The Knowledge Engineering Review, Vol. 4: 1, 1989, 53–71

Parallelism in knowledge-based machines

APOSTOLOS N. REFENES
Department of Computer Science, University College London, UK

Abstract

The application area of *knowledge-based expert systems* is currently providing the main stimulus for developing powerful, parallel computer architectures. Languages for programming *knowledge-based* applications divide into four broad classes: Functional languages (e.g. LISP), Logic languages (e.g. PROLOG), Rule-Based languages (e.g. OPS5), and, what we refer to as self-organizing networks (e.g. BOLTZMANN machines).

Despite their many differences, a common problem for all language classes and their supporting machine architectures is parallelism: how to de-compose a single computation into a number of parallel tasks that can be distributed across an ensemble of processors. The aim of this paper is to review the four types of language for programming *knowledge-based expert systems*, and their supporting *parallel machine architectures*. In doing so we analyze the concepts and relationships that exist between the programming languages and their parallel machine architectures in terms of their strengths and limitations for exploiting parallelization.

1 Knowledge processing and parallel computation

Over the last 30 years the focal point of computing has shifted from numeric processing in the 60s, onto data processing in the 70s, and onto knowledge processing in the 80s and 90s. This is being mirrored by the rapid development of new programming styles and computer architectures that are sympathetic to symbolic processing and parallel computation. Although most of these new languages and architectures are termed "general purpose" they rarely prove to be equally suited for all classes of problems. It is therefore important to understand the computational mechanisms underlying the major programming styles, so as to make useful observations about their strengths and weakness and in particular their potential for exploiting parallelism.

1.1 *Knowledge Processing*

Knowledge processing is the area of computer science concerned with the design of so-called "intelligent" computer systems. These systems are required to exhibit characteristics normally associated with human intelligence. A knowledge-based program has a number of distinctive properties. Firstly, the program contains an explicit representation of empirical human knowledge. Secondly, this representation of knowledge is relatively easy to read and understand. Thirdly, the program is able to provide, like a human, an explanation of its reasoning. Lastly, the program is easily modifiable, as when additional knowledge is included.

Within the area of knowledge processing two contrasting approaches are identifiable. Firstly, there is the *symbol manipulation* approach according to which knowledge is a collection of assertions represented by formal symbols—in a suitable language—and, "thinking" is the process of making deductions by manipulating these symbols. Secondly, there is the *connectionist* approach according to which "thinking" emerges from the neural connections forming and re-forming in the brain and connectionist computers attempt to emulate the way in which neurons in the brain perform computations. Although the symbol manipulation approach has so far had the greatest

impact in the design of parallel computer architectures, a number of massively parallel connectionist machines are under rapid development in various laboratories. The spectrum of programming languages for knowledge processing and their corresponding parallel computer achitectures is illustrated in Table 1.

Following the *symbol manipulation* approach are "general-purpose" languages such LISP (Abelson and Sussman, 1984) and PROLOG (Clocksin and Mellish, 1981), although specialized knowledge-based languages such as CMU's OPS5 (Forgy, 1981) are finding increased usage. Following the *pattern manipulation* approach are semantic network languages such as NETL (Fahlman, 1980) and massively parallel computer networks such as the MIT Connection machine (Hillis, 1984) and the CMU BOLTZMANN machine (Fahlman et al., 1983).

functional languages and in particular LISP are the cornerstone of AI applications in the USA. LISP-like functional languages are characterised by their ability to treat code and data as first class citizens which is an essential feature for suporting dynamically evolving knowledge-based systems. Functional languages, and in particular the more recent *reduction* languages (Kluge and Schmittgen, 1987), derive much of their strengths from the fact that they are based on rigorous mathematical formalisms (i.e. λ-calculus (Church, 1941), and recursion equation systems).

logic languages and in particular Prolog, are the cornerstone of AI research in Europe and Japan. Logic languages trace their roots to a branch of mathematics known as the (horn-clause subset of the first order) predicate calculus (van Emden and Kowalski, 1976).

rule-based languages are extensively used for the construction of knowledge-based expert and consultation systems. In rule-based languages, the knowledge base consists of rules in the form "IF *condition* THEN *action*". Each rule (i.e. production) consisting of a conjunction of conditions corresponding to the "IF" part of the rule and a set of actions corresponding to the "THEN" part of the rule.

connectionist languages are used for describing semantic networks. A semantic network is a knowledge representation formalism using a directed graph. Here vertices represent objects and concepts (e.g. sets) and the arcs represent relations (e.g. membership). In a connectionist computer each processor is connected to its "nearest neighbours" in a regular pattern that matches the flow of data and control in the target computation.

Each class of programming language is supported by a corresponding parallel machine architecture. To support the *symbolic processing* languages many new computers are being developed. At the present time symbolic processing computers are constructed from highly microprogrammed

Table 1 Computer Architectures and Programming Languages for Knowledge processing

	Symbol processing			*Pattern processing*
Programming languages				
	Functional	Logic	Rule-based	Connectionist
	Lisp, Miranda Hope, FP	Prolog, Parlog Spinlog	OPS5	NETL NIL
Computer architecture				
	Reduction machines	Inference machines	Production machines	Self organizing networks
	GRIP, ALICE	WAM, KABU WAKE	DADO	BOLTZMANN

workstations such as the SYMBOLICS 3600 (Moon, 1985) for LISP and ICOT PSI (Uchida et al., 1983) for PROLOG. To support the *pattern processing* approach are massively parallel architectures consisting of tens of thousands of simple processors which are generally not explicitly programmable. Instead, many of the connectionist machines incorporate built-in algorithms for learning and cognition.

Despite their many differences, a common problem for all architecture and language classes is parallelism; how to de-compose a single computation into a number of parallel tasks that can be distributed across an ensemble of processors. In the following section we discuss the principle ways of orchestrating parallel computations.

1.2 *Parallel Computation*

The implementation of many concurrent applications on general purpose computer networks appears to dictate a specific network structure. May et al. (1987) taxonomize concurrent applications into three groups according to the kind of parallelism that characterises them:

Algorithmic parallism. This group contains applications in which parallelism is extracted by decomposing the *algorithm* into a number of smaller steps which can be executed in parallel. Generally, these steps are fairly simple and are often performed repetitively within a loop. The best known example of algorithmic parallelism is the pipeline. Algorithmic parallelism often offers speed-ups proportional to the number of processors used. The limiting factor is that the throughput of the pipeline is restricted by the throughput of the slowest element in the pipeline. Therefore in order to have the potential for maximum speed-up a pipeline must be balanced; i.e. each stage in the pipeline must process data at the same rate. Although algorithmic parallism is well understood and has been successfully exploited in traditional numerical and scientific computations it is not, as we shall see, easily found in symbolic processing.

Geometric parallelism. This group contains applications where parallelism is obtained by distributing the *data* to be processed across an ensemble of processors in such a way that the original geometric structure is preserved. Each processing element in the network is responsible for processing (moderate amounts of) local data and occasionally synchronizing with its neighbours between computational steps. Examples of geometric parallelism are provided by statistical mechanics such as modeling the magnetic properties of iron (May et al., 1987). Often, geometric parallelism requires moderate amounts of local memory in each processing element, and communication is the major performance bottleneck. Only if the data dependencies are characterised by a geometric locality of reference this kind of parallelism offers speed ups proportional to the number of processors used.

Farming parallelism. This group contains applications in which a network of processors are used to process data delivered to them by a controlling processor i.e. *farmed out.* Except for the initial and final delivery of the data, there is little synchronization between the processing elements. These applications consist of two parts. The first is responsible for farming out the work and the second is usually an application-specific process. Examples of such parallelism can be found in operating systems i.e. processes in Unix, various graphics applications, and as we shall see, in most symbolic processing languages.

