PROGRESSES IN PARALLEL RANDOM NUMBER GENERATORS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÜLİN KAŞIKARA TENEKECİOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Supervisor

Examining Committee Members

Prof. Dr. Volkan Atalay	(METU, CENG)	
Prof. Dr. Ayşe Kiper	(METU, CENG)	
Assoc. Prof. Dr. Veysi İşler	(METU, CENG)	
Assoc. Prof. Dr. Tayyar Şen	(METU, IE)	
Assist. Prof. Dr. Halit Oğuztüzün	(METU, CENG)	

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: GÜLİN KAŞIKARA TENEKECİOĞLU

Signature :

ABSTRACT

PROGRESSES IN PARALLEL RANDOM NUMBER GENERATORS

Tenekecioğlu Kaşıkara, Gülin
M.Sc., Department of Computer Engineering
Supervisor : Prof. Dr. Ayşe Kiper

September 2005, 132 pages

Monte Carlo simulations are embarrassingly parallel in nature, so having a parallel and efficient random number generator becomes crucial. To have a parallel generator with uncorrelated processors, parallelization methods are implemented together with a binary tree mapping. Although, this method has considerable advantages, because of the constraints arising from the binary tree structure, a situation defined as problem of falling off the tree occurs. In this thesis, a new spawning method that is based on binary tree traversal and new spawn processor appointment is proposed to use when falling off the tree problem is encountered. With this method, it is seen that, spawning operation becomes more costly but the independency of parallel processors is guaranteed. In Monte Carlo simulations, random number generation time should be unperceivable when compared with the execution time of the whole simulation. That is why; linear congruential generators with Mersenne prime moduli are used. In highly branching Monte Carlo simulations, cost of parameterization also gains importance and it becomes reasonable to consider other types of primes or other parallelization methods that provide different balance between parameterization cost and random number generation cost. With this idea in mind, in this thesis, for improving performance of linear congruential generators, two approaches are proposed. First one is using Sophie-Germain primes as moduli and second one is using a

hybrid method combining both parameterization and splitting techniques. Performance consequences of Sophie-Germain primes over Mersenne primes are shown through graphics. It is observed that for some cases proposed approaches have better performance consequences.

Keywords: Parallel random number generation, parameterization methods, linear congruential genetators, binary tree mapping, problem of falling off the tree

ÖZ

PARALEL RASTGELE SAYI GENERATÖRLERİNDE GELİŞMELER

Tenekecioğlu Kaşıkara, Gülin
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ayşe Kiper

September 2005, 132 sayfa

Monte Karlo simülasyonları doğaları gereği paralel yapıya sahip oldukları için, paralel ve verimli bir rasgel sayı üreticisinin mevcut olması çok önemlidir. Aralarında korelasyon olmayan işlemcilere sahip paralel bir üretici oluşturabilmek için, paralelleştirme metodları, ikili ağaç yapısı eşleştirmesi ile birlikte uygulanır. Bu şekilde bir paralelleştirmenin hatırı sayılır bir çok avantajı olmasına rağmen, ikili ağaç yapısının getirdiği kısıtlamalardan dolayı ağaçtan düşme durumu oluşur. Bu tez çalışmasında, ikili ağaç yapısının taranması ve yeni üretim işlemcisinin atanması işlemlerine dayanan bir üretim metodu önerilmektedir. Bu yeni metod ile, üretim işlemi süre olarak artmış fakat, paralel işlemcilerin birbirinden bağımsız olmaları garantilenmiştir. Monte Karlo simulasyonlarında, rasgele sayı üretim süresinin, toplam simulasyon süresi ile karşılaştırıldığında farkedilmeyecek seviyede olması gerekir. Bu nedenle, doğrusal kongruent üreticilerde, modülüs olarak Mersenne asalları kullanılır. Çok dallanan bir Monte Karlo simülasyonunda, parametrikleştirme maliyeti de önem kazanır. Bu gibi durumlarda, parametrikleştirme maliyeti ile rasgele sayı üretim maliyeti arasında farklı bir denge kurabilen, farklı asalları veya paralleştirme metodlarını incelemek gerekebilir. Bu düşünceden yola çıkarak, tez kapsamında, doğrusal kongruent üreticilerin performans değerlerinin yükseltilmesine yönelik iki yaklaşım önerilmektedir. İlk yaklaşım, modülüs olarak

Sophie-Germain asallarının kullanılması, ikinci yaklaşım ise, parametrikleştirme ve bölme metodlarını birleştiren hibrit bir paralelleştirme metodunun kullanılmasıdır. Sophie-Germain asallarının, Mersenne asallarına göre performans ölçümleri grafikler ile gösterilmekte ve bazı durumlarda, önerilen yaklaşımların daha iyi performans değerleri oluşturduğu görülmektedir.

Anahtar Kelimeler: Paralel rasgele sayı üretimi, parametrikleştirme metodları, doğrusal kongruent üreticiler, ikili ağaç eşleştirmesi, ağaçtan düşme problemi

To My Husband

ACKNOWLEDGMENTS

I would like to thank to my supervisor Prof. Dr. Ayşe Kiper for her guidance, advice and criticism in writing of this thesis.

I would like to express my deepest gratitude to my husband, Gökhan who encouraged and supported me throughout the study.

I would like to thank to my parents and my sister, who motivated me all the time and always be with me whenever needed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Random Numbers and Monte Carlo Simulations

The property of randomness lies on the foundation of stochastic simulations and MC (Monte Carlo) methods. Simulating stochastic methods require a source of randomness. A question which follows is what can be considered random? In ordinary language, the word random is used to express apparent lack of purpose or cause. This suggests that no matter what the cause of something, its nature is not only unknown but the consequences of its operation are also unknown. Various disciplines handle the word random differently. In natural sciences, random takes on an operational meaning. Eveything that have undetermined or uncontrolled causes, is considered as random. Whereas, in statistics, random means some event happens with some probability distribution. In computing, the term random generally refers to generating or using a set of truly random sequence of RNs (Random Numbers) within some set range. For the purposes of simulation and MC computations, randomness becomes a valuable resource since these methods require a large supply of RNs, or means to generate them on demand [1].

RNs are used extensively in simulation of stochastic systems, statistical experiments, modeling, probabilistic algorithms, computer games, gambling machines and in numerical analysis with MC methods. In simulation, RNs are used to randomly pick event outcomes based on statistical or experiential data. In statistics, RNs are used to generate data with a particular distribution to calculate statistical properties when analytic techniques fail. In modeling, RNs are used to model random processes in nature such as those arising in ecology or economics. Moreover, RNs are widely used in cryptography to hide information from others. Besides, RNs are also used in games, computer programming, for interaction with the user or for decision making [2, 3].

RNs can be classified according to source of randomness. Formally, there are three types of RNs. Truly random, pseudorandom and quasirandom numbers. Truly random is defined as exhibiting true randomness. These RNs can be the result of a physical process such as timing clocks, circuit noise, Geiger counts, or bad memory. Pseudorandom numbers are defined as having the appearance of randomness, but exhibiting a specific repeatable pattern. Quasirandom is defined as filling the solution space sequentially in fact; these sequences are not at all random. They can be defined as half way between random and uniform grid.

RNs are computed using deterministic algorithms. So, they are defined as pseudorandom numbers. In this thesis, what is meant by an RN is in fact a pseudorandom number. In an RN sequence, next number is created by a function called generator, which takes as input one or more previous numbers from the sequence and generates the next number according to the predefined formula. The resulting sequence created by this way is expected to look statistically independent and uniformly distributed. RN sequences possess the following properties as a result of their cyclical structures. The sequence consists of finite number of integers and begins to repeat itself when the period is exceeded. Besides, the sequence is traversed in a particular order and the integers in it need not be distinct [4].

There are several RNGs (Random Number Generator). The most commonly used generators are LCG (Linear Congruentail Generator) and LFG (Lagged Fibonacci Generator). The LCG is first proposed for use by Lehmer in 1949 and is referred as the Lehmer generator. LCG can be defined by a recursion formula which in short represented as LCG(a, b, m, X_0) where a is the multiplier, b is additive constant, X_0 is seed, and m is the modulus. The LFG is becoming popular since it offers a simpler and faster method for obtaining higher periods. LFG is defined by LFG(l, k, m) where l and k are the lag values and m is the modulus. With proper chosen parameters, good RN sequences can be constructed from these generators.

When RN generation is considered, the question comes in mind, why use a deterministic computer algorithm instead of a truly random mechanism for generating RNs? Using a program is more convenient than throwing dice or picking balls from a box and entering the corresponding numbers on a computer's keyboard, especially when thousands of RNs are needed for a computer experiment. Attempts have been made at constructing RNGs from physical devices such as noise diodes, gamma-ray counters, and so on, but these remain largely impractical and unreliable, because they are not practical, and it is generally not true

that the successive numbers that they produce are independent and uniformly distributed. Besides, there are problems related with such sequences like being too slow, expensive and having low quality. In order not to have a purely deterministic algorithm, combining the output of a well designed RNG with some physical noise can be considered [5].

RNs are used on various fields. Among these fields, the area that makes the most extensive use of RNs is MC methods. MC methods are described as any computational method that uses RNs as an essential part of the algorithm. MC methods can be used in everything from economics to nuclear physics. With MC methods, a complex system is sampled in a number of random configurations, and that data can be used to describe the system as a whole [6].

When using MC simulation, RNs are used to determine attributes of particles, and interactions of particles with the medium. By stating the expected properties of attributes of particles, the desirable properties for RN sequences can be understood. For an MC simulation to be effective, the attributes of each particle should be independent of those attributes of any other particle and the attributes of particles should be able to fill the entire attribute space in a uniform way. When these aspects are applied to RNs, there appear several requirements for RNGs. Briefly speaking; the most important requirements can be summarized as being uncorrelated, uniform and having long period. When parallel architectures are considered, it is worth to consider additional aspects like low initialization overhead and no inter-processor communication [4].

The MC methods are often referred to as the method of last resort since they consume large computing resources and require very long runs. As computers become more powerful, MC methods become more commonly used. In a large-scale MC simulation literally millions or even billions of RNs are required [4]. That is why; the process of RN generation should ideally be very efficient. By the advent of supercomputers and more advanced parallel architectures, in order to get higher speed, MC methods begin to make extensive use of parallel computers, since these calculations are particularly well suited to such architectures and often require very long runs. A common way to parallelize MC is to put identical clones on the various processors, only the RN sequences are different. It is therefore important that to have a parallel MC, the underlying RNs must also be created in parallel in an uncorrelated manner [7]. Necessity to get higher speed becomes the driving force for the parallelization of RNs. Although, MC methods are known as embarrassingly parallel, the truth of this notion depends highly on the quality of the PRNG (Parallel Random Number Generator) used [4].

In many problems for which RNs are most heavily used like MC methods, it has been discovered that the quality of the RNs can influence the results. This is especially true in large-scale simulations on parallel supercomputers, which consume huge quantities of RNs, and require parallel algorithms for random number generation. As computers become more powerful and MC methods become more commonly used, the quality of RN sequence and the parallelization schemes become more important [3] .

There are two main approaches for the parallelization of RNG, which are called splitting and parameterization. Splitting approach depends on the idea of dividing the underlying sequence of RNs into different processors. Each processor uses the same iteration formula but with widely separated seed values [7]. Parameterization is another approach for generating parallel RNs. The exact meaning of parameterization depends on the type of RNG used. This method identifies a parameter in the underlying recursion of an RNG that can be varied. Each valid value of this parameter leads to a recursion that produces a unique, full-period stream of RNs [8,9].

When the PRNGs are considered, one should take care of the following aspects. The generation of the streams must be reproducible and without any inter-processor communication. The generator must be portable between serial and parallel platforms. Lastly, the generator must provide high quality RNs in a computationally inexpensive and scalable manner [10]. In order to achieve all these aspects, a technique for mapping a large number of parameterized RNGs onto a binary tree is utilized to permit an efficient, portable and reproducible MIMD implementation [11]. The point of using a binary tree to map the parallel processors is that one defines an entire subtree with each assignment and ensures that processors elsewhere in the computation can not accidentally assign the same processor. In addition, the computation of what node and subtree follow can be done with only local information, without any inter-processor communication [11]. Although, this mapping works fine for a considerable number of proseccors, problems can arise from the fact that the binary tree structure puts an upper limit on the number of processors that a processor can spawn with its local information. This problem is named as the problem of *Falling off the tree*.

When considering the parameterization of LCG, modulus plays an important role and lies on the heart of two costly operations. First one is named as the initialization cost occurring as a result of parameterized iteration, and the second one is the cost per RN generation resulting from the modular reduction [10]. In order to reduce the cost per RN generation, Mersenne primes are used as moduli [10]. By using a Mersenne prime, the cost of modular reduction is reduced but the initialization cost remains still.

1.2 Motivation

In this thesis, the subject of parallel random number generation is discussed from two different perspectives. Firstly, the binary tree mapping is considered and a solution is proposed to the falling off the tree problem. Secondly, performance consequences of LCGs with prime moduli are examined and two approaches are proposed in order to obtain better performance measures.

Problem of falling off the tree can be a disastrous situation if application in hand requires huge amounts of independent processors. In order to handle this problem, firstly, a naïve solution is given where the seed is changed and new processors are pointed to nodes elsewhere in the tree that may or may not be independent from those already been used [11]. Later, another solution is given in [2]. This solution enables the user to have access to all the independent processors. What is left to the user is to pick a generator with as many processors as maximum number of processors needed. So long as an upper bound can be established that is smaller than the number of processors for some generator, the user need not fear falling off the tree. But in such a case, the user must do the bookkeeping and take great care not to choose a processor that has already been used [12]. In this thesis, a new spawning method that is based on several spawn routes and two spawn algorithms is suggested as a solution to the problem of falling off the tree. Binary tree structure is traversed according to different spawn routes like breadth first search, inorder search, preorder search, postorder search, random search and least recently used search and the available processor is assigned as the new parent and from now on, this processor acts as if it is the original processor that requested the spawn call and starts creating new processors only with its local information. Although this method requires inter-processor communication for determining the new spawn processor, it can be useful since it enables automatic assignment of the next spawn processor without user interaction. By this way, not only the spawning operation can continue until all the nodes in the binary tree are exhausted, but

also, it is assured that all the newly created processors are independent from each other and from the already existing ones. To minimize the inter-processor communication, two spawn algorithms are also proposed. These algorithms rely on the fact that a processor which has spawned before is more probably to spawn in the future.

As a part of this thesis, the parameterization of LCG is implemented on PVM (Parallel Virtual Machine) system. The initialization and RN generation costs of LCGs are examined in detail. When the moduli is high, it is reasonable to ask if the reduced cost of modular reduction obtained when using a Mersenne prime is balanced by the increased initialization cost. In situations, where several processors with short periods are needed, one should consider other schemes that have a different balance between the cost per RN and the initialization cost. With this idea in mind, two approaches that are mentioned as future research topics in [10] are implemented and their effects on the performance consequences of LCGs are discussed.

First approach is based on changing the type of prime that is used as moduli. Instead of using a Mersenne prime, a Sophie-Germain prime is used as modulus. As a result of its nature, Sophie-Germain prime reduces the initialization cost considerably. The price paid for this is having to use standard modular multiplication [10]. LCG with Sophie-Germain prime is implemented and the comparison based analyis of Mersenne prime and Sophie-Germain prime is done case by case through speed up and efficieny graphics.

Second approach is based on the idea of changing the parallelization method, which is accomplished by utilizing a technique where splitting and parameterization are used together. By this way, the number of parallel processors available is increased by using several subsequences from each full period cycle. This improvement would allow the same number of parallel processors to become spawnable with a smaller modulus and with a smaller period length. Thus it would also speed up the cost of RN generation [10].

As a result, in this thesis, not only LCG is implemented with Sophie-Germain prime, but also, it is implemented through a parallelization method that combines parameterized iteration with sequence splitting or leapfrog methods. For which situations, this parallelization method is suitable is determined.

1.3 Organization of Thesis

The organization of the thesis is as follows. In **Chapter 2**, the theory of RN generation is covered by explaining the details of two well known generators. In **Chapter 3**, methods for parallelization of RNGs are discussed. In addition, the mapping scheme of processors on the binary tree and the problem of falling off the tree is given. In **Chapter 4**, the proposed solution to the falling off the tree problem is explained in great detail together with comparison based anaylsis, several tree charts, and graphics. In **Chapter 5**, the implementation of parameterized LCG on PVM system is discussed. Both architectural and algorithmic details are given. The emphasis is given to the enhancements on the LCG with prime moduli. To make LCG more efficient, proposed approaches are discussed. Advantages and disadvantages of using Sophie-Germain prime over Mersenne prime are explained. The resulting sequences that are created by the hybrid parallelization method are shown. Case analysis is done in order to determine the best approach for each case. Finally, in **Chapter 6**, all the comments are made, open questions and future are stated.

CHAPTER 2

SERIAL RANDOM NUMBER GENERATION

SRNGs (Serial Random Number Generators) are used for generating an array of numbers that have a random distribution. Generation is done by a function called generator, which is defined as, when applied to a number, yields the next number in the sequence. The SRNGs used in practice do not actually generate numbers that are truly random. Only the resulting sequence looks statistically independent and uniformly distributed.

SRNGs require the user to specify an initial value, or seed. Initializing the generator with the same seed gives the same sequence of RNs. If different sequences are needed, different seeds must be used.

Many widely used SRNGs have been shown to have quite poor randomness properties that lead to incorrect results in certain applications. It is better to use an SRNG that has been thoroughly tested. For applications in which RNs are only used occasionally, the quality of the generator doesl probably not matter, however in applications which use a lot of RNs, such as MC simulations, the quality of the generator is important and poor generators can lead to incorrect results [3].

2.1 Requirements for Serial Random Number Generators

An SRNG should produce a random number sequence that has the following properties,

- uniform distribution
- uncorrelation,
- never repeating itself,
- satisfying statistical test for randomness,
- having long period
- being reproducible,

- being fast
- being portable,
- changeable by adjusting seed values,
- easy to split into many independent subsequences,
- requiring low memory resources.

These properties are for ideal case. In practice it is impossible to satisfy all these requirements exactly. For practical purposes, it is required that the period of the sequence be much larger than the number of RNs needed for the application, and that the correlations be small enough that they do not noticeably affect the outcome of a computation [3].

In many applications that use RNs require that the SRNG be of highest quality. This requirement is in contradiction with the desire for being fast. A fast generator requires a minimal number of very simple operations, and it is this simplicity that often leads to problems with the quality of such generators. It is logical to sacrifice a little speed for much better randomness properties. While using an SRNG, it is usually better to be slow than sorry [3].

2.2 Linear Congruential Generators

The most commonly used RNG is the LCG. It is based on the iteration in Formula 2.1.

$$X_n = (a X_{n-1} + b) \bmod m \qquad (2.1)$$

In Formula 2.1, m is the modulus, a is the multiplier, and b is the additive constant that may be set to 0. The next number is generated using the random integer X_{n-1}, the integer constants a, b, and the integer modulus m. To get started, the algorithm requires an initial seed X_0. The entire sequence is referred as LCG(a, b, m, X_0). The appearance of randomness is provided by performing modulo arithmetic or remaindering. Note that the next result, X_n, depends on only the previous integer, X_{n-1}. This is a characteristic of LCGs which minimizes storage requirements, but at the same time, imposes restrictions on the period. (a, b, m) must be chosen carefully for a long period, good uniformity and randomness properties. The size of the modulus constrains the period, and is usually a prime or a power of 2 [3]. The period length for an LCG can be defined by the Theorem 2.1. The proof of the Theorem 2.1 can be found in [13].

Theorem 2.1 The LCG produces a sequence of period length m if and only if (b,m) = 1, a ≡ 1 (mod p) for all primes p dividing m, and a ≡ 1 (mod 4) if 4 | m [4].

For different choices of modulus and additive constant, generators behaves differently. When m is a power of two, the full period of $m = 2^M$ is obtained if and only if a ≡ 1 (mod 4), and b is odd (often chosen as 1). By using a power of two modulus, the process of performing modulus operation becomes very efficient but it causes correlation on least significant bits. All that a different choice of the initial seed does is shift the starting point in the sequence already determined by a, b, and m.

When additive constant b is zero, generator is termed as multiplicative congruential generator. The maximum period is 2^{M-2} (one quarter of the modulus), and is obtained if and only if a ≡ 3 (mod 8) or a ≡ 5 (mod 8) and the initial seed is odd. Low order bits are not random [4].

When the modulus m is prime, the maximum period length is m-1, and it is obtained when a is primitive modulo m. To show that a is primitive modulo m, it is sufficient to show that $a^{\Phi(m)/q} = 1(\text{mod } m)$ for all prime divisors q of m where Φ(m) is the Euler Phi function which gives the number of relatively prime integers less than m. In the case of m being prime, Φ(m) has the value m-1. Even when b is non-zero, the maximum period of this generator is still one less than the modulus. Thus, for LCG with a prime modulus, using a non-zero b does not increase the modulus [4].

When debugging, it is important to implement the algorithm to reproduce the same stream of RNs on successive runs. If the run is a debug run, the seed should be set to constant initial value, such as a large prime number. Otherwise, the initial seed should be set to a random odd value [3].

The costliest task when generating RNs is the modular reduction since it requires an integer division. When modulus is power of two, integer division and remaindering can be accomplished much more efficiently. With a divisor of 2^M, after the multiplication of aX^n, the next seed is obtained simply by performing a logical AND of X^n with a mask of (M-1) ones, right justified. On the other hand, when modulus is prime, standard modular reduction is used. To reduce this cost, Mersenne prime of the form $p = 2^k - 1$ where k is prime can be used. It is shown that such p's lead to fast modular reduction methods which use only a few

integer additions and subtractions. This technique is quite useful in practice, since it makes possible to implement long integer modular arithmetic without using multiple precision operations [14].

LCGs are commonly used since they are easy to implement, fast and adequate for most applications. Many commonly used LCGs use a modulus m that is a power of 2 since it is fast and convenient to implement on a computer. However, this approach produces highly correlated low order bits and long-range correlations for intervals that are a power of 2. To avoid these problems, it is best to use a modulus that is prime rather than a power of 2 [3]. In [15], it is found that LCG(7^5, 0, $2^{31} - 1$, X_0) has good results in spectral tests.

2.3 Lagged Fibonacci Generators

The name of the generator comes from the Fibonacci sequence. LFGs generate RNs from the iterative scheme in Formula 2.2.

$$X_n = X_{n-l} \Theta X_{n-k} \,(\text{mod } m) \qquad (2.2)$$

As seen in Formula 2.2, l and k are the lags, satisfying the conditions $l > k > 0$ and Θ is any binary arithmetic operation (addition, multiplication or XOR). The current X is determined by the values of X from l and k places ago. For an LFG, l initial values $X_0, X_1,.....,X_{l-1}$ are needed. This method requires storing the l previous values in the sequence in an array called a lag table. In addition, for most applications of interest m is a power of two, that is $m = 2^M$ [3].

As in LCGs, the parameters l, k and m must be carefully chosen to provide good randomness properties and the largest period. An advantage of this generator is that the period can be made arbitrarily large by just increasing the lag l. This also improves randomness properties since smaller lags mean higher correlations between the numbers in the sequence. One problem with LFG is that l words of memory must be kept current, where as LCG requires only that the last value of X is saved [4]. With proper choice of l, k and the first l values of X, the period, P, of this generator is equal to $(2^l - 1) \times 2^{(M-1)}$. Proper choice of l, and k here means that the trinomial $x^l + x^k + 1$ is primitive over the integers mod 2. The only condition on the first l values is that at least one of them must be odd [16].

The value of the modulus, m, does not by itself limit the period of the generator as it does in the case of an LCG. Note also that LFG is computationally simple. An integer add, a logical AND (to accomplish the mod 2^M operation), and the decrementing of two array pointers are the only operations required to produce a new RN. The major drawback in the case of this type of generator is the fact that l words of memory must be kept current. An LCG requires only one, the last value of X generated [4].

Conceptually, a Fibonacci generator acts the same as a linear shift register, and if M = 1 that m = 2^1, then the generator is a binary linear shift register. Since the state transformation of the shift register content is a linear operation, a matrix equation describing it can be given as shown in Figure 2.1.

The action of the shift register can be described by the Formula 2.3. In Formula 2.3, x_n is the entire vector after n time steps. When the vector x_0 has been given some initial set of values, for different n values, Formula 2.3 can be rewritten in terms of the initial vector x_0 as can be seen in Table 2.1. In the special case where n is taken as period p, $\mathbf{A}^p$ becomes equal to **I**, the identity matrix.

$$x_n = \mathbf{A}x_{n-1} \pmod{2^M} \tag{2.3}$$

$$\mathbf{A} = \begin{pmatrix}
0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 1 \\
1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\
0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1 & 0
\end{pmatrix}$$

Figure 2.1 Matrix equation for LFG

Table 2.1 Iteration formula

Iteration	Formula
1	$x_1 = \mathbf{A}x_0$
2	$x_2 = \mathbf{A}x_1 = \mathbf{A}^2x_0$
3	$x_3 = \mathbf{A}x_2 = \mathbf{A}^2x_1 = \mathbf{A}^3x_0$
n	$x_n = \mathbf{A}^nx_0$
p	$x_P = \mathbf{A}^Px_0$

The randomness properties of LFGs are best when multiplication is used, with addition being next best, XOR being by far the worst. LFGs using addition are the most popular because they are very simple and very fast. Each RN can be generated with a single addition and a modulus operation. Great care must be taken when choosing the lags for additive LFGs. Usually, much too small lags to give adequate randomness properties are chosen in many applications. Increasing the lag improves the randomness properties of the generator. A lag greater than 1000 is recommended for an additive LFG. The randomness properties can be improved by using multiple lags by combining three or more previous elements of the sequence, rather than two [3].

The choice of the lag may affect the speed of the generator, depending on the computer used. If a vector processor is used, a larger lag may improve performance, since the vector lengths are larger. If a scalar processor with limited cache is used, having a large lag may reduce the performance [3].

Multiplicative LFGs have seen little use. Although, slower than additive LFGs, they are as fast as 32-bit LCGs. They can be used with lags smaller than additive LFGs. One of the problems of multiplicative LFGs is handling the possible overflow of multiplication [3].

The algorithmic complexity of SRNGs can be defined as O(n) where n is the amount of RNs required. When considering SRNGs, what are more important are the properties of the generator that are inherited from the underlying recurrence relation like having long period or good randomness properties. Computational complexity of the recurrence relation is not of great concern. Though the complexity of SRNGs is fairly good, the problem arises when n

is in the range of millions, which is the case occurring most of the time. In problems where RNs are heavily used, execution of the SRNG slows the execution of problem considerably. In order to fasten random number generation, SRNGs are parallelized by several different methods forming a new class of RNGs, PRNGs which will be discussed in the next chapter.

CHAPTER 3

PARALLEL RANDOM NUMBER GENERATION

The goal of parallel RN generation is to design an RNG that produces random sequences of integers on each processor in a parallel computing environment [17]. Driving force to parallelize SRNGs comes from the necessity to get higher speed in MC applications. In order to get higher speed, these applications make extensive use of the parallel computers, since these calculations are particularly well suited to such architectures and often require very long run [7].

There are several methods for parallelization of SRNGs. These methods all assume a good source of sequential RNs which is transformed in some manner to a sequence of normally distributed RNs. There are several researches going on in order to improve parallelization methods of different SRNGs according to the requirements of an ideal PRNG. As a result of these researches, a software package named as SPRNG (Scalable Library for Pseudorandom Number Generation) based on MPI (Message Passing Interface) was created. Details of this package can be found in [20].

3.1 Requirements for Parallel Random Number Generators

In addition to the requirements for an ideal SRNG, a PRNG should possess the following additional properties:

- Generator should work for any number of processors
- Individual sequences on each processor should satisfy the requirements of a good SRNG.
- Sequences on different processors should be uncorrelated.
- Same sequence of RNs should be produced for different numbers of processors, and for the special case of a single processor [17].

- Speed of generation of the numbers on each processor and the amount of memory required per processor should be independent of the number of processors.
- There should be no data movement between processors. Thus, after the generator is initialized, each processor should generate its sequence independently of the other processors.

As with the ideal sequential generator, in practice it is not feasible to meet all these requirements. Among the above mentioned requirements of PRNGs, the most important one is the requirement that there should be no inter-processor correlation. This issue did not arise in the case of SRNGs [3].

3.2 Types of Parallel Random Number Generators

There are three general approaches to the generation of RNs on parallel computers; centralized, replicated, and distributed. In the centralized approach, a sequential generator is encapsulated in a task from which other tasks request RNs. This avoids the problem of generating multiple independent random sequences, but is unlikely to provide good performance. Furthermore, it makes reproducibility hard to achieve. The response to a request depends on when it arrives at the generator, and hence the result computed by a program can vary from one run to the next [18].

In the replicated approach, multiple instances of the same generator are created. Each generator uses either the same seed or a unique seed, derived, for example, from a task identifier. Clearly, sequences generated in this fashion are not guaranteed to be independent and, indeed, can suffer from serious correlation problems. However, the approach has the advantages of efficiency, and ease of implementation and should be used when appropriate [18].

In the distributed approach, responsibility for generating a single sequence is partitioned among many generators, which can then be parceled out to different tasks. The generators are all derived from a single generator; hence, the analysis of the statistical properties of the distributed generator is simplified. Most commonly used methods for parallelizing SRNGs are based on distributed approach [18].

3.3 Techniques for Parallelization

There are several methods for creating distributed PRNGs. These methods can be grouped under two basic techniques, which are splitting and parameterization. The underlying idea behind the splitting technique to parallelize a sequential generator is taking the elements of the sequence of RNs it generates and distribute them among the processors in some way [8]. On the other hand, parameterizing technique identifies a parameter in the underlying recursion of an SRNG that can be varied. Each valid value of this parameter leads to a recursion that produces a unique, full-period stream of RNs [19].

Finding a good PRNG is a very difficult problem. One of the reasons is that, any small correlations that exist in the sequential generator may be amplified by the method used to distribute the sequence among the processors, producing stronger correlations in the subsequences on each processor. Inter-processor correlations may also be introduced. Also, the method used to initialize a PRNG is at least as important as the algorithm used for generating the RNs, since any correlation between the seeds on different processors could produce strong inter-processor correlations.

3.3.1 Splitting Techniques

Splitting techniques based on the following concept. In order to parallelize a sequential generator, take the elements of the sequence of RNs it generates and distribute them among the processors in some way.

There are several methods to do this, which differs slightly but have the same basic concept, like sequence splitting, leapfrog, independent sequences, and random tree method. A top-down approach can be taken to choose a splitting scheme and an SRNG. There are five properties that make an SRNG suitable for splitting. These are,

- existence of a fast-leap-ahead algorithm,
- period long enough to be split,
- serial pseudo randomness,
- substream independence,
- fast serial implementation [9].

By considering these five important factors, two different splitting techniques are discussed.

3.3.1.1 Sequence Splitting

Sequence splitting method for parallelizing RNGs is to split a serial RN sequence into non-overlapping contiguous sections, each generated by different processors. If there are N processors, and the period of the serial sequence is P, then the first processor gets the first P/N RNs, the second processor gets the second P/N RNs, etc. This method requires a fast way to advance the serial sequence P/N steps. It turns out that LCG is a good candidate for this, but also, it is possible to use additive LFGs since jumping ahead is done only once in the initialization of the generator [3, 8].

A possible problem with this method is that although the sequences on each processor are disjoint, this does not necessarily mean that they are uncorrelated. In fact, it is known that LCG with power of two modulus, causes long-range correlations, which could become short-range interstream or inter-processor correlations in parallel generators [3]. One danger of this method is that if the user happens to consume more RNs than expected, then the sequences could overlap. Another disadvantage of this kind of generator is that it does not produce the same sequence for different number of processors [3].

3.3.1.2 LeapFrog Method

In this approach, the sequence of a serial generator is partitioned in turn among multiple processors like a deck of cards dealt to card players. If there are N processors, each processor leapfrogs by N in the sequence. For example, processor i gets X_i , X_{i+N} , X_{i+2N} , etc. To produce the same sequence of RNs for different number of processors, this method can be used. In order to use this method, jumping ahead in the sequence should be done easily. This can be done quite easily with LCGs but not practical for LFGs [3, 8].

This method has serious problems that long-range correlations in the original sequence can become short-range inter-stream correlations in the parallel generator. Since, it is known that LCGs using a power of two modulus have correlations between elements in the sequence that are a power of two apart. Moreover, for many parallel computers, the physical number of processors is a power of two. So, this method becomes useless. It may be adequate for some applications. If it is to be used, number of processors must be fixed and the modulus must be prime [3].

In some circumstances, it can be known that a program requires a fixed number of generators. In this case, leapfrog method can be used to generate sequences, which are guaranteed not to overlap for a certain period [3]. Let N be the number of sequences required. Two generators, L and R, are used and their corresponding a values, a_L and a_R, are defined as a and a^N respectively as shown in Formula 3.1 and Formula 3.2.

$$L_{k+1} = a\ L_k \bmod m \tag{3.1}$$

$$R_{k+1} = a^N\ R_k \bmod m \tag{3.2}$$

N different right generators ($R^0 \ldots R^{N-1}$) are created by taking the first N elements of L as their starting values. The name leapfrog method refers to the fact that the i^{th} sequence R^i consists of L^i and every N^{th} subsequent element of the sequence generated by L. As this method partitions the elements of L, each subsequence has a period of at least P/N, where P is the period of L. In addition, the N subsequences are disjoint for their first P/N elements. In Figure 3.1, the leapfrog method with three processors can be seen. Each of the three right generators selects a disjoint subsequence of the sequence constructed by the left generator's sequence [3].

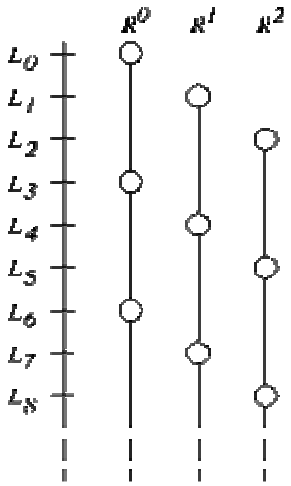


Figure 3.1 Leapfrog method with three processors

3.3.2 Parameterizing Techniques

The parameterization technique is one of the latest techniques for generating parallel RNs. The exact meaning of parameterization depends on the type of the SRNG. This method identifies a parameter in the underlying recursion of an SRNG that can be varied. Each valid value of this parameter leads to a recursion that produces a unique, full-period stream of RNs. Each processor is given the same SRNG but with a different set of parameter values. Hence, each processor executes the same general algorithm, and the same piece of code, only the parameters passed in initialization is varied from processor to processor [8,9]. There are two different methods for parameterization, which are cycle parameterization and parameterized iteration [7]. The emphasis will be on the parameterized iteration technique.

3.3.2.1 Parameterized Iteration

Parameterized iteration, lies on the fact that the iteration function can be parameterized. Here sequence i gets iteration function T_i. This is the case in the parameterization of LCGs [7]. The most important parameter of an LCG is the modulus m. Its size constraints the period, and for implementation reasons it is always chosen to be either prime or a power of two [20].

The parameterization method used is based on the type of modulus that has been chosen. When m is prime, a method based on use of the multiplier a as the parameter has been proposed. An alternative way to parallelize LCGs is to parameterize the additive constant when the modulus is a power of two [20]. In the next section, parameterization of power of two case is considered.

3.3.2.1.1 Parameterization of Power of Two Modulus

This method is first proposed as a way for providing PRNG for NYU Ultra-Computer. This technique has some interesting advantages over parameterization by multiplier. However, this technique, also has considerable disadvantages [20].

To parameterize power of two modulus, a set of additive constants that are pair-wise relatively prime are chosen. A logical choice can be to choose b_j as the jth prime. By this way, pair-wise relative primality is ensured. However, difficult problem of computing the jth prime arises when j gets higher [20].

Important advantage of power of two parameterization is that, spectral tests show it has good inter-stream independence. The disadvantage is, it is needed to compute inverse function of $\Pi(x)$, where $\Pi(x)$ is the number of primes less than x. If large number of streams are to be provided, fast algorithms can be used for the computation of $\Pi(x)$. Regardless of the efficiency of the algorithm, it is known to be a difficult computation with respect to computational complexity [20].

With a power of two modulus, cost of modular multiplication is far less than prime modulus case. On the other hand, a major shortcoming of LCGs modulo a power of two compared with prime modulus LCGs derives from the fact that the least significant bits of the power of two modulus have short period and are highly correlated. These aspects make the parameterization of power of two modulus less preferable [20].

3.3.2.1.2 Parameterization of Prime Modulus

To parameterize a prime modulus LCG, one can vary either the modulus or the multiplier or the additive constant. To vary the modulus is not acceptable because the modulus is chosen in a way to optimize the modular multiplication (the number theoretic properties of this modulus are used to optimize the modular multiplication). Thus, using a different modulus on different parallel processors lead to RN generation codes with very different execution times per RN [10].

There are two choices left, parameterizing the additive constant or the multiplier. If it is chosen to parameterize the multiplier when modulus is prime it can be shown that there is a set of initial conditions that makes the difference of a pair of prime modulus LCGs with same multiplier constant as show in [10]**.** This leads to RNs that are correlated. One more advantage of parameterizing the multiplier over the additive constant is that when parameterizing the multiplier, all the additive constants can be chosen as zero. This speeds the implementation, as only one modular multiplication and no modular addition is required per RN generation.

