การจำลอง L-SYSTEMS แบบเฟ้นสุ่มชนิดขนานของการเติบโตของลำต้นและกิ่งต้นไม้

นางสาวสุภาพร คำกลัด

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคณนา ภาควิชาคณิตศาสตร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2545
ISBN 974-17-3319-4

PARALLELIZING STOCHASTIC L-SYSTEMS SIMULATION OF PLANT STEM AND BRANCH GROWTH

Miss Supaporn Kamklad

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computational Science
Department of Mathematics
Faculty of Science
Chulalongkorn University
Academic Year 2002

ISBN 974-17-3319-4

Thesis Title	PARALLELIZING STOCHASTIC L-SYSTEMS SIMULATION OF PLANT STEM AND BRANCH GROWTH
By	Miss Supaporn Kamklad
Field of Study	Computational Science
Thesis Advisor	Professor Chidchanok Lursinsap, Ph.D.
Thesis Co-advisor	Associate Professor Suchada Siripant

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

..... Dean of Faculty of Science

(Associate Professor Wanchai Phothiphichitr, Ph.D.)

THESIS COMMITTEE

..... Chairman

(Assistant Professor Peraphon Sophatsathit , Ph.D.)

..... Thesis Advisor

(Professor Chidchanok Lursinsap, Ph.D.)

..... Thesis Co-advisor

(Associate Professor Suchada Siripant)

..... Member

(Associate Professor Jack Asavanant, Ph.D.)

สุภาพร คำกลัด: การจำลอง L-SYSTEMS แบบเฟ้นสุ่มชนิดขนานของการเติบโตของลำต้นและกิ่งต้นไม้. (PARALLELIZING STOCHASTIC L-SYSTEMS SIMULATION OF PLANT STEM AND BRANCH GROWTH) อ. ที่ปรึกษา: ศาสตราจารย์ ดร.ชิดชนก เหลือสินทรัพย์, อ.ที่ปรึกษาร่วม: รองศาสตราจารย์ สุชาดา ศิริพันธุ์ จำนวนหน้า 69 หน้า. ISBN 974-17-3319-4.

ระบบวิธีของ Lindenmayer หรือ L-system นั้นเป็นที่รู้จักกันมานานแล้วในฐานะทฤษฎีทางคณิตศาสตร์ที่ใช้ในการอธิบายรูปแบบการเจริญเติบโตของต้นไม้ต่างๆ โดยหลักการสำคัญของระบบวิธีนี้ก็คือการสร้างส่วนต่างๆของต้นไม้ด้วยการนำชิ้นส่วนใหม่เข้าแทนที่ชิ้นส่วนเดิมตามกฎเกณฑ์ที่ได้กำหนดไว้ และเนื่องจากการแทนที่ชิ้นส่วนนี้สามารถทำได้โดยลำพังและไม่กระทบถึงส่วนอื่นๆของต้นไม้นั้น จึงเห็นได้ว่า อาจจะนำงานทั้งหมดมาแบ่งออกเป็นส่วนย่อยๆ แล้วส่งให้เครื่องคอมพิวเตอร์หลายๆ เครื่องจัดการพร้อมๆกันได้ โดยใช้อัลกอริทึมในการประมวลผลแบบขนาน ซึ่งโปรแกรมที่ใช้ในงานวิจัยครั้งนี้เขียนขึ้นด้วยภาษาซี ประกอบกับชุดโปรแกรม Message-Passing Interface ซึ่งช่วยให้กลุ่มเครื่องคอมพิวเตอร์สามารถทำงานได้แบบขนาน และจากการวิเคราะห์หาความเป็นไปได้ในการเพิ่มประสิทธิภาพการทำงานของ Stochastic L-system นี้ ปรากฏว่า ค่า speedup factor มีค่าค่อนข้างต่ำเมื่อจำนวนรอบในการแทนค่าน้อย แต่เมื่อจำนวนรอบในการแทนค่ามากกว่า 7 พบว่า speedup factor นั้นกลับมีค่าที่น่าพอใจ โดยค่าสูงสุดของ speedup factor ในงานวิจัยครั้งนี้มีค่าประมาณ 19.6 เมื่อจำนวนรอบในการแทนค่าเท่ากับ 9 และใช้การประมวลผล 9 กระบวนพร้อมกัน.

ภาควิชา	คณิตศาสตร์	ลายมือชื่อนิสิต	
สาขาวิชา	วิทยาการคณนา	ลายมือชื่ออาจารย์ที่ปรึกษา	
ปีการศึกษา	2545	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม	

4272449323 : MAJOR COMPUTATIONAL SCIENCE
KEY WORD: L-SYSTEM / PARALLEL COMPUTING / MPI / STOCHASTIC / PLANT

SUPAPORN KAMKLAD: PARALLELIZING STOCHASTIC L-SYSTEMS SIMULATION OF PLANT STEM AND BRANCH GROWTH. THESIS ADVISOR: PROFESSOR CHIDCHANOK LURSINSAP, Ph.D., THESIS COADVISOR: ASSOCIATE PROFESSOR SUCHADA SIRIPANT, 69 pp. ISBN 974-17-3319-4.

Lindenmayer system or L-system has been recognized for a long time as a mathematical theory of plant development. The principle of system is based on a rewriting language in which each part of plant is produced by substituting along with production rules. Since substituting on each part of the plant can be performed simultaneously and independently without interfering to each other parts it is clearly seen that the whole job can be separated and performed by group of computers. In this research, a parallel algorithm for stochastic L-system has been developed and examined. The program was implemented in C programming and Message-Passing Interface package to enable the system running on virtual parallel machines. The result was investigated for feasibility to improve operating time of stochastic L-system. The speedup factor is poor for lower derivation length, but it become pretty good at derivation length greater than 7. Maximum value of speedup factor for this research is approximately 19.6 at derivation length of 9 with 9 processes operated simultaneously.

Department Mathematics Student's signature .....
Field of study Computational Science Advisor's signature .....
Academic year 2002 Co-advisor's signature .....

Acknowledgements

I would like to express my deeply gratitude feeling to numerous people who have directly and indirectly contributed to this research. I am thankful to all of them for their encouragement and support. Especially Professor Dr. Chidchanok Lursinsap and Associated Professor Suchada Siripant at The Advanced Virtual and Intelligent Computing (AVIC) Research Center, who always guided, helped and took great care of me all the time, that helped me complete this research.

I would like to thank the thesis committee, Assistant Professor Dr. Peraphon Sophatsathit and Associate Professor Dr. Jack Asavanant for their valuable advice.

I would also like to thank Mr. Nitass Sutaveepramochanon for making cluster, Mr. Paisan Tooprakai and Mr. Kasemsant Kuphanumat for solving problem on LINUX, and my friends, Miss Kingkarn Sookhanaphibarn, Mr. Maytee Bamrungrajhirun, Miss Kodchakorn Na Nakornphanom and others dear friends for their helps.

Finally, I would like to thank my parents and Dr. Wiwat Sidhisoradej for their love, care, encouragement and being with me through all the obstacles.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The study of plant morphology and plant growth has interested researchers, not only for survival reasons, but also because of the desire to understand nature and to appreciate the beauty we perceive in natural forms.

In 1960s, a biologist, Aristid Lindenmayer, presented a first model for cellular growth using string-rewriting mechanisms. This formalism, known as Lindenmayer-System or L-System, makes use of parallel replacements. It focused on the topological relationships of single cells and larger plant parts. Visualization methods for precise geometric descriptions were formulated later. One of these methods is inspired by the cursor movement commands provided by the LOGO programming language, that make the visualization of state changes easy. The popularity of this approach is derived from its lucid presentation in the groundbreaking work published in 1990 by Prusinkiewicz and Lindenmayer [1].

1.1 Problem Identification

The inspiration of this research comes from the paper entitled *Animating Plant Growth in L-system by Parametric Function Symbol* [2]. It revealed some difficulty in a rewriting language in which each part of plant was produced by substituting along with production rules. Just after a few iterative substitutions, there are too many symbols being substituted, and it consumed very long time to keep on proceeding.

In fact, the production of L-system can be cut to pieces, and each can be performed simultaneously and independently without interfering to one another. It is clearly seen that the whole job can be separated and performed by each one of group of computers. It is worthy constructing a parallel algorithm for a stochastic L-system. Exploration on the result could confirm the hypothesis which state time consumed should be improved if appropriated load balancing scheme for multiprocessor is applied. The program is developed and implemented using C programming language, associated with Message-Passing Interface package. The system is based on virtual parallel machine.

What here it is worth or not to employ a parallel computing to L-system calculation. The primary objective of this research is to determine the feasibility of applying parallel computing and MPI to stochastic L-system calculation and improving the executing time.

1.2 Background

1.2.1. L-systems

In 1968, Aristid Lindenmayer introduced L-systems, which provided a mathematical formalism for parallel grammars well adapted to the modeling of growth phenomena [1]. In 1984, Alvy Ray Smith, a computer graphics researcher

showed how L-systems could be used to synthesize realistic images. He also pointed out the relationship between the concept of Fractals and L-systems [3]. L-systems used to generate plants with or without inflorescence, cell growth and geometric patterns such as Indian kolams or mathematical ‘monsters’ such as the Von Koch or Hilbert curves. Many geometric patterns and tilling can be generated using L-systems. The problem of describing patterns and tilling using L-systems is largely unexplored.

Hammel and Prusinkeiwicz [4] extended the notation of L-systems with turtle interpretation to facilitate the construction of such objects. The extension was based on the interpretation of the entire derivation graph generated by L-systems, as opposed to the interpretation of individual words. The illustration of the proposed method by applying it to visualize the development of compound leaves, a seashell wins a pigmentation pattern, and a filamentous bacterium expanded the horizon of the application of L-system.

Samal, Peterson, and Holliday [5] recognized the naturally occurring objects that had been a difficult task in computer vision. One of the keys to recognize objects was the development of a suitable model. One type of model, the fractal, had been used successfully to model complex natural objects. A class of fractals, the L-system, had not only been used to model natural plants, but had also aided in their recognition. They extended the work in plant recognition using L-systems in two ways. Stochastic L-systems were used to model and generate more realistic plants. Furthermore, to handle the complexity of recognition, a learning system was used that automatically generated a decision tree for classification. Results indicated that the approach used here has great potential as a method for recognition of natural objects.

Chua mei Chen and Hsu Wen Jing [6] presented a type of formal language called L-system which was developed by Lindenmayer (1968). Similar to formal languages, L-systems defined a method by which a string of symbols could be rewritten or parsed into another string using a set of rewrite rules. X-machines were generalized state automata. They took another look at this way of generating plants and provided a convenient way of taking component L-systems that exhibited some properties and combining them using a x-machine so that each L-system contributed its properties or behavior. The result was a plant, which had some characteristics of each of its components.

Stefanovski, Loskovska and Mihajlov [7] introduced a model for implementing recursive objects, defined by L-systems, in a ray tracing based system for realistic visualization. The model was based on constructive solid geometry (CSG). Recurrent CSG-graphs were used for internal representation of recursive objects. The graphs allowed one to build up the scene during visualization, i.e. generating only those primitive objects which might be affected by the ray. The reduction of objects minimized the necessary memory space. For better performance, an efficiency scheme with super-bounding volumes was used.

Schaefer, Jr. [8] described the genetic programming paradigm using Lindenmayer system re-writing grammars which was proposed as a means of specifying robot behaviors for autonomous navigation of mobile robots in uncertain environments. The concise nature of these algorithms and their inherent expansion capabilities held promises as a method of overcoming communication bandwidth and time-of-flight limitations in the transmission of navigation, guidance, and control algorithms of planetary rovers. The results of this early research showed much

promise as a viable programming technique for evolutionary robotics and embedded systems.

1.2.2. Plant Model

Lintermann and Deussen [9] presented a rule-based approach combined with traditional geometric modeling techniques that allowed easy generation of many branching objects including flowers, bushes, trees, and even nonbotanical objects. A set of components describing structural and geometrical elements of plants mapped to a graph that formed the description of a specific plant and generated the geometry. Users got immediate feedback on what they had created-geometrical parameters, tropisms, and free-form deformations could control the overall shape of a plant. They demonstrated that their method handled the complexity of most real plants.

Fracchia and Ashton [10] described that the investigation of mechanisms responsible for the morphogenesis of complex biological organisms was an important area in biology. *P. Patens* was an especially suitable plant for this research because it was a rather simple organism, facilitating its observation, yet it possessed developmental phenomena analogous to those which occurred in higher plants, allowing the extrapolation of hypotheses to more complex organisms. The visualization consisted of three components: biological data collection, computer modeling (using L-systems), and model verification. The simulated developmental process was quite realistic and provides an excellent means for verifying the underlying hypotheses of morphogenesis.

Yi-Cheng Lin and Sarabandi [11] investigated a coherent scattering model for tree canopies based on a Monte Carlo simulation of fractal generated trees. In contrast to the incoherent models based on the radiative transfer theory, the present model was capable of preserving the relative phase of individual scatterer which gave rise to the coherent effects and predicting the absolute phase of the backscattered field or equivalently the scattering phase center. In the procedure for Monte Carlo simulation, the first tandem generation of tree architectures was implemented by employing the Lindenmayer systems (L-systems), a convenient tool for creating fractal patterns of botanical structures. Since the generating code of tree structures was faithful in preserving the fine features of the simulated tree types, this study provided an efficient approach to examine the effects of tree structures on the radar backscatter. After generating a tree structure, the electromagnetic scattering problem was then treated by considering the tree structure as a cluster of scatterers comprised of cylinders (trunks and branches) and disks (leaves) with specified position, orientation, and size. The scattering solution was obtained by invoking the single scattering theory for a uniform plane wave illumination. In this solution scattering from individual tree components when illuminated by the mean field was computed and then added coherently. The mean field at a given point within the tree structure included the attenuation and phase change due to the scattering and absorption losses of vegetation particles. Finally, the backscattering coefficients were simulated at different frequencies based on the results of the Monte Carlo simulations obtained from a large number of independent trees.

Mock [12] presented the *Wildwood* project in which a genetic algorithm was applied to a simplified L-system representation in order to generate artificial-life style plants for virtual worlds. Acting as a virtual gardener, a human selected which plants to breed, producing a unique new generation of plants. An experiment

involving a simulation-style fitness function was also performed, and the virtual plants were adapted to maximize the fitness function.

Chuai-aree, S., Siripant, S. and Lursinsap,C. [13] animated the plant growth by using the iteration of L-system, but at each time step of development the plant model was not smooth and continuous. They proposed an animating plant growth in L-system by parametric functional symbols to the length, size and position of each component of the plant. The developments of plant growth seemed to be smoother and more naturals as well as realistic. This prototype could be used to generate the realistic model of any plant based or bracketed L-system.

1.2.3. Parallel Computing

Poovarawan, Y. and Uthayopas, P. [14] presented the overview of High Performance Computing, the required components, and its applications. Current state of the HPC researches and facilities in Thailand had also been reviewed along with the HPC related research conducted in Kasetsart University. In summary, HPC was a technology that had an impact on Thailand competitiveness. Yet, much more qualified manpower and broader recognition of the field were seriously needed. Afterwards, when more understanding were obtained via an analysis and visualization, the experiments could be conducted to verify the simulation results or collect more information to fine-tune the model. The clear advantages of this second approach were faster turn-around time and much less cost. However, the computing power needed to solve these kinds of problems was enormous. This was the rational for recent emerging of Computational Science, which was the study of techniques and tools to tackle compute-intensive applications.

Uthayopas, P., Angskun, T. and Maneesilp, J. [15] introduced their experiences in constructing a parallel computer from cluster of cheap PCs. This parallel computer could be programmed using PVM and MPI standard message passing interface. Applications developed on this machine were portable to most commercial supercomputer such as IBM SP System, SGI Power Challenge. The steps of system integration and the application of this system were presented. They found that key factor in building this kind of system was the integration of suitable hardware and software systems. This technology was important in providing affordable supercomputing power for research and academic communities in Thailand.

Lin, Hsu and Lee [16] introduced the single-step-searching problem, which was defined as follows. They were given a graph where each vertex was associated with a weight. Assume that every edge of graph was of equal length. A fugitive might be hidden in any edge. They were asked to assign searchers to vertices to search the entire graph in one step such that no fugitive could escape. The cost of a searching plan was related to the weights of the vertices in which the searchers were initially located. Their goal was to minimize the cost of the searching plan. A parallel algorithm based upon The EREW model was proposed to solve this problem. Their algorithm applied the tree contraction technique. The critical point was that they had to transform a general tree into a binary tree, including pseudo-nodes, in order to apply this tree contraction technique. A new algorithm was devised to solve the problem on the transformed binary tree. It could be proved that this new algorithm was correct, as it produced a correct solution for the original tree. Their algorithm had an optimal speed-up.

1.3 Objective of the Research

Due to parallel algorithm and implementation of the algorithm under MPI environment is capable to reduce time for production rules substitution under stochastic L-system in plant stem and branch growth. The primary aim of this study is to obtain a parallel technique to cut down time for the production rules substitution. Anyway, most of operation in L-system calculating is string substitution, which is very fast action for any computer. It is very important to decide it is worth or not to involve parallel computing to L-system calculation. This question is primary objective of this research, it is to determine feasibility of applying parallel computing and MPI to stochastic L-system calculation and improving the executing time.

Chapter 2

Lindenmayer Systems

Lindenmayer Systems – or L-systems – were conceived as mathematical models of plant graphically growth. Fundamentally, their emphasis was on plant topology, but with limited ability to comprehensive higher plant modeling. Several geometric interpretations of L-systems were proposed with a view to turning them into significance tool for plant modeling.

2.1 Rewriting systems: concepts and history

The primary concept of L-systems is that of rewriting. In general, it is an importance technique for illustrating characteristic of complex objects by successively replacing parts of an initial object using a set of rewriting rules. The *snowflake curve* (Figure 2.1) is one of well-known model constructed using L-systems performed by von Koch in 1905 [17]. This model can be initiated with two fragments, an *initiator* and a *generator*. In each stage of rewriting, all straight lines will be replace with an imitation of the *generators*, adjusted their scale to have the same end point. Then next step will go on.

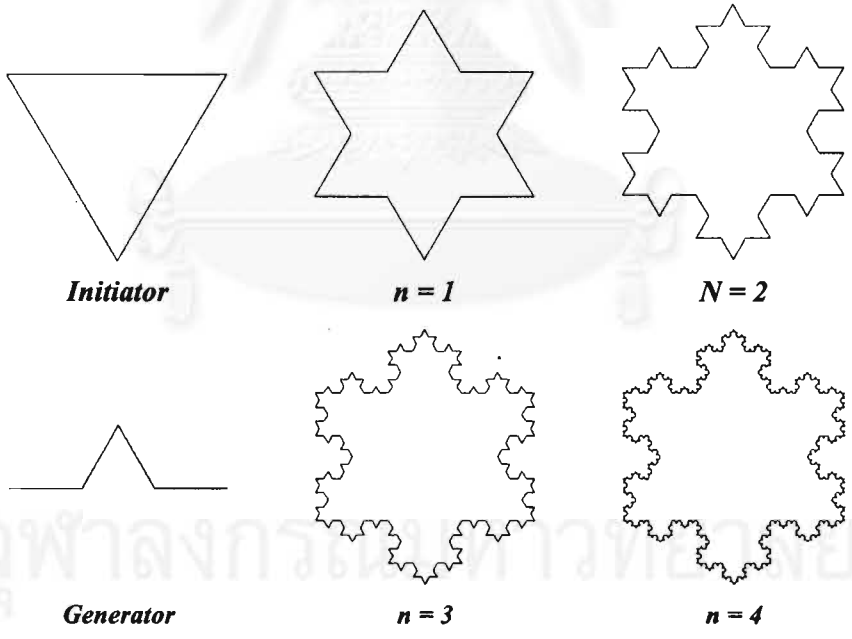


Figure 2.1 Construction of the Snowflake curve.

