A dominant point-based parallel algorithm that finds all longest common subsequences for a constrained-MLCS problem

Armel Nkonjoh Ngomade, Jean-Frédéric Myoupo, Vianney Kengne Tchendji

Version of Record: https://www.sciencedirect.com/science/article/pii/S1877750319301942
Manuscript_d02a6b3d597d83331d67f9872d45d600

A dominant point-based parallel algorithm that finds all longest common subsequences for a constrained-MLCS problem

Armel Nkonjoh Ngomade[1], Jean Frédéric Myoupo[2] and Vianney Kengne Tchendji[1]

[1]*Department of Mathematics and Computer Science, University of Dschang, Dschang, Cameroon*

[2]*Computer Science Lab-MIS, University of Picardie Jules Verne, Amiens, France*

Abstract: The work presented in this paper consists of searching for all the longest common subsequences among r sequences of equal length n which exclude a particular string. This problem is named the constrained-multiple longest common subsequence (constrained-MLCS) and is a general case of the constrained-LCS. Due to its importance, particularly in bioinformatics, the constrained-MLCS is widely studied. Thus, the solution proposed here is a coarse-grained multicomputer-based algorithm that uses an existing serial algorithm for local computation and a master-slave paradigm. This solution requires $O\left(\frac{|D|\times|\Sigma|+|CSSs|\times|MLCS|}{p}\right)$ local computation time on each processor, $O(|MLCS|)$ communication rounds and $O\left(\frac{nr|\Sigma|}{p}\right)$ time for preliminary works. $|\Sigma|$ is the alphabet's size, p is the number of processors used, $|D|$ is the number of dominants generated during the entire resolution process, $|CSSs|$ and $|MLCS|$ are the number of all common subsequences, and the length of the longest common subsequences respectively. The experiments performed indicate that our proposed algorithm is scalable both with the number of processors and the number of input sequences.

Keywords: Constrained-multiple longest common subsequence**,** Dominant point, Coarse Grained Multicomputer.

1. Introduction

With important applications in various fields such as computational biology, information retrieval and file comparison [2, 13, 24], the multiple longest common subsequence problem, denoted MLCS problem, is a classic NP-hard problem which consists in finding the longest subsequence shared between two or more sequences. In fact, according to the number of sequences, this problem can be classified into two cases:

1. An **LCS** problem which is searching for the longest common subsequences of exactly *two* sequences;
2. An **MLCS** dealing with at least three sequences.

Considerable efforts have been made during the last three decades to find solutions to this problem, but the most significant and applicable contributions deal only with the particular case of two or three sequences [4, 11, 18, 28]. Indeed, with the increasing volume of biological data and the widespread use of sequence analysis tools, an efficient MLCS algorithm, applicable to many sequences, will have a significant impact in bioinformatics.

Many sequential and parallel algorithms have been recently proposed [17, 22, 28, 29] for the MLCS problem. Depending on the model on which the solutions are based, these algorithms can be classified into two groups: the dominant point-based and the dynamic programming-based approaches. Using the dominant point-based approach has an enormous advantage over classical dynamic programming approaches because it greatly reduces the size of the search space by orders of magnitude [28]. Although this approach seems suitable, it also suffers from unnecessary and redundant storage, computations, comparison and deletion of multidimensional match points [17, 22] that hinder its application on long and numerous sequences.

To fill the gaps in the dominant point-based approach, in this work we look at a variant of the MLCS problem called the Constrained-MLCS problem. In fact, our aim is to propose a parallel algorithm based on an ameliorated version of the dominant point approach [22] and also based on the Bridging Coarse Grain BSP/CGM (Bulk Synchronous Parallel/Coarse Grained Multicomputer) model [3, 7, 27]. The CGM seems best suited for designing algorithms that are not too dependent on an individual architecture. A BSP/CGM machine is a set of p processors, each having its own local memory of size m with a connection to a router able to deliver point-to-point messages. A BSP/CGM algorithm alternates between local computations and global communication rounds. A CGM computation/communication round corresponds to a BSP super-step with a communication cost of $g \times m$, where g is the cost of communicating a word in the BSP model. Therefore, an efficient BSP/CGM algorithm must have maximum speedup and must use minimum number of communication rounds [7].

Our main contribution in this paper is to propose a parallel algorithm based on the serial solution of Z. Peng and Y. Wang [22] for the Constrained-MLCS. The motivation to use this particular sequential algorithm is that the data structure (Leveled-DAG) used provides a better optimization of time and space. In fact, existing dominant point-based algorithms have to generate a huge number of nodes and save them all in memory, while the Leveled-DAG

approach can timely eliminate all the nodes in the graph that cannot contribute to the construction of MLCS. At any moment, only the nodes in the current level as well as some nodes in the previous levels are saved, therefore, the Leveled-DAG is much smaller than the DAG constructed by the existing dominant point-based algorithms, which can save a lot of memory space and allows to solve larger problems. Our contribution can be summarized as follows:

- We introduce and clearly define a new variant of the MLCS problem.
- We also propose an efficient CGM-based algorithm that searches all the common subsequences between a set of sequences.
- To take the constraint into consideration, we first define a suitable data structure called *CSS-graph* which is used to store all the previously found common subsequences and we propose an efficient parallel algorithm to construct it and to perform the validation of the constraint.
- We do some analysis of our proposed algorithms resulting in the solution requiring $O\left(\frac{|D||\Sigma|+|CSSs||MLCS|}{p}\right)$ local computation time on each processor, $O(|MLCS|)$ communication rounds and $O\left(\frac{nr|\Sigma|}{p}\right)$ for the preliminary works. In this analysis, p, n, r denote respectively, the number of processors used, the length and the number of sequences and $|D|$, $|CSSs|$, $|MLCS|$, $|\Sigma|$ are the number of dominant points generated by the sequences, the number of common subsequences, the length of the longest common multiple subsequences and the size of the alphabet respectively.
- We perform experiments with sample sequences from a real biological NCBI database [23]. The collected results show that our solution is suitable for long sequences and is sufficiently scalable both with the number of input sequences and with the number of processors.

This paper is organized as follows: Section 2 is dedicated to introducing key concepts and for presenting recent research related to the MLCS problem. In Section 3, we present and analyze our parallel algorithm to solve the Constrained-MLCS problem. Experimental results and analyses are presented in Section 4. Finally, in Section 5, we summarize the paper and provide directions for future research.

2. Constrained-MLCS problem and Related works

In this section, we will first provide a formal definition of the MLCS and Constrained-MLCS problems; second, we will describe the dominant point approach used to solve the MLCS problem in the research literature and finally, we review related works on exact and approximate, sequential and parallel algorithms for the MLCS problem.

2.1. The Constrained-MLCS problem

The MLCS problem with a string-exclusion constraint, denoted STR-EC-MLCS, consists of searching all common subsequences of several sequences that are of maximum length and which exclude a string.

Definition 1. Let Σ be a finite set of symbols called an *alphabet.* A *sequence* $\boldsymbol{x}$ of length $\boldsymbol{n}$ over the alphabet Σ is defined as: $\boldsymbol{x} = x_1 x_2 \dots x_n$. The ith character of $\boldsymbol{x}$ is denoted $\boldsymbol{x}[i]$. A sequence $\boldsymbol{y} = x_{i_1} x_{i_2} \dots x_{i_k}$ is a *subsequence* of $\boldsymbol{x}$ if $\forall j, 1 \leq j \leq k: 1 \leq i_j \leq n$ and $\forall r, t, 1 \leq r \leq t \leq k: i_r \leq i_t$. A *substring* of $\boldsymbol{x}$ is a subsequence of successive symbols within $\boldsymbol{x}$.

Definition 2. Let $S = \{S_1, S_2, \dots, S_d\}$ be a set of sequences over alphabet Σ. The **MLCS** problem for set S consists in finding all sequences $\boldsymbol{x}$ such that:

i. *$\boldsymbol{x}$ is a subsequence of S_i for each i;*
ii. *$\boldsymbol{x}$ is the longest among all sequences satisfying (i).*

In general, there exists more than one MLCS between many sequences.

Definition 3. For a given set of sequences $S = \{S_1, S_2, \dots, S_d\}$ and a string P_{ct}, the Constrained-MLCS searches all sequences $\boldsymbol{x}$ such that:

i. *$\boldsymbol{x}$ is a subsequence of $S_i, 1 \leq i \leq d$;*
ii. *$\boldsymbol{x}$ doesn't entirely contain P_{ct};*
iii. *$\boldsymbol{x}$ is of maximal length satisfying (i) and (ii).*

This problem has already been addressed in the literature [9, 10, 18, 20], but it was only for the simple case of two input sequences.

An example of this problem is depicted in Figure 1 where the input sequences are ACTAGCTA, TCAGGTAT and CTAAGTTA.

Notations

Hereafter we will use the following notations:

- **MLCS** stands for multiple longest common subsequence;
- |MLCS| is the length of the MLCS;
- **ST** is the table of successors which indicates for a sequence $\boldsymbol{x}$ the position of the next character identical to each symbol of the alphabet after a given position;

- **CSSs** stands for all common subsequences between multiple sequences;
- **STR-EC-MLCS** stands for string exclusion multiple longest common subsequence;
- r is the number of sequences in a MLCS;
- p denotes the number of processors used;
- n is the common length of the r sequences;
- D is the set of all dominant points generated during the search process;
- P_{ct} is the constraint string (or sequence) in a Constraint-MLCS problem.

They will be widely used in the following.

Figure 1. Illustration of the MLCS and the Constrained-MLCS problems with input sequences ACTAGCTA, TCAGGTAT, CTAAGTTA and a constraint string P_{ct} =TA.

2.2. The dominant point-based approach

The dominant points approach, introduced by Hischberg [4, 19], consists of reducing the search to the exploration of a smaller set of dominant points rather than all the positions in a square matrix as done in the dynamic programming technique. This method has been widely used not only for the LCS problem [1, 5, 19] but also for the extended case of more than two sequences [4, 13, 28].

2.2.1 Preliminaries: definitions

Definition 4. Over a set of sequences $S = \{S_1, S_2, \dots, S_d\}$, a vector $v = (v_1, v_2, \dots, v_d)$ is called a *match point* if $S_1[v_1] = S_2[v_2] = \cdots = S_d[v_d] = \alpha$. Hereafter match $v(\alpha)$ will denote the fact that the vector v is a match point over S. In Figure 1, $X[4] = Y[3] = Z[3] = \mathrm{A}$, thus the point v = (4, 3, 3) is a match point.

Definition 5. Given two match points $v = (v_1, v_2, \dots, v_d)$ and $w = (w_1, w_2, \dots, w_d)$, we say that v ***dominates*** w denoted by $v \leq w$ if $v_i \leq w_i$ for $1 \leq i \leq d$. In a similar way, we say that v ***strongly dominates*** w if $v_i < w_i, 1 \leq i \leq d$. In contrast, v ***doesn't dominate*** w if there exists an $i, 1 \leq i \leq d$ such that $w_i \leq v_i$. This relation is denoted. $v \nleq w$. We also say that w is a ***successor*** of v if w strongly dominates v and no other match point z exists such that $w < z < v$.

We can observe in Figure 1 that (2, 2, 1) ≤ (4, 3, 3), (2, 2, 1) ≰ (3, 1, 2) and (4, 3, 3) is a successor of (3, 1, 2) on the symbol T.

Definition 6. A match $v = (v_1, v_2, \dots, v_d)$ is called a dominant of level ***k*** or a ***k****-dominant* if:

i. $M(v) = k$ *where* M *is a matrix computed by a dynamic programming technique using the standard recursive formula for the MLCS problem (see [22]);*

ii. *There is no further match point on the same symbol that dominates v.*

We denote D^k and D as the set of all k-dominant points and the set of all dominant points respectively.

Definition 7. A point z in a set B is called the minimal point of B if for all points $q \in \mathrm{B} - \{z\}, such\ that\ q \nleq z$. We denote by ***minima (B)*** the set of all the minimal points of B.

2.2.2 Principle of the method

The key idea of this method is based on the observation that only the dominant points can contribute to the construction of the MLCS. The search space of the dominant point method can be represented as a Direct Acyclic Graph (DAG) in which:

i. *A node represents a match point;*

ii. *The edges* $\langle z, q \rangle$ *represent the fact that q is a successor of z.*

This approach consists of constructing the DAG starting from level 0 to level $|MLCS|$. Therefore, initially, the DAG contains only the source $(0,0,\dots,0)$ and the final $(\infty,\infty,\dots,\infty)$ nodes, which don't have incoming and outgoing edges respectively. From here, the DAG is constructed level by level as follows: at the first level, $k = 0$ and $D^k = \{(0,0,\dots,0)\}$, and with a forward iteration procedure, the (k + 1)-dominants D^{k+1} are computed based on the k-dominants D^k. At the end, the DAG will be fully built and an MLCS will be any longest paths starting from the source node to the final node.

A simple example (*case of two sequences*) of this method is illustrated in Figure 2 where S_1 = AGCTGA and S_2 = CAGATCAGAG. In this figure, the nodes in *gray* and *black* represent those that have been removed either by the *minima* operation or because they appear more than one time in the same level. For example, to build level 1, we look for all the successors of the node (0,0) on all the symbols of the alphabet. Here, we have the nodes A(1,3), G(2,3), C(3,1) and T(4,5) among which G(2,3) and T(4,5) are not minimal nodes because (1, 2) ≤ (2, 3) and (1, 2) ≤ (4, 5). In level 2, the node (4,5) is duplicated, thus, one should be removed (CT(4, 5)). It should be noted that the final node is the successor of nodes having no successors. In this figure, the nodes in dark gray are nodes leading to the MLCS.

Figure 2. A DAG constructed using the dominant point method using sequences S_1 = AGCTGA and S_2 = CAGATCAGAG.