Naturally, the three kinds of parallelism are not mutually exclusive and a number of applications (e.g. solid modeling) show aspects of both alogirithmic and geometric decomposition. Interestingly, most symbolic processing languages do not allow the user to explicitly specify the way in which the program is to be executed on the parallel architecture. Instead, the exploitation of parallelism is achieved by implicit disciplines (which, nevertheless, use a combination of these three kinds of decomposition to exploit parallelism). In the following sections of this paper we examine the various computational styles, their corresponding computer architectures, and the ways in which they exploit parallel de-composition.

2 Functional languages and reduction machines

In the Functional style of programming, a "program" is regarded as a collection of functions in the true mathematical sense. A function is applied to its (input) arguments resulting in output values, and may define complex operations over list structured arguments, for example. Functional programs possess some important properties not usually found in Procedural styles. 'Referential transparency" ensures that the value of an expression is determined only by its definition and not its computational history. Freedom from side-effects and the restriction of assignment operations also contribute to a more "pure" model of computation.

Reduction machines constitute a novel approach to computer programming and program execution. They are designed to accept formal or declarative specifications of application problems as inputs, and return problem solutions as outputs. Formal problem specifications are programs which specify *what* is to be computed rather than *how* it is to be computed. This is achieved by means of coherent mathematical expressions or by sets of function definitions. Problem solutions are deduced from the respective specifications by *a process* of meaning-preserving transformations (Kluge, 1979). The order in which these transformations are to be carried out is determined by the structure of the problem and the availability of resources. The absence of procedural elements renders functional languages free of side effects and thus, suitable for parallel execution.

2.1 *Computational model*

The most important construct of a functional programming language (supported by a reduction system) is an equation of the general form:

$$f(e_1, \ldots, e_n) = expr(\cdots f \cdots e_1 \cdots e_n \cdots) \quad \text{where} \quad expr(e_i, \ldots) = \ldots$$

defining a function f of n arguments. The left hand side, referred to as the function head, specifies the variable f as a function name and the variables $e_1, \ldots, e_n$ as the formal parameters of the function. The expression on the right hand side, referred to as the function body, may contain one or several occurrences of the variables f and $e_1, \ldots, e_n$. It specifies the computation of the function value. If the f occurs at least once within the function body we have a so-called recursive function. Following the key word *where* we may (recursively) insert further function definitions which are local to $expr(\ldots)$ and may use any of the variables f, or $e_1, \ldots, e_n$ as their global parameters.

Expressions are either atomic or are constructed from function applications. Atomic expressions are variables, primitive function symbols, decimal numbers, character strings, and the empty list. Atomic expressions vary from language to language. Typically, the primitive functions of the language include:

- the binary arithmetic functions +, −, *, etc, which map decimal numbers to decimal numbers;
- the binary relational functions >, <, =, < >, etc, which map decimal numbers into the Boolean constants TRUE, or FALSE;
- the unary list processing functions *cons*, *head*, *tail* which are used to compose and decompose list structures.

Function applications are generally denoted as $func(arg_1, \ldots, arg_n)$; where *func* is either the name of a similarly defined function or a primitive function, and $arg_1, \ldots, arg_n$ are argument expressions. In addition, functional languages provide *if-then-else* clauses of a standard form and *lists* (or nested sequences) of expressions.

Execution of functional languages is based on systematically transforming expressions to equivalent and simpler expressions using the definitions as rewrite rules until they contain no further functions, only constants (or so-called constructor functions). Reducible expressions are substituted by expressions that have the same meaning. Systems that support variables use the universal substitution of the λ-calculus (Church, 1941) to insert expressions in expressions. Given an expression, *expr*, there are, in general, a number of alternative reduction sequences some of which

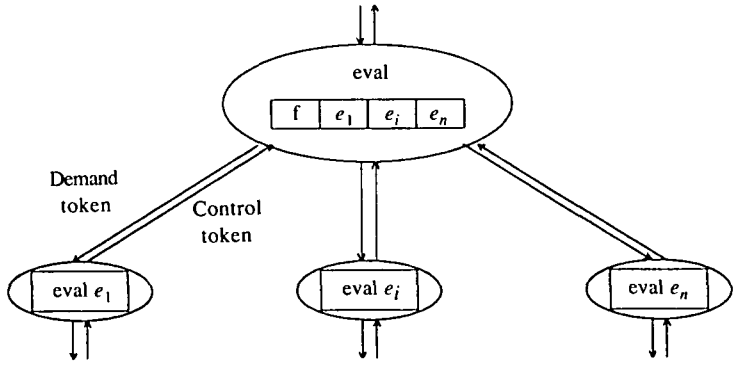

Figure 1 Farming De-composition in Functional Programs.

may not terminate. However, reduction machines must feature the Church–Rosser property which demands that: (a) all terminating sequences of reductions applied to an expression yield the same constant expression; and (b) if an expression has a meaning, then there must exist at least one sequence of reductions applied to it that terminates. In contrast to conventional computing systems, variables do not represent contents of memory cells but merely serve as placeholders and are used in their strict mathematical sense. The substitution of bound variables follows a strict discipline of the λ-calculus whereby the argument of a substitution is reproduced for every free occurrence of the bound variable within the body of the enclosing *lambda*-expression (i.e. β-reduction of the λ-calculus (Church, 1941)).

Given a functional program which is in effect a reducible expression, it is possible to spawn off all subexpressions as autonomous programs that are potentially reducible in parallel. The expressions being dealt with are function applications and hence the sub-expressions are the arguments to the function. Therefore, parallel evaluation of a function application such as $f(e_1, \ldots, e_i, \ldots, e_n)$ causes all the arguments $e_1, \ldots, e_n$ to be evaluated in parallel prior to being used by the function f (see Figure 1). This approach is an extended version of the call-by-value evaluation mechanism in which every argument of a function application is evaluated prior to function call. Conceptually, the computer system consists of two parts: the first is responsible for preparing and farming out the reducible expressions, and the second is responsible for evaluating these expressions.

To support this model internally each processing element often supports multi-tasking. Like any processor farm, to obtain maximum multi-processor parallelism which is proportional to the number of processors used, the synchronization between processing elements must be kept to a minimum. This requires coarse grain parallelism, fast context switching, and connectivity topologies with minimal contention. The absence of either algorithmic or geometric parallelism is the limiting factor in the construction of large scale reduction machines.

In the following section we give an overview of reduction machines with a description of the Imperial College ALICE machine.

2.2 *Imperial College ALICE*

The ALICE (Applicative Language Idealized Computing Engine) reduction machine (Darlington and Reeve, 1983) is being developed at Imperial College, London. ALICE is a parallel graph reduction machine based on a switched network organization. The ALICE project aims to provide efficient parallel evaluation both of Functional, Logic, and combined Functional-Logic languages. Another design aim is that construction of the machine should be possible using only a very limited range of VLSI components.

The ALICE machine organization consists of 4 types of unit, as illustrated by Figure 2. These are: a distribution system for processible (and vacant) packet identifiers, the Processing Agents, an Interconnection Network, and the Packet Pool Segments (i.e. memories).