When modulus m is prime, a method based on use of multiplier a as the parameter has been proposed. To parameterize a, when m is prime, first the family of permissible a's must be determined (The family of a's that makes the iteration formula to have the maximum period m-1). When m is prime, to obtain the maximal period, a condition on a is to be

primitive modulo m. With a primitive modulo m, any choice of additive constant b gives period m-1 so b is chosen as 0 [20]. An integer a is primitive modulo m if it obeys the rules explained in Appendix A.

Given primitivity, if a and α are primitive elements modulo m, then $\alpha = a^i \pmod m$ for some i relatively prime to Φ(m) where Φ(m) is the number of relatively prime numbers to m that are less than m. When m is prime, Φ(m) = m-1. Thus, a single reference primitive element a and an explicit enumeration of the integers that are relatively prime to m-1 furnish an explicit parameterization for the jth primitive element, $a_j = a^{lj} \pmod m$ where lj is the jth integer relatively prime to m-1. If all of the primitive elements modulo m can be parameterized then parameterization of all the full period LCG sequences modulo m will be accomplished. To calculate the other primitive elements, a single reference primitive element must be known. The parameterization is reduced to an explicit computation of the jth number relatively prime to m-1 [21].

Table 3.1 Algorithm of LCG parameterization

LCG Algorithm		
1	Find primitive element modulo m Assign it as the multiplier	a
2	Prime factorization of m – 1	$\{p_1,\ldots,p_n\}$
3	Find the jth element relatively prime to m – 1	l_j
4	Compute the new multiplier for the jth processor	$a_j = a^{lj} \pmod m$
5	Start iteration according to the recursion formula	$X_n = (a X_{n-1} + b) \bmod m$

In summary, in order to parameterize the LCG via the multiplier, start with a reference value of a that is primitive element modulo m and choose the multiplier for the jth stream as $a_j = a^{lj} \pmod m$ where lj is the jth integer relatively prime to m-1. From here, it can be seen that there can be at most Φ(m) processors with disjoint sequences. An open question is whether the prime modulus must be chosen to minimize the cost of computing lj or to

minimize the cost of modular multiplication modulo m. The use of Mersenne prime minimizes the computational cost of modular multiplication whereas the use of Sophie Germain prime minimizes the cost of computing l_j [10]. The algorithm of the LCG parameterization is given in Table 3.1. In this thesis, parameterization of LCG with prime modulus is implemented. Modular multiplication and modular exponentiation operations are computed according to the *Russian Peasant Algorithm*. When Mersenne prime is used, modular reduction is done with *Mersenne Reduction Algorithm*. Otherwise, normal modular reduction is used. The details of these algorithms will be discussed in Chapter 5.

3.3.2.2 Cycle Parameterization

Cycle parameterization makes use of the fact that some PRNGs have more than one cycle. If the seeds are chosen carefully, then it can be assumed that each random sequence starts out in a different cycle so two sequences will not overlap. Thus the seeds are parameterized that is, sequence i gets a seed from cycle i, the sequence number being the parameter that determines the cycle [7]. In other words, consider a single SRNG that has full-period cycles that fall into different ECs (Equivalence Classes) depending on the initial seed. This generator is then seeded appropriately to ensure that each parallel processor uses a different EC. This is the case for the LFG, which is parameterized through its initial values [11].

LFG has relatively short period with respect to the size of its seed. However, the short period is more than made up for with the huge number of full-period cycles it contains. LFG is defined by the Formula 2.2 and denoted as LFG(l, k,m). This generator has maximum possible period as given in Formula 3.3, if and only if at least one of the seeds is odd (Proof can be found in [22]). The number of seeds that give the maximum possible period is given in Formula 3.4. Since each of these seeds is in a maximum possible period cycle, the number of cycles with maximum possible period is calculated with respect to Formula 3.5. Each of these full period cycles is called an EC [16].

The use of these ECs will be the key to parallelizing this generator. Thus, firstly, all ECs are enumerated in some way and then using this enumeration, a unique seed is created for each EC. To derive an explicit enumeration, one seed from the full period must be chosen to serve as the representative of the entire EC. This representative seed is called EC's canonical form. In other words, in order to derive an explicit enumeration, given any arbitrary seed, it

must be transformed into a seed with a canonical form Additionally, application of this procedure to seeds from different ECs produces different canonical form seeds. As a result, a single seed is produced which is the representative for its EC and is in canonical form [16].

$$P = (2^l - 1)\, 2^{(m-1)} \tag{3.3}$$

$$S = (2^l - 1)\, 2^{(m-1)l} \tag{3.4}$$

$$E = (2^l - 1)2^{(m-1)l} / (2^l - 1)2^{(m-1)} = 2^{(m-1)(l-1)} \tag{3.5}$$

b_{m-1}	b_{m-2}	-	-	b_1	b_0	
-	-	-	-	-	b_{l-1}	x_{l-1}
	-	-	-	-	b_{l-2}	x_{l-2}
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	b_{01}	x_1
0	0	0	0	0	b_{00}	x_0

Figure 3.2 Canonical form for LFG

After canonical form transformation comes the enumeration of different ECs. Since the number of ECs is as in Formula 3.5, a set of (l-1)(m-1) bits are needed to specify a unique EC. The canonical form has already specified the l least significant bits so it could be hoped that the canonical form gives the explicit enumeration in Figure 3.2. In fact this explicit enumeration also shows the construction of all the EC representatives. This enumeration leaves exactly (l-1)(m-1) bits to be specified in the canonical form and yields exactly E different possibilities as computed by Formula 3.5 [16]. To make a parallel implementation,

the key is to associate each independent parallel processor in the computation with a unique parallel processor identifier, K. This K is then used to select the Kth EC for this processor. This procedure works without difficulty provided that the parameters for the generator are chosen so that no K is required in the computation that exceeds E-1 [11].

The simplest way to parallelize this generator is to associate the Kth parallel processor with EC number K. Although, it is simple and has ease in computation, this naïve approach has some shortcomings. Starting all of the pseudorandom sequences from seeds that are in canonical form, leads to so-called *flat spots*. This is because the seeding values used for small EC numbers are numerically small themselves. Thus the initial segments will start and remain numerically small for an unacceptably long stretch. Worse than that, since all of the ECs start from their canonical forms in the naive implementation, all low-numbered ECs will suffer from flat spots that are lined up with respect to their cycles. So not only is a low-numbered EC initially distorted, all of those with similar EC number will be similarly distorted [11].

This natural numbering of the ECs, will cause flat spots of the lowest numbered ECs to appear at the very beginning of their cycles, giving the initial appearance of both non randomness and high cross-correlation. An alternative approach is to renumber the ECs so that the first ECs chosen will not have flat spots and neighboring ECs will have very different representatives. This new reordering can be accomplished by the aid of a high quality LCG. Although the details of such an implementation is beyond to scope of this thesis, the details can be found in [11].

3.4 Method for Instantiating Processors

When parallelization of RNGs is considered, the following concepts must be taken with great care. Firstly, the generator must be able to provide a reproducible stream of parallel RNs. This reproducibility must hold independent of the number of processors used in the computation. Besides, the generator must allow for the creation of unique RN streams on a parallel machine without any inter-processor communication. The generator must be portable between serial and parallel platforms. Furthermore, the generator must provide high quality RNs in a computationally inexpensive and scalable manner [10]. In order to achieve all these aspects, a technique based on mapping of parameterized RNGs onto a binary tree is proposed in [19] to permit a portable and reproducible MIMD implementation.

Each parallel processor also has the ability to create new child processors. The point of using a binary tree to map the parallel processors is that, one defines an entire subtree with each assignment and insures that processors elsewhere in the computation can not accidentally assign the same processor. In addition, the computation of child processors' identifiers and subtree follow can be done with only local information, without any inter-processor communication [11].

Binary tree can be defined as a data structure where it consists of a node called root together with two binary trees called the left subtree and the right subtree of the root. In a binary tree structure, there is a parent-child relationship where each parent has at most two direct children. If parent has an identifier K, then the two children of the parent have the identifiers, 2K and 2K+1 respectively. The structure of the binary tree with seven nodes is given in Figure 3.3.

To produce a child processor with a process identifier K that is guaranteed to be distinct from others created elsewhere in the computation, using a binary tree mapping is essential. In the binary tree structure as shown in Figure 3.3, nodes are representing real processors. Each node must contain in its simplest case, the task id of the related processor and a node number which shows from which node in the binary tree to start if spawning child processors is needed. By this way, the assignment of process identifiers to newly created child processors becomes a local computation based only on the parent's processor identifier. For instance, when the processor for node K is required to create n children, it does so by assigning the n nodes closest and below it on the binary tree. This assures a local computation. In particular, if the processor assigned to node K has two children, they receive nodes 2K and 2K+1 respectively.

There are two main operations related with parallel processors and binary tree structure. These operations are defined as *Initialization* and *Spawning*. How and according to which rules the structure of binary tree changes with respect to these operations are described in the following sections.

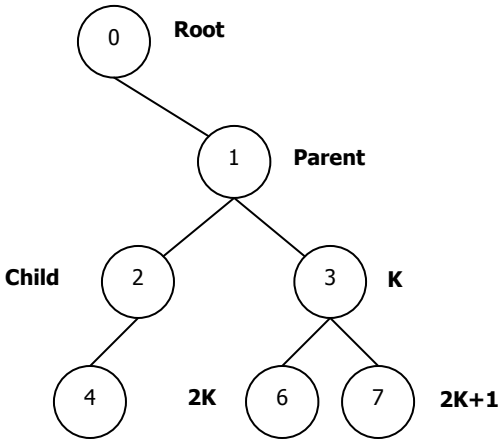


Figure 3.3 Binary tree structure

3.4.1 Initialization Algorithm

Initialization operation as its name implies is related with the initial organization of the parallel processors. Organization of processors on the binary tree structure, assignment of initial values, and the interaction between the parallel processors are the main parts of the initialization algorithm. In this section, the initialization algorithm is considered only with respect to its effects on the binary tree structure. The details of the overall initialization algorithm together with parallel calls to processors will be given in Chapter 4.

As described in the previous section, binary tree structure consists of nodes where each node represents a processor. When this is the case, it becomes essential that a node contains all the needed information for a processor to be able to spawn with only its local information. That is why; a node structure consists of in its simplest form from three attributes as can be seen from the Figure 3.4. First one is the task id of the related processor which is the process identifier of the processor. The second one is the node number. The third one is the node number of the first child processor to be created as a result of spawning operation. This attribute is defined as the *Spawn Pointer* and plays an important role during spawning operation.

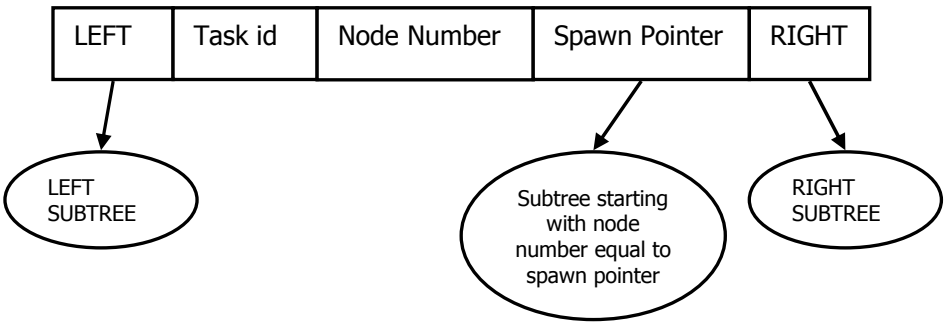


Figure 3.4 Node structure

Suppose a program needs N initial generators which will run independently from each other. One processor is created for each generator and these generators are placed at nodes n = 0, 1, 2, …. , n_{max} where n_{max} = N – 1 of the binary tree. While organizing the binary tree structure, for each node, according to the its node number, spawn pointer values are calculated with respect to the Formula 3.6 where SP represents spawn pointer [11].

$$SP_n = 2^j (2n + 1) \text{ for the smallest j such that } 2^j (2n + 1) > n_{max} \quad (3.6)$$

To illustrate the initialization operation, consider an example situation where a program needs initially three generators and where the maximum number of processors is restricted to 16. These generators are placed at nodes n = 0, 1 and 2 of the tree where n_{max} becomes 2. For each generator, spawn pointer values are calculated according to the Formula 3.6. The state of the binary tree after the initialization of three processors is shown in Figure 3.5. In Figure 3.5, spawn pointers are represented by arrows and the dashed circles are the nodes that are not created yet but pointed by the spawn pointers. For instance, node 1's spawn pointer points to 3. This means that when a spawn call is requested by node 1, this node will begin to create new child processors starting from node 3 and continue to produce within the subtree where node 3 is the root. This subtree is defined as the spawn pool of the node 1. In other words, *Spawn Pool* of a node can be defined as the subtree rooted by the spawn pointer of that node.

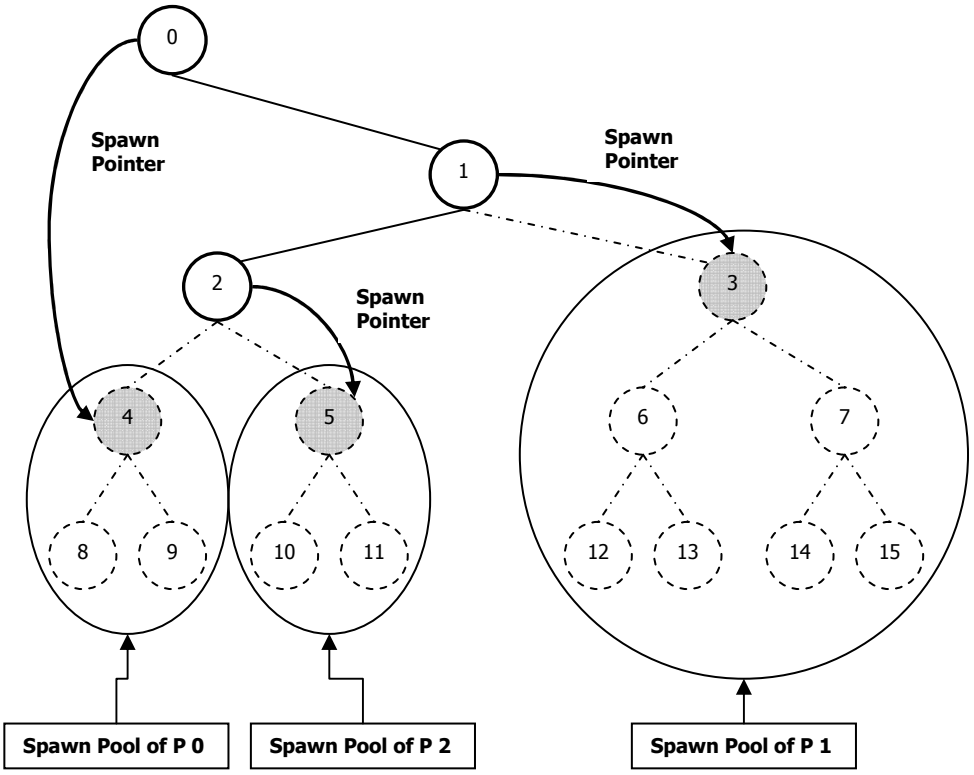


Figure 3.5 Initialization of binary tree with three processors

3.4.2 Spawning Algorithm

Spawning operation can be described as a processor being parent, wants to create new child processors. It is important that a parent processor could do this creation operation with only its local information, without a need for inter-processor communication. That is why; processors are mapped into a binary tree and spawn pointer and spawn pool concepts are introduced. Spawn pointers are providing a way to know where to start spawning in the binary tree whereas spawn pools are setting the boundaries between the parent processor and the other parts of the tree preventing those two processors accidentally assign same node numbers to different processors.

When a processor wants to spawn N new child processors, it gives the node number which is pointed by its spawn pointer to the first processor spawned. Then, continues to spawn within its spawn pool according to the rules of the binary tree structure. While spawning new child processors, there are two important aspects. First one is, it is required that these child processors must have been assigned spawn pointers, in case they will need to spawn in the future. Spawn pointers of child processors are calculated with the Formula 3.6 where n_{max} is changed to the maximum numbered node that is created during spawning. Second one is related with the spawn pointer of the parent processor. It must be adjusted according to the new organization of the binary tree with respect to the Formula 3.6. Other processors are not related with these spawn pointer adjustments since spawning is accomplished within the boundaries of the spawn pool of the parent processor. This adjustment of spawn pointers can be defined as dividing the spawn pool of parent processor between the newly created processors and itself.

To illustrate the spawning operation, consider an example situation where a program needs initially three generators and where the maximum number of processors is restricted to sixteen. Initial state of the binary tree together with spawn pools and spawn pointers can be seen in Figure 3.5. Assume that processor 1 needs to spawn one more processor. When this is the case, processor 1 creates the new child processor and gives this new processor its spawn pointer as its node number. In other words, processor 1 spawns processor 3 according to its spawn pointer. After giving the node number, with respect to the Formula 3.6, spawn pointer of processor 3 is calculated as seven and then the spawn pointer of processor 1 is updated according to Formula 3.6 to six, since $n_{max} = 3$. What is happened actually can be clearly seen when Figure 3.5 and Figure 3.6 are compared. In Figure 3.5, it is seen that, processor 1 has a spawn pool consisting of seven processors which means that processor 1 can spawn at most seven processors. When processor 1 spawns one processor, its spawn pool is divided into two between itself and the newly created child processor which can be seen in Figure 3.6 clearly.

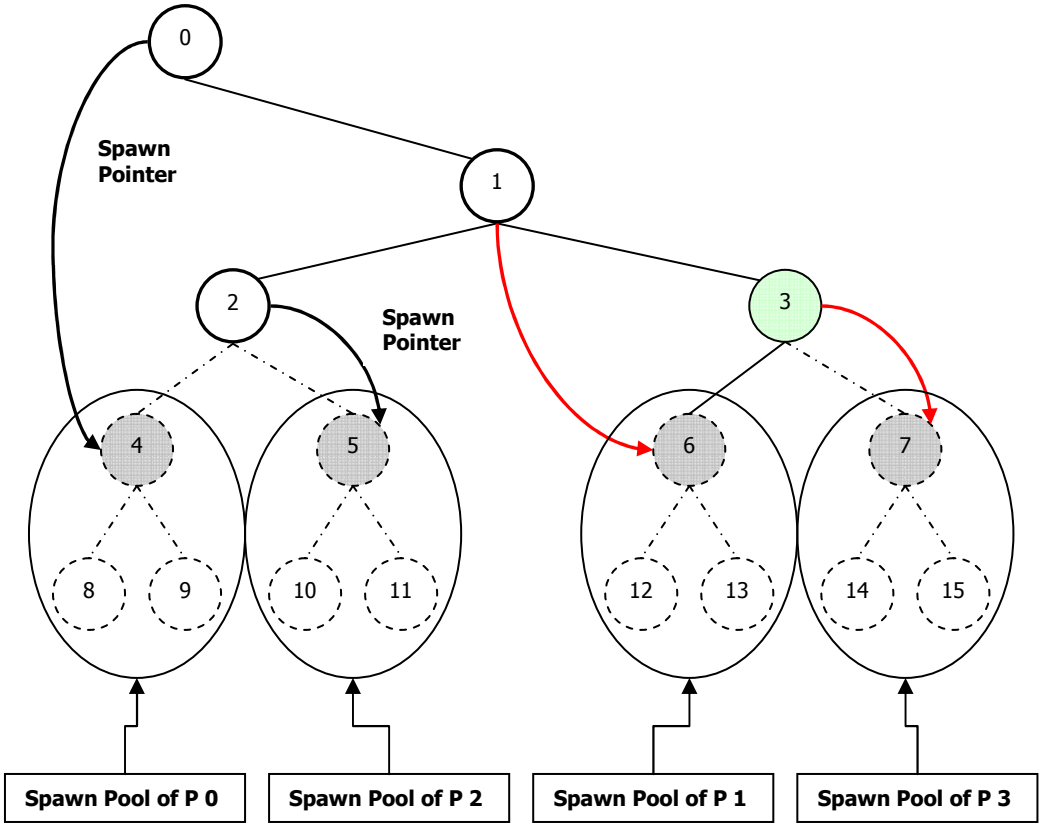


Figure 3.6 Processor 1 spawns 1 processor

Consider another example. In Figure 3.7, the initial state of the binary tree with three processors numbered from 0 to 2 is displayed together with spawn pointers and spawn pools. Here, the maximum number of processors is restricted to thirty two. Since three processors are already spawned, there are twenty nine processors left. These twenty nine processors are the overall spawn pool of the three processors and this spawn pool is divided among three processors according to their spawn pointers.

As can be seen from Figure 3.7, processor 1 has the spawn pointer three so the subtree starting from node 3 is processor 1's spawn pool, giving fifteen processors to processor 1 to spawn. In a similar manner, when processor 2 is considered, it is seen that, processor has the spawn pool as the subtree starting from node 5, having a total of seven processors to

spawn. Lastly, processor 0 has the spawn pool as the subtree rooted by node 4. Like processor 2, processor 0 has seven processors to spawn. In summary, the spawn pool of twenty nine processors is divided among three processors in a way that processor 1 gets fifteen, whereas processor 2 and 0 gets seven processors each.

Assume that processor 1 wants to spawn three processors. Processor 1 has a spawn pool as the subtree rooted by node 3. It can spawn fifteen processors at most. According to the spawn pointer, processor 1 assigns the first processor created, the node number of 3. The other two processors get the node numbers 6 and 7 respectively. During this operation, the spawn pool is also divided among processor 1 and newly created child processors. The result of the spawn pool division is clearly seen in Figure 3.8. Consider the case where processor 0 wants to spawn one processor. Processor 0 has a spawn pool rooted by node 4. It can spawn seven processors at most. According to the spawn pointer, processor 0 assigns the newly created processor the node number of 4. Also, the spawn pool of processor 0 is divided among itself and the newly created processor.

All these examples point to one fact: Spawning operation is accomplished within the boundaries of spawn pool. The spawn pool of the parent processor is divided among itself and the newly created child processors. Spawning operation relies on dividing spawn pool concept and updating of spawn pointers. It is these properties that enable spawning operation to be done independently and without needing inter-processor communication. Whereas, it is also these properties that put constraint on the number of child processors that a processor can spawn and constitute a situation called falling off the tree whose details will be given in the next section.

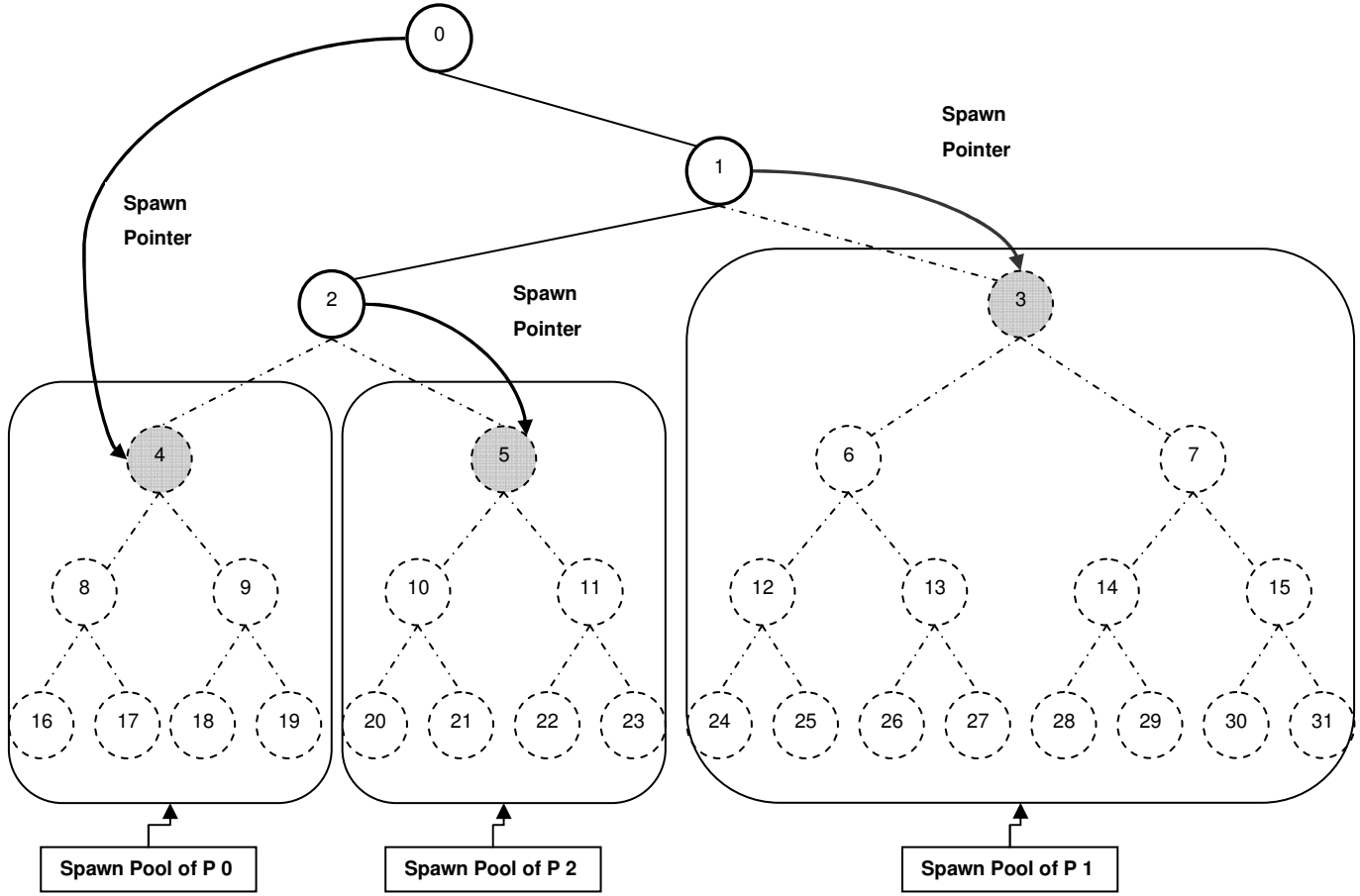


Figure 3.7 Initial state of binary tree: Spawn Pools and Spawn Pointers

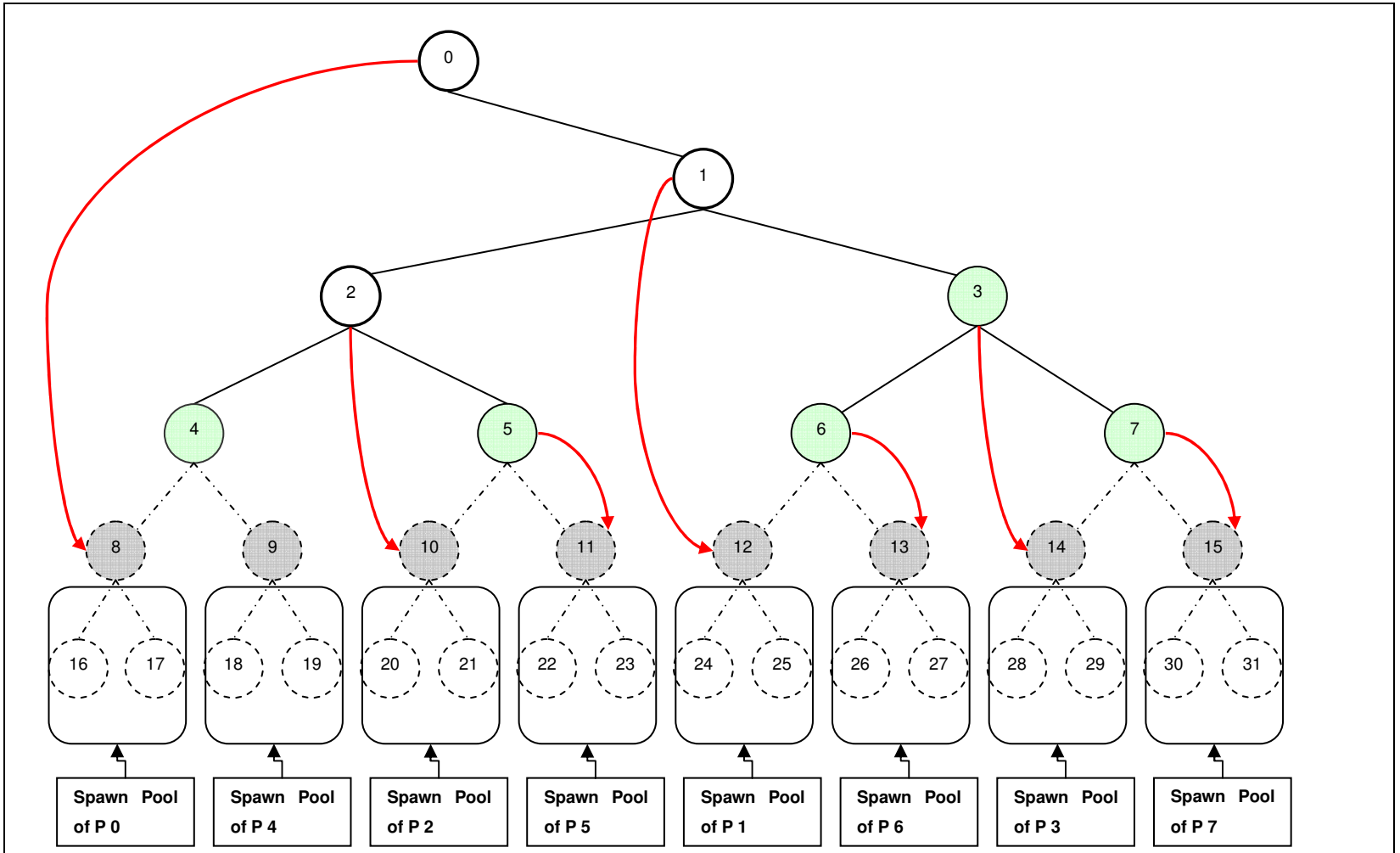


Figure 3.8 State of binary tree after three spawning operations

3.4.3 Problem of Falling off the Tree

Binary tree mapping works fine for a considerable number of processors, however, problems can arise from the fact that the binary tree structure puts an upper limit on the number of processors that a processor can spawn with its local information. This problem is named as the problem of *Falling off the tree*. It is defined as the condition of having made a spawn call requesting more processors than in the spawn pool of the processor given as input. This can be a serious problem when a processor needs to spawn several times.

The problem is clearly stated in Figure 3.10. Assume that there are initially two processors and the maximum number of processor is restricted to sixteen. Initially, processor 0 has seven processors in its spawn pool and wants to spawn one more processor. It spawns processor 2. Its spawn pool is divided into two and its spawn pointer is updated. Now, processor 0 has at most three processors to spawn. Again, processor 0 wants to spawn two processors. It spawns processors 4 and 8. When it comes to dividing the spawn pool, since there is only one processor left, not all of the processors are assigned to a spawn pool. As can be seen from Figure 3.9, only the processor 4 is assigned a spawn pool consisting of one processor. The other child processor 8 and the parent processor 0 assigned a null spawn pointer, meaning that they are unable to spawn any more. When such a situation is considered, it is seen that, upper limit for number of processors is sixteen, and so far only five processors are spawned leaving eleven processors. There are still processors left in the overall spawn pool but processor 0 is unable to spawn since there are not any processors left in its spawn pool.

Problem of falling off the tree can be a serious problem in applications where processors tend to spawn new processors continuously. In such cases, after a while, the RN sequences of independent processors will begin to correlate leading to incorrect computations. For instance, consider the situation where the path of each neutron is determined by a random sequence created by a processor. When there are correlations among individual sequences, the related neutrons will have the same path, and the quality of the computation becomes questionable.

As a solution to the falling off the tree problem, one proposal is changing the seed and pointing new processors to nodes elsewhere in the tree that may or may not be independent from those already been used. Another proposal is to enable the user to have access to all the independent processors. What is left is to pick a generator with as many processors as maximum number of processors needed. But in such a case, the user must do the bookkeeping and take great care not to choose a processor that has already been used [2]. In this thesis, a solution to the problem is suggested which will be described in Chapter 4.

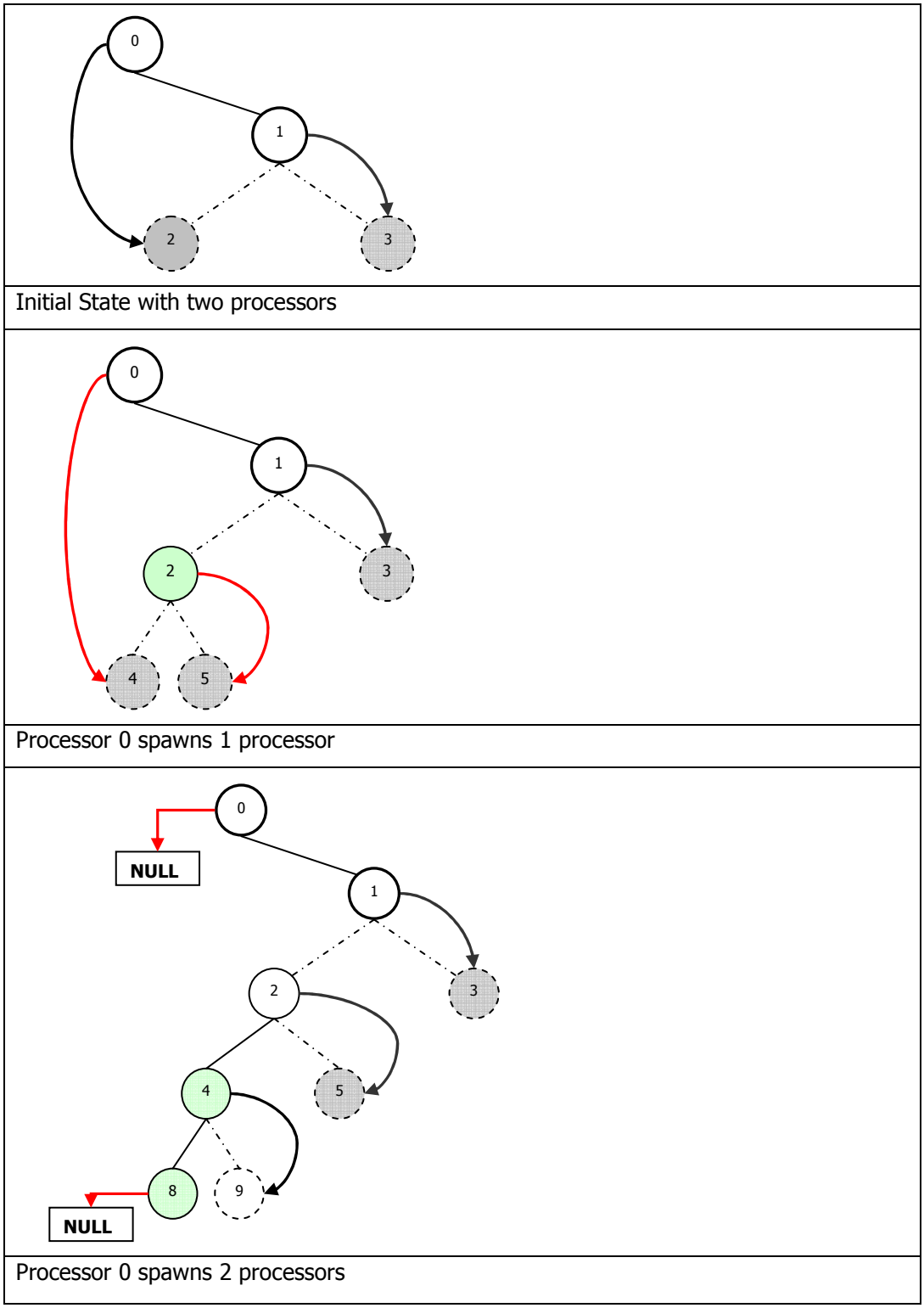


Figure 3.9 Problem of falling off the tree

CHAPTER 4

IMPROVEMENTS ON BINARY TREE MAPPING

Ideally, a PRNG should be able to produce individual sequences without inter-processor communication. To reduce inter-processor communication and to enable a processor to spawn new processors with only its local information, together with binary tree mapping, concepts of spawn pointer and spawn pool are exposed. But what if a processor wants to spawn more processors than in its spawn pool? The problem of falling off the tree appears. In such cases, if it is still important to spawn new processors without inter-processor communication, then there is no chance but to accept the correlated sequences on newly created processors. On the other hand, if minimized cost of inter-processor communication is acceptable, then there can be other solutions to the problem of falling off the tree, where it is guaranteed that uncorrelated sequences on individual processors are created. In fact, this is a serious trade-off between inter-processor communication and correlation that the users of RNGs must consider.

In this chapter, firstly, the implementation details of a parallel LCG on PVM system is given. Secondly, a solution to the problem of falling off the tree that requires inter-processor communication is proposed. This new solution consists of several search routes and two algorithms that minimizes the need for inter-processor communication. Lastly, detailed analysis of the proposed solution is madethrough graphics.

4.1 Implementation on PVM system

Although LFG has better quality consequences, in this thesis, LCG is chosen to be implemented. Since in LCG, memory handling and parameterization operations are carried out easily. In this section, the architectural and algorithmic details of LCG implementation are given. The implementation consists of five main algorithms as shown below:

- Algorithm for parameterization of LCG
- Initialization algorithm
- Spawning algorithm
- Algorithm for solving the falling of the tree problem
- RN generation algorithm

Although, the overall structure of implementation is considered, only the initialization, spawning and RN generation algorithms will be detailed in this section. The algorithm for parameterization of LCG will be considered in Chapter 5 and the algorithm for solving the falling off the tree will be fully clarified in Section 4.2.

