CSCI 1760 - Final Project Report
A Parallel Implementation of Viterbi's Decoding Algorithm

Shay Mozes

Brown University
shay@cs.brown.edu

Abstract. This report describes parallel Java implementations of several variants of Viterbi's algorithm, discussed in my recent paper [1]. The aim of this project is to study the issues that arise when trying to implement the approach of [1] in parallel using Java. I compare and discuss the performance of several variants under various circumstances.

1 Preliminaries

A comprehensive background and presentation of Viterbi's algorithm and the use of repetitions and compression for speeding up its running time are given in the attached [1]. To keep things concise, I repeat here just the necessary minimum.

Let Σ denote a finite alphabet and let $X \in \Sigma^n$, $X = x_1, x_2, \ldots, x_n$ be a sequence of observed letters. A Markov *model* is a set of k states, along with emission probabilities $e_k(\sigma)$ - the probability to observe $\sigma \in \Sigma$ given that the state is k, and transition probabilities $P_{i,j}$ - the probability to make a transition to state i from state j.

The Viterbi algorithm (VA) finds the most probable sequence of hidden states given the model and the observed sequence, i.e., the sequence of states $s_1, s_2, \ldots, s_n$ which maximize

$$\prod_{i=1}^{n} e_{s_i}(x_i) P_{s_i, s_{i-1}} \tag{1}$$

The dynamic program of VA calculates a vector $v_t[i]$ which is the probability of the most probable sequence of states emitting $x_1, \ldots, x_t$ and ending with the state i at time t. v_0 is usually taken to be the vector of uniform probabilities (i.e., $v_0[i] = \frac{1}{k}$). v_{t+1} is calculated from v_t according to

$$v_{t+1}[i] = e_i(x_{t+1}) \cdot \max_j \{P_{i,j} \cdot v_t[j]\} \tag{2}$$

We call the computation of v_{t+1} from v_t a Viterbi step. Clearly, each Viterbi step requires $O(k^2)$ time. Therefore, the total runtime required to compute the vector v_n is $O(nk^2)$. The probability of the most likely sequence of states is the maximal element in v_n. The actual sequence of states can be reconstructed in linear time if we keep track, along the computation, of the maximizing arguments in (2).

It is useful for our purposes to rewrite VA in a slightly different way. Let M^σ be a $k \times k$ matrix with elements $M^\sigma_{i,j} = e_i(\sigma) \cdot P_{i,j}$. We can now express v_n as:

$$v_n = M^{x_n} \odot M^{x_{n-1}} \odot \cdots \odot M^{x_2} \odot M^{x_1} \odot v_0 \tag{3}$$

where $(A \odot B)_{i,j} = \max_k \{A_{i,k} \cdot B_{k,j}\}$ is the so called max-times matrix multiplication. VA calculates (3) from right to left using matrix-vector multiplication in $O(nk^2)$ time. However, the operations in (3) are associative and can be performed in any order, albeit with longer running time.

In [1] we showed how to use repetitions in the input sequence X to obtain an asymptotic speedup. The simplest idea is to precalculate a matrix $M(W)$ for each possible sequence $W = w_1, w_2, \ldots, w_\ell$ of ℓ characters:

$$M(W) = M^{w_\ell} \odot M^{w_{\ell-1}} \odot \cdots \odot M^{w_2} \odot M^{w_1} \tag{4}$$

Here ℓ is some small integer ($\ell = O(\log n)$). We then compress X into the corresponding sequence of n/ℓ characters of a larger alphabet, by grouping each ℓ characters together. I refer to this idea as compressed Viterbi. Note that appropriate changes must be made in order to be able to trace back the optimal sequence of states in this case. For details on this and on other improvements on Viterbi's algorithm, see [1]).

