Parallel String Matching

Hector Acosta and Abed Haque

Abstract

There are many computational problems that that involve finding a specific pattern in a given set of data. String matching refers to the process of finding one or more strings (pattern) within a larger string (text). In this project we explore several algorithms for the string matching problem and analyze their time and work complexity. We then specifically deconstruct Vishkin's optimal parallel string matching algorithm and implement it in Java.

Introduction

In order to describe string algorithms, it makes sense to define some notation first. Let alphabet Σ be a finite set consisting of symbols. A string *x* is then defined as a finite sequence of elements in Σ. The length of x is denoted by |x|, the concatenation of two strings *x*, *y* is denoted by *xy*, and the *ith* element of *x* is denoted by x(i). Let then, *x* and *y* be two strings of length *m* and *n* respectively. A string *x* occurs in *y* if X(j) = Y(i + j -1) for all *j* such that 1 <= j <= m, we can also say that string *x* matches string *y* at the *ith* position.

In the parallel paradigm, matching is performed in two phases. We first perform the pattern analysis phase on the pattern to precompute a data structure that can speed up the matching process. In the algorithm we explored, this is called the *WITNESS* array. The text analysis phase then makes use of this pre-computed information to identify all the occurrences of a pattern *p* in text *T.* This paradigm is especially useful when the overhead of precomputing the data structure is less than the time spent in the matching operation. For example, when the match operation is going to be performed over large |T|.

Sequential String Matching Algorithms

Brute-force

We will start with the most naive sequential string matching algorithm, the brute-force algorithm. The brute-force algorithm simply iterates over the entire text T and then iterates over the entire pattern P to check if all values of T matches the corresponding values of P. This means that *O(m)* operations would occur *n* times, resulting in a total of *O(nm)* operations. If there were *n*m* processors available, this algorithm could be implemented in O(1) time on a CRCW PRAM since each of the *m* comparisons could be done in a single step in parallel, and the output array, *MATCH*, is simply computed with a logical AND over all booleans of the *m* outcomes.

Even though the parallel brute-force algorithm is fast, it has a prohibitively large number of operations. There are several sequential algorithms and parallel algorithms that can be done with only *O(n)* operations.

A java implementation is now presented:

```
public int match(String haystack, String needle)
    for(int i = 0; i < haystack.length(); i++ ) {
        for(int j = 0; j < needle.length() && i+j < haystack.length(); j++ ) {
            if(needle.charAt(j) != haystack.charAt(i+j)) {
                break;
            } else if (j == needle.length()-1) {
                return i;
            }
        }
    }
    return -1;
}
```

Knuth-Morris-Pratt (KMP)

It is worthwhile to mention the sequential KMP string matching algorithm for two reasons. First, the KMP algorithm is a classic well-performing sequential algorithm that we can use as a baseline for for comparison of our final parallel string matching implementation. Second, the KMP algorithm utilizes a pattern analysis phase in order to precompute a table to reduce the complexity of the algorithm.

The Knuth-Morris-Pratt algorithm makes optimizations that allows us to skip iterations in both the outer and inner loops of the brute force algorithm. We start by explaining the former. In the inner loop, we can use the information of the partial matches to skip outer loop iterations for which no match is possible. Additionally, by storing an extra array that keeps track of the last partial match we can avoid testing a number of characters equal to the overlap length. KMP has complexity O(n).

Vishkin's Parallel Matching Algorithm

WITNESS Array

Like many efficient string matching algorithms, Vishkin's parallel matching algorithm utilizes a fundamental pattern analysis phase. This phase uses a *witness function* Φ_{wit} that generates the *WITNESS* array. Let the pattern, *Y*, be a string of length *m* and period *p.* Let

$\pi(Y) = min(p,\ ceil(\frac{m}{2})) - 1$

The witness function can be defined as:

$\Phi_{wit}[0] = -1$

$\Phi_{wit}[i] = k$, *for* $0 < i < \pi(Y)$

Where *k* is any index for which $Y(K) \neq Y(i + k)$ for $i > 0$. In other words, the *WITNESS* array value, *k*, gives us an index of the element in which the pattern *Y* is different when compared to a version of *Y* that is shifted by *i* elements.

The *WITNESS* array can be computed sequentially in $\pi(Y)$ time with $\pi(Y)$ operations. The witness function can further be optimized to be run in O(*log(m))* time with *O(m)* operations.

Duel Function

The duel function utilizes the *WITNESS* array to very quickly eliminate positions that are guaranteed to not match with a given pattern. The duel function has three inputs: the text string Z of length n, a *WITNESS* array of the pattern string *Y* of length $m \leq n$, and two indices *i* and *j* where $0 \leq i < j \leq (n-m)$ such that $(j - i) < \pi(Y)$. The output of the duel function is either *i* or *j*, indicating the index that survived elimination. The algorithm is as follows:

```
k: = WITNESS[j - i]
if Z[j + k] ≠ Y[k] return i
else return j
```

If *i* and *j* are strictly chosen within the given constraints, it can be shown that *Y* cannot match *Z* at the position eliminated by *duel(i, j)*. Given the definition of the *WITNESS* function, it is evident that we cannot simultaneously have the the equalities *Z[j + k] = Y[k]* and *Z[j + k] = Y[k + j - 1]* occur at the location *j* in *Z*.

With a precomputed *WITNESS* array, the duel function can be implemented in *O(1)* time. The next section describes how the duel function is efficiently used in the text analysis of the parallel string matching algorithm.

