CHAPTER 3

An Efficient Fast Pruned Parallel Algorithm (EFPLCS) for finding Longest Common Subsequences in BioSequences

EFPLCS is an efficient and fast approach to identify the Longest Common Subsequence between biosequences. Identifying Longest Common Subsequence between two or more biosequences is an important problem in computer science due to its computational complexity and applicability to the field of Biology. This algorithm achieves its efficiency in using computational resources by doing specific pruning steps in the process of identifying the Longest Common Subsequence.

3.1. Overview of EFPLCS

Given two sequences over an alphabet $\Sigma$, algorithm EFPLCS generates successor tables and creates Initial Identical Pairs (IIDP). Starting from early non redundant IIDP, applies the Prune A, Prune B, Prune C, Prune D and Prune E steps, while it keeps identifying their successors at each level until there are no more successors. This is the point of longest common subsequence(s) having been identified. The level number also means the length of LCS i.e. length of LCS, $|\text{LCS}(X, Y)| = \text{maximum level}$. The number of LCS identified between the given strings is given by the number of pairs in the last level. It is possible to have more than one or none string as LCS. Now backtrack to identify the LCS. Backtracking can be carried out in parallel if there is more LCS and hence all can be obtained concurrently. Parallel implementation of this EFPLCS algorithm is possible. In this case one each node can be assigned an Initial Identical Character Pair to identify LCS and the central node identifies the longest LCS of all.
3.1.1. Pruning strategies

- **Prune A:** Pruning redundant IIDP
- **Prune B:** A pair which does not have an occurrence in the other sequence.
- **Prune C:** Pairs which are surely not a participant for longest LCS
- **Prune D:** Pairs which cannot produce successors in the next level
- **Prune E:** In sequential implementation release memory of those elements belonging to shorter LCS pairs

3.1.2. Features of EFPLCS

- Identifies all the possible LCS
- Precision in identifying LCS is 100%
- Faster and resource efficient than some of the algorithms
- The time complexity is O(L) for sequential implementation and O|LCS| for parallel implementation
- Memory complexity as max [4*(n+1)+4*(m+1), L]
- Parallel Implementation of algorithm is feasible.

3.2. Methodology of EFPLCS

3.2.1. Successor Table

Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_m)$ and Character set $CH = \Sigma = (A,C,G,T)$ then $x_i, y_i \in \Sigma$.

The Successor Table of $X$ and $Y$ are denoted as $TX$ and $TY$. $TX$ and $TY$ are two dimensional arrays with elements $\Sigma \times X$ and $\Sigma \times Y$. The size of $TX$ is $|\Sigma| \times (|X| + 1)$ and the size of $TY$ is $|\Sigma| \times (|Y| + 1)$. The entries into successor table are defined as below:
\[ T(i,j) = \begin{cases} \min\{k \mid k \in S(i,j)\} & \text{when } S(i,j) \neq \Phi \\ - & \text{otherwise} \end{cases} \quad \text{…………..(Eq.1)} \]

\[ S(i,j) = \{k \mid x_k = CH(i), \quad \text{for } k > j\} \]

where \( i = 1, 2, 3, \ldots, |\sum| \) and \( j = 0, 1, 2, \ldots, n \) i.e. length of sequence.

When ,

\( T(i,j) \neq - \), the values indicate the forthcoming position of \( CH(i) \) in the sequence;

\( T(i,j) = - \), i.e. no value, it indicates that there is no more occurrence of \( CH(i) \) after \( j^{th} \) position.

Example:

Let \( X = \text{“TCGTAC”} \) and \( Y = \text{“ATGCTAA”} \).

Therefore \( |X| = n = 6; \ |Y| = m = 7 \).

Their successor Tables TX and TY are shown in Table 3.1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( j = ) CH(i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX:</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td></td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td></td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( j = ) CH(i)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>TY:</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td></td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
3.2.2. Identical Character Pair (IDP)

For sequences X and Y, if \( x_i = y_j = CH(k) \), then \((i, j)\) is an identical pair of CH(k). The set of all identical character pairs of X and Y is denoted as S(X, Y).

