Parallel Algorithms for Text Analysis

Hector Acosta
Abed Haque

The String-Matching Problem

- Given a **Text** string of length ***n*** and a **Pattern** string of length ***m*** we want to find all occurrences of the pattern in the string
- The result will be a MATCH array that indicates all positions where the **Pattern** occurs in the **Text**

Applications

- Bioinformatics
- Plagiarism Detection
- Spam Filters
- Search Engines
- Databases
- Molecular Biology
- Speech Recognition
- Computer Vision

Terminology

- **Text** - The string that we are searching. Length of n.
- **Pattern** - The string that we are trying to find in Text. Length of m.
- Pattern **Prefix** - A substring from index 0 to i, where $0 \leq i \leq m$
- Pattern **Suffix** - A substring from index j to m, where $0 \leq j \leq m$
- A Pattern **Period** - A substring *X* if the $Pattern = X^k X'$, where *X* is concatenated with itself *k* times and *X'* is a prefix of *X*
- **The Period** of a Pattern - the shortest period of the Pattern
- A Pattern is **periodic** its period $p \leq m/2$

The Pattern *abababab a* is **periodic** with the period *ab*

Sequential Algorithms

Sequential Algorithm - Brute Force

- Most naive approach
- Try to match the pattern against the text for every position in the text up to index (n - m)
- O(m) operations required for each position
- Time complexity: O(nm)
- Work complexity: O(nm)
- Not very efficient!

Sequential Algorithm - Knuth-Morris-Pratt (KMP)

- Utilizes **Pattern Analysis** to prevent unnecessary backtracking
- Checks to see if there is a suffix that is also a prefix in the pattern that is already matched at the point of failure
- Time complexity: O(n)
- Work complexity: O(n)

Parallel Algorithm

Parallel Algorithm - Brute Force

- Assign a processor to every match and implement on a CRCW PRAM
- Time complexity: O(1)
- Work complexity: O(nm)
- Fast, but not work optimal!
- We could still use this if we were able to reduce the number of comparisons to a small number...

Goal:
Want a parallel algorithm with time complexity better than **O(n)** and an optimal work complexity of **O(n)**

Paradigm for handling pattern-matching problems

1. **Pattern Analysis**:

Preprocess the **pattern** to extract information about its structure, and store it in an array

2. **Text Analysis**:

The actual processing of the **text** using both the **pattern** and the information obtained from the **Pattern Analysis** phase

Pattern Analysis - The Witness Array

- Fundamental in parallel string-matching algorithms
- Keeps track of indices where elements are different when comparing the Pattern to a shifted version of itself
- Length of $min(p, ceil(m/2))$ for a text pattern Y of length m and period p
- W[0] = -1
- W[i] = k, where k is any index for which $Y(k) \neq Y(i+k)$ for $i > 0$

a	c	a	a	d	d
	a	c	a		

Example: When shifted by 1 and compared to itself, possible values of k are 0 and 1.

Witness Array Construction

	0	1	2	3	4	5	6	7
Pattern:	**a**	**a**	**b**	**a**	**a**	**c**	**d**	**a**
	a	a	b	a				
		a	a	b	a			
			a	a	b	a		
				a	a	b	a	
W:	-1	1	0	2				

Goal:
Efficiently eliminate indices where the pattern cannot occur

The duel(i, j) function

- Inputs:
  - A text string of length n
  - A WITNESS array of a pattern of length $m \leq n$
  - Two indices i and j of the text (where $0 \leq i < j \leq n-m$ and $(j - i) <$ WITNESS length)
- Output:
  - One of either i or j. The index that is **not** returned is an index where the pattern **cannot** occur in the text

The duel(i, j) function

- Recall: The WITNESS array provides us with an index of a pattern where the element is ***guaranteed*** to be different for a given **offset** with itself
- Notice that (j - i) is this offset
- Duel over many indices ➔ greatly reduce the possible locations where the pattern can occur
- duel() only takes O(1) time!

