LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR -85-810

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE A DISTRIBUTED IMPLEMENTATION OF FUNCTIONAL PROGRAM EVALUATION

LA-UR--85-810

DE85 009573

AUTHOR(S) Joseph H. Fasel, C-10
Robert J. Douglass, C-10
Randy Michelsen, C-10
Paul Hudak, Yale University

SUBMITTED TO Paper to be presented at AI-85, Long Beach, CA, April 30, 1985

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

FORM NO 836 R4
ST NO 2629 5/81

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

A DISTRIBUTED IMPLEMENTATION OF FUNCTIONAL PROGRAM EVALUATION

by

Joseph H. Fasel, Robert J. Douglass, Randy Michelsen[1]
University of California
Los Alamos National Laboratory
Los Alamos, New Mexico

and

Paul Hudak[2]
Yale University
New Haven, Connecticut

1. INTRODUCTION

1.1. Artificial Intelligence and the Need for Parallel Processing

Tremendous industrial and governmental interest in artificial intelligence has been sparked by successful applications involving the use of expert systems in such fields as geology, chemistry, medicine, and computer engineering; by the application of natural-language and speech processing to provide a more human-oriented interface for computerized information systems; and by the development of computer vision systems for industrial robotics and military guidance and surveillance applications.

AI applications developed to date are relatively small, yet they still consume a large quantity of computer resources and run rather slowly on conventional computer architectures. By the 1990s the Strategic Computing Plan of the U.S. Department of Defense's Defense Advanced Research Projects Agency (DARPA) anticipates expert systems with 30,000 rules executing at about 12,000 productions per second, as opposed to today's machines, which execute about 25 to 50 productions per second [7, 13]. In the area of computer vision, DARPA estimates that "one trillion von Neumann operations per second are required" to perform their applications and that 20 billion operations per second will be required for some speech understanding applications [7]. For natural-language processing, DARPA envisions the need for "massively parallel computational devices."

The Japanese Fifth Generation Project is even more ambitious, aiming at expert systems containing more than 10,000 Horn clauses and executing at the rate of 50 million to 1 billion logical inferences per second (LIPS), as opposed to their estimates of today's machines, which run at 10 to 100 thousand LIPS [26]. Other new industrial and government programs have sprung up with similar, if not as specifically quantified, goals; for example, the Microelectronics and Computer Technology Corporation in the U.S. and the ESPRIT program in Europe [12, 2].

The traditional von Neumann architecture is approaching execution speed limits imposed by the speed of light and the time to communicate bits of information electronically within a single processor. Clearly, obtaining the required speeds will require massive parallelism in the execution of artificial intelligence programs. Dozens of papers are entering the literature, advocating different approaches to achieving the necessary parallelism, many of which involve special-purpose parallel processors [10, 11, 28, 9, 27, 30, 5, 14].

---

[1]Work done under the auspices of the U.S. Department of Energy

[2]Research supported in part by NSF Grant MCS-8302018

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

1.2. The Functional and Logic Models

A common premise of fifth-generation computing is that the traditional "word-at-a-time" von Neumann computing model is inadequate as a foundation for parallel computing. Extending such a sequential model to the parallel world is like putting on a shoe that doesn't fit – it makes much more sense to begin with a foundation that has a nonsequential semantic base. There are several computing models that fit this more demanding description, but the two most popular ones are the *functional model* (based on *reduction* in the lambda calculus) and the *logic model* (based on *resolution* in first-order predicate calculus). Both offer several important advantages over conventional computing models. For example, their lack of side effects makes synchronization issues trivial, and the lack of a single "thread of control" simplifies the manifestation of parallelism. Both models promise to be valuable as a basis for fifth-generation computing.

It is not entirely clear which of these two models is better, and indeed it is only through ongoing research that we may be able to shed some light on the question. In this paper we will explore the details of a strictly *functional model* called *DAPS* (for Distributed Applicative Processing Systems). DAPS represents the culmination of our research in graph-reduction techniques on parallel architectures. Although the logic model has its clear advantages (such as the fact that resolution is a superior mechanism for variable binding in function invocations), we see the following as advantages of the functional model:

(1) The functional model is able to support higher order functions (that is, functions passed as arguments or returned as values from expressions), whereas the typical logic model restricts programs to first-order logical statements for reasons of efficiency.

