Distributed Applicative Processing Systems: Project Goals, Motivation, and Status Report

Paul Hudak

Technical Report YALEU/DCS/TR-~~317~~ 321
May 1, 1984

Yale University
Department of Computer Science
New Haven, CT

This research was supported in part by NSF Grant MCS-8106177.

Table of Contents

Distributed Applicative Processing Systems: Project Goals, Motivation, and Status Report

by
Paul Hudak

Abstract

Most existing parallel computers are successful only because of their ability to take advantage of the regularity and predictability of many programs, especially those found in scientific computation. On the other hand, there is a large class of programs (such as found in AI) characterized by irregularity and unpredictability, for which existing machines are poorly suited. Despite this problem, we argue that there are fundamental similarities in the dynamic behaviors of these programs, for which parallel computer systems can be specially tailored.

We describe herein a class of network computers called *Distributed Applicative Processing Systems*, or *DAPS*, that is based on the notion of dynamic task creation and distribution, using an underlying graph reduction model of program execution. The model is completely decentralized, and includes mechanisms to automatically perform "on-the-fly" garbage collection and task management. Our goal is to develop a model that exhibits parallelism in a way transparent to the user, and that executes unpredictable programs as effectively as predictable ones. We are testing the model by simulation. This paper describes the motivation, goals, and current status of the DAPS project.

1. Introduction

There have been a considerable number of proposals in recent years for machines that have the outward appearance of a "dense, closely-coupled network of processing elements", machines that we refer to as *network computers*. Such machines have several very appealing features: they have the potential for a great deal of parallelism, they are highly extensible, and with the advent of VLSI they are fairly easy to build. However, *none* 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 consistent *global* computation 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 non-deterministic computations. Such issues become extremely complex in the decentralized, distributed environment of the network computer, and cannot be taken for granted.
3. There has been almost no empirical study of the quantitative performance requirements of such machines. What should the ratio of processor speed to communications bandwidth be, what is the appropriate network topology, and how much memory should each processor have? All of these questions (and more) need to be answered before actually committing to a particular design.
4. There is little to be learned from the success of parallel machines in scientific computing, because these machines depend heavily on a program's *regularity* and *predictability* to get a high degree of parallelism. Such program behavior is common in scientific computing, but is absent from AI programs and others for which we would like the fifth generation machines to perform well. New techniques need to be developed to deal with dynamic, unpredictable behavior.

For the past several years we have been developing a class of network computers called **Distributed Applicative Processing Systems** [14], or **DAPS**, in which we have specifically addressed the problems raised above. The DAPS model has a firm underlying computation strategy derived from the lambda calculus, together with powerful mechanisms for managing the logistics of a highly-parallel computation. Furthermore, we have begun studying the dynamic behavior of the irregular, unpredictable programs that are characteristic of AI and a few other application areas. From this we have developed a technique called "diffusion scheduling" to handle the dynamic and unpredictable spawning of parallel tasks during a computation.

In this report we hope to justify the DAPS approach and describe the status of our current research. We expect the results of our work to be useful to existing network computer designs, but our long-term goal is to design and build a general-purpose DAPS machine that does equally well on unpredictable programs as it does on predictable ones. We do not believe that the user wishes to concern him/herself with the details of parallel computation -- such mechanisms should occur transparently to the user's program. To reach this goal our current work has centered upon careful simulations to gather the necessary empirical data to allow sound reasoning to guide our design. At this stage in the research, this is a far more economical and flexible approach than building a real machine.

2. What is DAPS?

2.1. Motivation

The DAPS philosophy is motivated primarily from the desire to solve the problems outlined above, coupled with the following important observations: First, to attain the degree of parallelism that we desire, the shared-memory model (or "von Neumann bottleneck") must go. On the other hand, carrying this to the extreme taken by data-flow researchers results in a great deal of communications overhead [11]. What we propose instead is a collection of memory partitions, to which each is assigned a small number of processing elements (perhaps just one). This leads naturally to a network computer model.

Second, any resource allocation mechanisms that depend solely on *static analysis* will be inadequate to deal with the dynamic behavior of many types of programs. One quickly concludes that the existing score of "supercomputers" will not do, since they rely almost exclusively on compile-time analysis. We propose instead an architecture that (perhaps with the *aid* of static analysis) makes *run-time* task distribution and resource allocation decisions.

Finally, a highly-parallel, extensible system cannot rely on *centralized knowledge* to accomplish dynamic resource allocation. Such a centralized database can only introduce another von Neumann bottleneck, a feature that the parallel system was trying to avoid in the first place. The only feasible solution is a *decentralized* mechanism that relies on incomplete knowledge of the system state to properly spawn new parallel tasks. Our solution is a technique called *diffusion scheduling* in which the workload tends to "diffuse" through the network of processor resources.

2.2. Overview

The foregoing observations have led us to a class of architectures that is radically different from conventional "von Neumann" machines. Physically, a DAPS machine is a highly-parallel, closely-coupled network of autonomous processing elements having only local store and communicating by messages. Logically, DAPS accomplishes program execution by *graph reduction*. All control in DAPS is entirely *decentralized*, including memory and task management, thus avoiding any bottlenecks to effective parallel computation.

