \leftarrow 1$ to $Npartitions$ do
  ▷ Loop thru the sequence tokens
  for $i \leftarrow 1$ to $L$ do
    Load_Data_From_Last_Partition(i)
    ...
    ▷ Loop through the model states
    ▷ of the current partition
    for $i \leftarrow 0$ to $MP$ do
      ▷ Core Viterbi code
      ...
    end for
    ▷ Update the special states
    ...
    Store_Data_For_Next_Partition(i)
  end for
end for
```

The outer loop (new middle loop) over the sequences mostly re-uses the same memory locations (except for the emission scores) that are accessed in the inner core loop. Consequently these locations should be kept in close cache. By limiting the model states loop to a pre-defined MP number of state-triplets, it can be assured that the whole sequence loop (the middle loop in the new layout) does not access more than roughly $\sim (176 \times M + 320)$ bytes (disregarding the emission scores). With this optimization, the memory required by the inner loop is cached in close memory and repeatedly accessed over the whole sequence loop while in cache, thereby drastically reducing the occurrence of cache misses. The maximum partition length (MP) is adjusted to achieve an optimal cache occupation, i.e., one that fills the available capacity of the innermost data cache. The resulting processing pattern with the partitioned model is represented in Figure 7.

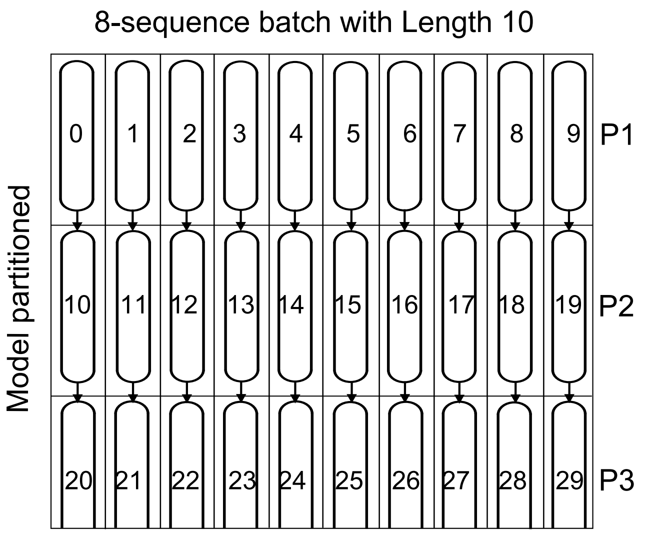

Fig. 7. Processing pattern of the adopted partitioned model,with an 8-sequence batch of length 10. The numbers represent the processing order of each partition. The arrows show the inter-partition dependencies.

Overall, the partitioned COPS implementation has an expected memory footprint of around $240 * M + 900$ bytes (corresponding to the original memory requirements of the non-partitioned COPS, plus the additional arrays that are required to store the inter-partition dependencies). It can thereby be estimated the value for the maximum partition length (MP) as the maximum model length that limits the memory footprint to the size of the L1D. Hence the value of MP can be determined by following the formula: $MP = \frac{L1Dsize - 900}{240}$.

Apart from a slight skew towards smaller lengths, justified by the sharing of the L1D cache with other variables not correlated with this processing loop, these estimated MP values coincide very consistently with the best partition lengths that were experimentically observed:

- 112 to 120 states, for 32KB L1D CPUs, (e.g. Intel Core, Core2, Nehalem, Sandy Bridge, Ivy Bridge and Haswell);
- around 48 states, for 16KB L1D CPUs, (e.g. AMD's Opteron Bulldozer and Piledriver);
- 216 to 224 states, for 64KB L1D CPUs, (e.g. AMD's Opteron K8, K10, Phenom I and II).

There are however two memory blocks that cannot be *strip-mined*:

- Emission scores, which must be refreshed (re-computed) for each new round of sequence tokens. These values are accessed only once, so it is counter-productive to consider their cacheability.
- Dependencies that must be exchanged between partitions. The last Match, Insert, and Delete contributions from each partition have to be carried on to the next partition, and so they have to be saved at the end of each partition. Each partition receives as input one line of previous states, with one state-triplet for each 8-fold round of sequences, and produces as output another line of values to be set to the next partition. These dependencies can be minimized to 3 values per sequence round (vE, Mnext, and Dcv) after re-

Listing 1 Pseudo-code of the proposed COPS approach, using SSE2 with 8x16-bit integers and saturated arithmetic.

