An Efficient Cache-Oblivious Parallel Viterbi Algorithm

Rezaul Chowdhury, Pramod Ganapathi, Vivek Pradhan,
Jesmin Jahan Tithi, and Yunpeng Xiao

Department of Computer Science, Stony Brook University, New York, USA

Abstract. The Viterbi algorithm is used to find the most likely path through a hidden Markov model given an observed sequence, and has numerous applications. Due to its importance and high computational complexity, several algorithmic strategies have been developed to parallelize it on different parallel architectures. However, none of the existing Viterbi decoding algorithms designed for modern computers with cache hierarchies is simultaneously cache-efficient and cache-oblivious. Being oblivious of machine resources (e.g., caches and processors) while also being efficient promotes portability. In this paper, we present an *efficient cache- and processor-oblivious Viterbi algorithm* based on *rank convergence*. The algorithm builds upon the parallel Viterbi algorithm of Maleki et al. (PPoPP 2014). We provide empirical analysis of our algorithm by comparing it with Maleki et al.'s algorithm.

Keywords: Viterbi algorithm; cache-efficient; cache-oblivious; recursive; divide-and-conquer; parallel; multi-instance; rank convergence;

1 Introduction

The Viterbi algorithm [36, 37] proposed by Andrew J. Viterbi in 1967, is a dynamic programming algorithm that finds the most probable sequence of hidden states, called the "Viterbi path" from a given sequence of observed events in the context of a hidden Markov model (HMM).

Motivation. The Viterbi algorithm has numerous real-world applications. Although it was originally used for speech recognition in CDMA technology, in the last 25 years, it has been heavily used in computational biology and bioinformatics for finding coding and non-coding regions of an unlabeled string of DNA nucleotides (i.e., gene finding) [3], prediction of protein-coding regions in genome sequences modeling families of related DNA or protein sequences and prediction of secondary structure elements in proteins [24], CpG island [17], promoter [29] and conserved elements detection [30]. Apart from computational biology, Viterbi algorithm is used in TDMA system for GSM [15], television sets [28], satellite and space communication [21], magnetic recording systems [23], parsing context-free grammars [22], and part-of-speech tagging [16]. Therefore, improving performance of Viterbi algorithm will likely to have impact in these areas as well.

When the input data becomes too large to fit into a cache, between two algorithms that perform the same set of CPU operations, the one that is more *cache-efficient*, i.e., causes fewer block transfers (or IO) between adjacent levels of caches is likely to run faster. Though there have been a lot of efforts and successes in parallelizing the Viterbi algorithm, there is little work in the realm of designing cache-efficient Viterbi algorithms that are also *cache-oblivious* [20], i.e., independent of cache parameters such as cache sizes and block sizes. Similarly, a *processor-oblivious* [12] algorithm does not use the number of processors in the algorithmic description. A cache- and processor-oblivious algorithm is more likely to be portable across machines. To the best of our knowledge, we present the first provably cache-efficient cache-oblivious parallel Viterbi algorithm.

We use *dynamic multithreading model* [14] and *ideal cache model* [20] to measure parallelism and serial cache complexity, respectively.

Related work. Several efficient cache- and processor-oblivious recursive divide-and-conquer algorithms for solving dynamic programs (DP) have been developed [2, 4, 7–11, 13, 31, 33, 34]. But the approach used in those papers assumes that the set and sequence of DP cell updates to be performed do not depend on the data values in the DP table which is not true in case of Viterbi DP.

One can use auto-parallelizers to parallelize sequential Viterbi programs. Fisher and Ghuloum [19] present a method in which loop body instances are represented in a closed form using function compositions. Reduction is then applied for parallelization. Chin et al. [6, 5] use second-order generalization and induction derivation to generate divide-and-conquer parallel programs. None of these methods exploit parallelism across stages. Also the generated parallel programs are not cache-efficient.

The parallel Viterbi algorithm [18] used for homology search in HMMER uses SSE2 instructions and reduces L1 cache misses. Though the phrase "cache-oblivious" appears in the title of the paper, the presented algorithm is not oblivious of the cache parameters as it uses loop-tiling with the tile size determined based on the size of the L1 data cache. Also the algorithm works only for three states, and it is not clear how the method behaves for arbitrarily large number of states as in the case of a general Viterbi algorithm.

The EasyPDP system [32] parallelizes the Viterbi algorithm and also reduces cache misses. However, it requires the user to specify loop tile sizes making it cache-aware. Also the reduction in cache misses is not significant.

The Viterbi algorithm is inherently sequential across stages which constraints parallelism along the time dimension. A parallel Viterbi algorithm presented in [26, 27] based on rank convergence is the first to exploit parallelism across stages. However, this algorithm is processor-aware and not cache-efficient.

Our contributions. Our major contributions are: (1) an *efficient cache-oblivious parallel multi-instance Viterbi algorithm* (Section 3), (2) an *efficient cache-oblivious parallel single-instance Viterbi algorithm* (Section 5) based on our multi-instance algorithm (Section 3) and Maleki et al.'s rank convergence algorithm (Section 4), and (3) *experimental results* (Section 6) comparing our algorithms with Maleki et al.'s algorithms on modern multicore platforms.

2 Cache-inefficient Viterbi algorithm

In this section, we formally describe the Viterbi dynamic program (DP), and describe a simple cache-inefficient Viterbi algorithm based on divide-and-conquer.