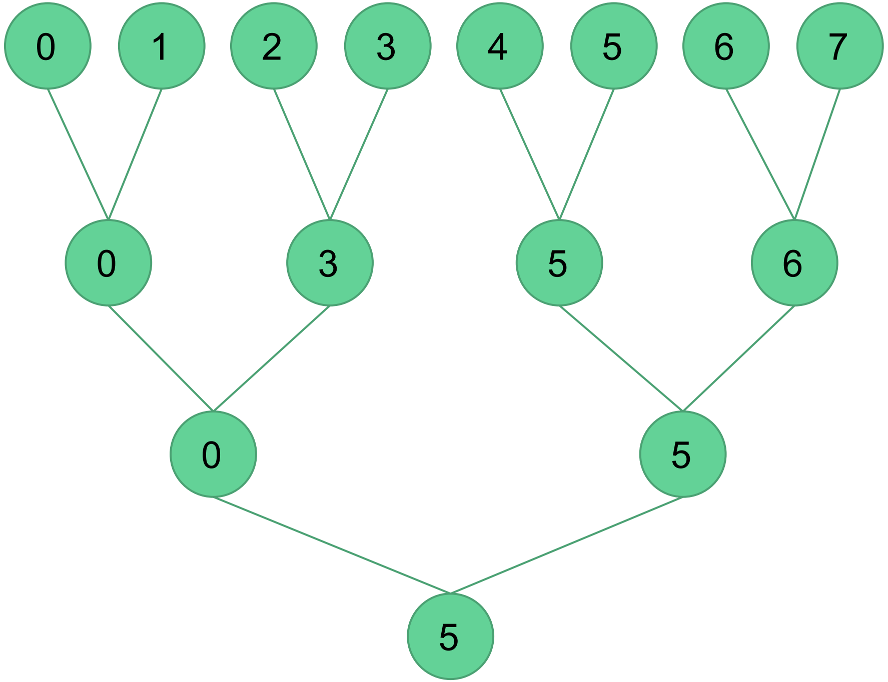
```java
public int duel(char[] text, int i, int j) {
    int k = witness[j - i];

    if (j + k >= text.length)
        return i;

    if (text[j + k] != pattern[k])
        return i;
    else
        return j;
}
```

treeDuel()

- duel(i, j) can practically be considered to be an associative operation
- The order of duels *can* affect the output, but for our purposes this output is still valid
- We can implement multiple duels as a balanced binary tree (reduce)
- We can safely eliminate all but one index from a substring if its length ≤ length(WITNESS array)



Parallel Text Analysis Algorithm

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

Java Implementation

Pattern Analysis - Witness Array Generation

```
static int[] generateWitness(char[] pattern) {
    int witnessLength = Math.max(2, Math.min((int) Math.ceil(pattern.length / 2.0), getPeriodLength(pattern)));
    int[] witness = new int[witnessLength];

    for (int offset = 0; offset < witness.length; offset++) {
        // Get 0 based index of the first differing character of
        // the shifted pattern
        for (int i = 0; i < witness.length; i++) {
            if (pattern[i + offset] != pattern[i]) {
                witness[offset] = i;
                break;
            }

            // shifted pattern matched exactly for every element
            if (i == witness.length - 1) {
                witness[offset] = -1;
            }
        }
    }

    return witness;
}
```

Text Analysis Step 1: Partition Text

```
public int[] nonPeriodicMatch(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 -> patternMatchesAtPosition(text, pattern, candidatePosition))
            .toArray();

    return results;
}
```

Text Analysis Step 2: Tree Duel

```java
public int[] nonPeriodicMatch(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 -> patternMatchesAtPosition(text, pattern, candidatePosition))
            .toArray();

    return results;
}
```

```java
public int treeDuel(char[] text, int start, int finish) {
    assert (finish <= text.length);
    assert (finish > start);
    if (finish - start == 1)
        return start;

    // duel is an associative operation, therefore we can use
    // the reduce operator
    OptionalInt result = IntStream.range(start, finish)
            .parallel()
            .reduce((i,j) -> duel(text, i, j));

    return result.getAsInt();
}
```

```java
public int duel(char[] text, int i, int j) {
    int k = witness[j - i];

    if (j + k >= text.length)
        return i;

    if (text[j + k] != pattern[k])
        return i;
    else
        return j;
}
```