```
▷ Loop through the partitions
for $p \leftarrow 1$ to $Npartitions$ do
    $Initialize\ Mmx,\ Imx,\ Dmx\ to\ -\infty$
    ▷ Loop through the sequence symbols
    for $i \leftarrow 1$ to $SequenceLength\ (L)$ do
        if p = 0 then
            ▷ First partition, initialize all to $-\infty$
            $xmxE \leftarrow Mnext \leftarrow Dcv \leftarrow -\infty$
        else
            ▷ Load data from previous partitions
            $xmxE \leftarrow PxmxE(i)$
            $Dcv \leftarrow PDcv(i)$
            $Mnext \leftarrow PMnext(i)$
        end if
        ▷ Loop thru the model states of the current partition
        for $k \leftarrow 0$ until $k = PartLength$ or $k + p \times PartLength > ModelLength$ step 8 do
            ▷ Load original scores, xmm 0 to 8
            $xmm[0] \leftarrow LOAD16(EMscoreSeq[0] + k)$
            $xmm[1] \leftarrow LOAD32(EMscoreSeq[1] + k)$
            ...
            ▷ Interleave 16-bit wide, xmm 8 to 15
            $xmm[8] \leftarrow UNPACK\_LOW16(xmm[0], xmm[1])$
            $xmm[9] \leftarrow UNPACK\_HIGH16(xmm[0], xmm[1])$
            ...
            ▷ Interleave 32-bit wide, xmm 16 to 23
            $xmm[16] \leftarrow UNPACK\_LOW32(xmm[8], xmm[10])$
            $xmm[17] \leftarrow UNPACK\_HIGH32(xmm[8], xmm[10])$
            ...
            ▷ Interleave 64-bit wide, xmm 24 to 31
            $xmm[24] \leftarrow UNPACK\_LOW64(xmm[16], xmm[18])$
            $xmm[25] \leftarrow UNPACK\_HIGH64(xmm[16], xmm[18])$
            ...
            ▷ Macro with the inner loop code, xmm 24 to 31
            ▷ Arguments: (model state index, xmm array index)
            $COMPUTE\_STATE\_TRIPLET(k + 0, 24)$
            $COMPUTE\_STATE\_TRIPLET(k + 1, 25)$
            ...
        end for
        ▷ Compute and update the special flanking states
        if $k + p \times PartLength < ModelLength$ then
            ▷ Not the last partiton, store data for next partition
            $PxmxE(i) \leftarrow xmxE$
            $PDcv(i) \leftarrow Dcv$
            $PMnext(i) \leftarrow Mnext$
        else
            ▷ Final partition, update the definitive pseudo-states
            $xmxC \leftarrow VMAX16(xmxC, xmxE)$
        end if
    end for
end for
return $Undiscretize(VMAX16(xmxC, t\_CT))$
```

Listing 2 Core inner loop code, used in the macro COMPUTE_STATE_TRIPLET State index, E.M. index). Variable k is the State index

```
▷ Use M value partially computed in last iteration
$Mnext \leftarrow VMAX16 \begin{cases} Mnext \\ xmxB + t_{BM}(k) \\ xmx[EMindex] \end{cases}$

$xmxE \leftarrow VMAX16(vE, Mnext)$

▷ Load scores from last column
$Dpv \leftarrow Dmx(k)$
$Ipv \leftarrow Imx(k)$
$Mpv \leftarrow Mmx(k)$

▷ Compute and store scores of this column
$Mmx(k) \leftarrow Mnext$
$Dmx(k) \leftarrow Dcv$
$Imx(k) \leftarrow VMAX16 \begin{cases} Mpv + t_{MI}(k+1) \\ Ipv + t_{II}(k+1) \end{cases}$

▷ Preempetive computation of next-column D score
$Dcv \leftarrow VMAX16 \begin{cases} Mnext + t_{MD}(k+1) \\ Dcv + t_{DD}(k+1) \end{cases}$

▷ Partially compute M score for next column
$Mnext \leftarrow VMAX16 \begin{cases} Mpv + t_{MM}(k) \\ Ipv + t_{IM}(k) \\ Dpv + t_{DM}(k) \end{cases}$
```

factoring the core code and moving the computation of Mnext with the 3 state dependencies to the end. The refactored inner loop code can be seen in Listing 2.