2.3. Related Works

Because of its crucial importance, this problem has received the attention of many researchers. Thus, several sequential and parallel solutions have been proposed. Among these solutions, some just calculate the length of the MLCS and others return the sequences corresponding to the MLCS. It should be noted that the dominant points technique is the most used.

Based on dynamic programming, Hsu and Du [11] proposed for an MLCS problem with r sequences of equal length n, an algorithm requiring $O(n^r)$ time and space. This algorithm returns a set of sequences which are LCS. To improve this, many other solutions have been proposed [1, 9, 19]. In [9], after proposing a CGM-algorithm to compute the length of the LCS, a backtracking method is used to find the corresponding LCS. Even with all these improvements, these solutions are still inefficient for practical use. For the LCS problem, many parallel solutions have been recently proposed [10, 26].

Recently, many dominant point-based solutions have been developed for the special case of two sequences [5, 12, 14, 20]. In [14], three dominant point algorithms for three or more sequences were proposed. One of the algorithms, *Algorithm A*, which was designed specifically for MLCS problems of three sequences, is much faster than the traditional dynamic programming algorithms for three sequences. However, *Algorithm A* finds dominant point sets by enumerating points of the same coordinate values in each dimension. As a result, its complexity increases rapidly with the growing number of sequences. The other algorithm, Koji and Imai's algorithm [14], works for an arbitrary number of strings. Apart from finding the length of the MLCS, these algorithms also proposed methods for recovering all MLCS.

For the MLCS problem, D. Korkin [15] proposed the first parallel MLCS algorithm requiring $O(|\Sigma||D|)$ time complexity, where $|D|$ is the number of dominant points in the graph. Later, Chen et al. [4] presented an efficient MLCS algorithm named FAST-LCS for DNA in which they introduced a novel data structure called a successor table to obtain the successors of nodes in constant time and they used a pruning operation to eliminate the nondominant nodes in each level. To improve this solution, Wang et al. [28] use the divide-and-conquer strategy, which is very suitable for parallelization to eliminate the nondominant nodes and proposed an efficient algorithm named Quick-DPAR. Many other improvements have also been proposed [16, 17, 22, 30, 31].

Among these recently proposed solutions, we have chosen as our sequential algorithm for local computation, the solution of Z. Peng and Y. Wang [22] because of its better

optimization of time and space. This solution consists in setting up a leveled-DAG that collects the partial MLCS until reaching the complete MLCS. The preliminary work consists in constructing the tables of successors. The entire execution process for sequences ACTAGCTA and TCAGGTAT of their solution is depicted on Figure 3. In this figure, the match point and the corresponding symbol are shown in each node. The partial LCSs are shown by strings near the nodes. The white nodes are newly generated and will be expanded later and the green ones are outdated and will be removed right away. The red nodes with incoming edges are left from the previous levels and cannot be removed at present. Steps (**A**), (**B**) and (**C**) generate the first, second and third level of nodes respectively. In step (**D**), no new node is created any more. Step (**E**) deletes the remaining outdated nodes and in step (**F**), only the end node is left.

The construction steps of the new graph are:

Step 0: Compute the entries of the successor tables of each input sequence;

Step 1: Build the first level of the Leveled-DAG: generate all the successors of the initial node as the first level by referring to the successor tables;

Step 2: Build the next level of the Leveled-DAG and delete the outdated nodes (*generate and delete*). A node is said to be outdated or useless when it can no more contribute to the search process of the MLCS. If there are nodes in the Leveled-DAG that have not been expanded, repeat the following two substeps:

Step 2.1: For each unexpanded node *z*, generate all its successors (*if a successor already exists in the graph, it does not need to be generated several times and only needs a pointer*), and if *z* has no successors, let the final node be its only successor;

Step 2.2: If $|partial\_LCS(z)|$ is the length of the partial LCS of the node *z* then, for each node *z* which does not possess an incoming arc (*nodes (1, 3), (2, 2) and (3, 1) in figure 3 at step B*), and for each successor *s* of z do the following:

- If $|partial\_LCS(z)| \geq |partial\_LCS(s)|$, delete the partial LCSs of *s*. Append the corresponding symbol of *s* to each partial LCS of *z*, and then save all the appended partial LCSs as the new partial LCSs of *s*;
- If $|partial\_LCS(z)| \geq |partial\_LCS(s)| - 1$, then append the corresponding symbol of *s* to each partial LCS of *z*, and add all the appended partial LCSs to the existing partial LCSs of *s*. Delete node *z* (*as well as its partial LCSs*) from the graph.

Step 3: Repeat Step 2.2, until only the final node is left in the graph;

Step 4: Output the partial LCSs saved in the end node, which are the real MLCS of the input sequences.

Figure 3. The Leveled-DAG constructed for sequences ACTAGCTA and TCAGGTAT.

The resulting algorithm returns all MLCS in a time complexity that is linear to the number of nodes in the graph, say $O(|D|)$ and a space complexity that depends on the maximum level of the Leveled-DAG $O(Max\_Level)$.

To find the length of the Constrained-LCS, many sequential solutions have been proposed [8, 25]. For parallel solutions, Deorowicz [8] proposed the first bit-parallel algorithm for the STR-IC-LCS problem and recently we proposed a CGM-based algorithm for a string-exclusion LCS [21]. To the best of our knowledge, our parallel algorithm, finding all the constrained-MLCS, is the first presented in all the research literature.

3. CGM Algorithm for the Constrained-MLCS problem

This section describes our proposed parallel algorithm. We first present an efficient CGM algorithm for preliminary works, second, we propose a parallel algorithm using the master-slave paradigm to find all the common subsequences (CSSs) between the input sequences and we finally describe an algorithm to validate all the previously found CSSs according to a constraint.

3.1. Preliminaries

For a STR-EC-MLCS problem with *r* sequences of equal length *n*, the calculation of the values of the entries of a successor table (ST) for a sequence $\boldsymbol{x} = x_1 x_2 \dots x_n$ is performed using equation (1) [22]:

$$ST[i,j] = min\{m | x_m = \alpha_i, m > j, 1 \le i \le |\Sigma|, 0 \le j \le n\} \quad (1)$$

where α_i is the ith symbol in Σ .

From this equation, the following dependency relationships are derived:

i. The computation of the entries of the tables ST for two sequences S_i and S_j is independent with $j \neq i$ and $1 \le i, j \le r$;

ii. When calculating the entries of a table ST, the computation of the values corresponding to each symbol of the alphabet is mutually independent. Formally, the computation of each $ST[i,j]$ for $1 \le i \le |\Sigma|$ is independent to $ST[k,j]$ for $1 \le k \le |\Sigma|$ with $k \neq i$;

iii. The computation of each $ST[i,j]$ for a sequence of size *n* with $1 \le j \le n$ depends on the symbols present in the sequence in positions $j+1, j+2, \cdots, n$.

Based on these dependencies, because our algorithm runs on a parallel computer with p processors, and in order to ensure load balancing among processors, we propose two distribution schemes:

Scheme 1: Equally distribute the sequences on processors. This formally consists of assigning to a processor i, the $\lceil r/p \rceil^{th}$ sequences. This distribution is easy to implement but has the drawback of leaving some processors idle if $r < p$;

Scheme 2: If $r < p$, then we assign a sequence to several processors as follows: we first assign a sequence to $\lfloor p/r \rfloor$ processors. If $r \times \lfloor p/r \rfloor < p$, then add a processor to the previous $\lfloor p/r \rfloor$ starting with the first block of processors. Although it is difficult to implement, this distribution scheme reduces the processor's idle time and ensures much better load balancing.

In our algorithm, we use the first scheme, because, in practice, the number of input sequences is usually greater than the number of processors. With this, in order to have the tables of successors of all the sequences, after the computation phase, all the processors will perform a global communication of ***all-to-all*** type. An overview of this solution is given in Figure 4 below.

Figure 4. Overview of preliminary work.

In this figure, the first step represents the task distribution process, the second, the local computation phase on each processor (*calculating the entries of the table of successors of each sequence assigned to it*) and the last, the global communication round between processors. Algorithm 1 presents local computations operations for a processor of rank i. The time and space complexity of this algorithm is summarized by lemma 1.

Algorithm 1: Local computation for preliminary work.

```
Data: The ⌈r/p⌉i^th sequences
Output: Successors tables of the ⌈r/p⌉i^th sequences
Begin
Foreach sequence x = x_1 x_2 ··· x_n do
   For j from 1 to n do
         Foreach symbol α_i ∈ Σ do
            ST[i,j] = min{m | x_m = α_i};
         End for
   End For
```

10 **End For**

11 **End**

Lemma 1: *On **p** processors, the preliminary work for the Constrained-MLCS problem with r sequences of equal length n requires* $O\left(\frac{nr|\Sigma|}{p}\right)$ *execution time and* $O(nr|\Sigma|)$ *space with a single communication round.*

Proof: According to equation (1), the computation of the entries of a successor table for a sequence of length *n* is done in $O(n|\Sigma|)$. Since each processor will evaluate at most $\lceil r/p \rceil$ sequences, it therefore requires $O(n|\Sigma|) \times \left\lceil \frac{r}{p} \right\rceil = O\left(\frac{nr|\Sigma|}{p}\right)$ time. Additionally, as each processor must have the table of successors of all the sequences, the space used is $O(n|\Sigma|) \times r$.

3.2. Parallel solution finding all common subsequences of the Constrained-MLCS problem

Among the existing sequential and parallel solutions for the MLCS problem, those based on the classical dominant point approach suffer from too much time and space consumption [28, 29]. Therefore, our aim here is to propose a parallel algorithm based on the efficient serial algorithm presented in [22]. Indeed, the main ideas for our solution are the following:

1. We will use the Leveled-DAG presented in [22] but since our goal is to have all common subsequences, we will no longer remove all outdated nodes (*nodes that can no longer contribute to the search process of other CSSs*).
2. At the last level (*the level where all MLCS are found*), all CSSs found since level 1 are returned in the form of a graph (*this representation optimizes the memory space necessary to store these CSSs because their identical prefixes are recorded in a single copy*).

Definition 8. When solving the MLCS problem using the dominant point approach, a ***subproblem*** can be defined as being the search of all the successors of a match point.

From this definition, the level *k* of a subproblem is defined as the length of the common subsequence that it generates. These subproblems can be represented as a multilevel acyclic graph defined as follows:

Definition 9. The DAG used to solve the MLCS problem is a graph in which:

1. A node is represented by:
   a. A match point $v(v_1, v_2, \dots, v_d)$;
   b. The symbol having produced the match point v;
   c. The subsequence(s) generated by the point v.
2. An edge represents the operation of calculating a successor from another match. Indeed, if v_1 and v_2 are two match points associated with symbols α_1 and α_2 respectively, then the edge $v_1 \rightarrow v_2$ represents the generation of subsequence $\alpha_1\alpha_2$;
3. The nodes that never have incoming and outgoing edges are respectively the initial and the final node;
4. The dependencies between the nodes are defined as follows:
   a. Two nodes which are not connected by an edge are mutually independent;
   b. The evaluation of one node may depend on the evaluation of another node of the same and the previous level. An example of this is shown in figure 3 where the evaluation of the node (7, 6) depends on the node of the same level (4, 3) and on the node of the previous level (5, 4);
   c. A node no longer having an incoming edge can no longer contribute to the search process of subsequences of other nodes and thus can be saved with all its generated subsequences.

From this definition, searching all CSSs consists in constructing the DAG starting from the initial node and ending at the final node by generating new nodes, saving the CSSs and removing the outdated nodes.

From the definition of the DAG the following characteristics emerge:

- *The DAG has an unknown form*: for two sets $A = \{A_1, A_2, \dots, A_r\}$ and $B = \{B_1, B_2, \dots, B_r\}$ of r sequences each, in the case where $|A_1| = |B_1|, |A_2| = |B_2|, \dots, |A_r| = |B_r|$, the DAG generated by the set A may be different from the one generated by the set B;
- *Uncertainty of generation of new nodes*: the transition from one level to another does not involve the generation of new nodes. With this DAG, it is impossible to exactly calculate the number of nodes generated by a level.

Given these characteristics, it is therefore a challenge for us to evaluate in parallel the nodes of a graph whose form is unknown and in which dependencies exist between the nodes

of the same and the previous level. This form requires a supervisor during the parallelization process because it will be necessary to synchronize the data so that each processor works on correct and updated data.

This algorithm uses the master-slave model. On a parallel machine having p processors, the processor p_0 will be the master, and the rest $\{p_1, p_2, \dots, p_{(p-1)}\}$ the slaves. The roles of each type of processor are as follows:

- **3.2.1 Master processor**:
  - Generate the first points of correspondence (*the successors of the initial node*);
  - Distribute all these nodes on slave processors;
  - Combine the partial results of slave processors after each step;
  - Based on the partial results, rebuild the next level by deleting the duplicates and reassigning the CSSs to the corresponding nodes.
- **3.2.2 Slave processor**:
  - Determine, using a specific algorithm, the nodes on which it will perform calculations;
  - Generate all the successors of each node assigned to it. In the case where a successor already exists, it is no longer generated and we just perform a redirection from the node to this successor.

Algorithm 2: Local algorithm for the master processor

```
Data: Successors table of all the r sequences
Output: The CSSs corresponding to the input sequences
Begin
    DAG = {initial node, final node};
    Succ(initial node) ← Generate_Next(initial node);
    Distribution of Succ(initial node) on the p-1 slave processors;
    Wait for the first results from the slave processors;
    Result's reception and partial recombination of DAG;
    Foreach duplicated node s of the same or previous level do
        Merge s;
    End For
    Foreach outdated node s do
        Save CSSs of s;
    End For
```

16 **End**

The CGM-based algorithm can therefore be summarized in three steps:

Step 1: The master processor generates the successors of the initial node and distributes them on all slave processors.

