A DECENTRALISED COMPUTING SYSTEM BASED ON DATA-FLOW

G. K. Egan, Member, IEEE
Department of Computer Science
University of Manchester
Oxford Road
M13 9PL
England.

Abstract

Models underlying conventional computing systems are unsuitable for decentralised control. The Data-Flow model of computation is used as the basis for a decentralised computing system consisting of an arbitrarily large number of conventional microprocessors which may communicate over simple asynchronous links.

Summary

The beginnings of a coherent Decentralised Control Theory has been developed in an attempt to overcome problems of communications, computational intractibility and reliability arising from the concept of centrality at the core of Modern Control Theory [12]. This theory, coupled with inexpensive computational power in the form of microprocessors, may lead to computing systems with control computation distributed over hundreds of processing elements.

However, the models underlying conventional computing systems of the von Neumann type, like Modern Control Theory, involve centrality, are not very practical and have some positive disadvantages [3].

In this paper a decentralised computing system is described [5] which is based on a variant of the Data-Flow model of computation [8][2].

The following are the key features of the system:

1) It consists of an arbitrarily large number of conventional microprocessor-based processing-elements, which, depending on the application, may or may not be geographically distributed. The processing-elements need not be identical.
2) The processing-elements communicate over simple bit-serial or byte-parallel asynchronous links.
3) The system copes with bursts of data in an orderly manner and normal computations need not be interrupted.
4) Computations are described by directed-graphs and the processing-elements evaluate, or execute, the graph interpretively.
5) Graphical languages for the system may be easily tailored to specific applications.

Most of the system and it's process environments are being simulated on a large conventional system (MU5) [6]. A number of 'typical' processing-elements have been constructed and are currently being commissioned. These are connected to MU5 and represent the real part of the system.

Data-flow

The Data-flow model of computation was originally developed by Karp and Miller at IBM's Thomas J. Watson Research Centre [8] and subsequently expanded by Adams to include conditional evaluation [2].

Data-flow uses a finite directed-graph to describe a computation. The edges or arcs of the graph are queues of data directed from one node to another. The nodes represent functions which map input data onto output data.

Data flows down the arcs as packets or tokens, each node requiring a specific number of tokens to trigger the node-function's evaluation. The evaluation or firing consumes tokens from the input arcs and places result tokens on the output arcs. The number of nodes eligible for firing at any instant depends only on the availability of data.

The models of Karp and Miller, and Adams have the property of determinacy: it does not matter in what order eligible nodes fire or how long the node-functions take to be evaluated; the result of the computation is always the same. This makes Data-flow well suited to a system of loosely coupled processing elements.

CH1551-1/80/0000-0165$00.75 © 1980 IEEE

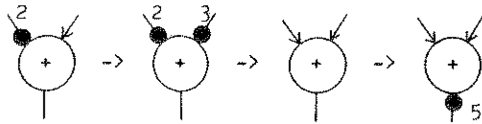


Fig. 1. Firing Example

A Control Example

Data-flow computations may be expressed in a graphical form which closely resembles the traditional block diagrams of control engineering. As with a block diagram, the overall data-flow graph may be decomposed into more manageable sub-graphs; this process is continued until some desired level of functional complexity is reached. Eventually the decomposition process will lead to sub-graphs which are the node-functions recognised by the computing system.

As an example, which may occur some way down the overall decomposition process, we take the case of a PID compensator [10] imbedded in a single-loop controller.

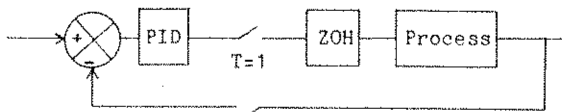


Fig. 2. Digital-PID based Controller

The PID block can be decomposed into sub-blocks for the proportional, integral and derivative components of the compensator.

