Exploiting Data Parallelism in Signal Processing on a Data Flow Machine

Peter Nitezki

Forschungszentrum Informatik
Technische Expertensysteme und Robotik
Haid-und-Neu-Str. 10-14
Karlsruhe 1
West Germany
Prof. U. Rembold
NITEZKI%FIX@IRA.UKA.DE

ABSTRACT

This paper will show that the massive data parallelism inherent to most signal processing tasks may be easily mapped onto the parallel structure of a data flow machine. A special system called STRUCTFLOW has been designed to optimize the static data flow model for hardware efficiency and low latency. The same abstractions from the general purpose data flow model that lead to a quasi systolic operation of the processing elements make explicit flow control of the data tokens as they pass through the arcs of the flow graph obsolete. We will describe the architecture of the system and discuss the restrictions on the structure of the flow graphs.

1 INTRODUCTION

Nearly all signal processing tasks show an algorithmic structure of extreme regularity both in the computing structure and the access pattern of their data. This inspired a variety of architectures that exploit this parallelity to provide the necessary computing power for real time operation. Some examples are processor arrays, systolic arrays and hard wired algorithms. Unfortunately their flexibility and programmability is quite low and the burden to parallelize an algorithm is completely put on the programmer.

Data flow computers that automatically exploit the fine grain parallelism of a program seemed to be very attractive to avoid these restrictions. In the last ten years several data flow systems have been proposed for signal processing applications /CHEN, HOGE, SAWA, SEAL, TEMM, VEDD/. However, pure data flow mechanisms turned out to be quite inefficient in processing large data structures and have a tendency to large overheads in data storage and flow control. Dynamic data flow systems need to enumerate any atomic data element in order to provide correct pairing of the operands. This makes a huge name space necessary which is only sparsely occupied. Static data flow schemes do not provide easy mechanisms for recursion and procedure calls and are therefore not much used. Most of the designs tried to overcome the problems by either restricting the parallelism of their machines to relatively low rates /HOGE, TEMM/ or by using data flow only for high level runtime scheduling of conventional computing resources /HONG, SEAL/. Another approach abandons some of the principles of pure data flow to gain efficiency and parallelism /LEIE, CHAS/. This is a step back to user controlled parallelism and brings back the problem of efficient programming of the individual tasks.

Some recent designs use the static principle together with static scheduling of the graphs. The massive use of parallelism is guarantied by compilers, which make efficient use of the parallelism in the application and map it automatically onto the hardware /VEDD, HONG, CAMP/.

2 THE STRUCTFLOW APPROACH

The design of STRUCTFLOW is guided by the fact that most signal processing algorithms show a very regular structure with nearly no conditional expressions. Furthermore the computational structure of the algorithm is relatively small containing typically less than about twenty nodes. The parallelity in the data structure is far more important. A 3x3 convolution on an image has for instance a computational structure of 16 nodes that has to be applied to 512x512 images. Therefore STRUCTFLOW has been adapted to the processing of streams, a concept first used in data flow machines by Dennis /DEN1/ and nowadays included in many modern single assignment languages /SISA/.

Streams can be looked at as one-dimensional data structures mapped onto an ordered sequence in time. Signals therefore may be directly be modeled as streams. By introducing separator symbols to structure such sequences, arbitrary data structures can be represented by streams. STRUCTFLOW provides means to handle both infinite and structured streams and exploits their inherent parallelity by dividing single streams into multiple streams and mapping those multiple streams onto multiple identical processing elements (PE) that execute a copy of the same graph. That kind of parallel data structures results in an order both in time and space. These multiple sequences of data elements may be automatically mapped onto the hardware structure. As signal processing is a task that does not require frequent activation and deactivation of a multitude of activities, the mapping may be statically scheduled and determined offline. The generation of the multiple streams from input signals and the synchronisation with the outer world is done by separate I/O-Modules. The handling of data items within the systems is purely asynchronous. Intermediate storage of data structures for restructuring and storage of large amount of input data is provided by structure memory (SM) modules (Fig. 1).

