# On Parallelization of General Recurrence Relation

*Dissertation submitted to Jawaharlal Nehru University*
*in partial fulfillment of the requirement*
*for the award of the degree of*

**MASTER OF TECHNOLOGY**
in
**COMPUTER SCIENCE AND TECHNOLOGY**

**By**

**NITIN KUMAR JAIN**

UNDER THE SUPERVISION OF

**PROF. C. P. KATTI**


**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES**
**JAWAHARLAL NEHRU UNIVERSITY**
**NEW DELHI-110067**

**JULY-2008**

# DECLARATION

I hereby declare that this dissertation entitled **"On Parallelization of General Recurrence Relation",** is being submitted by me to School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi in partial fulfillment of the requirement for the award of the degree of **Master of Technology** in Computer Science & Technology, is a record of original work done by me under the supervision of **Prof. C. P. Katti.**

The results submitted in this dissertation have not been submitted in part or full at any other University or Institution for the award of any degree or diploma.

Nitin Jain

**Nitin Kumar Jain**

जवाहरलाल नेहरू विश्वविद्यालय

**SCHOOL OF COMPUTER & SYSTEMS SCIENCE**
**JAWAHARLAL NEHRU UNIVERSITY**
**NEW DELHI – 110067 (INDIA)**

# CERTIFICATE

This is to certify that the dissertation entitled **"On Parallelization of General Recurrence Relation"**, being submitted by Mr. Nitin Kumar Jain to the **School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi** in partial fulfillment of the requirement for the award of the degree of **Master of Technology in Computer Science and Technology**, is a record of original work done by him under the supervision of Prof. C. P. Katti. To the best of my knowledge this work has not been submitted in part or full to any other University or Institution for the award of any degree or diploma.

**Prof. C. P. Katti**
**(Supervisor)**

8/8/08

**Prof. Parimala N.**
**Dean, SC & SS**
**Jawaharlal Nehru University**
**New Delhi, India**

# Acknowledgement

It gives me immense pleasure to record my humble gratitude towards all those who helped and encouraged me through my dissertation, several of them deserve special mention:

Above all, I wish to express my sincere appreciation to my honorable supervisor **Prof. C. P. Katti**, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi for his valuable guidance, enthusiasm & encouragement. He has been an inspiration to me and a great teacher. It was really an unforgettable experience to work under his guidance. I would like to express gratefulness to him for developing in me independent problem solving approach. I would like to appreciate his interest in my queries and their practical suggestions. It would have been really impossible to complete this dissertation without his invaluable support and patience.

I also wish to express a sincere thankfulness to **Prof. Parimala N.**, Dean and Professor of School of Computer and System Sciences, Jawaharlal Nehru University for providing me this wonderful opportunity and necessary infrastructure to carry out this dissertation. I express my heart-felt indebtedness to my friends and appreciate their help in all manners they could. I wish to express my admiration to all of them for their love and affection and being with me through thick and thin.

At last, I would like to thank my parents, who formed part of my vision and taught me the good things that really matter in life. I would like to share this moment of happiness with them.

**Nitin Kumar Jain**

# TABLE OF CONTENTS

# List of Figures

# List of Tables & Graphs

# Abstract

In today's world many scientific and engineering problems are in the form of linear and non linear recurrence equations. A parallel system is required to solve these problems by multiple processors. Here we are going to discuss some example of linear and non linear recurrence equations [LD, 90] and trying to find out the speedup to enhance the parallel computing. We are considering the tree architecture to calculate the speedup of our recurrence relation examples. We finally compare the speedup performance of both the recurrence relations.

# Chapter 1
# Introduction

## 1.1 Parallel computing

A **Parallel Computing** system is a computer having more than one processor, in which the each discrete part is solved concurrently, then each part is further broken down into a series of instructions, these instructions from each part execute simultaneously on different CPUs [Q, 94].

**Parallel Processing** is the simultaneous use of more than one CPU to execute a program. Ideally, parallel processing makes a program run faster because there are more engines (CPUs) running it. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other [Q, 94].

Most computers have just one CPU, but some models have several. There are even computers with thousands of CPUs. With single-CPU computers, it is possible to perform parallel processing by connecting the computers in a network. However, this type of parallel processing requires very sophisticated software called distributed processing software.

Note that parallel processing differs from multitasking, .in which a single CPU executes several programs at once. Parallel processing is also called parallel computing.

In Parallel Computing multiple processors solve a computational problem [1].

- It uses multiple CPUs.
- A single problem is broken into separate discrete problems.
- Each separate problem is further broken in to instructions.
- Instructions of each part execute simultaneously on different CPUs.

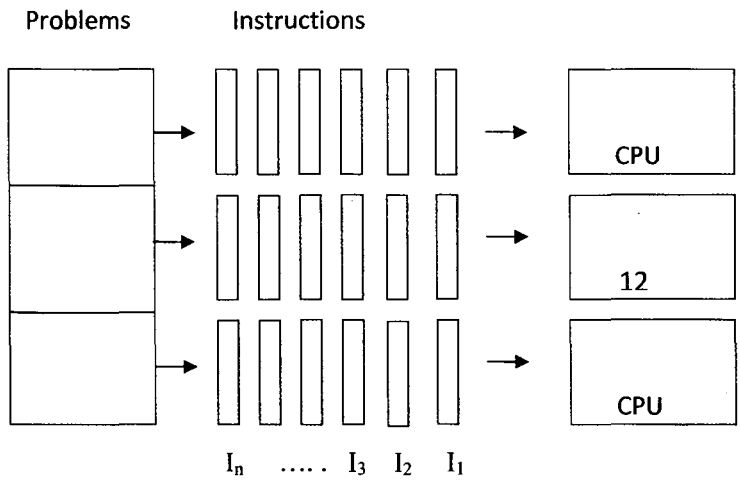



**Figure 1.1: Parallel Computation of a problem**

## 1.2 Historical aspects of parallel computing

"Since real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software and a supporting architecture that are naturally parallel [2]."

The study of computer architecture involves both hardware organization and programming / software requirements. From the hardware implementation point of view, the abstract machine is organized with CPU's caches, buses, microcode, pipeline, physical memory etc. Therefore architecture covers both instruction set architecture and machine implementation code [KH, 93].

### 1.2.1 Von Neumann Architecture

For the last past four decades, virtually all computers have followed a common machine model known as the *von Neumann* computer model, named after the Hungarian mathematician John von Neumann. A Von Neumann computer uses the stored program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory [KH, 93].

Basic Architecture includes:

- Both, data instructions and program are store in memory.
- Program instructions are coded data which tell the computer to do something.
- Data is used by the program as information.
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then *sequentially* performs them.

A basic *Von Neumann Architecture* is shown in figure.

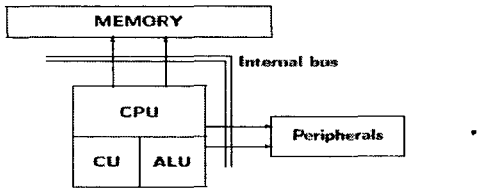



**Figure 1.2: Von Neumann Architecture**

This architecture was built as a sequential machine executing scalar data and because of this, these machines were slow.

Later look ahead techniques were developed to prefetch instruction in order to develop I/E (instruction fetch / decode and execute) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: one is two use multiple functional units simultaneously and the other is to practice pipelining at various processing levels.

### 1.2.2 Flynn's Classification

A widely accepted and often used classification scheme was proposed by Michael Flynn in 1972. Flynn classifies computers by the number of instructions and data-streams, respectively [RC, 81]. The following four classes are distinguished:

| SISD | SIMD |
|---|---|
| Single Instruction, Single Data | Single Instruction, multiple Data |
| **MISD** | **MIMD** |
| Multiple Instruction, Single Data | Multiple Instruction, Multiple Data |

**Table 1.1: Flynn's Classification Scheme**

A Stream simply means a sequence of items (data or instructions) as executed or operated on by a processor. Four main types of machine organizations can be found [RC, 81].

- **SISD** – single instruction / single data stream. This is the conventional serial Von Neumann computer in which there is one stream of instructions (and consequently only one processing unit) and each arithmetic instruction initiates one arithmetic operation, leading to a single data stream of logically related arguments and results. It is irrelevant whether pipelining is used to speed up the processing or not. A machine of this call is sometimes referred to as a scalar computer.

- **SIMD** – single instruction / multiple data stream. This is a computer that retains a single stream of instructions but has vector instructions that initiate many operations. Each element of the vector is regarded as a member of a separate data stream. Hence, there are multiple data streams. This classification includes all machines with vector instructions and machines belonging to this class are often called vector computers.

- **MISD** – multiple instruction stream / single data stream. This is essentially an empty class because it implies that several instructions are operating on the same data simultaneously. We will not consider this class of machines.

- **MIMD** – multiple instruction stream / multiple data stream. Multiple instructions streams imply the existence of several instruction processing units and therefore, necessarily, several data streams. This class is quite

general and includes all forms of multiprocessor configurations from linked mainframes (LANS and WANS) to a large arrays of microprocessors.

One can see that only two of the categories described above are of interest as far as parallel processing is concerned – i.e. the SIMD and MIMD machines.

