Improving Performance of the Viterbi Algorithm with Parallelism

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.

We believe that parallelism affords significant performance gains when applying the Viterbi algorithm to NLP as well, so the goal of this experiment was to utilize parallelization techniques on the Viterbi and Forward algorithms to reduce the running time of these polynomial time algorithms. 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

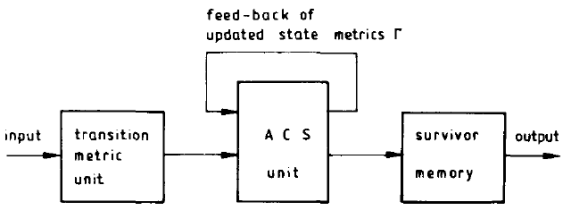

Figure 1: Pipeline structure of Viterbi decoder

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, Γ_{n+1,z_i}, as the add-compare-select (ACS) unit of the

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

Viterbi algorithm, shown in Figure 1. The formula to find this updated state is shown below (where λ is the transition probability):

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

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.

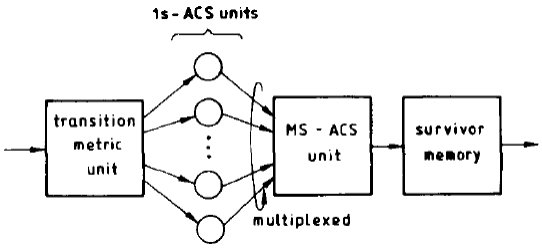

Figure 2: Multiplexed structure of Viterbi decoder

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

List/VB	$\rightarrow$	List/VB
[the/DT flights/NNS]	$\rightarrow$	the/DT flights/NNS
from/IN	$\rightarrow$	from/IN
[Baltimore/NNP]	$\rightarrow$	Baltimore/NNP

Figure 3: Example of preprocessor converting Penn Treebank files

is shown in Table 3. Our preprocessor also split the data up into multiple sections which helped later on during parallelization.

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.

We initially set out to explore three levels of parallelism:

1. Our most coarse-grained parallelism is simply processing different sentences in parallel. For the most part, this classifies as "embarrassingly parallel," since there is no coordination needed between the threads as they process the data. The only coordination required at the end of the forward algorithm is multiplying all of the results to get the full observation probability, which can be accomplished with a strait-forward parallel reduction.

2. Our fine-grained parallelism involves parallelizing within the Viterbi and forward algorithms. Each time step involves an $O(n^2)$ calculation that is essentially a matrix-vector multiplication (with a unit vector).

3. The final level of parallelism that we explored but did not implement involved parallelizing across time steps of the forward algorithm. Recall that each step involves the recursion, where v refers to the forward lattice, t the time step, j the current state, n the number of states, b the observation probability, and a the transition probability:

$$v_{t,j} = b_j(t) \sum_{k=0}^{n} v_{t-1,k} a_{k,j} \tag{2}$$

We were hoping to be able to modify this such that we could calculate the values of the lattice with a parallel prefix sum, however the recursion is exponential not linear (there is a factor of n steps at each of the t iterations), so we could not find a way to do this.

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 defined the problem size to be the number of words processed.

We ran 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, since the computation is not I/O bound and running more than a total of 12 threads degrades.

We ran the implementation over 5%, 10%, and 20% of the full dataset when determining scaling, and against the full dataset to determine peak performance.

We compared performance of the GNU and Intel compilers.

4.2. Results

We achieved relatively good scaling as shown in the strong scaling plot, Figure 4, and our peak performing configuration was able to process the entire Wall Street Journal dataset in about 3.5 seconds. While the speedup is not perfect, we are able to maintain performance increases as we increase threads all the way up to the cores available on the machine. Note that in the scaling figures the number of threads is the total number of both coarse and fine-grained threads. The scaling plots report the best performing of the thread combinations.

