*Project Report*

On

# Parallel String Matching Problem

(SMP)

Submitted by

PRANSHU MAHESHWARI (170001035)

PRAYAG JAIN (170001037)

Computer Science and Engineering

3rd year

Under the Guidance of

Dr. Surya Prakash


Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2019

# Index

# 1. Introduction

In computer science, string-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. With parallel string we try to reduce the time complexity of searching a pattern in a given string. We try to reduce complexity of already existing algorithms then finally we come up with our own algorithm.

- **The Problem Statement**

The *string matching problem* (SMP) consists of finding substring (generally pattern) P in text T. In the basic form both P and T consist of characters in the same alphabet Σ.

- **Applications**

Common applications of string matching include spell checking, record linkage. With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application. String matching is also used in spam filtering.

# 2. Algorithm Analysis

## 2.1 Naïve string-matching algorithm (Brute-Force)

### 2.1.1 Sequential Naïve string-matching algorithm

There are many approaches to solve SMP. The simplest way to find a substring in the given text is a brute force algorithm. It compares all characters of the text with the beginning of the substring. If the characters match, it compares the next characters in the text. The brute force algorithm does not need any additional preparations or space but its time complexity is O(n*m) in the worst case and O(n) in the average case, where n is the size of the text and m is the size of the substring.

### 2.1.2 Parallel Naïve string-matching algorithm

To parallelize the Naïve string-matching algorithm we divide the text T in *NUM_THREADS* parts and each processor is given a part of the text of size *T/NUM_THREADS + M - 1* and the processor performs the searching using the sequential brute force algorithm. This reduces the complexity from O(n*m) to O(n*m / NUM_THREADS).

## Code for Parallel Naïve string-matching algorithm

```
int partSize = n / NUM_THREADS;
int remainder = n % NUM_THREADS;

double start = omp_get_wtime();

#pragma omp parallel num_threads(NUM_THREADS) private(tid,start,end) shared(t,a,remainder,partSize,m)
{
    tid=omp_get_thread_num();
    long long j;

    if(tid == 0) {
        start = tid;
        end = partSize - 1;
    } else {
        start = (tid * partSize) - m;
        end = (tid * partSize) + partSize - 1;
    }
    for(long long i=start;i<=end-m;i++) {
        for(j=0;j<m;j++)
            if(t[i+j] != a[j])
                break;
        if(j == m) {
            #pragma omp critical
                v.push_back(i);
        }
    }
}
if(remainder != 0) {
    long long j;
    start = (NUM_THREADS+1) * partSize;
    end = n;
    for(long long i=start;i<=end-m;i++) {
        for(j=0;j<m;j++)
            if(t[i+j] != a[j])
                break;
        if(j == m) {
            #pragma omp critical
                v.push_back(i);
        }
    }
}
```

## 2.2 KMP (Knuth Morris Pratt) Pattern Searching

KMP algorithm is based on finite automaton. The states in the algorithm are marked with symbols that should match at the moment. There are two transitions from each state. One of them corresponds to the matching of the characters, while the other one corresponds to the mismatching. The automaton goes to the next state, if the comparison issuccessful, otherwise it goes to the previous state. Prefix-function is used for construction of the KMP automaton. This function is associated with the pattern P and gives the information about the positions of the different prefixes of the string in the patternP. Prefix-function $\pi:\{1,2,\ldots,m\}\rightarrow\{0,1,\ldots,m-1\}$that is associated with the pattern P[1. . . m] is defined as $\pi[q] = \max\{k:k < q\&PkPq\}$. In other words, $\pi[q]$ is the length of the longest prefix P that is the suffix of Pq. The time complexity is O(n+m).

KMP can also parallelized by dividing the text T into *NUM_THREADS* parts of size *T/NUM_THREADS + M - 1* and each processor performing sequential KMP on each part respectively.

**Code for KMP string-matching algorithm**

```c
void computeLPSArray(char* pat, int M, int* lps);

void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int lps[M];

    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        else if (i < N && pat[j] != txt[i]) {

            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

void computeLPSArray(char* pat, int M, int* lps)
{
    int len = 0;

    lps[0] = 0;

    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0) {
                len = lps[len - 1];


            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

## 2.3 Shift-AND string-matching algorithm

### 2.3.1 Sequential Shift-AND string-matching algorithm

In a general case we need to find all occurrences of pattern P in text T. Let us generalize the problem and assume that we should to find occurrences of all possible prefixes of pattern P: P(1) =p1, P(2) =p1p2, . . . , P(m) =p1p2. . . pm, where pi is the i-th symbol of pattern P. This method of string matching is known as Shift AND algorithm. The time complexity of the algorithm is O(m*n).

**Code for Shift-AND string-matching algorithm**