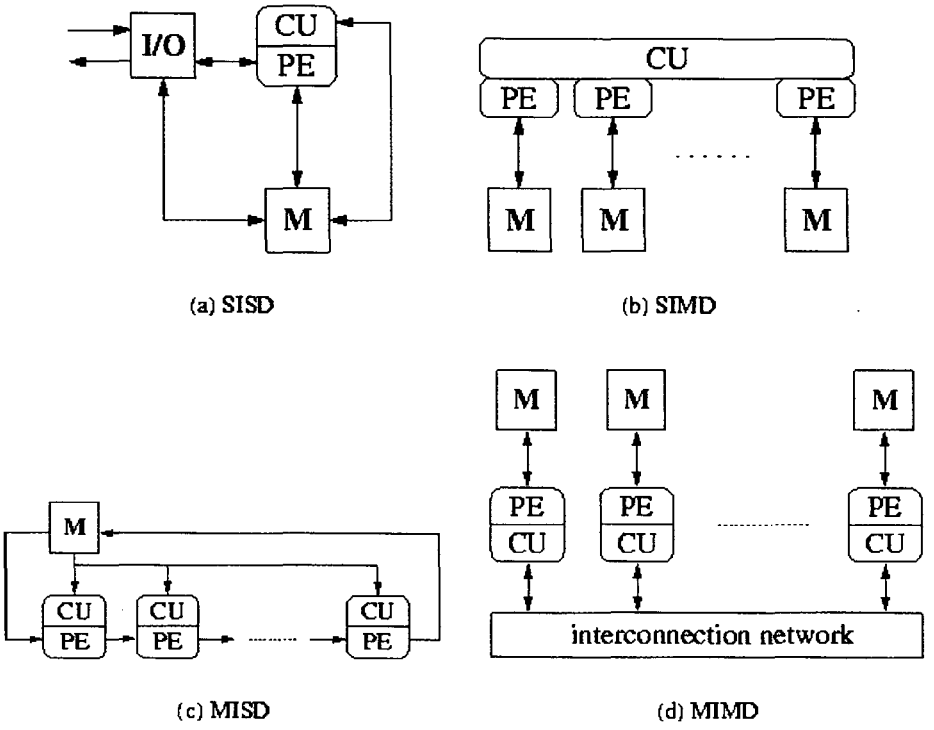



**Figure 1.3: Flynn Classification**

## 1.3 Need of Parallel Computing

Computer architects have been applying parallelism into various levels of hardware to increase the performance of computer sciences, for the decades. To achieve the extremely high speed demanded by contemporary science, architecture must now incorporate parallelism at the uppermost level of the system. Today's, the fastest computers in the world use high level parallelism concept. These computers are leading to new scientific discoveries [1].

Traditional serial computers are characterized by the occurrence of a single locus of control that tells about the next instruction to be executed. During the execution of each instruction data is operated. That is fetched from the global memory one at a time. So, only one instruction is execute at a time. In this processing, total computation is slow by the speed of the memory access and the speed of the input-output devices. A number of methods have been developed for alleviating these bottlenecks, cache memories and pipelining are some examples.

### 1.3.1 Limitations with Serial Computing

Both practical and physical reasons are major constraints to faster serial computers [1].

- **Economic Limitations**: Making a faster single processor is expensive procedure. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

- **Transmission Speed**: The speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm / nanosecond) and the transmission limit of copper wire (9 cm / nanosecond). Increase speeds necessitate increasing proximity of processing elements.

- **Limits to Miniaturization**: Processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.

Parallel computing is currently an area of intense research which is motivated by a variety of factors. These have always been a need for the solution of very large computational problems, but it is only recently that technological advances have raised the possibilities of massive parallel computation and have made the solution of such problems possible. Furthermore, the availability of powerful parallel computers is generating interest in the new types of problems that were not addressed in the past. Accordingly, the development of parallel computing algorithms is guided by the

interplay between old and new computational needs on the one hand, and technological progress on the other [4].

The needs of faster computation have been in number of areas such as computational fluid dynamics and weather prediction as well as image processing. In these applications there is a large number of calculations to be performed. The desired to solve more and more complex problem has always been running ahead of the capabilities of the time and has provided a driving force for the development of faster, and possibly parallel computing machines. The above mentioned types of problems can be easily decomposed along a spatial dimension and have therefore been prime candidate for parallelization, with a different computational unit (processor) assigned the task of manipulating the variables associated with the small region in space. Furthermore in such problems, interactions between variables are local in nature, thus leading to the design of parallel computers consisting of number of processors with nearest neighbor connection [1].

### 1.3.2 Benefits of Parallel Computing

The important reasons for using parallel computing are [1].

- It saves time (wall clock time)
- Solves larger problems
- Gives concurrency (to perform many operations at a time)
- Taking help of non-local resources by using available computer resources on a wide area network, or even the Internet.
- Cost savings by using multiple low cost computing resources. This cost can not save in supercomputing.
- Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

## 1.4 Hardware Implementation of Parallel Computing

There are two categories of parallel computers, which we have described below. These physical models are distinguished by having a shared common memory or unshared distributed memories [5].

### 1.4.1 Shared Memory

General Characteristics:

- Shared memory parallel computers use very widely, but generally they have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processor.
- Shared memory can be divided in to two main classes based upon memory access times: UMA and NUMA [1].

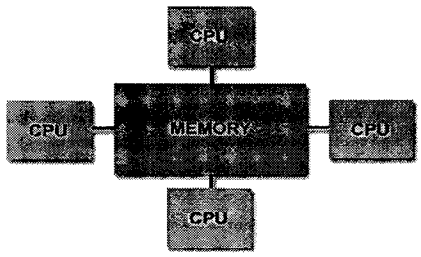



**Figure 1.4: Shared memory Architecture**

#### 1.4.1.1 Uniform Memory Access (UMA):

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines.[3]
- It has identical processors.
- Equal access and access time to memory

#### 1.4.1.2 Non-Uniform Memory Access (NUMA):

- Often made by physically linking between two or more SMPs.
- One SMP can directly access memory of other SMP.
- Not all processor have equal access time to all memories.
- Memory across link is slower.

Advantages of Shared Memory:

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Disadvantages of Shared Memory:

- Primary disadvantage is the lack of ***scalability*** between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory – CPU path, and for cache coherent systems, geometrically increase traffic associated with cache management.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

***Scalability***: The ***scalability*** of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements [AAGV, 03].

### 1.4.2 Distributed Memory

Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory [6].

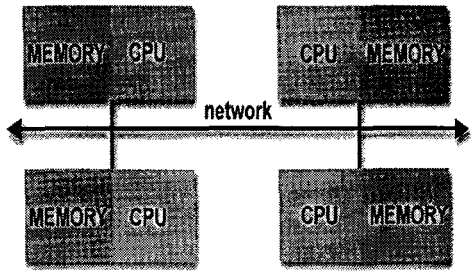



**Figure 1.5: Distributed Memory Architecture**

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.

- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Advantages of Distributed Memory:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages of Distributed Memory:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.

## 1.5 Aspects of Parallel Computing in Communication

The Exploration of parallelism has created a new dimension in computer science. In order to move parallel processing into the mainstream of computing, *H.T. Kung* (1991) has identified the need to make the significant progress in inter-processor communication in parallel architectures [DT, 01].

In various parallel algorithms, the time spent for inter-processor communication is sizable fraction of the total time needed to solve a problem. In that case algorithm experiences a substantial **Communication delay or Communication penalty**.
Communication Penalty (CP) is defined as ratio

$$CP \quad \frac{T_{Total}}{T_{Comp}}$$

Where

$T_{total}$ is the time required by the algorithm to solve the given problem, and

$T_{comp}$ is the corresponding time that can be attributed just to computation that is the time that would be required if all communication were instantaneous.

Distributed computing system can be viewed as a network of processors connected by communication links. Each processor uses its own local memory for storing some problem data and intermediate algorithmic results and exchanges information with other processors in group of bits called packets using the communication link of the network. The size of the packet can be widely varying from a few ten's of bits to some thousand of bits.

A shared memory can also be viewed as a communication network, since each processor can send information to every other processor by storing it in a shared memory.

Another technique called "Store and Forward" can also be adopted for packet switched data communication model. Here a packet must travel over a route involving several processors, may have to wait at any one of the processors for some communication resource to become available before it gets transmitted to next node.

There are four types of **communication delays** [SB, PC]

- **Communication Processing** Time: This is the time required to prepare the information for transmission. For example, making information in packets, affix addressing and control information to the packets, select the communication link on which to send each packet to appropriate buffers.

- **Queuing Time:** When information is assembled into packets for sending on some communication links, it must wait in a queue for the start of transmission, this wait is called queuing time. There are many reasons for waiting, for example, the link may be temporarily unavailable because other information packets are using it or scheduled to use it ahead for the given packet, and another reason is that by limitation of needed resources it may be

necessary to delay the transmission of packets. No proper buffer space at the destination processor at the time of transmission.

- **Propagation Time:** This is the time required in between the end transmission of last bit of the packet at the transmitting processor and the acceptance of the last bit of the packet at the receiving processor. It is called propagation time.

- **Transmission Time:** This is the time that required for transmission of all bits of the packets, one or more of above given times may be negligible.

One or more of above given times may be negligible. It's depending on system and algorithms.

## 1.6 Parameters in Parallel Algorithms

There are mainly two parameters in any sequential algorithm, namely the running (execution) times complexity and space complexity. A good sequential algorithm has a small running time and uses as little space as possible. [Q, 94] Running time, number of processors, and cost are the three principal criteria that are typically used for evaluating parallel algorithm.

- **Running time:** Since speeding up solution of a problem is the main reason for building parallel computers, an important measure in evaluating a parallel algorithm is its running time which is defined as the time taken by the algorithm to solve a problem on a parallel computer. A parallel algorithm is made up of two kinds of steps. In the computation step, a processor performs a local arithmetic or logic operation. In the communication step data is exchanged between the processors via the shared memory or through the interconnection network. Thus the running time of a parallel algorithm includes the time spent in both the computation and the communication steps. The running time depends not only on the algorithm but also on the machine on which the algorithm is executed.

- **Number of processors**: Another important criterion for evaluating a parallel algorithm is the number of processors required to solve the problem.

- **Cost:** The cost of a parallel algorithm is defined as the product of the running time of the parallel algorithm and the number of processors used.

To be able to assess the merits of a parallel algorithm, one needs to count the number of time unit steps needed. For this purpose it is convenient to introduce the idealized notion that during such a time unit step exactly one arithmetical operation can be carried out in the parallel mode.