Tasks like image sequence analysis and algorithms with runtime dependent access patterns require intermediate storage. The individual modules are connected through a global network providing stream routing. The term stream routing refers to the property that individual streams traveling on the network maintain their sequence in time although then may share a particular physical link with other streams. STRUCTFLOW is designed to use a packet switching cube connected cycles network due to its fixed degree and logarithmic growth. The routing elements implement a static deterministic routing algorithm that preserves the integrity of streams and enables the scheduler to determine throughput offline.

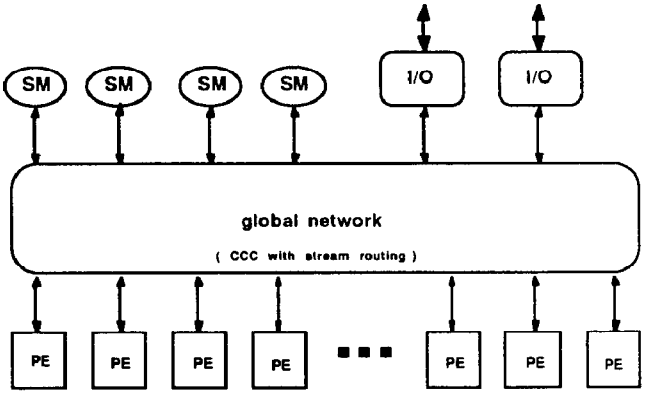

Fig. 1: The global structure of STRUCTFLOW

2.1 ABSTRACTIONS FROM PURE DATA FLOW

STRUCTFLOW uses the special features of streams to ease the design of the PEs. By restricting the data flow principle a very efficient hardware utilization can be achieved.

First if the sequence of data elements in time and space is strictly kept the elements of a stream don't have to be enumerated. The global network and the PE's are designed to keep the sequence in time in all cases. That means a token that enters a module first will be processed first and its resulting token will also exit first. Therefore only the name of the stream has to be attached to a data element, so the position of an element in a data structure is not explicitly accessible except in storage modules where structures can be arbitrarily addressed. Storage modules contain simple address generators to generate frequently used access patterns to form the

streams. All other patterns have to be generated in PEs to form a stream of addresses for the SMs.

Second, all data flow graphs will be processed serially in a pseudo systolic mode of data flow. All operands on the inbound arcs of the graph will be input in one wave, so the first data token of the second wave will only enter the processor if the first wave has been completely consumed. This quasi systolic operation allows the elimination of flow control within the PEs. As a consequence only the links across the network need to store an acknowledge arc that allow to implement a flow control protocol. The wavefront operation within the PEs avoids token clash without any special acknowledgement of the token consumption.

These two features ease the design of the PEs as we will see later and they have a lot of potential to reduce latency in the processing of the streams. There are of course some restrictions on the structure of the graphs and on the programming of conditionals and iterations, but most of them can be handled by a compiler, and there is not much use of general recursion in signal processing applications anyhow.

3 The Architecture of a STRUCTFLOW Processing Element

The PEs in STRUCTFLOW consist of three units (Fig. 2). Two of them, the matching unit (MU) and the processing unit (PU) form the classical data flow ring. The communication unit (CU) handles the flow control of the inbound and outbound arcs and forms the wavefronts for the quasi systolic processing of the graphs. Online loading of programs is also handled by this Unit.

3.1 STRUCTFLOW Instruction Set and its Implementation

For the ease of implementation the current version of STRUCTFLOW PEs use 16 bit integer arithmetic. The integer operators implemented today are addition, subtraction, multiplication, division, modulo function, supremum, infimum, and all relational operators on the ordered set of integers. Furthermore all 16 functions on two boolean operands are implemented.

Data items are endowed with a 3 bit type tag. It distinguishes integers and Booleans and allows the coding of special tokens for stream structuring and arithmetic exceptions. For the handling of arithmetic exceptions the set of integers is augmented by a *positive* and a *negative infinity value* which denotes overflow in the respective direction. All operators are functionally extended to allow arithmetic operations on these infinity values. Type mismatches or undefined value combinations like multiplying infinity by zero result a *bottom* -token. These extensions allow exception handling without any context sensitive elements.

