McGill University
School of Computer Science

An Efficient Solution of Tridiagonal Linear Systems on A Dataflow Architecture

ACAPS Technical Memo 20
May 9, 1991

Guang R. Gao
Clifford Sobkowicz†

Advanced Computer Architecture and Program Structures Group

†Environment Canada
Atmospheric Environment Service

3480 University St • Montreal • Canada • H3A 2A7

Contents

List of Figures

Abstract

This paper presents an efficient implementation of tridiagonal linear systems based on *dataflow software pipelining*. The solution method maps the kernel of the computation based on *LU-Decomposition* method into a cluster of loops. Each loop is computing a linear recurrence, and is translated into *software pipelined* dataflow code. In our implementation, elements of arrays are pipelined through the cluster of loops, and fine-grain parallelism in all loops is exploited simultaneousely in two ways:

- *intra-loop parallelism:* within one loop,
  - independent instructions from the body of the same iteration may be issued simultaneousely — called *spacial parallelism* within a loop;
  - instructions from several different iterations at successive stages of the software pipline may also be issued simultaneousely — called *temporal (pipelined) parallelism* within a loop;
- *inter-loop parallelism:* withing the cluster of loops, multiple instructions from different loops can be issued and exeecuted concurrently — called *temporal (pipelined) parallelism* between loops.

The solution is implemented on the McGill Dataflow Architecture Model (MDFA). Simulation results are presented which demonstrate that sufficient parallelism can be exploited to achieve a speedup factor between 5-17, while executing on one dataflow processor with different number of instruction pipelines. Multiple instructions are issued and executed in the dataflow software pipeline, keeping instruction pipelines in the processor usefully busy. The performance impacts of architecture factors and code optimization are discussed.

1 Introduction

Tridiagonal systems of linear equations play an important role in scientific and engineering computation. They arise from processing continuum models of physical systems on digital computers. For example, they are at the heart of finite-difference approximations to second order partial differential equations over a finite distance with boundary conditions at each end [30, 20].

In a continuum model dependent variables, such as temperature, electric potential, etc. vary continuously over variables such as time and space. Changes in a dependent parameter at a fixed point in the model are affected only by the state of the model in close proximity to the point. The model can be described by partial differential equations. When approximated in discrete form each variable is directly effected by its neighbours only. The effects of distant variables must be propagated through intermediate variables [30].

An example of a continuum model is Poisson's equation for potential as a function of charge density:

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = -C(x,y)$$

In one dimension the second derivative

$$\frac{d^2 V(x)}{dx^2} = -C(x)$$

is estimated in discrete form by

$$\frac{(V_{i+1} - V_i) - (V_i - V_{i-1})}{1/N} = -C_i$$

were the domain is represented by $N+1$ evenly spaced points numbered from 0 to N.

This gives rise to the tridiagonal system:

$$V_{i-1} - 2V_i + V_{i+1} = -C_i/N$$

Algorithms for solving linear systems in general are not appropriate to tridiagonal systems because excessive redundant computation is performed on coefficients known to be zero. A number of algorithms have been developed to solve tridiagonal systems involving only the tridiagonal elements of the coefficient matrix. This paper is interested in parallel solutions for tridiagonal linear systems. Algorithms for tridiagonal systems typically involve solving linear recurrences. This presents a challenge to efficient parallel processing.

The problem of solving tridiagonal systems efficiently on parallel computers has been addressed in a number of ways in the last decade [30]. A most prominent algorithm is based on the cyclic reduction technique, first proposed by Golub and Hockney and applied by Buzbee *et al* [5]. One approach uses such parallel techniques to solve the recurrences established by the LU-decomposition method of Gaussian elimination. One such algorithm, known as recursive doubling, was suggested by Stone [29], originally designed for the Illiac IV, and later modified for vector processors. Another approach results from consideration of the requirements of parallel processing and subsequent development of inherently more parallel algorithms. The *odd-even cyclic reduction* algorithm is based on this principle. A major difficulty with cyclic reduction based techniques is the overhead of data rearrangement between computation steps, which could lead to considerable performance degradation on vector computers, such as Cray and NEC. Another problem is the considerable variation in the degree of parallelism between computation steps. This raises the speculation that for sufficiently small systems a sequential algorithm could out perform cyclic reduction. The research presented here draws from Gaussian elimination and cyclic reduction but introduces a new model of computation to the problem.

Dataflow architectures have been proposed as an alternative to traditional von Neumann architectures [1, 4, 7, 17, 28]. In the dataflow model, a computation is described by a data flow graph—a directed graph with nodes that represent actors and arcs that transmit tokens which carry the values to be processed by these actors. Actors become activated for execution when tokens carrying operand values are at their input arcs. In the dataflow model, program execution is intrinsically data-driven in that the order of execution of particular operators is constrained only by data dependencies.

The most widely acknowledged feature of dataflow architectures is their ability to exploit *fine-grain parallelism* — parallelism at the machine instruction level. We argue that this is particularly useful in solving tridiagonal linear systems. Due to the data depencece in the recurrences which dominate the algorithms, the parallelism which can be effectively exploited by traditional vectorization and parallelization methods become very limited. Dataflow architectures provide a solution to overcome such limitations.

This paper presents an efficient implementation of tridiagonal linear systems based on *dataflow software pipelining*. The solution method maps the kernel of the computation based on *LU-Decomposition* method into a cluster of loops. Each loop is computing a linear recurrence, and is translated into *software pipelined* dataflow code. In our implementation, elements of arrays are pipelined through the cluster of loops, and fine-grain parallelism in all loops is exploited simultaneously in two ways:

- *intra-loop parallelism:* within one loop,
  - independent instructions from the body of the same iteration may be issued simultaneously — called *spatial parallelism* within a loop;
  - instructions from several different iterations at successive stages of the software pipeline may also be issued simultaneously — called *temporal (pipelined) parallelism* within a loop;
- *inter-loop parallelism:* within the cluster of loops, multiple instructions from different loops can be issued and executed concurrently — called *temporal (pipelined) parallelism* between loops.

The solution is implemented on the McGill Dataflow Architecture Model (MDFA). Simulation results are presented which demonstrate that sufficient parallelism can be exploited to achieve a speedup factor of between 5 and 17, while executing on one dataflow processor with a varying number of instruction pipelines. Multiple instructions are issued and executed in the dataflow software pipeline, keeping instruction pipelines in the processor usefully busy. The performance impacts of architecture factors and code optimization are discussed.

The reset of the paper is organized as follow. In section 2, we briefly introduce the tridiagonal linear systems of equations to be studied in this paper. In section 3, we briefly introduce the concept of dataflow software pipelining. In section 4, a solution scheme of the tridiagonal systems on dataflow machines is presented. It is based on a scheme which maps linear recurrences into software pipelined dataflow code. In section 5, we introduce briefly the target architecture — the McGill Dataflow Architecture Model (MDFA). In section 6, the simulation results on the target dataflow architecture are studied. In particular, the performance impacts of the architecture factors as well as code optimization methods are discussed. The conculsions are stated in section 7.