4.1.1 Architecture

Parallel LCG is implemented on a cluster system using PVM. Cluster system consists of three separate computers of nearly equal computational resources as can be seen in Figure 4.1.

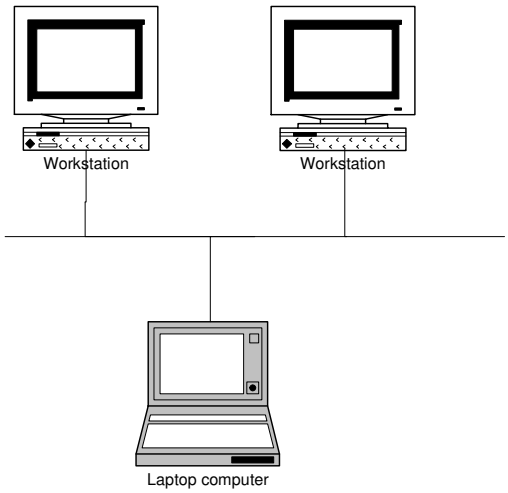


Figure 4.1 Architecture of implementation

4.1.2 Algorithmic Structure

Parallel LCG implementation consists of two separate programs written on PVM. These programs are named as MASTER and SLAVE respectively. MASTER program handles user input, creates initial processors, initializes binary tree structure and keeps binary tree up to date. Moreover, as a result of the proposed solution to the problem of the falling off the tree problem, MASTER program is responsible from directing spawn calls to available processors with respect to new spawn routes and algorithms. Whereas, SLAVE program represents individual processors and responsible from RN generation and spawning new child processors. Whole implementation relies on the communication between MASTER and SLAVE programs together with interaction with the binary tree structure. In this section, firstly, the node structure of the binary tree is represented. Secondly, the structure of MASTER and SLAVE programs are given. Lastly, the three basic operations, initialization, spawning and RN generation are explained with respect to MASTER SLAVE interaction.

4.1.2.1 Node Structure

Binary tree structure is the foundation of the initialization and spawning operations. It must be correctly initialized and updated with each new processor created. Now, it is reasonable to ask the question of what must be the structure of an individual node in practice? In binary tree structure, each node represents a real processor and as explained in Section 3.4.1, it must contain all the needed information for a processor to be able to spawn with only its local information. Besides, as a result of the new proposed solution to the falling off the tree problem, it becomes necessary to extend the structure further. The structure of a binary tree node can be seen in Figure 4.2. Each field in the node structure has a special purpose and will be explained briefly one by one.

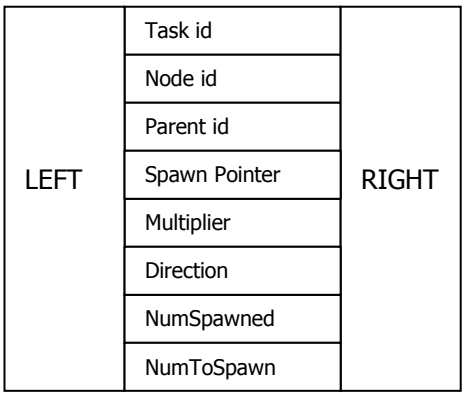


Figure 4.2 Node structure

Task id represents the task id of the processor given by the PVM system. It is needed for sending messages to real processors easily and makes a mapping between the node and the real processor. **Node id** is the node number needed for the binary tree mapping. **Parent id** is the node id of the processor that has spawned this node. **Spawn pointer**, represents the first node in the spawn pool of the node. Spawn pointer can be NULL, meaning that spawn pool of the node is empty. **Multiplier** is the local value computed by the processor and needed for RN generation formula. The details of the multiplier computation are given in Chapter 5. **Direction** represents the way that must be taken when traversing the spawn pool of the node. **NumSpawned** is the number of processors spawned by the node so far. **NumToSpawn** is the maximum number of processors that can be spawned at once. Parent id, Direction, NumSpawned, NumToSpawn are newly added fields for the proposed solution of the falling off the tree problem. The necessity of these fields will be clarified as it is progressed through the details of the proposed solution.

4.1.2.2 Structure of SLAVE Program

In the implementation, SLAVE program corresponds to an individual processor. In fact, the structure of the SLAVE program represents the life cycle of a processor. That is why; the flow of SLAVE program is explained in terms of processor's actions. The life cycle of a processor consists of three phases. In the first phase, initial data is taken from parent processor. If it is initialization operation, parent is the MASTER program. Otherwise, if it is spawning operation, parent is another processor. After taking the initial data, in the second phase, with this data, computations for LCG parameterization are performed. The algorithmic details of these computations are given in Chapter 5.

Briefly speaking, as a result of these computations, processor becomes ready to generate RN sequence that is guaranteed to be uncorrelated from other processors' sequences. In the third phase processor goes into an endless loop, waiting for action commands coming from the MASTER program. The flow of actions can be seen in Figure 4.3 and Figure 4.4 together with PVM calls. There are basically two important actions of a processor. First one is spawning child processors and second one is generating RNs. Algorithm for spawning is given in Section 4.1.2.5 and algorithm for generating RN is explained in Section 4.1.2.6.

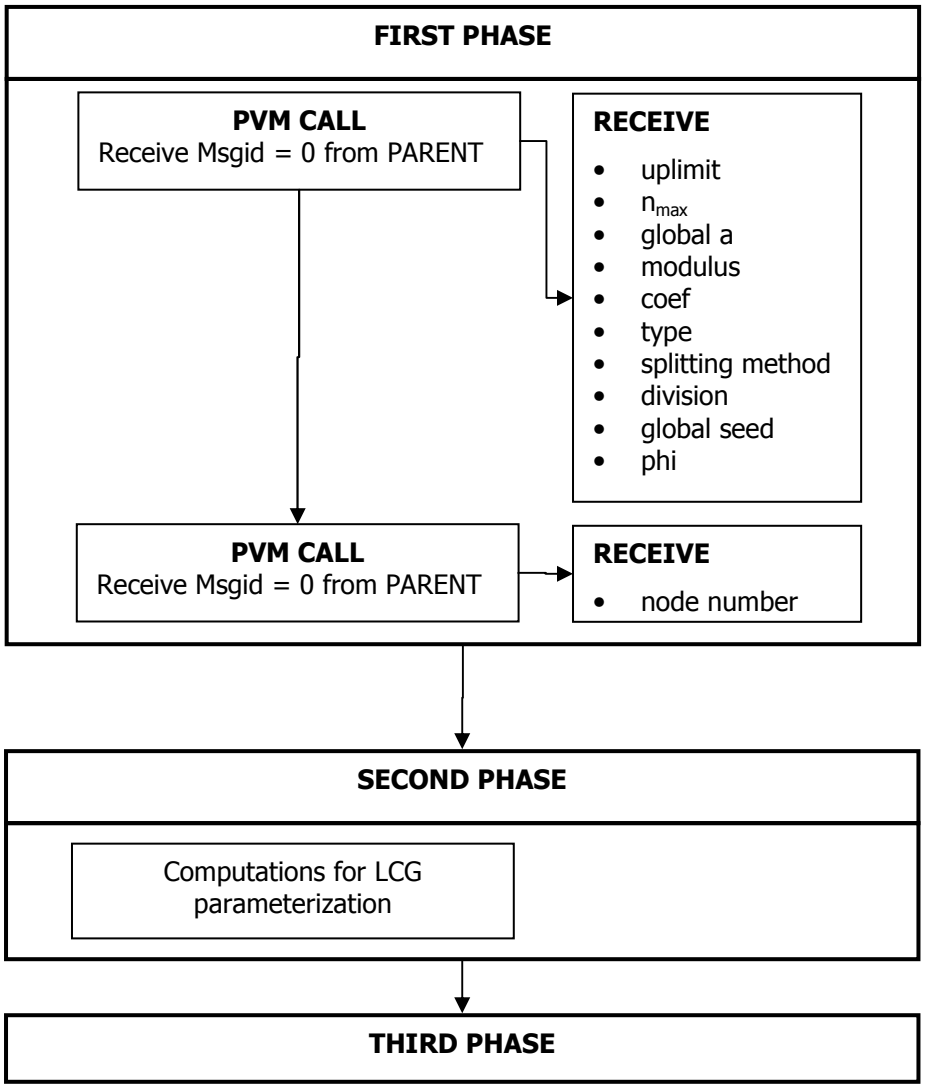


Figure 4.3 Algorithmic structure of slave program: Three Phases

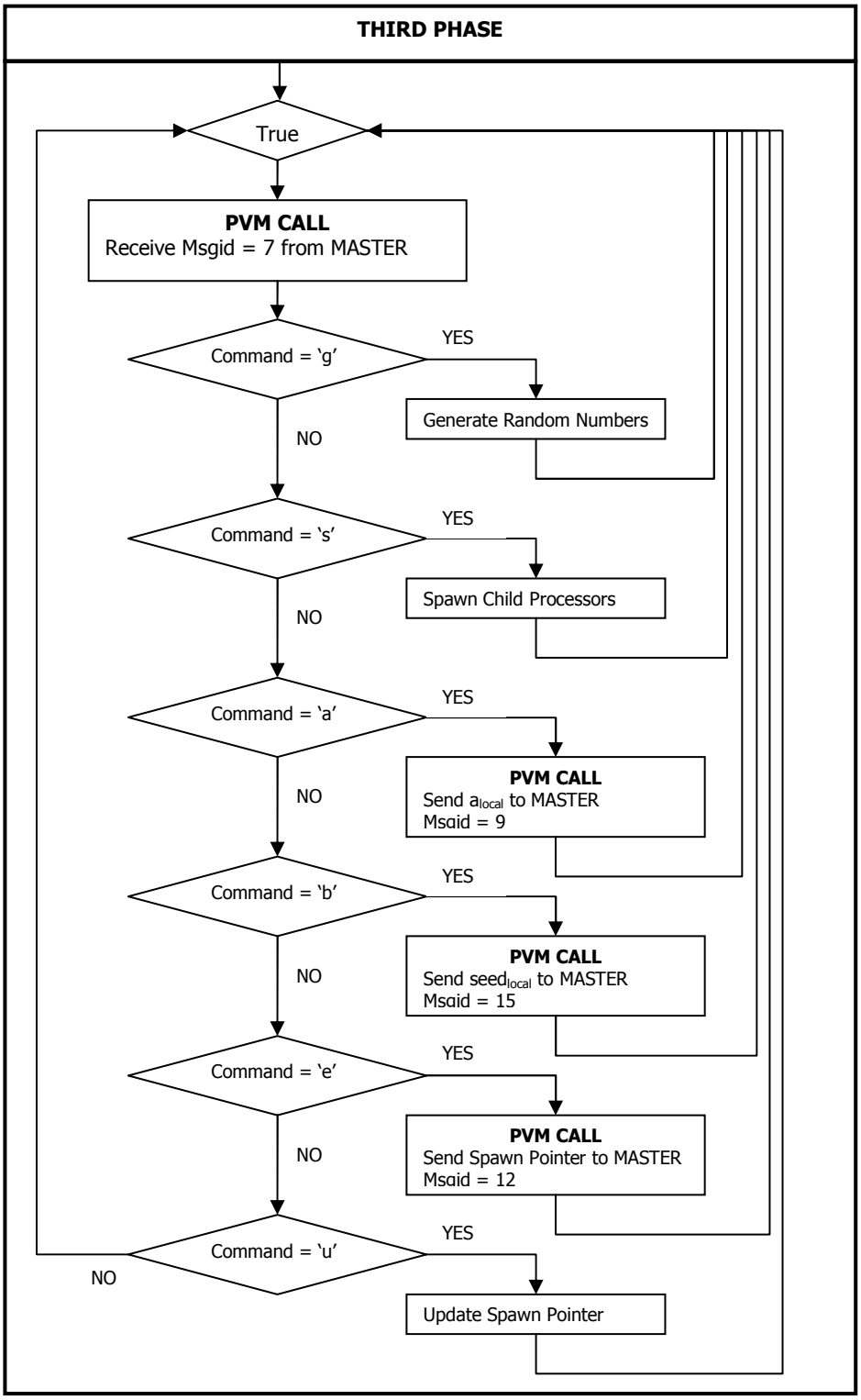


Figure 4.4 Algorithmic structure of the third phase

4.1.2.3 Structure of MASTER Program

MASTER program as its name implies, is responsible from the following operations:

- Coordinating user commands
- Creating and initializing binary tree structure
- Updating binary tree structure
- Displaying RN sequences
- Searching for a new parent for spawn call

MASTER program takes user commands and makes the relevant actions. Program starts by receiving *initialize* command together with related parameters like number of processors (nproc) from the user. By receiving *initialize* command, MASTER program executes the step called *Initialize PVM* in Figure 4.5. Briefly speaking, initialization operation takes the following actions:

- Creates real processors and broadcasts initial data,
- Creates binary tree structure,
- Creates nodes in the binary tree and fills the node structure appropriately.

After executing *initialize* command, now MASTER program becomes ready for the other operations like spawning and generating numbers. When *spawn* command is received together with the parent processor and number of child processors (nchild), MASTER program executes the steps that are represented as *Spawn PVM* and *Spawn Search PVM* in Figure 4.5.

Spawning operation can be summarized as follows:

- Sends parent processor the spawn call,
- Receives the number of processors spawned,
- Adjusts the binary tree structure according to the newly created processors,
- In cases where parent processor is out of spawn pool and there are more to spawn, searches the binary tree according to spawn routes and assigns an available processor as the new parent,
- Adjusts the binary tree structure according to the newly created processors with respect to the spawn algorithms.

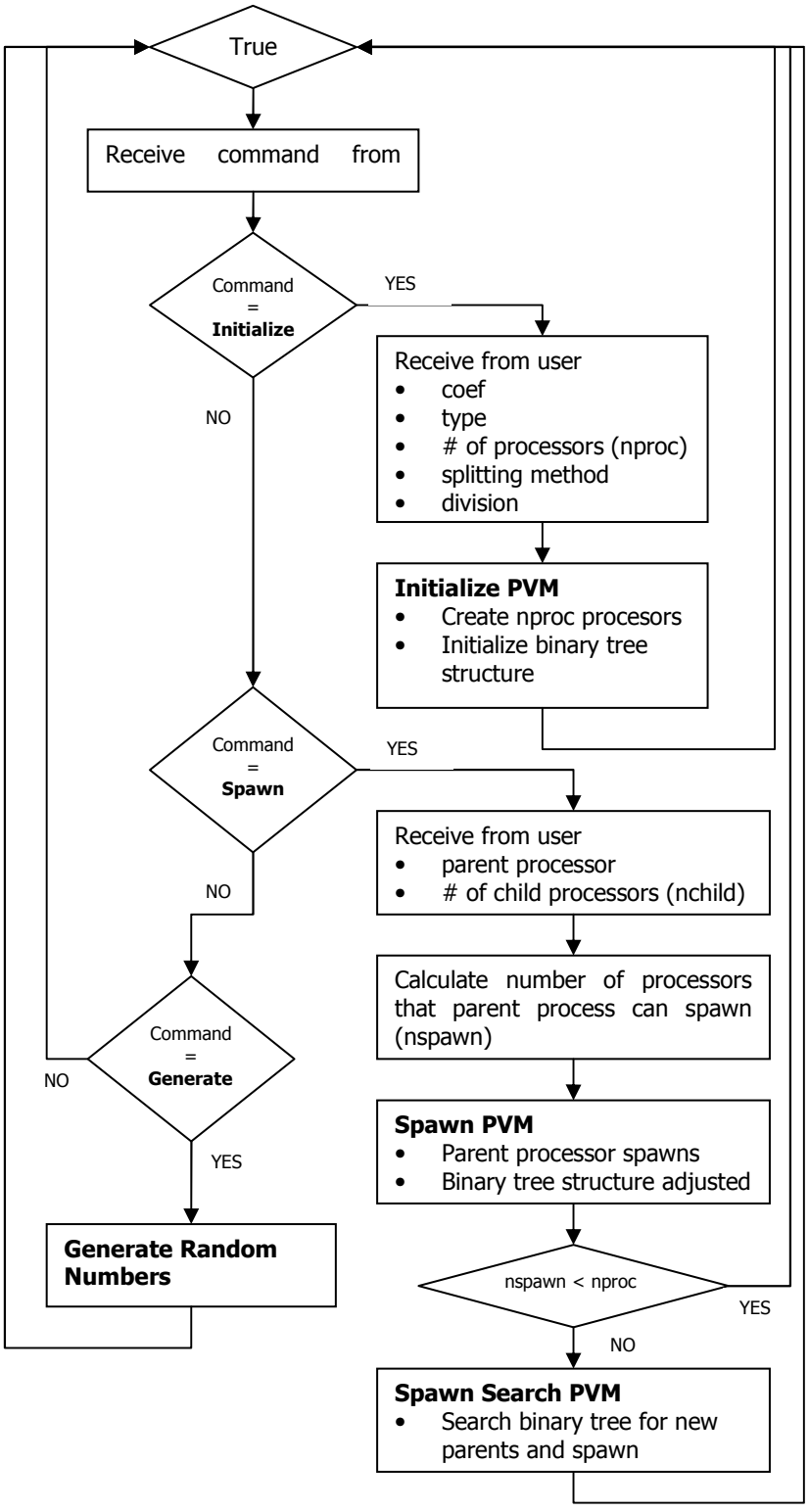


Figure 4.5 Algorithmic structure of master program

The fourth and fifth steps are as a result of the new proposed solution and are considered in detail in Sections 4.2 and 4.3. Apart from spawn and initialize commands, there is another command which is called *generate.* As a result of this command MASTER program executes the step *Generate Random Numbers* in Figure 4.5. In summary, RN generation operation is explained by the following steps:

- Sends each processor in the binary tree *generate* command,
- Receives from each processor in the binary tree, the RN sequences,
- Displays the RN sequences on screen and outputs to files.

In the following sections, details of initialization, spawning and RN generation operations are explained one by one.

4.1.2.4 Initialization Operation

Initialization operation is the implementation of the initialization algorithm that is explained in Section 3.4.1. In Figure 4.6, the algorithm of the initialization operation is displayed in terms of MASTER SLAVE interaction.

MASTER program spawns processors and broadcasts initial data to SLAVE program. Then, MASTER program sends each new processor their node number. Meanwhile, SLAVE program receives initial data and its node number and makes the related computation for LCG parameterization and begins to wait in an endless loop for the other action commands. MASTER program sends SLAVE program *send multiplier* command. SLAVE program receives the command, sends MASTER program its local multiplier value and continues to wait in the loop. Meanwhile, MASTER program initializes binary tree structure by creating nodes corresponding to new processors. When nodes are created in the binary tree, initialization operation ends.

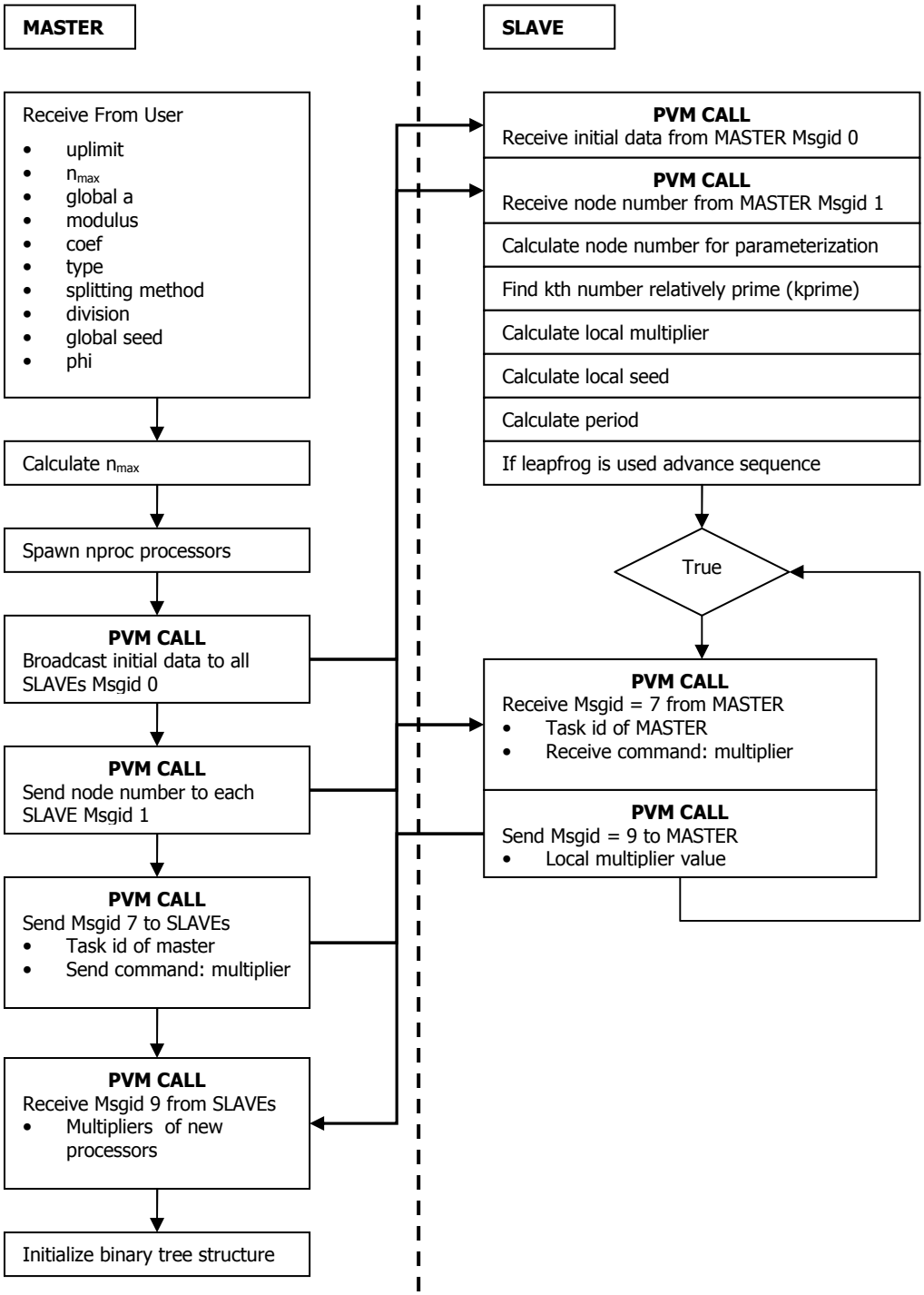


Figure 4.6 Algorithm for initialization operation

4.1.2.5 Spawning Operation

Spawning operation is the implementation of the spawning algorithm that is explained in Section 3.4.2. Spawning operation is represented by *Spawn PVM* in Figure 4.7. This operation mainly consists of two parts. First part is the creation of real processors and the second part is the adjustment of the binary tree structure according to these newly created processors. Second part is represented by *Spawn Binary Tree Structure* in Figure 4.7 and details of this algorithm together with a more detailed flow chart of spawning operation with PVM calls can be found in Appendix B. Furthermore, in Figure 4.8, algorithm of the spawning operation is displayed in terms of MASTER SLAVE interaction.

As can be seen from Figure 4.8, during spawning operation, there are interactions between at least three processors which are MASTER, PARENT and the newly created CHILD processors. PARENT and CHILD processors are represented by SLAVE program and can be defined as SLAVE PARENT and SLAVE CHILD. Spawning operation starts by MASTER receiving the *spawn* command from user together with parent node and the number of child processors (nchild). Firstly, MASTER processor finds the PARENT processor in the binary tree structure and checks whether this PARENT processor can spawn nchild processors. After making these computations, MASTER processor sends PARENT processor *spawn* command. PARENT processor spawns new child processors and sends back the number of spawned processors to MASTER processor. PARENT processor also broadcasts initial data to CHILD processors and sends node numbers to each CHILD processor. CHILD processor receives the initial data and node number from PARENT processor and makes the computations for LCG parameterization. Then, CHILD processor goes into an endless loop and waits for action commands. Meanwhile, PARENT processor sends the task ids of the CHILD processors to MASTER processor. MASTER processor sends each CHILD processor *send multiplier* command. Each CHILD processor sends its local multiplier value to MASTER program and continues to wait in the loop for other commands. MASTER processor, after receiving all the related information, starts adjusting the binary tree structure with respect to newly created processors. Spawning operation ends after the binary tree adjustment is finished.

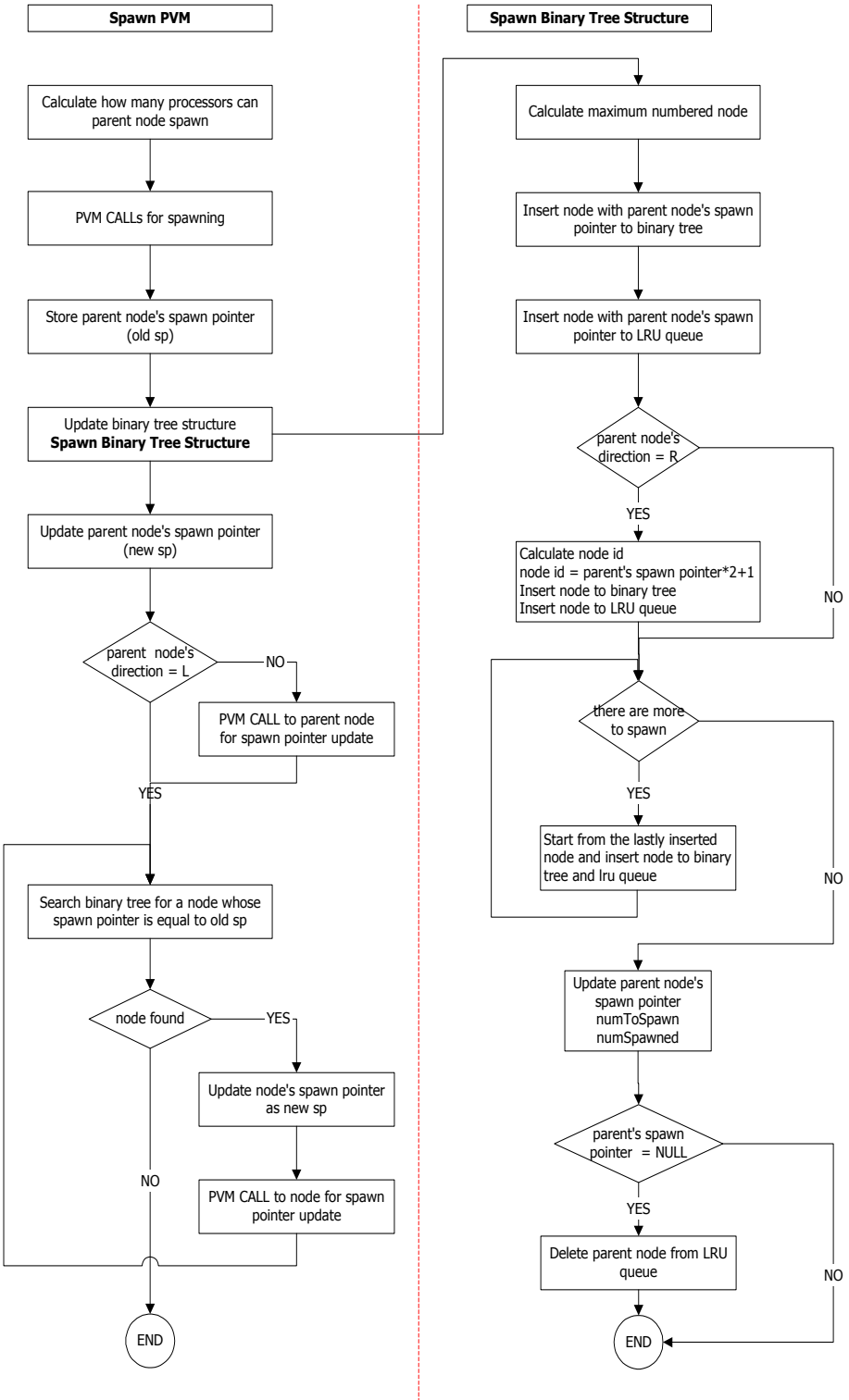


Figure 4.7 Flow chart of spawn operation

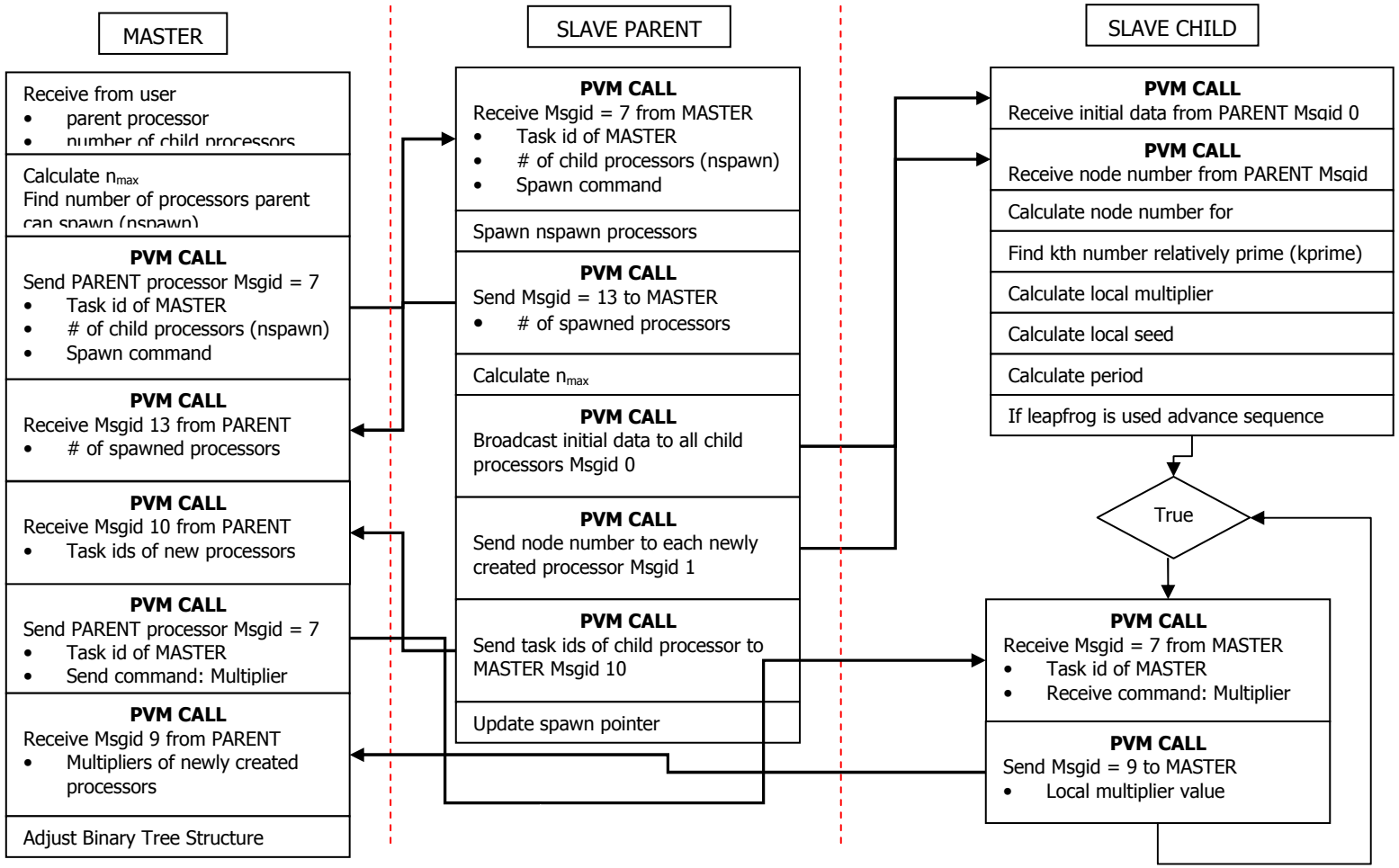


Figure 4.8 Algorithm for spawn operation

4.1.2.6 Random Number Generation Operation

Each processor represented by SLAVE program is responsible from RN generation. MASTER program receives *generate* command from user and sends this command to all the processors in the binary tree structure. Meanwhile, all processors are waiting inside the loop for action commands. Slave programs (processors) receiving *generate* command, start to produce the next RN in the sequence and send that number back to MASTER program. After sending the number, if period is not reached, SLAVE programs start to wait for the *continue* command. Meanwhile, MASTER program receives the numbers from each SLAVE program and prints these numbers to screen and then if period is not reached sends *continue* command to each processor for the next number generation. This interaction continues until the period is reached. Then, RN generation operation ends.

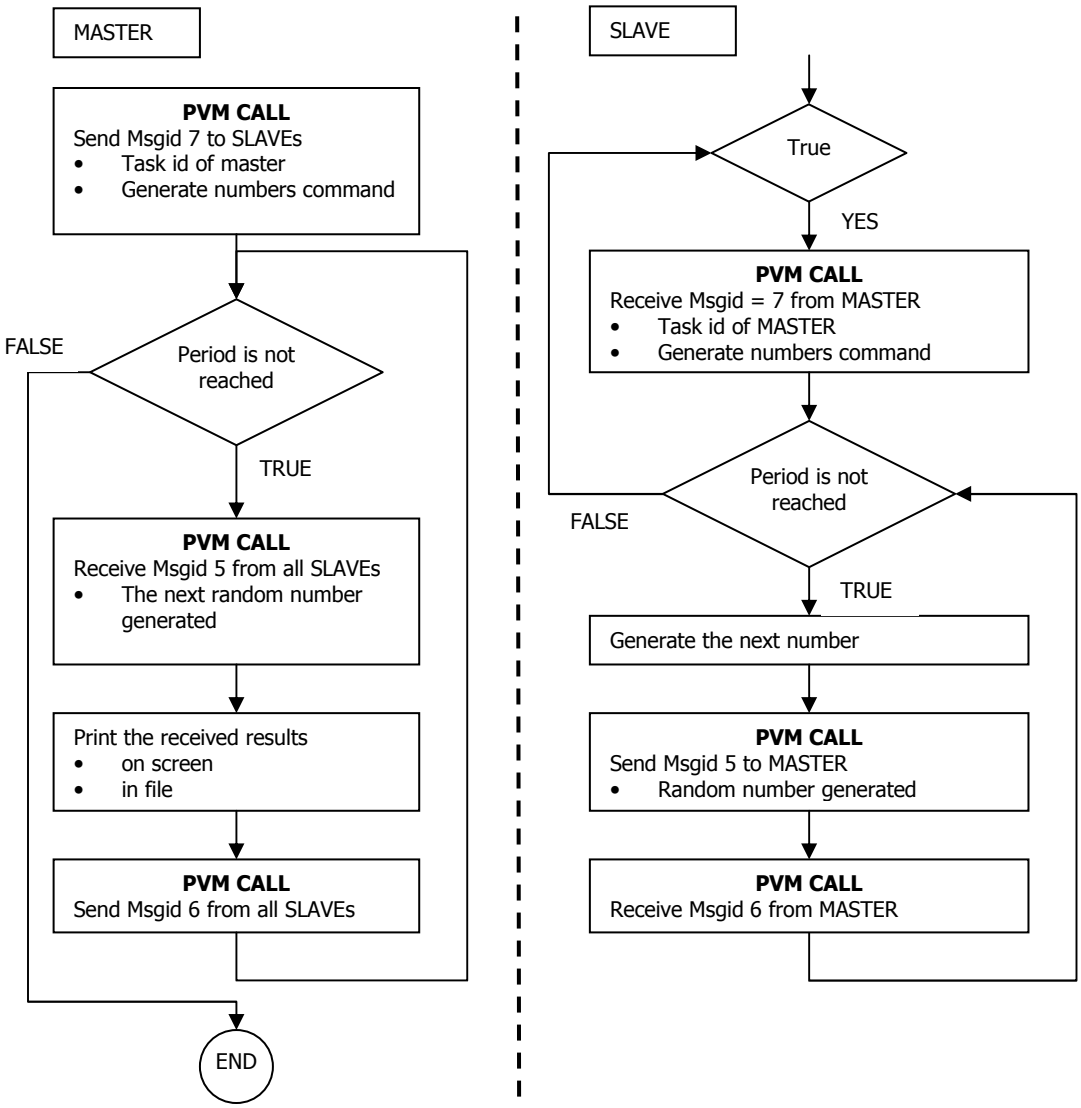


Figure 4.9 Algorithm for RN generation

4.2 Algorithms Related with Spawn Routes

As a solution to the problem of falling off the tree, methods based on traversing binary tree, are proposed. When falling off the tree problem occurs, binary tree is traversed. Traversal starts from root and continues according to the chosen method until a node with not null spawn pointer is reached. When such a node is found, search operation stops and found node is assigned as the new parent. After this assignment is made, *spawn* command is send to the related processor. This operation continues until all the child processors are spawned. By this way, newly assigned parents make the spawning operation in place of the original parent and it is assured that the child processors have uncorrelated sequences.

Problem of falling off the tree occurs when parent node has null spawn pointer or when parent node does not have enough processors in its spawn pool. In both situations, binary tree is traversed but in the second case, before traversing the binary tree, a spawn call is made with the original parent processor. When it is thought in terms of inter-processor communication, second case is more costly than the first case, since it requires one more spawn call in total in order to accomplish the overall spawning operation.

The proposed methods which are defined as spawn routes are different from each other on the way they are traversing the binary tree. The flow chart of the proposed solution, with seven spawn routes and two spawn algorithms can be seen in Figure 4.10. The details of these seven routes are given in the following sections. The comparison of these routes together with running time and cost analysis are given in Section 4.4.

