LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

# A Maximally Pipelined Tridiagonal Linear Equation Solver

Computation Structures Group Memo 254
June 1985

**Guang R. Gao**

The research described in this paper and its two companion papers [12, 13] was supported by the Department of Energy and the National Science Foundation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

# A Maximally Pipelined Tridiagonal Linear Equation Solver

[1]

GUANG R. GAO

*Laboratory for Computer Science*
*Massachusetts Institute of Technology*
*Cambridge, MA 02139*
*June. 1985*

**Abstract**

A new method of solving tridiagonal systems of linear equations is proposed which introduces parallelism in a way that may be effectively exploited by a suitable parallel computer architecture. The scheme is based on the program transformation techniques which can produce machine code to be executed in a maximally pipelined fashion. Compared to the existing parallel solution techniques such as the cyclic reduction algorithms, the new method has the following advantages: (1) it eliminates the substantial data rearrangement overhead incurred by many existing parallel algorithms; (2) it sustains a relatively constant parallelism during various phases of program execution; (3) the code generation is independent of the length of the vectors to be computed, hence is more flexible; (4) in general the size of machine code is much smaller and hence is more efficient in terms of memory usage. The new method is also numerically stable. Using the new method, we outline the code structure of a maximally pipelined tridiagonal linear equation solver for a static data flow supercomputer.

---

1. The research described in this paper and its two companion papers [12, 13] was supported by the Department of Energy and the National Science Foundation.

CONTENTS

## FIGURES

## 1. Introduction

Tridiagonal systems of linear equations form a very important class of linear algebraic equations. For example, the heart of finite difference solutions of PDEs (partial differential equations) consists of tridiagonal systems of equations. Consequently, efficient solutions for such equations become crucial for many numerical algorithms.

It is well known that a tridiagonal system of linear equations can be solved on a conventional computer using the classical *Gaussian elimination algorithm*, which, though has been proved most effective on serial computers, is sequential in nature and hence unsuitable for parallel computers without drastic alteration.

In the past decade, new techniques have appeared for solving tridiagonal systems of equations with parallel computers [16, 20]. The best known parallel algorithm is based on the cyclic reduction technique, first proposed by Golub and Hockney and applied by Buzbee et al, for solving tridiagonal system of equations efficiently [14,3]. One approach is using such parallel technique to solve the recurrences established by the LU-decomposition method of Gaussian elimination algorithm. One such algorithm, known as recursive doubling suggested by Stone [23] and originally designed for Illiac IV, was later modified for other vector computers [24]. Another approach has resulted from considering the needs of parallel processing in the first place and trying to design fundamentally new algorithms which are inherently more parallel. The *odd-even cyclic reduction* algorithm is base on such a principle. As will be discussed in section 2, a major difficulty with the algorithms based on cyclic reduction technique is the overhead of data rearrangement between computation steps which may lead to considerable performance degradation in its implementation on vector computers, such as Cyber 205 and Cray. Another problem is in the considerable variations of degree of parallelism between computation steps. This may raise the speculation that, for sufficient small vector size n, the sequential algorithm would run faster than cyclic reduction.

In this paper, a new method for solving tridiagonal systems of equations is proposed

which introduces parallelism in a way that may be effectively exploited by a suitable parallel computer architecture. The algorithm is based on the maximally pipelined solution of linear recurrences presented in a companion paper [12]. It performs a program transformation of the recurrences generated in the Gaussian elimination method to produce machine code which can be executed in a maximally pipelined fashion. Compared to the existing parallel solution techniques such as cyclic reduction algorithms, the new method has the following advantages: (1) it eliminates the substantial data rearrangement overhead incurred by many existing parallel algorithms; (2) it sustains a relatively constant parallelism during various phases of program execution; (3) the code generation is independent of the length of the vectors to be computed, hence is more flexible; (4) in general the machine code size is much smaller and hence is more efficient in terms of memory usage. The new method is also numerically stable. Based on this scheme, the code structure of a maximally pipelined tridiagonal equation solver is outlined for a static data flow supercomputer.

In Section 2, following a brief statement of the problem, we present the major recurrences established by LU-decomposition technique. These recurrences are important to later discussions. We survey related work on parallel tridiagonal solution methods, with particular emphasis on the cyclic reduction technique. This discussion includes both the recursive doubling algorithm for solving the linear recurrences generated in Gaussian elimination and the odd-even cyclic reduction algorithm. The disadvantages of these algorithms are discussed. In section 3, we present a basis for the pipelined tridiagonal system solver, i.e., the pipelined solution of linear recurrences, which is based on the principles developed earlier in [4, 9], followed by an extension described in one companion paper [12]. Section 4 develops the machine code structure of a fully pipelined tridiagonal solver and outlines its advantages over other parallel solutions. In Section 5 we discuss a future extension of the pipelined solution scheme. Section 6 briefly addresses the numerical stability aspects of the new method based on the result of a second companion

paper [13]. The conclusions and future research suggestions are in Section 7.

## 2. Background and Related Work

In this section we state briefly the problem of tridiagonal system of linear equations and review the directed methods for solving them — such as the Gaussian elimination algorithm, in particular the linear recurrences established by the LU-decomposition technique which are the starting point of our new approach. We also survey the related work of parallel tridiagonal solution methods, such as the well-known cyclic reduction technique. Our discussion includes both the recursive doubling algorithm for solving the recurrences directly established by the Gaussian elimination method and the odd-even cyclic reduction algorithm. The disadvantages of these algorithms are discussed.

### 2.1 Statement of The Problem

We consider the solution to the following tridiagonal set of linear equations:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & & \\ 0 & a_3 & b_3 & \ddots & \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \cdot \\ \cdot \\ k_n \end{pmatrix} \qquad (2.1)$$

or expressed in matrix-vector notation

$$\mathbf{Ax} = \mathbf{k} \tag{2.1}$$

In this paper, our major concern will be the case where the coefficient matrix **A** is positive definite or at least pivoting is not required.

## 2.2 LU-Decompusition

There are a number of serial methods for solving the tridiagonal system as expressed in (2.1). The maximally pipelined solution method to be developed in this paper is based on the well-known *LU-decomposition* technique [8]. The Stone's recursive doubling algorithm to be discussed later is also based upon such technique. In this method, we find two matrices, L and U, such that

(1) LU = A;

(2) L is a lower bidiagonal matrix;

(3) U is an upper bidiagonal matrix with 1 on its principal diagonal.

When A is non-simpler, its LU decomposition is unique. In fact, it is shown that

$$L = \begin{pmatrix} e_1^{-1} & 0 & \cdots & 0 \\ a_2 & e_2^{-1} & & \vdots \\ 0 & a_3 & e_3^{-1} & \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_n \quad e_n^{-1} \end{pmatrix}$$

and

$$U = \begin{pmatrix} 1 & u_1 & 0 & \cdots & 0 \\ 0 & 1 & u_2 & \ddots & \vdots \\ \vdots & \ddots & 1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & u_{n-1} \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}$$

where

$$\begin{aligned} u_1 &= c_1/b_1 \\ u_i &= c_i/(b_i - a_i u_{i-1}) \qquad i = 2,3...n-1 \qquad (2.3) \\ e_i &= u_i/c_i \end{aligned}$$

After computing L and U, it is relatively straightforward to solve the system of equations by a two-step process. First, letting $Y = Ux$, we have

$$Ly = K \qquad (2.5)$$

$$Ux = y \qquad (2.6)$$

and together, we have $Ax = LUx = Ly = k$.

The equation $Ly = k$ can easily be solved for y as follow.

$$\begin{aligned} y_1 &= k_1/b_1 \\ y_i &= (k_i - a_i y_{i-1})/(b_i - a_i u_{i-1}) \qquad i = 2,3...n \qquad (2.7) \end{aligned}$$

Note that in the solution process, as indicated by (2.7), there is no need to compute $e_i$ explicitly unless the matrix L is needed in other places. Next, we solve Ux = y for x by noting that

$$x_n = y_n$$
$$x_i = y_i - u_i x_{i+1} \qquad i = n-1,n-2...1 \qquad (2.8)$$

The two steps of (2.7) and (2.8) are often called *forward-elimination* and *backward-substitution.* The recurrences (2.3), (2.7) and (2.8) constitute a complete solution for Ax = k. and a sequential algorithm to perform such a solution is the so-called *Gaussian elimination algorithm.* We can observe that these recurrences cannot be evaluated directly using vector operations or on array of processors. Hence new solution techniques have been proposed, such as the cyclic reduction algorithms which we will presented next.