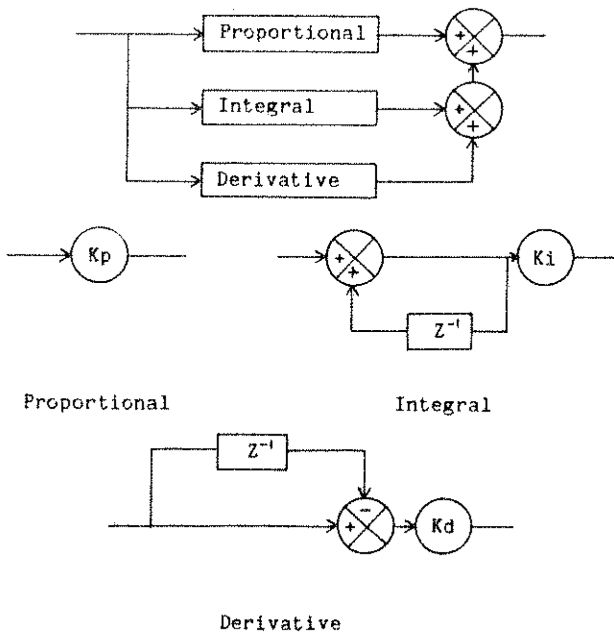


Fig. 3. Block Diagram Decomposition

Similarly the PID data-flow graph can be decomposed into sub-graphs.

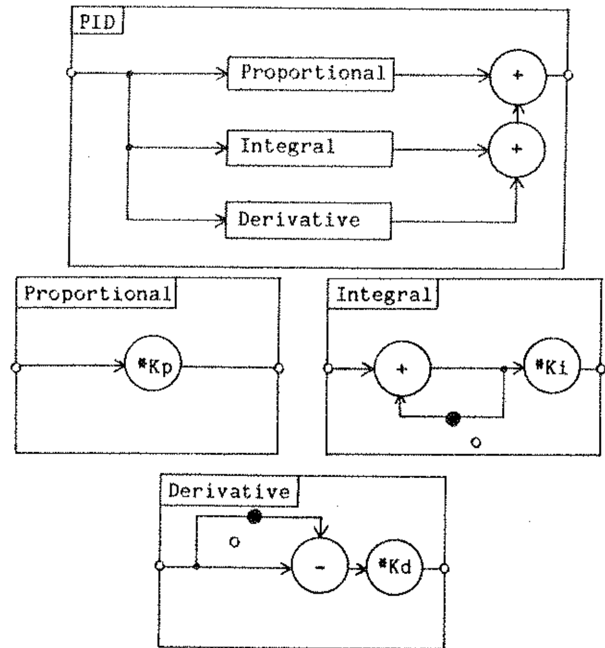


Fig. 4. Data-flow Decomposition

In this case, placing an initial token on an arc has the effect of delaying subsequent tokens on that arc by T. In general, initial tokens placed on arcs give the state of the graph at t=0 and as such represent an encapsulated history of the graph's evaluation. The assumption in this case is that the compensator input has been zero for all kT, k≤0 .

Communications in Data-flow

All the information needed to transmit a token from one node to another is contained in the data-flow graph. Graphs are held in the computing system as <u>node-descriptions</u>. The node-descriptions hold, along with other information, the graph connectivity in the form of <u>destinations</u>. Each destination specifies the <u>input-point</u> and <u>name</u> of a successor node to which result tokens should be sent.

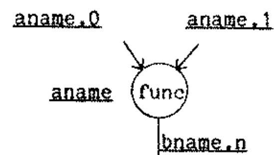


Fig. 5. Input-Points

In this system the graph is partitioned and the partitions allocated statically to processing-elements; the node name is sub-divided into the name of the processing-element and the name of the node within that element. The partitions will to a large degree follow the natural structure of the control task.

Node-Function Complexity

Provided the data-flow interface between nodes is maintained, the level of node-function complexity is completely at the discretion of the system designer. Nodes with primitive arithmetic and control functions give flexibility, but bring with them the penalty of high token traffic in the system's communication structure. Flexibility can be retained and communication traffic reduced by implementing common sub-graphs as node-functions. Kinematic models of adaptive industrial manipulators require the evaluation of Sine and Cosine frequently in co-ordinate transformations [4]; in this application it may be better to implement Sine and Cosine as node-functions.