\((TX(k, 0), TY(k, 0))\), where \(k = 1,2,3,4\) are known by special name Initial Identical Character Pair (IIDP), the IIDPs of X and Y. In this case these are \((5,1), (2,4), (3,3), (1,2)\). The IIDPs are eligible to get entry into Pair Table at level 1. All other Identical Pairs (IDP) get their entry into the appropriate level as the case may be. Pair table is a data structure in working memory with entries of eligible identical pair.

3.2.3. Efficient Pruning Techniques

PRUNE-A (Identifying non redundant IIDPs)

In the IIDP, i.e. at initial level, eliminate redundant pairs which are unlikely to generate LCS.

Example: \((5, 1), (2, 4), (3, 3), (1,2)\) are IIDPs. Since \((2,4) > (1,2)\), \((2,4)\) is unlikely to produce a longer LCS than \((1,2)\). Hence IIDP \((2, 4)\) can be pruned. Similarly IIDP \((3, 3)\) can also be pruned because \((3, 3) > (1, 2)\). Thus \((2, 4)\) and \((3, 3)\) are identified to be redundant IIDPs and non redundant and useful IIDPs are only \((5, 1)\) and \((1, 2)\). In this case, 50% reduction in IIDPs at initial level, reduces the redundant successor generation and hence computation time. Of course the final number of useful IIDPs depends on the two sequences getting compared.

PRUNE - B

An entry in successor table is ‘-‘when there are no more repeat of that character. Any successor pair having a no value member i.e. , ‘-‘ is redundant.
Example: (-, 6), (6, 4), (-, 3), (-, 2) are successors at level 1 for IIDP (5, 1) corresponding to character A. Except for (6, 4) all are having no value members. Hence these are redundant and can be pruned.

In another case (5, 6), (6, -), (3, -) and (4, 5) are successors of IIDP (2, 4) corresponding to Character C. Here (6, -) and (3, -) have no value members and become redundant successor pairs. Therefore it can be pruned. Only useful successor pairs are (5, 6) and (4, 5).

**PRUNE - C**

Among the successors of the same parent, a successor is said to be redundant if it satisfies the condition as below:

Let \((i_1, j), (i_2, j), \ldots, (i_m, j), (k, l)\) be the successors in the same level of a parent, such that \(i_1 < i_2 < \ldots i_m < k\) and \(j < l\).

Then \((i_1, j)\) alone is the non redundant successor. Therefore if \((i_1 < i_2)\) and \((j < l)\), then prune \((i_2, l)\).

**PRUNE - D**

An entry in successor table is ‘-‘ when there are no more repeat of that character. The whole column having an entry of ‘-‘ means that the string has ended. Therefore, while identifying successor for a pair \((i, j)\), if either members of column i of the TX table or members of column j of TY table, are all having no entry, then the pair \((i, j)\) is redundant in the next level. Hence it does not qualify to get entry into Pair table; prune in this level itself. This is a lemma to Prune A.

**Pruning E:**

Prune E is useful in Sequential implementation of EFPLCS algorithm.
Take the 1st eligible IIDP. Identify the LCS starting with this IIDP character. Store the LCS and |LCS| arrived at. Do the same with next IIDP. Compare the length of 2nd LCS arrived with the previous one. Discard the lesser length one and retain only the longer one. Continue the iteration until all the IIDPs are over. This will let us hold only the longest LCS arrived so far and hence operate with less memory.

3.3. EFPLCS Algorithm

3.3.1. Algorithm EFPLCS(X,Y) - Parallel version

**Input:** Query data sequence X of length m
Reference Data Sequence Y of length n
Let X, Y be contained in alphabet set \( \Sigma \) = k.