In this section we describe the salient features of DAPS, returning later with more detail on certain features as they relate to this proposal.

2.2.1. DAPS is based on graph reduction.

In DAPS a program is represented as a directed graph (called the **computation graph**) whose vertices are labeled with primitive operators and values, and whose edges reflect data dependencies between operators and values (cf. [2, 4, 7, 23, 25, 26, 35]). Program execution is accomplished via transmutations (called **reductions**) to the graph. 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. Figure 2-1 shows several examples of the reduction process. (This figure is intended to convey the general flavor of graph reduction -- we have left out many of the details of an actual implementation.)

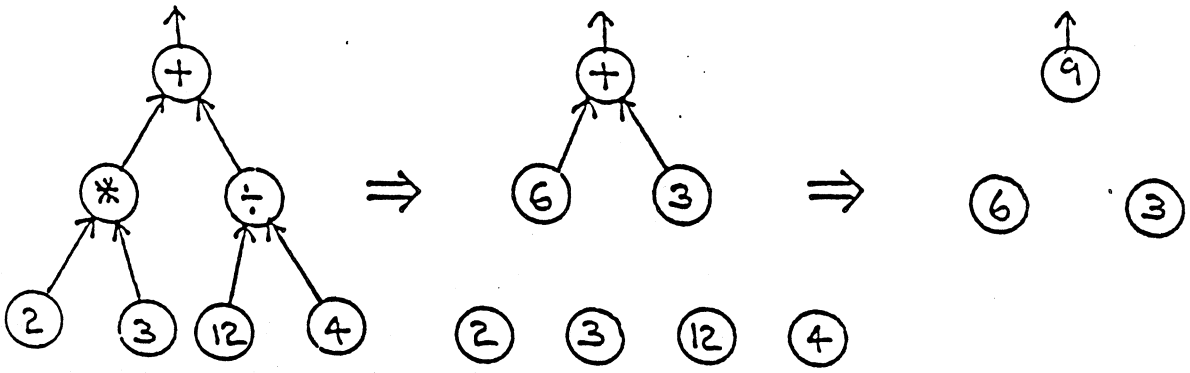
There are several significant advantages to the graph reduction approach: First, program and data are the same, simplifying the architecture and facilitating important AI tasks such as graph searches in semantic networks (see Section 4). Second, it is fairly easy for a compiler to determine the parallel components of a computation graph through straightforward flow analysis. Finally, it is possible to describe important properties of a parallel computation in terms of *graph connectivity*, allowing the development of effective strategies to deal with the logistics of a highly-parallel computation (as discussed in paragraph 2.2.5 below).

2.2.2. DAPS is highly parallel.

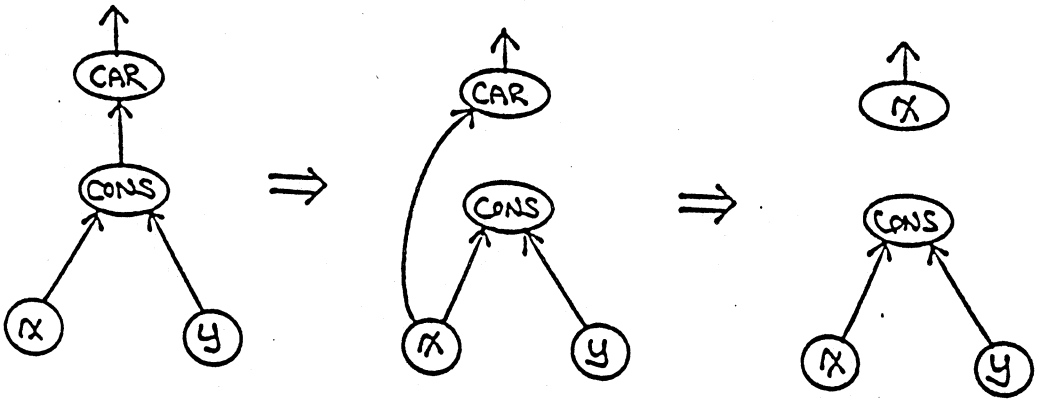
The DAPS parallel execution strategy amounts to dividing the computation graph into **partitions** that are assigned to autonomous processing elements (**PE**'s) in a closely-coupled network. Internally, each PE consists of a **local store** in which the partition is maintained, a **task queue** containing tasks ready for execution, a **communications system** through which tasks are spawned to other PE's, and two **processors**, one responsible for graph reduction, the other for house-keeping functions. The number and complexity of the primitive operations ("machine code") of a PE are minimal, corresponding to the chosen set of primitive graph mutations and data operators. DAPS thus supports the notion of a RISC architecture and is amenable to VLSI design.