In general any region of a data-flow graph may be implemented as a complex node-function (i.e. in hardware), the only requirement being that the data-flow interface is preserved on input and output arcs.

Tokens

The basic item in a data-flow system is the token. Tokens carry the data used by the graph in its evaluation and they may also carry a description of the data-flow graph itself.

Tokens are divided into six fields.

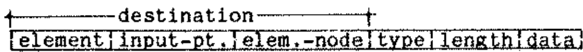


Fig. 6. Token Format

The fields and their functions are as follows:

1) element: the name of the destination processing-element.
2) input-point: the input point on the destination node.
3) element-node: the name of the destination node within the processing-element.
4) type: the type of the token data.
5) length: the length of the token data.
6) data: the data carried by the token.

At first inspection, the inclusion of type and length fields may seem extravagant, particularly if token size is to be minimised. However, if the data-field length is carried, then the token size may be variable and need no longer be constrained by the need to cater for the longest data type.

Some benefits which result from type and length fields are:

1) Average token size is reduced.
2) The node-function set is reduced. Type coercion is performed automatically where 'sensible'.
3) Tokens of one type cannot masquerade as tokens of a different type. The system checks all node-function operands for valid type.
4) Non-significant data-field bytes may be suppressed if necessary.
5) Complex objects such as node-descriptions may be sent through the system as tokens.

An example of an integer token is:

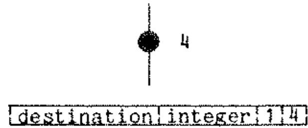
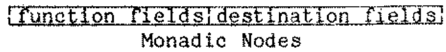


Fig. 7. Token Example

Nodes

Nodes carry their function-name, possibly literal operands and, in the form of destinations, the graph connectivity. Like tokens, they have a number of fields but the exact format varies with the node-function.



Monadic Nodes

Diadic Nodes

Literal Form

Fig. 8. General Node Formats

The function fields are as follows:

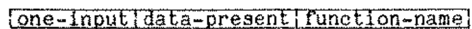


Fig. 9. Function Fields

A simple example of the literal node form, that of adding some constant literal to a data token, would be:

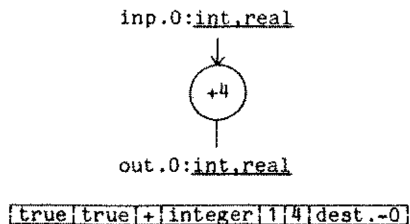


Fig. 10. Add Constant

In this case the node has one input, a literal is present and the function-name is addition; the literal is of type integer, length one byte and value four. The result-token is to be sent to destination-0.

An example of the diadic node form is the

Switch node. This node requires a bitstring token on input-point-0 and a token of any type on input-point-1. If the bitstring token is 'true' (the least significant bit of the data field is set) then the token on input-point-1 is sent to destination-1 otherwise it is sent to destination-0.

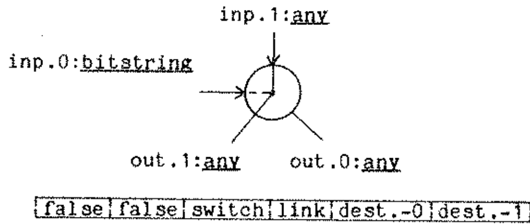


Fig. 11. Switch

With two-input nodes, tokens do not arrive simultaneously on both input-points. Therefore some mechanism must be provided to allow tokens to wait on one input-point until a complementary token arrives on the other input-point. The link points into a linked-list structure, which contains tokens queued on one input-point of two-input nodes.

Because of the practical difficulties of matching more than two input tokens, nodes have a maximum of two input-points. There is no real limit to the number of output-points but, for graph partitioning and 'stability' reasons, the initial limit is two.

