Chapter

4

Design Issues of Multiprocessors for Artificial Intelligence

Benjamin W. Wah

University of Illinois, Urbana, IL

Guo-jie Li

Academia Sinica, Beijing, People's Republic of China

4.1 Introduction

In recent years, artificial intelligence (AI) techniques have been widely used in various applications, such as natural-language understanding, computer vision, and robotics. As AI applications move from the laboratories to the real world and as AI software grows in complexity, the computational throughput and cost are increasingly important concerns. The conventional von Neumann computers are not suitable for AI applications because they were designed mainly for sequential and deterministic numeric computations. Extensive efforts have been devoted to investigate and develop efficient AI architectures [Wah 86]. This chapter provides a state-of-the-art assessment of AI-oriented systems and discusses the major issues involved in such designs. Chapter 7 discusses various architectural classes of AI machines with case studies.

4.1.1 Characteristics of AI computations

To develop a special-purpose computer to support AI applications, the requirements of these applications must be fully understood. Many

conventional numeric algorithms are well analyzed, and bounds on their computational performance have been established. In contrast, many AI applications are characterized by symbolic processing, nondeterministic computations, dynamic execution, large potential for parallel and distributed processing, management of extensive knowledge, and an open system.

Symbolic processing. Data are generally processed in symbolic form in AI applications. Primitive symbolic operations, such as comparison, selection, sorting, matching, logic set operations (union, intersection, and negation), contexts and partitions, transitive closure, and pattern retrieval and recognition, are frequently used. At a higher level, symbolic operations on patterns such as sentences, speech, graphics, and images may be needed.

Nondeterministic computations. Many AI algorithms are nondeterministic; that is, it is impossible to plan in advance the procedures to execute and to terminate with the available information. This is attributed to a lack of knowledge and a complete understanding of the problem; it may result in exhaustively enumerating all possibilities when the problem is solved or in a controlled search through a solution space.

Dynamic execution. With a lack of complete knowledge and anticipation of the solution process, the capabilities and features of existing data structures and functions may be defined and new data structures and functions created while the problem is actually being solved. Further, the maximum size for a given structure may be so large that it is impossible to allocate the necessary memory space ahead of time. As a result, when the problem is solved, memory space and other resources may have to be dynamically allocated and deallocated, tasks may be dynamically created, and communication topology may be dynamically changing.

Large potential for parallel and distributed processing. In parallel processing of deterministic algorithms, a set of necessary and independent tasks must be identified and processed concurrently. This class of parallelism is called AND-parallelism. In AI processing, the large degree of nondeterminism offers an additional source of parallel processing. Tasks at a nondeterministic decision point can be processed in parallel. This latter class is called OR-parallelism.

Knowledge management. Knowledge is an important component in reducing the complexity of solving a given problem; more *useful* knowledge means less exhaustive searching. However, many AI problems may have very high inherent complexity, hence the amount of useful knowledge may also be exceedingly large. Further, the knowledge acquired may be fuzzy, heuristic, and uncertain in nature. The representation, management, manipulation, and learning of knowledge are, therefore, important problems to be addressed.

Open system. In many AI applications, the knowledge needed to solve the problem may be incomplete because the source of the knowledge is unknown at the time the solution is devised, or the environment may be changing and cannot be anticipated at design time. AI systems should be designed with an open concept and allow continuous refinement and acquisition of new knowledge.

In general, there are two basic approaches to improving the computational efficiency of processing AI tasks: having heuristic knowledge to guide searches and using faster computers. In the following sections, these approaches are discussed.

4.1.2 Heuristic searches

The key performance-related feature of AI computations is their nondeterminism, which results from a lack of complete understanding of the solution process. In other words, when a problem becomes well understood and can be solved by a deterministic algorithm, we usually cease to consider it "intelligent," although the problem may still be symbolic [Simon 86].

The starting point of conventional computations is deterministic algorithms, whereas efficient deterministic algorithms to solve a given AI problem are a result of the knowledge accumulated and the gradual refinement of the computations. This involves the succinct choice of an appropriate knowledge-representation scheme, the learning mechanisms to acquire the related knowledge, and a suitable architecture to support the computations. Good heuristics designed from previous experience may allow a complex problem to be solved efficiently, even on a serial processor.

Since the mid-1960s, it has become apparent to the AI community that inference alone is often inadequate to solve real-life problems. To enhance the performance of AI algorithms, they must be augmented with knowledge and metaknowledge of the problem domain in addition to formal reasoning methods. *Metaknowledge* refers to the control information to guide the search. This realization gave birth to *knowledge engineering* and *knowledge-based systems*, the field of applied AI [Feigenbaum 83]. Since knowledge stored in any knowledge-based

system may be incomplete and inaccurate, combinatorial searches are still needed.

4.1.3 Faster technologies and parallel processing

An AI computer system must support both knowledge-based management and heuristic searches. Faster technologies and parallel processing are means to improve the computational efficiency. For many applications, such as natural-language understanding and computer vision, the current achievable performance is much lower than that needed. For example, according to the Defense Advanced Research Project Agency's (DARPA) strategic computing proposal, it was estimated that an equivalent of 1 trillion von Neumann computer operations per second were required to perform the vehicle-vision task at a level that would satisfy the long-range objective of the Autonomous Vehicle Project [Spectrum 83]. At best, current sequential computers of reasonable cost achieve processing rates below 100 million operations per second, which implies at least 10^4 times improvement in performance is required.

Newer technologies can help in designing faster computers. For example, using GaAs high-electron-mobility transistors (HEMTs), it was estimated that for a computer with over 500,000 gates operating at 77 K and 15 levels per pipeline stage, the cycles times would be 2.7 ns with 5 W and 3200 gates per chip, and 2.0 ns with 20 W and 5200 gates per chip, respectively [Amdahl 87]. In contrast, a liquid-cooled Cray 2 supercomputer built using emitter-coupled logic (ECL) technologies has 8 levels per pipeline stage, more than 500,000 gates, and operates at 300 K and 4.1 ns per cycle. The delay of one ECL gate level is approximately translated into 1.5 GaAs HEMT gate levels; hence, correcting the cycle time of the Cray 2 supercomputer into HEMT technologies and 15 levels results in a 5.1-ns cycle time for the Cray 2 computer. In short, there is a factor of 2 in using the newer technologies available today.

Another way to decrease the cycle time is to reduce the interconnect delay. It was estimated that with GaAs HEMTs operating at a 2-ns cycle time, the switching, fan-out, and interconnect delays were approximately 2 percent, 10.5 percent, and 87.5 percent of the cycle time, respectively [Amdahl 87]. Although superconductivity can be used to reduce the interconnect delays, it is less desirable with GaAs technologies, due to the high impedance in the gates, and more desirable with ECL technologies. When combined, these newer technologies available today may allow one to two orders of magnitude improvement in the cycle times of computers.

The trend in designing AI computers has been toward applying faster technologies and parallelism to process computation-intensive

AI tasks. Examples of parallel AI systems currently available or under research/development include ALICE, Aquarius, Butterfly, Concurrent Lisp machine, Connection Machine, DADO, FAIM-1, FFP, iPSC, Japanese Fifth-Generation Computer System (FGCS), NETL, NON-VON, Rediflow, SOAR, SPUR, and ZMOB [Wah 86]. Some of these computers, such as the Aquarius, Butterfly, iPSC, and ZMOB, were designed for both numeric and symbolic processing.

Recently, there is another trend to design small-grain massively parallel architectures for AI applications. These architectures are sometimes called *connectionist systems;* they are composed of a very large number of simple processing elements. Knowledge of a given entity in such systems is distributed on a number of processing elements and links, and each processor or link may be shared by multiple entities. The use of connections rather than memory cells as the principal means to store information leads to the name *connectionism* [Fahlman 87]. The resemblance to neurons in a brain also results in the term *neural networks*. Connectionist and neural systems are discussed in Chap. 16. Many computers can simulate connectionist systems. An example is the Connection Machine developed by Thinking Machines Inc., which can perform neural-network simulations two to three orders of magnitude faster than serial machines of comparable cost [Hillis 85, Waltz 87].

The high performance of many parallel AI computers is achieved through associative processing and *data-level parallelism*. This approach is suitable for operations on large databases, such as sorting, set operations, statistical analysis, and associative pattern matching. Yet data-level parallelism is not intelligent enough. For general AI applications involving heuristic searches, control-level parallelism should be involved. Unfortunately, early experience with multiprocessor architectures for Hearsay-II [Fennell 77], Eurisko [Lenat 84], OPS5 [Forgy 84], and others has led to a belief that parallel AI programs will not have a speedup of more than one order of magnitude. A possibly revolutionary approach to designing parallel languages and systems for AI processing may be needed.

One misconception in parallel processing is to use the total computing power of a parallel system to characterize the rate at which a given AI application is processed. Due to the nondeterminism in AI computations, a high computing power does not always imply a shorter completion time. Since most AI applications involve heuristic searches, resources may be devoted to fruitless searches, which use more computing power but do not help to decrease the time to find a solution. In fact, anomalies may happen such that increasing the degree of parallelism may even increase the completion time in nondeterministic searches [Wah 85, Li 87, Li 86*a*]. What is important is how to allocate resources so only useful tasks are performed. The

question of solving an AI problem in a parallel processing environment is still largely unanswered.

Another misconception about parallel processing is that it can be used to extend the solvable problem size of AI problems. Due to the high complexity of AI problems, parallel processing is useful *in improving the computational efficiency*, but *not in extending the solvable problem size* [Wah 85]. For example, a problem of size N and complexity N^k can be solved in N^k time units by a sequential processor. Assuming that N processors are used, X, the new problem size that can be solved in the same amount of time, satisfies the following equation:

$$N^*N^k = X^k$$

The left-hand side of the above equation represents the total computing power in N^k units of time with N processors, and the right-hand side represents the number of operations to be performed in solving a problem of size X. Solving the above equation yields

$$X = N^{1+1/k}$$

Table 4.1 summarizes the results for other instances. It is assumed that the size of the problem solved by a sequential processor is N, that the number of parallel processors ranges from 1 to 2^N, that linear speedup is achievable, and that the same amount of time is allocated to both sequential and parallel processing. The first column of Table 4.1 shows the complexities of solving the problem optimally, and the other columns show the corresponding sizes of the same problem that can be evaluated in the same amount of time for various number of processors. The extension in problem size is minimal when the problem involved is complex. This is evident in the last row in which the problem solved has exponential complexity. In this case only a logarithmic increase in problem size is achieved when a polynomial num-

TABLE 4.1 Relative Problem Sizes Solvable in a Fixed Amount of Time Assuming Linear Speedup†

Complexity to find optimal solution	Number of processors					
	1	N	N^2	N^3	N^k	2^N
N	N	N^2	N^3	N^4	N^{k+1}	$N2^N$
N^2	N	$N^{1.5}$	N^2	$N^{2.5}$	$N^{k/2+1}$	$N2^{N/2}$
N^3	N	$N^{1.33}$	$N^{1.67}$	N^2	$N^{k/3+1}$	$N2^{N/3}$
N^k	N	$N^{1+1/k}$	$N^{1+2/k}$	$N^{1+3/k}$	N^2	$N2^{N/k}$
2^N	N	$N + \log N$	$N + 2 \log N$	$N + 3 \log N$	$N + k \log N$	$2N$

†When sequential processing is used, problem size is N.

ber of processors are used, and a linear increase results with an exponential number of processors.

In essence, parallel processing alone cannot circumvent the difficulty of combinatorial explosion. The power of multiprocessing should not be overemphasized and must be combined with heuristic information to solve complex AI problems. Currently, methods for combining heuristic information and massive parallelism are still largely unknown. The publication in 1985 of the Sixth-Generation Computing System development proposal shows a serious intention in Japan to go beyond the current FGCS activities and address the AI aspects of computations [Agency 85].

4.1.4 Design issues of parallel AI architectures

The essential issues in designing a computer system to support a given AI application can be classified into the representation level, the control level, and the processor level. The *representation level* deals with the knowledge and methods used to solve a given AI problem and the means to represent it. Design issues related to the representation level are discussed in Sec. 4.2. The *control level* is concerned with the detection of dependencies and parallelism in the algorithmic and program representations of the problem. Design issues related to the control level are presented in Sec. 4.3. The processor level addresses the hardware and architectural components needed to evaluate the algorithmic and program representations. Issues related to the processor level are discussed in Sec. 4.4. Examples of issues in each level are shown in Table 4.2.

Developing an AI architecture requires solutions to many issues in each level. Yet some of these issues are still open at this time. In this chapter, we do not provide an exhaustive survey of all reported projects and their relevant issues. Instead, we discuss some important issues concerning the three levels and illustrate the solutions with a number of representative systems.

4.2 Representation Level

Since 1950, knowledge-representation schemes have been widely discussed in the literature [Brachman 85, Dreyfues 86]. The representation level is an important element in the design process and dictates whether or not the given problem can be solved in a reasonable amount of time. Although various paradigms have been developed, most existing knowledge-representation methods and AI languages were designed for sequential computations, and the requirements of parallel processing were either not taken into account or were only secondary considerations. Moreover, many designers of AI computers

TABLE 4.2 Examples of Issues in Designing AI Computers

Representation level:
Choosing an appropriate knowledge representation
Representing metaknowledge
Acquiring and learning domain knowledge and metaknowledge
Representing knowledge in a distributed fashion
Declaring parallelism in AI languages
Control level:
Analyzing data dependencies
Synchronization
Maintaining consistency
Partitioning AI problems
Deciding granularity of parallelism
Dynamic scheduling and load balancing
Efficient search strategies
Tradeoffs on using heuristic information
Predicting performance and linear scaling
Processor level:
Defining computational models
Developing methods to pass information
Designing hardware for overhead-intensive operations
Designing interconnection structure for load balancing and communication of guiding and pruning information
Managing large memory space

start with a given language or knowledge-representation scheme; hence the representation level is already fixed. Research in designing AI computers has focused on automatic methods of detecting detect parallelism and providing hardware support for time-consuming operations in a given representation but has not aided users much in collecting and organizing knowledge or in designing efficient algorithms.

4.2.1 Domain-knowledge representations

Domain knowledge refers to objects, events, and actions. From an implementation point of view, the criteria to evaluate a representation scheme for a multiprocessing system are its declarative power, the degree of knowledge distribution, and its structuralization.

Declarative vs. procedural representations. The major knowledge-representation paradigms used today can roughly be classified into declarative and procedural ones, although most practical representation schemes combine features from both. Declarative representations specify static knowledge, while procedural ones include static knowledge as well as the control information that operates on this static knowledge. Horn clauses (or even first-order logic), semantic networks, and rule-based production systems are examples of declarative representations, while Lisp

programs are procedural representations. Frames combine both declarative and procedural information to represent structured knowledge. Attached to each frame is various heuristic information, such as a procedure on using the information in the frame.

A declarative approach allows hiding procedural control-flow information, thereby resulting in an easily created, modified, and understood knowledge representation. Declarative representations are referentially transparent; that is, the meaning of a whole can be derived solely from the meaning of its parts and is independent of its historical behavior. This may significantly increase program productivity because of its user orientation and user friendliness.

Declarative representations offer higher potential for parallelism than procedural ones for the same problem, because a declarative representation specifies tasks as a set, while a procedural representation may overconstrain the order of execution by the implicit order of statements. Parallel versions of procedural representations, such as parallel Lisp programs, achieve a limited amount of concurrency, while relying on programmers to specify the parallel tasks [Halstead 84, Halstead 88]. However, parallelism in a declarative representation may be restricted by the implementation of the language translators. For example, interpreters for rule-based production systems can be viewed as pattern-directed procedure invocations. Although pattern matching may provide a rich source of parallelism, the match-select-act cycle is a bottleneck and restricts the potential parallelism. Fewer restrictions are seen in the implementation of logic programming and semantic networks. This is the key reason behind the Japanese FGCS project's choosing logic as the basic representation. It has also been reported that if 256K processing units were used, the Connection Machine, using a semantic-network representation, can execute four orders of magnitude faster than a sequential Lisp machine with respect to a number of object-recognition problems [Flynn 85].

A disadvantage of declarative representations is that their nondeterminism is usually associated with a large search space that may partly counteract the gains of parallel processing; whereas procedural schemes allow the specification and direct interaction of facts and heuristic information, hence eliminating wasteful searches. A tradeoff between the degree of parallelism and the size of the search space must be made in designing a representation scheme.

Distributed-knowledge representations. A second criterion to use in evaluating a representation scheme is its degree of distribution. In a local representation, each concept is stored in a distinct physical device, and each device may be shared among multiple concepts. Although this simplifies their management, the knowledge will be lost if

the device fails. Most current AI systems adopt the local representations.