## 2.3 Cyclic Reduction Technique

In this paper, we make no attempt to survey all parallel algorithms for tridiagonal linear equation solvers, but review only two well-known methods which will be compared with the pipelined solution algorithm. These methods are based on the cyclic reduction technique.

### 2.3.1 Recursive Doubling Algorithm

The recursive doubling algorithm proposed by Stone [23] began with the observation that the formula required by LU factorization, such as (2.7) and (2.8), are first order linear recurrences. The equation (2.3) appears not to be a linear recurrence. However, if we introduce a new variable $q_i$ such that $u_i = -q_i/q_{i+1}$, then (2.3) can be transformed into the following second-order linear recurrence:

$$a_i q_{i-1} + b_i q_i + c_i q_{i+1} = 0 \qquad i = 2,3,...,n-1 \qquad (2.9)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$

We note that (2.9) can be expressed in the matrix notation as follow:

$$\begin{pmatrix} q_i \\ q_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -a_i/c_i & -b_i/c_i \end{pmatrix} \begin{pmatrix} q_{i-1} \\ q_i \end{pmatrix} \qquad (2.10)$$

Stone pointed out that (2.10) is similar to a first-order linear recurrence except that the sequence is now a set of vectors instead of scalar values. The recursive doubling algorithm [23,24] treats (2.7), (2.8) and (2.10) as general first-order linear recurrence and using standard cyclic reduction method for handling linear recurrences to solve them.

It is helpful to review the cyclic reduction technique for solving linear recurrences before we outline its disadvantages. For instance, we consider the evaluation of the sequence of $x_i$ from the following first-order linear recurrence relation.

$$x_i = a_i x_{i-1} + b_i \qquad \text{for } i = 2...n \qquad (2.11)$$

where $x_1, a_1, ..., a_n$ and $b_1, ..., b_n$ are known values. The basic idea of standard cyclic reduction technique is to back up the recurrence in (2.11) such that a new recurrence can be obtained which relates every other term, every fourth term, every eighth term, etc.

For example, from (2.11) we have

$$\begin{aligned} x_i &= a_i a_{i-1} x_{i-2} + a_i b_{i-1} + b_i \\ &= a_i^{(1)} x_{i-2} + b_i^{(1)} \end{aligned} \qquad (2.12)$$

where $a_i^{(1)} = a_i a_{i-1}$, $b_i^{(1)} = a_i b_{i-1} + b_i$. The superscript (1) denotes the fact that this is a first

level backup. Such a backup process can be repeated (in a cyclic fashion) and we obtain a set of equations as follow.

$$x_i = a_i^{(l)} x_{i-2^l}^{l-1} + b_i^{(l)} \tag{2.13}$$

where

$$a_i^{(l)} = a_i^{(l-1)} a_{i-2^l}^{(l-1)} \tag{2.14}$$

$$b_i^{(l)} = a_i^{(l-1)} b_{i-2^l}^{(l-1)} + b_i^{(l-1)} \tag{2.15}$$

with $l = 0,1...\log_2 n$, $i = 2,3...n$. An important observation is that if any of $a_i$, $b_i$ or $x_i$ is outside the defined range, its value can be taken as zero. Therefore, when $l = \log_2 n$, all $x_i$ are solved by

$$x_n = b_n^{(\log_2 n)}$$

At level l, $m_l$—the number of operations for generating corresponding coefficients is roughly given by $m_l = 3(n-2^{l-1})$. This suggests the following observation:

(1) high parallelism exists at certain phases (steps) of the algorithm, i.e., at a fix level l, (2.14) and (2.15) can be evaluated for all i in parallel;

(2) the parallelism grows roughly linearly with the size of the vectors — i.e. n in this case;

(3) the useful parallelism decreases as the computation progressing through different phases.

Several difficulties with the above standard cyclic reduction solution for linear recurrences exist. First, the amount of parallelism varies between phases of computation. This will increase the difficulty of fully utilize the parallelism of the machine. For a pipelined vector computer, this requires that the machine should efficiently support vector

operations of variable vector lengths, For processor arrays with $n \ll N$ (N is the number of processors), it does not produce a balanced work load on all processors during the program execution. When $n > N$, the allocation of the vectors becomes a challenging task to assure both high speed of computation and good utilization of resources. Moreover, the number of operations is O(nlogn), a significant increase from O(n) for sequential Gaussian elimination methods. As pointed out by Lambiotte & Voigt [19], for vector computers the total number of operations is also an important factor. Therefore, even though vector operations may be applied, at some point the nlogn operations become the dominate factor and the vector algorithm will be slower than the scalar algorithm.

### 2.3.2 Odd-even Reduction Algorithm

The *odd-even cyclic reduction algorithm* is perhaps the most successful cyclic reduction algorithm applied to solve tridiagonal systems [3, 15, 7]. It starts directly from the system of equations defined by (2.1), i.e.,

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = k_i \qquad i = 1,2...n^1$$

The algorithm first eliminates the odd numbered variables in the even numbered equations by performing elementary row operations. In each level, we cut down the total number of equations by 1/2, hence, in $\log_2 n$ levels, the middle element $x_{n/2}$ can be computed directly from the coefficients. The remaining unknowns can be found by a refilling procedure. This algorithm also involves the recursive calculation of coefficients for equations at each level. One important advantage of the odd-even reduction over the recursive doubling algorithm is that it reduces the number of operations considerably at each level, and the total number of operations is on the order of O(n).

---

1. In the remaining discussion of this section, we assume n is a power of 2, but this is not an essential assumption.

One major difficulty with odd-even reduction is the data rearrangement of variable and coefficient vectors between phases of computation. For example, on the Cyber 205 one cannot apply vector operations directly to every other elements of the vector. Thus extra operations must be employed to reformat those elements into a new vector [19]. On the Cray it is possible to access elements of a vector at a fixed increment, but this may result in a performance degradation [2, 17]. Because of the overhead of data rearrangement, the cyclic reduction algorithm may run slower than a serial algorithm for sufficiently small n [20].

Another problem is the degree of variation of parallelism between different phases of computation. Because the parallelism decreases very rapidly, this problem becomes more serious than that for the recursive doubling algorithm. A parallel version of odd-even reduction algorithm has been proposed to keep a high parallelism throughout the computation. However, it increases the number of operations significantly to O(nlogn) [16].

## 3. Pipelined Solution Scheme for Linear Recurrences

The challenge of mapping numerical algorithms onto parallel computers is devising the method and arranging the computation so that the architecture features of a particular machine can be fully utilized.

The model of parallel computers for the pipelined algorithm developed in this paper is based upon computers built on data flow principles. In particular, we use the *static data flow machine* as a target machine to develop the machine code structure. The readers who are unfamiliar with the basic concept of data flow architecture are referred to [5,6].

## 3.1 Overview of the Pipelined Solution Scheme

In cyclic reduction scheme, the goal is to increase the speed through fully exploiting the parallelism in the original problem. High concurrency is obtained by replicating the operations as much as necessary to compute all elements in the result vector in parallel. Unfortunately, this technique incurs significant overhead of data rearrangements. Moreover, it requires varying amount of parallelism during different phases of computation.

In contrast, we propose a new method of solving tridiagonal systems which can explore and organize the parallelism in a way that best matches a suitable computer architecture. The new algorithm, named *pipelined tridiagonal solver*, is developed from a maximally pipelined mapping scheme for solving linear recurrences. In this section we briefly outline the key idea of the basic scheme.