**Output:** string LCS of length |LCS| which is /are Longest Common Subsequence(s) between X and Y.

**Procedure:**

**Step 1:** Build Successor Table TX and TY for the X and Y sequences over the alphabet \( \Sigma \).

**Step 2:** IIDP entry into Pair Table

**Step 2a:** Find all the Initial Identical Pairs(IIDP) of X and Y sequences over \( \Sigma \) such that \( \{TX(k,0),TY(k,0) | k=1,\ldots,k\} \)

**Step 2b:** Apply Prune-A on IIDPs.

**Step 2c:** Add the potentially useful IIDPs to the Pair Table.

Pair Table Data Structure: (i, j, level, Predecessor, Record number)

\( i = TX(k,0) \)
\( j = TY(k,0) \)
level =1 for IIDP, otherwise level = parentlevel +1.
Predecessor = $\emptyset$ | IIDP ,
= parent level |otherwise
Record number = last entry in pair Table + 1

Step 3: Producing Successors level wise
Repeat steps 3.1 thru 3.2 until no more successors are found

Step 3.1:
For all the current level identical pairs in pair table parallel do
3.1.1. identify all direct successors
3.1.2. Apply Prune B, C and D.

Step 3.2:
Add the potentially useful successors to pair table.

Step 4: Backtracking to collect LCS
For each pair in the max level, parallel do
$\text{Pred} =$ predecessor; LCS($r$) = $X_i$ ;
While $\text{pred} \neq \emptyset$ do
Read the pair table entry of Record number
Set $\text{pred} =$ predecessor; LCS ($r'$) =$X_i$ ;
LCS of $X,Y$ is stored in the array LCS . $|\text{LCS}| =$ max level.

3.3.2. Algorithm EFPLCS($X,Y$) - (Sequential version)

Input: Query data sequence $X$ of length $m$
Reference Data Sequence $Y$ of length $n$
Let $X,Y$ be contained in alphabet set $|\Sigma| = k$.

Output: string LCS of length $|\text{LCS}|$ which is /are Longest Common Subsequence(s)
between $X$ and $Y$. 
**Procedure:**

**Step 1:** Build Successor Table TX and TY for the X and Y sequences over the alphabet $\sum$.

**Step 2:**

- **IIDP entry into Pair Table**
  - **Step 2a.** Find all the Initial Identical Pairs (IIDP) of X and Y sequences over $\sum$ such that $\{TX(k,0),TY(k,0) | k=1,\ldots,k\}$
  - **Step 2b.** Apply Prune-A on IIDPs.
  - **Step 2c.** Add the potentially useful IIDPs to the Pair Table.

Pair Table Data Structure: $(i, j, \text{level}, \text{Predecessor}, \text{Record number})$

- $i = TX(k,0)$
- $j = TY(k,0)$
- level = 1 for IIDP, otherwise level = parentlevel + 1.
- Predecessor = $\emptyset$ | IIDP,
  
  
  = parent level | otherwise

Record number = last entry in pair Table + 1

**Step 3:** Producing Successors level wise for each IIDP

- **Take the 1st IIDP do Step 3 and 4**
  - Repeat steps 3.1 thru 3.2 until no more successors are found

**Step 3.1:**

- For the current IIDP in pair table
  - 3.1.3. Identify all direct successors
  - 3.1.4. Apply Prune B, C and D.

**Step 3.2:**

- Add the potentially useful successors to pair table.

**Step 4:** Backtracking to collect LCS
For each pair in the max level, parallel do
Pred = predecessor; LCS (r) = X_i;
While pred ≠ ∅ do
    Read the pair table entry of Recordnumber
    Set pred = predecessor; LCS (r’) =X_i;
LCS of X,Y is stored in the array LCS-current . |LCS-current| = max level.
Take one by one each IIDP in the pair table and do Steps 3 to 5
When no more IIDP in pair table exit to step 6

**Step 5:** Apply Prune E|
Go back to step 3

**Step 6:** The LCS is content of LCS and exit.

### 3.4. Execution and Result Analysis

The execution time and memory requirement are proportional to number of
IIDP at level 1 and the |LCS|. The lengthier the sequence X and Y, the higher the
chances of |LCS| being large and hence the resource requirement for finding solution
also is high.