2 Tridiagonal systems of linear equations

In tridiagonal systems of linear equations only the major diagonal and the diagonals directly above and below the major diagonal are non zero. In matrix notation they are of the following form:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_3 & b_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ \vdots \\ k_n \end{pmatrix}$$

A tridiagonal system can be represented and solved using the vectors a, b, c, x, and k.

Gaussian elimination can be performed directly on the tridiagonal elements of a coefficient array. The coefficient matrix A is decomposed into a lower bidiagonal matrix L and an upper bidiagonal matrix U such that $A = LU$. For a nonsingular A the decomposition is unique with:

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

and

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

so that

$$LU = \begin{pmatrix} e_1^{-1} & e_1^{-1}u_1 & 0 & \cdots & 0 \\ a_2 & a_2u_1 + e_2^{-1} & e_2^{-1}u_2 & \ddots & \vdots \\ 0 & a_3 & a_3u_2 + e_3^{-1} & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & e_{n-1}^{-1}u_{n-1} \\ 0 & \cdots & 0 & a_n & a_nu_{n-1} + e_n^{-1} \end{pmatrix}$$

Thus $e_1^{-1} = b_1$, and vector u is solved by the following recurrence:

$$\begin{aligned} u_1 &= c_1/b_1 \\ u_i &= c_i/(b_i - a_iu_{i-1}), \quad i = 2, 3, \ldots, n-1 \end{aligned} \tag{1}$$

and, $e_i = u_i/c_i$.

As $Ax = LUx = k$ if $Y = Ux$ then

$$Ly = k \tag{2}$$
$$Ux = y \tag{3}$$

Equation 2 is solved by the following recurrence:

$$\begin{aligned} y_1 &= k_1/b_1 \\ y_i &= (k_i - a_iy_{i-1})/(b_i - a_iu_{i-1}), \quad i = 2, 3, \ldots, n \end{aligned} \tag{4}$$

Subsequently x can be found by the recurrence:

$$\begin{aligned} x_n &= y_n \\ x_i &= y_i - u_i x_{i+1}, \quad i = n-1, n-2, \ldots, 1 \end{aligned} \tag{5}$$

The matrix U is factored by the recurrence (1). The forward elimination is performed by the recurrence (4) and the backward substitution by recurrence (5). Note that the values for e_i in the matrix L are not required.

3 Dataflow Software Pipelining

The dataflow model of computation, which has been proposed as an model for design high-performance computer architectures [3, 2, 6, 9, 21]. In the dataflow model, a computation is described by a data flow graph—a directed graph with nodes that represent actors and arcs that transmit tokens which carry the values to be processed by these actors. Actors become activated for execution when tokens carrying operand values are at their input arcs. In the dataflow model, program execution is intrinsically data-driven in that the order of execution of particular operators is constrained only by data dependencies.

For dataflow computers, a machine-level program can be regarded as a collection of instruction cells that comprise the directed graph, each node in the graph corresponding to an instruction and each arc specifying the destination of its resultant value following the instruction's execution. Unlike von Neumann computers, dataflow computers have no program counter or other form of centralized sequential control mechanism, and consequently, the order of instruction execution is restricted only by data dependencies within the data flow programs.

Fine-grain parallelism exists in a data flow machine level program, as shown in Figure 1. It consists of seven actors divided into four stages. In Figure 1 (a), actors 1 and 2 are enabled by the presence of tokens on their input arcs, and thus can be executed in parallel. Parallelism also exists between actors 3 and 4, and between actors 5 ad 6. In static data flow architecture, pipelining means arranging the machine code such that successive computations can follow each other through one copy of the code. If we present a sequence of values to the inputs of the data flow graph, these values can flow through the program in a pipelined fashion. In the configuration of Figure 1 (b), two sets of computation are pipelined through the graph, and the actors in stage 1 and 3 are enabled and can be executed concurrently. Thus, both forms of parallelism are fully exploited in the graph.

The power of fine-grain parallelism in a data flow computer can be derived from machine-level programs that form a large pipeline in which multiple actors in different stages are

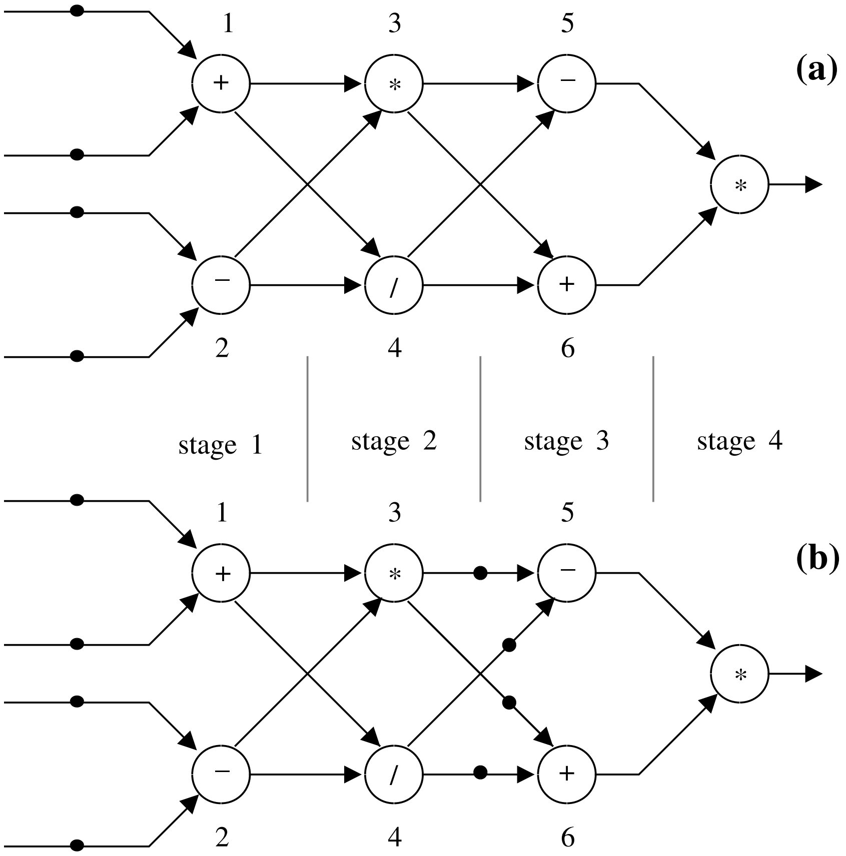

Figure 1: Pipelining of Data Flow Programs

executed concurrently. Each actor in the pipe is activated in a totally data-driven manner, and no explicit sequential control is needed. With data values continuously flowing through the pipe, a sustained parallelism can be efficiently supported by the data flow architecture.

