Indian Journal of Science and Technology

Constraint Based Periodic Pattern Mining in Multiple Longest Common Subsequences

G. M. Karthik[1*] and Ramachandra V. Pujeri[2]

[1]SACS MAVMM Engineering College, Kidaripatti (PO), Algarkoil via, Madurai, Tamil Nadu, 625301, India; gmkarthik16@gmail.com

[2]KGiSL Institute of Technology, Saravanampatti, Coimbatore, Tamil Nadu, 641035, India; sriramu.vp@gmail.com

Abstract

The periodicity search in longest common subsequences in multiple strings has a number of application, is an interesting data mining problem. Periodicity is very common practice in longest common subsequence mining algorithm. This work introduces a new parallel algorithm for finding periodicity in multiple strings. Few existing algorithms lacks in poor scalability, lacks in finding all longest pattern, and for finding symbol, partial and full periodicity. We designed the algorithm using FP-tree for finding periodicity for most common longest substring in multiple sources. We introduce a parallel algorithm for *Constraint Based Periodic Pattern Mining (CBPPM)* algorithm, which takes $O(kN)$ for finding periodicity and $O((N \times L \times h)/p)$ time for MLCS pattern. We tested parallel algorithm on a coarse-grained multi-computer (BSP/CGM) model with $p < \sqrt{m}$ processors that takes $O(N \times L\sqrt{p})$ space per processor, with $O(\log p)$ communication rounds. We derive a practical implementation that works better for arbitrary length of input sequence. The algorithm is noise resilient, and shown its performance in presence of replacement, insertion, deletion, or mixture of these types of noise. We experimented with synthetic and real data reveals a near linear speedup with scalable performance. The comparative study shows algorithm's applicability and effectiveness, generally more noise resilient.

Keywords: Frequent Pattern (FP) Tree, Multiple Longest Common Subsequence (MLCS), Periodicity Mining, Noise Resilient, Parallel Processing, Data Mining.

1. Introduction

The uniform interval of time to reflect certain behavior of an entity is vital in many applications such as frequently sold products in a retail market, interval pattern in DNA sequences, stock growth, transactions in superstore, gene expression data analysis [20, 8, 3] etc. Identifying repeating (periodic) patterns could reveal important observations about the behavior and future trends of the case represented by the time series [2], and hence would lead to more effective decision making. The multiple longest common subsequences (MLCS) are an NP-hard problem [7], with vital application in bioinformatics and computational biology, mostly in DNA and protein sequence analysis. Several algorithm addresses simplest case of MLCS of two strings [7, 5], or special case of three or more strings [32, 18, 25]. MLCS is widely used in DNA and protein sequence analysis, in search for *motif* or set of motifs given a protein family. With the increase volume of biological data, we expect that MLCS algorithm will have a significant impact on computational biology methods and their applications.

To find the periodic pattern in MLCS, we propose a new and efficient pattern enumeration approach based on the ideas of frequent pattern mining techniques. First, we have developed an efficient parallel version of CBPPM algorithm in BSP/CGM model with a near-linear speedup. A novel, compact *Frequent Pattern tree* (FP-tree) like *TRIE* data structure, called *consensus tree* is constructed,

*Corresponding author:
G. M. Karthik (gmkarthik16@gmail.com)

which enables a highly parallelized search along the tree branches. The construction of consensus tree detects symbol, sequence, and segment patterns without periodicity, within subsection of the series. The growth of the tree is restrained by providing additional mining constraints. Using the strategy of *Constraint-Based Mining* [21, 14, 26, 31], we restraint growth of FP-tree using *user-specified constraints* [26], such as level constraint [14] and rule constraint [31]. Secondly, the algorithm looks for all periods starting from all positions available in a particular node of consensus tree. All the node of the consensus tree exists based on confidence greater than or equal to the user-specified periodicity threshold. In time series, there are three types of periodic patterns (*symbol / Sequence / Segment*) can be detected [10]. Integrating two techniques and developed an efficient algorithm known as *Constraint Based Periodicity Pattern Mining (CBPPM)* technique to solve MLCS problem and to find periodic pattern in MLCS. We also demonstrate through empirical evaluation that *CBPPM* is more scalable and faster than existing methods. *CBPPM* algorithm is proposed based on two points; the first we search for all subsequences of any length among given input strings. The second one is that we search for all instances of all subsequence in the input strings. Implicit user-defined constraint play vital role in pruning the search space of the FP-tree and influence time complexity.

The remainder of the paper is organized as follows: In Section 2, we recall the basic definitions and existing techniques available to solve to periodicity in MLCS. Section 3 we present our parallel algorithms, and discuss their time and space complexities. In Section 4, we compare the performance of the algorithms with existing algorithms. Finally, in section 5, some concluding remarks and plans about future works are given.

2. Basic Definitions and Related Works

The multiple longest common subsequences (MLCS) problem can be defined as follows: Let $S = \{s_1, s_2, \ldots s_N\}$ be a set of N sequences of length $L_1, L_2, \ldots, L_N$, correspondlingly, over a finite symbol set Σ with $|\Sigma|$=R, such that $|S_i| = L, 1 \le i \le N$, and positive integer d (represents *mutation level*, means number of *de-generative* characters in a subsequence) and q (subsequence must present in number of input sequences) such that $0 \le d \le L$ and $1 \le q \le N$. A *subsequence* of s_t is called *longest common substring* (*pattern* or *center string*), can be obtained from at least q input sequence contains a substring in s_t's d-neighborhood whose length is $|s_t| \ge |s_{t1}, s_{t2}, \ldots, s_{tj}|$, where $0 < t \le j \le max[L_1, L_2, \ldots, L_N]$. Note that for a given set of input sequence there can be more than one MLCS. In the case $q = 2$ *MLCS* problem is simply called the *longest common subsequence* problem (LCS). The MCLS is widely used in bioinformatics and computational biology, and most direct implementation in a protein sequence analysis is a search for a motif or set of motifs given a protein family.