The most extensively studied and the best-understood rewriting systems operate on character strings. The first formal definition of such a system was given at the beginning of this century [18], but a wide interest in string rewriting was spawned in the late 1950s by Chomsky's work on formal grammar [19]. The concept of rewriting was applied to describe the syntactic features of natural languages. A few years later Backus and Naur introduced a rewriting-based notation in order to provide

a formal definition of the programming language. Their accomplishments have had various applications in computer science.

In 1968, Aristid Lindenmayer, a biologist, introduced a new type of string rewriting mechanism, called L-systems [19]. There is an essential difference to Chomsky grammars in the method of applying productions. In Chomsky grammars productions are applied sequentially, whereas in L-systems they are applied in parallel and simultaneously replace every letters in a given string. This difference exhibits the biological motivation of L-systems. Productions are proposed to capture cell divisions in which numerous divisions occur at the same time. Parallel production application has a fundamental impact on the formal properties of rewriting systems.

2.2 Deterministic and Context-Free of L-systems

Deterministic and context-free of L-systems, commonly called DOL-systems, is an elementary class of L-systems. It can be comfortably illustrated as an example that introduces the main concept in intuitive terms.

Firstly, let's consider strings compose of two symbols ***a*** and ***b***, which is possible occur many times in a string. Each symbol is associated with a *rewriting rule*. The rule $a \rightarrow ab$ means that any letter ***a*** in a string is to be substituted by the string ***ab***, and the rule $b \rightarrow a$ means that the any letter ***b*** is to be substituted by ***a***. The rewriting proceeding begin from a string called *axiom*.

Next, assume that the string is composed of only one letter ***b***. In the beginning step, the axiom ***b*** is substituted by ***a***, using rewriting rule $b \rightarrow a$. Then, in the second step, the letter ***a*** is substituted by ***ab***, using production $a \rightarrow ab$. The word ***ab*** are simultaneously rewritten in the next step. Thus, ***a*** is substituted by ***ab***, ***b*** is substituted by ***a***, and the result is string ***aba***. Similarly, the string ***aba*** produce string ***abaab*** which in turn outgrows to string ***abaababa***, then ***abaababaabaab***, and so on as illustrated in Figure 2.2

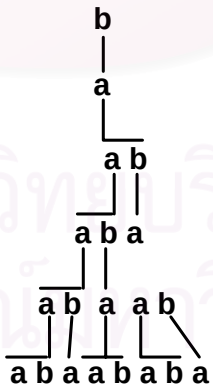


Figure 2.2 Elementary example in Deterministic and Context-Free L-system

Mathematical definitions describing DOL-systems and their operation are given below [1]. Let V denote an alphabet, V^* the set of all words over V. A string OL-system is an ordered triplet $G = \langle V, \omega, P \rangle$ where V is the alphabet of the system, $\omega \in V^+$ is a nonempty word called the axiom and $P \subset V \times V^*$ is a finite set of productions. A production $(a, \chi) \in P$ is written as $a \rightarrow \chi$. The letter a and the word χ

are called the predecessor and the successor of this productions, respectively. It is assumed that for any letter $a \in V$, there is at least one word $\chi \in V^*$ such that $a \rightarrow \chi$. If no production is explicitly specified for a given predecessor $a \in V$, the identity production $a \rightarrow a$ is assumed to belong to the set of productions P. An OL-system is deterministic (noted DOL-system) if and only if for each $a \in V$ there is exactly one $\chi \in V^*$ such that $a \rightarrow \chi$.

Let $\mu = a_1 \ldots a_m$ be an arbitrary word over V. The word $\nu = \chi_1 \ldots \chi_m \in V^*$ is directly generated by μ, noted $\mu \rightarrow \nu$, if and only if $a_i \rightarrow \chi_1$ for all $i = 1, \ldots, m$. A word ν is generated by G in a derivation of length n if there exists a developmental sequence of words $\mu_0, \mu_1, \ldots, \mu_n$ such that $\mu_0 = \omega$, $\mu_n = \nu$ and $\mu_0 \rightarrow \mu_1 \rightarrow \ldots \rightarrow \mu_n$.

Another commendable example illustrating the operation of DOL-systems is the formalism of simulating the development of a fragment of a multicellular filament. This system can be found in the blue-green bacteria Anabaena catenula and various algae [1]. The symbols a and b represent cytological states of the cells (their size and readiness to divide). The subscripts l and r indicate cell polarity, specifying the positions in which daughter cells of type a and b will be produced. The development is described by the following L-system:

$$
\begin{array}{lcll}
n & = & 4 & \\
w & : & a_r & \\
p_1 & : & a_r & \rightarrow \; a_l\, b_r \\
p_2 & : & a_l & \rightarrow \; b_l\, a_r \\
p_3 & : & b_r & \rightarrow \; a_r \\
p_4 & : & b_l & \rightarrow \; a_l
\end{array}
$$

Beginning from only one cell a_r (the axiom), the following succession of string is generated:

a_r

$a_l\, b_r$

$b_l\, a_r\, a_r$

$a_l\, a_l\, b_r\, a_l\, b_r$

$b_l\, a_r\, b_l\, a_r\, a_r\, b_l\, a_r\, a_r$

$\ldots$

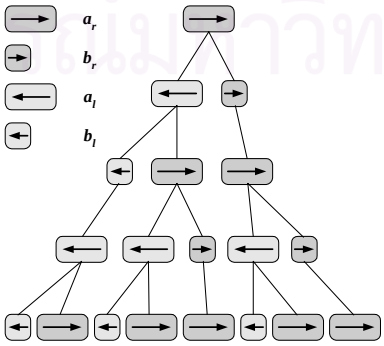


Figure 2.3 Development of a filament (Anabaena catenula) simulated using a DOL-system.

Under a microscope, the filaments appear as a sequence of cylinders of various lengths, with ***a***-type cells longer than ***b***-type cells. The corresponding schematic image of filament development is shown in Figure 2.3. Note that due to the discrete nature of L-systems, this model does not capture the continuous expansion of cells between subdivisions.

2.3 Turtle interpretation of strings

In place of illustration of more complex plants, more sophisticated graphical explanation of L-systems in needed. Frijters and Lindenmayer [20], and Hogeweg and Hesper [21] published the first available figures of this direction in 1974. In both cases, L-systems were used essentially to determine the branching topology of the modeled plants. The geometric aspects, such as the lengths of line segments and the angle values, were added in a post-processing phase. Smith [22], who established the capacity of L-systems for realistic image synthesis, subsequently extended the results of Hogeweg and Hesper.

The basic idea of turtle interpretation is given as follows. A status of the turtle is defined as a triplet (x,y,α), where the (x,y) represent the turtle's *position* in Cartesian coordinates, and the angle α represents the *direction* in which the turtle is facing. Given the *step size* d and the angle *increment* δ, the turtle can respond to commands represented by the following symbols in Table 2.1.

Table 2.1 Two-dimensional Turtle interpretations.

Symbols	Turtle Response
F	Moves forward one step with distance d. The turtle status shift from (x,y,α) to (x',y',α) by transformation equation $x' = x + d\cdot\cos\alpha$ and $y' = y + d\cdot\sin\alpha$, with a line drawing from (x,y) to (x',y').
f	Moves forward one step with distance d. The turtle status shift from (x,y,α) to (x',y',α) by transformation equation $x' = x + d\cdot\cos\alpha$ and $y' = y + d\cdot\sin\alpha$, without any line drawing.
+	Rotate counterclockwise by angle δ. The new turtle status is $(x,y,\alpha+\delta)$.
-	Rotate clockwise by angle δ. The new turtle status is $(x,y,\alpha-\delta)$.

One of commendable examples for illustrating the application of turtle is *quadratic Koch Island*. The representation of the system is given by string ν, the initial state of the turtle is (x_0,y_0,α_0) and fixed parameters d and δ. The turtle interpretation of ν is the figure (set of lines) drawn by the turtle in response to the string ν in Figure 2.4 b. Precisely, this method can be applied to illustrate the strings which are generated by L-systems. For example, Figure 2.5 presents four

approximations of the *quadratic Koch Island*. These figures were obtained by interpreting strings generated by the following L-system:

$$\omega \quad : \quad F\text{-}F\text{-}F\text{-}F$$
$$p \quad : \quad F \quad \rightarrow \quad F\text{-}F\text{+}F\text{+}FF\text{-}F\text{-}F\text{+}F$$

The images correspond to the strings obtained in derivations step of *0* to *3*. The angle increment δ is fixed at *90°*. The step length *d* is decreased by four times between subsequent images, to maintain the distance between the endpoints of the successor polygon equal to the length of the predecessor segment.

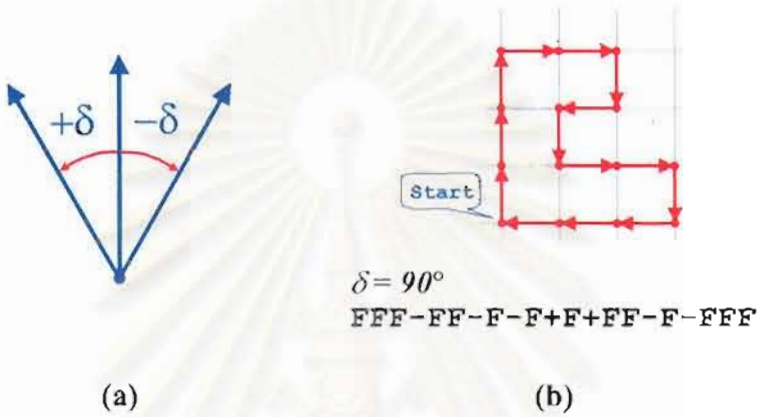


Figure 2.4 The Turtle Interpretation, in case of *step size d* is one unit, and *angle increment* δ is *90°*

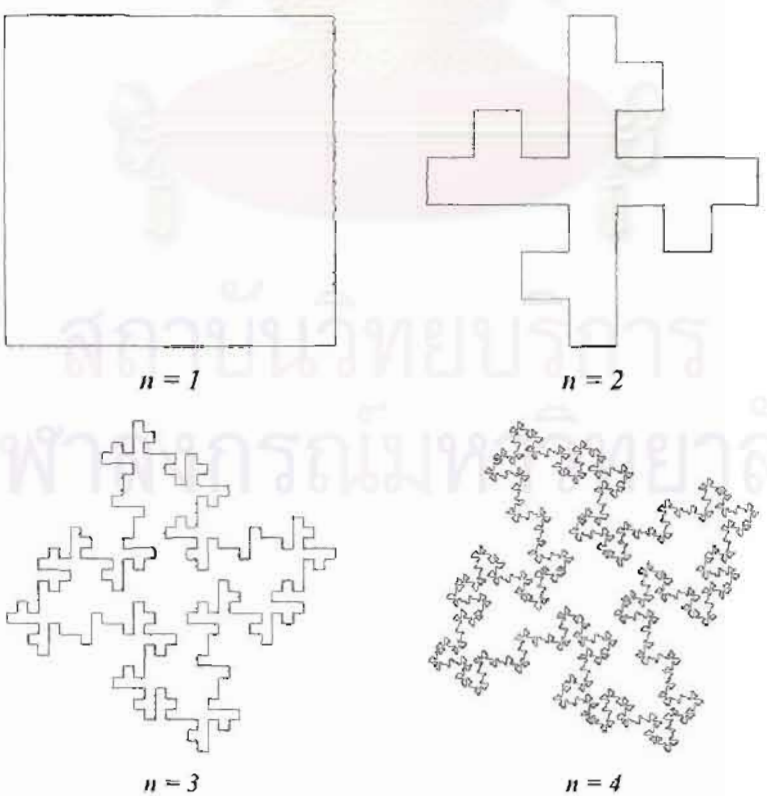


Figure 2.5 Generating a *quadratic Koch Island*.

2.4 Branching structures

The turtle interpretation of string as a sequence of line segments has worthy capability to result various type of drawing in L-system, but it is not more than just a single line. Nevertheless, most of the plant in the nature is branch structure. Thus, a mathematical description of tree-like shapes and the methods for generating are needed.

2.5 Axial tree

A *rooted tree* is defined as a set of edges that are labeled and directed. The edge sequences form paths from a distinguished node, called *root*, to the *terminal nodes*. One of subtypes of rooted tree is an *axial tree.* Each node of an *axial tree* has at most one outgoing straight-distinguished segment. Where other remaining edges are called lateral segment.

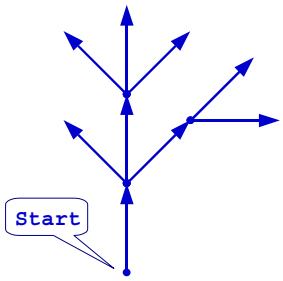
2.6 Bracketed OL-systems

The definition of a tree in L-systems does not particularize the data structure for representing an *axial tree*. One possibility is to use a list representation with a tree topology. Alternatively, the *axial tree* can be represented using strings with bracket [1]. An extension of turtle interpretation is required for strings with brackets and the operation of bracketed L-systems. Two more symbols are introduced to delimit a branch. The turtle interpretation is described in Table 2.2.

Table 2.2 Two-dimensional Turtle interpretations (extension).

Symbols	Turtle Response
[	Push the current status, position and orientation, of the turtle onto a pushdown stack. The other attributes such as the color, width and style of lines might be saved to stack as well.
]	Pull a status from the stack and assign it as the current status of the turtle. No other action to perform.

An example of an axial tree and its string representation are shown in Figure 2.6. Derivations in bracketed OL-systems proceed as in OL-systems with out brackets. The brackets replace themselves. Some examples of two-dimensional branching structures generated by bracketed OL-systems are shown in Figure 2.7.



$\delta = 45°$

F[+F][-F[-F]F]F[+F][-F]F

Figure 2.6 Bracketed string representation of an axial tree.

(a)

n **= 5, δ = 25.7°**

axiom: **F**

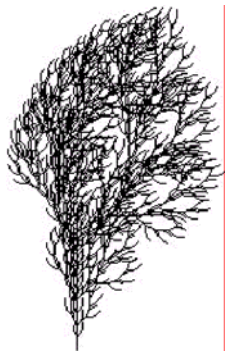
rule: **F→F[+F]F[-F]+F**

(b)

n **= 5, δ = 20°**

axiom: **F**

rule: **F→F[+F]F[-F][F]**



(c)

n **= 4, δ = 22.5°**

axiom: **F**

rule: **F→ FF-[-F+F+F]+[+F-F-F]**

(d)

n **= 7, δ = 20°**

axiom: **X**

rules: **F→FF**

X→F[+X]F[-X]+X

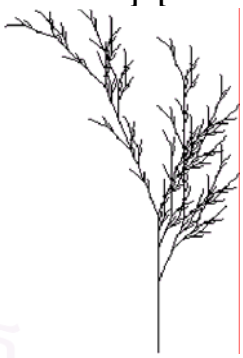
(e)

n **= 7, δ = 25.7°**

axiom: **X**

rules: **F→FF**

X→F[+X][-X]FX



(f)

n **= 5, δ = 22.5°**

axiom: **X**

rules: **F→FF**

X→F-[[X+X]+F[+FX]-X

Figure 2.7 Examples of plant-like structures generated by bracketed OL-systems.

2.7 Stochastic L-systems

It can be seen that all plants generated by the same deterministic L-system have to be exactly alike. An attempt to pose them in the same picture would produce a remarkable uniformity. In order to prevent this effect, it is necessary to introduce specimen-to-specimen variations that will preserve the general aspects of a plant but will modify its details.

The principle of Stochastic L-systems is variation achieved by randomizing the turtle interpretation, the L-system, or both. Anyway, randomization of the interpretation alone has a limited effect. Due to the modification of geometric aspects of a plant, such as the stem lengths and branching angles, is unable to change underlying topology. On the other hand, stochastic application of productions may affect both the topology and the geometry of the plant.

A stochastic OL-system is an ordered quadruplet $G_\pi = \langle V, \omega, P, \pi \rangle$. The alphabet V, the axiom ω and the set of productions P are defined as in an OL-system. Function $\pi : P \rightarrow (0,1]$, called the *probability distribution*, maps the set of productions into the set of production probabilities. It is assumed that for any letter $a \in V$, the sum of probabilities of all productions with the predecessor an is equal to 1.

The derivation $\mu \Rightarrow \nu$ is called a stochastic derivation in G_π if for each occurrence of the letter a in the word μ the probability of applying production p with predecessor a is equal to $\pi(p)$. Thus, different productions with the same predecessor can be applied to various occurrences of the same letter in one derivation step.

A simple example of a stochastic L-system is given as following.

$$
\begin{array}{lcll}
\omega & : & F & \\
p_1 & : & F & \xrightarrow{0.33} \quad F[+F]F[-F]F \\
p_2 & : & F & \xrightarrow{0.33} \quad F[+F]F \\
p_3 & : & F & \xrightarrow{0.34} \quad F[-F]F
\end{array}
$$

The production probabilities are listed above the derivation symbol $\rightarrow$. Each production can be selected with approximately the same probability of $1/3$. Examples of branching structures generated by this L-system with derivations step of 5 are shown in figure 2.8. Note that the results generated through Stochastic L-Systems are different for every derivation process. It makes these structures look like different specimens of the same plant species.

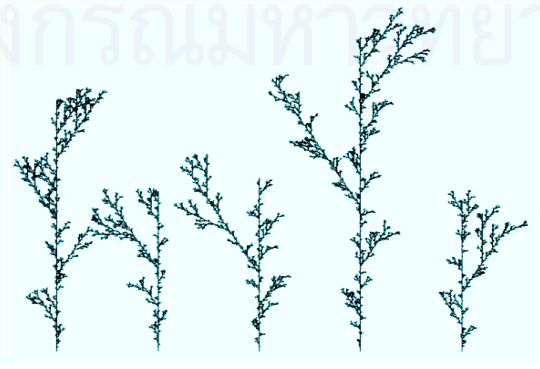


Figure 2.8 Stochastic branching structures

Chapter 3

Parallel Computing

3.1 Parallel Programming Paradigm

Even though, in present, a personal computer has become more and more powerful, but many of scientific fields still require much more computing power than can be achieved by a single personal computer. A supercomputer might be a reliable solution. However, it might be too expensive for some research group. Fortunately, cluster computing is another alternative potential solution that provides high computing power with an acceptable price.

A cluster composes of a group of personal computers connected through a high-speed network. It differs from a typical computer network in that each node in the cluster collaborates with each other to solve a problem. The cluster is capable of providing an equal computing power to a supercomputer, but it is possible to increase the probability of a node failure.

High performance computing with cluster requires a special application designed. This application must be able to divide tasks and provide these divided tasks to each node in cluster. The development of such an application requires communication library that helps developer to send and receive data between each node. In order to make application portable and independent from any specific communication library, a standard library of internode communication is required. Message-Passing Interface or MPI is one of the most considered standard libraries in cluster computing applications.