2 Parallel implementation

Throughout this report I shall use two different terms to describe work which can be done in parallel. The first is *parallelization in the state space* (or parallelization in k), which refers to the computation in parallel of a single matrix vector multiplication (e.g., the calculation of $v_{t+1} = M^{x_t} \odot v_t$ in parallel). The second is *parallelization in the sequence space* (or parallelization in n), which refers to the computation in parallel of the product of several matrices. In VA, the dependency $v_{t+1} = M^{x_t} \odot v_t$ dictates that only a single matrix-vector multiplication be performed in parallel. Hence, for VA, only parallelization in k is applicable. If we choose, however, to calculate (3) in some other order, then parallelization in the sequence space is possible as well.

The following variants were implemented. All variants take as input a set of $|\Sigma|$ k by k matrices, an input sequence of length n and an initial vector with k elements. The output is the optimal sequence of $n+1$ states. It is assumed that the entries of the matrices are the logarithms of probabilities, so that numerical accuracy is not an issue. This means that instead of max-times multiplication we actually perform max-plus multiplication.

- o - Sequential Viterbi's algorithm. One thread performs $O(nk^2)$ work using $O(nk)$ space.
- p - Sequential compressed Viterbi's algorithm. The length of the precalculated matrices ℓ is also given as input. One thread performs $O(|\Sigma|^\ell k^3 + \frac{nk^2}{\ell})$ work using $O(|\Sigma|^\ell k^2 + \frac{nk}{\ell})$ space.
- z - Same as the p-variant except that the pre-calculation and traceback are done in parallel.
- v - Parallel Viterbi's algorithm using matrix-vector multiplication (parallel only in k). Each matrix vector multiplication $u = M \odot v$ is divided into p_k blocks, each of length $bs = k/p_k$. Thread i computes bs consecutive elements of u. The maximal possible speedup with respect to the p-variant is p_k.
- m - Parallel Viterbi's algorithm using matrix-matrix multiplication (parallel both in k and in n). Consists of three steps:
  1. Divides the input sequence into p_n block tasks, each of length n/p_n. Each task consists of calculating the product and traceback information of n/p_n consecutive matrices (this is the parallelization in n part). p_k threads are assigned to perform matrix-matrix multiplications within each block (parallelization in k), so concurrently we have $p_n \cdot p_k$ threads. The outputs of this step are p_n k by k matrices, and p_n traceback information consisting of a total of nk^2 integers. Each thread performs $O(\frac{nk^3}{p_n p_k})$ work.
  2. Runs Viterbi's algorithm on the p_n matrices from (1).
  3. Expand the output of the small Viterbi instance of (2) into the optimal sequence of states for the original problem using the traceback information from (1). This is done parallel in n using p_n threads, each performing $O(n/p_n)$ work.
- x - Parallel compressed Viterbi's algorithm. Performs the precalculation and traceback steps sequentially, and Viterbi on the compressed sequence using the parallel v-variant.
- xx - Parallel compressed Viterbi's algorithm. Performs the precalculation and traceback steps in parallel, and Viterbi on the compressed sequence using the parallel v-variant.
- y - Parallel compressed Viterbi's algorithm. Performs the precalculation and traceback steps sequentially, and Viterbi on the compressed sequence using the parallel m-variant.
- yy - Parallel compressed Viterbi's algorithm. Performs the precalculation and traceback steps in parallel, and Viterbi on the compressed sequence using the parallel m-variant.

Note that the m-variant, which is parallel in n, performs matrix-matrix multiplications, while the v-variant, which is only parallel in k performs matrix vector multiplications. This means that the m-variant performs $O(nk^3)$ work while the v-variant performs $O(nk^2)$ work. On the other hand the m-variant uses $p_n \cdot p_k$ threads, while the v-variant can only use p_k threads. We would therefore expect that in instances where k is much larger the number of processors, the v-variant can achieve maximal parallelization and would outperform the m-variant. On the other hand, if k is small, we may expect that the m-variant will outperform the v-variant. The same kind of reasoning applies to the y- and x-variants.