The algorithms EFPLCS and FASTLCS [86] were executed on Intel
Dualcore/1.6GHz/1GB Desktop Computer system. The implementations of
algorithms were done in C++. The popular FASTLCS works in similar fashion to
EFPLCS.
The genomic database “aedb” subset from EBI\textsuperscript{3} and the protein sequences “pdb” from EBI ftp server\textsuperscript{4} were used. Execution was done with 3 Datasets of Genome sequences and Protein sequences on FASTLCS and EFPLCS Algorithms. Since test on one pair of sequence would take very less time, the datasets were iterated 50 times.

The performance comparison of EFPLCS is done on four accounts. The number of Successor records updated in pair table is considered as one of the features for performance comparison and effectiveness of pruning. Two other features used are CPU time and memory utilization. The fourth feature is qualitative. i.e. completeness to identify the correct and complete LCS is compared.

3.4.1. Dataset-1

Genome Dataset-1 had a query genome sequence of |50|. Different Y sequences of length from |40| to |150| were considered and duplicated for iterative purpose. As seen from Table.3.2, EFPLCS is able to identify LCS by generating only one 3\textsuperscript{rd} or less records and hence the memory used is also that much less. This had been possible by the additional pruning taken up by EFP_LCS. The FASTLCS goes out of bound early for want of more main memory during execution.

The resultant LCS sequences were compared for both length and number of LCS. EFPLCS is found to produce identical result with FASTLCS. EFPLCS is found to work consistently well for larger sequences than FASTLCS as the memory usage is less by 40% to 70%. EFPLCS computational efficiency (CPU Time) i.e. speedup in

\textsuperscript{3}ftp://ftp.ebi.ac.uk/pub/databases/fastfiles/asd/
\textsuperscript{4}ftp://ftp.ebi.ac.uk/pub/databases/pdb_seq/
finding solution is faster by 40% to 70% than FASTLCS. The precision of EFPLCS is the same as that of FASTLCS.

Table 3.2. Results with DATASET-1

<table>
<thead>
<tr>
<th>Y sequence length</th>
<th>FAST_LCS</th>
<th>EFPLCS</th>
<th>Speedup in CPU time in %</th>
<th>Memory Efficiency in %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max. Records generated</td>
<td>Memory used in KB</td>
<td>CPU time in secs</td>
<td>Max. Records generated</td>
</tr>
<tr>
<td>40</td>
<td>105264</td>
<td>1233</td>
<td>0.531</td>
<td>32451</td>
</tr>
<tr>
<td>50</td>
<td>502000</td>
<td>5882</td>
<td>2.516</td>
<td>121470</td>
</tr>
<tr>
<td>60</td>
<td>2755866</td>
<td>32295</td>
<td>12.656</td>
<td>573656</td>
</tr>
<tr>
<td>70</td>
<td>8033993</td>
<td>94148</td>
<td>33.953</td>
<td>2037663</td>
</tr>
<tr>
<td>80</td>
<td>14438168</td>
<td>169197</td>
<td>60.562</td>
<td>3490658</td>
</tr>
<tr>
<td>90</td>
<td>28879413</td>
<td>338430</td>
<td>118.540</td>
<td>7693702</td>
</tr>
<tr>
<td>100</td>
<td>Failed</td>
<td></td>
<td></td>
<td>9800183</td>
</tr>
<tr>
<td>110</td>
<td>Failed</td>
<td></td>
<td></td>
<td>12754824</td>
</tr>
<tr>
<td>120</td>
<td>56614872</td>
<td>663455</td>
<td>223.040</td>
<td>15126615</td>
</tr>
<tr>
<td>130</td>
<td>Failed</td>
<td></td>
<td></td>
<td>15744843</td>
</tr>
<tr>
<td>140</td>
<td>Failed</td>
<td></td>
<td></td>
<td>16507549</td>
</tr>
<tr>
<td>146</td>
<td>61859249</td>
<td>724913</td>
<td>244.100</td>
<td>16518431</td>
</tr>
</tbody>
</table>

