INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms
International
300 N. Zeeb Road
Ann Arbor, MI 48106
Parallel solution to linear recurrence

Mitra, Kamal Vedprakash, M.S.
Rice University, 1987
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ______
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) _________ lacking when material received, and not available from school or author.
12. Page(s) _________ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages ______
15. Dissertation contains pages with print at a slant, filmed as received ______
16. Other ____________________________

University Microfilms International
Rice University

Parallel Solution to Linear Recurrence

by

Kamal V. Mitra

A thesis submitted
in partial fulfillment of the
requirements for the degree

Master of Science

Approved, thesis committee:

Peter J. Varman, Assistant Professor of
Electrical and Computer Engineering, Director

J. Robert Jump, Professor of Electrical and
Computer Engineering

James B. Sinclair, Associate Professor of
Electrical and Computer Engineering.

Houston, Texas
Dedicated to
my eternal Lord and spiritual master

His Divine Grace Kirtanananda Swami Bhaktipada

om ajnana timirandhasya jnana anjana salakaya
caksur unmilitam yena tasmair sri Guru vair namah

I was born in the darkness of ignorance and my spiritual master opened my eyes with the torchlight of knowledge. I offer my respectful obeisances unto my spiritual master.
Acknowledgement

I offer my respects and homages unto the Supreme Lord Sri Krishna (who is God Himself) who is giving us the intelligence to do scientific research. I would like to thank my Advisor Prof. Peter J. Varman for his help and guidance in the research reported in this thesis. I would also like to thank my friend and colleague Kshitij Doshi for his timely help in preparing this thesis.
ABSTRACT:

In this thesis we present an optimal time parallel solution to the problem of first order linear recurrence. Given a system of n first order equations, the proposed parallel algorithm solves it in time $O(n/p + \log p)$ on a multiprocessor system with p processors. The multiprocessor system may be operating in synchronous or asynchronous mode. Our solution is not very restrictive in its requirement of processor interconnection. It requires only that certain processors be able to communicate with certain other processors as compared to some of the previous solutions which require a shared memory model of a parallel machine and require up to n simultaneous requests to memory to be serviced in $O(1)$ (constant) time. We map this algorithm on models of parallel machines such as the hypercube, a single bus connected system, a linear array of processors and a d-dimensional mesh.
TABLE OF CONTENTS

1. Introduction ............................................................................................................. 1
2. Algorithm for Unbounded and Bounded Models ...................................................... 10
3. Solving First Order Linear Recurrence System on the Hypercube ......................... 22
4. Performance on Various Architectures .................................................................. 42
5. Conclusion ............................................................................................................... 49

References .................................................................................................................... 50
Chapter 1

Introduction

An important aspect of research in parallel processing is the design of parallel algorithms. A parallel algorithm for a problem specifies how several processors may be brought to work on a single task to reduce the time required for its solution. In certain cases, the existing sequential algorithms for the problem, provide a basis for the design of a parallel algorithm. As a somewhat extreme example consider the problem of adding two vectors. In this case the independence of the operations on different vector elements provides a natural decomposition of the problem, for parallel processing. The sequencing of operations implied by the sequential algorithm for this problem does not imply the necessity of carrying out the operations in that order. On the other hand, there are several instances where it is necessary to follow the order of operations of the sequential algorithm in order to preserve its correctness.

As an example consider the solution of the first order linear recurrence problem, for which a straightforward sequential algorithm is presented in Fig. 1. A and B are known vectors, and it is desired to compute the output vector X.

Begin

\[ X[0] = B[0]; \]
For \( i = 1, 2, \ldots, n-1 \) do

\[ X[i] = A[i]*X[i-1] + B[i]; \]

End.

**Figure 1.**

In this case there is no obvious way to allow several processors to work on the problem to reduce the execution time. The difficulty of parallelizing known sequential solutions to computational problems has resulted in the search for efficient parallel algorithms. These algorithms
have as their primary goal a solution in which there are very few stages that necessarily have to be executed in serial order. Within each stage, all of the operations can be executed in parallel on different processors. For a sequential algorithm with a polynomial execution time, one is usually interested in finding an algorithm that uses only a polynomial number of processors, but achieves polylog execution parallel time, i.e. \( \log^k n \) for some \( k > 0 \). This line of research has led to the discovery of several parallel algorithms on a model commonly referred to as the unbounded model of parallel processing.

The unbounded model of parallel processing is one in which we assume we have a multiprocessor system with an unlimited number of processors. In such a model there is only one aim in the design of parallel algorithms - viz., to reduce the parallel execution time. Furthermore very few restrictions are placed on the capability of the processors to communicate with each other. The goal of research on such a model is to discover parallelism in problems which appear at first to require a large number of operations to be performed in sequential order.

In practice, a multiprocessor system differs from the idealized unbounded model in several respects. Firstly, the number of processors in the system at any time is fixed. Secondly, there are usually significant restrictions on the patterns of communication between processors that can be efficiently supported by the multiprocessor. Finally, overheads incurred in synchronization between stages of the parallel algorithm are neglected in the idealized model. We examine each of these issues in turn.

In a typical multiprocessor we are looking for parallel algorithms which can provide a linear decrease in the execution time. That is, if \( T_{seq} \) is the execution time for a problem of size \( n \) on a uniprocessor and \( T_p \) is the parallel execution time on a system with \( p \) processors, then a good parallel algorithm for such a bounded model of multiprocessor is one where \( T_p = c (T_{seq}/p) \), for some \( c \) independent of \( p \). We say such an algorithm achieves linear speedup. This problem of simultaneously achieving a small parallel execution time as well as achieving linear speed up is not always easy. For example \( p = 1 \), trivially satisfies the requirement of linear speedup. It is often not difficult to arrive at partitioning schemes for a problem,
whereby a small number of processors, can be employed to achieve linear speedup. However, doing so for a large value of \( p \) is often quite difficult. On the other hand, solutions in the unbounded model seldom achieve linear speedup, but have very small execution times. The reason for this is that small parallel times are often obtained at the expense of performing more operations than are required by the sequential algorithms.

The communication capabilities of different multiprocessors vary greatly. These can be roughly classified as shared memory models where processors communicate by reading and writing from a common shared memory or network models in which processors communicate by sending messages to each other. In both cases, there are a variety of submodels, differing in the capacity for simultaneous communications. Examples in the first category include the bus connected Balance Computer and the IBM RP3, which communicate using shared memory, but differ greatly in the amount of concurrency possible in the communication. Examples of the second category range from linear arrays like the CMU WARP to Intel's hypercube, which differ greatly in the communication patterns that can be effectively supported. It is important to design parallel algorithms with minimal communication requirements, so that they can be efficiently mapped onto the limited communication facilities provided by the systems.

Finally, in order to preserve the correctness of the parallel algorithm it may sometimes be necessary for processors to wait for synchronization. Such a situation may arise when a processor must read the value of a variable only after it has been updated by another processor. Contention for shared resources (e.g. simultaneous memory requests to the same module) and the consequent delay may be another cause for reduced performance.

Thus there are several factors which need to be considered in designing a good practical algorithm.

1. The total number of operations performed by the parallel algorithm should not (significantly) exceed those required by the best sequential algorithm for the problem.
2. The communication requirements should be minimized.

3. The number of synchronizations required and contention time should be minimized.

There is a relationship between the unbounded and bounded models which is known as Brent's Theorem. This theorem states that given a parallel solution which performs $X$ operations and takes time $T_{par}$ in an unbounded model, the algorithm can be simulated with $p$ processors, if we neglect all overheads incurred in assigning tasks to processors, in time $T_p$ bounded by:

$$T_p \leq \frac{X}{p} + T_{par}.$$ 

In other words if the number of operations performed by the parallel solution (i.e. $X$) is significantly greater than that performed by the best sequential algorithm, we can never hope to achieve linear speedup using this parallel algorithm.

In this thesis we present an optimal parallel solution to the problem of first order recurrence on a bounded system with $p$ processors. This solution is an improvement over previous solutions in terms of the operation count, the processing time, and its communication requirements, and is optimal along all three dimensions. The solution is valid for all values of $p \geq 1$ and simultaneously achieves linear speedup, requires the minimal execution time (up to constant factors) and performs the least amount of communication possible.

The time complexity for our algorithm is $O(\frac{n}{p} + \log p)$, where $n$ is the size of the recurrence system and $p$ is the number of processors. In particular using $\frac{n}{\log n}$ processors, the algorithm achieves $O(\log n)$ time. Since $\log n$ is known to be a lower bound on the execution time for this problem on any parallel machine (independent of $p$) our solution achieves optimal order time with $\frac{n}{\log n}$ processors. When the number of processors is less than $\frac{n}{\log n}$ the performance is $O(\frac{n}{p})$, which is optimal with respect to the number of processors.
We consider several models of multiprocessor systems. These include a linear array, $d$-dimensional arrays, a bus-connected shared-memory multiprocessor, and the hypercube. We analyze the performance of the parallel solution taking into account the communication delays. We show how the synchronisation required between successive stages of code can be dispensed with on a network supporting blocking receives.
Uniform Recurrence Systems

The solution of uniform recurrence equations is a problem that frequently arises in numerical algorithms or signal processing applications. This problem requires the determination of a sequence \( x_1, x_2, \ldots, x_n \) where each \( x_i \) is a function of the previous \( m \) \( x \)'s namely \( x_{i-1}, \ldots, x_{i-m} \). A common example is a time-varying linear system where the state of the system at time \( i \) is \( x_i \) and can be computed from the equations:

\[
x_1 = a_1 x_0 + b_1; x_2 = a_2 x_1 + b_2; \ldots
\]

where \( a_i \) and \( b_i \) represent the internal dynamics of the system and \( x_0 \) is the initial condition.