## 1.7 Applications of Parallel Computing

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence. [KH, 93]

Some examples: Rush hour traffic, Planetary and galactic orbits, Automobile assembly line, Tectonic plate drift, Daily operations within a business.
The great challenge applications areas are

- Ocean modeling can not be accurate with out supercomputing MPP systems. Ocean depletion research demands the use of computers in analyzing the complex chemical and dynamical mechanism involved. Both ocean and ozone modeling affect the global climate forecast.

- High speed civil transport aircraft are being aided by computational fluid dynamics running on super computers. Fuel combustion can be made more efficient by designing better engine models through chemical kinetics calculation.

- Rational drug design is being aided in the research for a cure cancer and acquired Immunodeficiency syndrome. Using a high performance computer, new potential agent has been identified that block the action of human Immunodeficiency virus protease.

- Catalyst for the chemical reactions is being designed with computers with many biological processes which are catalytically controlled by enzymes. Massive

parallel quantum models demand large simultaneous to reduce the time required to design catalyst and to optimize the properties.

- The magnetic recording industry relies on the use of computers to study magneto static exchange interactions in order to reduce noise in metallic thin films used to coat high density discs.

- Other important areas demanding computational support includes digital anatomy in real time medical diagnosis, air pollution reduction through computational modeling, design of protein structure by computational biologist, image processing and understanding, and technology linking research to education.

## 1.8 Thesis Roadmap

The rest of the thesis is organized as follows. In Chapter 2, we have discussed about recurrence relation and various examples of recurrence equations and areas where recurrence relations can be used. In chapter 3, we have discussed about first order general recurrence relation with its mathematical implementation in parallel computing, and in chapter 4, we have calculated the speedup of linear and nonlinear recurrence relations and shown the results. In chapter 5, we have given the conclusion of the dissertation.

# Chapter 2

# Recurrence Relation

## 2.1 Recurrence

A ***Recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs [CLR, 00].

The evaluation of a recurrence gives a problem for a parallel computer because the definition itself is given in terms of sequential evaluations, and only one term is evaluated at a time, which gives no scope for parallel computation. Recurrence can be applied in solution of linear equations by Gauss-elimination method, in all matrix manipulations that require the inner product of vectors, in all iterative methods, in all solutions of ordinary differential equations etc [RC, 81].

## 2.2 Recurrence Relation

A Recurrence Relation for a *sequence* $\{a_n\}$ is a formula which expresses the $n^{th}$ term of the sequence in terms of one or more of the previous terms of the sequence [VL, 06].

A finite ordered list is called a ***sequence***.

The individual terms in the sequence are called terms. For the sequence ($a_0$, $a_1$, $a_2$,.....$a_n$) of real numbers, an equation relating all but a finite numbers of terms $a_r$, to one or more of its previous terms is called a *recurrence relation* with given initial values about the beginning of the sequence. The initial values are also called initial conditions, boundary conditions, or basis for the sequence.

When a recurrence relation is known and one wants to determine explicitly the sequences which satisfies the given recurrence relation. Many natural functions are easily expressed as recurrences [VL, 06].

$$a_n = a_{n-1} + 1, a_1 = 1 \Rightarrow a_n = n(polynomial)$$
$$a_n = 2a_{n-1}, a_1 = 1 \Rightarrow a_n = 2^{n-1}(\exp onential)$$
$$a_n = na_{n-1}, a_1 = 1 \Rightarrow a_n = n!(factorial)$$

A mathematical relationship expressing $f_x$ as some combination of $f_i$ with $i < n$. When formulated as an equation to be solved, recurrence relations are known as recurrence equations, or sometimes difference equations [7].

A *recurrence relation* is an equation which gives the valued of an element of a sequence in terms of the values of the sequence for smaller values of the position index and the position index itself. If the current position of n of a sequence S is denoted by $S_n$ then the next value of the sequence expressed as a recurrence relation would be of the form

$$S_{n+1} = f( S_1, S_2, S_3 \ldots\ldots\ldots\ldots S_{n-1}, S_n, n)$$

Where $f$ is any function. An example of a simple recurrence relation is

$$S_{n+1} = S_n + (n+1)$$

which is the recurrence for the sum of integers from 1 to n+1? This could also be expressed as

$$S_n = S_{n-1} + n$$

## 2.2.1 Examples of Recurrence Relation

**Recursive Solution for Factorial of a number**

For a given number n, Factorial of a number **n!** can be recursively defined as [9]:

$0! = 1$ and $n! = n * (n-1)!$ For $n>=1$

In this definition, the equation is

$n! = n * (n-1)!$ For $n>=1$

is the recurrence relation. It defines each terms of the sequence of factorial as a function of immediately preceding term.

In order to find out the values of the terms in a recursively defined sequence, one must know the values of a specific set of terms in the sequence, usually the beginning terms. The assignments of values for these terms give the set of initial value, which is $0! = 1$. Knowing this value one can compute values for the other terms in the sequence from the recurrence relation [K, 03].

For Example,

1! = 1 * (0!) = 1*1 = 1

2! = 2 * (1!) = 2 * 1 * (0!) = 2 * 1 * 1 = 2

3! = 3 *(2!) = 3 * 2 * (1!) = 3 * 2 * 1 * (0!) = 3 * 2 * 1 * 1 = 6

4! = 4 * (3!) = 4*3*(2!) = 4*3*2*(1!) = 4*3*2*1*(0!) =4*3*2*1*1 = 24

5! = 5 * (4!) = 5*4*(3!) = 5*4*3*(2!) = 5*4*3*2*(1!) =5*4*3*2*1*1 = 120

and so on.

**Recursive solution for Fibonacci sequence**

One more example of the sequence that is defined by recurrence relation is the sequence of Fibonacci sequence [9].

It is a sequence of numbers first created by Leonardo Fibonacci (fi-bo-na-chee) in 1202. It's a very simple series, but its results and applications are limitless. Fibonacci mathematics is a constantly expanding branch of number theory, with more and more people being drawn into the complex details of fibonacci's legacy.

The first two numbers in the series are one and one. To obtain each number of the series, we simply add the two numbers that came before it. In other words, each number of the series is the sum of two numbers preceding it. Historically, some mathematicians have considered zero to be a Fibonacci number, placing it before the first 1 in the series. It is known as *Zeroth Fibonacci number*, and has no real practical merit [9].

Fibonacci numbers satisfy the recurrence relation

$$\mathbf{F_n = F_{n-1} + F_{n-2} \text{ for } n >= 3}$$

Here also $F_n$ is also defined as a function of two preceding terms, we must know two consecutive terms of the sequence in order to compute the subsequent ones.

For the Fibonacci numbers the initial condition are $F_1 = 1$ and $F_2 = 1$.

**Recursive Solution for Tower of Hanoi**

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883 [10].

Given a tower of 3 disks, initially stacked in increasing size on one of three pegs. The objective is to transfer the entire tower to one of the other pegs (the rightmost one) moving only one disk at a time and never a larger one onto a smaller. Solution to this problem can also be solved using recurrence relations.

Let call the three pegs, peg 1 (Source), peg 2 (Auxiliary) and peg 3 (Destination). The objective of the game is to transfer all the discs from peg 1 to peg 3. [TR, 06]

For a given number N of disks, the problem appears to be solved as the following steps:

1. Move the top N-1 disks from 1 to 3 (using 2 as an intermediary peg)
2. Move the bottom disk from 1 to 2.
3. Move N-1 disks from 3 to 2 (using A as an intermediary peg)

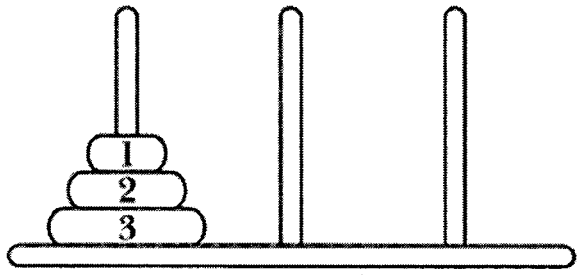



**Figure 2.1 Tower of Hanoi**

For N = 3 it can be shown as

1. Move disk 1 from peg 1 to peg 3
2. Move disk 2 from peg 1 to peg 2
3. Move disk 1 from peg 3 to peg 2
4. Move disk 3 from peg 1 to peg 3
5. Move disk 1 from peg 2 to peg 1
6. Move disk 2 from peg 2 to peg 3
7. Move disk 1from peg 1 to peg 3