Formal specification. The Viterbi DP is described as follows. We are given an observation space $O = \{o_1, o_2, \ldots, o_m\}$, state space $S = \{s_1, s_2, \ldots, s_n\}$, observations $Y = \{y_1, y_2, \ldots, y_t\}$, transition matrix A of size $n \times n$, where $A[i, j]$ is the transition probability of transiting from s_i to s_j, emission matrix B of size $n \times m$, where $B[i, j]$ is the probability of observing o_j at s_i, and initial probability vector (or initial solution vector) I, where $I[i]$ is the probability that $x_1 = s_i$. Let $X = \{x_1, x_2, \ldots, x_t\}$ be a sequence of hidden states that generates $Y = \{y_1, y_2, \ldots, y_t\}$. Then the matrices P and P' of size $n \times t$, where $P[i, j]$ is the probability of the most likely path of getting to state s_i at observation o_j and $P'[i, j]$ stores the hidden state of the most likely path (i.e., Viterbi path) are computed as follows. $P[i, j] = I[i] \cdot B[i, y_1]$, and $P'[i, j] = 0$ when $j = 1$. Otherwise (i.e., when $j > 1$):

$$P[i, j] = \max_{k \in [1,n]}(P[k, j-1] \cdot A[k, i] \cdot B[i, y_j]),$$
$$\text{and } P'[i, j] = \operatorname{argmax}_{k \in [1,n]}(P[k, j-1] \cdot A[k, i] \cdot B[i, y_j]),$$

```
LOOP-VITERBI(P, A, B)
1. for j ← 2 to t do
2. par for i ← 1 to n do
3. for k ← 1 to n do
4.     P[i, j] ← max (P[i, j],
       P[k, j − 1] × A[k, i] × B[i, y_j])
```

```
VITERBI-D&C(P, A, B)
1. for j ← 2 to t do
2.     X ← P[.., j]; U ← P[.., j − 1]
3.     V ← A; W ← B[.., y_j]
4.     A_vit(X, U, V, W)
```

```
A_vit(X, U, V, W)
1. if V is a small matrix then
2.     A_loop−vit(X, U, V, W)
3. else
4.     par: A_vit(X_1, U_1, V_11, W_1),
           A_vit(X_2, U_1, V_12, W_2)
5.     par: A_vit(X_1, U_2, V_21, W_1),
           A_vit(X_2, U_2, V_22, W_2)
```

Fig. 1. Iterative and recursive Viterbi algorithms.

Cache-inefficient algorithm. An iterative parallel and a recursive divide-and-conquer-based parallel Viterbi algorithms are given in Figure 1. As per the Viterbi recurrence, each cell (i, j) of matrix P depends on all cells of P in column $j - 1$, all cells of A in column i, and the cell (i, y_j) of B. The function $\mathcal{A}_{vit}$ fills jth column of P denoted by X using $(j - 1)$th column denoted by U using a divide-and-conquer approach. To compute each column of P, the entire matrix of A has to be read. Hence the recursive algorithm is cache-inefficient. In both algorithms, the stages are computed sequentially, however, all cells in each stage (or timestep) are computed in parallel.

Complexity analysis. The serial cache complexity of the iterative algorithm is computed as $\sum_{j=1}^{t} \sum_{i=1}^{n} \mathcal{O}(n/B) = \mathcal{O}(n^2 t/B)$, and that of the divide-and-conquer algorithm is computed as follows. Let $Q_{\mathcal{A}}(n)$ denote the serial cache complexity of $\mathcal{A}_{vit}$ on a matrix of size $n \times n$. Then $Q_{\mathcal{A}}(n) = \mathcal{O}(n^2/B + n)$ if $n^2 \leq \gamma_{\mathcal{A}} M$, and $4Q_{\mathcal{A}}(n/2) + \mathcal{O}(1)$, otherwise; where, $\gamma_{\mathcal{A}}$ is a suitable constant. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}(n^2/B + n)$. Thus, the serial cache complexity of the recursive algorithm is $\mathcal{O}(n^2 t/B + nt)$ when n^2 is too large to fit in cache.

Both the iterative and recursive algorithms have spatial locality, but they do not have any temporal locality. Hence, these algorithms are not cache-efficient.

The span (i.e., runtime on a machine with an unbounded number of processors) of the iterative algorithm is $\Theta(nt)$, as there are t time steps and it takes n time steps to fully update a cell of P. The span of the recursive algorithm is computed as follows. Let $T_{\mathcal{A}}(n)$ denote the span of $\mathcal{A}_{vit}$ on a matrix of size $n \times n$. Then $T_{\mathcal{A}}(n) = \Theta(1)$ if $n = 1$, and $2T_{\mathcal{A}}(n/2) + \Theta(1)$, otherwise. Solving, $T_{\mathcal{A}}(n) = \Theta(n)$, which implies that the span of the recursive algorithm is $\Theta(nt)$.

3 Cache-efficient multi-instance Viterbi

In this section, we present a novel cache-efficient cache-oblivious Viterbi algorithm for multiple instances of the problem.

