Parallelized Rabin-Karp Method for Exact String Matching

Predrag Brođanac[1], Leo Budin[2], and Domagoj Jakobović[3]
[1]*V. High School, Zagreb, Croatia*
[2,3]*University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia*
E-mail(s): predrag.brodjanac1@zg.t-com.hr, leo.budin@fer.hr, domagoj.jakobovic@fer.hr

Abstract. *Exact string matching refers to the search of each and any occurrences of a string in another string. Nowadays, this issue presents itself in various segments in a great deal, starting from standard routines for exact search, which routines are implemented into programs for text editing and processing, through databases and all the way to their various applications in other sciences. One of the sciences where, among other, this kind of search has been applied on a substantial level is biology, and especially in the segment concerning DNA chains. There are numerous different more or less efficient algorithms to solution of this problem. One of more efficient algorithms is Rabin-Karp algorithm, whose complexity is linear. This work provides us with one way to parallelize this algorithm for performance on multiprocessor systems.*

Keywords. Multiprocessing, string matching, Rabin-Karp, Python.

1. Introduction

Today technology arrives as culprit for lack of space to accelerate the work of processors. Space for acceleration exists in multiprocessor (multi-core) computers, where processes (cores) communicate directly by a shared memory [1]. However, in order for the possibilities of multiprocessor computers to be used actually, suitable software is required. Such software ought to be written in such a manner that parts of the software can, more or less, be executed independently, and consequently each on its own processor.

Currently there exists a whole sequence of programming languages enabling a certain form of parallel programming. Certain programming languages have gone even one step further and have developed program structures, which themselves will parallelize a part of the code program without its programmer having to know the method how it had been done. This work shall consider parallelizing on a slightly lower, but not the lowest, of levels (of the processor cores themselves). To illustrate we will be using the Python programming language and one of its modules – the multiprocessing. We will parallelize Rabin-Karp algorithm for exact strings matching with linear complexity.

2. Exact String Matching

Exact string matching boils down to search of any occurrence of a string *A* in string *B*. School ("naïve") way for a search such as this one comes down to comparison of all the symbols of string *A* with the first *len(A)* symbols of the string *B*, whereby *len(A)*is the number of symbols of string *A*. Provided that the first *len(A)* symbols of string *B* and all the symbols of string *A* match, then one can talk about one occurrence of string *A* in string *B*. Further, the same comparison applies to *len(A)* symbols of string *B* starting from the second symbol. This procedure is repeated until the end of string *B*.

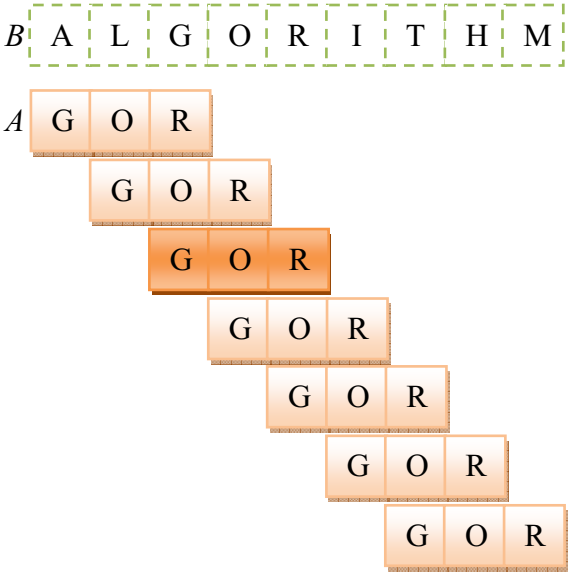

Figure 1: "Naive" string matching

This manner is the slowest one and its complexity is $\Theta(m(n - m + 1))$, whereby $n = len(B)$, $m = len(A)$.