(2) The generality of resolution makes it less efficient for the most common operations, which are typically no more complex than a conventional procedure call.

(3) The functional model can accomplish *controlled* search by enumerating alternatives, rather than relying on the automatic backtracking in unification, which the programmer often wishes to defeat.

(4) Many of the features that we will describe as important to DAPS apply equally to logic computing models. Indeed, we feel that we can support logic programming in our model by using one of several reduction-oriented approaches to resolution.

1.3. Summary

In this paper, we explore the potential of the functional model, particularly as it pertains to architecture. In Section 2, we describe the *graph-reduction* operational model of computation and its relation to AI problems. In Section 3, we discuss a class of architectures that implement graph reduction and a prototype implementation in this class being developed at Los Alamos. Finally, we speculate on the applicability of graph reduction to some other classes of architecture.

2. GRAPH REDUCTION AND AI

2.1. Graph Reduction

We have argued that the functional computing model is one of the most viable foundations for parallel computing. (Indeed, it is on this same foundation that research on dataflow machines lies.) Functional program evaluation strategies are derived from the lambda calculus [3], whose nonsequential nature is quite suitable for parallel computing. In particular, these strategies exhibit the well-known *Church-Rosser Property*, which indirectly states that no matter what computation order is chosen in executing a program, it is *guaranteed* to return the same result (assuming termination). This marvelous determinacy is invaluable in parallel systems – it means that programs may be written and debugged in a functional language on a sequential machine, and then the *same* programs may be executed on a parallel machine for improved performance. The key point here is that in functional languages the parallelism is *implicit* and supported by the underlying semantics. There is *no* need for special message-passing constructs or other communications primitives, *no* need for synchronization primitives.

and in general *no* need for special "parallel" constructs such as **parbegin...parend**.

Given that implementation of the functional model is a reasonable goal to pursue, what kind of physical architecture is most appropriate? That is, given that there are several reducible expressions (redexes) in a program graph, what mechanism should be used to direct processing power to those locales? There are, of course, many possibilities, but early on we rejected several ideas, including dataflow (since we view the distribution strategies as too fine-grained), multiprocessors with shared memory (since we view the memory as an ultimate bottleneck), and systolic machines (since the indeterminate behavior of recursion appears to be intractable in such a machine). The architecture that we have settled on instead is one in which a large number of processing elements (PEs) having only local store and communicating by messages are interconnected in a homogeneous communications network. This idea is not new and is sometimes referred to as a *network computer* or *ensemble architecture*. It is especially attractive because it appears to be well suited for VLSI design and is extensible; there are a growing number of real machines that fit its description.

Unfortunately, few (if any) of the network computers in existence today have succeeded in extracting a reasonable amount of parallelism from a general class of programs. They have only performed well when the parallel components of the program have been explicitly stated by the user, or explicitly allocated on particular processors, and generally only for problems that have inherent regularity and predictability. No one has demonstrated the overall feasibility of the network computer for accomplishing the parallel execution of a general range of programs, or even for a restricted class of say, AI programs. We feel that the network computer model is indeed a viable approach to "fifth-generation" computer design, but current efforts have been lacking in several respects:

(1) Most network computers lack a coherent *global* computation strategy (such as provided by the functional computing model). The machines are too often treated simply as an array of conventional processors that "communicate by messages," resulting in *ad hoc* solutions to a very difficult problem.

(2) Little attention has been paid to the complex issue of *managing* a highly parallel, decentralized computation, especially in the presence of "eager" and nondeterministic computations. Such issues become extremely complex in the decentralized, distributed environment of the network computer and cannot be taken for granted.

(3) There is little to be learned from the success of parallel machines in scientific computing, since these machines depend heavily on a program's *regularity* and *predictability*. Such program behavior is common in scientific computing, but is rare in AI and other more data dependent computations. New techniques need to be developed to deal with dynamic, unpredictable behavior.