It is easy to see that a standard recursive divide-and-conquer algorithm has no temporal locality because to compute each column of P ($\Theta(n^2)$ work), we have to scan the entire matrix A ($\Theta(n^2)$ space). We can exploit temporal cache locality by solving multiple instances of the problem simultaneously. The existing method that uses multiple instances [25] is cache-inefficient.

Two problems that have the same transition matrix A and emission matrix B are termed two instances of the same problem. The spoken word recognition problem can be considered as an example of multi-instance Viterbi problem. The core idea of the algorithm comes from the fact that by scanning the transition matrix A only once, a particular column of matrix P can be computed for n instances of the problem.

```
VITERBI-MULTI-INSTANCE-D&C(P_1, P_2, ..., P_q, A, B, t)
1.  for j ← 2 to t do
2.      X ← [P_1[.., j], P_2[.., j], ..., P_q[.., j]]
3.      U ← [P_1[.., j − 1], P_2[.., j − 1], ..., P_q[.., j − 1]]
4.      V ← A
5.      W ← [B[.., y_1j], B[.., y_2j], ..., B[.., y_qj]]
6.      A_vit(X, U, V, W)

A_vit(X_{n×q}, U_{n×q}, V_{n×n}, W_{n×q})
1.  if X and V are small matrices then
2.      A_loop−vit(X, U, V, W)
3.  else if q > n do
4.      par: A_vit(X_L, U_L, V, W_L), A_vit(X_R, U_R, V, W_R)
5.  else if q < n do
6.      par: A_vit(X_T, U_T, V_11, W_T), A_vit(X_B, U_T, V_12, W_B)
7.      par: A_vit(X_T, U_B, V_21, W_T), A_vit(X_B, U_B, V_22, W_B)
8.  else
9.      par: A_vit(X_11, U_11, V_11, W_11), A_vit(X_12, U_12, V_11, W_12),
             A_vit(X_21, U_11, V_12, W_21), A_vit(X_22, U_12, V_12, W_22)
10.     par: A_vit(X_11, U_21, V_21, W_11), A_vit(X_12, U_22, V_21, W_12),
             A_vit(X_21, U_21, V_22, W_21), A_vit(X_22, U_22, V_22, W_22)
```

Fig. 2. Cache-efficient multi-instance Viterbi algorithm.

Consider Figure 2. In the function $\mathcal{A}_{vit}(X, U, V, W)$, the matrix U is an $n \times q$ matrix obtained by concatenating $(j-1)$th columns of q matrices $P_1, P_2, \ldots, P_q$, where P_i is the most likely path probability matrix of problem instance i. The algorithm computes X, which is a concatenation of jth columns of the q problem instances. Each problem instance i has a different observations vector $Y_i = \{y_{i1}, y_{i2}, \ldots, y_{it}\}$. Matrix W W is a concatenation of $B[y_{1,j}], B[y_{2,j}], \ldots, B[y_{q,j}]$. We use X_T, X_B, X_L, and X_R to represent the top half, bottom half, left half, and right half of X, respectively. Executing the divide-and-conquer algorithm once computes the second column of all matrices P_1 to P_q. Executing the algorithm again computes the third column of the q matrices. Executing the algorithm t times, the last column of all problem instances would be filled. Note that for each time step (or observation step), W needs to be reconstructed.

Complexity analysis. The serial cache complexity of the algorithm in Figure 2 is computed as follows. Let $Q_{\mathcal{A}}(n,q)$ denote the serial cache complexity of $\mathcal{A}_{vit}$ on a matrix of size $n \times q$, and let n and q be powers of two. Then $Q_{\mathcal{A}}(n,q) = \mathcal{O}\left(n^2/B + n\right)$ when $n^2 + nq \leq \gamma_{\mathcal{A}} M$; $Q_{\mathcal{A}}(n,q) = 8Q_{\mathcal{A}}\left(n/2, q/2\right) + \mathcal{O}\left(1\right)$ when $n = q$; $Q_{\mathcal{A}}(n,q) = 2Q_{\mathcal{A}}\left(n, q/2\right) + \mathcal{O}\left(1\right)$ when $n < q$; and $Q_{\mathcal{A}}(n,q) = 4Q_{\mathcal{A}}\left(n/2, q\right) + \mathcal{O}\left(1\right)$ when $n > q$; where, $\gamma_{\mathcal{A}}$ is a suitable constant. Solving, the cache complexity of the algorithm for t timesteps is $t \times Q_{\mathcal{A}}(n,q) = \mathcal{O}\left(n^2qt/(B\sqrt{M}) + n^2qt/M + n(n+q)t/B + t\right)$. As the algorithm exploits temporal locality, it is cache-efficient. The span of the algorithm remains $\Theta\left(nt\right)$.

4 Viterbi algorithm using rank convergence

We briefly describe and improve Maleki et al.'s Viterbi algorithm [26] below.

Preliminaries. We rewrite the Viterbi recurrence using *log-probabilities* (i.e., logarithms of all probabilities) as follows so that we can replace multiplications with additions: $P[i,j] = I[i] + B[i,y_1]$ if $j = 1$, and $P[i,j] = \max_{k\in[1,n]}(P[k,j-1] + A[k,i] + B[i,y_j])$ if $j > 1$.