The equation used to compute \( x_i \) is called a recurrence equation, and together with initial values for some of the \( x_i \)'s represents a complete problem description. Formally, a recurrence problem consists of a set of recurrence equations \( x_i = f_i(x_{i-1}, \ldots, x_{i-m}), \) \( i = m+1, \ldots, n \) and some boundary values, which may consist of any of the following:

1. \( x_1, \ldots, x_m \) - This is an initial value problem.

2. \( x_{n-m+1}, \ldots, x_n \) - This is a final value problem.

3. A mixture of \( m \) initial and final values.

The above problem is referred to as the general \( m \)th order recurrence problem. When \( m = 1 \), each \( x_i \) is only dependent on the previous value i.e. \( x_{i-1} \), and the problem is referred to as the first order recurrence system.

The problem for which a parallel solution is proposed in this thesis is the solution of a first order linear recurrence system and is defined as follows.

**Problem Definition:**

Given \( x_0 = b_0 \), and the coefficients \( a_i, b_i, 1 \leq i \leq n-1 \), which relate \( x_i \) to \( x_{i-1} \) by:
\[ x_i = a_i \cdot x_{i-1} + b_i; \]

find the values of the variables \( x_1, \ldots, x_{n-1}. \)

**Previous Work**

The straightforward sequential solution (Fig. 1) to the first order linear recurrence system requires \( O(n) \) time. One of the first parallel solutions for the problem, in the unbounded model, was proposed by Kogge and Stone [1]. They proposed a parallel algorithm for the efficient solution of a general class of recurrence equations. This algorithm works on the principle of recursive doubling. Each term to be computed is expressed as the sum of two equally complex subterms. The individual subterms are then computed on different processors. This process of successive splitting is continued until the complexity of the subterms is reduced to a constant. This algorithm performs \( O(n \, \log n) \) number of operations and requires \( \log n \) parallel steps. Their solution is very demanding in terms of communication requirements. It assumes that each processor can communicate with every other processor in unit time, and that several such communications can be performed simultaneously. The total number of remote accesses to data made in the course of the algorithm (which we term the communication complexity) is \( O(n \, \log n) \). On a machine with limited communication facilities, like a single bus multiprocessor, the communication time prevents any significant speedup.

The number of processors, \( p \), required by the above solution has to be at least equal to \( n \).

If this condition is not met, the authors propose that the algorithm be used \( \frac{n}{p} \) times to calculate \( p \) elements of the series at a time. This implies that the time for solution on \( p \) processors is \( O\left( \frac{n}{p} \log p \right) \). Hence this solution never achieves linear speedup for any \( p > 1 \). In particular, when using \( \frac{n}{\log n} \) processors, the time complexity of the solution is \( O(\log^2 n) \).
Another solution to the linear recurrence problems was proposed by Chen and Kuck [2]. They proposed a solution to the general $m^{th}$ order recurrence system i.e. a system of equations wherein each variable is dependent on the value of $m$ previous variables. For a system of $n$ equations their solution requires a minimum of $O(m^2 \cdot n)$ processors and the execution time is $O(\log (m + 1) \cdot \log n)$. For the first order case, their solution requires $O(\log n)$ time and requires $n$ processors. The communication requirements match those of the algorithm by Kogge and Stone [1].

Two of our goals in parallel processing is to achieve the minimum processing time and to use the minimum number of processors. Of course it may not be possible to achieve both of these ends simultaneously. The solutions discussed above have been able to obtain the optimum time but with a minimum of $n$ processors. In practice it may be impractical to have such a large system. In such a case one would like to have solutions which are optimal up to a constant factor with respect to the available number of processors. Thus in the case of the first order problem this would mean an $O(\frac{n}{p})$ execution time using $p$ processors.

Gajski [3] proposed an algorithm for solving linear recurrence systems on parallel and pipelined machines. Given a system with $p$ processors, this solution solves a system of $n$ first order recurrence equations in time $O(\frac{n}{p})$. However this solution is only valid for the range $1 \leq p \leq \sqrt{n}$. This is generally a problem in parallel processing. It becomes increasingly difficult to efficiently utilize very large numbers of processors.

A recent solution which approaches the goals outlined is the one proposed by Carlson and Sugla [4]. Their approach is one of reorganizing the binary tree associated with computations. Given a system with $p$ processors they present a parallel algorithm to compute every $^nC_p^{th}$ $x_i$ (i.e. a total of $p \times s$) in $O(\frac{n}{p} + \log p)$ time. In order to compute all the $n$ values of the $x_i$'s, their solution requires the use of $p \log p$ processors. Alternatively stated, with $p$ processors it requires time $O(\frac{n \cdot \log p}{p} + \log p)$ time to compute all $n$ values of $x_i$. Ideally, one would like an algorithm which solves for all the $n$ values in $O(\frac{n}{p} + \log p)$ time using $p$. 
processors. Algorithm A which we present in the next chapter is a step in that direction.
Chapter 2

Algorithm on Unbounded and Bounded Model

In this chapter we first present a parallel algorithm for the solution of the first order linear recurrence on the unbounded model of parallel computation. Using \( n \) processors, we solve the first order recurrence system in \( 2\log n \) parallel steps, i.e. in \( O(\log n) \) time. In contrast to previous algorithms [1,2] which require \( O(n \log n) \) operations and have \( O(n \log n) \) communication complexity, this algorithm requires only a total of \( O(n) \) operations and has \( O(n) \) communication complexity. Note that this algorithm can be simulated in time \( O(\frac{n}{p} + \log n) \) time using any fixed number, \( p \), of processors. In a later section, we use this algorithm to obtain a parallel solution that is more suitable for a fixed-size multiprocessor than direct simulation.

Informal Description

Consider a system of \( n \) first order linear recurrence equations to be solved on a system with \( n \) processors. For convenience we will assume that \( n = 2^k \), for some positive integer \( k \). The system of equations is given by:

\[
x_0 = b_0, \quad \text{and,} \\
x_i = a_i x_{i-1} + b_i, \quad 1 \leq i \leq n-1.
\]

It is convenient to view the computation by means of a sequence of dependency graphs \( G_0, G_1, \ldots, G_i, \ldots \). Node \( i, 0 \leq i \leq n-1 \), in \( G_i \) represents the variable \( x_i \). With node \( i \) we associate a label \( s_i \) equal to the pair \( <p_i, q_i> \). (The meaning of \( p_i \) and \( q_i \) will be explained presently.) There is a directed edge (referred to as an arc) from node \( i \) to node \( j \) whenever

\[
x_i = p_i x_j + q_i.
\]

That is, \( x_i \) can be directly calculated if \( p_i, q_i \) and \( x_j \) are known. In this case we say that \( x_i \) is directly dependent on \( x_j \).
The dependency graph at the start of the computation, $G_0$, consists of the nodes $i$, $0 \leq i \leq n-1$, and the arcs $(i, i-1)$, $1 \leq i \leq n-1$. The labels $s_i = <a_i, b_i>$, $i = 1, \ldots, n-1$. Figure 2 depicts $G_0$ for the case $n = 16$.

We define an operation $\text{jump}$ on the vertices of $G_t = <V_t, E_t>$ as follows. Let $i \in V_t$ such that $(i, j), (j, k) \in E_t$. Then the jump operation on $i$ consists of the steps:

i) replace arc$(i, j)$ by arc$(i, k)$ and ii) update $s_i = <p_i, q_i>$ to $<p_j, p_j, q_i + p_i, q_j>$.

We also define a second function $\text{compute}$ on the vertices of $G_t$ as follows. Let $i \in V_t$, such that $(i, j) \in E_t$ and $(j, k)$ not $\in E_t$ for any node $k$. The compute operation on $i$ consists of the steps:

i) remove arc$(i, j)$ and ii) evaluate $x_i = p_i + x_j + q_i$.

In terms of the recurrence, these operations correspond to backward and forward substitutions, respectively. For instance, suppose $x_i = p_i x_j + q_i$, and $x_j = p_j x_k + q_k$. The jump operation on $x_i$ corresponds to substituting for $x_j$ in the equation for $x_i$. Performing this substitution yields:

\[ x_i = (p_i p_j) x_k + (p_i q_j + q_i). \]

By updating $p_i$ and $q_i$ to the quantities between the first and second pairs of parentheses respectively, the jump operation results in $x_i$ being directly expressed in terms of $x_k$. The compute operation is even simpler. Suppose the value of $x_j$ is known and that $x_i$ has been expressed in terms of $x_j$. Performing a compute on $i$ corresponds to evaluating $x_i$ by substituting the value of $x_j$ in the equation relating $x_i$ and $x_j$.

The algorithm consists of two phases referred to as the $\text{Up Sweep}$ and $\text{Down Sweep}$, respectively. The Up Sweep consists of $k$ iterations. The $i^{th}$ iteration of the Up Sweep transforms $G_t$ to $G_{t+1}$, by performing jump and compute operations on different vertices of $G_t$.

The input to the Down Sweep is the dependency graph $G_k$.

The Down Sweep consists of $k - 1$ iterations. In the $i^{th}$ iteration of Down Sweep, the graph $G_{n-k-i+1}$ is transformed to $G_{n-k+i}$ by performing compute operations on selected vertices
of \( G_{s+k-1} \).