3.2 Networks

In parallel computing, a network refers to the connection of processors and memories together in a parallel architecture. Ideally, one wants each processor to be connected to any other processor, but it becomes uneconomical when the number of processors is large (Figure 3.1), and many solutions have been established. There are various applicable network topologies. A brief summary of each topology is given in the following sections.

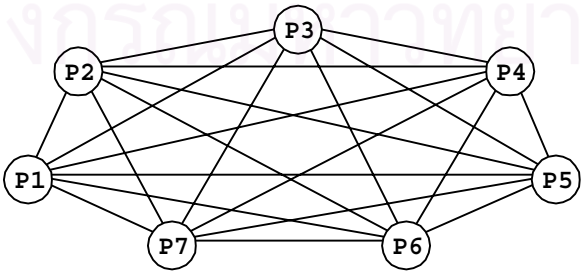


Figure 3.1 There are $\frac{P \cdot (P-1)}{2}$ links, where P is a number of processes. It make connecting all processors together becomes seriously expensive.

3.2.1. Linear topology

In Linear topology, processors are organized in an ascending order from 0 to P-1. Excluded the first and the last processors, each processor has two neighbors, its predecessor and its successor. Although the topology is simple, the data must pass through a number of processors in order to reach the destination. This results in long communication delays, especially between the first and the last processors.

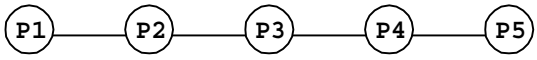


Figure 3.2 Linear topology.

3.2.2. Ring Topology

A ring topology can be obtained by connecting the first and the last processors of Linear topology to each other. A ring can be uni-directional (the communication is established in only one direction, clockwise or counter-clockwise) or bi-directional (the communication is established in both directions). The ring structure can still cause long communication delays between components.

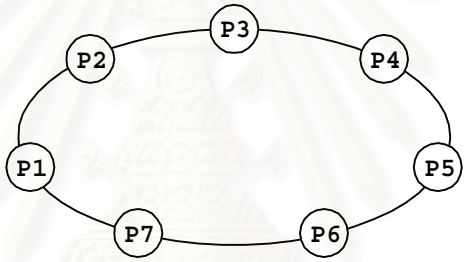
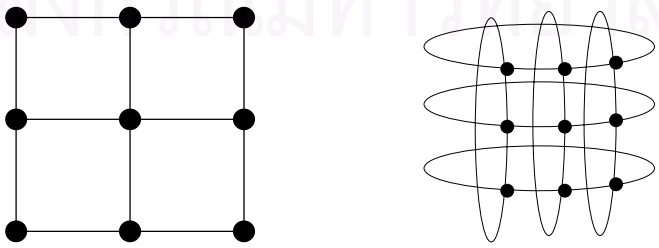


Figure 3.3 Ring Topology.

3.2.3. Two-Dimensional Mesh

In two-Dimensional Mesh topology, the processors are arranged in a two-dimensional matrix. Each processor is connected to four neighbors (top, down, left and right). There are two sub-types of this topology. One is the mesh with wrap-around connections between processors in the same row or column. The other is the mesh without such wrap-around connections. The mesh topology can be generalized to more than one dimension. In an *n-dimensional mesh*, each processor is connected to two neighbors in each direction.



(a) (b)

Figure 3.4 (a) No wrap-around connections, (b) Wrap-around connections

3.2.4. Binary Tree

1. **Static tree network** composed of processors connected resembling a complete binary tree. Each processor is connected to one parent in previous level and two children in succeeding level. An exception is root processor has no parent connection, and leaf nodes have no any children (Figure 3.5 a).

2. Dynamic tree network has the binary tree structure similar to static tree network. However, nodes at intermediate levels and root are switching elements, and just only leaf nodes are processors (Figure 3.5 b).

To communicate among processors, a message is sent from processor up the tree until it reaches the processor or the switch at the root of the smallest subtree containing both the sources and destination processors. Then the message is sent down the tree toward the destination processor.

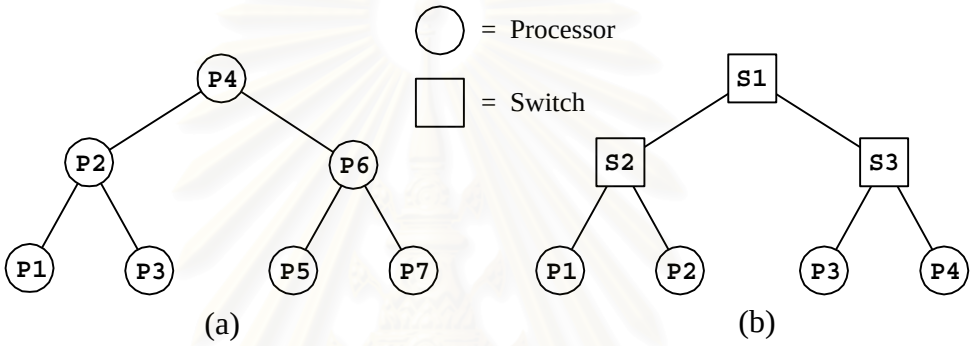


Figure 3.5 (a). Static binary tree interconnection network with 7 processors, (b) Dynamics binary tree interconnection network with 4 processors and 3 switches.

3.2.5. Star

A star-connection network, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. The star-connected network is similar to bus-based networks. Communication between any pair of processor is routed through the central processor, just as the shared bus forms the medium for all communication in bus-based network. The central processor is the bottleneck in the star topology.

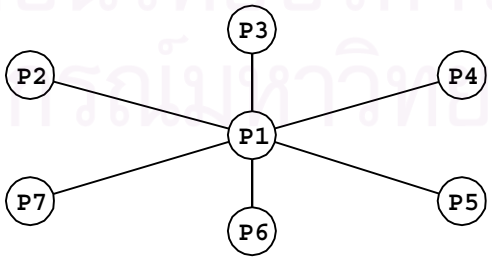


Figure 3.1 A star connected network of seven processors.

3.3 Search Algorithm for Discrete Optimization Problems

3.3.1. Discrete optimization problems

Discrete optimization problems (DOPs) is normally formulated in terms of finding a (minimum cost) solution path in a graph from an initial node to a goal node and solved by graph/tree search. It can be formally stated as: ***Given a finite discrete set S and a function f(x) defined on the elements of S, find an optimal element*** x_{opt} ***such that*** $f(x_{opt}) = min\{f(x)|x \in S\}$**.** In most problems, the set ***S*** is quite large. Consequently, it is very laborious to enumerate the elements in ***S*** for determining x_{opt}**.** The parallel processing is perhaps the way to obtain acceptable performance, and increase possibility to solve the problems.

3.3.2. Sequential Depth-First Search (DFS)

The search begins by expanding the initial node by generating its successors. Then, one of the most recently generated nodes is expanded in the similar way as its parent. If this node does not have any successors then backtracking is done, and a remaining node is selected for expansion. Three search methods that use the depth-first search strategy are the following.

1. **Simple Backtracking** is a method that terminates on finding the first solution. This solution is obviously not guaranteed to be the minimum cost solution.

2. **Depth-First Branch-and-Bound (DFBB)** is an algorithm which searches continue even after finding the solution. Whenever a new solution is found, the current best solution is updated.

3. **Iterative Deepening A*** keeps on expanding nodes in a depth-first fashion until the total cost of the selected node reaches a given threshold (which is increased in each iteration). The algorithm continues until a goal node is selected for expansion. It might appear that IDA* performs a lot of redundancies. But in practice, the redundancy is minimal and the algorithm finds an optimal solution.

3.3.3. Sequential Best-First Search.

Best-first search technique uses heuristics to direct a search through the spaces that is more likely to yield solutions. A* algorithm is a commonly used best-first search technique. A* makes use of a heuristic evaluation function, ***f***, defined over the nodes of the search space. For each node ***x***, ***f(x)*** gives an estimate of the cost of the optimal solution path passing through node ***x***.

A* maintains a list of nodes called **OPEN** which holds the nodes which have been generated but not expanded. This list is sorted on the basis of the ***f*** values of the nodes. The nodes with the lowest ***f*** values are expanded first. The main drawback of this scheme is that it runs out of memory very fast since its memory requirement is linear in the size of the search space explored.

3.3.4. Parallel Depth-First-Search Algorithms

A general procedure for parallel DFS is as follows. Each processor searches a disjoint part of the search space in a depth-first way. When any one finishes searching its part, it tries to get an unsearched part from the other processors. When a goal is found, all of them quit. If the solution does not exist, then all the processors would run out of work, and then terminate. [23]

Since searching is in a depth-first manner, the state space can be represented by a stack. The depth of stack is the depth of the node being explored

currently. Each processor maintains its own local stack. When the local stack is empty, it takes some of the untried alternatives of the stack of another processor. This process continues until all processors go idle or a solution is found.

3.3.5. Parallel Best-First Search

Parallel DFBB is similar to those of parallel DFS. Only little modification is keeping all the processors informed of the current best solution path. Whenever a processor finds a solution path better than the current best known, it broadcasts the solution to all the other processors to update their current best solution path. Parallel formulations of DFBB have been shown to yield linearly increasing speedups for many problems and architectures [24, 25].

3.3.6. Load-Balancing Schemes

In parallel DFS and DFBB, the selection of a target processor for a work request can be done in a number ways. This section reviews three dynamic load-balancing schemes: asynchronous round robin, global round robin, and random polling.

1. **Asynchronous round robin**: Each processor maintains its own pointer that determines the target processor of a work request. Every time a work request is made, the pointer is read and incremented (modulo the number of processors). Nevertheless, it is possible for two or more processors to request work from the same target processor nearly at the same time.

2. Global round robin: A global pointer is maintained at a designated processor. This pointer determines the target of a work request. Every time a work request is made, the pointer is read and incremented (modulo the number of processors). Though the global round robin scheme minimizes the total number of work requests for a wide class of problems, accessing the global pointer might forms a bottleneck. It is possible to degrade the performance.

3. Random polling: A processor is selected at random and the work request is targeted to this processor. It makes random polling does not suffer from such a drawback. However, in special case, on machines that have hardware support for concurrent access to a global pointer, the performance of the global round robin scheme would be better than random polling.

3.4 Message-Passing Interface (MPI)

3.4.1. Narrative

Message passing is a programming paradigm used widely on parallel computing, especially on networks of personal computers. Although there are many variations, the basic concept of processes communicating through messages is successful acceptable. Over the past decades, a worthy development has been made in casting significant applications into this paradigm. Vendors have implemented their own variance. However, the message-passing system can show it is efficient and portable. It is a good occasion to introduce the standard for both the syntax and semantics of routine library that will be practicable to a wide range of users and efficiently implementable on any plate forms. This attempt has been accomplished by

the *Message Passing Interface Forum*, a group of more than 80 people from 40 organizations, representing vendors of parallel systems, industrial users, industrial and national research laboratories, and universities.

3.4.2. Basic concept

There are two simple reasons in practicing message passing: (1) To exchange data between the parallel processes, and (2) to synchronize the processes. In view of human's communication, usually message passing, it is very easy to accept this idea.

One of the remarkable concepts of MPI is the degree of portability across different machines. This means that the same message-passing source code can be executed on any machines as long as the MPI library is available. It can run on a network of computers as a set of processes running on a single computer.

Another benefit offered by MPI is the ability to run transparently on cluster of computers with distinct architectures. It is possible for an MPI implementation to span on virtual computing model that hides many architectural differences. The user needs not to worry whether the code is running on the group of machines of the identical type or not. The MPI implementation will automatically do any necessary data conversion and utilize the correct communications protocol. Anyway, MPI does allow implementations that are targeted to a single system.

An MPI programming consists of autonomous processes, executing their own code (need not be identical for each process). Normally, each process executes in its own memory space, although shared-memory is possible. The processes can be sequential, or multi-threaded, with threads possibly executing concurrently. The processes communicate via calls to MPI communication standard subroutines. The definition of a message passing standard provides vendors with a clearly defined base set of subroutines that they can implement efficiently hardware supports.

3.4.3. Processes communication

The fundamental communication mechanism among processes of MPI is the data transmission between a pair of processes, or *point to point communication.* Most of MPI coding is built around the point to point approach.

Another choice of communication is *collective communications*. The data are transmitted among all processes in a group specified by an intracommunicator object defining the group of participating processes and providing a communication domain for the operation.

3.4.4. Processes Topologies

A topology is an optional attribute that one can give to an communicator. It is possible to assist the runtime system in mapping the processes onto hardware. Normally, each process in the group is assigned a rank between 0 and n-1. However, in many parallel applications, a linear ranking of processes does not adequately reflect the logical communication pattern of the processes. It might be more practicable arranging processes in topological patterns such as two-dimensional grids. Commonly, the logical process arrangement is described as the *virtual topology*.

The virtual topology is at liberty to be accomplished in the same way as physical processors, if this helps to improve the communication performance on a system.

The communication pattern of processes can be represented by a graph. The nodes stand for the processes, and the edges connect processes that communicate with each other. Since communication is most often symmetric, the graphs are assumed to be undirected graphs.

3.5 MPICH

MPICH is a portable implementation of the full MPI specification for a wide variety of parallel computing environments, including workstation clusters and massively parallel processors (MPPs). MPICH contains, along with the MPI library itself, a programming environment for working with MPI programs. The programming environment includes a portable startup mechanism, several profiling libraries for studying the performance of MPI programs, and an X interface to all of the tools. The document describes how to compile, test, and install MPICH and its related tools, the portable implementation of the MPI Message-Passing Standard Details using the MPICH implementation are presented in a User's Guide for MPICH [26].

3.5.1. Downloading MPICH

The easiest way to get MPICH is downloading via the web page *www.mcs.anl.gov/mpi/mpich/download.html*. For alternatively way, file named *mpich.tar.gz* is available for anonymous ftp on *ftp.mcs.anl.gov* in directory *pub/mpi*. To unpack the *mpich.tar.gz* file, move it into a build directory, assume the directory name is */tmp*. Make sure that enough available space is enough (larger than 100MB is advisable). Then, apply following command.

```
% cd /tmp
% tar zxovf mpich.tar.gz
```

However, if *tar* program on the system does not accept the *z* option, use

```
% cd /tmp
% gunzip -c mpich.tar.gz | tar zxovf -
```

Finally, It is necessary to check the web page *www.mcs.anl.gov/mpi/mpich/buglist-tbl.html* for any patches that might be needed. The patch page has instructions on applying the patches.

Chapter 4

Method of Experiments

4.1 Mathematical Model

A stochastic L-system examined in this experiment is based on bracketed L-system, it given as:

Axiom:

ω : F

Rules:

p_1 : $F \xrightarrow{0.33} F[+F]F[-F]F$

p_2 : $F \xrightarrow{0.33} F[+F]F$

p_3 : $F \xrightarrow{0.34} F[-F]F$

Where the quantities revealed over the arrows represent the production probabilities for each rule. The symbols utilized to represent plant structure are described in Table 4.1.

Table 4.1 Two-dimensional interpretations of L-system symbols in visualization.

Symbols	Interpretation
F	Moves forward with distance ***d***. The status is changed from ***(x, y, α)*** to ***(x′, y′, α)*** by equation $x' = x + d \cdot \cos\alpha$ and $y' = y + d \cdot \sin\alpha$. Draws a line from ***(x, y)*** to ***(x′, y′)***.
+	Rotate counter clockwise by angle δ. The status is changed from ***(x, y, α)*** to ***(x, y, α+δ)***
-	Rotate clockwise by angle δ. The status is changed from ***(x, y, α)*** to ***(x, y, α-δ)***
[	Push the current status onto a stack.
]	Pull status from the stack and then assign to current status.

The experimental program was built using C-programming language, and parallel components in program were implemented utilizing Message-Passing Interface (MPI). The program was performed on two sets of computer clusters. The first cluster composed of 3 computers include of two *Silicon Graphics O2* computer and one *Silicon Graphics OCTANE2*, working with IRIX operation system Release 6.5. The second cluster composed of four *HP Workstations x2000*, each of them was installed with Mandrake Linux Release 9.0.

The program was performed for various derivation lengths (1-9), and different amount of process numbers (2-11). A program written in familiar manner – sequential method – is performed for comparison. Time spent in each executing were recorded and interpreted.

The load balancing technique in this program was developed by means of Round Robin load balancing method. Instead of idle process look around for some undone job obtainable from active processes, an active process is looking for idle processes and sends divided job to them. The hypothesis is this dynamic technique might yield an improved performance due to number of processing units.

4.2 Main Program Algorithm

The main program can be described in two sections. First section is the code performing as master process, and the second one is the code performing as slave or child process. In fact, all programs on each machine are identical. Just during the starting execution, the program evaluates its own rank. Only a process of zero rank works as a master process and the others work as slaves or child processes.

An algorithm for master process can be briefly described. It initiates system by sending the axiom and all rules to the child process. Then, it calls one of child processes to perform the job, and waits the response from the child processes, if some asks for an idle process. Finally, when all child processes finish their jobs, the master receives all the results, sorts the jobs and then finishes the process.

In child process, it waits for a string sent from the other processes. The string is generated accordingly to the specified stochastic L-system grammar. If the generated string becomes too long, the child process is authorized to request from the master process for another idle process. If idle process presents, the job is divided and second half is sent to the idle process. If none exists, the process resumes its job.

The whole detail of algorithm for both sections is as follows:

4.2.1. Master Process

```
GET rules, axiom, and desired derivation length
SEND rules to all child processes
SEND axiom and derivation length to the first child processes
WHILE (job not finished) DO
    WAIT & RECEIVE status from child process
    IF child process asks for idle process THEN
        SEEK for idle process from idle table
        IF found an idle process THEN
            SEND back rank of idle process
        ELSE
            SEND message UNKNOW to tell no any idle process exists
        ENDIF
    ENDIF
        IF child process requests to send back result THEN
        RECEIVE result string
        APPEND result_string into result_list
    ENDIF
ENDWHILE
SEND process_control=NO_MORE_JOB to all child processes
SORT result_list on their label
WRITE down final result string
```

4.2.2. Child Process

```
WHILE control declares don't terminate DO
    RECEIVE control
    IF control state next message is "rules" THEN RECEIVE rules
    IF control state next message is "job" THEN
        RECEIVE job
        WHILE job is not finished yet DO
            PERFORM job
            IF job is too much THEN
                ASK for an idle process
                IF there are some idle processes THEN
                    DIVIDE job into two parts
                    SEND the second part to the idle process
                ENDIF
            ENDIF
        ENDWHILE
        SEND result to the master process
    ENDIF
ENDWHILE
```

4.3 Visualize Program Algorithm

The result coming out from main program is string of characters. To present it graphically, we need a program to interpret the string. The following algorithm is used for transforming result string from main program to set of xy-coordinate, which finally to be plotted by program GNUPLOT.

