Project Report: Comparison of Parallel KMP vs Parallel Rabin Karp on Heterogenous Architecture

Shivam Jaitly, Shivani Bhoite
CSC 8530: Parallel Algorithms
Department of Computer Science, Georgia State University
Spring, 2020

Abstract- String matching algorithm is widely used in many application areas such as bioinformatics, network intrusion detection, computer virus scan, among many others. KMP (Knuth-Morris-Pratt) algorithm is commonly used for its fast execution time compared with many other string-matching algorithms when applied to different input texts. In our project, we will compare a high performance parallel KMP algorithm on the Heterogeneous High-Performance Computing (HPC) architecture based on the general-purpose multicore microprocessor and the Graphic Processing Unit (GPU) to another popular string-matching algorithm which is parallel Rabin Karp. We will try and compare the performance of the both the algorithm on different types of input data of varying sizes. The implementation of parallel algorithms mainly focuses on optimizing the CPU-GPU memory hierarchy by optimizing the data transfer between the CPU memory and the GPU memory with the string-matching operations on the GPU. After implementation of the parallel KMP and parallel Rabin-Karp we will compare the results of both the algorithms and thus conclude our results. The experimental results show that the optimized parallel KMP algorithm leads up to ~5 times faster execution time

***Keywords*— parallel KMP, GPU, CUDA, parallel Rabin- Karp**

I. INTRODUCTION

String matching algorithm are commonly used in various applications such as the keyword matching for the given input text, the intrusion detection in the network system, among many others [1]. The lack of space to accelerate the work of processors in one of the major hindrances in multiprocessor architecture. The main aim of the of the program to be executed on the multiprocessor system is that each part of the software is independent of the other part so that they can be executed independently on different processors to give the real meaning of parallel architecture.

The exact string-matching problem consist of the following. We have a given pattern P in the string we want to search which has length of say m and given set data that is text T of length say n and we assume that it valid when the equality holds (n>=m). The problem constitutes to finding all the occurrence of the of Pattern P in the text T. The addition to this task is the task that we have to find the pattern P in the text T and count the number of times that pattern exists in the given string or to get all the indices of the points where the pattern in the string starts and returns the starting index of the pattern in the text T.

Since this is very popular problem and is used for different tasks across various domains. So, there are many algorithms present in the world for solving this problem in both sequential and parallel domain.[2] We choose two of the most popular and efficient algorithms to solve this problem that is KMP an Rabin-Karp algorithm. We also tried to parallelize these to algorithm efficiently and focus our results after comparing the two algorithms on different size and type of data and conclude our result. We also want to compute the Speedup of the parallel KMP and parallel Rabin-Karp algorithm that we implemented. We will try to find out from experiments which is the best algorithm. We will also compute how much speedup we get from the parallel KMP or parallel Rabin Karp by comparing our results and then come to the conclusion. We will also implement and compare the KMP sequential results with the brute force approach of the string matching. By comparing the brute force approach with the sequential KMP we are trying to prove that sequential KMP performs better than the brute force approach and thus we use it further to compute the speedup of both the parallel KMP and parallel Rabin – Karp keeping the Text T and pattern P common. The real reason for comparing KMP and Rabin-Karp is that they are two best algorithms for the string-matching problem sequentially and by our experiments we tend to find that how much performance of each algorithm improves when they are parallelized. We also want to find out that whether KMP is still better than Rabin-Karp after parallelizing and uses the potential of run on parallel architecture better than its competitor.

We went ahead with this type of project because of the following reasons First, we will be implementing two most popular algorithms for string matching and gain hands on experience for our Parallel Algorithms class. Second, the string-matching problem is still very hot research area in parallel computing domain as we found many recent papers on how to improve the current algorithms in the parallel architecture to perform better. We have the opportunity to implement these algorithms and analyse the result to conclude our results.