Qingguo Wang et al [25] have given basic definition of MLCS problem, which is fixed parameter traceable with respect to the length of sequences. Existing techniques widely used dominant point approaches, applied to a case of two sequences [1, 11]. In [18], Algorithm A uses three input sequence. FAST-LCS [32], Hakata and Imai's C algorithm [18] and Qingguo Wang's Quick DP algorithm [25], works for arbitrary number of strings. To speed up the computation, parallel MLCS algorithms are developed [25]. PRAM algorithms for LCS and are presented in [25] for two input strings. Lu and Lin [21] proposed parallel algorithm with $O(\log^2 m + \log n)$ time complexity with $\frac{mn}{\log n}$ processors when $\log^2 m \log\log m \le \log n$. Xu et al. algorithm takes $O\left(\frac{mn}{p}\right)$ time, where $1 \le p \le \max(m,n)$. Qingguo Wang et al [25] take $|\Sigma|\log^d$ n time complexity, where d represents number of input strings.

The existing algorithm [4, 15, 17] requires the user to specify the period and patterns occurring with that period, otherwise which look for all possible periods in the sequence. The algorithms specified in [23, 24, 19, 10], looks for all possible periods by considering the range. COVN [23] fails to perform well when the sequence contains insertion and deletion noise. *WARP* [24] can detect segment periodicity; it cannot find symbol or sequence periodicity. Sheng et al., [6] developed algorithm based on [16] *ParPer* to detect periodic patterns in a section of the sequence; their algorithm requires the user to provide the expected period value. Huang and Chang [19] and *STNR* [10] presented their algorithm for finding periodic patterns, with allowable range along the time axis. Both finds all type of periodicity by utilizing the time tolerance window and could function when noise is present. *STNR* [10] can detect patterns which are periodic only in a subsection of the sequence.

In this paper, we develop a parallel version algorithm capable of detecting subsequence of all possible length and finding positions of all instances of these patterns, then

CBPPM algorithm to mine periodicity from given input MLCS sequence. From generated subsequence, finding the longest path in the consensus tree represents the longest common subsequence present in q input strings (If $q = 2$ represents the LCS problem). We design CBPPM algorithm to mine subsequence from given input strings to construct consensus tree, using it to calculate the periodicity among MLCS. Parallel CBPPM algorithm proposed in BSP/CGM Coarse Grained Multicomputer to find periodic pattern in MLCS with three or more input sequences.

3. Constraint Based Periodic Pattern Mining Algorithm

3.1 Frequent Pattern Tree Construction

We will now present a parallel computing version of CBPPM algorithm for MLCS problem. For simplicity, we consider the number of processors p to be a power of 2 and m to be a multiple of p, in which $p-1$ (*slave/local*) processor used for constructing consensus tree, and one processor (*master/Global*) used to find longest common subsequence. The Parallel CBPPM algorithm divides the number of input sequence into $p-1$ subsets of size $[p-1]$ that do not overload. In each p_i processor construct their own consensus tree T_{p_i} with given number of input sequences. All generated subsequence of T_p from each processor given to p_1 (master) to find the longest common subsequence and periodicity among them. Each processor construct consensus tree based on user specified rule and level constraints.

CBPPM algorithm uses the TRIE like structure (called *consensus tree*) for shared representation of all subsequence. CBPPM finds all subsequence $S_t \in \sum^t$ with any length l, $0 \le d < l \le L$ such that for each S_t, there are at least q sequences of S containing an *x-mutated* copy ($x \le d$) along with their instances. The consensus tree constructed by CBPPM algorithm shown in Figure 1, there are $|\Sigma|$ branches grown out from each non-leaf node. Each subsequence is mapped to sequence represent by a path from root to particular node (leaf/nonleaf node). Each node contains pointers to all subsequences mapped to S_t, where a pointer (j, k, e) points to a subsequence are starting at the k th position of the j th sequence and node containing pointers pointing to less than q input sequences, with level of mutation $e \le d$. Each node has $|\Sigma|$ branches only if nodes satisfy prescribed *support* and *confidence* value. A path from root to any node in consensus represents a subsequence.

Based on the constraints, the pre-pruning the nodes happens at each level like backward closure property. The consensus tree's growth is restrained using rule and level constraints. The number of levels in the consensus tree is at most $L = \max\{L_1, L_2, \ldots, L_N\}$ of the sequences. Nodes with *confidence value* as $conf(S_t) = \dfrac{N - \sup(S_t)}{N - q} < 1$ will be pruned; used as *antimonotonic constraint* [31]. The support *value* $sup(S_t)$ of subsequence, stand for number of pointers in each input sequences. A node in the consensus tree will not branch out if a support value is $\le q$, used similar to *monotonic constraint* [31]. Each instance in a consensus node has to satisfy degree of mutation $e \le d$, otherwise that particular instance is dropped, and we used it like succinct constraint. The longest paths from root node to any leaf node in consensus tree represent the longest common subsequence in given input sequence. In consensus tree contains more than one longest path which represents the MLCS. CBPPM algorithm performs many comparisons between the subsequence using Hamming distance. We use bitwise comparison with complexity $O\left(\left(ln_{\,2}|\Sigma| \times i\right)/\mathrm{w}\right)$. Bitwise comparison is better than Hamming distance calculation when $N > 2$. MLCS problem is fixed parameter tractable with respect to (l, d) with finite and fixed symbol set $|\Sigma|$. The MLCS is fixed with the parameter L and $|\Sigma|$, since l and d are bounded with L.