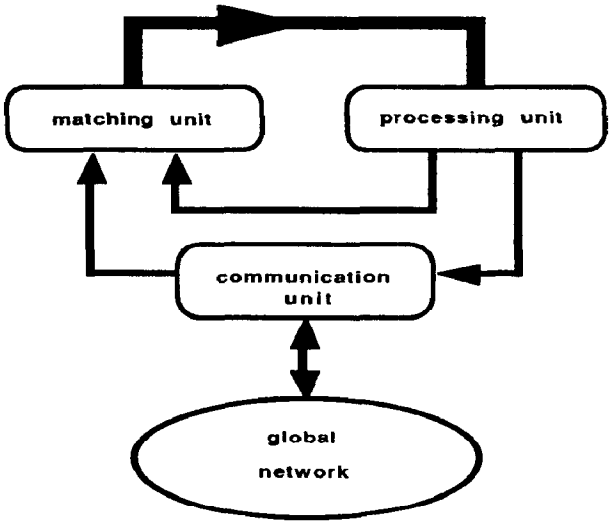

Fig. 2: Overview of the Processing Element

Iteration and conditional constructs are supported by special control operators. The **watch**-operator selects a data element of its input stream by the control of a boolean input. In case of a *True* the data input is copied to the output, in case of a *False* the result is a *bottom* -token. The **select**-operator chooses among its input operands that one, that is non-*bottom*. In case of two *bottom*-inputs *bottom* is the result. In case of two non-*bottom* values one is chosen arbitrarily. These operators have two non-stream-preserving counterparts, **guard** and **choice**, that suppress the output of a *bottom* token and therefore allow nondeterministic behaviour. Although those nondeterministic are important in the efficient programming of conditionals and iterations they are

not safe in any case. The correct propagation of wavefronts and the correctness of the streams must be ensured by the compiler.

Two instructions provide flow control and duplication of streams. The **identity**-operator may serve as pipeline buffer element or as a semaphore or gate mechanism using the regular matching mechanism. The **copy**-operator interprets its second operand as the destination name of the second outgoing arc and provides stream copying and switching. The same interpretation of the second operand shows the **switch**-operator that does not duplicate its left operand but sends it to the port specified by the right one. In case of a *bottom* token in the right operand the destination field specifies the default destination. Additional output operators may be used to partition graphs and sharing them among several PEs, or for transmission of a stream to the I/O or structure memories.

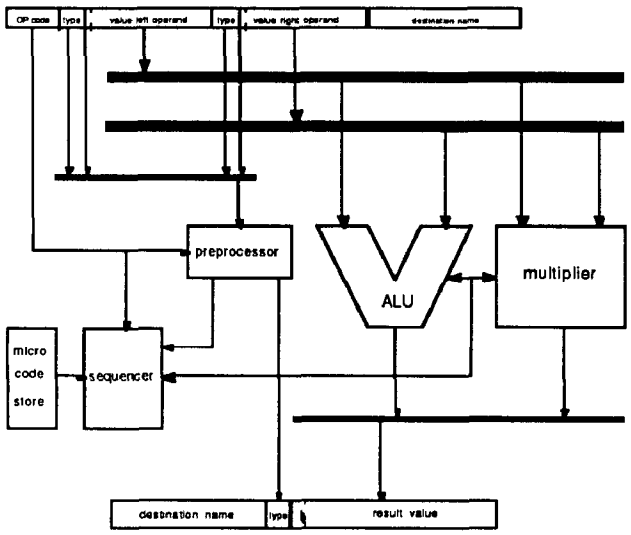

Fig. 3: The Processing Unit of a SRUCTFLOW PE

In the current first version of the STRUCTFLOW PU /AUGE/ (Fig. 3) all operators are processed by a microprogrammed ALU with an attached multiplier chip. Microprogramming has been chosen for the flexibility of the design in this phase of research and evaluation. The processing of the type tag together with the sign bit which also holds the boolean values is done by a function table preprocessor. This preprocessing needs no extra time as it is done in parallel with the ALU operation and speeds up microprograms as they do not have to examine the operands explicitly.