We found that the optimal thread combinations either used all coarse-grained or all fine-grained threads (see Figure 5). While there's nothing inherent the algorithm that would cause this, we think that processor affinity could be the source of the slowdown. Since the TACC machines have a non-uniform memory architecture (NUMA), the two sockets per node do not share the same memory, so threads scheduled on different sockets incur communication costs when writing to the same address space. OpenMP does not provide an API to specify processor affinity, so we can't deterministically avoid this situation.

We also found what appears to be a bug in g++ 4.7. When scheduling only fine-grained threads in g++ compiled binaries, the running time increases drastically (Figure 6).

If we kept the coarse-grained threads at one, but removed the outer OpenMP directive, the running time dropped to the expected amount, suggesting that the source of the bug is g++'s handling of the nested OpenMP directives. This is a relatively new feature of OpenMP that is poorly specified[3] and saw slow adoption among the compilers. Our suspicions were further confirmed when we did not see the problematic behavior on g++ 4.8.

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

[3] `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`

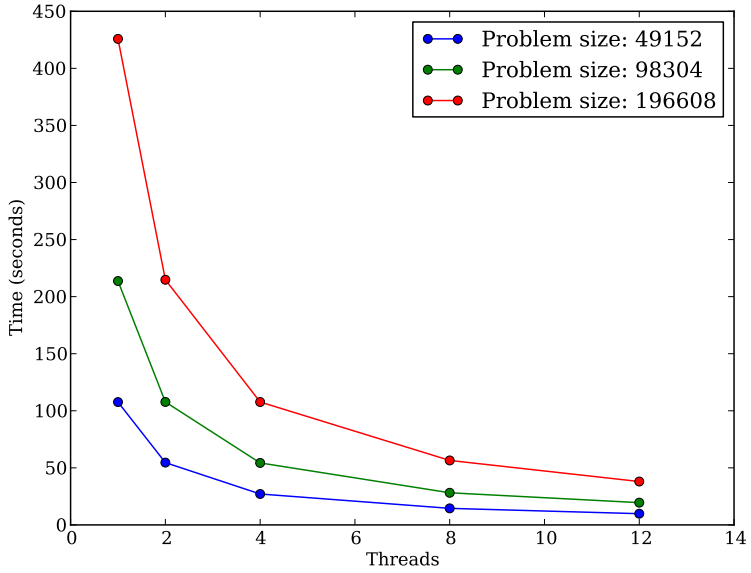

Figure 4: Strong scaling

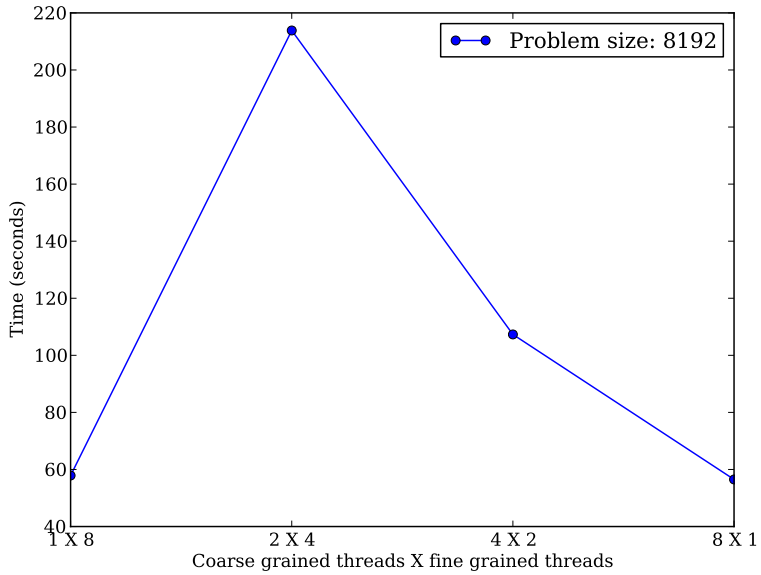

Figure 5: Coarse versus fine grained parallelism

We were also caught off guard by how much better the GNU-compiled code performed than the Intel-compiled code. The Intel compiler, being made by the same company that manufactures the chip, is generally thought to perform better, however in our tests the GNU-compiled code ran almost twice as fast (Figure 7). We expect that this is a configuration issue, though we did ensure that the `-O3` flag was used in all performance benchmarks.