Step 2: Construction of the next level as follows:

Step 2.1: For each unexpanded node (*node for which the successors have not been generated*) generate its successors. If it has no successors then the final node is taken as its only successor.

Step 2.2: If $|z|$ denotes the length of the common subsequence of a node z, then for each node z having no incoming arc and for each successor s of z, do the following:

- If $|z| \geq |s|$ then replace all CSSs of s by CSSs of z which have been concatenated with the corresponding s;
- If $|z| \geq |s| - 1$ then append the CSSs of z with the corresponding symbol of s and add them to the CSSs of s;
- Save the CSSs of z.

Step 3: Repeat Step 2 until the final node remains the single unexpanded node in the graph.

Algorithms 2 and 3 describe respectively the work of the master and slave processors during each step of the search process of all the CSSs. In Algorithm 2, the procedure *Generate_Next* generates the successors of a node. It should also be noted that the sending phase of the successors of the initial node is a *one-to-all* communication.

In Algorithm 3:

- The function *compute_nb_node (E)* is used to calculate the number of nodes on which a processor will perform its jobs. In the case where this number is zero, the indexed processor will be waiting during the corresponding step;
- During the first communication phase, the processors having performed local calculations will communicate their results to the others;
- The second communication phase consists in sending to the master processor by the slave processors, information which accelerates the process of saving the CSSs.

To optimize memory space, the slave processors will perform local deletions and only the supervisor will keep all found CSSs that are the results from algorithms 2 and 3.

Algorithm 3: Local algorithm for slave processors

1 **Data**: Set E of unexpanded nodes.

```
Output: a set of generated nodes with their corresponding CSSs.
Begin
    nb_node ← compute_nb_node (E);
    For i from 1 to nb_node do
        Foreach symbol α∈Σ do
            Succ ← Succ ∪ Generate_Next(node_{i, α});
        End For
    End For
    First communication phase of all-to-all type;
    Collect information and possibility delete outdated nodes;
    Second communication phase of all-to-one type;
End
```

Figure 5 is an illustration of the overall process of our parallel algorithm searching all existing CSSs between the sequences ACTAGCTA, TCAGGTAT and CTAAGTTA. In this illustration, we have 4 processors among which 1 is a supervisor and 3 are slaves. In figures 5.a and 5.b, the green and red nodes represent the initial and the final node respectively. The step (E1) is the generation of the successors of the initial node by the supervisor. (E2) is the parallel generation of the successors of nodes of the previous level. (E3) represents the first synchronization by the supervisor. (E4) is the second local computation on slave processors and (E5) is the corresponding synchronization phase on the master. The third local computation and synchronization are represented by (E6) and (E7) respectively. After (E7), the master processor will no longer synchronize data because only one node remains in the DAG. Step (E9) shows how only active slave processors compute the last subproblem. This last step exists to create links between the last unexpanded node and the final node.

Figure 5.a First five steps to find the CSSs between three sequences with four processors.

Figure 5.b. Last ending steps for the search process of CSSs of three sequences using 4 processors.

Figure 5. Entire search process of all existing CSSs between ACTAGCTA, TCAGGTAT and CTAAGTTA sequences with 4 processors.

Lemma 2: *On **p** processors, the overall search process requires* $O\left(\frac{|D|\times|\Sigma|}{p}\right)$ *local computation time and* $O(|MLCS|)$ *communication rounds.*

Proof: The evaluation of a node requires $O(|\Sigma|)$ time. The distribution scheme used here guarantees, in the worst case, $\frac{|D|}{p}$ nodes per processor. Which gives us a local calculation time of $O(|\Sigma|) \times \frac{|D|}{p} = O\left(\frac{|D| \times |\Sigma|}{p}\right)$. Since the DAG has a maximum of $O(|MLCS|)$ levels and each level induces two communication rounds, we conclude that the entire resolution process requires $O(2|MLCS|) = O(|MLCS|)$ rounds.

3.3. Coarse-Grained algorithm validating all common subsequences

This parallel algorithm works on a set of *p* processors and can be summarized as follows:

- Definition of a data structure which will optimize the storage space used by all common subsequences. This backup will be done only on the master processor during the process of backing up CSSs;
- Distribute the constraint on each processor;
- Group all the common subsequences according to their level *(a level k groups the CSSs of length k)*;
- Validate the CSSs in parallel level after level starting by the levels of the maximum length;
- If one or more sequences of a level *k* respect the constraint, then the validation process is completed and the CSSs of the current level can be returned as MLCS.

3.3.1 Data structure used to store all CSSs

The idea here is motivated by the fact that during the process of searching for CSSs, CSSs of length *k* are found directly after searching for all common subsequences of length *k-1*. Thus, subsequences of length *k-1* are prefixes of subsequences of length *k*. The structure defined here is a graph whose preview is given in Figure 6 for the sequences ACTAGCTA, TCAGGTAT and CTAAGTTA.

3.3.2 Parallelization constraints

The previous data structure results in the following constraints:

- A level *k* is evaluated only if all CSSs of the nodes of the level *k+1* have all been validated and none of them respects the constraint;
- The evaluation of nodes belonging to the same level and of two CSSs belonging to the same nodes are mutually independent;

- The number of nodes to evaluate on a level is not predictable. Additionally, the evaluation costs of two nodes (*the number of* CSS*s to validate*) of the same level can be different.

To ensure load balancing between processors, we will not distribute nodes but, a set of CSSs.

Figure 6. Data structure storing all CSSs.

In Figure 6, a CSS of length k will be represented by the path from the source node to one node in level k. For example, the CSS "*CAGA*" is represented by the path $\{C(2,2,1) \rightarrow A(4,3,3) \rightarrow G(5,4,5) \rightarrow A(8,7,8)\}$. The evaluation of one level consists in finding all CSSs generated by nodes of the level and distributes them on each slave processor. Thus, to evaluate level 5, two processors will validate in parallel "TAGTA" and "CAGTA". With "TA" as a constraint, these two sequences will not respect it. Therefore, level 4 can be evaluated, and during the evaluation, the set of CSSs of this level will be equally distributed on processors and will be processed simultaneously. At the end of this level, only "CAGA" will respect the constraint and the validation process will be stopped.

3.3.3 Parallel algorithm

This solution is based on the previous parallelization constraints and is summarized in the five next steps:

1. The master processor distributes the graph to the others processors;
2. Evaluate the level $|MLCS|$. The evaluation of a level consists in looking for all the CSSs generated by nodes of the level and then for each CSS to carry out a validation;
    a. To evaluate a level k, we search all nodes of the level by assigning them indexes. For each node, we memorize the number *nb* of CSSs that it has generated;
    b. When this number is known, each processor calculates the number of CSSs that it will evaluate, say $\left(\frac{\sum nb}{p}\right)$. The distribution here is linear starting from the lowest indexed nodes.
    c. After the local calculations (*validation of the* CSS*s*), if there exists among the validated CSSs those that respect the constraint, then, the corresponding processor will carry out a *one-to-all* communication in order to notify the other processors that an MLCS has already been found.
3. If after Step 2, no CSSs validate the constraint, then the $|MLCS| - 1^{\text{th}}$ level is evaluated;

4. Repeat Steps 2 and 3 until level $|P_{ct}|$. Indeed, if we could reach the level $|P_{ct}|$ without having found an MLCS, since all the CSSs of length less than the constraint necessarily respect it, then the final solution will be all the CSSs of the level $|P_{ct}| - 1$.

Algorithm 4: Local validation for a processor rang k

```
Input: The graph G of CSSs and the constraint Pct
Output: A set of MLCS not containing Pct
Begin
 If k == 0 then
   Send (G);
 else
   Receive(G);
 End if
 j = |MLCS|;
While j > = length(P) do
    nb_css = compute_nb_css(G, j);
For i from 1 to nb_css do
   Validate(CSS_i, P_ct);
   If (valid(CSS_i, Pct)) then
     add (CSS_i, list);
   End if
End For
If not_empty (list) then
    Send(list);
    Break;
End if
j = j-1;
End while
End
```

Algorithm 4 is the executed pseudo code on a processor of rank *k*. In this algorithm, the procedure *validate*(CSS_i, P_{ct}) verifies if CSS_i contains the constraint string P_{ct}. This verification is performed using simple comparisons. The procedure *compute_nb_ css*(G, j) finds the number of CSSs to be validated by a processor on a level according to the principle

of Steps 2.a and 2.b described above. The complexity analysis of this algorithm and the global solution is presented in lemma 3 and theorem 1 respectively.

Lemma 3: *Algorithm 4 requires* $O\left(\frac{|CSSs|\times|MLCS|}{p}\right)$ *local computation time and* $O(|MLCS|)$ *communication rounds, where p is the number of processors used.*

Proof: This proof is similar to that of Lemma 2.

Theorem 1: *Our solution for the MLCS problem with a string exclusion constraint requires* $O\left(\frac{|D|\times|\Sigma|+|CSSs|\times|MLCS|}{p}\right)$ *local computation time on each processor,* $O(|MLCS|)$ *communication rounds and* $O\left(\frac{nr|\Sigma|}{p}\right)$ *time for preliminary work.*

Proof: This result is derived from Lemmas 1, 2 and 3.

4. Results and discussion

In this section, we present and analyze the results obtained from the implementation of our algorithms in order to validate our theoretical predictions.

4.1. Simulation environment

The following configurations were made during the experiments:

- C was used as a programming language;
- The MPI library (*OpenMPI version 3.1.1*) provided inter-processor communication;
- Tests of our programs were carried out on the Dophin cluster of the MATRICS platform of the University of Picardy Jules Verne that has the following configuration:
  - *i.* Sixty computation nodes with 28 cores each (*1680 cores in total*). Forty-eight nodes called thin nodes with 128 GB of RAM and 12 thick nodes with 512 GB of RAM;
  - *ii.* Each node is composed of 2 processors where a processor is an Intel Xeon E5-2680 V4 processor (35 MB Cache, 2.40 GHz);
  - *iii.* 2 login nodes that are not intended to perform calculations but to provide the job submission environment;
  - *iv.* The operating system used was CentOS Linux version 7.4.1708;
  - *v.* One NFS server (85 TB) with 10 Gbits;
  - *vi.* A BeeGFS (300 TB) 100 Gbits Distributed File System.
  - *vii.* All nodes are interconnected with Omni-Path links providing 100Gbps throughput.

- The data (*sets of input sequences*) used are actual biological data from the NCBI (*National Center for Biotechnology Information*) database [23]. The constraint is generated randomly according to date sizes.

During these experiments, we conducted 3 types of tests:

1. The first consists in varying the number of processors (or MPI processes) used for a fixed number of input sequences (*the size of these sequences being identic*). Thus, this number varies from 1 to 128 for a number of 100, 500, 900 and 1000 sequences of length 100;
2. The second uses 32 processors and varies the number of input sequences of fixed sizes from 3 to 900;
3. In the third test, the number of input sequences and the number of processors is fixed at 10 and 16 respectively. Only the size of the input sequences varies from 100 to 40000.

The communication time in this work is the sum of the effective data transfer time and processor idleness. The execution time is obtained by averaging the results of 5 sets of tests.

4.2. Results

Figure 7. Execution times using $p \in \{2, 4, 5, 8, 16, 32, 64, 80, 115, 120, 128\}$ processors with 100, 500 and 900 sequences respectively.

With 100, 500 and 900 sequences of length 100, Figure 7 shows the final execution time of our algorithm with different numbers of processors. This figure shows that for a reduced number of processors (*less than 32*), the speed-up rapidly increases Additionally, for a number of processors greater than 32, the speedup doesn't increase drastically. This is due to irregularities that occur during the generation of the DAG's nodes, which cause some processors to be idle during the resolution process. It can therefore be concluded from this figure that our solution is sufficiently scalable with the number of processors and input sequences.

Figure 8.a. Memory used for $p \in \{1, 4, 16, 128\}$ and $r \in \{3, 4, \ldots, 900\}$.

Figure 8.b. Evolution of speedup for $p \in \{2, 4, 16, 32, 64, 128\}$ for r=100 and n=100.

Figure 8. Evaluation of memory used and speedup of our algorithm.

An evaluation of the memory used by our algorithm is depicted in figure 8.a. The results here is the mean of the memory used by each processor during the resolution. The first observation here is that on a single processor, memory consumption of our algorithm is high (≈ 60Gb for only 50 inputs sequences of length 100). We can also observe that by increasing the number of processors, we are decreasing considerably the memory used by each. The highly memory consumption is justified by the fact that the increase of the number input sequences implies an increase not only of the number of nodes per level in the graph but also the space taken by a node.

Figure 8.b gives the speedup of our algorithm. For recall, if T_s and T_p are respectively the sequential execution time and the execution time on p processors, the speedup = T_s / T_p and the efficiency = speedup / p. We can observe in this figure that the speedup is increasing according to the number of processors used. This is a proof of the sufficient scalability of our algorithm with the number of processors.

Figure 9. Communication, computation and execution times for $p = 32$ and $r \in \{3, 4, \dots, 900\}$.

Figure 9 shows that the communication time does not increase according to the number of input sequences. This is justified by the fact that the number of communication rounds does not depend on the number of input sequences but rather on the size of the resulting MLCS. This communication time is approximately 20% of the execution time, and this is due to the idleness of processors caused by a possible unbalanced load, the master-slave alternation and the quantity of data share between processors. Ensuring load balancing is a major task for our algorithm. From figure 9, it appears that the processors work much more than they communicate.

Figure 10: Communication time, local computation and execution times for $p = 16, r = 10$ with an input sequence length $n \in \{100, 200, \dots, 9000\}$.

