Method

The matching process is inherently sequential, since we need to process the string one character at a time. It may be parallelized, however, through the use of an optimistic approach. Assume we have n threads. We divide the input string into n pieces. The normal thread gets the first piece of the string, and performs matching as above.

The n optimistic threads each perform matching on their own portion of the string, but since they are not sure what state the DFA will be in to start with, they simulate matching from every possible state simultaneously.

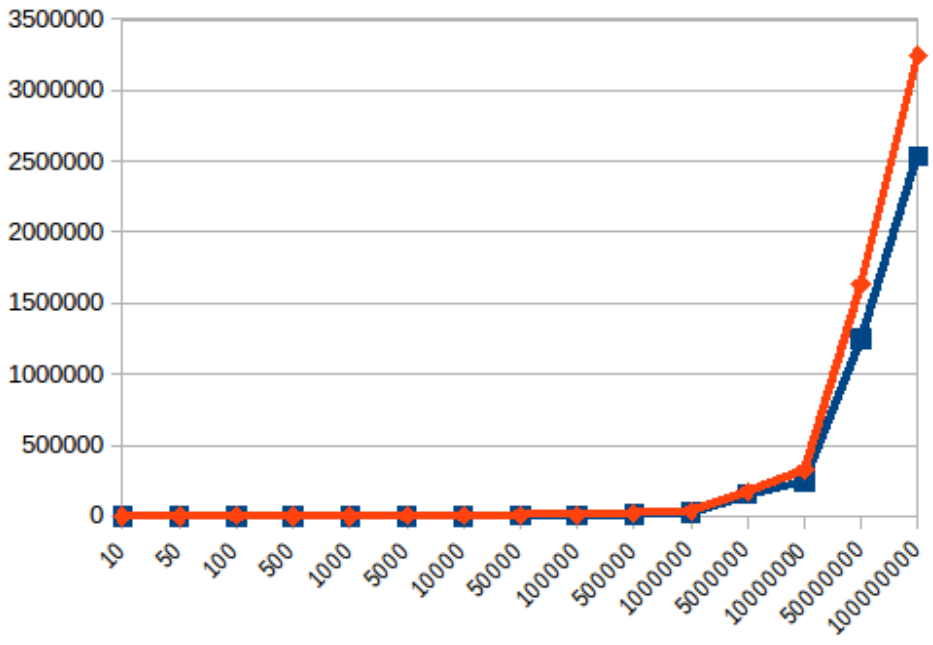
Once the normal thread reaches the end of its input fragment i, it looks at the mapping produced by the thread handling the next fragment $i + 1$. This repeats until the matching process is completed for the entire string

In my code I have taken variables as the string size, no of state for the fsm, size of the alphabet, and the thread count.

The state mapping in generated in code randomly and so is the string.

So every segmentof string is executed s(no_of_states) times
so total no. Of segments is equal to the no. Of threads (t).
n is the size of the string
effectively the code runs on the string for a parallel program is s*n/t
while for serial program it runs for n

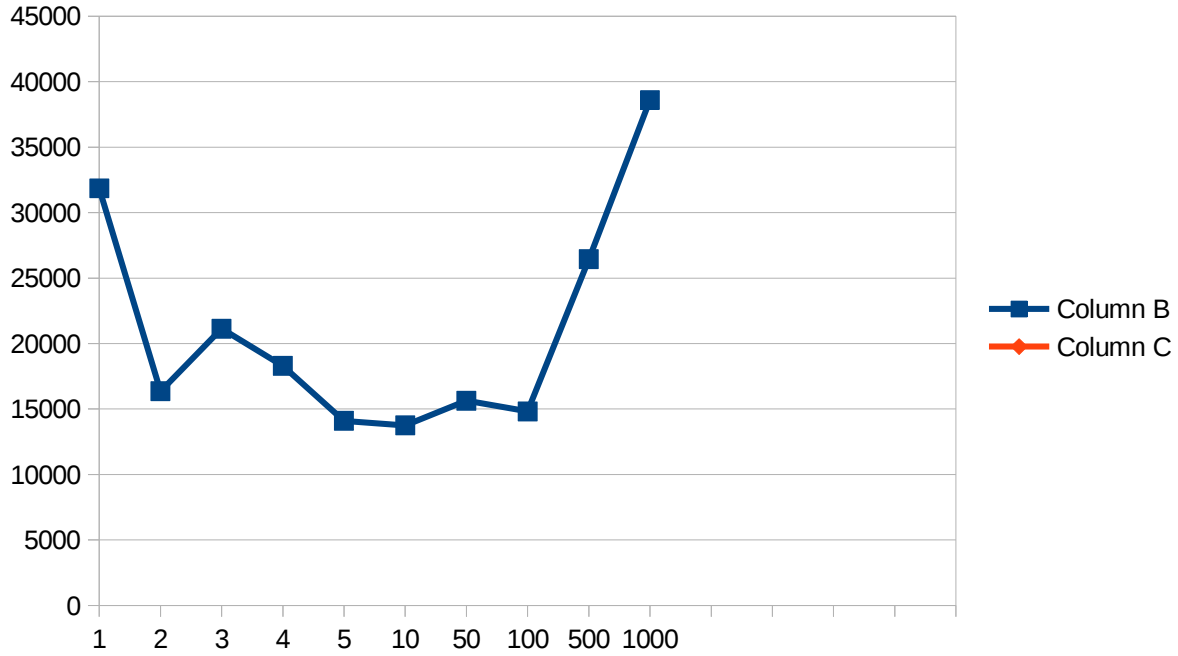
Parallel will perform theoritically better than serial
if: no_of_threads (t) > no_of_states (s)