4.3.1. Visualize Program

```
GET L-system string
Prepare status-stack
SET turtle current-status (position x, y and angle α) to zero
SET turtle next-status (position x, y and angle α) to zero
FOR each character from the string
    IF character is '['  THEN push current-status into status-stack
    IF character is ']'  THEN pull status-stack into current-status
    IF character is '-'  THEN COMPUTE next-status α' = α - δ
    IF character is '+' THEN COMPUTE next-status α' = α + δ
    IF character is 'F' THEN
        COMPUTE next-status x' = x - d·sin(α) and y' = y + d·sin(α)
        DRAW line from (x, y) to (x', y')
        SET current-status = next-status
    ENDIF
NEXT character
```

Chapter 5

Experimental Result

5.1 Sequential execution

The experimental programs, for generating an L-system tree string, are firstly performed in sequential manner on a single processor. It was executed on HP workstations X2000 for approximately 100 times for each derivation length of 2, 3, 4, 5, 6, 7, 8 and 9 respectively. The averaged time spent for each derivation length are calculated and plotted against the length as shown in Figure 5.1. Since the result of L-system algorithm used is in order of $O(m^{n+1})$ [see Appendix C], it will be helpful to plot the relation in semi-log graph. Another excellent demonstration for the $O(m^{n+1})$ relationship is correlation between derivation length with length of result string generated from L-system program (shown in Figure 5.2).

It should be noticed that in Figure 5.2, the relationship is very precisely resembled to an exponential curve (appearance as a straight line in semi-log graph). However, the curve in Figure 5.1 turns to a straight line after the derivation length of five. The numerical values of time spent and the averaged result string length in the sequential method are shown in Table 5.1.

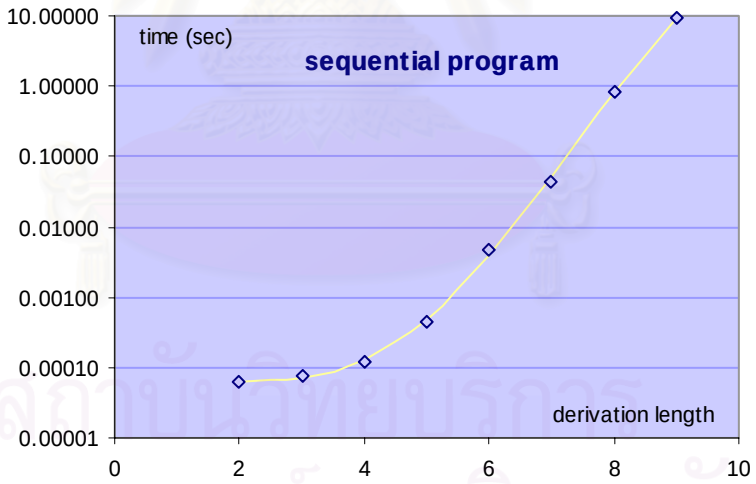


Figure 5.1 Averaged time spent for each derivation length in

Table 5.1 Numerical values of averaged time spent and averaged length of result string in the sequential method

Derivation Length	Averaged Time Spent(second)	Average String Length(character)
2	0.0000634	30
3	0.0000767	124
4	0.0001204	446
5	0.0004374	1,545
6	0.0047668	6,352
7	0.0446541	20,464
8	0.8367362	80,971
9	9.4074299	292,193

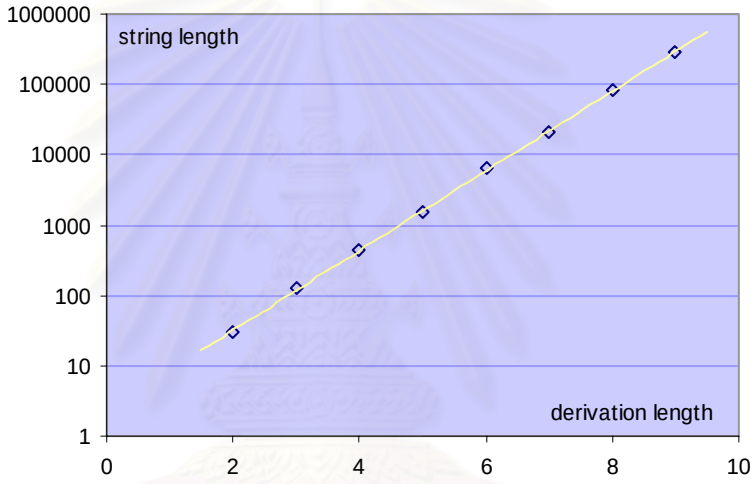


Figure 5.2 Averaged length of result string for derivation lengths generated from L-system program.

5.2 Parallel execution on SGI cluster

On SGI cluster – include of SGI O2 and SGI OCTANE2 – the parallel version of program is performed for 1, 2, 3 and 4 processes. They yield unexpected result. The time spent was supposed to be decreased during number of process increase. But on contrary, for every length of derivation, the time spent dropped at process number of 2, and rapidly increased for more number of processes (figure 5.3 and table 5.2).

The appropriate explanation for the time increasing might be the network connection among SGI system is linked with the Internet, which is possible that the information sent among SGI machines have to reroute to somewhere outside before arrives destination. Moreover, due to the program need very large space for result string, the very limited swap space of SGI machines might extend the executing time, and it is too difficult to rearrange this swap space.

Due to it was obviously seen the purposeless to perform more experimental, the experiment for more number of processes had not been longer perform.

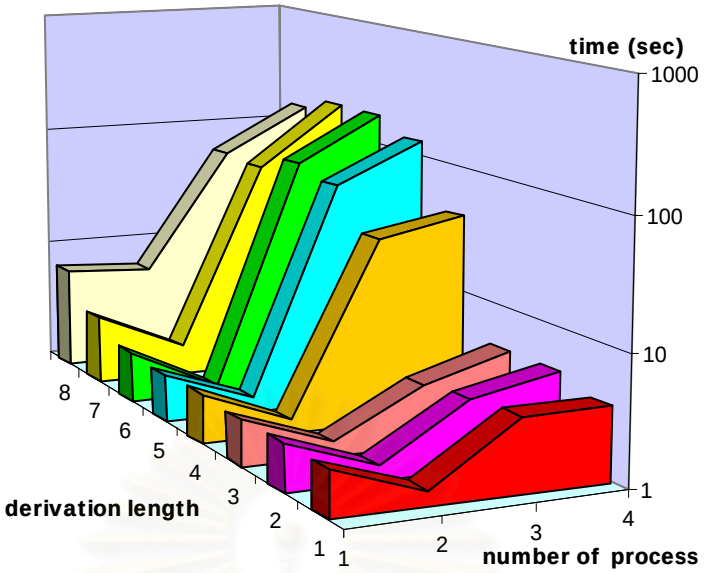


Figure 5.3 Averaged time spent for derivation lengths from parallel program executing on SGI cluster.

Table 5.2 Averaged time spent for derivation lengths from parallel program executing on SGI cluster.

Derivation	Number of Process							
Length	2	3	4	5	6	7	8	9
2	2.2676	2.3308	2.3086	2.4466	2.3508	2.5565	3.531	6.6724
3	1.3147	1.2943	1.2907	1.2702	1.30633	1.1425	1.7483	5.9937
4	3.7294	3.3248	2.919	30.422	61.5402	73.9475	54.0233	57.1711
5	3.6858	4.6652	4.584	40.3185	126.2795	172.3375	170.7212	135.7317

5.3 Parallel execution on HP workstation cluster

Executing of the program on four HP workstations X2000 cluster were performed approximately 100 times for each of derivation length of two to nine and process numbers of 2 to 11. The system yielded more appropriate results. However, the experiments show that the size of string sent among processes yield significant effect on performance. In the parallel version program, the constant named MAXSTRING defined maximum length of string transferred among processes. If it was defined to be 180,000, the program is capable to execute faster, but was unable to calculate in case derivation length larger than eight. The result is demonstrated graphically in Figure 5.4 and in Table 5.3. It can be seen that the result revealed what has been supposed to be. Increasing of number of processes made averaged time spent in calculation decreased for length of derivation larger than seven.

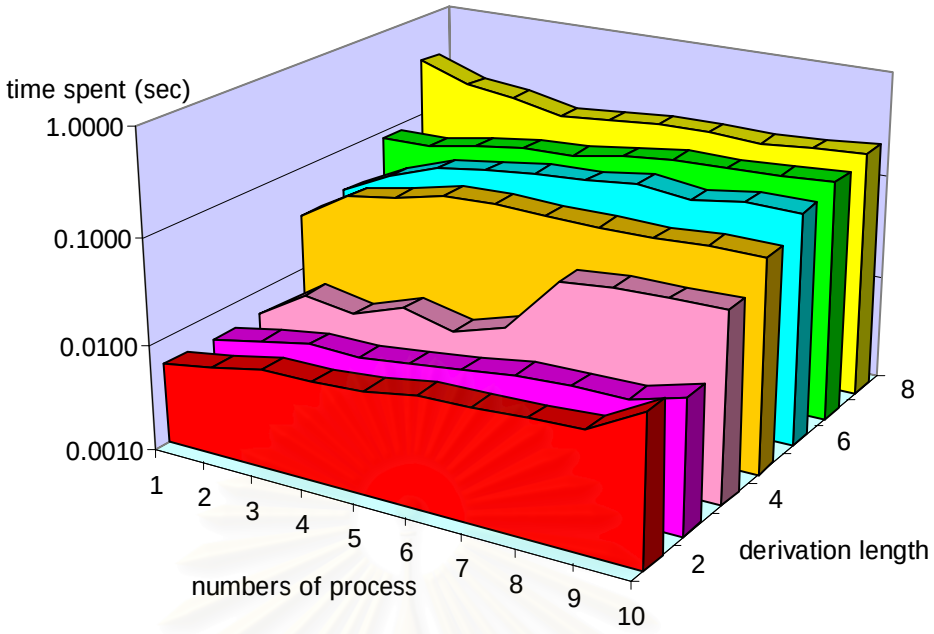


Figure 5.4 Averaged time spent for derivation lengths from parallel program executing on HP workstation cluster, with MAXSTRING = 180,000.

Table 5.3 Averaged time-used for derivation lengths from parallel program, executing on HP workstation cluster, with MAXSTRING = 180,000.

Derivation Length	Number of Process						
	2	3	4	5	6	7	8
2	0.0057	0.0056	0.0058	0.0332	0.0377	0.0831	0.3497
3	0.0068	0.0068	0.0113	0.0642	0.0649	0.0831	0.2352
4	0.0084	0.0083	0.0095	0.0777	0.0937	0.1012	0.2052
5	0.0086	0.0081	0.0137	0.0982	0.1112	0.1166	0.1625
6	0.0092	0.0092	0.0105	0.1042	0.1271	0.1233	0.1795
7	0.0112	0.0106	0.0146	0.0993	0.1329	0.1488	0.1960
8	0.0114	0.0124	0.0505	0.0970	0.1543	0.1670	0.2047
9	0.0120	0.0129	0.0561	0.0960	0.1362	0.1673	0.1912
10	0.0130	0.0128	0.0573	0.1033	0.1572	0.1765	0.2095
11	0.0238	0.0177	0.0593	0.1022	0.1565	0.1930	0.2249

In case the maximum length of string sent among processes was defined to be 290,000, the program is able to calculate up to derivation length of nine. However, speedup factor was a little bit slowed down. The result is shown graphically in Figure 5.5 and in Table 5.4.

In addition, averaged executing time-used on each process for derivation lengths of 8, with maximum length of string sent among processes of 120,000, is shown in Table 5.5.

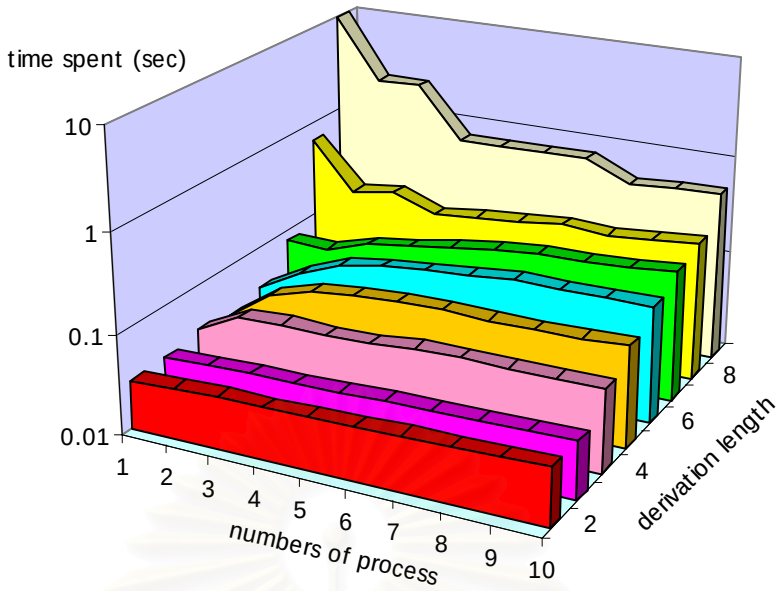


Figure 5.5 Averaged time spent for derivation lengths from parallel program executing on HP workstation cluster, with MAXSTRING = 290,000.

Table 5.4 Averaged time-used for derivation lengths from parallel program, executing on HP workstation cluster, with MAXSTRING = 290,000.

Derivation Length	Number of Process							
	2	3	4	5	6	7	8	9
2	0.0306	0.0307	0.0364	0.0313	0.0390	0.0798	0.6345	9.6150
3	0.0314	0.0316	0.0590	0.0619	0.0663	0.0773	0.2138	2.2977
4	0.0325	0.0326	0.0620	0.0837	0.0977	0.1046	0.2443	2.3639
5	0.0325	0.0327	0.0596	0.0936	0.1157	0.1202	0.1689	0.7015
6	0.0339	0.0336	0.0613	0.0968	0.1277	0.1399	0.1879	0.7220
7	0.0346	0.0347	0.0667	0.0990	0.1389	0.1614	0.2043	0.7281
8	0.0356	0.0357	0.0684	0.0959	0.1519	0.1789	0.2304	0.7386
9	0.0356	0.0357	0.0670	0.0974	0.1509	0.1800	0.2135	0.4805
10	0.0367	0.0366	0.0665	0.0978	0.1538	0.1923	0.2318	0.5075
11	0.0377	0.0376	0.0675	0.1033	0.1514	0.2075	0.2522	0.5330

Table 5.5 Averaged time-used on each process for derivation lengths of 8 from parallel program, executing on HP workstation cluster, with MAXSTRING = 120,000.

Numbers of Process	Total Time (second)	Averaged Executing Time on Each Process (second)							
		P1	P2	P3	P4	P5	P6	P7	P8
2	0.6996	0.6958							
3	0.2382	0.2133	0.2053						
4	0.2701	0.1263	0.2430	0.1982					
5	0.1891	0.1535	0.1424	0.1280	0.1302				
6	0.2190	0.1524	0.1587	0.1433	0.1360	0.1244			
7	0.2331	0.1482	0.1456	0.1486	0.1430	0.1309	0.1380		
8	0.2460	0.1482	0.1435	0.1421	0.1470	0.1351	0.1398	0.1480	
9	0.2178	0.1334	0.1217	0.1100	0.1007	0.1009	0.1114	0.1237	0.1143

In comparison of parallel computation with a sequential computation, it is helpful to be established as *speedup factor*, ratio of execution time using one processor with execution time using multiprocessor. The speedup factor for execution on HP workstation cluster are calculated and sketched as shown in Figure 5.6 and Figure 5.7.

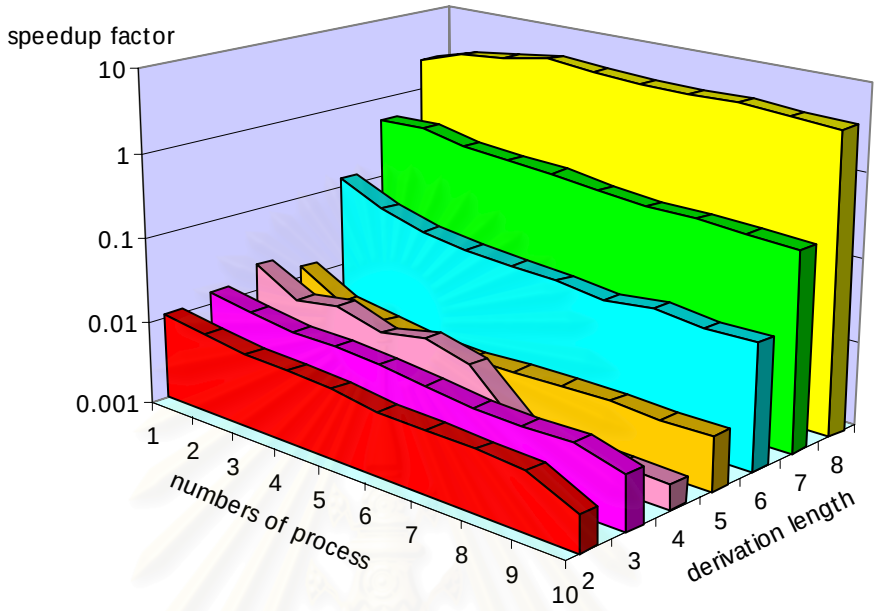


Figure 5.6 *Speedup Factor* from parallel program executing on HP workstation cluster, with MAXSTRING = 180,000.

