Published in the Proceedings of the Third International Conference on Supercomputing, 1988, Vol. 1, pp. 484-493.

SUPERCOMPUTERS FOR SYMBOLIC PROCESSING

B. W. Wah, M. B. Lowrie, and G.-J. Li

ABSTRACT. In this paper, we provide a detailed survey on the motivations, design, applications, current status, and limitations of supercomputers designed for symbolic processing. A major difference between symbolic and conventional numeric applications is that the knowledge used in symbolic applications may be fuzzy, uncertain, indeterminate, and ill-represented. As a result, the collection, representation, and management of knowledge is more difficult in symbolic applications than in conventional numeric applications. We survey various techniques for knowledge representation and processing, from both the designers' and users' points of view. Software architectures are discussed from a supercomputing perspective; parallelism is emphasized. Hardware structures and features for support of symbolic processing are discussed, and the state-of-the-art parallel computers for symbolic processing are overviewed.

I. INTRODUCTION

Recent proliferation of applications of computers suggests that the processing of symbols rather than numbers will be the basis for the next generation of computers. This is highlighted by the numerous research efforts in Japan, Europe, and the United States.

Supercomputers have been designed with tremendous numeric processing power. These computers are of tremendous value for scientific applications and the extension of man's knowledge in scientific areas. Unfortunately, supercomputing machines aimed at numeric applications have weak symbolic processing power. The disparity between symbolic and numeric operations, therefore, calls for different architectures for symbolic processing and innovative research in computers for symbolic processing. In this paper a review of the state of the art in computers for symbolic processing is presented. The emphasis of the paper is on the machines and techniques which provide the greatest speed: supercomputers for symbolic processing.

In a discussion of supercomputers for symbolic processing, it is first necessary to find a common ground on precisely what symbolic processing, or a symbolic application is. A model of computation which enables a precise definition of symbolic processing is presented in the references[49]. The model concentrates on the parameters of the functions performed in the computation. A function which takes analog or numeric input is an analog or numeric function, respectively. A function which takes symbols, or entities which are not quantitative, is symbolic. Thus, symbolic computation is the evaluation of symbolic functions. A symbolic application is an application with symbolic input and output. Typical symbolic applications include general problem solving, database management, expert systems, natural language processing, learning, and many other applications. Specific programs for accounting, text processing, or even compilation of computer programs also fall into the area of symbolic applications.

The remainder of the introduction is devoted to a discussion of the design process. This discussion is applicable to computer design in general. It will, however, highlight the areas which receive more attention in symbolic processing systems than in numeric systems, and provides a basis for the organization of the rest of the paper.

B. W. Wah and M. B. Lowrie are with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801. G.-J. Li is with the Academia Sinica, Beijing, China.

Research was partially supported by National Aeronautics and Space Administration Grant NCC 2-481 and National Science Foundation Grant MIPS 85-19649. Research of M. B. Lowrie was also supported by a Ph.D. Scholarship from AT&T Bell Laboratories.

I.A. Computer System Design Process

Computer solution of an application can be viewed as follows. The process begins with identification of the application, and a way for representing the problem and the knowledge necessary to solve the problem. If the application is to be solved on an existing system, then the general techniques for representing knowledge are largely dictated by the software architectures available. In any case, the problem is mapped into software, which is then mapped onto the computer hardware. This process of solving a computer problem is outlined in Figure 1. This method of looking at computer solution of applications is useful because it enables an interesting perspective on the design of computer systems.

Depending on the starting point of the design process, the design can be classified as a top-down or a bottom-up approach. Top-down design begins with the specification and analysis of the application. A knowledge representation is then designed, tailored to the needs of the application. From this, a language is designed, and the system is mapped into software, microlevel and subsystem-level structures. This corresponds to the design of features from left to right in Figure 1, although the process may have to undergo many iterations if it is discovered that the functional requirements of the design cannot be implemented using current technology. The FGCS project can be viewed as a system designed using a top-down design approach.

In contrast, a bottom-up design first selects the technology and design options such as data- or demand-driven calculation. A language and application suitable for implementation with these concepts are then sought for. As above, this design process may have to undergo many iterations if it is found that the applications cannot be suitably supported by the computer designed. This corresponds to a right to left order of design, from Figure 1. ZMOB and the Butterfly Multiprocessor are instances designed using this technique.

A short cut to the top-down approach is to compromise between the top-down and bottom-up approaches, resulting in a middle-out design approach. The middle-out approach begins with the selection of an appropriate and well established knowledge representation scheme that is the most suitable for the application. The representation scheme should already have a well-developed programming environment that can be modified later for the needs of this specific system. Primitives for concurrent execution may be added to allow users to annotate concurrent tasks in the application. The hardware and software architectures are then designed, and the selection of features to incorporate includes consideration of previously used and designed structures. The middle-out approach can further be classified into top-first and bottom-first. In a top-first middle-out approach, the designers start with a well defined knowledge representation scheme and tailor it to the given application. ALICE and FAIM-1 are examples of architectures designed using this philosophy. In a bottom-first middle-out approach, the designers first develop the architecture to support a well established representation scheme before mapping the application to the scheme chosen. DADO is an example developed using this approach.

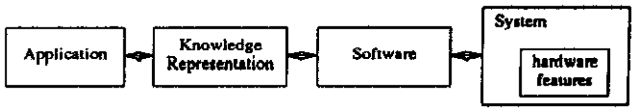

Figure 1. Solution Process.

I.B. Organization of the Paper

The architecture of a computer largely depends on the nature of processing performed on the computer. In Section II, the software architectures of symbolic supercomputers are presented. As in the top-down view of Figure 1, the process begins with the identification and characterization of the application, and development or selection of a technique for representing knowledge. Section II follows this organization by beginning with characteristics of symbolic applications, followed by a discussion of knowledge representations appropriate for symbolic processing. Section II emphasizes the software architectures and features appropriate for supercomputing environments. As most symbolic supercomputing environments endeavor to exploit parallelism, this includes a discussion of parallelism from the symbolic processing viewpoint.

Identification of the characteristics of the application and software environment enables identification of processing tasks and operations which most greatly benefit from hardware support. In Section III, hardware features including both micro and subsystem level architectures for supercomputing symbolic processing engines are presented. A complete system can be seen as a combination of hardware and software features, guided by a set of design philosophies. In Section IV, supercomputing systems for symbolic processing applications are presented. Thus, the paper presents the topics of Figure 1 in a top-down fashion.

II. SYMBOLIC PROCESSING TECHNIQUES

In order to develop effective software architectures for symbolic processing, it is necessary to anticipate some of the features of the application domain. The features prevalent in symbolic computations are enumerated in Table 1. These features do not necessarily apply to every instantiation of a symbolic processing problem and are intended as design guidelines that target general symbolic processing techniques.

Software architectures are usually derived with an underlying technique or techniques for representing knowledge. In Section II.A, the knowledge representation schemes most widely employed for symbolic processing are discussed. When this is complete, software architectures for symbolic supercomputing can be presented. Most symbolic supercomputing machines endeavor to exploit a high degree of parallelism. In Section II.B, the issue of parallelism is explored from a symbolic processing perspective. Finally, software architectures for symbolic supercomputing are presented in Section II.C.