Recently, distributed representations have been proposed. In this scheme, a piece of knowledge is represented by a large number of units and distributed among multiple physical devices, and each device is shared among multiple knowledge entities. The resulting system is more robust because the failure of one physical device may cause some but not all information to be lost in multiple knowledge entities. Neural networks [Hopfield 82] and the Boltzmann machine [Hinton 84] are examples in this class. The proposed Boltzmann machine consists of a very large network of binary-valued elements that are connected to one another by bidirectional links with real-value weights. The weight on a link represents a weak pairwise constraint between two hypotheses. A positive weight indicates that the two hypotheses tend to support one another, while a negative weight suggests that the two hypotheses should not both be accepted. The quality of a solution is then determined by the total cost of all constraints it violates.

Another interesting distributed knowledge-representation scheme, called sparse distributed memory (SDM), has been proposed by Kanerva [Kanerva 86]. The SDM has a 1000-bit address to model a *random* sample of 2^{20} physical locations. Given a 1000-bit read/write address, the locations in the SDM that are within 450 bits of this address are selected associatively. Statistically, nearly 1000 memory locations will be selected. The word read is a statistical reconstruction by a majority rule. The SDM model was designed with an analogy to the human brain and can perform pattern computations such as looking up patterns similar to a given pattern and generating a pattern that is an abstraction of a given set of similar patterns [Denning 86]. Although it is much simplified with respect to the human brain, its concept may lead to a new class of computers suitable for pattern computations.

Distributed representations are generally fault-tolerant in that within a large parallel network with a few faulty units, the remaining pattern is still usable. This property is very attractive for wafer-scale integration. The disadvantage of distributed representations is that they are hard for an outside observer to understand and modify, so automatic learning schemes must be employed. An open problem at this time is to combine local and distributed representations by decomposing a large knowledge base into partitions and using a local representation for each.

Structuralization of knowledge. A third criterion to use in evaluating knowledge-representation schemes is their structuralization, which is

related to the inference time and the amount of memory space required to store the knowledge.

In general, the more structured a knowledge representation is, the less inference time and the more memory space are needed. An experimental comparison of efficiency has been reported for four kinds of knowledge-representation schemes for a pilot expert system, namely, the simple production system, structured production system, frame, and logic [Niwa 84]. It was found that the volume of the knowledge base was different for the four schemes. In one instance, both production systems have 263 rules and 15K characters, the frame system has 213 frames and 29K characters, and the logic system has 348 clauses and 17K characters. The memory space required by the frame system is the largest because some related pieces of knowledge have to be replicated in different frames. Since at most one conclusion is allowed in each Horn clause, the space of the logic system is larger than that of the production systems. The experimental results also show that, with respect to forward and backward reasoning, the frame system is the fastest, while the logic system is the slowest. The efficiency of the frame system is relatively insensitive to the size of the knowledge base because related pieces of knowledge are connected to one another by pointers, thereby limiting searches. Inference time for the simple production system is moderately sensitive to changes in the size of the knowledge base, while that for the logic system is markedly sensitive to changes in size.

Structured knowledge representations are usually desirable as long as the memory space needed is reasonable. To achieve this end, metaknowledge may be included in the knowledge base to reduce the search overhead needed. There are two problems in using metaknowledge. First, it consumes more memory space and may increase the overheads in memory management and communication. Second, metaknowledge in a poorly understood domain may be fallible and may lead the search in the wrong direction, thereby increasing the total search time. Theoretical studies and experimental comparisons are urgently needed to address this space-time tradeoff.

4.2.2 Metaknowledge representations

Metaknowledge includes the extent and origin of domain knowledge of a particular object, the reliability of certain information, the possibility that an event will occur, and the precedence constraints. In other words, metaknowledge is knowledge about domain knowledge. Metaknowledge can be considered to exist in a single level or in a hierarchy [Bowen 85]. In a hierarchical form, metaknowledge is used to decide which domain-dependent actions to perform, while *meta-metaknowledge* is the control knowledge about metaknowledge.

Higher level metaknowledge is commonsense knowledge known to humans.

The use of metaknowledge allows one to express the partial specification of program behavior in a declarative language, hence making programs more aesthetic, simpler to build, and easier to modify. It facilitates incremental system development; that is, one can start from a search-intensive program and incrementally add control information until a possibly search-free program is obtained. Last, many knowledge-representation schemes and programming paradigms, such as logic, frames, semantic networks, and object-orient programming, can be integrated with the aid of metaknowledge [Genesereth 83a, Bowen 85a].

Metaknowledge can be classified as deterministic and statistical according to the correctness and efficiency considerations.

Deterministic metaknowledge. Deterministic metaknowledge is related to the correct execution of the algorithm. Metaknowledge about precedence relationships results from a better understanding of the problem; this helps reduce the resource and time complexities. For instance, to solve the problem of sorting a list, it is necessary to analyze the problem, find the appropriate representation, and evaluate the necessary tasks. A list of n elements can be sorted by searching in parallel in O(log n!) average time (= O(n • log n)) one of the $n!$ permutations that contain the sorted elements; however, an algorithm such as Quicksort contains functionally dependent subtasks and can sort the list in O(n • log n) average time using one processor. In general, the more deeply we understand the problem to be solved, the larger is the set of necessary precedence constraints and the more efficient is the solution to the problem.

Many AI languages allow programmers to specify the sequence of executions in a serial computer, but the metaknowledge to specify the correct execution in a multiprocessing environment is incomplete or missing. In programs written in pure declarative languages, the static aspects of the represented knowledge are stressed, while the controls are left to the compiler/interpreter. For instance, in a logic program, a clause "a : – a_1, a_2, a_3." means that 'a' is true, if 'a_1,' 'a_2,' and 'a_3' are true, but nothing about their functional dependencies is represented. The sequence of executions in a serial computer is correct because a definite search order is imposed, but the precedence relationships among subgoals are unknown to the scheduler in a multiprocessor.

In a number of AI languages, such as Prolog, the type and meaning of variables and functions are dynamic and query-dependent and cannot be completely specified at compile time. To use metaknowledge in this regard, the semantic meaning of subgoals and operations can be

specified, which can be interpreted as precedence relationships by the scheduler at run time. In logic programming, the method to represent semantic information in a general and efficient way is still open.

The *metarules* used must be sufficient and precise such that all precedence relationships can be derived unambiguously and easily. An important consideration is the scope within which metarules can be applied. Commonsense metarules should be included to operate on more specific metarules specified by the programmers. Using the metarules, the interpreter/compiler generates the necessary synchronization primitives.

Several researchers have addressed the above problem. Gallaire and Lasserre used metaknowledge expressed as a general or special control strategy in a Prolog-like interpreter [Gallaire 82]. In their approach, metaknowledge is made explicit through metarules, each of which describes an action to be undertaken by the interpreter whenever the interpreter focuses its attention on an object involved in the metarule. In LP, a Prolog equation-solver learning system [Silver 86], control information is expressed in a declarative representation, and inference is performed at the metalevel. Search at the object level is replaced by search at the metalevel. Research is necessary to provide a practical method to unambiguously specify the needed synchronization through metaknowledge.

Statistical metaknowledge. Statistical metaknowledge can be used to enhance the computational efficiency of an AI program. Warren used a simple heuristic and reordered only the goals of compound queries written in pure Prolog [Warren 81]; even so, he typically obtained query speedups of an order of magnitude. The probability of a subgoal's success and the associated search cost have been found to be useful in guiding the search of logic programs [Li 85, Gooley 88]. In general, clauses in Prolog with the same head should be ordered such that those likely to succeed with a smaller expected search cost are searched first. In contrast, subgoals within a clause should be ordered such that those likely to fail with a smaller expected search cost are searched first.

In many expert systems, the *belief* and other measurements of accuracy of the information have been widely used. For example, in MYCIN, the *confidence factor* (CF) is used to decide among alternatives during a consultation session [Buchanan 84]. A CF of a rule is a measurement of the association between premises and actions. A positive CF indicates that the evidence confirms the hypothesis, while a negative CF indicates disconfirming evidence.

The representation of metaknowledge about uncertainty is an ac-

tive topic in AI. Several methods, such as Fuzzy logic and Dempster-Shafer theory, are studied currently. The proper choice is still unclear.

4.2.3 AI languages and programming

Conventional imperative languages are inefficient and complex to program for symbolic and pattern processing; hence, the design of AI programming languages has had a central role in the history of AI research. Frequently, new ideas in AI were accompanied by a new language that was natural for expressing the ideas.

To enhance programmer productivity and take full advantage of parallel processing, declarative languages have been designed for AI programming. Function-oriented, logic-oriented, and object-oriented languages are the major programming paradigms today. Lisp is an early and widely used functional language; it is characterized by symbolic computations, representation of information by lists, and recursion as the only control mechanism. Numerous imperative features have been incorporated into different dialects of Lisp, so most Lisp programs are not actually declarative, but a large enough subset allows declarative programming to be done.

Hybrids of programming paradigms have been developed. One simple approach to combining features from two languages is to provide an interface between the two. Examples include Loglisp [Robinson 82], Funlog [DeGroot 85*a*], and OIL [Davis 85]. Providing features from multiple languages within a single unified framework, such as Lambda Prolog, has also been proposed. A different approach called *narrowing* involves replacing pattern matching in functional languages by unification [Reddy 85]. Logic programs can then be expressed as functions. Recently, three commercial programming tools, KEE, ART, and LOOPS, have been introduced, which provide a mechanism to allow multiple paradigms to be used in a program.

New AI languages feature large declarative power, symbolic processing constructs, representing information by lists, and using recursion as the only control mechanism. These languages differ in their expressive power, their ease of implementation, and their ability to specify parallelism and to include heuristic knowledge. A language-oriented AI computer will inherit all the features and limitations of the language it implements. It should be noted that no single paradigm is appropriate for all problems, because one language may be more "natural" than another depending on the requirements and the personal view. Hence, intelligent systems should allow multiple styles, including function-oriented, object-oriented, and logic-oriented paradigms.

Expressive power vs. ease of implementation. Functional languages, such as pure Lisp [McCarthy 78], Backus's FP [Backus 82], HOPE [Bailey 85], and VAL [McGraw 80], share many features with logic languages, including their declarative nature, reliance on recursion, and potential for execution parallelism. Yet they have vital individual features as well. First, in functional programs, input and output variables are fixed, while in logic programs, the modes of variables are query-dependent. For example, the statement z = plus(x,y) in a functional program implies that x and y are inputs and z is output. In contrast, in a logic program, the goal sum (X,Y,Z) has eight possible combinations of modes of variables X, Y, and Z. For instance, (in, out, in) means that $Y = Z - X$. Second, in a functional program, only constant and constructor functions can appear in the output, while in a logic program, logic variables can be used as output. Third, pure functional programs are deterministic, and no search is needed, while logic programs are inherently nondeterministic and require searches. Finally, functional programming provides the ability to write high-order functions; that is, a function can be passed as an argument. In contrast, Prolog is a first-order language, although some logic programming languages are not.

The first three properties, especially the nondirectionality, make logic languages more expressive in the sense that a single logic program corresponds to several functional programs. Moreover, logic and functional programs are executed using *resolution* and *reduction* (or *term rewriting*), respectively. Note that resolution can use input information implicit in the patterns to cut down the size of the set to be examined. For example, to solve the append subgoal append([P], [Q,R], [1,2,3]), resolution makes no distinction between inputs and outputs and uses the input information (length of the lists) to select the appropriate clauses and produce bindings for the variables involved. However, in the corresponding functional formulation ([P], [Q,R]) = split([1,2,3]), all possible splits of [1,2,3] are produced, and the one that splits the list into [P] and [Q,R] will be selected. The above example illustrates that reduction can lead to overcomputation as compared to resolution.

The crucial disadvantage of functional programming lies in the difficulty of representing inherent nondeterminism in AI problems. Although the recursive formulation and the leftmost-outermost reduction of functional programs enable depth-first searches naturally, it is difficult to write a heuristic search program with a pure functional language since heuristic searches are inherently history-sensitive. In fact, best-first-search programs written in Lisp include a lot of "setq" and "prog" statements, which are not pure functional primitives [Winston 84]. Due to their less expressive power for representing nondeterminism and their inefficiency in dealing with large data

structures, pure functional languages are unsuitable for general AI applications.

Although logic languages are more expressive, their implementations, especially in a parallel processing environment, are more difficult due to the nondirectionality of variables. The dynamic nature of modes requires run-time analysis. In contrast, the run-time behavior of functional programs is much simpler to control than that of logic programs, particularly in a parallel context. Techniques, such as graph reduction and dataflow, have been developed for the parallel evaluation of functional languages. Furthermore, Lisp has only a few primitive operators and provides unique list structures to compound data objects. These features simplify the implementation of Lisp compilers/interpreters. In fact, Scheme, a dialect of Lisp, has been implemented in a single chip [Sussman 81]. The implementation, however, may be complicated by the dynamic nature of Lisp programs and primitives with side effects introduced in practical functional languages. Dynamic features, such as random accesses to linked lists, garbage collection, frequent function calls, and dynamic binding of functions, incur extensive run-time overheads.

Obviously, it would be advantageous if the simple controls of functional languages could be implemented in the more expressive logic languages. Considerable efforts have been devoted to combine functional and logic programming [DeGroot 85*a*]. One approach to simplifying logic languages is to introduce directionality of modes of variables [Reddy 85]. This method degrades its expressive power to that of first-order functional languages. Others attempt to extend functional languages to achieve the expressive power of logic languages but retain most of the underlying functional simplicity. An example is HOPE with unification [Darlington 85]. Unfortunately, up to now, a language that has good expressive power and yet is flexible enough for parallel execution does not exist. Efforts are needed in this direction.

Specification of parallelism. Since parallel processing was not a consideration when most existing AI languages, such as Lisp and Prolog, were designed, the precedence restrictions implicit in a sequential execution order cannot be detected easily in a parallel execution. To extend these languages in a parallel processing environment, explicit primitives may have to be included.

In a pure functional language (dataflow language), the meaning of an expression is independent of the history of computations performed prior to the evaluation of this expression. Precedence restrictions occur as a result of function application. Notions such as side effects do not exist; hence all arguments and distinct elements in a dynamically created structure can be evaluated concurrently. For example, to compute the average of numbers in a list s, (1.(2.(3.nil))), using the function

average(s) = div(sum(s), count(s)), the computations of sum(1.(2.(3.nil))) and count(1.(2.(3.nil))) can proceed concurrently. It has been reported that implementations of functional languages on parallel computers seem easier than on sequential ones [Darlington 84].

Note that Lisp and many of its dialects are not pure functional languages. Referential transparency is lost in most Lisp languages due to side effects. The precedence restrictions are represented not only in function calls but also in procedures.

Several parallel Lisp languages have been proposed and implemented. Multilisp, developed by Halstead, has been implemented on a 128-processor Butterfly computer. Concurrency in Multilisp can be specified by means of the **pcall** and **future** constructs [Halstead 86*a*]. **pcall** embodies an implicit fork-join. For example, (**pcall** A B C) results in the concurrent evaluation of expressions A, B, and C. The form (**future** X) immediately returns *future* (a pseudo value) for X and creates a task to concurrently evaluate X, hence allowing concurrency between the computation and the use of X. When the evaluati ·n of X yields a value, it replaces the *future*. The **future** construct is good in expressing mandatory parallelism but is quite expensive in the current Multilisp implementation.

Another parallel Lisp language, Concurrent Lisp [Sugimoto 81], is extended from Lisp 1.5 and has three additional primitive functions to specify concurrency: Starteval for process activation, and CR (critical region function) and CCR (conditional critical region function) for mutual exclusion. A multiprocessing program written in Concurrent Lisp is a set of cooperating sequential processes, each of which evaluates its given form. Similar to P/V primitives, CR and CCR have enough power to express process interactions.

In Parlog, a parallel logic programming language [Clark 84b], every argument has a mode declaration that states whether the argument is input (?) or output (↑). For example, in the following statements:

```
mode merge(?,?,↑).

merge([U|X],y,[U|Z])←merge(X,Y,Z).
```

the first two lists are merged to form the result. In Concurrent Prolog [Shapiro 83], a read-only annotation (?) is used. For example,

```
merge([U|X],Y,[U|Z])← merge(X?,Y,Z).
```

indicates that X must have a value before merge(X?,Y,Z) can be invoked. Another way to specify the concurrency is to use different symbols to distinguish between parallel AND and sequential AND, such as "," and "&" in Parlog. Guarded clauses are used in Parlog and Concurrent Prolog to partly specify parallelism. A guarded clause has a format h :- g | b., where g is the *guard* of the clause and b is its *body*. Subgoals in the body can only be evaluated when all subgoals in the

guard have succeeded and values bound have been committed to the body.