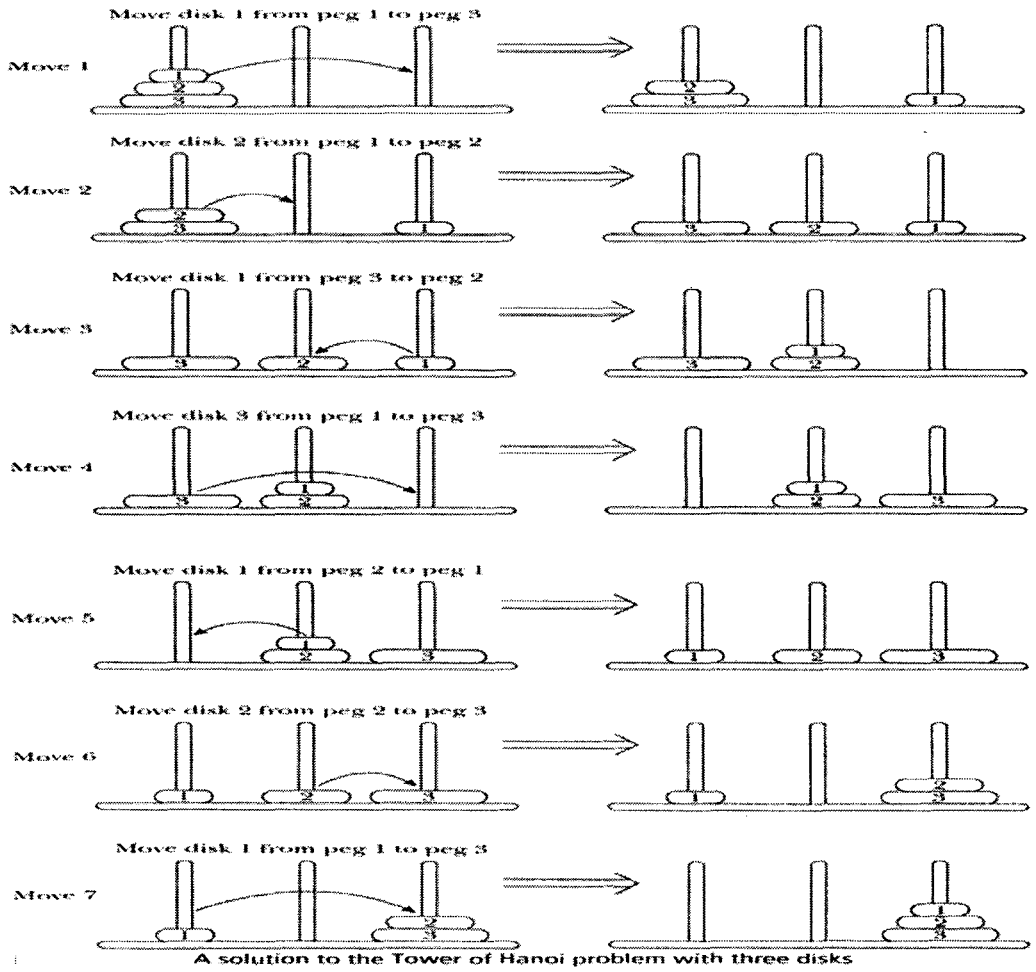



**Figure 2.2 Solution of Tower of Hanoi problem with 3 disks [10]**

Since step 1 requires moving n-1 disks from one peg to another, the minimum number of moves required in step 1 if just $m_{n-1}$. It then takes one move to accomplish step 2, and another $m_{n-1}$, moves to accomplish step 3. This analysis produces the recurrence relation

$$m_n = m_{n-1} + 1 + m_{n-1}$$

which gives the equation

$$\mathbf{m_n = 2 * m_{n-1} + 1 \text{ for } n >= 2}$$

Further only one is required to win the game with disk 1, so the initial value for this sequence is $\mathbf{m_1 = 1}$. By using the recurrence relation and the initial condition we can determine the number of moves required for any number of discs.

**Recursive solution for number of edges in a complete graph**

A complete graph is a graph in which there exists an edge between every pair of vertices. A complete graph is sometimes referred to as a *universal graph* or a *clique*, the degree of every vertex is *n-1* in a complete graph G of *n* vertices. The total number of edges in G is $\frac{n*(n-1)}{2}$. The complete graph with *n* vertices is denoted by $K_n$ [N, 04].

To determine the number in a complete graph $K_n$ with n vertices is also a recursive problem i.e. how many edges need to be drawn to obtain $K_n$ from $K_{n-1}$. Thus the number of edges in $K_n$ is satisfies the recurrence relation

$$\mathbf{e_n = e_{n-1} + (n-1) \text{ for } n >= 2}$$

Definition of $e_n$ involves only the preceding term $e_{n-1}$ and so the value which is needed is $e_n$. Since the complete graph with 1 vertex ha no edges, we see that $e_1 = 0$. This is the initial condition for the sequence.

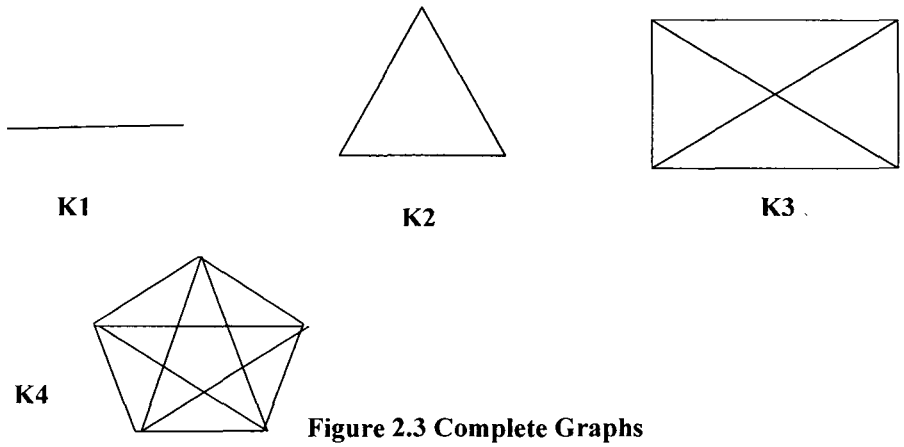



**Figure 2.3 Complete Graphs**

**Recursive Trees**

Drawing a picture of the backsubstitution process gives you a idea of what is going on [9].

We must keep track of two things –

(i) The size of the remaining argument to the recurrence, and

(ii) The additive stuff to be accumulated during this call.

*Example:* $T(n) = 2T(n) + n^2$

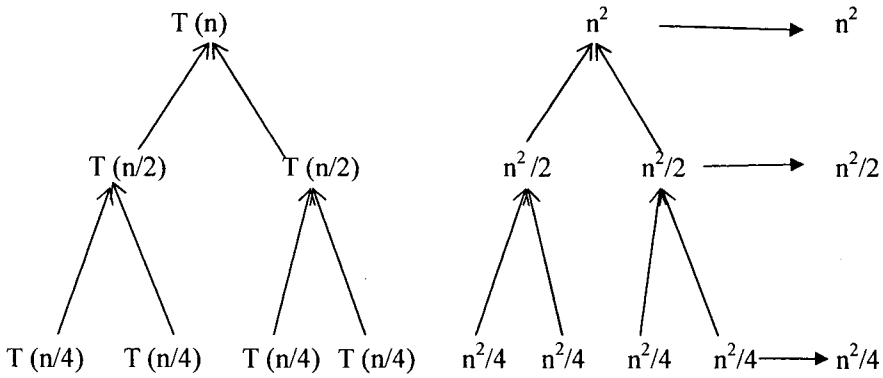



**Fig. 2.4 Recursive Tree**

The remaining arguments are on the left, the additive terms on the right.

Although this tree has height $\log_2 n$, the total sum at each level decreases geometrically, so:

$$T(n) = \sum_{i=0}^{\infty} \frac{n^2}{2^i} = n^2 \sum_{i=0}^{\infty} \frac{1}{2^i} = \Theta(n^2)$$

The recursion tree framework made this much easier to see than with algebraic backsubstitution.

**Matrix Multiplication**

The standard matrix multiplication algorithm for two $n \times n$ matrices is $O(n^3)$ [9].

$$\begin{vmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{vmatrix} * \begin{vmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{vmatrix} = \begin{pmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{pmatrix}$$

Strassen discovered a divide-and-conquer algorithm which takes

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \text{ time.}$$

Since $O(n^{\log_2 7})$ dwarfs $O(n^2)$, case 1 of the master theorem applies and $T(n) = O(n^{2.81})$.

This has been ``improved" by m recurrences until the current best in $O(n^{2.38})$.

## 2.2.2 Solving Recurrences

For solving recurrences there are mainly three methods, these methods are for obtaining bounds on the solution [CLR, 00].

(i) In the *Substitution method*, we guess a bound and then use mathematical induction to prove that our guess is correct.

(ii) The *Iteration method* converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.

(iii) The *master method* provides bounds for recurrences.

The master method depends on the following theorem [CLR, 00]:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence.

$$T(n) = a\,T(n/b) + f(n)$$

Where we interpret n/b to mean either *floor (n/b)* or *ceil (n/b)*. Then T (n) can be bounded asymptotically as follows.

(a) If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon > 0$,

then $T(n) = \Theta(n^{\log_b a})$.

(b) If $f(n)=O(n^{\log_b a})$, then $T(n)=\Theta(n^{\log_b a}\lg n)$.

(c) If $f(n)=O(n^{\log_b a+\varepsilon})$ for some constants $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some constants $c < 1$ and all sufficiently large $n$, then $T(n)=\Theta(f(n))$.

## 2.3 Generalization of Fibonacci sequence For Parallel Computing

Parallelism plays an important role in parallel data processing particularly where numerical algorithms are concerned. Fibonacci sequence formula can also be solved using parallel approach. This recurrence formula for Fibonacci sequence can be generalized to third order formula. [SU, 04]

### 2.3.1 Formula of Second order Recurrence for parallel processing

The following formula is for solving the recurrence relation serially [SU, 04].

**Let** $\mathbf{f_0 = 1, f_1 = b}$

$$\mathbf{f_i = b_i + f_{i-1} + a_i f_{i-2}} \qquad \text{..... 2(a)}$$

$$\mathbf{where\ a_i, b_i \in IR}$$

$$\mathbf{2 \leq i \leq N = 2^n}$$

Since $f_0$ and $f_1$ are given, so $f_2$, $f_3$,....$f_{n-1}$, $f_n$ can be calculated as

$f_2 = b_2 f_1 + a_2 f_0 = b_2 b + a_2$

$f_3 = b_3 f_2 + a_3 f_1 = b_3 (b_2 b + a_2) + a_3 b$

.....

.....

$f_{n-1} = b_{n-1} f_{n-1} + a_{n-1} f_{n-3}$

$f_n = b_n f_{n-1} + a_n f_{n-2}$

} ..... 2(b)

Equation (2.2) can be written as

$$\mathbf{F_i = A_i * F_{i-1}} \qquad \text{..... 2(c)}$$

Where $F_j = \begin{pmatrix} f_j \\ f_{j-1} \end{pmatrix}$ and $A_i = \begin{pmatrix} b_i & a_i \\ 1 & 0 \end{pmatrix}$

with ( j = 1, 2 ... N and i = 2, 3... N)

in more general way equation (2.3) is

$$F_i = (A_i * A_{i-1}) * F_{i-2}$$

.....