The maximally pipelined code mapping scheme for linear recurrence is based on the principle established in our previous work [4,9]. A more thorough treatment of mapping linear recurrences along this line can be found in [12]. A key step in a pipelined mapping of (2.11) is backing up the recurrence in a manner similar to that in cyclic reduction as described in section 2.3.1. However, a fundamental difference exists in the philosophy — the backup procedure here is taken as a way to allow multiple elements to be computed by the same piece of machine code in a maximally pipelined fashion. Therefore, instead of fully expanding the recurrence to the end, the backup is only performed to an extent that a constant degree of parallelism can be maximally and gracefully explored throughout the computation. The maximally pipelined throughput is the major concern of this scheme.

The form of parallelism which is of most interest to us is the potential of *maximally pipelined* execution of machine level data flow programs [4]. The power of pipelined computation in the data flow computer lies in the possibility of machine level programs that form one large pipeline in which many instructions in various of stages are in concurrent execution. It is beyond the scope of this paper to discuss such scheme in detail,

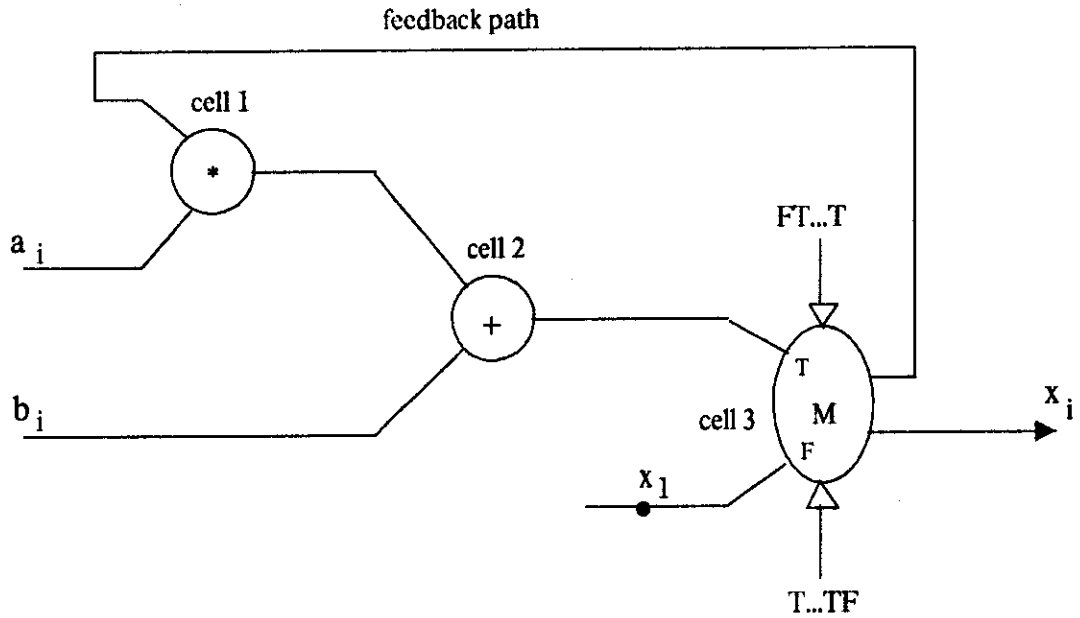
a background review can be found in [4,9,10]. In the following, we present an example to illustrate how such scheme is used in the pipelined mapping of linear recurrences.

### 3.2 Maximally Pipelined Mapping of Linear Recurrences

An attractive way to implement recurrence on data flow computers is to introduce feedback paths in the data flow graph. This, however, presents particular problems when maximum pipelining of the program is desired. A *direct translation* of the first order recurrence is shown in Fig. 1. The value $x_i$ depends on the value of $x_{i-1}$, therefore, a *feedback path*, such as the one marked in the graph, is generated. The key is to understand the role of the merge operator (denoted by M in Fig. 1): (1) under the *merge control* input values (<FT...T>), the initial output value of the loop is taken from the second input of the merge, i.e., $x_1$. (2) the upper output of M is routed under the *feedback control* values, i.e. <T...TF>, therefore all but the last two elements of the array will be fed back; and (3) the

---

**Fig. 1. The Pipelined Mapping of a First-Order Linear Recurrence**



lower output of the merge is forwarded as the output of the loop unconditionally. Due to the existence of cycles, the instruction graph produced by such a scheme, in general, cannot be fully pipelined. More specifically, the feedback link between the output of cell 3 and the input of cell 1 prevents the whole graph from being fully pipelined.

The problem of the above example and its solution have been studied by Dennis and Gao in [4, 9]. As indicated by the author in a companion paper [12], the problem is essentially a mismatch between the *dependence delay*— the dependence inherent in the recurrence (i.e., $x_i$ depends on $x_{i-1}$, therefore, at least a two-stage feedback delay is required) and the *computational delay*—the actual length of the loop in the data flow graph generated by the direct translation scheme (3 stages in this example). In [12], the author described a solution for such a problem on a static data flow computer, based on the concept of *companion functions* [4,18]. It is essentially a way to remove the dependence of $x_i$ on $x_{i-1}$, thus, easing the feedback constraints in order to match the computational delay of the data flow graph. For the above example, we have:

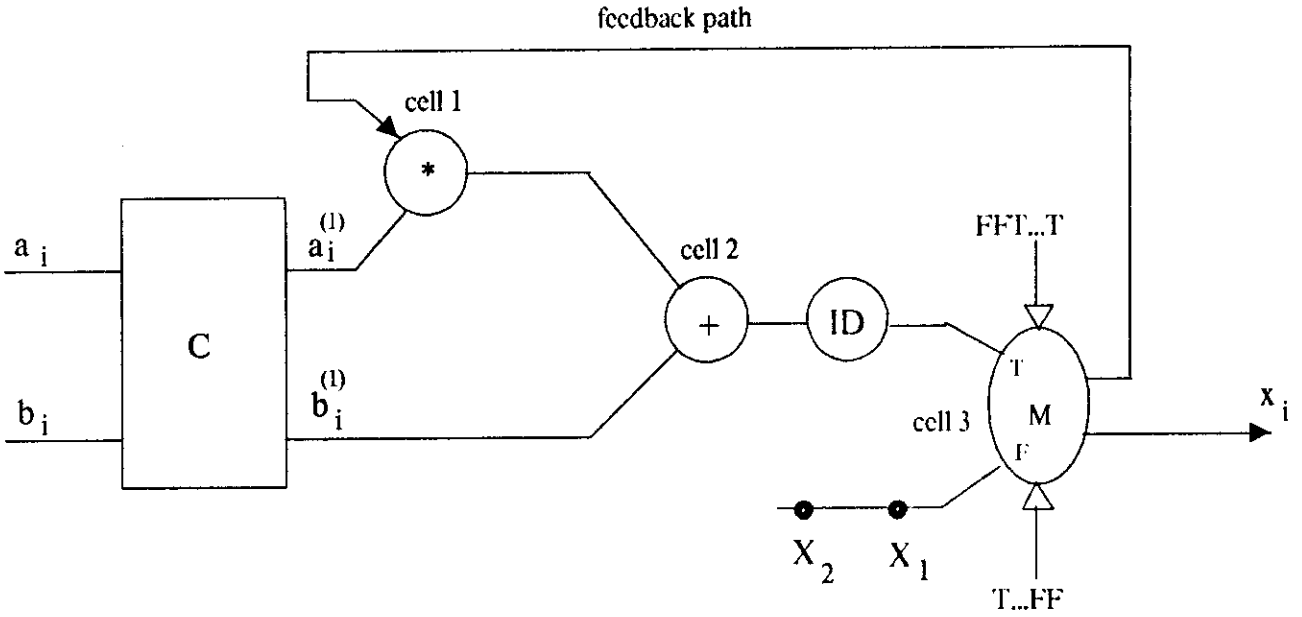
$$\begin{aligned} x_1 &= b_1 \\ x_2 &= a_2 b_1 + b_2 \\ x_i &= a_i a_{i-1} x_{i-2} + a_i b_{i-1} + b_i \qquad \text{where } i \geq 3 \end{aligned} \tag{3.1}$$

