# Clever Title

Aaron Stacy Calvin MacKenzie

## 1. Introduction

Originally developed by Andrew Viterbi to decode signals over noisy channels, the Viterbi algorithm proved to be useful in a wide variety of areas. Within the field of Natural Language Processing (NLP), the Viterbi algorithm is most commonly used with Hidden Markov Models (HMM) to find the most likely sequence of part-of-speech tags, and other related tagging tasks.

Closely related to the Viterbi algorithm is the Forward algorithm, which is used within NLP to efficiently compute observation likelihood. While not as popular as the Viterbi algorithm, the Forward algorithm is a fundamental algorithm within NLP. The implementations of the two algorithms are actually quite similar and they share the same running time of $O(TN^2)$.

Work on improving the Viterbi algorithm typically comes from the realm of signal processing, rather than NLP, where these gains greatly help with the decoding of signals. Typically, these improvements deal with a hardware viterbi decoder and ways to modify this hardware in order to enhance the parallelization of the decoding problem.

Rather than approaching this task from a hardware perspective, we wanted to improve these algorithms with parallelization techniques. We implemented a parallel version of the Viterbi and Forward algorithms using the OpenMP API [1] and characterized the performance over a range of workloads. We tested this on the Wall Street Journal dataset from the Penn Treebank and varied the size of the dataset to measure performance. We ran the algorithm on the TACC cluster in order to find strong and weak scaling characteristics.

## 2. Related Work

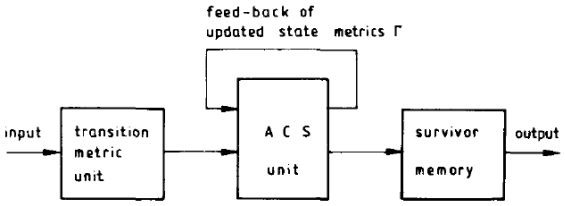


Figure 1: Pipeline structure of Viterbi decoder

The work by Fettweis and Meyr (1989) [2] describes a successful attempt at parallelizing a key bottleneck of the Viterbi algorithm: the add-compare-select component. The authors describe the process of obtaining the updated state metric, $\Gamma_{n+1,z_i}$, as the add-compare-select (ACS) unit of the Viterbi algorithm, shown in Figure 1. The formula to find this updated state is shown below (where $\lambda$ is the transition probability):

[1] `http://openmp.org/wp/`

$$\Gamma_{n+1,z_i} = \max_{\text{all possible } z_k \to z_i} (\Gamma_{n,z_k} + \lambda_{n,z_k \to z_i}) \quad (1)$$

This equation is at the heart of the Viterbi algorithm, and forces the current state to depend on all previous states with its recursive nature.

Even though the ACS unit is only one component of the Viterbi algorithm, the authors explain that "since the ACS unit is much more complex, it is the bottleneck which limits the throughput rate." To combat this, the authors introduce the concept of an $M$-step trellis, as opposed to the traditional 1-step trellis used in the Viterbi algorithm. This can be used in the ACS loop, but will take $M$ times as long. Utilizing a combination of the original 1-step trellis and the $M$-step trellis allows the authors to parallelize the Viterbi algorithm in combination with a multiplexer.

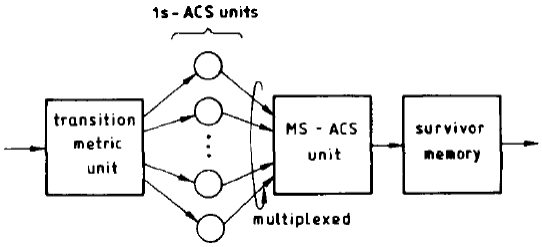


Figure 2: Multiplexed structure of Viterbi decoder

The new pipeline structure is shown in Figure 2. This is implemented in hardware using a systolic array, which is a pipelined structure of processing units called cells. Each of these cells can be computing in parallel and combined at the end, speeding up the overall process.