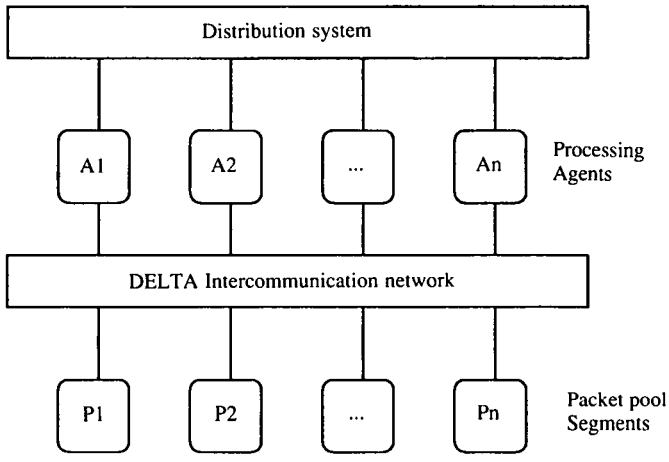

Figure 2 IC ALICE Machine Organization

Conceptually ALICE can be viewed as a pool of packets (P_i) and a collection of processing agents (A_i). A processing agent (A_i) executes the following sequence of tasks (cf. agent farming decomposition):

- Remove some processible packet from the pool i.e. a packet whose function field contains a rewritable packet and whose status is processible.
- Decide whether it is reducible i.e. whether the arguments are constant or themselves require reduction.
- Determine the appropriate rewrite rule.
- Generate the packets corresponding to the "rewrite" and place them in the pool.

Two distinct types of access are made to the *packet pool*, the first is directed at a packet with a specific identifier, for example to send a control signal, while the second is an access to find any processable or vacant packet. These types of access are supported by separate mechanisms in the physical architecture. The first mechanism is based on a random access memory in which a global address corresponds directly to a packet's identifier. The second mechanism is based on the circulation of packets around two slotted communications rings containing processible and vacant packets.

The set of processible packet identifiers and the set of vacant packet memory locations are distributed independently around the Processing Units. Each processing unit has unique instantaneous access to the contents of each set of identifiers. Thus, for example, when a processing unit requires to generate a new packet it would simply use the memory location whose address currently appears in its window on the set of vacant packet identifiers.

ALICE represents the graph of a program as a collection of packets, each packet specifying one node of the graph. As shown below a packet has three primary and three secondary fields:

← Primary →			← Secondary →		
ID	Function	Arg-list	Status	Ref-count	Signal-list

Primary fields contain the information required to represent a node, while secondary fields contain control information required by the evaluation mechanism. Fields have the following functions: *ID*—the unique identifier used to reference the packet; *FUNCTION*—the task associated with the node; *ARGUMENT-LIST*—the references and values defining the arguments of the node; *STATUS*—defines the state of the packet (e.g. processible); *REFERENCE-COUNT*—used to implement reference count garbage collection; and *SIGNAL-LIST*—the identifiers of packets to be signalled when the node produces its result.

Program execution proceeds by progressively rewriting expressions, using the appropriate recursion equation as a *rewrite rule*. Execution may be viewed as being initiated by "demand" tokens and resumed by "control" tokens which are wake-up signals indicating when a result is available. When a demand token is received, the packet corresponding to the node will be in one of three basic states: needs executing—in which case the demanding node's identifier is appended to the signal list and the demanded node is "forced"; already executing—in which case the identifier is merely appended to the signal list; and lastly already reduced—in which case the demanding node can be signalled immediately. This kind of *demand-driven* (Treleaven et al., 1982) execution is a typical example of farming parallelism. In ALICE, like in most reduction machines, the granularity of parallelism is fairly fine since "demand" tokens must be propagated to the leaves of the reduction tree which comprise the primitive or constructor functions of the language. Thus, the absence of coarse grain processing presents the limiting factor in obtaining parallelism speed-ups proportional to the number of processors used.

2.3 *Other projects*

Research into reduction machines has been going on since the late 70s. This initial work produced some interesting examples of serial reduction machines most of which have been claimed to execute their corresponding software at speeds comparable to those of general purpose mini-computers. Table 2 below shows some example systems.

Table 2

Computer	*Processors*	*Parallelism*	*Connectivity*	*Software*	*References*
IC ALICE	4-128 Transputers	—	Switched	HOPE	(Darlington and Reeve, 1983)
UCL GRIP	100 MC6800	5m red/s	Bus	Miranda	(Peyton-Jones et al., 1985)
REDIFLOW	1/more Xputers	medium grain	Switched	Lisp	(Keller et al., 1984)
NORMA	4-per node	0.2m red/s	Bus	SASL	(Richards, 1985)
GMD Reduction	serial	—	Internal bus	FP	(Keller et al., 1978)
Cambridge SKIM	serial	—	Internal bus	LISP	(Clarke et al., 1980)

Research to construct parallel reduction machines has intensified since the mid 80s. Most parallel reduction machines currently under construction are basically medium scale (i.e. 1–100 processors) networks composed of traditional control flow devices like the Transputer (e.g. ALICE), or the Motorola 68000 series (e.g. UCL GRIP). These control flow devices are often highly "microprogrammed" to emulate the behaviour of a graph or combinator reduction machine.

3 Logic languages and inference machines

Logic programming languages trace their roots to a branch of mathematics known as predicate calculus (van Emden and Kowalski, 1979). Most logic languages currently in use are based on the Horn-Clause (Kowalski, 1978) subset of first-order predicate calculus, in which an inference usually consists of a number of assumptions that are claimed to imply a certain conclusion. The basic construct in a logic language is the *predicate*. A predicate in Horn Clause logic has the form:

$$A_0 :- B_{00}, \ldots, B_{0m}$$
$$\vdots$$
$$A_n :- B_{n0}, \ldots, B_{nm}.$$

The clauses A_0 through to A_n represent a set of alternative conclusions where clause A_i is true if all conditions B_{i0} through to B_{im} are true.

Underlying most Logic language implementations there is a restricted theorem prover for first-order predicate calculus. The most common mechanism for this purpose is provided by resolution

although other mechanisms such as the logical connection method are finding increasing use (Bibel and Buchberger, 1985).

3.1 *Computational model*

The predominant Logic programming language is PROLOG (Clocksin and Mellish, 1981). To illustrate PROLOG, consider a programming example to *append* two lists, as expressed in two Prolog clauses:

```
append([ ], L,L)                                    ; C1
append([H | L1], L2, [H | L3]) :- append(L1, L2, L3).  ; C2.
```

There are two interpretations of these clauses, procedural and declarative. Procedurally, clause C_1 states that the result of appending the empty list (i.e. "[]") to any list L is L. The alternative Clause C_2 states that to append the list with head H and tail L1 to list L2, then construct a list with head H and tail L3, where list L3 is constructed by appending L1 to L2. When a call to append is received, PROLOG attempts to match the call with C_1. Failure to match causes *backtracking*. That is, any binding of variables produced by matching to C_1 is relinquished and the underlying interpreter attempts to match the call with the (next) alternative clause C_2.

Declaratively, the clauses can be viewed as the set of relations defined by *append* such that C_1 is a *fact* (e.g. L is L appended to [ ]) and C_2 is a *rule* (e.g. [H | HL] is L2 appended to [H | L1] if L3 is L2 appended to L1). Since the predicate is expressed as a relation it is possible to use it in different ways.

Underlying most PROLOG interpreters is an inference machine which is usually based on the interpretation mechanism of the Edinburgh University DEC-10 PROLOG due to Warren (Warren, 1977). This stack-oriented mechanism is a practical way of implementing the important PROLOG operations, namely clause invocation, unification and backtracking. In such interpreters, excution is performed by using three independent stacks: local, global, and trail stacks. Figure 3 shows the execution environment. The object codes are loaded into a heap area. The interpreter fetches both the caller and callee codes and executes unification between them by creating environments in the three stacks. The environment for each clause is represented by a local frame and a global frame plus some trail entries.