We can note that (3.1) is the same as (2.12). This transformation is interesting to us because $x_i$ now depends on $x_{i-2}$ instead of $x_{i-1}$. Therefore, we can map our example, now expressed as in (3.1), into a data flow graph as shown in Fig. 2. Note that we have introduced two additional pipelines $a_i'$ and $b_i'$ as denoted by the dotted lined box C, where

$$\begin{aligned} a_i^{(1)} &= a_i a_{i-1} \\ b_i^{(1)} &= a_i b_{i-1} + b_i \qquad \text{where } i \geq 3. \end{aligned}$$

This added pipeline is named the *companion pipeline* in [4], and its structure is shown in Fig. 3. To understand how the scheme works we first examine the loop. The role of the

**Fig. 2. Maximually Pipelined Mapping of First-Order Linear Recurrences**



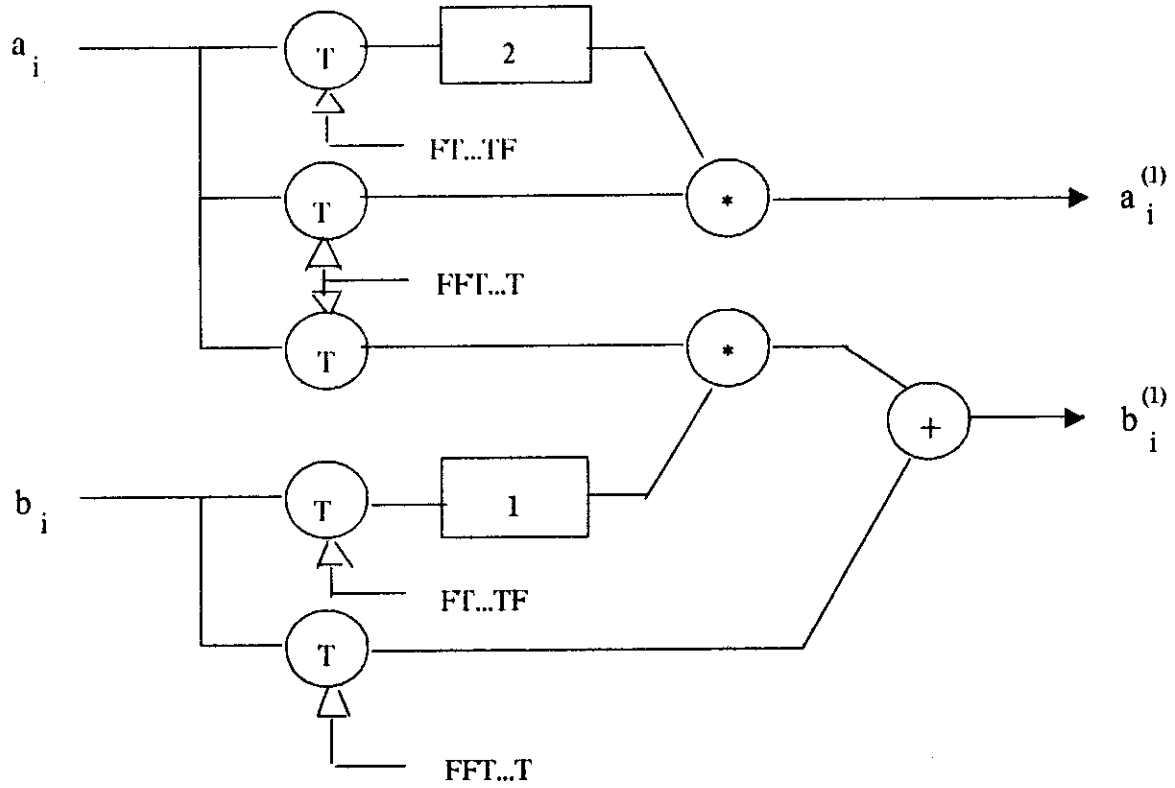

merge operator is as before except that two initial values are presented to the second input of the merge, i.e. $x_1 = b_1$, $x_2 = a_2b_1 + b_2$. The ID cell plays the role of a FIFO of size 1. It is inserted to achieve the computational delay needed in the feedback path to match the dependence delay. The rest of the graph is self-explanatory and the reader should be convinced that the graph is maximally pipelined.

The transformation shown in the above example is equivalent to dividing the first-order linear recurrence into two equivalent classes of computation. In fact, since $x_i$ now depends on $x_{i-2}$, we may split the sequence of results into two subsequences:

$$X' = \langle x_1, x_3, x_5 \ldots x_{2i-1} \ldots \rangle$$
$$X'' = \langle x_2, x_4, x_6 \ldots x_{2i} \ldots \rangle.$$

Either sequence can be computed independently by sharing the same loop, and the companion pipeline provides the appropriate input coefficients.

**Fig. 3. The Companion Pipeline fo Example (3.1)**




The advantage of this scheme is obvious. First, it does not require data rearrangement during the computation. In fact, it even eliminates the requirement that the input vectors be completely filled before the computation starts. If the input coefficient vectors themselves are generated by some preceding code block in a pipelined fashion, the producer-consumer type of interface technique (as described in [9]) can be applied to save considerable storage space for the intermediate values. Moreover, we observe that the degree of parallelism remains constant (5 floating point arithmetic operations and several other operations) in the computation.[1] The high throughput is achieved by the maximally

1. Here the variation of the parallelism during the start and finish time of the computation is not considered

pipelined execution of each actor in the data flow program, hence, the storage usage by the machine code is very efficient. Finally, there are no essential limitation on the length of the vectors which can be computed.

The principle of maximally pipelined solution described in this section can also be applied to a general linear recurrence [12].

## 4. A Maximally Pipelined Tridiagonal Solver

In this section, we apply the principle of pipelined solution of linear recurrences described in the last section to develop the machine program structure for a maximally pipelined tridiagonal system solver.

Starting from LU-decomposition, we can observe that the major equations, such as (2.7) and (2.8), are first-order linear recurrences. Therefore, the pipelined method as described in the last section can be applied directly. Now consider (2.9) which can be transformed into the following second-order linear recurrence.

$$q_i = \alpha_i q_{i-1} + \beta_i q_{i-2} \qquad i = 3,4...n \qquad (4.1)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$ and

$$\alpha_i = -b_{i-1}/c_{i-1}$$
$$\beta_i = -a_{i-1}/c_{i-1}$$

Performing one level backup we obtain

$$q_i = \alpha_i^{(1)} q_{i-2} + \beta_i^{(1)} q_{i-3} \qquad i=4,5...n \qquad (4.2)$$

where $q_1 = 1$, $q_2 = -b_1/c_1$, $q_3 = b_2b_1/c_1c_2 - a_2/c_2$, and

$$\alpha_i^{(1)} = \alpha_i \alpha_{i-1} + \beta_i$$

$$\beta_i^{(1)} = \alpha_i \beta_{i-1} \qquad i = 4,5...n \qquad (4.2.1)$$

Fig. 4 shows a maximally pipelined data flow machine level program for mapping (4.2.1). The loop in the middle of Fig. 4 can easily be understood by noting its similarity with the loops in Fig. 2. The code in the dotted lined box is the companion pipeline generating values for $\alpha_i^{(1)}$ and $\beta_i^{(1)}$. The node labeled N performs a negation of its input. The boolean value sequences C0 - C5 can be found in Fig. 6. The boxes denote the FIFO buffers which are introduced for balancing the graph [6,9] to achieve maximum pipelining, and the number written inside the box is the number of stages in that buffer. It is easy to check that Fig. 4 correctly computes (2.9) and it is maximally pipelined.

