Distributed Execution of Functional Programs Using Serial Combinators

PAUL HUDAK, MEMBER, IEEE, AND BENJAMIN GOLDBERG

Abstract — **A general strategy for automatically decomposing and dynamically distributing a functional program is discussed, suitable for parallel execution on multiprocessor architectures with no shared memory. The strategy borrows ideas from data flow and reduction machine research on one hand, and from conventional compiler technology for sequential machines on the other. One of the more troublesome issues in such a system is choosing the right granularity for the parallel tasks. As a solution we describe a program transformation technique based on *serial combinators* that offers in some sense just the "right" granularity for this style of computing, and that can be "fine-tuned" for particular multiprocessor architectures. We show via simulation the success of our approach.**

Index Terms — **Combinators, distributed computing, functional programming, graph reduction, lambda calculus, load-balancing, multiprocessing, parallel computing.**

I. Introduction

In recent years there has been considerable interest in parallel architectures of various sorts, especially ones characterized as "a large number of autonomous processing elements" interconnected in various ways. We refer to such machines collectively as *multiprocessors,* although in this paper we will concentrate on ones having a regular, sparse interconnect with no shared memory. The interest in these machines is not surprising since they are in a sense the most "obvious" way to get vast amounts of parallelism, and they are relatively easy to build. Indeed, several manufacturers are now producing commercial multiprocessors whose price/performance ratio appears to be quite favorable. To fully exploit these architectures, one must obtain near-maximal performance from each of the individual processors simultaneously. This is not an easy task.

The majority of researchers interested in programming such machines view the overall system in its most literal, concrete form, i.e., as a network of individual machines communicating in some cooperative manner. Our viewpoint is somewhat different: we wish to treat the multiprocessor abstractly as a single logical entity (sometimes called a *network computer* or *ensemble architecture*), and our goal is to provide a way to *automatically* decompose and dynamically distribute a user's program for parallel execution. Our view also differs from the conventional one in that we wish to build a *general-purpose* machine that can execute many users' parallel programs simultaneously, rather than a dedicated multiprocessor whose programs are statically mapped for optimal performance. With these goals in mind, it is not surprising that our work has centered upon developing a "virtual machine" with the desired properties, that we believe can be implemented efficiently on top of a conventional multiprocessor. We accomplish this by borrowing ideas from data flow and reduction machine research on one hand, and from conventional compiler technology for sequential machines on the other.

Manuscript received February 1, 1985; revised May 30, 1985. This work was supported in part by the National Science Foundation under Grant MCS-8302018 and in part by a grant from Burroughs Corporation. A preliminary version of this paper was presented at the IEEE 1985 International Conference on Parallel Processing, St. Charles, IL, Aug. 1985.

The authors are with the Department of Computer Science, Yale University, New Haven, CT 06520.

One of the more troublesome issues in such a design is choosing the right granularity for the parallel tasks — if they are too large, parallelism might be lost, and if too small, communication costs might dominate the computation. Our theory of *serial combinators* offers a medium granularity that we consider to be quite suitable for this style of computing, and that can be "fine-tuned" for particular multiprocessor architectures.

In the next section we describe in more detail the problems to be solved in reaching our goal and briefly outline our solutions. Serial combinators are described in detail in Section III. Then in Section IV we describe the virtual graph reduction machine that forms the foundation for multiprocessor realizations. Simulation results are given in Section V. Finally, in Section VI we discuss other related efforts and point to future research directions.

II. Problem Description

There are three fundamental obstacles in the way of achieving the goals stated in Section I: 1) choosing a language base with ample and easily detectable parallelism; 2) choosing an evaluation model that does not rely on any "centralized" data or control that could act as a bottleneck; and 3) choosing the right process granularity, especially for architectures with nontrivial communication costs. Each of these issues is discussed in more detail below.

A. Language Base

If one is to be able to automatically decompose programs for parallel execution one must choose a language whose semantics provides ample opportunities for parallelism and whose syntax makes it easy to detect such parallelism. The class of *functional languages* (also known as *applicative* or *data flow* languages) is particularly good at satisfying these requirements. Aside from their "standard" virtues, which have been well argued elsewhere [1], [5], [8] and are beyond the scope of this paper, functional languages have another feature that becomes virtuous when viewed in the light of parallel computation. Specifically, the standard evaluation models for functional languages exhibit the well-known *Church–Rosser property* [4], 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 property is invaluable in parallel systems.