The major writable areas of store are the three stacks shown in Figure 3. The information below the tops of the local and global stacks is randomly accessible while the trail stack is maintained as a conventional push-down list. The global stack contains the value cells or global variables; variables which are arguments of structured data. Since structured data is shared among clauses, storage occupied by this type of variable cannot be reclaimed until execution fails or backtracking occurs. The local stack determines the actions to be taken on the success or failure of a goal and maintains

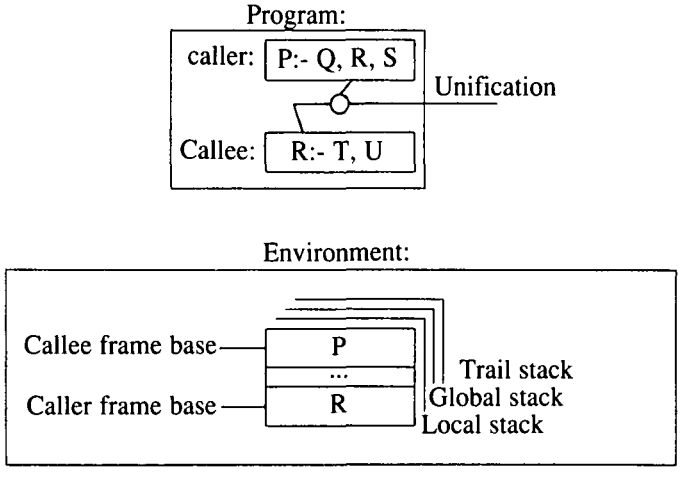

Figure 3 PROLOG Execution Environment

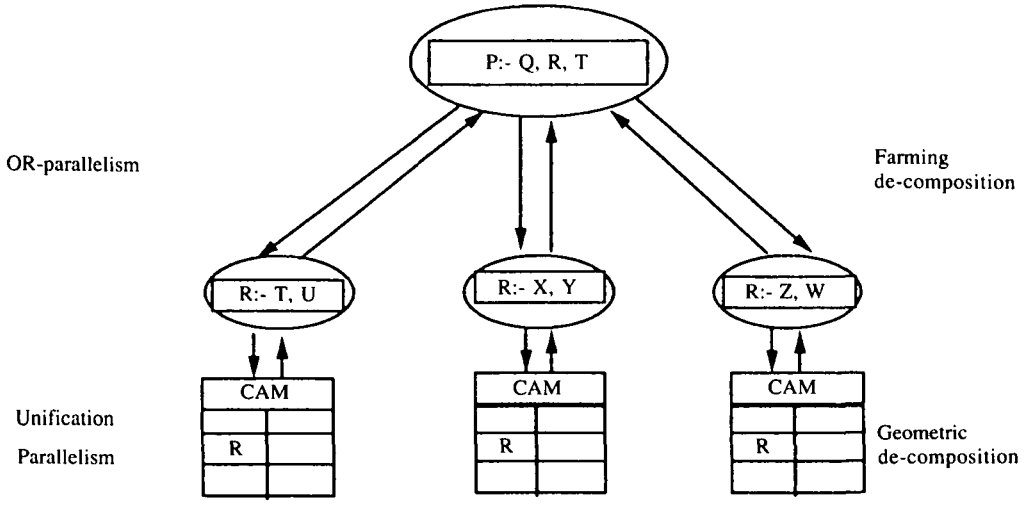

Figure 4 Farming and Geometric Decomposition of Prolog Programs

additional control information. Lastly, the trail stack is used to maintain the binding of variables, which may have to be reset on backtracking. For most of the time, the PROLOG interpreter is in the process of trying to unify the head of some clause against an existing goal.

Parallelism in the execution of logic programs can be obtained in three broad areas: (a) *OR*-parallelism—all alternative clauses satisfying a goal can be evaluated in parallel; (b) *AND*-parallelism—all the subgoals of a selected clause can be evaluated in parallel; and (c) *Unification*-parallelism—all arguments of a goal and the corresponding arguments of a selected clause can be matched in parallel by the unification algorithm.

As shown in Figure 4 above, OR-parallelism is another example suitable for farming de-composition. There are three reasons for this: firstly, alternative clauses can be evaluated independently; secondly, there is relatively little communication between the processing elements; lastly, each alternative clause involves coarse grain parallelism since it essentially invokes another copy of the interpreter. Likewise, Unification-parallelism can be exploited within a single processing element either by using associative memories (cf. geometric de-composition) for speeding up the search of the data base, or by using more complex techniques to pipeline the process of unification (cf. algorithmic parallelism). In contrast, although AND-parallelism can, in principle, be treated by a processor farm there is a requirement for high synchronization between processing nodes that use shared (logical) variables.

3.2 *Fujitsu Kabu-wake*

KABU-WAKE is a novel inference machine developed at the Fujitsu Laboratories in the framework of the Japanese FGCS Programme. Essentially, the KABU-WAKE method is based on sequential logic processing and OR-parallelism. The effectiveness of the method is currently undergoing testing on an experimental machine developed at Fujitsu Laboratories. This description of KABU-WAKE is based on (Sohma et al., 1985).

The experimental KABU-WAKE machine, as shown in Figure 5, comprises a network of 16 processing elements (one of which is dedicated to I/O) interconnected to form two networks: the CONTROL network and DATA network.

Each *Processing Element* (PE) is based on the Motorola 68010 and supports a parallel inference interpreter for the KABU-WAKE method. When a PE receives a request for a job from an idle PE, the PE divides its current task and dispatches one half of it to the requesting PE. Each PE is capable of 1 KLIPS, which is comparable to C-PROLOG running on the VAX 11/780.

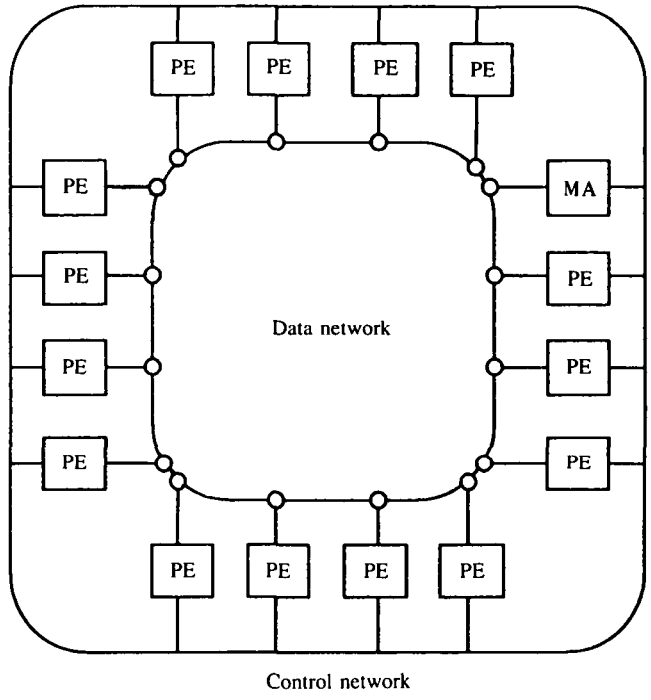

Figure 5 KABU-WAKE Machine organization

The CONTROL network is used to route information packets concerning the execution status of the PEs. As these packets go by, they can be examined by busy PEs. Once a busy PE has detected an idle PE, the busy PE may farm-out half of its workload (search tree) to the idle PE. The DATA network is used to transfer this split-job, referred to as KABU. When splitting a search tree, the involved variables are changed to a status as if all the subsidiary processes before the split position had failed and backtracked. Binding and unbinding of variables is based on tagging them with a unique time stamp. As new subgoals are created, new level numbers are assigned corresponding to the depth of the subgoal. As the search tree is split, a simple comparison of the level of the subgoal and that of the variable determines whether the variable must be unbound or not. Transferred branches of the search tree are also tagged by a so-called "rule-number" to indicate the position from which to start execution. Finally, KABU-WAKE uses a special-purpose "request reception" flag to indicate whether a search tree is splittable.