Let \( b \) be a loop variable. In the first iteration of Up Sweep \((b = 0)\), \( n/2 - 1 \) vertices, viz. \( 3, 5, 7, \ldots, n - 1 \) are selected to perform a jump operation. In the same iteration, vertex 1 performs a compute operation and evaluates \( x_1 \). (See Figure 3.) At the end of this iteration, \( x_3, x_5, \ldots, x_{n-1} \) have direct dependencies on \( x_1, x_3, \ldots, x_{n-3} \) respectively. In the next iteration \((b = 1)\), \( n/4 - 1 \) vertices viz. 7, 11, 15, \ldots, \( n-1 \), are selected to perform a jump. In the same iteration, vertex 3 performs a compute and evaluates \( x_3 \) using the known value \( x_1 \). In general, when the loop variable has value \( b \), the vertex \( i = 2^{b+1} - 1 \) (only one such vertex exists) performs a compute using the already computed value of the variable \( x_j \) where \( j = 2^b - 1 \). Simultaneously \((n/2^{b+1}) - 1 \) vertices perform a jump operation. This process is continued for \( k = \log n \) steps. Thus at the end of \( k \) steps, \( x_1, x_3, x_7 \ldots x_{n-1} \) are known. The dependencies in \( G_k \) at the end of the Up Sweep are as follows: \( n/2 - 1 \) vertices are directly dependent on a vertex differing in index by 1, \( n/4 - 1 \) vertices are directly dependent on vertices differing in index by two, and so on, with 1 vertex depending on a vertex \( n/4 \) indices away from it. Note that in the last step of the Up Sweep \( x_{n-1} \) is evaluated.

In the Down Sweep, the dependencies set up in the Up Sweep are used to evaluate the remaining \( x_i \)'s. In the first iteration of the Down Sweep, all the vertices which had edges directed into vertices \( n/4 \) indices away, perform a compute and evaluate their respective \( x_{7} \). In the next iteration, all vertices which had dependencies on vertices \( n/8 \) indices away compute and evaluate their respective \( x_i \)'s. Thus at the end of \( \log n - 1 \) iterations of the Down Sweep, the values of all the \( x_i \)'s, \( 0 \leq i \leq n-1 \), are known and the algorithm terminates.

Formal Description

We now describe the algorithm more formally. The number of processors in the system is \( n = 2^k \). Later we discuss the case when the number of processors is not a power of two.
Node $i$ of $G_0$ is associated with processor $i$ (denoted by $P_i$).

We define the condition $C(r)$ for the selection of processors in iteration $r$ as follows:

$$C(r) = \{ j \mid j = (2^r - 1) \mod 2^{r+1}, \text{ and } j \neq 2^r - 1 \},$$

where,

$$m = n \mod b \Leftrightarrow m \text{ and } n \text{ are congruent modulo } b.$$ 

$C(r)$ is the set of all $j$'s which satisfy the above condition.

Note that $|C(r)| = 2^{k-r} - 1$. For $n = 16$ one may verify that

$C(0) = \{2, 4, 6, 8, 10, 12, 14\}$

$C(1) = \{5, 9, 13\}$

$C(2) = \{11\}$

$C(3) = \{\}$

$X[i,j], P[i,j], Q[i,j]$ are memory locations accessed by $P_j$. $P[j]$ and $Q[j]$ are initialized to $a_j$ and $b_j$ respectively, for all $j$, $1 \leq j \leq n-1$, and $X[0]$ is initialized to $b_0$.

Algorithm A is divided into two phases: The Up Sweep and the Down Sweep. The Up Sweep involves $\log n = k$ iterations and the Down Sweep involves $k - 1$ iterations. As discussed previously, during the Up Sweep direct dependencies are set up between variables $x_i$ and $x_j$, where $i$ and $j$ differ by a power of two. Also during this phase the values of the variables $x_j$ are evaluated for $j = 2^m - 1$, where $0 \leq m \leq k-1$. During the Down Sweep, we utilise the dependencies set up in the Up Sweep to compute the values of the remaining $x_j$'s.

**Algorithm A**

Begin
Up-Sweep-A;
Down-Sweep-A;
End.
Up-Sweep-A

for b = 0 to k-1 do

/* Processors synchronize at the start of the iteration */
begin

/* Select processors that perform a jump operation */
if \((j \in C(r) \text{ or } j = 2^{r+1} - 1)\), for all \(r, \ b+1 \leq r \leq k\), do in parallel
begin

\[ P[j] \leftarrow P[j] \cdot P[j - 2^b]; \]
\[ Q[j] \leftarrow Q[j] + P[j] \cdot Q[j - 2^b]; \]
end

if \((j = 2^{b+1} - 1)\) do /*For this one processor \(P_j\), evaluate the value \(x_j\)*/
begin

\[ X[j] \leftarrow P[j] \cdot X[j - 2^b] + Q[j]; \]
end

end for

/* End of Algorithm Up-Sweep-A */

Down-Sweep-A

for b = k-2 downto 0

/* Processors synchronize at the beginning of the iteration */
begin

if \((j \in C(b))\) do in parallel

/* Calculate the values of \(x_j\) */
begin

\[ X[j] \leftarrow P[j] \cdot X[j - 2^b] + Q[j]; \]
end end for
/* End of Algorithm Down-Sweep-A */

Correctness and Analysis

In this section we illustrate the working of the algorithm for $n = 16$ and performs $O(n)$ operations. A small modification of the algorithm permits efficient implementation on a hypercube. The modified algorithm is presented in Chapter 3, where a formal proof of correctness is also provided.

Consider the processors that perform a jump operation in the $b^{th}$ iteration, $0 \leq b \leq k-1$. For $b = 0$, the processors are those in $C(1)$, $C(2)$, and $C(3)$ as well as processors $2^2 - 1$, $2^3 - 1$ and $2^4 - 1$. These correspond to the processors 5, 9, 13, 11, 3, 7 and 15. The processor that performs the compute operation is processor $2^1 - 1$, which is processor 1. It may be verified from Figure 3 that this is faithful to the informal description. In the iteration when $b = 1$, the processors performing a jump are those in $C(2)$ and $C(3)$ and processors 7 and 15 i.e $P_7$, $P_{11}$ and $P_{15}$. The processor performing a compute in this iteration is $P_3$. When $b = 2$, the processors performing a jump are those in $C(3)$ (which is an empty set) and $P_{15}$. A compute is performed by $P_7$. Finally, in the last iteration, only $P_{15}$ does anything, and performs a compute.

We perform a similar trace for the Down Sweep. Iterations in this phase correspond to values of the loop index variable from 2 down to 0. For $b = 2$, the processors that perform a compute are those in $C(2)$, which in this case is only $P_{11}$. It uses the value computed by the processor with index $11 - 2^2$, i.e $P_7$. It may be seen that at the end of the Up Sweep, $x_{11}$ had a direct dependence on $x_7$ and that the value of $x_7$ was known. For $b = 1$, processors which perform a compute are those in $C(1)$, i.e $P_5$, $P_9$ and $P_{13}$. The variables $x_5$, $x_9$ and $x_{13}$ had direct dependences on the variables $x_3$, $x_7$ and $x_{11}$ at the end of the up sweep. The values of $x_3$ and $x_7$ were known at the end of the up sweep and $x_{11}$ was computed in the first iteration of down sweep. For $b = 0$, the processors $P_0$, $P_2$, $P_4$, $P_6$, $P_8$, $P_{10}$, $P_{12}$ and $P_{14}$ perform a
compute operation.

The number of operations performed by the algorithm can be computed as follows. In the Up Sweep, the number of jump operations performed in the $b^{th}$ iteration is

\[
\sum_{r=b+1}^{k-1} (|C(r)| + 1)
\]

\[
= \sum_{r=b+1}^{k-1} 2^{k-r-1}
\]

\[
= 2^{k-b-1} - 1
\]

The number of computes performed in the $b^{th}$ iteration is 1. Hence the number of operations (i.e., the number of jump operations plus the number of compute operations) in the $b^{th}$ iteration is $2^{k-b-1}$. The total number of operations performed in Up-Sweep-A is therefore given by:

\[
\sum_{b=0}^{k-1} 2^{k-b-1} = 2^k - 1 = n - 1.
\]

The analysis for Down-Sweep-A is similar. The number of compute operations (and hence the total number of operations, since there are no jump operations in the Down-Sweep) performed in the $b^{th}$ iteration is $|C(b)|$. Summing over all values of $b$ from $k-2$ to 0, we get for the number of operations in Down-Sweep-A:

\[
\sum_{b=0}^{k-2} 2^{k-b-1} - 1
\]

\[
= 2^k - k - 1 = n - \log n - 1.
\]

The total number of operations performed by Algorithm A is therefore $2n - \log n - 2$ which is O(n).

The analysis for the time complexity is obvious. Up-Sweep-A consists of $k$ iterations and Down-Sweep-A consists of $k-1$ iterations. Within each iteration, a processor does at most one operation, requiring O(1) time. Hence, the time complexity is O(log $n$).

Since, only O(n) operations are performed the number of remote accesses to data that are made (the communication complexity) is also bounded by O(n).
Notes

In the discussion above, we assumed that the number of processors (which is equal to the number of data elements) is a power of two. It is easy to see that the assumption is not really restrictive. If $n$, the number of equations satisfies $2^{k-1} < n \leq 2^k$, we assume that we have $2^k$ processors, of which $P_n, P_{n+1}, P_{n+2}, \ldots, P_{2^k-1}$ are to evaluate the non-existent values $x_n, x_{n+1}, x_{n+2}, \ldots, x_{2^k-1}$. We run the algorithm unchanged -- $k$ iterations for Up-Sweep-A and $k-1$ iterations for Down-Sweep-A. (Of course, the synchronization at the start of each iteration should involve only actual processors $P_0, \ldots, P_{n-1}$.) The number of iterations in Up-Sweep-A and Down-Sweep-A will therefore be $\lceil \log n \rceil$ and $\lfloor \log n \rfloor - 1$ respectively. Therefore in the worst case we increase the number of iterations by at most an additive factor of two. In contrast, the obvious way of using $2^{k-1}$ of the available $n$ processors, would result in a worst-case degradation by a multiplicative factor of two. Furthermore, the program details would have to be appropriately modified.

Limited Number of Processors

In this section we present an algorithm for a bounded model of a multiprocessor system. The number of processors here is limited to $p$ ($p < n$). The modified algorithm consists of three phases. The first and the last phases are performed independently and concurrently by each processor using data that is local to it. The second stage consists of Algorithm A executed on a $p$ processor multiprocessor.