Dataflow software pipelining was proposed as a model for structuring fine-grain parallelism. Here is a summary of some of the main results. A graph is *balanced* if every path from an input node to an output node contain exactly the same number of actors. A graph is maximally pipelined if it is balanced [13]. To achieve maximum pipelining, a basic technique (called *balancing*) is used to transform an unbalanced dataflow graph into a balanced graph by introducing FIFO buffers on certain arcs. To optimally balance a graph, a minimum amount of buffering is introduced into dataflow graphs such that their execution can be fully pipelined. It is known that optimal balancing of an acyclic dataflow graph can be formulated into certain linear programming problems which have efficient algorithmic solutions. A dataflow compiler will use these algorithms to perform code optimization.

For conventional pipelined processor architectures, a technique also called "software pipelining", is proposed where an iteration of a loop is activated before its preceding iteration is completed – thus multiple instructions are in concurrent execution [27, 31, 26]. Apart from the differences in target architecture models, a major distinction here is that the dataflow software pipelining is done at a "fine-grain" level – an instruction is a unit of scheduling, while the software pipelining cited above is done at a "coarse-grain" level – the unit for scheduling being an iteration of a loop.

4 A Solution by Dataflow Software Pipelining

As we demonstrated in Section 2, the solution of a tridiagonal linear system is equivalent to the solution of the three recurrences: (1), (4) and (5). The two recurrences (4) and (5) are first-order linear recurrences. The recurrence (1), as to be demonstrated in this section, can also be translated into an equivalent second-order linear recurrence. This section describes the scheme we adopted in this paper for mapping linear recurrence into software pipelined dataflow graphs.

4.1 Dataflow Software Pipelining of Linear Recurrences

Linear recurrences share with arithmetic expressions a role of central importance in scientific numerical computations. The solving of such recurrences can create bottlenecks in a parallel computer because of the sequential constraints implied by the recursive definition; therefore, it is important to find efficient solutions for computing linear recurrences on these machines.

The technique presented in this section is based upon the use of a *companion function* to transform a linear recurrence so that a *companion pipeline* is introduced in the result data flow graph, to achieve maximum pipelining. The concept of a companion function and the general optimization technique based upon companion functions are described in [23, 22, 24, 25]. The application of companion pipelines for maximum pipelining of data flow graphs was studied in [11, 12, 15].

A first-order linear recurrence is described by the following equations:

$$\begin{aligned} x_1 &= b_1 \\ x_i &= a_i x_{i-1} + b_i, \quad i = 2, \ldots, n \end{aligned} \tag{6}$$

It can also be specified by the following **for-construct** expression:

```
X =
    for i from 1 to n
        T from array_empty
    construct
        let x1 = B[1]
        in
            if i = 1 then x1
            else
                A[i]*T[i-1] + B[i]
            endif
        endlet
    endfor
```

Applying the basic mapping rule and optimization to the code block produces the result data flow graph shown in Figure 2.

In the above figure there are four T-gate actors numbered 1–4. The upper T-gate (labeled 1) selects `B[1]` for the initial value `X[1]`, and the T-gate (labeled 2) selects the first $n-1$ values of `X` for feedback. The lower two T-gate actors (labeled 3 and 4) select the value sequences `A[2],...,A[n]` and `B[2],...,B[n]`, respectively. We note that `MM` selects the initial value $\mathtt{x}_1$ from its input port 1 as its first output `X[1]`. The remaining $n-1$ outputs `x[2],...,x[n]` are selected by `MM` from input port 2.

The loop in Figure 2 has a length of four (from α through nodes 2, 5, 6, and 7 back to α); thus, the loop has the capacity to process two elements of the result array concurrently ($\mathtt{x}_i$ and $\mathtt{x}_{i-1}$) when run at the maximally pipelined rate.[1] However, the recurrence constraint of (10.1) prevents $\mathtt{x}_i$ and $\mathtt{x}_{i-1}$ from being processed concurrently.

[1] Here we assume `MM` counts as one node.

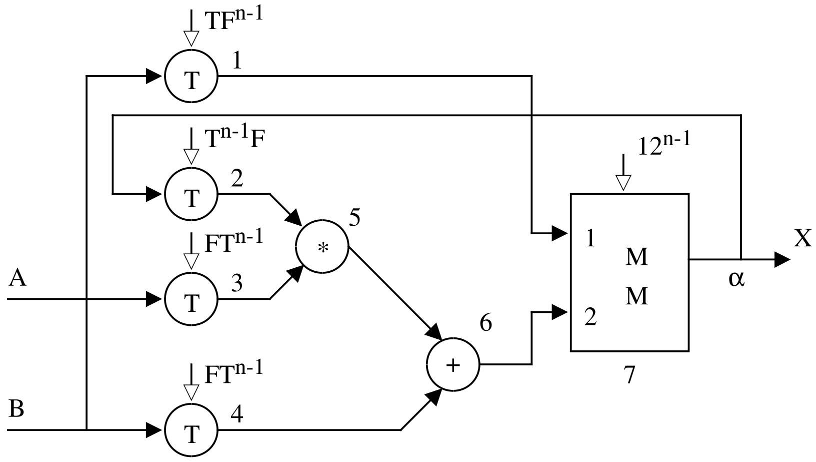

Figure 2: The Data Flow Graph of a First-Order Linear Recurrence

A solution, based upon the use of a companion function, is to relax the constraint imposed by the recurrence, i.e., to perform a transformation which removes the dependence of $\mathbf{x}_i$ on $\mathbf{x}_{i-1}$. Equation (10.1) can therefore be rewritten as

$$x_i = a_i^{(1)} x_{i-2} + b_i^{(1)}, \quad i = 3, \ldots, n \tag{7}$$

where $x_1 = b_1$, $x_2 = a_2 b_1 + b_2$, and

$$a_i^{(1)} = a_i a_{i-1} \tag{8}$$

$$b_i^{(1)} = a_i b_{i-1} + b_i \tag{9}$$

The corresponding **for-construct** expression is shown in Figure 3.

The companion pipeline can be generated automatically by the basic mapping scheme as long as an adequate source-level transformation of the program block is performed; thus, we have illustrated how introducing a companion pipeline can be combined with the basic code mapping scheme to systematically generate efficient data flow graphs.