One can argue for the suitability of functional languages in parallel computing in yet another way that is perhaps more intuitive and thus more convincing. The argument is simply that *there are no side-effects*. Most experienced programmers recognize the importance of at least minimizing side-effects, but the importance of doing so in a parallel system is intensified significantly, due to the careful synchronization required to ensure correct behavior when side-effects are present. Without side-effects, there is no way for concurrent portions of a program to adversely affect one another — indeed, this is simply another way of stating the Church–Rosser property.

To summarize, in functional languages parallelism is implicit, easy to detect, and supported by the underlying semantics. Our intent in providing a general-purpose system is that one might write and debug a functional program on a sequential machine and then run the same program on a parallel machine for improved performance. There is generally no need for special message-passing constructs or other communications primitives, no need for process creation/synchronization primitives, and no need for special "parallel" constructs such as "parbegin · · · parend."

B. Evaluation Model

Most conventional languages rely on a sequential stack-based evaluation model. The stack usually serves two distinct (indeed, separable) purposes: 1) it provides a data structure through which lexically bound variables may be referenced (often called the *static chain*), and 2) it provides a return mechanism for recursive procedure calls (often called the *dynamic chain*). The stack serves these purposes well since its last-in, first-out behavior matches the depth-first evaluation that characterizes most sequential execution strategies. For this same reason it is a particularly bad structure for general parallel computation. Even if parallelism were possible, the centralized nature of the environment (allocated on the stack) would act as a bottleneck.

There is actually considerable motivation for abandoning stack allocation even for sequential computers. For example, higher-order functions, central to functional languages as well as Scheme dialects of Lisp [25], [22], require the equivalent of a "closure," which in general must be heap allocated (in this paper we use "heap" in the Lisp implementation sense). Furthermore, coroutines and other "exotic" control structures (including upward continuations in Scheme) require at a minimum multiple stacks, which in turn require some degree of heap allocation.

An alternative evaluation model, that appears well suited to parallel computation, is *graph reduction*. Although graph reduction is simply an operational rendition of reduction in the lambda calculus [26], we prefer to view it as a generalization of conventional stack-based evaluation, in which activation records are allocated in a heap instead of on a stack. We refer to the resultant structure as the *program graph*.

In a parallel environment, how does graph reduction serve the two purposes described earlier, i.e., the static and dynamic chains? It is easy to imagine several reducible expressions (called *redexes*) being available simultaneously in the program graph, and the Church–Rosser property permits us to reduce them simultaneously. Thus parallel execution is achieved essentially by making the dynamic chain tree-like. But what about the bottleneck in the static chain? In its simplest form, graph reduction will have the same problem as the stack-based model — even though the environment structure is now tree-like, the upper levels of the tree will still behave as a bottleneck to shared data. Naively making multiple copies of the environment is an overkill and quite inefficient. A mechanism is needed to distribute just enough of the environment to the proper places while avoiding excessive communication costs.

We can solve this more troublesome problem by basing our graph reduction on the *combinatory calculus* [4] instead of the lambda calculus (for our purposes the two are equivalent in expressive power). Schonfinkel [24] showed that by using a fixed set of small constant functions called *combinators*, all bound variables may be removed from a program. This somewhat surprising result means that the ubiquitous environment, central to the notion of beta-reduction in the lambda calculus, can be eliminated entirely! In Section III we will describe in detail how this takes place.

C. Process Granularity

Of the many possible program execution models, it is probably safe to say that a fixed set of combinators offers in some sense the *finest* granularity of computation, even finer than data flow. However, if one studies the performance figures for existing multiprocessors, it becomes apparent that the ratio of interprocessor communication time to CPU instruction speed is generally quite high, typically anywhere from 10 to 100. Thus, it seems that we must find relatively large "grains" of parallelism for our overall strategy to be successful. We initially became motivated to do this after observing via simulation that with a fixed set of combinators it is possible for a purely sequential computation (i.e., one whose data dependences preclude any parallelism) to become decomposed for execution on several processors [11]. Clearly, this is a wasted effort.