After partitioning, the overall performance behaved remarkably close to what had been predicted, maintaining the same level of caches misses and computation speed for any model length (see Figure 8). As it can be seen in this figure, COPS even managed to be slightly faster than HMMER's ViterbiFilter for models up to ∼1200. For longer models, COPS gains a close to 2-fold speedup over ViterbiFilter, due to the latter's cache degradation. In fact, when compared to the non-partitioned COPS code, the partitioned version was about 50% faster for long models (>1000 bps).

D. Wave-front multi-threading of the model partitions

After having a partitioned model, each partition can be seen as a 'chunk' of data to process. This data layout seems particularly suitable for an additional level of parallelization: multi-threading using a wave-front pattern of partitioned chunks. The speedup is expected to be sub-linear, but still this additional parallelization level is an interesting approach, which could be applied to other areas (e.g.; single tasks that cannot be trivially decomposed into independent threads).

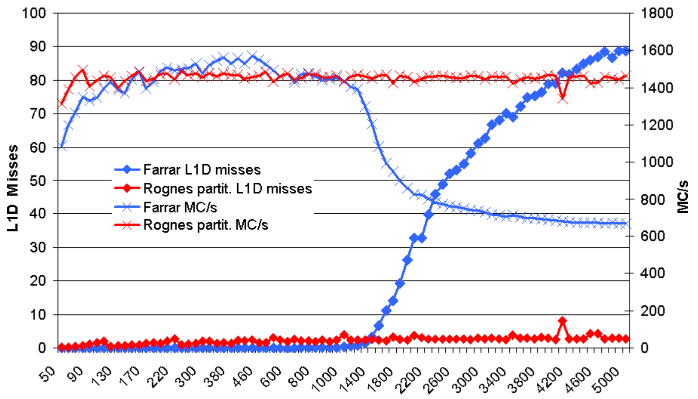

Fig. 8. Cache profiling results of HMMER's Farrar-based ViterbiFilter and of the new partitioned COPS, on an Intel i7 Sandy Bridge with 32KB of L1D cache. Measured in millions of cells computed per second (MC/s) and billions of L1 data cache misses.

With a partitioned model, it is possible to add SMP multi-threading to parallelize the partitions. Some number, N, of partitions can be simultaneously processed by N threads, following a wave-front technique — each thread starts its partition of the i-th sequence residue when the previous thread has finished the previous partition of the same i-th sequence residue (Figure 9). Synchronization of the threads, for each iteration of the outer loop (each sequence residue), is thus required.

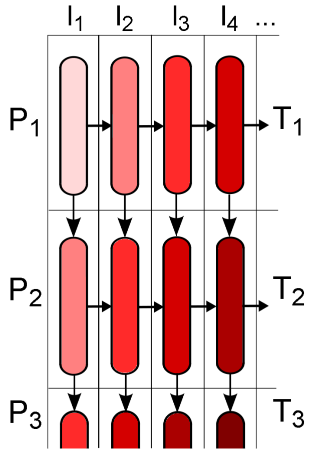

Fig. 9. Multi-threaded Wave-front pattern. Blocks are processed concurrently in a diagonal pattern (same color for concurrent partitions).

There are two synchronization concerns involved in the multi-threaded wave-front strategy: synchronization of all threads at the start of the algorithm (since it is run multiple times using the same N threads), and synchronization of data between threads processing inter-dependent chunks. For the inter-call synchronization, the method chosen was to use per-thread local call counters and a single global counter. The master thread increments the global counter, and each worker thread compares the global value against its local counter, yielding the CPU if they do not match. s For the synchronization and communication between the partitions, it was chosen a simple and efficient solution:

- $Nthreads$ arrays of flags, one for each pair (partition, sequence symbol), wherein threads can wait and signal one another.
- Unique arrays (with length equal to the sequence length) for the data that needs to be exchanged. Only one array is needed for each type of data, because only one transfer of data can occur at any given time for each sequence symbol (i.e.; the data is propagated

only one thread at a time, every time the active thread finishes and passes its data to the next partition thread). Only one thread is active processing a sequence index at a given time, any other thread is either in a previous index or waiting on the preceding partition thread.