Exception Handling

One of the system's token types is the ? or 'don't-know'. While a ? may be used in a variety of applications such as partial pattern matching, it's major use is to communicate information about 'exceptions' occurring during the evaluation of the graph to the graph itself. Exceptions fall into two classes:

1) Faults in the evaluation or attempted evaluation of node-functions for example function argument type exceptions (inc. input-output), arithmetic exceptions. With this class of exception a ? token is propagated to the succeeding nodes. ? tokens propagated in this manner retain the original reason for the exception and the destination at which that exception occurred. The ? token can not be used as a control token on conditional path nodes (Pass-if-Present, Pass-if-True, Pass-if-False, Switch).
2) Destination exceptions for example non-existent node, inactive input-point etc. Because no successor node exists for this class of exception, a reserved exception-node is defined in each processing-element. A token of type destination is sent to input-point-1 of the node and any ? arriving at the other input of the exception node is sent to that destination.

Input and Output Nodes

Input and output nodes are reserved and associated with particular devices and processing-elements. The actions of input and output nodes are as follows:

1) Input: a response-destination, which remains valid until another arrives, is sent to input-point-1; to input-point-0 is sent a token of any type. Depending on the nature of the device, the associated input node will eventually respond with valid data or a ?. If no response destination has been specified, a ? is sent to the processing-element's exception-node.
2) Output: a response-destination, as for input, is sent to input-point-1; to the other is sent data to be output. The node responds with a copy of the original data or a ?. If the output action fails and no response destination is specified, a ? is sent to the processing-element's exception-node.

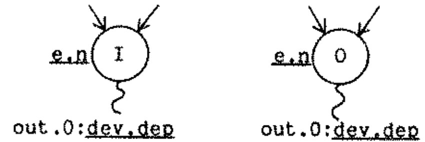


Fig. 12. Input and Output Nodes

Storage Nodes

In many applications it is necessary, from time to time, to update 'constants' in, for example, the difference equations which represent a digital compensator. While it is possible to retain such semi-permanent information by circulating it through the graph, this is, to say the least, not very efficient.

For this situation and others like it a storage node is provided. Input-point-1 of the storage node receives written tokens while input-point-0, when receiving any token, causes a copy of the last token written to input-point-1 to be sent to the successor node. If no token has been written a ? is sent instead.

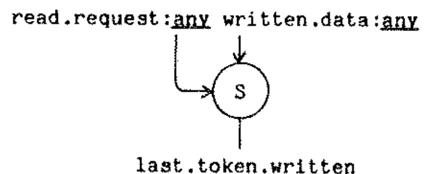


Fig. 13. Storage Node

A Further Example

The example below uses storage,input and output nodes in the implementation of a simple controller. Setpoint tokens arrive and are held in a storage node. Tokens arrive at the clock.tick input at 'approximately' the sampling rate T. The process output is staticised in a storage node for interrogation by other regions of the graph.

Provided it's time sense is maintained by clock-ticks, the controller maintains the process output at the last specified setpoint.

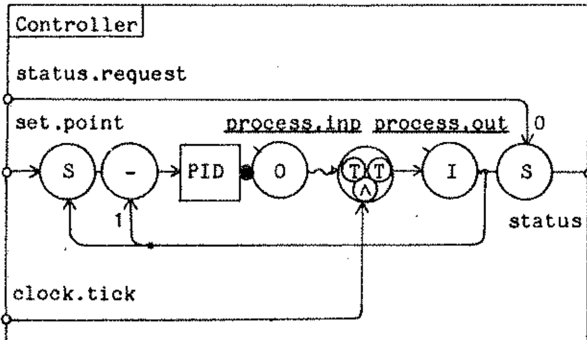


Fig. 14. Controller Sub-graph

System Architecture

In the following description of the system architecture, any detailed discussion of inter-processor communication has been omitted. This is intentional as the type and nature of the communication structure will vary dramatically from application to application, perhaps even within an application. The only requirement placed on the communication structure is that tokens sent from one processing-element to another will not overtake tokens previously sent.