We rewrite the first-order linear recurrence in (2.7) as

---

**Fig. 4. Pipelined Tridiagonal Linear Equation Solver -- Part 1**

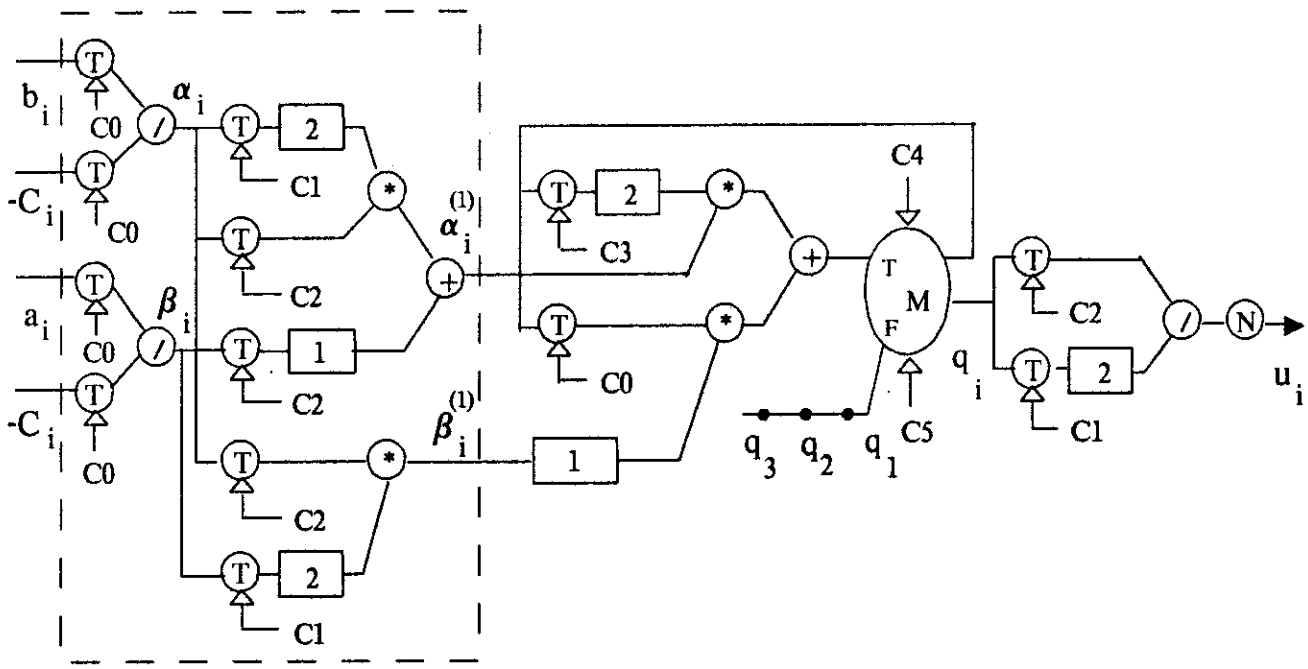
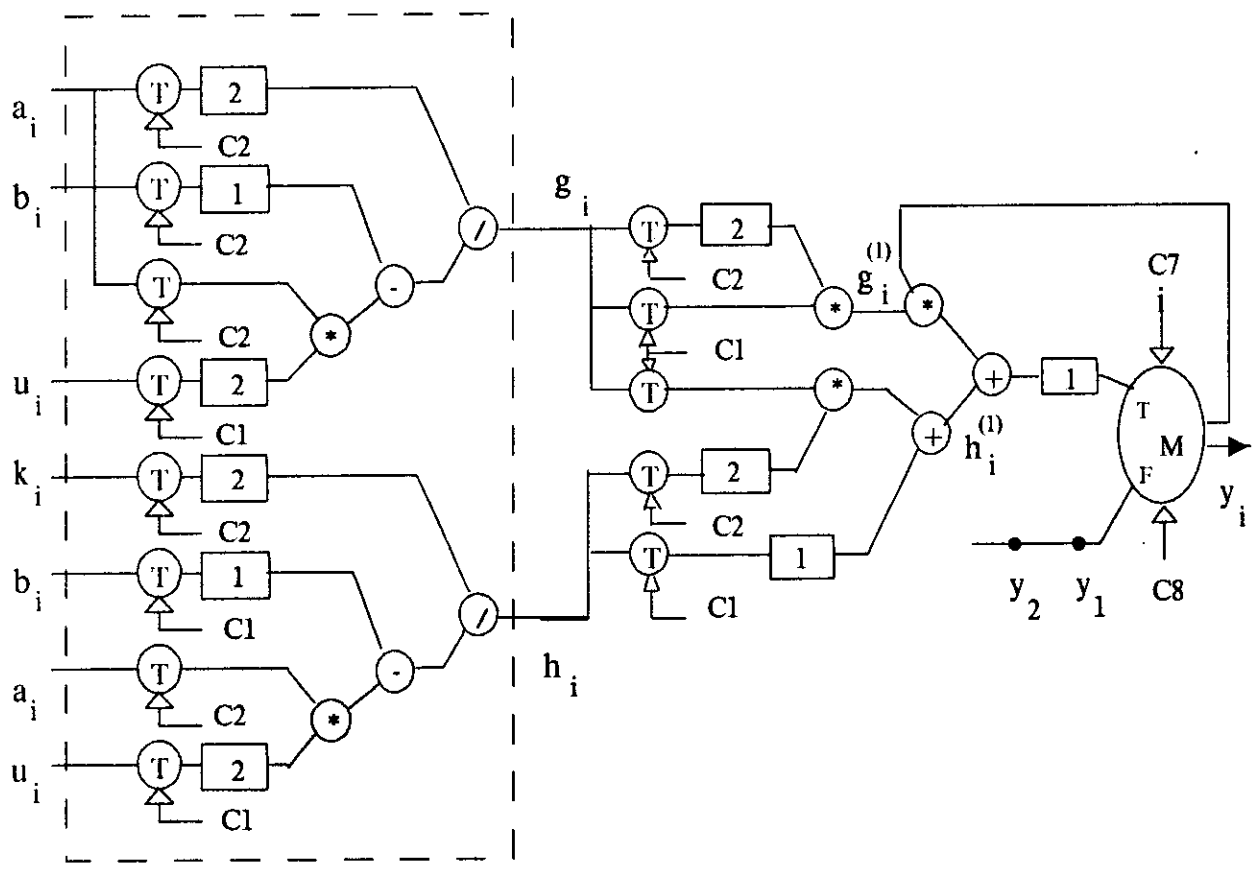


**Fig. 5. Pipelined Tridiagonal Linear Equation Solver -- Part 2**




$$y_i = g_i y_{i-1} + h_i \qquad i=2,3...n \qquad (4.3)$$

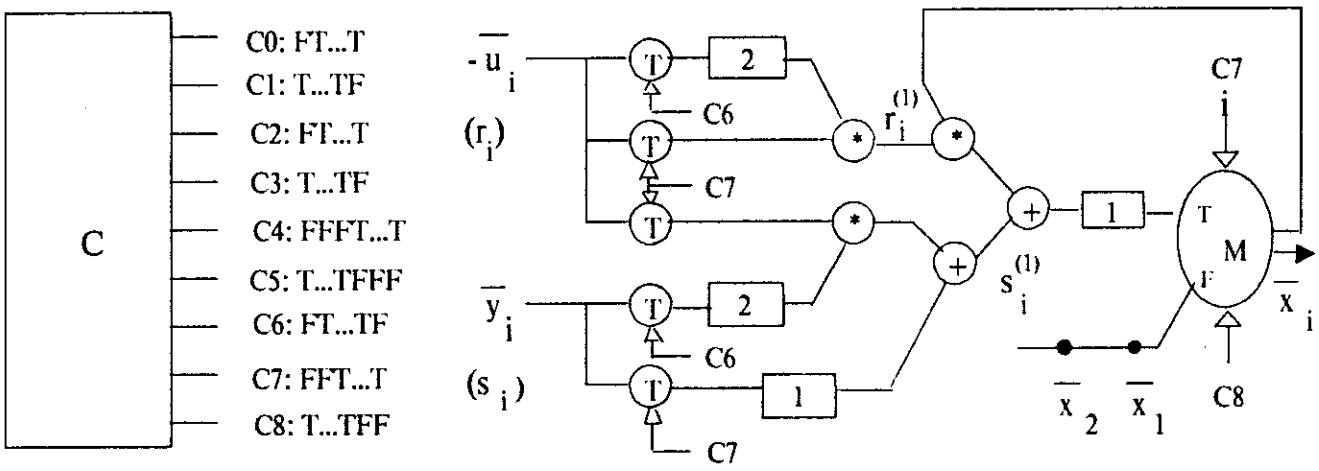
where $y_1 = k_1/b_1$, $g_i = -a_i/(b_i - a_i u_{i-1})$, $h_i = k_i/(b_i - a_i u_{i-1})$. Performing one level backup we obtain

$$y_i = g_i^{(1)} y_{i-2} + h_i^{(1)} \qquad i = 3,4...n \qquad (4.4)$$

where $y_1 = k_1/b_1$, $y_2 = (-a_2 k_1 + b_1 k_2)/(b_2 - a_2 u_1) b_1$, and

$$g_i^{(1)} = g_i g_{i-1}$$
$$h_i^{(1)} = g_i h_{i-1} + h_i$$

**Fig. 6. Pipelined Tridiagonal Linear Equation Solver -- Part 3**




---

Fig. 5 shows a maximally pipelined mapping of (4.4). The dotted lined box is the companion pipeline and the boolean sequences C1,C2,C7,C8 can be found in Fig. 6.