This synchronization technique has the precious benefit of not using any locks or blocking primitives. No memory barriers are needed, because stale data is not a problem for the program. Data that is not updated will cause no consistency problems, rather it will merely lead to more waiting (yielding) cycles. The flags are only reset by the master thread, before starting the algorithm.

In order to maintain cache efficiency, the partition length must also be adapted to allow for the optimal use of the available threads, minimizing thread idleness. Ideally, the number of partitions should be a multiple of the number of threads, although this may not always be possible or the optimal case in practice (for instance, it could lead to a large number of small partitions, which would increase the inter-partition overhead).

III. Results and Discussion

To conduct a comparative and more comprehensive evaluation of the proposed approach, COPS was ran against the isolated ViterbiFilter implementation of HMMER 3.1b1, which uses the striped approach based on Farrar's work. The adopted benchmarks, consisted on Dfam database of Human DNA HMMs [12], and two genomes retrieved from the NCBI archive: *Homo Sapiens* (Human) and *Macaca Fascicularis* (Macaque).

All the timings were measured in total walltime, by using the Linux *ftime* function. The benchmarks were run on two different machines:

- Intel Core i7-3770KK, with Ivy Bridge architecture, 6-core, 3.50 GHz, with 32KB of L1 data cache;
- AMD Opteron 6276, with Bulldozer architecture, 8-core, 2.3 GHz, with 16KB of L1 data cache.

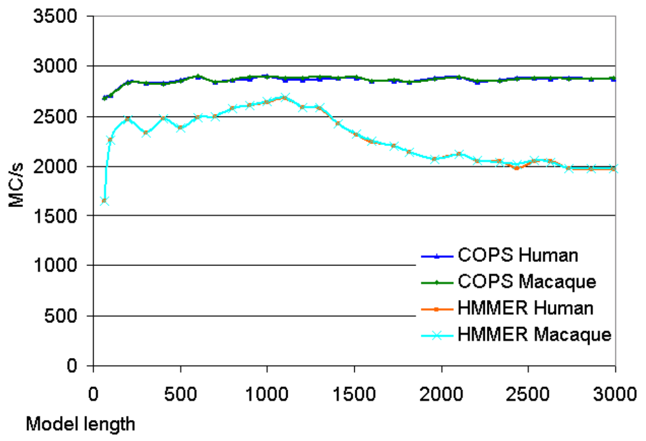

Fig. 10. Performance of COPS and HMMER's striped ViterbiFilter, on an Intel Core i7 Sandy Bridge, 32KB of L1D cache (millions of cells per second).

Figure 10 and Figure 11 represent the performance (in millions of cell updates per second) of the two implementations and the speedup of the presented approach, respectively, when

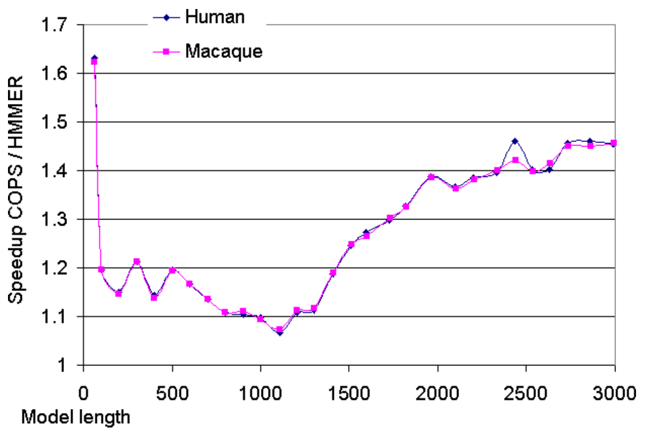

Fig. 11. Speedups of COPS against HMMER's striped ViterbiFilter, on an Intel Core i7 Sandy Bridge, 32KB of L1D cache.