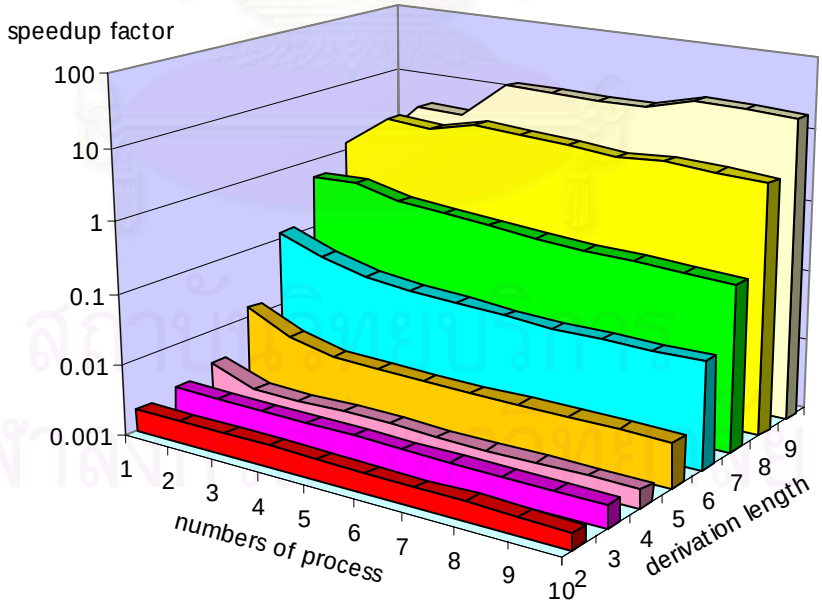


Figure 5.7 *Speedup Factor* from parallel program executing on HP workstation cluster, with MAXSTRING = 290,000

5.4 Additional Outcomes

Due to the previous graphics on the relation between execution time, the number of processes and the derivation length are plotted in a semi-log graph. But for some instance, it might be more certain if the relation is shown in a normal graph. Some of the interesting relations are demonstrated in following Figures:

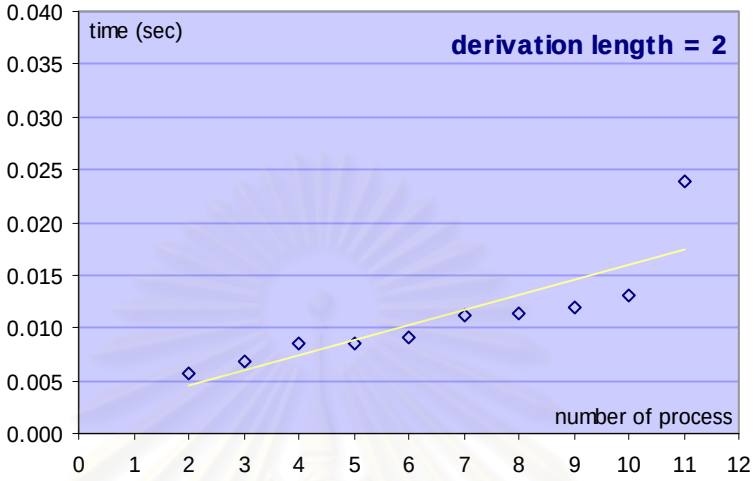


Figure 5.8 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 2 and MAXSTRING = 180,000

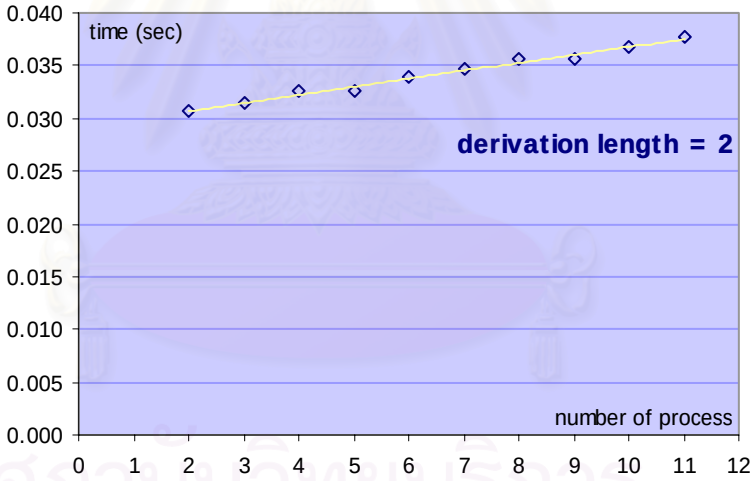


Figure 5.9 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 2 and MAXSTRING = 290,000

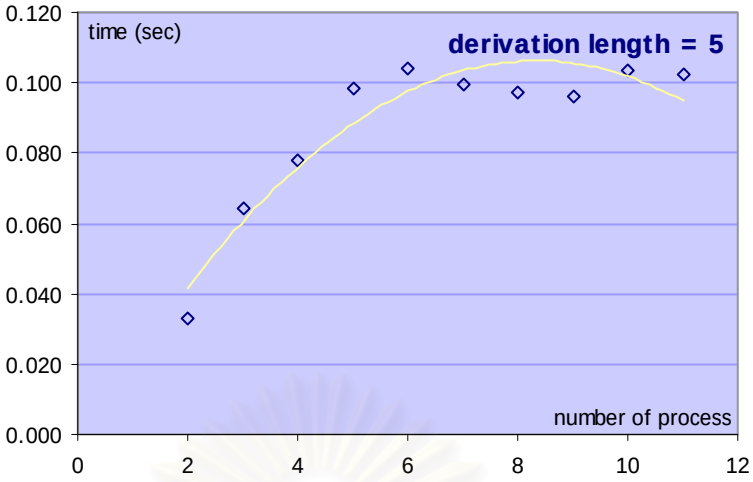


Figure 5.10 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 5 and MAXSTRING = 180,000

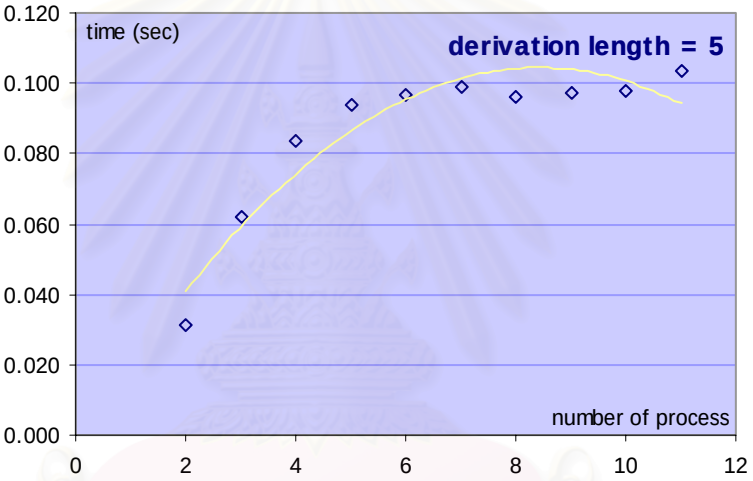


Figure 5.11 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 5 and MAXSTRING = 290,000

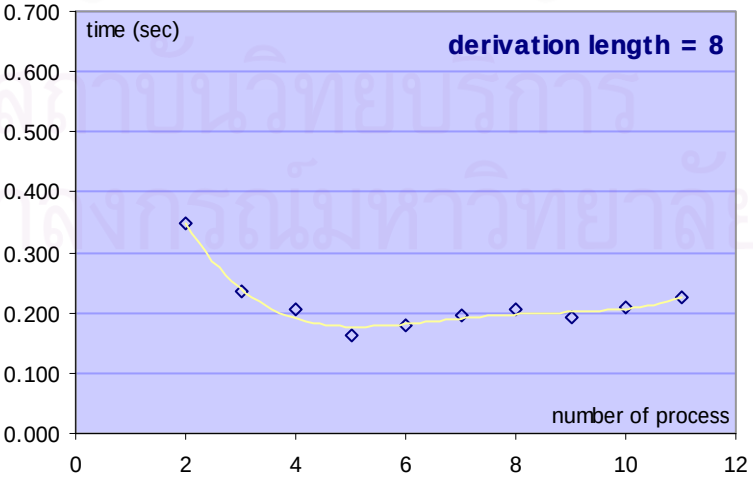


Figure 5.12 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 8 and MAXSTRING = 180,000

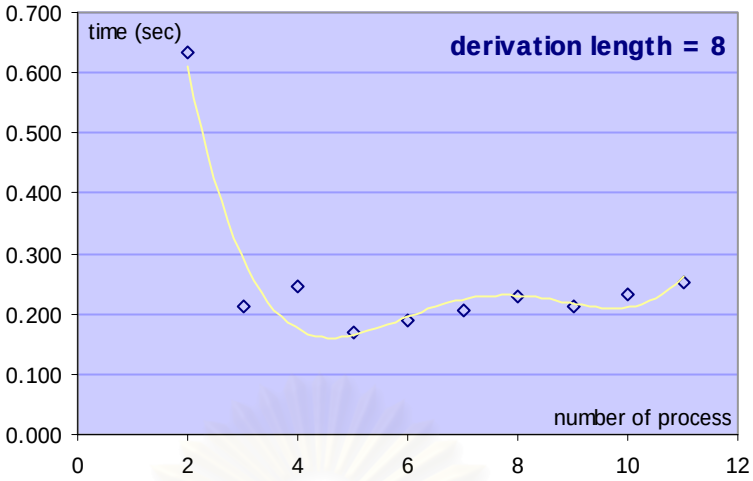


Figure 5.13 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 8 and MAXSTRING = 290,000

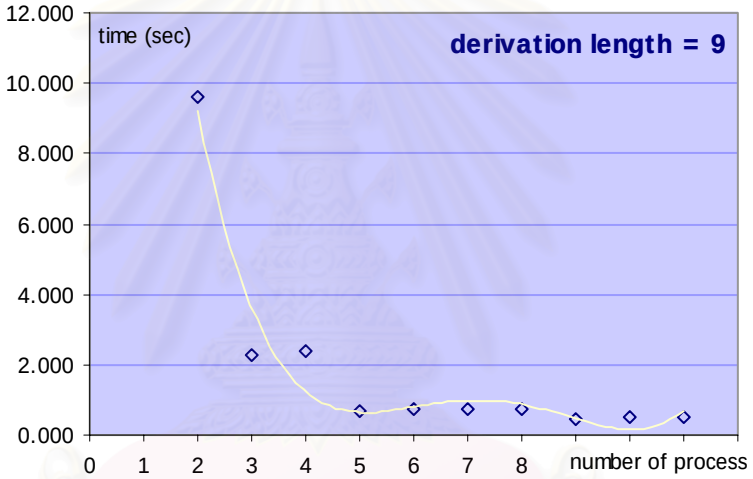


Figure 5.14 The relation of execution time and number of process performed on HP workstation cluster, with derivation length = 9 and MAXSTRING = 290,000

Chapter 6

Conclusion

The overhead of MPI in this experiment is mainly affected by information transferring among processes. Comparisons between the sequential and parallel programs (Table 5.1, 5.3 and 5.4) show very large difference of executing times for derivation length of two. It revealed MPI overhead is very large. In addition, Figure 5.8 and 5.9, the derivation length of two, shows the linearly dependent of executing time with amount of process involved in calculation. They both increase in the same way. It is definitely seen that executing time is longer in case of larger MAXSTRING, the constants defining maximum length of string transferred among processes. Comparison of Figure 5.8 and 5.9 could be applied major effect of MAXSTRING as well.

On the other hand, in region which derivation length is larger than seven, result string size becomes very large, and the main job is much extensive. It is clearly seen that executing time was significantly effected by number of process. The Charts in Figures 5.12, 5.13 and 5.14 revealed that the executing time is largely decreased with respect to the added number of processes.

It can be seen from the result that the parallel computing with MPI assistance yields very good efficiency if workload of MPI overhead is insignificant when compared to the main tasks. The speedup factor is notable worthy if the derivation length is greater than seven. The maximum value of speedup factor for this research is approximately 19.6 at derivation length of nine with nine processes operated simultaneously.

The expectation that computation speed could be improved along with high number of processors has been accomplished. However, because of extensive duration of bi-directional communication among processes, it is worth to applying parallel computing technique if the overhead of parallel method is minor for whole system.

References

1. Prusinkiewicz, P., Lindenmayer, A. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York: **1990**.
2. Chuai-aree, S., Siripant, S., and Lursinsap, C. Animating Plant Growth in L-System by Parametric Function Symbols. *Proceeding of International Conference on Intelligent Technology 2000 (December 2000):* 135-143.
3. Smith, A.R. Plants, fractals, and formal languages, *ACM SIGGRAPH* vol. 18, no.3 (1984): 1-10.
4. Hammel, M. S., and Prusinkiewicz, P. Visualization of developmental processesby extrusion in space-time. *Proceedings of Graphics Interface '96* (May 1996): 246-258.
5. Samal, A., Peterson, B., and Holliday, D.J. *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, 1 (Nov 1994): 183-187.
6. Chen, C.M., and Jing, H.W. *A Simulation Study of Plant Hybridization Using L+ System* [online]. Available from: http://bashful.ice.ntnu.edu.tw/~jason/summary_index.htm [2003, Jan]
7. Stefanovski, S., Loskovska, S., and Mihajlov, D. Representation and realistic rendering of objects defined by L-systems. *Electrotechnical Conference, 1998. MELECON 98., 9th* Mediterranean , 1 (May 1998): 86-90.
8. Schaefer, C.G., Jr., Morphogenesis of path plan sequences through genetic synthesis of L-system productions. *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress*, 1 (1999): 365.
9. Lintermann, B., and Deussen, O. Interactive modeling of plants. *Computer Graphics and Applications, IEEE*, 19 (Jan 1999): 56-65.
10. Fracchia, F.D., and Ashton, N.W. A visualization tool for studying the development of the moss Physcomitrella patens. *Visualization, 1995. Visualization '95. Proceedings., IEEE Conference*, (Oct 1995): 364 -367, 475.
11. Yi-Cheng Lin, and Sarabandi, K. A coherent scattering model for forest canopies based on Monte Carlo simulation of fractal generated trees. *Geoscience and Remote Sensing Symposium, 1996. IGARSS '96. 'Remote Sensing for a Sustainable Future.', International*, 2 (May 1996): 1334 –1336.
12. Mock, K.J. Wildwood: the evolution of L-system plants for virtual environments. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence, The 1998 IEEE International Conference*, (May 1998): 476-480.
13. Chuai-aree, S. *An Algorithm for Simulation and Visualization of Plant Shoots Growth*. Master's Thesis, Department of Mathematics, Faculty of Science, Chulalongkorn University, **2000**.

14. Poovarawan, Y., and Uthayopas, P. *High Performance Computing: Needs and Application for Thailand* [online]. **1997**. Available from: http://prg.cpe.ku.ac.th/publications/yuen.pdf [2003, Mar]

15. Uthayopas, P., Angskun, T., and Maneesilp, J. *Building a Parallel Computer from Cheap PCs: SMILE Cluster Experiences* [online]. **1998**. Available from: http://prg.cpe.ku.ac.th/publications/anscse2.pdf [2003, Mar]

16. Lin, J.S., Hsu, F.R., and Lee, R.C.T. A parallel algorithm for the single step searching problem. *Parallel Architectures, Algorithms and Networks*, **1994**. *(ISPAN) International Symposium*, (Dec 1994): 278-285.

17. Von Koch, H. *Acta Mathematica*. 30:145-174, **1905**.

18. Salomaa. *Formal Languages*. Academic Press, New York, **1973**.

19. Chomsky., N. *Three models for the descriptions of language*. IRE Trans. On Information Theory, 2(3):113-124, **1956**.

20. Frijters, D. and Lindenmayer, A. L System. *Lecture notes in Computer Science 15*. Springer-Verlag, Berlin: 24-52, **1974**.

21. Hogeweg, P. and Hesper, B. A model study on biomorphological description. *Pattern Regognition*, 6:165-179, **1974**.

22. Smith, A. R. Plants, fractals, and formal languages. *Computer Graphics*, 18, 3 (July 1984), ACM SIG-GRAPH, New York: 1-10, **1984**.

23. Hwang., K. *Advanced Computer Architecture with Parallel Programming*. McGraw Hill Inc., **1992**.

24. Arvindam, S., Kumar, V. and Nageshwara Rao, V. Floorplan optimization on multiprocessors. *Proceedings of the 1989 International Conference on Computer Design (ICCD-89)*, **1989**.

25. Arvindam, S., Kumar, V., and Nageshwara Rao., V. Efficient parallel algorithms for search problems: Applications in VLSI cad. *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October **1990**.

26. Gropp, W. and Lusk, E. *Installation and User's Guide to MPICH*, a Portable Implementation of MPI, Argonne National Laboratory, Argonne, IL, **2001**.