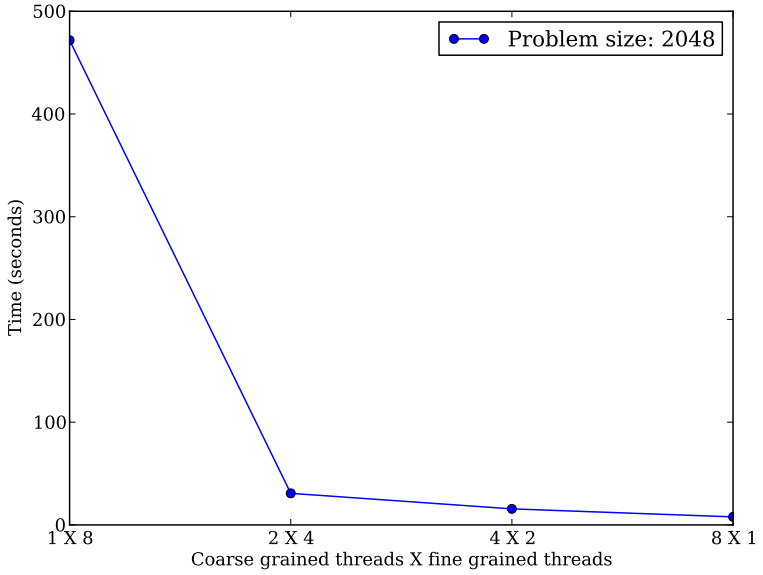

Figure 6: Coarse versus fine grained parallelism

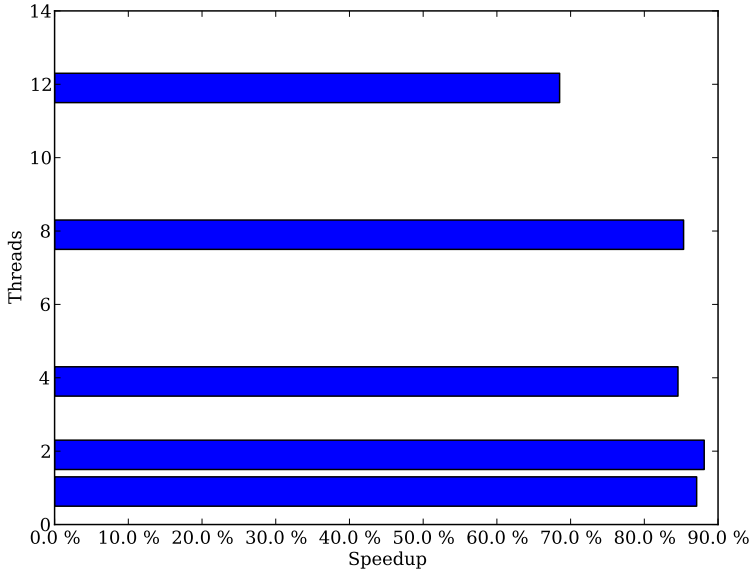

Figure 7: Speedup of GCC compiler over Intel compiler

5. Future Work

Our future work would include further investigation of the poor performance we saw when combining levels of parallelism. The two approaches we would take to address this is using a version of OpenMP that supports processor affinity specification, such as the latest Intel compiler

or clang-openmp[4]. Furthermore we could address affinity concerns by using MPI[5] to create a distributed parallel system. This would also allow us to take advantage of more parallelism and decrease the time required to arrive at a solution.

6. Conclusion

We applied a number of parallelization techniques to the Viterbi and forward algorithms for processing data in regards to parts-of-speech. We were able to acheive significant speedup through multiple levels of parallelism and by overlapping file I/O with the actual computation. We believe that we could acheive even greater gains by using a distributed algorithm, and that the techniques we found could be used to improve performance on a broader range of applications of the Viterbi algorithm including learning.

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.

[4] `http://clang-omp.github.io`

[5] `http://www.dmoz.org/Computers/Parallel_Computing/Programming/Libraries/MPI`