In worst case the number of developed nodes is $N(L-i+1)$, where each node can produce $\Sigma_{(j=0)^i}$ $(i \mid j)(|\Sigma|-1)^j$ variations with mismatched. This makes space complexity $O(N \times L \times f(d, l))$, which is roughly bound by $O(N \times L)$. Time complexity is not gained since we generate all possible subsequence, but will be gained back in space complexity. CBPPM can also produce mutated copy, the maximal number of node at each level i exceed $N(l - i + 1)$. We have used rule and level constraints, which do not test all possibilities; this would raise the time complexity to $|\Sigma|^L$. Hence time complexity is roughly bounded to

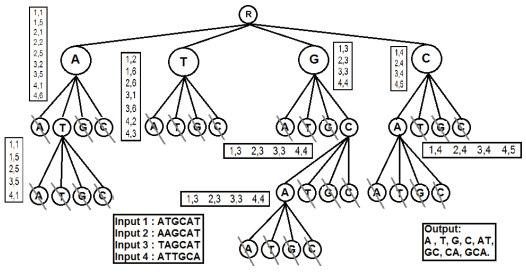


Figure 1. CBPPM algorithm construct consensus tree with $N = 4$, $q = 4$, $d = 0$.

$O(N \times L)$, with optimal low space complexity. Hence the time and space complexity for each processor is same as CBPPM algorithm.

To find the longest subsequence without mutation, consider all instances with value $e = 0, 0 \le e \le d \le L$, of a subsequence obtained from consensus node. To increase the accuracy of this process, value of mutation must be $d \cong L$, but it affects time and space complexity of CBPPM algorithm in larger scale. From this analysis, the total work is not parallelized and the work done during the communication steps is significantly smaller, compared to the amount of the parallelized work. Hence the running time of the parallel version of our algorithm that reflects the sequential version of CBPPM algorithm's efficiency. Figure 2 shows that the number of nodes generated with number of *slave* processors. Figure 3a shows the time taken for communication between the numbers of processors, and Figure 3b shows the time taken for different length of input sequences (with eight processors). Figure 2 and Figure 3 shows the great advantage of this approach, in contrast to classical dynamic programming approaches.

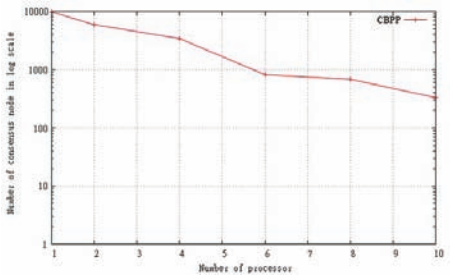


Figure 2. The number of node generated in consensus tree for number of processor using CBPPM algorithm for multiple random sequences of length 100.

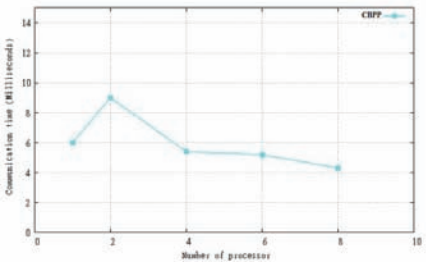


Figure 3a. The communication time with number of processor for MCLS sequences on three random sequences using CBPPM algorithm.

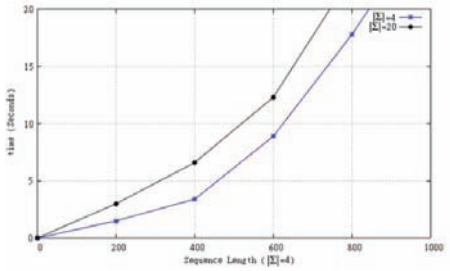


Figure 3b. The communication time with number of processor for MCLS sequences on three random sequences using CBPPM algorithm with different symbol set size.

3.2 Periodic Pattern Mining

The master processor p_1 does two processes, first, partition of number of input sequence based on number of processors, and secondly finding periodic pattern among generated subsequence. The most difficult part of the algorithm involves the problem in finding periodic pattern in MLCS. As mentioned above, we utilize the consensus tree node with its pointer for periodicity detection algorithm. Our algorithm is linear-distance-based; we take the difference between any two successive position pointers leading to *Difference vector*, represented in *Difference Matrix (Diff_matrix)*. *Diff_matrix* is not kept in the memory but this is considered only for the sake of explanation. Figure 4

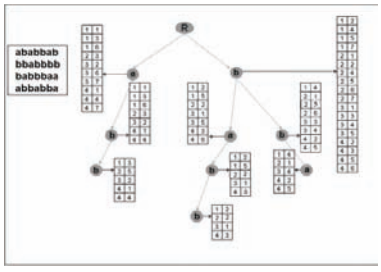


Figure 4a. An example of consensus tree structure constructed using CBFP algorithm with $d = 0$, $\Sigma = \{a, b\}$ and S = {(ababbab), (bbabbbb), (babbbaa), (abbabba)}, $N = 4$, $L = 7$, $q = 4$, and $|\Sigma| = 2$.

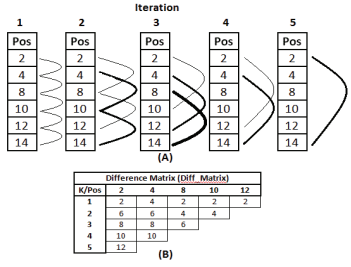


Figure 4b. Difference Matrix calculation for 'ab' pattern from FP tree node pointers.

presents how the *Diff_matrix* is derived from the position pointers of a particular node of consensus tree. From the matrix the periodicity is represented by *(S, K, StPos, EndPos, c)*, denoting the pattern, period value, starting position and ending position, and number of occurrences respectively for a particular consensus node (which denote a pattern). *CBPPM* algorithm scans the difference vector starting from its corresponding position (*Pos*), and increases the frequency count of the period (*K*) if and only if the difference vector value is periodic regard to the *StPos* and *K*. Algorithm 1 in appendix formally represent the formation of *Diff_matrix* form consensus node pointers.