It is clear that the above approach for specifying parallelism by users detracts from the objective of declarative programming, which separates logic from control, or "what" from "how." Both mode declarations in Parlog and read-only annotations in Concurrent Prolog impose a fixed execution order on subgoals, which may be inefficient in parallel processing. On the other hand, distinguishing the guard from the body cannot completely specify the precedence relationships because subgoals in the guard and body may be dependent. The use of guards is also complicated by a lack of general methodology for selecting subgoals in the guard. Moreover, precedence relationships are a partial order, so the distinction between sequential AND and parallel AND, which are linear orders, is insufficient to specify all precedence relationships. Last, owing to the nondeterministic behavior of AI programs, users cannot always specify the parallelism perfectly. A desirable parallel AI language should allow its compiler to detect the parallelism and schedule parallel executions as efficiently as possible.

Object-oriented languages. *Object-oriented programming* holds promise as a framework for concurrent programming that can be extended to databases and knowledge bases. The prediction that "object-oriented programming will be in the 1980s what structured programming was in the 1970s" [Rentsch 82] is becoming a reality. The first object-oriented language was Smalltalk [Goldberg 83]. A variety of object-oriented languages include Loops [Stefik 86], Actor [Agha 86], CommonObjects [Snyder 85], and many others [Wegner 86]. Recently, CommonLoops was suggested as a standard for object-oriented extensions to Common Lisp by the Lisp community [Bobrow 85].

Object-oriented programming has been used to express different concepts, but the concept of an object is the common feature in these languages. Objects are entities that combine the properties of procedures and data. Object-oriented programming replaces the conventional operator-operand concept by messages and objects. All actions in an object-oriented program result from sending messages among objects. A selector in the message specifies the operation to be performed. An object responds to messages using its own procedures (called *methods*) for performing operations. Message sending supports *data abstractions*, a concept that is necessary but not sufficient for the language to be object-oriented. Object-oriented languages must additionally support the management of data abstractions using *abstract data types* and the composition of abstract data types through *inheritance*. Inheritance is used to define objects that are almost like other objects. In fact, object-oriented programming should be characterized

by the nature of its type mechanisms rather than the nature of its communication mechanisms; that is, object-oriented programming can be defined as

$$\text{Object-oriented programming} = \text{data abstraction} + \text{data types} + \text{type inheritance}$$

Object-oriented programming is a paradigm for organizing knowledge domains while allowing communications. Concurrent models, operating systems, and coordination tools are built from low-level objects, such as processes, queues, and semaphores. Hewitt's Actor model is a formalization of the ideas of object-oriented languages; in his model an actor is the analog of a class or type but has the added effects of parallelism [Hewitt 77]. Computations in the Actor model are partial orders of inherently parallel events having no assignment commands. The language Act3, based on the Actor model, combines the advantages of both object-oriented and functional programming [Agha 85]. To support object-oriented programming, appropriate objects representing data structures should exist at the hardware level as machine data-structure types. This gives birth to the *Datatype Architecture* [Giloi 85]. The Apiary network architecture is based on the Actor model [Hewitt 80, Hewitt 84].

A major problem in the representation level lies in the large amount of knowledge needed to define a good representation and the imprecise nature of this knowledge. Efforts have been directed at the automatic acquisition of domain knowledge and metaknowledge and toward the design of a language that is more expressive and yet easy to implement in a parallel processing environment. The design of a systematic method to generate alternate representations is particularly desirable. The methodology should start with the problem specification, use automated tools to transform the problem specifications into problem representations, compare alternate representations, and use metaknowledge to guide the generation of different representations.

4.3 Control Level

There are four basic issues regarding the control level of computer-system design. Maintaining a consistency of knowledge is important because incomplete and inconsistent knowledge is often dealt with in AI computations. Because multiprocessing is widely used in AI computations, related issues include the decomposition of a problem (or program) into subproblems, the synchronization of cooperating processes, and the scheduling of processes for efficient execution. Although the design issues for the control level are similar to those for

traditional multiprocessing systems, AI problems often start with different representations; hence their solutions for the control level may be very different from traditional ones.

4.3.1 Consistency maintenance

Traditional logic is *monotonic* because new axioms are only added to the list of provable theorems and never cause any to be withdrawn. However, knowledge-based systems on changing real-world domains have to cope with the maintenance of consistent deduction. Classic symbolic logic lacks the tools to deal with inconsistencies caused by new information. *Nonmonotonic reasoning* has been developed to deal with this problem [Winograd 80].

Early attempts at consistency maintenance evolved around the explicit manipulation of statements. The major system developed was STRIPS, which dealt with the manipulation of blocks of various sizes, shapes, colors, and locations by a robot [Fikes 71]. In STRIPS, the entire database is searched for inconsistencies when the robot moves a block. *System-applied inference* refers to a system in which the architecture provides a mechanism to automatically maintain the consistency of the database. The widely publicized system of this nature was Microplanner [Sussman 70]. In Microplanner, the operators of STRIPS are replaced by "theorems." There is no automatic inference mechanism, and the programmer is required to encode all possible implications of a theorem. An improvement to STRIPS is Doyle's truth maintenance system (TMS), in which the reasons for beliefs are recorded and maintained, and these beliefs can be revised when discoveries contradict assumptions [Doyle 79]. To attach a justification to a fact, a TMS is designed with a goal that efficiently links consequences and their underlying assumptions. In TMS, each relation has an associated IN and OUT node. The statement at this node is true if the statements in the IN list are known to be true and the statements in the OUT list are not true.

A different approach to handle consistency maintenance was adopted in designing the IBM YES/MVS expert system that operates on a System 370 computer under the MVS operating system [Schor 86]. This expert system is used to schedule a real-time system in which contradiction occurs between the changed facts and the previous consequences. The system removes inconsistent deductions and computes new consequences in accordance with the changed facts. The consistency maintenance mechanism has three parts: recognition of inconsistencies, modification of the resultant state to remove inconsistencies and rededuce consistent consequence, and hidden control to ensure that all inconsistencies are detected and corrected properly.

Experience on the design of YES/MVS shows a pitfall in which cor-

recting an inconsistency may cause another inconsistency, which in the process of being corrected reintroduces the first inconsistency. It was also found that knowledge represented in a style for consistency maintenance turned out to be quite modular, and maintaining it has been easier than initially expected.

Nonmonotonic logic has been demonstrated to be feasible but inefficient to implement in a large system. To allow the system to be used in real time, hardware support has to be provided on the time-consuming operations. Fundamental operations such as standard database functions may have to be implemented in hardware. The management of a virtual memory system to support frequent additions and deletions in a TMS is an important design issue. The maintenance of the appropriate storage organization such that locality is maintained among relations affecting each other is a nontrivial problem. Finally, parallel processing may introduce additional problems of consistency; efficient parallel architectures to process concurrent queries have to be investigated.

4.3.2 Partitioning

In parallel computations, determining the granularity, or the minimum size of a subproblem that should be computed by a single processor, depends on the inherent parallelism in the problem to be solved. Partitioning can be implemented at different levels. At the higher levels, a complex AI problem is partitioned into several functional tasks, each of which is processed by a functionally distributed computer system. At the lower levels, the control graph of the program is partitioned into atomic operations, each of which can be processed independently.

Partitioning can be performed by users at design time or compilers at compile time or processors at run time. In the first method, programmers use a parallel language to specify and partition problems. These languages can define parallel tasks and the associated data communications. Design issues of parallel languages were discussed in Sec. 4.2.3. In this section, we discuss static and dynamic partitioning.

Inherent parallelism and granularity. The proper granularity of parallelism should be determined from the inherent parallelism in the problem and the communication overheads involved in synchronization and scheduling. In general, finding the optimal granularity is difficult; however, the degree of parallelism inherent in the problem may provide useful information to guide the design of the architecture.

An example to illustrate the choice of the proper granularity is-

shown in the design of parallel rule-base systems. Forgy et al. observed that each OPS5 production, when it fires, manipulates a few (usually 2 to 3) working memory elements and affects only a small number (20 to 30) of productions [Forgy 84]. According to this analysis, it appears that only limited speedups are available and that massive parallelism may not be needed. To improve the degree of parallelism, further efforts should be devoted (1) to investigate parallel match algorithms, (2) to design efficient partitioning strategies, and (3) to develop techniques to rewrite sequential OPS5 programs into versions more suitable for parallel processing.

Gupta estimated that the hardware utilization will be around 2 percent if the Rete match algorithm is mapped directly onto the DADO architecture [Gupta 84]. He recommended partitioning OPS5 production rules into 32 subsets to exploit the modest amount of production-level parallelism.

Based on Gupta's algorithm, Hillyer and Shaw studied the execution of production systems on the NON-VON computer, a heterogeneous system with 32 large processor elements (LPEs) and 16K small processor elements (SPEs) [Hillyer 86]. Each SPE has 64 bytes of RAM to store a condition-element term. The large number of SPEs, which can be viewed as an active memory of LPEs, perform intraproduction tests in a massively associative fashion. The performance is predicted at a rate of more than 850 productions fired per second using hardware comparable in cost to a VAX-11/780. This shows that two orders of magnitude of speedup is achievable by properly partitioning production systems.

The partitioning algorithm used may have significant effects on performance. If a majority of node activations occur within a single partition, then the performance will not be good. Some researchers have reported heuristics for partitioning production systems, such as assigning productions that are sensitive to the same context, goal, or task, to different processors in a round-robin fashion. However, preliminary results have shown that these strategies do not bring significant improvement as compared to random partitioning [Oflazer 84]. The efficiency of intelligent partitioning strategies using knowledge previously known remains an open question.

In a multiprocessing system, it is hoped that equal-sized tasks are distributed evenly to all processing units. The above example, however, has shown that this may be impractical because the problems to be solved may have irregularly structured control-flow and dataflow graphs and data-dependent workloads. In practice, efficient heuristic methods may have to be used to partition the task graph into granules that can be executed in parallel. Important related issues to be studied in this case are the design of heterogeneous architectures and the dynamic distribution of workload.

Compiler detection of parallelism. Based on the data dependencies in a program, a compiler may be able to detect the parallel modules in it and partition the program at compile time. An example is the postcompiler of FAIM-1, called an *allocator,* which performs dataflow analysis on the procedural code and inference connectivity analysis on the logic behavior in order to statically distribute the fragments to the processing elements [Anderson 87]. Similar work has been done on partitioning programs for numeric applications [Kuck 86].

Detection of parallelism in logic programs has centered on detecting AND-parallelism and OR-parallelism. AND-parallelism in logic programs involves the simultaneous execution of subgoals in a clause. Due to shared variables, concurrent execution of two or more subgoals in a clause may result in binding conflicts. Detection of AND-parallelism is based on the analysis of *input-output modes* of arguments in a subgoal. The input and output variables in a logic program denote the direction of binding transfers during unification, in a way similar to the input and output arguments in procedure calls. However, an argument in a logic program can be in the input mode in one instance and in the output mode in another, or may remain unbound. This dynamic behavior prohibits a complete static analysis. Previous research, therefore, developed methods either to provide primitives for users to specify the modes or to automatically assign modes to arguments that can be analyzed at compile time and leave the rest to be resolved at run time.

Approaches to compile-time detection of AND-parallelism have followed two basic approaches:

1. *Restricted AND-Parallelism.* DeGroot describes a technique that restricts the allowable parallel execution of logic programs to certain, well-defined fork-join type execution graphs [DeGroot 84]. Such graphs can be expressed with a recursive execution graph expression language. Each graph expression contains a conditional, run-time test used to see if the expression can be executed in parallel or if it must execute sequentially. When expressions execute in parallel, inner expressions recursively test subsequent conditions at run time to see if they too may execute in parallel or if they must execute sequentially. The resulting behavior is similar to a downward-branching tree attached to an upward-branching tree, similar to the traditional fork-join paradigm. By restricting parallel execution to this form, DeGroot's restricted AND-parallelism (RAP) model can be efficiently implemented in an extended Warren Abstract-Machine for Prolog [Hermemegildo 88]. DeGroot also describes an efficient typing algorithm that can be used to reduce the cost of the run-time checks as detailed in Chap. 13.

2. *Static-Data Dependency Analysis.* Chang, Despain, and DeGroot studied another model which does not restrict the types of

parallel behavior but instead restricts the dynamic behavior of the Prolog clauses to a single behavior [Chang 85, DeGroot 85*b*]. Unlike DeGroot's original model, which can achieve multiple parallel execution behaviors for a given clause, the static data-dependency model uses a global program dataflow analysis to select a single, worst-case behavior for each clause. In this way, no run-time tests are required, as they are in the RAP model; however, this reduced run-time cost is traded for potentially greater parallelism. The global dataflow analysis determines when two or more terms are coupled through shared variables. Like DeGroot's model, no two subgoals are allowed to execute in parallel if they share coupled terms. Because the condition of being coupled is a dynamic state, the static data-dependency model simply assumes the worst case; that is, if two terms are ever coupled, then it is assumed that they are always coupled.

Other heuristic methods of checking types at compile time are also possible. Tung and Moldovan have also investigated a number of heuristics to infer the modes of a given variable and mark all possible input/output modes of arguments in the clauses [Tung 86].

Compiler detection of parallelism has the advantages of reduced run-time overhead and programming efforts. Its disadvantage is that it may not be able to detect all the inherent parallelism in a highly expressive AI language and may have to be combined with user declaration and dynamic detection. The restrictions of compiler detection are briefly summarized below.

Special cases. The extraction of parallelism from data-dependency analysis is based on the assumption that if two subgoals do not share any unbound variable then they can be executed concurrently. This assumption is not true in some special features of the language, such as outputs in Prolog. A solution to this problem is proposed in Chap. 13.

Procedural dependencies. A procedural dependency exists between two subgoals if their execution order is fixed by their semantics. For example, in the following clause:

```
a(X):   test_for_ok(X), work_on(X).
```

the subgoal "test_for_ok(X)" must be executed first. Note that the subgoals in this example cannot be executed concurrently even if X is grounded, because the second subgoal may contain meaningless, inaccurate, or unbound work unless the first subgoal is true. In declarative languages such as Prolog, it is difficult to specify the semantics of subgoals without specifying its explicit control for parallelism. A solution to this problem is proposed in Chap. 13.

Exponential complexity. It may be difficult to define all possible combinations of modes at compile time because they grow exponentially with the number of potential output variables.

Dynamic detection of parallelism. Many data dependencies in a highly expressive AI language cannot be resolved until run time. For example, a subgoal p(X,Y) in a logic program may be called as p(X,X), which is a coupling dependency on a query with coupled terms introduced at run time. This dependency cannot be detected at compile time. Due to the dynamic nature of AI computations, an AI computer should provide a mechanism to map the program and data onto hardware dynamically.

In general, the computational model can be represented as a token-flow graph with four kinds of nodes: AND-decomposition, OR-decomposition, AND-join, and OR-join. The tokens passed along the edges can be demand tokens, data tokens, or control tokens. Conery and Kibler described an AND/OR-process system based on a producer-consumer model that dynamically monitors variables and continually develops data-dependency networks to control the order of execution of subgoals, never allowing two potential procedures with the same variable to be executed in parallel [Conery 85]. An ordering algorithm, called a *connection rule,* is used to dynamically determine a generator for each unbound variable. When a subgoal is completed, it is checked to ensure that it did produce all variable bindings it was supposed to; otherwise, the ordering algorithm is evaluated again. Improvements were made to the above scheme to further reduce the run-time overhead and extract more parallelism [Lin 86, Kim 86].

Since dynamic partitioning must be repeatedly executed at run time, it may reduce the performance gains and could even produce negative gains. The tradeoff between static partitioning by an intelligent compiler and dynamic partitioning by a sophisticated operating system is an important issue to be addressed in parallel AI processing. Dynamic partitioning is closely related to dynamic scheduling, and related issues will be discussed in a subsequent section.

Bottleneck analysis. An important issue in partitioning is to evenly decompose the problem, so bottlenecks in performance do not exist. It is easy to see that if a bottleneck requires a fraction of the total computations, then the speedup cannot be more than the reciprocal of this fraction, regardless of how the rest is partitioned. It is well known that the performance bottleneck of an application executing on a vector computer is its scalar code. Similarly, the performance bottleneck of a parallel AI computation is its sequential part (sequential infer-

ence or I/O). An important problem is to find the bottleneck in the problem to be solved.