2.2.3. DAPS is decentralized.

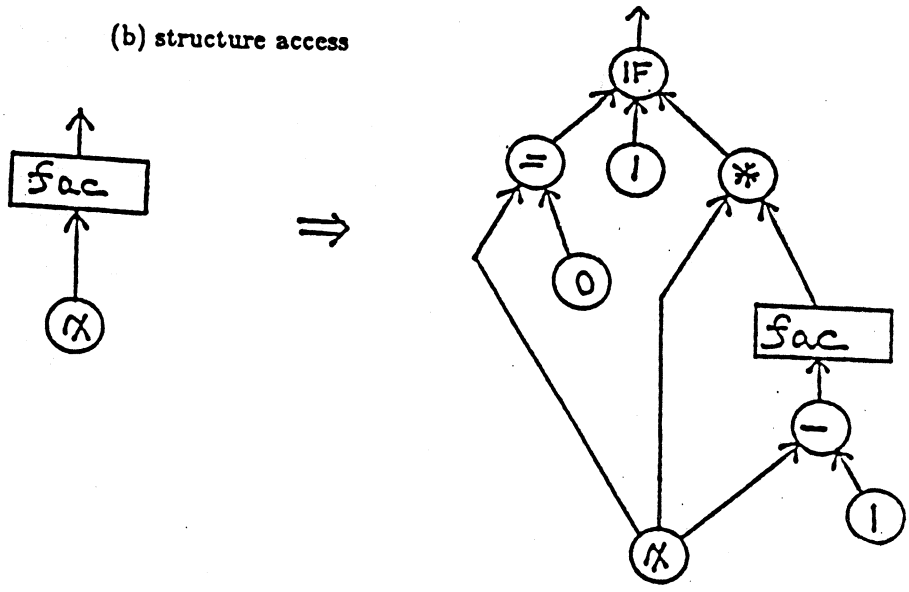
The network topology is completely *homogeneous*, meaning not only that each PE is identical, but also that the *interconnections* between them are the same. There is no concept of a "parent" or "child" node, and to ensure extensibility each PE has a fixed number of neighbors. The network provides *logically complete communication*, in that any PE may spawn tasks to



(a) arithmetic expression



(b) structure access



(c) function invocation via "macro-expansion"

Figure 2-1: Examples of Graph Reduction

any other (via a store-and-forward message protocol), although it is less expensive to communicate with a PE's immediate neighbors.

2.2.4. The *reduction process* accomplishes program execution.

Logically, DAPS should be viewed as *two* processes working simultaneously on the computation graph; this is reflected by the two processors comprising each PE. The objective of the **reduction process** is program execution. Each task in this process is responsible for a primitive operation such as data manipulation or a primitive graph mutation. Through diffusion scheduling, expanding portions of the graph are allocated to other PEs to try balancing processor load while maintaining locality of reference (more on this in Section 3.2). Since all knowledge is decentralized, these run-time decisions are made based on incomplete knowledge of the system state.

2.2.5. The *house-keeping process* handles the logistics of parallel computation.

Central to the DAPS philosophy is the inclusion of mechanisms to automatically perform most of the house-keeping chores required to manage a large parallel computation. Simple problems such as garbage collection become non-trivial in such an environment, and cannot be taken for granted. The house-keeping process executes *concurrently with the reduction process* (thus avoiding annoying interruptions in program execution), and is completely decentralized. Its global behavior dynamically accomplishes (1) garbage collection, (2) deletion of eagerly invoked computations subsequently found to be irrelevant, (3) detection of deadlock, and (4) updating of task priorities. This process is the most novel yet crucial aspect of DAPS, and is an extension of our work on *decentralized graph-marking algorithms* [15, 16, 17, 18].

3. Further Development

Our previous research has fairly well developed the "mechanistic details" of the DAPS model, as discussed in the last section. In some sense enough detail exists to build a prototype machine. Our preferred approach, however, has been to refine certain particularly important aspects of the model, to test them via simulation, and to study the dynamic behavior of programs for which we want DAPS to perform well, in particular AI programs. In this section we will elaborate on these ideas further.

Although we feel strongly that the DAPS model is ideal for the parallel execution of AI programs (and others), it is not clear what the appropriate *distribution and scheduling strategies* should be. The *distribution* strategies determine *what* should be distributed for parallel execution, and the *scheduling* strategies determine *where* to distribute the work. Our solution to

the first problem is the use of **serial-combinators**, and to the second problem, a technique called **diffusion scheduling**.

3.1. Serial-combinators

A **combinator** is a lambda expression composed only of applicative forms, and containing no free variables [5]. It turns out that any deterministic program can be rewritten into an equivalent one using a fixed set of constant combinators. Indeed, based on this idea we have built a compiler for ALFL (a functional language developed at Yale) targeted for a conventional machine [19]. We have also done experiments in distributed combinator reduction on DAPS [20]. The following idea is an extension of the latter.

Hughes generalized the notion of a combinator to include ones derived from a given source program that he calls **super-combinators** [21]. In this way the program itself determines the set of combinators that it is to be defined in terms of, creating ones that are maximal (in number of terms) for that program. Thus their execution is in general more efficeint than a composition of many finer-grained combinators. A **serial-combinator** is our own refinement of a super-combinator to *maximal expressions having no concurrent substructure*; i.e., all of its subexpressions must be computed serially. The idea is that such expressions are most efficiently computed on a single PE -- there is no advantage to subdividing the computation further, since that can only add communications cost to the already serial computation. Serial-combinators are therefore the smallest objects that are distributed for parallel execution in DAPS.

