LANGUAGE CONSIDERATIONS OF PARALLEL PROCESSING SYSTEMS PART ONE: Concurrent microprocessing systems

INFORMATICA 2/87

UDK 681.3.06:519.682/.683

Peter Kolbezen
Institut »Jožef Stefan«, Ljubljana

ASTRACT - Full parallelism offered by the multi-processor is not still fully exploited. Much work that has been done in structured programming to separate a mono-processor program into well-defined modules, and attempts to systematize the interactions between modules, have helped to achieve a more disciplined approach to software development with much benefit to multi-mikroprocessor software.

This paper presents various issues relevant to language aspects of parallel processing systems. The objective is to present a discussion of issues and some of the current approaches rather than a well-developed metodology of software, which has yet to be developed. New approaches to parallel processing architecture are briefly outlined too.

O JEZIKIH SISTEMOV PARALELNEGA PROCESIRANJA. PRVI DEL: Konkurenčni mikroprocesorski sistemi. Popolna sočasnost, ki jo omogoča materialna oprema večprocesorskih sistemov, še ni dovolj izkoriščena. Da bi se ta cilj dosegel, je bilo med drugim vloženo že veliko napora tudi v strukturirano programiranje, ki deli enoprocesorski program v dobro definirane module, poskuša sistemizirati akcije med moduli, pomaga doseči bolj urejen pristop k razvoju programske opreme in ji daje številne prednosti.

Članek podaja zaključke, ki izhajajo iz jezikovnega vidika na sisteme paralelnega procesiranja. Obravnavani so zgolj rezultati in novejši poskusi reševanja problemov programske opreme. O kaki bolj dovršeni metodologiji programske opreme pa ni moč govoriti, saj je le-ta še vedno v razvojnih fazah.

Na kratko so opisane tudi nekatere najvidnejše računalniške arhitekture, ki še posebej učinkovito podpirajo paralelno procesiranje.

1. INTRODUCTION

High level languages and their translators have become essential for writing application programs for mono-processor systems. The same, however, cannot be said for multi-microprocessor systems. The immense variety of applications and hardware architectures, and the diversity of philosophies about how systems shoud be structured, makes it extremely difficult to design languages that are likely to be widely accepted. It still remains a difficult challenge to design a high level language which is sufficiently general and modular to accommodate a large number of architectural types of machines /1/. In the absence of bold and fresh ideas to express concurrency, it is then natural that current thinking is along the lines for extending or generalizing the sequential programming languages /2/. At least it is known that using this approach one has something that works for an isolated microprocessor which forms a constitutent part of the whole system. Thus a sequential language enables individual software modules to be written. This is a rather primitive approach, however, where concurrency (which requires a control and communication structure), synchronization for resource sharing, efficiecy and robustness aspects are outside the language consideration.

A further difficulty stems from the fact that the language issues and runtime support aspects cannot be isolated totally. The attributes of the kernel are important in deciding whether or not certain issues need to be dealt with at the language level.

Most of the language proposals in the concurrent programming area also have an underlying model of distributed computing. The many of these languages are in the research phase and any have not been implemented, also there is little hard practical experience. Most of the time the underlying model is not explicitly stated.

Event if one attempts to extract the underlying model from a proposal, it is not always an easy task. Sometimes the model and languages issues become inseparable. The choice of the model would affect the programming methodology and the proof techniques for a language based on that model. A model provides a conceptual framework in which to discuss and understand the behaviour of concurrent computations, and is intended to capture the underlying philosophy of a programming language.

A high level language is a medium which not only enables us to obtain a machine executable code but, perhaps more importantly allows us to formulate an application precisely. In this sense, there is a greate vacuum for a vehicle to describe concurrent applicatios formally.

Another difficulty in using languages applicable to multi-microprocessor systems is the necessity for a translator. Translator writing immediately requires the specification of the target machine. It is desirable that the translator also runs on the target machines. Since there is no architectural uniformity, this requires a translator design which is capable of running on widely varying configurations. Ideally, a translator also should take advantage of the structure and hence be modular. This requires significant departure from compiler writing for mono-processor systems.