The DAPS model directly addresses all of the issues raised above. Physically, it can be viewed as a network computer. But logically, it has a coherent global computation strategy derived from the lambda calculus, together with effective mechanisms for managing the logistics of a highly parallel computation. All control is entirely decentralized, including memory and task management, thus avoiding any bottlenecks to parallelism. Furthermore, it uses a technique called *diffusion scheduling* to handle the dynamic and unpredictable spawning of parallel tasks during a computation, a feature that is crucial to good performance on the less regular, unpredictable programs that are characteristic of AI and other application areas. The DAPS model is discussed in more detail below.

2.2. The Irregularity of AI Computation

Most AI problems lack regularity and predictability. In contrast to many scientific applications which perform computations on very regular data objects, such as fixed sized arrays, AI programs process complex data structures represented as lists, sets, and graphs that are allocated and deallocated in a variable pattern during execution. Much of the processing in symbolic computation involves following linked lists by tracing chains of pointers through scattered locations in memory. Execution speeds for single serial processors are therefore limited by the time it takes to make random retrievals of individual words of data from memory.

Parallel execution of AI programs must also be dynamic, because for many AI problems the size and form of the computation are determined dynamically as the computation unfolds, and an architecture for AI should reflect this fact. In particular, it needs mechanisms to dynamically spawn concurrent processes, to automatically manage the parallel tasks and their storage needs, and to respond to the changing processing requirements of the computation with load-balancing strategies to shift work from overloaded to idle processors.

Rule-based expert systems provide an example of the irregularity of AI computation. They consist of a set of facts describing a particular problem domain and a set of inference rules that can be applied to the facts to deduce new facts. Rules consist of two parts: a condition part that is a conjunction of test conditions and an action part that specifies changes or additions to be made to the facts if the conditions are satisfied. Starting with a goal that is to be established as true, an expert system searches for some combination of rules and facts that will establish the goal. The form of this search for most expert systems is that of an AND/OR tree. OR nodes represent points in the search where there are alternative rules or facts that might lead to a solution, and AND nodes represent points where all of several test conditions in the condition part of one rule must be satisfied in order to apply the rule [10].

An AND/OR search tree is a very dynamic structure. It is created on-the-fly for each new goal to be established, and it grows and shrinks depending on the particular facts and rules in the knowledge base. Such a computation is markedly different from a typical scientific computation, such as solving a matrix of coefficients for a system of partial differential equations. If AND/OR trees are searched concurrently, the search must be a dynamic one with processes spawned and managed automatically during execution. The graph-reduction model is a good one for such unpredictable computations.

Graph reduction is also quite attractive for more regular AI tasks as well, for example, low-level computer vision and speech processing [11, 28, 9]. Many low-level vision algorithms operate on fixed sized arrays representing the picture elements (pixels) in an image. We have experimented with automatically parallelizing and executing with the DAPS graph-reduction scheme described below a functional specification of a stereo vision disparity algorithm, which matches corresponding parts in a stereo pair of pictures to determine their distance from the camera. We have been able to obtain speedups comparable to or slightly better than the speedups obtained by explicitly hand-coding parallelism for the same algorithm in HEP Fortran. The speedups for the graph reduction were obtained using the DAPS simulator discussed below and compared with parallel speedups obtained using the Denelcor HEP. In addition, the functional specification of the algorithm is about 20 times smaller than the HEP Fortran program and mirrors the original equations of the algorithm definition [11].

3. GRAPH-REDUCTION ARCHITECTURE

3.1. DAPS

In the DAPS functional model, a program is represented as a directed graph whose vertices are labeled with primitive operators and values and whose edges reflect data dependencies between operators and values (*cf.* [1, 4, 8, 22, 24, 25, 31]). Program execution is accomplished via transmutations (called *reductions*) to the graph, which essentially mimic the reduction rules of the lambda calculus. This includes not only relabeling vertices with their ultimate value but also changing the connectivity of the graph, which can result in the implicit deletion of some vertices as well as the explicit creation of new ones. (New vertices, for example, are added as the result of a function invocation.) The graph thus expands and contracts as the computation progresses, as shown in Figure 1.