Table 1. Characteristics of Symbolic Processing

Characteristic	Definition	Examples
Indeterminate Computations	Nondeterministism caused by --Uncertain Knowledge	Fuzzy logic, Confidence Factors
	--Incomplete Knowledge	Heuristic Search, Automatic Function Generation
	--Inconsistent Knowledge	Nonomonotic Logic, Truth Maintenance
Knowledge Processing	Domain and Metalevel Knowledge[1]	Heuristics, default reasoning, formalization of belief
Symbolic Primitives	Primitive Operations	comparison, sorting, selection, matching, logical operations, transitive closure, etc.
Parallel and Distributed Processing	AND-Parallelism	Task decomposition
	OR-Parallelism	Parallel Search

II.A. Knowledge Representation

Standard scientific supercomputing machines have focused on procedural representations. These representations, implemented in codes such as fortran, are useful for scientific applications where the algorithm is deterministic. The characteristics presented in Table 1 highlight the complexity and nondeterminism in symbolic processing. It is this complexity which has promoted the development more flexible representations, although they frequently correspond to a loss in processing efficiency. Consequently, the knowledge representations promoted for symbolic applications are frequently more flexible and allow easier exploitation of parallelism. Unfortunately, this is at the price of unnecessary computation, and greater difficulty in incorporation of metaknowledge. This decreases the efficiency of the computation. Much of the research in this area represents a trade-off between expressiveness and tractability.

The knowledge representations that have received the greatest attention include predicate logic, production systems, semantic networks, procedural languages, and connectionist representations. These representations are summarized in Table 2. Despite a great deal of effort devoted to research in knowledge representation, very little scientific theory is available to either guide the selection of an appropriate representation scheme for a given application or transform one representation into a more efficient one. Although a number of knowledge representation schemes have been proposed, none is clearly superior to the others for all applications.

Table 2. Knowledge Representation Schemes.

Representation	Definition	Strengths	Weaknesses
Predicate Logic	first order logic, Horn clauses.	natural form of expression; formal environment; programming without concern of details of implementation; parallelism.	separation of representation and control of processing; inefficient and unnecessary search; difficulty in incorporation of meta-knowledge.
Production Systems	collection of condition-action rules, usually in if-then form.	ability to incorporate declarative and procedural knowledge; natural condition action format; parallelism.	cannot reach expert level behavior[12]; high control overhead.
Semantic Networks	directed graph whose nodes represent objects, concepts, or situations, and whose arcs represent relationships between nodes	expressive power of logic[30]; default reasoning; graph search form of inference; parallelism.	simple semantic networks can only express a collection of variable-free assertions; difficult user control of processing.
Procedural	knowledge represented as a collection of modules expressed in a procedural language such as a Lisp dialect	capable of representing heuristic knowledge; extended logical inferences, such as plausible reasoning; elimination of wasteful search.	limited by the available constructs; difficulty in program creation and modification; difficult detection of parallelism.
Connectionist	concepts are represented over a number of modules, or units; units that have a positive correlation with an input feature activate, and those with negative correlation exhibit inhibitory signals;	allow automated procedures for learning concepts and representations; great potential for parallelism in computation.	difficulty of interpreting the system state and the internal representations; lengthy training/programming period.

Table 3. Issues on Parallel Knowledge Processing.

Issue	Definition	Comments
Deterministic and nondeterministic	Concurrent execution of multiple units of computation, all of which are necessary for job completion.	Low overhead guarantees speedup; tasks must be independent; pure functional programming is deterministic.
	Multiple potential solutions evaluated in parallel; parallelism used to replace or augment backtracking.	Easy to implement--always independent; nondeterministic nature may lead to anomalies in parallelism.
Granularity	Size of units of computation to be executed by a single functional unit.	Difficult to determine; is a function of knowledge representation, problem complexity, the shape of the search tree, distribution of processing times, and the dynamic nature of the problem[31].
Data level and control level	Data stored one element per processor; executes in SIMD fashion.	Can be used for large database operations, sort, set operations, statistical analysis,..[50]. Can be implemented in memory -- referred to as *active memory*[14].
	Independent control for parallel tasks.	Major type of parallelism used; MIMD systems; detection of parallelism can be more difficult than in numeric programs; nondeterministic nature requires dynamic mapping.
User defined and system defined	User specifies portions of program which can execute in parallel.	In numeric processing, DOACROSS is a typical example. The FUTURE construct in Multilisp is a symbolic construct.
	Parallelism is detected and exploited automatically by the compiler or run-time software and hardware.	Fully distributed representations allow massive system-level parallelism; some systems may employ both--the user only aids in indicating available parallelism.

II.B. Parallelism in Symbolic Processing

Humans are often thought of as the most efficient symbolic processing engines. Some researchers claim that symbolic problems can, therefore, be most effectively solved using techniques similar to those employed in the human brain. Naturally, the techniques used for parallelism are highly dependent on the technique for representing knowledge. The effort to model the brain, at its lowest level, is exemplified by neural network systems (see III.B). The issues on parallel processing, as related to symbolic processing, are summarized in Table 3.

II.C. Software Architectures for Symbolic Supercomputing

As with the choice of the knowledge representation scheme, the major goals in software languages are ease and ability to represent the solution of the application and the promotion of efficient execution of the algorithm. Once a technique, or techniques, for representing knowledge is selected, the major features of the language become apparent. In the case of Prolog, the choice of a logic representation dominates the characteristics of the language.

The software technique for implementing these features may not be as clear, however. For instance, Lisp was a procedural language developed for symbolic processing. In its implementation, functional programming with recursion and list-structured data were selected. These are not obvious choices, but they arise out of the use of a procedural representation. The selection of these features is also made on the basis of ease of representation and efficiency of processing. As another example, additional "impure" features may be added to the implementation of a knowledge representation scheme for efficiently supporting computation and/or providing flexibility to the user. CUTs and side effects in Prolog are well-known examples. This aspect can be seen as a procedural addition to a logic representation, which is done to support the efficient *implementation* of logic programming.

The extension of conventional computer languages for symbolic processing is an issue that has been explored extensively. By their nature, conventional computer languages are based on procedural representations. By examining the characteristics of symbolic processing the desirable features to incorporate into a conventional von Neumann programming language become apparent. Such features include data structures, symbolic primitives, recursion, and others. Unfortunately, conventional languages such as Fortran have not proven to provide adequate support for symbolic processing. In particular, the languages are not sufficiently flexible to enable simple encoding of very complex symbolic operations. For this reason, the new and less conventional languages for symbolic processing are emphasized in this section.

As was observed in Section II.A, the emphasis in the design of new representations for symbolic processing problems has been in the area of adding declarative and distributed features to existing representation schemes. Part of the motivation for this emphasis is referential transparency, or freedom from side-effects. This relieves some of the programming burden of the users, allowing easier programming of complex applications. In this Section, three paradigms for the design of software languages which promote referential transparency are discussed: *functional languages*, *relation languages*, and *object-oriented languages*. The emphasis of this section is on the use of these system in supercomputing environments. To date, attention on symbolic supercomputing has focused on exploitation of parallelism.

Functional Programming Languages

The functional programming approach does not employ states, program counters, or other sequence-related computational constructs. A program is a function in the mathematical sense. The program, or function, is applied to the input, and the function is evaluated to the desired output. A functional approach can be thought of as a language based on Lambda Calculus; operators are applied to data or results of further function evaluations. Examples of functional language include pure Lisp, Hope, and Dataflow Languages.

In a functional language, the meaning of an expression is independent of the history of any computation performed prior to the evaluation of the expression. Precedence restrictions occur only as a result of function application. Notions such as side-effects and shared memory do not exist in functional programs. The lack of side-effects results in the marvelous determinacy property that is valuable in parallel processing. Regardless of the order of computations of the arguments of a function, the same result (assuming termination) is guaranteed. Hence, all arguments and distinct elements in dynamically created structures in a functional program can be evaluated concurrently. The key point is that parallelism in functional languages is implicit and supported by their underlying semantics. There is no need for special message-passing constructs, synchronization primitives, or constructs for specifying parallelism.