We rewrite the recurrence above as $s[t-1] = s[0] \odot A_1 \odot A_2 \odot \cdots \odot A_{t-1}$, where $s[j]$ is the jth solution vector (or column vector $P[..,j]$) of matrix P, the $n \times n$ matrix A_i is a suitable combination of A and B, and $\odot$ is a matrix product operation defined between two matrices $R_{n\times n}$ and $S_{n\times n}$ as $(R \odot S)[i,j] = \max_{k\in[1,n]}(R[i,k] + S[k,j])$.

The *rank* of a matrix $A_{m\times n}$ is r if r is the smallest number such that A can be written as a product of two matrices $C_{m\times r}$ and $R_{r\times n}$. Vectors v_1 and v_2 are *parallel* provided they differ by a constant offset. For example, $\langle 1,2,3,4\rangle$ and $\langle 5,6,7,8\rangle$ are two parallel vectors.

```
VITERBI-RANK(s[0..t − 1], A, B)
1.  p ← #processors
    ⟨ Forward phase ⟩
2.  par for i ← 1 to p do
3.    l_i ← t(i − 1)/p; r_i ← ti/p
4.    if i > 1 then s[l_i] ← random vector
5.    for j ← l_i to r_i − 1 do
6.      s[j + 1] ← VITERBI(s[j], A, B[.., y_{j+1}])
    ⟨ Fix up phase ⟩
7.  converged ← false
8.  while !converged do
9.    par for i ← 2 to p do
10.     conv_i ← false; s ← s[l_i]
11.     for j ← l_i to r_i − 1 do
12.       s ← VITERBI(s, A, B[.., y_{j+1}])
13.       if s is parallel to s[j + 1] then
14.         conv_i ← true; break
15.       s[j + 1] ← s
16.   converged ← ∧_i conv_i
```

Fig. 3. Processor-aware parallel Viterbi algorithm using rank convergence as given in Maleki et al. paper [26].

Original algorithm. The algorithm, shown in Figure 3, consists of two phases: (i) parallel forward phase, and (ii) fix up phase. In the forward phase, the t stages are divided into p segments, where p is the number of processors, each segment having $\lceil t/p \rceil$ stages (except possibly the last stage). The stages in the ith segment are from l_i to r_i. The initial solution vector of the entire problem is the initial vector of the first segment and it is known. The initial solution vectors of all other segments are initialized to non-zero random values. A sequential Viterbi algorithm is run in

all the segments in parallel. A stage i is said to converge if the computed solution vector $s[i]$ is parallel to the actual solution vector s_i. A segment i is said to converge if $\text{rank}(A_{l_i} \odot A_{l_i+1} \odot \cdots \odot A_j)$ is 1 for $j \in [l_i, r_i - 1]$.

In the fix up phase a sequential Viterbi algorithm is executed for all segments simultaneously. The solution vectors computed in different segments (except the first) might be wrong. But eventually they will become parallel to the actual solution vectors if *rank convergence* occurs. If rank convergence occurs at every segment then the solution vectors at every stage will be parallel to the actual solution vectors. Otherwise, the fix up phase is run again and again until rank convergence occurs at some point. In the worst case, which rarely happens in practice, the fix up phase will have to be executed a total of $p-1$ times for rank converngence to happen.

Improved algorithm. The algorithm described above is processor-aware, and we make it processor-oblivious as follows.

We chose a suitable segment size c (say 256) that is feasibly large, then use a parallel for loop to solve those t/c segments simultaneously. Unlike Maleki et al.'s algorithm, we need to make sure that the segments are non-overlapping at their boundaries and then adjust the fixup phase accordingly as shown in Figure 4.

```
VITERBI-RANK-IMPROVED(s[0..t − 1], A, B)
1.  n ← 2^k; t ← 2^{k+k'}; c ← 2^8
    ⟨ Forward phase ⟩
2.  size ← c; q ← t/size
3.  par for i ← 0 to q − 1 do
4.    l_i ← i × size, r_i ← l_i + size − 1
5.    if i > 0 then s[l_i] ← random vector
6.    for j ← l_i to r_i do
7.      s[j + 1] ← VITERBI(s[j], A, B[.., y_{j+1}])
    ⟨ Fixup phase ⟩
8.  u[0..t − 1] ← s[0..t − 1]; converged ← false
9.  for (j ← log c to (log t)-1) & !converged do
10.   size ← 2^j; q ← t/(2 × size)
11.   par for i ← 0 to q − 1 do
12.     l_i ← (2i + 1) × size − 1
13.     r_i ← l_i + size; conv_i ← false
14.     for j ← l_i to r_i do
15.       u[j+1] ← VITERBI(u[j], A, B[.., y_{j+1}])
16.       if u[j + 1] is parallel to s[j + 1] then
17.         conv_i ← true; break
18.       s[j + 1] ← u[j + 1]
19.   for i ← 0 to q − 1 do
20.     converged ← converged ∧ conv_i
21.   if converged = true then break
```

Fig. 4. Processor-oblivious parallel Viterbi algorithm using rank convergence.

Here is how the algorithm works. Let the initial segment size is c (i.e., c consecutive time steps). For convenience we chose $c = 2^i$ where $i \in [\log c, \log t]$. We divide t time steps into t/c independent segments each of size c. Similar to Maleki et. al.'s algorithm, the first solution vectors of all except the first segment are initialized to non-zero valid random probability values. Then in the forward phase we run serial Viterbi algorithm on all segments simultaneously. At the end of the forward phase solution vectors till the c^{th} column (i.e., all columns in the first segment) will have correct log-likelihood values. Other segments will have values computed from the random values chosen initially which may or may not be parallel to the expected values.