Since serial-combinators have no free variables, they can be easily implemented via graph reduction [35], and their detection can be done at compile-time as an extension to our work in [19]. Modifications our underway to our current compiler to include serial-combinator analysis, and that compiler will then become the "front end" to the DAPS simulator (discussed further below).

3.2. Diffusion Scheduling

Diffusion scheduling attacks the problem of deciding *where* to distribute new work. This decision is crucial to obtaining good performance, but must be done *quickly*, at *run-time*, and based on *decentralized knowledge*. It is not possible, therefore, to make optimal decisions -- a heuristic solution is required that attempts to balance processor load (thus increasing the degree of parallelism) while still maintaining locality of reference (thus minimizing commuications overhead). Given a new task **t** created on a PE **n**, the heuristics behind diffusion scheduling spawn **t** for execution either on **n** or on an immediate neighbor of **n**, based on a weighted sum of several factors, including:

1. The processing load on n (i.e., the number of tasks waiting for execution on n's task queue).
2. The memory load on n (i.e., 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 t to other nodes in the network.

The effect of (1)-(3) is to push tasks away from busy nodes, 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 3.2 for a hypothetical two-dimensional network. We are currently investigating several variations on the method by which the weighted sum is computed.

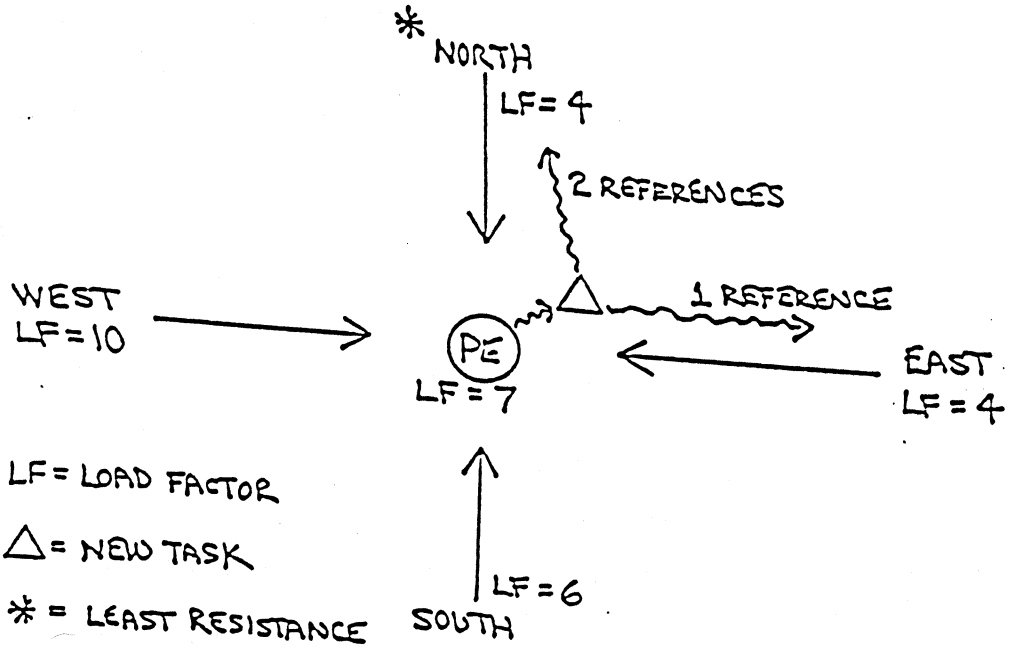


Figure 3-1: Diffusion Scheduler Chooses to Spawn Task to the North

3.3. The DAPS Simulator

We have recently completed a prototype simulator for DAPS on a network of Apollo workstations [20]. With it we can easily vary physical parameters such as network topology (and its size and degree), communications bandwidth, and the structure of each PE, as well as logical parameters such as the house-keeping process, the reduction process, and of course the distribution and scheduling heuristics. The simulator includes mechanisms to time-stamp

messages so that simulated parallelism can be measured, but it can also be run by simulating a set of PE's on an arbitrary number of workstations, thus emulating true parallelism.

We are using the simulator in two ways. First, we are experimenting with refinements to our reduction and diffusion scheduling strategies. The simulator's flexibility makes it possible to vary network and processor parameters in response to quantitative data as it is gathered. For example, the optimal values of the physical parameters mentioned in the last paragraph cannot be adequately determined until sufficient data has been gathered via simulation. Second, we have begun using it to empirically analyze the dynamic behavior of unpredictable programs, with special attention to be paid to AI applications. This is discussed further in the next section.

4. Relationship to AI Programming

In recognition of the importance of AI in future programming tasks, we plan to make AI the "applications test-bed" for DAPS. The programs found in AI are precisely the kind that existing "super-computers" do not execute efficiently, and are the kind that we are tailoring DAPS especially for.