Processing-element Sub-structure

The internal structure of processing-elements will also be application dependent. Nevertheless, the experimental structure described below is sufficiently general and flexible to explore many applications; it also closely parallels conventional microprocessor system architectures.

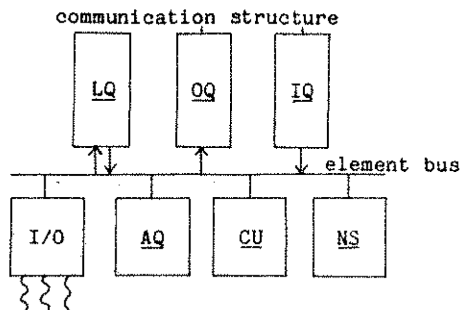


Fig. 15. Processing-element Sub-structure

Each processing-element is comprised of several sub-units. The major units are as follows:

1) Computational-Unit (CU): A shared controller and computational unit.
2) Node-Description-Store (NS): A conventional store containing descriptions of graph nodes assigned to this particular processing-element.
3) Input-Queue (IQ): A hardware FIFO buffer through which the processing-element receives tokens from other processing-elements.
4) Local-Queue (LQ): A software or hardware queue containing tokens generated by the processing-element which are destined for itself.
5) Output-Queue (OQ): A hardware FIFO buffer through which the processing-element transmits tokens to other processing-elements.
6) Arc-Queue-Store (AQ): A conventional store holding tokens which are queued on one arc of two-input-arc nodes.

The linked-list structure within the Arc-Queue-Store is the mechanism by which processing-elements preserve the queueing of tokens on arcs. Each two-input node-description has a link into the Arc-Queue-Store; the first entry in the Arc-Queue-Store, pointed at by the link, has the following fields:

1) input-point: the input-point on which tokens are currently queued; tokens are never queued on both input-points simultaneously as the node-function is evaluated immediately.
2) present: indicates that a token is queued on an input-point.
3) head: a pointer to the first token-entry on the arc.
4) tail: a pointer to the last token-entry on the arc.

Arc-Queue-Store token-entries have the following fields:

1) type: the token data type.
2) length: the length of the token data field.
3) data: the token data.
4) next: a pointer to the token-entry holding the next token queued on the arc.

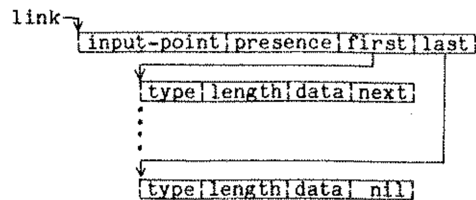


Fig. 16. AQ List Structure

Processing-element Operation

Processing-element operation proceeds as follows:

The Computational-Unit scans the Local and Input

Queues until a token is present. When a token arrives, the destination node-description is examined to see if it requires one or two operands:

1) If the node requires one operand, the Computational-Unit evaluates the node-function and writes any result-tokens to either the Local or Output Queue.
2) If the node has two inputs, the Arc-Queue-Store is accessed via the link for the second operand token. If the token input-points are complementary, a token is removed from the head of the arc queue and the node-function evaluated; otherwise the arriving token is placed on the tail of the arc queue.

The Computational-Unit then returns to scan the Local and Input Queues.

The basic operation is modified when the element supports input-output transducers. As with all conventional systems, the data rate of most input-output transducers will be slow compared with the processor. The Computational-Unit may initiate an input-output action and then continue with normal processing. Eventually the action will be accomplished and, _if_ a response destination has been specified, the transducer may interrupt the Computational-Unit. The Computational-Unit resets the transducer status and places the response-token on either the Local or Output Queue. The Computational-Unit then resumes normal processing.

Real Processing-elements

The association between processing-element sub-units and conventional system sub-units is straight forward. In the experimental elements the Computational-Unit is based on a Zilog Z80A microprocessor [14]. The data-flow interpreter resides in erasable read only memory and will occupy from 2K to 16K bytes depending on the node-function and data-type set supported. The Arc-Queue-Store, Node-Store and Local-Queue reside in random access memory, although the Node-Store could also reside in read only memory. MOS FIFOS are used for the Input and Output Queues.