3.2 THE MATCHING OPERATION IN THE STRUCTFLOW PROCESSOR

Previous designs of data flow machines show matching as the major time consuming factor. Our intention was to keep matching as easy and fast as possible. The work of Dennis et al. /DEN2/ was the most encouraging influence. STRUCTFLOW uses static dataflow on nodes with exactly two inbound edges and exactly one destination. The **copy**-operator fits fully into this scheme as copying is done by a microprogram in the PU that interprets the second operand in a special way. Matching is strict so every operation will only be enabled when both operands are present. Modern memory technology makes direct storage of the data tokens feasible. This eliminates the need of an associative mechanism which needs extra time and hardware. Matching is achieved in a single read-modify-write cycle on the node store, where the node name in the data token directly addresses the memory.

Since all flow graphs are executed in a systolic pipelining of streams, no local flow control on the edges of the graph is needed. This frees the node or matching store of the need of backlink pointers for the acknowledge arcs and makes extra flow control steps obsolete. In addition to the effect of simplification of the matching outlined above, the speed of matching is at least doubled by this mode of operation. The matching unit /GALL/ therefore needs only a small finite state machine as matching controller. Only two transitions are needed per matching of one token: In the first step a token is selected from two input registers where the source is determined by the state of the token queue (Fig. 4). In the second step the node store is accessed and the three match control bits are examined. If there is no match, the token is stored. Otherwise, the token and the stored operand together with the instruction code are forwarded to the node formation unit. In this unit the different fields of the node are suitably arranged and sent to the PU as an instruction packet.

The matching control is driven by three bits of the node store. The first bit determines whether a token has yet been stored in this location in a prior operation. The second bit determines whether the present operand is a constant which is not to be deleted after a successful match. The third bit is set if the stored operand is on the left port of that node.

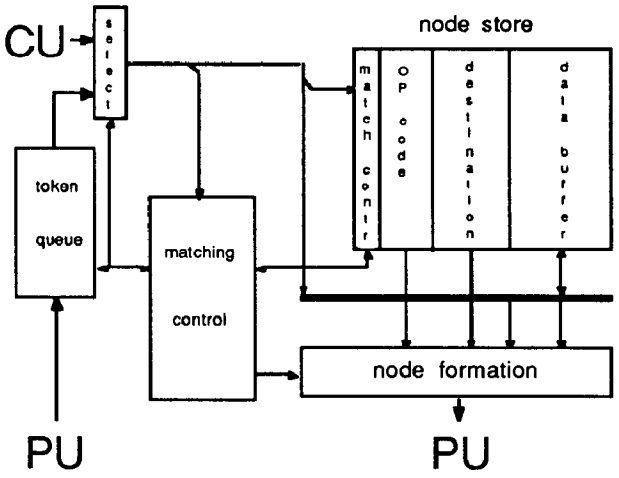

Fig. 4: The Matching Unit of a STRUCTFLOW PE

The quasi systolic wavefront processing frees the MU from flow control on the internal data path. The traffic of tokens on the input arcs however is influenced by the global network and the external timing of the I/O. STRUCTFLOW thus has to provide an acknowledge mechanism on the inbound edges of the flow graphs to prevent so-called token clash - two data items on the same arc at the same time. The interface of MU and CU is designed to work in lockstep and allows rejection of tokens to prevent that situation.

3.3 Flow Control and the Role of the Communication Unit

The CU performs mainly two tasks. First is the flow control of the in- and outbound arcs and the formation of the input wavefronts. Second is the online loading of program information in the node store of the MU and in its I/O control store.