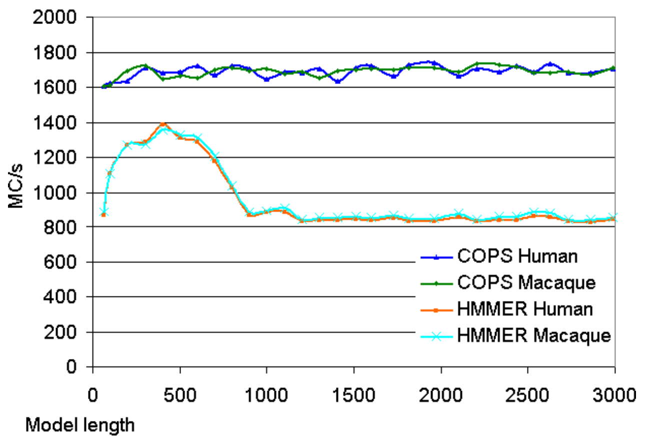

Fig. 12. Performance COPS and HMMER's striped ViterbiFilter, on an AMD Opteron Bulldozer, 16KB of L1D cache (millions of cells per second).

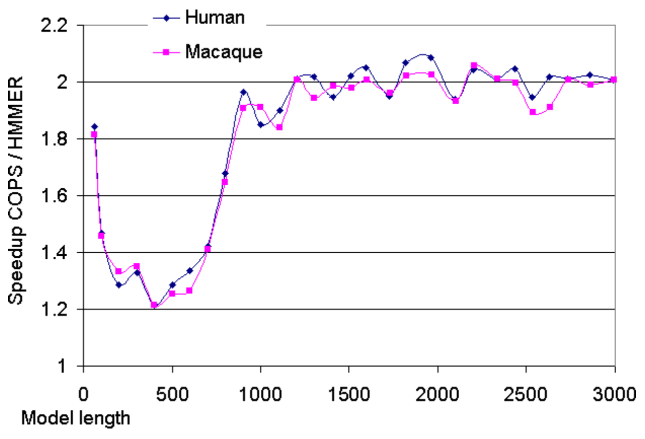

Fig. 13. Speedups of COPS against HMMER's striped ViterbiFilter, on an AMD Opteron Bulldozer, 16KB of L1D cache.

using the Intel core i7 processor. Figure 12 and Figure 13 represent similar results, observed in the AMD processor.

For short models (< 100 bps), the penalizing overhead of Farrar's Lazy-F loop is clearly evident. As a result, HMMER's ViterbiFilter has a very poor performance on these models. In contrast, the proposed COPS solution does not suffer from this problem and presents a much smaller performance penalty in these small models (mainly from the initialization costs between each inner-loop execution). As a result, COPS achieved a considerable 1.7-fold speedup vs HMMER in these models.

For medium-length models (between 100 and 500 bps on 16KB-L1D machines, and up to ∼1000 on 32KB-L1D machines), HMMER implementation is about as good as the proposed COPS, reducing the observed speedup to about 1.2-fold. These correspond to the model lengths wherein the striped version does not exceed the size of the innermost data cache.

For longer models, from 500 bps or 1000 bps (depending on the L1D size), it can be observed that the performance of HMMER quickly deteriorates as the model's length increases, and the memory requirements of the standard Farrar approach reach the maximum that the innermost caches can provide (usually 32K on Intel's L1D cache). In contrast, the proposed inter-sequence COPS is able to consistently maintain the same performance level with increasingly long models, thus achieving a 2-fold speedup on AMD's, and a 1.5-fold on Intel's, against the HMMER version for longer models.

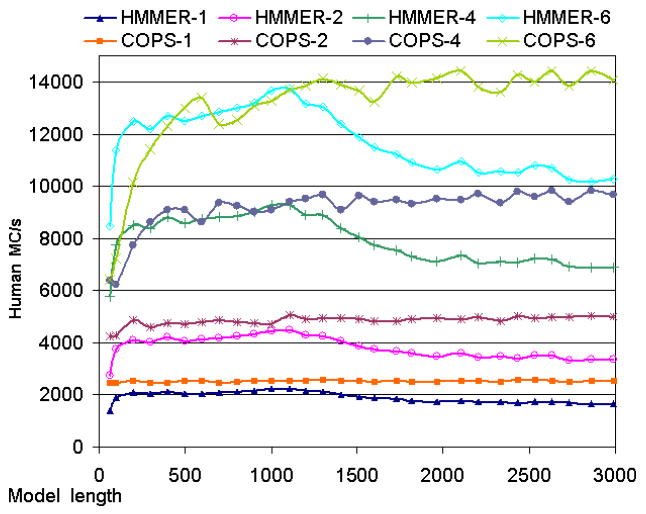

Fig. 14. Performance of the multi-threaded COPS and HMMER's ViterbiFilter, on the Intel Core i7 (values in millions of cells per second).