2. FEATURES OF CONCURRENT LANGUAGE

Some of the desired features of a concurrent language can be listed as follows /3/:

- expressive power or richness - provability - ease and efficiency of implementation - easy of use - readability of resulting programs - impact of changes - extent of concurrency possible

Expressive power or richness This refers to the ability of the model/language in being able to express certain behaviours, i.e. the richness to be able to model certain computations like recursion, non-determinism, and so on. This property is also referred to as completeness or adequancy. An increase in expressive power is likely to be accompanied by an increase in the difficulty of proving programs. While it is desirable to have simplicity as one of the goals, it is not advisable to have that as the overriding criterion.

Provability One may be interested in proving many properties, like partial correctness, freedom from deadlocks, termination, fairness, etc. The presense of some constructs would make it extremely difficult, if not impossible, to prove certain properties. For example, at the current state of the art of program proving, the presence of time-outs could make the achievement of the tractability of proofs almost impossible. Of course, an important consideration is the power of the language used for specifyng assertions about program properties. The assertion language or the logic used should be rich enough to be able to specify formally various desired properties /4/.

Formalization of the semantics of constructs is an important prerequisite for program proving. While researches have been discussing all of the above properties for a long time, there are very few well defined techniques or formal methods to illustrate the existence of the necessary properties.

Ease and efficiency of implementation The implementation of certain features may be quite difficult to achieve. It is not sufficient merely to define primitives whose functionality makes them worth implementing. It must also be possible to deliver that functinality with reasonable efficiency. In most aplications the efficiency, or costliness, is likely to be an important consideration. While some constructs might be implemented easily, the efficiency of such implementations may not necessarily be good. The practicality of mechanisms would be measured by the efficiency of their implementations.

Ease of use The presence of powerful features does not mean that they would be easy to use. Normally high level constructs and good abstractions capabilities make thing easier. Ease of use and expressive power are complementary criteria. A model/language being rich enough to express a certain type of computation does not automatically mean that it could be done in an easy way-certain ingenious, awkward and obscure ways have to be resorted to. Constructs which reflect intuitive ways of abstractions would be appealing to the user.

While writing programs, language primitives should allow coherent combinations. Avoiding subtle interactions among primitives would make them easier to use and help reduce errors. The flexibility of the constructs is also an important factor in the ease of their use.

Readability of resulting programs Any proposal for new language features should be scrutinized closely to determine the effect of the proposed facility on program structure. The mechanisems should be such that they discourage complex and confusing structures. The presense of high level and very powerful constructs could lead to easily comprehensible programs. Of course, this may not always be the case. The ability to compose the process structure hierarchically should be of great benefit. In general, constructs that are easily verified are likely to be easily understood.

Impact of changes If the constructs do not include or force a high degree of modularity, a change in the definition of one process may necessitate many changes throughout the rest of the system. This would be highly undesirable, particularly if the number of the processes involved is quite large. Permitting a great degree of autonomy in the definition of processes would help a good deal in reducing and localizing the impact of changes.

Extent of concurrency possible The greater the degree of concurrency the constructs permit to be expressed, the better. But the overheads involved in supporting such concurrency should not be such as to offset the advantages gained through the increase in parallelism.

3. HIGH PARALLEL PROCESSING ARCHITECTURES

It is agreed by all concerned that the key to fifth generation computer architectures is a much higher degree of parallelism than is incorporated into computers at present. It is likely that there will be a number of layers of parallelism: closely coupled processing elements reflecting the parallelism inherent in inference or knowledge base processing operations, looser coupling between the various subsystems in a fifth generaton computer, and distributed processing across local and wide area networks of computers /3/.

At present there are two types of close-coupled parallelism implemented in computers: processing arrays and pipelines. Processing arrays are vectors of identical processing elements which act synchronously to perform identical opreations on arrays of data. Pipelines are used for multi-stage operations /7/ such as floating-point multiplications, where each element of the pipeline carries out one step of operation, and passes its intermediate result to the next element. Operations on successive sets of data can take place at intervals of one step. Parallel processing of this type, known as "regular" parallelism, will undoubtedly find a place in fifth generattion computers, but mechanisms to deal with irregular parallelism are the main topic for research. Three approaches are present today: parallel control flow, dataflow and graph reduction /9,10/.

Traditionally, by parallel control flow each step of a program is executed in sequence, under the control of a single program counter which determines the lowlevel operation to be carried out next. The flow of control is implicit in the structure of the program. Each statement in the module is a call to a more detailed processing procedure. Therefore, if a parallel computer system and programming language were available, the processing procedure are called at the same time. They executed in parallel, and the control module waits until all processing procedure are complete before continuing. Programming languages such as concurrent Pascal and Ada have facilities for operations of this sort computers, but it remains to be seen whether this approach, which is only a slight variation on conventional sequential processing, will be adequate for the radical demands of fifth generation architectures.