Implementation of this algorithm which enters all the results into the global array is:

```
for i in range(0, len(A) -
        len(B)):
    match = True
    for j in range(0, len(B)):
        if B[j] != A[i + j]:
            match = False
    if match:
        R.append(i)
```

3. Rabin-Karp Algorithm

Along with this manner, there exist multitudes of more or less efficient algorithms. One of the more efficient ones, whose complexity is $\Theta(m) + \Theta(m(n - m + 1))$ but whose medium operation time is better by far is Rabin Karp algorithm. Basic principle of this algorithm resembles greatly the algorithm described above. The only difference is that we will not be checking all the symbols of B string each time, starting from some letter **s**, as we would have done with the "naive" algorithm. The principal idea is that a string converts into a number and that comparison of strings leads down to comparison of two numbers, which occurs in a considerably shorter period of time than the comparison of two strings does. Furthermore, the upper side of this method is that when shifting, one does not have to compare strings letter by letter once more, but will compare values of the numbers, and new number from the previous number we will acquire by rejecting the first digit and adding a new digit at the end.

Suppose we have a default string A whose length ($len(A)$) is equal to m. We mark that string by $A[0..m - 1]$ and let p be the numerical value of the string A. Calculation of numerical value should be bijective (further on we will explain one of the possible manners of its realization). Further, let us have a default string $B[0..n - 1]$ and let t_s be numerical value of a part of string B, which has m symbols and starts on place **s** in the string $B(B[s..s + m - 1])$, for $s = 0, 1,...n - m$. Clearly, in this case strings $A[0..m - 1]$ and $B[s..s + m - 1]$ shall be equal if and only if $p = t_s$. Provided that the values of number p can be determined in $\Theta(m)$, and all the values of t_s for $s = 0, 1,...n - m$ in $\Theta(n – m + 1)$ period of time, the following demonstrates that complexity of this algorithm equals $\Theta(m) + \Theta(n – m + 1) = \Theta(n)$, which is considerably faster in comparison to $\Theta(m * (n - m))$, which was the complexity of the "naive search"[5].

Further on, we shall suppose that A and B strings comprise of capital letters of the English alphabet, with total number of 26. Further on, let us have defined $N(c)$ function, which shall retrieve the numerical value of letters in the following manner: $N('A') = 0$, $N('B') = 1$... $N('Z') = 25$. In this case the p value will be calculated by Horner algorithm:

$$p = N(A[m - 1]) + 26 * (N(A[m - 2]) + 26 * (N(A[m - 3]) + ... + 26 * (N(A[1]) + 26 *(N(P[0]))...). \quad (1)$$

Analogously,

$$t_s = N(B[s+ m - 1]) + 26*(N(B[s + m - 2]) + ... + 26*(N(B[s + 1]) + 26*(N(B[s]))...). \quad (2)$$

The (2) makes it clear that

$$t_{s+1} = 26*(ts – 26^{m-1}N(B[s])) + N(B[s + m]). \quad (3)$$

For example, if there is default string $B = 'BCDEF'$ and given that $m = 4$, in this case the following is: $N('B') = 1$, $N('C') = 2$, $N('D') = 3$, $N('E') = 4$ i $N('F') = 5$. Further:
$t_0 = N('E') + 26 * (N('D') + 26 * (N('C') + 26 * N('B')))$, that is:
$t_0 = 4 + 26*(3 + 26*(2 + 26(1))) = 19010$.

According to the above formula it will be:
$t_1 = 26(t_0 – 17576 * 1) + 5 = 37289$. On the other hand, numerical value of $'CDEF'$ string equals: $5 + 26(4 + 26(3 + 26(2))) = 37289$. The aforementioned shows that the $'CDEF'$ string has been acquired from $'BCDE'$ string in constant time.

Problem of this algorithm are sizes of the numbers p and t_s. These numbers clearly depend upon on m and numbers of symbols in the alphabet. However, for the requirements of a "rough" comparison, it will suffice to observe values p MOD q and t_s MOD q as well, where q is a prime number. If p MOD $q \neq t_s$ MOD q, then we can say for sure that strings $A[0..m - 1]$ and $B[s..s + m - 1]$ do not match. Whereas, if p MOD $q = t_s$ MOD q, then we cannot say with certainty that the strings are a match, and an additional cross check is required. This additional cross check is nothing else but the "naive" comparison of strings. For q it is best to take such a number that

$D * q$ is the biggest possible number that can be saved into the register, whose length equals the length of computer word(s), where D equals the number of alphabet symbols over which alphabet words A and B have been defined.