To support the graph-reduction process there is a single global address space representing a free space of available cells with which the program graph is constructed (and may be thought of as one large *heap*, in the Lisp sense). Each PE is responsible for one contiguous portion of this address space, and thus parallelism comes to bear when concurrent redexes reside in the address spaces of different PEs. A *task queue* is maintained on each PE that essentially contains pointers to vertices in the graph

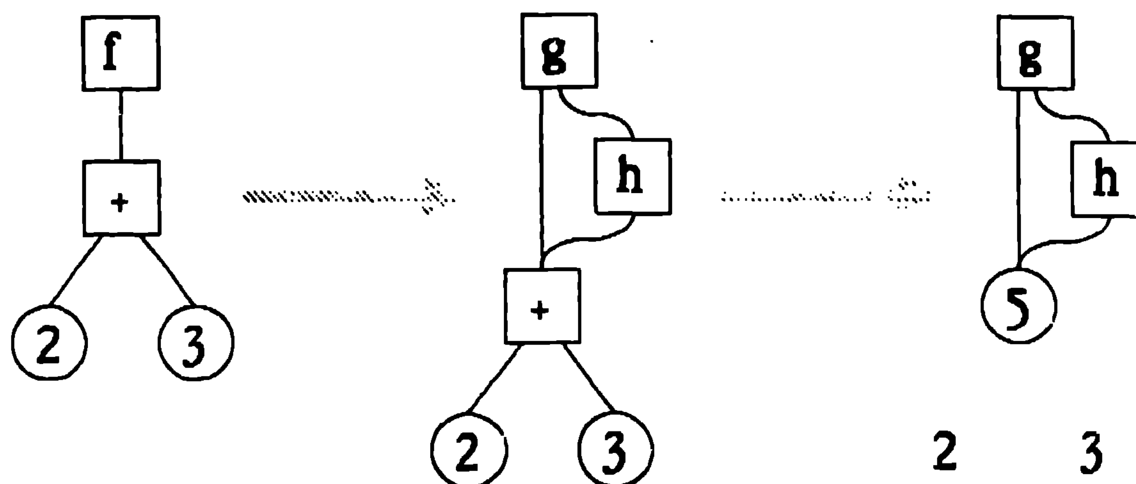

Figure 1: Possible reduction sequence for f(x) = g(x,h(x)).

representing available redexes. Execution occurs by initially mapping the program graph onto th global address space, and as the evaluation unfolds, expanding portions of the graph are allocated on neighboring PEs for increased parallelism. The program thus "diffuses" through the network: the method for doing this is described below.

There are several significant advantages to the graph-reduction approach:

(1) Program and data are the same, simplifying the architecture and facilitating important AI tasks such as graph searches in semantic networks.

(2) It is fairly easy for a compiler to determine the parallel components of a computation graph through straightforward flow analysis.

(3) It is possible to describe important properties of a parallel computation in terms of *graph connectivity*, allowing the development of effective solutions memory and task management problems.

(4) Highly parallel computation involves frequent *context switches*, since breadth-first evaluations are usually performed to maximize the chance for parallelism. Using graph reduction, the overhead for context switching is essentially zero, since all data relevant to the computation are embedded in the program graph.

3.1.1. Memory and task management

Underlying the graph-reduction machinery is a powerful set of housekeeping processes that perform crucial tasks such as garbage collection, irrelevant task deletion, and task prioritization. These processes are based on parallel, decentralized algorithms that execute *concurrently with graph reduction* and are intended to be part of the underlying machinery, not something that the user need be concerned with. (These algorithms have been described in detail elsewhere [17, 18, 19] and are not central to this discussion.)

3.1.2. Serial combinators

An important question in any parallel computing environment is what the "granularity" of the parallel tasks should be. In this section we discuss the notion of *serial combinators* used in DAPS.

A *combinator* [29, 6] is simply a lambda expression that has no free variables and is a *constant applicative form*; that is, it contains only bound variables and constants combined by application. These properties are crucial for a graph reducer, since they allow computed values to overwrite the nodes from which they were derived, without making copies of the bodies of functions. Hughes [21] introduced the notion of a *supercombinator*, derived from a lambda expression by abstracting out the *maximally free expressions*; that is, the largest subexpressions all of whose literals are either constants (including basic functions) or free variables. Together with a few optimizations, the resulting supercombinators have a very useful property: execution of the original program results in a *fully lazy evaluation*, in that no subexpression internal to a combinator body need be recomputed as a result of reapplication of the combinator to the same argument in several different places. This is a generalization of laziness as defined by Henderson and Morris [16], which guarantees only that *arguments in a function call* are never computed more than once. Supercombinator execution is also generally more efficient than a composition of many finer grained combinators.

