Auto-complete using an N-gram language model

**Introduction:**
Auto-complete is a ubiquitous and useful technology that allows for suggestions in real time while writing text. Auto-complete must generate “likely” (to be defined), context-based suggestions at a speed faster than that of a human typist to be useful.

Because of auto-complete’s time-sensitive nature, it is important that auto-complete software runs quickly. Luckily, many aspects of the auto-complete problem are non-sequential and are therefore parallelizable.

**N-grams:**
We define the likelihood of a word occurring at position $t$ in a sentence conditional on the preceding $t-1$ words as:

$$P(w_t | w_{t-1}, w_{t-2}, ..., w_1) = \frac{Freq(w_1, w_2, ..., w_{t-1}, w)}{Freq(w_1, w_2, ..., w_{t-1})}$$

Here, $(w_1, w_2, ..., w_{t-1}, w)$ represents an ordered sequence of words within a sentence. We dub these ordered sequences N-grams, where N is based on the length of the sequence. We pick the word with maximum likelihood as our auto-correct suggestion and break ties between two words based on their respective frequencies in the training text. To account for the denominator of the above equation potentially being zero, we can add a small constant.

To compute the N-grams and their respective frequencies, we can use a mapreduce approach in which we divide training text into chunks and compute the frequencies of all N-grams in parallel and combine the result.

**Tries:**
To look for potential suggestions based on the incomplete word at position $t$ in a sentence ($w_t$ in the equation above), we use a trie data structure. Each node in the trie will represent a prefix of a string and the subtrees of a given node contain all words the current node is a prefix of. For example, the node representing prefix “fa” might have the children “family”, “farm”, “fame”, and “farce” contained in its subtrees. We can perform a DFS to obtain the potential matches with a certain prefix.

The DFS can be parallelized as we can start a thread of computation when searching a new branch of the trie. We can also experiment with the granularity of concurrency by placing limits on the number of levels of the trie in which we generate new threads.
Auto-complete and Haskell Concurrency:

With auto-complete, we have the opportunity to implement the mapreduce framework while also using a concurrent DFS search on a trie, offering several fruitful areas of concurrency. Other than the initial I/O for reading training text, auto-complete is parallelizable and we believe we will observe a dramatic speedup through concurrency.