.....

$$F_i = (A_i * A_{i-1} * \text{.....} * A_2) * F_1$$

Now for a value of N

$$F_N = (A_N * A_{N-1} * \text{.....} * A_2) * F_1 \qquad \text{..... 2(d)}$$

### 2.3.2 Communication time of two matrices in parallel

Parallel execution of the recurrence relation comes to the execution of the following equation.[TY,99]

$$F_N = (A_N * A_{N-1} * \text{.....} * A_2) * F_1$$

Where $A_2$ ..... $A_{N-1}$ are matrices.

The parallel multiplications of matrices on hypercube architecture perform as follows. For N = 9, number of arrays for multiplication is eight i.e. from $A_9$ to $A_2$.

Take a 3 dimension hypercube with 8 nodes. Put all matrices at each node. Send four matrices on remaining matrices. It takes one unit communication time. Now hypercube has 2 dimensions and each node have two matrices. Multiplication of each pair of matrices takes one unit arithmetic time. In this way, after every communication dimension of hypercube decreases by one. So we can solve equation 2(d) parallely and serially.

## 2.4 Recursive Doubling

Here we present a brief idea about recursive doubling as given in [SU, 03] and [3].

Let us consider a set,

$$M = \{m_1, m_2, m_3\text{--------}m_N\} \text{ having } N = 2^n \text{ elements.}$$

Let $\Theta$ be an arbitrary associative operation that can be carried out on set M.

For example, $\Theta \varepsilon \{+,-,*, \max, \min, ---\}$

Now the expression ( $m_1 \Theta m_2 \Theta$ ------ $\Theta m_N$ ) is formed both serially and in parallel and a comparison is made. For n = 4, we have the following schemes:

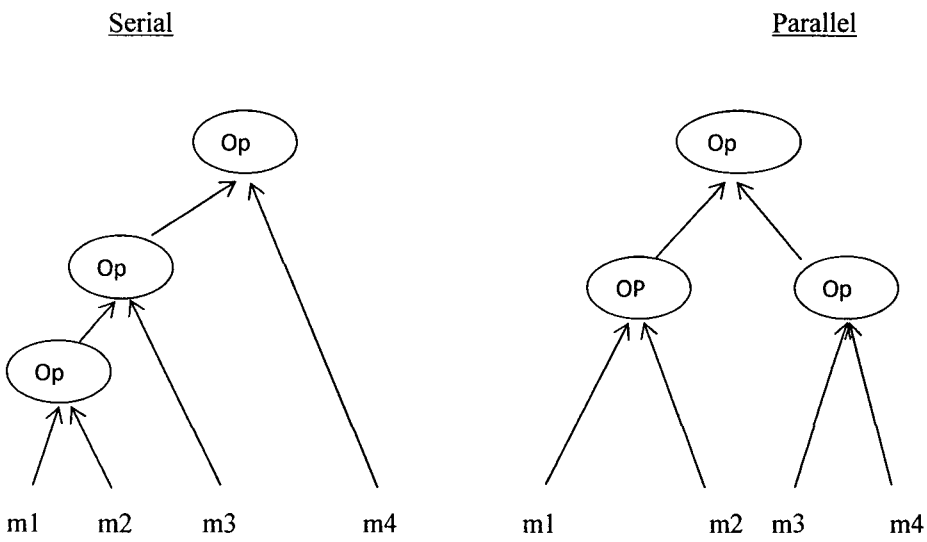



**Fig. 2.5 Tree Representation of Serial and Parallel computation**

Generally, Recursive doubling with $N = 2^n$ elements requires $log_2 N$ parallel steps. Serial implementation requires $N-1$ steps.

## 2.5 n$^{th}$ Order linear Recurrence formula

In general, a linear recurrence relation has the form [SU, 04].

$$f_i = a_{i1}f_{i-1} + a_{i2}f_{i-2} + \ldots\ldots\ldots + a_{in}f_{i-n} \qquad \ldots\ldots\ldots 2(e)$$

$f_0 = 1$,

$f_1 = \alpha_1, \ldots\ldots\ldots\ldots = f_{n-1} = \alpha_{n-1}$

where $a_{ij}$'s and $\alpha_i$'s are constants.

now (2.5) is basically a serial process, where by putting the values of $f_0$, $f_1$, $f_2$ ------$f_{p-1}$, we obtain our first value as $f_p$ and subsequently putting values of ($f_{i-1}, f_{i-2}$, ------- $f_{i-p}$), we obtain the new value $f_i$, where i = p, p+1, p+2 ------N.

As a first simple example, we consider a second order recurrence relation [SU, 04]

$f_0 = 1$

$f_i = b_i * f_{i-1} + a_i * f_{i-2}$ $\quad a_i, b_i \varepsilon R \quad 2 \leq i \leq N = 2^n \quad \ldots\ldots\ldots$ 2(f)

This yields a sort of Fibonacci sequence, let

$$F_i = \begin{pmatrix} f_i \\ f_{i-1} \end{pmatrix} \text{ and } A_i = \begin{pmatrix} b_i & a_i \\ 1 & 0 \end{pmatrix} \qquad \{j = 1, 2, 3\ldots\ldots N,\ i = 2, 3 \ldots N\}$$

Thus, equation (2.6) can be written as

$F_i = A_i F_{i-1} = A_i A_{i-1} \ldots\ldots\ldots A_2 F_1$ and $F_N = A_N \ldots\ldots\ldots A_2 F_1$

Since matrix vector multiplication is associative, $F_N$ computed using the Recursive doubling technique in O ($\log_2 N$) steps by comparison with O (N) steps using a serial calculation.

# Chapter 3

# General First Order Recurrence

In this Chapter, we present a generalization of first order recurrences, not necessarily linear. Ideally multiprocessor systems with $p$ processors can achieve $p$ times speed-up for a given program execution. In most cases, however this ideal speed-up can not be achieved.

## 3.1 General First-Order Recurrences

Given $x_1$, as the initial value, let

$$x_i = f(\alpha_i, x_{i-1}) \qquad \text{..... 3(a)}$$

where $\alpha_i$, for each **i**, is a set of parameters; i > 0, are (real) scalars, and $f(.,.)$ is a function with approximately defined domain and co-domain. The function $f(.,.)$ is called the recurrence function [LD, 90].

There are some examples:

$$\left.\begin{array}{ll} (1) & x_1 = b_1 \\ x_i = a_i + x_{i-1}\,, & N \geq i \geq 2 \\ \alpha_i = (a_i, b_i) & \end{array}\right\} \qquad \text{..... 3(b)}$$

This is the standard first-order linear recurrence.

$$\left.\begin{array}{ll} (2) & x_1 = b_1 \\ x_i = b_i x_{i-1}^{\,a_i}\,, & N \geq i \geq 2 \\ \alpha_i = (a_i, b_i) & \end{array}\right\} \qquad \text{..... 3(c)}$$

This is a non-linear recurrence.

$$\left.\begin{array}{ll} (3) & x_1 = \dfrac{a_1}{c_1} \\ x_i = \dfrac{a_i + b_i x_{i-1}}{c_i + d_i x_{i-1}}, & N \geq i \geq 2 \\ \alpha_i(a_i, b_i, c_i, d_i) & \end{array}\right\} \qquad \text{..... 3(d)}$$

This is a recurrence involving a rational function.

$$\left.\begin{array}{ll} (4) & x_1 = b_1 \\ x_i = b_i + \dfrac{a_i}{x_{i-1}}, N \geq i \geq 2 & \\ \alpha_i = (a_i, b_i) & \end{array}\right\} \quad \ldots\ldots\ldots 3(e)$$

This is the continued fraction expansion.

Now if $a_i \equiv 1$, then equation 3(b) becomes

$$x_i = f(\alpha_i, x_{i-1}) = b_i + x_{i-1},$$

where the recurrence function, $f(.,.)$ is associative.

But if the $a_i$'s are not identically equal to unity in equation 3(b), then in general $f(.,.)$ is not associative, that is,

$$f(\alpha, f(\beta, x)) \neq f(f(\alpha, \beta), x) \quad \ldots\ldots\ldots 3(f)$$

Here $f(\alpha, \beta)$ may not even be defined. Thus in general when the function $f(.,.)$ does not satisfy equation 3(f), parallel algorithms for computing $x_i$'s are not obvious at all [LD, 90].

Now we develop a sufficient condition on $f(.,.)$ with which parallel algorithms for computing $x_i$'s can be derived. This sufficient condition is expressed in terms of another function $g(.,.)$ related to $f(.,.)$, which is defined as follows [LD, 90].

The recurrence function $f(.,.)$ is said to have a *companion function* $g(.,.)$ [1] If for all the values of the parameters $\alpha_1$ and $\alpha_2$.

$$f(\alpha_1, f(\alpha_2, x)) = f(g(\alpha_1, \alpha_2), x) \quad \ldots\ldots\ldots 3(g)$$

In the following, we first derive the companion function for the above four examples, given above in the same order

$$\left.\begin{array}{ll} (1) & x_i = a_i x_{i-1} + b_i \\ = (a_i a_{i-1}) x_{i-2} + (a_i b_{i-1} + b_i) & \\ g(\alpha_i, \alpha_{i-1}) = \alpha_i^{(1)} & \\ = (a_i a_{i-1}, a_i b_{i-1} + b_i) & \end{array}\right\} \quad \ldots\ldots 3(h)$$

$$\left.\begin{array}{ll} (2) & x_i = b_i x_{i-1}^{a_i} \\ x_i = b_i b_{i-1}^{a_i} x_{i-2}^{a_i a_{i-1}} & \end{array}\right\}$$

$$g(\alpha_i, \alpha_{i-1}) = \alpha_i^{(1)} \qquad \text{..... 3(i)}$$

$$= (a_i a_{i-1}, b_i b_{i-1}^{a_i})$$

(3) $$x_i = \frac{a_i + b_i x_{i-1}}{c_i + d_i x_{i-1}}$$

$$x_i = \frac{(a_i c_{i-1} + b_i a_{i-1}) + (a_i d_{i-1} + b_i b_{i-1}) x_{i-2}}{(c_i c_{i-1} + d_i a_{i-1}) + (c_i d_{i-1} + d_i b_{i-1}) x_{i-2}}$$

$$g(\alpha_i, \alpha_{i-1}) = \alpha_i^{(1)} \qquad \text{.....3(j)}$$

$$= (a_i c_{i-1} + b_i a_{i-1}, a_i d_{i-1} + b_i b_{i-1}, c_i c_{i-1} + d_i a_{i-1}, c_i d_{i-1} + d_i b_{i-1})$$

(4) $$x_i = b_i + \frac{a_i}{x_{i-1}} \qquad \text{..... 3(k)}$$

Rewriting $x_i$ as

$$x_i = \frac{a_i + b_i x_{i-1}}{x_{i-1}} \qquad \text{..... 3(l)}$$

It can be seen that this is a special case of 3(c), with $c_i = 0$ and $d_i = 1$.

**Theorem** [LD, 90]

If a recurrence function *f(., .)* has an associated companion function *g(., .)* satisfying equation 3(e), then for $\alpha_1$, $\alpha_2$, $\alpha_3$ and x,

$$f(g(\alpha_1, g(\alpha_2, \alpha_3)), x) = f(g(g(\alpha_1, \alpha_2), \alpha_3), x)$$

i.e. *g(., .)* is associative with respect to *f(., .)*.

So to understand the importance of this theorem, let us calculate

$$x_5 = f(\alpha_5, x_4)$$

$$x_5 = f(\alpha_5, f(\alpha_4 x_3))$$

$$x_5 = f(\alpha_5, f(\alpha_4, f(\alpha_3, x_2)))$$

$$x_5 = f(\alpha_5, f(\alpha_4, f(\alpha_3, f(\alpha_2 x_1))))$$

$$x_5 = f(\alpha_5, f(\alpha_4, f(g(\alpha_3, \alpha_2), x_1)))$$

$$x_5 = f(\alpha_5, f(g(\alpha_4, g(\alpha_3, \alpha_2)), x_1))$$

$$x_5 = f(g(\alpha_5, g(\alpha_4, g(\alpha_3, \alpha_2))), x_1)$$

$$x_5 = f(g(g(\alpha_5, \alpha_4), g(\alpha_3, \alpha_2)), x_1)$$

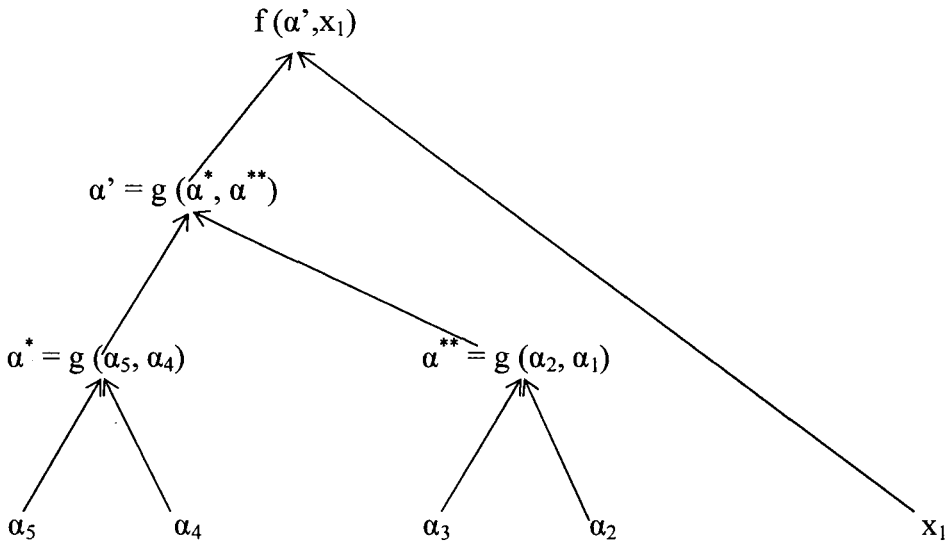



**Fig. 3.1 An illustration of the above theorem**

## 3.2 Computational characteristics of First Order linear Recurrence

In First order Linear Recurrence (FOLR), equation 3(h) holds and here we are expanding this equation and finding the data dependency flow, which we have shown in figure 3.2 [8].

$$x_i = a_i x_{i-1} + b_i, \text{ for } 2 \leq i \leq N$$

By expanding the loop in figure 3.1, we get data dependency flow as shown in figure 3.2.

**pseudo code of equation 3(h)**

```
x[0] = b[0]
    Do (i = 1: N-1)
        x[i] = a[i] * x[i-1] * b[i]
```

Enddo

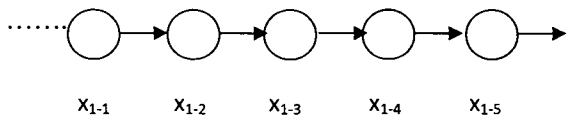



**Figure 3.2 Data dependency flow for pseudo code**

In figure 3.2, a circle with subscript $X_{j-1}$, tells the $j^{th}$ statement of the i-th iteration of the loop which we have given in pseudo code i.e. Fig.3.1, there is also the loop carried dependency between x[i]'s of the two consecutive iterations. So its not possible to parallelize it directly by decomposing the data. The nested form of equation 3(h) reduces the number of multiplications. So the nested form of this equation can not be parallelized because of these data dependencies [8].

Now, when we unfold this nested form of equation 3(h) in to the expanded form of equation 3(m), then we get the ability to change the dependency pattern.

We have shown the expanded form of first order linear recurrence is expanded into $x_j$, when the given expanded form is dependent only on $x_j$ as shown in equation 3(h).

$$x_i = a_i x_{i-1} + b_i$$

$$x_i = a_i\left(a_{i-1}x_{i-2} + b_{i-1}\right) + b_i$$

$$x_i = a_i a_{i-1} x_{i-2} + a_i b_{i-1} + b_i$$

$$x_i = a_i a_{i-1}\left(a_{i-2}x_{i-3} + b_{i-2}\right) + a_i b_{i-1} + b_i$$
$$x_i = a_i a_{i-1} a_{i-2} x_{i-3} + a_i a_{i-1} b_{i-2} + a_i b_{i-1} + b_i$$

.....

.....

$$x_i = x_j \prod_{k=j+1}^{i} a_k + \prod_{k=j+1}^{i}\left[ b_k \prod_{l=k+1}^{i} a_l \right] for 1 \le j \prec i \le N \quad \text{....3(m)}$$

Figure 3.3 shows the extreme case of expanded form, which is expanded into $x_1$ [8].

In this expanded form of equation 3(h), the computation of $x_i$ is dependent on the value of $x_1$ instead of the immediate preceding $x_{i-1}$. But the expanded form of first order linear recurrence increases the number of multiplication operations, compared to the nested form of first order linear recurrence, the data dependencies, here the data dependencies are reduced which were preventing parallelization [8].

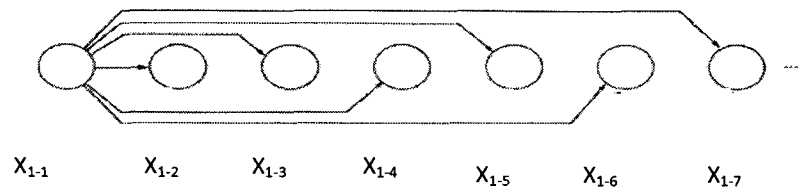



**Figure 3.3 Dependency flow of expanded form of First Order Linear Recurrence**

The expended from of First order linear recurrence reduced both the loop carried dependency and also the overhead of increased multiplication operations.

Now we discuss some advantage and disadvantage of expanded form of the First Order Linear Recurrence and try to find out the balancing point between the nested form and the expanded form. For the simplicity, we calculate only multiplication operations in calculating the execution time and ignore other operations such as loop control and parallel execution overhead. We consider only the number of multiplication as a measure of the execution time [8].

Let $n_{i,j}$ be the number of multiplications needed for calculating $x_i$ in expanded form of FOLR, which is dependent only on $x_j$ . The formula to compute $n_{i,j}$ is given by

$$n_{i,j} = \text{number of multiplication in} \left( x_i \prod_{k=j+1}^{i} a_k \right) \quad +$$

number of multiplication in $\left( \sum_{k=j+1}^{i} \left[ b_k \prod_{l=k+1}^{i} a_l \right] \right)$

$n_{i,j} = (i-j) + [(i-j-1) + (i-j-2) + ..... + 1]$

$$n_{i,j} = \sum_{k=1}^{i-j} k, \text{ where } 1 \leq j < i \leq N. \qquad \text{..... 3(n)}$$

Hence the total number of multiplications is computing from $x_1$ to $x_N$, is given by

$$\sum_{i=2}^{N} n_{i,j} = \sum_{i=2}^{N} \sum_{k=1}^{i-j} k \qquad \text{..... 3(o)}$$

The time to execute the nested form of first order linear recurrence in sequential is N, while the time to execute the expanded form of first order linear recurrence is parallel is given by equation 3(o) [8].

Let $p$ be the number of processors used for parallel execution of the expanded form of first order linear recurrence, then the relationship between p and N (number of elements in array x) is given by

$$\frac{1}{p} \sum_{i=2}^{N} \sum_{k=1}^{i-j} k = N \qquad \text{..... 3(p)}$$

Lets take an example where we take j=1, and N=100, the number of processors p, needed to keep the execution time equal to that of serial execution of nested form of first order linear recurrence would be 1667 [8].

Thus, practically it is impossible to parallelize the first order linear recurrence by simply using the expanded form of first order linear recurrence. Consequently, for the successful parallel execution of first order linear recurrence, the overhead of extra multiplications should be mitigated by evenly distributing them to each processor [8].

# Chapter 4

# Parallel Algorithm for Solving Recurrences

## 4.1 Algorithm for General First Order Linear Recurrence

The simplest example of recurrence problem is the first order recurrence in which $x_i$ depends only on $x_{i-1}$. The log-sum algorithm for solving $x_i = a_i + x_{i-1}$ is an example of it. By introducing the parallelism into the solution of this recurrence came from the associativity of addition, which allows us to rewrite standard serial evaluation [3].

$$X_4 = a_4 + (a_3 + (a_2 + a_1))$$

Or

$$X_4 = (a_4 + a_3) + (a_2 + a_1)$$

Here we shown the two equations, from which we can apply parallelism on second equation by two processors, one computing $(a_4 + a_3)$ and the other computing $(a_2 + a_1)$ as shown in figure. For N number of calculation we have N/2 parallel additions at the first step, N/4 at the second step, ….., and $N/2^k$ at the $k^{th}$ until, at the ceil($\log_2 N$)-th step, $x_N$ is computed with one final addition [3].

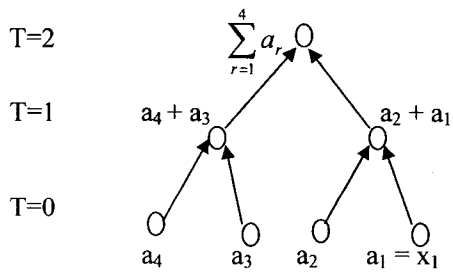



**Figure 4.1 computation of $x_4$**

Let us take another example to calculate $x_9$

The Equation which we are taking here is a linear equation, and then we will calculate the Speedup of that linear equation [LD, 90].

$$x_i = a_i x_{i-1} + b_i$$

Now By the definition of general first order recurrence [LD, 90] in equation 3(a), we know

$$x_9 = f(\alpha_9, x_8)$$

$$x_9 = f(\alpha_9, f(\alpha_8, x_7))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, x_6)))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, x_5))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, x_4)))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, f(\alpha_4, x_3))))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, f(\alpha_4, f(\alpha_3, x_2)))))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, f(\alpha_4, f(\alpha_3, f(\alpha_2, x_1))))))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, f(\alpha_4, f(g(\alpha_3, \alpha_2), x_1)))))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(\alpha_5, f(g(\alpha_4, g(\alpha_3, \alpha_2)), x_1))))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(a_6, f(g(\alpha_5, g(\alpha_4, g(\alpha_3, \alpha_2))), x_1)))))$$

$$x_9 = f(\alpha_9, f(\alpha_8, f(\alpha_7, f(\alpha_6, f(g(g(\alpha_5, \alpha_4), g(\alpha_3, \alpha_2)), x_1)))))$$

.....

.....

.....

$$x_9 = f(g(g(g(\alpha_9, \alpha_8), g(\alpha 7, \alpha_6)), g(g(\alpha_5, \alpha_4), g(\alpha_3, \alpha_2))), x_1)$$

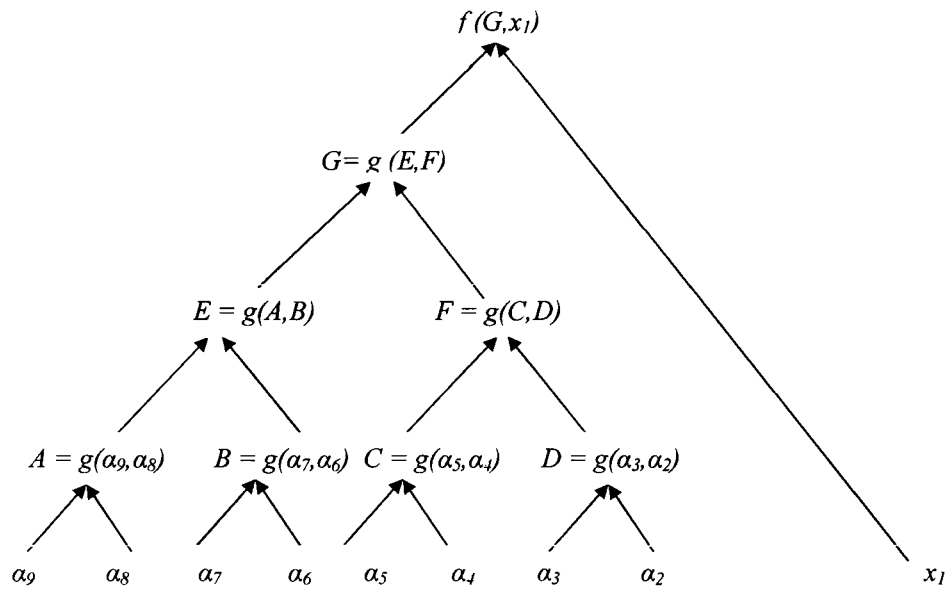



**Figure 4.2 Calculation of $x_9$ by the algorithm**

**Algorithm to calculate $x_n$**

```
/* Given array1 [N], here N is the size of the array to store α values (α1, α2,...αn)
   Given array2 [N], array2 is used to store α values for computation.
   Declare variables I, J, K, P
*/
P = N
FOR I = 1 TO P STEP 1 DO
        array2 [I] = array1 [I]
END
FOR I =1 TO (log2N) STEP 1 DO IN PARALLEL
        K = 0
                FOR J = 1 TO P STEP 2 DO IN PARALLEL
                        array2 [K] = g (array2 [J], array2 [J+1]
                        K = K+1
                END
```

P = P/2

END

COMPUTE f (array2 [0], $x_1$)

**Procedure to calculate $x_9$**

Here $\alpha$ values are given as $(\alpha_2, \alpha_3, \ldots. \alpha_9)$ which is stored in *array1 [8]* and we will store these values in *array2 [8]* for computation and $N = 8$, $x_1$ given as initial value

$P = 8$ therefore $(log_2 8) = 3$

I[st] iteration

array2 [0] = g (array2 [1], array2 [2])

array2 [1] = g (array2 [3], array2 [4])

array2 [2] = g (array2 [5], array2 [6])

array2 [3] = g (array2 [7], array2 [8])

II[nd] iteration

array2 [0] = g (array2 [1], array2 [2])

array2 [1] = g (array2 [3], array2 [4])

III[rd] iteration

array2 [0] = g (array2 [1], array2 [2])

Now computation of **f (array2 [0], $x_1$)** will give value of $x_9$.

## 4.2 Calculation of Speedup for a Linear Recurrence Equation

Let us take a linear equation to calculate the speed up of a linear recurrence [LD, 90].

$x_n = a_n x_{n-1} + b_n$ , where $x_1$ is given

To calculate speedup, we require the time taken in serial computation as well as the time taken in parallel computation.

So for serial computation

$$x_1 = b_1 + a_1 x_0$$

It is taking two operations, one is multiplication and another is addition.

now $$x_2 = b_2 + a_2 x_1$$

It is taking four operations, one is multiplication and another is addition and two operations of $x_1$, hence total four operations.

Now $$x_3 = b_3 + a_3 (x_2)$$

It is taking six operations, four were previous and two new operations.

Similarly $$x_4 = b_4 + a_4 (x_3)$$

It is taking eight operations, six were previous and two new operations.

$$x_5 = b_5 + a_5 (x_4)$$

It is taking ten operations, eight were previous and two new operations.

Hence we have to calculate total number of operation in serial computation; this is understood by seeing the number of operation on $x_2$, $x_3$, $x_4$, $x_5$ and so on, that the total number of operations are in multiple of 2. So on solving, we get total ***2n*** operation in serial communication.

Now we have to calculate time taken in parallel computation,

Now for Parallel computation, as we have shown solution for the value of $x_9$ in section 4.1.

As this example shows for calculation of *g* function it is taking 3 operations at each level of the tree and it is also taking 2 operations in calculating function of *f*, and we also know that in a tree of level *n* its time complexity is $(\log_2 n)$.

Hence total time taken in parallel computation is **$(3 * \log_2 n + 2) * AT$**, here we are assuming only Arithmetic time in operations and ignoring the communication time.

So speedup is

Speedup = time taken in serial computation / time taken in parallel computation

$$S_p = \frac{2n}{(3 * \log_2 n + 2) * AT}$$

Where AT is the arithmetic time required to perform operations.

Here both serial and parallel communications will take arithmetic time (AT), but in our example we are neglecting AT from both type of communication.

## r a Non Linear Recurrence

, 90]

operations in Serial communications.

For n=1, $x_2 = -2x_1^2$

here it is easily shown that there are two operations and both are multiplications

now for n=2, $x_3 = -2x_2^2$

there are also two operations and the two previous operations, so total number of operations are four.

For n=3, $x_4 = -2x_3^2$

There are total six operations, among which two are new and four are previous.

For n=4, $x_5 = -2x_4^2$

There are total eight operations, same as in earlier cases.

So now we can easily guess that in the generalization of this non linear equation there are total number of operations is ***2n*** in serial computation.

Now we have to calculate the total number of operations in parallel computation.

Now same as we have calculated for the different values of n, we will again calculate in same manner.

$x_{n+1} = -2x_n^2$ is the non linear recurrence equation, where $x_1$ is given.

For n=1, $x_2 = -2x_1^2 = (-2)^3(x_0)^4$

For n=2, $x_3 = -2x_2^2 = (-2)^7(x_0)^8$

For n=3, $x_4 = -2x_3^2 = (-2)^{15}(x_0)^{16}$

For n=4, $x_5 = -2x_4^2 = (-2)^{31}(x_0)^{32}$

and so on.

So on generalizing this equation, we get the general solution for calculating $x_n$.

$$x_n = (-2)^{2^n-1}\left(x_0^{2^n}\right)$$

Now let us calculate the number of operations this equation is taking in parallel.

After finding the generalized equation, implement it on tree topology, so that we can calculate number of operations this equation is taking.

By taking an example to calculate $x_4$.