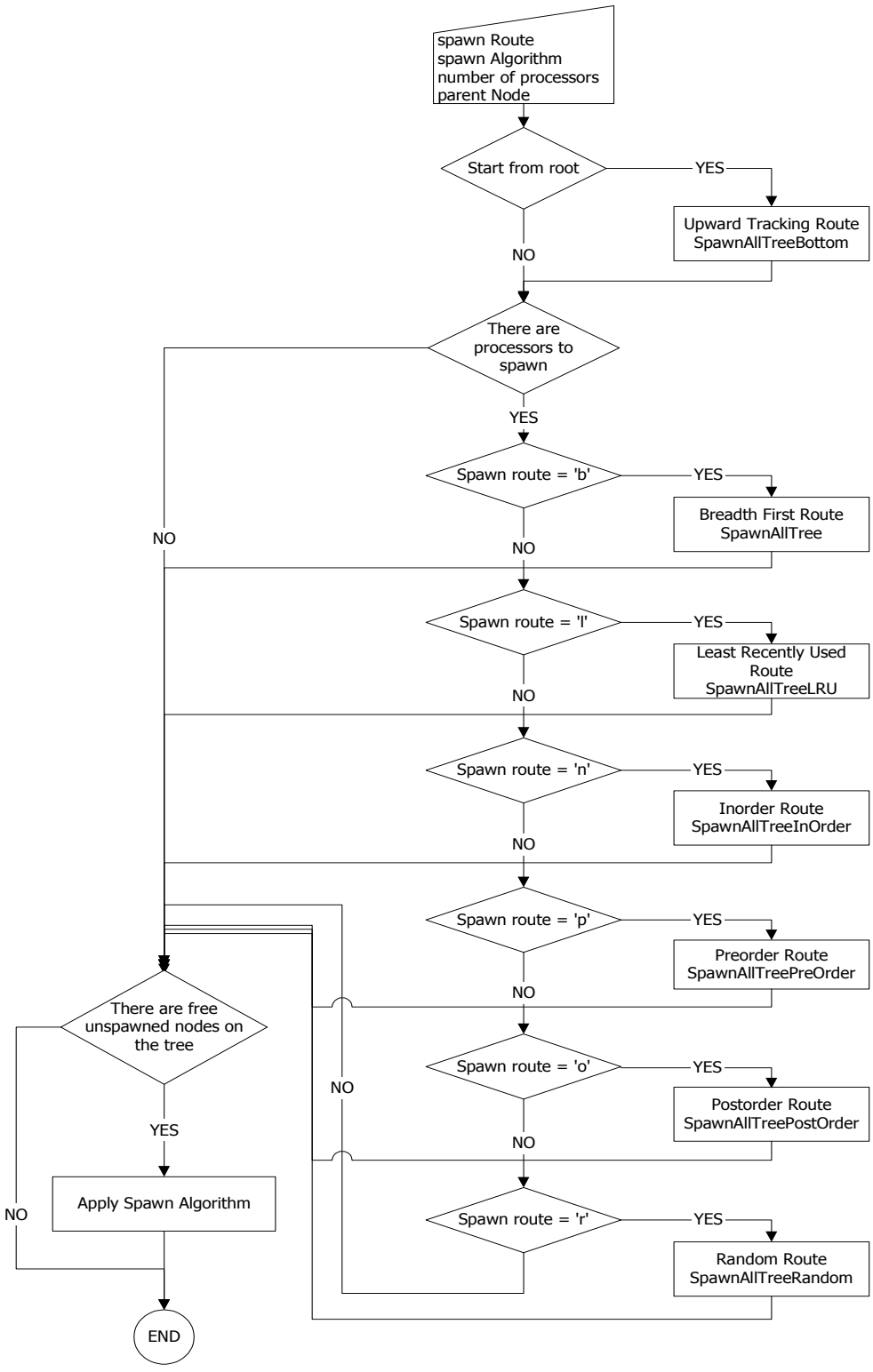


Figure 4.10 Flow chart of spawn search PVM

4.2.1 Breadth First Route

First route is called *Breadth First Route.* As its name implies, in this route, binary tree is traversed in breadth first manner as shown in Figure 4.11. Starting from node 0, until a node with not null spawn pointer is found, binary tree is traversed level by level. The algorithmic details of this route are given in Table 4.1.

Table 4.1 Algorithm for breadth first search

SpawnAllTree	
Input	Pointer to tree structure, number of child processors to spawn
Output	Last node spawned
1	Traverse tree in breadth first manner
2	If node's spawn pointer is not null, assign this node as the new parent node and send *spawn* command to parent processor, return number of processors spawned
3	If all processors are spawned or there is not any processor left to spawn, end, else continue to traverse the tree in breadth first manner

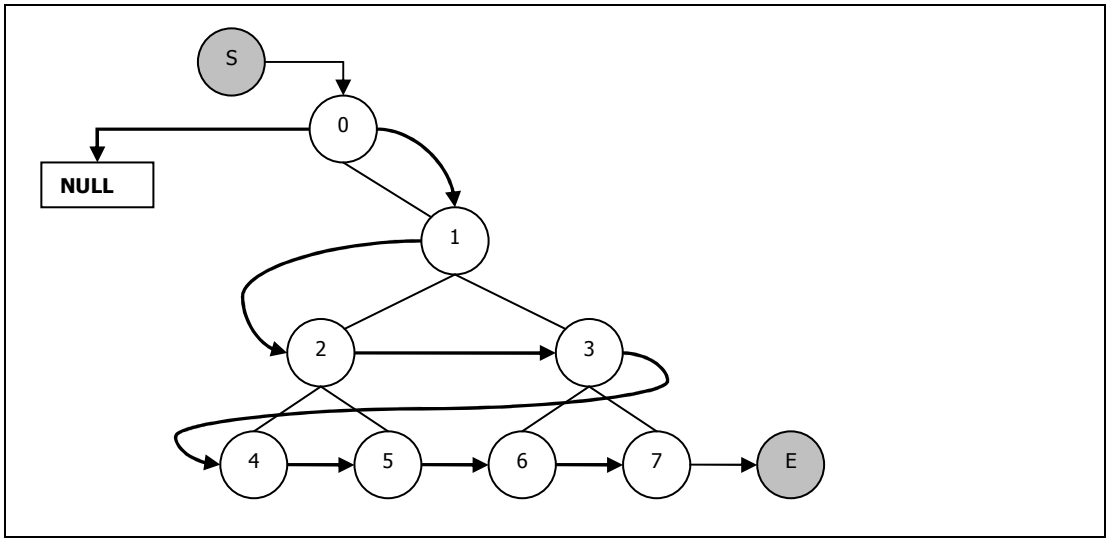


Figure 4.11 Tree chart for breadth first search

4.2.2 Inorder Route

Second route is called *Inoder Route*. In this route, binary tree is traversed in inorder manner as shown in Figure 4.12. Starting from node 0, until a node with not null spawn pointer is found, binary tree is traversed. The algorithmic details of this route is given in Table 4.2. This route is recursive in nature, and nodes are traversed in order 0, 4, 2, 5, 1, 6, 3, and 7.

Table 4.2 Algorithm for inorder route

SpawnAllTreeInOrder	
Input	Pointer to tree structure, number of processors to spawn
Output	Last node spawned
1	Traverse tree in inorder manner
2	If node's spawn pointer is not null, assign this node as the new parent node and send *spawn* command to parent processor, return number of processors spawned
3	If all processors are spawned or there is not any processors left to spawn end, else continue to traverse the tree in inorder manner

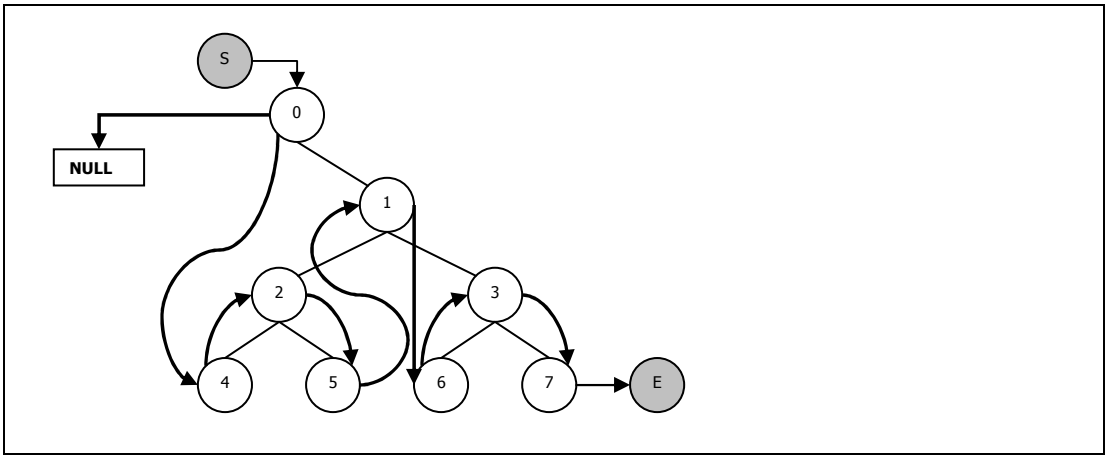


Figure 4.12 Tree chart for inorder route

4.2.3 Preorder Route

Third route is called *Preorder Route*. In this route, binary tree is traversed in preorder manner as shown in Figure 4.13. Starting from node 0, until a node with not null spawn pointer is found, binary tree is traversed. The algorithmic details of this route are given in Table 4.3. This route is recursive in nature, and nodes are traversed in order 0, 1, 2, 4, 5, 3, 6 and 7.

Table 4.3 Algorithm for preorder route

SpawnAllTreePreOrder	
Input	Pointer to tree structure, number of processors to spawn
1	Traverse tree in pre order manner
2	If node's spawn pointer is not null, assign this node as the new parent node and send *spawn* command to parent processor, return number of processors spawned
3	If all processors are spawned or there is not any processors left to spawn end, else continue to traverse the tree in pre order manner

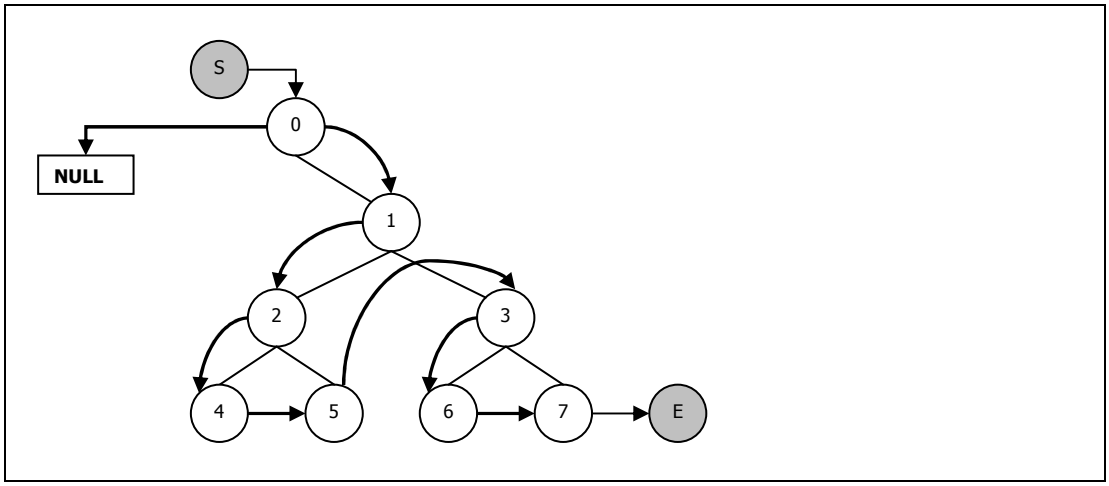


Figure 4.13 Tree chart for preorder route

4.2.4 Postorder Route

Fourth route is called *Postorder Route*. In this route, binary tree is traversed postorder manner as shown in Figure 4.14. Starting from node 0, until a node with not null spawn pointer is found, binary tree is traversed. The algorithmic details of this route are given in Table 4.4. This route is recursive in nature, and nodes are traversed in order 4, 5, 2, 6, 7, 3, 1, and 0.

Table 4.4 Algorithm for postorder route

SpawnAllTreePostOrder	
Input	Pointer to tree structure, number of processors
Output	Last node spawned
1	Traverse tree in post order manner
2	If node's spawn pointer is not null, assign this node as the new parent node and send *spawn* command to parent processor, return number of processors spawned
3	If all processors are spawned or there is not any processors left to spawn end, else continue to traverse the tree in post order manner

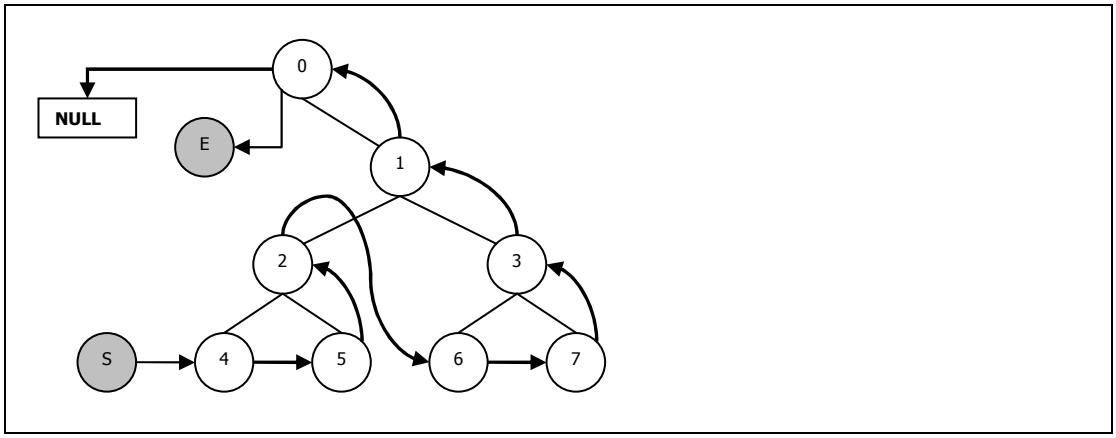


Figure 4.14 Tree chart for postorder route

4.2.5 Upward Tracking Route

Fifth route is called *Upward Tracking Route*. In this route, it is searched for the highest numbered node whose parent is the node that makes the spawn call. If such a node is found and it has not null spawn pointer, then it is assigned as the new parent and search continues like this way, until all the child processors are spawned. In fact, in this route, search area is restricted with the initial spawn pool of parent processor. Consider the situation in Figure 4.15, the initialization operation is accomplished with four processors from 0 to 3. When this is the case, initially, processor 0 has the spawn pool as the subtree rooted by node 4. After many spawning operations, processor 0 has null spawn pointer but it wants to spawn more, according to this route, first place to look is the subtree starting with node 4, in other words, initial spawn pool of the processor 0. If spawn call could not be fully covered with this route, then search can continue with one of the other six methods. The algorithmic details of this route are given in Table 4.5.

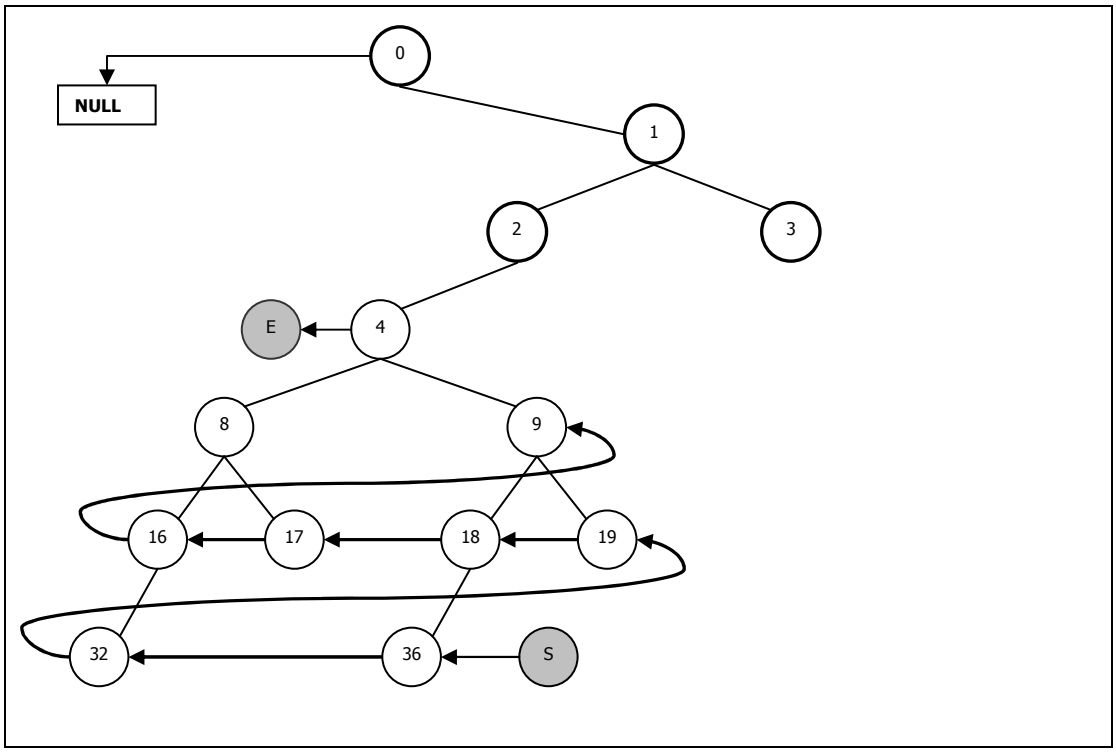


Figure 4.15 Tree chart for upward tracking route

Table 4.5 Algorithm for upward tracking route

SpawnAllTreeBottom	
Input	Pointer to tree structure, parent node id, number of processors to spawn
Output	Last node spawned
While true	
1	Find highest number node whose parent is equal to parent node id and spawn pointer is not null
2	If there is such a node, assign it as the new parent node and send *spawn* command to the parent node, return number of processors spawned
3	If there is not any processors left to spawn assign new parent node as last spawned node
EndWhile	

4.2.6 Least Recently Used Route

Sixth route is called *LRU* (Least Recently Used) *Route*. In this route, with all the nodes in the binary tree structure having not null spawn pointer, a LRU table is created. Then the elements in the LRU table are considered according to the number of processors spawned so far (NumSpawned) and the maximum number of processors that can be spawned by the node (NumToSpawn). For this route, node structure is extended with two new fields which are called NumSpawned and NumToSpawn respectively.

This route relies on the idea that if a processor spawns it is more likely to spawn in the future. So, LRU table is sorted in ascending order with respect to NumSpawned and in descending order with respect to NumToSpawn. The first element of the LRU table is assigned as the new parent node and makes the spawning operation. After finishing the spawning operation, since NumSpawned and NumToSpawn values of the parent node are changed, it is needed to sort the LRU table once more. If there are more to spawn, the assignment procedure continues as explained above. After a LRU table is created in

memory, in order to keep LRU table up to date, results of spawning operations like new processors creations and changes in spawn pointer values are reflected to the table. While new processors are inserted into the LRU table, the processors with null spawn pointers are deleted from LRU table. The algorithmic details of this route are given in Table 4.6.

Table 4.6 Algorithm for least recently used route

SpawnAllTreeLRU	
Input	Pointer to tree structure, number of processors to spawn
Output	Last node spawned
1	If LRU queue is not created, create LRU queue from binary tree structure
2	Sort LRU queue according to least spawned, most to spawn
While LRU queue is not empty and there are more to spawn	
3	Assign first node of the queue as parent node
4	Send *spawn* command to the parent node and return number of processors spawned
5	If there is not any processors left to spawn assign parent node as last spawned node and end
6	Sort LRU queue according to least spawned, most to spawn
EndWhile	

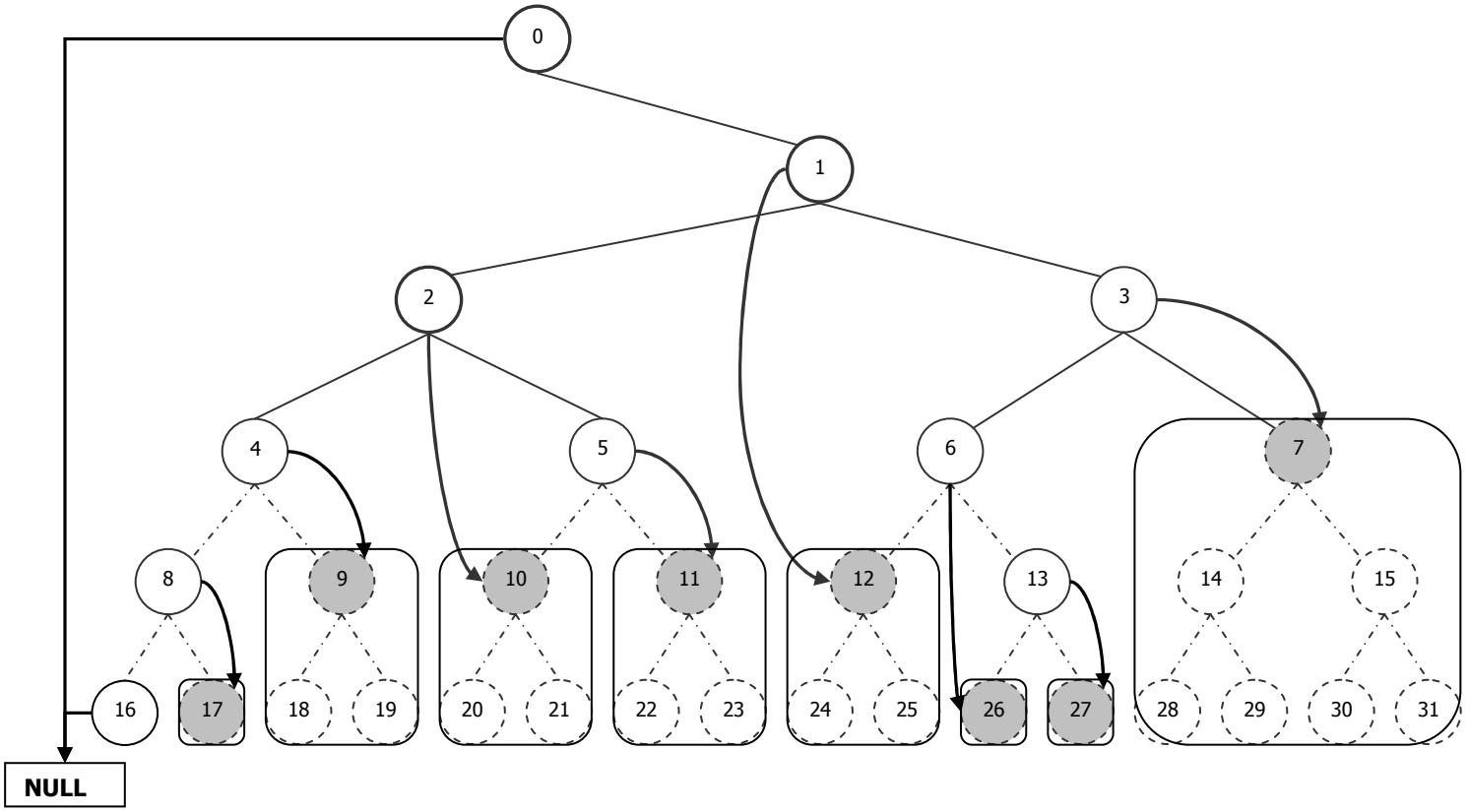


Figure 4.16 State of binary tree after three spawning operations

Consider the example in Figure 4.16, where upper limit for number of processors is thirty two and processor 0 has a null spawn pointer. In such a case, when processor 0 wants to spawn two more processors by using LRU route, firstly, binary tree structure is converted into LRU table and this table is sorted according to NumSpawned and NumToSpawn. The resulting table looks like Table 4.7. For instance, processor 16 is not in LRU table since it has null spawn pointer. So, if there is such a LRU table then the first element of the LRU table which is the node number 3 is assigned as the new parent and makes the spawning operation in place of processor 0. During spawning operation, LRU table is also modified, two new processors 7 and 14 are added into the table and processor 3 is moved to another place in the table since its NumSpawned and NumToSpawn values are changed. The state of the LRU table after spawning operation can be seen in Table 4.8.

Consider that processor 0 again wants to spawn four processors by using LRU route. When this is the case, according to Table 4.8, processor 4 is assigned as the new parent, but this processor is able to spawn at most three processors so at first step only three processors are created (9, 18 and 19). Since, new child processors have null spawn pointers. They are not taken into LRU table. Besides, since processor 4 has a null spawn pointer, it is taken out from the table, leaving it as in Table 4.9. After adjusting LRU table, as a second step, a parent must be found for spawning one more processor. The first element of the LRU table which is processor 5 is assigned as the new parent. As a result of the spawning operation, processor 11 is created. Since processor 11 has not null spawn pointer, it is added to the LRU table, and the modifications on processor 5 are also reflected to the LRU table. As a result, LRU table has the form as in Table 4.10.

Table 4.7 Least recently used table before Spawn(0,2)

Processor Number	NumSpawned	NumToSpawn
3	**0**	**7**
4	0	3
5	0	3
8	0	1
13	0	1
2	1	3
6	2	1
1	2	3

Table 4.8 Least recently used table after Spawn(0,2)

Processor Number	NumSpawned	NumToSpawn
4	**0**	**3**
5	0	3
7	**0**	**1**
8	0	1
13	0	1
14	**0**	**1**
2	1	3
3	**1**	**1**
6	2	1
1	2	3

Table 4.9 Least recently used table before Spawn(0,4)

Processor Number	NumSpawned	NumToSpawn
5	**0**	**3**
7	0	1
8	0	1
13	0	1
14	0	1
2	1	3
3	1	1
6	2	1
1	2	3

Table 4.10 Least recently used table

Processor Number	NumSpawned	NumToSpawn
7	0	1
8	0	1
13	0	1
11	**0**	**1**
14	0	1
5	**1**	**1**
2	1	3
3	1	1
6	2	1
1	2	3

4.2.7 Randomly Chosen Route

The last route is called *Randomly Chosen Route*. In this route, with all the nodes in the binary tree structure having not null spawn pointer, an index queue is created. Then, randomly an index is chosen and the row in the table with that index is assigned as the new parent. If there are more processors to spawn, assignment is accomplished as explained above. The algorithmic details of this route are given in Table 4.11.

Consider the example in Figure 4.16, where upper limit for the number of processors is thirty two and processor 0 has a null spawn pointer. In such a case, when processor 0 wants to spawn two more processors by using randomly chosen route, firstly, from the nodes of the binary tree structure having not null spawn pointers, an index queue is created. The structure of the queue can be seen in Figure 4.17. The queue has eight elements so the RN must be chosen between one and eight. If it is assumed that the RN is chosen as seven, then, the node on the seventh position of the queue is assigned as the new parent.

Table 4.11 Algorithm for randomly chosen route

SpawnAllTreeRandom	
Input	Pointer to tree structure, number of processors
Output	Last node spawned
1	Traverse the tree, push the nodes with not null spawn pointer to queue
While queue is not empty	
2	Randomly chose an index of the queue, assign it as parent
3	Send parent node *spawn* command and return the number of processors spawned
4	If parent node's spawn pointer is null, delete parent node from queue
5	If more processors to spawn continue. Otherwise, end
EndWhile	

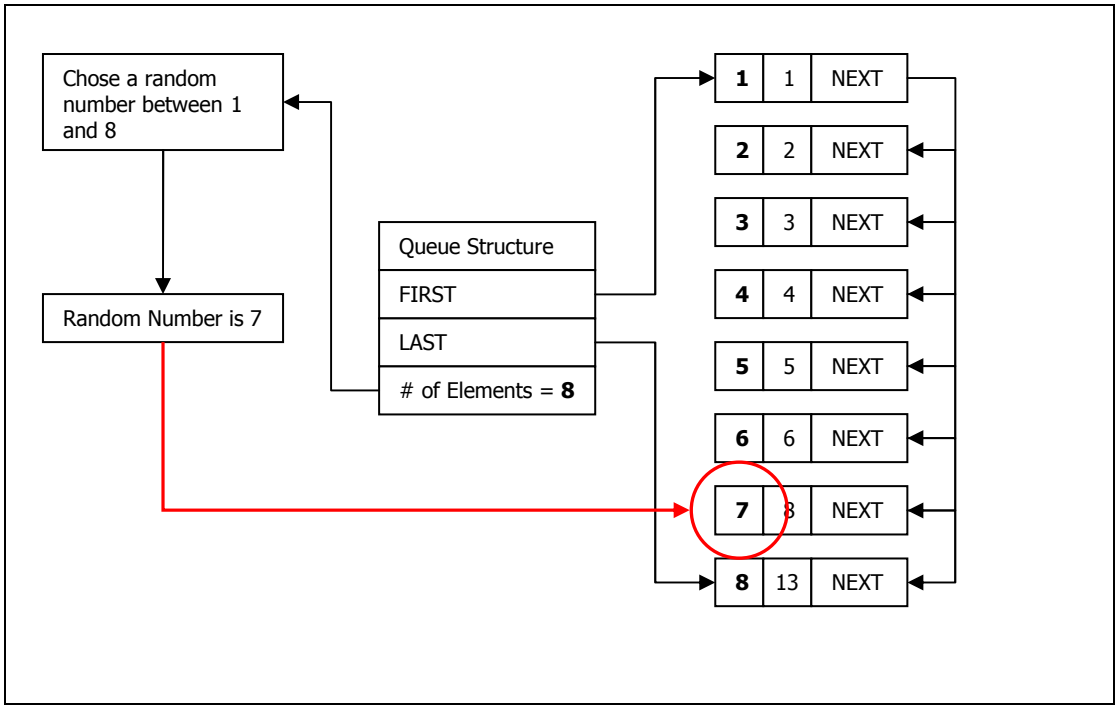


Figure 4.17 Random search

4.3 Algorithms Related with Spawn Pointers

As a solution to the problem of falling off the tree, in the previous section, several spawn routes are explained. These routes are based on binary tree traversal and parent node assignment. After new parent is determined, spawn call is made to that parent processor through PVM calls. When spawn routes are considered from cost perspective, two extra costs are encountered which do not exist in an ordinary spawning operation. These costs can be defined as the cost of traversing binary tree and the cost of making a spawn call for the new assigned parent processor. Especially, second cost has more significance since it appears as an inter-processor communication cost which must be minimized as much as possible. In order to reduce inter-processor communication cost, proposed solution is enhanced with two algorithms which are called spawn algorithms.

Spawn algorithms are based on the fact that a processor spawned before is more likely to spawn in the future. That is why; if a processor has null spawn pointer and wants to spawn, after spawning operation is accomplished through spawn routes, original processor's spawn pointer is updated according to these algorithms and the updated values are sent to related processors through PVM calls. By this way, if the same processor wants to make a spawn call in the future, it can do this without need for binary tree traversals and extra spawn calls since, it has non zero spawn pointer value. These spawn algorithms extends the node structure further by adding a new field called **Direction** and changes the spawn pointer calculation formula slightly. The details of these algorithms are given in the following sections. The flow chart of spawn algorithms can be seen in Figure 4.18.

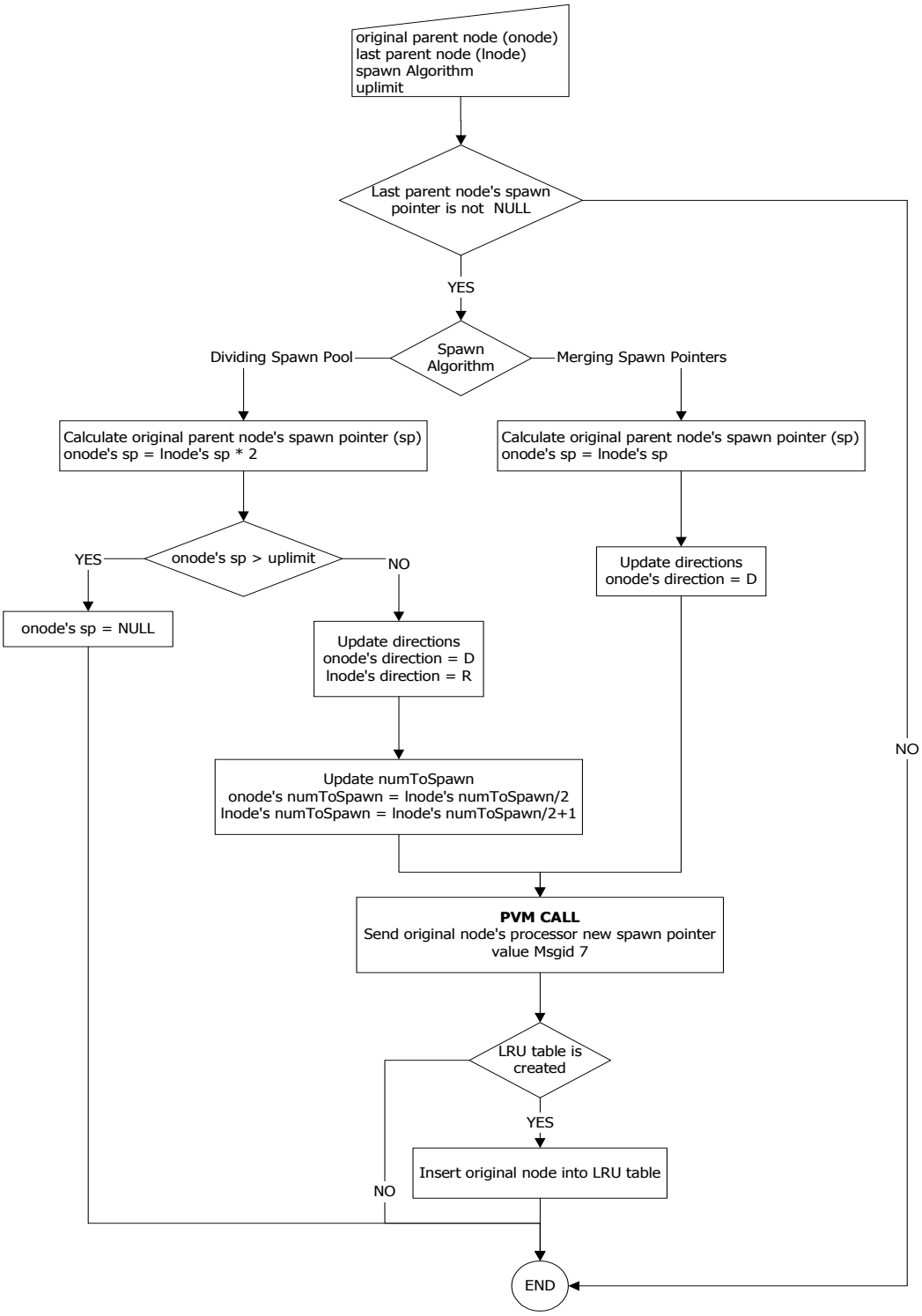


Figure 4.18 Flow chart of spawn algorithms

4.3.1 Dividing Spawn Pool

First algorithm is called *Dividing Spawn Pool*. As its name implies, it is based on the idea of dividing the spawn pool of the new parent processor into two, between itself and the original processor. For this algorithm, **Direction** field is inserted into node structure. It can have three values (L, D and R). Initially, when a node is created, its direction is given as L. Direction field shows which way to take from the root node of the spawn pool while advancing deeper into the tree and directly affects the calculation of spawn pointer. The flow chart for calculating spawn pointer values with respect to different direction values can be seen in Figure 4.19. If parent node's direction is L, its spawn pointer is updated according to Formula 3.6. When its direction is D, new spawn pointer value is found by doubling the old spawn pointer until it is higher than n_{max}. When its direction is R, firstly, the spawn pointer value of the first child node created is found. Then, parent's spawn pointer is updated by doubling the child node's spawn pointer value. During all these calculations, calculated value is checked against the upper limit for the number of processors and updated as null in cases where it is higher than the upper limit.

Consider the example in Figure 4.20 where initial state of a binary tree is shown. Processor 0 has a null spawn pointer and wants to spawn two more processors. If it is assumed that breadth first route is used, then processor 1 is assigned as the new parent processor and makes the spawn call in place of processor 0. The state of the binary tree after spawning operation is shown in Figure 4.21. Two new processors are created (6, and 12) and processor 1's spawn pointer is updated. Now, processor 1 has a spawn pool as subtree rooted by node 24. According to dividing spawn pool algorithm, this spawn pool must be divided between processor 1 and processor 0 and the direction fields of each node must be updated. As a result of this division operation, processor 0 gets the subtree rooted by 48 as its spawn pool. Also, its direction is changed to D. On the other hand, processor 1 has a spawn pool of starting from node 24 and going right to subtree rooted by node 49 and its direction is updated to R. As a last remark, consider that node 24 is defined as *Temporary Node* in Figure 4.21. When the spawn pool is divided into two between processor 1 and processor 0. Processor 0 gets the spawn pointer as node 48. But in fact, node 48 is the child of node 24 according to the binary tree structure. Meaning that, in order to have a node 48, it is compulsory to have node 24. But there can be situations where processor 0 spawns before processor 1. In such cases, problems occur since a node is wanted to be created

whose parent has not been created yet. To prevent such situations, a temporary node is created in the binary tree structure. It is important to note that, this node is only for consistency purposes and it has no correspondence with a real processor. Later, for instance, when processor 1 decides to spawn, this temporary node is converted into a real node with a specific processor assignment.