On the other hand, little work has been done on larger grains of parallelism. The strategy used by Keller and Lin [19] constrains the granularity to that implied by source-level function definitions, thus placing the burden on the programmer. Such user-defined functions not only may contain internal parallelism that may be lost, but they are also seldom combinators. This means that separate mechanisms are required to import values for free variables, and the useful property of "full laziness" may not be realized; that is, it is possible for a partially applied function to recompute some of its subexpressions if it is shared. Hughes' *supercombinators* [17] solve the latter problem, but are targeted for sequential machines and thus may contain internal parallelism that will be lost.

The goal then is to retain the environmentless nature of combinators and their usefulness in a parallel graph reducer, while maximizing their granularity and ensuring that parallelism is not lost. It turns out that if one is willing to do without a *fixed* set of combinators (an idea first suggested by Hughes [17]), then one can infer *program-derived* combinators that have the desired properties. We call these *serial combinators,* which we argue to be in some sense at just the right level of granularity, because of the following.

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, ensuring that no extraneous communication costs are incurred because of too fine a granularity.

We have also made some pragmatic refinements to serial combinators to increase their efficiency. In particular, we take into consideration strictness properties of functions, common subexpressions, complexity of subexpressions, and the overhead for distributing a computation. We discuss serial combinators and their refinements in more detail in the next section.

III. Serial Combinators

The reader is assumed to have some familiarity with the lambda calculus and combinatory calculus, although we will begin with a review of the basic ideas. A more theoretical discussion may be found in [2] or [4]. We will then proceed with a description of serial combinators, for which a more thorough treatment is contained in [14].

A. A Brief Review of Combinators

Consider a function f defined by $f(x) = \mathbf{exp}$. A functional object equivalent to f may be obtained by *abstraction* of the free variable x from **exp**, which is written as $[x]\mathbf{exp}$. Application of this function to a value v is written as $([x]\mathbf{exp})v$. The interaction between abstraction and application is defined by the simple rule

$$([x]\mathbf{exp})x = \mathbf{exp}. \tag{1}$$

In the lambda calculus $[x]\mathbf{exp}$ is written $\lambda x.\mathbf{exp}$, and the process of applying it to a value v is called *beta-reduction*. Logically, beta-reduction causes substitution of v for all free occurrences of x in **exp**, but it is more typically implemented by providing an *environment* to **exp** in which the value of x is bound to v. The environment ensures that $(\lambda x.\mathbf{exp})x = \mathbf{exp}$.

Alternatively, using the *combinatory calculus,* one may abstract x from **exp** according to the following rules:

$$[x]x = I$$
$$[x]y = K\,y, \quad \text{if } x \neq y$$
$$[x](e1\ e2) = S([x]e1)\,([x]e2)$$

where S, K, and I are primitive functions called *combinators* that are defined (assuming function application associates to the left) by

$$I\,x = x$$
$$K\,y\,x = y$$
$$S\,f\,g\,x = f\,x(g\,x).$$

Using these rules it is easy to show that this method of abstraction obeys the rule given in (1). Note that through repeated abstractions of this sort, all bound variables may be eliminated from a program. Thus, the "substitution" operation in the lambda calculus becomes meaningless, and there is no need for an environment.

The combinators S, K, and I form a rather small, fixed set of primitives, which is quite attractive. However, as argued earlier, there are reasons for wanting a larger granularity. Technically, a combinator 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 that are combined by application. These are the crucial properties 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. With this generalization Hughes [17] introduced the notion of a *supercombinator,* as explained below.

A *free expression* in a lambda expression $\lambda v.e$ is defined as any subexpression of e in which v does not occur as a free variable. A *maximally free expression* is a free expression which is not part of any larger free expression. In a nutshell, starting with a program P, Hughes' algorithm for generating supercombinators is then the following.