Algorithm for a Bounded Model

Algorithm A proposed in the previous section solves the first order linear recurrence problem in $O(\log n)$ time using $n$ processors. It has $O(n)$ communication complexity. In practice, for large values of $n$, the actual number of processors may be less than this. In such a case the need arises for an algorithm which is adaptable to the available system size and provides a
speedup which is linear with respect to the number of processors in the system. In other words, although we may not be able to attain the minimum parallel execution time, we would like an algorithm which is optimal with respect to the system size. For the linear recurrence problem this would mean an execution time of $O\left(\frac{n}{p}\right)$. In this section we propose a strategy for achieving this bound.

One approach to achieving an execution time of $O\left(\frac{n}{p}\right)$ on a machine with less than $n$ processors is to simulate the operations of the original algorithm on the system with $p$ processors ($p < n$). Essentially, each operation may be considered as a process and scheduled on a processor. As discussed in Chapter 1, Brent's Theorem tells us that if there exists a parallel algorithm which performs $X$ operations and takes time $T_{par}$ one can simulate the algorithm on a machine with $p$ processors in time $T_p \leq X/p + T_{par}$. Since algorithm A performs $O(n)$ operations and takes time $O(\log n)$, it can be simulated in time $O\left(\frac{n}{p} + \log n\right)$.

There are, however, several disadvantages to this approach for achieving linear speedup on a system with a small number of processors. Firstly, the granularity of the simulation would be too fine. The overheads incurred in scheduling a process, which performs only a small operation when invoked, could easily dominate the actual processing time. Secondly, unless careful thought is given to deciding which processes will be scheduled on a processor, there could be a large amount of communication ($O(n)$ remote requests in the worst case). Depending on the communication structure of the machine one may lose any advantages of the parallel algorithm.

We propose now a parallel scheme that is a combination of serial and parallel processing. The scheme consists of three stages - a serial stage (in which processors independently and concurrently compute on data locally available to them), followed by a parallel phase where processors communicate with each other, finally followed by another serial stage where processors work independently and concurrently. On a system of $p$ processors, in the first stage we employ serial processing to construct a $p$ equation first order recurrence
system from the initial \( n \) equation system. The second stage employs Algorithm A, discussed in the previous section, to solve the \( p \) equation system using \( p \) processors. Finally, in the third stage all of the processors concurrently and independently perform serial processing to evaluate all the \( n \) variables.

Let \( n \) be the number of equations and \( p \) the number of processors where \( n = m_p \). The system of equations is given by \( x_0 = b_0 \) and \( x_i = a_i x_{i-1} + b_i \), \( 1 \leq i \leq n-1 \). Partition this system into \( p \) sets, each of which comprises of \( m \) consecutive equations. Now allocate the system equally to all the \( p \) processors, i.e. \( P_0 \) gets coefficients corresponding to equations 0 through \( m-1 \), \( P_1 \) gets coefficients corresponding to equations \( m \) through \( 2m-1 \), and so on, with \( P_m \) getting the coefficients of equations \((p-1)m \) through \( pm - 1 \). In general, \( P_i \) gets the coefficients corresponding to equations \( im \) through \((i+1)m - 1 \).

We define a new set of variables \( w_i = x_{(i+1)m-1} \), \( 0 \leq i \leq p-1 \). Using the \( m \) sets of coefficients allocated to it, \( P_i \), \( (i > 0) \) calculates new variables \( u_i \) and \( v_i \). These express the variable \( w_i \) directly in terms of \( w_{i-1} \), i.e.

\[
w_i = u_i \times w_{i-1} + v_i, \quad 1 \leq i \leq p-1.
\]

\( P_0 \) computes \( w_0 \) using \( x_0 \) and the coefficients \( a_j, b_j \), \( 1 \leq j \leq m-1 \). \( P_i, i > 0 \), performs back substitution to reduce the system of equations allocated to it, to a single equation relating \( w_i \) to \( w_{(i-1)} \). The algorithm for this phase is presented in Figure 4 below.
Algorithm executed by $P_i$, $1 \leq i \leq p - 1$

Begin

$$u_i = a_{(i+1)m-1};$$

$$v_i = b_{(i+1)m-1};$$

For $j = 1$ to $m - 1$ do

Begin

$$v_i = v_i + u_i (b_{(i+1)(m-1-j)});$$

$$u_i = u_i (a_{(i+1)(m-1-j)});$$

End

End /* End of Algorithm */

Figure 4

$P_0$ executes the algorithm of Figure 1, with the limit of the iteration replaced by $m - 1$. The time taken by $P_i$, $i > 0$, to perform this back substitution, and by $P_0$ to evaluate $x_{m-1}$ is clearly $O(m)$. In terms of the dependency graph notation introduced earlier, the dependencies between variables at the end of this stage is shown in Figure 5, for the case $n = 16$ and $p = 4$.

The next stage computes the value of the variables $w_i$, $0 \leq i \leq p - 1$. The problem here is to solve a system of $p$ equations (comprising of the equations $w_i = u_i + w_{i-1} + v_i$, $0 \leq i \leq p - 1$) with $p$ processors. Each of these variables is related by a direct dependency to the variable which is indexed one less than it and $w_0$ is known. The distribution of the coefficients of this system are exactly those required by algorithm A. Therefore, at this stage we apply the parallel algorithm A discussed in the previous section, to compute $w_i$, $1 \leq i \leq p - 1$. This requires time $O(log p)$.

At the end of the second stage all the values of $w_i$, $0 \leq i \leq p - 1$, are known. In one parallel step, each $P_i$, $i > 0$, reads the computed value of $w_{i-1}$ from $P_{i-1}$. Thus, each such $P_i$ has available the value of $x_{im-1}$. 

In the third and final stage of the algorithm, each of the processors computes the values of the remaining \((m-1)\) \(x_i\)'s in the system allocated to it, using forward substitution, i.e., using the algorithm of Figure 1 with the indices appropriately adjusted. This takes each of the processors time bounded by \(O(m)\).

Thus at the end of the three stages, the value of all the variables \(x_i\), \(0 \leq i \leq n-1\) are known. The total time taken by the algorithm is bounded by the sum of the times for each stage, and is given by \(O(m + 2 \log p + m)\), which is \(O\left(\frac{n}{p} + \log p\right)\). Thus we have realized a parallel solution for the first order linear recurrence problem which takes optimal execution time (up to small constants) with respect to the available number of processors.

**Summary**

In this chapter we first presented an algorithm (Algorithm A) that solves an \(n\) equation first order recurrence system in \(O(\log n)\) time using \(n\) processors. The advantage of this algorithm over previous schemes is that it performs only a total of \(O(n)\) operations and has \(O(n)\) communication complexity. We then presented a scheme that solves the problem in time \(O\left(\frac{n}{p} + \log p\right)\), when the number of processors \(p\), is less than \(n\). This scheme is a combination of two phases of serial processing (which are independently and concurrently performed in each of the processors) and a phase in which processors communicate with each other. The latter phase is implemented using the operation and communication efficient parallel algorithm A.

In the next chapter we discuss the solution of an \(n\) equation first order linear recurrence system on a \(p\) processor hypercube. We also show how the global synchronization required at various steps of the algorithm can be avoided if the machine supports blocking receives.
Figure 2 G0

G1

G2

G3

G4

G5

G6

G7

Figure 3
Figure 5. $n=16, p=4$
Chapter 3

Solving First Order Linear Recurrence System on the Hypercube

In this chapter we consider the problem of mapping the parallel algorithm for the solution of a first order linear recurrence system on a hypercube. We first discuss the problem in directly implementing Algorithm A of the previous chapter on a hypercube; we then present a modified algorithm, Algorithm B, which maps directly onto such a machine. A formal proof of the correctness and an analysis of the time complexity is provided. Finally, we present an asynchronous implementation where global synchronization between stages of the algorithm is not needed. This implementation assumes a communication protocol that supports blocking receives.

Hypercube Model:

A $k$-dimensional hypercube is a processor network having $p$ processors, where $p = 2^k$, $k \geq 0$. In such a system every processor is connected to $k$ (i.e. $\log p$) other processors, as follows. Let the binary representation of an integer $j$, $0 \leq j \leq 2^k - 1$, be denoted by the $k$-tuple $<j_{k-1}, j_{k-2}, j_{k-3}, \ldots, j_1, j_0>$. The processors have indices in the range $0$ to $p - 1$. Denote the processor with index $j$ as $P_j$. $P_j$ is directly connected to $P_i$ if and only if the binary representations of $j$ and $i$ differ in exactly one bit.

Consider a 4-dimensional hypercube, consisting of 16 processors. For this case, each processor would be connected to four other processors. The interconnection between processors is shown in Figure 6.

We define $\tau$ to be the time necessary to transmit a unit of data between processors which are directly connected by a link. The exact size of a unit of data is left unspecified in our model. With respect to our algorithm the largest quantity of data that must be handled in one communication step consists of two coefficients of the system of equations. We therefore consider this as a unit of data. We also denote by $t$ the time to perform the two multiplications
and one addition of coefficients required in a jump operation.
Solving the Recurrence System

The strategy to solve an $n$ equation first order recurrence system on a $p$ processor hypercube, follows the decomposition method presented in the previous chapter. The equations are partitioned equally among the processors; the processing consists of two serial phases which bracket a phase in which processors communicate. Since, once the data is distributed to the processors, the first and third phases are not dependent on the processor interconnection, we are only concerned with the implementation of the second phase on a network.

Consider mapping algorithm A directly on a $p$ processor hypercube. During the Up Sweep we find that processors which communicate with each other are always directly connected. (This statement is formally justified later in this chapter). For now, by examining Figures 3 and 6, one can see the correctness of this statement for $p = 16$. Hence it takes only $\tau$ units of time for communication during each iteration of Up-Sweep-A.