Experience with designing the FIDO vision system at Carnegie-Mellon University has shown that an unbalanced partitioning algorithm can substantially degrade the performance [Klinker 86]. Adding Warp, a systolic system with a peak processing rate of 100 MFLOPS, to a host (a Sun computer and three "stand-alone processors") seems to only double or triple the speed of the FIDO loop. This means that Warp is definitely underutilized; functions on the stand-alone processors, either in preprocessing or postprocessing in using the Warp array, take up a substantial amount of time. It is expected that proper partitioning of vision algorithms will improve its performance significantly.

4.3.3 Synchronization

Synchronization refers to the control of deterministic aspects of computations, while scheduling handles mainly the nondeterministic aspects. The objective of synchronization is to guarantee the correctness of parallel computations such that the results of execution in parallel are the same as those of a sequential execution; that is, the parallel execution is serializable. In some nondeterministic problems, the generation of the same set of results as a sequential execution may not be necessary. For example, a user may wish to obtain a small subset of answers from a large set; the particular answers obtained do not have to be the same in the serial and parallel cases. In this case requirements on synchronization can be relaxed in parallel processing.

Many synchronization primitives used in AI processing are the same as those used in conventional computers. Examples include semaphores, test-and-set, full/empty bits, fetch-and-add, and synchronization keys. In addition, new or extended concepts related to synchronization have been introduced by AI researchers, such as the blackboard and actors. In this section, we will survey the synchronization of AI computations in the control and data levels and mechanisms using shared memory and message passing.

Two levels of synchronization. In procedural languages, if a statement precedes another statement in the program, the implication is that this statement should be executed before the second statement if the two statements share common variables; that is, control-level synchronization is implicit when data-level synchronization is needed. This implicit execution order may overspecify the necessary precedence constraints in the problem.

On the other hand, if the tasks are specified as a set using a declarative language, then control-level synchronization is absent, and they can be processed concurrently if they do not share common variables. If they have common variables but are semantically independent,

then they can be processed sequentially in an arbitrary order to maintain data-level synchronization.

The difficulty of specifying control-level synchronization when tasks are semantically dependent is a major problem in declarative languages such as Prolog. For example, the decomposition of a set into two subsets in Quicksort must be performed before the subsets are sorted. Hence, the tasks for decomposition and for sorting are both semantically dependent and data-dependent. To overcome this problem, programmers are provided with additional tools, such as specifying the input/output modes of variables in a Prolog program, to specify control-level synchronization. These primitives may have side effects and may not be able to completely specify all control-level synchronization in all situations. These problems may have to be dealt with at run time until sufficient information is available.

In general, process activations and deactivations can be considered as control-level synchronization, while passing arguments in procedure calls can be considered as data-level synchronization. Both methods can be implemented through a shared memory or by message transfers.

Shared memory. In tightly coupled multiprocessor systems, synchronization is done through a shared memory. Examples of such existing and proposed AI computers include the Aquarius [Despain 85], Concurrent Lisp machine [Sugimoto 83], Concert Multilisp machine [Halstead 86*b*], and Parallel Inference Engine [Goto 84]. In what follows, we discuss synchronization using blackboards and show methods using shared variables in logic programs.

Blackboard. Historically, the *blackboard* model was developed for abstracting features of the HEARSAY-II speech-understanding system [Erman 80]. The model is usually viewed as a problem-solving framework; however, we discuss only its control aspect here. The model consists of three major components: a knowledge source, a blackboard data structure, and control. The knowledge to solve the problem is partitioned into knowledge sources that are kept independently. The data needed to solve the problem concerned include input data, partial solutions, alternatives, and final solutions, which are kept in a global database, the blackboard. The blackboard can be divided into multiple blackboard panels that correspond to the hierarchy of solution space. Knowledge sources result in changes in the blackboard, which lead to a solution to the problem. Communications and interactions among knowledge sources take place solely through the blackboard. A monitor is needed to ensure that no more than one knowledge source can change the blackboard at one time. There are a set of control modules that monitor changes in the blackboard and decide the appropriate ac-

tion to take next. The sequence of knowledge-source invocations is dynamic.

The blackboard model provides a useful framework for diverse types of knowledge to cooperate in solving a problem and has been used for many AI applications. Its implementation is similar to that of a critical section in operating systems. In the pure model, the solution is built one step at a time. Currently, extensive research on concurrent access to blackboards is conducted.

Hayes-Roth has proposed a more powerful *blackboard control architecture* in which control information (metaknowledge) is also stored and updated on a separated control blackboard [Hayes-Roth, 85]. This approach adapts to complex control plans as a whole. Operational strategies, heuristics, and scheduling rules can change repeatedly in the course of problem solving.

Synchronization via shared-memory variables. Although Lisp contains a "pure function" subset, it also supports many functions with side effects, such as *rplaca, rplacd, set,* and input/output functions. These side effects, which result from procedural dependencies and global (or free) variables, resemble problems in conventional parallel languages. In fact, some shared-memory multiprocessors, such as Concert and Butterfly, support both Multilisp, Simultaneous Pascal, and other parallel languages [Halstead 86*b*]. Multilisp provides a simple method to wait for values generated in the future. However, as in other languages, procedure activations in Multilisp may not be well nested, and an activation can terminate before another activation it contains. This exception-handing problem has to be addressed in programming the system [Halstead 85].

Pure Prolog is a single-assignment language. Under this restriction, the distinction between a shared-memory variable and a communication channel vanishes. Since a logic variable is not allowed to be rewritten through side effects, conventional hardware-synchronization mechanisms, such as test-and-set, full/empty-bit method, and fetch-and-add, are no longer needed in multiprocessing pure logic programs [Lindstrom 84]. The popular strategy taken now is to provide the programmer with a mechanism to delay process reduction until enough information is available so that a correct decision can be made. Currently, the Concurrent Prolog group is concentrating their efforts on Flat Concurrent Prolog, a subset of Concurrent Prolog. In Guarded Horn Clause (GHC) [Ueda 85], ICOT's current choice for Kernel Language 1, OR-parallelism was eliminated from Concurrent Prolog, and a strict synchronization rule that suspends a subgoal if it tries to write in the parent environment is adhered. This rule made the read-only annotation somewhat superfluous. Although it simplifies the imple-

mentation of GHC, some expressive power is lost due to a weaker notion of unification [Takeuchi 86].

Joins. As similar to conventional *fork-join* primitives, static joins can be used for synchronization in parallel AI processing. For example, in multiprocessing logic programs, a parent node can activate its children in parallel, and each child begins producing all possible answers. The parent waits for each child to complete, collects their answers, computes the "join" of their answers, and passes the entire set of results as its answer. This approach uncovers the greatest AND-parallelism in a logic program but is efficient only if the program consists mostly of deterministic procedures and clauses; that is, most variables have only a single binding. For nondeterministic AI problems, joins are impractical because the nondeterminism increases the uncertainty whether a given AND node should be evaluated. Note that if joins are computed dynamically, that is, a parent node collects separate answers from each child as they are produced, then the data-level synchronization employed forms a pipelined computation called *dynamic joins*. This scheme is discussed later with respect to synchronization in semantic networks.

Message passing. In passing messages, a communication channel between the sender and receiver processes is required. Synchronization via messages can be achieved through software protocols or specialized hardware. Many existing and proposed AI computers pass around messages of arbitrary complexity and perform complex operations on them. The computing elements are complex, and the communication costs are high. Alternatives to passing messages are discussed in this section.

Message passing in production systems. Reasoning using forward chaining in production systems has different behavior from reasoning using backward chaining. The behavior in forward chaining is illustrated in OPS5, whose interpreter repeatedly executes a match-select-act cycle. In the match phase, all rules whose conditions are satisfied by the current content of the working memory are selected. This is called the *conflict set*. In the select phase, conflict resolution is performed to select one of the productions in the conflict set. In the act phase, the working memory is modified according to the action part of the selected rule. Although the three phases can overlap in a multiprocessing environment, synchronization must be performed to ensure that the result is consistent with that of a sequential execution; that is, all changes in the conflict set must be known prior to the completion of conflict resolution in the next cycle.

Synchronization in the efficient Rete interpreter for OPS5 is based on a dataflow graph, which can be viewed as a collection of tests that

progressively determine the productions ready to fire. Inputs to the graph consist of changes to the working memory encoded in tokens. Output tokens specify changes that must be made to the conflict set. Tokens are sent via messages in a multiprocessing system.

Marker passing and value passing. Marker passing has been studied as an alternative to message passing. In such systems, communications among processors are in the form of single-bit markers. An important characteristic is that there is never any contention: if many copies of the same marker arrive at a node at once, they are simply ORed together. The order of markers to be passed is determined by an external host.

Marker passing is suitable for systems implementing semantic networks. Nodes in the semantic network are mapped to processors in the system. An example of such a system is NETL [Fahlman 79]. A basic inference operation in semantic networks is set intersection. Analogous to dynamic joins in databases, set intersections are implemented using data-level synchronization. If an object with n properties is searched, then n commands are sequentially broadcast to all corresponding links, the associated nodes are marked, and the node with n markers reports its identity to the controller. Marker passing is adequate for many recognition problems; however, it may not be sufficient to handle general AI problems. The Connection Machine was originally developed to implement marker passing to retrieve data from semantic networks, but its current version has more powerful processing units that can manipulate address pointers and send arbitrary messages.

In value passing, continuous quantities or numbers are passed around the system, and simple arithmetic operations are performed on these values. Like marker-passing systems, there is no contention in value passing: if several values arrive at a node via different links, they are combined arithmetically, and only one combined value is received. In this sense, value-passing systems can be considered as an analog computer. Examples of value-passing systems are the Boltzmann machine [Fahlman 83], and other "neural" computation systems [Hopfield 85].

Marker-passing systems do not gracefully handle recognition problems in which the incoming features may be noisy. These problems can be better handled by a value-passing system in which each connection has an associated scalar weight that represents the confidence on the incoming values. Many iterative-relaxation algorithms that have been proposed for solving low-level vision and speech-understanding problems are ideally suited to value-passing architectures.

Object-oriented and actor approaches. In the object-oriented approach, and in particular, the Actor model, an actor is a virtual computing unit that is defined by its behavior when messages are received. Actors communicate via point-to-point messages that are buffered by a mail system. The behavior of an actor consists of three kinds of actions: (1) communicating with specific actors of known mail addresses; (2) creating new actors; and (3) specifying a replacement that will accept the next message. Actor languages avoid the assignment command but allow actors to specify a *replacement.* Replacements can capture history-sensitive information, while allowing concurrent evaluation of data-independent expressions [Agha 86]. Message passing in actors, which can be viewed as a parameter-passing mechanism, differs from both call-by-value and call-by-reference.

4.3.4 Scheduling

Scheduling is the assignment of ready tasks to available processors. It is especially important when there is nondeterminism in the algorithm. Scheduling can be static or dynamic. Static scheduling is performed before the tasks are executed, while dynamic scheduling is carried out as the tasks are executed. The actions to be performed in scheduling include (1) determination of dependent tasks, (2) static reordering of tasks at compile time, (3) dynamic selection of tasks at run time when free processors are available, and (4) determination of the number of processors to solve a given class of problems cost-effectively. All schedules can be considered as a search strategy based on a search tree or search graph [Pearl 84].

Identifying dependencies. Parallel scheduling of AI programs is complicated by their dynamic functional and shared-variable dependencies and the high expressive power of many AI languages. Due to high expressive power, the same program can be used to represent many different dependencies, each of which may be scheduled differently. Identifying dependencies at compile time is also difficult due to the dynamic and nondeterministic nature of executions.

If functional dependencies exist among tasks, then the scheduler must find these dependencies dynamically; if there are only shared-variable dependencies and no functional dependencies, then the scheduler has to compare the merits of all possible schedules. Both examples are not practical because of the high dynamic overhead. As discussed earlier, solutions to detect dependencies are not satisfactory at this time.

A viable approach is to identify the possible dependencies at compile time, statically order all sibling nodes in a search tree for each case,

and schedule them according to a parallel depth-first strategy. A simple method was proposed by Warren [81], which orders the subgoals in a clause according to the number of possible solutions generated under the given subgoal. Our experimental simulations indicated that the worst-case evaluation time resulting from this method can be worse than the case without reordering, but the best-case time can be 2 to 30 times better. Warren's method does not consider the effects of backtracking, the possible dependencies among subgoals and clauses, and the overhead of finding the solutions. We have proposed a method to represent the effects of backtracking as an absorbing Markov chain [Li 86*b*]. By assuming that sibling nodes are independent, they are reordered to minimize the total expected search cost of the program. Heuristics have been developed to reorder subgoals when they are dependent and have side effects. Our preliminary simulations indicated that the performance is substantially better than that of Warren's method.

Selection strategies. Suppose in the course of evaluating an AI program, n active tasks and m processors are available, $1 \leq m < n$. The ideal scheduling algorithm should select m active tasks such that this decision will minimize the expected computational time. It is difficult to design such an optimal selection algorithm because (1) the metrics to guide the search are estimated heuristically and may be fallible, (2) the metrics may be dynamically changing during the search, and (3) problem-dependent precedence restrictions may exist that cannot be detected at compile time. As a result, unexpected anomalies may occur when parallel processing is applied.

The potential parallelism in an AI computation can be classified into two types: deterministic parallelism and nondeterministic parallelism. *Deterministic parallelism* refers to the concurrent execution of two or more units of computations, all of which are necessary for the completion of the given job. The computational units can be tasks, processes, and/or instructions. Since all units of computation, which are performed concurrently, have "AND" relations, this kind of parallelism is traditionally called AND-parallelism. *Nondeterministic parallelism* refers to the search of multiple potential solutions in parallel. Since all potential solutions have "OR" relations, this kind of parallelism is traditionally called OR-parallelism.

Although AND-parallelism is treated as deterministic and OR-parallelism as nondeterministic in conventional studies, the selection of descendents of an AND task to evaluate is also nondeterministic, because the aim is to select one that fails as soon as possible. Hence, scheduling is important for tasks that are nondeterministic but may not be specific with respect to AND- or OR-parallelism.

In nondeterministic searches, heuristic information to guide the

scheduler in selecting nondeterministic tasks is more important than the design of parallel processors, because the number of processors is almost always smaller than the number of processable tasks.

As an example, in selecting nodes to evaluate in a branch-and-bound search tree, which is an OR tree with lower-bound values to guide the search, the problem is reduced to finding the m smallest number from n numbers. Table 4.3 shows the results obtained by

TABLE 4.3 Selecting the m Smallest Numbers from n Numbers

Approach	Time complexity in each iteration	Space/hardware complexity for selection	Accuracy of selection
Multistage selection network	$O(\log m \cdot \log n)$	$O(n \cdot \log^2 m)$	1.0
Single-stage network	O(m)	O(n)	1.0
No-wait policy	O(1)	O(m)	0.63

three architectural approaches. In the first approach, a multistage selection network was designed to perform the selection exactly [Wah 84]. In the second approach, a single-stage ring network was used to shuffle the nodes until a complete selection was obtained [Wah 84*b*]. In the third approach, a *no-wait policy* was applied. It was recognized that the heuristic information to guide the search might not be always accurate. Hence, the most promising task in local memory was always evaluated in each cycle, while the fetch of the more promising tasks from other processors was initiated. It was found that on the average, a minimum of 63 percent of the desirable tasks to be selected were selected by the no-wait policy without any additional overhead on selection, assuming that the m most promising tasks were randomly distributed among the processors [Wah 84*b*, Wah 85].

The management of the large memory space to store the heuristic information and the large number of intermediate nodes in the search tree is another difficult problem to solve. A tradeoff must be made to decide for a given amount of heuristic information and a given architectural model, whether the amount of heuristic information should be increased or decreased, and how effective should the new heuristic information be.

The memory space required to store enough heuristic information to avoid backtracking is often prohibitive. For example, assume that all solution trees of a complete binary AND/OR tree with n levels are equally likely. The leaves are assumed to be OR nodes and are at level 0, while the root is an AND node and is at level n. We have that $f(n)$, the total number of solution trees, satisfy the following recurrence:

$$f(n) = \begin{cases} 1 & n = 0 \text{ or } n = 1 \\ 4f^2(n-2) & n \geq 2 \end{cases}$$

$$= 2^{2(2^{n/2}-1)} \tag{4.1}$$

For $n = 0$, there is only one node; hence there is one solution tree. For $n = 1$, the root is an AND node with two descendents (see Fig. 4.1*a*). Again, this represents one solution tree. For the general case, each node in level $n - 2$ has $f(n-2)$ solution trees (see Fig. 4.1*b*). A solution tree for the root at level n consists of picking two nodes in level $n - 2$, a total of four combinations. Each pair of nodes selected in level $n - 2$ represents two solution trees, all possible combinations of which will yield a new solution tree. This is depicted in Fig. 4.1*b*.