In general, for an arbitrarily large alphabet (D letters) and an appropriate function N, formula (3) carries the following outline:

$$t_{s+1} = (D*(ts - (D^{m-1} \ MOD \ q) * N(B[s])) + N(B[s+m])) \ MOD \ q \quad (4)$$

4. Realization of the Rabin-Karp Algorithm in Python Programming Language

Our focus will be an alphabet consisting of 4 symbols: *A*, *C*, *T*, *G*. We will create a method that will verify whether *B[n .. n + m - 1]* equals *A*, and this will be performed by the "naive" algorithm:

```
def equal(n):
    for i in range (0, m):
        if B[n + i] != A[i]:
            return False
    return True
```

The following method retrieves the numerical value of the symbol in the following way: for the letter *A* it retrieves *0*, for *C* it retrieves *1*, for *G* it retrieves *2*, and for *T* it retrieves *3*:

```
def N(c):
    tmp = {'A' : 0, 'C' : 1, 'G'
     : 2, 'T' : 3}
    return tmp[c]
```

The following method retrieves the numerical value of a string:

```
def strToNum (s):
    t = 0
    for i in range (0, len (s)):
       t = (t * D + N(s[i])) % q
    return t
```

Finally, the method representing the realization of the algorithm retrieves all the positions of string *A* into string *B*:

```
def allPositions ():
    _k = strToNum (B[:m])
    if _k == p:
        if equal(0):
            R.append (0)
    for i in range (0, n - m):
        _k = (D * (_k - N(B[i])
         * H) + N(B[i + m])) % q
        if (_k == p):
            if (equal (i + 1)):
                R.append (i + 1)
```

The initial conditions of the method above are as follows:

```
R = []
q = 1073741789
D = 4
m = len (A)
H = D ** (m - 1) % q
p = strToNum (A)
```

A and *B* are arbitrary strings consisting solely of symbols: *A*, *C*, *T* and *G*.

5. Multiprocessor Programming

Current progress in the area of processor acceleration is based upon improvement of parallel architectures, and not any longer on processor operation acceleration. Momentary challenge in the modern IT science is to parallelize standard sequential routines that use in programs [1]. The starting point here will be multiprocessor systems with shared memory, i.e. multi-core systems.

Standard sequential programming does not take much account of "background" actions on the level of the memory itself. First, whole program modules are copied from the disk into the central memory, and then smaller amounts of data are transferred into processor cache memory. This kind of complex organization of memory for a programmer remains hidden in sequential programming. Parallel programming often requires from a programmer to stoop to a lower level [1].

Regarding the problems considered herein, on account of the fact that the data have been read solely from the shared memory, we will not be required to descend onto a lower level, therefore it will not be a point of our discussion herein.

5.1. Multiprocessing Package

Nowadays there are multitudes of programming languages enabling us to perform parallel programming: Java, C#, C++, Python, ... To illustrate the solution of our problem, we will be using the Python. *Multiprocessing* module has been developed in the Python for the purpose of parallel programming. Besides the multiprocessing programming, Python provides multithreading programming, as well. In the presented application, where no intensive communication is needed, the choice of both techniques may be justified. However the multiprocessing model has been chosen due to the fact that in practice it could be performed on several computers as well, and not solely on a single multi-processor computer.

Since the proposed algorithm is ideally parallel (the communication is negligible), the same model could be deployed on multiple machines, especially if the search string is very large.

Multiprocessing module contains routines enabling a method to be initiated and performed at the same time. Module with routines for parallel programming inside the Python we will initiate with the following instruction:

```
from multiprocessing import *
```

This module contains *Process* class, whose constructor has a number of parameters:

```
Process(target = method_name,
  args = (arg1, arg2,...))
```

This method shall create a new process, that will perform the *method_name* method, and the values transferred to the method have been set by values *arg1*, *arg2*,... We will initiate the process by calling the *start()*and *join()* method.