Parallel program execution is based on all 16 PEs having a distinct copy of all the PROLOG clauses. Each individual PE processes a search-tree sequentially using a depth-first search. The executing PE may release the OR-branches of its search tree to other PEs when requested by an idle PE. When a PE is requested to split its search tree, the PE does so at the branch closest to the root of the tree and thus, half the search tree is dispatched. More requests propagate the split in the search-tree into smaller and smaller branches. Eventually, the leaf nodes of the tree are evaluated and propagated backwards. During this demand-driven evaluation process there are two distinct forms of de-composition: farming out parallelism by the internal nodes and unification (e.g. geometric) parallelism both by the internal and the leaf nodes. Although KABU-WAKE uses only farming parallelism it is easy to see that there is a potential for combining the other two forms of parallelism in the evaluation of PROLOG. The reported advantages (Sohma et al., 1985) are typical of inference machines which restrict the grain of parallelism. They include: (a) low parallelism overhead for individual PEs which are capable of executing at the same speed as for sequential inference; (b) minimizable communication between PEs; and (c) there is no danger of an unexpected increase in the number of OR processes since this is limited by the number of PEs. The KABU-WAKE inference machine illustrates an increasingly accepted rationale for constructing small scale MIMD computers for logical inference in which individual processing nodes incorporate powerful SIMD components like CAMs and pipelines (e.g. SRI pipelined Prolog machine (Tick and Warren, 1984)).

3.3 *Other projects*

Several other projects to develop parallel inference machines are underway in a number of research Laboratories mainly in Japan and Europe. Table 3 below shows some example systems.

Table 3

Computer	*Processors*	*Parallelism*	*Connectivity*	*Software*	*Reference*
Tokyo Univ. PIE	16 MC68000	OR-Processes	Switched	Prolog	(Moto-Oka et al., 1984)
ICOT PIM-R	8–16	OR, AND	Switched	Conc. Prolog	(Onai et al., 1984)
BULL Multi-SCHUSS	256	unification	Bus	Prolog	(Gonzales-Rubio and Rohmer, 1985)
Logical Connection	8 MC68000	connection method	L-switch	Prolog	(Bibel and Buchberger, 1985)
SRI-Pipelined Prolog	Pipeline	unification	—	Prolog	(Tick and Warren, 1984)
Kyoto Univ. Prolog	—	OR, AND	Bus	Prolog	(Ohbuchi, 1985)

With the exception of a few examples, again most parallel inference machines currently under construction are basically small to medium scale (i.e. 1–256 processors) networks composed of traditional control flow devices. Most of these networks are basically used as prototypes to investigate future implementation in VLSI, and there are examples of all three kinds of decomposition to exploit parallelism.

4 Rule-based languages and production machines

Rule-based languages have been extensively used for a variety of tasks including research in artificial intelligence, cognitive psychology, and learning systems. Rule-based languages centre on large empirical knowledge-bases from which assertions can be drawn (Brownston et al., 1985). Such empirical knowledge-bases are constructed from rules of the (general) form:

R1: If Conception(C) and Causal-Knowledge(K)
  Then Action(A) and Assertion(S)
R2: . . .

These rules may be used in two ways, either reasoning *forwards* from a condition to an action, or reasoning *backwards* by hypothesizing an action and using the rules to verify the condition.

4.1 *Computational model*

The most popular of rule-based languages is OPS5 (Forgy, 1981, 1984), developed at Carnegie-Mellon University. The design of OPS5 centres on three main components namely, *Production* memory—which is composed of a set of IF–THEN rules called Productions, *Working* memory—which is a database of assertions about a specific domain, and the *Recognise-Act* interpreter—the OPS5 control mechanism.

Each Production consists of a conjunction of conditions corresponding to the "IF" part of the rule and a set of actions corresponding to the "THEN" part of the rule. The actions associated with a production operate on the Working Memory changing its state. The most important action types are: "MAKE"—creates and adds one new element; "REMOVE"—deletes one or more elements; "MODIFY"—changes one or more sub-elements of an existing element; and "WRITE"—displays information on the user terminal. OPS5 rules are of the form:

(P ⟨rule-name⟩ $(C_0) \ldots (C_n)$
  $\rightarrow (A_0) \ldots (A_m)$)

with C_i being the conditions and A_i the actions of the rule. In general, the meaning of an OPS5 production rule is: when the state of Working Memory is such that conditions C_0 through to C_n are all true, the actions A_0 through to A_m are to be executed.

Working memory contains two kinds of elements, namely vectors of symbols e.g. (TASK START), and elements with associated attribute-value pairs e.g. (COMPUTER ~NAME VAX); an attribute being preceded by a tilde.

Figure 6 shows an OPS5 production named *append*. In this Figure, Working Memory contains a task to append the lists LIST1, and LIST2 to LIST3 and the two lists named LIST1 and LIST2 with elements (a b c) and (1 2 3) respectively.

The single production rule in Figure 6 states that when Working Memory is such that there is: a task to append Lists L1 and L2 to form List L3; and a list L1 say named HEAD, and a list L2 say named TAIL; then (→) make a new list L3 by firstly copying (SUBSTR) into L3 all the elements of HEAD and then copying all the elements of TAIL.

The OPS5 control mechanism is a so-called Recognize-Act Interpreter, executing a production-system program by performing the following operations in a simple loop:

> MATCH—determine which rules have satisfied conditions, and construct the "conflict set"
> CONFLICT RESOLUTION—select one rule that is satisfied, if none satisfied halt
> ACT—perform the actions of the selected rule.

Thus, for the rule *append* above, the Recognize-Act Interpreter will initially match the conditions C_1, C_2, C_3 against its Working Memory elements, and bind the variables ⟨L1⟩, ⟨L2⟩ and ⟨L3⟩ to the patterns they match (e.g. LIST1, LIST2 and LIST3). For a successful match the Recognize-Act Interpreter will perform the action A_1 (i.e. MAKE) which creates the new List L3 in Working Memory.

Regarding the OPS5 Interpreter's performance, by far the most important operation is the MATCH. In all OPS Interpreters MATCHing is based on the Rete match algorithm (Forgy, 1982). Using the Rete algorithm, the Interpreter represents the left-hand sides of the productions as a dataflow graph whose input consists of changes in Working Memory and whose output consists of changes in the conflict set. These changes are encoded in data structures called tokens. As tokens flow through the graph, the graph nodes perform the necessary operations which are essentially tests and which may result in new tokens being generated. In the graph there are four kinds of nodes (i.e. constant test, memory, two-input and terminal nodes) that perform two types of tests (i.e. intra-condition and inter-condition tests). The four types of node are:

- Constant-test nodes: These nodes test constant features of Working Memory elements.
- Memory nodes: These nodes maintain the matcher's state. They store lists of tokens that match individual conditions or groups of conditions.

```
Working Memory:
        (TASK APPEND-LIST LIST1 LIST2 LIST3)
        (LIST LIST1 a b c)
        (LIST LIST2 1 2 3)
Production Memory:
        (P APPEND
        (TASK APPEND-LIST ⟨L1⟩ ⟨L2⟩ ⟨L3⟩)  ; C1
        {(LIST ⟨L1⟩) ⟨HEAD⟩}               ; C2
        {(LIST ⟨L2⟩) ⟨TAIL⟩}               ; C3
        →
        (MAKE LIST ⟨L3⟩ (SUBSTR ⟨HEAD⟩ 3 INF)
                (SUBSTR ⟨TAIL⟩ 3 INF)))    ; A1
```

Figure 6 OPS5 Production *append*

- Two-input nodes: These nodes access the information stored by the memory nodes to determine whether groups of conditions are satisfied.
- Terminal nodes: Terminal nodes are concerned with changes to the conflict set.

The two types of tests are:

- Intra-condition tests: These tests involve single working memory elements and serve the purpose of identifying various features of a single Working Memory element.
- Inter-condition tests: These tests refer to multiple condition elements.