Since all solution trees are equally likely, the entropy I of the heuristic information to guide the search at the root such that a correct decision is always made without backtracking is

$$I = \sum_{j=1}^{f(n)} \frac{1}{f(n)} \log_2 f(n)$$

$$= 2(2^{n/2} - 1) \tag{4.2}$$

which is exponential with respect to the height of the tree.

To manage the large memory space incurred by the storage of intermediate subproblems that may lead to solutions, we have investigated three alternatives to support branch-and-bound algorithms with a best-first search, the results of which are displayed in Table 4.4. In a direct implementation, the best-first search was implemented on an existing virtual-memory system, a VAX-11/780 computer running 4.2

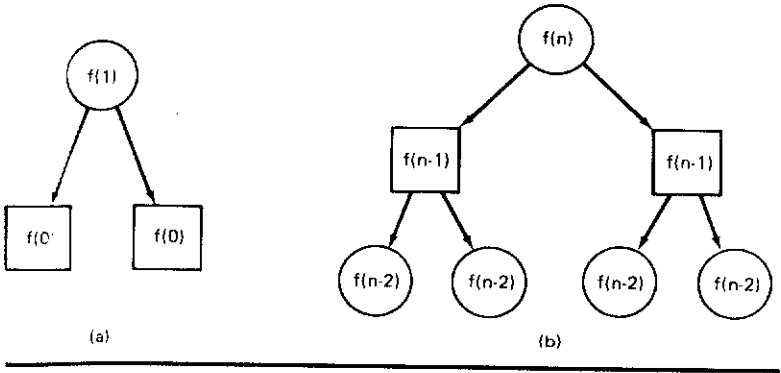

Figure 4.1 (*a*) A binary AND/OR tree with two levels; (*b*) a binary AND/OR tree with n levels. (Circles represent AND nodes; boxes represent OR nodes.)

TABLE 4.4 Relative Times to Complete a Branch-and-Bound Algorithm for Various Memory-Management Techniques

Approach	0/1 integer programming problems	0/1 knapsack problems
Direct implementation	1	1
Modified virtual memory	0.6	0.1
No-wait policy	0.1	0.001

BSD UNIX. In the second approach, a modified virtual memory with specialized fetch and replacement policies was designed to adapt to the characteristics of the search algorithm. In the third approach, the no-wait policy discussed above was used to select subproblems in the main memory without waiting for the most promising subproblem to be accessed from the secondary memory. Again, the no-wait policy is superior in performance [Yu 88].

The nondeterministic nature of computations and the fallibility of heuristic guiding may lead to anomalies of parallelism. When n processors are used to solve the problem, the resulting speedup as compared to a single processor may be less than 1, greater than n, or between 1 and n. The reasons for this anomalous behavior are due to (1) ambiguity in the heuristic information, (2) more than one solution node, and (3) approximation and dominance tests [Li 84]. As a result, subtrees searched under serial processing will be terminated, and the search will be misled into a different part of the search tree.

In summary, scheduling is important when there is nondeterminism in the problem. Good heuristic metrics to guide the search are usually difficult to design and depend on statistics such as success probabilities, search costs, and problem-dependent parameters. Tradeoffs must be made among the dynamic overhead incurred in communicating the heuristic-guiding information, the benefits that would be gained if this information led the search in the right direction, and the granularity of tasks. In practice, the merits of heuristic guiding are not clear, since the heuristic information may be fallible. As a result, some AI architects do not schedule nondeterministic tasks in parallel. The excessive overhead coupled with the fallibility of heuristic information also leads some designers to apply only static scheduling to AI programs.

Pruning. Pruning can be considered as a negative form of heuristic guiding which guides the search to avoid subproblems that will never lead to better or feasible solutions. Pruning is useful in both backward and forward chaining. In backward reasoning, problems are decomposed into smaller subproblems and evaluated independently. There are usually redundant evaluations of the same task in different parts

of the search tree when the search trees are recursive. Likewise, in forward reasoning, the more primitive facts are reduced to form more general facts until the query is satisfied. Unnecessary results are generated because it is not clear which reduction will lead to a solution of the problem.

Pruning in search problems can be carried out by dominance relations. When a node P_i dominates another node P_j, it implies that the subtree rooted at P_i contains a solution with a value no more (or no less) than the minimum (or maximum) solution value of the subtree rooted at P_j.

As an example, consider two assignments, P_1 and P_2, on the same subset of objects to be packed into a knapsack in the 0/1 knapsack problem. If the total profit of the objects assigned to the knapsack for P_1 exceeds that of P_2 and the total weight of the objects assigned in P_1 is less than that of P_2, then the best solution expanded from P_1 dominates P_2.

When parallel processing is used, it is necessary to keep the set of *current dominating nodes* (denoted by N_d) in memory [Wah 85]. These are nodes that have been generated but not yet dominated. In general, N_d can be larger than the set of active nodes. A newly generated node P_i has to be compared with all nodes in N_d to see whether P_i or any nodes in N_d are dominated.

If N_d is small, it can be stored in a bank of global data registers. However, centralized comparisons are inefficient when N_d is large. A large N_d should then be partitioned into m subsets, $N_d^0, \ldots, N_d^{m-1}$, and distributed among the local memories of the m processors. A subproblem $P_{i,j}$ generated in processor i is first compared with N_d^i; any subproblems in N_d^i, dominated by $P_{i,j}$ are removed. If $P_{i,j}$ is not dominated by a subproblem in N_d^i, it is sent to a neighboring processor and the process repeats. If it has not been dominated by any node in N_d, $P_{i,j}$ eventually returns to processor i and is inserted into N_d^i.

There are several problems associated with the use of dominance tests in AI applications. First, dominance relations are very problem dependent and cannot be derived by a general methodology. Most dominance relations have been developed for dynamic programming problems. To derive a dominance relation in a search process, a dominance relation is hypothesized and a proof is developed to show that the dominance relation is correct. Some progress has been made in using learning-by-experimentation to derive dominance relations for dynamic programming problems [Yu 86]. However, automatic proof techniques are largely missing. Moreover, learning-by-experimentation is applicable if there are a very small number of dominance relations that are used frequently in the problem. In many AI applications coded in Prolog, there are a large number of dominance

relations, each of which is used infrequently in the program. Some special cases can be solved, such as finding redundant computations in recurrences [Chen 87]. For the general situation, it is sometimes difficult to find these dominance relations without human ingenuity. Second, many dominance relations are related to the semantics of the applications. A good language to represent semantics is missing at this time. Last, the overhead of applying dominance relations is usually very high, and sequential and parallel implementations will incur prohibitive overhead.

Granularity of parallelism. When a parallel computer system with a large number of processors is available, it is necessary to determine the granularity of parallelism, that is, the size of tasks that will be executed as an indivisible unit in a processor. Since many AI problems can be represented by AND/OR trees, some processors have to be idle when nodes close to the root are evaluated. The proper number of processors should be chosen to match the inherent parallelism in the problem to be solved.

The proper granularity is a function of the problem complexity, the shape of the AND/OR tree, and the distribution of processing times of tasks. Many of these parameters are dynamically changing and data-dependent, and only special instances can be analyzed [Li 86c]. An important functional requirement for parallel processing of AI programs is the ability to dynamically distribute the workload. For a system with a small granularity, an efficient interconnection network is required to transfer data and control information. In a loosely coupled system with a coarse grain, an effective load-balancing mechanism is also needed.

4.4 Processor Level

The VLSI technology that has flourished in the past 10 years has resulted in the development of many special-purpose computers for AI processing. Architectures for AI processing can be classified into the microlevel, macrolevel, and system-level architectures. Microlevel and macrolevel architectures are discussed in the next two sections. Sections 4.4.3 through 4.4.7 briefly discuss the system-level architectures. A taxonomy of architectures implementing AI systems is discussed in Chap. 7.

4.4.1 Microlevel architectures

The microlevel architectures consist of architectural designs that are fundamental to applications in AI. In the design of massively parallel AI machines [Fahlman 83], some of the basic computational problems

recognized are set intersection, transitive closure, contexts and partitions, best-match recognition, Gestalt recognition, and recognition under transformation. These operations may not be unique to AI, and many exist in other applications as well. Due to the simplicity of some of these operations, they are usually implemented directly in hardware, especially in systolic arrays. Many other basic operations can also be implemented in VLSI. Examples include sorting and selection, computing transitive closure, string and pattern matching, selection from secondary memories, dynamic programming evaluations, proximity searches, and unification.

Some AI languages such as Lisp differ from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of numbers or bit fields of a fixed size. The instruction set must be designed according to the storage structure [Steele 80]. Additional concepts that are well suited for list processing are the tagged-memory [Moon 87] and stack architectures.

4.4.2 Macrolevel architectures

The macrolevel is an intermediate level between the microlevel and the system level. In contrast to the microlevel architectures, macrolevel architectures are (possibly) made up of a variety of microlevel architectures and perform more complex operations. However, they are not considered a complete AI system but can be taken as more complex supporting mechanisms for the system level. The architectures can be classified into those that manage data, such as dictionary machines, database machines, and structures for garbage collection, and those that search.

A dictionary machine is an architecture that supports the insertion, deletion, and searching for membership, extremum, and proximity of keys in a database [Schmeck 85]. Most designs are based on binary-tree architectures; however, designs using radix trees and a small number of processors have been found to be preferable when keys are long and clustered [Fisher 84].

Database machines depend on an architectural approach that distributes the search intelligence into the secondary and mass storage and relieves the workload of the central processor. Extensive research has been carried out in the past decade on optical and mass storage, back-end storage systems, and database machines. Earlier database machines developed were mainly directed toward general-purpose relational database management systems. Examples include the DBC, DIRECT, RAP, CASSM, associative array processors, text retrieval systems, and CAFS [Hsiao 79, Langdon 79, Babb 82]. Nearly all current research on database machines to support knowledge databases

assumes that the knowledge database is relational, hence research is directed toward solving the disk paradox [Boral 83] and enhancing previous relational database machines by extensive parallelism [Tanaka 84, Muraka 84, Shibayama 84, Sakai 84]. Commercially available database and back-end machines have also been applied to knowledge management [Kellogg 83, Neches 84].

Searching is essential to many applications, although unnecessary combinatorial searches should be avoided. The suitability of parallel processing to searching depends on the problem complexity, the problem representation, and the corresponding search algorithms. Parallel algorithms and architectures to support divide-and-conquer, branch-and-bound, and AND/OR-graph search have been developed [Wah 85].

Extensive research has been carried out in supporting dynamic data structures in a computer with a limited memory space. *Garbage collection* is an algorithm that periodically reclaims memory space no longer needed by the users [Cohen 81]. This is usually transparent to the users and could be implemented in hardware, software, or a combination of both. For efficiency reasons, additional hardware such as stacks and reference counters are usually provided.

4.4.3 Functional-programming-oriented system-level architectures

The objective of writing a functional program is to define a set of (possibly recursive) equations for each function [Darlington 84]. Data structures are handled by introducing a special class of functions called *constructor functions*. This view allows functional languages to deal directly with structures that would be termed "abstract" in more conventional languages. Moreover, functions themselves can be passed around as data objects.

The design of the necessary computer architecture to support functional languages thus centers around the parallel evaluation of functional programs (function-oriented architectures) and the mechanisms of efficient manipulation of data structures (list-oriented architectures).

In *function-oriented architectures,* the design issues center on the physical interconnection of processors, the method used to "drive" the computation, the representation of programs and data, the method to invoke and control parallelism, and the optimization techniques [Vegdahl 84]. Desirable features of such architectures should include a multiprocessor system with a rich interconnection structure, the representation of list structures by balanced trees, and hardware supports for demand-driven execution, low-overhead process creation, and storage management.

Architectures to support functional-programming languages can be

classified as uniprocessor architectures, tree-structured machines, data-driven machines, and demand-driven machines. In a uniprocessor architecture, besides the mechanisms to handle lists, additional stacks to handle function calls and optimization for redundant calls and array operations may be implemented [Steel 79, Turner 79, Castan 82]. Tree-structured machines usually employ lazy evaluations but suffer from the bottleneck at the root of the tree [Mago 85, Davis 79, O'Donnell 81]. Dataflow machines are also natural candidates for executing functional programs and have tremendous potential for parallelism. However, the issue of controlling parallelism remains unresolved. A lot of the recent work has concentrated on demand-driven machines that are based on reduction machines on a set of load-balanced (possibly virtual) processors [Treleaven 82*a*, Darlington 83, Kluge 83, Keller 84*a*, Keller 84*b*, Clarke 80, Treleaven 80].

List-oriented architectures are architectures designed to efficiently support the manipulation of data structures and objects. Lisp, a mnemonic for "list processing language," is a well-known language to support symbolic processing. There are several reasons why Lisp and list-oriented computers are really needed. First, to relieve the burden on the programmers, Lisp was designed as an untyped language. The computer must be able to identify the types of data, which involves an enormous amount of data-type checking and the use of long strings of instructions at compile and run times. Conventional computers cannot do these efficiently in software. Second, the system must periodically perform garbage collection and reclaim unused memory at run time. This amounts to around 10 to 30 percent of the total processing time in a conventional computer. Hardware implementation of garbage collection is thus essential. Third, due to the nature of recursion, a stack-oriented architecture is more suitable for list processing. Last, list processing usually requires an enormous amount of space, and the data structures are so dynamic that the compiler cannot predict how much space to allocate at compile time. Special hardware to manage the data structures and the large memory space would make the system more efficient [Fitch 80, Deering 85].

The earliest implementation of Lisp machines were the PDP-6 computer and its successors the PDP-10 and PDP-20 made by the Digital Equipment Corporation [McCarthy 78]. The half-word instructions and the stack instructions of these machines were developed with Lisp's requirements in mind. Extensive work has been done for the DEC system 10s and 20s on garbage collection to manage and reclaim the memory space used.

The design of Lisp machines was started at MIT's AI Laboratory in 1974. CONS, designed in 1976 [Knight 74], was superseded in 1978 by a second-generation Lisp machine, the CADR. This machine was a

model for the first commercially available Lisp machines, including the Symbolics LM2, the Xerox 1100 Interlisp workstation, and the Lisp Machine, Inc., Series III CADR, all of them delivered in 1981. The third-generation machines were based on additional hardware to support data tagging and garbage collection. They are characterized by the Lisp Machines, Inc.'s Lambda supporting Zetalisp and LMLisp; the Symbolics 3600 supporting Zetalisp, Flavors, and Fortran 77; the Xerox 1108 and 1132 supporting Interlisp-D and Smalltalk; and the Fujitsu FACOM Alpha Machine, a back-end Lisp processor supporting Maclisp. Most of the Lisp machines support networking using Ethernet. The LMI Lambda has a NuBus developed at MIT to produce a modular, expandable Lisp machine with multiprocessor architecture.

A single-chip processor to support Lisp has been implemented in the MIT SCHEME-79 chip [Sussman 81]. Other experimental computers to support Lisp and list-oriented processing have been reported [Griss 77; Taki 79; Goto 79; Deutsch 78; Nagao 79; Greenfeld 81; Sansonnet 80, 82*a*, 82*b*; Puttkamer 83]. These machines usually have additional hardware tables, hashing hardware, tag mechanisms, and list processing hardware, or are microprogrammed to provide macroinstructions for list processing. A Lisp chip built by Texas Instruments (TI) implements over half a million transistors on a 1-cm^2 chip for 60 percent of the functions in a TI Explorer. The implementation on a single chip results in five times improvement in performance [Matthews 87]. Experimental multiprocessing systems have been proposed to execute Lisp programs concurrently [Williams 78; McKay 80; Model 80; Guzman 81; Sugimoto 81, 83; Hill 86]. Dataflow processing is suitable for Lisp because these programs are generally data-driven [Yamaguchi 84, 83; Amamiya 82, 84]. Other multiprocessing architectures to support list processing have been proposed and developed [Coghill 79, Hewitt 80, Giloi 82, Treleaven 82*a*, Diel 84].

Architectures have also been developed to support object-oriented programming languages. First developed in 1972 by the Xerox Corporation, Smalltalk is recognized as a simple but powerful way of communicating with computers. At MIT, the concept was extended to become the Flavors system. Special hardware and multiprocessors have been proposed to directly support the processing of object-oriented languages [Plotkin 83, Ishikawa 84, Suzuki 84, Ungar 84].