3 Programming

3.1 Java

I implemented all of the above variants in Java. I decided to use Java for two main reasons. First, Java is the programming language we used throughout the course. As described in class, the Java concurrency package provides convenient mechanisms for distributing tasks between threads in a pool. Second, I wanted to use this opportunity to remind myself of programming with Java, which I last did about seven years ago. I should note that implementing this project in C++ would probably result in faster running times. Searching the web, however, I found numerous postings claiming that in certain benchmarks, Java is as efficient as C [2–4]. As a very rough benchmark I wrote and compared two basic sequential VA implementations, one in C++ and the other in Java. My results show that C++ (G++ with -O3) is roughly 1.5 times faster than Java (1.5.0), but since this was not the objective of this project, I did not invest the time and effort to make those results reliable.

3.2 Environment

All runs were made on `plover.cs.brown.edu`, a Sun UltraSPARC T2 Processor with eight UltraSPARC V9 cores. Each core has 8 threads and 8/16KB data/instruction cache. Cores are interconnected via a full cross bar. L2 cache is shared among all cores and consists of eight 4MB banks. Java version is 1.5.0_07. Real clock time was measured using `System.nanoTime()` and averaged over ten repeats of each run. To take the effects of runtime optimization into consideration, each test run was repeated 3 times before beginning the time measurements.

3.3 A note on the recursive approach to matrix multiplication

One of my first attempts in distributing the work among multiple threads used the recursive approach presented in class and in the textbook. There, this approach is presented as an efficient alternative to just having a single thread computing each matrix element. The drawback of the latter approach is that there is a large overhead associated with creating short lived threads. It turns out that in practice the recursive approach is also problematic. Consider the recursion tree for the matrix addition. During execution, it is likely that threads will be created for most of the internal nodes (which do no real work) before any of the leaves actually runs. This is a waste of threads in comparison with a solution where a single thread divides the matrix into some number of blocks (like the base of the recursion, this number depends on the architecture) and submits these blocks as tasks to be carried out by other threads in the pool. This way there is only one thread generating work, instead of half of the threads generating work in the recursive approach. The same argument applies to other tasks, such as multiplying a sequence of matrices. It is tempting to apply the recursive approach, dividing into two halves at each step, but it turns out that it is far better to do it by dividing in advance into linear blocks.

4 Results

The wide use of Viterbi's algorithm in many different fields makes it interesting to test the performance of the variants for different values of input parameters, especially with respect to n and k. In computational biology, for example, n is typically very large (a few millions), while k may be as small as 2. In natural language processing, n is much smaller (a few hundreds), but k might be bigger (a few thousands). I did not have time to perform a systematic mapping of the relevant parameter space, or even come close to that. Instead I tried to find interesting instances on which to compare the performance.

4.1 Small n, Large k

For an instance with small n ($n = 20$) and large k ($k = 1000$), one would expect the m-variant to be useless. Indeed, the running times are so long that they are not reported here. On the other hand, this is the case where the parallelization in k of the v-variant should yield a speedup. The results are shown in Figure 4.1. We see that indeed, we get a speedup, but the speedup saturates for $k > 4$. Looking at the data and L2 data cache misses (not shown) we see that for more than 4 threads, the number of cache misses starts to rise and is probably the cause for the poor scalability with increasing k. Obviously, for these values of n, running the m-variant is not profitable, because we increase the running time by a factor of 1000, but can only gain a speedup of less than 20. Similarly, running the compressed variants is not beneficial, because the sequence is not long enough to contain many repetitions.