NA- Not Applicable

Beyond the computational overheads necessitated by pruning, the pruning has very effectively reduced the redundancy and hence EFPLCS is faster than FASTLCS. It is to be noted from the Dataset-I results that FASTLCS has failed to converge with some of the Y sequences, whereas EFPLCS has succeeded to identify LCS over those
critical ranges too. The average speedup achieved by EFPLCS over FASTLCS on this Dataset-1 was 67% and memory efficiency achieved was 70%. The performance is plotted in Figure 3.1 and figure 3.2.

Figure 3.1. CPU Time with DATASET-1

Figure 3.2. Memory used with DATASET-1
### 3.4.2. Dataset-2

Genome Dataset-2 is an experiment using *aedb* genome sequences of variable length. 100 such sequences were taken as Y sequences. The |Y| was ranging from 3 to 54. Each of the five sequences of length between 17 and 54 were used as X sequence and iterated. The dataset-2 is executed using both EFPLCS and FASTLCS. A sample execution statistics generated while execution of the program is shown in Figure.3.3 and Figure.3.4 for EFPLCS and FASTLCS respectively.

<table>
<thead>
<tr>
<th>PMopt</th>
<th>Xlen</th>
<th>Ylen</th>
<th>#LCS</th>
<th>#Recs</th>
<th>ET</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>1</td>
<td>54</td>
<td>30</td>
<td>1</td>
<td>29084</td>
</tr>
<tr>
<td>2)</td>
<td>1</td>
<td>54</td>
<td>25</td>
<td>5</td>
<td>13704</td>
</tr>
<tr>
<td>3)</td>
<td>1</td>
<td>54</td>
<td>24</td>
<td>3</td>
<td>5582</td>
</tr>
<tr>
<td>4)</td>
<td>1</td>
<td>54</td>
<td>31</td>
<td>5</td>
<td>26713</td>
</tr>
<tr>
<td>5)</td>
<td>1</td>
<td>54</td>
<td>29</td>
<td>8</td>
<td>35901</td>
</tr>
<tr>
<td>6)</td>
<td>1</td>
<td>54</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7)</td>
<td>1</td>
<td>54</td>
<td>6</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>8)</td>
<td>1</td>
<td>54</td>
<td>27</td>
<td>32</td>
<td>34907</td>
</tr>
<tr>
<td>9)</td>
<td>1</td>
<td>54</td>
<td>42</td>
<td>5</td>
<td>152174</td>
</tr>
<tr>
<td>10)</td>
<td>1</td>
<td>54</td>
<td>37</td>
<td>5</td>
<td>157428</td>
</tr>
<tr>
<td>11)</td>
<td>1</td>
<td>54</td>
<td>42</td>
<td>4</td>
<td>453704</td>
</tr>
</tbody>
</table>

Figure 3.3. EFPLCS Execution statistics on Dataset 2- a sample

---

Figure 3. 4. FASTLCS Execution Statistics on Dataset2 – a sample

Figure 3. 5. CPU Time with Dataset-2
On this Dataset-2 aedb genome sequences, EFPLCS has yielded 35% speedup in CPU activity and 42% memory efficiency. On |X|=24, in spite of the CPU time being indifferent, memory efficiency is achieved. The result of Dataset-2 is enumerated in Table 3.3. Figure 3.5. and Figure 3.6 show the graphical view of comparison.