Owing to the different motivations and objectives of various functional-programming-oriented architectures, each machine has its own distinct features. For example, the Symbolics 3600 [Moon 87] was designed for an interactive program development environment where compilation is very frequent and ought to appear instantaneous to the user. This requirement simplified the design of the compiler and results in only a single-address instruction format, no indexed and indi-

rect addressing modes, and other mechanisms to minimize the number of nontrivial choices to be made. On the other hand, the aim in developing SOAR [Ungar 84] was to demonstrate that a reduced instruction set computer (RISC) could provide high performance in an exploratory programming environment. Instead of microcode, SOAR relied on software to provide complicated operations. As a result, more sophisticated software techniques were used.

4.4.4 Logic and production-oriented system-level architectures

Substantial research has been carried out on parallel computational models of utilizing AND-parallelism, OR-parallelism, and stream parallelism in logical inference systems, production systems, and others. The basic problem of their exponential complexity remains open at this time.

Sequential Prolog machines using software interpretation, emulation, and additional hardware support such as hardware unification and backtracking [Tick 84] have been reported. Single-processor systems for production systems using additional data memories [Lenat 77] and a RISC architecture [Forgy 84] have been studied.

New logic programming languages suitable for parallel processing have been investigated. In particular, the use of predicate logic [Emden 82]; extensions of Prolog to become Concurrent Prolog [Shapiro 83], Parlog [Clark 84*a*], and Delta-Prolog [Pereira 84]; and parallel production systems [Uhr 79] have been developed. One interesting parallel language is systolic programming, which is useful as an algorithm design and programming methodology for high-level-language parallel computers [Shapiro 84].

Several prototype multiprocessor systems for processing inference programs and Prolog have been proposed, some of which are currently under construction. These systems include multiprocessors with a shared memory [Borgwandt 84]; ZMOB, a multiprocessor of Z80s connected by a ring network [Weiser 85]; Aquarius, a heterogeneous multiprocessor with a crossbar switch [Despain 85]; and MAGO, a cellular machine implementing a Prolog compiler that translates a Prolog program into a formal functional program [Koster 84]. Techniques for analyzing Prolog programs such that they can be processed on a dataflow architecture have been derived [Bic 84, Hasegawa 84, Irani 84, Ito 85, Amamiya 86]. An associative processor has been proposed to carry out propositional and first-order predicate calculus [Dilger 83].

DADO is a multiprocessor system with a binary-tree interconnection network that implements parallel production systems [Stoflo 87].

Non-Von is another tree architecture to evaluate production systems at a lower level of granularity [Shaw 87].

4.4.5 Distributed problem-solving systems

Knowledge in an AI system can sometimes be represented in terms of semantic nets. Several proposed and experimental architectures have been developed. NETL [Fahlman 79], and its generalization to THISTLE [Falhman 83], consists of an array of simple cells with marker-passing capability to perform searches, set-intersections, inheritance of properties and descriptions, and multiple-context operations on semantic nets. Thinking Machine Inc.'s Connection Machine is a cellular machine with 65,536 processing elements. It implements marker passing and virtually reconfigures the processing elements to match the topology of the application semantic nets [Hillis 85]. Associative processors for processing semantic nets have also been proposed [Moldovan 84].

Some AI architectures are based on frame representations and may be called object-oriented architectures. For example, the *Apiary* developed at MIT is a multiprocessor actor system [Hewitt 80]. An efficient AI architecture may also depend on the problem-solving strategy. A general form called connectionist architectures evolve from implementing neurons in brains [Fahlman 87]. The basic idea of the *Boltzmann machine* is the application of statistical mechanics to constrained searches in a parallel network [Hinton 84]. The most interesting aspect of this machine lies in its domain-independent learning algorithm [Askley 85].

Including control in stored knowledge results in a distributed problem-solving system. These systems are characterized by the relative autonomy of the problem-solving nodes, a direct consequence of the limited communication capability. With the proposed formalism of the Contract Net, contracts are used to express the control of problem solving in a distributed processor architecture [Smith 81]. Related work in this area includes Petri-net modeling [Pavlin 83], distributed vehicle-monitoring testbed [Lesser 83], distributed air-traffic control system [Cammarata 83], and modeling the brain as a distributed system [Gevins 83, Fritz 84].

4.4.6 Hybrid systems

It has been suggested that a combination of Lisp, Prolog, and an object-oriented language such as Smalltalk may be a better language for AI applications [Takeuchi 83]. This approach can be carried out in two ways. First, multiple AI languages can be implemented using microprogramming on the same computer, so programs written in these

languages can be executed independently. For example, Prolog is available as a secondary language on some Lisp machines. A version of a Prolog interpreter with a speed of 4.5 kilologic inferences per second (KLIPS) has been developed for Lisp Machine's Lambda. A second approach is to design a language that combines the desirable features from several AI languages into a new language. Some of the prototype multiprocessors, such as ZMOB and MAGO, were developed with a flexible architecture that can implement object-oriented, functional, and logic languages. FAIM-1, a multiprocessor connected in the form of a twisted hex-plane topology, was designed to implement the features of object-oriented, functional, and logic programming in the OIL programming language [Anderson 87]. Currently, a parallel version of Scheme similar to MultiLisp is being implemented. HOPE, a hybrid functional and logic language, is currently being implemented on ALICE [Smith 83].

4.4.7 Fifth-generation computer projects

The Fifth-Generation Computer System (FGCS) project was started in Japan in 1982 to further the research and development of the next generation of computers. It was conjectured that computers of the next decade will be used increasingly for nonnumeric data processing such as symbolic manipulation and applied AI. The goals of the FGCS project are

1. To implement basic mechanisms for inference, association, and learning in hardware.
2. To prepare basic AI software in order to utilize the full power of the basic mechanisms implemented.
3. To implement the basic mechanisms for retrieving and managing a knowledge base in hardware and software.
4. To use pattern recognition and AI research achievements in developing user-oriented people-machine interfaces.
5. To realize supporting environments for resolving the software crisis and enhancing software production.

The FGCS project is a marriage between the implementation of a computer system and the requirements specified by applications in AI, such as natural-language understanding and speech recognition. Specific issues studied include the choice of logic programming over functional programming; the design of the basic software systems to support knowledge acquisition, management, learning, and the intelligent interface to users; the design of highly parallel architectures to support inferencing operations; and the design of distributed-

function architectures that integrate VLSI technology to support knowledge databases [Treleaven 82*b*, Kawanobe 84, Uchida 85].

A first effort in the FGCS project has been to implement a sequential inference machine, or SIM [Yokoi 84]. Its first implementation consists of two medium-performance machines for software development known as the Personal Sequential Inference machine (PSI) and Cooperative High-Speed Inference machine (CHI) [Taki 84]. The PSI and CHI have been further implemented in custom LSIs into PSI-II and CHI-II. The PSI-II has been found to have a performance that ranges from 100 to 333 KLIPS for various benchmark programs. Another architectural development is on the knowledge-base machine, Delta [Murakami 85].

The current efforts in the intermediate stage are directed at the parallel inference machine, or PIM, and the multi-PSI computers [Murakami 85]. As an intermediate target, PIM-I is being built now. It consists of about 100 processing elements, with a total speed of 10 to 20 MLIPS including overhead caused by the operating system for PIM, PIMOS. Eight processing elements with private caches in a cluster are connected through a shared memory, and a switching network is used to connect the clusters. Each processing element will be implemented in standard-cell VLSI chips. The machine language is KL1-B, based on GHC [Sato 87]. Last, the development of the basic software system acts as a bridge to fill the gap between a highly parallel computer architecture and knowledge information processing [Furukawa 84]. The PIMOS was designed as a single unified operating system to control the parallel hardware [Taki 86]. It will be built on the multi-PSI (version 2) system that is currently being developed. Each PE consists of a PSI-II with a 16-MW main memory and interfaces to the mesh interconnection network. The KL1-B interpreter is implemented in firmware and is expected to attain a speed of 100 to 150 KLIPS [Ichiyoshi 87].

In the final stage, a parallel computer with about 1000 PEs and attaining 100 MLIPS to 1 GLIPS is expected to be built. Although the projects are progressing well, there is the recognition that more research on exploiting intelligence is needed rather than on brute-force parallelism. The proposal of the Sixth-Generation Computer System Project is an indication of efforts in this direction [Agency 85].

The Japanese FGCS project has stirred intensive responses from other countries. The British project is a 5-year, $550 million cooperative program between government and industry that concentrates on software engineering, intelligent knowledge-based systems, VLSI circuitry, and people-machine interfaces. Hardware development has focused on ALICE, a Parlog machine using dataflow architectures and implementing both HOPE, Prolog, and Lisp [Smith 83]. The European Commission started the $1.5 billion, 5-year European Strategic Program for Research

in Information Technologies (Esprit) in 1984 [FGCS 84]. The program focuses on microelectronics, software technology, advanced information processing, computer-integrated manufacturing, and office automation. In the United States, the most direct response to the Japanese FGCS project was the establishment of the Microelectronics and Computer Technology Corporation in 1983 [Spectrum 83]. The project has an annual budget of $50 to $80 million per year. It has a more evolutionary approach than the revolutionary approach of the Japanese and would yield technology that the corporate sponsors can build into advanced products in the next 10 to 12 years. Meanwhile, other research organizations have formed to develop future computer technologies of the United States in a broader sense. These include DARPA's Strategic Computing and Survivability, the semiconductor industry's Semiconductor Research Corporation, and the Microelectronics Center of North Carolina [Spectrum 83].

4.5 Design Decisions of AI-Oriented Computers

The appropriate methodology to design an AI computer should utilize a top-down design approach: functional requirements should be developed from the problem requirements, which are mapped into hardware based on technological constraints. Similar to the design of conventional computers, a bottom-up design approach is not adequate since special requirements of the applications may not be satisfied. Before a design decision is made, it is important to understand the applicability of the system to a class of problems and to then strive for high performance in a prototype implementation. Thus knowing that an m-processor system gives a k-fold increase in performance over a single processor is more important than knowing the maximum instruction rate of a prototype. Proper understanding and analysis of the problem is probably more important than applying brute-force parallelism randomly in the design.

The issues classified in Table 4.2 provide a view to the sequence of design decisions made in developing a special-purpose computer to support AI processing. The various approaches can be classified as top-down, bottom-up, and middle-out.

Top-down design decisions. This approach starts by defining, specifying, refining, and validating the requirements of the application; devising methods to collect the necessary knowledge and metaknowledge; choosing an appropriate representation for the knowledge and metaknowledge; studying problems related to the control of correct and efficient execution with the given representation scheme; identifying functional requirements of components; and mapping

these components into software and microlevel, macrolevel, and system-level architectures subject to technological and cost constraints. The process is iterative. For example, the representation of knowledge and the language features may be changed or restricted when it is discovered that the functional requirements found cannot be mapped into a desirable and realizable system with the given technology and cost. In some projects, the requirements may be very loose and span across many different applications. As a result, the languages and knowledge-representation schemes used may be oriented toward general-purpose usage. The Japanese FGCS project is an attempt to use a top-down approach to design an integrated user-oriented intelligent system for a wide spectrum of applications.

Bottom-up design decisions. In this approach, the designers first design the computer system based on a computational model, such as dataflow, reduction, and control flow, and the technological and cost limitations. Possible extensions of existing knowledge-representation schemes and languages developed for AI applications are implemented. Finally, AI applications are coded using the representation schemes and languages provided. This is probably the most popular approach to apply a general-purpose or existing system for AI processing. However, it may result in inefficient processing, and the available representation schemes and languages may not satisfy the application requirements completely. The ZMOB and Butterfly multiprocessor are examples in this class.

Middle-out design decisions. This approach is a shortcut to the top-down design approach. It starts from a proven and well-established knowledge-representation scheme or AI language (most likely developed for sequential processing) and develops the architecture and the necessary modifications to the language and representation scheme to adapt to the application requirements and the architecture. This is the approach taken by many in designing special-purpose computers for AI processing. It may be subdivided into top-first and bottom-first, although both may be iterative. In a top-first middle-out approach, studies are first performed to modify the language and representation scheme to make it more adaptable to the architecture and computational model. Primitives may be added to the language to facilitate parallel processing. Nice features from several languages may be combined. The design of the architecture follows. ALICE and FAIM-1 are examples of architectures designed using this approach. In the bottom-first middle-out approach, the chosen language or representation scheme is mapped directly into architecture by providing hardware support for the overhead-intensive operations. Applications are

implemented using the language and representation scheme provided. LISP computers are examples designed with this approach.

4.6 The Future

Although many AI computers have been proposed or built, Lisp computers are probably the only architecture that have had widespread use for solving real AI problems. This is probably due to the large investment in software for many applications coded in Lisp. At present, there is no comprehensive methodology for designing parallel AI computers. Research on AI in the past three decades and the recent experience in building AI computers have led to a view that the key issue of an AI system lies in understanding the problem rather than efficient software and hardware. In fact, most underlying concepts in AI computers are not new and have been used in conventional systems. For example, hardware stack and tagged memory were proposed before they were used in Lisp computers. However, the above argument does not imply that research on hardware and architectures is not necessary.

To support efficient processing of AI applications, research must be done in developing better AI algorithms, better AI software management methods, and better AI architectures. The development of better algorithms can lead to significant improvements in performance. Many AI algorithms are heuristic in nature, and upper bounds on performance to solve these problems have not been established as in traditional combinatorial problems. As a consequence, the use of better heuristic information, based on commonsense or high-level metaknowledge and better representation of the knowledge, can have far greater improvement in performance than improved computer architecture. Automatic learning methods to aid designers in systematically acquiring and managing new knowledge to be available in the future are very important.

Better AI software management methods are essential in developing more efficient and reliable software for AI processing. AI systems are usually open and cannot be defined based on a closed-world model. The language must be able to support the acquisition of new knowledge and the validation of existing knowledge. Probabilistic reasoning, fuzzy knowledge, and nonmonotonic logic may have to be supported. The verification of the correctness of an AI program is especially difficult due to the imprecise knowledge involved and the disorganized way of managing knowledge in a number of declarative languages and representation schemes. Traditional software engineering design methodologies must be extended to become knowledge engineering to accommodate the characteristics of knowledge in AI applications. Automatic programming is important to aid designers to generate the AI software from specifications.

The role of parallel processing and innovative computer architectures lies in improving the processing time of solving a given AI problem. It is important to realize that parallel processing and better computer architectures cannot be used to overcome the exponential complexity of exhaustive enumeration (unless an exponential amount of hardware is used) and are not very useful to extend the solvable problem space. For a problem with a size that is too large to be solved today by a sequential computer in a reasonable amount of time, it is unlikely that it can be solved by parallel processing alone, even if a linear speedup can be achieved. The decision to implement a given algorithm in hardware depends on the complexity of the problem it solves and its frequency of occurrence. Problems of low complexity can be solved by sequential processing or in hardware if they are frequently encountered; problems of moderate complexity should be solved by parallel processing; and problems of high complexity should be solved by a combination of heuristics and parallel processing.

In many AI systems developed today, tasks and operations implemented in hardware are those that are frequently executed and have polynomial complexity. These tasks or operations are identified from the languages or the knowledge-representation schemes supported. The architectural concepts and parallel processing schemes applied may be either well-known conventional concepts or new concepts for nondeterministic and dynamic processing. The role of the computer architects lies in choosing a good representation, recognizing overhead-intensive tasks to maintain and learn metaknowledge, identifying primitive operations in the languages and knowledge-representation schemes, and supporting these tasks in hardware and software.

Acknowledgment

This research was supported by the National Aeronautics and Space Administration under contract NCC 2-481.

Bibliography

[Agency 85] Agency, Science and Technology. *Promotion of Research and Development on Electronic and Information Systems That May Complement or Substitute for Human Intelligence.* Science and Technology Agency, Tokyo, 1985.

[Agha 86] G. Agha. *Actor: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[Amamiya 86] M. Amamiya, M. Takesue, R. Hasegawa, and H. Mikami. "Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine." *Proc. 13th Annual International Symposium on Computer Architecture,* pp. 10–19, Tokyo, June 1986.

[Amdahl 87] G.M. Amdahl. "Tampered Expectations in Massively Parallel Process ing and Semiconductor Industry." Keynote Address, Second International Conference on Supercomputing, Santa Clara, CA, May 1987.

[Anderson 87] J.M. Anderson, W.S. Coates, A.L. Davis, R.W. Hon, I.N. Robinson, S.V. Robison, and K.S. Stevens. "The Architecture of FAIM-1." *Computer*, vol. 20, no. 1, pp. 55–65, January 1987.