To illustrate such features consider the following OPS5 pattern:

(EXPRESSION ^Name ⟨N⟩ ^ARG1 0 ^OP + ^ARG2 ⟨X⟩),

When the pattern matcher processes this pattern it looks for Working Memory elements which have the following features:

- The class of the element must be EXPRESSION
- The value of the ARG1 must be the number "0"
- The value of the OP must be the atom "+"

Features are identified by linear sequences of special purpose nodes in the network. The other class of tests, the inter-condition tests, results from having a *variable* occur in more than one condition in the rule. This type of test is analogous to binding values to variables. Such tests are performed by the two-input nodes that join two paths in the network into one, and so on. Finally a special terminal node is used to represent the Production.

4.2 *Columbia University DADO*

DADO is a parallel, tree structural machine designed at Columbia University with the aim of providing significant performance improvement in the execution of Rule-Based programming languages. When fully developed, DADO will comprize a very large number of primitive processing elements (in the order of 100,000). This description of DADO is primarily based on (Stolfo and Shaw, 1982).

The DADO machine organization is configured as a binary tree, as shown in Figure 7. In the prototype DADO architecture, each node in the tree is a primitive Processing Element (PE) comprizing processor, small local memory, and so-called I/O switch. In the DADO-2 prototype, the *Processor* is an Intel 8751, accompanied with 4K of EPROM, and 16K bytes of RAM. In addition, there is a specialised I/O switch, and three 8-bit wide links. In the binary tree topology of DADO, two of the links are used to connect a PE to its descendant PEs, while the third link connects the PE to its ancestor.

PEs in the DADO machine are capable of operating either in Single-Instruction-Multi-Data stream (SIMD) or Multi-Instruction-Multi-Data stream (MIMD) mode. In SIMD mode, the same instruction is broadcast to all PEs from the ancestor PE, and all receiver PEs execute the instruction

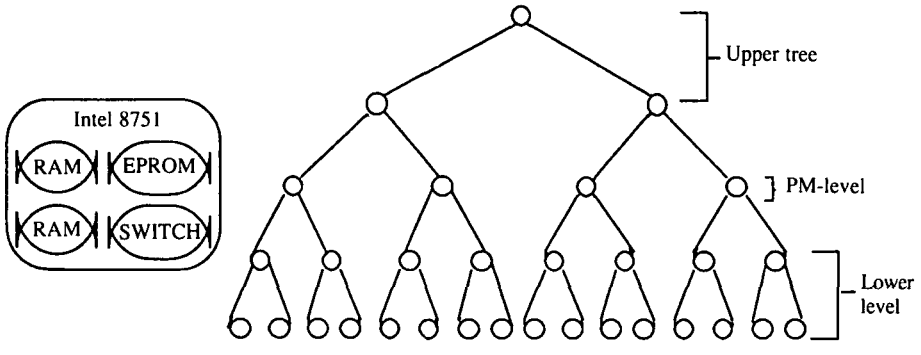

Figure 7 The DADO Tree Machine Organization

with different data elements. In MIMD mode, PEs and their descendents, are virtually disconnected from ancestor PEs and operate autonomously. However, subtrees of PEs, within the DADO structure, may be allowed to operate in SIMD mode without the entire structure doing so.

Programs for the DADO machine are organised in a way that reflects the OPS5 notion of Working Memory and Production Memory. For the execution of Rule based languages the machine is configured as a three-level tree: the *Upper portion* of the tree, the *Production Memory (PM)-level* and the *Lower portion* of the tree (comprizing all PEs below the PM-level). The *Upper portion* is used to select one of the productions to be executed and to broadcast the actions resulting from the execution. The *Production Memory* PM-level stores the productions, with each PE storing a single production. The *Lower portion* of the tree stores the working Memory elements that are so-called "relevant" to the production rule in the ancestor PM-level PE. This division is illustrated in Figure 7.

The system execution cycle consists of three phases: MATCH, SELECT (i.e. conflict resolution) and ACT. In the DADO machine these phases are implemented as follows:

MATCH—All PEs at the PM-level are set to MIMD mode and simultaneously match their contents against the contents of their respective Working Memory subtrees. As a result each PE computes a so-called "priority" rating.

SELECT—All PEs enter SIMD mode and are instructed to broadcast their "priority" ratings to their parents. Parent PEs discard smaller priority ratings and broadcast, up the tree, a tag identifying higher priority siblings (together with the winner priority rating) so that a single "winner" reaches the root.

ACT—The PE with the highest priority is instructed to instantiate and return its right-hand side which is then broadcast to all PEs at the PM-level. These PEs are then set to MIMD mode and are instructed to update their Working Memory subtrees according to the actions of the selected rule.

DADO is a tightly coupled system which attempts to exploit both geometric and algorithmic parallelism. Geometric parallelism is exploited during the MATCH phase of execution, whereas algorithmic parallelism is exploited during the SELECT phase of execution. The limiting factor in the construction of large-scale Production machines is not so much the communications overheads (which nevertheless are considerable) as the fact that the parallelism inherent to Rule-Based systems themselves is restricted (Forgy et al., 1984).

4.3 *Other projects*

Several other projects to develop parallel production machines are underway in a number of research Laboratories mainly in the United States. Table 4 below shows some example systems.

Table 4

Computer	*Processors*	*Parallelism*	*Connectivity*	*Software*	*References*
Columbia DADO	1K MC68000	Geometric	Tree	OPS5	(Stolfo and Shaw, 1982)
Fairchild FAIM-1	—	100 * VAX780	Mesh	OPS5	(Davis and Robison, 1985)
Maryland ZMOB	256 Z80	Professor farm	Bus	Demons	(Rieger, 1981)
BBN BUTTERFLY	56 MC68000	Geometric	Banyan Switch	C	(BBN, 1985)
Columbia NON-VON	4-32	O(2)	Tree	OPS5	(Hillyer and Shaw, 1983)

With the exception of the Columbia University DADO most parallel production machines currently under construction are basically small to medium scale networks.

5 Connectionism and self-organizing networks

Within the area of knowledge processing Connectionism represents an alternative approach to symbol manipulation. Connectionist models are based on the hypothesis that "thinking" emerges

from the neural connections in the brain forming and re-forming, and self-organizing networks attempt to emulate the way in which neurons perform cognition and other knowledge processing computations. Based on this model and the theory of Cellular Automata, connectionism explores diverse paths in machine design.

5.1 *Computational model*

The establishment, strength, and reciprocal feedback of interneural connections can be taken as a model of knowledge processing in the brain. For instance (Feldman, 1985) uses the example in Figure 8 to illustrate the connections in the various elements of the sentence *Bob threw a ball.*

In a connectionist network the sentence may be taken to mean either "Bob *propelled* a *sphere*" or "Bob *sponsored* a *dance* (for charity)". To disambiguate the sentence, the various connections are formed and reformed, according to the context, until a consistent (sometimes called stable) state is reached. In principle, it is possible to partition the network and distribute it across an ensemble of processors that are responsible for performing the computation required to re-arrange these connections. Such computer networks are generally referred to as self-organizing networks.

Four main hypotheses, applied to computer design, set self-organizing network machines apart from other knowledge processing architectures. Firstly, in self-organizing network machines the hardware structure is pre-eminent, since high-level processing cannot be abstracted from the hardware that performs it. Secondly, self-organizing networks are "massively" parallel with designers considering 10,000–1,000,000 processors working in parallel. It must be noted however, that as important a factor is the number of connections per processor and not merely the number of processors. Thirdly, self-organizing networks provide the possibility of solutions (though not necessarily optimal ones) to NP-hard problems. Lastly, and probably more important, self-organizing networks are largely unprogrammed. Only general instructions are given and the network finds its solution by setting into a stable state rather than by following the explicit instructions of an algorithm.