Flow control is implemented through the input and output module /SCAR/ (Fig. 5). The input module maintains a hardware queue for any of the input arcs which form a fixed subset of the name space of the MU. The state of the queues controls an X-On/X-Off protocol on acknowledge arcs that are stored in the CU's control memory. That control store also contains information which arcs belong to a wavefront and therefore controls the information of input wave fronts. The input module scans that control store to form a sequence of input tokens for the MU. In case that every token is accepted the wavefronts of the active graphs are scanned sequentially. If a token is rejected or a queue is empty that wavefront is suspended and the next one will be processed. In the next scan the suspended wavefront will be resumed with the suspended arc quite like an interrupt in a von Neumann machine.

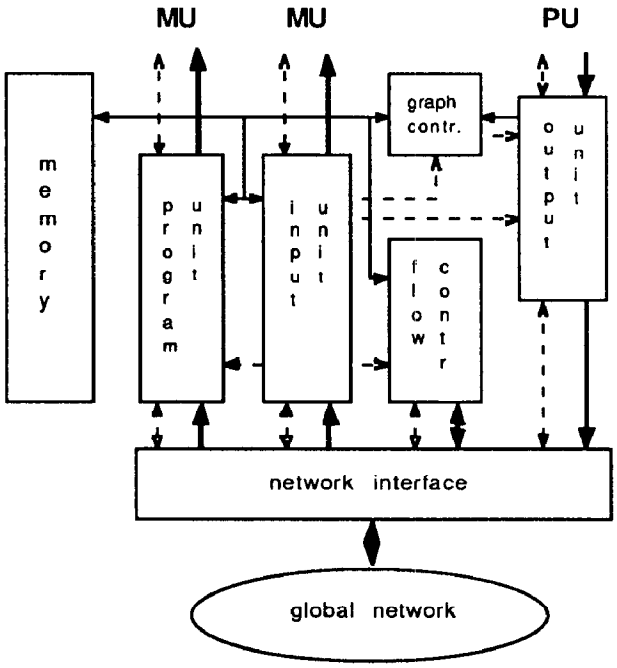

Fig. 5: The structure of the Communications Unit

The graph control module scans the streams for start or end tokens to enable or disable the wavefront formation and controls the use of particular graphs.

Program loading consists mainly of the task of scanning the network input for program tokens and storing of the information in the appropriate store. The program stream is therefore structured by special control tokens and has a particular structure to determine the different parts of

the program eg. node information and wavefront information.

4 PROGRAMMING ISSUES

Some of the design decisions of this system affect the programming of the machine. The most important issue is that all graphs are executed in a quasi systolic fashion. The consequence is that all possible paths that lead to a node have to be of equal length. The finite depth of the queues in the CU also places an upper bound on the depth of a graph as there is no means to stop wavefronts that entered the MU. In case of acyclic flow graphs these graphs have to be balanced with respect to the root and with an upper bound on their depth. All these restrictions may be met automatically at compile time by inserting identity operations forming a pipeline delay of appropriate length. In case of the commutativity and distributivity of certain operators the graphs may be automatically restructured to their maximally broad equivalents. These problems are all covered by a paper of Montz /MONT/ who investigated these issues for the MIT static data flow project.

4.1 THE PROGRAMMING OF CONDITIONALS

The programming of conditional expressions has to deal with the same restrictions as other acyclic graphs. The most straightforward solution is to evaluate both branches in parallel and to select the desired result at the end with a selector node (a merge in terms of Dennis who gave an example for this style in /DEN2/). This solution shows all desired properties like cleanliness (no tokens remain in the graph after execution of the conditional), safety (no deadlock may occur), and strictness (the result will only produced when all input tokens have been supplied) in a very straightforward manner, but wastes a lot of the necessary computing power as only one of the results will be used.

The particular properties of the STRUCTFLOW PE help to overcome this problem. Any token will be processed as soon as it enters the MU and neither the MU nor the PU does some reordering on the packets they act on. So, if two different subgraphs of equal depth are fed with two consecutive wavefronts, the graph that got the first wave will produce the first result. If now the condition is evaluated, a switch might distribute the wavefronts and a nonstrict merge may form the correct result stream. STRUCTFLOW does not provide nonstrict operators. This dilemma is a fake, as only one wavefront can arrive at a time and a knot /VEE1/ can be employed. The term knot refers to an extension of flow graphs to flow nets where two arcs of the net point at the some port of an operator. It is easy to show that in our case of conditional wavefront processing no token clash or violation of strictness and cleanliness can occur.

