Related Work :

Brute Force is a primary method which preprocesses neither the text nor the pattern. From left to right, BF performs a character by character analysis. The sliding window is moved one place to the right after a match or mismatch, and the matching is restarted from the first character of the pattern. A major drawback of BF is its high time consumption. There are approaches that combine the dynamic programming approach and DFA that are based on Deterministic Finite Automata. These methods aren't always scalable for large sequences due to the use of a finite automaton. Furthermore, because of the use of dynamic programming, the memory requirements are much higher. The KMP algorithm was proposed by Knuth which performs the comparison from the left side. When a mismatch occurs, KMP pushes the sliding window to the right by keeping the longest overlap of the matched text's suffix and the pattern's prefix. The output of this algorithm is linear. The KMP algorithm works well when the alphabet size is high, but it takes a long time to run when the alphabet size is small or the pattern length is short. From right to left, the Boyer-Moore algorithm look for a pattern in the text. This algorithm looks for the pattern's last character first. It computes the shift increment at the end of the matching phase. When a mismatch arises, two helpful rules (bad character and good suffix) are used to reduce the number of comparisons. The Boyer-Moore algorithm's drawback is that its preprocessing time is dependent on the pattern length and alphabet size. The Divide and Conquer Pattern Matching algorithm is based on comparisons. The text is checked for the pattern's rightmost character at the start of the DCPM's preprocessing phase. The rightmost character table contains the index of the findings. The text is scanned once more to detect pattern's leftmost character. The indexes are saved in the leftmost character table in the case of sameness. DCPM determines the window boundaries by using these two tables. In other words, the elements of the tables are investigated based on the length of the pattern. When the distance between the windows' leftmost and rightmost character equals the pattern length, a window is found. As a result, DCPM requires two passes through the text as well as some calculations to decide the windows. Total sameness is achieved when all of the pattern's characters and the text's windows are identical.

Methodology:

A pattern matching algorithm in this context must be able to search in datasets ranging from gigabytes to terabytes in size, as well as complete genomes with 3 billion base pairs [13]. The DNA sequences, on the other hand, are extremely lengthy. As a result, the time spent matching with the pattern is regarded as the most important metric. Whenever character t[s] of the text is aligned with character p[0], substring t[s..s+m−1] is called by the current window of the text. The text is scanned in the preparation phase of the proposed pattern matching algorithms to discover the windows of size m. The algorithms next test the pattern's characters one by one against those in the window in the matching phase to determine the pattern's total occurrence. After a complete match or mismatch, the other windows are checked for text matching.

- ***FIRST-LAST PATTERN MATCHING ALGORITHM:*** simple First-Last Pattern Matching (FLPM) algorithm is proposed in this section. FLPM is a DCPM upgrade that includes preprocessing and matching phases. ***A.preprocessing phase:*** Because FLPM is built on comparisons, the FLPM preprocessing phase scans text t to identify the text windows that will be used by the matching phase later. The first character of pattern p, i.e. p[0], is searched in text t at the start of the preprocessing step. The search is carried out during the interval t[0..n m] because the length of p is m. In contrast to the DCPM methodology , the FLPM algorithm checks the correspondence of p[m1] to t[s + m 1] at order to establish the end boundary of the window whenever character p[0] is aligned with the character in position s (in which 0 s n m) of text t, i.e., t[s] = p[0]. If these two characters are also the same, the t[s..s+m1] window is chosen as a candidate interval to be checked more accurately in the next phase. After then, the preprocessing phase continues to diagnose further windows. Although this technique concentrates on the first and last character of the pattern in recognising windows, it can be modified to include characters in other positions. ***B. MATCHING PHASE:*** The matching step explores the windows once the preprocessing step has identified them in order to discover all occurrences of the pattern in the text. As a result, the characters p[1..m 2] must be compared with characters t[si +1..si +m2] for the start index of each window, say si, determined in the previous step. Because their similarity was already confirmed in the previous phase, p[0] = t[si] and p[m 1] = t[si + m 1], the first and last character of the *pattern* and the window, respectively, should not be compared again. If the pattern is

completely identical to the text window, then this window is an answer. Otherwise, there's a mismatch. this phase continues by investigating the next windows.