Finally, (2.8) can be conveniently treated as a first-order linear recurrence by introducing new variables $\bar{x}_i$, $\bar{y}_i$, $\bar{u}_i$ such that $\bar{x}_i = x_{n-i+1}$, $\bar{y}_i = y_{n-i+1}$, $\bar{u}_i = u_{n-i+1}$. Hence, (2.8) can be rewritten as

$$\bar{x}_i = r_i \bar{x}_{i-1} + s_i \qquad i = 2,3...n \qquad (4.5)$$

where $\bar{x}_i = \bar{y}_i$, $r_i = -\bar{u}_i$ and $s_i = \bar{y}_i$. We can note that (4.5) is a standard first-order linear recurrence, hence we can solve it by one level backup:

$$\bar{x}_i = r_i^{(1)} \bar{x}_{i-2} + s_i^{(1)} \qquad i = 3,4...n \qquad (4.6)$$

where $\bar{x}_1 = \bar{y}_1$, $\bar{x}_2 = \bar{u}_2 \bar{y}_1 + \bar{y}_2$ and

$$r_i^{(1)} = r_i r_{i-1}$$
$$s_i^{(1)} = r_i s_{i-1} + s_i$$

Fig. 6 shows a maximally pipelined mapping of (4.6).

Now we have constructed a complete machine code structure for a maximally pipelined tridiagonal solver as shown by Fig. 4 - Fig. 6. Fig. 4 and Fig. 5 can be combined into one maximally pipelined data flow graph by observing that the sequence of values of $u_i$ produced by Fig. 4 can be directly fed into Fig. 5. The interface between the outputs of Fig. 4 and Fig. 5 and the inputs of Fig. 6 cannot be connected directly. The main reason is that the order in which the elements $y_i$ and $u_i$ are generated by Fig. 4 and Fig. 5 is opposite to the order in which they are used for the maximum pipelining of Fig. 6. Hence, we should first store the values of $u_i$, $y_i$ into two arrays. Then, Fig. 6 will access the arrays in a reverse order. The code in Fig. 4 and Fig. 5 will sustain a constant parallelism such that there are 20 floating point operations and a number of other operations are concurrently in pipelined operation. When only the code of Fig. 6 is in execution , the parallelism will be reduced to a constant of 5. Although there is such a change of degree of parallelism between the forward elimination and backward substitution phases of the computation, they are entirely stable during each phase, hence are easily to be handled by the processors. Furthermore, In Fig. 4 - Fig. 6, the pattern of runtime data routing is regular, thus eliminating the data rearrangement problem for cyclic reduction. Moreover, it essentially can work for tridiagonal systems regardless of their size, hence, has more flexibility and generality than the cyclic reduction scheme.

The reader may wonder if the 5 to 20 folds of parallelism available in the pipelined algorithm may not meet the appetite of a supercomputer. We argue that the major concern should be how the parallelism in the algorithm can be most effectively used by a suitable architecture. First, the new scheme maintains a relatively constant amount of parallelism and relatively simple data routing pattern. Thus, the resource management and allocation problems are more easy to handle, thereby providing better opportunity of parallel

processing when the machine has extra power. Second, one is often faced with solving a set of m independent tridiagonal systems (say, m=64), as frequently occurs in the solution of PDEs [16]. In this case, the new scheme can be best used by generating m independent pipelines for each system to obtain 20xm folds of parallelism (near 1000 if m=64 !). Finally, the new scheme is flexible enough to be extended to obtain more parallelism when such a requirement does occur. We will briefly address such extensions in the next section.

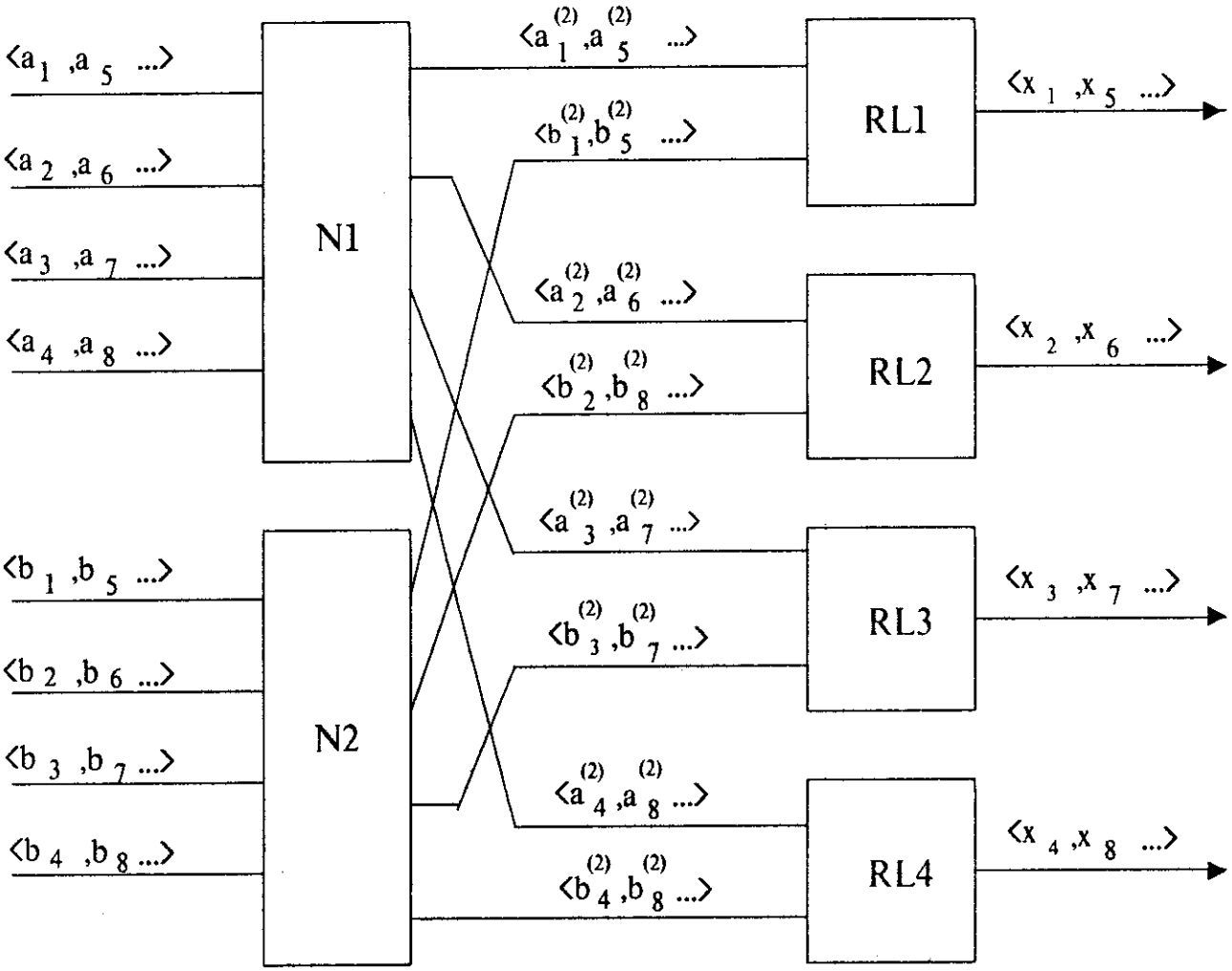
## 5. The Multi-level Pipelined solution Scheme—An extenssion

In the last section, we have presented a tridiagonal linear system solver based on maximally pipelined solution technique for linear recurrences. A fixed amount of parallelism is obtained in the implementation illustrated in Fig. 4 - Fig. 6 where one level backup is performed. A natural question now arises: how far can we push this technique to get more parallelism, and what is the trade-off?