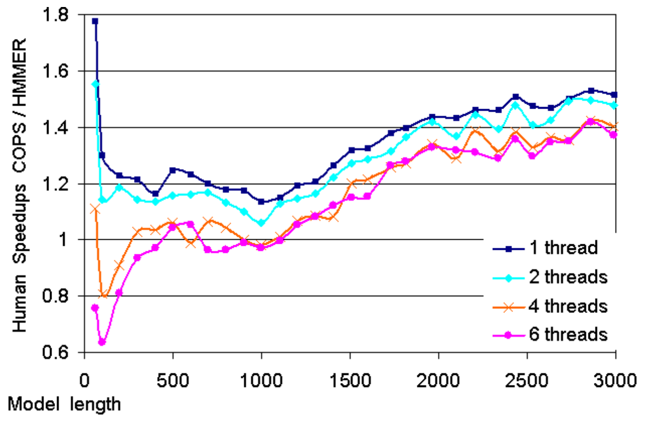

Fig. 15. Speedups of the multi-threaded COPS vs HMMER's ViterbiFilter, on the Intel Core i7 (values in millions of cells per second).

A. Evaluation of the Wave-front Multi-threading

The multi-threaded version of COPS was evaluated by comparing it against a standalone HMMER's ViterbiFilter running in a multi-threaded workpool. The same datasets and target machines described earlier were also used for this evaluation.

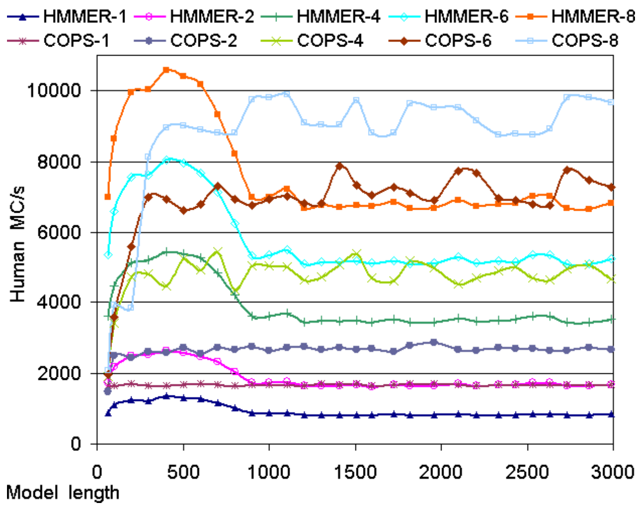

Fig. 16. Performance of the multi-threaded COPS and HMMER's ViterbiFilter, on the AMD Opteron (values in millions of cells per second).

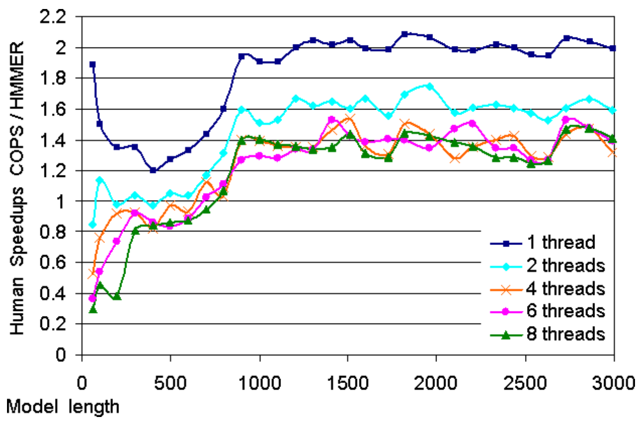

Fig. 17. Speedups of the multi-threaded COPS vs HMMER's ViterbiFilter, on the AMD Opteron (values in millions of cells per second).

The experimental results show that HMMER's Intra-task implementation is able to maintain a linear speedup over the number of threads/scores. This is was the expected outcome, given that it is processing each alignment task independently, and there are no inter-thread dependencies (except for the shared work pool).

In the case of COPS, the results show a clear sub-linear speedup, with a considerable drop in the added speed per core, as the number of cores increase. This was also an expected outcome, since there is substantial communication between the threads processing the partitions. The start and end delays inherent in the wave-front pattern also hindered the multi-threading performance.