CBPPM algorithm calculates all patterns which are periodic starting from any position and continues till the last occurrence of the pattern. FP tree node which contains pointers (*pos*) accessed as a continuous pattern for *Diff_matrix* calculation. Such types of periodicity calculation are very useful in real time DNA sequences. The existing algorithms [10] do not prune or prohibit the calculation of redundant periods; the immediate drawback is reporting a huge number of periods, which makes it more challenging to find the few useful and meaningful periodic patterns within the large pool of reported periods. Our algorithm reduces the number of comparison of pointers which are used for calculation periodicity. In Algorithm 2 we empowered to use *p* periods only one time for each and every position pointers from that *Diff_matrix* is calculated. *Diff_matrix* is able to assist in finding periodicity for every starting position with different *p* periods. Our algorithm not only saves the time of the users observing the produces results, but also saves the time for computing the periodicity by the mining algorithm itself.

The master processor p_1 does two processes, first, partition of number of input sequence based on number of processors, and secondly finding periodic pattern among generated subsequence. Time and space complexity of master processor is minimum when compare to time and space required by single processor involved in consensus tree computing. We use $\log p$ rounds to join the results, in which partial solutions are joined to give a single solution. The time necessary for the $p-1$ processors to solve the MLCS subproblem in parallel is $O\frac{(m\times N\times L)}{p}$. After $\log p$ rounds we have the solution of the original problem. The sum of times of all the union steps is $O\left(N\times L\times\sqrt{m}\left(1+\left(\log m\right)\right)/\sqrt{p}\right)$ and it takes $O\left(N\times L\sqrt{p}\right)$ space. Based on rule and level constraints few subsequences will be pruned, which may be present some input of any other *local* processor. Such subsequences must be communicated among the local processor which increase in time and space complexity. Instead problem is handled using the mutation factor *d*, which represent number of character is misplaced in a subsequence. Usually mutation value is taken as $d=\left[\frac{m}{p-1}\right]$, which handle above problem to an optimal level.

4. Experimental Results

We designed and implemented the parallel version of CBPPM algorithm for MLCS problem. The algorithm was implemented on the message-passing interface (MPI) system and run on local IBM SP3 machine. We have used *Scalasca* parallel processing tool which runs in Dell NVIDIA Linux cluster system, and aids in testing our parallel CBPPM in massively parallel processing (MPP) systems. The algorithms were tested on set of strings similar to the length of nucleotide and protein sequences, ranging between 100 and 5000 with $|\Sigma|=4$ and $|\Sigma|=20$.

4.1 Analysis of CBPPM Algorithm to Find MLCS

The parallel CBPPM algorithm was implemented using *Scalasca* parallel processing tool. The reason is it supports MPP environment which provides efficient performance. Our CBPPM algorithm consist of master thread which runs on master processor and FP tree creation by slave thread which runs in other slave processors. Master thread divides the number of input sequence based on available slave processor and assigns the input sequence along with constraints to each slave processor. After all slave processor complete consensus tree creation, they generate all subsequence and report it to master processor. Then, the master thread performs bitwise comparison among the all reported subsequence and report the longest subsequence with/without mutation value. CBPPM algorithm is compared with Hakata and Imai's A and C algorithms [18], Quick-DP algorithm [25]. The A algorithm is designed for three sequence MLCS problem, and C algorithm work with any number of sequence MLCS problem. Quick-DP, has consistent speed up than Hakata and Imai's algorithm.

Our algorithm takes more time than both Hakata and Imai's algorithm and Quick-DP, because our implementation generates all subsequence of three random DNA sequences of various lengths. Our implementation has higher precision and de-generative forms of MLCS can

be generated. Figure 5 shows that CBPPM compared with Quick-DP, Hakata and Imai's and FAST-LCS [32], for more than three random sequence of MLCS problem. From Table 1 it is clear Quick-DP has benchmark result than Hakata and Imai's algorithm and FAST-LCS algorithm for both alphabet size $|\Sigma| = 4$ and $|\Sigma| = 20$. CBPPM algorithm has moderate running time on long sequences. FAST-LCS is fast enough than Hataka and Imai's algorithm, but compared to Quick-DP less time efficient. The CBPPM algorithm's performance is far better than other existing technique is discussed in [13] and [12]. Time performance of CBPPM remains same as it checks for all possible subsequence irrespective of the data set. CBPPM performs better than WARP and STNR [13].

The running time of parallel CBPPM is compared with Quick-DPPAR (parallel version of Quick-DP) [25], on sequences of various lengths. Quick-DPPAR using 8-processor on five random DNA sequences is published in [25]. The Figure 6 demonstrates the efficiency of parallel CBPPM with Quick-DPPAR. Based on running time, Parallel CBPPM is less efficient compare to Quick-DPPAR in terms of time complexity. We test out parallel CBPPM with few set of protein sequences from the family of melanin-concentrating hormone receptors (MCHRs). From Pfam database [28, 27], few collection of KRAB-containing zinc-finger repressor protein families were used as a test data [29]. We used very few protein families listed in Table 2, which CBPPM solves MLCS with optimal solution. It is very difficult to compare the execution or computation time of CBPPM with existing techniques like MUSCLE [25] and ClustalW [22]. Since MUSCLE and ClustalW extract common subsequences by counting number of residues that are in common among all alignment sequences. Quick-DP, MUSCLE and ClustalW fails when pair wise identity among sequences is poor. But CBPPM identify those subsequences, since it is incorporated with mutation factor, de-generated sequences will be generated even when sequences pair wise identity is poor. CBPPM running time is always worse when compared to existing techniques like Hataka and Imai's algorithm and Quick-DP shown in Figure 6. CBPPM takes more time in generating all subsequences and along their instances. Hence computation time of CBPPM is not good as compared to dominant point approaches. CBPPM is recommended for finding a longest common subsequence with mutation. CBPPM algorithm is far better than existing technique, which are related in pattern mining is described in [13, 12]. The idea of using FP-tree in solving MLCS problem can be accomplished by CBPPM algorithm.