Text Analysis

The text analysis section of the parallel algorithm consists of three steps:

1. Partition the text into 2*n/m* blocks, with each block containing no more that *m*/2 consecutive characters
2. For each block, eliminate all but one candidate position by using the duel function with a balanced binary tree
3. For each remaining candidate position, verify whether the pattern occurs by using the brute-force algorithm

After the text is partitioned into *m*/2 sized blocks in Step 1, each block is operated on in parallel. The most significant part of the parallel algorithm occurs in Step 2. The duel function can be practically treated as though it were an associative operation. It is true that the order of operation when dueling several inputs can potentially result in different outcomes. However, this is acceptable for our application, as our only goal in this step is to properly eliminate the vast majority of indices from being considered. Therefore, Step 2 can be viewed as a balanced binary tree with each internal node *u* containing the index that is returned by the function *duel(i, j)*, where *i* and *j* are the indices stored in the children of node *u*. With this implementation Step 2 can be performed in parallel in *O(log(m))* steps with *O(m)* work.

By the time we arrive at Step 3, we have reduced our problem from *n* potential matching indices down to only *2n/m* candidates for matching indices. This is a significant reduction, especially when dealing with large values of *m*! We can now use the parallel brute-force algorithm in *O(1)* *O(m)* work for each candidate.

Java Implementation of Vishkin's Parallel Algorithm

We now go over our implementation of Vishkin's algorithm using Java Streams. The implementation can be splitted into four primary functions: generateWitness(), duel(), treeDuel(), and match().

We first we explore our match implementation.

```
public int[] match(char []text) {
    if (text.length < pattern.length)
        return new int[0];
     int chunkSize = witness.length;
     int[] results = IntStream.range(0, text.length)
             .parallel()
             .filter(i -> i % chunkSize == 0)
             .map(start -> treeDuel(text, start, Math.min(start + chunkSize, text.length)))
             .filter(candidatePosition -> memcmpOffset(text, pattern, candidatePosition))
             .toArray();
     return results;
12.     }
```

Line 5 creates an exclusive Integer range from 0 to text length. The stream is then parallelized in line 6. Since we're only interested in the starting position of each section, we filter these positions in line 7. This corresponds to Jaja's step 1. "Partition T into ceil(2n / m) blocks containing no more than m/2 consecutive characters". In line 8, each one of these ranges gets reduced to candidate positions by using our treeDuel() function. Let's first consider a basic treeDuel() function as described in the algorithm:

```
1.  for (int i = 0; i < nextPowerof2(len); i++) {
2.      for (int j = 0, retIdx = 0; j < (finish - start)/ Math.pow(2, i); j += 2, retIdx++) {
3.          if (start + j + 1 == finish) {
4.              ret[retIdx] = ret[j];
5.              break;
6.          }
7.          int d1 = ret[j];
8.          int d2 = ret[j + 1];
9.          ret[retIdx] = duel(text, d1, d2);
10.     }
11. }
12. return ret[0];
```

The above code should result familiar, it's a parallelizable reduce operation that performs duel as its binary operation. Jaja describes this as follows: "For each block T eliminate all but one position as a possible candidate for a matching by using a balanced binary tree, where each internal node *u* contains the index returned by the function *duel(i, j)* and *i* and *j* are the indices stored in the children of *u*." Let's now consider the Java Stream version of this code:

```
1.  public int treeDuel(char[] text, int start, int finish) {
2.     OptionalInt result = IntStream.range(start, finish)
3.             .parallel()
4.             .reduce((i,j) -> duel(text, i, j));
5.     return result.getAsInt();
6.  }
```

The stream version takes advantage of the fact that the duel() function is an associative operation, therefore using the reduce() operation (line 3) is equivalent. As a final step, the match function (line 9) performs a brute force search over the candidate matches and returns the result to the caller.

Benchmarks

The presented code was benchmarked using the Caliper framework. In order to offset the overhead that's introduced by using Streams as opposed to simple for loops, we benchmarked our code against the Stream version of a brute force sequential match. Code follows:

```
public int[] bruteForceSequentialMatch(char[] text){

   int[] results = IntStream.range(0, text.length - pattern.length)
            .filter(i -> patternMatchesAtPosition(text, pattern, i))
            .toArray();

    return results;
}
```

As the length of the pattern, and the amount of possible matches increased, we observed better performance on Vishkin's algorithm. Benchmarks on an 8 core computer for 1 to 8 threads were also performed and can be found in the addendum.

Conclusion

The string matching algorithms presented earlier allow for efficient string matching by combining some common optimization techniques. Precomputing information (to generate a witness array) and reducing a balanced binary tree to compute the possible string matches are recurring themes in the world of parallel algorithms.

Vishkin's parallel string matching algorithm maximizes parallelism and drastically reduces the search space, allowing for O(m) time complexity as opposed to O(nm) in the brute force algorithm. As the benchmarks indicate, parallel string matching outperforms brute force search, as the pattern size increases.

References

- JáJá, J. (2001). An introduction to parallel algorithms. Reading, Mass.: Addison-Wesley.
- Crochemore, M., & Rytter, W. (2003). Jewels of stringology: text algorithms. New Jersey: World Scientific.

Addendum

- https://microbenchmarks.appspot.com/runs/b180348d-2ebd-42a5-92b1-3b8824e8afeb#r:scenario.benchmarkSpec.parameters.numThreads
- https://microbenchmarks.appspot.com/runs/aa0b3fdd-a6d2-4499-b19f-349b0f8c2ed3#r:scenario.benchmarkSpec.parameters.numThreads
- https://microbenchmarks.appspot.com/runs/f2436060-a649-4808-bf37-1861cd5db2ec#r:scenario.benchmarkSpec.methodName,scenario.benchmarkSpec.parameters.size