The report is divided into five sections. Section I deals with the introduction of the topic and also discussed the significance of the parallel KMP algorithm in the recent research of the string-matching problem domain. Section II deals with the background of the Naïve string-matching algorithm, KMP sequential, sequential Rabin-Karp and the parallel KMP and parallel Rabin-Karp algorithm. This gives this insight of each algorithm and how each algorithm works for the string-matching problem. Section III discusses about the experiments and approaches we took to compare the different algorithms and discusses about the different scenarios in the experiments. Section IV covers the results of the experiment. Section V covers the conclusion of the report on which algorithm parallel KMP or parallel Rabin-Karp performs better on parallel architecture. Also, it discusses which algorithm gave better speedup in comparison to other.

II. Background

A. Naïve Algorithm

Naive pattern searching is the simplest method among other pattern searching algorithms and is also called as brute force approach for string matching.

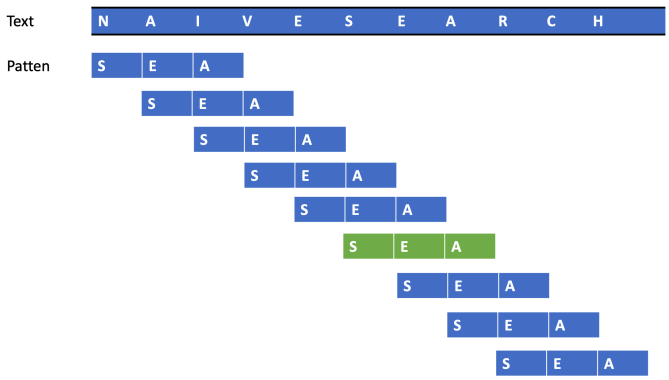


Fig. 1 Implementation of Naïve Algorithm

It searches of any occurrence of a string A in string B. A brute force way for a search such as this one comes down to a 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.

B. KMP Sequential

Knuth, Morris, and Pratt developed the KMP algorithm. This algorithm uses the prefix and the suffix of the reference pattern string in the process of pattern matching.

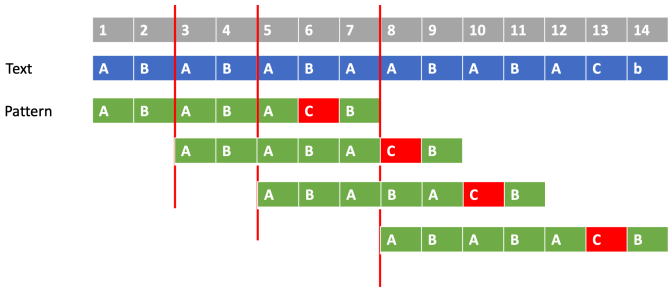


Fig. 2 Implementation of KMP Algorithm

First, a pre-processing step is conducted for the reference pattern string to generate a Failure Table. Starting with the beginning of the pattern string, we count the number of occurrences of the pattern matching with the prefix and the suffix at the current index of the string until we reach the end of the pattern string. We store these count values in the Failure Table. The pattern string and the Failure Table have the same length. Next, we conduct the pattern matching of the reference string with the input text using the Failure Table. When a character in the pattern matches with the character in the input text, the pattern match continues for the next character. When there is a mismatch, then we skip by the number of characters stored in the Failure Table. Then the whole window of the pattern string is shifted by the number to the forward direction of the input text string.

C. Rabin-Karp Sequential

Rabin-Karp algorithm uses a hash function for finding a substring from a string. The hash function determines the feature value of a particular syllable fraction. It converts each string into a number, called a hash value. The comparison of strings leads down to a comparison of two numbers, which occurs in a considerably shorter period than the comparison of two strings does. Rabin-Karp algorithm determines hash value based on the same word [3].