Now, what left is to demonstrate how the recurrence (10) can be transformed into a linear recurrence. This is important since dataflow software pipelining can also be applied to second-order linear recurrences. In Appendix B we include a short discussion of how this can be done for the recurrence (10) for the purpose of completeness. The readers are referred to [15] for a detailed discussion.

```
X =
    for i from 1 to n
        T from array_empty
    construct
        let x1 = B[1],
            x2 = A[2] * x1 + B[2],
            AP = A[i] * A[i-1],
            BP = A[i] * B[i-1] + B[i]
        in
            if i = 1 then x1
            elseif i = 2 then x2
            else
                    AP * T[i-1] + BP
            endif
        endlet
    endfor
```

Figure 3: A First-Order Linear Recurrence with Backup

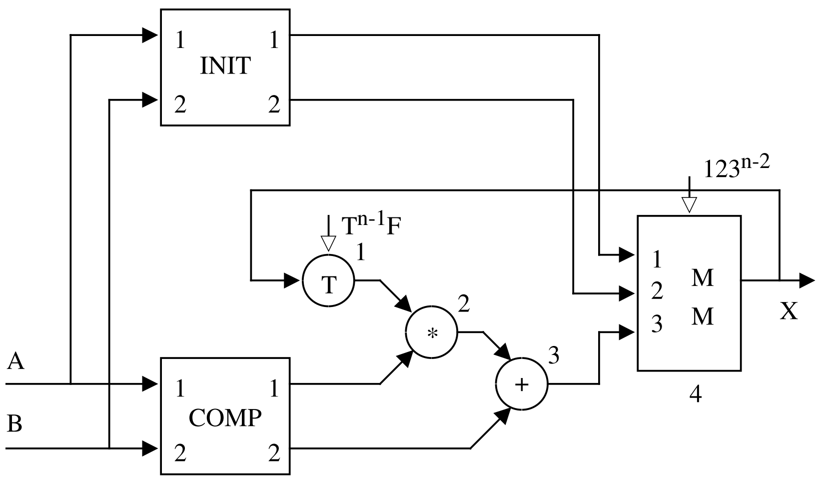

Figure 4: A Maximally Pipelined Data Flow Graph for a FLR

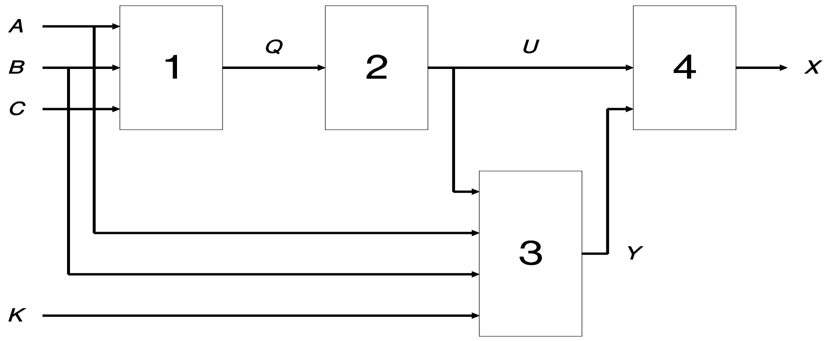

Figure 5: A Dataflow Software Pipeline for a Tridiagonal System

4.2 A Dataflow Software Pipeline for a Tridiagonal System

In Figure 5, we show a top level diagram of a dataflow software pipeline of a tridiagonal linear system. Here A,B,C and K are input arrays, which are fed into the software pipe in a pipelined fashion.

Box 1,2,3,4 are each a software pipelined dataflow code block. Box 1 computes the vector Q defined by the second-order linear recurrence according to the equation (13) in Appendix B. Box 2 computes the vector U from the equation (10). Box 1 and 2 together computes the recurrence (10) in a pipelined fashion. The vector U is sent as an input to both Box 3 and Box 4. Box 3 will compute vector y in a pipelined fashion according to the linear recurrence (4). Finally, the linear recurrence (5) is computed by Box 4, and the final solution vector X is generated.

The internal structure of each boxes as pipelined dataflow code is shown in Appendix (Section: C).

5 The Target Architecture: The McGill Dataflow Architecture Model

The McGill Dataflow Architecture (MDFA) implements the static dataflow model of computation, but it can be extended to efficiently support concurrent (possibly recursive) function

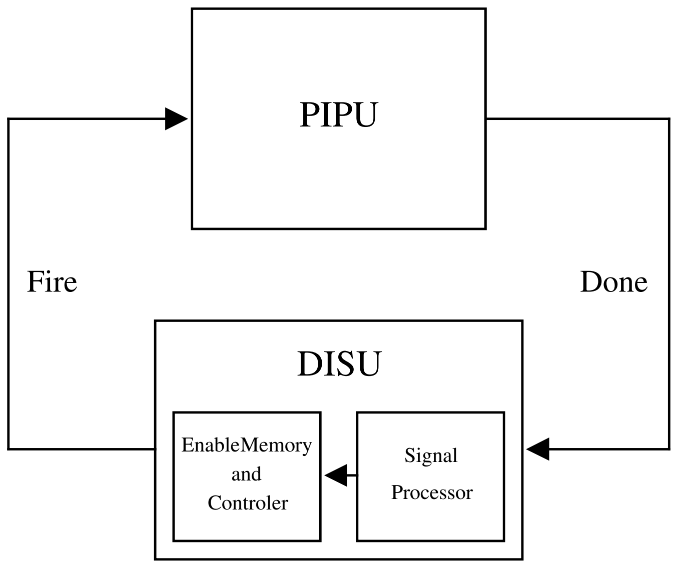

Figure 6: An Argument-Fetching Dataflow Processor

invocations — the subject of another paper [16]. In this paper, we will restrict ourselves to the basic architecture features essential to our discussions.

The McGill Dataflow Architecture has two basic components: a *pipelined instruction processing unit* (PIPU) and a *dataflow instruction scheduling unit* (DISU). A block diagram of this architecture is shown in Figure 6. The PIPU is an instruction processor that uses conventional techniques to achieve fast pipelined operation. Likewise, it is structured similar to a conventional pipelined processing element, except that it has no instruction counter. The PIPU is connected to the DISU by two links: a *fire* link and a *done* link. The fire link transmits the addresses of enabled instructions to the PIPU from the DISU, and following instruction execution, the PIPU informs the DISU of instruction completion across the done link. It does this by transmitting a *done* signal containing the address of the completed instruction, together with a condition code used by the DISU to control the issuance of conditional signals [14]. The DISU, on the other hand, holds a signal graph for the dataflow instructions allocated to the processing element, and it also maintains the pool of enabled instructions available for execution in the PIPU. In this architecture there is no actual data token traffic, even though instruction scheduling is data-driven.

One of the principle logic components within the DISU is the *enable memory*. The enable memory holds flags for several thousand dataflow instructions and selects the enabled instructions according to scheduling priority logic. In an experimental enable memory device, designed by the author, a row logic scheduler has been implemented, called "*Row Watcher*". The row logic selects each row, in turn, and reads the contents of any non-zero row into the column logic. Then, the column logic scans the selected row from left to right, issuing a fire command for each bit that is set [8].

The computation in our dataflow model is specified by a *dataflow program tuple* $\{P, S\}$ and its operational semantics, where P (P-code) is a set of three-address instructions, and S (S-code) is a directed graph called a *signal graph* (cf. fig. 7). An entry of P-code in the program tuple is a three-address instruction (called a *p-instruction* or *p-node*), similar to that in a conventional architecture. S-code contains a set of *s-nodes* each containing a list of actors to notify when its corresponding p-instruction has been executed, an enable count indicating the number of signals to be received before the actor can fire, and a reset count to reset the enable count value once it has fired. Each s-node address is identical to its corresponding p-instruction address.