27. Lindenmayer, A. *Journal of Theoretical Biology*, 18:280-315, **1968**.

Appendices

Appendix A

Program Listing

```
 1: /*   modify on 14 MAR 2003 treealfa2.c   */
 2:
 3: #include<stdio.h>
 4: #include<stdlib.h>
 5: #include<string.h>
 6: #include<time.h>
 7: #include<sys/types.h>
 8: #include<math.h>
 9: #include"mpi.h"
10:
11: #define ran() ((double) rand()/RAND_MAX)
12:
13: #define MAXITER             500
14: #define MAXSTRING          200000
15: #define PROCESSMAX           10
16: #define MAXLOAD             100  /* l-string will be seperate if > MAXLOAD */
17: #define MAXLEVEL             10  /* maximum level for job separation */
18: #define MAX_RESULT_LIST      100
19: #define MAX_RULE_CHAR        255
20: #define MAXRULE               20
21:
22: #define TRUE                   1
23: #define FALSE                  0
24: #define EQUAL                  0
25: #define LESSTHAN              -1
26: #define GREATERTHAN            1
27: #define IS_GREATER_THAN       >0
28: #define IS_EQUAL              ==0
29:
30: /******** Constanst for processes status ********/
31:
32: #define START_PROGRAM         -1
33: #define UNKNOW                -2  /* UNKNOW has to be negative number */
34: #define PROCESS_IDLE           0
35: #define PROCESS_BUSY           1
36: #define ASK_FOR_IDLE_PROCESS   4
37: #define RESULT_READY           8
38:
39: #define STATUS_TAG           501
40: #define IDLE_PROCESS_TAG     502
41: #define L_STRING_TAG         503
42: #define RULE_TAG             504
43:
44: /*********************************************/
45:
46: #define ROOT_PROCESS           0
47:
48: /******** Constanst for loop control ***********/
49:
50: #define PERFORM_LOOP           1
51: #define TERMINATE_LOOP         0
52:
53: /******** Constanst for process control *********/
54:
55: #define NEW_JOB_COMING         1
56: #define RULE_COMING            2
57: #define NO_MORE_JOB            0
58:
59: #define PROCESS_CONTROL_TAG   504
60:
61: /******** Macro ********************************/
62:
63: #define PP printf ("[P%1d] ", rank)
64: #define PRINT_LABEL_OF(X) for (kr=0;kr<X.index;kr++) printf ("%d ",X.label[kr])
65: int kr ;
66:
67: /*****GLOBAL VARIABLE*************/
68:  int    rank, process_n;
69:  int    idle_table[PROCESSMAX];
70:
71: /*********************************************/
72:
73: /*********************************/
74: /*    Declare MPI derived type    */
75: /*********************************/
```

```
 76:
 77:  struct  L_String
 78:     {
 79:      int     iter ;              /* iteration needed           */
 80:      char    label[MAXLEVEL] ;   /* label for this string      */
 81:      char      str[MAXSTRING] ;  /* string to be operated      */
 82:     } ;
 83:
 84:  MPI_Datatype  MPI_L_String ;
 85:
 86:  MPI_Datatype  typ1[3]  =  {MPI_INT, MPI_CHAR,    MPI_CHAR} ;
 87:  int           len1[3]  =  {
 88:                             1,
 89:                             MAXLEVEL,
 90:                             MAXSTRING
 91:                            };
 92:  MPI_Aint      dis1[3]  =  {
 93:                             0,
 94:                             sizeof(int),
 95:                             sizeof(int)+MAXLEVEL*sizeof(char)
 96:                            } ;
 97:
 98:  struct Rule
 99:     {
100:      int     num_rule ;
101:      float   prob [MAXRULE] ;
102:      float   probx [MAXRULE] ;
103:      char    pred [MAXRULE] [MAX_RULE_CHAR] ;
104:      char    succ [MAXRULE] [MAX_RULE_CHAR] ;
105:     } ;
106:
107:  MPI_Datatype  MPI_Rule ;
108:
109:  MPI_Datatype  typ2[5]  =  {MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_CHAR, MPI_CHAR} ;
110:  int           len2[5]  =  {
111:                             1,
112:                             MAXRULE,
113:                             MAXRULE  ,
114:                             MAXRULE * MAX_RULE_CHAR ,
115:                             MAXRULE * MAX_RULE_CHAR
116:                            } ;
117:  MPI_Aint      dis2[5]  =  {
118:                             0,
119:                             sizeof(int),
120:                             sizeof(int) + MAXRULE*sizeof(float),
121:                             sizeof(int) + MAXRULE*sizeof(float)
122:                                         + MAXRULE*sizeof(float),
123:                             sizeof(int) + MAXRULE*sizeof(float)
124:                                         + MAXRULE*sizeof(float)
125:                                         + MAXRULE*MAX_RULE_CHAR*sizeof(char)
126:                            } ;
127:
128: /********* End Declaration ********/
129:
130: int  All_process_idle();
131:
132: void SwapLStr ( struct L_String *x, struct L_String *y) ;
133:
134: main(int argc,   char *argv[])
135: {
136:
137:  time_t t1;
138:
139:  int    send_iter[]={1,4};  /* iteration that mom send to each son */
140:  int    i, j, k, count;
141:
142:  int    idle_processor = UNKNOW ;
143:
144:  char   send_string[PROCESSMAX][MAXSTRING];
145:
146:  char   result_string[MAXSTRING];
147:  char   cut_result_string[MAXSTRING];
148:  char   temp_result[MAXSTRING];
149:  char   temp_string[MAXSTRING];
150:  char   finished_str[MAXSTRING];
151:
152:  int    cut_num = 0;
153:
154:  int    loop1_control ;
155:  int    process_control ;
156:
157:  int    process_status = UNKNOW ;
158:
159:  int    swap_occur ;
160:
161:  double sttime, entime;
162:  double sttime1, entime1;
```

```
163:
164:  char  *Lptr  ;
165:  char  sub_str [MAXSTRING] ;
166:  float ran_val ;
167:  float upper_bound, lower_bound ;
168:
169:  struct L_String sent_message, recv_message  ;
170:  struct L_String result_list [MAX_RESULT_LIST] ;
171:  int             result_index  =  0 ;
172:  struct Rule  rule_all;
173:
174: /*********************************/
175: /*     Initialize MPI Engine.     */
176: /*********************************/
177:
178:  MPI_Status  status;
179:
180:  printf ("[PX] ");  printf ("START PROCESS\n") ;
181:
182:  MPI_Init(&argc, &argv); /* initial MPI variable  */
183:  MPI_Comm_rank(MPI_COMM_WORLD, &rank);      /* set number for each processor  */
184:  MPI_Comm_size(MPI_COMM_WORLD, &process_n); /* set total number of processor  */
185:
186: /********* End Initialize. ********/
187:
188: /*********************************/
189: /*    Commits MPI derived type    */
190: /*********************************/
191:
192:  MPI_Type_struct (3, len1, dis1, typ1, &MPI_L_String) ;
193:  MPI_Type_commit (&MPI_L_String) ;
194:
195:  MPI_Type_struct (5, len2, dis2, typ2, &MPI_Rule);
196:  MPI_Type_commit (&MPI_Rule);
197:
198: /********* End Declaration *******/
199:
200: /*
201:    NOTE:
202:    ====
203:
204:    parent: rank == 0
205:    child:  rank != 0
206:
207:    Communication: parent --> child
208:            1) PARENT: Send process control.  CHILD: Recv process control.
209:            2) PARENT: Send Infomation.       CHILD: Perform appropriate action.
210:
211:    Communication: child --> parent
212:            1) PARENT: Recv process status.   CHILD: Send process status.
213:            2) PARENT: Perform the action.    CHILD: Send information.
214:
215: */
216:
217: if (rank == ROOT_PROCESS) {
218:
219: /*************************************************************/
220: /*                                                           */
221: /*           P A R E N T   P R O C E S S   P A R T           */
222: /*                                                           */
223: /*************************************************************/
224:     sttime1 = MPI_Wtime();
225:     sttime  = MPI_Wtick();
226:     rule_all.num_rule = 3;
227:
228:     rule_all.prob[0]  = 0.33;
229:     strcpy(rule_all.pred[0] ,"F");
230:     strcpy(rule_all.succ[0] ,"F[+F]F[-F]F");
231:
232:     rule_all.prob[1]  = 0.33;
233:     strcpy(rule_all.pred[1] ,"F");
234:     strcpy(rule_all.succ[1] ,"F[+F]F");
235:
236:     rule_all.prob[2]  = 0.34;
237:     strcpy(rule_all.pred[2] ,"F");
238:     strcpy(rule_all.succ[2] ,"F[-F]F");
239:
240:     strcpy(send_string[1], "F");
241:     strcpy(send_string[2], "");
242:
243:     /*=======================================================*/
244:     /*    First Section:                                     */
245:     /*          Initiate job. Separate job, and send all     */
246:     /*          suparated job to child processes.            */
247:     /*=======================================================*/
248:
249:
```

```
250:     for( i=1; i<process_n; i++) idle_table [i] = PROCESS_IDLE ;
251:
252:     for( i=1; i<process_n; i++)
253:     {
254:         rule_all.probx[0] = 0.0 ;
255:         for( k=1; k<=rule_all.num_rule; k++)
256:         {
257:            rule_all.probx[k] = rule_all.probx[k-1] + rule_all.prob[k-1] ;
258:         }
259:
260:         PP; printf ("SEND RULE : ROOT --> CHILD:%d\n",i      );
261:         for (j=0; j<=rule_all.num_rule; j++)
262:         {
263:             PP; printf("     : j=%d  prob=%f  probx=%f  pred=%s  succ=%s \n",
264:                        j, rule_all.prob[j], rule_all.probx[j],
265:                        rule_all.pred[j], rule_all.succ[j]);
266:         };
267:
268:         process_control = RULE_COMING ;
269:         MPI_Send ( &process_control, 1, MPI_INT,
270:                    i, PROCESS_CONTROL_TAG, MPI_COMM_WORLD );
271:         MPI_Send ( &rule_all, 1, MPI_Rule,
272:                    i, RULE_TAG , MPI_COMM_WORLD );
273:     }
274:
275:     /* zone mom send iteration & string to son processor     */
276:     for( i=1; i<2; i++)
277:     {
278:         sent_message.iter     =  send_iter [i-1] ;
279:         if (i == 1 ) strcpy(sent_message.label, "1");
280:         if (i == 2 ) strcpy(sent_message.label, "2");
281:         if (i == 3 ) strcpy(sent_message.label, "3");
282:         if (i == 4 ) strcpy(sent_message.label, "4");
283:         if (i == 5 ) strcpy(sent_message.label, "5");
284:
285:         strcpy (sent_message.str, send_string [i]);
286:
287:         PP; printf ("SEND MESSAGE : ROOT --> CHILD:%d\n",i      );
288:         PP; printf ("     : iter     = %d\n", sent_message.iter );
289:         PP; printf ("     : label    = %s\n", sent_message.label);
290:         PP; printf ("     : str      = %s\n", sent_message.str  );
291:
292:         idle_table[i] = PROCESS_BUSY;
293:
294:         process_control = NEW_JOB_COMING ;
295:         MPI_Send ( &process_control, 1, MPI_INT,
296:                    i, PROCESS_CONTROL_TAG, MPI_COMM_WORLD );
297:         MPI_Send ( &sent_message, 1, MPI_L_String,
298:                    i, L_STRING_TAG , MPI_COMM_WORLD );
299:     }
300:
301:     /*=======================================================*/
302:     /*    Second Section:                                    */
303:     /*          Wait and de periodically read status of      */
304:     /*          child processes. Then perform appropriate    */
305:     /*          action for that status.                      */
306:     /*=======================================================*/
307:
308:     /* Watching for in case process need some information */
309:     /* loop while check that every son processor finished its work? */
310:
311:     while (All_process_idle() == FALSE)
312:     {
313:         for (i = 1; (i < process_n) && (All_process_idle() == FALSE); i++)
314:         {
315:            /*-----------------------------------------------------
316:             |  Wait and receive status/request from child process. |
317:
318:             -----------------------------------------------------*/
319:             PP; printf ("CHECKING FOR IDLE/BUSY/REQUESTED PROCESS\n") ;
320:             PP; printf ("     : - before receive status from child process -\n");
321:             PP; printf ("     : idle_table[1]     = %d\n", idle_table[1]    );
322:             PP; printf ("     : idle_table[2]     = %d\n", idle_table[2]    );
323:
324:             if (idle_table[i] != PROCESS_IDLE)
325:             {
326:                MPI_Recv ( &process_status, 1, MPI_INT,
327:                           i, STATUS_TAG, MPI_COMM_WORLD, &status );
328:             }
329:             else
330:             {
331:                process_status = PROCESS_IDLE ;
332:             }
333:
334:             PP; printf ("     : - after receive status from child process -\n");
335:             PP; printf ("     : from processor    = %d\n", i                );
336:             PP; printf ("     : process_status    = %d\n", process_status   );
```

```
337:            PP; printf ("     : idle_table[1]     = %d\n", idle_table[1]    );
338:            PP; printf ("     : idle_table[2]     = %d\n", idle_table[2]    );
339:
340:            /*if son processor has idle_table = PROCESS_IDLE
341:                 it's mean that son processor has finished his job */
342:
343:           /*-------------------------------------------------------
344:            |  Perform appropriate action for the status/request.  |
345:            -------------------------------------------------------*/
346:            if (process_status == PROCESS_IDLE) idle_table[i] = PROCESS_IDLE ;
347:            if (process_status == PROCESS_BUSY) idle_table[i] = PROCESS_BUSY ;
348:
349:            PP; printf ("     : -- after CHANGE status table.\n") ;
350:            PP; printf ("     : idle_table[1]     = %d\n", idle_table[1]    );
351:            PP; printf ("     : idle_table[2]     = %d\n", idle_table[2]    );
352:
353:            if (process_status == ASK_FOR_IDLE_PROCESS)
354:            {
355:                PP; printf ("SEEKING FOR IDLE PROCESS FROM IDLE TABLE.\n") ;
356:
357:                idle_processor = UNKNOW ;
358:                for (j = 1; (j < process_n) && (idle_processor == UNKNOW); j++)
359:                {
360:                    PP; printf ("     : idle_table[%d]     = %d\n",
361:                                j, idle_table[j] );
362:
363:                    if (idle_table[j] == PROCESS_IDLE)
364:                    {
365:                            idle_processor = j;
366:                    }
367:                }
368:
369:                PP; printf ("     : idle_processor     = %d\n", idle_processor);
370:
371:                if (idle_processor != UNKNOW)
372:                   idle_table [idle_processor] = PROCESS_BUSY ;
373:
374:                PP; printf ("SEND IDLE PROCESS NUMBER.\n") ;
375:                PP; printf ("     : to                = %d\n", i );
376:                PP; printf ("     : idle_processor    = %d\n", idle_processor );
377:
378:                /* send No. of processsor j  to processor i  */
379:                MPI_Send ( &idle_processor, 1, MPI_INT,
380:                           i, IDLE_PROCESS_TAG, MPI_COMM_WORLD );
381:            }
382:            if (process_status == RESULT_READY)
383:            {
384:              /*-------------------------------------------------------
385:               |       Receive Result and collect it to List.         |
386:               -------------------------------------------------------*/
387:               MPI_Recv ( &recv_message, 1, MPI_L_String,
388:                          MPI_ANY_SOURCE, L_STRING_TAG, MPI_COMM_WORLD, &status);
389:
390:               PP; printf ("RECV MESSAGE : CHILD --> ROOT.\n"          );
391:               PP; printf ("     : iter     = %d\n", recv_message.iter );
392:               PP; printf ("     : label    = %s\n", recv_message.label);
393:               PP; printf ("     : str      = %s\n", recv_message.str  );
394:
395:               strcpy (result_list[result_index].label, recv_message.label);
396:               strcpy (result_list[result_index].str  , recv_message.str  );
397:               result_index ++ ;
398:
399:               idle_table[i] = PROCESS_IDLE ;
400:            }
401:        }
402:     }
403:
404:     /*=======================================================*/
405:     /*    Last Section :                                     */
406:     /*          Print all result from list and send          */
407:     /*          HALT control to all child process.           */
408:     /*=======================================================*/
409:
410:    /*-------------------------------------------------------
411:     |               Print result list.                      |
412:     -------------------------------------------------------*/
413:     PP; printf ("RESULT LIST:\n");
414:     for (i=0; i<result_index ; i++)
415:     {
416:         PP;
417:         printf ("    i=%2d ", i);
418:         printf ("label= %s",result_list[i].label);
419:
420:         printf (" str=<%s>\n", result_list[i].str        );
421:     }
422:
423:    /*-------------------------------------------------------
```

```
424:     |             Sort result list                          |
425:     ---------------------------------------------------------*/
426:     swap_occur = TRUE ;
427:     while (swap_occur == TRUE)
428:     {
429:        swap_occur = FALSE ;
430:        for (i=0; i<result_index-1; i++)
431:        {
432:           if (strcmp(result_list[i].label,result_list[i+1].label)IS_GREATER_THAN)
433:           {
434:              SwapLStr (&result_list[i], &result_list[i+1]);
435:              swap_occur = TRUE ;
436:           }
437:        }
438:     }
439:    /*---------------------------------------------------------
440:     |             Count End Time for Total Process           |
441:     ---------------------------------------------------------*/
442:
443:     entime1 = MPI_Wtime();
444:     entime  = MPI_Wtick();
445:     PP; printf ("TIME THAT COUNT BY MPI_Wtime = %lf\n", entime1  - sttime1);
446:     PP; printf ("TIME THAT COUNT BY MPI_Wtick = %lf\n", entime   - sttime );
447:     PP; printf ("PROCESS TERMINATED\n") ;
448:    /*---------------------------------------------------------
449:     |               Print result list.                       |
450:     ---------------------------------------------------------*/
451:     PP; printf ("RESULT LIST: AFTER SORT\n");
452:     for (i=0; i<result_index ; i++)
453:     {
454:         PP;
455:         printf ("    i=%2d ", i);
456:         printf ("label= %s",result_list[i].label);
457:         printf (" str=<%s>\n", result_list[i].str );
458:     }
459:
460:     strcpy (finished_str, "") ;
461:     for (i = 0; i < result_index; i++) strcat(finished_str, result_list[i].str) ;
462:     count = strlen (finished_str);
463:     PP;printf(" FINISHED STRING LENGTH = %d\n",count);
464:     PP;printf(" FINISHED STRING = $%s$\n",finished_str);
465:
466:
467:     /*---
468:      |    Do terminate all process.   |
469:                                    ---*/
470:     for (i = 1; i < process_n; i++)
471:     {
472:        process_control = NO_MORE_JOB ;
473:        MPI_Send ( &process_control , 1, MPI_INT,
474:                   i, PROCESS_CONTROL_TAG, MPI_COMM_WORLD );
475:     }
476:
477: /*****************************************************************/
478: /*              E N D   P A R E N T   P R O C E S S              */
479: /*****************************************************************/
480:   }else{
481: /*****************************************************************/
482: /*                                                               */
483: /*               C H I L D   P R O C E S S   P A R T             */
484: /*                                                               */
485: /*****************************************************************/
486:
487:     PP; printf ("JOB START : IN CHILD PROCESS:%d\n", rank) ;
488:     PP; printf ("     : process_status     = %d\n", process_status );
489:
490:     loop1_control = PERFORM_LOOP ;
491:
492:     while (loop1_control == PERFORM_LOOP)
493:     {
494:       /*=======================================================
495:        ||      Waiting for control from other process.       ||
496:        ||      At this point, process is in IDLE state.      ||
497:        =======================================================*/
498:        MPI_Recv ( &process_control, 1, MPI_INT,
499:                   MPI_ANY_SOURCE, PROCESS_CONTROL_TAG, MPI_COMM_WORLD, &status );
500:
501:        PP; printf ("WAIT/RECEIVE PROCESS CONTROL : IN CHILD PROCESS\n") ;
502:        PP; printf ("     : process_control = %d\n", process_control );
503:
504:        if (process_control == NO_MORE_JOB)
505:        {
506:           process_status = PROCESS_IDLE ;
507:           MPI_Send ( &process_status, 1, MPI_INT,
508:                      0, STATUS_TAG, MPI_COMM_WORLD );
509:
510:           loop1_control = TERMINATE_LOOP ;
```

```
511:        }
512:
513:        if (process_control == RULE_COMING)
514:        {
515:          /*---------------------------------------------------
516:           |               Receive Rule.                       |
517:           ---------------------------------------------------*/
518:           MPI_Recv ( &rule_all, 1, MPI_Rule,
519:                      MPI_ANY_SOURCE, RULE_TAG, MPI_COMM_WORLD, &status );
520:
521:           PP; printf ("RECV RULE :  \n");
522:           for (i=0; i<=rule_all.num_rule; i++)
523:           {
524:               PP; printf("     : i=%d  prob=%f  probx=%f  pred=%s  succ=%s \n",
525:                           i, rule_all.prob[i], rule_all.probx[i],
526:                              rule_all.pred[i], rule_all.succ [i] );
527:           };
528:
529:           loop1_control = PERFORM_LOOP ;
530:        }
531:
532:        if (process_control == NEW_JOB_COMING)
533:        {
534:          /*---------------------------------------------------
535:           |               Receive Message.                    |
536:           ---------------------------------------------------*/
537:           MPI_Recv ( &recv_message, 1, MPI_L_String,
538:                      MPI_ANY_SOURCE, L_STRING_TAG, MPI_COMM_WORLD, &status );
539:
540:           PP; printf ("RECV MESSAGE :  --> CHILD\n");
541:           PP; printf ("     : iter     = %d\n", recv_message.iter );
542:           PP; printf ("     : label    = %s\n", recv_message.label);
543:           PP; printf ("     : str      = %s\n", recv_message.str  );
544:
545:           process_status = PROCESS_BUSY ;
546:           MPI_Send ( &process_status, 1, MPI_INT,
547:                      0, STATUS_TAG, MPI_COMM_WORLD );
548:
549:           /* print for check the receiving */
550:
551:           PP; printf ("INFORMATION : IN CHILD PROCESS\n") ;
552:           PP; printf ("     : recv_message.str  = %s\n", recv_message.str );
553:           PP; printf ("     : process_status    = %d\n", process_status   );
554:
555:          /*---------------------------------------------------
556:           |               Perform job.                        |
557:           ---------------------------------------------------*/
558:           strcpy ( result_string, recv_message.str) ;
559:           count = 0;
560:           time(&t1);
561:           srand((long)t1);              /* put time in second to get seed */
562:
563:           for (i = 0; i < recv_message.iter; i++)
564:           {
565:            /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
566:             |               Do string operation                      |
567:             - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
568:
569:             /* PP; printf ("STRING OPERATION.\n"       ); */
570:             PP; printf ("STRING OPERATION.\n"       );
571:
572:             strcpy (result_string , "") ;
573:
574:             for (Lptr=recv_message.str; *Lptr != '\0' ; Lptr++)
575:             {
576:                strncpy (sub_str, Lptr, 1) ;
577:                strcat (sub_str, "") ;
578:                PP; printf ("     : sub string <%s>\n", sub_str);
579:                if (strcmp (sub_str,"F") IS_EQUAL)
580:                {
581:                   ran_val = 0.5 ;
582:                   ran_val = ran() ;
583:                   PP; printf ("     : random = %f\n", ran_val);
584:                   for (k=0 ; k<rule_all.num_rule; k++)
585:                   {
586:                      upper_bound = rule_all.probx[k+1] ;
587:                      lower_bound = rule_all.probx[k] ;
588:                      if ( ran_val >= lower_bound && ran_val < upper_bound)
589:                      {
590:                         strcat (result_string, rule_all.succ[k]) ;
591:                         PP; printf ( "     : k=%2d result_string <%s>\n",
592:                                      k, result_string );
593:                      }
594:                   }
595:                }
596:                else
597:                {
```

```
598:                 strcat (result_string, sub_str);
599:              }
600:           }
601:
602:           strcpy (recv_message.str, result_string);
603:
604:           count = strlen(result_string);
605:           PP; printf ("   iteration i=%2d count=%2d result_string <%s>\n",
606:                       i, count, result_string);
607:
608:          /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
609:           |    If there is to much job, give some to other.      |
610:           - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
611:           if (count > MAXLOAD && i < recv_message.iter-1)
612:           {
613:               PP; printf ("REQUEST FROM ROOT FOR OTHER IDLE PROCESS\n");
614:               PP; printf ("     : process requested = %d\n", rank     );
615:
616:              /*---
617:               | Ask for idle processor from ROOT process. |
618:                                                     ---*/
619:               process_status = ASK_FOR_IDLE_PROCESS ;
620:               MPI_Send ( &process_status, 1, MPI_INT,
621:                          0, STATUS_TAG, MPI_COMM_WORLD );
622:               MPI_Recv ( &idle_processor, 1, MPI_INT,
623:                          0, IDLE_PROCESS_TAG, MPI_COMM_WORLD, &status );
624:               process_status = PROCESS_BUSY ;
625:
626:               PP; printf (" --> : idle_processor    = %d\n", idle_processor);
627:
628:               if (idle_processor != UNKNOW) /* if there are any idle process */
629:               {
630:                  PP; printf ("IDLE PORCESS FOUND, PERFORM SEPARATE JOB\n");
631:
632:                  /**** Divide string into two part ****/
633:                  cut_num = count / 2 ;
634:
635:                  strcpy(temp_string, result_string);
636:
637:                  PP; printf ("     : -- before seperation -- \n"            );
638:                  PP; printf ("     : temp_string       = %s\n", temp_string  );
639:                  PP; printf ("     : result_string     = %s\n", result_string);
640:
641:                  strncpy(temp_result, temp_string , cut_num);
642:
643:                  PP; printf ("     : -- after  seperation -- \n"            );
644:                  PP; printf ("     : temp_result       = %s\n", temp_result  );
645:
646:                  strcpy(cut_result_string, temp_string + cut_num);
647:
648:                  strcpy(result_string, temp_result);
649:
650:                  PP; printf ("     : result_string     = %s\n", result_string);
651:                  PP; printf ("     : cut_result_string = %s\n",
652:                                                        cut_result_string);
653:
654:                 /*---
655:                  | Prepare information to be sent. |
656:                                                  ---*/
657:
658:                  sent_message.iter     =  recv_message.iter - i - 1;
659:
660:                  strcpy (sent_message.label, recv_message.label);
661:                  strcat (recv_message.label, "1");
662:                  strcat (sent_message.label, "2");
663:
664:                  strcpy (recv_message.str, result_string);
665:                  strcpy (sent_message.str, cut_result_string);
666:
667:                  PP; printf ("SEND MESSAGE FROM PROCESS %d --> %d\n",
668:                                rank, idle_processor);
669:                  PP; printf ("     : iter     = %d\n", sent_message.iter );
670:                  PP; printf ("     : label    = %s\n", sent_message.label);
671:                  PP; printf ("     : str      = %s\n", sent_message.str  );
672:
673:                  process_control = NEW_JOB_COMING ;
674:                  MPI_Send ( &process_control, 1, MPI_INT,
675:                             idle_processor,PROCESS_CONTROL_TAG,MPI_COMM_WORLD);
676:
677:                  MPI_Send ( &sent_message, 1, MPI_L_String,
678:                             idle_processor, L_STRING_TAG ,MPI_COMM_WORLD);
679:
680:               }else{
681:                  /*
682:                  In case idle_processor = UNKNOW, this mean
683:                  there are no any processor is free. This
684:                  process has to perform the job by itself
```

```
                    for next one iteration step, and back to
                    look for idle process again later.
                    */
                 }/*end if*/
               }
             }
            /*-------------------------------------------------------
             |     Job Finish, send result back to ROOT process.    |
             -------------------------------------------------------*/
             sent_message.iter     =  0 ;
             strcpy (sent_message.label, recv_message.label);
             strcpy (sent_message.str  , result_string);

             PP; printf ("SEND RESULT BACK TO ROOT PROCESS.\n");
             PP; printf ("     : iter     = %d\n", sent_message.iter );
             PP; printf ("     : label    = %s\n", sent_message.label);
             PP; printf ("     : str      = %s\n", sent_message.str  );

             process_status = RESULT_READY ; /* idle_table will be set to IDLE.*/
             MPI_Send ( &process_status, 1, MPI_INT,
                        ROOT_PROCESS, STATUS_TAG, MPI_COMM_WORLD);
             MPI_Send ( &sent_message, 1, MPI_L_String,
                        ROOT_PROCESS, L_STRING_TAG, MPI_COMM_WORLD);

             loop1_control = PERFORM_LOOP ;
          }/*end if*/
       }
/*****************************************************************/
/*                E N D   C H I L D   P R O C E S S              */
/*****************************************************************/
  }
MPI_Finalize();/* finished using MPI function */
} /*  end main  */


void SwapLStr ( struct L_String *x, struct L_String *y)
{
   struct L_String temp ;

   temp.iter = x->iter ;
   strcpy (temp.label,x->label);
   strcpy (temp.str  ,x->str);

   x->iter = y->iter ;
   strcpy (x->label,y->label);
   strcpy (x->str  ,y->str);

   y->iter = temp.iter ;
   strcpy (y->label,temp.label);
   strcpy (y->str  ,temp.str);
}


int All_process_idle()
{
   int i, check_process_idle;

   check_process_idle = TRUE;

   /*
   PP; printf("CHECK ALL PROCESS IDLE\n");
   */
   for(i=1; i<process_n; i++)
   {
       /*
       PP; printf("     : i = %d idle_table = %d \n", i,idle_table[i] );
       */
       if(idle_table[i] != PROCESS_IDLE) check_process_idle = FALSE ;
   }
       /*
       PP; printf("     : return = %d \n", check_process_idle );
       */
   return check_process_idle;
}
```