In the fix up phase, we start fixing from the second segment as in the original Maleki et al.'s algorithm. However, in each fix up phase, we work on alternative segments always leaving the first segment of the prior fix up phase. After each fix up phase, the size of each segment being considered doubles and number of segments becomes half with respect to the previous phase. At the end of each

fix up phase, we check whether the computed solution vectors are parallel to those in the forward phase, and if the answer is 'yes' for all segments under consideration, the program terminates. Otherwise, the next fix up phase is run. In the worst case, the fix up phase is executed $\lambda \in [1, \log(t/c)]$ times after which all results are guaranteed to be correct since by that time the result from the original input propagates till the end. In the worst case, the program is like a serial Viterbi algorithm with a constant factor overhead.

Complexity analysis. Let $T_1^F(n,t)$, $Q_1^F(n,t)$, $T_\infty^F(n,t)$, and $S^F(t)$ denote the work, serial cache complexity, span, and the steps for convergence, respectively, of algorithm $F \in \{O, I\}$, where O represents the original rank convergence algorithm and I denotes our modified algorithm. Let $f(t)$ be the number of segments in algorithm O. Note that for Maleki et al.'s original algorithm $f(t) = p$. Let the number of times the fix up phase is executed in O and I be λ_O and λ_I, respectively. Then $\lambda^O \in [1, f(t)]$ and $\lambda^I \in [1, \log(t/c)]$.

Work. $T_1^O(n,t) = \Theta\left(n^2 t \cdot \lambda_O\right)$, and $T_1^I(n,t) = \Theta\left(n^2 t \cdot \lambda_I\right)$. In the worst case, $T_1^O(n,t)$ is $\Theta\left(n^2 t \cdot f(t)\right)$, and $T_\infty^I(n,t)$ is $\Theta\left(n^2 t \cdot \log t\right)$.

Serial cache complexity. As there is no temporal locality, $Q_1^O(n,t) = \mathcal{O}\left(T_1^O(n,t)/B\right)$ and $Q_1^I(n,t) = \mathcal{O}\left(T_1^I(n,t)/B\right)$, when n^2 does not fit into the cache.

Span. $T_\infty^O(n,t) = \Theta\left(n(t/f(t)) \cdot \lambda_O\right)$, as the number of stages in each segment is $\Theta\left(t/f(t)\right)$, and the span of executing each stage is $\Theta\left(n\right)$. In the worst case, $T_\infty^O(n,t)$ is $\Theta\left(nt\right)$. $T_\infty^I(n,t)$ is computed as follows. In the ith fix up phase, the number of stages in each segment is 2^i. Hence, the span of executing all stages for λ_I iterations in the fix up phase is $\Theta\left(\sum_{i=\log c}^{(\log c)+\lambda_I} 2^i\right) = \Theta\left(2^{\lambda_I}\right)$. Then $T_\infty^I(n,t) = \Theta\left(n2^{\lambda_I}\right)$. In the worst case, $T_\infty^I(n,t)$ is $\Theta\left(nt\right)$.

Steps for convergence. Let the rank of the matrix $A_1 \odot A_2 \odot \cdots \odot A_t$ be k. For the original algorithm, $(S^O(t) - 1) \times (t/f(t)) < k \leq S^O(t) \times (t/f(t))$, which implies $S^O(t) = \lceil kf(t)/t \rceil$. Similarly, for the improved algorithm, $2^{S^I(t)-1+\log c} < k \leq 2^{S^I(t)+\log c}$, which implies $S^I(t) = \lceil k/c \rceil$.

5 Cache-efficient Viterbi algorithm

In this section, we present an efficient cache- and processor-oblivious parallel Viterbi algorithm based on recursive divide-and-conquer, as shown in Figure 5. The algorithm is derived by combining ideas from the cache-efficient multi-instance Viterbi algorithm (see Section 3) and the improved parallel Viterbi algorithm based on rank convergence (see Section 4).

Recall that in the multi-instance Viterbi algorithm works on the i^{th} solution vectors, $s[i]$, of different instances of the problem and generates the $(i+1)^{th}$ solution vectors, $s[i+1]$, of the instances cache-efficiently. To develop a cache-efficient Viterbi algorithm, in the forward phase, we divide t time steps into t/c independent segments each of size c as we did in the improved parallel Viterbi algorithm using rank convergence shown in Figure 4, As before, we chose $c = 2^i$

where $i \in [\log c, \log t]$. Since each segment is independent, we can assume that these segments are different instances of the same Viterbi problem. Therefore, we can use the cache-efficient multi-instance Viterbi algorithm to solve these t/c instances simultaneoulsy. Again, the first solution vectors of all except the first segment are initialized to non-zero valid random probability values.

The fix up phase is similar to that of the VITERBI-RANK-IMPROVED algorithm (see Figure 5 and 4), except that now we use cache-efficient Multi instance Viterbi algorithms to compute the next solution vector of all segments at once instead of using Viterbi algorithm to compute an entire segment independently. As before, we start fixing from the second segment since the first segment is already fixed after the forward phase.