Although many AI systems have at least gross structural components, their run-time behavior is usually quite unpredictable. This is perhaps an inherent characteristic of "knowledge-based" systems, since their behavior depends directly on the knowledge base, which is often changing. The structural components themselves are also rather unstable -- the possibilities are as varied as the theories in AI -- and there is simply no analog to the "array" that is at the heart of the regularity in scientific computation. As a result, architectures that rely on predictability or regularity are doomed to perform poorly when executing AI programs. The DAPS model, on the other hand, *assumes* such irregularity, and is designed to deal with it dynamically.

Despite our claims about AI programs, they do have certain things in common, and are not beyond run-time behavioral analysis. For example, a common characteristic of many AI programs is *graph searches*. Indeed, it is easy to argue that graph searching is an inherent behavioral aspect of "knowledge-based" systems, where data-retrieval operations on a graph of conceptual dependencies (semantic networks) are very common (we speak of semantic networks generically here, ignoring their differences [27, 33]). These searches fall nicely into the DAPS model, where the knowledge-base takes the form of a distributed program graph, and the searches occur in a highly-parallel manner via the spawning of tasks.

The DAPS model is able to support both sides of two on-going arguments in the AI community: the so-called declarative/procedural argument [1, 36], and the use of logic versus

semantic networks in representing knowledge [6, 22, 28]. Our intuition tells us that these arguments are of a high-level representational flavor, and that the *dynamic behaviors* of programs differing only in these respects are actually very similar. For example, the unification and backtracking mechanisms that underly resolution in logic programming languages are not unlike explicit inferencing mechanisms on semantic networks. We feel that it is possible to effectively support the underlying commonalities by a suitably designed machine. Just as a good understanding of the regular behavior of matrix algorithms preceded the design of effective vector machines, we feel that a good understanding of the dynamic behavior of AI programs should precede the design of a fifth generation computer.

4.1. Sources of Parallelism in AI Programming

Clearly those AI theories containing *explicit* parallelism, such as those modeling perceptual processes, could be made to run much faster on a parallel machine. Our primary interest, however, is in the *implicit* parallelism to be gleaned from a more general class of AI programs. As a simple but non-trivial example, consider a typical database of an expert system. The knowledge may appear as rules of the form "IF <condition> THEN <implication>", or in a production system as "<situation> → <action>" [30]. In both cases the enabling conditions take the form of a logical formula of an arbitrary number of terms, each of which could be computed in parallel. Furthermore, the inferencing mechanisms may explore the enabling and invocation of several rules simultaneously.

As a more general example, consider the set of Horn clauses making up the following Prolog program [3]:

$$\mathbf{A1} \leftarrow \mathbf{B11} \wedge \mathbf{B12} \wedge \ldots \wedge \mathbf{B1k}_1$$

$$\mathbf{A2} \leftarrow \mathbf{B21} \wedge \mathbf{B22} \wedge \ldots \wedge \mathbf{B2k}_2$$

$$\ldots$$

$$\mathbf{An} \leftarrow \mathbf{Bn1} \wedge \mathbf{Bn2} \wedge \ldots \wedge \mathbf{Bnk}_n$$

together with the goal $\mathbf{G} \equiv \mathbf{C1} \wedge \mathbf{C2} \wedge \ldots \wedge \mathbf{Cj}$. Any *resolution theorem prover* could be used to prove the goal through refutation. Intuitively, the strategy is to prove **G** by first proving the atomic formulae **C1** through **Cj** in turn. Each term is proved by *unifying* it with the head of one of the clauses, say **Ai**, which reduces the problem to proving the conjunction $\mathbf{Bi1} \wedge \mathbf{Bi2} \wedge \ldots \wedge \mathbf{Bik}_i$. The algorithm then recurses. In most Prolog systems the resolution process utilizes variations of Robinson's unification algorithm [31, 32] to reduce each goal to an empty conjunction of terms (indicating success), or to a non-empty conjunction with no more unifications possible (indicating failure). On a conventional *sequential* computer, the overall

process depends on extensive backtracking mechanisms to recover from failure. On the other hand, with an appropriate *parallel* machine, several unifications could be attempted simultaneously. The problem reduces to computation of a large AND/OR tree, where AND nodes represent the conjunction of terms, and OR nodes correspond to the possibility of a goal being unifiable with the heads of several different Horn clauses. An arbitrary number of sub-trees could be explored in parallel.

It should be clear that this style of execution of a Prolog program will generate a great deal of parallelism. Indeed, one might argue that a judicious invocation of parallel tasks will be required to avoid overwhelming the parallel resources! For example, consider the simultaneous unification of a goal with two clauses -- it is only necessary that *one* be successful, since it is the OR of the results that determines overall success. Imagine that one unification succeeds quickly, but that the other takes a very long time (or perhaps does not terminate at all!). Clearly this is a waste of resources, since the latter task may recursively generate an arbitrary mass of other parallel (and equally irrelevant) tasks.

The above problem highlights the need for a *dynamic* task manager, since it is quite doubtful that static analysis alone would be sufficient to make effective decisions (indeed the undecidability of the halting problem prevents us from knowing whether a sub-process will terminate). Recall that diffusion scheduling, for example, considers *processor load* before deciding where and when to allocate a new task, and this information is obviously only available at run-time. We are investigating several strategies for attaining parallelism of this sort.

