CS 140 Assignment 5:
NFA Based String Matching

Assigned February 10, 2010

Due by 11:59 pm Wednesday, February 24

The purpose of this assignment is for you to gain experience in a common real-world scenario: You are given an existing sequential program, and you will parallelize it using Cilk++. Your job is to convert the sequential program into a parallel one, without introducing any data races, and get a reasonable speedup. For grading purposes, both correctness and performance will count.

For this assignment, there is one part of the program you will parallelize (because it will be executed many times on large inputs), and another part that you will leave sequential (because it only runs once). You will measure speedup only on the part you parallelize.

1. The problem domain

The underlying problem is to locate a "target" character string that fits a particular description, within a particular set of text "data". Versions of this problem show up in many different applications, ranging from the "find" command in a word processor to the reconstruction of a biological genome from DNA sequencing data. Finding a simple string in a word processor is an easy computation; but when the strings and the data get very long, parallel computing must come into play.

For this homework, you will parallelize a sequential code for a very basic and important string matching problem. There are two inputs: first, a "regular expression" that describes the set of target strings you're looking for; second, a string (or maybe a list of strings) to be checked against the regular expression to see if they match the target set.

The program first converts the regular expression into a so-called "nondeterministic finite state automaton" or NFA, which is a description of an abstract machine that recognizes strings that match the regular expression. You don't need to parallelize this conversion, because it only happens once on a small data set. The second step is the core of the algorithm, and the most important part: Here the program takes a particular input string and uses the NFA to decide whether or not the string is in the target set. You will parallelize this core part of the algorithm, which checks one single input string against the NFA. The input string can be extremely long in practice, so parallel efficiency makes a difference in this step.

2. What to implement

You will start with the serial code at http://www.cs.ucsb.edu/~cs140/cilk/nfa.cpp, and produce a parallel version of it. This program accepts two input files, one that contains the regular expression representing the target set, and one that contains a list of strings to be checked to see whether they are targets.

Compile the serial program using: >> **g++ -O2 nfa.cpp -o nfa**

Execute the serial program using: >> **./nfa regex.in strings.in accepted.out**

Sample regex and strings files are available on the course web site, along with the corresponding output file.

For example, the regular expression, **a(a|b)+bc** matches any string that starts with the character '**a**', ends with '**bc**', and in the middle has a substring of length at least one that contains only **a**'s and **b**'s. This regular expression matches the string **abbbc** but not the string **abc.**

So, what do you actually need to change to parallelize this code, and what can you leave alone?

The program first converts the regular expression into an NFA. You don't need to parallelize (or even look at) that part of the code. In fact, we put that code in a different header file, http://www.cs.ucsb.edu/~cs140/cilk/construct.h. You will need this header to compile your program, but you shouldn't modify it.

The next part of the code reads the input string and uses the NFA to decide whether any substring of the input string matches the target set as defined by the regular expression. You **are** expected to parallelize the part of the code that matches any substring of the given input string. In particular,

1) You **should** parallelize the breadth-first search operation within the matchprefix() function. For that, you must convert the global data structures to appropriate hyperobjects. This is a non-trivial task, and is the bulk of the homework.

2) You can, if you wish, perform a different matchprefix() for every possible suffix of the input strng in parallel.

3) You may use your imagination to find any other way of paralellizing the computation on a single input string.

You are **not**, however, allowed to parallelize across the different input strings. These are intended to be separate, independent test cases for your parallel program. Each string in the strings.in file should be processed in its proper order, one after the other.

3. Theoretical background

The rich theory of formal languages defines a hierarchy of languages, based on their computability and expressiveness. The simplest and easiest-to-recognize languages are *regular languages*. They can be recognized in linear time using abstract machines called finite state automata. A finite state automaton is a machine that contains a finite number of states, transitions between states, and actions performed during transitions.

For string matching, you can think of a finite state automaton as a directed graph, where states are vertices and transitions are edges. Each edge has a label that is either a character from the alphabet, or an empty (epsilon) label. A finite state automaton has two kinds of special states: one *start* state (s0), and some number of *final* states.

The automaton processes an input string one character at a time. The automaton starts in state s0, and follows edges (transitions) from one state to another. The automaton can follow an edge labelled by a character only if the edge label matches the next character in the input string; in that case, it consumes that character and moves on to the following character in the input string. The automaton can also follow an epsilon edge (with an empty label) without consuming the input character.