Unfortunately, during the Down Sweep we find that processors which need to communicate do not always have a direct link between them. A clear example of this in a 16 processor hypercube is $P_8$ which requires $x_7$ from $P_7$ in the last iteration of Down-Sweep-A. There is no direct link between $P_7$ and $P_8$. In fact, the shortest communication path between $P_7$ and $P_8$ requires four link traversals. One such communication path which could be followed is $P_7$ to $P_{15}$ to $P_{11}$ to $P_9$ to $P_8$. This data transfer would take time at least $4 \tau$, even assuming that all other data transfers taking place at this step can be arranged so as to not interfere with each other.

In general for a $p$ processor hypercube, it may be necessary to transmit the data over $\log p$ links on the last iteration of Down-Sweep-A, $\log p - 1$ links in the iteration before the last and so on. In general, the delay between successive iterations of Down-Sweep would have a growth rate of approximately $\tau \log p$ units of time. This results in an $O(\log^2 p)$ time implementation of algorithm A on a $p$ processor hypercube.
We propose a modification of algorithm A that requires only a constant delay between successive iterations of Down-Sweep-A. The routing is planned in such a manner that the communication required in any iteration of the Down Sweep is always between processors which are directly connected. Hence the delay between successive stages will be no more \( \tau \). We call this modified version Algorithm B and show that it maps efficiently onto a hypercube.

**Hypercube Algorithm**

In this section, we present algorithm B for the hypercube. Like algorithm A of Chapter 2, it consists of two phases denoted by Up-Sweep-B and Down-Sweep-B respectively. The algorithm follows the same principles as Algorithm A, performing jump and compute operations on the same sets of vertices of the dependency graph. The variables \( X[j] \), \( P[j] \) and \( Q[j] \) are as defined for algorithm A. The new variable \( Temp[j] \) associated with \( P_j \) is used to route computed \( x_i \)'s, to processors which cannot access them by a direct link.

**Algorithm B**

Begin
Up-Sweep-B;
Down-Sweep-B;
End.

**Up-Sweep-B**

For \( b = 0 \) to \( k-1 \) do

/* Processors synchronize at the start of the iteration */

begin

/* Select processors that perform a jump operation */

if \((j \in C(r) \text{ or } j = 2^{r+1} - 1)\), for all \( r \), \( b+1 \leq r \leq k-1 \), do in parallel

begin
P[i] ← P[i] * P[i - 2^b];
Q[i] ← Q[i] + P[i] * Q[i - 2^b];

end

if (j = 2^b+1 - 1) do /* For this one processor P_j, evaluate the value x_j */
    begin
        Temp[i] ← X[i - 2^b];
        X[i] ← P[i] * Temp[i] + Q[i];
    end

end for

/* End of algorithm for Up Sweep */

Down-Sweep-B

For b = k-2 downto 0

/* Processors synchronize at the beginning of the iteration */

begin

if (j ∈ C(b)) do in parallel

/* Calculate the values of x_j */

begin

    Temp[i] ← Temp[i + 2^b]
    X[i] ← Q[i] + P[i] * Temp[i];

end

if (j ∈ C(r) or j = 2^r+1 - 1, b+1 ≤ r ≤ k-1)

/* Processors synchronize at the start of this stage*/

/* Read the value to be transmitted in the next iteration */

begin

    Temp[i] ← X[i - 2^b]

end
end

end for

/* End of algorithm B : Down Sweep */

/* End of Hypercube algorithm */
For the case \( p = 16 \), we discuss the computation of \( x_9 \). First notice that Up-Sweep-B is essentially the same as Up-Sweep-A; the only difference is that the processors which have computed the final values of their \( x \) variables (i.e., \( P_1, P_3, P_7 \), and \( P_{15} \)) have also stored certain information in their respective \( Temp[\cdot] \) variables. For instance, \( Temp[j] \) for any such \( P_j \) is the \( x_k \) which \( P_j \) used to compute the final value of \( x_j \). In particular, \( P_{15} \) which computed \( x_{15} \) in the last iteration using \( x_7 \) has \( Temp[15] = x_7 \). Similarly, \( Temp[7], Temp[3] \) and \( Temp[1] \) contain \( x_3, x_1 \) and \( x_0 \) respectively.

In the first iteration of Down-Sweep-B, \( P_{11} \) obtains the value stored in \( Temp[15] \) (which is \( x_7 \)), and thereby computes \( x_{11} \). Note that there is a direct link between \( P_{11} \) and \( P_{15} \), but no such link between \( P_{11} \) and \( P_7 \). After computing \( x_{11} \), this value is written back to \( Temp[15] \) of \( P_{15} \). Thus, at the end of the first iteration of Down-Sweep-B, \( Temp[15] \) has the value \( x_{11} \), and \( Temp[11] \) has the value \( x_7 \). In the next iteration, \( P_5, P_9 \), and \( P_{13} \) wish to compute their final \( x \) values, by obtaining the values of \( x_3, x_7 \) and \( x_{11} \) respectively. At this time \( Temp[7] \) contains \( x_3 \) (set during the Up-Sweep-B), \( Temp[11] \) contains \( x_7 \) (set in the previous iteration) and \( Temp[15] \) contains \( x_{11} \) (set in the previous iteration, following the compute by \( P_{11} \)). Thus, by reading from the \( Temp[\cdot] \) location of the processor indexed two greater than its own index, \( P_5, P_9 \) and \( P_{13} \) obtain the required values of \( x \) to complete their computation. Without such a strategy \( P_5, (P_9 \) and \( P_{13} \) would require a two-link communication to get the value of \( x_3 \) (respectively, \( x_7 \) and \( x_{11} \)).

Correctness and Complexity Analysis

In this subsection, we formally demonstrate that a link exists between every pair of processors which communicate during the course of the execution of Algorithm-B on a \( 2^k \) processor hypercube. This would show that the algorithm terminates in \( O(\log p) \) time. In the next subsection, we show that when it terminates, the values of \( x_i \), \( 0 \leq i \leq p - 1 \), have been correctly computed.
Lemma 4.1:
(a) If \( j \in C(r) \), \( s+1 \leq r \leq k-1 \), then \( (j - 2^s) \in C(s) \).

(b) If \( j \in C(s) \), \( 0 \leq s \leq k-1 \), then \( (j + 2^s) \in C(r) \) or \( j = 2^{r+1} - 1 \), \( s+1 \leq r \leq k-1 \).

Proof:
(a) \( j \in C(r) \) implies
\[
j = \alpha \times 2^{s+1} + 2^r - 1, \quad \alpha > 0.
\]
Hence,
\[
j - 2^s = \alpha \times 2^{s+1} + 2^r - 2^s - 1.
\]
Since \( r > s \), \( 2^{s+1} \) divides \( 2^r \) and \( 2^{r+1} \).
Hence,
\[
j - 2^s = \beta \times 2^{s+1} + 2^r - 1, \quad \beta > 0.
\]
Hence, \( j - 2^s \in C(s) \).

Lemma 4.2:
If \( j \in C(r) \) and \( 0 \leq b \leq k-1 \), then \( j \) and \( j - 2^b \) are directly connected by a link for all \( r, b+1 \leq r \leq k-1 \).

Proof:
Let \( j \in C(r) \). Then \( j \) is of the form \( j = \alpha \times 2^{r+1} + 2^r - 1, \alpha \neq 0 \). The binary address of \( P_j \) is therefore \( (j_{r-1}, j_{r-2}, j_{r-3}, \ldots, j_{r+1}, 0, 1, \ldots, 1) \). The address consists of \( k \) bits and the least significant \( r+1 \) bits have the pattern 011..1; i.e., a zero followed by \( r \) consecutive ones. Now, \( 2^b \) has the \( b+1 \) bit binary representation 100..0, i.e., a one followed by \( b \) consecutive zeroes.

Since \( b < r \), the bit address of \( P_{j-2^b} \) is given by \( (j_{r-1}, j_{r-2}, j_{r-3} \ldots, j_{s+1}, 0, 0, 1, 1, \ldots, 1) \) differing in exactly the \( b^{th} \) bit from that of \( P_j \). It follows that \( P_j \) and \( P_{j-2^b} \) are directly connected.
Lemma 4.3:

If \( j = 2^r - 1 \) and \( 0 \leq b \leq k - 1 \), then \( P_j \) and \( P_{j-2^b} \) are directly connected by a link for all \( r, b+1 \leq r \leq k \).

Proof:

Follows from the observation that \( j \) and \( j - 2^b \) differ in exactly the \((b + 1)^{st}\) least significant bit.

By examining algorithm Up-Sweep-B and Lemmas 4.2 and 4.3, we see that all processors communicating in Up-Sweep-B are directly connected by a communication link.

Lemma 4.4:

If \( j \in C(b) \) and \( 0 \leq b \leq k - 1 \), then \( P_j \) and \( P_{j+2^b} \) are directly connected by a link.

Proof:

If \( j \in C(b) \), then the binary address of \( P_j \) is \( \langle j_{k-1}, j_{k-2}, j_{k-3}, \ldots, j_{b+1}, 0, 1, 1, \ldots, 1 \rangle \), i.e., the address consists of \( k \) bits and the least significant \( b+1 \) bits have the pattern 011...1; that is, a zero followed by \( b \) consecutive ones. Now \( 2^b \) has the \( b+1 \) bit binary representation 100..0, i.e., a one followed by \( b \) consecutive zeroes.

Hence, \( j \) and \( j + 2^b \) will differ in exactly the \((b + 1^{st})\)bit.

From algorithm Down-Sweep-B and Lemmas 4.3 and 4.4, it follows that all communication during this phase of the algorithm is between processors that are directly connected by a link.