An interesting point of note are the wide variations in the serial fraction metric along model lengths, reflecting an efficiency of multi-threading that depends on the specific model length. Some model lengths allow for a perfect work decomposition between partitions and threads, while others do not, and leave some unbalanced workloads. For the shorter models, the speedup per added thread deteriorates faster because the model length restricts the number of threads that can effectively be used: many threads processing a short model inevitably leads to either very short partitions, or idle threads from an unbalanced work distribution.

IV. Conclusions

A new vectorization of the Viterbi decoding algorithm was proposed to process arbitrarily sized HMM models. The presented algorithm is based on a SSE2 inter-task parallelism scheme, similar to the DNA alignment algorithm proposed by Rognes [2]. Besides the adopted alternative vectorization approach, the proposed algorithm introduces a new partitioning of the Markov model that allows a significantly more efficient exploitation of the cache locality. Such optimization, together with an improved loading of the emission scores, allows the achievement of a constant processing throughput, regardless of the innermost-cache size and of the dimension of the considered model.

According to the extensive assessments and evaluations that were conducted, the proposed vectorized optimization of the Viterbi decoding algorithm proved to be a rather competitive alternative implementation, when compared with the state of the art HMMER3 decoder. Being always faster than such already highly optimized HMMER's ViterbiFilter implementation, the proposed implementation provides a constant throughput and proved to offer a processing speedup as high as 2, depending on the considered HMM model size.

This work also explored the potential of multi-threading parallelization using a wave-front pattern. This consisted in an Intra-task parallelization of each COPS execution, as opposed to an Inter-task trivial parallelization with independent executions. The results were fairly good although the parallelization is clearly not scalable for a large number of threads, due to the communication overhead. For a small number of threads however, the obtained speedup through multi-threading was significant, not much lower than a linear speedup (i.e. 6.5-fold speedup for 8 threads). Hence, it has been shown to be an interesting parallelization avenue to employ in some areas, e.g., for very large models that are searched only a few times.

Future work may still explore other avenues for increasing the exploited parallelization using the adopted inter-task strategy, together with the newly proposed partitioning method (e.g.: application of the same partitioning model to the striped approach). Such future work may also extend this approach to Intel's recent instruction-set extension AVX2, allowing the processing of twice more vector elements at a time.

V. Acknowledgment

The author would like to thank Professor Nuno Roma and Professor Luís Russo, for their valuable ideas, guidance and support.

References

[1] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[2] T. Rognes, "Faster smith-waterman database searches with inter-sequence simd parallelisation," *BMC bioinformatics*, vol. 12, no. 1, p. 221, 2011.

[3] T. Smith, M. Waterman *et al.*, "Identification of common molecular subsequences," *J. mol. Biol*, vol. 147, no. 1, pp. 195–197, 1981.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[5] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer, "Accelerating hmmer on gpus by implementing hybrid data and task parallelism," in *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*. ACM, 2010, pp. 418–421.

[6] S. Derrien and P. Quinton, "Hardware acceleration of hmmer on fpgas," *Journal of Signal Processing Systems*, vol. 58, no. 1, pp. 53–67, 2010.

[7] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler, "Hidden markov models in computational biology: Applications to protein modeling," *Journal of molecular biology*, vol. 235, no. 5, pp. 1501–1531, 1994.

[8] K. Karplus, C. Barrett, and R. Hughey, "Hidden markov models for detecting remote protein homologies." *Bioinformatics*, vol. 14, no. 10, pp. 846–856, 1998.

[9] S. R. Eddy, "Profile hidden markov models." *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[10] ——, "Accelerated profile hmm searches," *PLoS Computational Biology*, vol. 7, no. 10, p. e1002195, 2011.

[11] J. Levon and P. Elie, "Oprofile - a system profiler for linux," http://oprofile.sourceforge.net, 1990.

[12] T. J. Wheeler, J. Clements, S. R. Eddy, R. Hubley, T. A. Jones, J. Jurka, A. F. Smit, and R. D. Finn, "Dfam: a database of repetitive dna based on profile hidden markov models," *Nucleic acids research*, vol. 41, no. D1, pp. D70–D82, 2013.