4.2 Analysis of CBPPM for Finding Periodicity in MLCS

CBPPM algorithm does not calculated redundant period, because which are supper-pattern has already been found periodic with same period value using *Diff_matrix*. Periodicity is calculated using *Diff_matrix* from bottom to top, hence algorithm does not check the redundant periods.The time performance of *CBPPM* compared to *ParPer*, *CONV*, *WARP* and *STNR* in three perspectives: varying data size, period size and noise ratio. First we compare *CBPPM* performance against *ParPer* [16], with synthetic data with varying data size from 1,00,000 to 10,00,000. The results are shown in Figure 7. *ParPer* only finds partial periodic patterns in the data namely symbol, segment and sequence patterns, and their complexity is $O(N^2)$. *STNR* [10], *CONV* [23] and *WARP* [24] are compared with size of the series varied from 1,00,000 to 10,00,00,000. Figure 8 shows *CBPPM* performs better than *WARP* and *STNR*, but worse than *CONV*. The run time complexity of *STNR* and *WARP* is $O(N^2)$, but for *CONV* is $O(nlogn)$. *CBPPM* performs better than *WARP* and *STNR* because *CBPPM* applies optimization strategies, mostly reduced the redundant comparison. *CBPPM* performance is shown in Figure 9 with varying period size from 5 to 100. *ParPer* [16] and *WARP* [24] get affected as the period size increased. Time performance of *CBPPM*, *CONV* and *STNR* [10] remains same as it checks for all possible periods irrespective of the data set.

The noise-resilient features in periodicity detection in presence of noise, is presented in [9, 10]. Three types of noise generally considered in time series data are replacement, insertion, and deletion noise. In order to deal with this problem, [10] used the concept of time tolerance into the periodicity detection process. The idea is that periodic occurrence can be drifted within a specified limit called time tolerance (denoted as *tt*), which is utilized in *CBPPM* algorithm. The *CBPPM* algorithm with time tolerance is presented in Appendix. In the case of noise ratio, we used a synthetic sequence of length 10,000 containing 4 symbols with embedded period size of 10. Symbols are uniformly distributed and generated in the same way as done in [23]. We used 5 combination of noise, i.e., replacement, insertion, deletion, insertion-deletion, and replacement-insertion-deletion. By gradually increased the noise ration

from 0.0 to 0.5, the confidence at period of 10 is detected. The time tolerance for all the experiments is ±2. Figure 10 show that our algorithm compares well with *WARP* [24], *STNR* [10] and performs better than *AWSOM* [30], *CONV* [23], and *STB* [9]. For most of the combination of noise, the algorithm detects the period at the confidence higher than 0.5. The worst results are found with deletion noise, which disturbs the actual periodicity. *CBPPM* shows consistent superiority because we consider asynchronous periodic occurrences which drift from the expected position within an allowable limit. This turns our algorithm a better choice in detecting different types of periodicity.

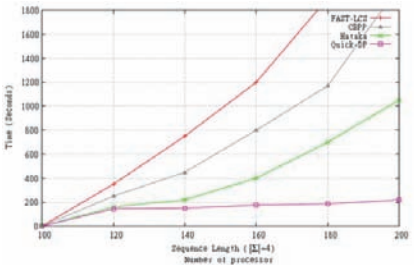


Figure 5. The average running time of CBPPM, Hataka and Imai's C algorithm, Quick-DP and FAST-LCS on MLCS protein of five random strings of different lengths.

5. Conclusion

In this paper, we presented novel algorithm uses FP tree as underlying structure for finding periodic pattern in MLCS. The algorithm can detect symbol, sequence and segment periodicity. In CBPPM, a highly parallel TRIE-like structure, the consensus tree, and fast level-wise search strategy based on downward closure property help to increase the efficiency. It can detect the redundant period which are pruned; before calculating confidence which in turn saves a significant amount of time. We tested the algorithm on both real and synthetic data in order to test its accuracy, effectiveness of reported results, and the noise resilience characteristics. Our algorithm runs in *O (k. N)* in the worst case for finding periodic patterns. Analysis of protein and genome sequence is one of the principle application are for the MLCS methods [32, 11, 25] using dominant point. Note that the space complexity for algorithm is optimal for all subsequences, since its space complexity is linear in the

Table 1. Average running time (seconds) of CBPPM, FAST-LCS, Quick-DP and Hataka and Imai's C algorithm for random five sequence of different length

Sequence length	Quick-DP		Hataka and Imai's C algorithm		FAST-LCS	CBPPM	
	$\|\Sigma\| = 4$	$\|\Sigma\| = 20$	$\|\Sigma\| = 4$	$\|\Sigma\| = 20$	$\|\Sigma\| = 4$	$\|\Sigma\| = 4$	$\|\Sigma\| = 20$
100	0.2	0	3.6	1.7	46.8	36	26
120	0.6	0.1	15.8	12.2	266	184	150
140	0.9	0.4	54.9	26.2	1430	1014	890
160	1.4	0.5	149.9	71.5	4801	2891	1450
180	2.2	0.8	426	203	17143	6434	2350
200	2.6	1.1	896	560	40262	9832	3122