The total time \( T(n, p) \) to solve the problem of size \( n \) on a \( p \) processor hypercube is bounded by the expression given below. This is based on the following assumptions. The time to load the coefficients and retrieve the results is not included. The time to communicate a pair of coefficients between directly connected processors is \( \tau \) and the time to perform two multiplications and one addition of the coefficients is \( r \). The synchronization time required at
various stages of algorithm B is assumed to be negligible. In a SIMD implementation, this assumption is justified. In an asynchronous environment where processors communicate by sending messages to each other, this algorithm can be implemented so that no global synchronization is needed (see the next section for such an implementation). In this case also the assumption is justified.

\[ T(n, p) = 2 \left[ \frac{n}{p} \right] \times t + (t + \tau) \times \log p + (\tau + t + \tau) \times (\log p - 1) \]

\[ = (2 \left[ \frac{n}{p} \right] + 2 \log p - 1) \times t + (3 \log p - 2) \times \tau. \]

**Proof of Correctness**

We now show that algorithm B correctly solves a first order linear recurrence system. The proof consists of two parts, one for Up-Sweep-B and one for Down-Sweep-B.

The following is known to be true from the initial conditions of the system.

**Initial Conditions:**

1. \(X[0] = x_0\) i.e. the value of the first variable is known.
2. \(x_i \rightarrow x_{i-1}\) for \(1 \leq i \leq n-1\), where the notation \(x_i \rightarrow x_j\) means that \(x_i\) has a direct dependence on \(x_j\).

**Lemma 4.5:**

At the end of the \(b^{th}\) iteration of Up-Sweep-B, \(0 \leq b \leq k-1\), the following conditions are true.

1. \(X[j] = x_j\), for all \(j = 2^{r+1} - 1, -1 \leq r \leq b\).
2. \(\text{Temp}[j] = x_{j-2r}\), for all \(j = 2^{r+1} - 1, 0 \leq r \leq b\).
[3] (a) $x_j \rightarrow x_{j-2^r}$, for all $j = 2^r + 1 - 1$ or $j \in C(r)$, $0 \leq r \leq b$.

[3] (b) $x_j \rightarrow x_{j-2^{r+1}}$, for all $j = 2^{r+1} - 1$ or $j \in C(r)$, $b + 1 \leq r \leq k - 1$.

Proof:

We will induct on $b$.

1. Base Conditions:

We show that the Lemma is true at the start of Up-Sweep-B, by substituting $b = -1$.

[1] Part 1 is true since $X[0] = x_0$ from the initial conditions.


[3] For $b = -1$, part 3(a) is vacuously true. Part 3(b) states that $x_i \rightarrow x_{i-1}$ for all $1 \leq i \leq k-1$. This is true from the initial state of the system.

Thus the base case is established.

2. Induction Step: As the induction hypothesis (IH) we assume [1] to [3] (denoted by IH1, IH2 and IH3 respectively) are true for $b = s - 1$. We shall show that it holds good after the $s^{th}$ iteration.

[1] From IH1, we know that $X[j] = x_j$ for all $j = 2^{r+1} - 1$, $r = -1, 0, 1, ..., s - 1$.

Since only $P_k$, $k = 2^{r+1} - 1$, updates the value of $X[i]$, these values are not altered in the $s^{th}$ iteration. Hence IH1 holds true for $-1 \leq r \leq s - 1$. Next we show that IH1 holds good for $r = s$. Consider processor $k = 2^{s+1} - 1$. From IH3(b), we have $x_k \rightarrow x_{k-2^r}$. Substituting for $k$, $x_k \rightarrow x_{2^{r-1}}$, i.e., if $x_{2^{r-1}}$ is made known to processor $k$ then it can compute the value $x_k$.

Looking back at the algorithm we see that processor $k = 2^{r+1} - 1$ reads the value $X[2^{r+1} - 1 - 2^r] = X[2^{s} - 1]$ from $P_{2^{r-1}}$. By IH1, $X[2^s - 1]$ is $x_{2^{s-1}}$. Thus, $x_{2^{s-1}}$ is computed by processor $k$ and stored in the location $X[2^{s+1} - 1]$. Hence IH1 holds.
for $r = s$. We have shown that IH1 holds true after the $s^{th}$ iteration.

[2] We now show that IH2 holds good at the end of the $s^{th}$ iteration. IH2 was true for $j = 2^{r+1} - 1$, $0 \leq r \leq s-1$, and the $\text{Temp}[]$ locations of these processors are not altered in the $s^{th}$ iteration. Hence IH2 holds for $0 \leq r \leq s-1$ at the end of the $s^{th}$ iteration. In the $s^{th}$ iteration, the $\text{Temp}[]$ value of processor $j = 2^{r+1} - 1$ is set to $X[j-2^r]$, which by IH1 is $x_{j-2^r}$. Thus IH2 holds for $r = s$ at the end of the $s^{th}$ iteration.

IH2 holds true after the $s^{th}$ iteration.

[3] We shall now show that IH3 holds. For $j = 2^{r+1} - 1$, $0 \leq r \leq s$ the dependencies represented by IH3 were true at the beginning of the $s^{th}$ iteration, and they are unchanged in the $s^{th}$ iteration. Similarly, for all $j \in C(r)$, $0 \leq r \leq s$, IH3 was true and remains true after the execution of the loop.

For $j$ belonging to $C(r)$ or for $j = 2^{r+1} - 1$, where $s+1 \leq r \leq k-1$, the dependencies are updated. From IH3(b), the dependency after the $(s-1)^{th}$ iteration was of the form $x_j \rightarrow x_{j-2^r}$. The dependency of $x_j$ is updated using the dependency of $x_{j-2^r}$ existing at the previous iteration. From Lemma 4.1, $j \in C(r)$ implies that $(j-2^r) \in C(s)$. If $j = 2^{r+1} - 1$ then also $(j-2^r) \in C(s)$. By IH3(b), the dependency of $x_{j-2^s}$ is on $x_{j-2^s-2^r}$, which is $x_{j-2^s-1}$. Thus, at the end of the $s^{th}$ iteration, $x_j \rightarrow x_{j-2^s}$. Thus IH3 holds true after the $s^{th}$ iteration.

Hence, Lemma 1 holds good, and we may use the conditions existing at the end of Up-Sweep-B as a set of initial conditions for Down-Sweep-B.
Correctness of Down-Sweep-B:

Initial conditions:

The following hold good at the end of Up-Sweep-B.

1. \( X[j] = x_j \), for all \( j = 2^{r+1} - 1, r = -1, 0, ..., k-1 \).

2. \( \text{Temp}[j] = x_{j-2^r} \), for all \( j = 2^{r+1} - 1, r = 0, ..., k-1 \).

3. \( x_j \rightarrow x_{j-2^r} \), for all \( j \in C(r) \) or \( j = 2^{r+1} - 1, r = 0, ..., k-1 \).

Lemma 4.6:

The following conditions hold true at the end of the iteration when the loop index has value \( b \), \( (b \) ranges from \( k-2 \) down to \( 0 \) \) of Down-Sweep-B.

1. \( X[i] = x_j \)
   
   \[ \text{if } j = 2^{r+1} - 1, \quad r = -1, 0, ..., k-1; \]

   \[ \text{if } j \in C(r), \quad r = b, ..., k-1; \]

2. \( \text{Temp}[i] \)

   \[ (a) = x_{j-2^r} \text{ for all } j \text{ satisfying either of the following conditions:} \]

   \[ j \in C(r), b \leq r \leq k-1 \text{ or} \]

   \[ j = 2^{r+1} - 1, b+1 \leq r \leq k-1. \]

   \[ (b) = x_{j-2^r} \]

   \[ \text{if } j = 2^{r+1} - 1, 0 \leq r \leq b. \]

Proof:

By induction on \( b \).

We show that the Lemma holds at the beginning of the Down-Sweep-B. Substitute \( b = k-1 \) to check base conditions:
[1] From initial condition 1, IH1 holds good for \( j = 2^{r+1} - 1 \), for \( r = -1, 0, \ldots, k-1 \).

The set \( C(k-1) \) is empty and hence this portion of IH1 holds vacuously.

[2] On substituting \( b = k-1 \), there is no \( j \) satisfying any of the conditions of IH2(a). Hence, this assertion is vacuously true. IH2(b) follows from initial condition 2.

We now complete the proof by inducting on the value of \( b \). We assume that the Lemma is true for \( b = s+1 \) and we show that it remains true after the next iteration i.e. \( b = s \).

[1]

For all those \( j \) selected by the first condition of IH1, the claim holds good after the \( s^{th} \) iteration, since it held good at the end of the \((s+1)^{th}\) iteration and it is not changed in the \( s^{th} \) iteration.

For the elements selected by the second condition of IH1, we note that if \( j \in C(r), r = s+1, \ldots, k-1 \), then they satisfied IH1 at the end of the \((s+1)^{th}\) iteration and these elements are not altered in the \( s^{th} \) iteration. Hence IH1 holds true for them.

For \( j \in C(s) \), the value of \( X[j] \) is altered in this iteration. From initial condition 3 (which holds good throughout the Down Sweep) \( x_j \rightarrow x_{j-2^r} \).

Let \( z = j + 2^r \). Consider the value of Temp \([z]\). By Lemma 4.1(b), \( z \in C(r) \), or \( z = 2^{r+1} - 1, s+1 \leq r \leq k-1 \).

In the first case, at the end of the \((s+1)^{th}\) iteration, Temp \([z]\) contained \( x_{z-2^r} \) (from the second condition of IH2(a)) i.e Temp \([z]\) had \( x_{j-2^r} \). In the second case, IH2(a) (first condition) asserts that Temp \([z]\) contained \( x_{z-2^r} \) i.e \( x_{j-2^r} \).
Since \( X[j] \) is updated using \( Temp[z] \) and \( x_j \rightarrow x_{j-2^r} \), IH1 will hold after the \( s^{th} \) iteration.

[2]

In the \( s^{th} \) iteration, for \( j \in C(s) \), \( Temp[j] \) gets updated to the value that \( Temp[j + 2^r] \) had at the end of the \((s+1)^{th}\) iteration. From the proof of part 1 above, we can conclude that at the end of the \( s^{th} \) iteration, \( Temp[j] \) will have the value \( x_{j-2^r} \).