Let us look at the example depicted in Fig. 6. For comparison it is the same small conditional that can be found in the paper of Dennis /DEN2/.

```
if C[i] then (A[i] + B[i])
else 5 * (A[i] * B[i] + 2)
endif
```

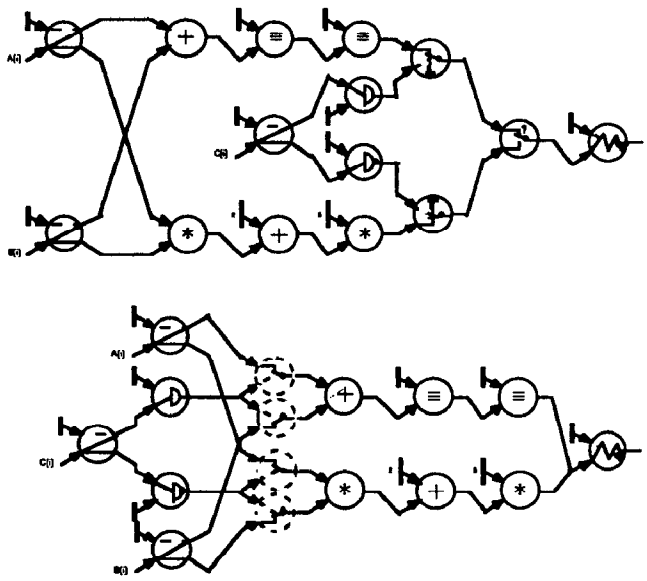

Fig. 6: Two structural variants of a conditional expression

In first graph we see a strictly deterministic implementation. The rightmost node provides the name of an external arc where the result will be used. The switches and logic gates to its left provide the conditional selection of the results of the two branches. The two copy nodes on the left distribute copies of the input to the two branches

(not depicted in the example of Dennis). In the upper branch the pipeline buffers can be found in the chaining of the two identity nodes. Please note that both branches execute wether the result will be used or discarded. The input arcs of the graph provide flow control so the arc named C needs no pipeline buffers.

The second graph shows the same conditional expression in an implementation with nondeterministic switches. Only in case that the branch will be chosen an input will be routed to it. The other branch will be inactive. Note the necessity for pipeline buffering to keep the wavefronts intact even in this case.

4.2 Iteration and Recursion

As all data structures are already mapped onto streams, the classical loop for scanning of data structures is obsolete. Iterations are only needed for recurrence relations, where the computation of a value requires previously computed ones. Dennis /DEN2/ gives a complete solution for that case of iteration. The only concern of the compiler or the programmer is to supply the number of initial values that matches the depth of the iteration cycle for that the wavefronts will be correctly formed and to control the switches that feedback the previous values. The problem with a 'do while' kind of iteration is that the nett consumption resp. production of tokens by the loop may vary and thus the wavefront processing gets out of step. As a consequence the input of the graph has to be gated to synchronize the loop. Input arcs are the only arcs that provide acknowledgement of tokens, so every gate has to be coded as an operation with an input arc for the wavefront element and a control arc that gets tokens from the termination test of the loop.

The example in Fig. 7 shows an implementation of a *forall* expression in VAL (also found in /DEN2/). Due to the stream processing no explicit control of iterations as in implicit languages occurs. The non-deterministic switches on the left provide an efficient construct to choose the elements of the input stream and saves the pipeline buffers necessary elseway. The control stream denoted through a sequence of boolean values can be formed quite easily and can be derived from the elements of the input stream. In this case a non-deterministic implementation of the conditional expression can be seen. The deterministic version would also be possible but much less effective.