6. Rabin-Karp Algorithm Parallelization

The principal idea of parallelizing the Rabin-Karp Algorithm is to split string *B* into *K* parts, and to start an individual process for each of the parts, where individual processes will search for occurrences of string *A* in that part of string *B***.** This requirement will make us modify the *allPositions* method a little, which method, for the part of string from *x* spot to *y* spot in the string *B***,** requires occurrence of the string *A***:**

```
def allPositions(x, y, R):
    tmp = ''
    for i in range(x, x + m):
        tmp = tmp + B[i]
    _k = strToNum(tmp)
    if _k == p:
        if equal(x):
            R.put(x)
    for i in range(x, y - m):
        _k = (D * (_k - N(B[i])
         * H) + N(B[i + m])) % q
        if (_k == p):
            if (equal(i + 1)):
                R.put(i + 1)
```

Further on, we encounter a problem of dividing string *B* into *K* equal parts. Suppose we have a default string of *n* symbols length, and we are searching for each and every occurrence of string with *m* symbols length, all of which parallel as the *K* process. For example, if the first process is processing *B[0 .. n / K - 1]* part of the string, and the second process is processing *B[n / K .. 2 * (n / K) - 1]* part of the string, in that case neither of the processes would be processing *B[n / K – m + 1 .. n / K + 1]* substring, for example. However, should we assert that the first process verifies all those substrings starting in positions 0, *1*, … *n / K*, then this process, for example, in order to verify a substring starting in place *n / K***,** will be processing *m – 1* symbols from the area of the second process. This manner provides certainly that all viable substrings will be verified. Therefore, the first process will be processing substring *B[0 .. n / K + m- 1]*, the second process will be processing *B[n / K .. 2 * n / K + m - 1]*, whereas the final process will be processing substring *B[(K - 1) * n / K .. K * n / K + m - 1]*, which approximately equals: *B[(K - 1) * n / K .. n + m - 1]***.** Evidently, this is not "fair". Namely, the second parameter of the substring, which is processed by the final process, most often supersedes the string length. This situation could have happened even in some substrings, which are processed by certain previous processes. We could bypass it by saying that if the last symbol of a substring (*B[(i - 1) * n / K .. i * n / K - 1]*) supersedes *n* , then we will be observing substring *[(i - 1) * n / K .. n - 1]***.** However, if *m* is a large number considering the number *n***,** then the number of operations performed by a certain process shall be unequal. For example, lets say that *n = 20*, *m = 10* and *K = 3*. According to the above considerations, the first process will be processing a part of string *B[0 .. 16]*, the second process will be processing

a part of string *B[7 .. 20]*, whereas the third process will be processing substring *B[14 .. 20]*. Even a small example such as this shows us that the first process will have to perform the amount of work far greater than the third process. Solution to this problem lies in the fact that *i*-th (*i = 0, ..., K - 2*) process is processing a part of string *B[round(i * (n – m + 1) / K).. round((i + 1) * (n – m + 1) / K) + m - 1]*, and that the final process is processing substring *B [round((K – 1) * (n – m + 1) / K).. n]*.

Therefore, the *allPositions* method would be started as *K* parallel processes in the following manner:

```
for i in range(K - 1):
   processes.append ( Process (
     target = allPositions, args
     = (round(i * d), round((i +
     1)* d) + m - 1, R, )))
processes.append(Process(target
  = allPositions, args =
  (round(i * d), n, R, )))
for pr in processes:
    pr.start ()
for pr in processes:
    pr.join ()
```

thus:

```
d = (n – m + 1) / K.
```

whereas *K* is an arbitrary number.

7. Experimental Results

For the purpose of testing of this algorithm, we will take a part of DNA chain from a fish of Zebrasome flavences family of sea fish. To compare the speed, testing was performed by "naive algorithm", serial Rabin-Karp algorithm and parallel Rabin-Karp algorithm all of which on a 4GB RAM memory computer and 4 core processor. The results for different sizes of search strings are displayed in the following table:

n	m	"Naive"	Serial Rabin-Karp	Parallel Rabin-Karp (K = 4)
10000	10	0,043s	0,026s	0,088s
	3000	7,276s	0,026s	0,097s
3479998	10	13,15s	3,553s	1,182s
	3000	3556s	3,888s	1,313s
	10000	12033s	3,936s	1,351s
6970438	10	27,81s	7,114s	2,286s
	3000	7042s	7,757s	2,520s
	10000	34458s	8,026s	2,598s
60258128	10	too long	81,87s	24,74s
	3000	too long	87,38s	27,25s
	10000	too long	89,13s	27,75s

Table 1: Comparison of speed of performance of the exact string search by different algorithms

Figure 1 shows the execution times of both variants of Rabin-Karp's algorithm, whereas the achieved speedup on 4 processors is shown in Figure 2.