In each fix up phase, we work on alternative segments always leaving out the first segment of the prior fix up phase (already fixed by this time). After each fix up phase, the size of each segment being considered doubles and number of segments halves with respect to the previous phase. For each step, we use multi-instance Viterbi algorithm to compute the $(i+1)$st solution vector from the ith solution vectors for all segments at once. At the end of each fix up phase, we check whether the computed solution vectors are parallel to those found in the forward phase, and if that is true for all segments under consideration, the program terminates. Otherwise, the next fix up phase is run. In the worse case, the fix up phase is executed for $\lambda \in [1, \log(t/c)]$ times after which all results are guaranteed to be correct.

```
VITERBI-CACHE-EFFICIENT(s[0..t − 1], A, B)
 1. n ← 2^k; t ← 2^{k+k'}; c ← 2^8
    ⟨ Forward phase ⟩ ____________________
 2. size ← c; q ← t/size
 3. par for i ← 0 to q − 1 do
 4.    l_i ← i × size, r_i ← l_i + size − 1
 5.    if i > 0 then s[l_i] ← random vector
 6. VITERBI-MI(s[l_0..r_0], s[l_1..r_1], ..,
       s[l_{q-1}..r_{q-1}], A, B, c)
    ⟨ Fixup phase ⟩ ____________________
 7. u[0..t−1] ← s[0..t−1]; converged ← false
 8. for (j ← log c to (log t) − 1) do
 9.    size ← 2^j, q ← t/(2 × size)
10.    par for i ← 0 to q − 1 do
11.       l_i ← (2i + 1) × size − 1
12.       r_i ← l_i + size; conv_i ← false
13.    VITERBI-MI(u[l_0..r_0], ..., u[l_{q-1}..r_{q-1}],
          A, B, size + 1)
14.    par for i ← 0 to q − 1 do
15.       r_i ← 2(i + 1) × size − 1
16.       if u[r_i] is parallel to s[r_i] then
17.          conv_i ← true
18.       else s[r_i] ← u[r_i]
19.    for i ← 0 to q − 1 do
20.       converged ← converged ∧ conv_i
```

Fig. 5. An efficient cache- and processor-oblivious parallel Viterbi algorithm using rank convergence. VITERBI-MI refers to VITERBI-MULTI-INSTANCE-D&C of Sec. 3.

Complexity analysis. Let $T_1(n,t), Q_1(n,t)$, and $T_\infty(n,t)$ be the work, serial cache complexity, and span of the cache-efficient Viterbi algorithm, respectively. Let $\lambda \in [1, \log(t/c)]$ be the number of times the fix up phase is executed.

$T_1(n,t) = \Theta\left(n^2 t \cdot \lambda\right)$. In the worst case, $T_1(n,t) = \Theta\left(n^2 t \cdot \log t\right)$. As in Section 4, $T_\infty(n,t) = \Theta\left(n2^\lambda\right)$. Finally, $Q_1(n,t) = \mathcal{O}\left(\sum_{i=\log c}^{(\log c)+\lambda} \left(Q_{\mathcal{A}}\left(n, t/2^i\right) \cdot 2^i\right)\right)$ $= \mathcal{O}\left(n^2 t\lambda/(B\sqrt{M} + n^2 t\lambda/M + (n(n2^\lambda + t\lambda))/B + 2^\lambda\right)$. If $n^2, t = \Omega\left(\sqrt{M}\right)$ and convergence happens after $\lambda = \mathcal{O}(1)$ iterations of the fix up phase, $Q_1(n,t)$ reduces to $\mathcal{O}\left(n^2 t\lambda/(B\sqrt{M}) + n^2 t\lambda/M\right)$ which further reduces to $\mathcal{O}\left(n^2 t\lambda/(B\sqrt{M})\right)$ when the cache is tall (i.e., $M = \Omega\left(B^2\right)$).

6 Experimental results

This section presents our implementation details and performance results.

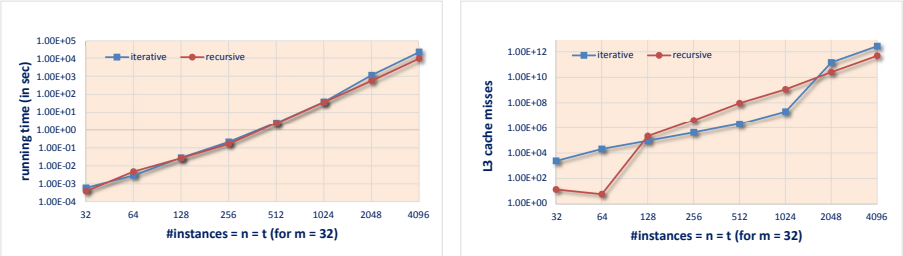

Fig. 6. Running time and L3 miss of our cache-efficient multi-instance Viterbi algorithm along with the multi-instance iterative Viterbi algorithm.

We used a dual socket 16-core (= 2 × 8-cores) 2 GHz Intel Sandy Bridge machine to run all experiments presented in the paper. Each core of this machine was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All the cores in a socket shared a 20 MB L3 cache, and the machine had 32 GB RAM shared by all cores. We used PAPI 5.2 [1] to count the L3 cache misses (event `PAPI_L3_TCM`) and `likwid` [35] (i.e., `likwid-perfctr`) to measure energy and power consumption of the program. The matrices A, B, and I were initialized to random probabilities. We used log-probabilities in all implementations and hence used additions instead of multiplications in the Viterbi recurrence. All matrices were stored in column-major order. We performed two sets of experiments to compare our cache-efficient algorithms with the iterative and the fastest known Viterbi (Maleki et al.'s) algorithms. They are as follows.