We have recently developed a *refinement* of a supercombinator called a *serial combinator* with the important additional property that *there is no concurrent substructure*, and there are no larger combinators with the same property. The purpose of this refinement should be obvious: if the combinator has no concurrent substructure, then there is no need to subdivide it further, since doing so can only add communication costs to an already sequential computation. To summarize, we argue that serial combinators are at the ideal level of granularity in the following sense:

(1) They are combinators, facilitating their use in a graph-reduction machine (especially parallel ones).

(2) They result in a fully lazy evaluation, guaranteeing that no extraneous computations are performed.

(3) They have no concurrent substructure, guaranteeing that no available parallelism will be lost.

(4) There are no larger objects having these same properties, so that no extraneous communication costs are incurred because of too fine a granularity.

Serial combinators are therefore the smallest objects that are distributed for parallel execution in DAPS.

3.1.3. Diffusion scheduling

Diffusion scheduling attacks the problem of deciding when and where to distribute new work [20]. This decision is crucial to obtaining good performance, especially in a nearest neighbor topology where communicating to immediate neighbors is cheapest. However, the allocation decisions need to be made *quickly*, at *run time*, and based on *decentralized* (and thus incomplete) knowledge. It is not possible, therefore, to make optimal decisions; heuristics are needed that balance the processors' loads (thus increasing the degree of parallelism) while maintaining locality of reference (thus minimizing communications overhead). Given a PE **p** about to create a new node **n**, the heuristics that underlie diffusion scheduling will allocate **n** either on **p** or on an immediate neighbor of **p**, based on a weighted sum of several factors, possibly including:

(1) The processing load on **p** (that is, the number of tasks waiting for execution on **p**'s task queue).

(2) The memory load on **p** (that is, how much free space is available).

(3) The processing and memory load of each neighboring PE.

(4) A weighted measure of the "direction" of references from **n** to other nodes in the network.

The effect of (1)-(3) is to push work away from busy processors, and the effect of (4) is to draw them toward those to which they have global references (thus maintaining locality of reference). In this way work "diffuses" through the network in the direction of least resistance, as suggested by Figure 2 for a hypothetical two-dimensional network (*cf.* [15, 23]).

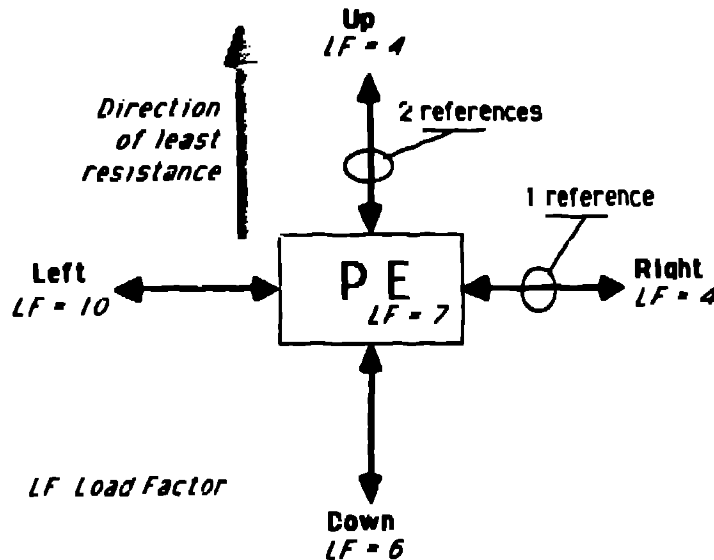

Figure 2: Diffusion scheduler spawns task upward.

3.1.4. Los Alamos implementation

There is currently under development at Los Alamos National Laboratory a system designed to provide a testbed for research in distributed graph reduction. This will allow us to further explore many facets of distributed reduction, including both architectural and language issues. The system is comprised of a group of Symbolics Lisp machines connected by a communications network, with the functionality of communication, scheduling, and reduction embodied in separate processes on each machine. As illustrated by Figure 3, the testbed system consists of a collection of processes replicated on each available Symbolics host.