The results presented in figure 10 shows that the percentage of communication time increases with the length of the sequences. This augmentation is explained by the fact that the number of communication rounds depends on the number of levels of the DAG, which is a function of the size of the CSSs between the input sequences. Thus, the probability to have long CSSs increases with the size of the sequences.

Figure 11.a: Percentage of local computation and communication according to the number of sequence number using 32 processors.

Figure 11.b. Percentage of communication and local computation based on the size of the sequences on 32 processors

Figure 11: Computation time vs Communication time

Figures 11.a and 11.b illustrate the percentages of communication and computation with two configurations (*with different sizes and numbers of sequences*). These figures show us that there is more communication when the size of the sequences increases than when the number of sequences increases. This result is in agreement with our theoretical predictions ($O(|MLCS|)$).

Figure 12. Load difference between 8 processors using $r \in \{3, 4, 5 \ldots, 900\}$ input sequences.

For 8 processors, Figure 12 illustrates each of their computation loads. It shows that the master processor (*supervisor*) has a maximum load compared to the slave processors. This is justified by the fact that during the CSSs search process, a part of the DAG data synchronization work, the supervisor backups the CSSs and rebuilds them during the validation process. This figure also shows that the processor's load sufficiently decreases from one slave processor to another. This is due to the uncertainty of generating new nodes from one level to another. It can also be noted that the higher the rank of a processor, the lower its load. Indeed, during the construction process of the DAG, if the number of nodes of a level is lower than the number of processors, then our distribution scheme of the tasks to processors will attribute only one node to a processor starting with the processor rank 1.

5. Concluding remarks

In this paper, we propose a parallel solution based on the sequential algorithm of Peng and Wang [22]. The solution proposed here is subdivided into two steps: we first propose an algorithm that searches all the common subsequences between the input sequences, and we then present a second algorithm that searches among all the subsequences for those that exclude the constraint and remain of maximum length. The analysis of this solution reveals that it requires $O\left(\frac{|D|\times|\Sigma|+|CSSs|\times|MLCS|}{p}\right)$ local computation time on each processor, $O(|MLCS|)$ communication rounds and $O\left(\frac{nr|\Sigma|}{p}\right)$ time for preliminary work.

Experimental results indicate that the proposed solution is efficient and sufficiently scalable on large numbers of input sequences with long sequences. Despite its efficiency, our

algorithm can be improved by implementing another distribution scheme that ensures better load balancing between processors. Additionally, reducing the latency time of processors will greatly reduce the runtime of our solution. This algorithm can also be optimized by parallelizing the work of the master processor (using a multicores processor as a master). Further research should be conducted on a general case of this problem, for example, by considering the string-inclusion case rather than the string-exclusion case.

Acknowledgement

We thank the anonymous reviewers whose valuable comments and suggestions have significantly improved the presentation and the readability of this work.

References

[1] A. Apostolico, S. Browne and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92(1) :3–17, 1992

[2] T. K. Attwood and J.B.C. Findlay. Fingerprinting g-protein-coupled receptors. Protein Engineering, Design and Selection, 7(2) :195–203, 1994

[3] T. Cheatham, F. Amr, S. Dan and L. Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, 1995, DOI:10.1109/HICSS.1995.375451

[4] Y. Chen, A. Wan and W. Liu. A fast-parallel algorithm for finding the longest common sequence of multiple biosequences. BMC bioinformatics, 7(4): S4, 2006.

[5] F. Y. L. Chin and C. K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size, Journal of Information Processing, Volume 13 Issue 4,pp. 463-469, 1990

[6] C. Rick. New algorithms for the longest common subsequence problem. Technical Report, University of Bonn, 1994, https://dl.acm.org/citation.cfm?id=895359

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. International Journal of Computational Geometry & Applications, 6(03) pp. 379–400, 1996

[8] S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. Fundamenta Informaticae, vol 99, pp. 409–433, 2010

[9] T. Garcia, J. F. Myoupo and D. Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 349–356, 2003,

[10] D. S. Hirschberg. Algorithms for the longest common subsequence problem. Journal of the ACM (JACM), 24(4) :664–675, 1977.

[11] W. J. Hsu and M. W. Du. Computing a longest common subsequence for a set of strings. BIT Numerical Mathematics, 24(1) :45–59, 1984.

[12] J. W. Hunt and T. G Szymanski. A fast algorithm for computing longest common subsequences. Communications of the ACM, 20(5) :350–353, 1977.

[13] S. Imre. Sequence comparison: some theory and some practice. LITP Spring School on Theoretical Computer Science, Lecture Notes in Computer Science vol 377, 1987, pp. 79–92.

[14] H. Koji and H. Imai. Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima. Optimization Methods and Software, 10(2) :233–260, 1998.

[15] D. Korkin. A new dominant point-based parallel algorithm for multiple longest common subsequence problem. Technical Report TR01-148, Univ. of New Brunswick, 2001 [on line https://www.cs.unb.ca/tech-reports/documents/TR01_148.pdf]

[16] Y. Li, Y. Wang and L. Bao. Facc : a novel finite automaton based on cloud computing for the multiple longest common subsequences search. Mathematical Problems in Engineering, 2012. http://dx.doi.org/10.1155/2012/310328

[17] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma and J. Huang. A novel fast and memory efficient parallel mlcs algorithm for long and large-scale sequences alignments. 32nd International Conference on Data Engineering, pp. 1170–1181, 2016.

[18] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. IEEE Transactions on Parallel and Distributed Systems, 5(8) :835–848, 1994.

[19] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. J. Comput. Syst. Sci., 20(1) :18–31, 1980.

[20] J. F. Myoupo and D. Semé. Time-efficient parallel algorithms for the longest common subsequence and related problems. Journal of Parallel and Distributed Computing, 57(2) :212–223, 1999

[21] J. F. Myoupo, A. N. Ngomade and V. K. Tchendji, V. K. Coarse-Grained Multicomputer Based-Parallel Algorithms for the Longest Common Subsequence Problem with a String-

Exclusion Constraint. International Conference on High Performance Computing and Simulation (HPCS), pp. 1038-1044, 2018

[22] Z. Peng and Y. Wang. A novel efficient graph model for the multiple longest common subsequences (mlcs) problem. Frontiers in genetics, 8 :104, 2017.

[23] Pseudomonas aeruginosa pao1 chromosome, complete genome. [Online]. Available at: http://www.ncbi.nlm.nih.gov/nuccore/110645304?report=fasta.

[24] T. F. Smith and M. S. Waterman, Identification of Common Molecular Subsequences, J. Molecular Biology, 147 :195–197, 1981.

[25] Y-T. Tsai. The constrained longest common subsequence problem. Information Processing Letters, 88(4) :173–176, 2003.

[26] N. Ukiyama and H. Imai. Parallel multiple alignments and their implementation on cm5. Genome Informatics vol 4, pp. 103–108, 1993. https://doi.org/10.11234/gi1990.4.103

[27] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM 33(8) :103–111, 1990.

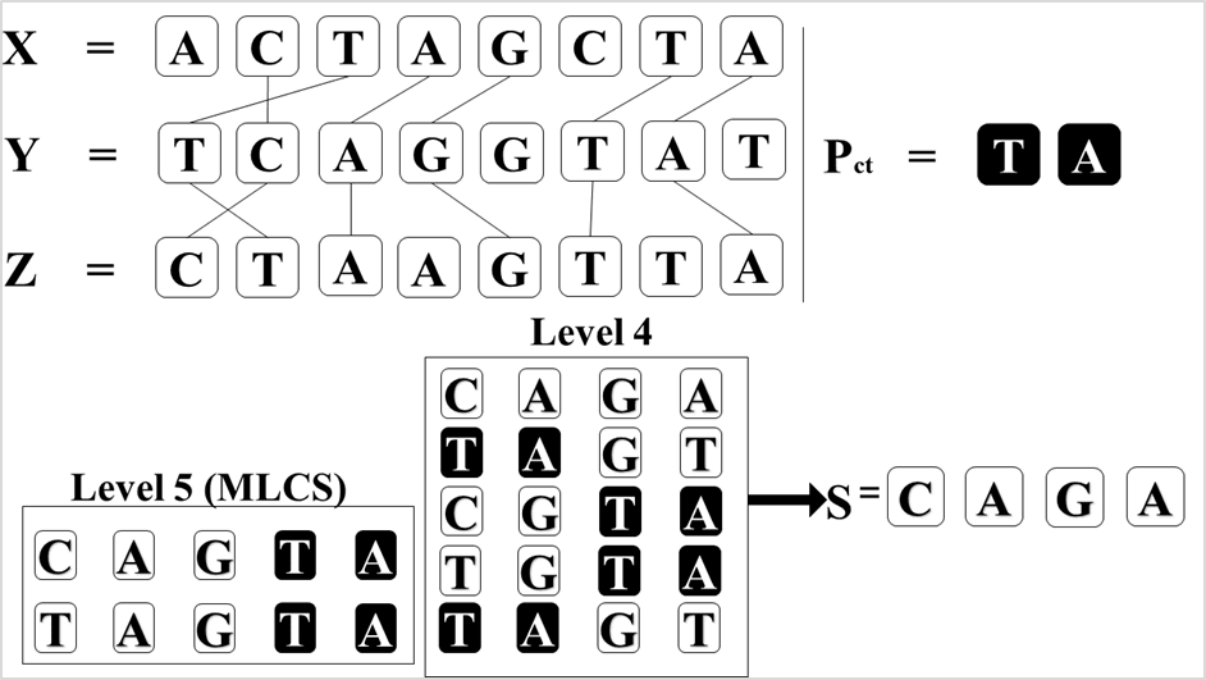
[28] Q. Wang, D. Korkin and Y. Shang. A fast multiple longest common subsequence (mlcs) algorithm. IEEE Transactions on Knowledge and Data Engineering, 23(3) :321–334, 2011

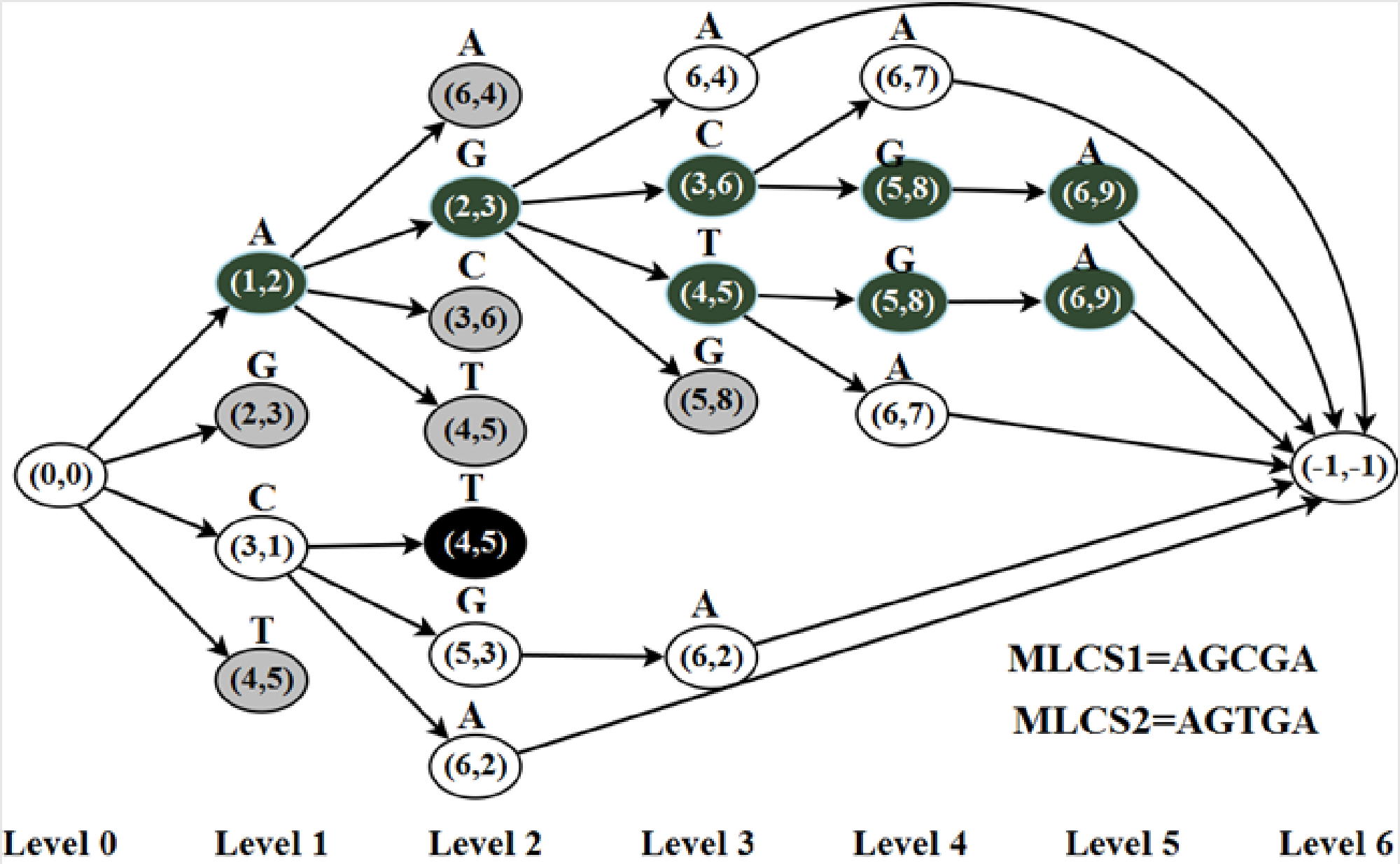
[29] Q. Wang, D. Korkin and Y. Shang. Efficient dominant point algorithms for the multiple longest common subsequence (MLCS) problem. Proceedings of the 21st international joint conference on artificial intelligence, pp. 1494-1499, 2009