Cache-efficient multi-instance Viterbi algorithm. We compared our cache-efficient multi-instance recursive Viterbi algorithm with the multi-instance iterative Viterbi algorithm. Both algorithms were optimized and parallelized. To construct matrix $W_{n\times q}$ (we chose q to be n in this case), instead of copying all the relevant columns of B, only the pointers to the respective columns were used. Wherever possible, pointer swapping was used to interchange previous solution vector (or matrix) and current solution vector (or matrix).

The running time and the L3 cache misses for the two algorithms are plotted in Figure 6. The number of stages n, which is also the number of instances was varied from 32 to 4096. Note that in the cache-efficient multi-instance Viterbi algorithm, the number of stages does not need to be the same as the number of instances. The variable m was fixed to 32 and the number of timesteps t was also kept the same as n (hence overall complexity is $O(n^4)$). The recursive algorithm ran slightly faster than the iterative algorithm in most cases when the number of instances increased. When n was 4096, our recursive algorithm ran around 2.26 times faster than the iterative algorithm.

Cache-efficient Viterbi algorithm. We compared our cache-efficient parallel Viterbi algorithm with Maleki et al.'s parallel Viterbi algorithm. Both implementations were optimized and parallelized and the reported statistics are averages

of 4 independent runs. In all our experiments, the number of processors p was set to 16. The plots of Figure 7 show the graphs of the running time and L3 cache misses for the two algorithms for $n = 4096$.

When $n = 4096$, we varied t from 2^{12} to 2^{18}, and kept m fixed at 32. Our algorithm ran faster and incurred significantly fewer L3 misses than Maleki et al.'s algorithm throughout. For $t = 2^{18}$, our algorithm ran 33% faster, and incurred a factor of 6 fewer L3 misses. Better cache performance led to lower DRAM energy consumption.

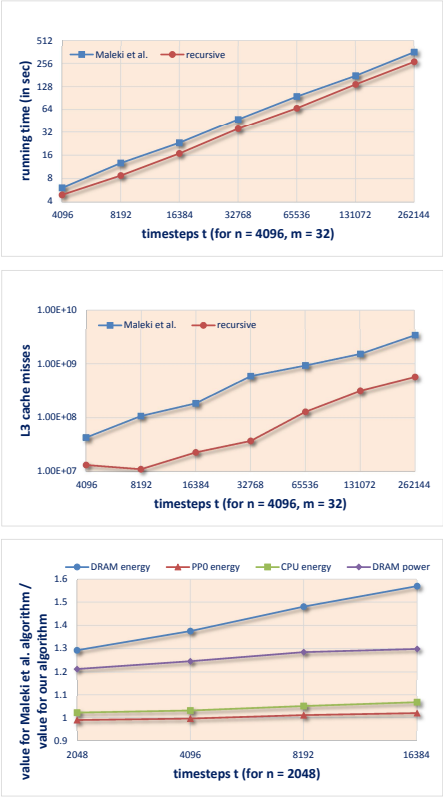

Fig. 7. Running time, L3 miss and energy/power consumption of our cache-efficient Viterbi algorithm along with the existing algorithms.

Energy consumption. We ran experiments to analyze the energy consumption (taking average over three runs) of our cache-efficient recursive algorithm and Maleki et. al.'s algorithm. Our algorithm consumed relatively less DRAM energy compared to the other algorithm.

We used the `likwid-perfctr` tool to measure CPU, Power Plane 0 (PP0), DRAM energy, and DRAM power consumption during the execution of the programs. The energy measurements were end-to-end, i.e., included all costs during the entire program execution. Note that the DRAM energy consumption is somewhat related to the L3 cache miss of a program as each L3 cache miss results in a DRAM access. Similarly, since CPU energy gives the energy consumed by the entire package (all cores, on chip caches, registers and their interconnections), it is related to a program's running time. PP0 is basically a subset of CPU energy since it captures energy consumed by only the cores and their private caches.

For $n = 2048$, t was increased from 2^{11} to 2^{14} while keeping m fixed to 32. Figure 7 shows that the DRAM energy as well as power consumption of our algorithm was significantly less because of the reduced L3 cache misses. When $t = 16384$, Maleki et al.'s algorithm consumed 60% more DRAM energy and 30% more DRAM power than ours.

Acknowledgment. Chowdhury and Ganapathi were supported in part by NSF grants CCF-1162196, CCF-1439084 and CNS-1553510.