Pure Lisp is a functional language. Many dialects of Lisp, however, are not purely functional. Operations on global variables, property lists, input/output, and other features incorporated in many Lisp languages create side-effects. Consequently, the property of referential transparency is lost in most practical Lisp languages. Moreover, precedence restrictions are represented not only by functional calls, but also in procedures.

With the presence of side effects, it is not straightforward to identify the parallel tasks as in a pure functional language. Users are required to identify independent tasks with special primitives. Several parallel Lisp languages have been proposed and implemented. Multilisp, developed by R. Halstead at MIT and implemented on a 128-processor Butterfly parallel processor, includes the usual Lisp side-effect primitives for altering data structures and changing the values of variables[18]. Concurrency is introduced by means of the **pcall** and **future** constructs[9]. Pcall embodies an implicit fork-join. For example, (**pcall** A B C) will result in the concurrent evaluation of expressions A, B, and C; while (**future** X) immediately returns a pseudo location for the value of X and creates a task to concurrently evaluate X. The use of **future** allows concurrency between the computation of a value and the use of that value.

Proponents of functional languages believe that their simplicity and elegance will conduce to more orderly, more rigorous, more verifiable, and ultimately more efficient programming. Opponents worry about losing expressiveness as a result of the expression-evaluation-only model. The crucial disadvantage of functional programming is that it is difficult to represent the inherent nondeterminism in AI problems. The recursive formulation and leftmost outermost reduction of functional programs enable depth-first search naturally, but it is difficult to write a heuristic search program by a pure functional language, since heuristic search is inherently history-sensitive. In fact, heuristic search programs written in Lisp include many "setq" and "prog" statements that are not pure functional primitives. Due to the inability of representing nondeterminism and the inefficiency of dealing with large data structures, pure functional languages are often less suitable for general symbolic applications. Their usefulness for deterministic symbolic applications is significant.

Relational Languages

There are two major forms of relational languages available: logic and production systems. The languages associated with these representations are referred to as relational since both emphasize the relation between a condition and an inference or rule.

Logic. The most popular logic based language is Prolog. Logic programming is a reasoning-oriented or deductive programming environment. Logic programming has recently received considerable attention because of its choice as the core computer language for the Fifth Generation Computer System Project in Japan.

The motivation of logic programming is to separate logic from control. However, logic programming implementations often include extralogical primitives to improve their run-time efficiency and flexibility in specification. For example, in Prolog, the CUT predicate is an extralogical control mechanism to define a similar construct as if-then in conventional languages. In addition, variables in a logic program are often nondirectional, meaning that a variable does not have to be defined as an input or output variable at compile time, and its mode can be changed at run time depending on the context. As a result, dependencies among subgoals are not defined at compile time. The solution is to require the users to specify the parallel processable tasks. In Parlog[6], every argument has a mode declaration that states whether the argument is input (?) or output (^). In Concurrent Prolog[42], a "read-only" annotation (?) is used. Users can also distinguish between "parallel AND" and "sequential AND" by using "," and "&", respectively.

Constructs can also be introduced to restrict parallelism until certain preconditions are satisfied. An example is a guard that has been adopted in Parlog. A guarded clause has the 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.

User specification of parallelism certainly detracts from the objective of declarative programming. Both mode declarations in Parlog and read-only annotations in Concurrent Prolog impose a fixed execution order on subgoals, which may be inefficient. Choosing the proper subgoals in the guard is sometimes difficult and is not guided by any general principle. Owing to the nondeterministic nature of AI applications, users cannot identify all parallel processable tasks perfectly. A better symbolic processing language should utilize both compile- and run-time detection of parallelism.

Production System. The other major form of relational language is based on production-system representations[5]. A production-system program consists of a set of data and a set of rules that can act on the data. A rule is composed of a left hand side (LHS) and a right hand side (RHS). The LHS is the antecedent or situation and represents the conditions necessary for application of the rule. The conditions are in the form of a Boolean combination of clauses[5]. The RHS is called the consequent and indicates a set of changes to the data memory to be performed when the conditions of its LHS are met. Thus, a production system can be viewed as a combination of matching the logic condition and modifying the data in a procedural fashion.

The most prevalent programming environment for implementation of a production-system representation is the OPS5 system[5]. The OPS5 system highlights the issues involved in designing production systems. OPS5 employs extensive data typing. The working memory (data) is viewed as a separate entity from the production memory, where the rules are stored. Strategies are required for matching data conditions with LHS of rules and for resolving conflicts when more than one rule has a match. OPS5 employs a Rete match algorithm that uses a graph to save partial results of matches[5].

Production systems provide a natural programming paradigm for if-then programming environments, such as those employed in expert systems. Unfortunately, algorithms with iterations and recursions are difficult to encode. In addition, rules are independent, and modulization of programs requires special attention of users. Thus, large programs are more difficult to develop[40].

Object-Oriented Languages

New languages and programming systems are being developed to simplify AI programming radically. *Object-oriented programming* holds promise as a programming framework and can be extended to concurrent systems, databases, and knowledge bases.

In conventional software, data and procedures are the main focus of the representation and are treated as separate entities. The choice of procedures and data is made by the programmer. In an object-oriented system, there is only one entity: the *object*. Objects may be manipulated like data or describe manipulation, like a procedure. Processing is performed by sending and receiving messages to and from the object that possesses the appropriate information. A selector in the message specifies the kind of operation. Message sending is uniform, and a message represents only what the sender wants (or the result returned) but does not include information on how to accomplish that. Objects respond to messages using their own procedures (called *methods*) for performing operations. Since all communication is done via messages, one method may not "call" another method. The technique for representing the procedural knowledge can be any knowledge representation scheme, although most implementations of object-oriented languages have employed procedural representations of control knowledge within the object.

In addition to objects and messages, object-oriented languages may also employ the concepts of class and instance. A class is a description of similar types of objects. Using classes, attributes of objects may be shared. In addition, classes provide a mechanism for inheritance or implicit sharing. Inheritance is used to define objects that are almost like other objects. Thus, classes provide an interface for the programmer to interact with the definition of objects.

Data abstraction is an important principle that is entailed through message sending. Object-oriented languages support both the management and collection of data through *abstract data types*, and the composition of abstract data types through an *inheritance* mechanism.

The *Actor* model, developed by Hewitt at MIT, is a formalization of the ideas of object-oriented language that also considers the added effect of parallelism. An actor is the analogue of a class or type instance. Computations in the Actor model are partial orders of events, inherently parallel and having no assignment commands. The Apiary network architecture has been proposed to support the actor model[22].

III. HARDWARE ARCHITECTURAL SUPPORT

So far, knowledge representation and techniques for encoding knowledge of symbolic applications (or symbolic languages) have been discussed. The features of a language must be carried out by either hardware or software. In some cases, these features are broken down by the compiler or interpreter into smaller pieces, each executed by a special piece of hardware.

In this Section, hardware features of symbolic processing machines are discussed. We begin with an analysis of how the features or tasks are identified which benefit the most from direct implementation in hardware. A discussion of hardware architectures that have been designed for symbolic processing computers follows. These architectures are classified into microlevel hardware features and subsystem architectures. A microlevel architecture is either a piece of hardware designed to support a feature of the language or processing techniques at its most fundamental level. It is specialized and does not, in general, provide useful computations outside of its role in the system. A subsystem-level architecture performs a complete and useful function by itself, but is often included as a portion of a larger system.

Identification of Tasks for Implementation in Hardware. New hardware architectures are best utilized for operations that the computer performs frequently. Counter to intuition, identification of these tasks is very difficult. Operations may be instructions, parts of instructions, groups of instructions, or frequently recurring tasks. Identification of new and valid areas for development of new hardware architectures is an important area of research.