5.2 *CMU Boltzmann machine*

Even in the esoteric world of Connectionist machines, the Boltzmann Machine research at Carnegie-Mellon University (CMU) represents a novel approach. It consists of massively parallel networks of simple processing elements, called units, working on large constraint satisfaction searches. This description is primarily based on (Fahlman *et al.*, 1983).

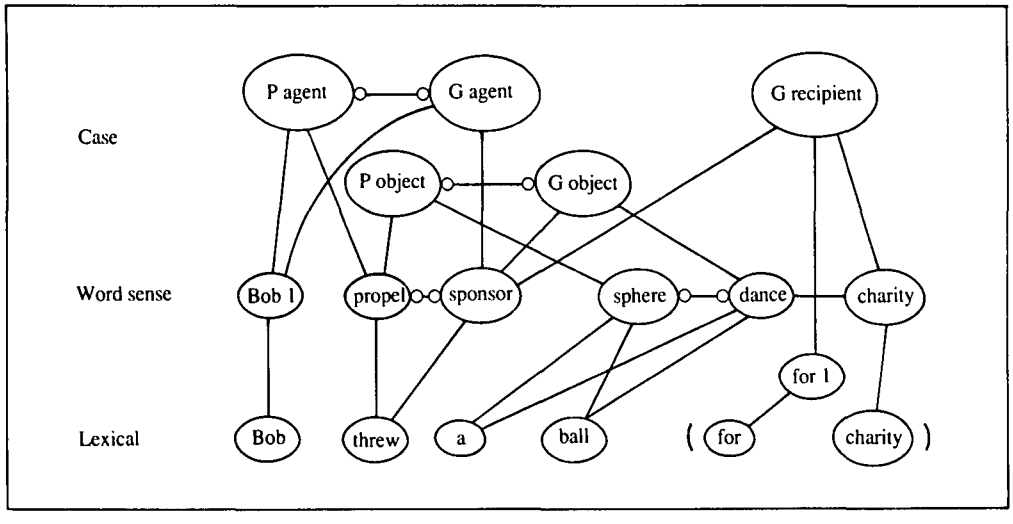

Figure 8 Connectionists model to understand the sentence "Bob threw a ball"

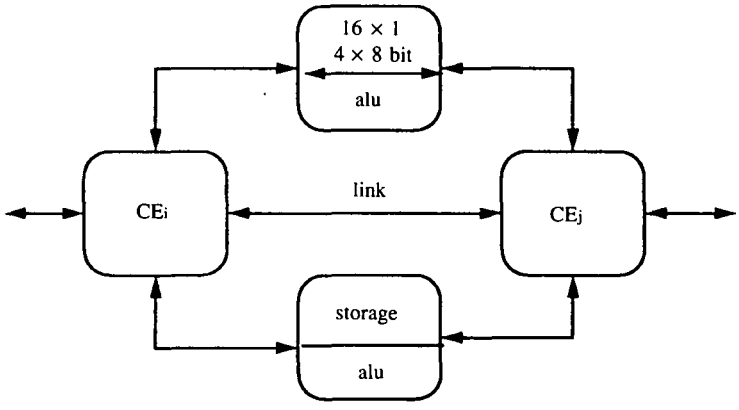

Figure 9 CMU Boltzmann Machine units

A Bolzmann unit, as shown in Figure 9, is logically a primitive computing element connected to other such units with bidirectional links. With each such link there is an associated weight signifying the strength of the interelation. The links themselves represent a so called "weak" pairwise constraint between the hypotheses. A positive value (weight) for the link L(i, j) indicates that the two hypotheses support each other, whereas a negative weight suggest that both hypotheses should not be accepted together.

Researchers at CMU have done preliminary design work on a Boltzmann machine, known as Thistle (Fahlman et al., 1963), combining two fundamental methods of communication of signals, namely "marker-passing" and "value-passing". In Thistle, each element or connection is used to represent local knowledge about a concept or assertion. Nodes are connected to any number of links and this node-link pattern forms the system's long-term memory and also some body of knowledge (as in semantic networks). Each primitive Computing Element (CE) has local storage for 16 single-bit markers and four 8-bit values. These values can be added, multiplied, scaled, compared to one another and passed from one to another, perhaps gated by various markers and multiplied by the weight associated with the link.

A Boltzmann unit, as shown in Figure 9, is either ON or OFF. Its state (ON/OFF) is a probabilistic function of all neighbouring units and their associated weight. The resulting structure is a machine that is in a certain state at any instance. This state can be assigned a value which is the sum of the weights of all the "ON" units in the system plus a threshold. This value is termed the *energy* of the system at that particular state. By supplying the system with a set of hypotheses some units are externally forced into particular states representing these hypotheses, and thus giving the system a certain "energy". The idea is to obtain the minimum such "energy" which is clearly obtainable by altering the weight of some units. Once the energy of the system is established it is interpreted as the extent to which these new hypotheses violate the built-in knowledge of the system. Once the *minimum energy* is established (by now the internal weights of the links have been changed) the best interpretation of the hypotheses is achieved. These hypotheses may now be accepted and perhaps become knowledge themselves. This process is performed by the learning algorithm. In general, the learning algorithm performs two tasks:

- Modeling the underlying environment.
- Controlling the learning of the machine.

To model the underlying environment a partition of two functional groups of units is used: a set of *visible* units (that are locked into specific states by the environment) and a set of *hidden* units whose states are used to represent more complex hypotheses about the states of the visible units. Then the modeling problem reduces to the problem of minimizing the discrepancies between the system's internal model and the environment. An information-theoretic measure of such discrepancies is given by the summation (G) of the probabilities of visible and hidden units over all states. Minimizing G is

then a process of changing the weights of the hidden units. Simple techniques (from physical models) are available to achieve this efficiently.

The other task of the learning algorithm is twofold. The first is to reduce the effect of the "noise" introduced by *estimating* the average probabilities in the visible and hidden units. This task can be achieved by collecting statistics over a longer period. The second last is to resolve confusion when an environment specifies that only a small subset of possible patterns over the visible units ever occur and hence the unused patterns must have probability zero which the Boltzmann machine cannot handle in finite space. A solution is engineered around the problem by manipulating the correct patterns so that a small noise is introduced. If the noise is sufficiently small every pattern will have *some* probability of occurring but the correct ones will dominate the statistics.

By far the most important operation in the Boltzmann machine is the computation required to minimise the energy over the network. The algorithm for performing this computation is based on a model from statistical mechanics which exhibits natural parallelism and can be de-composed geometrically. Each computing element (CE) in the network is required to perform moderate amounts of numeric computation and to communicate with near neighbours. However, due to the symmetry of the de-composed data structure (e.g. grid) self-organizing networks like the Boltzmann network offer parallelism speed-ups which are fairly close to being proportional to the number of processors used. There are two limiting factors in the construction of large scale (i.e. hundreds of thousands of nodes) Boltzmann-like self-organizing networks: firstly, in order to obtain maximum speed each individual CE must be a relatively fast (and thus expensive) number-cruncher; and secondly the performance of the learning algorithms degrades rapidly with the addition of further knowledge.

6 Conclusions and future trends

Research for the construction of parallel computer systems to support the application domain of knowledge processing is following two contrasting approaches. The *symbol* manipulation approach attempts to develop computer architectures suitable for the execution of Very High Level symbolic processing languages such as Lisp, Prolog, and OPS5. These parallel computers tend to be constructed of highly microprogrammed traditional control flow devices. The trend is towards parallel computer networks composed of moderate numbers of identical units and a regular interconnection network between these units. Thus, these architectures are believed to be suitable for implementation in VLSI and Waferscale technology. The best current example of such identical units is the INMOS TRANSPUTER microcomputer (Barron et al., 1983). However, devices like the TRANSPUTER provide only rudimentary support for symbolic processing in terms of their ability to manipulate structured data and their support for associative processing (cf. unification). What we can expect to see in the future are TRANSPUTER-like devices enhanced with mechanisms for supporting symbolic processing such as structured memory and associative components, for example.