Graph Time vs size of word with constants

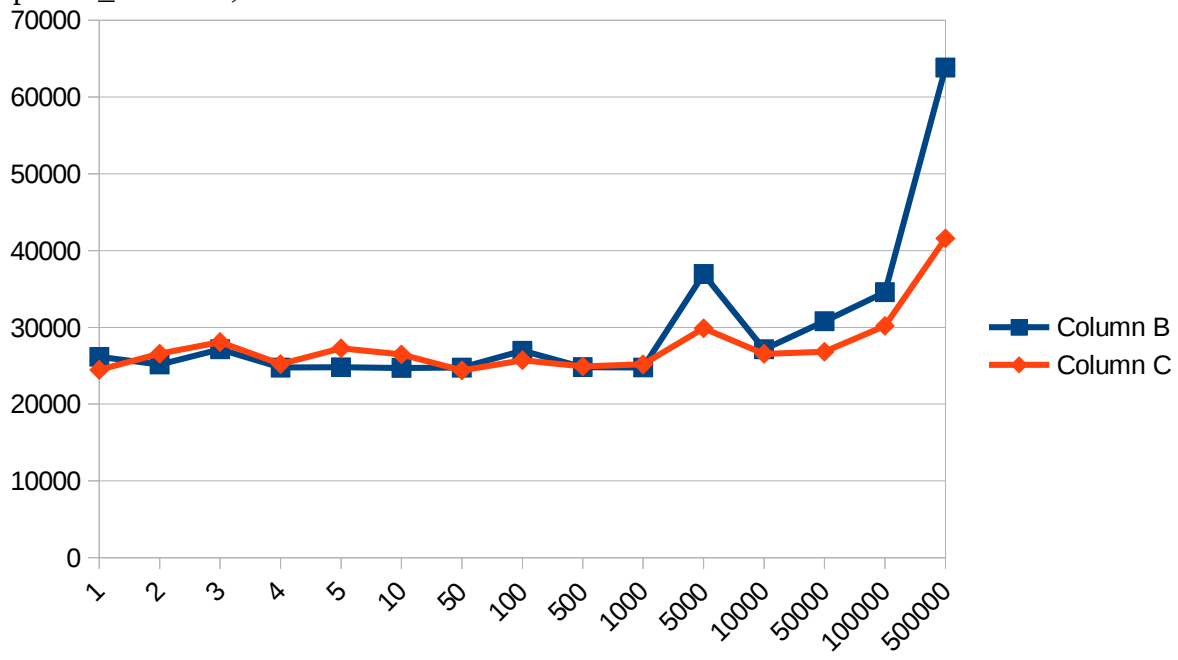
AS:
int no_of_states = 3;
int alphabet_size = 10;
int thread_count = 10;

Red- Parallel
Blue - sequential



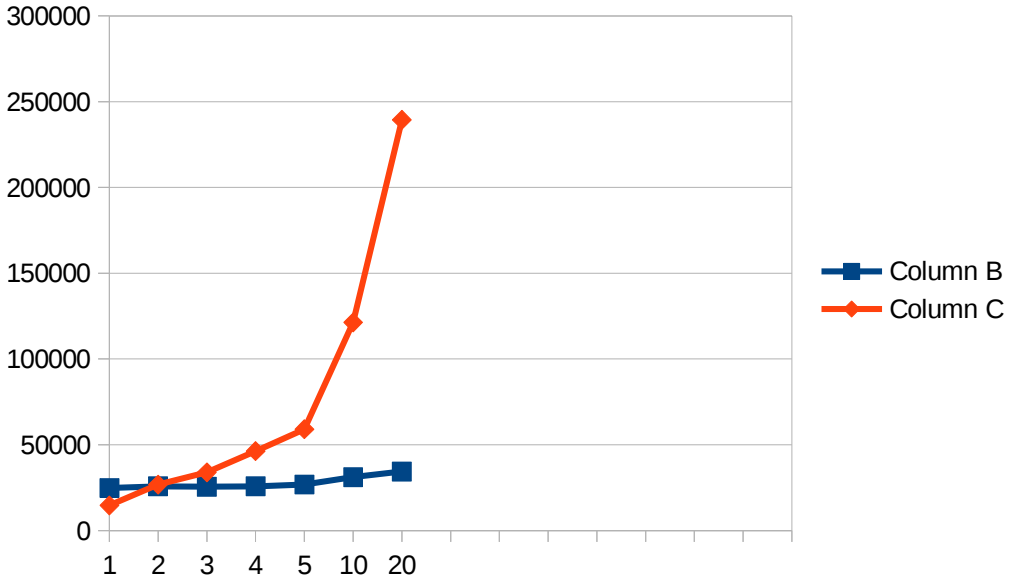
Graph Time vs thread counts with constants

AS:
int string_size = 1000000;
int no_of_states = 1;
int alphabet_size = 10;



Graph Time vs size of alphabet with constants

AS:
int string_size = 1000000;
int no_of_states = 2;
int thread_count = 10;
Red- Parallel
Blue – sequential



Graph Time vs state with constants
AS:
int string_size = 1000000;
int alphabet_size = 10000;
int thread_count = 10;
Red- Parallel
Blue – sequential