Take the first-order linear recurrences, such as (2.11) or (3.1), as an example. As pointed in [12], the pipelined solution does not change basic structure of the loop. It increases the parallelism by maximally utilizing the few instructions in the loop. Backing up one level, as shown in Fig. 2, is equivalent to multiplexing the computation of odd and even elements with the same code. In [12], it is also shown that this code is saturated with activities so that further backups with one copy of the code do not help in improving performance. In addition, it may cause extra overheard in terms of both delay and memory space. What should we do if the machine still has extra processing power? Naturally enough, we should construct multiple pipelines in the code.

Fig. 7 shows a high-level view of an example of the machine code structure which has 4 times as much parallelism as that in Fig. 2. The key is to introduce more concurrency by performing one more level of backup and utilizing multiple copies of code. For example, we can first transform (2.11) into the following form:

Fig. 7. A Multiple Pipeline Solution




---

$$x_i = a_i^{(2)} x_{i-4} + b_i^{(2)} \qquad (5.1)$$

where $x_1 = b_1$, $x_2 = a_2 b_1 + b_2$, $x_3 = a_3 (a_2 b_1 + b_2) + b_3$, $x_4 = (a_3(a_2 b_1 + b_2) + b_3) + b_4$

and

$$a_i^{(2)} = a_i a_{i-1} a_{i-2} a_{i-3} \tag{5.2.1}$$

$$b_i^{(2)} = a_i a_{i-1} a_{i-2} b_{i-3} + a_i a_{i-1} b_{i-2} + a_i b_{i-1} + b_i \tag{5.2.2}$$

The code modules RL1-RL4 are used to compute the subsequences $\langle x_1, x_5, x_9 ... \rangle$, $\langle x_2, x_6, x_{10} ... \rangle$, $\langle x_3, x_7, x_{11} ... \rangle$ and $\langle x_4, x_8, x_{12} ... \rangle$ respectively. Each box is a copy of the code in Fig. 2.

The boxes N1 and N2 are code which generate the coefficient sequences of (5.2.1) and (5.2.2), splitting them into subsequences to be used by RL1-RL4. The construction of N1 and N2 should be straightforward. In Fig. 8 and Fig. 9 we include a mapping of N1. The code for N2 is more tedious and is omitted. Also omitted are the FIFOs for the purpose of simplicity.

We can note that, with both multiple levels of backup and multiple copies of pipelined code, a significant gain in the amount of parallelism can be achieved, while still retaining the major advantages of the pipelined solution scheme, as illustrated in the previous sections.

The overhead with the multiple pipelined scheme includes (1) more operations to be performed due to the multiple backup; (2) the requirement that input coefficients should be generated (or stored) in a way which can be efficiently accessed in the desired order.

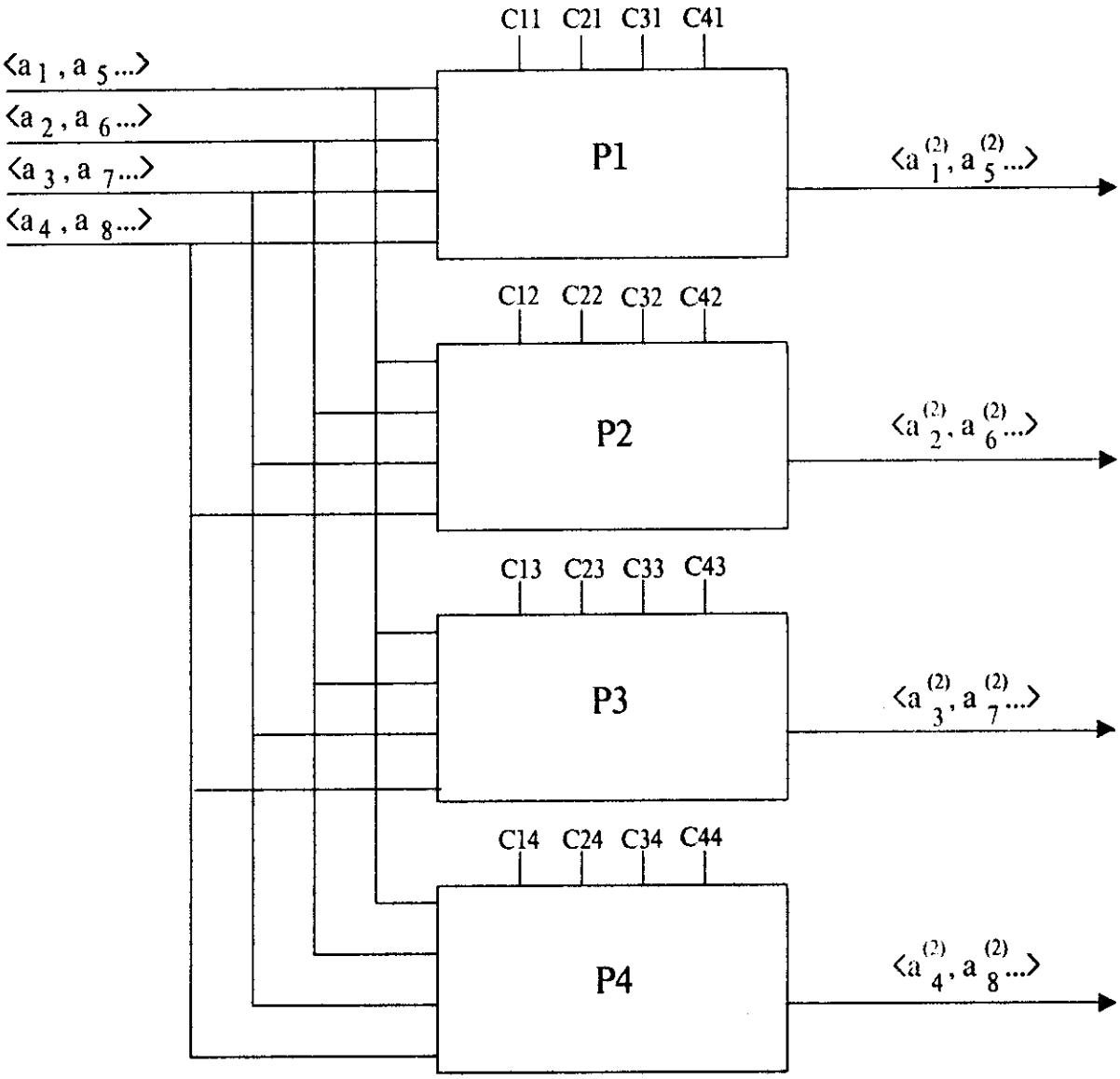
The selection of an adequate number of copies of pipelines and suitable backups should based on both the performance tradeoff and the machine architecture.

## 6. The Stability Aspect of the Solution

In developing new parallel algorithms, high importance attached not only to the speed of the greatest computation rate, but to the numerical stability problems as well.