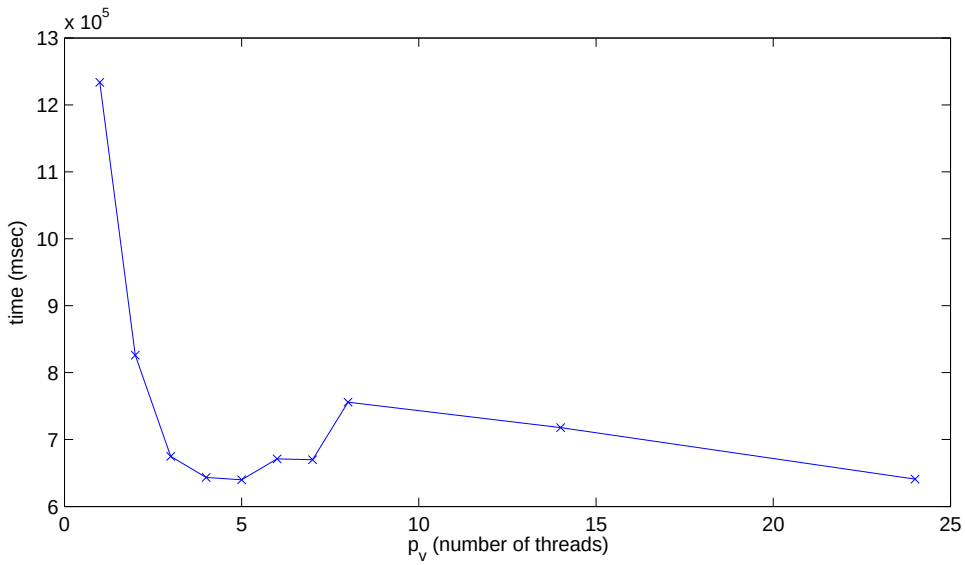

Fig. 1. Average running time for $n = 20$ and $k = 1000$ as a function of the number of threads (p_v) for the v-variant

4.2 Moderate n and k

With moderate values of k ($k = 50$) and n ($n = 5,000$), the performance of the compressed variants should theoretically be better. Figure 4.2 shows a comparison of the sequential p-variant with the parallel z- x- and xx-variants when 4 threads are used for the parallelization in k (I did check other parameters as well, but the results for 4 threads were fastest, as discussed above), for different

compression rates ℓ. First note that the compression helps (compare to the sequential running time without compression). This is an algorithmic speedup. Next we see that the p- and x- variants behave approximately the same, while the z- and xx- variants, which perform the pre-calculation in parallel do better. The xx-variant which does both the precalculation and the compressed viterbi in parallel, achieves the best overall running time for $\ell = 2$. As ℓ increases, the running is dominated by the pre-calculation step. A few runs I made showed that this step does not scale well beyond 16 concurrent threads.

	1	2	3	4	5	6
p	1363	927	1725	5474	21000	85800
z		760	801	1756	5801	22300
x		660	1495	5342	20900	85400
xx		525	631	1592	5706	22500

Fig. 2. Average running time (in milliseconds) for different compression rates ℓ (columns) and the three variants p x and xx. The sequential running time without compression is shown in the upper leftmost table entry. Sequence length is $n = 5000$ and number of states $k = 50$ ($|\Sigma| = 4$).

4.3 Large n, small k

When k is small, practically no speedup can be obtained by parallelization in k. On the other hand, since n is large, a speedup may be achieved by compression and by parallelization in n.

For $n = 10^6$, $k = 2$, the sequential algorithm takes an average of 1263 milliseconds to complete. As mentioned above, one might expect that the m-variant which runs parallel in n would outperform the sequential algorithm. However, with four threads ($p_n = 4$) the m-variant runs in 2124 milliseconds, 1728 msecs with $p_n = 14$, and around 1600 msecs for higher values of p_n (in all of these $p_k = 1$ since the matrices are small). It appears that the concurrency causes many cache misses which actually make the parallel algorithm slower than the sequential one.

The sequential compressed algorithm does yield a speedup. Although there are many cache misses in this case as well, the decrease in the amount of work does provide speedup, though not by a factor of ℓ. Here, however, we do obtain further speedup with any of the variants that run the precalculation step in parallel. The best of these is the z-variant. Figure 4.3 shows the performance of various variants for different values of ℓ ($n = 10^6$, $k = 2$). It seems that the parallel speedup is only due to the precomputation step and not because of the parallelization in n of Viterbi's algorithm itself.