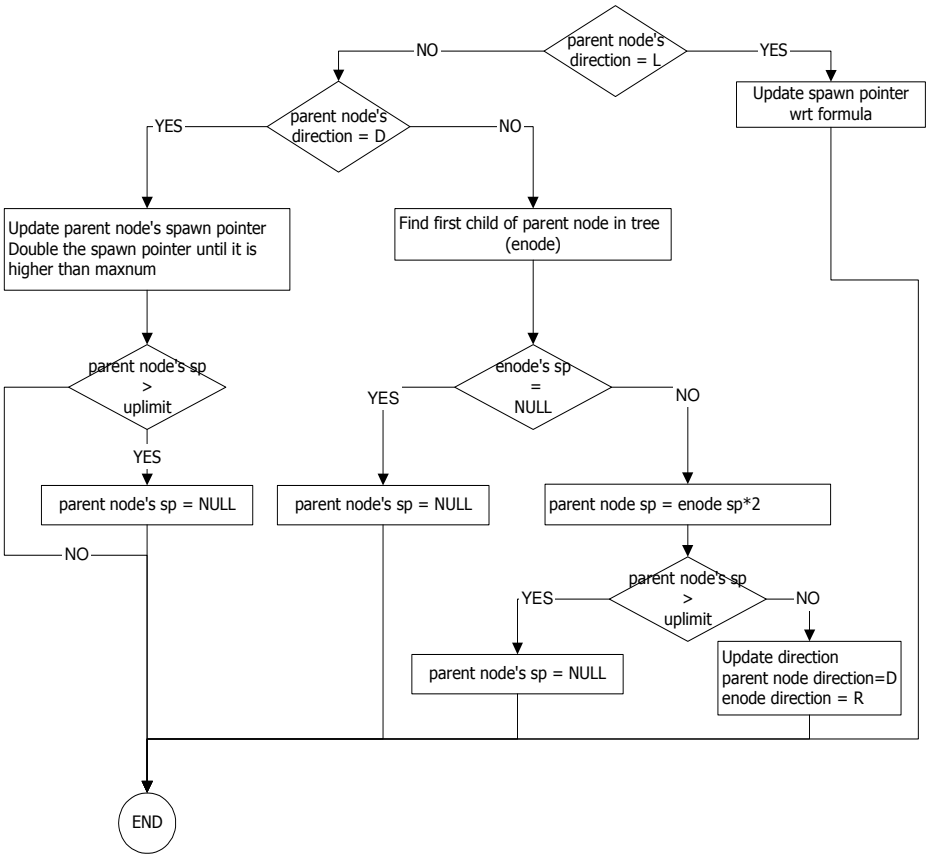


Figure 4.19 Flow chart of calculating spawn pointer values

After processor 1 spawns two processors and spawn pointers are updated, consider the case where processor 0 wants to spawn one processor. Since, processor 0 has non zero spawn pointer, without a need for spawn route, spawning operation can be carried out. As a result, processor 48 is created and spawn pointer of node 0 is updated according to the flow in Figure 4.19. Since only one processor is created, n_{max} is 48. New spawn pointer value of processor 0 is calculated by doubling the old value until it is higher than forty eight, causing it to be ninety six. The state of the binary tree after processor 0 spawns one processor can be seen in Figure 4.22.

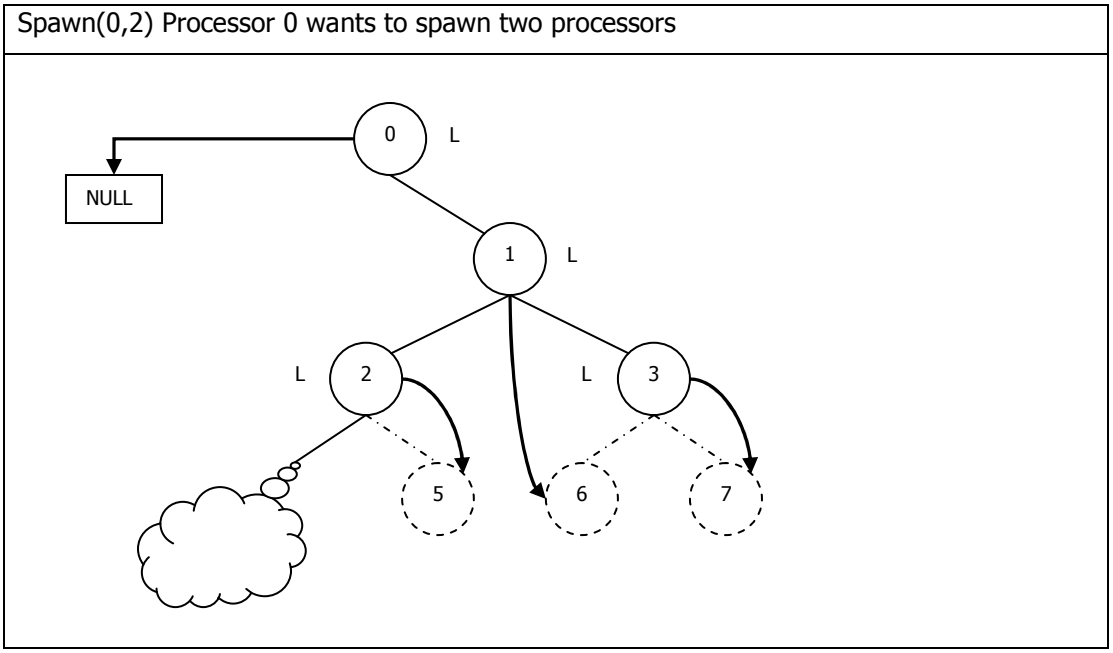


Figure 4.20 Initial state of the binary tree

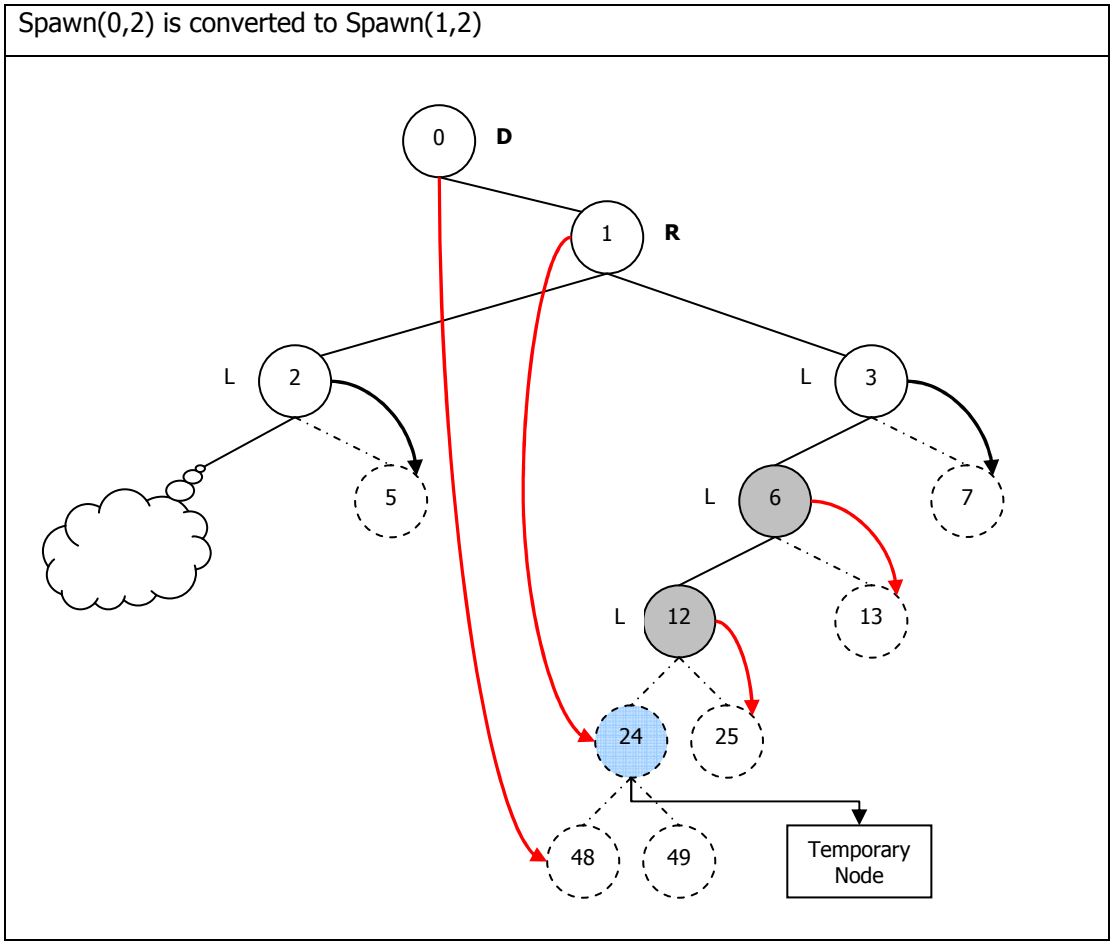


Figure 4.21 State of the binary tree after Spawn(1,2)

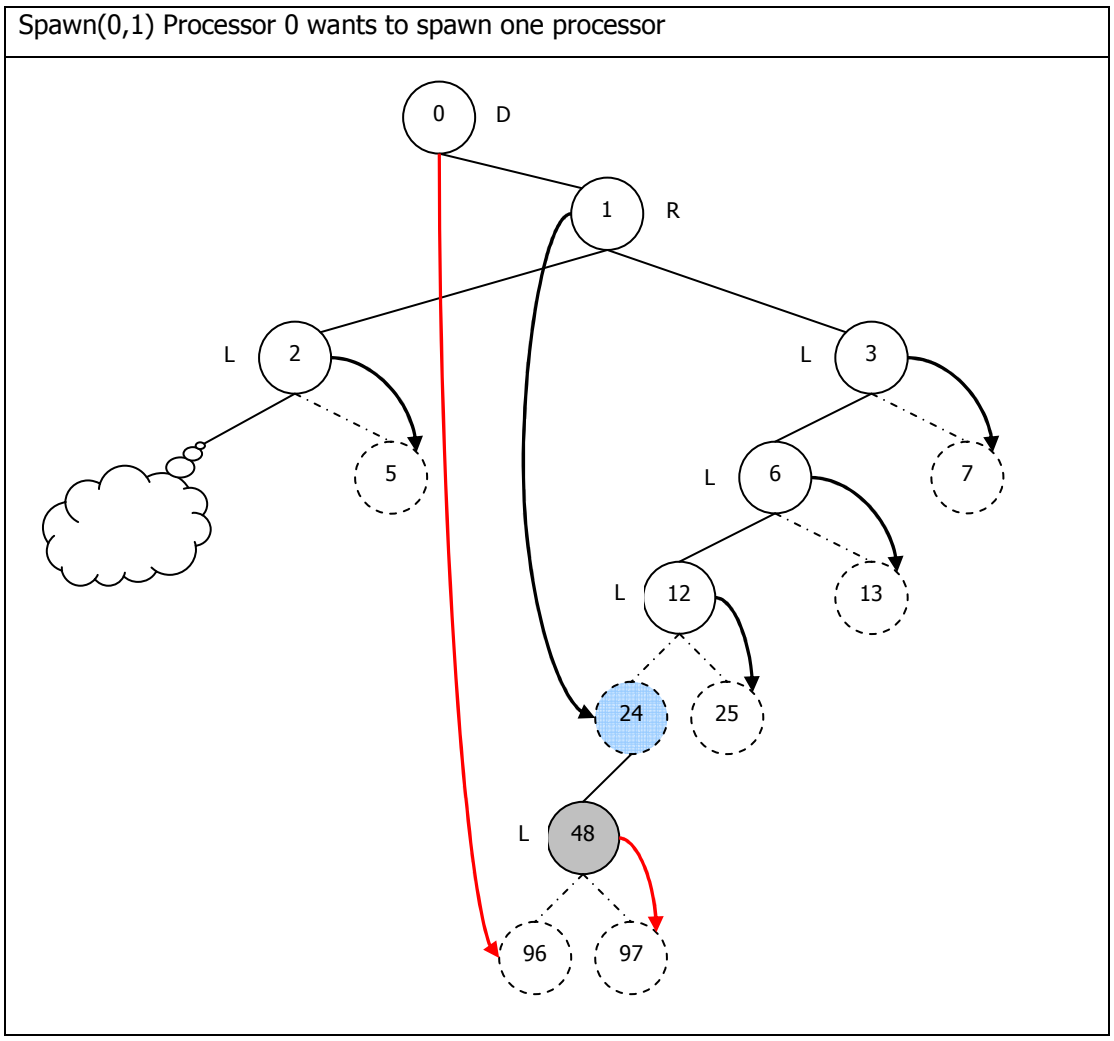


Figure 4.22 State of the binary tree after Spawn(0,1)

As another example, consider the case where processor 1 wants to spawn two new processors. Since processor 1 has direction R, after creating processor 24, it is continued from right subtree and processor 49 is created. Since these newly created processors have direction L, their spawn pointer values are calculated according to the Formula 3.6. Calculation of spawn pointer of processor 1 can be described as dividing the spawn pool of processor 24 between itself and processor 1. In order to update spawn pointer of processor

1, firstly, the spawn pointer of processor 24 is found, which is node 98. Then, processor 1's spawn pointer is calculated by doubling the spawn pointer value of processor 24. So, processor 1's spawn pointer is updated to node 96. Besides, processor 1's direction is updated to D. Whereas, processor 24's direction is changed to R. The state of the binary tree structure after processor 1 spawns two processors can be seen in Figure 4.21.

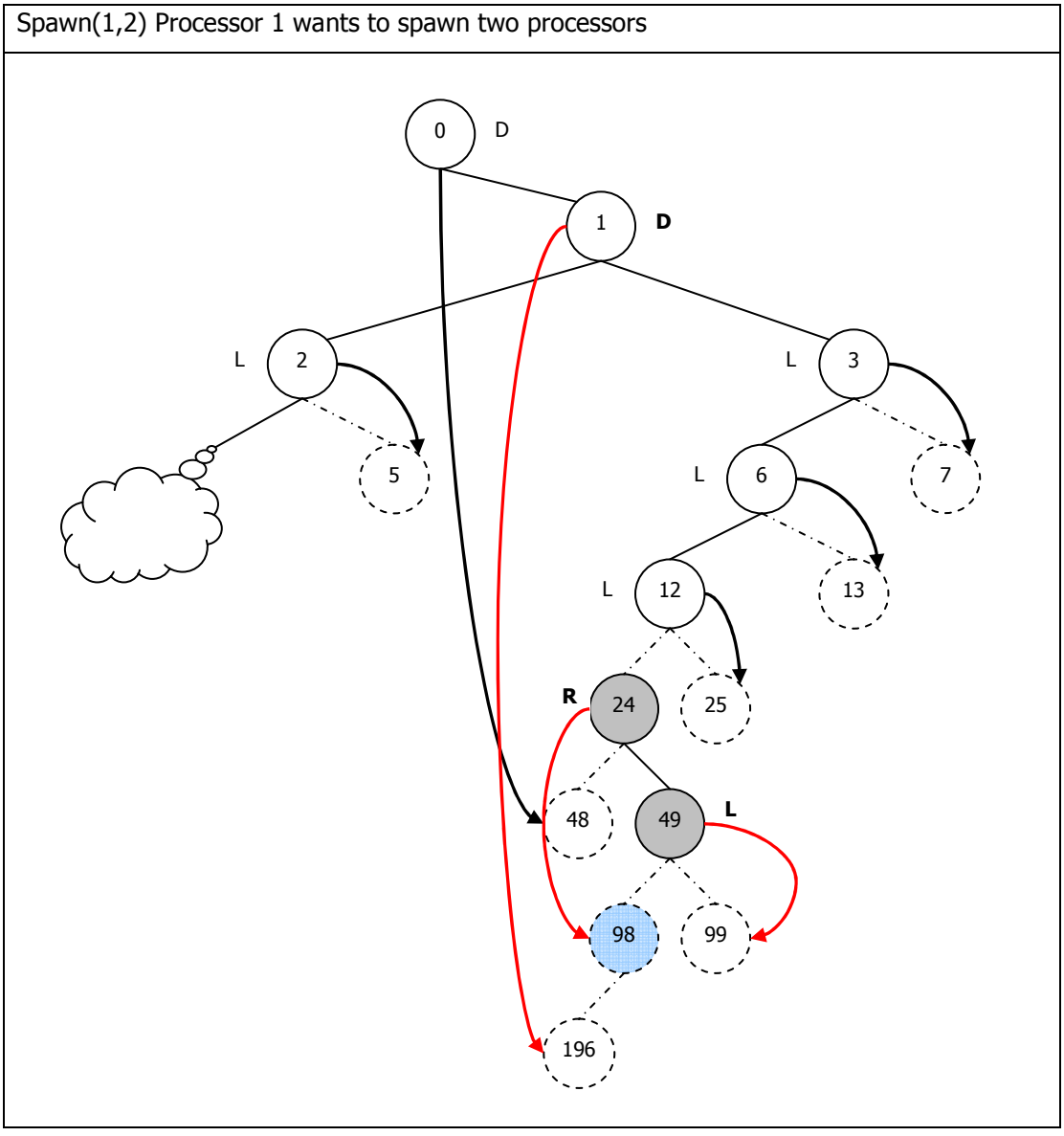


Figure 4.23 State of the binary tree after Spawn(1,2)

4.3.2 Merging Spawn Pointers

Second spawn algorithm is named as *Merging Spawn Pointers*. In this algorithm, unlike the previous algorithm, spawn pool is not divided among processors but several processors are directed to the same spawn pool. In this algorithm, when a processor spawns, not only the spawn pointer of that processor is updated but also other processors pointing to the same spawn pool need to update their spawn pointer values. This algorithm requires an overall adjustment of spawn pointer values on the binary tree structure. Besides, these updated values must also be sent to their corresponding processors.

This algorithm is suitable for architectures where spawning operation is delivered from a single program since there can be correlations when two processors pointing to the same spawn pool want to spawn simultaneously. Consider the example binary tree structure in Figure 4.24. Here, processor 0 has null spawn pointer and wants to spawn two more processors. If it is assumed that binary tree is traversed in breadth first route, then processor 1 is assigned as the new parent processor. After this assignment, processor 1 accomplishes the spawning operation by creating processors 6 and 7 respectively and updates its spawn pointer to node 24 by using the Formula 3.6. According to merging spawn pointers algorithm, processor 0 is also assigned to the spawn pool of processor 1 by updating the spawn pointer value to node 24. Besides, processor 0's direction value is changed to D. The state of the binary tree structure after processor 1 spawns two processors can be seen in Figure 4.25.

After processor 1 spawns two processors, processor 1 and processor 0 points to the subtree rooted by node 24. When processor 0 wants to spawn one more processor, it accomplishes the spawning operation locally and its spawn pointer value is updated to forty eight by doubling the old spawn pointer value until it is higher than the maximum numbered node created. Updated value is also sent to the processor 0 through PVM calls. After spawning operation ends, the binary tree structure is traversed looking for the nodes having spawn pointer value equal to twenty four. Node 1 is found. Its spawn pointer value is updated to forty eight and this value is sent to the processor 1 through PVM calls. The state of the binary tree structure after spawning operation can be seen in Figure 4.26.

Spawn(0,2) Processor 0 wants to spawn two processors

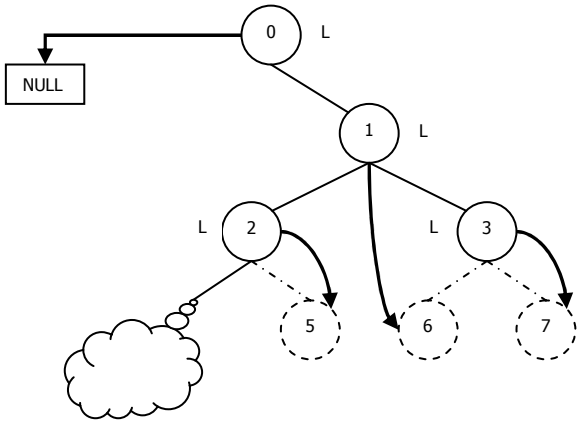


Figure 4.24 Initial state of the binary tree

Spawn(0,2) is converted to Spawn(1,2)

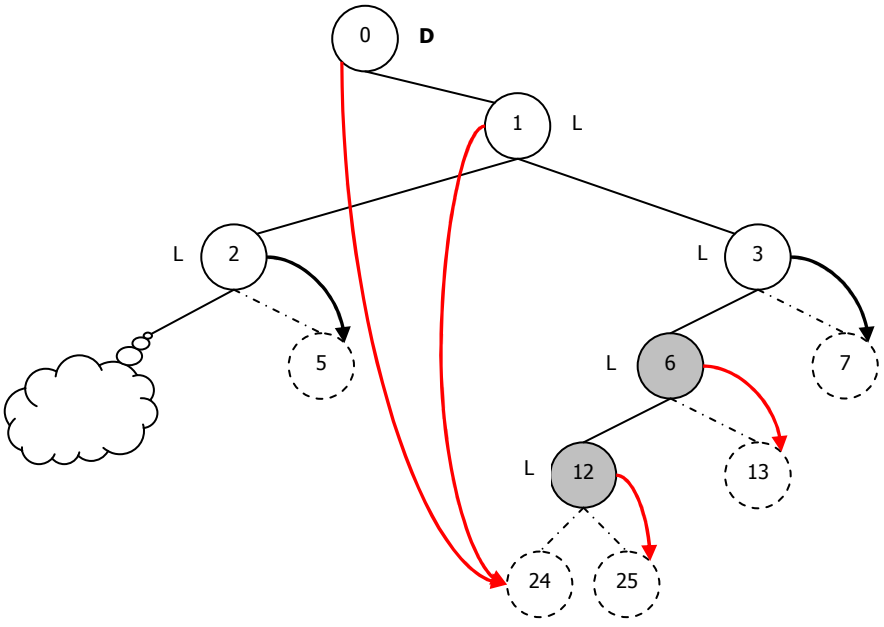


Figure 4.25 State of the binary tree after Spawn(1,2)

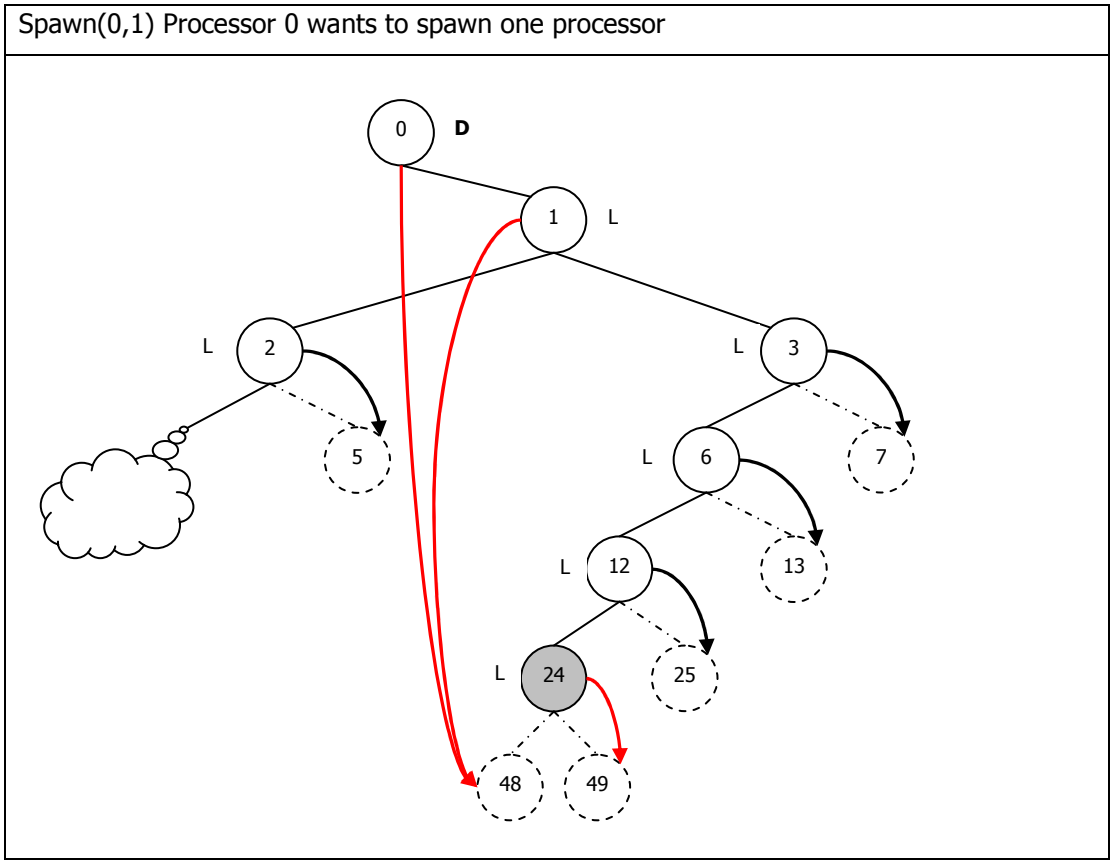


Figure 4.26 State of the binary tree after Spawn(0,2)

4.4 Analysis of Improvements

In this thesis, a solution to the problem of falling off the tree is proposed. This solution is based on the idea of traversing the binary tree and finding new spawn parents. In order to accomplish such assignments, several spawn routes are proposed. When the result of such a solution is considered from performance perspectives, it is seen that two extra costs appear, binary tree traversal cost and the cost of the spawn call that is to be made for the new spawn parents. In order to reduce these costs, proposed solution is enlarged with two spawn algorithms. The details of these spawn routes and spawn algorithms are given in the previous sections. Now, the performance consequences of these routes and algorithms will be examined and the best method to use will be defined.

Before analyzing the performance aspects of the proposed solution, it is better to define the performance criteria for the proposed solution. The spawning operation is accomplished by the new solution must be made with minimum number of spawn calls and in the resulting binary tree structure, spawn pools must be equally distributed and the number of null spawn pointer assigned processors must always be minimized. These two criteria are determined in order to reduce the extra costs that are gained as a result of the proposed solution. All spawn routes and spawn algorithms must be examined with respect to these performance criteria.

While searching the binary tree structure to find the new spawn parent, the chosen spawn route determines the path to be taken through the tree. As going deeper into the binary tree structure, the probability of choosing a processor with a small spawn pool as the new spawn parent increases. Choosing such a processor leads to two disastrous situations and disobeys the rules of performance criteria. First one is the case where the spawn request can not be covered with that spawn parent, since it does not have enough elements in its spawn pool. When this is the case, several binary tree traversals and spawn calls are accomplished until the spawn request is fully covered. In the second case, although, the spawn request is covered at once, the spawn pool is so small that it can not be divided among the parent and the newly created processors. This leads to spawn parent and newly created processors to have empty spawn pools leading to an increase in the number of null spawn pointer assigned processors. So, in order to have better performance, it is recommended that spawn request should be handled from the processors in the upper levels of the binary tree. As a result, it can be stated that, spawn routes that start traversing the binary tree from upper levels have better performance consequences.

In Figure 4.27, the binary tree traversal time for different sized binary trees are represented. For sparse binary trees, traversal time of the whole binary tree is low. Whereas, in massive binary trees with high amounts of nodes, binary tree traversal time increases considerably. In fact, as can be seen from the trend line in Figure 4.27, there is a linear relation between number of nodes and binary tree traversal times.

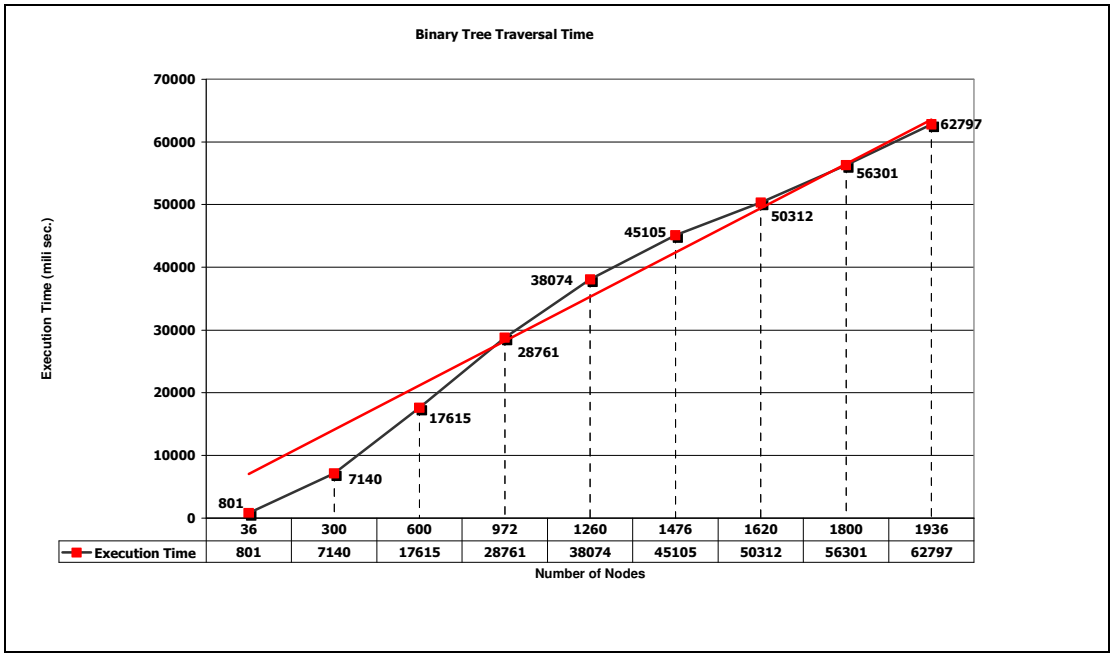


Figure 4.27 Binary tree traversal time

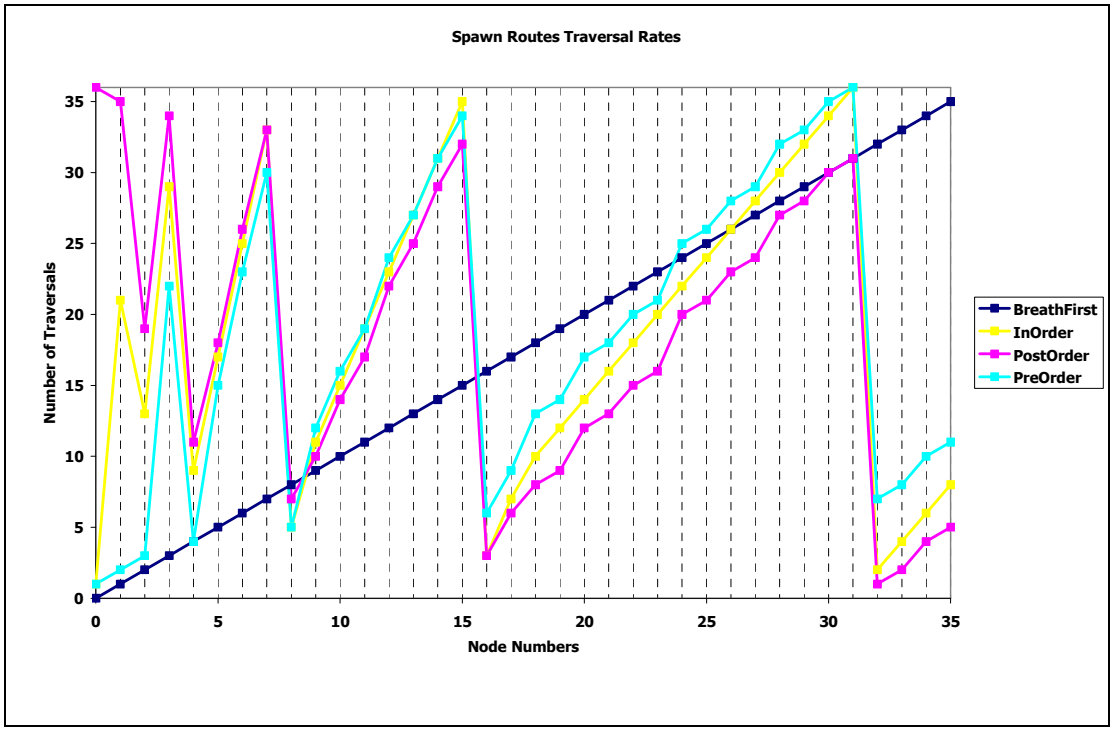


Figure 4.28 Spawn routes traversal rates

When spawn routes are considered according to performance criteria, it is seen that breadth first and preorder routes are better in performance than inorder, postorder and upward tracking routes. Since, they are traversing the binary tree in top to bottom approach. In Figure 4.28, for different spawn routes, the number of traversals needed for finding a specified node is shown. For instance, for finding node 0, in breath first route, number of traversals needed is zero. While in postorder route, thirty five traversals are required in order to reach node 0. Likely, in order to reach node 15, in breadth first route, fifteen traversals, in inorder route, thirty five traversals, in postorder route, thirty two traversals and in preorder route, thirty four traversals are needed. Number of traversals vary greatly from route to route, leading to considerable differences in traversal times. When examined from this point of view, breadth first route seems to be the most stable route. Since, traversal time is directly related with the node number to be found. For inorder route and postorder routes, node 32 requires the least number of traversals. For preorder route and breadth first route, node 0 needs the lowest number of traversals. This situation appears from the fact that, preorder and breadth first routes start searching from the top of the binary tree whereas, inorder and postorder routes start from bottom.

When choosing the spawn route, two facts must be kept in mind. First one is, it is better to choose a route that searches from top to bottom since, the nodes in the upper levels have higher number of elements in their spawn pools. Second one is, it is preferable to keep the traversal time as small as possible. When these consequences are considered, initially, it becomes reasonable to start with breadth first route until a certain threshold and then continue with a route that starts searching from the bottom. This threshold value for a binary tree with at most thirty six nodes can be easily recognized in Figure 4.28. Until node 15, the traversal time of breadth first route is the best when compared with the others. After that, it is seen that, breadth first route becomes the worst one. While, postorder route is the best. It can be stated that, spawn route must be chosen according to the current state of the binary tree. If the number of nodes with null spawn pointer values in the upper levels is high, then it is better to start with a route that starts searching from bottom. Otherwise, routes starting from top can be chosen together with a threshold value when reached, it should be switched to a route that starts searching from bottom.

Random route and LRU route are different in structure since they are not based on binary tree traversal while searching for the new spawn parent. In random route, every time, binary tree is converted into an index queue. As the binary tree structure gets deeper, the cost of this conversion operation increases considerably. That is why; for cases with vast amount of processors, this route is not recommended. In LRU route, at first, LRU table is created from binary tree structure. Once it is created, no binary tree traversals remain since LRU table always kept current. In Figure 4.29, the LRU table creation time is given for different sized binary trees. As the binary tree grows, the time to convert it into LRU table increases. LRU table is sorted in ascending order with respect to NumSpawned and in descending order with respect to NumToSpawn. The first element of the LRU table is assigned as the spawn parent. By this way, the processor which spawned less and with the widest spawn pool is appointed as the spawn pointer. Although, LRU route tries to make a uniform distribution among processors' spawn pools, it requires, two costly operations, which are conversion of binary tree into LRU table and sorting of LRU table. Conversion cost can be thought of as an initialization cost. Whereas, sorting cost is encountered during each spawn parent search. The overall execution cost of LRU route is given in Figure 4.30. Initially, binary tree with thirty six nodes is converted into LRU table and is sorted accordingly. Then, as the number of nodes increases, only the sorting operation is required. LRU route on its own, can be thought as unattractive but when combined with a spawn algorithm, it can become feasible.