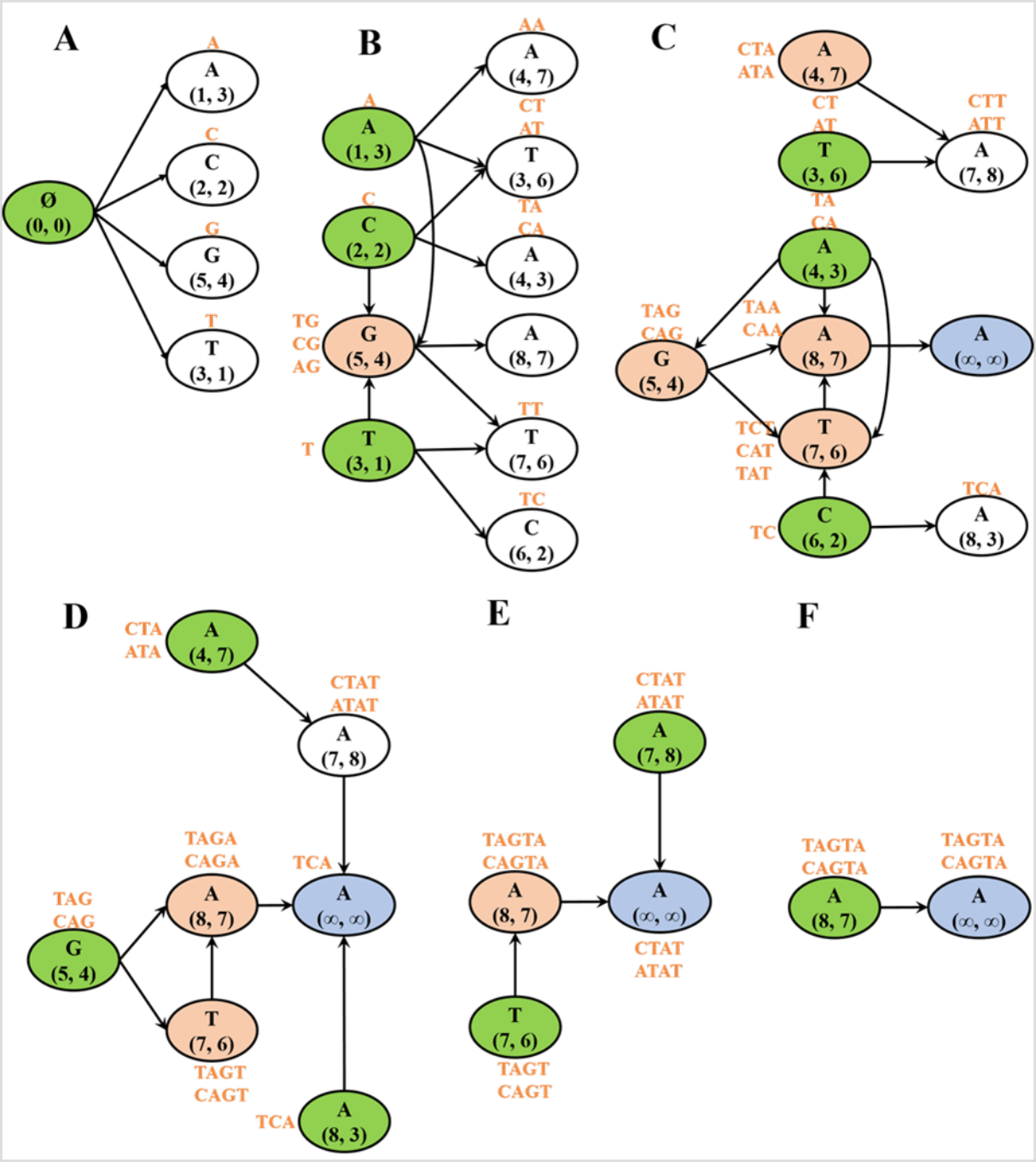
[30] Yang Jiaoyun , Xu Yun, and Shang Yi. An efficient parallel algorithm for longest common subsequence problem on GPUs. Proceedings of the World Congress on Engineering, volume 1, pp. 499–504, 2010. [on line at *http://www.iaeng.org/publication/WCE2010/WCE2010_pp499-504.pdf*]

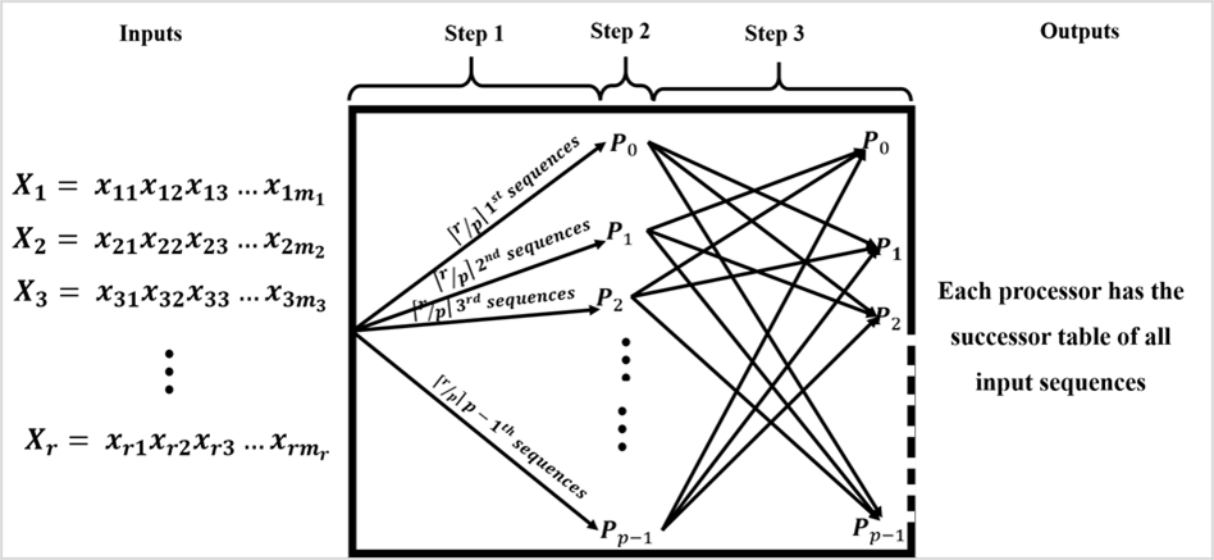
[31] J. Yang, Y. Xu, G. Sun and Y. Shang. A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization. IEEE Transactions on Parallel and Distributed Systems, 24(5) :862–870, 2013.

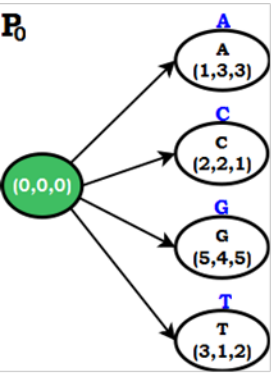
[32] D. Zhu and X. Wang. A simple algorithm for solving for the generalized longest common subsequence (lcs) problem with a substring exclusion constraint. *Algorithms*, 6(3) :485–493, 2013.


X = A C T A G C T A
Y = T C A G G T A T
Z = C T A A G T T A
Pct = T A
Level 4
C A G A
T A G T
C G T A
T G T A
T A G T
Level 5 (MLCS)
C A G T A
T A G T A
S = C A G A

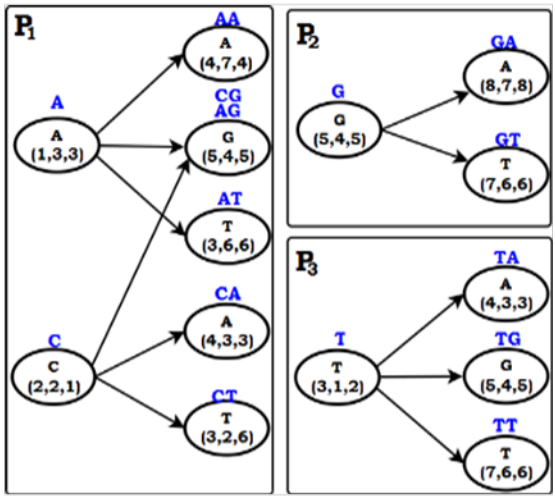

A
(6,4)
A
(6,4)
A
(6,7)
G
(2,3)
C
(3,6)
G
(5,8)
A
(6,9)
A
(1,2)
C
(3,6)
T
(4,5)
G
(5,8)
A
(6,9)
G
(2,3)
T
(4,5)
G
(5,8)
A
(6,7)
(0,0)
(-1,-1)
C
(3,1)
T
(4,5)
T
(4,5)
G
(5,3)
A
(6,2)
A
(6,2)
MLCS1=AGCGA
MLCS2=AGTGA
Level 0
Level 1
Level 2
Level 3
Level 4
Level 5
Level 6


A
Ø
(0, 0)
A
A
(1, 3)
C
C
(2, 2)
G
G
(5, 4)
T
T
(3, 1)
B
A
A
(1, 3)
C
C
(2, 2)
TG
CG
AG
G
(5, 4)
T
T
(3, 1)
AA
A
(4, 7)
CT
AT
T
(3, 6)
TA
CA
A
(4, 3)
A
(8, 7)
TT
T
(7, 6)
TC
C
(6, 2)
C
CTA
ATA
A
(4, 7)
CT
AT
T
(3, 6)
CTT
ATT
A
(7, 8)
TA
CA
A
(4, 3)
TAG
CAG
G
(5, 4)
TAA
CAA
A
(8, 7)
A
(∞, ∞)
TCT
CAT
TAT
T
(7, 6)
TC
C
(6, 2)
TCA
A
(8, 3)
D
CTA
ATA
A
(4, 7)
CTAT
ATAT
A
(7, 8)
TAGA
CAGA
A
(8, 7)
TCA
A
(∞, ∞)
TAG
CAG
G
(5, 4)
T
(7, 6)
TAGT
CAGT
TCA
A
(8, 3)
E
CTAT
ATAT
A
(7, 8)
TAGTA
CAGTA
A
(8, 7)
A
(∞, ∞)
CTAT
ATAT
T
(7, 6)
TAGT
CAGT
F
TAGTA
CAGTA
A
(8, 7)
TAGTA
CAGTA
A
(∞, ∞)

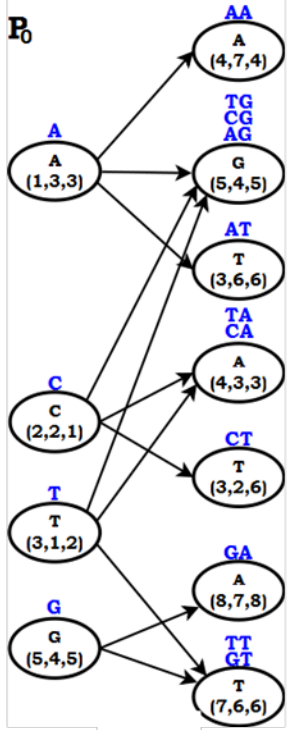

Inputs
Step 1
Step 2
Step 3
Outputs
$X_1 = x_{11}x_{12}x_{13} \dots x_{1m_1}$
$X_2 = x_{21}x_{22}x_{23} \dots x_{2m_2}$
$X_3 = x_{31}x_{32}x_{33} \dots x_{3m_3}$
$X_r = x_{r1}x_{r2}x_{r3} \dots x_{rm_r}$
$\lceil r/p \rceil 1^{st}$ sequences
$\lceil r/p \rceil 2^{nd}$ sequences
$\lceil r/p \rceil 3^{rd}$ sequences
$\lceil r/p \rceil p-1^{th}$ sequences
P_0
P_1
P_2
P_{p-1}
P_0
P_1
P_2
P_{p-1}
Each processor has the successor table of all input sequences


P0
(0,0,0)
A
A
(1,3,3)
C
C
(2,2,1)
G
G
(5,4,5)
T
T
(3,1,2)

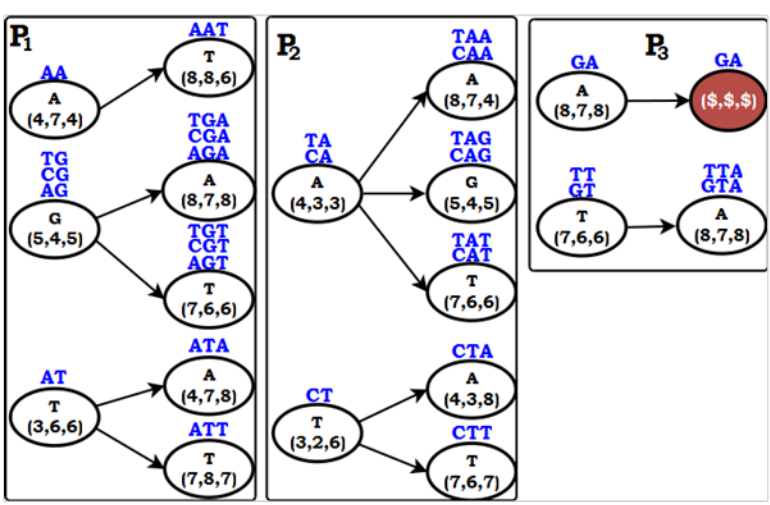
E1


P1
A
A
(1,3,3)
C
C
(2,2,1)
AA
A
(4,7,4)
CG
AG
G
(5,4,5)
AT
T
(3,6,6)
CA
A
(4,3,3)
CT
T
(3,2,6)
P2
G
G
(5,4,5)
GA
A
(8,7,8)
GT
T
(7,6,6)
P3
T
T
(3,1,2)
TA
A
(4,3,3)
TG
G
(5,4,5)
TT
T
(7,6,6)