The *network server* is the communication interface to the network connecting the processors. It distributes incoming messages to the appropriate local processes and accepts messages for transmission to neighboring processors. Typical messages include requests for values of nodes in the locally resident portion of the program graph, values returned as the result of previous requests, or spawned task requests.

The *system scheduler* is an implementation of the decentralized diffusion scheduling algorithm discussed above. Information characterizing the available capacity of neighboring processors, periodically provided by the scheduler's counterparts on these processors, is used for load balancing during the distribution or spawning of tasks. Failure to receive such information from a neighboring processor, indicating probable processor failure, may be used to initiate rescheduling of tasks previously distributed to the effected processor. The scheduler implicitly defines the term "neighboring PE" through its perception of the network topology. This knowledge of adjacency is loaded into the scheduler process as part of the system initialization procedure. This allows us to experiment with different interconnection topologies, even though the communications medium in this testbed implementation is in fact a global bus.

The locally resident portion of the program graph is maintained by the *reduction manager*. As the program graph unfolds during reduction, portions of the graph are dynamically allocated and reduced under control of the reduction manager resident on each processor. This process responds to requests generated as a result of reduction activity, on the local or neighboring processors.

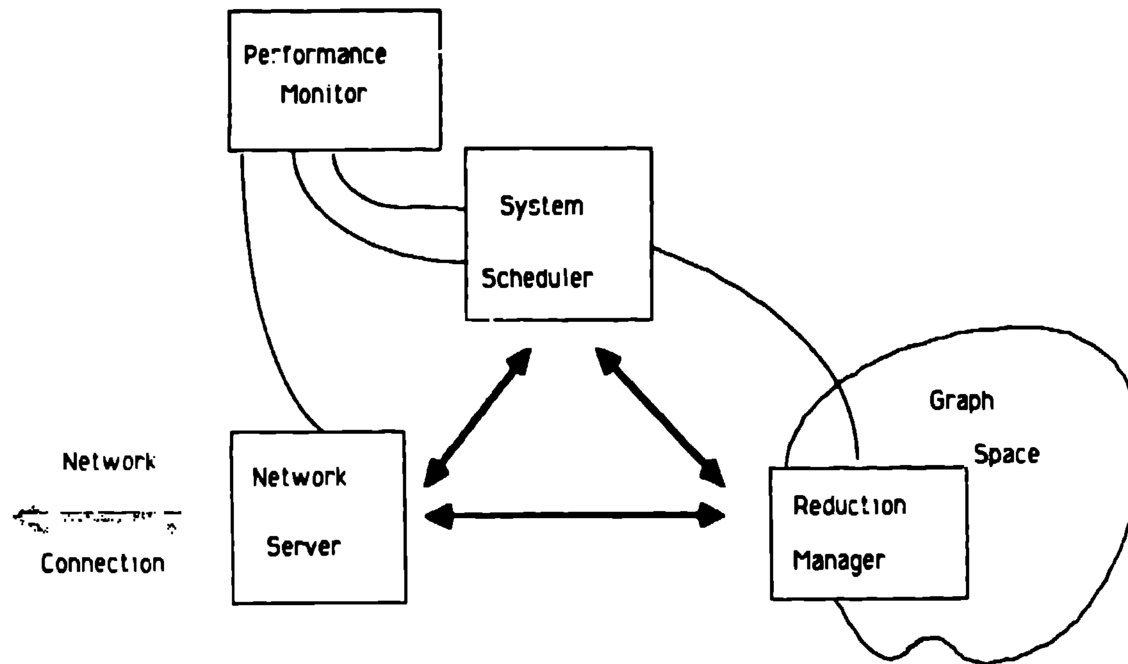

Figure 3: Symbolics implementation block diagram.

The *performance monitor* collects data reflecting performance of the reduction system processes resident on each processor. Locally pertinent information regarding network traffic, reduction activity, and interprocess communication is maintained on each processor. The collection activity is controlled by a software toggle.