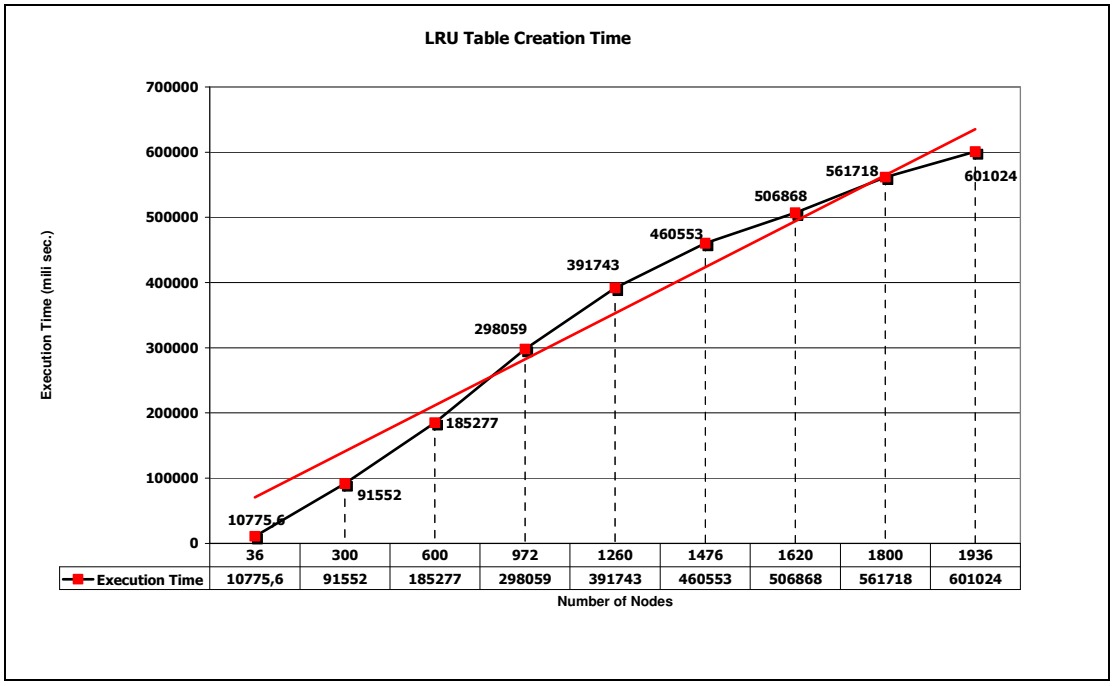


Figure 4.29 LRU table creation time

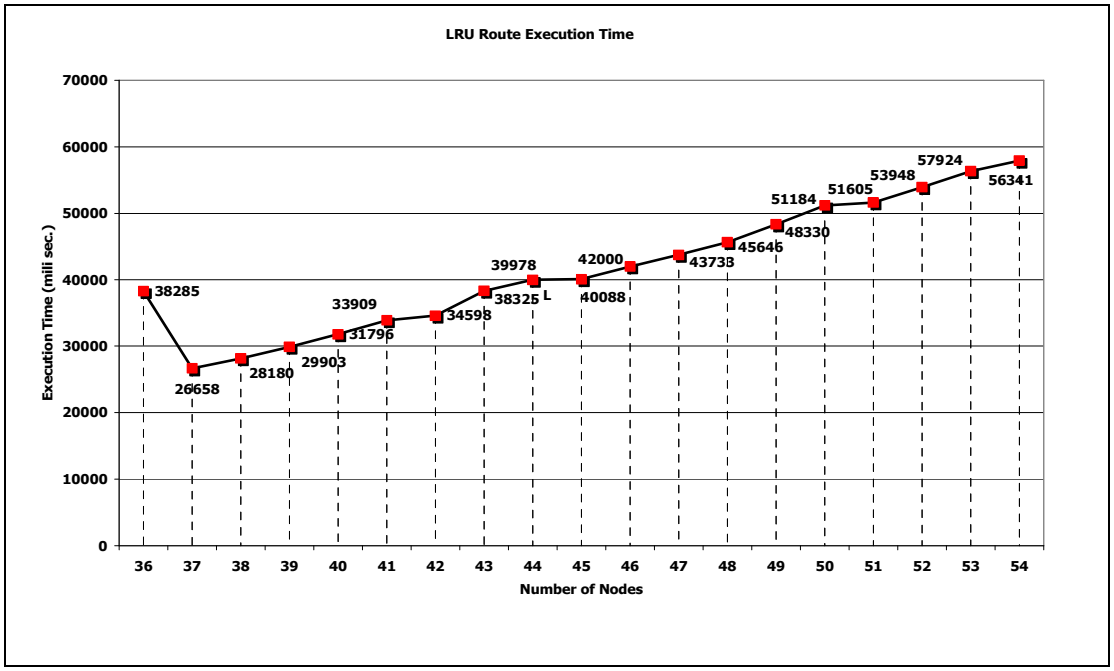


Figure 4.30 LRU route execution time

Before analyzing the performance of two spawn algorithms, it is necessary to understand the effects of the binary tree traversal and spawn call costs on the spawning operation. The effects can be described by considering three spawn operations. First case is the ordinary spawning operation where there is no need for binary tree traversal, only related spawn call is made to the parent processor. In the second case, parent processor has a null spawn pointer that is why, binary tree is traversed to find the new spawn parent. Spawn call is made to the new spawn pointer. If the spawn request is covered, execution stops, else it continues with traversing the binary tree. In the second case, there is at least one binary tree traversal cost and one spawn call cost. In the third case, parent processor does not have a null spawn pointer but it does not have enough elements in its spawn pool either. In such a case, firstly spawn call is made to the parent. Then, binary tree is traversed in order to find the new spawn parent. Spawn call is made to the new spawn pointer. If the spawn request is covered, execution stops, else it continues with traversing the binary tree. In the third case, there are at least two spawn call costs and one binary tree traversal cost. The cost analysis of these three cases can be seen in Table 4.12. When considering the spawning operation with null spawn pointer from cost perspective, it is seen that best approach is to find a spawn parent that is able to cover the spawn request at once. But this is not often possible, so a more reasonable approach can be to be able to cover the spawn request with minimum number of spawn calls and binary tree traversals.

Table 4.12 Cost analysis of spawning operation

	Spawn Call Cost	**Binary Tree Traversal Cost**
1st Case	1	-
2nd Case	>=1	>=1
3rd Case	>=2	>=1

In Figure 4.31, an example case that clarifies the three cases of spawn operation is given. The initial state of the binary tree contains four nodes. Firstly, processor 0 spawns three processors simultaneously through normal spawning operation according to case 1. So, in Figure 4.31, while processor number increases from four to seven, total execution time remains unchanged. After that processor 0 attempts to spawn two more processors. Only one processor can be spawned by processor 0, for the other one, binary tree is traversed in breadth first route and processor 1 is assigned as the new spawn parent. In order to spawn two processors, two spawn calls and one binary traversal are accomplished as can be seen in step 4 of the Table 4.13, leading to a case 3 situation. Execution time is leading to a peak. Then, processor 0 wants to spawn two more processors. Since, its spawn pool is empty, binary tree is traversed according to breadth first route and again processor 1 is assigned as the new spawn parent. Fortunately, spawn request can be covered from the spawn pool of processor 1, leading to a case 2 situation. This time, a small increase in execution time is observed. Example case continues in this manner with peaks and small increases. Execution time for a spawn operation can be summarized as follows. As can be seen from Figure 4.31, case 1 has no effect on execution time. That is why; it can be defined as the ideal case. Whereas, case 2 and case 3 lead to an increase in the execution time that is directly related with the number of processors joining to the spawning operation. If the spawn call can be handled with less number of processors, then, there happens a small increase in execution time. Otherwise, there appears a peak. It is to be sure that, case 3 takes longer execution time than case 2. In fact, when the number of processors that are used to cover a spawn request increases, execution time increases considerably. Such situations are represented in Table 4.13 with bold letters. The most costly operation is making the spawn call since it necessitates inter-processor communication. That is why; spawning operations should be handled in a way that minimizes the number of spawn calls. Execution times for several spawn sequences can be seen in Figure 4.32. As it is seen in Figure 4.32, there can be situations with very low cost while on the other hand, there can be situations where execution time has enormous values. Such difference in execution time directly depends on the properties and necessities of the application.

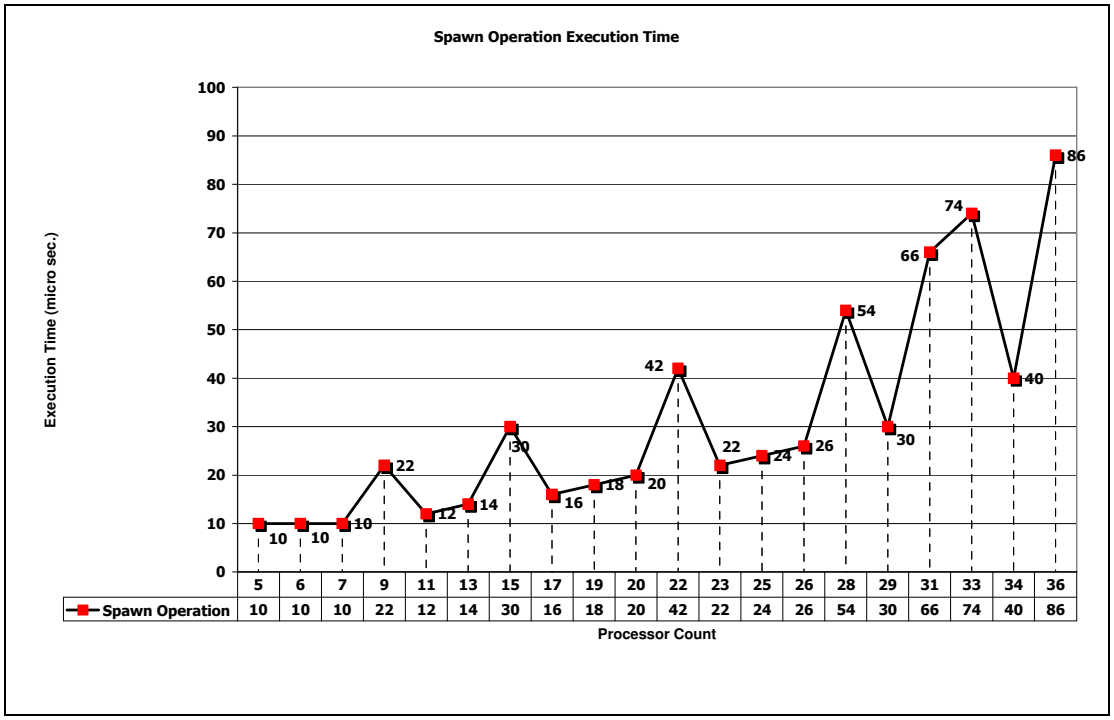


Figure 4.31 Spawn operation execution time

When the cost of the spawning operation is increased through the new spawning method, to minimize these costs, two spawn algorithms are proposed. These algorithms are based on the assumption that if a processor spawns it is more probable that it will spawn in the future. With this idea, the existing spawn pools are shared among processors with null spawn pointers. By this way, a more uniform distribution of spawn pools is accomplished and the number of processors with null spawn pointer values is decreased. Especially, *dividing spawn pool algorithm* is more cost effective, since after dividing the spawn pool between the spawn parent and the original processor, original processor continues to spawn as in case 1. Whereas, in *merging spawn pointers algorithm*, there appears an extra cost of traversing the binary tree and adjusting the spawn pointer values after each spawning operation. The worst case for these spawn algorithms occurs when the underlying assumption of these algorithms does not work as expected. In other words, consider the case with two processors A and B. A wants to spawn but it has null spawn pointer. Binary tree is traversed and B is chosen as the spawn parent and B spawns in place of A. Then, B's

spawn pool is divided among itself and A according to the *dividing spawn pool* algorithm. After that, A never spawns again but B continues to spawn several times. In such a case, by restricting the spawn pool of B to one half, in fact, something worse is accomplished.

Table 4.13 Spawn calls and binary tree costs

	Spawn Command	**Executed SC**	**New Nodes**	**Formula**	**Time**	**Proc. #**
1	S(0,1)	S(0,1)	4	s	10	5
2	S(0,1)	S(0,1)	8	s	10	6
3	S(0,1)	S(0,1)	16	s	10	7
4	**S(0,2)**	**S(0,1)**	32	2s+bt	22	9
		S(1,1)	6			
5	S(0,2)	S(1,2)	12, 24	s+bt	12	11
6	S(0,2)	S(2,2)	5, 10	s+2bt	14	13
7	**S(0,2)**	**S(2,1)**	20	2s+5bt	30	15
		S(3,1)	7			
8	S(0,2)	S(3,2)	14, 28	s+3bt	16	17
9	S(0,2)	S(4,2)	9,18	s+4bt	18	19
10	S(0,1)	S(5,1)	11	s+5bt	20	20
11	**S(0,2)**	**S(5,1)**	22	2s+11bt	42	22
		S(6,1)	13			
12	S(0,1)	S(6,1)	26	s+6bt	22	23
13	S(0,2)	S(7,2)	15,3	s+7bt	24	25
14	S(0,1)	S(8,1)	17	s+8bt	26	26
15	**S(0,2)**	**S(8,1)**	34	2s+17bt	54	28
		S(9,1)	19			
16	S(0,1)	S(10,1)	21	s+10bt	30	29
17	**S(0,2)**	**S(11,1)**	23	2s+23bt	66	31
		S(12,1)	25			
18	**S(0,2)**	**S(13,1)**	27	2s+27bt	74	33
		S(14,1)	29			
19	S(0,1)	S(15,1)	31	s+15bt	40	34
20	**S(0,2)**	**S(16,1)**	33	2s+33bt	86	36
		S(17,1)	35			

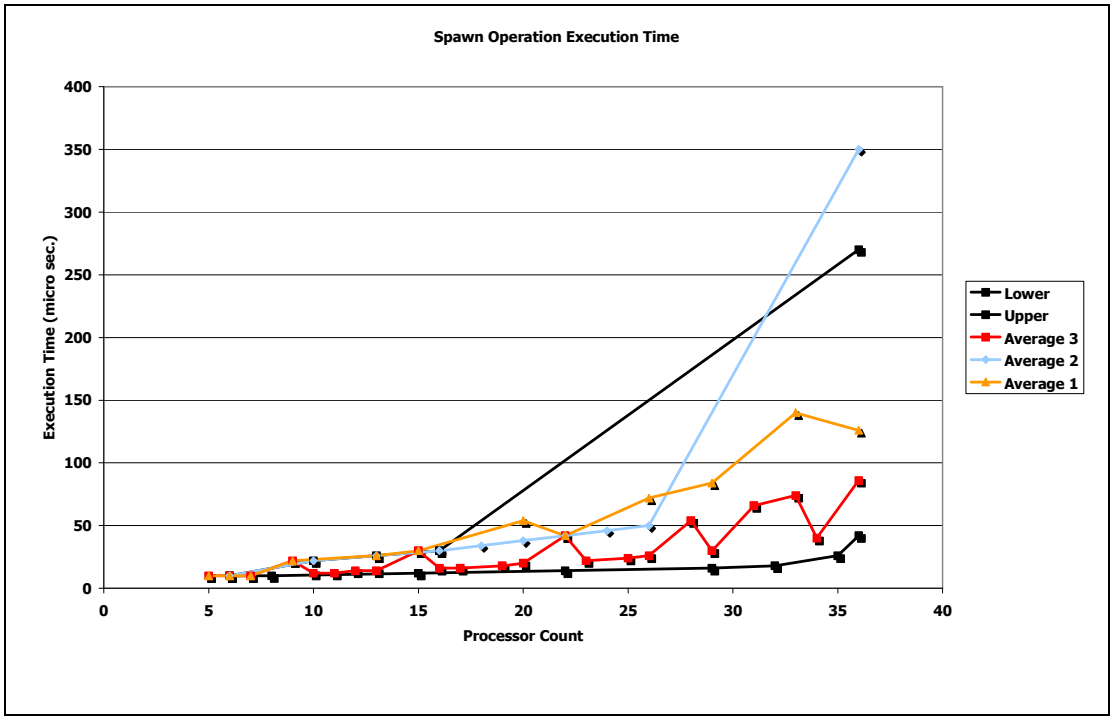


Figure 4.32 Several spawn operation sequences execution times

After all these comments, which route and which algorithm must be chosen in order to have a cost efficient spawning operation must be determined. As a route, it is feasible to choose routes like breadth first route and preorder route. When there are so many processors with null pointer values, it can be efficient to choose LRU route and continue with it, in order to bring the binary tree to a more uniform state. When LRU route is combined with *dividing spawn pool algorithm*, it is highly probable that good performance values are reached.

No matter which spawn route or spawn algorithm is used, the solution to the problem of falling off the tree is a more expensive operation than an ordinary spawning operation since it relies on inter-processor communication. If an application requires processors with spawning capabilities and if it is important to have disjoint sequences in each processor, then, this proposed solution can be used by paying the price of inter-processor communication cost.

CHAPTER 5

ENHANCEMENTS IN PARALLELIZING LCG WITH PRIME MODULUS

LCGs are commonly used since they are one of the oldest methods for RN generation and they are easy to compute. An LCG can be defined as LCG(a, b, m, X_0). Here, the most important parameter is modulus since its size constraints the period. There are two possibilities for the choice of modulus, power of two and prime. According to the type of modulus, parameterization methods of LCG differ. Details of LCG parameterization can be found in Section 3.3.2.1. In this thesis, LCG with prime modulus is implemented since LCG with power of two modulus is highly correlated on least significant bits.

When considering the parameterization of LCG, there appear two costly operations. First one is named as the initialization cost, finding the jth number relatively prime to m-1, arising from parameterization and the second one is the cost per RN generation resulting from the modular reduction. In order to reduce the cost per RN generation, Mersenne prime is used. Besides, as future work, using Sophie-Germain prime is stated as a different approach [10]. Since, it forms a completely different balance between initialization cost and RN generation cost. As part of this thesis, LCG parameterization is implemented with both Mersenne prime and Sophie-Germain prime.

When parallelizing LCG, splitting methods can also be considered as another alternative. It is well known that the structure of LCG is well suited to splitting methods like sequences splitting and leapfrog. When splitting methods are used, though the period lessens, the only cost encountered is the cost of RN generation. Unlike parameterized iteration there is no initialization cost. Although splitting methods are easy to implement and less costly than parameterization, they have well know defects as explained in Section 3.3.1. How will be the consequences if a method based on both splitting and parameterization methods is used for

parallelizing LCG? This question is asked for the first time, as a future research topic in [10]. In this thesis, such a hybrid method is implemented for LCG that combines sequence splitting with parameterized iteration. The effects of this hybrid method on the costs of LCG are tried to be determined.

In this chapter, firstly, the implementation details of parallel LCG with prime modulus are given. Then, both Mersenne prime and Sophie-Germain prime implementations are compared from different perspectives with several graphics. Lastly, the algorithmic details of hybrid method are explained together with its effects on performance.

5.1 Parallel LCG Implementation

The costliest tasks of parallel parameterized LCG are the cost of modular reduction which is defined as the cost per RN generation and the initialization cost arising from the parameterized iteration. In order to reduce these costs, special types of primes must be chosen. Two candidates are Mersenne prime and Sophie-Germain prime.

In 1999, Jerome Solinas introduced families of moduli called the generalized Mersenne numbers and showed a small weight prime moduli which is suitable for Mersenne modular reduction of the form $p = 2^k - 1$ where k is prime [14]. It is shown that such p's lead to fast modular reduction methods which use only a few integer additions and bitwise shifting. This technique is quite useful in practice, since it makes possible to implement long integer modular arithmetic without using multiple precision operations [10]. It becomes optimal to choose a Mersenne prime as modulus.

When modulus is not so high, it is reasonable to ask if the reduced cost of modular reduction obtained when using a Mersenne prime is balanced by the increased cost required in computing the jth number relatively prime to m-1 during initialization. If the number of RNs required per processor is large, reduced cost per RN is preferred to a stiff initialization cost. However, in highly branched MC computations, one often uses only a few hundred to a few thousand RNs before branching. Thus, one should consider other schemes that have different balance between the cost per RN and the initialization cost. A possible approach is to consider using Sophie-German primes instead of Mersenne primes as modulus [10].

Sophie Germain prime is a prime of the form m = 2q + 1 where q itself is prime. This special prime, has an explicit enumeration of the primitive elements modulo m. The price payed for this explicit enumeration is having to use standard modular multiplication [10].

The parameterization algorithm of the parallel LCG implementation is given in Figure 5.1 with MASTER SLAVE interaction. In order to parallelize LCG with parameterization, firstly, MASTER program receives input from user. Then, according to the type of modulus, modulus is calculated. After that, a multiplier value that satisfies the conditions for maximum period is determined. As explained in Section 2.2, in order to have a maximum period, multiplier must be primitive element modulo m. For a number a to be primitive element modulo m it must satisfy Formula 5.1 for all prime divisors q of m.

$$a^{m-1/q} = 1 (\text{mod } m) \tag{5.1}$$

After calculating multiplier value, maximum number of processors with disjoint cycles is calculated. LCG parameterization is accomplished by finding the jth number relatively prime to m-1 for the jth processor. So, maximum number of processors can not be higher than the number integers that are relatively prime to m-1. This value is represented as Φ(m-1) and called as Euler Phi Function. It is easily calculated by the Formula 5.2.

$$\Phi(m-1) = (m-1) * \Pi (1 - 1/p) \text{ for all prime divisors p of m-1.} \tag{5.2}$$

After calculating upper limit, global seed value is calculated. Then, all these initial data are sent to the newly created child processor together with binary tree node number through PVM calls. After receiving these initial data and the node number, each processor begins to make the relevant computations related with LCG paralelization. Only after these computations are carried out truly, the processor becomes ready for RN generation.

Parameterization operation starts when processor receives initial data from MASTER processor. Assume there is a processor with node number j. First of all, processor j finds the jth number relatively prime to m-1. This operation is named as enumeration operation. The method for finding the jth number differs by the type of modulus. When Mersenne prime is

used, it is defined as *Mersenne Enumeration*. Whereas, for Sophie-Germain prime, it is named as *SP Enumeration* and it is easier since, because of its nature, there exists an explicit enumeration for finding the jth number relatively prime. After jth number relatively prime to m-1 is computed by the jth processor, multiplier for the jth processor is calculated by the Formula 5.3, where l_j is the jth number relatively prime to m-1 and a_j is the multiplier value of the jth processor. After computing the multiplier, parameterization operation ends. Now processor j is ready to create an RN sequence that is guaranteed to be disjoint from other processors' sequences.

$$a_j = a^{l_j} \pmod{m} \tag{5.3}$$

When the parameterization algorithm in Figure 5.1 and the Formula 2.1 is considered, it is seen that, while generating RNs or when parameterizing an LCG, modular arithmetic operations like modular reduction, multiplication and exponentiation are heavily used. Basically, it can be stated that, for a parallel parameterized LCG, not only an enumeration algorithm is needed but also, there must be efficient modular reduction, modular exponentiation and modular multiplication algorithms. In the following two sections, the details of enumeration algorithms and the details of algorithms used for modular arithmetic operations are given with respect to the type of modulus chosen.

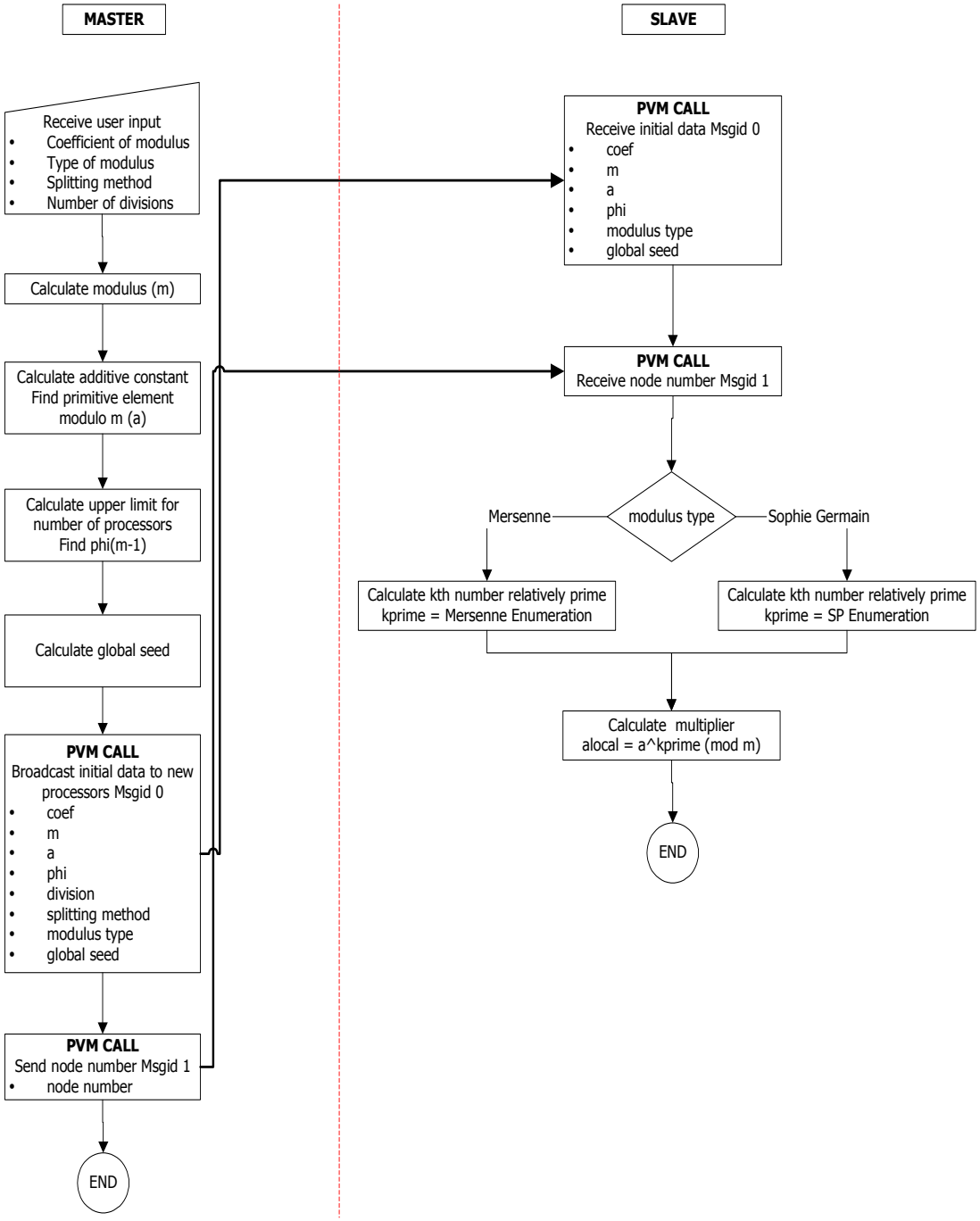


Figure 5.1 Parameterization algorithm of parallel LCG

5.1.1 Enumeration Algorithms

Enumeration algorithm or in other words, the algorithm for finding the jth number relatively prime to m-1 is the foundation of LCG parameterization via multiplier. The cost of this operation is referred as the initialization cost and its performance depends highly on the type of the modulus chosen. The efficiency of an enumeration algorithm directly affects the performance consequences of a parallel LCG implementation. For the two choices of prime moduli, Mersenne prime and Sophie-Germain prime, algorithms for enumeration operation differ considerably having completely different performance consequences.

For LCG with Mersenne prime, an efficient algorithm for finding the jth number relatively prime to m-1 can be found in [10]. In this thesis, Mersenne enumeration is accomplished that is based on linear search via finding GCD (Greatest Common Divisor). GCD operation is implemented according to Euclid's GCD algorithm. Euclid's algorithm is an efficient way to find the GCD of two numbers a and n, given a is less than n, it is based on the fact that if a and b have divisor d then so does a-b, a-2b and so on. C code of Euclid's algorithm can be seen in Table 5.1. Because of GCD computation of several times, Mersenne enumeration, does not have good performance when compared with Sophie-Germain enumeration.

Table 5.1 Euclid's GCD

```
int GCD(int n1, int n2)
{
   int gcd;
   while(n1%n2!=0)
   {
      gcd = n1%n2;
      n1=n2;
      n2=gcd;
   }
   return gcd;
}
```

Sophie-Germain prime is a prime of the form m = 2q + 1 where q itself is prime as explained before. In this case m – 1 = 2q so the integers that are relatively prime to m-1 are all the odds except q. Because of this explicit enumeration, Sophie-Germain enumeration has better performance than Mersenne Prime enumeration. On the other hand, Mersenne prime has an efficient modular reduction algorithm which is called as *Mersenne Reduction*. The details of Mersenne reduction together with other modular arithmetic algorithms are given in the next section.

5.1.2 Modular Arithmetic Algorithms

Parameterization of LCG and its iterative scheme for RN generation is based on modular arithmetic operations like modular reduction, modular multiplication and modular exponentiation. While calculating the multiplier according to the Formula 5.3, modular exponentiation is used. Moreover, creating RNs with Formula 2.1 depends on modular multiplication operation. That is why, special algorithms are used for computing these operations efficiently with integer arithmetic on 32 bit systems.

5.1.2.1 Mersenne Reduction Algorithm

Mersenne prime is a special type of prime number represented as $p = 2^k - 1$ where k is prime. Such prime numbers are known to have an efficient modular reduction algorithm where reduction is accomplished with only addition and shifting operations. When compared with normal reduction operation where division is used, it is clear that Mersenne reduction algorithm has better performance.

Table 5.2 Mersenne reduction

```
uint MersenneReduction(unit number, uint mprime, uint coef)
{
  unsigned int i, itot, old;
  old=number;
  itot=0;
  if(number < mprime)
    return number;
  else
    if(number == mprime)
      return 0;
  else{
    do{
      itot=0;
      do
      {
        i = old;
        i = i & mprime;
        old = old >> coef;
        itot = itot + i;
      }
      while(old!=0);
      old = itot;
        }
    while(old>mprime);
    return old;
        }
}
```

In Mersenne reduction algorithm, number to be reduced is separated into two parts; k lower bits where k is the Mersenne coefficient and y higher bits which begin at bit position k. Finally the reduction happens by adding the y higher bits after shifting them right to index 0 to the lower k bits and repeating this until no higher bits are left. If the result equals to Mersenne prime then it is set to zero. The C code of the algorithm can be found in Table 5.2. Other modular arithmetic operations like modular multiplication and modular exponentiation can be implemented by using special multiplication or exponentiation algorithms in combination with a modular reduction algorithm. In the following section, such a special algorithm and its variation according to modular reduction operation is explained.

5.1.2.2 Russian Peasant Algorithm

While computing local multiplier during parameterizaton and while generating RNs, modular exponentiation and modular multiplication operations are used. In order to accomplish these operations, an approach based on the *Russian Peasant method* is used. Russian Peasant algorithm is one of the earliest algorithms that have been discovered and it is for multiplication and exponentiation operations. This algorithm is also utilized in [23] while implementing a random number generator package with splitting techniques.

Table 5.3 Modular multiplication

```
uint RussianMul(uint n, uint x, uint m, uint coef, char type)
{ unsigned int P;
  while ((n & 1) == 0)
  { if(type=='m')
      x = MersenneReduction(x<<1, m, coef);
    else
     x= NormalReduction(x<<1, m, coef);
    n >>= 1;
  }
  P = x;
  n >>= 1;
  while (n > 0)
  { if(type==m)
     x = MersenneReduction(x<<1, m, coef);
    else
     x= NormalReduction(x<<1, m, coef);
    if ((n & 1) != 0)
    { if(type=='m')
       P = MersenneReduction(P+x, m, coef);
      else
       P = NormalReduction(P+x, m, coef);
     }
     n >>= 1;
  }
  return P; }
```

The fundamental definition of exponentiation, for positive integral exponents is defined by Formula 5.4.

$$x^n = x \times . \times x \text{ (n times)} \quad (5.4)$$

If that definition is directly turned into code, that would be wasteful since an algorithm based directly on that definition would use nine multiplications to compute x^{10}. Instead, since $x^{10} = ((x^2)^2)^2 \times x^2$, x can be multiplied by itself to obtain x^2, multiply x^2 by itself to obtain x^4 and then x^4 by itself to obtain x^8, and finally multiply x^8 by x^2 to obtain the result. By this way, x^{10} is computed in four steps instead of nine. This simple observation is the basis of the exponentiation algorithm of the Russian Peasant algorithm. In order to turn exponentiation into modular exponentiation, multiplication operations are done modulo m. The C code of the modular exponentiation operation is given in Table 5.4. As can be seen from Table 5.4, when Mersenne prime is used, modular multiplication operation with Mersenne reduction is utilized.

After considering both enumeration and modular arithmetic operations, it is understood that, there is not any prime modulus having good performance characteristics for all types of operations. Mersenne prime is good at modular arithmetic operations. Whereas Sophie-Germain prime has an efficient explicit enumeration for finding the jth number relatively prime to m-1. When this is the case, the problem is studied from a different perspective and instead of changing the modulus, a different approach based on changing the parallelization method of LCG is proposed as an alternative for adjusting performance consequences of LCG implementation.

Table 5.4 Modular exponentiation

```
uint RussianExp(uint n, uint x, uint m, uint coef, char type)
{
  int P;
  while ((n & 1) == 0) {
    x = RussianMul(x,x,m,coef,type);
    n >>= 1;              }
  P = x;
  n >>= 1;
  while (n > 0) {
    x = RussianMul(x,x,m,coef,type);
    if((n & 1) != 0)
       P = RussianMul(P,x,m,coef,type);
    n >>= 1;    }

  return P;
}
```

5.2 New Technique: Hybrid Method for Parallelization

When LCG parameterization is considered, not only is the period of the LCG a function of the modulus, but so is the total number of full-period LCGs. Since many branching MC computations require many available generators when using the binary tree mapping, large modulus is required in these situations to give deep binary trees. One drawback of this fact is that a large modulus is used for reasons other than the total number RNs needed in a particular computation. This is a clear weakness for parallel LCGs since the computational cost per RN increases as the number of processors needed increases. As a solution to this problem, a hybrid method that is based on both splitting and parameterization is implemented. By this way, number of available parallel processors is increased by using several subsequences from each full period cycle. How the sequence is divided into subsequences is determined by the splitting method used. This improvement allows the same number of parallel processors to be furnished with a smaller modulus, and thus it also speeds up the cost of computing individual RN [10].

New hybrid method extends parameterized iteration method further by adding two splitting methods, sequence splitting and leapfrog. This extension consists of three major additions. First one is the calculation of node number, second one is the assignment of seed values and the third one is the computation of the multiplier value for leapfrog method. Why such additions are needed and how they are implemented are explained in detail in the following paragraphs.

Algorithm of the hybrid method is shown in Figure 5.2. This method combines sequence splitting and leapfrog methods with parameterized iteration when parallelizing LCG. When compared with the algorithm in Figure 5.1, it is seen that steps that are shown in red in Figure 5.2, are added as a result of this combination. There appear two new fields that are crucial for this method, first one is for the splitting method and the second one is for the number of subsequences. Splitting method determines how a sequence is divided. Whereas, the number of subsequences, determines the maximum number of independent processors.

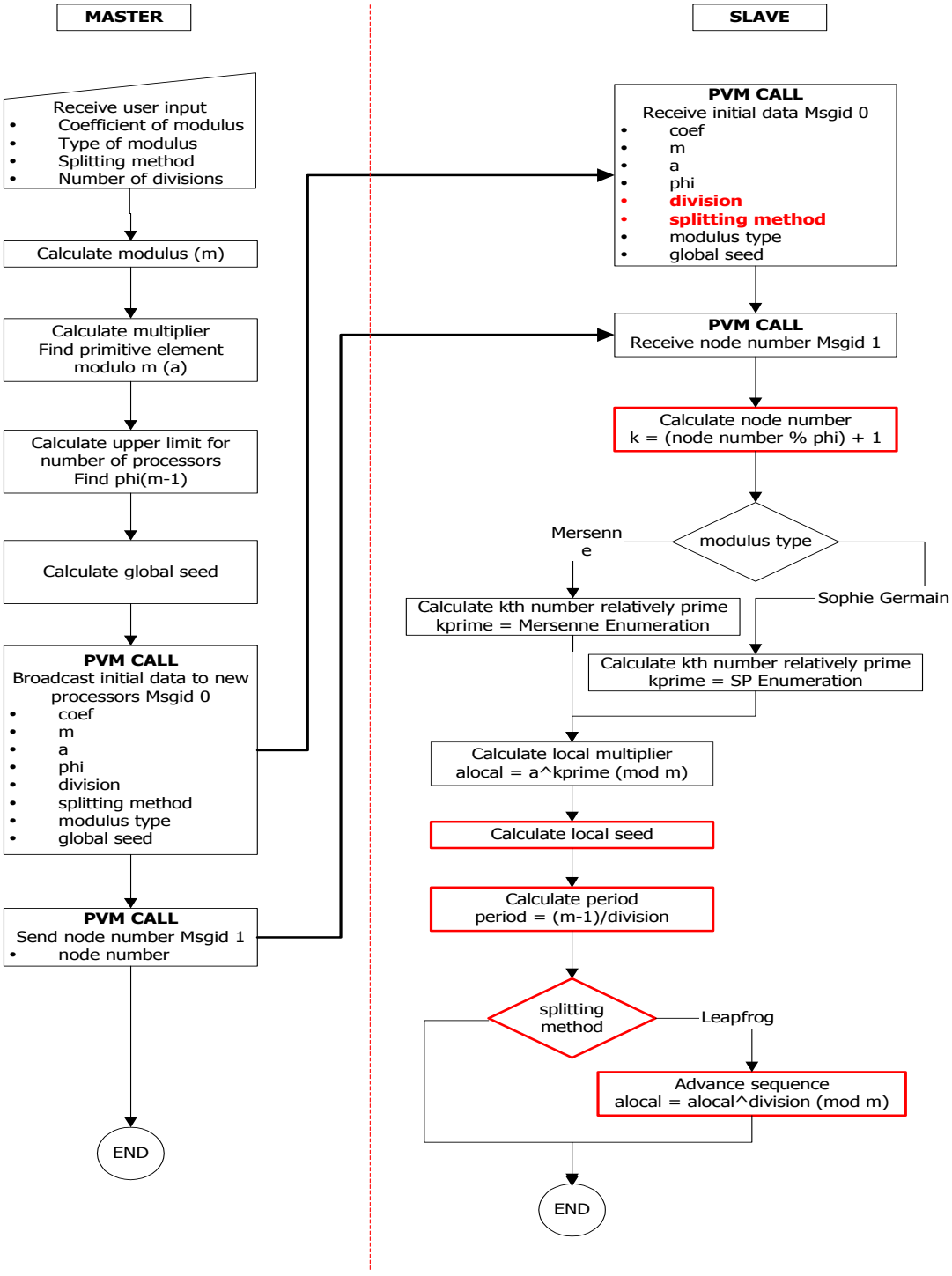


Figure 5.2 Parameterization algorithm of parallel LCG: Hybrid Method

According to the algorithm in Figure 5.2, after SLAVE processor receives initial data and node number from MASTER program, it calculates the node number in order to use it during enumeration operation. Recall that in the parameterization algorithm, operations are computed with respect to the binary tree node number. For instance, for a processor being in node 3 of the binary tree, 3^{rd} number relatively prime to m-1 is found and Formula 5.3 is applied for determining the multiplier. In the hybrid method, for a processor in node n of the binary tree, n value is converted by taking n MOD Φ(m-1). With this new value of n, multiplier is calculated. By making such an adjustment, nodes in the binary tree that are in the same remainder class of MOD Φ(m-1) have the same multiplier value thus, creating different parts of the same RN sequence.