4.2. The Importance of Eager Evaluation

The simultaneous computation of several terms in a logical formula is an example of an **eager computation**, since all of the values are not necessarily needed. Performing eager computations (resources permitting) has the effect of increasing the *effective parallelism* of a program, which is quite important since studies have shown that many programs simply do not have enough *strict parallelism* to effect an overall speed-up of more than a factor of 5 or so. From an AI standpoint this argument is strengthened by Fahlman's observation that we currently expend too much effort "optimizing" graph searches; or, in his words, "we can rearrange the order of the paths to be searched to gain efficiency, but it is dangerous to leave anything out" ([8],p.8).

A key aspect of the DAPS graph reduction strategy is the differentiation of **eager tasks** (whose results might not be needed) from **vital tasks** (whose results are known to be needed). The inclusion of eager tasks seems innocuous enough, until one considers that:

1. Eager tasks "compete" with vital ones, so it is important to assign a higher *priority* to the latter, especially when resources are limited.
2. It may be subsequently discovered that the result of an eager computation *is* needed, in which case all of the resulting eager tasks that comprise that (sub)computation are now vital. We thus need a mechanism to *dynamically upgrade task priorities.*
3. Worse, one may discover that an eagerly invoked computation is *not* needed -- the tasks comprising such a computation then become **irrelevant.** These tasks behave no differently than any others, and may distribute through the system generating an arbitrarily large (and irrelevant) parallel workload. We therefore need a mechanism to *find and delete irrelevant tasks.*
4. If shared subexpressions are allowed, a *fourth* type of task arises when an eager task is dereferenced from the vertex that initially spawned it, but is still accessible from some other vertex that has not requested its value yet. We refer to this as a **reserve task**, and it requires a unique priority as well.

As can be seen, allowing eager computations (which we see as crucial to getting the high degree of parallelism that we desire) creates an interesting logistical problem. An initially eager task may expand into a highly parallel workload of many other tasks, each of whose priority is subject to change as the computation proceeds. An important advantage of DAPS is the inclusion of mechanisms to automatically prioritize tasks, detect and delete irrelevant tasks, as well as perform system-wide garbage collection and deadlock detection. As discussed in Section 2, these algorithms are part of the house-keeping process, and execute *concurrently* with program execution. Just as all LISP systems provide automatic garbage collectors, DAPS provides these house-keeping mechanisms to make the subtle details of parallel computation transparent to the user's program. We can't over-emphasize the importance of this aspect of the DAPS model.

5. Relationship to Other Work

We are not alone, of course, in our quest for a viable "fifth generation" computer. We discuss in this section the relationship of our work to several other significant efforts. This discussion is useful in further highlighting our philosophy and research goals.

First, we must mention the Japanese Fifth Generation Computer Systems (FGCS) Project [29, 34], being administered by Japan's Institute for New Generation Computer Technology (ICOT). The Japanese overall research plan is actually quite complex, and still evolving. One of their chief goals is the design of "knowledge information processing systems" or **KIPS**, whose underlying computation model is based on first-order predicate calculus. The machine language of KIPS will thus have a Prolog flavor, and performance will be measured in "logical inference per second", or **LIPS**. Another goal of the Japanese plan is the development of a high-performance database machine to support expert systems.

Although the Japanese goals are quite bold (indeed radical), we see several flaws in their overall plan. First, we believe it is overly restrictive to base a machine on logic programming. Prolog is a very effective language for certain applications, but it has by no means become universally adopted, even in the AI community alone. To quote Feigenbaum and McCorduck;

> The Prolog language has features as well as flaws. One good feature is a logical calculus... Its flaw is that the knowledge so represented is often opaque, incomprehensible, and arcane. A second good feature of Prolog is that it solves problems by proving theorems in first-order predicate calculus using computationally fast methods (that can be made even faster by using parallelism). The user never has to be concerned with the details of the problem-solving process.... [but] The last thing a knowledge engineer wants to do is abdicate control to an "automatic" theorem-proving process that conducts massive searches without step-by-step control exerted by knowledge in the knowledge base. Such uncontrolled searches can be extremely time-consuming. The parallelism that can be brought to bear is a mere palliative, a Band-aid... [10]

Even the inferencing mechanisms that underly the Prolog model are overly restrictive, in that new inferencing theories are emerging in AI whose behavior might be obscured when "forced" into the Prolog model, as well as make them less efficient than a more explicit encoding of their strategy.

The DAPS computation model is essentially more primitive than that of KIPS, and as a result more general. It is just as easy to build a flexible Prolog interpreter in the "micro-code" of a DAPS machine as it is to build any other inferencing engine. Similarly, a variety of high-level languages could easily be supported -- not only Prolog and LISP, but also AI specific languages and systems such as SRL, KL-one, PSN, KRL, Planner, etc. Rather than restrict the design to a specific language or inferencing strategy, DAPS instead concentrates on the issues of initiating and coordinating highly-parallel computations, issues that have not been emphasized in the Japanese research plan.