For a number of reasons, one of the most promising architectural models, certainly for the inference processing subsystems of a fifth generation computer, is dataflow architecture. It can cope with irregular as well as regular close-coupled parallelism, it is flexible and extensible, it has the potencial for very high data throughputs, and it reflects, at hardware level, the type of parallelism inherent in inference processing. The central idea of dataflow architecture is: a network of processing elements is set up, which reflects the logical structure of the task to be carried out, and items of data flow between the elements. Each elenents operates at its own pace, and waits until it has a complete set of intermediate inputs before it "fires". There are two techniques for the control of such a network. In the totally data-driven approach, each element waits passively for data to arrive, whereas in the demand-driven regime each element issues requests "upstream" for data when it is ready for it. In general a dataflow computer or computer subsystems has three requirements:

- to store representations of program graphs,
- to implement some form of data tokens to flow through the graphs, and
- to provide suitable instruction processing facilities.

Each requirement poses certain probleas, some of which are quite severe. Program graphs in practice will contain hundreds of thousands, if not millions, of arcs and nodes, and may not always reduce to the neat tree structure. Furthemore, if, as is almost certain, program contains recursive definitions, portions of the structures will be re-entrant. In this example the graph (recursive program graph) of this inference needs to replicate itself repeatedly during processing. Further, most of the data processed in knowledge-based systems does not consist of single items, but of large structures which would cause unacceptable overheads if they were passed through a dataflow network in their entirety. This problem is being approached by using pointers to the data structures in the dataflow networks, and accessing the structures from fixed memory only when they are required.

Three lines of research are being followed in response to these difficulties. The first is to regard a dataflow task as fixed at compile time, and to prohibit re-entrat code /12,13/. This static approach is ilustrated in Fig.1, which uses a network of binary processors each with two alternative output channels.

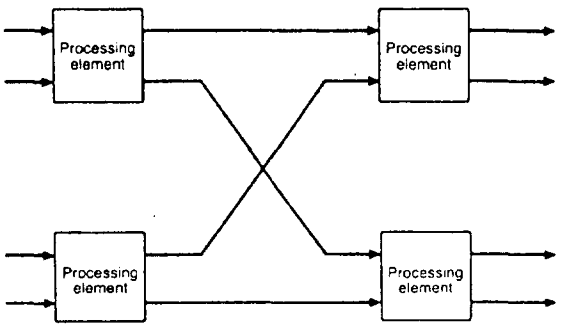


Fig.1-A static dataflow network(delta network).

The dynamic approach gets round the problem of re-entrat code by allowing replication of portions of the network at run time. This has the virtue of simplicity, and may become increasingly feasible as hardware constrains slacken. Fig.2. illustrated one possible configuration using this technique /14/.

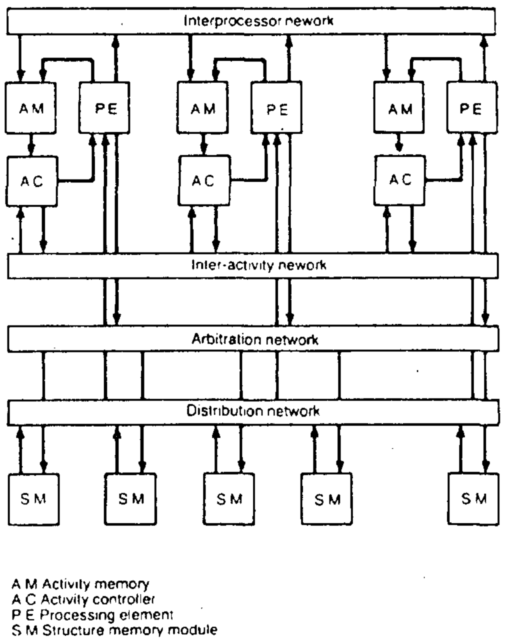


Fig.2-A dynamic dataflow architecture.

The line of development which holds out the most promise in the short term is the tagged system, variations of which are under development at MIT and Manchester University. Each data items flowing through the network carries with it an identification tag, which specifies its type (for example it may be a pointer to a large data structure held in fixed store) and its position in the program. The tags enable data items to be paired and matched with appropriate instructions for processing. The tags also indicate the level of recursion if re-entrant code is used. One node of a data-flow systems using this approach /15/ is illustrated in Fig.3.