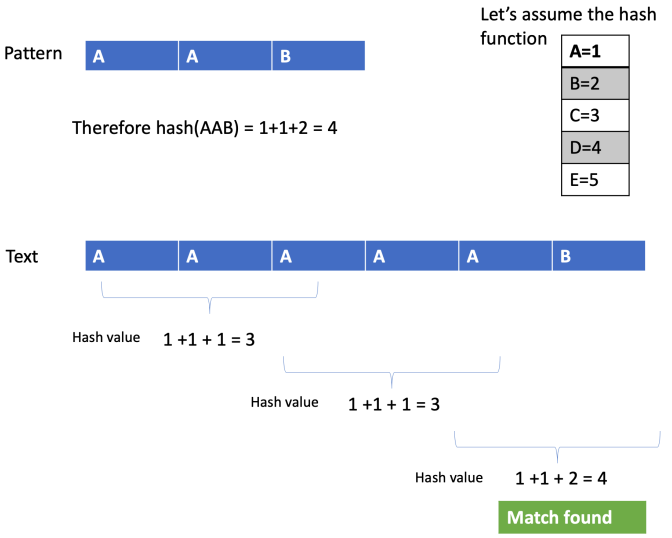


Fig. 3 Implementation of Rabin Karp Algorithm

There are two barriers to determining the hash value. First, many different strings are in a particular sentence. This problem can be solved by assigning multiple strings with the same hash value. The next problem is not necessarily the string that has the same hash value match to overcome it for each string is assigned to the brute-force technique. Rabin-Karp requires a sizeable prime number to avoid possible hash values similar to different words.[3]

D. Parallel KMP-

KMP parallelization improve the performance of the KMP algorithm on parallel architecture especially for the string with larger sizes. By using the concept of parallel algorithm, a very large text is divided into chunks which are independent of the pattern size. Then the same pattern is executed on different chunks of the text in parallel and thus helps in improving the performance of the of the sequential KMP algorithm.

Pseudo Code -

```
Declare and Definition of the index X

N ← length[Text]
M ← length[Pattern]
preKMP ← Compute the prefix function
results ← stores the index of the
occurrence of pattern.
x← blockDim.x*blockIdx.x+threadIdx.x
if x<N  then
start←x*M*chunk
stop←(x+1)*M*chunk+M-1

if stop>N then stop←N
endif

    i←0, j←start
while j<stop
while i>-1 and P[i] !=T[j]
i←preKMP[i] end while
i++ and j++
if i>=m then results j←j-i
i← preKMP[i]
endif, endwhile, endif
```

E. Parallel Rabin Karp-

In parallel Rabin-Karp Algorithm, the string into split into K parts, and individual processes is for each component is started parallelly.

If given a string B containing n elements each, the first process is processing B[0 .. n / K] part of the string, and the second process is processing B[n / K .. 2 * (n / K)] part of the string. This is how the parallel Rabin Karp algorithm is executed.

One of the problems we faced while parallelization of Rabin-Karp is to decide if the division is to be done in the length of (n/K) parts. Suppose we have a text of length n and the pattern to be searched of length m. There is a possibility that the pattern is between [0-n/K] and [K+1..2*(n/K)+1], and these substrings are running on different parallel processors. [2] After going paper, we found out that the solution is first to assert the first process verifies all those substrings starting in positions 0, 1, ... n / K. After that, this process, for example, to verify a substring starting in place n / K, will be processing m – 1 symbols from the area of the second process. This manner certainly provides 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]. We can bypass 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].

III. Experiments

For the sake of experiment, we compare the results by applying different types of data which are as follows –

- DNA sequences.
- English text.
- Numbers.

We use the Naïve pattern matching algorithm, sequential KMP, parallel KMP and parallel Rabin Karp for our experiment.

For each set of the algorithm we use the following type of text and pattern-

- The input text with various lengths which varies in length from 10^4 to 10^6 characters in length to get the performance of each algorithm and to get the better performance estimate of each algorithm. For ease of understanding we have called this length as ‘n’ in the paper.

- For the experiment purpose in each case and for every run we take into account the length of

pattern which is 4 character and less than in 10 in each case so that the uniformity of the experiment is maintained, and we can get the comparable results. For ease of understanding we have called this length as 'm' in the paper.

After taking the data and implementing both the sequential and parallel implementations of our proposed projects, we started running each of the algorithm on each set of data and start gathering the results. Also, for the implementation of the algorithm in parallel we have only used CUDA C and for the implementation of the sequential algorithm we used C language to maintain the standards of what is being taught our class. Then all the algorithms are run one by one on each of the above-mentioned inputs and then the running time of each algorithm is recorded which is presented in the following section of the report.