[Askley 85] D.H. Askley, G.E. Hinton, and T.J. Sejnowski. "A Learning Algorithm for Boltzmann Machines." *Cognitive Science*, vol. 9, no. 1, pp. 147–169, 1985.

[Babb 82] E. Babb. "Joined Normal Form: A Storage Encoding for Relational Databases." *Trans. on Database Systems*, vol. 7, no. 4, pp. 588–614, ACM, December 1982.

[Backus 82] J. Backus. "Function-Level Computing." *Spectrum*, vol. 19, no. 8, pp. 22–27, August 1982.

[Bailey 85] R. Bailey. "A Hope Tutorial." *Byte*, vol. 10, no. 8, pp. 235–258, August 1985.

[Bic 84] L. Bic. "Execution of Logic Programs on a Dataflow Architecture." *Proc. 11th Annual International Symposium on Computer Architecture*, pp. 290–296, June 1984.

[Bobrow 85] G. Bobrow, et al. "CommonLoops: Merging Common Lisp and Object-Oriented Programming." *Tech. Rep.* ISL-85-8, Xerox Palo Alto Research Center, August 1985.

[Boral 83] H. Boral and D. DeWitt. "Database Machine: An Idea Whose Time has Passed?" *Database Machines*, pp. 166–167, Springer-Verlag, Berlin, 1983.

[Borgwardt 84] P. Borgwardt. "Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors." *Proc. International Symposium on Logic Programming*, pp. 2–11, February 1984.

[Bowen 85] K. Bowen. "Meta-Level Programming and Knowledge Representation." *New Generation Computing*, vol. 3, no. 4, pp. 359–383, 1985.

[Brachman 85] R. Brachman and H. Levesque (eds). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA, 1985.

[Buchanan 84] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Experts Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA, 1984.

[Cammarata 83] S. Cammarata, D. McArthur, and R. Steeb. "Strategies of Cooperation in Distributed Problem Solving." *Proc. 8th International Joint Conference on Artificial Intelligence*, pp. 767–770, William Kaufman, Los Altos, CA, August 1983.

[Castan 82] M. Castan and E.I. Organick. "M3L: An HLL-RISC Processor for Parallel Execution of FP-Language Programs." *Proc. 9th Annual Symposium on Computer Architecture*, pp. 239–247, 1982.

[Chang 85] J.H. Chang, A.M. Despain, and D. DeGroot. "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis." *Proc. COMPCON Spring*, pp. 218–225, 1985.

[Chen 87] H.-Y. Chen and B.W. Wah. "The "RID-REDUNDANT" Procedure in C-Prolog." *Proc. International Symposium on Methodologies for Intelligent Systems*, Charlette, NC, October 1987.

[Clark 84*a*] K. Clark and S. Gregory. "PARLOG: Parallel Programming in Logic." Research Rep. DOC 84/4, Imperial College, London, England, 1984.

[Clark 84*b*] K. Clark and S. Gregory. "Note on System Programming in PARLOG." *Proc. International Conference on Fifth-Generation Computer System*, pp. 299–306, 1984.

[Clarke 80] T. Clarke, P. Gladstone, C. Maclean, and A. Norman. "SKIM—The S, K, I Reduction Machine." Conf. Record of Lisp Conference, Stanford University, Menlo Park, CA, 1980.

[Cohen 81] J. Cohen. "Garbage Collection of Linked Data Structures." *Computing Surveys*, vol. 13, no. 3, pp. 341–367, September 1981.

[Conery 85] J.S. Conery and D.F. Kibler. "AND Parallelism and Nondeterminism in

Logic Programs." *New Generation Computing*, vol. 3, no. 1, pp. 43–70, 1985.

[Darlington 83] J. Darlington and M. Reeve. "ALICE and the Parallel Evaluation of Logic Programs." Preliminary Draft, Dept. of Computing, Imperial College of Science and Technology, London, England, June 1983.

[Darlington 84] J. Darlington. "Functional Programming." In F.B. Chambers, D.A. Duce, and G.P. Jones (eds). *Distributed Computing*, Academic Press, London, 1984.

[Darlington 85] J. Darlington, A.J. Field, and H. Pull. "The Unification of Functional and Logic Languages." Tech. Report, Imperial College, London, England, February 1985.

[Davis 79] A.L. Davis. "A Data Flow Evaluation System Based on the Concept of Recursive Locality." *Proc. National Computer Conference*, pp. 1079–1086, 1979.

[Davis 85] A.L. Davis and S.V. Robison. "The FAIM-1 Symbolic Multiprocessing System." *Proc. COMPCON*, pp. 370–375, Spring 1985.

[Deering 85] M.F. Deering. "Architectures for AI." *Byte*, pp. 193–206, April 1985.

[DeGroot 84] D. DeGroot. "Restricted AND-Parallelism." *Proc. International Conference on Fifth Generation Computers*, pp. 471–478, November 1984.

[DeGroot 85*a*] D. DeGroot and G. Lindstrom (eds). *Logic Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[DeGroot 85*b*] D. DeGroot and J.-H. Chang. "A Comparison of Two AND-Parallel Execution Models." *Hardware and Software Components and Architectures for the 5th Generation, AFCET Informatique*, pp. 271–280, Paris, March 1985.

[Denning 86] P. Denning. "A View of Kanerva's Sparse Distributed Memory." RIACS Tech. Rep. TR-86.14, NASA Ames Research Center, Moffett Field, CA, June 1986.

[Despain 85] A.M. Despain and Y.N. Patt. "Aquarius—A High Performance Computing System for Symbolic/Numeric Applications." *Proc. COMPCON*, pp. 376–382, February 1985.

[Deutsch 78] P. Deutsch. "Experience with Microprogrammed Interlisp Systems." *Proc. MICRO*, vol. 11, November 1978.

[Diel 84] H. Diel. "Concurrent Data Access Architecture." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 373–388, 1984.

[Dilger 83] W. Dilger and J. Muller. "An Associative Processor for Theorem Proving." *Proc. Symposium on Artificial Intelligence*, pp. 489–497, 1983.

[Doyle 79] J. Doyle. "A Truth Maintenance System." *Artificial Intelligence*, vol. 12, no. 3, pp. 231–272, 1979.

[Dreyfus 86] H. Dreyfus and S. Dreyfus. "Why Expert Systems Do Not Exhibit Expertise." *IEEE Expert*, vol. 1, no. 2, Summer 1986.

[Emden 82] M.H. van Emden and G.J. de Lucena-Filho. "Predicate Logic as a Language for Parallel Programming." In S.-A. Tarnlund and K. Clark (eds). *Logic Programming*, pp. 189–198, Academic Press, New York, 1982.

[Erman 80] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." *Computing Surveys*, vol. 12, no. 2, pp. 213–253, June 1980.

[Fahlman 79] S. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. Series on Artificial Intelligence, MIT Press, Cambridge, MA, 1979.

[Fahlman 83] S.E. Fahlman and G.E. Hinton. "Massively Parallel Architectures for AI: NETL, THISTLE, and BOLTZMANN Machines." *Proc. National Conference on Artificial Intelligence*, pp. 109–113, 1983.

[Fahlman 87] S.E. Fahlman and G.E. Hinton. "Connectionist Architecture for Artificial Intelligence." *Computer*, vol. 20, no. 1, pp. 100–109, January 1987.

[Feigenbaum 83] E.A. Feigenbaum. "Knowledge Engineering: The Applied Side." In J.E. Hayes and D. Michie (eds). *Intelligent Systems: The Unprecedented Opportunity*, pp. 37–55, Ellis Horwood, Chichester, England, 1983.

[Fennell 77] R.D. Fennell and V.R. Lesser. "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay-II." *IEEE Trans. Computers*, vol. C-26, no. 2, pp. 98–111, February 1977.

[FGCS 84] "ESPRIT: Europe Challenges U.S. and Japanese Competitors." *Future Generation Computer Systems*, vol. 1, no. 1, pp. 61–69, 1984.

[Fikes 71] R.E. Fikes and N.J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence*, vol. 2, nos. 3 and 4, pp. 189–208, 1971.

[Fisher 84] A.L. Fisher. "Dictionary Machines with a Small Number of Processors." *Proc. 11th Annual International Symposium on Computer Architecture*, pp. 151–156, June 1984.

[Fitch 80] J. Fitch. "Do We Really Want a Lisp Machine?" *SEAS/SMC Annual Meeting, January 1980.*

[Flynn 85] A.M. Flynn and J.G. Harris. "Recognition Algorithms for the Connection Machine." *Proc. International Joint Conference on Artificial Intelligence*, pp. 57–60, 1985.

[Forgy 84] C. Forgy, A. Gupta, A. Newell, and R. Wedig. "Initial Assessment of Architectures for Production Systems." *Proc. National Conference on Artificial Intelligence*, pp. 116–120, August 1984.

[Fritz 84] W. Fritz and The Intelligent System. *SIGART Newsletter*, no. 90, pp. 34–38, October 1984.

[Furukawa 84] K. Furukawa and T. Yokoi. "Basic Software System." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 37–57, 1984.

[Gallaire 82] H. Gallaire and C. Lasserre. "Metalevel Control For Logic Programs." In K.L. Clark and S.-A. Tarnlund (eds). *Logic Programming*, pp. 173–185, Academic Press, New York, 1982.

[Genesereth 83] M.R. Genesereth. "An Overview of Meta-Level Architecture." *Proc. National Conference on Artificial Intelligence*, pp. 119–124, 1983.

[Gevins 83] A.S. Gevins. "Overview of the Human Brain as a Distributed Computing Network." *Proc. International Conference on Computer Design: VLSI in Computers*, pp. 13–16, 1983.

[Giloi 82] W.K. Giloi and R. Gueth. "Concepts and Realization of a High-Performance Data Type Architecture." *International Journal of Computer and Information Sciences*, vol. 11, no. 1, pp. 25–54, 1982.

[Goldberg 83] A.J. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Gooley 88] M.A. Gooley and B.W. Wah. "Efficient Reordering of Prolog Programs." *Proc. 4th International Conference on Data Engineering*, Los Angeles, February 1988.

[Goto 84] A. Goto, H. Tanaka, and T. Moto-oka. "Highly Parallel Inference Engine PIE—Goal Rewriting Model and Machine Architecture." *New Generation Computing*, vol. 2, no. 1, pp. 37–58, 1984.

[Goto 79] E. Goto, T. Ida, K. Hiraki, M. Suzuki, and N. Inada. "FLATS, A Machine for Numerical, Symbolic and Associative Computing." *Proc. 6th International Joint Conference on Artificial Intelligence*, pp. 1058–1066, August 1979.

[Greenfeld 81] N. Greenfeld and A. Jericho. "A Professional's Personal Computer System." *Proc. 8th International Symposium on Computer Architecture,* pp. 217–226, 1981.

[Griss 77] M. Griss and M. Swanson. "MBALM/1700: A Microprogrammed Lisp Machine for the Burroughs B1726." *Proc. MICRO-10,* 1977.

[Gupta 84] A. Gupta. "Implementing OPS5 Production Systems on DADO." *Proc. International Conference on Parallel Processing,* pp. 83–91, 1984.

[Guzman 81] A. Guzman. "A Heterarchical Multi-Microprocessor Lisp Machine." *Proc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management,* pp. 309–317, November 1981.

[Halstead 85] R. Halstead Jr. and J. Loaiza. "Exception Handling in Multilisp." *Proc. International Conference on Parallel Processing,* pp. 822–830, August 1985.

[Halstead 86*a*] R. Halstead. "Parallel Symbolic Computing." *Computer,* vol. 19, no. 8, pp. 35–43, August 1986.

[Halstead 86*b*] R. Halstead Jr., T. Anderson, R. Osborne, and T. Sterlig. "Concept: Design of a Multiprocessor Development System." *Proc. International Symposium on Computer Architecture,* pp. 40–48, June 1986.

[Hasegawa 84] R. Hasegawa and M. Amamiya. "Parallel Execution of Logic Programs based on Dataflow Concept." *Proc. International Conference on Fifth Generation Computer Systems,* pp. 507–516, 1984.

[Hayes-Roth 85] B. Hayes-Roth. "A Blackboard Architecture for Control." *Artificial Intelligence,* vol. 26, no. 3, pp. 251–321, July 1985.

[Hermenegildo 88] M. Hermenegildo. *A Restricted AND-Parallel Execution Model and Abstract Machine for Prolog Programs.* Academic Press, New York, 1988.

[Hewitt 80] C. Hewitt. "The Apiary Network Architecture for Knowledgeable Systems." *Conf. Record of Lisp Conference,* pp. 107–117, Stanford University, Menlo Park, CA, 1980.

[Hewitt 84] C. Hewitt and H. Lieberman. "Design Issues in Parallel Architectures for Artificial Intelligence." *Proc. COMPCON Spring,* pp. 418–423, February 1984.

[Hillis 85] W.D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[Hillyer 86] B.K. Hillyer and D.E. Shaw. "Execution of OPS5 Production Systems on a Massively Parallel Machine." *J. Parallel and Distributed Computing,* vol. 3, no. 2, pp. 236–268, 1986.

[Hinton 84] G.E. Hinton, T.J. Sejnowski, and D.H. Askley. "Boltzmann Machine: Constraint Satisfaction Network that Learns." Tech. Rep., Carnegie-Mellon University, Pittsburgh, PA, 1984.

[Hopfield 85] J. Hopfield and D. Tank. "'Neural' Computation of Decisions in Optimization Problems." *Biologic Cybernetics,* vol. 52, pp. 141–152, 1985.

[Hsiao 79] D.K. Hsiao (ed.). "Special Issue on Database Machines." *Computer,* vol. 12, no. 3, March 1979.

[Ichiyoshi 87] N. Ichiyoshi, T. Miyazaki, and K. Taki. "A Distributed Implementation of Flat GHC on the Multi-PSI." *International Conference on Logic Programming,* 1987.

[Irani 84] K.B. Irani and Y.F. Shih. "Implementation of Very Large Prolog-Based Knowledge Bases on Data Flow Architectures." *Proc. 1st Conferences on Artificial Intelligence Applications,* pp. 454–459, December 1984.

[Ishikawa 84] Y. Ishikawa and M. Tokoro. "The Design of an Object-Oriented Architecture." *Proc. 11th International Symposium on Computer Architecture,* pp. 178–187, 1984.

[Ito 85] N. Ito, H. Shimizu, M. Kishi, E. Kuno, and K. Rokusawa. "Data-Flow Based Execution Mechanisms of Parallel and Concurrent Prolog." *New Generation Computing*, vol. 3, pp. 15–41, 1985.

[Kanerva 86] P. Kanerva. *Parallel Structures in Human and Computer Memory*. RIACS Tech. Rep. TR-86.2, NASA Ames Research Center, Moffett Field, CA, January 1986.

[Keller 84] R.M. Keller, F.C.H. Lin, and J. Tanaka. "Rediflow Multiprocessing." *Proc. COMPCON Spring*, pp. 410–417, 1984.

[Kellogg 83] C. Kellogg. "Intelligent Assistants for Knowledge and Information Resources Management." *Proc. 8th International Conference on Artificial Intelligence*, pp. 170–172, William Kaufman, Los Altos, CA, 1983.

[Kim 86] S. Kim, S. Maeng, and J.W. Cho. "A Parallel Execution Model of Logic Program Based on Dependency Relationship Graph." *Proc. International Conference on Parallel Processing*, pp. 976–983, August 1986.

[Klinker 86] G. Klinker, E. Clune, J. Crisman, and J. Webb. "The Implementation of a Complex Vision System on Systolic Array Machine." Tech. Rep. Department of Computer Science, CMU, May 1986.

[Kluge 83] W.E. Kluge. "Cooperating Reduction Machines." *IEEE Trans. Computers*, vol. C-32, no. 11, pp. 1002–1012, November 1983.

[Knight 74] T. Knight. "The CONS Microprocessor," AI Working Paper 80, MIT, Cambridge, MA, November 1974.

[Koster 84] A. Koster. "Compiling Prolog Programs for Parallel Execution on a Cellular Machine," *Proc. ACM'84 Annual Conference*, pp. 167–178, October 1984.

[Kuck 86] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. "Parallel Supercomputing Today and the Cedar Approach," *Science*, pp. 967–974, February 1986.

[Langdon 79] G.G. Langdon Jr. (ed). "Special Issue on Database Machines." *Trans. on Computers*, vol. C-28, no. 6, June 1979.