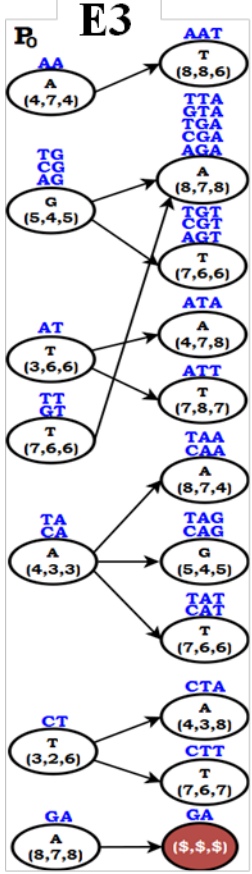
E2


P0
A
A
(1,3,3)
C
C
(2,2,1)
T
T
(3,1,2)
G
G
(5,4,5)
AA
A
(4,7,4)
TG
CG
AG
G
(5,4,5)
AT
T
(3,6,6)
TA
CA
A
(4,3,3)
CT
T
(3,2,6)
GA
A
(8,7,8)
TT
GT
T
(7,6,6)

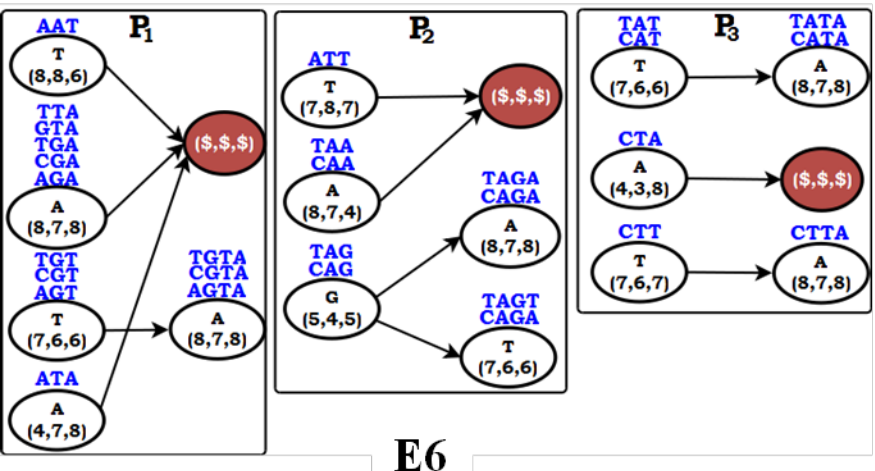
E3


P1
AA
A
(4,7,4)
TG
CG
AG
G
(5,4,5)
AT
T
(3,6,6)
AAT
T
(8,8,6)
TGA
CGA
AGA
A
(8,7,8)
TGT
CGT
AGT
T
(7,6,6)
ATA
A
(4,7,8)
ATT
T
(7,8,7)
P2
TA
CA
A
(4,3,3)
CT
T
(3,2,6)
TAA
CAA
A
(8,7,4)
TAG
CAG
G
(5,4,5)
TAT
CAT
T
(7,6,6)
CTA
A
(4,3,8)
CTT
T
(7,6,7)
P3
GA
A
(8,7,8)
TT
GT
T
(7,6,6)
GA
($,$,$)
TTA
GTA
A
(8,7,8)

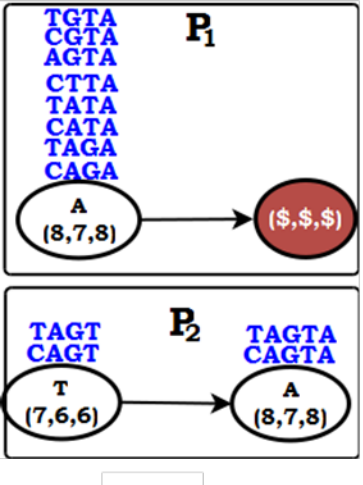
E4


P0
AA
A
(4,7,4)
TG
CG
AG
G
(5,4,5)
AT
T
(3,6,6)
TT
GT
T
(7,6,6)
TA
CA
A
(4,3,3)
CT
T
(3,2,6)
GA
A
(8,7,8)
AAT
T
(8,8,6)
TTA
GTA
TGA
CGA
AGA
A
(8,7,8)
TGT
CGT
AGT
T
(7,6,6)
ATA
A
(4,7,8)
ATT
T
(7,8,7)
TAA
CAA
A
(8,7,4)
TAG
CAG
G
(5,4,5)
TAT
CAT
T
(7,6,6)
CTA
A
(4,3,8)
CTT
T
(7,6,7)
GA
($,$,$)

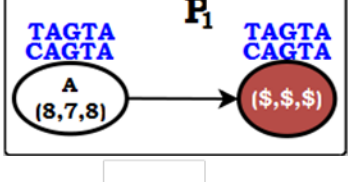
E5


P1
AAT
T
(8,8,6)
TTA
GTA
TGA
CGA
AGA
A
(8,7,8)
TGT
CGT
AGT
T
(7,6,6)
ATA
A
(4,7,8)
($,$,$)
TGTA
CGTA
AGTA
A
(8,7,8)
P2
ATT
T
(7,8,7)
TAA
CAA
A
(8,7,4)
TAG
CAG
G
(5,4,5)
($,$,$)
TAGA
CAGA
A
(8,7,8)
TAGT
CAGA
T
(7,6,6)
P3
TAT
CAT
T
(7,6,6)
CTA
A
(4,3,8)
CTT
T
(7,6,7)
TATA
CATA
A
(8,7,8)
($,$,$)
CTTA
A
(8,7,8)

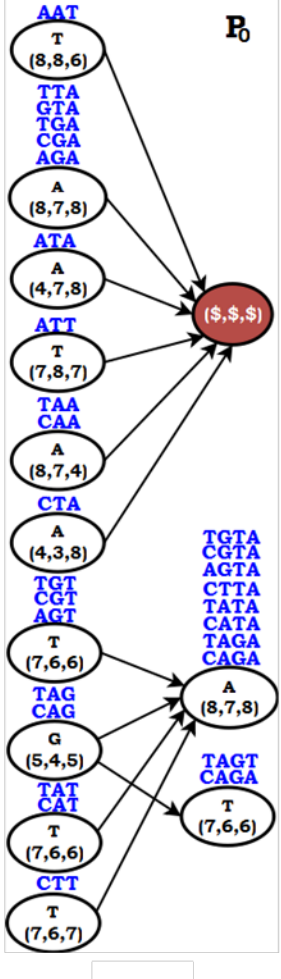
E6


P1
TGTA
CGTA
AGTA
CTTA
TATA
CATA
TAGA
CAGA
A
(8,7,8)
($,$,$)
P2
TAGT
CAGT
T
(7,6,6)
TAGTA
CAGTA
A
(8,7,8)

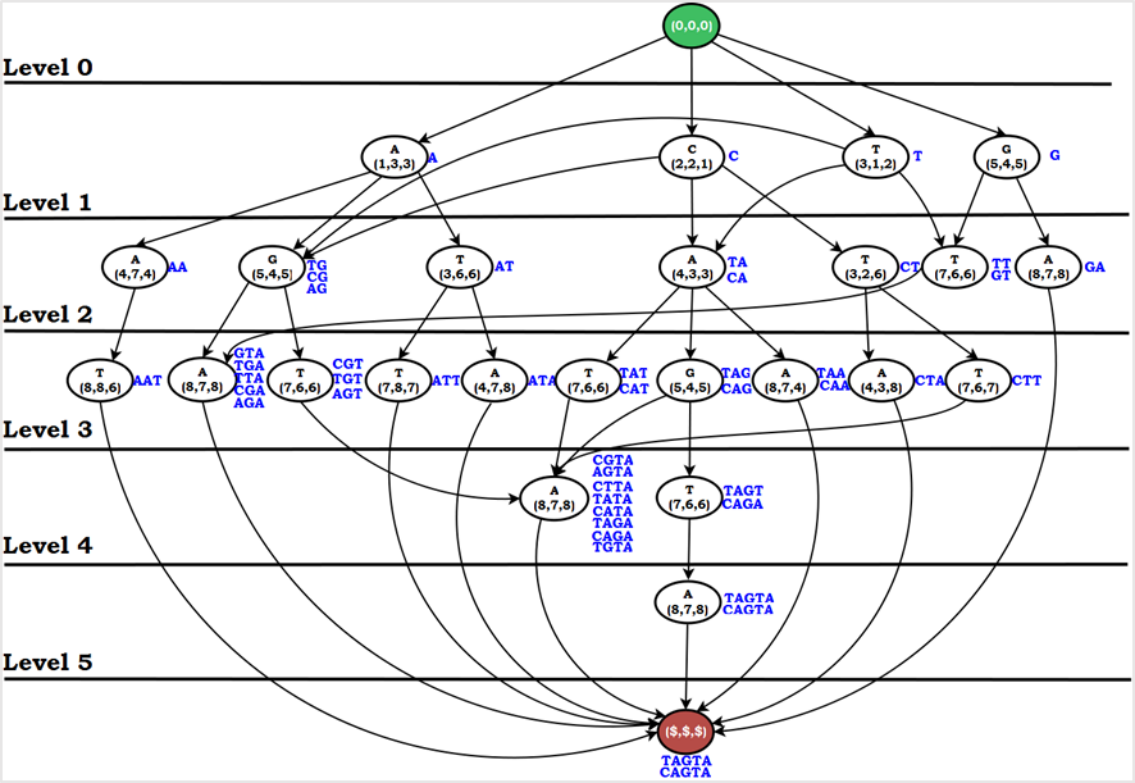
E8

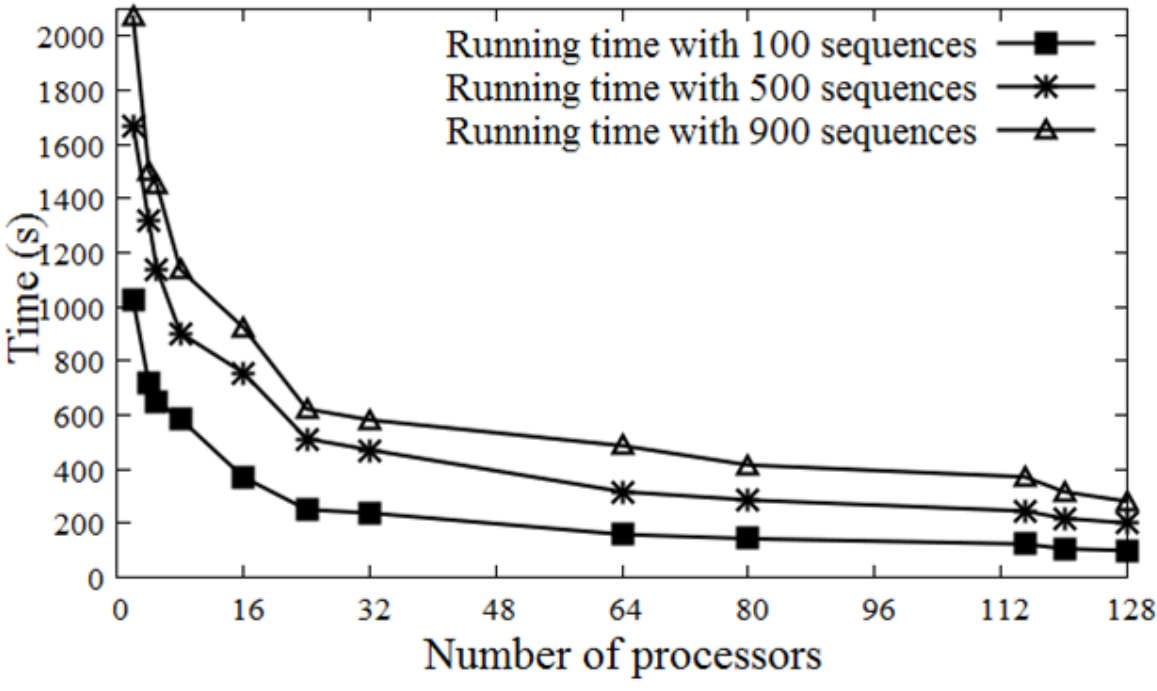

P1
TAGTA
CAGTA
A
(8,7,8)
TAGTA
CAGTA
($,$,$)