4.4 Small n, small k

The results of the previous subsections suggest that parallelization is problematic due to the cache misses caused by the large amount of data used throughout the computation. It is therefore interesting to check whether parallelization does work when little data is used, for example, $n = 1000$, $k = 4$. To my knowledge, such parameters are not often used in real life situations. Also note that when the overall running time of the sequential algorithm is small, the overhead of managing multiple threads becomes significant. Indeed, in this case, the parallel m-variant is slightly faster than the sequential o-variant ($9,000$ μsec using 4 threads compared to about $12,000$ μsec sequentially). Increasing the number of threads increases the running time. It is difficult to determine whether this is due to the administration overhead, or due to memory contension and cache misses (adding counters changes the running time significantly).

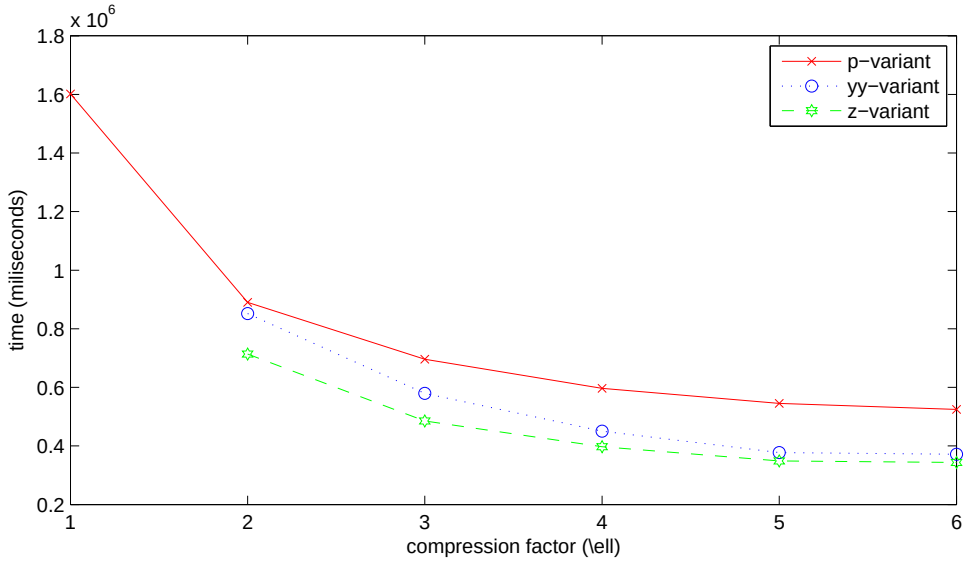

Fig. 3. Average running time for $n = 10^6$ and $k = 2$ as a function of the compression rate ℓ for the p-, yy- and z-variants. the parallel variants (yy- and z-) used 64 threads each.

5 Conclusions

Although Viterbi's algorithm is essentially a sequence of matrix vector multiplications, which in principal is an embarrassingly parallelizable task, I found it difficult to obtain any speedup in practice, and the speedups one does get do not scale well with the number of threads used. We also see that it is important to tailor the kind of parallelization employed to the specific set of input parameters.

- For small n and k one should use parallelization in n without compression (the m-variant).
- For small n and large k, parallelization in k without compression works best.
- For moderate to large n, it is best to use compression with parallelization of the precalculation step. If k is large enough then additional parallelization in k helps a little. Parallelization in n, however, does not help in this case.

It is discouraging that even for this apparently easily parallelizable problem we could not gain more than a factor of 2 speedup due to parallelization alone. This is in spite of the fact that we had 64 threads and a relatively large L2 cache at our disposal. On the positive side, we were able to combine algorithmic ideas with parallelization to gain (somewhat) better running times than we could get using either technique separately.

References

1. Yury Lifshits, Shay Mozes, Oren Weimann and Michal Ziv-Ukelson, Speeding Up HMM Decoding and Training by Exploiting Sequence Repetitions, to apear in Algorithmica (2007).
2. `http://research.sun.com/techrep/2002/abstract-114.html`
3. `http://osnews.com/story.php/5602/Nine-Language-Performance-Round-up-Benchmarking-Math-and-File-IO`
4. `http://www.idiom.com/ zilla/Computer/javaCbenchmark.html`