Text Type	n	m	Naïve	Sequential KMP	Parallel KMP	Parallel Rabin
Numbers	10^4	<10	1.5	1.3	0.4	6.0
	10^5		8.2	6.16	1.1	9.9
	10^6		80.2	56.7	10.2	53.0
Text	10^4		1.4	1.5	0.5	3.0
	10^5		6.8	5.5	1.0	27.1
	10^6		76.0	30.1	4.6	96.7
DNA	10^4		1.5	1.4	0.4	3.2
	10^5		8.0	7.9	1.1	27.0
	10^6		76.5	34.2	4.6	104.3

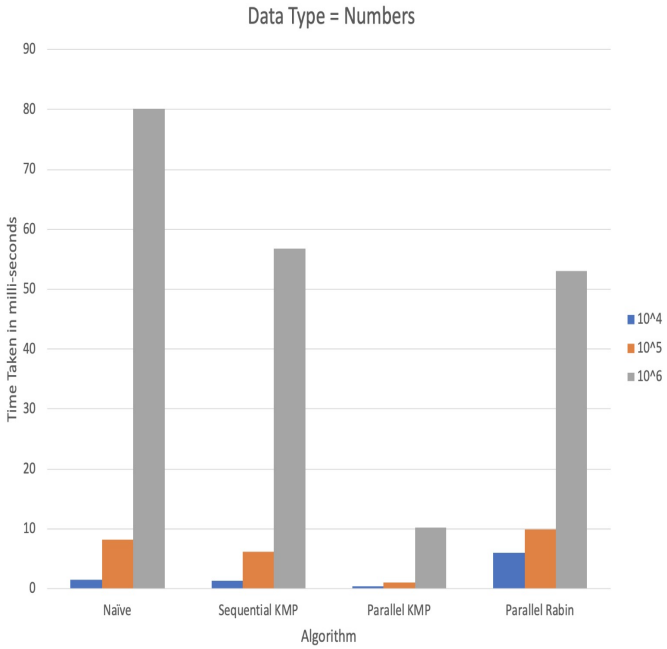
TABLE I. Experiment Results and execution time (milliseconds) of different algorithms on different input size text

The results in the Table I clearly illustrate what we have asserted before the experiment results. The speed of parallel KMP is better than the other set of algorithms on the same data in terms of running time. The sequential KMP is also one of the efficient string-matching algorithms.

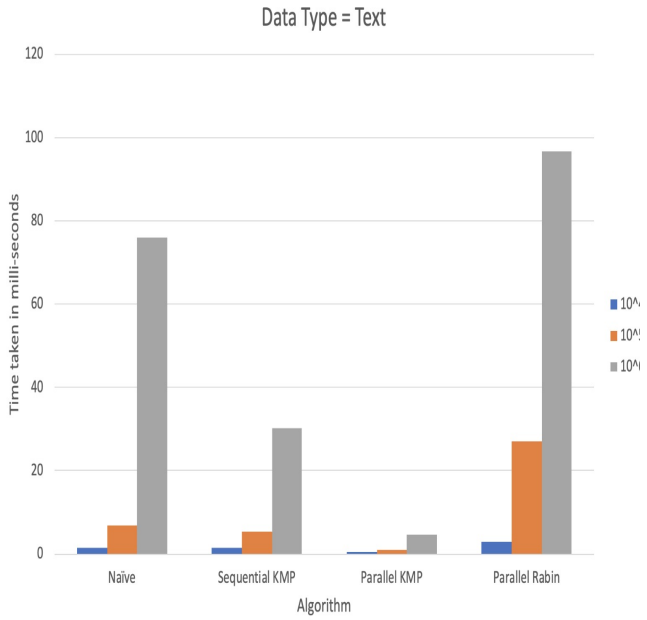
Further, it is evident from the results that parallel KMP performs well on all sizes of input text, and also on all types of data and give approximately a speedup of ~5 times than what we take as experiment to be best sequential string-matching algorithm that is sequential KMP. Also, from the results we can see that sequential KMP performs better than the naïve brute force string matching algorithm in term of running time.