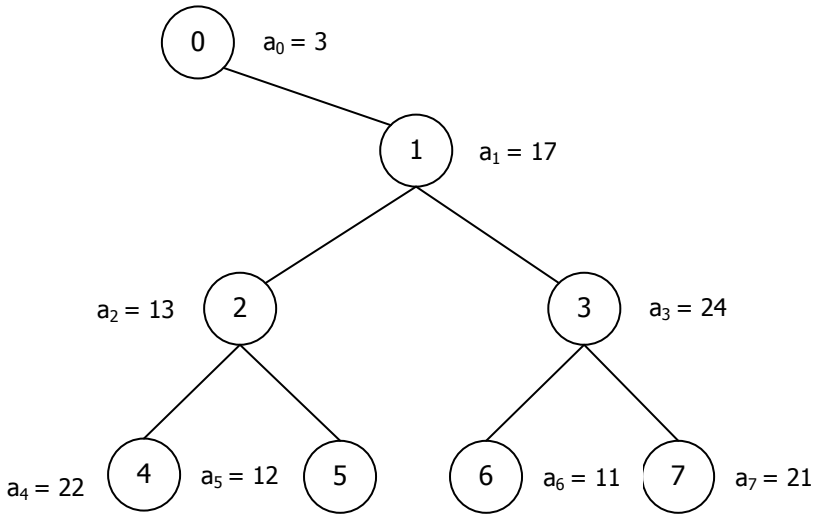


Figure 5.3 Parameterized iteration: LCG(3, 31, 5)

Consider the situation in Figure 5.3, for LCG(3, 31, 5) with Mersenne prime. When only parameterized iteration is used, there appears eight independent processors. On the other hand, when parameterized iteration is combined with a splitting method and a single disjoint sequence is divided into three subsequences, there can be at most twenty four independent

processors as seen in Figure 5.4. In fact, there are eight disjoint sequences of period thirty. By dividing an independent sequence into three parts, twenty four processors are obtained with period reduced to one third. Here, sequences are divided among processors in different divisions according to the splitting method chosen. In Figure 5.4, there are three divisions. That is why; a single sequence is divided among three processors in different divisions but in the same remainder class of MOD 8. Processors 0, 8 and 16 are such an example. They are in the same remainder class, creating the different parts of the same sequence. For the example situation in Figure 5.4, list of processor groupings is given in Table 5.5.

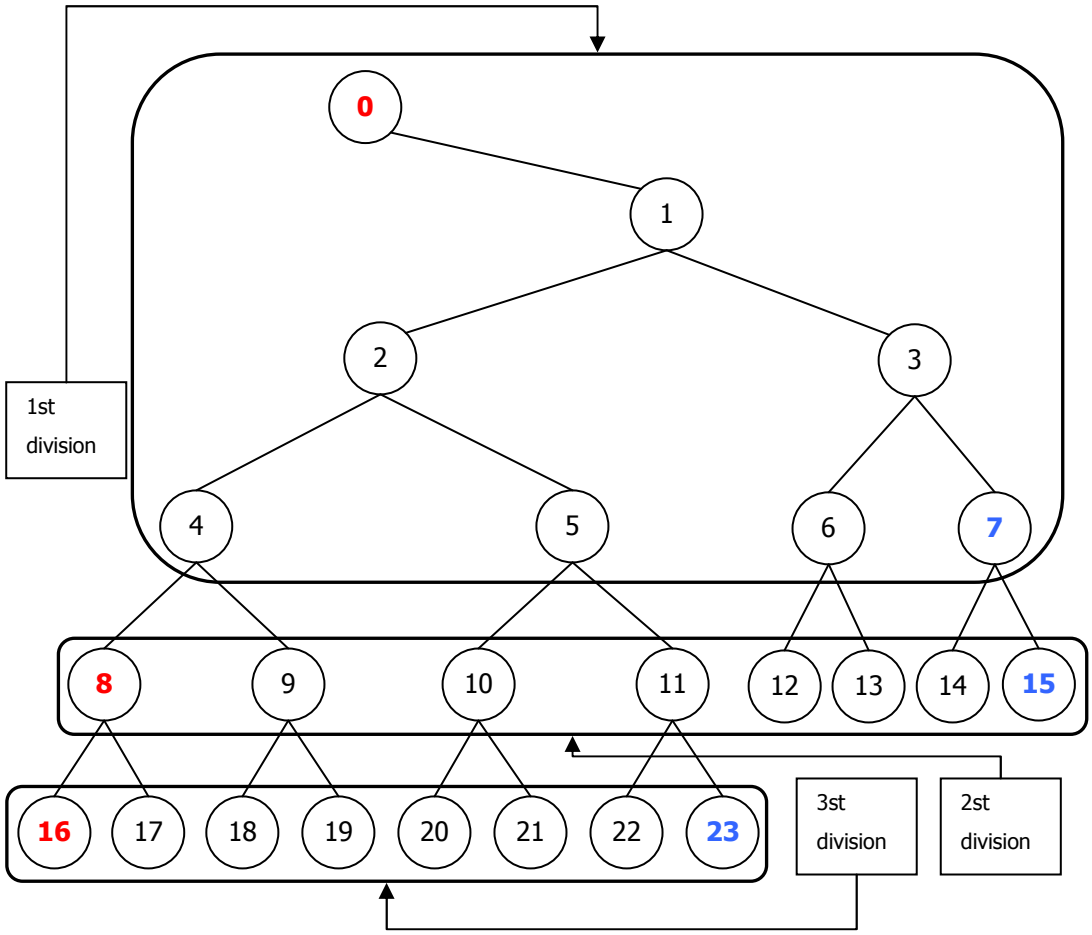


Figure 5.4 Parameterized iteration & splitting method LCG(3, 31, 5)

Table 5.5 Remainder class MOD Φ(m-1)

MOD Φ(30) = MOD 8	R	Multiplier	a_n
0 ≡ 8 ≡ 16	0	$a_0 = a^{l0} \pmod m$	3
1 ≡ 9 ≡ 17	1	$a_1 = a^{l1} \pmod m$	17
2 ≡ 10 ≡ 18	2	$a_2 = a^{l2} \pmod m$	13
3 ≡ 11 ≡ 19	3	$a_3 = a^{l3} \pmod m$	24
4 ≡ 12 ≡ 20	4	$a_4 = a^{l4} \pmod m$	22
5 ≡ 13 ≡ 21	5	$a_5 = a^{l5} \pmod m$	12
6 ≡ 14 ≡ 22	6	$a_6 = a^{l6} \pmod m$	11
7 ≡ 15 ≡ 23	7	$a_7 = a^{l7} \pmod m$	21

After calculating the multiplier value, each SLAVE processor computes its local seed to determine its starting point in the sequence. According to the type of splitting method used, seed creation differs. The algorithm for creating local seed in terms of different splitting methods can be seen in Figure 5.5. When sequence splitting is used, sequence is advanced P/D steps each time where D is the number of subdivisions and P is the period. The corresponding value is assigned as the seed. Whereas, in leapfrog method, the first D numbers are given as the seed values to corresponding processors.

When processors 0, 8 and 16 are considered, in order to divide the sequence between these processors by using sequence splitting method, seeds must be assigned by advancing the sequence. In sequence splitting, processor 0 gets the global seed value. For processor 8, sequence is advanced ten steps and the 10^{th} value in the sequence is given as the seed. Lastly, sequence is advanced twenty steps and the 20^{th} value in the sequence is assigned as the seed to processor 16. By such assignments, processor 0 creates the 1^{st} division of the whole sequence, processor 8 creates the 2^{nd} division and processor 16 creates the 3^{rd} division. When leapfrog method is used as the splitting method, processor 0 gets the first number in the sequence, processor 8 gets the second number and processor 16 gets the third number in the sequence as the seed value. In Table 5.6, seed assignment and sequence division for both Sequence splitting and leapfrog methods are given.

After seed assignment is completed, period reduction is done by each processor. Since overall sequence is divided among processors, the period of the subsequences lessen accordingly. If the overall period is thirty and if there are three divisions as in Table 5.6, then period of each processor is reduced to ten.

Processors become ready for RN generation after all these calculations are accomplished. In sequence splitting, RNs are computed by using the Formula 2.1. Whereas, in leapfrog, multiplier must be enhanced in a way that enables easy advancing of the overall sequence.

When using the hybrid method, choosing the splitting method also becomes important. The comparison of several splitting methods suitable for LCG is given in Section 3.3.1. According to the characteristics of the generator used, the most appropriate method can be chosen. When LCG is considered, it is seen that sequence splitting and leapfrog methods are the most suitable ones. As explained in Section 3.3.1, these methods differ only in the way of dividing the overall sequence between processors.

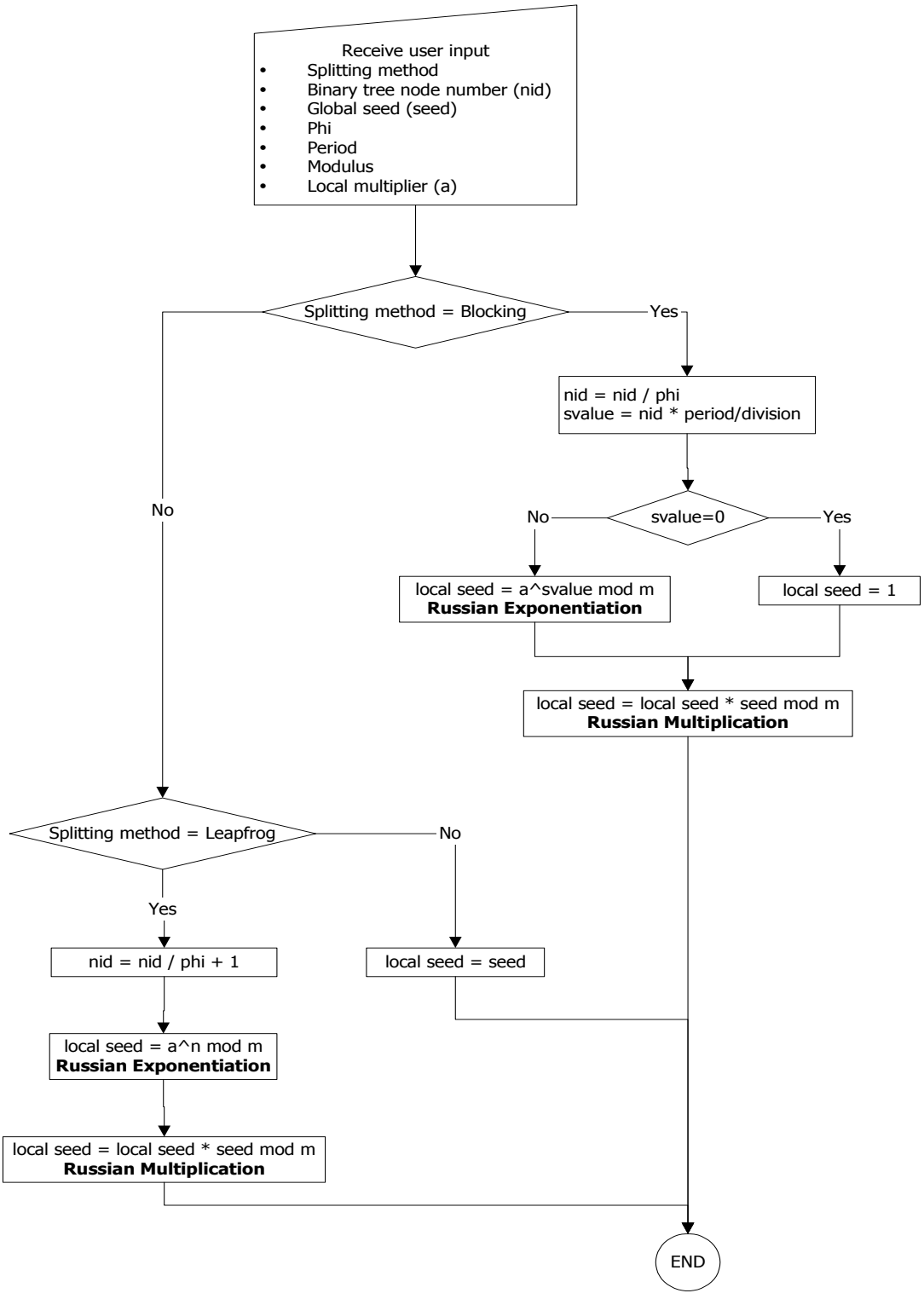


Figure 5.5 Flow chart of seed creation algorithm

Table 5.6 Splitting methods

Sequence Splitting					Leapfrog			
	1st	2nd	3rd			1st	2nd	3rd
15	15				**15**	15		
14	14				**14**		14	
11	11				**11**			11
2	2				**2**	2		
6	6				**6**		6	
18	18				**18**			18
23	23				**23**	23		
7	7				**7**		7	
21	21				**21**			21
1	1				**1**	1		
3		3			**3**		3	
9		9			**9**			9
27		27			**27**	27		
19		19			**19**		19	
26		26			**26**			26
16		16			**16**	16		
17		17			**17**		17	
20		20			**20**			20
29		29			**29**	29		
25		25			**25**		25	
13			13		**13**			13
8			8		**8**	8		
24			24		**24**		24	
10			10		**10**			10
30			30		**30**	30		
28			28		**28**		28	
22			22		**22**			22
4			4		**4**	4		
12			12		**12**		12	
5			5		**5**			5

To make the whole picture clear, two graphics of LCG with Mersenne prime, are presented that show the state of the sequence divided with both sequence splitting and leapfrog methods. In Figure 5.5, sequence generated by LCG with Mersenne prime of $2^7 - 1$ is shown together with its subsequences shared among three processors by the sequence splitting method. Overall sequence has a period of $2^7 - 2$. With sequence splitting method, it is provided that P0 creates the first 42 numbers, P1 creates the second 42 numbers, and P3 generates the third 42 numbers. In order to accomplish such divisions, P1 and P2 advance in the sequence and assign the last number to be created by the previous processor as their seed values. Unlike leapfrog method, sequence splitting needs advancing operation only during initialization while calculating the seed values.

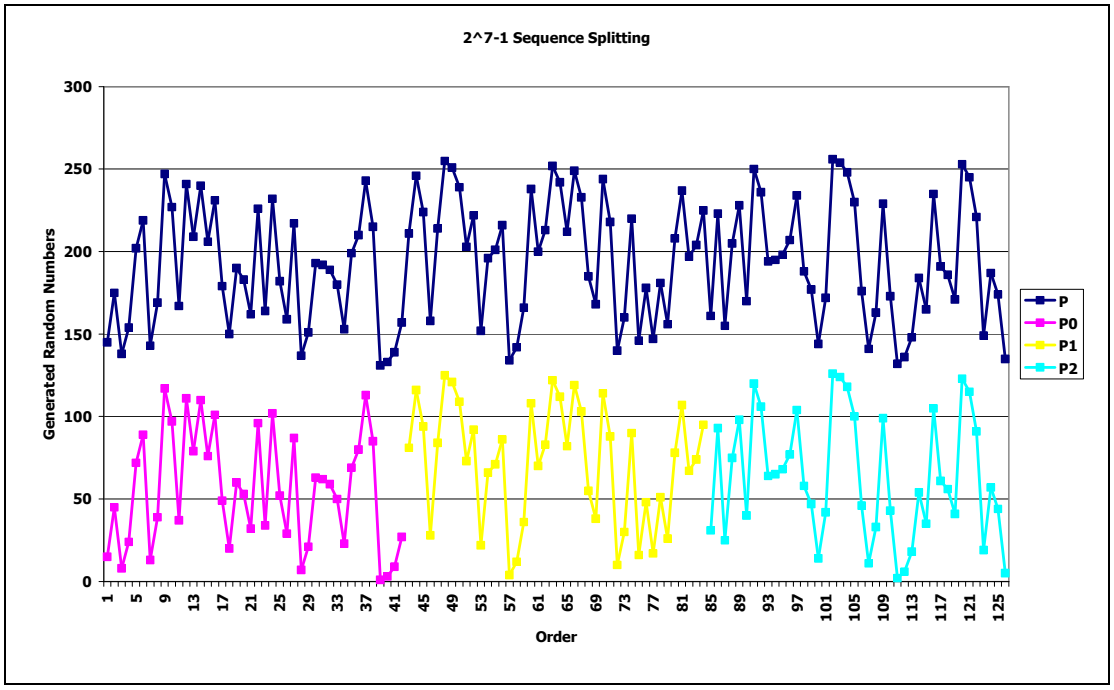


Figure 5.6 Mersenne prime sequence splitting

The subsequences generated by dividing the sequence using leapfrog method are shown in Figure 5.6. The overall sequence is created by LCG with Mersenne prime of $2^7 - 1$. Among the three processors, the sequence is divided in a manner that each processor creates numbers that are three places apart in the sequence. In order to accomplish such a distribution, the underlying RNG must have an easy to advance structure. For LCG, this is easily accomplished by changing the value of the multiplier in the iterative schema. The behaviour of the splitting method is independent from the type of moduli chosen for LCG as long as it is prime. Graphics for LCG with Sophie-Germain modulus, for both sequence splitting and leapfrog methods can be seen in Appendix C.

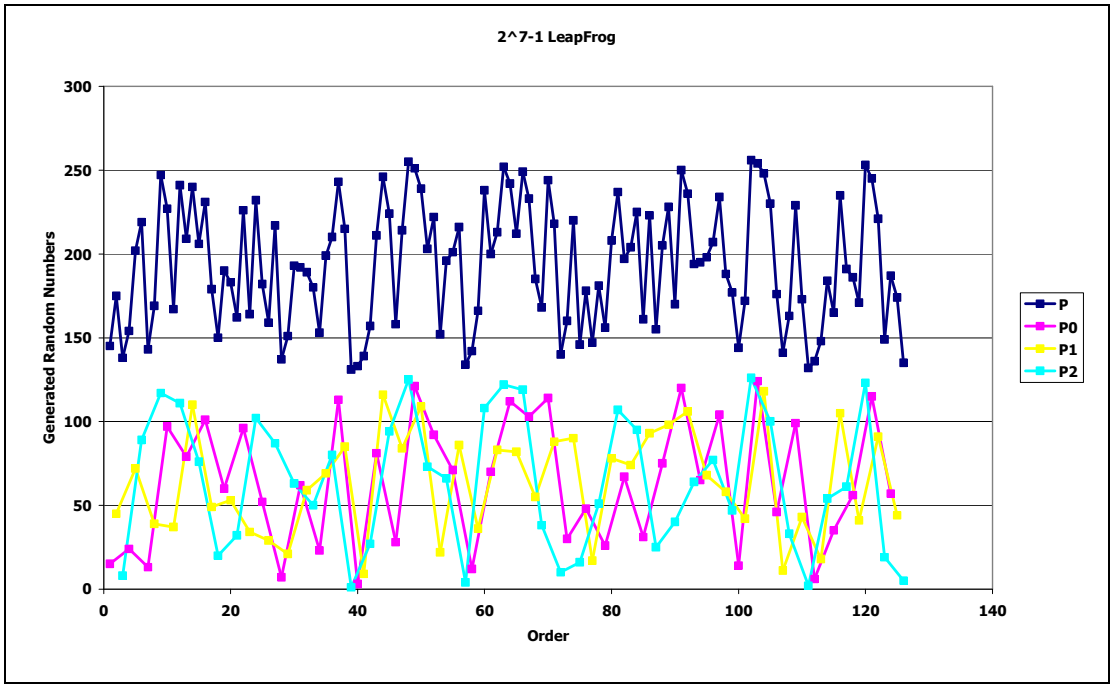


Figure 5.7 Mersenne prime leapfrog method

New hybrid method is based on the idea of increasing the upper limit for number of processors by reducing the period of each processor. This property of the method can be considered as its major drawback especially in situations where high amounts of RNs are

needed. But in other cases, where total number of RNs needed is not so high, this method can be preferred since with a smaller modulus, more processors can be spawned. This is a serious trade-off between having higher number of processors and having higher number of RNs. In the next section, all these newly proposed methods are analyzed case by case. The best method for each case is tried to be determined.

5.3 Analysis of Enhancements

In this thesis, in order to improve performance measures of LCG with prime moduli two different approaches are presented. In the first approach, the type of modulus is changed from Mersenne prime to Sophie-Germain prime. In the second approach, the type of parallization method is changed, instead of using parameterized iteration method, a hybrid method which combines two splitting methods with parameterized iteration method is used. In this section, the analysis of these approaches are given in terms of execution time via graphics. While comparing Sophie-Germain prime with Mersenne prime, speed up and efficiency graphics are also presented. Finally, in order to determine which approach suits best to which situation, these two approaches are examined case by case.

5.3.1 Sophie-Germain Prime vs. Mersenne Prime

Sophie-Germain prime, according to its nature, has an explicit parameterization when finding the kth number relatively prime to m-1. This enumeration operation can be accomplished by just checking if the number is odd which is done easily by bitwise operations. The execution time comparison of Mersenne prime and Sophie-Germain prime is given in Figure 5.8. For both Mersenne prime of $2^{31} - 1$ and Sophie-Germain prime of 2.147.483.579, several k values are chosen and the execution time for calculating these kth numbers relatively prime to m-1 are measured. The results are really interesting, since Sophie-Germain prime enumeration seems considerably faster than Mersenne prime enumeration. This great difference between Sophie-Germain and Mersenne prime implementations appears from the fact that, Mersenne prime enumeration linearly searches the sequence from 2 to m-2 by checking the greatest common divisor of each number with m-1. Whereas, Sophie-Germain enumeration depends only on checking if a number is odd or not through bitwise operators.

Another feature of Sophie-Germain prime is having a high Φ(m-1) value. The upper limit for the maximum number of independent processors increases considerably when Sophie-Germain prime is used. In the case of Sophie-Germain prime, there are situations where the upper limit for the number of processors is nearly two times higher than the Mersenne prime case, which means that nearly same amount of indepedent processors can be furnished by using a lower Sophie-Germain prime as modulus.

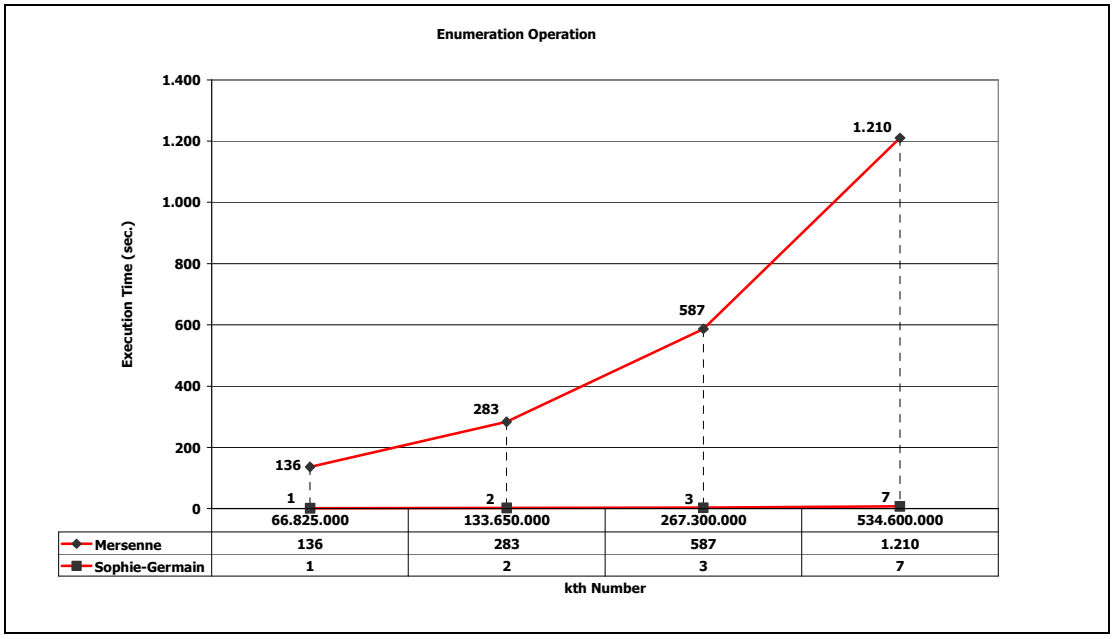


Figure 5.8 Execution times for enumeration operations

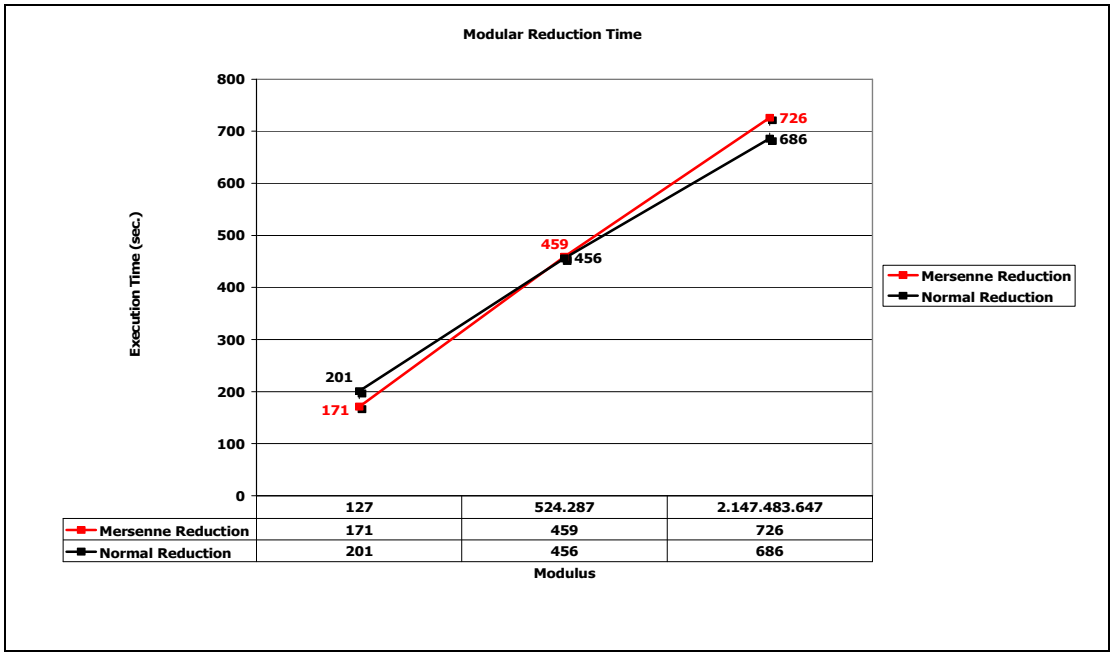


Figure 5.9 Execution times for modular arithmetic operations

Up to this point, only the superiorities of Sophie-Germain prime over Mersenne prime are considered. In order to possess these superior features of Sophie-Germain prime, what is paid as a price is using standard modular reduction. Unlike, Mersenne prime, Sophie-Germain prime does not have an efficient modular reduction algorithm. The execution time values of an LCG generating one billion RNs with both Mersenne prime and Sophie-Germain prime are presented in Figure 5.9. Three different Mersenne primes and Sophie-Germain primes are taken as modulus values. It is seen that for smaller modulus values, the execution time of Mersenne prime is better than Sophie-Germain prime. As the modulus gets higher, for both Mersenne prime and Sophie-Germain prime, RN generation time increases and by the time, the difference between these costs decreases. In practice, when the m approaches a few hundred bits in size, the cost of the shift and add modular reduction for a Mersenne prime is comparable to standard modular reduction. Thus it makes sense to consider using Sophie Germain primes when large moduli are needed [10].

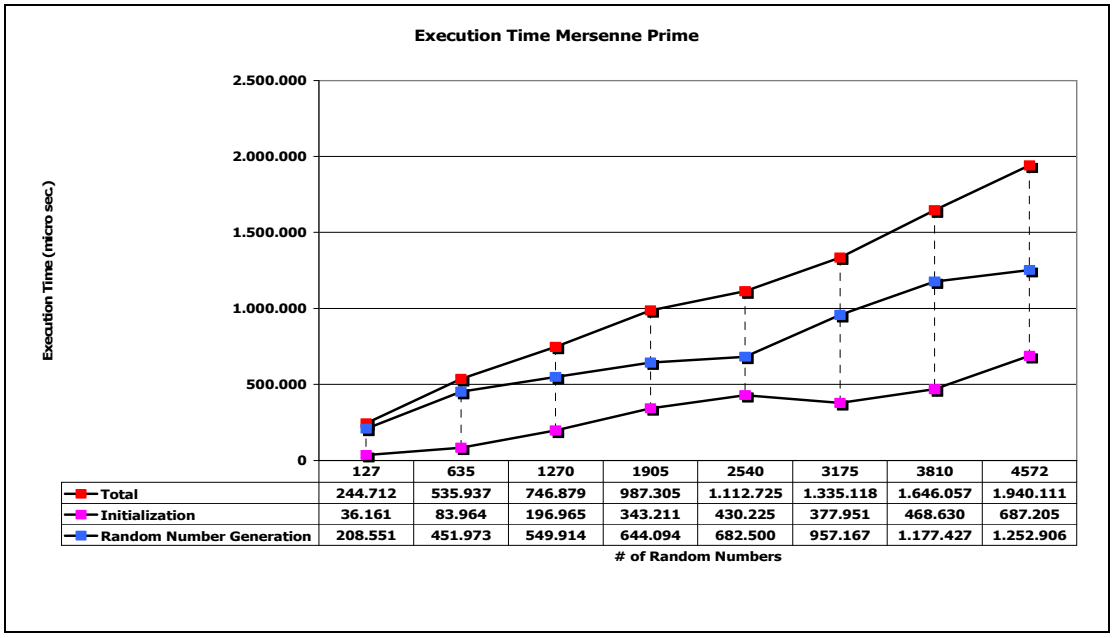
After comparing execution times of Sophie-Germain prime and Mersenne prime on operation basis, now it is time to see the overall cost of RN generation for each prime. The execution time of Mersenne prime for different amounts of RN is shown in Figure 5.10. Whereas, in Figure 5.11, the execution time of Sophie-Germain prime is shown. As can be seen, for Mersenne prime, initialization cost is high, RN generation cost is low. Whereas, for Sophie-Germain prime, initialization cost is low, RN generation cost is high. That is why; in total, cost of Mersenne and Sophie-Germain prime is very near.

In Figure 5.12, speed up graphic for both Mersenne prime and Sophie-Germain prime is given together, with respect to theoritical speed up value. Speed up is calculated by the Formula 5.5 where N is the number of processors. For one processor, execution time for E numbers is measured. Then for N processors, the execution time of each processor creating E/N numbers is measured. The ratio of these two measurement gives the speed up.

$$S = T1/TN \tag{5.5}$$

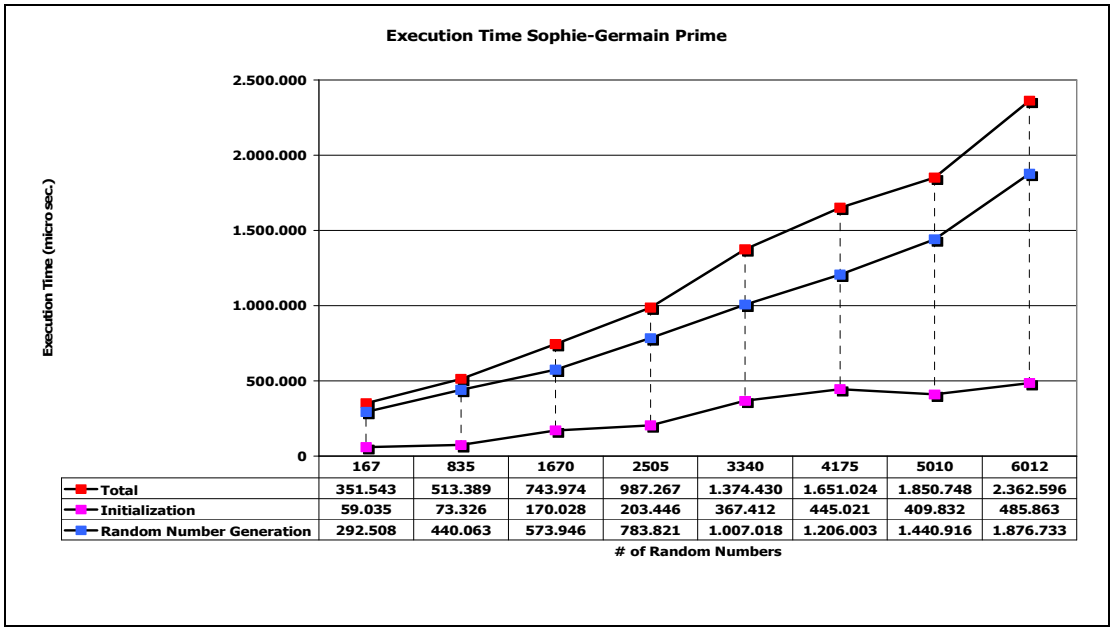
In Figure 5.13, only the speed up values of Mersenne and Sophie-Germain prime are displayed. Because of the initialization costs in more than one processor case, ideal speed up values are not reached for both types of prime moduli. Howover, as can be seen, Sophie-Germain prime has a slightly better speed up values than Mersenne prime since the initialization cost of Sophie-Germain prime is much lower than Mersenne prime. After calculating the speed up value, efficiency is calculated by the Formula 5.6. The effect of this decrease can also be seen in Figure 5.14 in terms of efficieny analysis.

$$E = 1/S \tag{5.6}$$



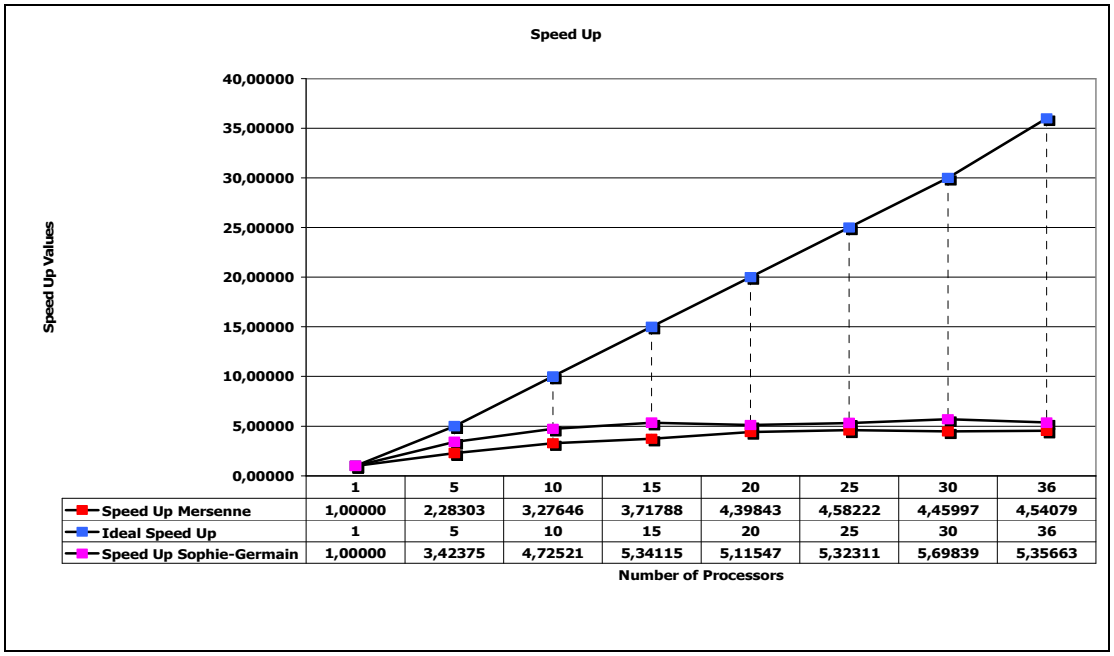
	127	635	1270	1905	2540	3175	3810	4572
Total	244.712	535.937	746.879	987.305	1.112.725	1.335.118	1.646.057	1.940.111
Initialization	36.161	83.964	196.965	343.211	430.225	377.951	468.630	687.205
Random Number Generation	208.551	451.973	549.914	644.094	682.500	957.167	1.177.427	1.252.906