'Typical' graphs will involve an interpretation overhead of the order of 20%. With the assistance of a hardware arithmetic unit, such as the Am9511 [1], computational throughput will increase but the interpretation overhead will rise to around 80%. These overheads are satisfactory where throughput is not the major consideration [9] and compare favourably with other interpretive systems.

System Status

A paper such as this can only attempt to give the flavour of the system. Several mechanisms have not been discussed at all; amongst these are:

1) Sub-graph and recursive sub-graph sharing, important in computationally intensive control tasks [11].
2) The stream mechanism of Weng, important in the development of functional languages [13].

Research with the system is now well established with work completed to date including:

1) Comprehensive simulation software which can evaluate graphs with the order of 10,000 nodes and model upwards of 100 processing-elements. The software is written in Pascal [7].
2) A translator supporting a textual encoding of the system's graphical target language. The translator allows macro sub-graph expansion and plants the necessary encapsulating primitives for shared sub-graphs [5][11].
3) Amongst other studies, algorithms for object recognition based on multiple-hypothesis testing using a laser range-finder [11].

Work in hand includes:

1) Application specific languages and their compilation on data-flow systems.
2) A detailed study of system aspects associated with time and non-determinacy.
3) 'Fast' purpose built processing-elements.
4) Communication techniques for systems of closely coupled processing-elements.
5) Computer assisted graph partitioning with practical constraints.

Acknowledgement

The system described in part in this paper was developed during the course of the author's doctoral research. The author is indebted to Professor D. Morris for his guidance, and the University of Manchester's Department of Computer Science for financial and engineering support.

References

[1] Advanced Micro Devices, _Am9511 Arithmetic Processing Unit (Product Sheet)_, Advanced Micro Devices Inc., Sunnyvale, California.

[2] D.A. Adams, 'A Model for Parallel Computations', in Hobbs (ed) _Parallel Processor Systems, Technologies and Applications_, pp311-333, Spartan Books, 1970.

[3] J. Backus, 'Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs', CACM Vol.21 No.8, pp613-641, Aug. 1978.

[4] P. Coiffet et al., 'Real Time Problems in Computer Control of Robots', Proceedings of the 7th International Symposium on Industrial Robots, pp145-152, Oct. 1977.

[5] G.K. Egan, 'A Study of Data-flow: It's Application to Decentralised Control', Ph.D. thesis, Dept. of Computer Science, University of Manchester, 1979.

[6] R.N. Ibbett and P.C. Capon, 'The Development of the MU5 Computer System', CACM Vol.21 No.1, Jan. 1978.

[7] K. Jensen and N. Wirth, Pascal-User Manual and Report, Springer-Verlag, New York, 1975.

[8] R.M. Karp and R.E. Miller, 'Properties of a Model for Parallel Computations: Determinacy, Termination and Queueing', SIAM J. Applied Mathematics, Vol.11 No.6, pp1390-1411, Nov. 1966.

[9] R. Mori et al., 'Microcomputer Applications in Japan', IEEE Computer, Vol.12 No.5, pp64-74, May 1979.

[10] K. Ogata, Modern Control Theory, Prentice-Hall Inc., Englewood Cliffs, N.J., 1970.

[11] C.P. Richardson, 'Object Recognition using a Dataflow Machine: Algorithms for a Laser Range-finder', M.Sc. dissertation, Dept. of Computer Science, University of Manchester, 1979.

[12] N.R. Sandell et al., 'Survey of Decentralised Control Methods for Large Scale Systems', IEEE Transactions on Automatic Control, Vol.AC-23 No.2, pp108-128, Apr. 1978.

[13] K.S. Weng, 'Stream-oriented Computation in Recursive Data-flow Schemas', Technical memo #68, Laboratory for Computer Science, Massachusetts Institute of Technology, Oct. 1975.

[14] Zilog, Z80/Z80A-CPU Technical Manual, Zilog, Cupertino, California, 1977.