But we can see parallelization of Rabin Karp algorithm does not give the satisfactory results when compared to the sequential KMP algorithm result and parallel KMP algorithm results, we feel due to inherent nature of the Rabin Karp algorithm. Performance of Rabin Karp depends the length of Text in which we are trying to find the pattern which we give. From the result it is evident that parallel Rabin Karp will perform only better if the size of text is very large beyond the values that we have taken in the experiment because Rabin Karp uses hash values and for calculating hash values it takes longer time so which will be helpful only when the size of input Text is very large.

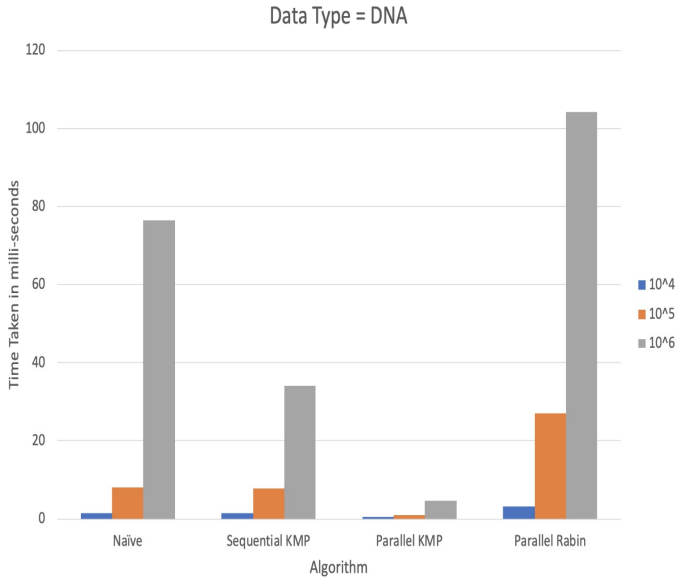
Further, results of the experiment show that parallelization KMP give efficient and expected results because it is due to the inherent nature of the KMP algorithm. Parallel KMP algorithm divide the input text into chunks to execute on different processors and take use of the parallelization by matching the string and using the KMP algorithm approach.
The result of the experiment shows that parallel KMP work with any size of the Text data and any size of the pattern that we want to search KMP performs better and efficient and gives us very minimal running time a compared to others and gives us higher speed on the parallel architecture that mean s it uses benefits of the parallel architecture better than parallelization of Rabin Karp algorithm.



a) Numbers input set.



b) English text input set



c) DNA input data of various size.

Fig. 4 Performance Comparisons of the different algorithms against parallel KMP

Fig 4 shows the performance of the various algorithms in terms of running time on different type of input text that is DNA data, English text data and numbers data and that too of different size and gives the overview of which algorithms perform better.

- It shows that the parallel KMP performs better in every scenario of the with all text type and sizes. This shows KMP efficiently parallelize and uses parallel architecture efficiently and the running time of the parallel KMP is very efficient when compared to others algorithm in the experiments.

- From the figure we can infer that the parallelization of Rabin Karp does not work efficiently with the input size of the experiment and thus we can assume that parallel Rabin Karp will give better running times when the input text size is very large. This happens because the Rabin Karp calculates the hash values and for smaller input text it proves to be not useful and thus gives bad performance.

Fig 5 shows the running time comparisons between sequential and parallel KMP and from the figure it is evident that parallelization of KMP reduces the running time significantly and performs better.