- ***PROCESSOR-AWARE PATTERN MATCHING ALGORITHM:*** The PAPM (Processor-Aware Pattern Matching) algorithm is described in this section. The way pattern p characters and text t characters are compared in this algorithm differs from the FLPM algorithm. PAPM compares words made up of many characters, whereas FLPM uses a character-based pattern matching algorithm. PAPM compares two words at the same time using a processor's processing capabilities. The processor can compare the data of two registers throughout an execution cycle. Because each byte is made up of eight bits, word len = b/8 is used to calculate the number of processable bytes (or word length) for this CPU. In other words, the processor can compare a word (including word len characters) to another by using its registers at different times. A 32-bit processor, for instance, can compare a four-character word to another word at the same time.
- ***LEAST FREQUENCY PATTERN MATCHING ALGORITHM:*** The Least Frequency Pattern Matching (LFPM) algorithm is a DNA-specific enhancement of the PAPM algorithm. The LFPM algorithm, on the other hand, can be used to different pattern matching applications. Because of the time cost, LFPM is a good choice when many patterns must be found in the text, but it is not an efficient alternative when only a few patterns must be searched. LFPM looks for a low-frequency word of the pattern in the text to limit the number of recognised windows. In other words, LFPM concentrates on a word that is expected to appear less frequently in the text than other terms in the pattern. LFPM computes the frequency of all possible words for the related alphabet in the first step before the preprocessing phase. Although this step's computations add time to the process, it significantly reduces the time required for the later steps. The current text or a similar dataset can be used as a reference to calculate the frequency of particular words. The Human Reference Genome (HRG) is used as a reference point to count all possible words with the length of word len over the finite alphabet 6 = A,C, G, and T. Adenine, Guanine, Cytosine, and Thymine are the four nucleotide bases of a molecule of DNA (or a DNA sequence). Because word_len is the length of a word in a particular computer, the number of all possible words over alphabet 6 is 4word_len. It is worth mentioning that this number is fixed for all b−bit computers.

Results:

This section compares the performance of the presented algorithms (FLPM, PAPM, and LFPM) with that of the Brute Force (BF), Boyer-Moore (BM), and Divide and Conquer Pattern Matching (DCPM) algorithms. The word length for the PAPM and LFPM algorithms was four bytes due to the use of a 32-bit computer. The HRG was used as a reference in LFPM to construct the frequency table. The simulations carried out using the C programming language. In each experiment, the reference was searched for ten patterns, with the average of the results provided. It should be noted that the frequency table is constructed with a time overhead in LFPM. This time overhead was fixed during the simulation because the HRG dataset was used for all experiments. There was a total of 12 milliseconds of overhead. However, this time overhead was ignored in the calculations because the table of frequency is created only once for any text or database worked on by LFPM. The results of the simulated algorithms' performance evaluation in the preprocessing phase, matching phase, and total phases in terms of time cost are presented as follows. ***A. THE TIME OF PREPROCESSING PHASE:*** It should be noted that the preprocessing phase time for various simulated algorithms over the pattern length is negligible for the BM because the pattern is only analysed during the preprocessing phase of this algorithm. In FLPM, one pass across the text is enough to find the pattern's first and last character at the same time. DCPM, on the other hand, requires two passes: one to find the pattern's leftmost character and another to find the pattern's rightmost. PAPM looks for the pattern's first word in the text. ***B. THE TIME OF MATCHING PHASE***: FLPM, PAPM, and LFPM outperform the other algorithms .The main reason for their superiority is because they specify a low number of windows in the previous phase. In addition, by employing word processing, PAPM and LFPM's inquiry to match the windows and the pattern is much faster than that of the simulated character-based algorithms, i.e., BF, DCPM, BM, and FLPM. ***C. TOTAL TIME:*** The usage of word processing by PAPM and LFPM minimises the amount of time it takes to complete pattern matching. In LFPM, the number of times the least frequent word is repeated differs from the number of times the pattern's other words are repeated. The bigger the value of this difference, the better the LFPM performance.