Project Objective

We aim to implement parallel regular expression matching via a data-parallel NFA implementation. We summarize our deliverables as followed:

1. Implementation of a module for regex parsing, NFA generation, sequential and parallel matching.

2. Rigorous evaluations of parallel matching in large text files.

3. Performance evaluation and comparisons with various $K$ cores and sequential implementation ($K = 1$), as well as other Haskell implementations if time left.

4. (If time left) Implementation of real-world regex applications.

5. (If a lot of time left) Lower bound computation of number of states needed for computationally equivalent DFA from NFA with $n$ states.

To clarify: We are not trying to find a needle in a haystack, but trying to determine if the whole text matches a regex pattern. As a crude example, we would check if a 1 GB long string of "a . . . b . . . c" matches the regex $R = ab^*c^*$.

Background

A regular expression (regex) is a search pattern that can be recognized with a finite state machine (FSM). Specifically, the underlying FSM for regex patterns is a nondeterministic finite automata (NFA). Formally, an NFA is a five-tuple $(Q, \Sigma, q_0, \delta, F)$ where $Q$ is the finite set of states, $\Sigma$ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow P(Q)$ is the transition function (where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$), $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accept states (Sipser).

To check if a string $w = w_0w_1 \ldots w_{k-1}$ satisfies a regex pattern $R$, $w$ can be run on the corresponding NFA $N$ with the initial state $q_0$. Then, for every character $w_i$ of $w$, the transition function is performed with the current state $q_i$ and the current character $w_i$. After performing the transition function $\delta(q_i, w_i)$, the NFA follows all possible resulting states in parallel. After reading the entire string $w$, if at least one instance of $N$ is in a state $q \in F$, then $N$ accepts $w$.

Instead of having $N$ branch with each possible next state, the set of reachable states $S_i$ can be recorded to achieve a simulation of $N$ that does not branch. This approach has roots in the NFA to DFA (deterministic finite automata) conversion algorithm from Sipser’s “Theory of Computation.” Upon initialization, $S_0 = \{q_0\}$. For each character $w_i$, $N$ updates the set of reachable states according to the equation $S_{i+1} = \bigcup_{q \in S_i} \delta(q, w_i)$. After reading the whole string $w$, if $S_k \cap F \neq \emptyset$, $N$ accepts $w$ (in other words, there is some state $q \in S_k$ that is an accept state). The worst-case runtime for such implementation of an NFA with $n$ states on a string $w$ (where $k = |w|$) is $O(nk)$. 
Approach

Despite their sequential nature, NFAs can be partially parallelized. First, partition the input string $w$ into $K$ chunks (or substrings) of similar length, where $C_i$ is chunk $i$ and $w = C_1 \ldots C_K$. Additionally, the transition function $\delta$ can be generalized to the transition lookup table $T_i : S_i \rightarrow S_{i+1}$, which takes in the set of reachable states $S_i$ and computes the next set of reachable states $S_{i+1}$ with the input $w_i$. For each chunk $C_n = w_i \ldots w_j$, the overall transition lookup table $T_i \rightarrow j : S_i \rightarrow S_{j+1}$ can be computed, which effectively merges the transition lookup tables $T_i, \ldots, T_j$ into a single transition lookup table that takes in the set of reachable states $S_i$ and produces the set of reachable states $S_{j+1}$ after all characters $w_i \ldots w_j$ in chunk $C_n$.

These $K$ transition lookup tables $T_{i_1 \rightarrow j_1}, \ldots, T_{i_k \rightarrow j_k}$ can be computed in parallel. Then, the initial set of reachable states $S_0 = \{q_0\}$ can be passed through all $K$ transition lookup tables in order. This operation would result in the final set of reachable states $S_k$. If intersection of $S_k$ and the set of accept states $F$ is non-trivial, then there exists an accept state that is reachable from $q_0$, and $N$ would accept $w$. The worst-case runtime for one chunk of this implementation of an NFA with $n$ states on a string $w$ (where $k = |w|$) that is split into $K$ chunks is $O(\frac{nK}{K})$. With at least $K$ cores, the computation on each chunk can be parallelized, and when $n < K$, the runtime of the parallelized version of this algorithm outperforms the sequential version (when $K = 1$).

Rough Outline of Algorithm

We summarize our approach with the following algorithm:

- Convert regex $R$ to NFA $N$ with $n$ states (where $n$ is bounded above by some integer for reasonable runtimes)
- Given $K$ cores, split input string $w$ into $K$ chunks $C_1, \ldots, C_K$.
- For every chunk $C_n = w_p \ldots w_q$,
  a. Run transition function $\delta$ on each character $w_s$ in chunk $C_n$ for every state $q \in Q$ to generate transition lookup table $T_s$.
  b. Combine all transition lookup tables $T_{i_1}, \ldots, T_{i_j}$ to generate transition lookup table $T_{i_1 \rightarrow j}$.
- Pass $S_0 = \{q_0\}$ through each of the $K$ transition lookup tables $T_{i_n \rightarrow j_n}$ for chunk $C_n$ to obtain the final set of reachable states $S_K$
- Compute the intersection of $S_k$ and the set of accept states $F$ to determine if $N$ accepts $w$.

References