Du et al. [1] proposed extensions to two known methods for parallelizing the computationally intensive kernel of the Viterbi algorithm. The paper is applied to computational biology, so the authors first investigate existing parallelization methods for related sequence alignment algorithms.

The authors also demonstrate how to divide the Viterbi algorithm into two parts: independent sections of the calculation do not depend on previous results, and are therefore more amenable to parallelization. By isolating the dependent ant parts of the computation, the sequential work is minimized and the authors achieve greater speedup.

Because of the difference in application domain, the specifics of the approach taken by Bader et al. cannot be directly applied to NLP. However when designing parallel algorithms the basic principles of locality always apply. There are two specific techniques used by Bader et al. that can be applied to a more general Viterbi algorithm: dividing dependent and independent calculations and streaming algorithms that overlap data transfer with calculation. We were able to use the latter in our implementation.

## 3. Method Overview

Before implementing any of the algorithms, we first wrote a preprocessor to format the Penn Treebank data. Initially, the Penn Treebank files contained `<word>/<label>` pairs on each line and this format was not suitable for us. The preprocessor we implemented removed the irrelevant information and each output line contained a sentence of `<word>/<label>` pairs. An example is shown in Table 3. Our preprocessor also split the data up into multiple sections which helped later on during parallelization.

| | | |
|---|---|---|
| List/VB | $\rightarrow$ | List/VB |
| [ the/DT flights/NNS ] | $\rightarrow$ | the/DT<br>flights/NNS |
| from/IN | $\rightarrow$ | from/IN |
| [ Baltimore/NNP ] | $\rightarrow$ | Baltimore/NNP |

Figure 3: Example of preprocessor converting Penn Treebank files

### 3.1. Sequential

Next, we needed to implement sequential versions of the two algorithms. From their definitions, the implementations of the Viterbi and Forward algorithms were straightforward. Since we were not concerned with the resulting scores from the algorithms, we did not construct the probability distribution from the training set, but rather used a random value for the probability of each transition, $P(t_i|t_{i-1})$, and emission, $P(w_i|t_i)$. However, we did use the actual words and tags from the dataset.

### 3.2. Parallelization

Using OpenMP, we were able to add parallelization to the sequential algorithms we had written. In addition, to achieve scalability beyond the limits of memory afforded by the system, the algorithms needed to divide the work into tractable chunks and process them incrementally by bringing a chunk into memory, processing it, and then either writing it out or reducing it, depending on the calculating. We utilized parallelization to overlap the I/O work with computation.

This parallelization allowed us to explore possible speedups by reading data and allocating the lattice in parallel with applying the Viterbi algorithm. In other words, while on thread or process is reading data from disk, the other can run the algorithm.

Note that this is the most coarse-grained parallelism we will be doing. Since each step of the sequence involves a significant amount of calculation we have the opportunity for more parallelization, however streaming algorithms only appear to be applicable at the most coarse level for our purposes.

## 4. Experimental Evaluation

We evaluated our implementation on the Wall Street Journal section of the Penn Treebank, which has 45,920 sentences and 1,107,365 part-of-speech tagged words.

We evaluated our work by running the algorithm on the Lonestar section of the Texas Advanced Computing Cluster (TACC) [2]. These machines run compute nodes with two sockets each, each socket containing six cores, giving each process up to 12 simultaneous processors.

### 4.1. Methodology

Since our implementation had two levels of parallelism, coarse grained and fine grained, we characterized performance from a single thread at each level up to 12. We did not look at more than a total of 12 threads, though, since the computation is not I/O bound and running more than a total of 12 threads would degrade performance.

[2] `https://www.tacc.utexas.edu/resources/hpc/lonestar`

### 4.2. Results

## 5. Future Work

## 6. Conclusion

## References


[1] Z. Du, Z. Yin, and D. Bader. A tile-based parallel Viterbi algorithm for biological sequence alignment on GPU with CUDA. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. IEEE*, 2010.

[2] G. Fettweis, & H. Meyr. Parallel Viterbi algorithm implementation: Breaking the ACS-bottleneck. *Communications, IEEE Transactions on*, 37(8), 785-790, 1989.