A final objection to the Japanese approach is their idea of a special database machine serving as a "backend" to speed up information retrieval. This approach is not extensible, and simply introduces another "von Neumann bottleneck", something that we are adamant about avoiding. As discussed earlier, the DAPS model supports highly-parallel graph searches, and does so in a flexible, decentralized manner.

The research plan whose underlying philosophy is most similar to DAPS is Keller's **Rediflow** multi-processor, a blend of data-flow and reduction ideas [24]. Although there are some gross differences between Rediflow and DAPS (such as the lack in Rediflow of mechanisms to deal with eager computation), there are two key similarities; the first being the use of graph reduction, the second being the dynamic scheduling of tasks (referred to as "load-balancing" in Rediflow). However, there are some important but subtle differences even in these features. For example,

Rediflow considers memory utilization of a PE as part of its diffusion heuristics (something that DAPS does not do), but does not use information about global pointers to help maintain locality (an important feature of the DAPS strategy). Second, Rediflow's graph reduction strategy is based heavliy on user-defined functions rather than using the more optimal serial-combinator analysis that we use. All in all, the two models are complimentary, and lend each other support in the viability of this style of machine.

Another related effort is Hewitt's **Apiary System** [12]. Physically the Apiary is similar to our current image of DAPS, although neither machine has been built. Logically there are some important differences, the foremost being that the Apiary is based on the ACTOR model of computation, rather than graph reduction. We feel that the graph reduction approach is more versatile for several reasons. First, it more directly supports the idea of graph searches in knowledge bases. Second, our global virtual graph space allows one to better coordinate the overall computation, thus facilitating the all-important house-keeping chores discussed earlier. Third, it more directly supports Algol-like languages, including LISP and functional programming languages.

The **Connection Machine** [13] is perhaps the most radical and AI-specific proposal that we have seen, but like the Japanese approach is overly restrictive (although in a different way). The machine is designed specifically for concurrent manipulation of knowledge stored in a semantic network, by mapping every node and link in the network to its own autonomous processing element and communications channel. Such a strategy provides the *potential* for a high degree of parallelism, but in reality only a small subset of the processing elements will be active at any given time. The graph partitioning strategy in DAPS tries instead to assign a *subgraph* of a semantic network to a single processing element. More specifically, the intent of the serial-combinator analysis is to partition the network in such a way that knowledge obtainable only through sequential inferences is resident in one partition. That is, if two nodes in a semantic net are always traversed sequentially, there is no advantage to assigning them to separate PE's as the Connection Machine does by default.

The Connection Machine is actually an extension of earlier work by Scott Fahlman on systems for representing "real-world knowledge". Fahlman has developed a system called **NETL** that is both a theory of knowledge representation and a design for a massively parallel machine to realize the theory [8]. Like the Connection Machine, NETL maps each node and link in the semantic network to a physical processor and communications channel, respectively. Special **marker passing** algorithms are used to perform highly-parallel graph searches to accomplish inferencing on the semantic network. Limitations of the marker-passing strategy have recently

led Fahlman to alter his model somewhat to accomodate **value-passing** strategies as well. From this effort has emerged a new model of knowledge representation called **Thistle**, together with a new parallel architecture called the **Boltzmann Machine** [9].

All of our comments about the overly restrictive nature of the Connection Machine apply to Fahlman's work as well. The theories of knowledge representation that underly all of these machines have not been adequately tested, and even if sound are not likely to be adopted as universal theories by the AI community. It should be noted that any of these theories could be implemented on DAPS (just as they could be implemented on a serial computer) with greater flexibility. The result would be more efficient use of processing power (since more PE's would be kept busy), but we would probably not achieve as high a degree of parallelism as the more "massively parallel" approach.

6. Project Summary

DAPS represents a class of network computers vastly different from conventional machines. Our underlying philosophy assumes unpredictable program behavior, highly-parallel decentralized computation, and integral memory and task management mechanisms. We feel that DAPS is a viable fifth generation computer model, but is sufficiently radical and untested that building hardware at this time is decidedly premature. We are instead embarking on a detailed study of the feasibility and effectiveness of our ideas, including a detailed simulation of a DAPS machine, where great flexibility is attained at an affordable cost. No hardware construction is planned until we have *proven* that parallelism can be adequately exploited by the network-computer model. Our long-term goal is to either retrofit an existing network computer with the DAPS strategies, or to design a DAPS machine from scratch.

7. Acknowledgements

Many thanks to Ben Goldberg, who built the current DAPS Simulator, and to David Kranz, who helped build the current ALFL Compiler. Also thanks to Bob Keller, whose influence has been most pervasive. Several other useful discussions have taken place with the many fine students and faculty at Yale.

References

[1] Barr, A., and Feigenbaum, E.A.
The Handbook of Artificial Intelligence.
HeurisTech Press, 1981.

[2] K. J. Berkling.
Reduction languages for reduction machines.
In *Proc. 2nd Annual Symposium on Computer Architecture*, pages 133-140. IEEE, 1975.

[3] Clark, K.L.
An introduction to logic programming.
Gordon and Breach, 1982, pages 93-112.