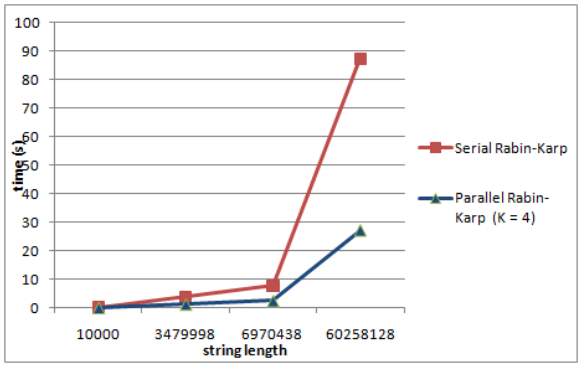

Figure 2: Comparison of serial and parallel Rabin-Karp algorithm for m = 3000

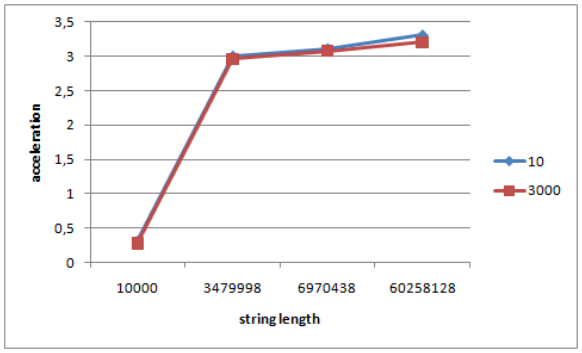

Figure 3: Acceleration of Parallel Version of the R-K Algorithm on 4 Processors

The results clearly illustrate what we had asserted at the beginning, that the speed of performance of the Rabin-Karp algorithm depends on the length of string *B*, in which we are searching for the occurrence of string *A*. Further, it is evident that for large strings the parallelized algorithm is over 3 times faster than the serial Rabin-Karp algorithm. For shorter strings, which require less than a second to perform the algorithm, the parallelization is not even required, and the small difference in speed of the parallel algorithm, compared to the serial one, is due to time required for the creation of a number of processes. The described results of relation between the serial and parallel algorithm have turned out as expected. Namely, the processes are independent amongst each other, and each and every process has received an equally large string.

8. Conclusion

As stated at the beginning, the problem of fining each and every occurrence of a string in another string is very frequent in practice. Modern computers are mainly multiprocessor computers. Table 1 shows that the accelerations acquired are important, and that parallelized version of this algorithm would accelerate greatly applications which use serial version of this algorithm. Certainly this is not the only parallelization of this algorithm, but is very simple and intuitive one because there is no need to take account of process synchronization due to the fact that the processes only read the data from the shared memory. Probably in the similar way certain algorithms from this area could be parallelized as well, which may have been more in use that the Rabin-Karp algorithm itself. Furthermore, this problem might be extended into the area of searching for "similar" substrings, which is especially applicable in bioinformatics. Smith-Waterman algorithm for local alignment of strings, which has been created for the purpose of discovering similarities between couples of DNA chains, leads down to dynamic programming and its complexity is $\Theta(n * m)$[2, 4]. Acceleration of this algorithm would be of great benefit to the searching of DNA data base.

9. References

[1] Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publisher, 2008

[2] M. C. Schatz, C. Trapnell, Fast Exact String Matching on the GPU, Technical report

[3] S. Viswanadha Raju, A. Vinayababu, Optimal Parallel algorithm for String Matching on Mesh Network Structure, Intenrational Journal of Applied Mathematical Sciences, Vol.3 No.2(2006), pp. 167-175

[4] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, J Mol Biol, vol. 147, pp. 195-197, 1981

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2005