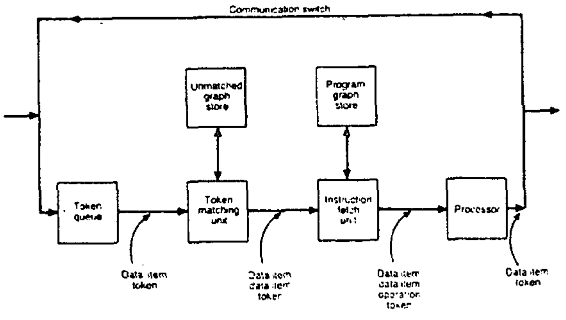
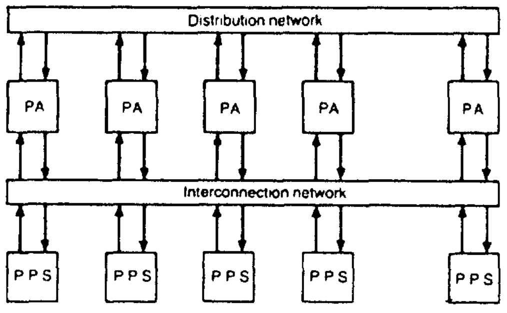


Fig.3-A tagged dataflow architecture.

The graph reduction architecture /16/ is the next variation on the dataflow approach discussed above. This variation is to evaluate functions by working directly on their graphical representations. As various portions of the graph are evaluated, they are repleaced by their intermediate results. Evaluation of each of the lowest nodes (which becomes a search to see whether such a node is present in the given relations), can proceed in parallel. The intermediate boolean results are then fed back through the graph as it is reduced, until a single result emerges. ALICE /17/ is the computer which incorporates graph reduction directly into its basic architecture. It is designed to be programmed in the applicative language HOPE, but can also support declarative languages such as PROLOG. The architecture of ALICE enables the parallel operations to be performed without any explicit instructions from the program. Each node in a program graph is represented as a packet within Alice. A packet consists of an identifier fields, a function or operator field, and one or more argument fields, which may be data values or references to other packets. There are also control fields used by the computer in its operation.

The general layaut of an ALICE computer is shown in Fig.4. It consists of a large segmented memory serving as a packet pool, and a number of processing agents. The processors and the memory segments are connected by a high-speed switching network which enables any processor to access any memory segment with minimal delay due to other access path. The configuration chosen is a delta network, comprising a large number of simple switching elements with four inputs and four outputs in a regular array. (Fig.1 shows a delta network of elements with two inputs and two outputs.) The network operates asynchronously, so that each request for a packet is propagated through the switches as rapidly as possible, and the packet is returned to the processor as soon as the access path is open.



PA Processing Agent
PPS Packet pool segment

Fig.4-Alice: overall structure.

Also linking each processing agent is a low bandwidth distribution network, which contains addresses of processable packets and empty packets. This network includes simple processing elements which transfer these addresses from one processing agent to another, in order to even out the queue of work waiting at each processor. ALICE uses the INMOS transputer /12,13 / as its basic processing element: each main processing agent cointains a number of transputers, and additional transputers provide the intelligence in the distribution network. The transputer is designed as a single-chip processing element for parallel computer architectures. It has an one-board memory, with high-speed DMA (for input and output channels, bypassing the processor) facilities and reception and transmission registers for data transfer between transputers. Its single sequential processor has a reduced instruction set (RISC processor) for maximum speed (instruction cycle time of 50 nanosecond). Transputer is designed for a very high throughput of data, even if the processing rate is not so high.

The transputer is designed to be programmable directly in Occam programming language /18/. It is intended to be incorporated in a distributed architecture, with individual transputers connected by a very high speed local area network. As such it is an ideal building block for many components of a multi-microprocesor fifth generation computer system.

4. CONCLUSION

With the increased interest in multi-micropro-cessor and distributed computing systems, there is emerging a large number of proposals and approach to handle them. In a multi-micropro-cessor design the architectural philosophy requires the interrelated consideration of application requirements, hardware communications, and software aspects. While more detailed treatment of software issues is left until second part which succeed of this paper, each of the considerations, as are: types of communications, priority, co-ordination of processes, process-procesor allocation, network visibility, control issues, and synchronization /19/ should be seen as having implications as to the structure, hardware, and software of a system.