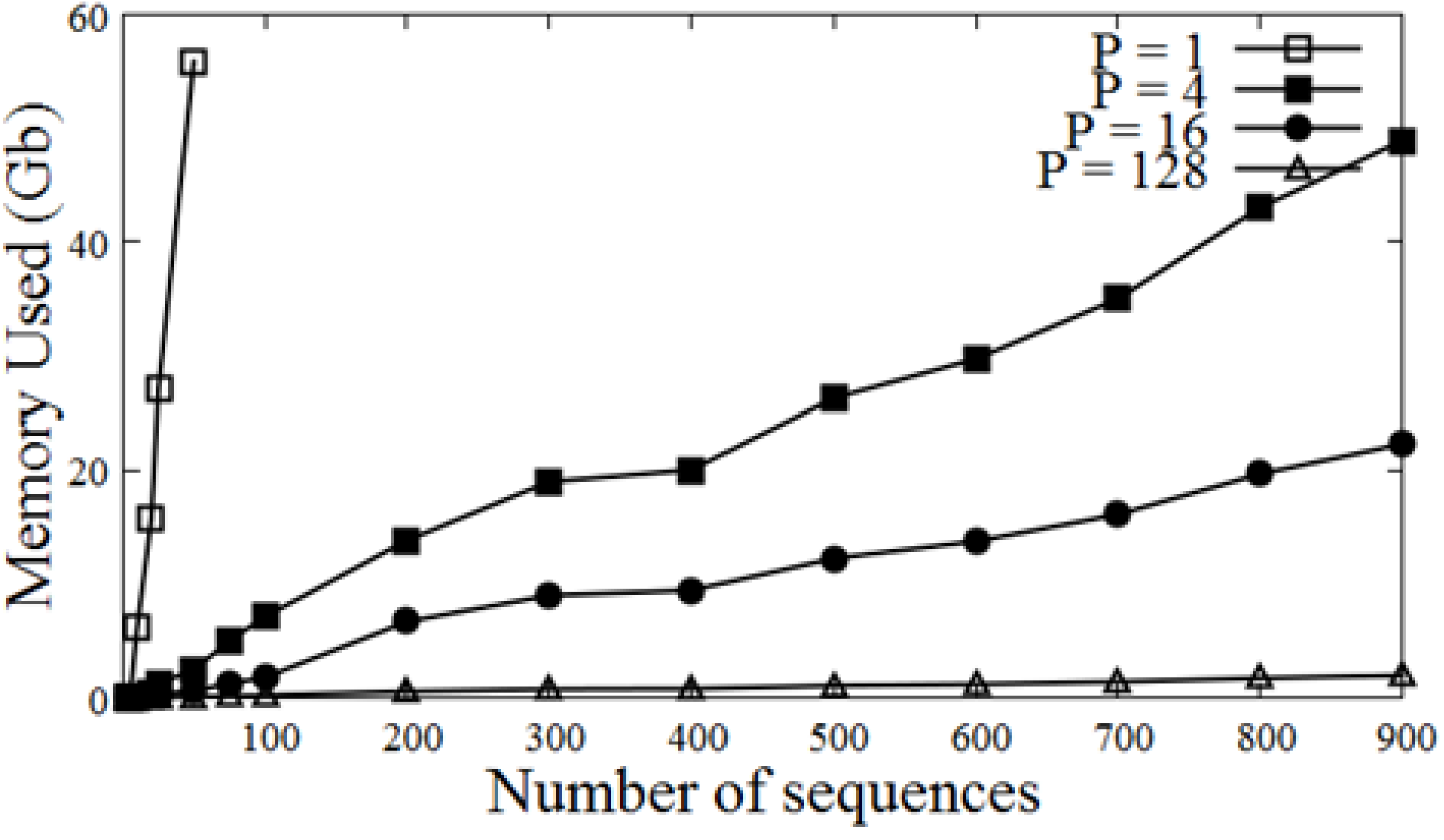
E9

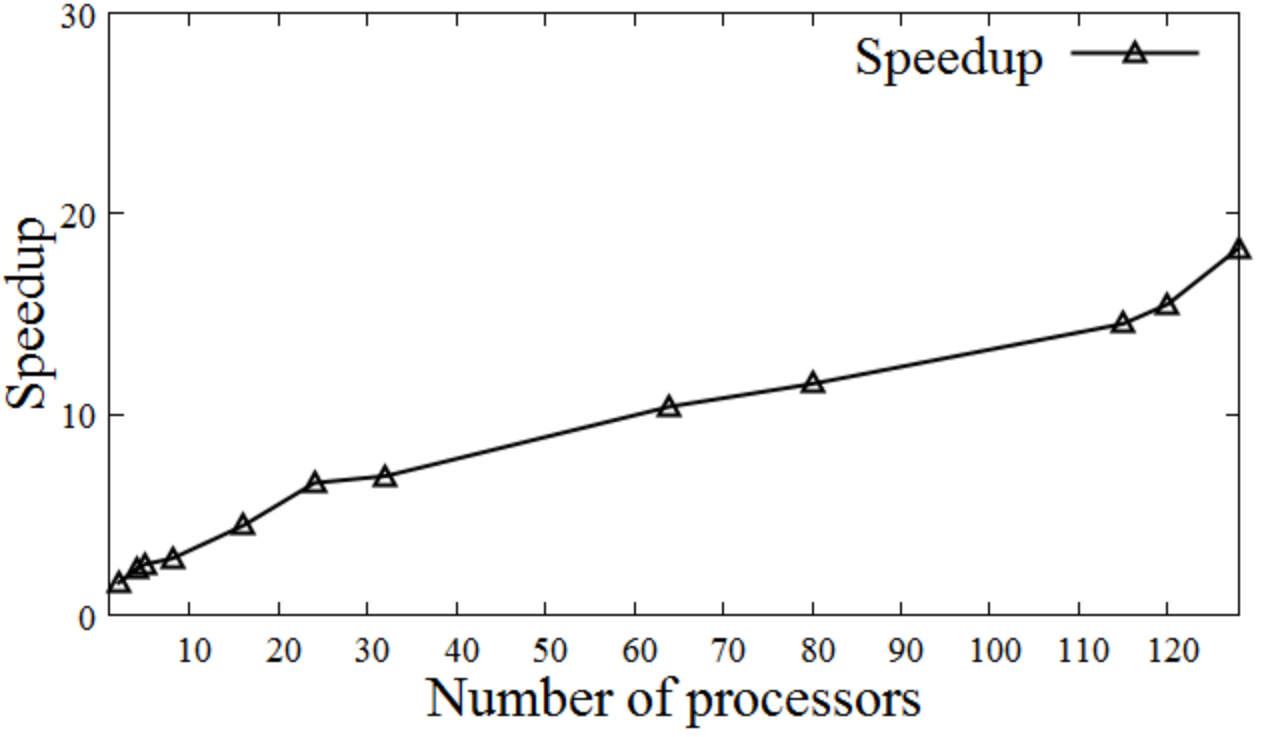

P0
AAT
T
(8,8,6)
TTA
GTA
TGA
CGA
AGA
A
(8,7,8)
ATA
A
(4,7,8)
ATT
T
(7,8,7)
TAA
CAA
A
(8,7,4)
CTA
A
(4,3,8)
TGT
CGT
AGT
T
(7,6,6)
TAG
CAG
G
(5,4,5)
TAT
CAT
T
(7,6,6)
CTT
T
(7,6,7)
($,$,$)
TGTA
CGTA
AGTA
CTTA
TATA
CATA
TAGA
CAGA
A
(8,7,8)
TAGT
CAGA
T
(7,6,6)

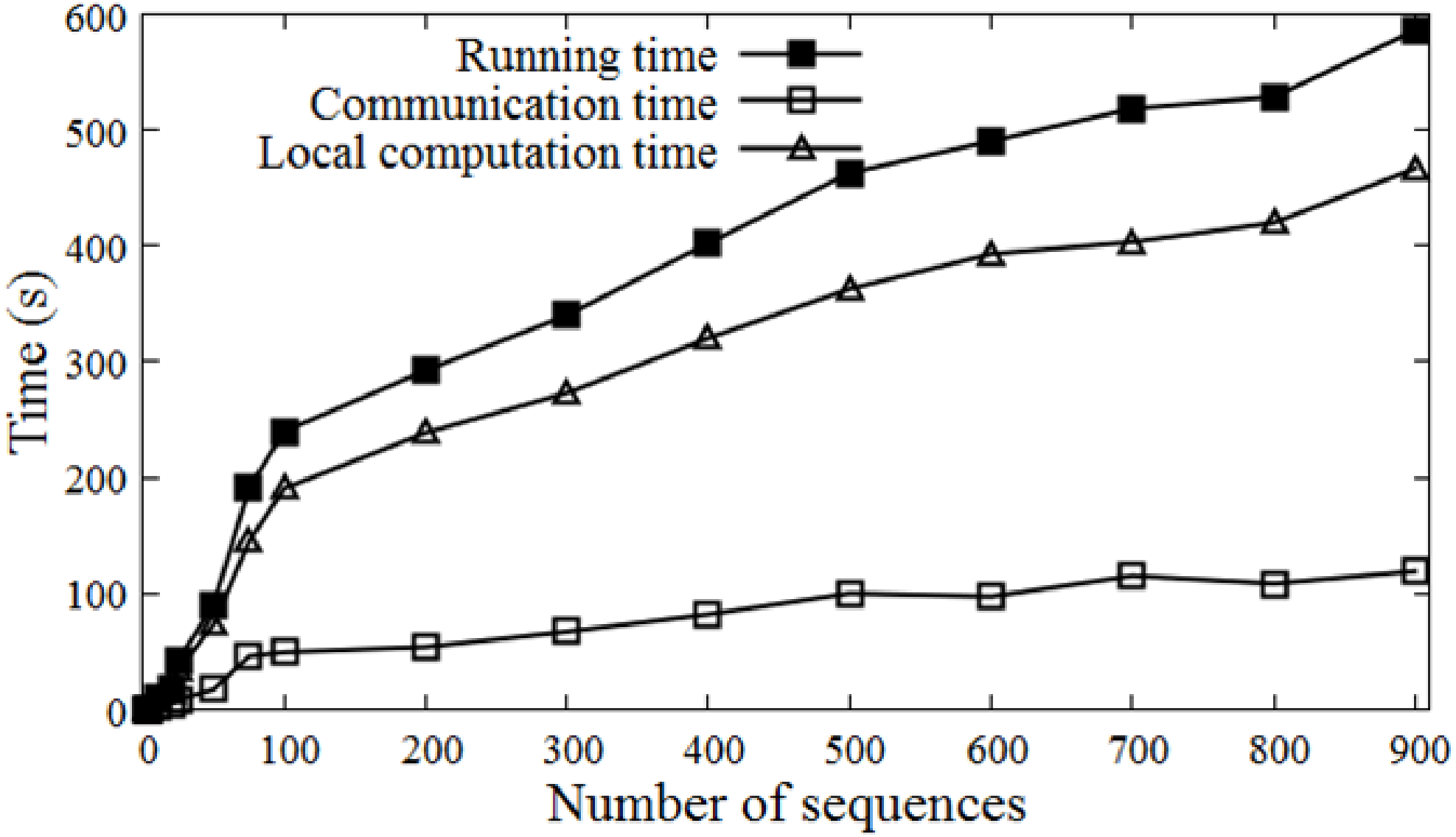
E7

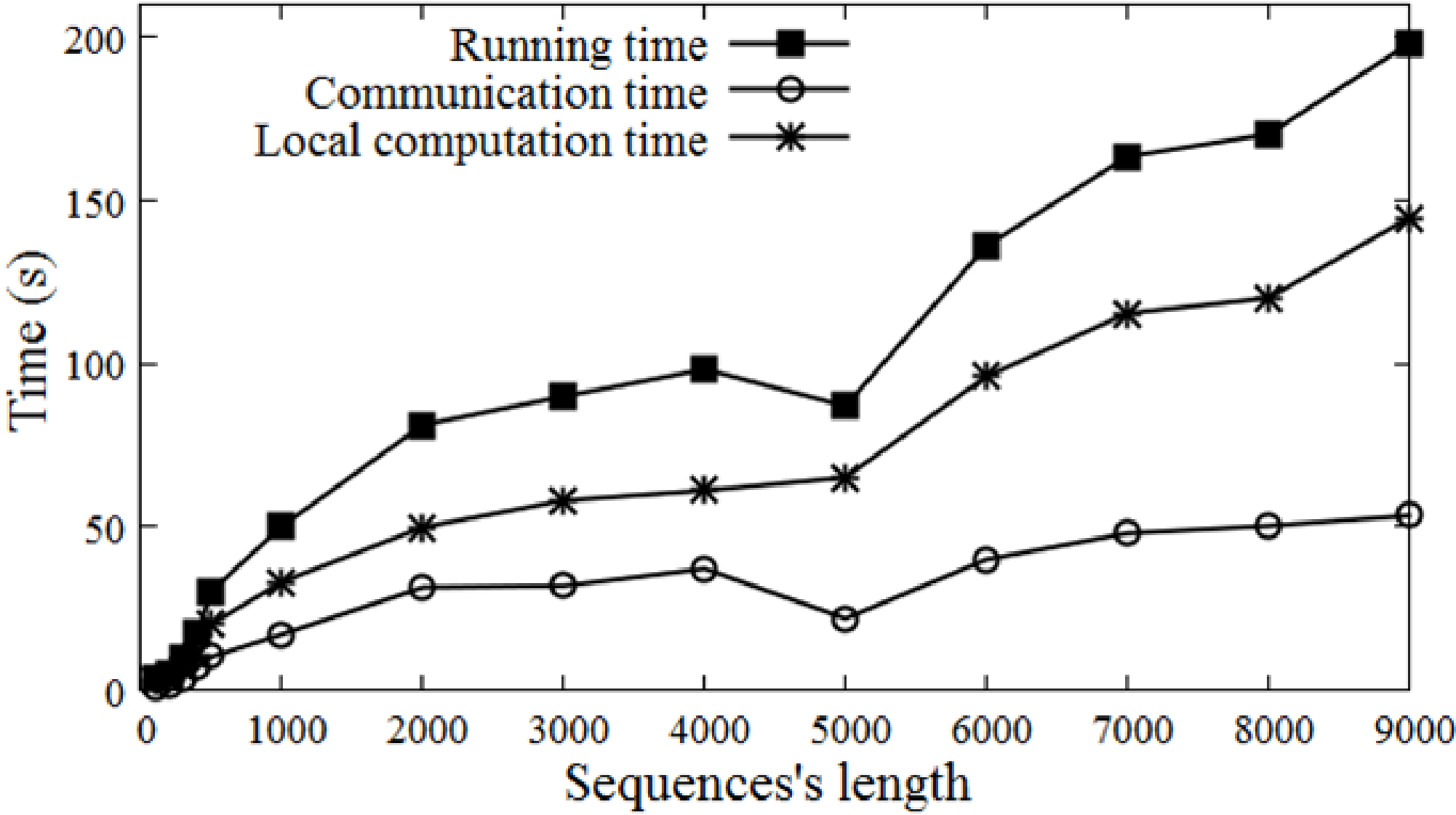

{0,0,0}
Level 0
A (1,3,3) A
C (2,2,1) C
T (3,1,2) T
G (5,4,5) G
Level 1
A (4,7,4) AA
G (5,4,5) TG CG AG
T (3,6,6) AT
A (4,3,3) TA CA
T (3,2,6) CT
T (7,6,6) TT GT
A (8,7,8) GA
Level 2
T (8,8,6) AAT
A (8,7,8) GTA TGA TTA CGA AGA
T (7,6,6) CGT TGT AGT
T (7,8,7) ATT
A (4,7,8) ATA
T (7,6,6) TAT CAT
G (5,4,5) TAG CAG
A (8,7,4) TAA CAA
A (4,3,8) CTA
T (7,6,7) CTT
Level 3
A (8,7,8) CGTA AGTA CTTA TATA CATA TAGA CAGA TGTA
T (7,6,6) TAGT CAGA
Level 4
A (8,7,8) TAGTA CAGTA
Level 5
{$,$,$}
TAGTA
CAGTA