The choice of a knowledge representation and software language dictates some desirable hardware architectures. The lisp language would create a need for support for: data typing, function calls, list structures and garbage collection, as well as individual commands such as car and cdr. Similarly, the Prolog language requires support for: condition matching, database functions, search operations such as backtracking, and unification. In addition to the language-defined hardware architectures, there are also symbolic operations that may be implemented with hardware support. Examples include primitive operations such as sorting and pattern matching.

In the 1960's, hardware costs were relatively high compared to software costs. This led to the efficient utilization of minimal hardware at the expense of increased software complexity. In the past two decades, hardware technologies have improved significantly, while application programming becomes increasingly more complex. Consequently, the hardware-software trade-off has to be reexamined; hardware may be employed to simplify the programming process.

III.A. Microlevel Hardware Features

In this section, hardware features of symbolic processing architectures are discussed. A microlevel hardware architecture is a unit dedicated to the support of a specific symbolic processing technique used in the system. As the number of features in symbolic applications and their languages is quite large, the number of corresponding possibilities for microlevel architectures is very large. Five representative examples of microlevel architectures are discussed in detail in this section: stacks, data tagging, garbage collection, pattern matching, and unification. These architectures were selected on their unique utility in symbolic applications. The significance of providing hardware support for these functions is illustrated in Table 4.

Hardware Stacks and Fast Stack-Access Techniques. Stack architectures support function calls. This is especially useful for Lisp and other functional programming languages. Fast stack operations are also useful for implementing Prolog. Data is often pushed and pulled from stacks in backtracking operations. The frames of a caller clause may be deeply buried in the stacks. For this reason, a stack architecture that can only store and retrieve data in the top of the stack is not adequate. In the Personal Sequential Inference Machine (PSI, a product of the Japanese Fifth Generation Project), a cache memory is employed. Several operations suited to stack access are carried out in the cache memory.

Tagged Memory. A conventional von Neumann computer does not distinguish between data and program. Both are stored as fixed-size binary words. Meaning is not inherent in the contents of storage but defined by the program manipulating the storage. A tagged architecture, however, relies on self-identifying representation at all levels of storage. Although tagging has been employed since the 1960's, early design considered tagging as a relatively unimportant and expensive peripheral concept. Tagged memory can be a key feature in symbolic processing computers today. Symbolic architectures often require identification of different types of physical and abstract data types, including integer, character, event, garbage, and others. During processing, it is necessary to identify the different operands employed in the computation. For this reason, the tagging of data to improve real-time type checking is appropriate.

Table 4. Microlevel Architectures and Their Significance.

Architecture	Significance
Stacks	For function calls and value binding; more than one stack may be used.
Tagging	Data type checking; up to an order of magnitude speed improvement for Lisp.
Garbage Collection	Reclamation of usable storage; accounts for 10 to 30 percent run-time in Lisp.
Pattern Matching	Fundamental operation; up to 90% of execution time in production systems.
Unification	Type of pattern matching for logic; over 60% of execution time in sequential logic programs.

The most common hardware support for data tagging is the allocation of extra bits in each word to represent its type. Data-type checking at run-time may be supported by additional hardware and overlapped with regular processing. The speed of symbolic computers is often linked to how effectively they emulate a tagged-memory architecture[10]. Even in untyped languages, data tagging can simplify the programming process. The programmer need no longer be concerned with the type of add instruction used, integer or long, for example.

Garbage Collection. Garbage collection refers to the process of identifying memory cells whose contents are no longer useful for the computation in progress. In this case, memory cells are contiguous groups of at least one memory word. The process involves marking these cells as available for future use and compacting free memory into contiguous blocks. It has been estimated that 10 to 30 percent of the execution time in large LISP programs is spent on garbage collection. The implication for interactive or real-time systems is great, as garbage collection often requires large continuous segments of time from the CPU.

Most recent research in the area of garbage collection has focused on parallel garbage-collection methods. Parallel garbage collection is garbage collection that is performed concurrently with program execution. The techniques often center around the use of coloring cells, or the division of memory space into (two) distinct regions. In the first case, tagging can be a useful microlevel hardware feature. Parallel garbage-collection processors can be designed with very simple and fast components, without becoming a bottleneck of the system. In a design proposed by Hibino[23], the collector processor cycle time was 200ns--6 times faster than typical processor cycle times.

Pattern Matching Hardware Support. Addition and multiplication are the mainstay of scientific computations. Similarly, pattern matching is the basic operation of symbolic processing. A pattern matcher may be employed for two major tasks: finding entries in a database and choosing an operation to execute next. For example, determining the applicable rule in a production system is a pattern matching problem. It has been estimated that 90% of execution time in a rule-based system employed for expert systems is spent in the matching phase. Therefore, hardware pattern-matching support can simplify the programming task and improve run-time efficiency.

In most symbolic representations, symbols are represented in form of strings. Conventional string-matching hardware can be classified into four categories. The first approach is associative memory. Although straightforward, it is difficult to operate on strings of variable lengths. The second technique is cascaded logic-memory arrays, called a cellular array. Each character in a pattern-string is stored in a cell and is compared with a character in the input string. A third technique is the finite-state-automata (FSA) method that uses a transition table to perform complex string matching. Finally, there is the dynamic programming technique that uses statistical characteristics of the general pattern in order to determine the parameter table for proximity matching. A survey of techniques for hardware pattern matching can be found in the references[37].

Pattern matching in symbolic processors differ from conventional database retrievals in that many symbolic applications may contain widely varying field lengths and that uncertainities in data forbid exact matching of patterns. The pattern matching hardware for a symbolic processor must be tailored to the representation(s) for which it will be used.

In matching under uncertainities, best-matching is required. Kanerva's Sparse Distributed Memory (SDM) is a system designed for best matching[29]. The proposed prototype consists of a virtual memory that is addressed by a 1024-bit address, and a small physical memory. Each word in the physical memory has the 1024-bit address for this word and the data fields. When a memory address is given, all locations in the physical memory with addresses that differ by less than 450 bits of the given address are accessed, and the corresponding data fields are combined together into a single response. The 450 bits are chosen so that approximately 0.1% of the memory words will respond on the average for a physical memory of 4 Mwords.

In a forward chaining rule-based production system, the objects to be matched are constants, and multi-pattern multi-object pattern matching is required. The Rete Match Algorithm is an efficient solution to this problem. A number of hardware implementations of this technique have been proposed, including tree architectures[44], SIMD Cellular Array Processor (CAP)[4], and tagged token data-driven multiprocessor[16].

Unification Hardware. Unification, a form of pattern matching, is the fundamental technique in logic programming. It determines if two terms can be made textually identical by finding a set of substitutions for variables in the terms, and replaces all occurrences of each variable by that variables substitution. In general, both terms to be resolved in unification are allowed to contain variables; hence, unification can be thought of as a bi-directional pattern matching operation. Since unification is applied extensively and is known to consume over 60 percent of the execution time in sequential executions, it is desirable that additional hardware or firmware support be available.

The Parallel Inference Engine (PIE) developed at the University of Tokyo employs special hardware, referred to as UNIRED, for unification and reduction. UNIRED may be characterized by the following features: tagged memory, high-speed local memory that can be accessed in parallel, parallel hardware stacks, and dedicated internal busses. The unify processor fetches a goal from a memory module and candidate clauses from definition memory. The unify processor then attempts to unify them, generate new goals, and return these goals to the memory module.

Parallel unification is also an area of great interest. Parallel unification can be performed either by unifying each term pair in two atoms simultaneously, or by finding many possible unifications concurrently. Unfortunately, the unification problem is proven to be log-space-complete in P. This means that it is not possible to perform parallel unification in $O(\log^k n)$ time using a polynomial number of processors for any k, where n is the total number of nodes and edges of the directed acyclic-graph representation of the clauses[53]. It has also been shown, however, that near linear speedup can be achieved in parallel unification. Thus, unification algorithms are parallelizable from a practical perspective. A mesh connected array of unifiers has been proposed to exploit AND-parallelism in unification and may achieve super-linear speedup[43].