Table 2. KRAB containing zinc-finger repressor sequence

Sub family	Protein	Species	Localization	Number of zinc finger	Expression pattern
A+B subfamily	HKrI8	Human	19	20	Ubiquitous
	RbaK	Human	7	16	Ubiquitous
	ZF5128	Human	19	9	Ubiquitous
A subfamily	HZF12	Human	19	9	Ubiquitous
A+b subfamily	ZNF222	Human	19	7	Ubiquitous
SCAN subfamily	ZFP95	Human	7	13	Ubiquitous

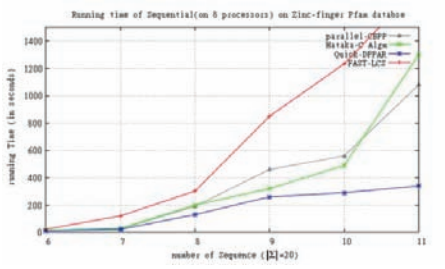


Figure 6. Comparing running time of the parallel CBPPM (on eight processors) with Quick-DPPAR, FAST-LCS, Hataka and Imai's C algorithm on zinc finger protein sequences from Pfam database.

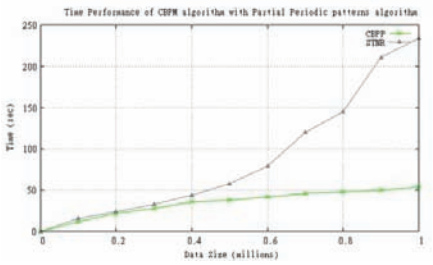


Figure 7. Time performance of CBPPM with ParPer algorithm.

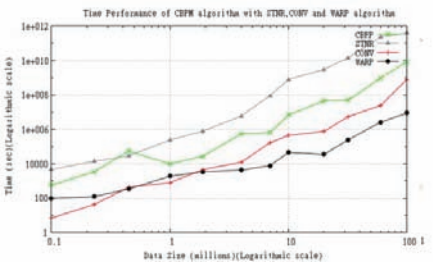


Figure 8. Time performance of CBPPM algorithm with STNR, CONV and WARP.

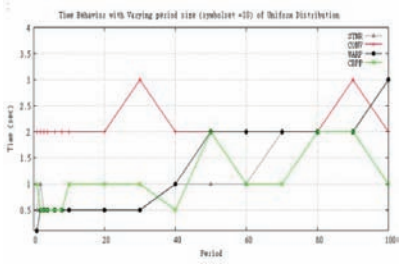
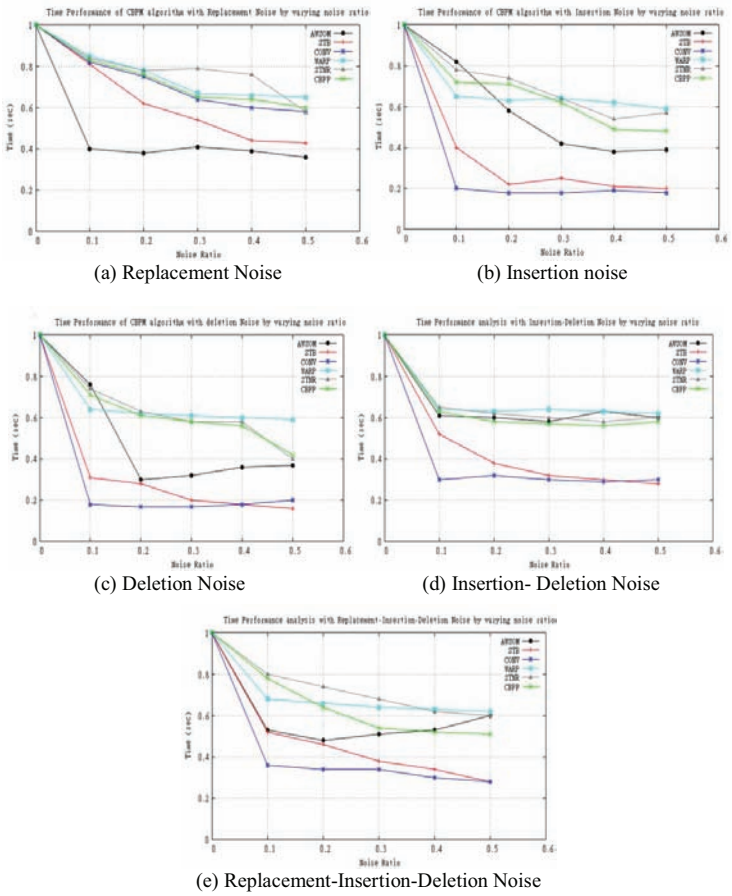


Figure 9. Time behavior with varying period size.

input size. We have found all algorithms, including ours, for solving MLCS are exponential in some parameters, which is inevitable. We have done only a few experiments in real biological data. In the future, we will apply this approach to solve more real problems in biological computation.