In the implementation of the MU the problem of general recursion was also considered. Gallinat /GALL/ could show that linear recursion could be programmed without changes to the hardware or the set of operators. The token queue in the MU takes the role of buffer memory where the tokens of a software controlled stack will reside. Although this kind of recursion is not very efficient, because all the tokens in the stack will pass the MU and the PU many times as they all travel around the data flow ring, it still proved feasible even on a data flow machine designed for stream processing.

4.3 The Efficiency of STRUCTFLOW Parallelism

The basic mechanism of procedure call and recurrence execution in STRUCTFLOW is graph locking. Most designers /VEE2/ abandoned this kind of control in data flow machines because it reduces the available parallelism and thus causes idle times due to uncompletely filled pipelines. STRUCTFLOW however has only very few stages in the data flow ring, so 5 to 10 tokens are enough to keep both MU and PU busy. Only pathological graphs might show less usable parallelism. Due to the sequential execution of the individual flow graphs more available parallelism only causes more tokens to wait in the queue. All of the massively parallel execution in STRUCTFLOW comes from the consequent parallelization of streams.

5 Project State

The present project is to bee seen in a as an academic enterprise in a restricted university research environment. It has been performed by one assistant and an a handful of students working on their theses. Up to now one PE has been built and undergos test. The design of the CU has lead to a prototype to be redesigned for proper integration into a network of processors. In some parts of the design the lack of tailored ICs re-

stricted the size and performance of the PE. Some activities in the direction of custom VLSI have been started.

6 ACKNOWLEDGEMENT

This research has been done in the context of the research programme "Physics and Application of Novel Sensors" and has been funded by the state of Baden-Württemberg. Also many thanks to Karl Kleine who helped to improve my poor English.

7 REFERENCES

/AUGE/ Augenstein, R.; *Entwurf und Realisierung einer Verarbeitungseinheit*; Masters Thesis, Fak. Electrical Eng., Universität Karlsruhe, 1987(in German)

/CAMP/ Campbell, M. L.; *Static Allocation for a Data Flow Multiprocessor*; IEEE Int. Conf. Parallel Processing, p. 511ff, 1985

/CHAS/ Chase, M.; *A Pipelined Data Flow Architecture for Digital Signal Processing*, VLSI Signal Processing, Sect. 18, IEEE Press, 1984

/CHEN/ Chen, S.; Ritter, G.X.; *A Reconfigurable Architecture for Image Processing*; IEEE, Int. Conf. Computer Design, p. 516ff, 1984

/DEN1/ Dennis, J. B.; *Data Flow Ideas for Supercomputers*, IEEE COMPCON Spring, p. 15ff, 1984

/DEN2/ Dennis, J.B.; Rong, G.G.; *Maximum Pipelining of Array Operations on Static Data Flow Machine*, IEEE, Int. Conf. Parallel Processing, p. 331ff, 1983

/GALL/ Gallinat, J.; *Entwurf und Aufbau einer Matching-Einheit*, Masters Thesis, Fak. Computer Science, Universität Karlsruhe, 1987 (in German)

/HOGE/ Hogenauer, E.B.; Newbold, R.F.; Inn, Y.J.; *DDSP - A Data Flow Computer for Signal Processing*, IEEE Int. Conf Parallel Processing, p. 126ff, 1982

/HONG/ Hong, Y.-C.; Payne, T. H.; Ferguson, L. O.; *An Architecture for a Dataflow Multiprocessor*, IEEE Int. Conf. Parallel Processing, p. 349ff, 1986

/LEIE/ Leier, W.; *A Small, High-Speed Data Flow Processor*, IEEE Int. Conf. Parallel Processing, p. 341ff, 1983

/MONT/ Montz, L.B.; *Safety and Optimization Transformations for Data Flow Programs*; TR 240, LCS, MIT, 1980

/SAWA/ Sawakar, P.S.; Forquer, T.J.; Derry, R.P.; *Programmable Modular Signal Processor - A Data Flow Computer for Real Time Signal Processing*, IEEE Int. Conf. Parallel Processing, p. 344ff, 1985