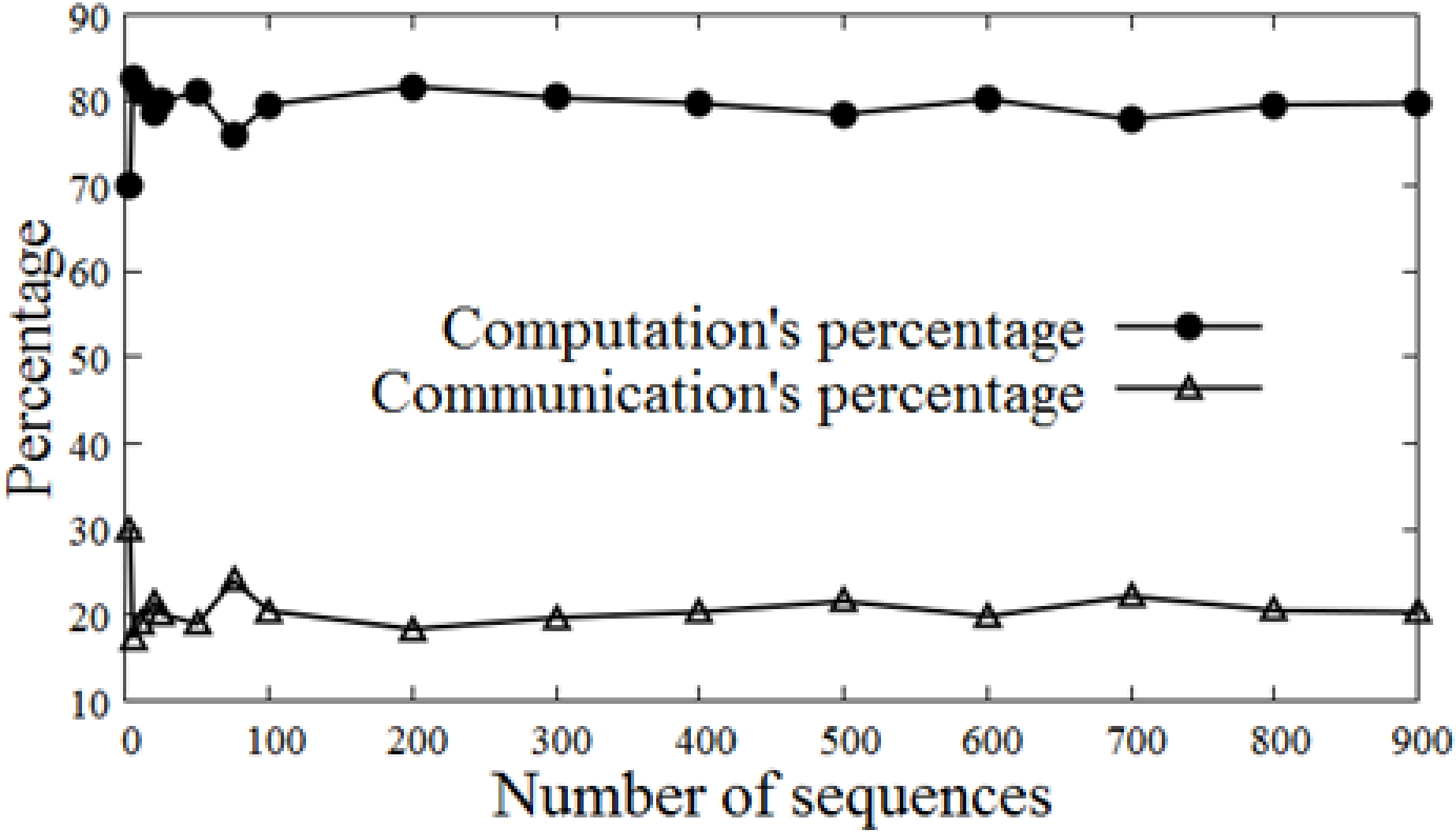

Running time with 100 sequences
Running time with 500 sequences
Running time with 900 sequences
Time (s)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
16
32
48
64
80
96
112
128
Number of processors

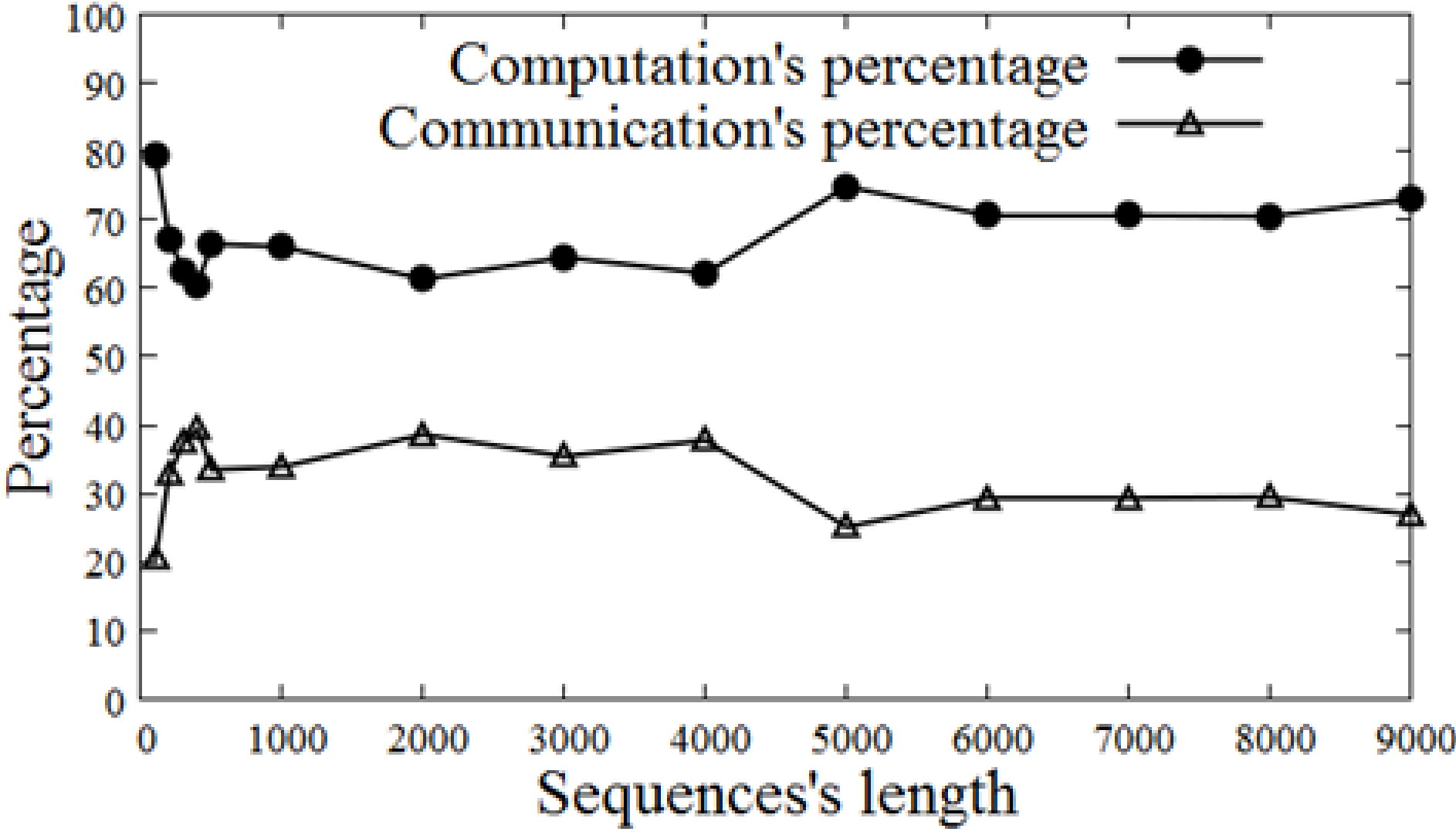

P = 1
P = 4
P = 16
P = 128
Memory Used (Gb)
0
20
40
60
100
200
300
400
500
600
700
800
900
Number of sequences

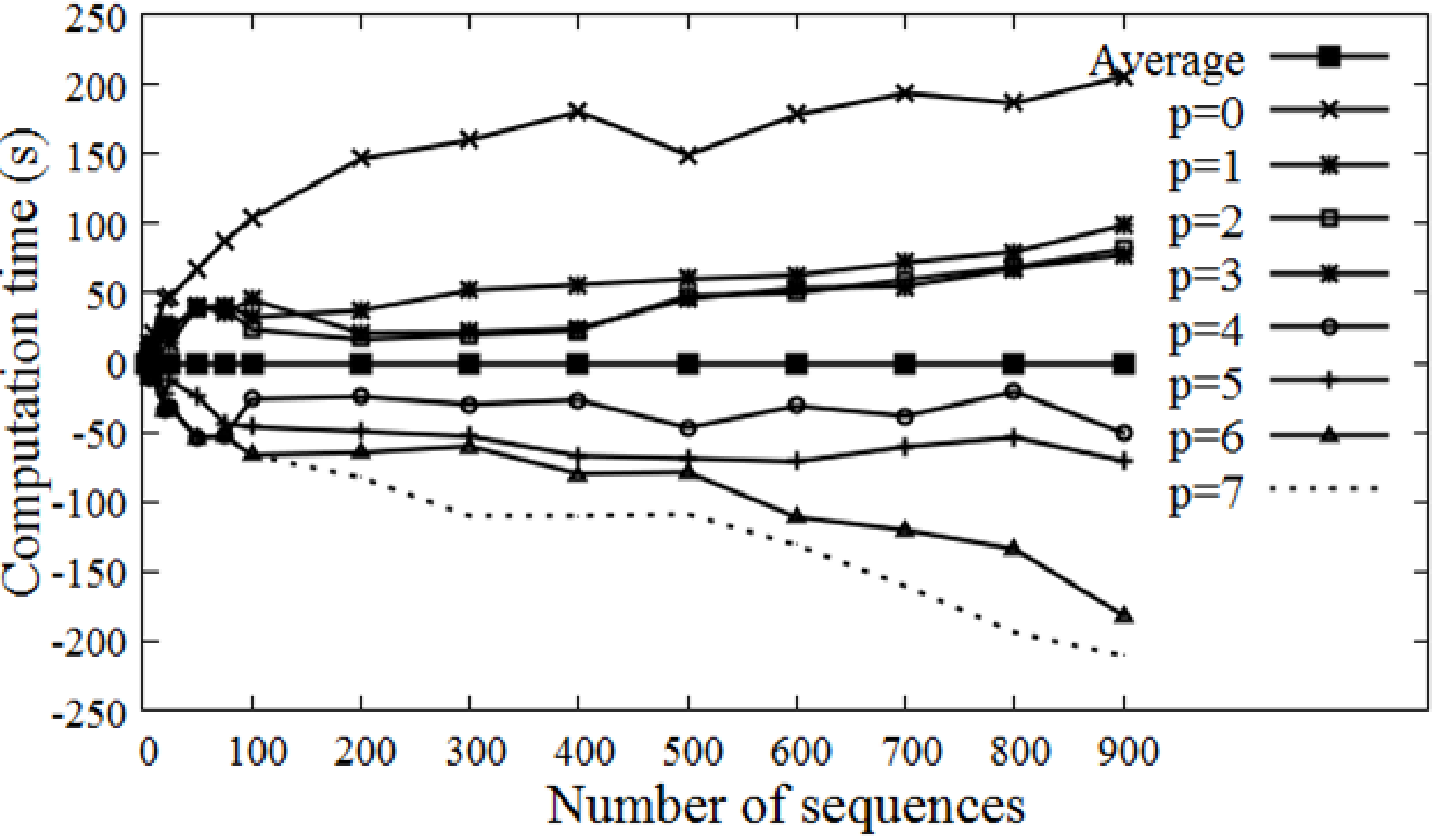

Speedup
Speedup
Number of processors
0
10
20
30
10
20
30
40
50
60
70
80
90
100
110
120


Running time
Communication time
Local computation time
Time (s)
Number of sequences
0
100
200
300
400
500
600
700
800
900


Running time
Communication time
Local computation time
Time (s)
Sequences's length
0
50
100
150
200
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Percentage
90
80
70
60
50
40
30
20
10
0
100
200
300
400
500
600
700
800
900
Number of sequences
Computation's percentage
Communication's percentage


Computation's percentage
Communication's percentage
Percentage
Sequences's length
0
10
20
30
40
50
60
70
80
90
100
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Average
p=0
p=1
p=2
p=3
p=4
p=5
p=6
p=7
Computation time (s)
Number of sequences
250
200
150
100
50
0
-50
-100
-150
-200
-250
0
100
200
300
400
500
600
700
800
900