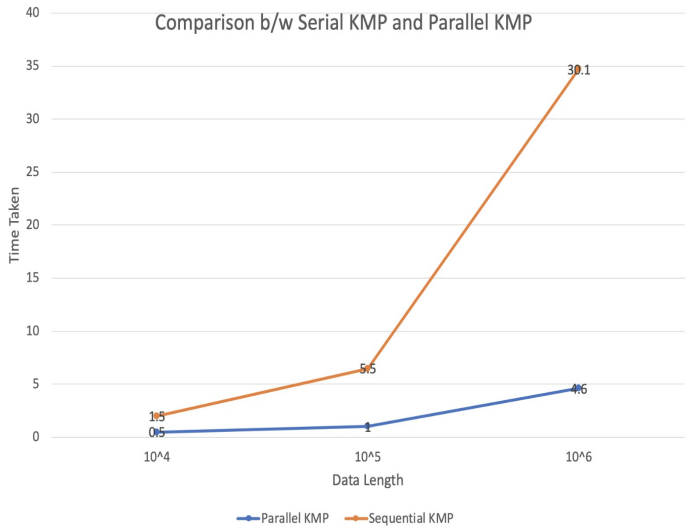


Fig. 5 Comparison of running time of Parallel and Sequential KMP.

IV. Results

Performance of the parallel algorithms also measured in the term of Speedup that is how much speedup the parallelization of the algorithm gives when compared to the running time of the best sequential algorithm for the same problem.

Speedup is formally given as the ratio of the sequential runtime for solving the problem to the time taken by parallel algorithm to solve the same problem in parallel architecture.

S =Ts/Tp

From the experiment results we can infer that speedup of parallel KMP comes out to be ~5 time when compared to sequential KMP algorithm.

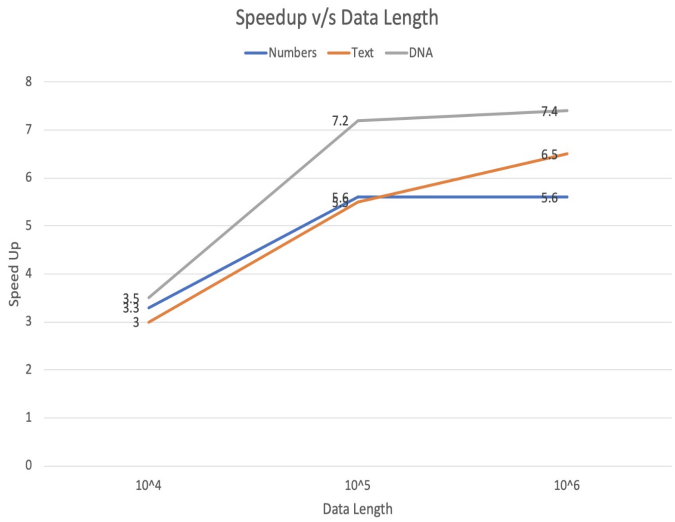


Fig. 6 Speedup of parallel KMP on various data input and different sizes.

Fig 6 gives a detailed comparison of the speedup achieved by Parallel KMP when tested on different input sizes that we considered in the experiment and across different types of input text. By taking into account the values from the experiment, calculating the speedups and then taking an average value we came to the result that parallel KMP in our experiment gives ~5 times speedup.

V. Conclusion

.

As stated in the beginning the string-matching algorithm is used in various domains. Since the usage of the algorithms is so wide the execution of the algorithm should be faster and done in a most efficient manner. This is where parallelization comes into the picture. We choose 2 of the most efficient string-matching algorithms and compared them. As seen from the Table I that the parallel algorithms perform better than the naïve algorithm of the sequential algorithmic implementation. In addition, we also assert that the parallel KMP algorithm performs better than the parallel Rabin-Karp algorithm. Thus, Parallel KMP upgrades the working of sequential KMP and is better than the algorithms mentioned above. It does this by reducing the execution time and producing the better results in less time.

References

[1] S. Park, D. Kim, N. Park and M. Lee, "High Performance Parallel KMP Algorithm on a Heterogeneous Architecture," 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, 2018, pp. 65-71.

[2] D.E. Knuth, J. James H. Morris, and V. R. Pratt, “Fast pattern matching in strings,” SIAM Journal on Computing, vol. 6, no. 2, pp. 323–350, 1977.

[3] Brođanac, Predrag & Budin, Leo & Jakobovic, Domagoj. (2011). Parallelized Rabin-Karp method for exact string matching.

[4] Shabaz, Er. Mohammad & Kumari, Er. (2017). ADVANCE-RABIN KARP ALGORITHM FOR STRING MATCHING.