Pseudo Code

```
Initialize MPI Engine
Declare necessary types and variables
IF (rank = 0) THEN
    GET rule_all, axiom_string and desired derivation_length
    SEND process_control=RULE_COMING to all child processes
    SEND rule_all to all child processes
    SET idle_table=PROCESS_BUSY for 1st child process
    attach label ‘1’ to axiom_string
    SEND process_control=NEW_JOB_COMING to 1st child processes
    SEND derivation_length & axiom_string to 1st child processes
    WHILE (there are some child process still busy) DO
        FOR all child process
            IF (this child process busy) THEN
                WAIT & RECEIVE process_status
                IF process_status = ASK_FOR_IDLE_PROCESS THEN
                    look for idle process from idle table
                    IF found an idle process THEN
                        SET idle_table=PROCESS_BUSY for this idle process
                        SEND rank of idle process to process requested for it
                    ELSE
                        SEND message UNKNOW to tell no any idle process
                    ENDIF
                ENDIF
                IF process_status = RESULT_READY THEN
                    RECEIVE result_string
                    APPEND result_string into result_list
                    SET idle_table for this process to PROCESS_IDLE
                ENDIF
            ENDIF
        NEXT child process
    ENDWHILE
    SNED process_control=NO_MORE_JOB to all child processes
    sort result_list on their label
    write down final result string
ELSE
    SET loop_control = PERFORM_LOOP
    WHILE (loop_control = PERFORM_LOOP) DO
        WAIT & RECEIVE process_control from another process
        IF (process_control = NO_MORE_JOB) THEN
            SEND process_status = PROCESS_IDLE to parent process
            SET loop_control = TERMINATE_LOOP
        ENDIF
        IF process_control = RULE_COMING THEN
            RECEIVE rule_all
            SET loop_control = PERFORM_LOOP
        ENDIF
        IF process_control = NEW_JOB_COMING THEN
            RECEIVE derivation_length, initial_string
            SEND process_status = PROCESS_BUSY to master process
            FOR all derivation step needed
                PERFORM substitute character in initial_string by means of rule_all
                IF result_string is too long THEN
                    SEND process_status=ASK_FOR_IDLE_PROCESS to master process
                    RECEIVE answer from master process
                    IF answer is not UNKNOW THEN
```

```
56:                     devide result_string to two sub-strings
57:                     APPEND ‘1’ to 1st sub-string label
58:                     APPEND ‘2’ to 2nd sub-string label
59:                     SEND process_control=NEW_JOB_COMING to the idle process
60:                     SEND 2nd sub-string to the idle process
61:                     COPY 1st sub-string to result_string
62:                 ENDIF
63:             ENDIF
64:             COPY result_string to initial_string
65:         NEXT derivation step
66:         SEND process_status=RESULT_READY to master process
67:         SEND result_string to master process
68:         SET loop_control = PERFORM_LOOP
69:       ENDIF
70:     ENDWHILE
71:  ENDIF
```

Appendix B

Tree Generated of stochastic L-system

In this research, a lot of stochastic L-system trees were generated. With appropriate visualized technique, most of them are look pretty good. Some of them are shown as following.

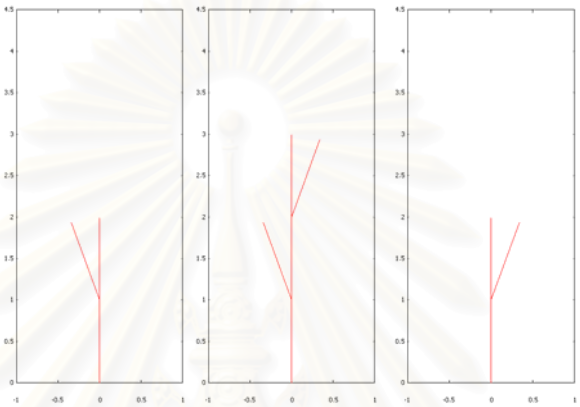


Figure B.1 Stochastic tree structure with derivation length of 1.

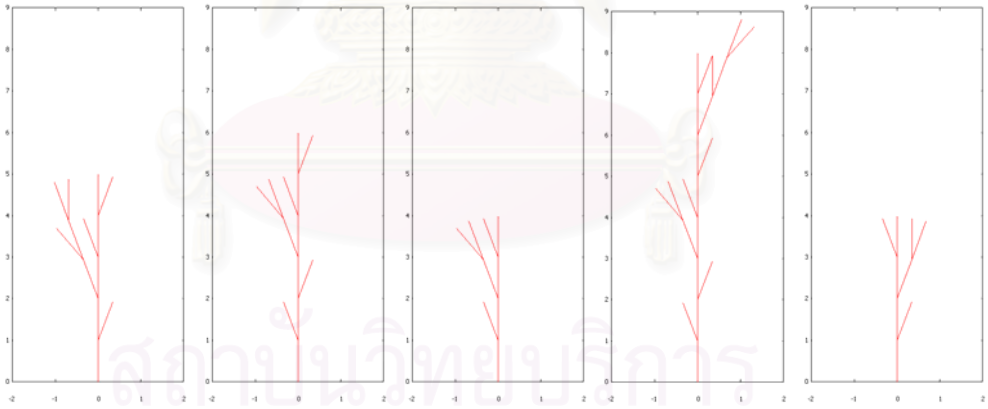


Figure B.2 Stochastic tree structure with derivation length of 2.

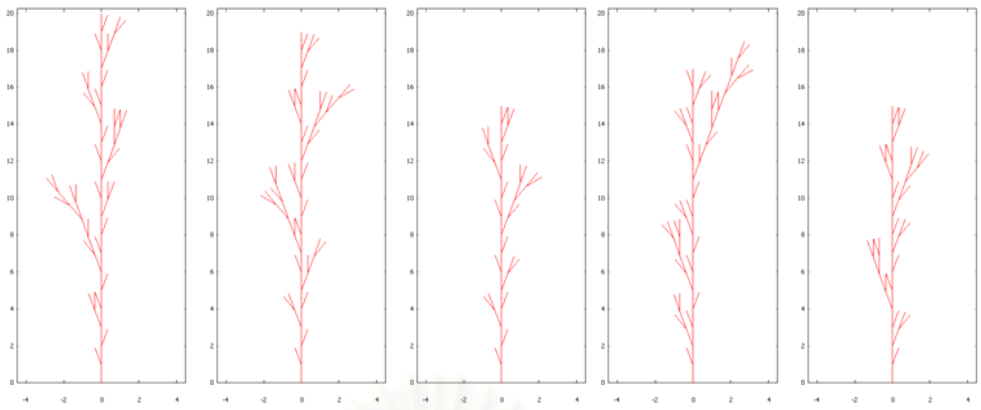


Figure B.3 Stochastic tree structure with derivation length of 3.

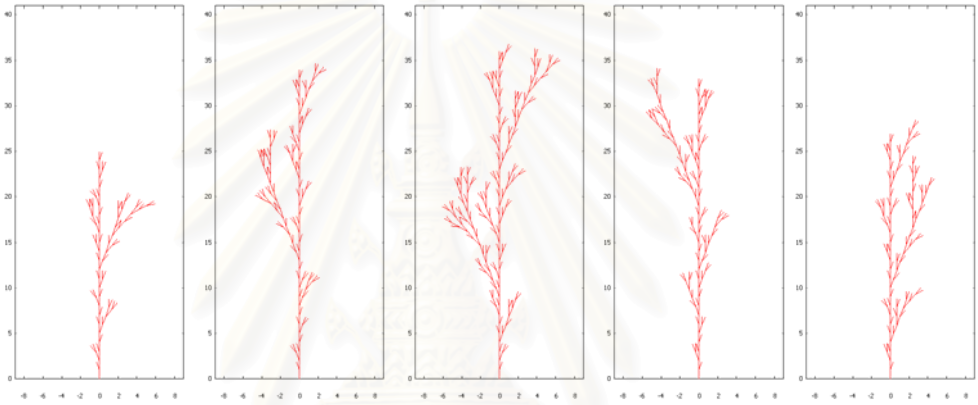


Figure B.4 Stochastic tree structure with derivation length of 4.

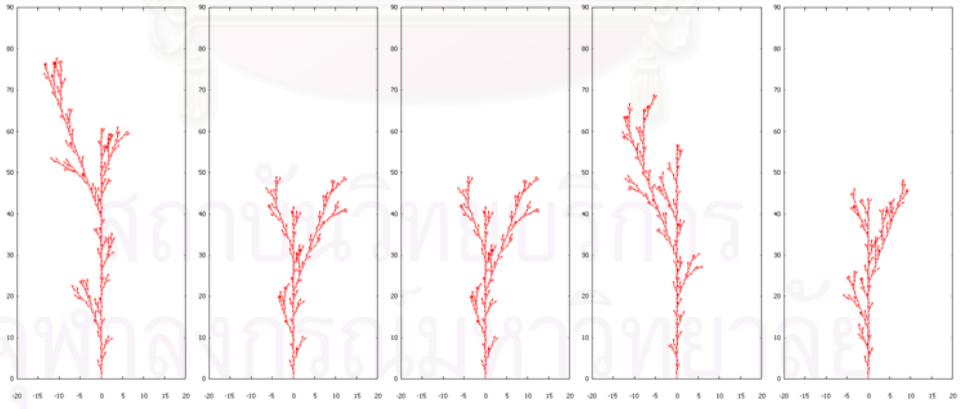


Figure B.5 Stochastic tree structure with derivation length of 5.

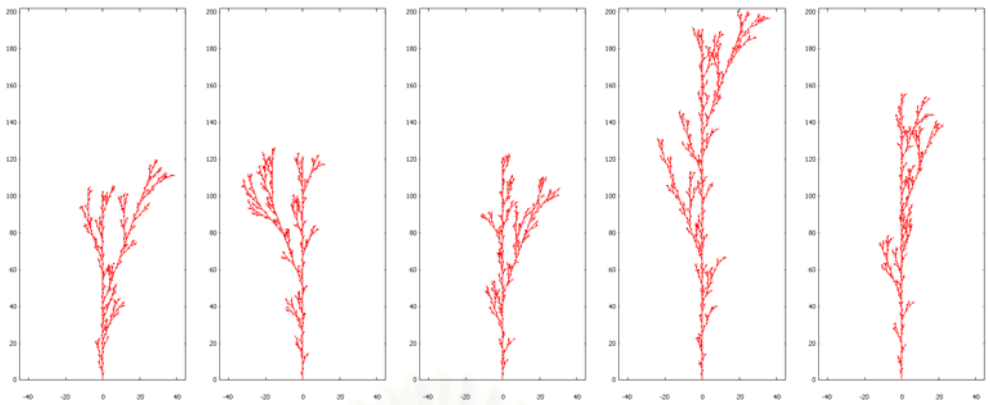


Figure B.6 Stochastic tree structure with derivation length of 6.

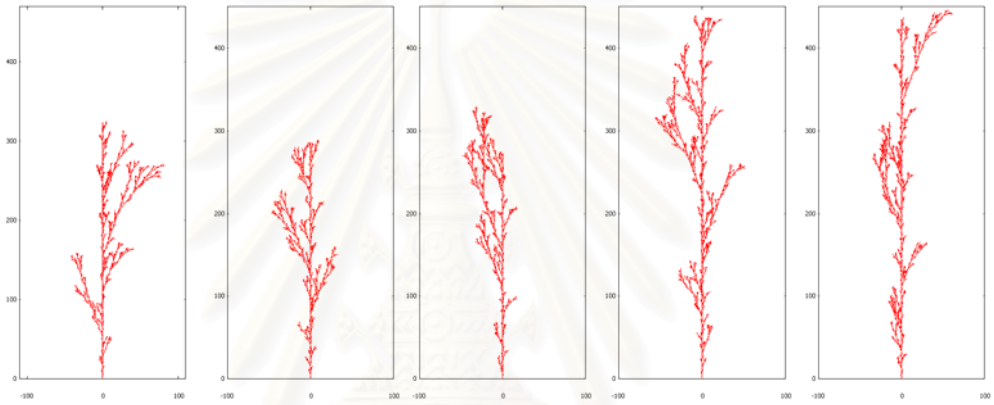


Figure B.7 Stochastic tree structure with derivation length of 7.

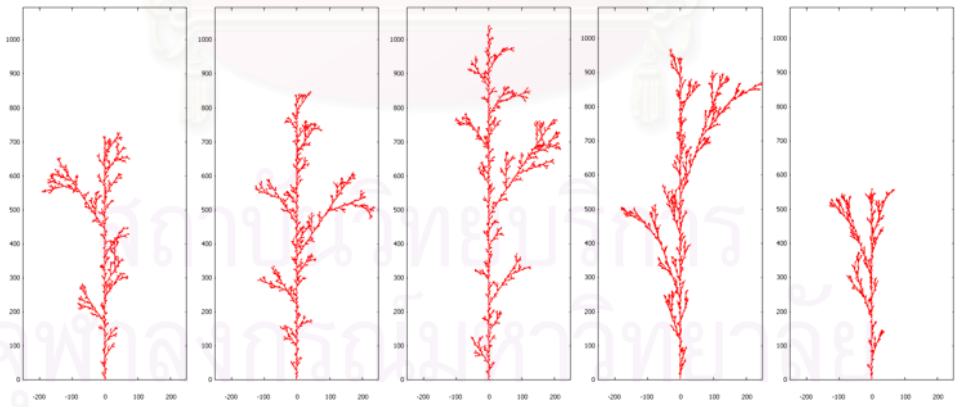


Figure B.8 Stochastic tree structure with derivation length of 8.