[4] Clarke, T., Gladstone, P., MacLean, Norman, A.
SKIM - the S, K, I reduction machine.
In Davis, R.E., and Allen, J.R. (editors), *The 1980 LISP Conference*, pages 128-135.
Stanford University, August, 1980.

[5] Curry, H.K., and Feys, R.
Combinatoroy Logic.
Noth-Holland Pub. Co., Amsterdam, 1958.

[6] Dahl,V.
Logic programming as a representation of knowledge.
Computer 16(10):106-111, October, 1983.

[7] Darlington, J. and Reeve, M.
ALICE: A multi-processor reduction machine for the parallel evaluation of applicative languages.
In *Functional Programming Languages and Computer Architecture*, pages 65-76. ACM, October, 1981.

[8] Fahlman, S.
NETL: A System for Representing and Using Real-World Knowlege.
MIT Press, Cambridge, MA, 1979.

[9] Fahlman, S.E., and Hinton, G.E.
Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines.
In *Proc. AAAI*, pages 109-113. AAAI, 1983.

[10] Feigenbaum, E.A., and McCorduck, P.
The Fifth Generation.
Addison-Wesley Publishing Company, Reading, MA, 1983.

[11] Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H.
A second opinion on data flow machines and languages.
Computer 15(2):58-69, February, 1982.

[12] Hewitt, C.
Design of the APIARY for actor systems.
In Davis, R.E., and Allen, J.R. (editors), *The 1980 LISP Conference*, pages 107-118. Stanford University, August, 1980.

[13] Hillis, W.D.
The Connection Machine.
Technical Report AI-Memo 646, MIT, Sept, 1981.

[14] Hudak, P.
Object and Task Reclamation in Distributed Applicative Processing Systems.
PhD thesis, University of Utah, July, 1982.

[15] Hudak, P. and Keller, R.M.
Garbage collection and task deletion in distributed applicative processing systems.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 168-178. ACM, August, 1982.

[16] Hudak, P.
Distributed Graph Marking.
Research Report 268, Yale University, January, 1983.

[17] Hudak, P.
Decentralized marking of an evolving graph.
submitted to TOPLAS , 1983.

[18] Hudak, P.
Distributed Task and Memory Management.
In Lynch, N.A., et al. (editors), *Proc. of Sym. on Prin. of Dist. Comp.*, pages 277-289. ACM, August, 1983.

[19] Hudak, P. and Kranz, D.
A combinator-based compiler for a functional language.
In *11th ACM Sym. on Prin. of Prog. Lang.*, pages 121-132. acm, January, 1984.

[20] Hudak, P. and Goldberg, B.
Experiments in diffused combinator reduction.
In *Sym. on LISP and Functional Programming*, pages to appear. ACM, August, 1984.

[21] Hughes, R.J.M.
Super-combinators: A new implementation method for applicative languages.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 1-10. ACM, August, 1982.

[22] Israel, D.J.
The role of logic in knowledge representation.
Computer 16(10):37-41, October, 1983.

[23] Keller, R.M.
Semantics and applications of function graphs.
Technical Report UUCS-80-112, Department of Computer Science, University of Utah, October, 1980.

[24] Keller, R.M. and Lin, F.C.H.
Simulated performance of a reduction-based multiprocessor.
IEEE Computer 17(7):to appear, July, 1984.

[25] Kluge, W., and Schlutter, H.
An architecture for direct execution of reduction languages.
In Chu et al. (editors), *Proc. of the International Workshop on High-Level Language Computer Architecture*, pages 174-180. May, 1980.

[26] Mago, G.A.
A network of microprocessors to execute reduction languages, Part I.
International Journal of Computer and Information Sciences 8(5):349-385, March, 1979 revised.

[27] Minsky, M.
A framework for representing knowledge.
AI Memo 306, MIT, June, 1974.

[28] Moore, R.C.
The role of logic in knowledge representation and commonsense reasoning.
In *Proc. Nat'l Conference Artificial Intelligence*, pages 428-433. American Assoc. for Artificial Intelligence, August, 1982.

[29] Moto-oka, T., et al.
Proc. Int'l Conf. 5th Generation Computer Systems.
Japan Information Processing Development Center, 1981.

[30] Quinlan, J.R.
Fundamentals of the knowledge engineering problem.
Gordon and Breach, 1982, pages 33-46.

[31] Robinson, A.
A machine-oriented logic based on the resolution principle.
J.ACM 12:23-41, 1965.

[32] Robinson, J.A.
Fundamentals of machine-oriented deductive logic.
Gordon and Breach, 1982, pages 81-92.

[33] Schank,R., Abelson, R.
Scripts, plans, and knowledge.
In *Proc. Int'l Joint Conference AI*, pages 151-157. AI, 1975.

[34] Treleaven, P., Lima, I.
Japan's fifth-generation computer systems.
Computer 15(8):79-88, August, 1982.

[35] Turner, D.A.
A new implementation technique for applicative languages.
Software-Practice and Experience 9:31-49, 1979.

[36] Winston, P.H.
Artificial Intelligence.
Addison-Wesley Publishing Co., Don Mills, Ontario, Canada, 1979.