```
int n = t.size(), m = a.size();
int D[m][n];
vector<int> index;

double start = omp_get_wtime();

for(int i=0;i<m;i++) D[i][0] = 0;
for(int i=0;i<n;i++) D[0][i] = (t[i] == a[0]) ? 1 : 0;

for(int i=1;i<m;i++) {
    for(int l=1;l<n;l++) {
        D[i][l] = (a[i] == t[l] && D[i-1][l-1]) ? 1 : 0;
    }
}

double end = omp_get_wtime();

for(int l=0;l<n;l++) {
    if(D[m-1][l])
        index.push_back(l);
}

for(auto it : index) {
    cout<<it<<" ";
}
cout<<"\n";
```

### 2.3.2 Parallel Shift-AND string-matching algorithm

We can parallelize the Shift AND algorithm by parallelizing the inner loop for each row and thus the time complexity can be reduced to O(m) time with O(n) processors on CREW PRAM model.

**Code for Parallel Shift-AND string-matching algorithm**

```
int n = t.size(), m = a.size();
int D[m][n];
vector<int> index;

double start = omp_get_wtime();

#pragma omp parallel for
for(int i=0;i<m;i++) {
    D[i][0] = 0;
}

#pragma omp parallel for
for(int i=0;i<n;i++) {
    D[0][i] = (t[i] == a[0]) ? 1 : 0;
}

for(int i=1;i<m;i++) {
    #pragma omp parallel for shared(a[i])
    for(int l=i;l<n;l++) {
        D[i][l] = (a[i] == t[l] && D[i-1][l-1]) ? 1 : 0;
    }
}

double end = omp_get_wtime();

for(int l=0;l<n;l++) {
    if(D[m-1][l])
        index.push_back(i);
}


for(auto it : index) {
    cout<<it<<" ";
}
cout<<"\n";
```

## 2.4 Dynamic Programming Based Solution

### 2.4.1 Sequential Solution

A matrix D[m+1][n+1] is constructed, where $D_{i,j}$ is the minimum number of differences between $a_1,a_2,...,ai$ and any contiguous substring of the text ending at $t_j$. If $D_{l,m} <= k$ then there must be an occurrence of the pattern in the text with at most k differences that ends at $t_m$. Time complexity of this algorithm O(n*m). The last row of D matrix gives the number of mismatches for pattern ending at that index. Thus it can be used to perform fuzzy(approximate) search as well.

**Code for DP based string-matching algorithm**

```
15
16      int n = t.size(), m = a.size();
17      int D[m+1][n+1];
18      vector<int> v;
19
20      double start = omp_get_wtime();
21
22      for(int i=0;i<=n;i++) D[0][i] = 0;
23      for(int i=0;i<=m;i++) D[i][0] = i;
24
25      for(int i=1;i<=m;i++) {
26          for(int l=1;l<=n;l++) {
27              int temp = (a[i-1] == t[l-1]) ? D[i-1][l-1] : D[i-1][l-1] + 1;
28              D[i][l] = min({ D[i-1][l] + 1, D[i][l-1] + 1, temp });
29          }
30      }
31
32      double end = omp_get_wtime();
33
34      for(int l=0;l<=n;l++) {
35          if(!D[m][l]) {
36              v.push_back(l);
37          }
38      }
39
```

### 2.4.2 Parallel Solution

This solution can be parallelized by running the n*m for-loop twice parallely, once row wise and the second time column wise. The time complexity of this is O(n+m) and it requires O(max(n, m)) processors on a CREW PRAM model.

**Code for Parallel DP based string-matching algorithm**

```
#pragma omp parallel for
for(int i=0;i<=n;i++) {
    D[0][i] = 0;
}

#pragma omp parallel for
for(int i=0;i<=m;i++) {
    D[i][0] = i;
}

for(int i=1;i<=m;i++) {
    #pragma omp parallel for shared(a[i-1])
    for(int l=1;l<=n;l++) {
        int temp = (a[i-1] == t[l-1]) ? D[i-1][l-1] : D[i-1][l-1] + 1;
        D[i][l] = min({ D[i-1][l] + 1, temp });
    }
}

for(int l=1;l<=n;l++) {
    #pragma omp parallel for shared(a[i-1])
    for(int i=1;i<=m;i++) {
        int temp = (a[i-1] == t[l-1]) ? D[i-1][l-1] : D[i-1][l-1] + 1;
        D[i][l] = min({ D[i][l-1] + 1, D[i][l], temp });
    }
}

double end = omp_get_wtime();