References

1. Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/`
2. Bille, P., Stöckel, M.: Fast and cache-oblivious dynamic programming with local dependencies. In: proc. LATA. pp. 131–142 (2012)
3. Burge, C., Karlin, S.: Prediction of complete gene structures in human genomic DNA. Journal of Molecular Biology pp. 268(1):78–94 (1997)
4. Cherng, C., Ladner, R.E.: Cache efficient simple dynamic programming. In: proc. AofA. pp. 49–58 (2005)
5. Chin, W., Tan, S., Teo, Y.: Deriving efficient parallel programs for complex recurrences. In: proc. PASCO. pp. 101–110 (1997)
6. Chin, W.N., Darlington, J., Guo, Y.: Parallelizing conditional recurrences. In: proc. Euro-Par. pp. 579–586 (1996)
7. Chowdhury, R.A., Ganapathi, P., Tithi, J.J., Bachmeier, C., Kuszmaul, B.C., Leiserson, C.E., Solar-Lezama, A., Tang, Y.: AutoGen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In: proc. PPoPP. p. 10. ACM (2016)
8. Chowdhury, R.A.: Cache-efficient algorithms and data structures: theory and experimental evaluation. Ph.D. thesis, PhD thesis, Department of Computer Sciences, The University opf Texas at Austin (2007)
9. Chowdhury, R.A., Ramachandran, V.: Cache-oblivious dynamic programming. In: proc. SODA. pp. 591–600 (2006)
10. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: proc. SPAA. pp. 207–216 (2008)
11. Chowdhury, R.A., Ramachandran, V.: The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. Theory of Computing Systems 47(4), 878–919 (2010)
12. Chowdhury, R.A., Ramachandran, V., Silvestri, F., Blakeley, B.: Oblivious algorithms for multicores and networks of processors. Journal of Parallel and Distributed Computing 73(7), 911–925 (2013)
13. Chowdhury, R.A., Le, H.S., Ramachandran, V.: Cache-oblivious dynamic programming for bioinformatics. IEEE/ACM Transactions on Computational Biology and Bioinformatics 7(3), 495–510 (2010)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press Cambridge (2001)
15. Costello, D.J., Hagenauer, J., Imai, H., Wicker, S.B.: Applications of error-control coding. IEEE Transactions on Information Theory 44(6), 2531–2560 (1998)
16. Cutting, D., Kupiec, J., Pedersen, J., Sibun, P.: A practical part-of-speech tagger. In: proc. ANLC. pp. 133–140. Association for Computational Linguistics (1992)
17. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge Univ. Press (1998)
18. Ferreira, M., Roma, N., Russo, L.M.: Cache-oblivious parallel SIMD Viterbi decoding for sequence search in HMMER. Bioinformatics 15(1), 165 (2014)
19. Fisher, A.L., Ghuloum, A.M.: Parallelizing complex scans and reductions 29(6), 135–146 (1994)
20. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: proc. FOCS. pp. 285–297 (1999)
21. Heller, J., Jacobs, I.: Viterbi decoding for satellite and space communication. IEEE Transactions on Communication Technology 19(5), 835–848 (1971)

22. Klein, D., Manning, C.D.: A* parsing: fast exact Viterbi parse selection. In: proc. NAACL. pp. 40–47 (2003)
23. Kobayashi, H.: Application of probabilistic decoding to digital magnetic recording systems. IBM Journal of Research and Development 15(1), 64–74 (1971)
24. Krogh, A., Larsson, B., Von Heijne, G., Sonnhammer, E.L.: Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. Journal of Molecular Biology 305(3), 305(3):567–580 (2001)
25. Liu, C.: cuHMM: A CUDA implementation of hidden Markov model training and classification. The Chronicle of Higher Education (2009)
26. Maleki, S., Musuvathi, M., Mytkowicz, T.: Parallelizing dynamic programming through rank convergence. In: proc. PPoPP. pp. 219–232 (2014)
27. Maleki, S., Musuvathi, M., Mytkowicz, T.: Low-rank methods for parallelizing dynamic programming algorithms. ACM Trans. on Parallel Comp. 2(4), 26 (2016)
28. Nam, H., Kwak, H.: Viterbi decoder for a high definition television (1998), `http://www.google.com/patents/US5844945`, US Patent 5,844,945
29. Ohler, U., Niemann, H., Liao, G.c., Rubin, G.M.: Joint modeling of DNA sequence and physical properties to improve eukaryotic promoter recognition. Bioinformatics 17(suppl 1), S199–S206 (2001)
30. Siepel, A., Bejerano, G., Pedersen, J.S., Hinrichs, A.S., Hou, M., Rosenbloom, K., Clawson, H., Spieth, J., Hillier, L.W., Richards, S., et al.: Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes. Genome Research 15(8), 1034–1050 (2005)
31. Tan, G., Feng, S., Sun, N.: Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In: proc. SC. p. 78 (2006)
32. Tang, S., Yu, C., Sun, J., Lee, B.S., Zhang, T., Xu, Z., Wu, H.: EasyPDP: An efficient parallel dynamic programming runtime system for computational biology. IEEE Transactions on Parallel and Distributed Systems 23(5), 862–872 (2012)
33. Tang, Y., Chowdhury, R.A., Luk, C.K., Leiserson, C.E.: Coding stencil computations using the Pochoir stencil-specification language. In: proc. HotPar (2011)
34. Tithi, J.J., Ganapathi, P., Talati, A., Aggarwal, S., Chowdhury, R.A.: High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In: proc. IPDPS (2015)
35. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: proc. ICPPW. pp. 207–216 (2010)
36. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. on Information Theory 13(2), 260–269 (1967)
37. Viterbi, A.J.: Convolutional codes and their performance in communication systems. IEEE Transactions on Communication Technology 19(5), 751–772 (1971)