/SCAR/ Scarchillo, A.; Entwurf und Aufbau einer Kommunikationseinheit; Masters-Thesis, Fak. Electrical Eng., Universität Karlsruhe, 1987 (in German)

/SEAL/ Seals, J.D.; Shively, R.R.; *EMSP: A Data Flow Computer for Signal Processing Applications*, VLSI Signal Processing, Sect. 6, IEEE Press, 1984

/SISA/ McGraw, J., et al.; *SISAL: Streams and Iteration in a Single-Assignment Language*; Lawrence Livermore Ntnl. Labs, 1983

/TEMM/ Temma, T., Mizoguchi, M.; Hanaki, S.; *Template-Controlled Image Processor TIP-1 Performance Evaluation*, IJCAI,1983

/VEDD/ Vedder, R.; Finn, D.; *The Huges Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing*, IEEE 12th Ann. Int. Symp. Computer Architecture, p. 324ff, 1985

/VEE1/ Veen, A.; *A Formal Model for Data Flow Programs with Token Coloring*; internal report, MC, Asterdam, 1981

/VEE2/ Veen, A.; *The Misconstrued Semicolon: Reconciling Imperative Languages and Dataflow Machines*; CWI Tract, MC, Amsterdam, 1986

```
A : array(integer) :=
 forall
        i in [0, m+1]     % range specification
        P : integer :=    % definition
                if (i = 0) | (i = m+1) then
                else 25 * (C[i-1] + 2 * C[i] + C[i+1])
                endif;
 construct B[i] * (P *P)  % accumulation
 endall
```

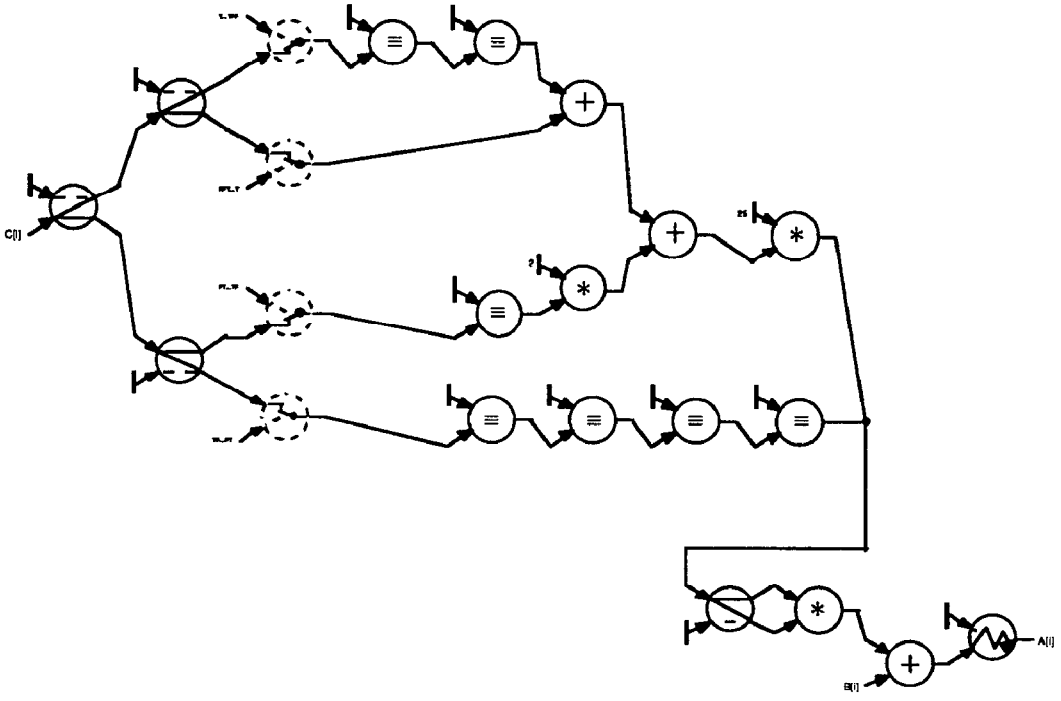

Fig. 7: Implementation of a forall construct