III.B. Subsystem Level Architectures

The subsystem-level architecture represents an intermediate level between microlevel and complete-system designs. In this section, we identify three techniques for classifying the subsystem-level architectures that address different types of knowledge processing. Three different types of subsystems will emerge from this analysis: data- and knowledge-base machines, inference machines, and neural networks.

Knowledge representation plays an important role in the way that a complete system integrates its components. The Japanese FGCS project emphasizes logic representations and stresses the development of separate knowledge-base and inference engines, which are integrated after their separate development. This viewpoint of the system architecture is illustrated in Figure 2a.

An alternative perspective is seen from the design of the Connection Machine. System design more closely reflects a semantic-network knowledge representation and memory-based reasoning. All the knowledge in the system is embodied in a large collection of facts; no intentional knowledge, or rules, are employed. For this type of system, the database is implemented directly on the architecture, and inferences are carried out in software and message exchanges. This perspective of system design is illustrated in Figure 2b.

The last design perspective is that of a hierarchical nature as shown in Figure 2c. An inference machine handles all meta-level inferences, while a knowledge-base computer deals with rules manipulating domain knowledge. A separate database computer carries out search and selection operations on the domain knowledge.

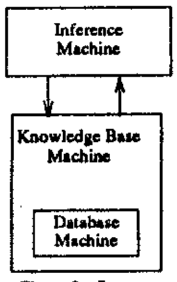

Figure 2a. Japanese FGCS Model

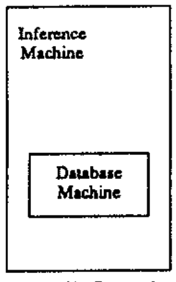

Figure 2b. Connection Machine Model

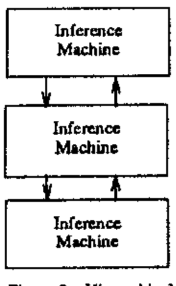

Figure 2c. Hierarchical Model

Following this approach, subsystem-level architectures for data and knowledge base management, and inference are discussed. Subsystem architectures based on neural networks are also presented. Neural networks are interesting due to their radically different approach to processing.

Database Architectures. Early studies on specialized database architectures emphasized the use of parallelism. Such designs include CASSM, RAP, and DIRECT[20]. Some later systems, such as the Connection Machine, are designed with massive parallelism for symbolic applications and can be applied for a number of specialized database functions[15]. A comparison of several early parallel database computers and a survey of commercially available database computers can be found in the references[20,32].

The fact that these systems are not commercially popular indicates their limited effectiveness. In fact, the bottleneck in database retrieval has proven to be disk input/output and not processor cycles[46]. As a result, very intelligent database processing using application-dependent knowledge and indexing may be preferable to massive parallelism[46].

Knowledge Base Architectures. The objectives and requirements of a knowledge-base computer are different from those of a database architecture. A knowledge base deals with a higher level of abstraction, and classes of objects as opposed to a simple collection of data and facts in a database. The nature of items in a knowledge base is more complex than those in a database. In addition, knowledge changes less frequently, and is usually more compact than the large set of data usually stored in conventional databases.

A knowledge-base subsystem should have a mechanism for either rejection of or truth maintenance for the insertion of inconsistent data or rules. Support for the inference mechanism is also desirable and may take the form of an automatic rule-selection mechanism, logic support, or special hardware for operations such as joins and projections of relations. Finally, the interface to the host computer should preferably be intelligent and may draw on the resources of the knowledge-base system.

The Japanese FGCS project is developing Delta, a combined knowledge and database computer. The motivation behind a specialized design is that the integrated system calls for a performance that cannot be met by commercial components. The system consists of a Control Processor, Relational Database Engine, and a Hierarchical memory. The Control Processor translates commands (received from an Interface processor that communicates with a parallel inference machine) into subcommands that the Relational Database Engine can perform. The Relational Database Engine operates through a data-path with the Hierarchical Memory, which is composed of semiconductor and magnetic disks[38].

Inference Machines. Inference engines are a key component of knowledge processing architectures. The structure of the inference engine is highly dependent on the knowledge representation and programming language employed. An important problem in designing inference machines is the architectural supports for searching the knowledge base. Deduction and search have been the dominant paradigms for symbolic machine inference over the last 30 years. An overview of approaches and research issues in this type of inference architecture can be found in the references[49].

Microlevel hardware features discussed in Section III.A are often major components of the inference subsystem. Dataflow, control-flow, and demand-flow techniques have been employed in designing inference machines for Lisp and logic. Although the theory behind data-driven and demand-driven computations appears very promising, no clear solution has been developed for many of the design considerations such as the proper granularity and architectural approach. It remains an open problem as to the proper trade-off among demand-flow, data-flow, and control-flow in an inference computer.

Neural Networks. A neural network can be viewed as containing eight components: processing units, state of activation, output function, pattern of connectivity, propagation rule, activation rule, learning rule, and an environment[41]. The first design consideration is on the representation in the set of processing units. There are two possible approaches: local and distributed. A local representation allocates one concept to its own unit. Hopfield's network solving the traveling salesman problem is an example of a local representation on a neural network[26]. While this may be an efficient solution to the problem, it does not perform learning. In contrast, in a distributed representation, each unit participates in the storage of many concepts. In this way, a processing unit in a distributed representation stores an abstract microlevel feature[41].

In a neural network, each unit has an associated activation level. This activation level can be analog, analog and bounded, or discrete. Its output is a function of its activation level. Output functions are usually threshold or sigmoidal functions. The output of each unit is distributed to a set of processing units. The emphasis that the outputs from other units have on a unit is determined by the weight of the connection between units. Finally, a node's activation is modified by the activation rule which is a function of inputs, their relative connection strengths, and the current activation level.

Neural networks have been proposed for a variety of applications. A neural network can be easily adapted as associative memory with capability for inexact matching. Its strengths include both speed and accuracy in the presence of noise. At a more complex level, neural networks can be used for speech recognition, and vision tasks such as letter recognition. Neural networks have been applied for solving combinatorial search problems, such as the traveling-salesman problem, although solutions for large search problems on neural networks are not yet of as high quality as those of good digital algorithms.

The design of neural networks is still plagued by a number of difficult problems. First, a neural network must be trained for a given application and must be re-trained when the system parameters change. There is no systematic method to generalize a neural network trained for one application and apply it for another application. Second, all the learning algorithms known today require extensive amount of training for good performance. For example, over 9,000 learning sweeps, each of which has 40 phases, are required to train a shifter network using the Boltzmann Machine learning algorithm[25]. Moreover, the learning speed depends on the configuration of the neural network, which cannot be selected systematically. Third, extremely large neural networks cannot be built with the current technology. Finally, with the limited size of neural network, it is necessary to partition the problem so that part of it can be learned by traditional methods and the other part by neural processing. However, knowledge representations in neural networks are drastically different from procedural and declarative representations in conventional symbolic processors. Systematic methods to integrate them are still missing.

IV. SYMBOLIC SUPERCOMPUTING SYSTEMS

This Section is a survey of existing supercomputing machines for symbolic processing. The popularity of symbolic processing has risen at the same time as a great deal of interest in parallel processing has emerged. As a result, designs for symbolic processing supercomputers have the common feature of endeavoring to exploit parallelism. The techniques for exploiting parallelism, however, vary widely. First, the methods of communication and synchronization in parallel symbolic processors are discussed. The rest of this section is devoted to discussions of specific symbolic processing supercomputers.