Regarding parallelism in symbolic processing computers, we have seen that the strategy of farming de-composition is widely used by reduction and inference machines. However, for farming parallelism to be worth exploiting, it is important to constrain its grain to manageable levels. *Reduction* machines such as the IC ALICE are managing to achieve this, to some degree, by permitting procedural operations in the machine (e.g. direct assignment of values to packets). *Inference* machines such as the Fujitsu KABU-WAKE have been successful in restricting farming parallelism to manageable levels by limiting farming de-composition exclusively to OR-processes. In addition, inference machines have a potential for exploiting parallelism by geometric de-composition during unification. Likewise, the exploitation of parallelism in *Production* machines is based on a combination of the latter two forms of de-composition, namely geometric and algorithmic.

The *Connectionist* approach to the construction of parallel knowledge processing systems follows a bottom-up path, in which the computer network is largely unprogrammed. These *self-organizing*

networks like the CMU Boltzmann machine tend to use geometric de-composition when attempting to exploit parallelism and appear to offer potential speed ups that are close to being proportional to the number of processors used.

References

Functional Languages and Reduction Machines

Abenson, H and Sussman, GJ, 1984. *Structure and Interpretation of Computer Programs*, Massachusetts: The MIT Press

Church, A, 1941. *The Calculi of Lambda-Conversion*, New Jersey: Princeton University Press

Clarke, TIW et al., 1980. "SKIM—The S, K, I Reduction Machine", Proc. 1980 LISP Conf. Stanford

Darlington, J and Reeve, M, 1983. "ALICE and the Parallel Evaluation of Logic Programs", 10th Int. Symp. on Computer Architecture

Keller, RM et al., 1978. "A loosely coupled applicative multiprocessing system", Proc. Nat. Computer Conf., AFIPS Press, pp 861–870

Keller, RM et al., 1984. "REDIFLOW Multiprocessing", Proc. IEEE COMPCON Spring 84, pp 410–414

Kluge, WE, 1979. "The architecture of a reduction language machine hardware model", Tech. Rep. ISF-Rep. 79.03, Gesellschaft fur Mathematik und Datenverarbeitung MBH Bonn.

Kluge, W and Schmittgen, C, 1987. "Reduction Languages and Reduction Systems", In: *Future Parallel Computers*, Lecture Note in Computer Science, vol. 272, Treleaven, P and Vanneschi, M, eds, pp 153–182

Moon, D, 1985. "Architecture of the Symbolics 3600", Proc. Twelth. Int. Symp. on Computer Architecture, pp 76–83

Peyton-Jones, S et al., 1985. "GRIP—a Parallel Graph Reduction Machine", University College London, Dept. of Computer Science, Technical Report 1665

Richards, H, 1985. "An Overview of the Burroughs NORMA", Internal Report Burroughs Corp., Austin Research Centre

Rieger, C et al., 1981. "ZMOB: A new computing engine for AI", Proc. 7th IJCAI-81, Vancouver, pp 955–960

Logic Languages and Inference Machines

Bibel, W and Buchberger, B, 1983. "Towards a Connection Machine for Logical Inference", Proc. Int. Sym. on Fifth Generation and Super Computers, Rotterdam, December 11–13, 1984, *Future Generations Computer Systems* **1** (3) pp 177–188

Clocksin, W and Mellish, C, 1981. *Programming in PROLOG*, New York: Springer-Verlag

van Emden, MH and Kowalski, RA, 1976. "The Semantics of Predicate Calculus as a Programming Language", *JACM* **23** (4) pp 733–742

Gonzales-Rubio, R and Rohmer, J, 1985. "From databases to Artificial Intelligence: A Hardware Point of view", Les Arcs, France: AIS, NATO

Kowalski, R, 1979. *Logic for Problem Solving*, Amsterdam: North-Holland

Moto-oka, T. et al., 1984. "The Architecture of a Parallel Engine PIE", Proc. Int. Con. on Fifth Generation Computer Systems, Tokyo, pp 479–488

Onai, R et al., 1984. "An Approach to a Parallel Inference Machine Based on Control-Driven and Data-Driven Mechanisms", Tech. Rep. TR-042, ICOT

Sohma, Y et al., 1985. "A New Parallel Inference Mechanism Based on Sequential Processing", Proc. IFIP TC-10 Work. Conf. on Fifth Gen. Comp. Arch., UMIST Manchester

Tick, E and Warren, DHD, 1984. "Towards a Pipelined Prolog-Processor", Proc. 1984 Int. Sym. on Logic Programming

Uchida, S et al., 1983. "Outline of a Personal Sequential Inference Machine: PSI", *New Generation Computing* **1** (1)

Warren, DHD, 1977. "IMPLEMENTING PROLOG—compiling predicate logic programs", Dept. of Artificial Intelligence, Univ. of Edinburgh, Research Report No. 39

Rule-Based languages and Production Machines

BBN, 1985. "BUTTERFLY Parallel Processor Overview", Tech. Rep. Bolt Beranek and Newman Labs. Inc., Cambridge, MA

Brownston, L et al., 1985. *Programming Expert Systems in OPS5*, New York: Addison-Wesley

Davis, AL and Robison, SV, 1985. "The Architecture of the FAIM-1 Symbolic Multiprocessing System", Proc. Int. Joint Conf. on Artificial Intelligence, pp 32–38

Forgy, CL, 1981. "The OPS5 user's Manual", Tech. Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University

Forgy, CL, 1982. "Rete: A Fast Algorithm for the Many Pattern/Many Object Match Problem", *Artificial Intelligence* **19** pp. 17–37

Forgy, C et al., 1984. "Initial Assessment of Architectures for Production Systems", Proc. Nat. Conf. for Artificial Intelligence AAAI '84, pp 116–120

Hayes-Roth, F et al., 1983. *Building Expert Systems*, New York: Addison-Wesley

Hillyer, BK and Shaw, DE, 1983. "Rapid Execution of AI Production Systems on the NON-VON Supercomputer", Tech. Rep., Dept. of Comp. Sc., Columbia Univ., New York

Stolfo, SJ and Shaw, DE, 1982. "DADO: A Tree-Structured Machine Architecture for Production Systems", Proc. Nat. Conf. on AI AAAI-82, pp 242–246

Connectionist Machines and Self-Organizing Networks

Fahlman, S, 1980. "Design sketch for a million-element NETL machine", Proc. AAAI-80, Stanford, pp 249–251

Fahlman, SE et al., 1983. "Massively Parallel Architectures for AI: NETL, Thistle and Boltzmann Machines", Proc. Nat. Conf. of AI, AAAI-83, pp 109–113

Feldman, JA, 1985. "CONNECTIONS", *Byte* **10** (4) pp 277–284

Hillis, WD, 1984. "The Connection Machine: A computer architecture based on cellular automata", *Cellular Automata*, Farmer, et al., ed, pp 213–229

Related Reviews

Ohbuchi, R, 1985. "Overview of Parallel Processing Research in Japan", IBM Japan Science Institute, JSI Research Report TR87-0003

Treleaven, PC et al., 1982. "Data Driven and Demand Driven Computer Architecture", *ACM Computing Surveys* **14** (1) pp 93–143

Parallelism and Communication

Barron, I et al., 1983. "Transputer does 5 or more MIPS even when not used in parallel", *Electronics* **56** (23) pp 109–115

May, D, Shepard, R and Keane, C, 1987. "Communicating sequential processes: Transputer and Occam", In: Treleaven, PC and Vaneschi, M, eds, *Future Parallel Computers*, New York: Springler Verlag