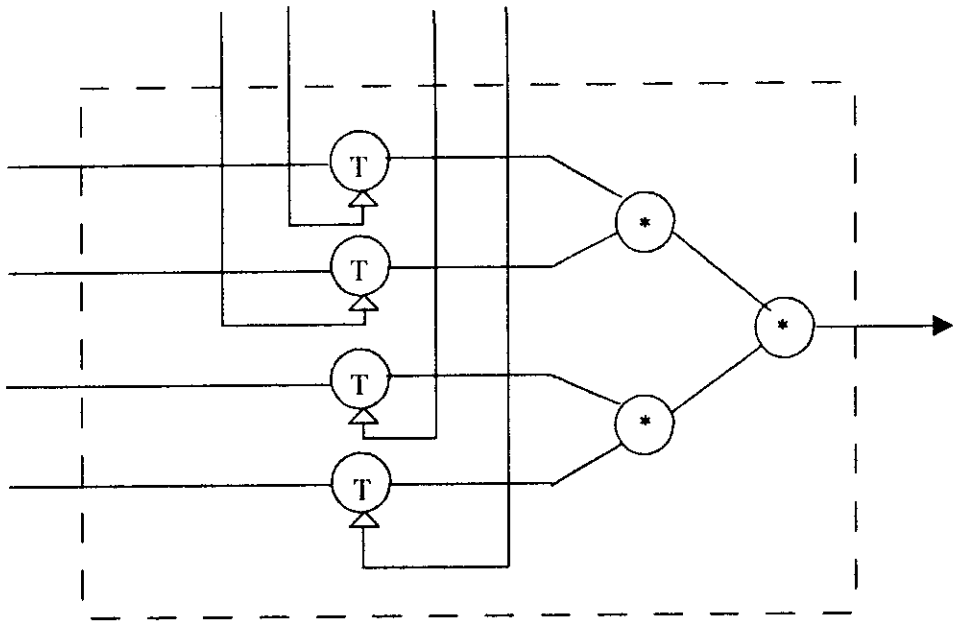
The stability aspect of the pipelined tridiagonal solver has been studied by the author in a second companion paper [13]. Based on the technique of *forward error analysis* [22], the author has shown that the pipelined tridiagonal linear equation solver is numerically

**Fig. 8. A Mapping of the Interface Code**




| | | | |
|---|---|---|---|
| C11: FT...T | C21: T...TF | C31: T...TF | C41: T...TF |
| C12: FT...T | C22: FT...T | C32: T...TF | C42: T...TF |
| C13: FT...T | C23: FT...T | C33: FT...T | C43: T...TF |
| C14: FT...T | C24: FT...T | C34: FT...T | C44: FT...T |

**Fig. 9. A Mapping of the Box P in Fig. 8**




fully stable. The author has also shown that this algorithm has the same degree of stability as a sequential Gaussian elimination algorithm.

As a remark, we note that the nonlinear recurrence expressed in (2.3) can also be directly solved in a maximally pipelined fashion, because it is easy to show that it has a companion function. However, it is due to the stability reason we prefer the solution proposed in this paper [13].

## 7. Simulation Results

The entire code for the pipelined tridiagonal linear equation solver has been translated into the machine code of a proposed static data flow supercomputer, and a preliminary simulation results of the code can be found in [21], which shows that the maximally pipelined throughput can be sustained for each of the three loops.

## 8. Conclusions and Future Research Suggestions

A maximally pipelined approach for solving tridiagonal systems of linear equations is proposed and a complete machine code structure for the algorithm is constructed. The new scheme has several important advantages over other existing parallel algorithms. Although, the primary target machine used in this paper is a static data flow computer, we expect that the principle can also be applied to other data flow computers, such as the dynamic data flow machine [1], although a different perspective of pipelining may required [11]. It is my belief that the basic ideas may also be useful for a conventional parallel machine architecture.

Future research is needed to implement such an algorithm on a real static data flow machine yet to be built. An automatic program transformation for both multiple backup and multiple copies of code presents an interesting challenge to the compiler construction for such computers.

## 9. Acknowledgements

The author is indebted to the constant encouragement and direction from Prof. Jack Dennis at MIT. Dr. Bill Ackerman and Kevin Theobald have read the draft and made interesting comments. Natalie Tarbet has provided valuable helps in the improvements of the English for this paper. Finally, the author is grateful to his wife, Gao Ping, for her typing of the entire manuscripts and preparation of the figures.

**References**

[1] Arvind, Gostelow,K.P. and Plouffe,W, "An Asynchronous Programming Language and Computing Machine", TR-114a, Dept. of Information and Computer Science, Univ. of California, Irvine, Dec. 1978.

[2] Boris, J., "Vectorized Tridiagonal Solvers", Naval Research Laboratory Report No. 3048, 1976.

[3] Buzbee, B. L. Golub G. H. and Neilson C. W., "On Direct Methods for Solving Poison's Equations", SIAM Journal of Numerical Analysis, 7., 1970.

[4] Dennis, J. B. and Gao, G. R. "Maximum Pipelining of Array Operations on Static Data Flow Machine", Proceeding of the 1983 International Conference on Parallel Processing, Aug 23-26, 1983.

[5] Dennis, J. B., "Data Flow for Supercomputers" To appear on the Proceeding of 1984 Compcon., March, 1984.

[6] Dennis, J. B., Gao G. R. and Todd, K., "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. on Computers, c-33, No. 7, July 1984.

[7] Ericksen, J., "Iterative and Direct Methods for Solving Poisson's Equation and Their Adaptability to ILLIAC IV", Center for Advanced Computation Document No. 60, University of Illinois at Urbana, Champaign, 1972.

[8] Forsythe, G. E. and Moler, C. B., "Computer Solution of Linear Algebraic Systems", Prentice-Hall, Englewood Cliffs, N. J., 1967.

[9] Gao, G. R. "An Implementation Scheme for Array Operations in Static Data Flow Computer" MS Thesis, Laboratory for Computer Science, MIT, Cambridge, MA, June 1982.

[10] Gao, G. R. "Homogeneous Approach of Mapping Data Flow Programs", Proceeding of the 1983 International Conference on Parallel Processing, Aug, 1984.

[11] Gao, G. R., "Maximally Pipelined Throughput and Its Token Storage Requirements for Dynamic Data Flow Processors", IBM Research Report, RC-10785, Computer Science, T.J. Watson Research Center, Oct. 1984.

[12] Gao, G. R. "Maximum Pipelining of Linear Recurrence on Static Data Flow Computers", Computation Structure Group Note 49, Lab. for Computer Science, Aug. 1985.

[13] Gao, G. R. "Stability Aspects of a Pipelined Tridiagonal Linear Equation Solver", Computation Structure Group Note 48, Lab. for Computer Science, Aug. 1985.

[14] Hockney, R. "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis", J. ACM. 12. 1965.

[15] Hockney, R., "The Potential Calculation and Some Applications", Methods Computational Phys. 9, pp. 135-211, 1970.

[16] Hockney, R. W. and Jesshope, C. R., "Parallel Computers", Adam Hilger Ltd., 1981

[17] Kershaw, D. "Solution of Single Tridiagonal Linear Systems and Vector Rization of the ICCG Algorithm on the Cray-1", in "Parallel Computations", Ed. by Rodrigue, G. et al., Academic Press, 1982.

[18] Kogge, P. M. "A parallel Algorithm for Efficient Solution of a General Class of Recurrence Equations." IEEE Trans. Comput., Vol. c-22, no. 8, Aug. 1973.

[19] Lambiotte, J. and Voigt, R., "The Solution of Tridiagonal Linear Systems on the CDC Star-100 Computer", ACM Trans. Math Software 1., 1975.

[20] Ortega, J. M. and Voigt, R. G., "Solution of Partial Differential Equations on Vector and Parallel Computers", NASA ICASE Report No. 85-1, 1985.

[21] Redkey, D. H., "Simulation Results for a Pipelined Tridiagonal Linear Equation Solver on a Static Data Flow Computer", Computation Structures Group Note 50, Lab. for Computer Science, M.I.T., Aug. 1985.

[22] Ronsch, W., "Stability Analysis in Using Parallel Algorithms", Parallel Computing, Vol. 1, No. 1, Aug. 1984.

[23] Stone, H., "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", J. ACM, 20., 1973.

[24] Stone, H., "Parallel Tridiagonal Equation Solvers", ACM Trans. on Math. Software, Vol. 1, 1975.