In multi-microprocessor systems the architectural structure, applications requirements, and varied software aspects like the operating systems /20/, communications infrastructure, and tools to aid application programming such as high level languages suitable for parallel programming, all form a tightly knit situation in which it is far more difficult to isolate the constituent parts and arrive at universally accepted solutions.

The requirements of the fifth generation for layers of parallelism and an emphasis on inference rather than numerical computation look like providing sufficient incentive. Even if the objective of a computer with enhanced intelligece is not attained, the new architectures will provide engines of unprecendent power for conventional computing. The move away from general-purpose processors to aggregations of special-purpose chips is likely to affect all branches of information technology. The increase in the scale of integration, and the advanced CAD systems for microchip production will find applications in every branch of microelectronics. The industrial, economic and political consequences of having access to, or not having access to the new generation of silicon foundries are farreaching.

5. REFERENCES

/1/ Bedina M., Distante F. SW & HW Resources Allocation in a Multiprocessor system: An Architectural Problem, Advances in Microprocessing and Microprogramming, EUROMICRO, 1984.

/2/ Tucker A.B., Jr. Programming languages (Second edition), McGraw-Hill Book Company, 1986.

/3/ Stankovič J.A., A Perspective on Distributed Computer Systems, IEEE Trans. on Comp., vol.C-33, NO.12, December 1984.

/4/ Chandy M., Misra J. Distributed simulation: A case study in design and verification of distributed programs, IEEE Trans. on Software Engineering, September 1978.

/5/ Ma P.R., Lee E.Y.S., Tsuchya M. "A task allocation model for distributed computing systems", IEEE Trans. on Comp., pp. 41-47, Jan. 1982.

/6/ Flynn M.J. Directions and Issues in Architecture and Language", Computer IEEE, October 1980.

/7/ Milutinović V. A High Level Language Architecture: Bit-Slice-Based Processor and Associated System Software, microprocessing and Microprogramming 12, 1983.

/8/ Kogge P.M. The Architecture of Pipelined Computers, McGraw-Hill Book Company, 1981.

/9/ Treleaven P. Fifth generation computer architecture analysis, in Moto-Oka, 1982, pp. 265-275.

/10/ Hwang K., Briggs F. Computer Architecture and Parallel Processing (International Student Edition), McGraw-Hill Book Company, 1986.

/11/ Brajak P. Paralelno procesiranje: arhitektura 90-tih godina, Zbornik jugoslovenskog savjetovanja o novima generacijama računala, MIPRO 86, Rijeka, Maj 1986, str.33-46.

/12/ Smith K. New computer breed uses transputers for parallel processing, Electronics 56,4, 1983, pp.67-68.

/13/ Mihovilovič B., Mavrič S, Kolbezen P. Transputer-osnovni gradnik večprocesorskih sistemov, Informatica 4/86, Ljubljana, 1986.

14/ Tanaka H., et.al. The preliminary research on the data flow machine and the data base machine as the basic architecture of fifth generation computer systes, in Moto-Oka, 1982.

/15/ Gurd J. Developments in dataflow architecture, in SPL Internacional, 1982.

/16/ Cripps M., Field A.J., Reeve M.J. The design and implementation of Alice: a parallel graph reduction machine, Byte Magazine, June 1985.

/17/ Darlington J., Reeve M.J. Alice: a multiprocessor reduction machine for the parallel evaluation of applicative languages, ACM Confernce on Functional Programming and Computer Architecture, October 1981, pp.65-75.

/18/ Inmos. Occam Programming Manuel, Prentice-Hall, 1984.

/19/ Kolbezen P. Analiza multiprocesorskega sistema, IJS Delovno poročilo Dp-4461, Univerza E.Kardelja, IJS, Ljubljana, December 1986.

/20/ Kolbezen P. Problemi načrtovanja multiprocesorskega sistema, IJS Delovno poročilo Dp-4462, Univerza E.Kardelja, IJS, Ljubljana, December 1986.

/21/ Foster M.J., Kung H.T. The design of special-purpose VLSI chips, IEEE Computer Magazine 13,1, 1980, pp.26-40.

/22/ Seitz C.L. Conccurent VLSI Architectures IEEE Trans. on Comp., Vol.C-33, No.12, Dec. 1984.