![Memory usage on Variable |query|](image)

**Figure 3.6. Memory Usage with Dataset-2**

<table>
<thead>
<tr>
<th>Query sequence length X</th>
<th>FASTLCS</th>
<th>EFPLCS</th>
<th>Speedup in CPU time %</th>
<th>Memory Efficiency %</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>29 KB</td>
<td>0.046</td>
<td>0.031</td>
<td>32.61</td>
</tr>
<tr>
<td>24</td>
<td>207</td>
<td>0.063</td>
<td>0.062</td>
<td>1.59</td>
</tr>
<tr>
<td>37</td>
<td>2602</td>
<td>0.235</td>
<td>0.141</td>
<td>40.00</td>
</tr>
<tr>
<td>48</td>
<td>128281</td>
<td>4.188</td>
<td>65873</td>
<td>2.438</td>
</tr>
<tr>
<td>54</td>
<td>128281</td>
<td>9.578</td>
<td>65873</td>
<td>5.844</td>
</tr>
</tbody>
</table>

Table 3.3. Results with Dataset-2
3.4.3. Dataset - 3

Genome Dataset-3 was experimented to bring out the data dependency behavior between the X and Y sequences in identifying LCS. Hence the X sequence was replaced with another sequence of \(|50\) in the Dataset-1. Here again EFPLCS continued to perform better with CPU speedup of 35% and memory efficiency of 42%. The details are shown in Table 3.4. On higher lengths when FAST_LCS fails EFPLCS continues to perform. In these cases the speedup and memory efficiency become “NA” meaning not applicable as the data for FAST_LCS were not available.

| Sequence length Y | FASTLCS | | | EFPLCS | | | Speedup in CPU time % | Memory Efficiency % |
|-------------------|---------|----------------|---------|---------|----------------|----------------|----------------|-----------------|----------------|
|                   | Max. Records generated | Memory used in KB | CPU time in secs | Max. Records generated | Memory used in KB | CPU time in secs |                      |                  |                  |
| 100               | 14493326 | 169843 | 64.922 | 7635007 | 89472 | 42.547 | 34.46 | 47.32 |
| 110               | 25582052 | 299789 | 108.047 | 14754224 | 172901 | 74.922 | 30.66 | 42.33 |
| 120               | 34280097 | 401719 | 144.375 | 19400963 | 227355 | 99.516 | 31.07 | 43.40 |
| 130               | 41080502 | 481412 | 175.641 | 23090835 | 270595 | 124.2 | 29.29 | 43.79 |
| 150               | Failed | 40747821 | 477513 | 203.48 | NA | NA |

3.4.4. Dataset – 4 (Protein Sequences)

The dataset is protein sequence pdb from EBI database. Protein sequences were tested with 4 different datasets like the case of Genome sequence.
Dataset- I had \(|X| = 40\) and \(|Y| = \{80,90,\ldots,170\}\).
Dataset – II had \(|X| = 80\) and \(|Y| = \{80,90,\ldots,170\}\)
Dataset – III had \(|X| = 58\) and \(|Y| \{100\) strings of varying length}
It was observed that EFPLCS provides almost the same efficiency as that of genome
data shown with dataset 1 through 3. However, EFPLCS was able to converge with
$|X|=80$ while FAST-LCS was able to converge only with $|X|=50$. The $|Y|$ limit for
EFPLCS was up to 258 on $pdb$ data while that for FAST_LCS was 146.

3.4.5. Performance on Longer Sequences

On Applying Prune-E the EFPLCS could converge with genome data length of
900 whereas FAST_LCS fails to converge for want of resource requirements. The
details are as below:

$X$ Length: 49

No. of $Y$ Strings: 10

$Y$ Length Max: 900

Average $Y$ Length: 900

Total Execution Time: 97.437

Maximum Records generated: 27152506

Memory used: 318193KB (318MB)

This shows it is possible to have long sequences compared within the available
resources with EFPLCS.

3.5. Conclusion

Since EFPLCS produces all the possible LCS, it is expected to give more
relevant results. On a Sequential pairwise implementation EFPLCS basis has been
proven to give efficiency in the use of computational resources and memory usage. It
is expected to give better memory efficiency in MSA implementation. If EFPLCS
were run on powerful computer, it could certainly cross the limits of most leading
LCS algorithms. The numeric limits on \(|X|\) and \(|Y|\) are computer system resource dependent and hence only comparative performance should be used to understand the EFPLCS ability to perform.