Side Note - Java Streams are great!

```diff
     public int treeDuel(char[] text, int start, int finish) {
         assert (finish <= text.length);
         assert (finish > start);
         if (finish - start == 1)
             return start;

-        int len = finish - start;
+        // duel is an associative operation, therefore we can use
+        // the reduce operator
+        OptionalInt result = IntStream.range(start, finish)
+                .reduce((i,j) -> duel(text, i, j));

-        int[] ret = IntStream.range(start, finish).toArray();
-
-        for (int i = 0; i < nextPowerof2(len); i++) {
-
-            for (int j = 0, retIdx = 0; j < (finish - start)/ Math.pow(2, i); j += 2, retIdx++) {
-                if (start + j + 1 == finish) {
-                    ret[retIdx] = ret[j];
-                    break;
-                }
-                int d1 = ret[j];
-                int d2 = ret[j + 1];
-
-                ret[retIdx] = duel(text, d1, d2);
-
-            }
-        }
-        return ret[0];
+        return result.getAsInt();
     }
```

Text Analysis Step 3: Brute Force candidates

```
public int[] nonPeriodicMatch(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 -> patternMatchesAtPositionParallel(text, pattern, candidatePosition))
            .toArray();

    return results;
}

public static boolean patternMatchesAtPositionParallel(char []txt, char []pat, int startPosition)
{
    if (pat.length + startPosition > txt.length)
        return false;

    return IntStream.range(0, pat.length)
                    .parallel()
                    .allMatch(i -> txt[i + startPosition] == pat[i]);
}
```

Algorithm Complexity

	Time Complexity	Work Complexity
KMP	$O(n) + O(m)$	$O(n)$
Brute Force	$O(nm)$	$O(nm)$
Vishkin	$O(\log(m))$	$O(n)$

Performance Benchmarks

SCENARIO.BENCHMARKSPEC.METHODNAME	SCENARIO.BENCHMARKSPEC.PARAMETERS.SIZE	BYTES (B)	RUNTIME (NS)	OBJECTS
bruteForce	10	192,678.667	1,446,020.880	4,260.667
bruteForce	20	382,702.423	3,336,104.063	8,452.099
bruteForce	5	90,252.757	822,072.550	1,991.892
matcher	10	77,415.543	64,281.544	1,873.743
matcher	20	136,878.871	89,913.514	3,323.012
matcher	5	48,528.000	56,101.168	1,167.000
HIDDEN DIMENSIONS (16)				
INVARIANTS (738)				

https://microbenchmarks.appspot.com/runs/f2436060-a649-4808-bf37-1861cd5db2ec#r:scenario.benchmarkSpec.methodName,scenario.benchmarkSpec.parameters.size

Performance Benchmarks

n = 10000

m = 151

SCENARIO.BENCHMARKSPEC.PARAMETERS.NUMTHREADS	BYTES (B)	RUNTIME (NS)	OBJECTS
1	685,848.000	660,118.755	16,863.000
2	686,281.200	379,642.863	16,871.550
3	687,061.333	324,205.749	16,887.000
4	688,118.197	340,587.847	16,908.437
5	687,809.569	377,352.852	16,902.314
6	689,109.765	379,975.723	16,928.368
7	688,529.674	348,549.896	16,916.442
8	688,810.963	292,815.429	16,922.074
HIDDEN DIMENSIONS (16)			
INVARIANTS (740)			

https://microbenchmarks.appspot.com/runs/aa0b3fdd-a6d2-4499-b19f-349b0f8c2ed3#r:scenario.benchmarkSpec.parameters.numThreads

Performance Benchmarks

n = 10000

m = 8

SCENARIO.BENCHMARKSPEC.PARAMETERS.NUMTHREADS	BYTES (B)	RUNTIME (NS)	OBJECTS
1	3,396,600.000	2,139,471.651	82,895.000
2	3,396,673.831	1,326,520.252	82,896.467
3	3,414,367.349	1,137,894.424	83,260.733
4	3,417,961.158	1,081,987.138	83,330.355
5	3,421,046.202	1,096,109.458	83,393.292
6	3,419,909.286	958,589.058	83,371.696
7	3,415,502.680	966,934.155	83,280.206
8	3,408,471.040	1,188,574.650	83,136.960
HIDDEN DIMENSIONS (16)			
INVARIANTS (740)			

https://microbenchmarks.appspot.com/runs/b180348d-2ebd-42a5-92b1-3b8824e8afeb#r:scenario.benchmarkSpec.parameters.numThreads

Future Work

- Periodic Case
- Further Stream investigation

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.