Fast String Matching

Neel Krishna - PES1201700029

Karan Panjabi - PES1201701142

Abstract

Modern day computers have a word-size of 64-bit or 32-bit (not as much prevalent now). That means the CPU can process an assembly instruction on data of size 64 bits. Every character in C is represented using 1 byte (8 bits). A lot of string matching algorithms compare character by character. This means that at CPU level, the 24 higher order bits have been padded with 0s and the last 8 bits represent the character. This wastes CPU power.

So, instead of comparing 8 bits/1byte/1character at a time, we can compare 64bits/8bytes/8characters in a single shot on a 64 bit machine or 4 characters on a 32-bit machine.

Brief Algorithm

The string matching algorithms based on shifts require us to find out the number of characters that match partially with the text (not Naive). Let;s say the word size is 32-bits and the text string is “ABCD” and the pattern string is “DEFG”. We can interpret the text and the pattern as an unsigned int32 and compare them directly using ==. However, since we also need some information of how many characters matched to figure out the shift amount, we can do two things: subtract the two integers or take an xor. Subtraction involves the usage of an adder at machine level, which involves the carrying-over of bits if the need arises. This is not as efficient as an XOR where a comparison can be done directly and doesn't require and carry bits to be forwarded to higher order bits. So, doing an XOR will give us sufficient information of how many characters matched, and then we can directly use the mismatched character to figure out the shift required using the shift tables used in the Horspool and Boyer-Moore algorithms.

The information about the number of characters that matched can be got from the single number that we get after XORing the word. There are ranges that that number can fall into which will correspond to the number of characters that matched. This is the crux of the algorithm.

```
int getmatchedpos(int32 num)
{
	static int32 low_3 = 1;
	static int32 high_3 = 255;

	static int32 low_2 = 1 << 8;
	static int32 high_2 = (1 << 16) - 1;

	static int32 low_1 = 1 << 16;
	static int32 high_1 = (1 << 24) - 1;
```

```
      if (num == 0)
          return 4;
      else if (num >= low_3 && num <= high_3)
          return 3;
      else if (num >= low_2 && num <= high_2)
          return 2;
      else if (num >= low_1 && num <= high_1)
          return 1;
      else
          return 0;
}
```

This test function takes the xor of the pattern and text and based on it's value gets the number of characters that match from the right of the string on a little endian system. The interesting part is that this function would give the reverse output in case of big endian system.

Naive String Matching

Improving the naive string matching algorithm was straightforward. Instead of treating a one-byte entity as one character, we treated an eight-byte entity as a character (handling the boundary case for the last character). The rest of the algorithm for matching remained more or less the same.

```
ws patws = *(ws *) (pattern + j);
ws tws = *(ws *) (text + s + j);
// if all chars match then, increase j by that 8, decrease charleft by 8
    if(patws == tws) {
        j += sizeof(ws);
        charleft -= sizeof(ws);
    }
```

Horspool and Boyer-Moore Algorithm

Improving the Horspools / Boyer Moore Algorithm requires the same steps. The standard algorithms need to get k -> the number of characters that have matched from the right. The challenge arises here as well when the pattern size is not a multiple of the word size. If the pattern size is greater than wordsize, then we need to keep on dividing the number of blocks of word size and keep checking until a mismatch occurs. If the number of characters left is less than the word

size then we need to pad zeros appropriately. Note that the remaining characters (less than wordsize) should be appropriately placed according to endianness.

Performance Report

Algorithm	**Input Size and Characteristics**	**Standard time (s)**	**Improved time (s)**
Naive	Text: 10^6 Pattern: 5000 Worst case input	13.9	2.7
	Text: 10^4 Pattern: 1000 Worst case input	0.074	0.014
Horspool	Text: 10^6 Pattern: 5000 Worst case input	9.7	2.9
	Text: 10^4 Pattern: 1000 Worst case input	0.026	0.025