6. References

1. Apostolico A, Browne S et al. (1992). Fast linear-space computations of longest common subsequences, Theoretical Computer Science, vol 92(1), No. 1, 3–17.
2. Weigend A, and Gershenfeld N (1994). Time series prediction: forecasting the future and understanding the past, Addison-Wesley.
3. Ptitsyn A A, Zvonic S et al. (2007). Permutation test for periodicity in short time series data, BMC Bioinformatics (BMCBI), vol 8, 395.
4. Berberidis C, Aref W et al. (2002). Multiple and partial periodicity mining in time series databases, Proceedings of European Conference Artificial Intelligence.
5. Rick C (1994). New algorithms for the longest common subsequence problem, Technical Report No. 85123-CS, Computer Science Department, University of Bonn.
6. Sheng C, Hsu W et al. (2005). Mining dense periodic patterns in time series data, Proceedings of 22nd IEEE International Conference on Data Engineering, 115.
7. Maier D (1978). The complexity of some problems on subsequences and supersequences, Journal of the ACM, vol 25(2), 322–336.
8. Glynn E F, Chen J et al. (2006). Detecting periodic patterns in unevenly spaced gene expression time series using lomb-scargle periodograms, Bioinformatics, vol 22(3), 310–316.
9. Rasheed F, and Alhajj R (2010). STNR: a suffix tree based noise resilient algorithm for periodicity detection in time series databases, Applied Intelligence, vol 32(3), 267–278.
10. Rasheed F, Al-Shalalfa M et al. (2011). Efficient periodicity mining in time series databases using suffix trees, IEEE Transactions on Knowledge and Data Engineering (TKDE), vol 23, No.1, 79–94.
11. Chin F Y, and Poon C K (1990). A fast algorithm for computing longest common subsequences of small alphabet size, Journal of Information Processing, vol 13, No.4, 463–469.
12. Karthik G M, and Pujeri R V (2008). Constraint based frequent pattern mining technique for solving GCS problem, proceedings of World Academy of Science, Engineering and Technology, vol 32, 672–679.
13. Karthik G M, and Pujeri R V (2012). Constraint based periodicity mining in time series databases, International Journal of Computer Network and Information Security, vol 10, 37–46.
14. Han J, Lakshmanan L V S et al. (1999). Constraint-based multidimensional data mining, IEEE Computer, vol 32, No. 8, 46–50.



(a) Replacement Noise (b) Insertion noise

(c) Deletion Noise (d) Insertion- Deletion Noise

(e) Replacement-Insertion-Deletion Noise

Figure 10. Time performance of CBPPM compared with STNR, CONV, ParPer, WARP, AWSOM, STB.

15. Han J, Gong W et al. (1998). Mining segment-wise periodic patterns in time related databases, Proceedings of ACM International Conference on Knowledge Discovery and Data Mining, 214–218.
16. Han J, Yin Y et al. (1999). Efficient mining of partial periodic patterns in time series database, Proceedings of 15th IEEE International Conference in Data Engineering, 106.
17. Yang J, Wang W et al. (2002). InfoMiner: mining partial periodic patterns with gap penalties, Proceedings of Second IEEE International Conference on Data Mining.
18. Hakata K, and Imai H (1998). Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima, Optimization Methods and Software, vol 10(2), 233–260.
19. Huang K Y, and Chang C H (2005). SMCA: a general model for mining asynchronous periodic patterns in temporal databases, IEEE Transaction on Knowledge and Data Engineering, vol 17, No. 6, 774–785.
20. Dubiner M et al. (1994). Faster tree pattern matching, Journal of ACM, vol 14(2), 205–213.
21. Lu M, and Lin H (1994). Parallel algorithms for the longest common subsequence problem, IEEE Trans. Parallel and Distributed System, vol 5, No. 8, 835–848.
22. Larkin M A, Blackshields G et al. (2007). Clustal W and Clustal X Version 2.0, Bioinformatics, vol 23, 2947–2948.
23. Elfeky M G, Aref W G et al. (2005). Periodicity detection in time series databases, IEEE Transactions on Knowledge and Data Engineering, vol 17, No. 7, 875–887.
24. Elfeky M. G, Aref W G et al. (2005). WARP: Time *WARP*ing for periodicity detection, Proceedings of Fifth IEEE International Conference on Data Mining, 138–145.
25. Wang Q et al. (2011). A fast multiple long common subsequence (MLCS), IEEE Transactions on Knowledge and Data Engineering, vol 23, No. 3, 321–334.
26. Ng R et al. (1998). Exploratory mining and pruning optimizations of constrained associations rules, Proceedings of 1998 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, 13–24.
27. Finn R D, Mistry J et al. (2006). Pfam: clans, web tools and services, Nucleic Acids Research, vol 34, D247–D251.

28. Finn R D, Tate J et al. (2008). The Pfam protein families database, Nucleic Acids Research, vol 36, D281–D288.
29. Urrutia R (2003). KRAB-containing zinc finger repressor proteins, Genome Biology, vol 4, No. 10, 231.
30. Papadimitriou S, Brockwell A et al. (2003). Adaptive, hands off-stream mining, Proceedings of 29th Very Large Data Bases (VLDB) conference, 560–571.
31. Lee S D, and De Raedt L (2004). Constraint based mining of first order sequences in SeqLog, Database Support for Data Mining Applications, 154–173.
32. Chen Y, Wan A et al. (2006). A fast parallel algorithm for finding the longest common sequence of multiple biosequences, BMC Bioinformatics, vol 7, S4.

Appendix

CBPPM Algorithm (FP Tree Construction)

CBPPM performs many compare operations for calculating distance between two patterns. Mainly, complexity *O (i)* deals with comparing the symbols of the two strings one by one in *ith* level of the consensus tree. For each candidate center string in each node is $\overline{j=0}, \sum_{j=0}^{\min(d,i)} \left(\frac{i}{j}\right)(|\Sigma|-1)^j$, calculated at most $N \times (L-i+1)$ distances. The time complexity is roughly bound by $O(N \times L)$. The secondary storage used for running the CBPPM is bound by $O(N \times (L-i+1))$. Therefore, the total space complexity is $O(N \times L)$. Testing all possibilities of patterns restrained by *rule* and *level* constraints this would raise the time complexity to $|\Sigma|^L$. We generate those candidates whose consensus strings satisfy the prescribed *rule* constraint. Therefore, using our strategy we raise time complexity by $O(N \times L)$ with quite low space complexity.