Figure 5.10 Execution time for Mersenne prime



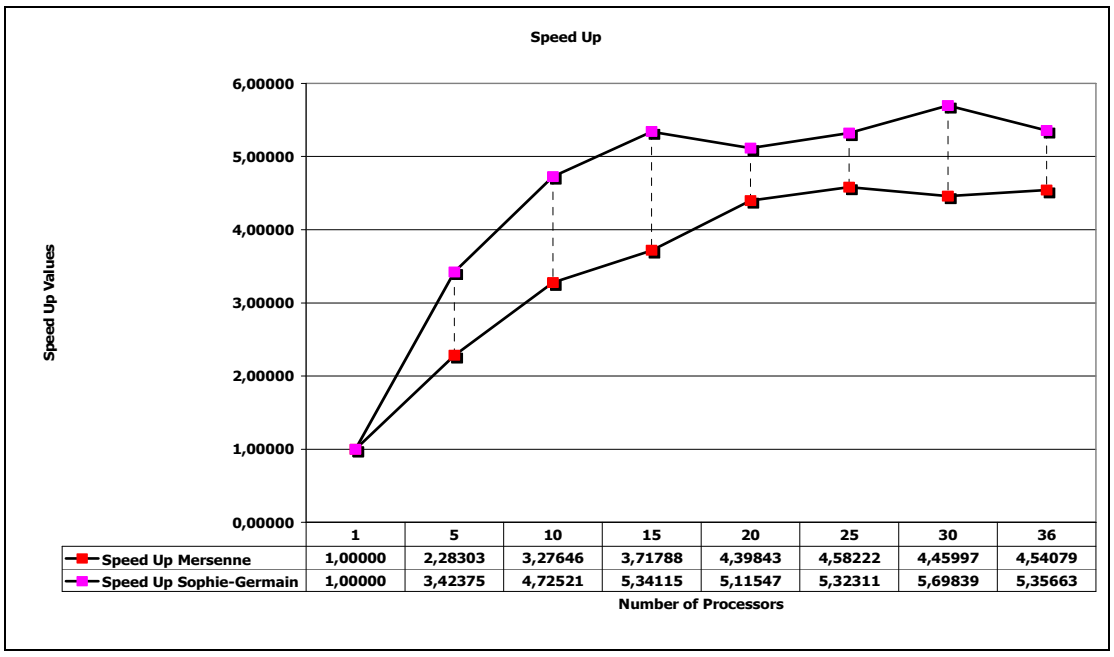
	167	835	1670	2505	3340	4175	5010	6012
Total	351.543	513.389	743.974	987.267	1.374.430	1.651.024	1.850.748	2.362.596
Initialization	59.035	73.326	170.028	203.446	367.412	445.021	409.832	485.863
Random Number Generation	292.508	440.063	573.946	783.821	1.007.018	1.206.003	1.440.916	1.876.733

Figure 5.11 Execution time for Sophie-Germain prime



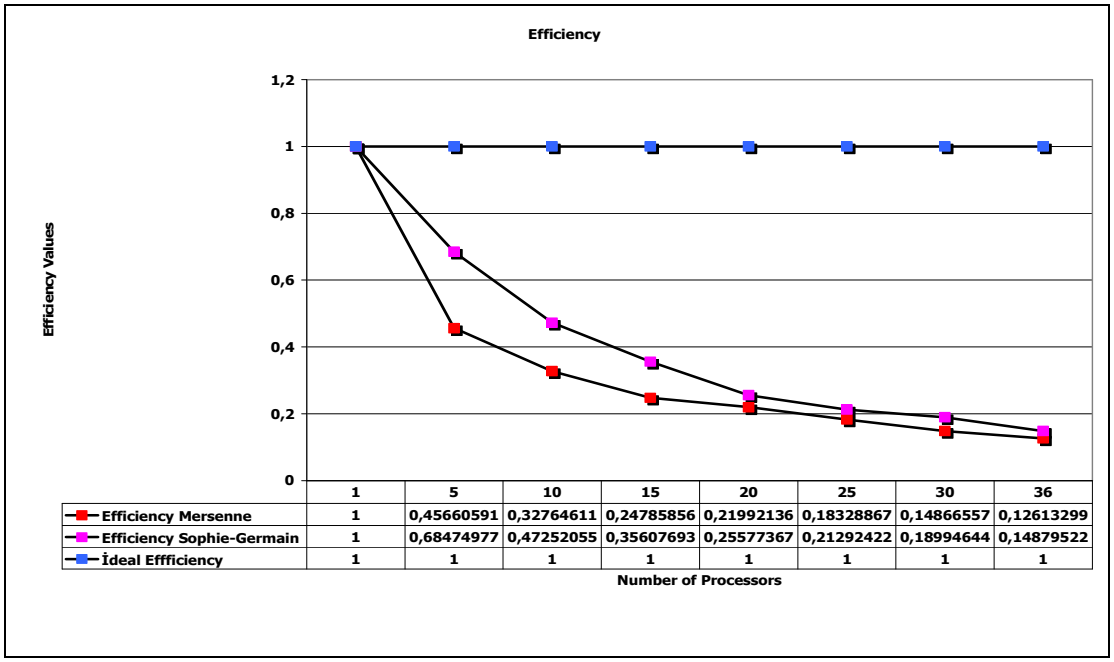
	1	5	10	15	20	25	30	36
Speed Up Mersenne	1,00000	2,28303	3,27646	3,71788	4,39843	4,58222	4,45997	4,54079
Ideal Speed Up	1	5	10	15	20	25	30	36
Speed Up Sophie-Germain	1,00000	3,42375	4,72521	5,34115	5,11547	5,32311	5,69839	5,35663

Figure 5.12 Speed up for Mersenne and Sophie-Germain primes



	1	5	10	15	20	25	30	36
Speed Up Mersenne	1,00000	2,28303	3,27646	3,71788	4,39843	4,58222	4,45997	4,54079
Speed Up Sophie-Germain	1,00000	3,42375	4,72521	5,34115	5,11547	5,32311	5,69839	5,35663

Figure 5.13 Speed up for Mersenne and Sophie-Germain primes



	1	5	10	15	20	25	30	36
Efficiency Mersenne	1	0,45660591	0,32764611	0,24785856	0,21992136	0,18328867	0,14866557	0,12613299
Efficiency Sophie-Germain	1	0,68474977	0,47252055	0,35607693	0,25577367	0,21292422	0,18994644	0,14879522
İdeal Effficiency	1	1	1	1	1	1	1	1

Figure 5.14 Efficiency for Mersenne and Sophie-Germain primes

5.3.2 Case Analysis

Two approaches are proposed in order to improve the performance consequences of an LCG with prime moduli. All of these methods have advantages over one another but none of them can be considered as applicable in all circumstances. In fact, according to the needs of the situation, the most appropriate approach must be chosen. For this purpose, in this section, four separate cases are determined and the best approach to be taken for each case is tried to be realized.

First case is defined as LCG used in an application where amount of RNs needed is high and the need for independent processors is low. When number of indenpendent processors is low, this means that enumeration operation is rarely used and has little effect on performance. On the other hand, since the RNs required is large, reduced cost per RN is desirable. The most suitable approach is using LCG with a Mersenne prime in order to make use of the cost reduction arising from the efficient modular reduction operation. When modulus is so large, using Sophie-Germain prime can be another alternative. But, this will not bring any performance gain, since enumeration operations are rarely utilized.

In the second case, LCG is used in an application where the need for RNs and independent processors are high. In such a circumstance, both RN generation cost and parameterization cost must be reduced. Using Sophie-Germain prime can be an alternative in order to reduce the cost of enumeration operations but this approach has no effect on the cost of RN generation. According to the acceptable performance thresholds of a particular application, decision between using Mersenne prime or using Sophie-Germain prime should be made.

Consider the third case as LCG utilized in an application where RNs needed per processor is low but several independent processors are required. Since, RNs needed is not so high, hybrid method can be used. Besides, in order to reduce the cost of enumeration operation, using Sophie-Germain prime can be preferred. According to the nature of Sophie-Germain prime, when combined with the hybrid method requires a very low modulus value in order to cover the requirements of the third case.

Last case is defined as LCG used in an application where the necessity for both RNs and independent processors are low. Since, RNs needed is not so high, hybrid method can be used together with Mersenne prime. Since, enumeration operation is seldomly used, by choosing Mersenne prime, the cost of RN generation can be reduced. Also, by using hybrid method, with a smaller modulus, required amount of processors and RNs can be created. Having a smaller modulus value also leads to a reduced cost of RN generation since it is shown that, Mersenne prime has better modular reduction performance for smaller modulus values. For these four separate cases, the best approach is tried to be determined. The summary of these cases can be found in Table 5.7.

As a last word, according to the performance consequences of LCG with prime moduli what can be stated is, every single application has its own requirements, in order to find the best approach, firstly, performance thresholds related with these requirements must be decided. Only after such determinations, the best approach for that specific application can be designated by taking into consideration the performance criteria for each approach.

Table 5.7 Four cases and best approaches

	Case	**Approach**
1	RNs needed = high Number of independent processors = low	Mersenne prime
2	RNs needed = high Number of independent processors = high	Sophie-Germain prime
3	RNs needed = low Number of independent processors = high	Sophie-Germain prime with hybrid method
4	RNs needed = low Number of independent processors= low	Mersenne prime with hybrid method

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, parallel RN generation is considered from two different perspectives. Firstly parallelization topology based on binary tree structure is taken into consideration and a solution to the problem of falling off the tree is given. Secondly, the performance consequences of parallel LCG with prime moduli are considered and several methods for improving the performance are given.

By the proposed solution to the falling off the tree problem, when a processor made a spawn request, no matter the state of its spawn pool, it is made possible to respond to that spawn request by traversing the binary tree structure and determining the new spawn parents. This spawning operation continues until the overall spawn pool of the generator vanishes. Since assignment is accomplished according to the rules of binary tree, it is guaranteed that each processor created has a completely different random behaviour from the other processors in the environment. Such an assignment brings up two additional costs to the spawning operation. First one is the cost of binary tree traversal and second one is the inter-processor communication arising from new spawn parent assignment. In order to reduce these costs, *dividing spawn pool* and *merging spawn pointer* algorithms are proposed. As discussed in Section 4.4, for different situations, the performance consequences of the proposed solution differ considerably. Especially, when dividing spawn pool algorithm is utilized, performance of the proposed solution gets better. If processors in an application tend to spawn continuously and if it is crucial that each processor has disjoint sequences, then, by bearing to some inter-processor communication, whole spawn pool of the generator can be appointed for the application. This is a trade-off between having independent sequences and having no inter-processor communication. The best decision can be given by considering the performance graphics given in Section 4.4 according to the requirements of the application.

The LCG with prime modulus is parallelized by an explicit parameterization of its multiplier. When this is the situation, there appear two different types of cost for an LCG. First one is the cost of parameterization and the second is the cost per RN generation. By choosing different types of prime moduli, it is possible to reduce these costs. But unfortunately, there are not any prime moduli, which have good performance characteristics for both parameterization and RN generation operations. In this thesis, performance of an LCG is discussed with respect to both Mersenne prime and Sophie-Germain prime. It is shown that, for some cases, Sophie-Germain primes show better performance than Mersenne primes. Whereas, Mersenne primes can not be left aside because they have high performance in modular reduction operations. For an LCG, having a high prime modulus is the cause of all costly operations. With the new hybrid method, because of its structure, same number of processors are provided with a lower modulus value. This decrease in modulus value, improves both the performance of parameterization and RN generation. Whereas, what is paid as a price is the considerable decrease in the period. As a result of all these investigations, it can be stated that there is not any method or any parameter which suits best for all types of applications. The choices must be made carefully with respect to the necessities of the underlying application.

As a future work in the area of parallel RN generation, the parallelization aspects of combined generators can be considered. They become preferable since they possess better quality consequences than the ordinary generators. Their parallelization schemes and their topological architectures can be further examined from performance and efficiency perspectives with respect to the requirements of PRNGs.

In the scope of this thesis, although, the theoretical information related with LFG is given, it is not implemented in PVM system. Implementation of LFG in PVM can be thought of as a future work. Besides, as Sophie-Germain primes are used as moduli, implementation of other types of special primes like Fermat primes can be considered. Although, Fermat primes are very rare, they are known to be very efficient in modular arithmetic operations like Mersenne primes.

REFERENCES

[1] Wikipedia The Free Encyclopedia, Random Numbers. http://en.wikipedia.org/wiki/Random_number, 2005 (Last accessed August 2005).

[2] Srinivasan A., Mascagni M., and Ceperley D., Testing Parallel Random Number Generators. *Parallel Computing*, 29:69-94, 2003.

[3] P. D. Coddington, Random number generators for parallel computers, http://www.npac.syr.edu/users/paulc/papers/NHSEreview1.1/, 1996 (Last accessed July 2005)

[4] Computational Science Education Project, Random Numbers. http://csep1.phy.ornl.gov/rn/rn.html, 1995 (Last accessed July 2005).

[5] P. L'Ecuyer, Random Numbers. In the *International Encyclopedia of the Social and Behavioral Sciences*, N. J. Smelser and Paul B. Baltes Eds., Pergamon, Oxford, 2002, 12735-12738.

[6] Computational Science Education Project, Introduction to Monte Carlo Methods. http://csep1.phy.ornl.gov/mc/mc.html, 1995 (Last accessed July 2005).

[7] Srinivasan A., Ceperley M. D., and Mascagni M., Random number generators for parallel applications. In D. Ferguson, J. I. Siepmann, and D. G. Truhlar, editors, *Monte Carlo Methods in Chemical Physics, Advances in Chemical Physic series*, Vol. 105, New York, John Wiley and Sons, pp 13-36 1999.

[8] Mascagni M., Theory and Software for Parallel Random Number Generation. *Proceedings of The Fourth International Conference on Supercomputing in Nuclear Applications* (SNA 2000), 2000.

[9] Mascagni M., Parallel Pseudorandom Number Generation. *SIAM News*, August, pp. 1,8-10, 1999.

[10] Mascagni M., Parallel Linear Congruential Generators with Prime Moduli. *Parallel Computing*, 24: 923-936, 1998.

[11] Pryor V. D., Cuccaro A. S., Mascagni M., and Robinson L. M., Implementation and usage of a portable and reproducible parallel pseudorandom number generator. In *Proceedings of Supercomputing `94*, IEEE, pp. 311-319, 1994.

[12] Parker J., Extensions and Optimizations to the Scalable, Parallel Random Number Generators Library. M.Sc. Thesis, Florida State University, 2003.

[13] Knuth E. D. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms Third Edition*, Addison Vesley, Reading, Massachusetts, 1998.

[14] Solinas A. J, Generalized Mersenne numbers. *Technical Report CORR 99-39*, Centre for Applied Cryptographic Research, University of Waterloo, 1999.

[15] Park S. K. and Miller K. W., Random number generators: good ones are hard to find. *Comm of the ACM* 31, 1192-1201, 1988.

[16] Mascagni M., Robinson L. M., Pryor V. D., and Cuccaro A. S., A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator. *J. Comput. Physics*, 15:211-219, 1995.

[17] Aluru S., Prabhu M. G., and Gustafson J., A random number generator for parallel computers. *Parallel Computing* Vol. 18, pp. 839-847, 1992.

[18] Foster Ian, Designing and Building Parallel Programs. http://www-unix.mcs.anl.gov/dbpp/, 1995 (Last accessed July 2005)

[19] Mascagni M., Parallel pseudorandom number generation. *SIAM News*, August, pp. 1,8-10, 1999.

[20] Mascagni M., SPRNG: A Scalable Library for Pseudorandom Number Generation. *Proceeding of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, MS11, March 22-24, 1999.

[21] Mascagni M., Serial and Parallel Random Number Generation. In *Quantum Monte Carlo in Physics and Chemistry*, P. Nightingale and C. Umrigar, editors, Springer-Verlag: New York, Berlin, pp. 277-288, 1999.

[22] Brent P. R., On the periods of generalized Fibonacci Recurrences. In the Pess, *Math. Comput.* 1994.

[23] L' ecuyer P., Cote S., Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, Vol 17, no, 1, March 1991.

APPENDIX A

FINDING PRIMITIVE ROOT MODULO M

In this section, finding primitive root modulo m is shown by an example.

Corollary A.1: Let r be a primitive root modulo m where m is an integer m > 1. Then r^u is a primitive root modulo m if and only if u and m are relatively prime.

Theorem A.1: The integer 7 is a primitive root of $2^{31} - 1$.

Proof A.1:

To show that 7 is a primitive root of $2^{31} - 1$, show that, $7^{(2^{31} - 2)/q} \neq 1 \pmod{2^{31} - 1}$ for all prime divisors q of $2^{31} - 2$.

To find factorization of $2^{31} - 2$, $2^{31} - 2 = 2(2^{30} - 1)$

$$= 2(2^{15} - 1)(2^{15} + 1)$$

$$= 2(2^5 - 1)(2^{10} + 2^5 + 1)(2^5 + 1)(2^{10} - 2^5 + 1)$$

$$= 2.3^2.7.11.31.151.331$$

if it is shown that $7^{(2^{31} - 2)/q} \neq 1 \pmod{2^{31} - 1}$ for q = 2,3,7,11,31,151,331, then it is known that 7 is a primitive root of $2^{31} - 1 = 2147483647$. Since,

$7^{(2^{31} - 2)/2} \neq 2147483546 \neq 1 \pmod{2^{31} - 1}$

$7^{(2^{31} - 2)/3} \neq 1513477735 \neq 1 \pmod{2^{31} - 1}$

$7^{(2^{31} - 2)/7} \neq 120536285 \neq 1 \pmod{2^{31} - 1}$

$7^{(2^{31}-2)/11} \neq 1969212174 \neq 1 \pmod{2^{31}-1}$

$7^{(2^{31}-2)/31} \neq 512 \neq 1 \pmod{2^{31}-1}$

$7^{(2^{31}-2)/151} \neq 535044134 \neq 1 \pmod{2^{31}-1}$

$7^{(2^{31}-2)/331} \neq 1761885083 \neq 1 \pmod{2^{31}-1}$

In practice the primitive root 7 is not used as the multiplier, since the first few integers generated are small. A larger primitive root is found by taking a power of 7 where the exponent is relatively prime to $2^{31} - 2$ (as stated in Corollary A.1). For instance, since GCD(5, $2^{31} - 2$) = 1, Corollary A.1 tells us that $7^5 = 16807$ is also a primitive root. Since GCD(13, $2^{31} - 2$) = 1, another possibility is to use $7^{13} = 252246292 \pmod{2^{31} - 1}$ as the multiplier.

APPENDIX B

FLOW CHARTS FOR INITIALIZATION AND SPAWNING OPERATIONS

In this section, the flow charts of initialization and spawning operations are given. In Figure B.1, initialization algorithm together with PVM calls is given. Moreover, in Figure B.2 and Figure B.3, the details of the spawning operation are displayed. In Figure B.2, spawning operation is considered from PVM calls perspective while in Figure B.3, the focus is the effects of the spawning operation on binary tree structure.

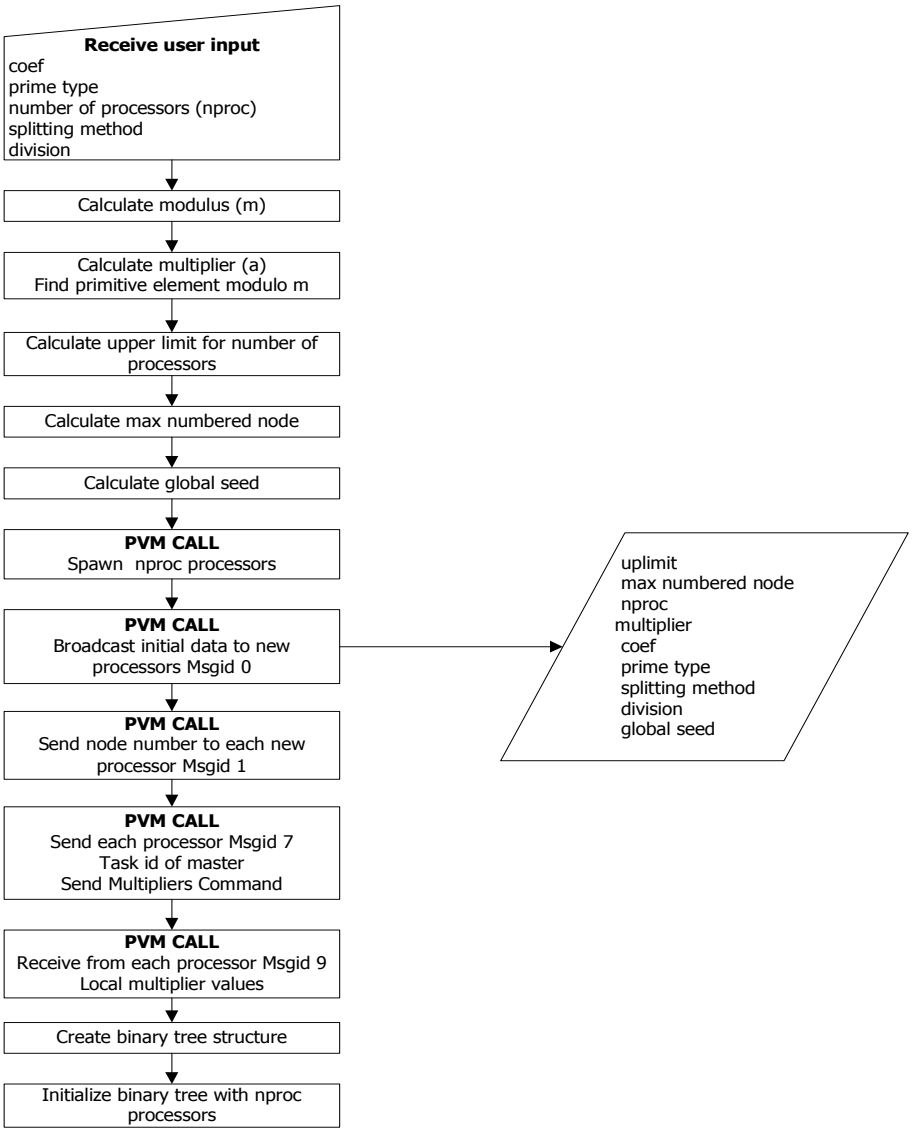


Figure B.1 Flow chart of initialization operation with PVM calls

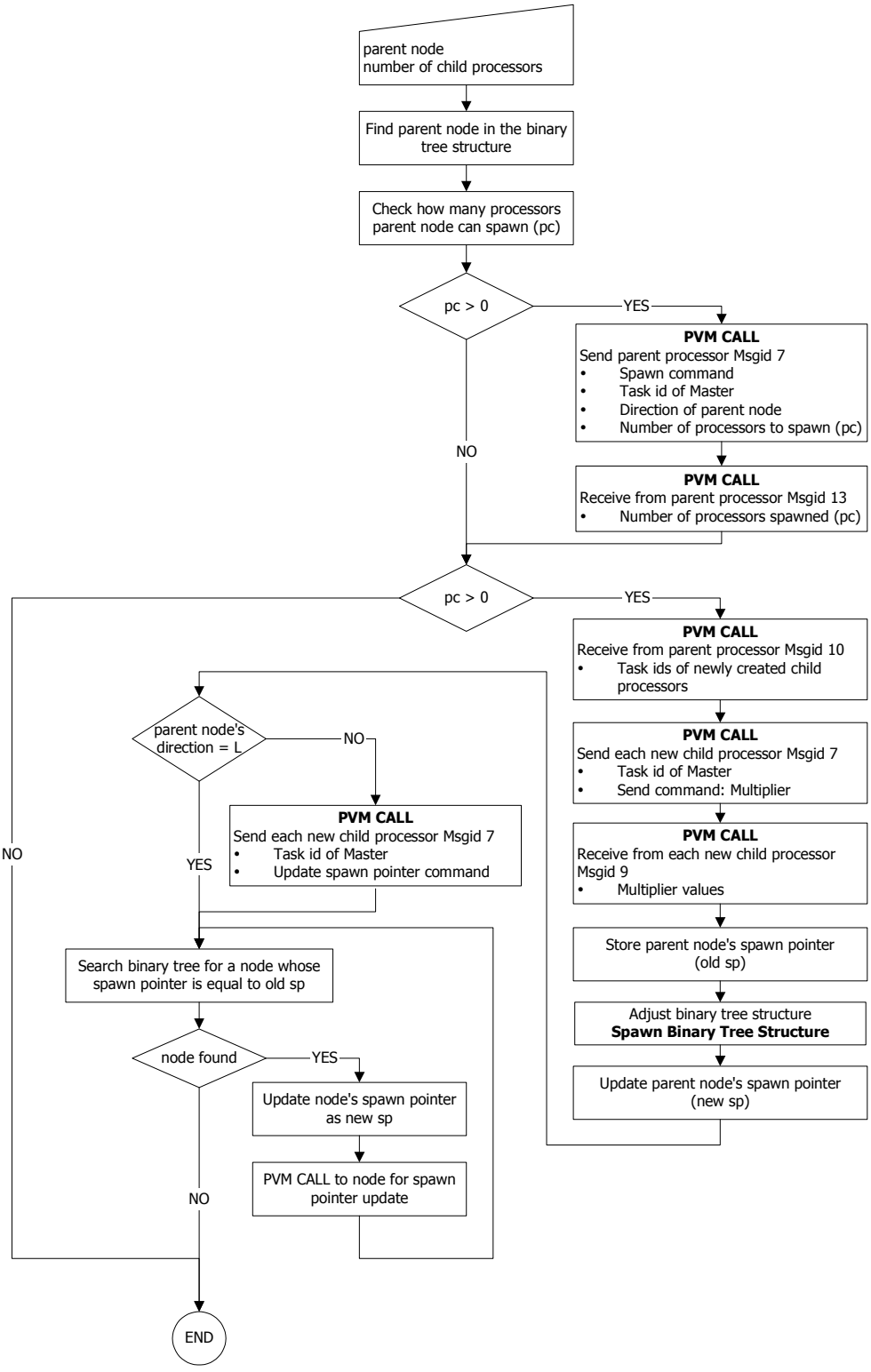


Figure B.2 Flow chart of spawn operation with PVM calls

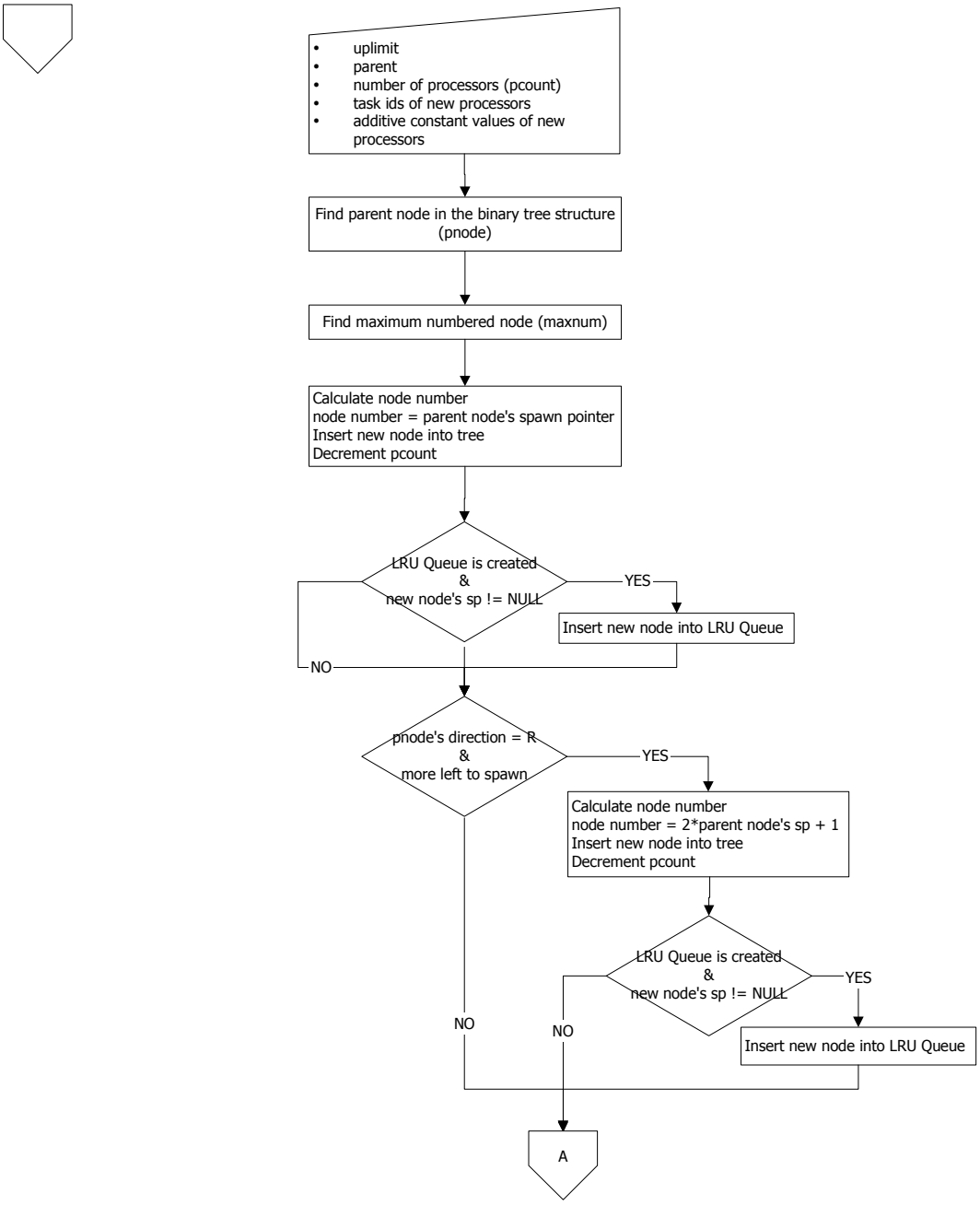


Figure B.3.1 Spawn operation on binary tree first part

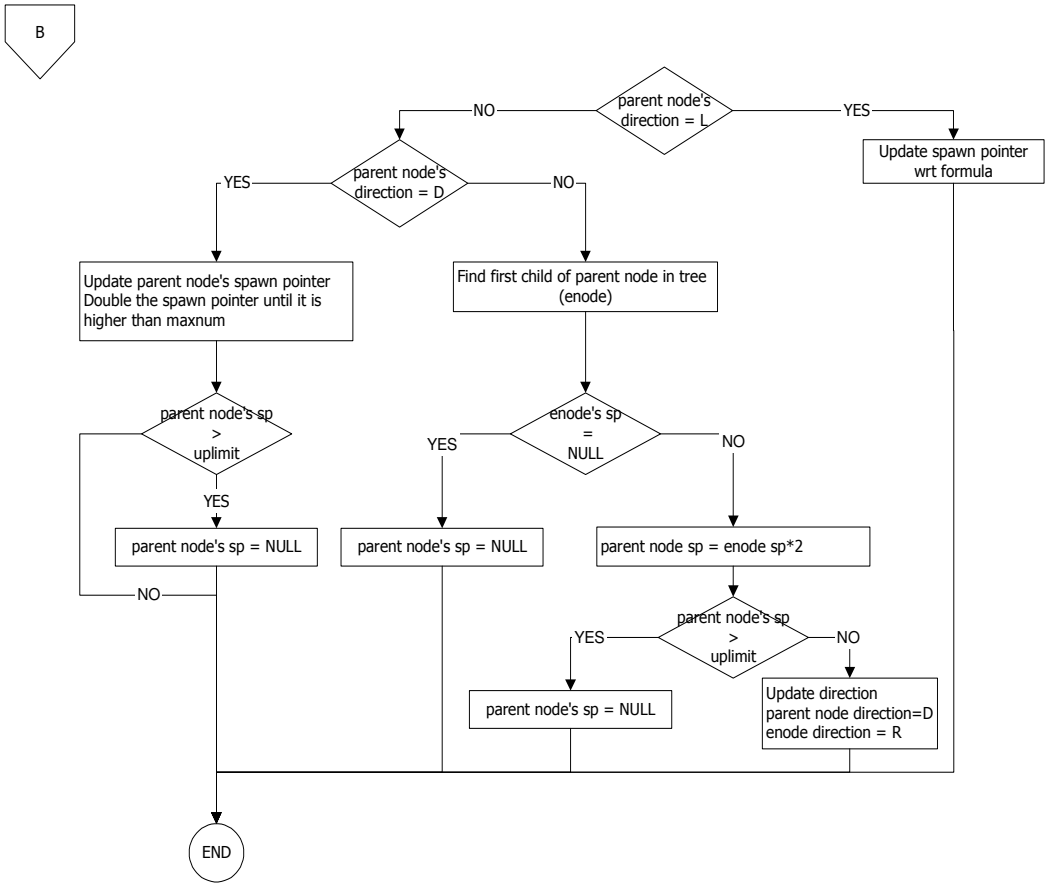


Figure B.3.2 Spawn operation on binary tree second part

APPENDIX C

GRAPHICS FOR SOPHIE-GERMAIN PRIME

Sequence generated by LCG with Sophie-Germain prime of 2*83+1 is shown together with its subsequences shared among three processors by the sequence splitting method in Figure C.1.

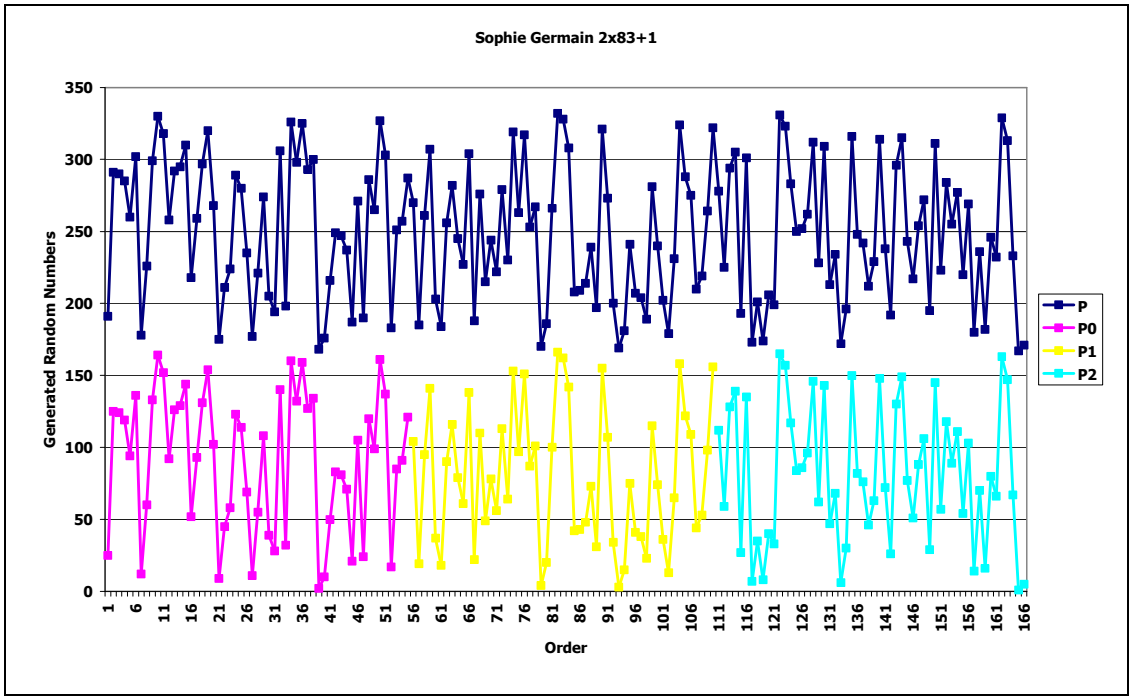


Figure C.1 Sophie-Germain prime sequence splitting method

Sequence generated by LCG with Sophie-Germain prime of 2*83+1 is shown together with its subsequences shared among three processors by the leapfrog method in Figure C.2.

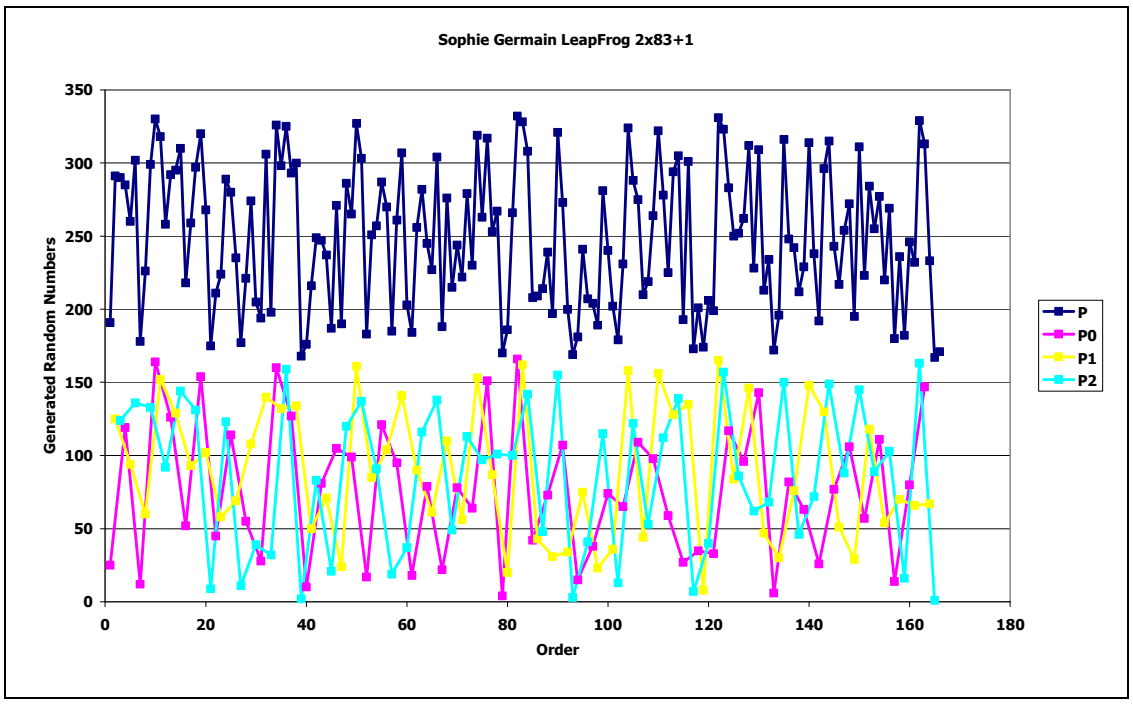


Figure C.2 Sophie-Germain prime leapfrog method