In Section II, it was seen that many of the software architectures were based on an underlying technique for representing knowledge. In Section III, it was illustrated that many of the hardware features are based on previously developed software architectures. This effect is continued in development of complete systems; most real systems are developed for an existing knowledge representation or software language. For this reason, the Section is divided into subsections on systems for: functional, logic, rule-based, object-oriented and connectionist representations.

Communication. Message passing, marker passing, and value passing are three predominant communication methods in parallel symbolic processors. Message passing is the conventional way of communication in which the information to be communicated is formulated into a message and sent over the interconnection network. The computing elements are generally complex, and the communication costs are high. Message passing is popular in many parallel symbolic processors.

Marker passing refers to the transfers of single-bit markers from one processor to another. A marker indicates the presence of a given property, and a set of markers indicate the conjunction of a set of properties. Each processor is simple and can store a few distinct marker bits. There is never any contention: if two markers arrive at the same destination, they are simply ORed together. Marker passing is especially suitable for implementing semantic networks and recognition problems in hardware. One such system is NETL[13]. The Connection Machine was originally designed as a marker-passing system, but was modified later to carry out more powerful processing and communication features[15].

The last method of communicating information is value passing. In this form, information is passed as continuous quantities. Only simple operations are performed on these numbers. The salient feature of this approach is that if several values arrive simultaneously at one point, they are combined into a single output by a mathematical function. Consequently, there is no contention in information transfer. Examples of value passing systems include the Boltzmann machine and other neural computation systems. Iterative relaxation techniques for problems such as low-level vision, speech understanding, and optimization all seem to be suited to value-passing architectures.

Synchronization. Synchronization refers to the control of concurrent accesses to shared items in a parallel processing system. It is important in message-passing systems because messages may result in contention for shared resources. In contrast, synchronization is not critical in marker-passing or value-passing systems because there is a predefined method of passing markers and combining values so that contention will not occur.

Synchronization is important when there are shared data items. In a program written in a procedural language, the order of statements dictates the order of execution. If two statements share a variable, the first is to be executed prior to the second. Therefore, synchronization control is implicitly defined by the order of statements when data sharing is necessary. In contrast, the order of execution in a program written in a pure declarative language is not defined. When two tasks share a common variable, the order of execution is indeterminate. As a result, explicit specification of synchronization control is needed when data sharing is present. However, most declarative languages lack facilities for explicit specification of synchronization control.

Synchronization can be carried out by shared memory or message passing. Shared memory is popular and has been used in systems such as Aquarius[11], Concurrent Lisp Machine[47], Concert MultiLisp Multiprocessor[19], and Parallel Inference Engine[36]. Blackboard architectures and shared variables are two applicable techniques for shared-memory synchronization.

There are three components in a blackboard architetcture: a knowledge source, a blackboard, and control[39]. The knowledge processing technique to solve the problem is partitioned into separate knowledge sources. The data, including input/output and partial solutions, are stored in the blackboard. The blackboard may be partitioned into smaller blackboards, forming a hierarchy of solution spaces. Knowledge sources manipulate the data in the blackboard in order to reach a solution. The only communication mechanism between these knowledge sources is the blackboard. A monitor is present to ensure that only one knowledge source is changing the blackboard at any time.

A more powerful blackboard architecture has been proposed in which control information (or meta-knowledge) is allocated a separate blackboard[21]. This approach is more flexible and suits the nondeterministic nature of symbolic processing.

Synchronization may also be achieved through shared-memory variables. Lisp languages that have been modified for parallel processing often contain shared variables for synchronization. Multilisp provides a mechanism for waiting for values to be generated in the future.

Synchronization in a message passing system is accomplished through a protocol implemented in hardware or software. In a standard message passing environment, the messages may be of arbitrary complexity. This is more appropriate, for example, in systems using an object-oriented programming technique.

IV.A. Parallel Functional Programming Computers

Lisp has enjoyed the longest tenure in the main stream of languages for symbolic processing. Consequently, the majority of special-purpose parallel processors designed to support functional languages are oriented toward Lisp or its dialects. Examples include Concert[19], EM-3[52], and a multi-microprocessor Concurrent Lisp system developed at Kyoto University[47]. In all these systems, users are required to specify to some extent the tasks to be decomposed. Compilers for automatic detection of parallelism in sequential Lisp programs is an area of active research. Table 5 presents some of the more publicized parallel systems for functional programming techniques.

The majority of computers that are designed for general-purpose applications have only a few features specifically appropriate to symbolic processing. As a result, the inference engine and knowledge base are not separated and are almost exclusively implemented by sophisticated software structures. Lisp is added as one of the several languages to

Table 5. Notable Parallel Functional Programming Computers.

Machine	Year/Status*	Primary Language	Interconnection	Communication	Features
Butterfly	1985/CA	Multilisp	Butterfly Switch	Shared memory	256 MC68000-series PEs; homogeneous, tightly coupled; general-purpose multiprocessor.
iPSC	1986/CA	Common Lisp	Hypercube	Message passing	256 Intel 80286 and 80386 processors; no shared environment; user decomposes program into concurrent processes that communicate by messages; general purpose multiprocessor.
Connection Machine	1986/CA	*Lisp, CM-Lisp	Hypercube	Message passing	Model CM-2 has 4096 bit-serial processors; users annotate Lisp programs for SIMD or multiple SIMD parallel processing; C*, Fortran, and Paris (CM-2 assembly language) are also supported.
Concurrent Lisp Machine	1983/PO	C-Lisp	Multiple busses	Shared memory	17 MC68000-series PEs; special cell interface; control stack; garbage collector.
EM-3	1984/PO	EMLISP	Modified Delta network	Message passing	List-like data-driven language; 16 MC68000-series PEs; special router chip; control for function evaluation.
Concert	1986/PO	Multilisp	Ringbus	Shared memory	32-64 MC68000-series PEs; network is segmented bus in shape of ring.
Alice	-/UC	Hope, Lisp, Prolog	Cluster of processors, ring buffer.	Message passing	Transputer as basic processor; reference counter for garbage collection.

* Status:
CA: Commercially available; PO: Prototype operational; HS: Hardware simulated;
UC: Under construction; SI: Simulations completed; PD: Paper design;

Table 6. Notable Parallel Computers for Logic Representations.

Machine	Year/Status*	Interconnection	Communication	Features
Aquarius	-/UC	Bus and crossbar	Shared memory	Heterogeneous MIMD; 16 PEs; synchronization through Goodman Cache; crossbar to shared memory modules; special Prolog, floating-point, and I/O processors.
Parallel Inference Engine (PIE)	1984/SI	Switching network	Shared memory	100's to 1000's of inference units; goal rewriting model; OR-parallelism; sequential AND processing; Activity Controllers to control inference tree; Unify Processors connected to Definition Memory containing program.
PIM-R	1986/HS	Multistage network	Shared memory	Many Inference Modules connected to Structure Memory units through network; structure copying.
PIM-D	1986/HS	Multistage network	Shared memory	Multiple PEs connected to Structure Memory; unfolding interpreter; asynchronous communication; streams for nondeterministic control.
PIM-I	-/UC	Hierarchy	Shared memory, message passing	100 PEs; 8-PE clusters interconnected with shared memory and parallel cache.

accompany their parallel computers. A Lisp compiler is used for decomposing tasks for parallel processing, and users are required to annotate tasks in various degrees. Examples of commercial multiprocessors include the Butterfly[9], Connection Machine[7], and the Intel iPSC concurrent computer[3].