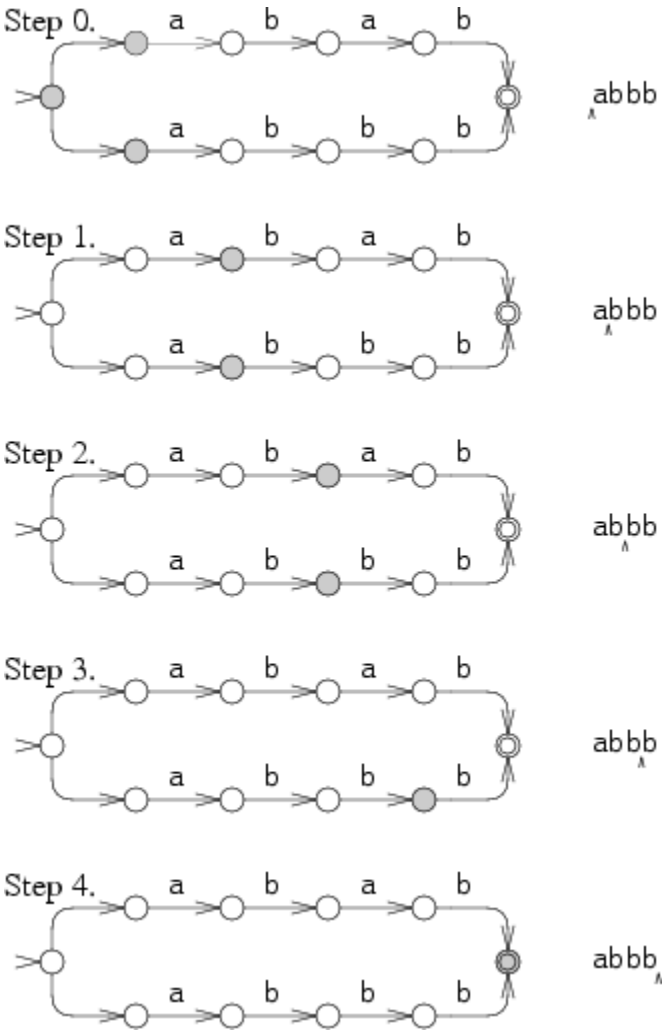
This can end in one of two ways: either the automaton reaches a state where it is stuck because no label matches the next input character and there is no epsilon transition, or it reads the last character of the input. If, after reading the last input character, the automaton is in one of the *final* states, the input string is declared to be a target string. If the automaton gets stuck, or if it finishes reading the string in a non-final state, the input string is declared to be a non-target string.

The tricky part of this is that, at any given point, the automaton might have more than one choice of what to do next -- there might be two or more transitions labelled with the next input character, or one or more epsilon transitions might be available. Therefore, the automaton has to explore all possible paths before concluding that the input string is not a target -- the rule is that if *any* possible path ends in a final state, the input string is a target.

There are several ways to explore the possible paths through this “non-deterministic” automaton. Theoretically, there is a way to convert any non-deterministic automaton to a deterministic one, where there is never any choice about which way to go; but that conversion can grow the size of the automaton exponentially, making it impractical. A better alternative is to explore the different possible paths through the automaton in parallel.

The original, “classical” way to explore all the paths was to use depth-first search with backtracking to follow one path at a time. However, for our parallel implementation, we use a breadth-first search that explores all the paths at the same time.

Here is an illustration of the breadth-first search approach with input string “abbb”. The caret ^ shows how much of the string has been consumed, and the shaded circles are all the states the NFA could possibly be in at that step. The start state is at the far left, and the only final state is at the far right. The string is accepted as a target because one of the paths (the bottom path) ends in the final state after consuming the entire string.



In the code, the breadth-first search is implemented using two lists of states. The current list (or *clist*) holds the current set of possible states (the shaded circles above), and the next list (or *nlist*) holds the new frontier that is one step ahead. The search is broken into phases, where each phase consists of examining all the states in the current list, computing their successors, and storing those successors that have not yet been seen into the next list. At the end of each phase, the nlist becomes the clist.

More information on this subject is at http://swtch.com/~rsc/regexp/regexp1.html

4. What to turn in

Report the parallel scaling of your code on large test inputs (which will be available a week before the deadline). Use the same methodology as hw3 for increasing the number of active cores (CILK_NPROC=n) for scaling.

Turn in your parallel nfa.cilk code and a set of plots that show the scaling behavior.

You may do this homework singly or in pairs.