Hence IH2 holds good for all \( j \in C(s) \).

For \( j \in C(r) \), \( 0 \leq r \leq s-1 \), IH2 was true before the beginning of the iteration and these values do not get changed. Hence IH2 remains true for these.

For \( j \in C(r) \) or \( j = 2^{r+1} - 1 \), \( s+1 \leq r \leq k-1 \), \( Temp[j] \) is updated by assigning to it \( X[j - 2^s] \) calculated in the first part of this iteration. From part 1 of the proof, we know that \( X[j - 2^s] = x_{j-2^r} \).

Hence IH2 holds good for these values of \( j \) as well.

Thus we have shown that both IH1 and IH2 hold good after the \( s^{th} \) iteration.

**Theorem 4.1**:

When Algorithm B terminates, \( X[j] = x_j \), for all \( j \), \( 0 \leq j \leq k-1 \) and \( Temp[j] = x_{j-1} \) for all \( j \), \( 1 \leq j \leq k-1 \).

**Proof**:

Immediately follows by substituting \( b = 0 \) in Lemma 4.6.

It may be noticed from Theorem 4.1 that at the termination of algorithm B, \( P_j \) has the value of \( x_{j-1} \) in \( Temp[j] \). Hence, when used as the middle phase of an algorithm on a
bounded processor hypercube, the third stage (which evaluates the $x$'s using forward substitution) can immediately begin executing.

Message-Based Implementation

In this section we describe an message-based implementation of algorithm B on a $p = 2^k$ processor hypercube. A motivation for this arises because achieving global synchronization of all the processors at various stages of algorithm B could degrade the time performance. If there is a global controller, the synchronization time may not be very significant, but if the processors have to synchronize using only local communications, it would add at least $O(\log p)$ time per synchronization to the time complexity.

In the model considered in this section, each processor runs its program essentially independent of other processors in the hypercube. The processors interact with each other when necessary by sending messages. To support this communication we assume two operations $\text{Send}(id, data)$ and $\text{Receive}(id, data)$. We describe the behavior of these operations below.

Consider $P_i$, $0 \leq i \leq p - 1$. If $P_i$ has a local variable named $msg$ that it wishes to send to $P_j$, it executes a statement $\text{Send}(j, msg)$ and then continues with the next statement of its program. If $P_j$ wishes to receive some message from $P_i$ into a local variable $data$ it executes a statement $\text{Receive}(i, data)$. We assume that in this case, $P_j$ will wait at this statement until a message from $P_i$ is received, at which point the message is copied to the variable $data$ in $P_j$. Therefore, the effect of $P_i$ executing $\text{Send}(j, msg)$ and $P_j$ executing $\text{Receive}(i, data)$ is as if the variable $msg$ was copied to variable $data$ by an assignment statement.

Message-based Program

In this section, we present the algorithm executed by each processor in performing algorithm B. The program falls into two different patterns, one for processors with ids of the form $2^r - 1$ and one for other processors.
The processor $P_j$ has significant local variables $X_j$, $Temp_j$ and $S_j$. $X_j$ and $Temp_j$ are the same as variables $X[j]$ and $Temp[j]$ of algorithm B. $S_j$ is actually the pair of variables $P[j]$ and $Q[j]$. Since these coefficients are always used as a pair, it is clearer to describe the algorithm by assuming they are packaged as one single variable.

We also define a dummy variable $dummy_j$ into which a pair of coefficients that is received is copied. We use the following shorthand for the computations of the jump operation. If $P_j$ updates its coefficients $S_j$ using received pair of coefficients $dummy_j$, we denote this operation by

$$S_j := Update(S_j, dummy_j).$$

Similarly, when $P_j$ evaluates its variable $X_j$ using $S_j$ and $Temp_j$, we denote this by

$$X_j := Compute(S_j, Temp_j).$$
/* Algorithm executed by processors of the type \( j = 2^t + 1 - 1, -1 \leq t \leq k-1 */

/* Code corresponding to Up-Sweep-B */

Begin

for \( b = 0 \) to \( t-1 \) do

begin

Receive\((j - 2^b, dummy_j)\); /* Receive coefficients */

\( S_j := Update(S_j, dummy_j) \); /* Update coefficients */

end for

Receive\((j - 2^t, Temp_j)\); /* Receive an x */

\( X_j := Compute(X_j, Temp_j) \); /* Evaluate \( x_j \) */

If \( t \neq k-1 \) then Send\((j + 2^{t+1}, X_j)\);

End /* End of Message-based Code for Up Sweep */

/* Begin iterations of Down Sweep */:

for \( b = t-1 \) down to \( 0 \)

begin

Send\((j - 2^t, Temp_j)\);

Receive\((j - 2^t, Temp_j)\);

end /* End of code for Down Sweep */

/* End of message-based code for processors \( j = 2^t + 1 - 1 */

/* Message-based Program for \( j \in C(t), \ 0 \leq t \leq k-1. */

/* Program for iterations of Up Sweep */

for \( b = 0 \) to \( t-1 \) do

begin

Receive\((j - 2^b, dummy_j)\); /* Receive coefficients */

\( S_j := Update(S_j, dummy_j) \)

end
end
Send(j + 2^t, S_j); /* Send coefficients */

/* Begin iterations of Down Sweep */
Receive(j + 2^t, Temp_j);
X_j := Compute(S_j, Temp_j);
Send(j + 2^t, X_j);
for b = t-1 down to 0
begin
  Send(j - 2^b, Temp_j);
  Receive(j - 2^b, Temp_j);
end
/* End of message-based code for processors j ∈ C(t), 0 ≤ t ≤ k-1. */

Example of Message-based Execution

Let us consider the operation of the algorithm for p = 8. We assume a time t for both the operations of updating coefficients as well as for evaluating an x, and τ time delay for transmitting either type of message.

Let us look at the operation of P_7 which determines the total execution time. P_7 falls into the first category of processors whose program was discussed above. With respect to that program, for this processor t = 2; it will execute two update operations, using coefficients received from P_6 and P_5 respectively. It then exits the first loop and receives the value of x_3 from P_3 into its variable Temp_7. It computes x_7. The last Send statement of the program for Up-Sweep is not executed, since there is no processor with index 15. Hence, program execution of code for the Down Sweep begins. P_7 first sends its variable Temp_7 containing x_3 to P_5 and then waits to receive the computed value of x_5 from P_5. It receives this into Temp_7 and immediately forwards it to P_6 (i.e. the send in the do loop with b = 0). Finally, it receives the
computed value of $x_6$ from $P_6$ and the algorithm terminates.

The time required for the execution is determined by the last processor $P_{15}$. The Up-Sweep for $P_{15}$ requires $(t + \tau)$ time per iteration and the Down Sweep time $(t + 2 \tau)$ per iteration. Thus, the time derived earlier for the explicitly synchronized solution is unchanged.
Figure 6. A 16 node hypercube
Chapter 4

First Order Recurrence on Various Architectures

In this chapter we discuss the solution of first order recurrence systems of equations on various multiprocessor architecture models. The models that we examine are as follows. First we consider a model of a shared-memory multiprocessor where the shared memory and the processors are connected by a single bus. We then consider a bus-connected system where processors communicate by making broadcasts over the bus. We conclude the chapter with a mapping of the solution onto a linear array and indicate how that solution can be generalized to higher dimensional arrays as well.

The solution for the shared memory single bus multiprocessor is an implementation of algorithm A. We analyze the performance on this model. A modified algorithm requiring only one phase (in contrast to the Up Sweep and Down Sweep of algorithms A and B) is proposed for the bus connected system with broadcasting and for the linear array. This scheme performs better than direct implementation of algorithm A (or B) on both these models, by exploiting the communication features of the architectures. On the other hand, this one phase strategy would be inefficient on both the hypercube and the shared memory bus connected system.

The problem that we are solving is the solution of a first order recurrence system of \( n \) equations using \( p \) processors. The strategy is, as before, a three phase process, where we have two phases of sequential processing performed independently and concurrently in each processor and a middle phase where the processors communicate. As before, we will only concentrate on the implementation of the middle phase, since the other phases are architecture independent, once the data is loaded into the individual processors.

Shared Memory Bus Connected System

The model consists of \( p \) processors that communicate with each other by reading and writing from shared memory locations. The system memory is assumed to be distributed
among all the processors. The processors are connected to each other by a single bus that can handle only one memory request at a time. If several processors make a request for the bus at the same time, the contention is resolved by a bus controller. In case of contention, the bus controller allocates the bus to processors either on a fixed priority basis or FCFS or random. (Our solution is not affected by the contention resolution logic.) Processors can access the portion of the system memory local to it, without contending for the shared bus. If a memory access is made by processor $i$ to the portion of memory local to a processor $j$ at the same time as processor $j$ is accessing that portion of memory, the contention must be resolved and one of the accesses given precedence. (For the scheme presented here, this situation will not arise.) We assume that there is hardware for globally synchronizing the processors.

Consider the execution of algorithm A on such a machine. The first (and every subsequent) iteration begins with a global synchronization between the processors. This synchronization is assumed to take time $t_{sync}$. This synchronization is necessary so that the processors can begin the next step of computation without affecting the correctness of the algorithm.

In the first iteration of $Up$–Sweep–$B$, $\frac{D}{2}$ processors want to read the value of data which is located in memory local to a processor with index one less than it. The processors in whose memories the remote data lies are not updating any of the data which is going to be read by other processors. Hence the problem of simultaneously trying to read and write in the same memory location does not arise and there is no need of locking the memory locations.