The generalized equation of $x_4$ is

$$x_4 = -2x_3^2 = (-2)^{15}(x_0)^{16}$$

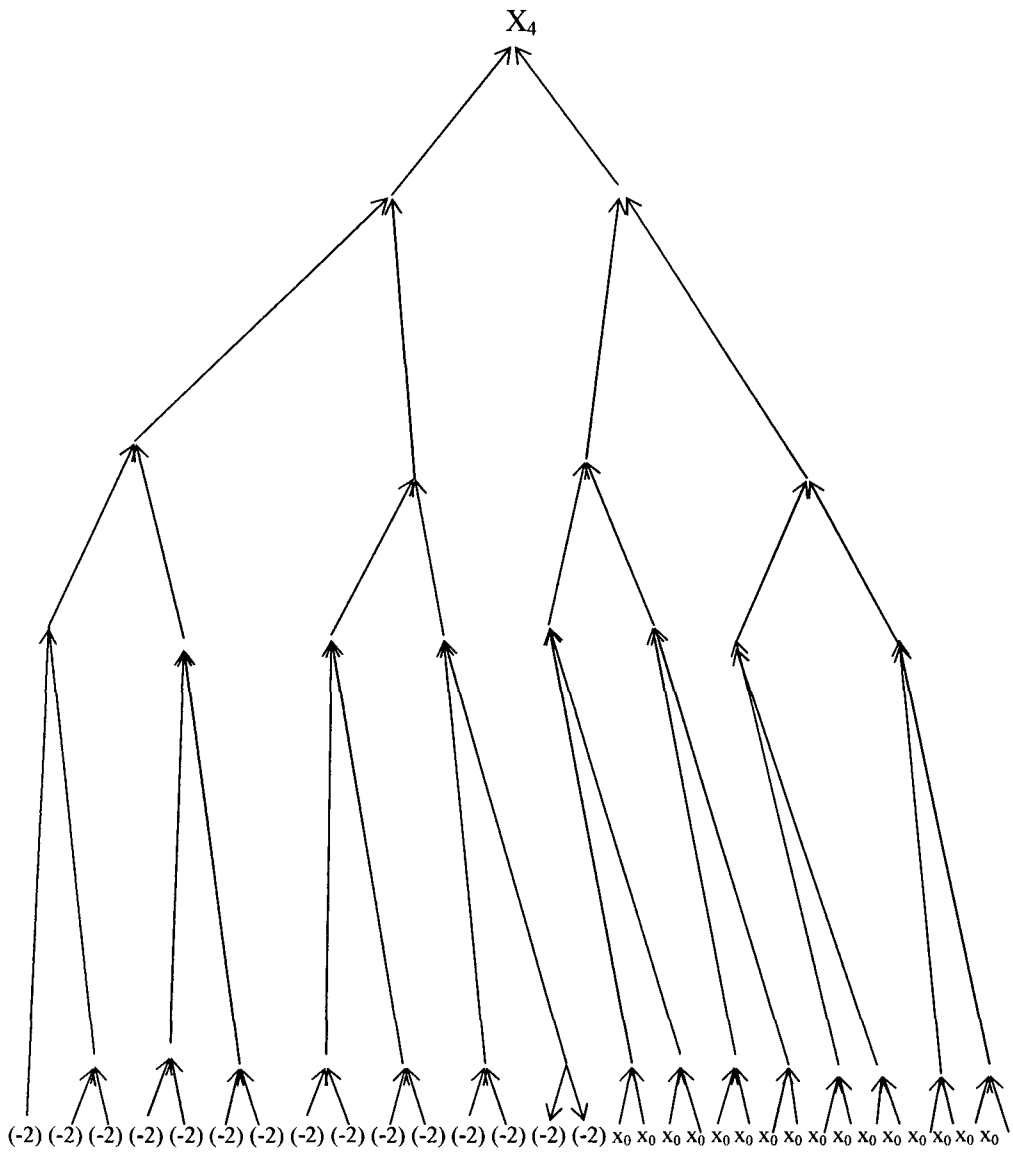



**Figure 4.3 Calculation of $x_4$ for Non Linear Equation**

Here it is easily seen that it is taking 5 operations, so we conclude that total number of operations taking in parallel for $n$ processors will be $n+1$.

So speedup is

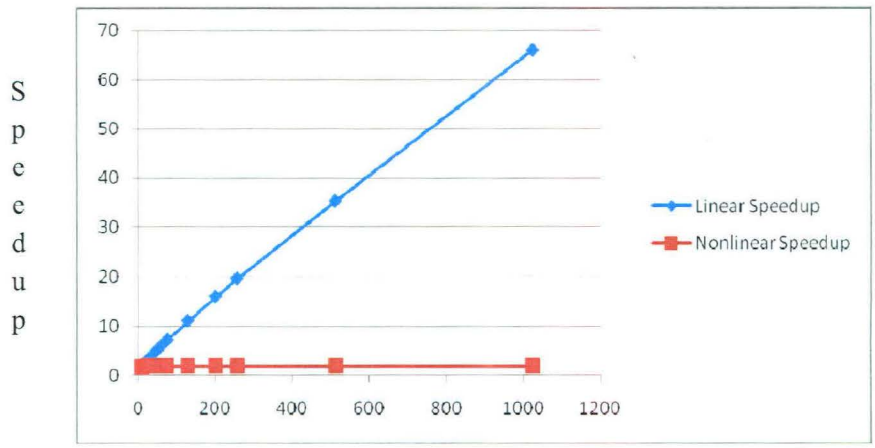
Speedup = time taken in serial computation / time taken in parallel computation

$$S_p = \frac{2n}{(n+1)}$$

Here both serial and parallel computations will take Arithmetic Time (AT), but in our example we are neglecting AT from both type of communication.

| Number of Processors n | Speedup of Linear Equation | Speedup of Non Linear equation |
|---|---|---|
| 8 | 1.454 | 1.778 |
| 16 | 2.285 | 1.882 |
| 25 | 3.138 | 1.960 |
| 32 | 3.764 | 1.939 |
| 40 | 4.452 | 1.951 |
| 50 | 5.282 | 1.960 |
| 64 | 6.4 | 1.969 |
| 75 | 7.251 | 1.973 |
| 128 | 11.13 | 1.984 |
| 200 | 16.04 | 1.990 |
| 256 | 19.692 | 1.992 |
| 512 | 35.310 | 1.996 |
| 1024 | 66.064 | 1.998 |

**Table 4.1 Speedup of linear and non linear equation**




**Graph 4.1 Speedup of Linear and Non Linear Equation**

Here Blue line is the speedup of Linear equation and Red line is the speedup of Non Linear equation.

In our example of Linear and non Linear equation, the speedup of linear equation is very much higher than speed up of non linear equation. The speed of non linear equation will be less than two in our example.

# Chapter 5

# Conclusion & Future Work

The Parallel Computing technique for solving the recurrence relation is better for a large data set. But if data set is not large enough, parallel techniques may not be needed. For small data set serial approach will give the improved results as there is no need of parallel computation.

In our examples of recurrence relation we have calculated speedup on both the linear and non linear equations, the non linear equation has less speed up, as compare to linear equation's speedup. Non linear recurrence relation in our example has speedup less than 2. In our dissertation we have implemented it on tree architecture, but it may vary from architecture to architecture.

It might be possible that it may produce better speedup if we implement it on Hypercube or Supercube architecture.

So the future work can be performed on parallelization of recurrence relation by using different architectures to improve the speedup.

# References


[LD, 90] S. Lakshmivarahan, Sudarshan K. Dhall, "Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems", McGraw Hill, 1990

[AAGV, 03] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, "Introduction to parallel Computing", Pearson Education ltd., 2003

[KH, 93] Kai Hwang, "Advances Computer Architecture: Parallelism, Scalability, Programmability", McGraw Hill, 1993

[Q, 94] Michael J. Quinn, "Parallel Computing: Theory and Practice", McGraw Hill, 1994

[RC, 81] R W Hockney, C R Jesshope, "Parallel Computers Architecture, Programming and Algorithms", Adam Hilger Ltd, Bristol

[K, 03] Kenneth H. Rosen, "Discrete mathematics and its applications", Tata McGraw Hill, 2003

[N, 04] Narsingh Deo, "Graph theory with its applications to engineering and computer science", Prentice hall of India 2004

[T, 73] Joseph Frederick Traub, "Complexity of Sequential and Parallel Numerical algorithm", Carnegie-Mellon University

[BA, 84] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and Hyperbus structures for a computer network", IEEE transactions on Computers, vol. C-33.

[FC, 92] T. Len Freeman, Chris Phillips, "Parallel Numerical Algorithm", Prentice Hall of India 1992

[SU, 04] U. Schendel, "Introduction to Numerical methods for Parallel Computers", john Wiley and Sons, 2004

[SB, PC] Shawn T. Brown, "Overview of Parallel Computing",

[TR, 06] Tom Leighton and Ronitt Rubinfeld, "Recurrences I" 6.042/18.062 Mathematics for computer science, October 06.

[TD, 01] Deepali Taneja, "Performance evaluation of numerical algorithms in certain parallel architecture systems", Dec 01.

[CLR, 00] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms, 2000.

[VL, 06] Vese Luminita, "Recurrence Relations", November 14, 2006.

[1] https://computing.llnl.gov/tutorials/parallel_comp/

[2] http://aeshen.typepad.com/aeshen/2007/02/parallel_comput.html

[3] Kogge P.M. "Parallel solution of recurrence problems", IBM J. Research development lab

[4] http://www.gup.uni-linz.ac.at/thesis/diploma/bernhard_reitinger.html

[5] http://www.math.pitt.edu/~sussmanm/.html

[6] http://www.cs.bath.ac.uk/Alwynbarry.htm

[7] http://mathworld.wolfram/recurrnce.html

[8] Hong-Soog Kim, Young-Ha Yoon, Dong Soo Han, "Parallel processing of first order linear recurrence on SMP machines". The journal of supercomputing.

[9] Link: http://cs-netlab-01.lynchburg.edu/courses/algorithms/recurenc.htm

[10] ww.cs.duke.edu/courses/spring05/cps130/lectures/skiena.lectures/lecture3.pdf

[11] http://www.cut-the-knot.org/recurrence/hanoi.shtml