On the Connection Machine, special chips containing 16 bit-serial processors and router circuits are developed. *Lisp in CM-2 allows users to specify a parallel variable (pvar), which is a first-class object with value for each processor in the computer[8]. Pvar's can be accessed concurrently (with possible masks) by all local or remote processors in a SIMD or multiple SIMD mode. CM-Lisp is a dialect of Common Lisp extended to allow fine-grained, data-oriented parallel processing. It provides higher-level data abstractions called zappings, which are similar in structures to arrays or hash tables. Broadcasts, reductions, and combinations can be specified.

Other general-purpose computers with limited support for parallel symbolic processing, such as ZMOB, are being developed at universities[51]. These computers can be viewed not as symbolic processing computers, but as general-purpose computers that are appropriate for both numeric and symbolic computations.

IV.B. Parallel Logic Architectures

In this section, parallel systems suitable for evaluating logic programs are presented. A summary of notable projects is shown in Table 6. There are two significant parallel logic systems developed at universities. The Aquarius multiprocessor developed at University of California, Berkeley, emphasizes a coupling of intensive numeric calculations and symbolic manipulations[11]. It intends to utilize parallelism at all levels of computation and considers cost as secondary to performance considerations. Another university project is the Parallel Inference Engine (PIE), being developed at the University of Tokyo[36]. The target is 1,000 processors, and a speedup of 170 has been estimated for 256 processors. PIE utilizes OR-parallelism only.

Probably the most massive effort at the development of parallel logic systems is contained in the Japanese Fifth Generation Computing System project (FGCS). The project discerns three major areas of development: problem solving and inference machines (hardware), knowledge-base management systems (software and algorithms), and an intelligent man-machine interface[35]. The project is divided into three stages. The initial stage explores basic computer technology and processing techniques. This stage has been completed. The middle stage is for the development of subsystems, and the construction of experimental subsystems. The final stage is devoted to the development of the complete system.

The initial-stage designs of the Parallel Inference Machines (PIM) were based on two concepts: reduction and dataflow (PIM-R, and PIM-D, respectively). The architectures for these computers were similar, but the technique for evaluation reflected these two philosophies. PIM-R and PIM-D have both been hardware simulated. PIM-I is a hardware design for the intermediate stage of the FGCS project. The target speed for the 100-processor PIM-I is 10 to 20 MLIPS, with target speed of 200-500 KLIPS

Table 7. Notable Parallel Machines for Production, Object-Oriented, and Connectionist Systems.

Machine	Year/Status* Type	Interconnection	Communication	Features
DADO1, DADO2	1986/PO Prod. Sys.	Binary tree	Message passing	DADO2: 1023 8-bit processors; 16K user memory; two modes: MIMD, and multiple SIMD; special I/O circuits.
Non-Von	1985/PO Prod. Sys.	Binary tree with leaf connections; connections to LPEs.	Message passing	Binary SIMD tree of Small PEs; leaves are connected in mesh; Large PEs connected by network, with connections to high-level nodes in tree; intelligent disk drives connected to LPEs.
PSM	1986/SI Prod. Sys.	Shared buses	Shared memory	32-64 processors; parallel Rete Match algorithm; PEs connected to memory modules through cache; local memory; hardware task scheduler.
FAIM-1	-/UC Obj.-Or.	Hexagonal mesh	Message passing	PEs = Hectogons; heterogeneous shared-memory multiprocessor: Instruction Stream memory, Post Office communication processor, Evaluation Processor, and others; three-port switch at edge of array for I/O and wrapping of connections.
Dragon	-/UC Obj.-Or.	Bus	Shared memory	Up to 10 32-bit workstation/processors; tightly coupled; associative cache at each processor.
Apiary	1980/PD Obj.-Or.	Single-stage network	Message passing	Implements Actor model; computations in the Actor model are partial orders of events with no assignment commands.
NETL	1979/PD Conn.	Multi-Level switching network	Marker passing	For semantic networks; million processors, each can store 16 markers; simple PEs; only Boolean functions.
THISTLE	1983/PD Conn.	Multi-level switching	Value passing	For semantic networks; similar to NETL, only with 8-bit value passing.
Connection Machine	1986/CA Conn.	Hypercube	Message passing	General-purpose SIMD and multiple SIMD processing; can be programmed for marker passing operations in semantic networks and simulating neural networks.
SNAP	1985/PD Conn.	Mesh with global bus	Message passing	Square array of identical processors; CAM in each PE for relationships between nodes; Communication Unit; Processing Unit.

* Refer to Table 5 for explanation of status.

for the individual processors[48]. The machine language for this computer will be KL1-B, which is based on Guarded Horn Clauses. The software development will be done on a network of PSI (Personal Sequential Inference) systems (multi-PSI)[28].

IV.C. Parallel Systems for Production Systems Computations

The exploration of computers for production systems has been carried out primarily at universities. Table 7 presents a summary of these projects.

The DADO1 and DADO2 projects[45] at Columbia University develop a class of computers based on tree architectures. The upper-level nodes synchronize and select rules, intermediate nodes match and store rules, and the leaves are for the working memory.

Another project at Columbia University is the Non-Von computer (Non-Von-1 is an earlier version)[24,27]. Unlike DADO2, Non-Von uses Smaller Processing Elements in a binary tree, subject to the control of Large Processing Elements. Most of the pattern matching tasks that are done in the Working Memory have small granularity and are more suitable to be executed on a large number of Small Processing Elements.

Finally, the PSM computer is a larger-grain machine that is specifically designed to support the OPS5 system and a parallel Rete-Match algorithm. Simulations have shown promising speedups, and that 32 processors are sufficient to exploit most parallelism for this system[17].

IV.D. Parallel Object-Oriented Architectures

Most development work on object-oriented programming has been on computers not specifically designed for object-oriented computation, such as the Intel iPSC. Two notable multiprocessors are designed specifically for object-oriented computations. FAIM-1[2] is a multiprocessor with special RISC processors connected by a hexagonal mesh. OIL, an intermediate language, was first developed for interfacing modules. However, the scope of the project has recently been changed to exclude the development of OIL and base programming of the computer on MultiScheme. The Dragon project is another object-oriented design project, but supports only 10 processors[34]. Table 7 shows a summary of the notable projects.

IV.E. Connectionist Processing

The connectionist implementations focus on correlations between nodes in a graph containing the knowledge and have been designed primarily for semantic networks. Other connectionist designs of interest are the neural networks. Unfortunately, current technology for design of neural networks have precluded development of neural networks in a roll greater than that of subsystem (see Section III.B).

The four designs for connectionist system for the implementation of semantic networks correspond to the three types of message passing environments. NETL utilizes the most elementary processing elements in a marker passing system[13]. THISTLE is a similar design, but employs value passing instead of marker passing[13]. The Connection Machine can be programmed to simulate marker passing in semantic networks and value passing in neural networks using massive data-level parallelism[7]. Finally, SNAP relies on message passing[33]. These machine are summarized in Table 7.

VI. REFERENCES

1. L. Aiello, C. Cecchi, and D. Sartini, "Representation and Use of Metaknowledge," *Proc. of the IEEE*, pp. 1304-1321, Oct. 1986.
2. 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, IEEE, Jan. 1987.
3. D. Billstrom, J. Brandenburg, and J. Teeter, "CCLISP on the iPSC Concurrent Computer," *Proc. Sixth Int'l Conf. on Artificial Intelligence*, pp. 7-12, AAAI, Seattle, WA, 1987.
4. R. Brooks and R. Lum, "Yes, An SIMD Machine Can Be Used For AI," *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 73-79, 1985.
5. L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5*, Addison Wesley Publishing Co., Reading, MA, 1985.
6. K. Clark and S. Gregory, "Note on System Programming in PARLOG," *Proc. Int'l Conf. Fifth Generation Computer System*, pp. 299-306, ICOT and North-Holland, 1984.