This implementation effort provides us the ability to evaluate combinator-based graph reduction in an actual multiprocessor environment. In addition, it allows us to further study related issues in compilation, scheduling, and communication.

3.2. Other Architectures

Having discussed the merits of graph reduction for distributed computing, we must add that we see no difficulty in applying it as well to shared-memory multiprocessor architectures, such as the Denelcor HEP, Cray X-MP, or NYU Ultracomputer. In fact, such a machine could be considered the ideal graph-reduction engine. Instead of the necessity to distribute the graph to the processors, the processors can directly reduce the graph in common memory; the ability to share values, one of the chief virtues of the graph-reduction model, is thus greatly enhanced. The problem of load balancing is vastly simplified; the processors can take work from a common queue of available reductions.

A variation on the shared-memory multiprocessor is a machine such as the BBN Butterfly, in which processor-memory pairs are connected by a global switch. Here, each processor has fast access to its associated memory and relatively slower but uniform access to any other memory. A likely strategy for load balancing in this case is again to use a common queue, but organized such that each processor can perform reductions that can be accomplished in its local memory before looking for work elsewhere.

Shared-memory multiprocessors thus appear to be a hospitable environment for parallel graph reduction, but we are devoting most of our attention to the more difficult problem of distributed computing because we believe that shared-memory architectures cannot ultimately deliver the performance gains that have motivated research in parallel processing. One cannot indefinitely add processors to a common memory without suffering a performance degradation from memory contention or increased memory access time. Indeed, shared-memory architectures that achieve a uniform memory access time generally do so by making this time uniformly worst-case. As these architectures are scaled up to large numbers of processors, this becomes significant. Thus, it appears that the only way to make effective use of massive parallelism is to distribute computing resources and exploit locality in programs to minimize communications overhead.

The attractiveness of shared memory aside from performance problems, together with the ultimate necessity for distribution, suggest that a hybrid approach is reasonable; we construct shared-memory multiprocessors of whatever size is beneficial, and these become the nodes in a distributed network. In such a system, we think of parallelism as being available on two levels: a tightly coupled, perhaps small-grained parallelism within a processing element, and loosely coupled, larger grained parallelism over the network as a whole. A third level should be considered, that of the functional units within a processor; it is here that current vector supercomputers achieve much of their performance, by pipelining these units so that data can be streamed through them at a high rate, and by overlapping the execution of instructions for independent units. For single-instruction, multiple-data (SIMD) parallelism, such processors achieve an execution rate that would be difficult to equal with asynchronous multiprocessing. To the extent, therefore, that SIMD parallelism is a factor in AI problems, we may expect vector processors to be important in fifth generation architecture. Perhaps future high-performance computing systems will be large-scale distributed networks of shared-memory, pipelined vector multiprocessors, like the Cray X-MP. Although the applicability of graph reduction to vector processing has yet to be evaluated, we suspect that any applicative model of computation is likely to be advantageous, considering that vectorization of scientific codes on current vector machines seems to be limited by the ability of optimizing compilers to determine data dependencies to extract parallelism from sequential imperative programs.

REFERENCES

1. Berkling, K. J. Reduction languages for reduction machines. *Proc. 2nd Ann. Symp. Computer Architecture* (1975). IEEE, pp. 133-140.
2. Braggaar, H. ESPRIT: Europe challenges U.S. and Japanese competitors in information technology. *Future Generations Computer Systems 1*, 1 (July 1984).
3. Church, A. *The Calculi of Lambda Conversion*. Annals of Mathematical Studies, vol. 6, Princeton University Press, Princeton, NJ, 1941.
4. Clarke, T. J. W., Gladstone, P. J. S., Maclean, C. D., and Norman, A. C. SKIM -- The S, K, I reduction machine. *Conf. Rec. 1980 Lisp Conf.* (Stanford, CA, August 1980), 128-135.
5. Conery, J., and Kibler, D. Parallel interpretation of logic programming. *Proc. Conf. Functional Programming Languages and Computer Architecture* (Portsmouth, NH, October 1981). ACM.
6. Curry, H. B., and Feys, R. *Combinatory Logic*. vol. I. North-Holland, Amsterdam, 1958.
7. DARPA. *Strategic Computing*. DoD, October 1983.
8. Darlington, J., and Reeve, M. ALICE: A multi-processor reduction machine for the parallel evaluation of applicative languages. *Proc. Conf. Functional Programming Languages and Computer Architecture* (Portsmouth, NH, October 1981). ACM, pp. 65-75.
9. De Mori, R., Laface, P., and Mong, Y. Parallel algorithms for syllable recognition in continuous speech. *IEEE PAMI 7*, 1 (January 1985.).