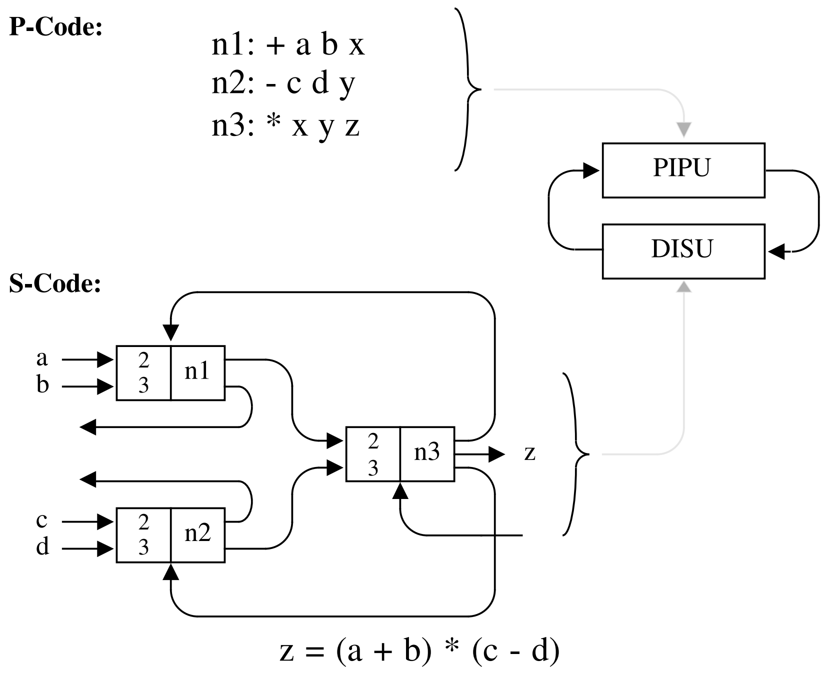

Figure 7: A Dataflow Program Tuple

When an instruction is about to be executed, a *fire signal* containing a p-instruction address, say i, is presented to the *fire* input. The PIPU will then process it in a fashion similar to that of any conventional architecture, i.e., it sends the instruction through the stages in the instrcution processing pipeline such as instruction fetch, operand fetch, execution, and result store. After the execution of an instruction is completed, the PIPU will deliver a *done signal* containing the p-instruction address (in this case, i) to the *done* link.

The scheduling of p-instruction execution is performed by the processing of S-code in the

DISU. The signal processor in the DISU fetches the corresponding signal list of the done signal from a signal list memory and sends a *count signal* containing an actor's address from the signal list to the enable controller. There, the enable counts of actors are fetched from enable count memory, decremented and checked if it is zero – in which case, the count value is reset and its address is sent to PIPU via the fire link.

5.1 Architectural Factors Affecting Fine-Grain Processing

We would now like to point out the features of this architecture which characterizes its capabilities for fine-grain computing. There are three factors affecting its fine-grain processing capabilities:

- the *PIPU capacity* which is the number of PIPU execution pipelines;
- the number of count signals the DISU can process per PIPU pipe beat, which we have termed the *DISU signal processing capacity*; and
- the hardware instruction scheduling mechanism, i.e., the enabled instruction scheduler.

The PIPU capacity specifies the degree of possible concurrency in instruction execution. The DISU signal processing capacity indicates the capability of the processing element in handling fine-grain synchronization. The instruction scheduler, in its present form, implements a "row-watcher" scheduler where the instructions are organized by rows and columns according to their addresses, and enabled instructions are selected one row at a time, scheduling them for execution in the next machine cycle. A fifo queue called a "fire queue" acts as a buffer between the DISU and the PIPU(s) to smooth out the firing rate. The scheduling method can affect the rate of enabled instructions sent to the execution unit(s), and thus affect its parallel processing capabilities.

6 Performance Evaluation

In this section, we describe simulation results of the proposed tridiagonal equation solver. The performance is analyzed and impacts of architecture factors and code optimization are discussed.

6.1 The Experiments

An experimental MDFA architecture testbed has been used throughout the simulation. The major code blocks in Figure 5 has been translated into the MDFA assembler code, i.e. A-Code graphs where simple nodes in the former are mapped into instruction tuples (see Section 5). An assembler(DASM) is used to generate executable code for an interpreter/simulator(AD) of the architecture. Execution time is reported in machine time units or cycles. The above MDFA architecture testbed is described in [18].

The simulator allows specification of the number of PIPU's available for parallel processing, the number of stages in each PIPU, and the DISU's capacity in signals per cycle. Scheduling of enabled instructions was according to a row watcher [19, sec. 6.2] algorithm using an enable memory with one bit for each instruction. When the DISU determines that an instruction can be executed the corresponding bit in the enable memory is set. The enable memory is arranged in rows and columns. The row watcher selects enabled instructions from one row at a time and schedules them for execution in the next cycle. The scheduled instructions either begin execution in an available PIPU or are placed in a *FIFO fire queue* to await for a free PIPU. Performance metrics reported by the AD simulator are listed in appendix A.

The arrays were generated with a pre-processor so as to allow adjustment of the size of the system. Initially a system with 32 dimensions was used in order to verify the code. For compaarison purpose, the same system was solved using the decomposition and solution routines from the *LINPACK* benchmark [10] in order to obtain values for the solution vector. The solution vector from the dataflow program was verified as equal to the solution from *LINPACK*.

6.2 The Effects of Software Pipelining

We perform experiments with two types of code structures:

- software pipelined but not balanced, and
- software pipelined and balanced.

The software pipelined but not balanced version of the A-code program required 124 nodes in the dataflow graph. The balanced version required an additional 8 nodes plus 11 FIFO buffers for balancing. The length of the FIFO buffers ranged from between 2 stages to 4 stages.

Both the unbalanced and balanced versions of the program were simulated with varying numbers of six stage PIPU's. The DISU capacity was maintained at three signals per PIPU as simulations indicated average per instruction requirements of 2.72 for the unbalanced version and 2.79 for the balanced.

The test data consisted of a tridagonal system generated as in section 6.1 but with 900 dimensions . This was considered a large system which would require long run times which

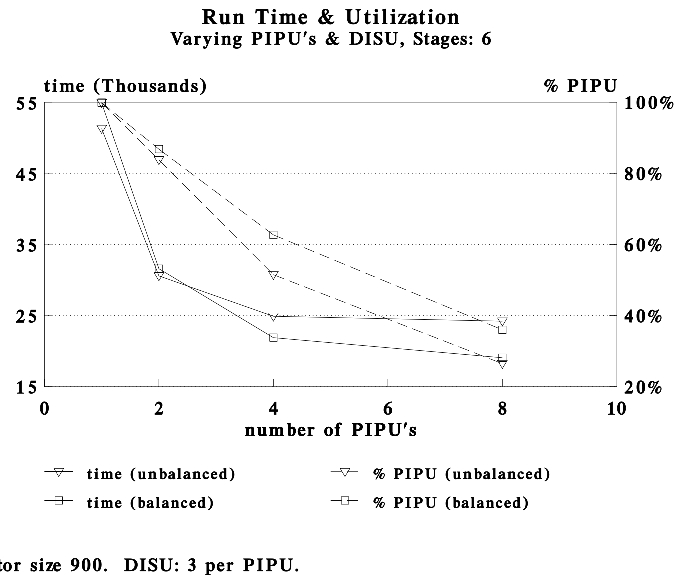

Figure 8: Simulation performance

would allow accurate comparison of the two versions of the program. Also long vectors would assure the execution was dominated by pipelined processing of the data rather than start up and shut down overhead.

The total run times and PIPU utilization are graphed as a function of the number of PIPU's in figure 8. The simulation results are summarized in Table 1.

First, we can observe that both balanced and unbalanced versions have enough fine-grain parallelism to saturate a single PIPU. However, the overall run time of the balanced version is greater because the additional instructions required to perform the balancing as well as the overhead of the additional signals. The added instruction level parallelism serves to increase pending instruction population and instruction delay.

The addition of a second PIPU plus additional DISU capacity results in close to a doubling in performance as indicated by the total run times. The unbalanced version is still faster because, as indicated by the figures for population of enabled instructions and average instruction delay, two PIPU's are insufficient to execute all the instructions being enabled by the highly parallel balanced software pipeline.

Four PIPU's provide sufficient capacity to exploit the additional parallelism introduced by balancing. The balanced version is finally marginally superior to the unbalanced.

With eight PIPU's the improvement in performance of the unbalanced version has appeared to level off while the balanced version has continued to improve according to the r_∞^{-1}

PIPU Stages: 6, DISU Signal Capacity (C): 3 per PIPU								
	Unbalanced				**Balanced**			
# PIPU's	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
PIPU Utilization	99.9%	83.8%	51.5%	26.5%	99.9%	86.8%	62.7%	36.0%
Pipelined Speedup	5.994	10.06	12.35	12.70	5.993	10.42	15.04	17.27
Avg. Count Signal sent per instruction **(S)**	2.719	2.719	2.719	2.719	2.786	2.786	2.786	2.786
Total Instructions Executed	51,325	51,325	51,325	51,325	54,939	54,939	54,940	54,940
Total Run Time	51,387	30,618	24,927	24,238	55,002	31,636	21,914	19,086
Total Idle Cycles	54	9,912	48,384	142,580	64	8,334	32,718	97,751
Total Count Signals	139,543	139,540	139,540	139,540	153,080	153,080	153,080	153,080
Population	11.237	6.917	4.056	3.158	27.169	21.409	8.661	5.955
Average instruction Delay	2.833	0.984	0.097	0.026	5.416	2.427	0.571	0.074

Table 1: Simulation performance metrics

values. The balanced version now takes 21% less time. The run time of the balanced has only been reduced by 12% with a doubling in PIPU's. It is suspected that low PIPU utilization could now be a factor.

6.3 Effects of architecture parameters

We have studied the impacts by key architecture factors on the performance of the tridiagonal solver. These factors are: (1) the number of PIPUs; (2) the DISU capacity; (3) the number of stages in a PIPU. Each parameter was varied separately while holding the others constant and the total run time and PIPU utilization were graphed as a function of the parameter value.

6.3.1 Number of PIPU's

In figure 9 the number of PIPU's was varied from 1 to 8 by 1, while the number of stages in each PIPU was set to 6 and the DISU had a signalling capacity of 4.

The graph indicates no improvement in performance beyond 3 PIPU's for the balanced version of the program (and 2 PIPU's for the unbalanced). This would indicate that DISU capacity is insufficient to enable instructions at a rate required to utilize more than 3 PIPU's. This is confirmed by the utilization curve. When the number of PIPU's is doubled from 4 to 8 the utilization is cut in half, from 35% to 17.5%, so there is no net gain in processing speed. In other word, the DISU capacity imposes the limit for a peek execution rate.

The balanced version requires more time because balancing requires additional dataflow instructions and additional signals for FIFO synchronization. A single PIPU is fully utilized by both versions and thus can not benefit from the additional parallelism. The execution limit imposed by the DISU prohibits additional PIPU's from exploiting the additional parallelism. Note that, in this implementation, a FIFO is actually made of a chain of identity instructions. Such inefficiency could be avoid if further optimization is performed [15].

An interesting observation is the performance with 2 PIPU's. The performance of the unbalanced version is optimal and it utilizes the PIPU's slightly more than the balanced. The unbalanced version becomes slower slightly with the third and forth PIPU and then becomes constant. The balanced version improves with the third PIPU and then becomes constant. The gap between the balanced and unbalanced widens from 3625 cycles with 1 PIPU to 3815 cycles with 2 and then narrows to 2732 with 3 and 2704 with 4 or more. Apparently with the unbalanced version sufficient parallelism is present to utilize 2 PIPU's effectively but the benefits of more are overridden by the expense of congesting the DISU, due to more instructions completing in parallel. With the balanced version the additional demand on the DISU prevents utilization of 2 PIPU's as effectively as with the unbalanced

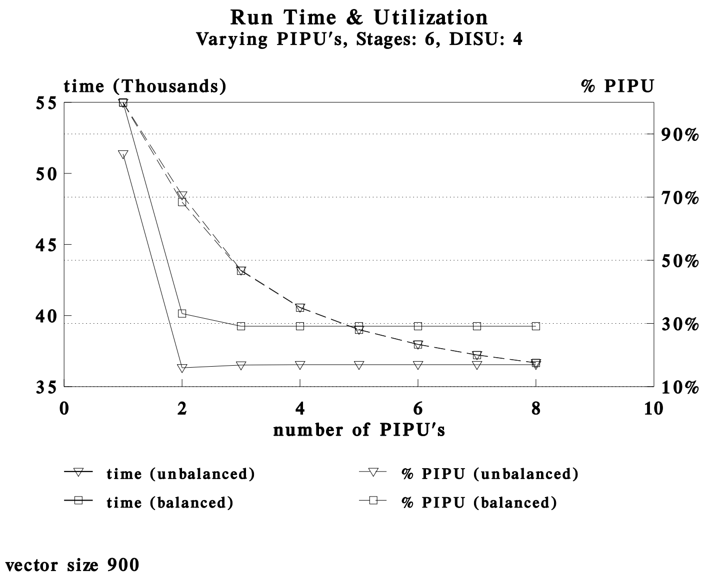

Figure 9: Effect of number of PIPU's

version. The additional parallelism in the balanced version utilizes a third PIPU to a greater degree than in the unbalanced version. As a result, the performance improved with the third PIPU, before becoming dominated by the DISU limit and becoming constant.

6.3.2 DISU capacity

In figure 10 the DISI capacity was varied from 2 to 16 by 2, while the number of stages in a single PIPU was set to 6.

We observer that the PIPU was under utilized by a DISU with a capacity of 2 and saturated by a capacity of 4 and beyond. Execution was subsequently limited by the PIPU.

The balanced version is slower because the PIPU must execute additional identity instructions to balance the dataflow graph. As we mentioned earlier, this may be avoided if FIFO implementation is improved.

6.3.3 Number of stages in PIPU's

In figure 11 the number of stages in each PIPU's was varied from 2 to 14 by 2. The number of PIPU's was set to 1 and the DISU had a signalling capacity of 4.

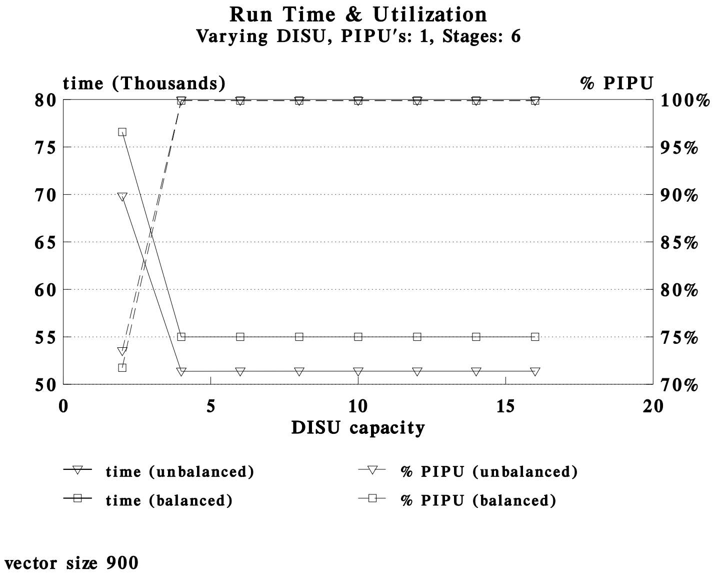

Figure 10: Effect of DISU capacity

The DISU was able to fully utilize a pipeline with up to 6 stages. Utilization and performance then decreased. The balanced version of the program utilized a pipeline with 8 or more stages more than the unbalanced but required more time to run due to the additional instructions and signals.

7 Conclusions

A tridiagonal linear systems based on dataflow software pipelining has been implemented on the McGill argument-fetching dataflow architecture testbed. Th effect of code optimization through code balancing were examined using the MDFA testbed. The effects of various machine parameters are also studied.

This study has indicated that dataflow software pipelining can offer an unique opportunity for exploiting fine-grain parallelism in solving tridiagonal linear systems. Although the code is dominated by loops computing several recurrences, there still exists enough instruction level parallelism to keep up two instruction pipelines (each with 6 stages) to be usefully busy!. Balancing of software pipelines can provide additional parallelism and improved performance.

As these improvements in performance are obtained by exposing parallelism in algorithms at vraious levels: within one loop (spacial and temporal pallelism), and between loops (temporal parallelism).

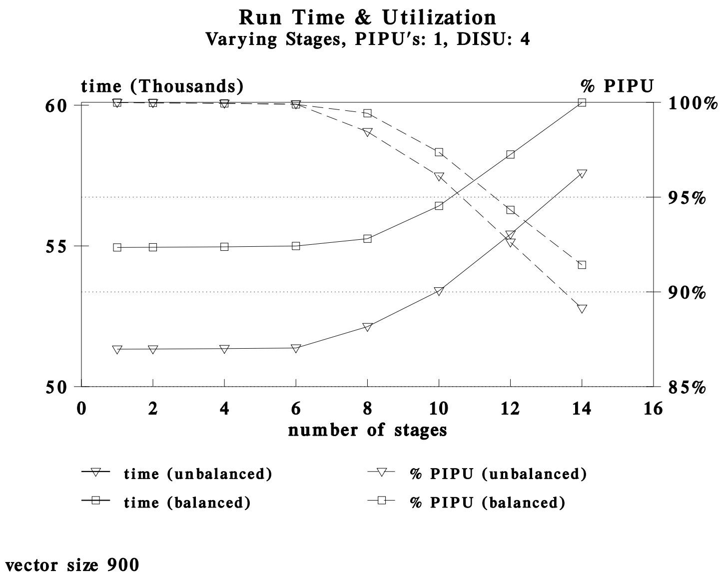

Figure 11: Effect of number of PIPU stages

Balancing of the software pipeline offers a potential increase in performance of 39% according to the study of the algorithm on the idealized machine. Realizing any improvement required four PIPU's. Thus performance can be improved with software pipelining and balancing but careful consideration must be given to the implementation of these techniques.

The study of the effects of the architecture parameters clearly indicated the necessity to match the execution and scheduling capacity. Either component can be a bottleneck which nullifies excess capacity in the other component. In other word, fine-grain synchronization is no free lunch, and we must structure the code to exploit the paralleism to match the architecture capacity.

A more extensive study of various combinations of number of PIPU's, length of PIPU pipelines, DISU capacity, and enabled instruction dispatch schemes could provide further information on potential weaknesses in these components and in there interactions.

8 Acknowledgment

We would like to thank the Natural Science and Engineering Research Council (NSERC) for a grant in supporting this work. We also enjoy the support from the members of ACAPS (Advanced Computer Architecture and Program Structures Group) at McGill University. In particular, we would like to thank Russel Olsen and Philips Wong for their discussions, and Jean-Marc Monti for his assistance with graphics. We wish to acknowedge the support of the

Atmospheric Environment Service of Environment Canada, especially Wayne Hodgins and Len Feldman, of the Canadian Meteorological Centre.

A Performance Metrics

Values reported by the AD macro simulator [32].

- *Total Instructions Executed* records the number of the three-address PIPU instructions that have been executed in the PIPU.
- *Total Run Time* records the number of virtual machine cycles elapsed.
- *Total Idle Cycles* records the total number of machine cycles where the instruction processing units (PIPUs) were idle, due to the lack of available execution instructions.
- *PIPU Utilization* of N PIPUs is the ratio

$$\frac{(\text{Total Run Time} * N) - \text{Total Idle Cycles}}{\text{Total Run Time} * N}$$

- *Pipelined Speed-Up* is the ratio

$$\frac{\text{Non-pipelined Sequential Time}}{\text{Total Run Time}}$$

  where "Non-pipelined Sequential Time" is the product of the "Total Instructions Executed" and the execution time per instruction. For example, if a PIPU unit has six stages, then in a sequential execution manner, the executon time per instruction is six machine cycles. Speedup can be achieved if pipelining is used effectively.

- *Total Count Signals* records the number of count signals in the DISU produced by the executed program. Conceptually, this metric represents the signal traffic in the DISU.
- *Population* records the average population in the enable memory and in the fire queue of the instructions that were ready to fire but which were delayed for more than one machine cycle, waiting in the enable memory or in the fire queue.
- *Average Instruction Delay* is the average no. of cycles an enabled instruction spends waiting in the fire queue.

B Transformation of Recurrence (1) into a Linear Recurrence

In order to perform the decomposition of the coefficient matrix by evaluating the vector u with the recurrence (1) (see Section 2) vector q is introduced such that

$$u_i = -q_i/q_{i+1}. \tag{10}$$

Substitution of u_i in 1 by 10 produces the relation

$$q_1 = 1, \tag{11}$$

$$q_2 = -b_1/c_1, \tag{12}$$

$$a_i q_{i-1} + b_i q_i + c_i q_{i+1} = 0, \quad i = 2, 3, \ldots, n-1. \tag{13}$$

This is restated as a second order linear recurrence

$$q_i = \alpha_i q_{i-1} + \beta_i q_{i-2}, \quad i = 2, 3, \ldots, n-1 \tag{14}$$

Where

$$\alpha_i = -b_{i-1}/c_{i-1}, \tag{15}$$

$$\beta_i = -a_{i-1}/c_{i-1}. \tag{16}$$

Substitution of q_{i-1} in 14 with the equivalent expression from 14 gives

$$q_3 = b_2 b_1/c_1 c_2 - a_2/c_2, \tag{17}$$

$$q_i = \alpha_i^{(1)} q_{i-2} + \beta_i^{(1)} q_{i-3}, \quad i = 4, 5, \ldots, n \tag{18}$$

Where

$$\alpha_i^{(1)} = \alpha_i \alpha_{i-1} + \beta_i, \tag{19}$$

$$\beta_i^{(1)} = \alpha_i \beta_{i-1}. \tag{20}$$

Thus the decomposition of the coefficient matrix is achieved with a third order recurrence involving only the second and third previous terms. Processing of the the current term can proceed in parallel with the immediately previous term.

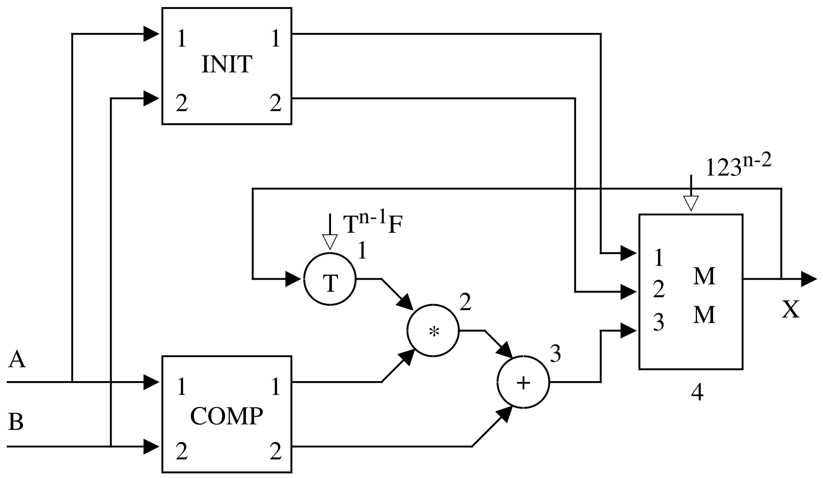

Figure 12: A Dataflow Software Pipeline for a Diagonal System

C The Structure of The Dataflow Code Blocks

In Figure 12, we show the internal dataflow code structure of the diagram in Figure 5. The code can be understood easily from the principle of mapping linear recurrences into pipelined dataflow code. Note that the dotted line boxes contain the companion pipelines introduced for the purpose of maximaly pipelining. The small solid line boxes denote FIFO buffers for balancing purpose, and the numbers indicate the size of the buffers. The boolean value sequences are generated by the box C.

In our simulation, the code is translated into machine code for the McGill Dataflolw Architecture.

References

[1] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[2] Arvind and et al. The tagged token dataflow architecture (preliminary version). Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., August 1983.

[3] Arvind, K. P. Gostelow, and W. Plouffe. *An Asynchronous Programming Language and Computing Machine*. Department of Information and Computer Science, University of California, Irvine, December 1978.

[4] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[5] B. L. Buzbee, G.H. Golub, and C. W. Neilson. On direct methods for solving poisson's equations. *SIAM J. Numerical Analysis, 7*, 1970.

[6] J. B. Dennis. First version of a data flow procedure language. Technical Report MIT/LCS/TM-61, Laboratory for Computer Science, MIT, 1975.

[7] J. B. Dennis. Evolution of the static dataflow architecture. In *Advanced Topics in Dataflow Computing*. Prentice-Hall, 1991.

[8] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the Supercomputing '88 Conference*, pages 368–373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[9] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *The Second Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[10] Jack J. Dongarra. The LINPACK benchmark: An explanation. In C. D. Polychronopoulos, editor, *Proceedings of the 1987 Conference on Supercomputing, Athens, Greece*, pages 457–474, Berlin, June 1987. Springer-Verlag, LNCS-297.

[11] G. R. Gao. An implementation scheme for array operations in static data flow computer. Master's thesis, Laboratory of Computer Science, MIT, Cambridge, MA, June 1982.

[12] G. R. Gao. Maximum pipelining linear recurrence on static data flow computers. Computation Structures Group Memo 49, Laboratory for Computer Science, MIT, August 1985.

[13] G. R. Gao. A maximally pipelined tridiagonal linear equation solver. *Journal of Parallel and Distributed Computing*, 3(2):215–235, June 1986.

[14] G. R. Gao. A flexible architecture model for hybrid dataflow and control-flow evaluation. ACAPS Technical Memo 07, School of Computer Science, McGill University, Montreal, January 1989.

[15] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[16] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proceedings of the PARBASE '90 Conference*, Miami Beach, FL, March 1990.

[17] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Towards efficient fine-grain software pipelining. In *Proceedings of the ACM International Conference on Supercomputing*, Amsterdam, the Netherlands, June 1990.

[18] G. R. Gao and Z. Paraskevas. Compiling for dataflow software pipelining. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, Illinois, August 1989. To be published by Pitman in their series Monographs in Parallel and Distributed Computing.

[19] G. R. Gao and Z. Paraskevas. Dataflow software pipelining: A case study. ACAPS Design Note 06, School of Computer Science, McGill University, Montreal, February 1989. Presented as a short paper at the International Conference on Supercomputing '89, Crete, Greece, June 1989.

[20] R. W. Hockney and C. R. Jesshope. *Parallel Computers.* Adam Hilger Ltd., Bristol, 1981.

[21] R. M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *AFIPS Conference Proceedings, vol. 48*, pages 613–622, 1979.

[22] P. M. Kogge. Maximum rate pipelined solutions to recurrence problems. In *Proceedings of the 1st Computer Architecture Symposium*, pages 71–76, December 1973.

[23] P. M. Kogge. A parallel algorithm for efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8), August 1973.

[24] P. M. Kogge. Parallel solutions of recurrence problems. *IBM Journal of Research and Development*, 18(2), March 1974.

[25] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, New York, 1981.

[26] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[27] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[28] S Sakai and et al. An architecture of a dataflow single chip processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 46–53, Israel, 1989.

[29] H. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1, 1975.

[30] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

[31] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.

[32] Y. B. Wong. AD reference manual (version 1.4). ACAPS Design Note 16, School of Computer Science, McGill University, Montreal, October 1989.