7. Thinking Machines Corporation, Connection Machine Model CM-2 Technical Summary, Tech. Rep. HA87-4, Cambridge, MA, April 1987.

8. Thinking Machines Corporation, **Lisp Reference Manual*, Version 4.0, Cambridge, MA, Oct. 1987.

9. A. S. Cromarty, "What Are Current Expert System Tools Missing?," *Proc. COMPCON Spring*, pp. 411-418, IEEE, 1985.

10. M. F. Deering, "Architectures for AI," *Byte*, pp. 193-206, McGraw-Hill, April 1985.

11. A. M. Despain and Y. N. Patt, "Aquarius--A High Performance Computing System for Symbolic/Numeric Applications," *Proc. COMPCON Spring*, pp. 376-382, IEEE, Feb. 1985.

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

13. S. E. Fahlman and G. E. Hinton, "Massively Parallel Architectures for AI: NETL, THISTLE, and BOLTZMANN Machines," *Proc. National Conf. on Artificial Intelligence*, pp. 109-113, AAAI, 1983.

14. S. E. Fahlman, "Parallel Processing in Artificial Intelligence," *Parallel Computing*, vol. 2, pp. 283-286, 1985.

15. S. E. Fahlman and G. E. Hinton, "Connectionist Architecture for Artificial Intelligence," *Computer*, vol. 20, no. 1, pp. 100-109, IEEE, Jan. 1987.

16. J.-L. Gaudiot, S. Lee, and A. Sohn, *Data-Driven Multiprocessor Implementation of the Rete Match Algorithm*, Tech. Rep., Dept. of Computer Science, University of Southern California, Los Angeles, CA, 1987.

17. A. Gupta, C. L. Forgy, A. Newell, and R. Wedig, "Parallel Algorithms and Architectures for Rule-based Systems," *Proc. 13'th Int'l Symp. on Comp. Arch.*, pp. 28-37, IEEE/ACM, 1986.

18. R. Halstead, "Parallel Symbolic Computing," *Computer*, vol. 19, no. 8, pp. 35-43, IEEE, Aug. 1986.

19. R. Halstead Jr., T. Anderson, R. Osborne, and T. Sterlig, "Concert: Design of a Multiprocessor development System," *Proc. Int'l Symp. on Comp. Arch.*, pp. 40-48, IEEE/ACM, June 1986.

20. P. B. Hawthorn and D. J. DeWitt, "Performance Analysis of Alternative Database Machine Architectures," *Trans. on Soft. Eng.*, vol. SE-8, no. 1, pp. 61-75, IEEE, Jan. 1982.

21. B. Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251-321, North-Holland, July 1985.

22. C. Hewitt and H. Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence," *Proc. COMPCON Spring*, pp. 418-423, IEEE, Feb. 1984.

23. Y. Hibino, "A Practical Parallel Garbage Collection Algorithm and Its Implementations," *Proc. 7th Annual Symp. on Comp. Arch.*, pp. 113-120, IEEE/ACM, May 1980.

24. B. K. Hillyer and D. E. Shaw, "Execution of OPS5 Production Systems on a Massively Parallel Machine," *J. Par. and Dist. Proc.*, vol. 3, no. 2, pp. 236-268, Academic Press, 1986.

25. G. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines," in *Parallel Distributed Processing*, ed. D. E. Rumelhart, J. L McClelland, vol. 1, MIT Press, Cambridge, MA, 1986.

26. J. J. Hopfield and D. W. Tank, *Disordered Systems and Biological Organization*, Springer Verlag, 1986.

27. H. A. H. Ibrahim, J. R. Kender, and D. E. Shaw, "Low-Level Image Analysis Tasks on Fine-Grained Tree-Structured SIMD Machines," *J. Par. and Dist. Proc.*, pp. 546-574, Academic Press, Dec. 1987.

28. N. Ichiyoshi, T. Miyazaki, and K. Taki, "A Distributed Implementation of Flat GHC on the Multi-PSI," *Int'l Conf. on Logic Programming*, 1987.

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

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

31. G. J. Li and B. W. Wah, "Optimal Granularity of Parallel Evaluation of AND-Trees," *Proc. Fall Joint Computer Conf.*, pp. 297-306, ACM/IEEE, Nov. 1986.

32. F. J. Malabarba, "Review of Available Database Machine Technology," *Proc. Trends and Applications*, pp. 14-17, IEEE, 1984.

33. D. I. Moldovan and Y. W. Tung, "SNAP: A VLSI Architecture for Artificial Intelligence Processing," *J. Par. and Dist. Proc.*, pp. 109-131, Academic Press, May 1985.

34. L. Monier and P. Sidhu, "The Architecture of the Dragon," *Proc. COMPCON*, pp. 118-121, Spring 1985.

35. T. Moto-oka, "Overview to the Fifth Generation Computer System Project," *Proc. 10th Annual Int'l Symp. on Comp. Arch.*, pp. 417-422, IEEE/ACM, June 1983.

36. T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The Architecture of a Parallel Inference Engine (PIE)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479-488, ICOT and North-Holland, 1984.

37. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *Trans. on Computers*, vol. C-28, no. 6, pp. 384-394, IEEE, June 1979.

38. 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, IEEE, June 1985.

39. H. P. Nii, "Blackboard Systems, Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective," *AI Magazine*, pp. 82-106, AAAI, Aug 1986.

40. C. V. Ramamoorthy, S. Shekhar, and V. Garg, "Software Development Support for AI Programs," *Computer*, pp. 30-42, IEEE, Jan. 1987.

41. D. D. Rumelhart, G. Hinton, and J. L. McClelland, "A General Framework for Parallel Distributed Processing," in *Parallel Distributed Processing*, ed. D. E. Rumelhart, J. L McClelland, vol. 1, MIT Press, Cambridge, MA, 1986.

42. E. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25-48, OHMSHA Ltd. and Springer-Verlag, 1983.

43. Y. Shih and K. B. Irani, "Large Scale Unification Using a Mesh-Connected Array of Hardware Unifiers," *Proc. Int'l Conf. on Par. Proc.*, pp. 787-794, Pennsylvania State Univ. Press, St. Charles, IL, 1987.

44. S. J. Stolfo and D. P. Miranker, "The DADO Production System Machine," *J. Par. and Dist. Proc.*, vol. 3, no. 2, pp. 269-296, Academic Press, 1986.

45. S. J. Stolfo, "Initial Performance of the DADO-2 Prototype," *Computer*, pp. 75-84, IEEE, Jan. 1987.

46. H. S. Stone, "Parallel Querying of Large Databases: A Case Study," *Computer*, pp. 11-21, IEEE, Oct. 1987.

47. S. Sugimoto, K. Agusa, K. Tabata, and Y. Ohno, "A Multi-Microprocessor System for Concurrent Lisp," *Proc. Int'l Conf. on Par. Proc.*, pp. 135-143, IEEE, 1983.

48. S. Uchida, "Inference Machines in FGCS Project," *Proc. VLSI'87 Int'l Conf.*, IFIP TC-10, WG 10.5, Aug. 1985.

49. B. W. Wah, M. B. Lowrie, and G.-J. Li, "Computers for Symbolic Processing," *Invited Paper, Proc. of the IEEE*, Submitted Dec. 1987.

50. D. L. Waltz, "Applications of the Connection Machine," *Computer*, vol. 20, no. 1, IEEE, Jan. 1987.

51. 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, IEEE, Feb. 1985.

52. Y. Yamaguchi, K. Toda, J. Herath, and T. Yuba, "EM-3: A Lisp-Based Data-Driven Machine," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 524-532, ICOT and North-Holland, 1984.

53. H. Yasuura, On the Parallel Computational Complexity of Unification, T.R. ER 83-01, Yajima Lab, Kyoto University, Japan, Oct. 1983.