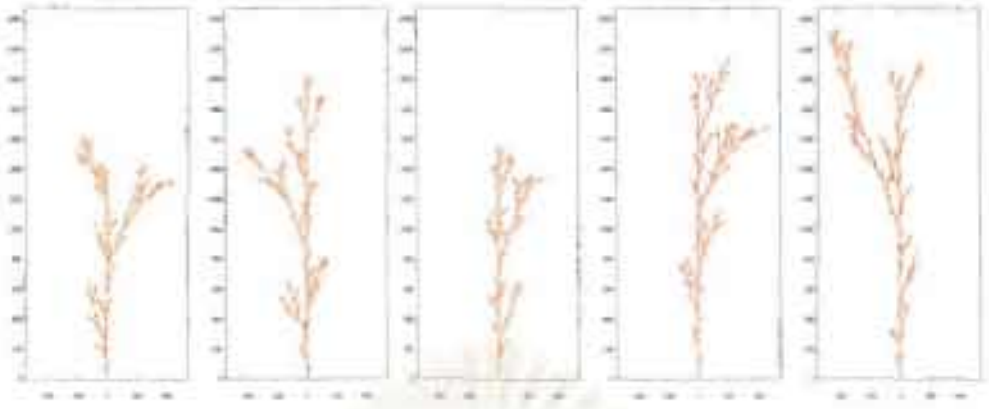


Figure B.9 Stochastic tree structure with derivation length of 9.

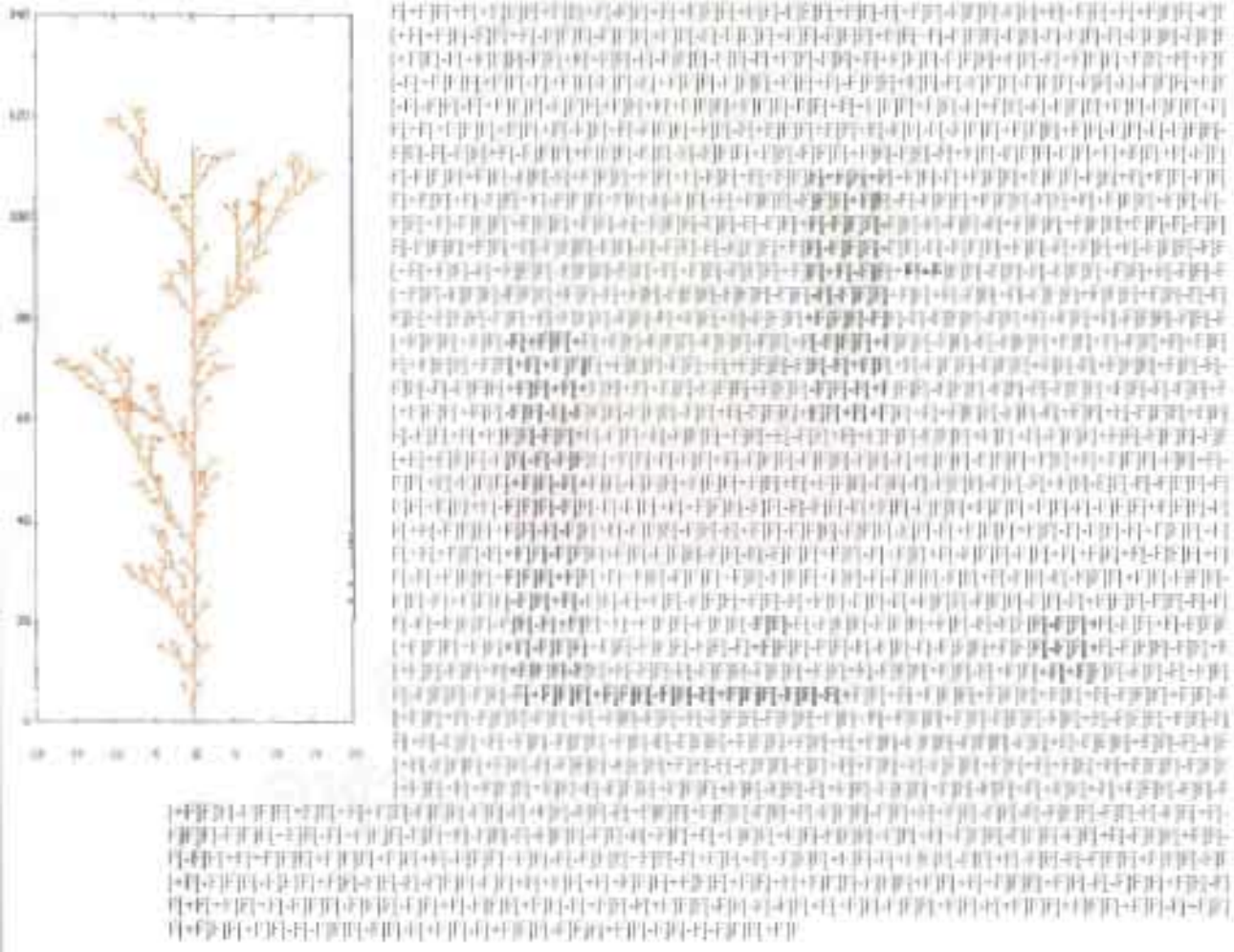


Figure B.10 String, consist of 4,157 charters, generated by Stochastic L-System with derivation length of 6 and their graphic visualization.

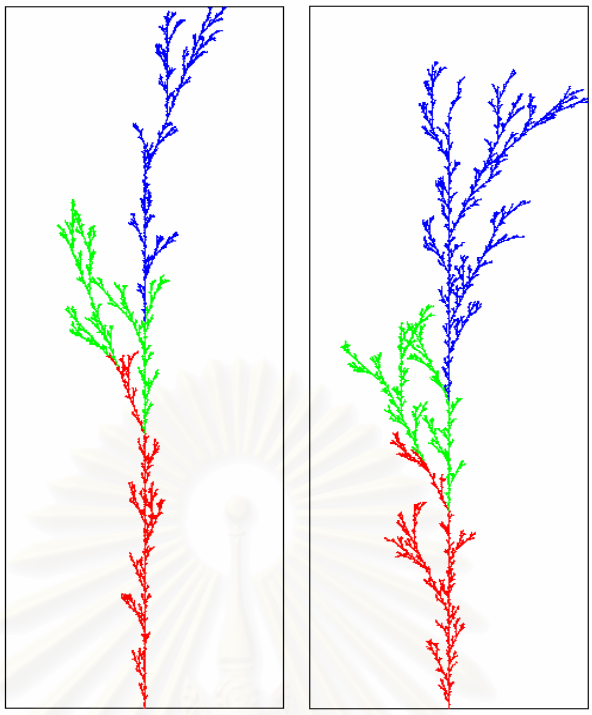


Figure B.11 Stochastic tree structure generated by 4 processes (1 parent and 3 children) with derivation length of 9. Each color represents string generated by each process.

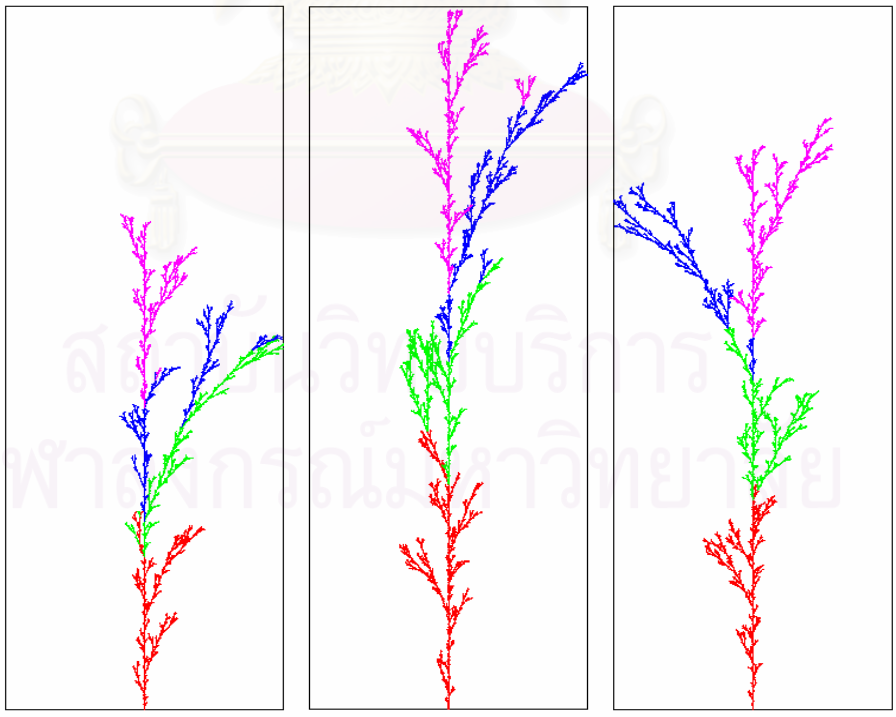


Figure B.12 Stochastic tree structure generated by 5 processes (1 parent and 4 children) with derivation length of 9. Each color represents string generated by each process.

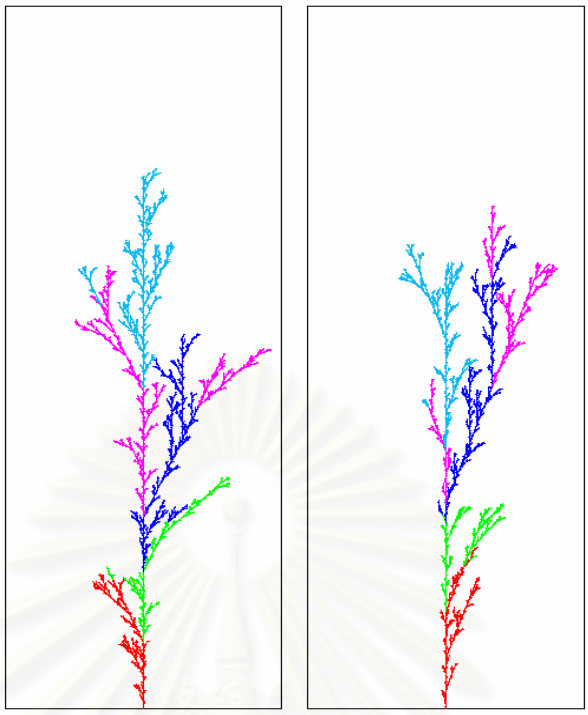


Figure B.13 Stochastic tree structure generated by 6 processes (1 parent and 5 children) with derivation length of 9. Each color represents string generated by each process.

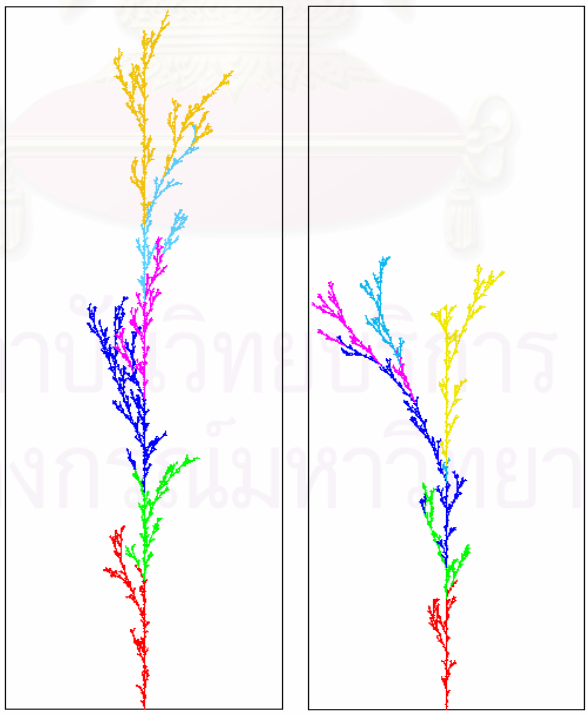


Figure B.14 Stochastic tree structure generated by 7 processes (1 parent and 6 children) with derivation length of 9. Each color represents string generated by each process.

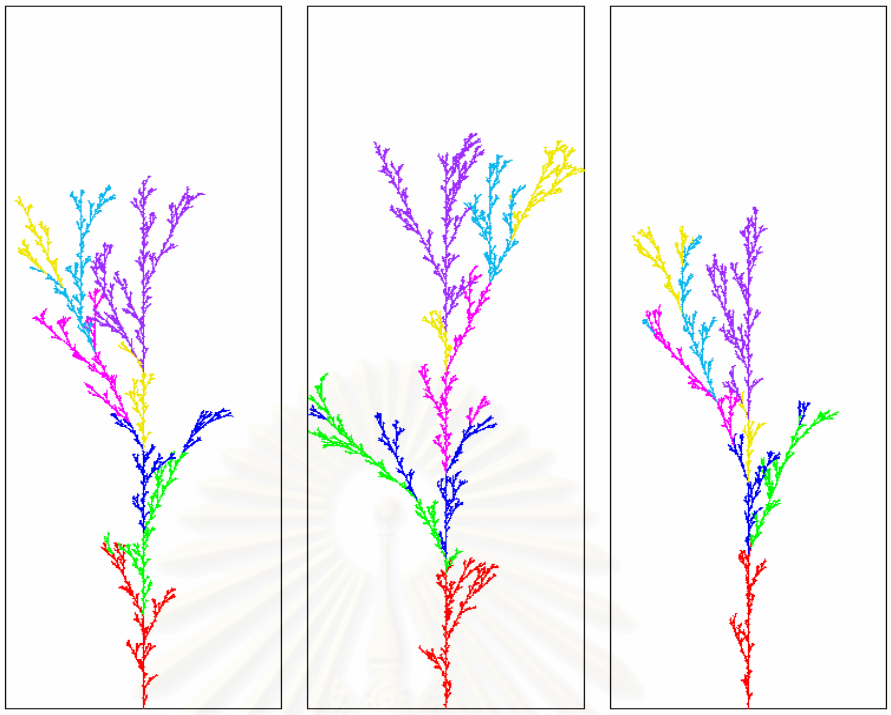


Figure B.15 Stochastic tree structure generated by 8 processes (1 parent and 7 children) with derivation length of 9. Each color represents string generated by each process.

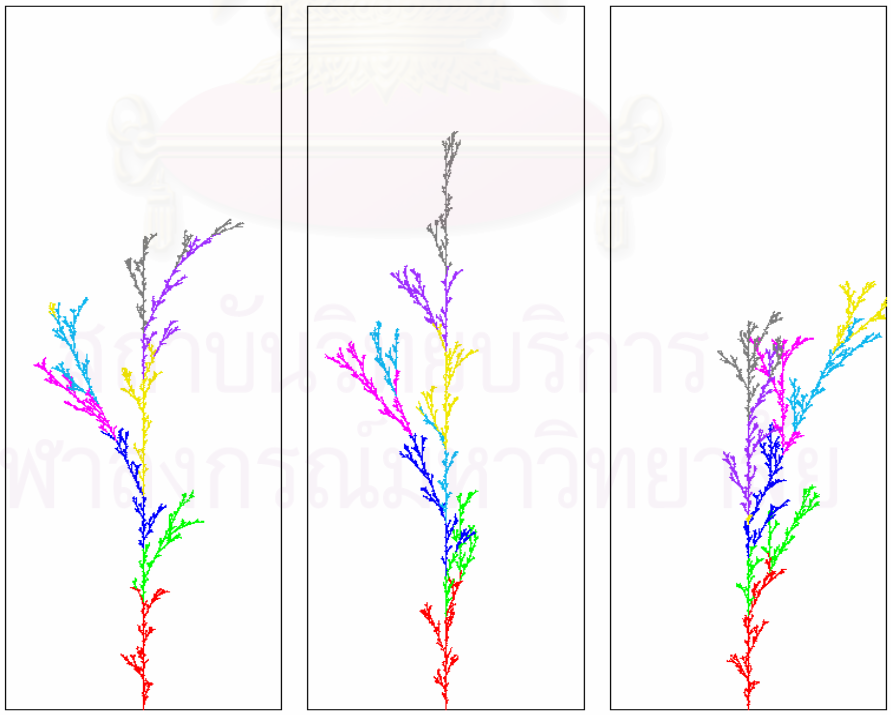


Figure B.16 Stochastic tree structure generated by 9 processes (1 parent and 8 children) with derivation length of 9. Each color represents string generated by each process.

Appendix C

The Growth of Functions

To show that the relationship between derivation length, n, with length of result string, $L(n)$, generated from L-system program is $O(m^{n+1})$, we need to verify there are constants C and p such that $|L(n)| \le C|m^{n+1}|$, whenever $n > p$. Suppose that

- k represents number of symbols in *axiom.*
- k_1 represents number of symbols in the *axiom*, which is not going to be substituted.
- k_2 represents number of symbols in the *axiom*, which is going to be substituted, due to rewriting rules.
- m represents number of symbols in rewriting rule which contain largest number of rewriting symbol.
- m_1 represents number of symbols in the rewriting rule, which is not going to be substituted.
- m_2 represents number of symbols in the rewriting rule, which is going to be substituted.

For an axiom:

$$n = 0, \quad L(0) = k = k_1 + k_2$$

Since only k_2 symbols are going to be substituted by a string of m symbols on the next derivation step. The upper bound of the length of final string can be calculated as follows.

$$\begin{aligned} n = 1, \quad L(1) &= k_1 + k_2 \cdot m \\ &= k_1 + k_2 \cdot (m_1 + m_2) \\ &= k_1 + k_2 \cdot m_1 + k_2 \cdot m_2 \end{aligned}$$

The resulting string contains only $k_2 \cdot m_2$ rewriting symbols. Hence,

$$\begin{aligned} n = 2, \quad L(2) &= k_1 + k_2 \cdot m_1 + (k_2 \cdot m_2) \cdot m \\ &= k_1 + k_2 \cdot m_1 + (k_2 \cdot m_2) \cdot (m_1 + m_2) \\ &= k_1 + k_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_2 \end{aligned}$$

$$\begin{aligned} n = 3, \quad L(3) &= k_1 + k_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_1 + (k_2 \cdot m_2 \cdot m_2) \cdot m \\ &= k_1 + k_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_1 + (k_2 \cdot m_2 \cdot m_2) \cdot (m_1 + m_2) \\ &= k_1 + k_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_2 \cdot m_1 + k_2 \cdot m_2 \cdot m_2 \cdot m_2 \end{aligned}$$

It can be assumed that

$$L(n) = k_1 + k_2 \cdot m_1 \left(1 + m_2 + m_2^2 + \mathrm{K} + m_2^{n-1}\right) + k_2 \cdot m_2^n$$

Since $k_2 \cdot m_1 \cdot m_2^i \le k_2 \cdot m_2^n$ for any $i < n$, $L(n)$ **becomes.**

$$L(n) = k_1 + \underbrace{k_2 \cdot m_1 + k_2 \cdot m_1 \cdot m_2 + k_2 \cdot m_1 \cdot m_2^2 + \ldots + k_2 \cdot m_1 \cdot m_2^{n-1}}_{\text{n-terms}} + k_2 \cdot m_2^n$$

$$\le k_2 \cdot m_2^n + \underbrace{k_2 \cdot m_2^n + \ldots + k_2 \cdot m_2^n}_{\text{n-terms}} + k_2 \cdot m_2^n$$

$$\le (n+2) \cdot k_2 \cdot m_2^n$$

$$\le k_2 \cdot m_2^{n+1}$$

The inequality $|L(n)| \le C|m^{n+1}|$ hold whenever $n > p$, if we take $C = k_2$ and $p = 2$. Hence, it can be concluded that the value of $L(n)$ is $O(m^{n+1})$.

Vitae

Supaporn Kamklad was born in 1962. She received a Bachelor Degree in Mathematics from Department of Mathematics, Chulalongkorn University in 1989. In 1990-1993, she worked for DATAMAT Co. Ltd. as an assistant system analyst. She is expected to complete Masters degree in Computational Science form Chulalongkorn University in 2003.