Algorithm 1: FP-tree construction

1. **For** each string *j* of given input sequence *N* **do**
2. **For** each symbol *k* of input string *j* of length *L* **do**
3. **If** the *k*th symbol *i*th sequence is $b_1 \in \Sigma$ **do**
4. Put (*j*, *k*, 0) in new node S_{b1}, find (*j*, *k*, 1) substring is in all $S_{b'1}$ for $b'_1 \neq b_1$ and *j* in $T_{b'1}$ for each $b'_1 \in \Sigma$ if and only if $sup(b_i) > threshold$.
5. **For** each *i*th sequence from 1 to *L* **do**
6. **Loop(1):**
7. **For** each substring's $conf(b_1, b_2, b_3, \ldots, b_{i-1}) \geq 1$ **do**
8. **Loop(2):**
9. **For** each entry (*j*, *k*, *e*) in each nodes $S_{b_1, b_2, \ldots, b_1}$ where $k < L - i + 1$ **do**
10. **Loop(3):**
11. **If** the $k + i$th element of the *j*th sequence is $b_{i-1} \in \Sigma$ and $sup(b_{i-1}) < q$ **do**
12. **Begin(1):**
13. **put** (*j*, *k*, *e*) in $S_{b_1, b_2, \ldots, b_0, b_{i-1}}$;
14. **if** $e < d$ then for all $b'_{i-1} \neq b_{i-1}$
15. **put** (*j*, *k*, *e* + 1) in $S_{b_1, b_2, \ldots, b_0, b_{i-1}}$ if and only if $conf\left(_{b_1, b_2, \ldots, b_0, b_{i-1}}\right) \geq 1$;
16. **End Begin 4;**
17. **If** $conf(b_{i-1}) < 1$ then Remove $S_{b_{i-1}}$;
18. **End Loop 3;**
19. **For** each node $S_{b_1, b_2, \ldots, b_{i-1}} \neq \phi$ **do**
20. **For** each node in next level $S_{b'_1, b'_2, \ldots, b'_0}$ with $distance(b_i, b'_i) \leq d$ **do**
21. **For** each $S_{b'_1, b'_2, \ldots, b'_1} \neq \phi$ and $conf\left(S_{b'_1, b'_2, \ldots, b'_i}\right) \geq q$ along with $distance(b_i, b'_i) \leq d$ **do**
22. **Loop(5):**
23. **If** $conf(b'_i) < 1$ then Remove $S_{b'_i}$
24. **Create** a new level in consensus tree with $T_{b'_1, b'_2, \ldots b'_1} \leftarrow T_{b'_1, b'_2, \ldots b'_1} \bigcup S_{b'_1, b'_2, \ldots b'_1}$
25. **If** no node exists in $T_{b'_1, b'_2, \ldots, b'_1}$ then
26. **Increment** *i* ; End Loop2;
27. **Else**
28. **Print** the output sequence $(b'_i, S_{b'_i})$;
29. **End Loop 5;**
30. **If** all $S_{b_1, b_2, \ldots, b_i, b_{i+1}}$ are removed then stop the program else output all pairs $b_1, b_2, \ldots, b_{i-1}; S_{b_1, b_2, \ldots, b_i, b_{i+1}}$
31. **Remove** all $S_{b_1, b_2, \ldots, b_i}$ and $T_{b_1, b_2, \ldots, b_i}$;
32. **End Loop 2;**
33. **i = i + 1;**
34. **End Loop 1;**

Algorithm 2: Difference Matrix (*Diff_matrix*) Algorithm

- **Input:** a MLCS (**S**) of size **N** contains position pointers **Pos;**
- **Output:** Difference Matrix (**A**) containing Difference vector;

1. **For** $i = 1$ *to* $N - 1$
2. **Begin** Loop 1:
3. **Assign** $j = 1$
4. **if** (j < N − i)
5. $A(j, i) = S_j - S_{j+i}$;
6. **if** $(j + 1 \neq j + i)$
7. **Then**
8. $t = j + 1$;
9. **While** $(t < j + i - 1)$
10. **Begin Loop** 2:
11. $A(t, i) = S_t - S_{t+i}$;
12. $t = t + i$;
13. **End Loop** 2;
14. **Endif;**
15. $j = j + i$;
16. **Endif;**
17. **End Loop** 1;

Algorithm 3: Constraint Based Periodicity Mining Algorithm (*CBPPM*)

- **Input:** *Diff_matrix* (**A**), and time tolerance value *tt*;
- **Output:** position of periodic patterns P;

1. **For** $K = N - 1$ *to* 1;
2. **Begin Loop** 1:
3. **For** $i = 1$ to $N - k$;
4. **Begin Loop** 2:
5. **Assign** $j = i$, $c = 1$;
6. **if** $(j + K \leq N - K$ **then**
7. **if** Difference $(A(j, K), A(j + K, K))$ is in between $(A(j, K) \pm tt)$
8. **then** c++;
9. **Endif;**
10. **if** $\exists P_{j,K+1}$ and Diff $(A(j, K), A(j, K + 1))$ is in between $(A(j, K) \pm tt)$
11. **then** $c = c + P_{j,k+1}(q)$;
12. **Endif;**
13. $j = j + k$;
14. **Goto step** 6:
15. **Else**
16. **Assign** $stPos = j$, $endpos = j + k$, $p = k$, $q = c$;
17. Project Periodicity $P_{j,K}(S, p, stPos, endPos, q)$;
18. **If** $(i + K > N -$ K)
19. **Break Loop** 2;
20. **Endif;**
21. **End Loop** 2;
22. **End loop** 1;