#pragma omp parallel for
for(int i=m;i<=n;i++) {
    if(!D[m][i]) {
        #pragma omp critical
            index.push_back(i);
    }
}
```

## 2.5 Our CREW PRAM Solution

We allocate m processors to each character in the text T and these m processors check if the pattern starts from that character or not. It can also be used to perform fuzzy searches in the text.
The time complexity is O(lg(m)) with O(n*m) processors on a CREW PRAM model.

**Code for our Parallel CREW string-matching algorithm**

```
int n = t.size(), m = a.size();
vector<vector<int> > check(m, vector<int>(n, 0));
int SUM[n];

double start = omp_get_wtime();

#pragma omp parallel for
for(long long i=0;i<n*m;i++) {
    int id = i / m;
    int j = i % m;
    if(a[j] == t[id+j]) {
        check[j][id] = 1;
    }
}

#pragma omp parallel for
for(int i=0;i<n;i++) {
    SUM[i] = 0;
    #pragma omp parallel for reduction(+:SUM[i])
    for(int j=0;j<m;j++) {
        SUM[i] += check[j][i];
    }
}

#pragma omp parallel for
for(int i=0;i<n;i++) {
    if(SUM[i] == m) {
        #pragma omp critical
            v.push_back(i);
    }
}

double end = omp_get_wtime();

for(auto it : v) {
    cout<<it<<" ";
}
cout<<"\n";
```

## 2.6 Our CRCW PRAM Solution

We allocate m processors to each character in the text T and these m processors compares the next m characters with the pattern. If a mismatch is found the processor writes 0 to the check[i] variable.
The time complexity is O(1) with O(n*m) processors on a Common CRCW PRAM model.

**Code for our Parallel CREW string-matching algorithm**

```
int n = t.size(), m = a.size();
vector<int> check(n, 1);

double start = omp_get_wtime();

#pragma omp parallel for
for(long long i=0;i<n*m;i++) {
    int id = i / m;
    int j = i % m;
    if(a[j] != t[id+j]) {
        check[id] = 0;
    }
}

#pragma omp parallel for
for(int i=0;i<n;i++) {
    if(check[i]) {
        #pragma omp critical
            v.push_back(i);
    }
}

double end = omp_get_wtime();

for(auto it : v) {
    cout<<it<<" ";
}
cout<<"\n";
```

# Results

| Algorithm | Time Complexity | # Processors |
|---|---|---|
| **Sequential Naïve** | O(n*m) | O(1) |
| **Parallel Naïve** | O(n*m/NUM_THREADS) | O(NUM_THREADS) |
| **KMP** | O(n+m) | O(1) |
| **Parallel KMP** | O((n+m)/NUM_THREADS) | O(NUM_THREADS) |
| **Shift And** | O(n*m) | O(1) |
| **Parallel Shift And** | O(m) | O(n) |
| **DP Solution** | O(n*m) | O(1) |
| **Parallel DP** | O(n+m) | O(n) |
| **Our CREW Solution** | O(lg(m)) | O(n*m) |
| **Our CRCW Solution** | O(1) | O(n*m) |

## The timing analysis of the algorithms

```
Sequential Brute Force Algorithm
Text size: 10000000
Pattern size: 7
Time: 0.0573285
```

```
Parallel Brute Force Algorithm
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
Time: 0.0156179
```

```
Our CRCW Algorithm:
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
Time: 0.424392
```

```
Our CREW Algorithm
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
Time: 0.030455
```

```
DP Based Algorithm
Text size: 10000000
Pattern size: 7
Time: 0.0261375
```

```
Parallel DP Based Algorithm
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
Time: 0.11827
```

```
Sequential Shift AND Algorithm
Text size: 10000000
Pattern size: 7
Time: 0.425879
```

```
Parallel Shift AND Algorithm
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
Time: 0.133095
```

```
Sequential KMP Algorithm:
Text size: 10000000
Pattern size: 7
Time: 0.097149
```

```
Parallel KMP Algorithm:
Text size: 10000000
Pattern size: 7
NUM_THREADS: 4
0.067735
```

## Project Link: Github

## References

[1] Bit parallel string matching, Alina Gutnova

June 21, 2006 Available: Bit parallel string matching - Semantic Scholar.

[2] General methods of sequence comparison

https://dornsife.usc.edu/assets/sites/516/docs/papers/msw_papers/msw-054.pdf

[3] https://www.wikipedia.org/

[4] https://cs.stackexchange.com/