In general in the execution of this algorithm on any parallel machine, processor $i$ never tries to read any data which lies in the memory of processor $j$ and which also needs to be updated by any other processor, including $j$. This is an explicit property of the algorithm and solves many of the problems associated with simultaneous reads and writes in the execution of parallel algorithms. Specifically other solutions to the first order linear recurrence problem such as the one proposed by Kogge and Stone [1] necessitate reading from a processor which wants to modify the same location.
All $\frac{p}{2}$ processors which want to read, contend for the bus and receive their data in some order. Thus the last processor to be serviced, gets the remote data after $\frac{p}{2} \tau$ time. It performs the updating of the variable in time $t$. Thus after time $\frac{p}{2} \tau + t + t_{\text{sync}}$, all the $\frac{p}{2}$ processors have finished the first iteration.

In the second iteration, the same process is repeated, with the difference that only $\frac{p}{4}$ processors want to access remote data. Thus this step takes time $\frac{p}{4} \tau + t_{\text{comp}} + t_{\text{sync}}$. In this manner after $\log p$ steps the Up Sweep terminates. The total time taken by the Up Sweep is given by:

$$T_{\text{Up-Sweep}} = (p - 1) \tau + \log p \ (t + t_{\text{sync}}).$$

In the Down Sweep the pattern of communication is reversed i.e. in the first step there is only one processor which wants to access remote data, in the second step there are two processors and so on. The mode of operation and the synchronization mechanism remain the same. Thus the total time taken by the Down Sweep phase of the algorithm is given by:

$$T_{\text{Down-Sweep}} = (p - \log p - 1) \tau + (\log p - 1) (t + t_{\text{sync}}).$$

Thus the total execution time of the algorithm $A$ on a bus connected shared memory system is given by:

$$T_A = T_{\text{Up-Sweep}} + T_{\text{Down-Sweep}}$$

$$= (2p - \log p - 2) \tau + (2 \log p - 1) (t + t_{\text{sync}}).$$

Assuming $1 \ll \log p \ll p$, we get for $T_A$:

$$T_A = 2p \tau + 2 \log p \ x \ (t + t_{\text{sync}}).$$

The total time taken for the solution of a system of $n$ first order linear recurrence equations on a bus connected shared memory multiprocessor of $p$ processors is given by

$$T = 2 \left\lceil \frac{n}{p} \right\rceil \tau + 2p \tau + 2 \log p \ (t + t_{\text{sync}}).$$
Broadcast-Bus Connected System and Linear Array

In this section we present a modification of algorithm A which is more efficient on a system with a broadcast bus and a linear array of processors. In fact on a linear array this algorithm (which we shall call Algorithm C) takes optimal time including the constant factors. First we quickly derive the execution time of algorithm A on a linear array so that the performance of algorithm C can be compared with it.

Mapping Algorithm A on a Linear Array

Consider a model of a linear array of \( p \) processors. Here each processor (except the two at the end) is connected to two adjacent processors.

Let us consider the direct mapping of algorithm A, discussed in Chapter 2 on such an array. Let \( \tau \) be the time required to transmit a unit of data between two adjacent processors. Also let \( t \) be the time taken for performing the computations in a jump or compute operation. In the first iteration of the Up Sweep processors are reading from adjacent processors. This takes \( \tau \) units of time. Thus the first iteration of the Up Sweep takes \( \tau + t \) time. In the second iteration processors which perform a computation have to read from processors which are separated by an index of two. This would take \( 2\tau \) units of time, followed by \( t \) time for computation. In this manner the total time taken for the Up Sweep is given by:

\[
T_{Up-Sweep} = \log p \ t + (p-1) \ \tau.
\]

Consider the down sweep of algorithm A executed on the array. Here in the first iteration processors would be reading from processors which are indexed \( p/4 \) apart and then computing their \( x_i \)'s. Thus the first step of the down sweep takes time \( t + \tau \frac{p}{4} \). In the next step the time is \( t + \frac{p}{8} \ \tau \) and so on. Thus the total time taken for the down sweep is given by

\[
T_{DownSweep} = (\log p - 1)t + (\frac{p}{2} - 1) \ \tau
\]

Thus the total time taken to execute the algorithm A on a linear array of processors is given by
\[ T = (2 \log p - 3) \cdot t + \left(\frac{3p}{2} - 3\right) \tau \]

Next we present a modification of the algorithm \( A \) for the linear array which takes time \((\log p \cdot t + p \cdot \tau)\) on a linear array. This improved performance is achieved by performing one sweep instead of two sweeps which are necessary in case of algorithm \( A \).

**Algorithm C**

Consider a system of \( p \) processors where \( p = 2^k \). The proposed algorithm consists of \( k \) iterations. Let the iterations be represented by the loop variable \( i \) which varies from 0 through \( k - 1 \). Before the start of the \( i^{th} \) iteration the processor array is divided into \( 2^{k-i} \) sections. Each section consists of \( 2^i \) consecutively indexed processors. These sections are numbered as \( S_m \) \( 0 \leq m \leq 2^{k-i} - 1 \). The processors in section \( S_m \) are \( P_{m2^i} \) to \( P_{(m+1)2^i-1} \). Thus after each iteration the number of sections is halved and the number of processors in each section is doubled.

The algorithm works as follows. At the beginning of the \( i^{th} \) iteration (for any \( i \) from 0 to \( k - 1 \)), the processors in the first section i.e. \( S_0 \), will have their respective \( x_i \)'s known. The \( x_i \)'s in all processors in the remaining sections, i.e. \( S_m \), \( 1 \leq m \leq 2^{k-i} \) will have direct dependencies on the \( x \) variable of the last processor in the immediately previous section, \( S_{m-1} \). i.e. \( P_{m2^i-1} \)

In the \( i^{th} \) iteration, the last processor in \( S_0 \) transmits its \( x \) value to all the processors in the section \( S_1 \). The processors in \( S_1 \) use this value to evaluate their respective \( x_i \)'s (i.e they perform a compute operation).

The processors in the remaining sections either update their dependencies or do nothing at all. All processors in odd numbered sections will update their dependencies (i.e perform a jump operation). The processors in the even numbered sections do nothing, except that the last processor of the section transmits the pair of coefficients necessary for a jump operation to all the processors in the following section. The processors in the immediately following sec-
tion (for which \(m\) is odd) receive the transmitted values and use this to update their dependencies. After \(k\) iterations, there is only one section and all processors in that section have evaluated their \(x\) variables.

Analysis for Linear Array and Broadcast-Bus Multiprocessor

We determine the time for the execution of algorithm C on a linear array. In the \(i^{th}\) iteration the time taken to transmit a value by the last processor in section \(S_m\) to all the processors in section \(S_{m+1}\) is proportional to the length (number of processors in) of \(S_{m+1}\), and equals \(2^i\tau\). Since all computations in the section can be overlapped, the time required for the \(i^{th}\) iteration is given by \((2^i \times \tau + \tau)\). The algorithm terminates when the number of the sections is reduced to one, i.e., after the iteration for which \(i = k - 1\). Hence the total time taken for the execution is given by:

\[
T = \sum_{i=0}^{k-1} (2^i \tau + \tau)
\]

Hence,

\[
T = \log p \tau + (p-1)\tau
\]

The analysis for a bus-connected system with broadcast is now discussed. In this model, each processor has its local memory, and processors communicate by sending a message over the bus. Several processors can simultaneously receive a message broadcast over the bus.

Algorithm C naturally maps onto such a system. In the \(i^{th}\) iteration there are \(2^{k-i}\) sections, and one processor in each even numbered section wishes to broadcast a message. This message will then be used by all processors in the immediately following odd numbered section. The number of broadcasts made in the \(i^{th}\) iteration is therefore \(2^{k-i-1}\). Summing over all values of \(i\) from 0 to \(k - 1\), and charging \(\tau\) time per broadcast, this communication accounts for \(\tau (2^k -1)\) time. Following each broadcast, processors perform one operation, which takes time \(\tau\) per iteration. Note that no explicit synchronization is needed since blocking receive is assumed. Hence, the total time for this model is:
\[ T = \tau (p - 1) + \log p \quad \tau. \]

d-dimensional mesh

In order to solve the recurrence problem on a d-dimensional mesh we apply the linear array algorithm (algorithm C) along each dimension treating each dimension as a linear array. As a result each of the values in a given dimension are expressed in terms of the last value of the previous dimension. Next algorithm C is applied in the next higher dimension. At the end of this process we start backtracking i.e. the computed value of the variables are passed onto the lower dimension, using which the processors compute their respective \( x_i \)'s.

If \( p \) is the number of processors and \( d \) is the number of dimensions then

\[ T_{d-array} = (\log p + d - 1) \tau + (2d - 1)p^{\frac{1}{d}} \tau \]
Chapter 5

Conclusion

In this thesis we have presented an optimal solution to the problem of first order linear recurrence system. In the first chapter we described some of the previous solutions and discussed their drawbacks. Next, in chapter 2, we presented parallel algorithms for implementation on unbounded as well as bounded models of multiprocessor systems. The algorithm A on the unbounded model, was presented. This algorithm takes $O(\log n)$ time using $n$ processors, which consists of two Sweeps was presented in a more formal manner. An analysis of the number of operations was performed and shown to be $O(n)$. This reflects an improvement over previous solutions that had $O(n \log n)$ operations. The algorithm for the bounded model used algorithm A as the middle stage between two stages of serial processing performed independently on each processor. Thus the algorithm takes $O(n/p + \log p)$ time on a $p$ processor system.

Next we discuss the mapping of the algorithm on various architectures to include communication costs. In chapter 3 we considered mapping on the hypercube. We showed that a slight modification of the algorithm A permits efficient mapping onto the hypercube. The implementation on a network supporting blocking receives was presented by writing the program in terms of the send and receive primitives. This enabled us to do away with the need for global synchronizations.

In chapter 4 we presented a modified algorithm (algorithm C specialized for a linear array of processors and a broadcast bus connected system. We also discussed the solution on a shared memory bus-connected system.

The next step in this research would be to investigate the performance of this algorithm on actual multiprocessor machines by experimentation or by simulation. It would be interesting to attack the more general problem of $m^{th}$ order recurrence.
References:


