LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

High Speed Data Flow Computer Architecture for the Solution of the Navier-Stokes Equations

Computation Structures Group Memo 225
March 1983

Jack B. Dennis

This research was supported under a Joint Research Interchange with the NASA Ames Research Center, Moffett Field, California, Interchange No. NCA2-OR425-101.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

At MIT, the Computation Structures Group of the Laboratory for Computer Science is applying knowledge gained in our study of data flow concepts of computer and program organization to the specification, design, programming and evaluation of supercomputers.

Data flow computers make a radical departure from conventional concepts of computer architecture because all instruction sequencing is controlled by the flow of data. Thus, data flow computers promise efficient parallel computation limited in speed only by data dependencies in the calculation being performed. These architectures also provide natural support for functional programming, which promises greatly improved programmability for large-scale applications. The programming language VAL, designed and implemented by the Computation Structures Group, is the first functional programming language developed expressly for high performance numerical computation.

This report focuses on the mapping of the algorithm for the solution of the Navier-Stokes equations onto a data flow machine. We discuss the structure of the Navier-Stokes code as translated from the original Fortran into VAL, and present a mapping of the VAL program onto a data flow supercomputer. On the basis of this analysis, we are able to calculate the number and types of hardware units required to build a data flow machine and the structure of the machine code for the Navier-Stokes problem using this machine such that it will outperform conventional super computers in the areas of speed, size, and power consumption.

Introduction

This report concerns the evaluation of data flow concepts for surpassing the performance of conventional supercomputers for the computations of numerical aerodynamics. For this study a specific computer program was chosen – a Fortran code for the implicit computation of Navier-Stokes problems over a three dimensional grid. The work accomplished by the study is:

1. Recoding the Fortran program into VAL, a functional style programming language designed at MIT as a source language for data flow computation.

2. A preliminary analysis of the VAL program to outline hypothetical machine code structures for use with a data flow supercomputer.

3. A proposed data flow computer configuration capable of performing the Navier Stokes computation at approximately 1000 MFLOPS on the basis of our analysis.

It appears that such a computer configuration could be built using on the order of 25,000 semiconductor devices (fewer than ten types) in NMOS technology. The machine would consume less power and occupy less space than present day supercomputers.

The study reported here uses the same methodology we have employed for a similar analysis of a global weather model computation, also provided to us by NASA in the form of a Fortran benchmark program. Our work on the weather code, which employs an explicit computation scheme, is reported in a paper submitted for publication [5]. We show that one time step of the nine-layer model using a 144 by 87 grid can be computed in five seconds on a data flow supercomputer running at about 250 million floating point operations per second (MFLOPS). The Navier-Stokes problem presents a more difficult challenge since the implicit algorithm has a less regular structure.

In the following sections, we describe the Fortran program for the Navier-Stokes computation and the form of data flow supercomputer chosen as the target architecture for analysis. Then we present the recoding of the Navier-Stokes computation in VAL and the program structures that allow us to make performance estimates for the proposed architecture. We conclude with a brief discussion of the work required to ensure that the estimated performance will be achieved using a feasible compiler and a practical data flow machine.

The Navier-Stokes Benchmark Problem

The computational problem for analysis is the solution of the Navier-Stokes equations on a three-dimensional grid using implicit finite-difference techniques. The size of the computational grid is chosen to be 100 by 100 by 100 to provide sufficient spatial definition for the modeling of turbulent flow, a major objective of computational aerodynamics. Since five scalar physical quantities are associated with each grid point, as well as three

coordinate values mapping the computational grid into physical space, the data base describing the state of the problem at the end of each step consists of eight million scalar values.

The equations for the three-dimensional problem factor into a set of 100 times 100 independent difference equation systems for each of the three grid dimensions. Each of the ten thousand linear systems for one grid dimension is a block tridiagonal system of 100 times 5 unknowns. The solution of these systems makes up the major part of the required computation. Computation at the rate of 1000 MFLOPS is needed to meet performance objectives – about one time step per two seconds of computation.

Rewriting the Navier-Stokes Code in VAL

The programming language VAL was developed by the Computation Structures Group of the MIT Laboratory for Computer Science as the source programming language for computations to be performed by data flow computation. VAL is designed so that programs are expressed in a form that allows easy determination of program parts that may be executed simultaneously. Each module of a VAL program includes an interface description that specifies each input value the module uses, each output value the module produces, and the data types of all inputs and outputs. Modules in VAL do not refer to global variables, do not modify their input values, and therefore do not have "side-effects". The language includes a powerful **forall** construct that expresses the simultaneous computation of the elements of an array. This construct finds frequent use in programs for the numerical solution of partial differential equations.

To rewrite the Fortran version of the Navier-Stokes code in VAL we constructed a module in VAL for each subprogram of the Fortran code. The first step was to identify the true input and output data for each module by analyzing the use of common data by the corresponding Fortran subprograms. Translating each module was reasonably straightforward although great care was needed to be sure that the correct generation of each Fortran variable was used in writing the VAL code. Figure 1a shows a portion of the Fortran subprogram Smooth and the corresponding code written in VAL is shown in Figure 1b.

```
      SUBROUTINE SMOOTH
      COMMON JMAX,KMAX,LMAX,SMU
      COMMON S(30,30,30)
      COMMON Q(30,30,30)
      COMMON D(30,30,30)
      SM1 = SMU*.5
      JM = JMAX-1
      JMM = JMAX-2
      KM = KMAX-1
      LM = LMAX-1
      DO 10 L = 2,LM
      DO 10 K = 2,KM
C
C     FOURTH-ORDER SMOOTHING
C
      DO 20 J = 3,JMM
      S(J,K,L) = S(J,K,L) - SMU * (
     1   Q(J+2,K,L) * D(J+2,K,L)
     2   - 4. * Q(J+1,K,L) * D(J+1,K,L)
     3   + 6. * Q(J,K,L) * D(J,K,L)
     4   - 4. * Q(J-1,K,L) * D(J-1,K,L)
     5   + Q(J-2,K,L) * D(J-2,K,L)
     6   ) / D(J,K,L)
   20 CONTINUE
C
C     SECOND-ORDER SMOOTHING AT THE BOUNDARIES
C
      S(2,K,L) = S(2,K,L) + SM1 * (
     1   Q(3,K,L) * D(3,K,L)
     2   - 2. * Q(2,K,L) * D(2,K,L)
     3   + Q(1,K,L) * D(1,K,L)
     4   ) / D(2,K,L)
      S(JM,K,L) = S(JM,K,L) + SM1 *
     1   (Q(JMAX,K,L) * D(JMAX,K,L)
     2   - 2. * Q(JM,K,L) * D(JM,K,L)
     3   + Q(JMM,K,L) * D(JMM,K,L)
     4   ) / D(JM,K,L)
   10 CONTINUE
      RETURN
      END
```

Figure 1a. The program module Smooth written in Fortran (reduced to one quantity and one direction for illustration).

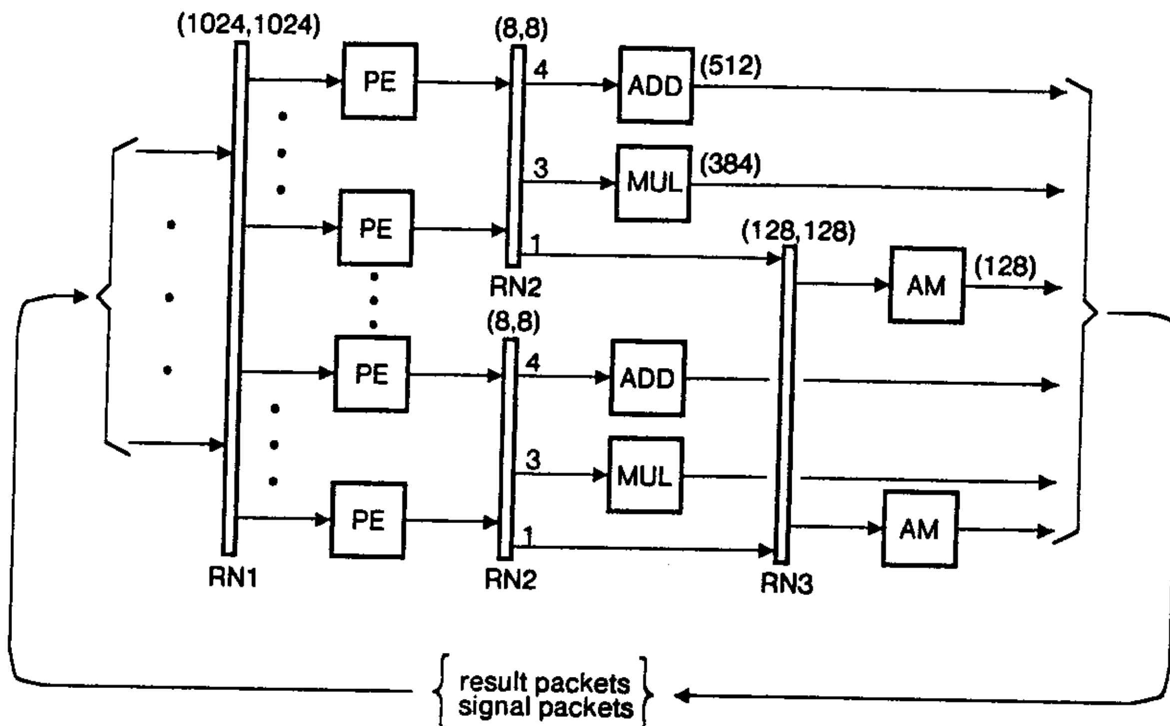
Each of the VAL program modules for the Navier-Stokes problem has been successfully translated by the VAL translator implemented at MIT, so each is known to be free of syntactic and some kinds of semantic errors. Since the VAL translator performs full compile-time type checking, we know in addition that each of the modules is type correct.

```
function Smooth(
   Q: three_dim;                    %% state values
   S: three_dim;                    %% residuals
   D: three_dim;                    %% Jacobian
   smu: real;                       %% smoothing parameter
   jmax, kmax, lmax: integer        %% index limits
   returns three_dim)               %% SA value

   type three_dim = array[array[array[real]]];

   let
      sm1: real := .5 * smu;
      SA: three_dim :=
      forall J in [1, jmax], K in [1, kmax], L in [1, lmax]
      construct
         if J = 1 | K = 1 | L = 1 | J = jmax | K = kmax | L = lmax

            %% boundary point -- no change

         then S[J, K, L]

         elseif J = 2 | J = jmax - 1 then

            %% point is next to boundary in J-direction
            %%  -- use second order formula

         S[J, K, L] + sm1 * (
               + Q[J + 1, K, L] * D[J + 1, K, L]
               - 2.0 * Q[J, K, L] * D[J, K, L]
               + Q[J - 1, K, L] * D[J - 1, K, L]
            ) / D[J, K, L]

         else

            %% interior point -- use forth order formula

            S[J, K, L] - smu * (
               + Q[J + 2, K, L] * D[J + 2, K, L]
               - 4.0 * Q[J + 1, K, L] * D[J + 1, K, L]
               + 6.0 * Q[J, K, L] * D[J, K, L]
               - 4.0 * Q[J - 1, K, L] * D[J - 1, K, L]
               + Q[J - 2, K, L] * D[J - 2, K, L]
            ) / D[J, K, L]
         endif
      endall;
   in SA
   endlet
endfun
```

Figure 1b. The program module Smooth written in VAL (reduced to one quantity and one direction for illustration).

A Data Flow Supercomputer

Data flow computer architecture is a fundamentally different way of organizing a computer system. The difference is the manner in which instruction execution is controlled: a data flow computer has no program counter. Rather, each instruction is activated when it has received (as the results of other instructions) the data on which it is to operate.



PE: Processing Element
ADD, MUL: Functional Units
AM: Array Memory Unit

RN1: Distribution Routing Network
RN2: Functional Unit Routing Network
RN3: Memory Routing Network

Figure 2. A Data Flow Supercomputer.

For high performance numerical computation we propose a concept of data flow architecture [3] in which the instructions of a machine level program are loaded into specific memory locations in the machine before computation begins, and only one instance of an instruction is active at a time. The overall structure of the proposed machine is shown in Figure 2. Instructions of a data flow program are held in the local memory of the *processing elements.* Each processing element is equipped to recognize which of the

instructions it holds have been enabled for execution by arrival of the operand values it needs. If an enabled instruction calls for a scalar arithmetic operation (a floating point addition or multiplication, for example), the instruction, including its operands, is sent to a *functional unit* capable of performing that operation. The *array memory* units are provided to hold arrays of data making up the (possibly very large) data base of the computation and are accessible from any of the processing elements. Instructions are provided that build array values or access elements of arrays stored in the array memory units. These instructions are sent to the appropriate array memory unit through the *memory routing network*. Other instructions such as those calling for duplicating data values, for Boolean operations, and for simple tests, are performed within the processing elements.

Instruction execution in a functional unit or array memory unit yields result packets each of which consists of a data value and a destination field that specifies the target instruction for the result packet. The result packets are sent to processing elements that hold the target instructions through the *distribution routing network*. The way in which a data flow processing element handles result packets, identifies enabled instructions, and initiates instruction execution has been described in several publications [2, 7, 3]. The use of packet routing networks in data flow computers is discussed in [3, 4] and analyses of efficient structures for routing networks have been done by Andy Boughton [1].

Because data flow computers are radically different from familiar sequential stored program computers, it is impossible to predict the performance of a data flow computer through a comparison of the speed and cost of its basic units with corresponding units of a conventional machine. Rather, it is necessary to choose specific computations for making a comparison.

The configuration of machine shown with 1024 processing elements and 128 array memory units is chosen to meet the computation and memory requirements of the Navier-Stokes benchmark computation, that is, 1000 MFLOPS computation rate and 40 megawords of array memory. The manner in which this configuration was derived is explained below.

Before this sort of machine can be constructed with confidence that it will meet

practical objectives, more experience and knowledge is needed in the areas of program transformation, optimum code generation, and the design of hardware units to meet performance and reliability requirements. Studies in these areas have high priority in the current work of the MIT Computation Structures Group [6, 8, 9].

Program Structure

The overall structure of the main loop of the VAL program for the Navier-Stokes problem is shown in Figure 3 and corresponds to the subprogram STEP in the Fortran version. The functions of the thirteen modules are as follows:

BC	Extends the solution to match boundary conditions of the problem.
MuTur	Calculation of residuals for turbulent flow.
RHS	Calculation of the "right hand side" values for the factored linear systems.
VisRHS	Calculation of "right hand side" for viscous flow.
Smooth	Spatial smoothing to stabilize the computation.
XXM, YYM, ZZM	Routines to calculate the "metrics" – differences that express the compression or expansion of space in the computational grid.
D	Difference computation for the coefficient matrices.
ABC	Calculation of coefficient matrices for one linear system .
F	Obtains right hand side values for one linear system from the S structure.
BTri	Solver for the block tridiagonal equation systems.
VisMat	Calculation of coefficients for viscous flow.

From the VAL program we have obtained the operation counts shown in Figure 4. The totals show that, as usual, adds and multiplies are in the majority, and roughly equal in number. Divisions are relatively rare. Reads and writes of the array memories are in the ratio of three to one and are less than one eighth as numerous as adds and multiplies. The proposed machine in Figure 2 has been chosen to support these operation counts. We assume that each cell block can recognize enabled instructions and deliver operation

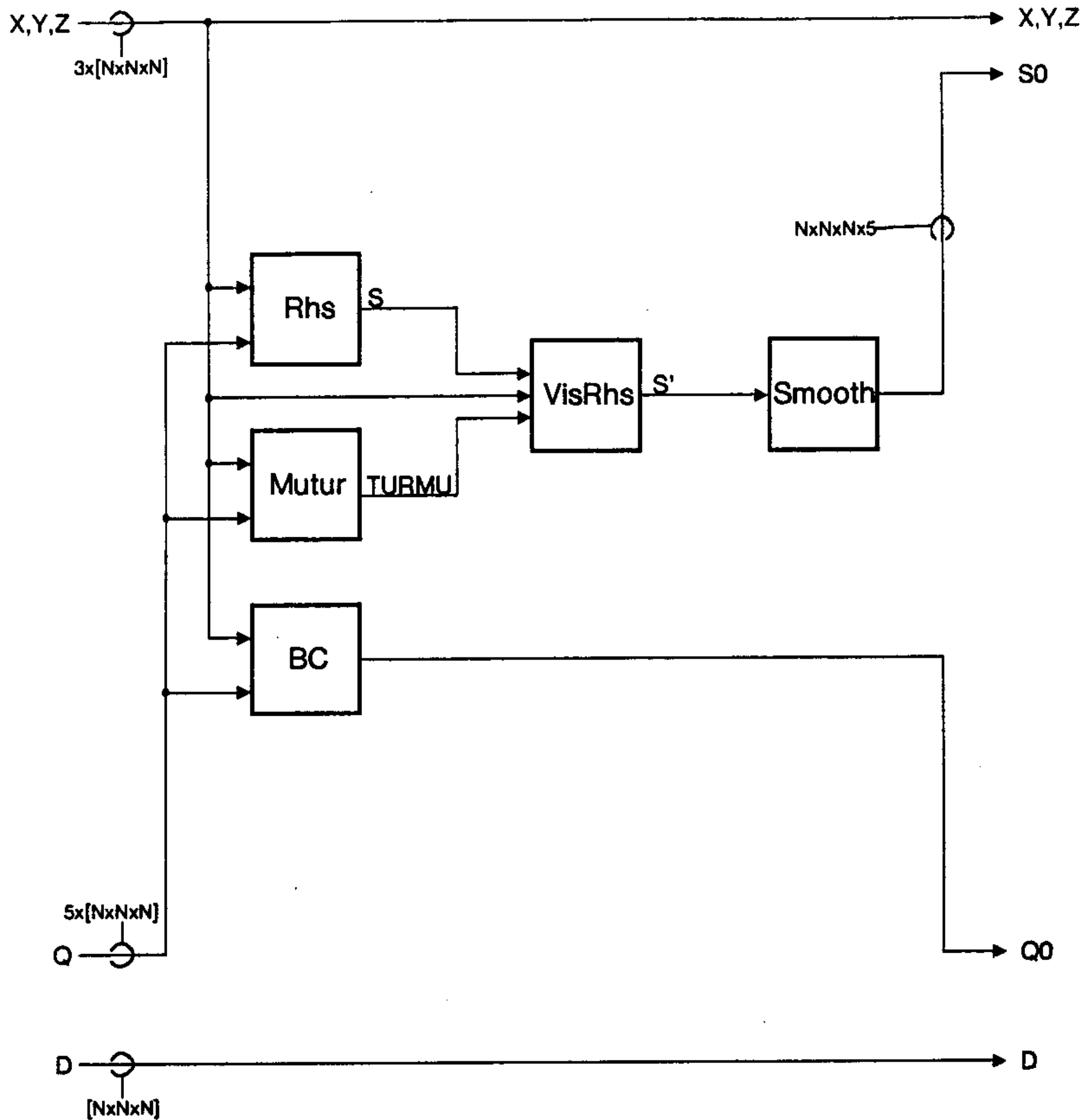


Figure 3a. Computation for one time step of the Navier-Stokes computation.

packets to the functional unit network at the rate of one million operation packets per second. These operation packets will call for arithmetic operations to be performed by functional units and read and write operations for the array memories. From the operation counts we see that the 1024 processing elements can carry out the Navier-Stokes computation at the rate of one time step per two seconds. This rate of computation involves handling the following rates of packet flow:

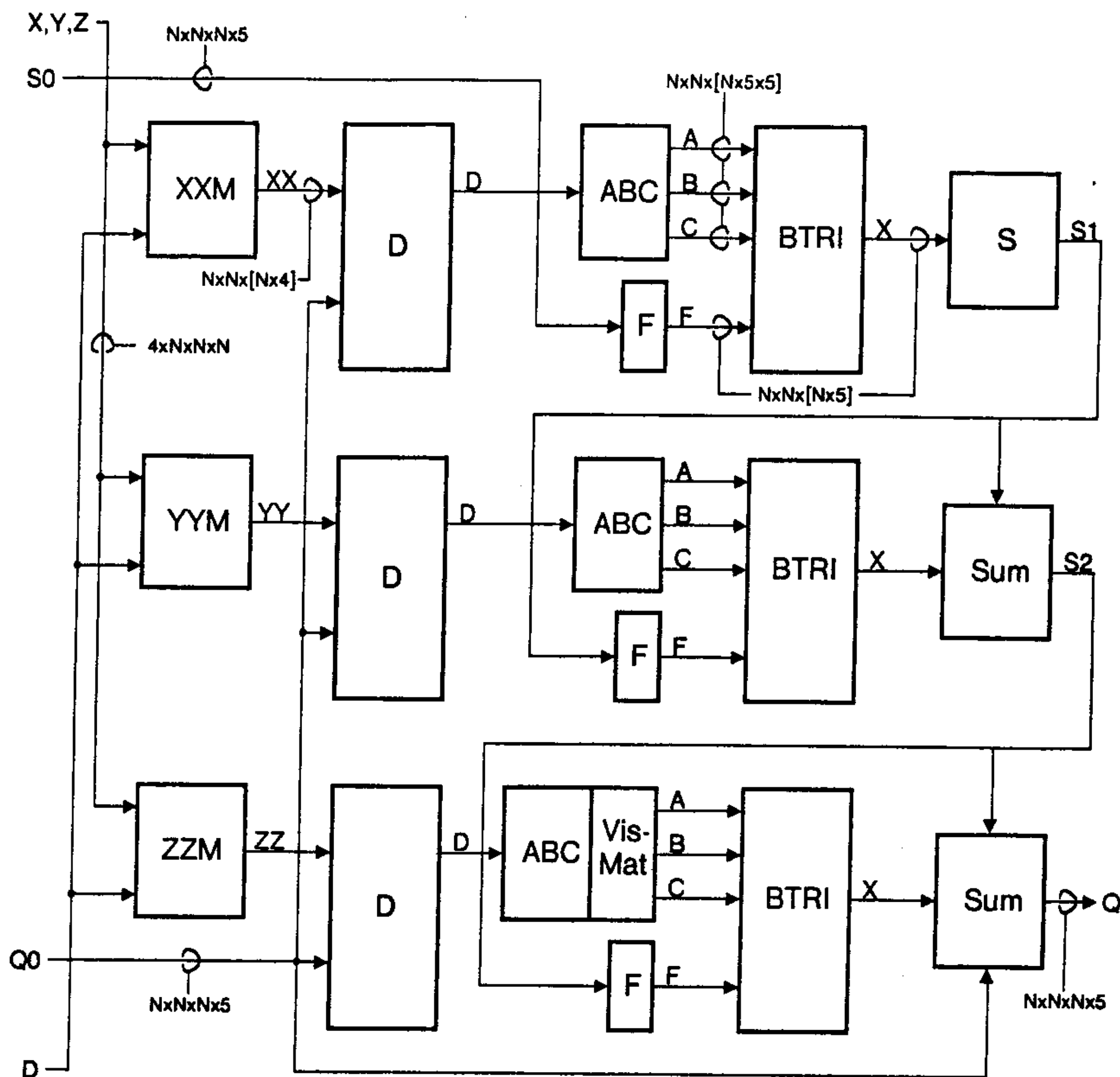


Figure 3b. Computation for one time step of the Navier-Stokes computation (cont.).

Operation packets		
	additions	353 MHz
	multiplications	476 MHz
	divisions	15 MHz
Array Memory Packets		
	writes	70 MHz
	reads	25 MHz
Total		939 MHz

The routing networks are assumed to be designed to support the rates of packet flow

Module	Add	Multiply	Divide	Read	Write
XXM	$60\ N^3$	$36\ N^3$		$27\ N^3$	$9\ N^3$
D	$66\ N^3$	$216\ N^3$	$6\ N^3$	$45\ N^3$	
ABC	$30\ N^3$	$9\ N^3$	$3\ N^3$		
F				$15\ N^3$	
VisRHS	$O(N^2)$	$O(N^2)$	$O(N^2)$		
BTri	$474\ N^3$	$600\ N^3$	$15\ N^3$		
Sum	$15\ N^3$			$15\ N^3$	$15\ N^3$
BC	$O(N^2)$	$O(N^2)$	$O(N^2)$	$10\ N^3$	$5\ N^3$
RHS	$O(N^2)$	$O(N^2)$	$O(N^2)$	$8\ N^3$	$5\ N^3$
MuTur	$O(N^2)$	$O(N^2)$	$O(N^2)$		
VisMat	$O(N^2)$	$O(N^2)$	$O(N^2)$		
Smooth	$60\ N^3$	$90\ N^3$	$5\ N^3$	$20\ N^3$	$15\ N^3$
Total	$705\ N^3$	$951\ N^3$	$29\ N^3$	$140\ N^3$	$49\ N^3$

Figure 4. Operation counts for the Navier-Stokes computation. The grid size N is 100 for the application studied.

required by this computation. These rates are feasible provided instructions for each phase of the computation are distributed evenly over the processing elements and that phases that run together do not depart seriously from the average demand on functional units and array memories.

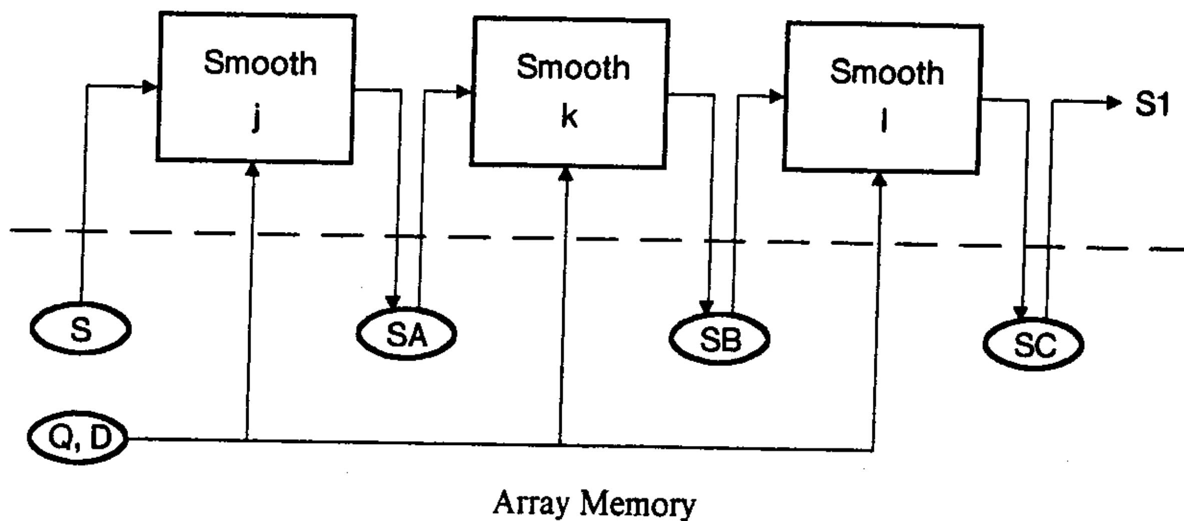
Machine Code Structure

In our concept of a data flow supercomputer, an instruction cannot be executed again until its results have been consumed by successor instructions. The compiler from the source programming language must ensure that this condition is met.

Since high performance can be achieved only if instructions are executed over and over, it is attractive to structure the machine code so that streams of data can flow through successive stages of instructions in pipeline fashion, each instruction operating on new data as soon as its previous result has been consumed. Pipelining of data flow machine code is done by arranging that each instruction, after it executes, send a packet conveying an

acknowledge signal to each instruction from which it received an operand value. The details of this kind of code structure are given in [5].

To illustrate how this idea is applied in the case of the Navier-Stokes problem, we consider the module Smooth, which involves about ten percent of the arithmetic in the time step computation. A fourth-order smoothing formula is applied to each of the five physical state quantities in each of the three grid dimensions. The input to this process is the structure Q of physical state quantities, the residuals S, and the Jacobian of the grid D.



Array Memory

Figure 5. The smoothing process is carried out in three phases – one for each dimension of the grid.

As shown in Figure 5, the computation proceeds in three phases in which the smoothing formula is applied in corresponding directions of the grid – j, k, l. Each phase reads its input values from the array memory and writes its results, a new version of S, as a new data structure in the array memory. As values of S are read from array memory, the space occupied becomes free and is available for the construction of new array values, perhaps the elements of SA generated by Smooth (j).

Figure 6 shows the machine code for smoothing one variable with respect to one dimension of the grid. This code corresponds to the VAL code shown in Figure 1, which treats only one component of data from Q and S. The instruction format and principles of machine program organization used here are explained in [5]. The elements of S, Q and D

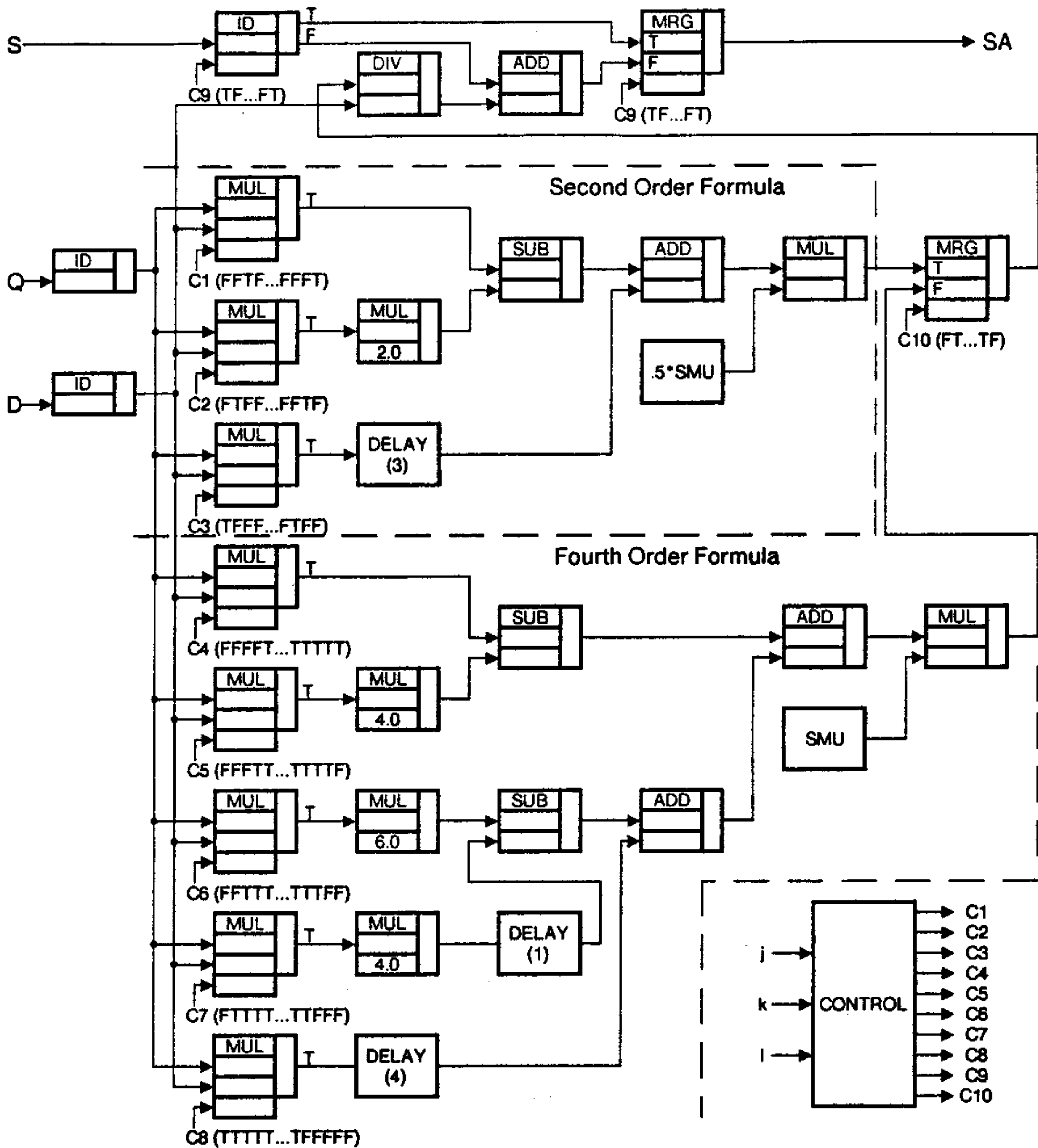


Figure 6. Illustrative data flow machine code for the Smooth module.

enter as successive operand values delivered to three identity (ID) instructions at the left of the diagram. (The machine code for retrieving and storing values in the array memory is not included.) The control section uses the indices j, k, and l of the elements to generate streams of boolean values C1, . . . , C10 which gate values from the data streams as required

to implement the smoothing formula. The actual control values shown at the multiply instructions correspond to j-values 1, . . . , N for those groups of elements where indices k and l are not on the boundary. The fourth order formula is applied for elements of SA that are two positions distant from the j-boundary and at least one position distant from a k- or l- boundary. An auxiliary second order formula is used for elements one position away from the j-boundary. For each element on a boundary, the value from S is used without change.

In this illustration of pipelined data flow machine code we have assumed for simplicity that data is delivered to the smoothing process as three streams of data – each one million elements long. In reality, this would not provide enough simultaneous work to keep the machine busy and would take much too long. It is necessary to duplicate the code shown 100 times and arrange that each copy processes one slice of data from the three-dimensional grid. Since this is still not fast enough, further concurrency must be exploited by processing several planes of data simultaneously. The code to accomplish this takes up proportionately more memory in the processing elements, but poses no new problems of machine code structure. The interesting challenge is the design of compilation techniques that can generate such code automatically.

In the Navier-Stokes computation a large fraction of the work is in computing solutions to the 30,000 linear systems for each time step. This is done by module BTri, using coefficient matrices and right-hand-side values computed by modules XXM, YYM, ZZM, D, ABC, F, and VisMat. In our proposed machine code structure, these eight program modules are operated as a single pipelined computation, receiving streams of elements of data structures Q0, S0, X, Y and Z from array memory. The computation proceeds in three phases corresponding to the three grid dimensions. Each phase uses results from the previous phase and builds a new data structure in the array memory: S1, S2, and Q, the state structure for the next time step.

The implementation of BTri is discussed in the next section of this report. Module Smooth has already been treated. The remaining modules, BC, RHS, MuTur, and VisRHS, account for only a small portion of the total computation because they do not

touch all points of the grid. These have not been analyzed in detail. We assume they would operate by processing streams of data retrieved from the array memories in pipeline fashion, to produce streams that define new output data structures built up in the array memories. The memory required for FIFO buffers to build balanced pipeline code for these modules is expected to be small. The buffers would be implemented within the processing elements. The structure of these modules and of the corresponding machine code is similar to code blocks of the global weather code, and no special problems are anticipated.

Performance

The problem of achieving high performance on a data flow supercomputer is essentially the problem of keeping all units of the machine busy doing productive activity. From the flow diagram we see that the computation for one time step progresses through phases, each phase involving different code and different data. To ensure full utilization of the machine, the instructions making up the code of each phase must be distributed evenly over all processing elements, and each major data structure must be distributed over all array memory units. In the following, we assume that the machine and its compiler are designed so that this balance is achieved. The implications of this hypothesis on the instruction set, memory organization, and design of the compiler are subjects of current study.

As discussed earlier, we have chosen the proposed machine configuration to provide sufficient memory bandwidth and processing capacity to achieve the rates required by the Navier-Stokes problem. This, however, does not guarantee that performance at those limits will be achieved. We must be certain that data dependencies in the machine code do not prevent sufficient instructions from being enabled to maintain full utilization of the functional units. We will examine this question now for the critical portion of the code – the linear system coefficient generation and solution.

The question is whether the required rate through the data flow pipeline can be maintained. Let us see what the rate must be to achieve two seconds per time step. The pipeline headway is the time available for the whole computation divided by the number of

data groups that must be fed through the pipeline:

$$(2 \times 10^6) / (3 \times 100 \times 100) = 67 \text{ microseconds}$$

The headway in practice must be shorter because we have neglected pipeline start up and shut down time (see the next section), and time must be allocated for computation by the remaining modules used in each time step. These may add about fifteen percent to the total computation time, so a pipeline headway of 50 microseconds should be sufficient to achieve full performance. To meet this requirement the instruction execution period of the machine must be less than half the pipeline headway – less than 25 microseconds. We believe this is a reasonable goal for the hardware configuration proposed.

Solving Block Tridiagonal Systems

A tridiagonal matrix is a square matrix having nonzero elements only on its principal diagonal and the two adjacent diagonals. A system of equations having a matrix of coefficients in tridiagonal form can be solved by specialized algorithms that use only order n steps, where n measures the problem size in terms of the number of rows of coefficient blocks of fixed size. The usual approach is to perform a forward substitution pass proceeding from upper left to lower right. The solution is completed by a back substitution pass that proceeds in the reverse direction – lower right to upper left.

The linear equation systems of the Navier-Stokes problem have block-tridiagonal coefficient matrices (Figure 7), which means they are tridiagonal with 5 by 5 blocks in place of scalar elements. Some algorithms for tridiagonal systems can be extended to block tridiagonal systems by substituting matrix operations for scalar ones, including replacing reciprocal by matrix inversion.

The diagram in Figure 8a shows the pattern of data flow for the linear reduction algorithm. Each step in the forward solution generates a set of intermediate results required at the corresponding point of the back substitution. The steps of the forward solution must proceed sequentially since each step requires values calculated by the immediately preceding step; the back substitution is sequential in the same way.

$$\begin{bmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & A_{n-1} & B_{n-1} & C_{n-1} \\ & & & A_n & B_n \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{n-1} \\ X_n \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_{n-1} \\ F_n \end{bmatrix}$$

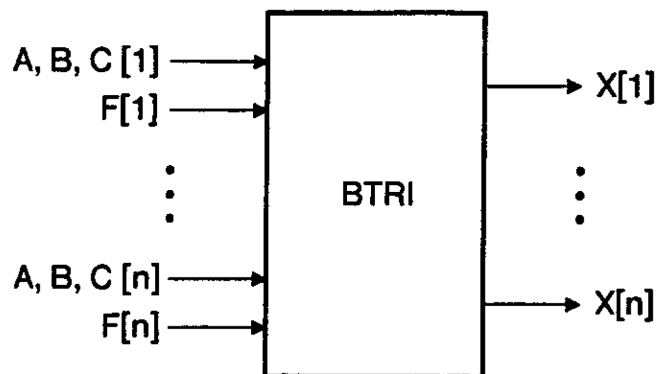
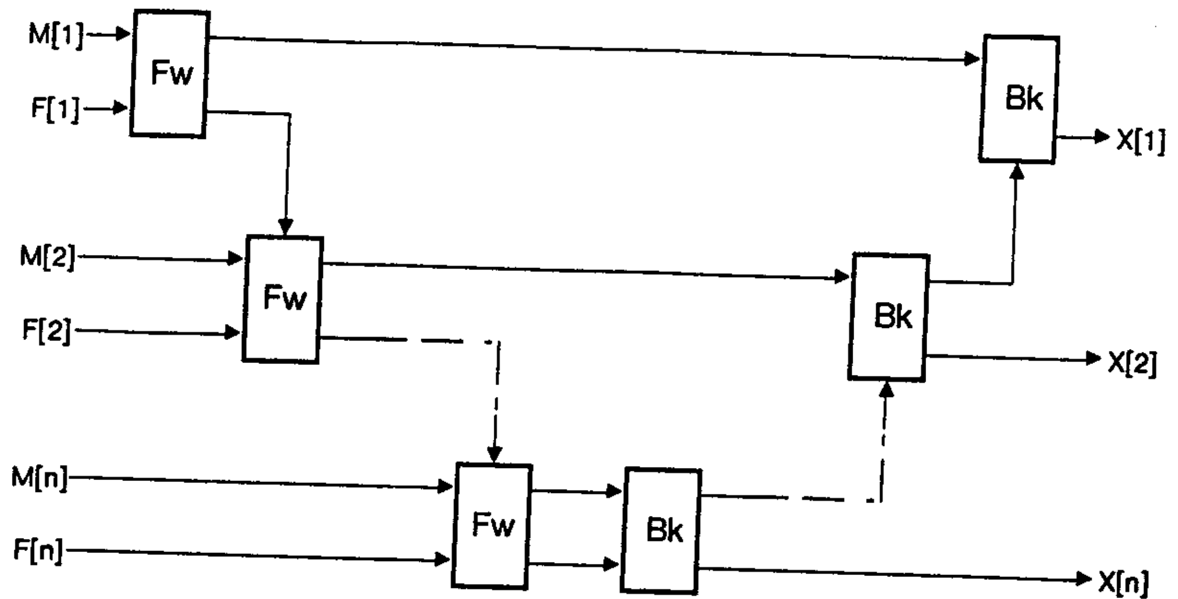


Figure 7. Block tridiagonal equation system and the input/output interface of module BTri.

The computation shown in the figure can be operated in pipelined fashion on a data flow computer to realize a high degree of concurrency; the ten thousand sets of coefficients and right-hand-side data can be fed into the pipe, one at a time, at the repetition rate of the pipeline. A major difficulty in doing this is the amount of storage required to hold the intermediate data waiting for use in the back substitution. For the problem under study, in which each system has 100 rows of 5 by 5 blocks, each step in the forward solution has a delay of 31 instruction periods, and generates 30 intermediate results. Each step in the back substitution has a delay of 19 periods. From this it follows that 75 megawords of storage is required for intermediate values to run a full pipeline.

An alternative approach is to use a cyclic reduction algorithm which will solve tridiagonal systems, also in order n steps, but in order log n time instead of order n time. Cyclic reduction offers more parallelism, but that is not the important issue here—there is enough parallelism in the simultaneous solution of 10,000 systems. As shown in Figure 8b,

(a) Linear reduction.



(b) Cyclic reduction.

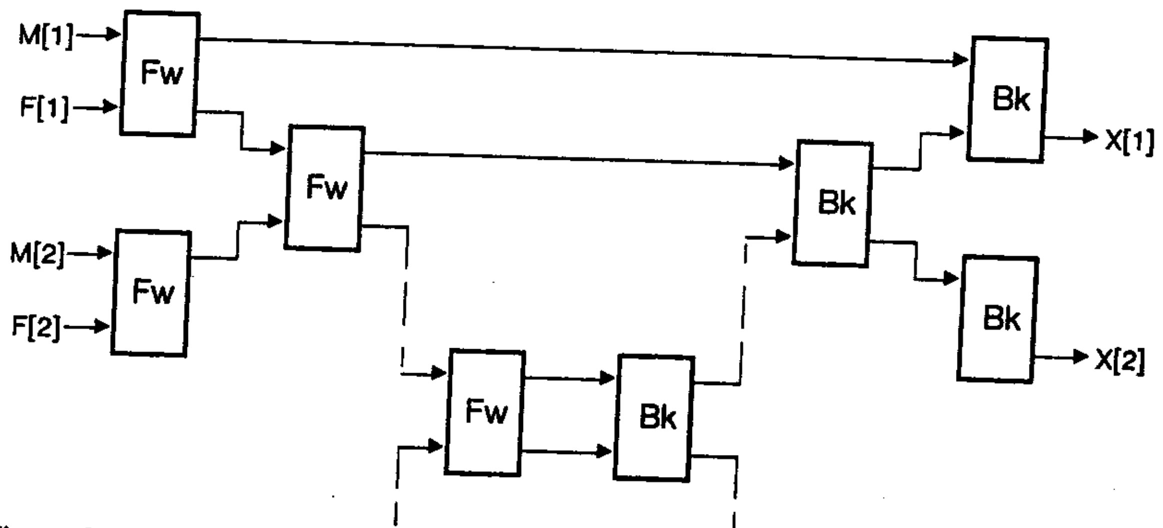


Figure 8. Flow diagram for two pipelined schemes for solving tridiagonal systems.

use of cyclic reduction reduces the length of the pipeline to 350 stages from 5000, and the storage required for intermediate values is only 1.4 megawords.

Conclusion

We have outlined the general principles that could be used to structure machine level programs for running the three-dimensional Navier-Stokes problem on a data flow

supercomputer. Our analysis gives reasonable expectation that a data flow computer having 1024 processing elements and 128 array memory units could carry out the computations at the desired 1000 MFLOPS rate. As presently envisioned, such a computer could be built using roughly 25,000 LSI parts in NMOS technology. Such a computer would be considerably smaller in size and power consumption than conventional supercomputers.

Nevertheless, our analysis of the Navier-Stokes problem is incomplete in several ways. The principal inadequacies stem from the fact that we have not chosen a specific instruction set for a practical high performance data flow computer. Consequently, wherever our estimates depend on instruction formats, inaccuracies may exist.

Another question concerns the coding of the "red tape" computations that compute array indices, control the routing of data values according to conditional outcomes, and so on. Our experience from study of the weather code and the Navier-Stokes code indicates that such instructions will form a minor portion of machine level programs, yet if these parts of the code are not carefully structured, the pipelined structures will not be able to operate at full speed.

The amount of array memory required to hold the problem data base is easily calculated; however, we have not done a careful count of the amount of FIFO buffer storage required to pipeline all parts of the computation. For some applications such buffering may use the array memory units. In the case of the Navier-Stokes code we expect that most FIFO buffering will be implemented in the instruction memories of the processing elements.

Other areas where our analysis is incomplete are in input/output and fault tolerance. We have only considered the main time step computation, omitting any support for problem specification or user analysis of results. Likewise, provisions for problem backup and fault tolerance have not been considered.

The primary work needed to establish viability of high performance data flow computers is demonstrating that a compiler can be constructed that generates the

hypothesized machine code structures. This needs development of appropriate theory and algorithms for transforming, optimizing, and generating machine code from programs expressed in VAL.

References

1. Boughton, G. A. Routing Networks in Packet Communication Architectures. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June, 1978.

2. Dennis, J. B. Packet Communication Architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975, pp. 224-229.

3. Dennis, J. B. Data Flow Supercomputers. *Computer 13*, 11 (November 1980), 48-56.

4. Dennis, J. B., Boughton, G. A., and Leung, C. K. C. Building Blocks for Data Flow Prototypes. Proceedings of the 7th Annual Symposium on Computer Architecture, May, 1980, pp. 1 - 8.

5. Dennis, Jack B., Gao Guang-Rong, and Kenneth Todd. A Data Flow Supercomputer. submitted for publication

6. Gao, G. R. An Implementation Scheme of Array Operations in Static Data Flow Machine. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June, 1982.

7. Gurd, J., and Watson, I. Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution. *Computer Design 19*, 6 (June 1980), 91-100.

8. Todd, K. W. High Level VAL Constructs in a Static Dataflow Machine. Tech. Rep. TR-240, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1981.

9. Todd, K. W. Function Sharing in a Static Data Flow Machine. Proceedings of the 1982 International Conference on Parallel Processing, August, 1982.