10. Douglass, R. A qualitative assessment of parallelism in expert systems. *IEEE Software 2*, 2 (May 1985).

11. Douglass, R., and Lincoln, Patrick. Automatic versus hand-code parallelization: A stereo-vision example. In K. Preston and L. Uhr (Ed.) *Parallel Computer Vision*. Academic Press, New York, 1985.

12. Fischetti, M. Next generation computers: The United States. *IEEE Spectrum 20*, 11 (November 1983).

13. Forgy, C., Gupta, A., Newell, A., and Wedig, R. Initial assessment of architectures for production systems. *Proc. NCAI* (Austin, TX, 1984.), 116.

14. Goto, A., Tanaka, H., and Moto-oka, T. Highly parallel inference engine PIE - Goal rewriting model and machine architecture. *New Generation Computing 2*, 1 (1984), 37.

15. Halstead, R. H. Jr. Reference tree networks: Virtual machine and implementation. MIT/LCS/TR-22, MIT Laboratory for Computer Science, 1979.

16. Henderson, P., and Morris, J. M. A lazy evaluator. *Proc. 3rd ACM Symp. Prin. Prog. Lang.* (Atlanta, GA, January 1976), 95-103.

17. Hudak, P., and Keller, R. M. Garbage collection and task deletion in distributed applicative processing sysystems. *Conf. Rec. ACM Symp. Lisp and Functional Programming* (Pittsburgh, PA, August 1982), 168-178.

18. Hudak, P. Distributed graph marking. Research Report 268, Dept. of Comp. Sci., Yale University, January 1983.

19. Hudak, P. Distributed task and memory management. *Proc. Symp. Prin. Distributed Computing* (Montreal, August 1983), 277-289.

20. Hudak, P., and Goldberg, B. Experiments in diffused combinator reduction. *ACM Symp. Lisp and Functional Programming* (Austin, TX, August 1984), 167-176.

21. Hughes, R. J. M. Super-combinators: A new implementation technique for applicative languages. *Conf. Rec. ACM Symp. Lisp and Functional Programming* (Pittsburgh, PA, August 1982), 1-10.

22. Keller, R M. Semantics and applications of function graphs. Technical Report UUCS-80-112, Dept. of Comp. Sci., Univ. of Utah, Salt Lake City, October 1980.

23. Keller, R. M., and Lin, F. C. H. Simulated performance of a reduction-based multiprocessor. *IEEE Computer 17*, 7 (July 1984), 70-82.

24. Kluge, W., and Schlutter, H. An architecture for direct execution of reduction languages. *Proc. Intern. Workshop on High-Level Language Computer Architecture* (May 1980), 174-180.

25. Mago, G. A. A network of microprocessors to execute reduction languages, Part I. *Intern. Journ. Comp. and Info. Sci. 8*, 5 (March 1979), 349-385.

26. Moto-oka, T., and Stone, H. Fifth-Generation Computer Systems: A Japanese project. *IEEE Computer 17*, 3 (March 1984), 6.

27. Pollack, J., and Waltz, D. Parallel interpretation of natural language. *Proc. Intern. Conf. Fifth Generation Computer Systems* (Tokyo, 1984).

28. Preston, K., and Uhr, L. *Multicomputers for Image Processing*. Academic Press, New York, 1982.

29. Schönfinkel, M. Über die Bausteine der mathematischen Logik. *Math. Annal. 92* (1924), 305-316.

30. Stolfo, S., and Miranker, D. DADO: A parallel processor for expert systems. *Proc. Int. Conf. on Parallel Processing* (August 1984). IEEE Computer Society Press, p. 74.

31. Turner, D. A. A new implementation technique for applicative languages. *Software Practice & Experience 9* (1979), 31-49.