[Lenat 77] D.B. Lenat and J. McDermott. "Less Than General Production System Architectures," *Proc. 5th International Joint Conference on Artificial Intelligence*, pp. 923–932, William Kaufman, Los Altos, CA, 1977.

[Lenat 84] D.B. Lenat. "Computer Software for Intelligent Systems." *Scientific American*, vol. 251, no. 3, pp. 204–213, September 1984.

[Lesser 83] V.R. Lesser and D.D. Corkill. "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks." *The AI Magazine*, pp. 15–33, Fall 1983.

[Li 84] G.-J. Li and B.W. Wah. "Computational Efficiency of Parallel Approximate Branch-and-Bound Algorithms." *Proc. International Conference on Parallel Processing*, pp. 473–480, 1984.

[Li 85] G.-J. Li and B.W. Wah. "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs." *Proc. International Conference on Parallel Processing*, pp. 123–130, June 1985.

[Li 86*a*] G.-J. Li and B.W. Wah. "Coping with Anomalies in Parallel Branch-and-Bound Algorithms." *IEEE Trans. Computers*, vol. C-35, no. 6, pp. 568–573, June 1986.

[Li 86*b*] G.-J. Li and B.W. Wah. "How Good Are Parallel and Ordered Depth-First Searches?" *Proc. of International Conference on Parallel Processing*, pp. 992–999, 1986.

[Li 86*c*] G.-J. Li and B.W. Wah. "Optimal Granularity of Parallel Evaluation of AND Trees." *Proc. 1986 Fall Joint Computer Conference*, pp. 297–306, November 1986.

[Li 87] G.-J. Li and B.W. Wah. *IEEE Trans. on Software Engineering*, "Computational Efficiency of Combinatorial OR-Tree Searches, 1987.

[Lin 86] Y.-J. Lin and V. Kumar. "A Parallel Execution Scheme for Exploiting AND-Parallelism of Logic Programs." *Artificial Intelligence*, pp. 972–975, August 1986.

[Lindstrom 84] G. Lindstrom and P. Panangaden. "Stream-Based Execution of Logic Programs." *Proc. International Symposium on Logic Programming*, pp. 168–176, February 1984.

[Mago 85] G. Mago. "Making Parallel Computation Simple: The FFP Machine." *Proc. COMPCON Spring*, pp. 424–428, 1985.

[McCarthy 78] J. McCarthy. "History of Lisp." *SIGPLAN Notices*, vol. 13, no. 8, pp. 217–223, 1978.

[McGraw 80] J.R. McGraw. "Data Flow Computing: Software Development." *IEEE Trans. Computers*, vol. C-29, no. 12, pp. 1095–1103, 1980.

[McKay 80] D. McKay and S. Shapiro. "MULTI—A Lisp Based Multiprocessing System." *Conf. Record of Lisp Conference*, Stanford University, Menlo Park, CA, 1980.

[Model 80] M. Model. "Multiprocessing via Intercommunicating Lisp Systems." *Conf. Record of Lisp Conference*, Stanford University, Menlo Park, CA, 1980.

[Moldovan 84] D.I. Moldovan. "An Associative Array Architecture Intended for Semantic Network Processing." *Proc. ACM'84 Annual Conference*, pp. 212–221, October 1984.

[Moon 87] D.A. Moon. "Symbolics Architecture." *Computer*, vol. 20, no. 1, pp. 43–52, January 1987.

[Murakami 84] K. Murakami, T. Kakuta, and R. Onai. "Architectures and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 18–36, 1984.

[Murakami 85] K. Murakami, T. Kakuta, R. Onai, and N. Ito. "Research on Parallel Machine Architecture for Fifth-Generation Computer Systems." *Computer*, vol. 18, no. 6, pp. 76–92, June 1985.

[Nagao 79] M. Nagao, J.I. Tsujii, K. Nakajima, K. Mitamura, and H. Ito. "Lisp Machine NK3 and Measurement of Its Performance." *Proc. 6th International Joint Conference on Artificial Intelligence*, pp. 625–627, August 1979.

[Neches 84] P.M. Neches. "Hardware Support for Advanced Data Management Systems." *Computer*, vol. 17, no. 11, pp. 29–40, November 1984.

[Niwa 84] K. Niwa, K. Sasaki, and H. Ihara. "An Experimental Comparison of Knowledge Representation Schemes." *The AI Magazine*, pp. 29–36, Summer 1984.

[O'Donnell 81] J.T. O'Donnell. "A Systolic Associative Lisp Computer Architecture with Incremental Parallel Storage Management." *Ph.D. Dissertation*, University of Iowa, Iowa City, IA, 1981.

[Oflazer 84] K. Oflazer. "Partitioning in Parallel Processing of Production Systems." *Proc. International Conference on Parallel Processing*, pp. 92–100, 1984.

[Pavlin 83] J. Pavlin. "Predicting the Performance of Distributed Knowledge-Based Systems: A Modeling Approach." *Proc. National Conference on Artificial Intelligence*, pp. 314–319, 1983.

[Pearl 84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.

[Pereira 84] L.M. Pereira and R. Nasr. "Delta-Prolog: A Distributed Logic Program-

ming Language." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 283–291, 1984.

[Plotkin 83] A. Plotkin and D. Tabak. "A Tree Structured Architecture for Semantic Gap Reduction." *Computer Architecture News*, vol. 11, no. 4, pp. 30–44, September 1983.

[Puttkamer 83] E. von Puttkamer. "A Microprogrammed Lisp Machine." *Microprocessing and Microprogramming*, vol. 11, no. 1, pp. 9–14, January 1983.

[Reddy 85] U.S. Reddy. "On the Relationship Between Logic and Functional Languages." In D. DeGroot and E.G. Lindstrom (eds). *Logic Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Rentsch 82] T. Rentsch. "Object Oriented Programming." *SIGPLAN Notices*, vol. 17, no. 9, pp. 51–57, September 1982.

[Robinson 82] J. Robinson and E. Sibert. "LOGLISP: Motivation, Design, and Implementation." In K. Clark and S. Tarnlund (eds). *Logic Programming*, Academic Press, New York 1982.

[Sakai 84] H. Sakai, K. Iwata, S. Kamiya, M. Abe, A. Tanaka, S. Shibayama, and K. Murakami. "Design and Implementation of Relational Database Engine." *Proc. Fifth Generation Computer Systems*, pp. 419–426, 1984.

[Sansonnet 80] J.P. Sansonnet, M. Castan, and C. Percebois. "M3L: A List-Directed Architecture." *Proc. 7th Annual Symposium on Computer Architecture*, pp. 105–112, May 1980.

[Sansonnet 82*a*] J. Sansonnet, D. Botella, and J. Perez. "Function Distribution in a List-Directed Architecture." *Microprocessing and Microprogramming*, vol. 9, no. 3, pp. 143–153, 1982.

[Sansonnet 82*b*] J.P. Sansonnet, M. Castan, C. Percebois, D. Botella, and J. Perez. "Direct Execution of Lisp on a List-Directed Architecture." *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 132–139, March 1982.

[Sato 87] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. "KL1 Execution Model for PIM Cluster with Shared Memory." *International Conference on Logic Programming*, 1987.

[Schmeck 85] H. Schmeck and H. Schroder. "Dictionary Machines for Different Models of VLSI." *IEEE Trans. Computers*, vol. C-34, no. 5, pp. 472–475, May 1985.

[Schor 86] M. Schor. "Declarative Knowledge Programming: Better Than Procedural." *Expert*, vol. 1, no. 1, pp. 36–43, Spring 1986.

[Shapiro 83] E. Shapiro and A. Takeuchi. "Object Oriented Programming in Concurrent Prolog." *New Generation Computing*, vol. 1, no. 1, pp. 25–48, 1983.

[Shapiro 84] E. Shapiro. "Systolic Programming: A Paradigm of Parallel Processing." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 458–470, 1984.

[Shaw 87] D.E. Shaw. "On the Range of Applicability of an Artificial Intelligence Machine." *Artificial Intelligence*, vol. 32, pp. 151–172, 1987.

[Shibayama 84] S. Shibayama, T. Kakuta, N. Miyazaki, H. Yokota, and K. Murakami. "A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor." *New Generation Computing*, vol. 2, no. 2, pp. 131–155, 1984.

[Silver 86] B. Silver. *Mete-Level Inference: Representing and Learning Control Information in Artificial Intelligence*. North-Holland, Amsterdam, 1986.

[Simon 86] H.A. Simon. "Whether Software Engineering Needs to Be Artificially Intelligent." *Trans. on Software Engineering*, vol. SE-12, no. 7, July 1986.

[Smith 83] K. Smith. "New Computer Breed Uses Transputers for Parallel Processing." *Electronics*, pp. 67–68, February 24, 1983.

[Smith 81] R.G. Smith and R. Davis. "Frameworks for Cooperation in DistributedProblem Solving." *Trans. on Systems, Man and Cybernetics,* vol. SMC-11, no. 1, pp. 61–70, January 1981.

[Snyder 85] A. Snyder. "Report ATC-85-1, Software Technology Lab., Hewlett-Packard Lab." Object-Oriented Programming for Common Lisp, Palo Alto, CA, 1985.

[Spectrum 83] "Special Issue on Tomorrow's Computers." *Spectrum* vol. 20, no. 11, pp. 51–58, 69, November 1983.

[Steel 79] G. Steel and G. Sussman. "Design of Lisp-Based Processor, or SCHEME: A Dielectric Lisp or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode." AI Memo 514, MIT, Cambridge, MA, March 1979.

[Steele 80] G.L. Steele Jr. and G.J. Sussman. "Design of a Lisp-Based Microprocessor." *Comm. of the ACM,* vol. 23, no. 11, pp. 628–645, November 1980.

[Stefik 86] M. Stefik and G. Bobrow. "Object-Oriented Programming: Themes and Variations." *AI Magazine,* Spring 1986.

[Stoflo 87] S.J. Stoflo. "Initial Performance of the DADO2 Prototype." *Computer,* vol. 20, no. 1, pp. 75–84, January 1987.

[Sugimoto 83] S. Sugimoto, K. Agusa, K. Tabata, and Y. Ohno. "A Multi-Microprocessor System for Concurrent Lisp." *Proc. International Conference on Parallel Processing,* pp. 135–143, 1983.

[Sussman 70] G. Sussman, T. Winograd, and E. Charniak. *Micro-planner Reference Manual.* Tech. Rep. AIM-203, MIT Press, Cambridge, MA, 1970.

[Sussman 81] G.J. Sussman, J. Holloway, G.L. Steel Jr., and A. Bell. "Scheme-79—Lisp on a Chip." *Computer,* vol. 14, no. 7, pp. 10–21, July 1981.

[Suzuki 84] N. Suzuki, K. Kubota, and T. Aoki. "SWORD32: A Bytecode Emulating Microprocessor for Object-Oriented Languages." *Proc. International Conference on Fifth Generation Computer Systems,* pp. 389–397, 1984.

[Takeuchi 83] I. Takeuchi, H. Okuno, and N. Ohsato. "TAO—A Harmonic Mean of Lisp, Prolog, and Smalltalk." *SIGPLAN Notices,* vol. 18, no. 7, pp. 65–74, July 1983.

[Taki 79] K. Taki, Y. Kaneda, and S. Maekawa. "The Experimental Lisp Machine." *Proc. 6th International Joint Conference on Artificial Intelligence,* pp. 865–867, August 1979.

[Taki 84] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, and A. Mitsuishi. "Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI)." *Proc. International Conference on Fifth Generation Computer Systems,* pp. 398–409, 1984.

[Taki 86] K. Taki. "The Parallel Software Research and Development Tool: Multi-PSI System." *France-Japan Artificial Intelligence and Computer Science Symposium,* pp. 365–381, 1986.

[Tanaka 84] Y. Tanaka. "MPDC-Massive Parallel Architecture for Very Large Databases." *Proc. International Conference on Fifth Generation Computer Systems,* pp. 113–137, 1984.

[Tick 84] E. Tick and D.H.D. Warren. "Towards a Pipelined Prolog Processor." *New Generation Computing,* vol. 2, no. 4, pp. 323–345, 1984.

[Treleaven 80] P. Treleaven and G. Mole. "A Multi-Processor Reduction Machine for User-Defined Reduction Languages." *Proc. 7th International Symposium Computer Architecture,* pp. 121–130, 1980.

[Treleaven 82*a*] P.C. Treleaven and R.P. Hopkins. "A Recursive Computer Architecture for VLSI." *Proc. 9th Annual Symposium on Computer Architecture,* pp. 229–238, April 1982.

[Tung 86] Y.-W. Tung and D. Moldovan. "Detection of AND-Parallelism in Logic Programming." *Proc. International Conference on Parallel Processing*, pp. 984–991, August 1986.

[Turner 79] D.A. Turner. "A New Implementation Technique for Applicative Languages." *Software—Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979.

[Uchida 85] S. Uchida. "Inference Machines in FGCS Project." *Proc. VLSI'87 International Conference*. IFIP TC-10, WG 10.5, August 1985.

[Ueda 85] K. Ueda. "Guarded Horn Clauses." Tech. Rep. TR-103, ICOT, Tokyo, 1985.

[Uhr 79] L.M. Uhr. "Parallel-Serial Production Systems." *Proc. 6th International Joint Conference on Artificial Intelligence*, pp. 911–916, August 1979.

[Ungar 84] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. "Architecture of SOAR: Smalltalk on RISC." *Proc. 11th Annual International Symposium on Computer Architecture*, pp. 188–197, 1984.

[Vegdahl 84] S.R. Vegdahl. "A Survey of Proposed Architectures for the Execution of Functional Languages." *IEEE Trans. Computers*, vol. C-33, no. 12, pp. 1050–1071, December 1984.

[Wah 84*a*] B.W. Wah and K.L. Chen. "A Partitioning Approach to the Design of Selection Networks." *IEEE Trans. Computers*, vol. C-33, no. 3, pp. 261–268, March 1984.

[Wah 84*b*] B.W. Wah and Y.W.E. Ma. "MANIP—A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems." *IEEE Trans. Computers*, vol. C-33, no. 5, pp. 377–390, May 1984.

[Wah 85] B.W. Wah, G.-J. Li, and C.F. Yu. "Multiprocessing of Combinatorial Search Problems." *Computer*, vol. 18, no. 6, pp. 93–108, June 1985.

[Wah 86] B.W. Wah and G.-J. Li. *Tutorial on Computers for Artificial Intelligence Applications*, IEEE Press, 1986.

[Waltz 87] D.L. Waltz. "Applications of the Connection Machine." *Computer*, vol. 20, no. 1, January 1987.

[Warren 81] D.H.D. Warren. "Efficient Processing of Interactive Relational Database Queries Expressed in Logic." *Proc. 7th International Conference on Very Large Data Bases*, pp. 272–281, 1981.

[Wegner 86] P. Wegner and B. Shriver (eds). "Special Issue on Object-Oriented Programming Workshop." *SIGPLAN*, vol. 21, no. 10, October 1986.

[Weiser 85] M. Weiser, S. Kogge, M. McElvany, R. Pierson, R. Post, and A. Thareja. "Status and Performance of the ZMOB Parallel Processing System." *Proc. COMPCON Spring*, pp. 71–73, February 1985.

[Williams 78] R. Williams. "A Multiprocessing System for the Direct Execution of Lisp." *Proc. 4th Workshop on Computer Architecture for Non-Numeric Processing*, August 1978.

[Winograd 80] T. Winograd. "Extended Inference Modes in Reasoning by Computer Systems." *Artificial Intelligence*, vol. 13, pp. 5–26, 1980.

[Winston 84] P.H. Winston and S. Horn. *Lisp*, 2nd ed., Addison-Wesley, Reading, MA, 1984.

[Yamaguchi 84] Y. Yamaguchi, K. Toda, J. Herath, and T. Yuba. "EM-3: A Lisp-Based Data-Driven Machine." *Proc. International Conference on Fifth Generation Computer Systems*, pp. 524–532, 1984.

[Yokoi 84*a*] T. Yokoi, S. Uchida, and ICOT Third Laboratory. "Sequential Inference Machine: SIM—Its Programming and Operating System." *Proc. Interna-*

tional Conference on Fifth Generation Computer Systems, pp. 70–81, 1984.

[Yu 86] C.-F. Yu and B.W. Wah. "Learning Dominance Relations in Combinatorial Search Problems." *Proc. Computer Software and Applications Conference*, October 1986.

[Yu 88] C.-F. Yu and B.W. Wah. "Efficient Branch-and-Bound Algorithms on a Two-Level Memory System." *IEEE Trans. Software Engineering*, vol. SE-14, no. 9, September 1988.