1) Find the leftmost, innermost lambda expression $L = \lambda v.\mathbf{exp}$.
2) Find the maximally free expressions, e_l through e_n, of L.
3) Create a new combinator (say α) defined by
$$\alpha\ i_1 \cdots i_n v = \mathbf{exp}[i_1/e_1, \cdots, i_2/e_2]$$
where formal parameter names i_1 through i_n do not occur free in **exp**. (The expression $e[x/y]$ denotes the result of substituting x for all free occurrences of y in e.)
4) Substitute $(\alpha\ e1 \cdots en)$ for L in P.
5) Repeat steps 1)–4) until step 1) fails.

Together with a few optimizations, the resulting supercombinators have a very useful property: execution of the original program results in a *fully lazy evaluation*. No subexpression internal to a combinator body needs to be recomputed as a result of a partially applied combinator being shared in several places.

A *serial combinator* is basically a refinement of a supercombinator in which several additional constraints have been added, the most important being that *it has no concurrent substructure,* and furthermore, it is not contained in any larger combinator with that 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. As argued earlier, serial combinators are small enough that no parallelism is lost, yet large enough that no extraneous communication costs are incurred.

We should also point out from a complexity standpoint that serial and supercombinators seem to have an inherent efficiency advantage over a fixed set of combinators, based on the following argument: consider a lambda expression **exp** with a free variable x that occurs at lexical depth n. Using a standard fixed set of combinators, at least n abstractions are needed to abstract x from **exp**, and likewise at least n reductions take place when the resulting expression is applied to an actual argument for x! Yet with a serial combinator the overhead is essentially constant, independent of the depth. Furthermore, research in the last 20 years has produced excellent compiler techniques for generating efficient code for lambda expressions, including optimal code for trees and single-level environments. These optimizations are easily applied to the bodies of serial combinators, suggesting an architecture (such as the multiprocessor that is of interest to us) that combines the elegance and generality of graph reduction with the efficiency of a conventional register machine. This particular argument is not made by Hughes, whose reduction strategy is to expand supercombinators into their graphical equivalents, after which normal graph reduction proceeds. Such a strategy has the advantage of avoiding the depth n complexity mentioned above, but fails to take advantage of other efficient compilation techniques.

In the remainder of this section we will first give an algorithm to compute "simple" serial combinators and then discuss some important refinements and their effect on the basic algorithm.

B. An Algorithm to Generate Simple Serial Combinators

Our examples are expressed in a language called ALFL [12], a lazy functional language with pattern-matching similar to SASL [28] and FEL [18]. In ALFL the basic program structure is an *equation group,* whose value is denoted by the keyword **result**. The equations may define local values or functions, and are evaluated "by demand." The expression **e1 → e2, e3** is like **if e1 then e2 else e3.**

We assume the reader to be familiar enough with this style of language that the examples will be self-explanatory. Most of the translation process, however, takes place on lambda expressions. The basic algorithm is as follows.

1) Translate the ALFL source program into a lambda expression. This is a straightforward translation as described in [13].
2) Convert the lambda expression into supercombinators using Hughes' method described in the last section, together with the optimizations of eliminating redundant combinators and ordering parameters optimally [17]. As a further optimization we avoid creating "trivial closures" (the partial evaluation of primitive functions) by not abstracting free expressions that consist solely of a primitive operator with an incomplete set of arguments, such as "+ x."
3) Convert to serial combinators, by doing the following for each supercombinator.
   a) Determine its concurrent substructure. In its basic form, this simply means finding primitive operators that are *strict* in more than one of their arguments, such as the basic arithmetic and relational operators (+, −, =, >, etc.). It might also include "eager" versions of cons and append, or even an "eager conditional."
   b) Starting from the outermost expression and working inward, for each subtree containing several concurrent subexpressions, retain one of the subexpressions in the definition of the current combinator (since it will represent part of the combinators' "sequential thread"), and compile each of the other subexpressions into a separate serial combinator (to allow them to be computed in parallel).
   c) For each serial combinator thus generated, repeat step 3b). Stop when no more refinements are made.

As an example, consider the following divide-and-conquer version of the factorial function, written in our source language ALFL:

```
{fac 0 = 1;
 ' n = pfac 1 n;
pfac l h = h=l → l,
           h=l+1 → l * h,
           (pfac l (l + h)/2) * (pfac ((l + h)/2 + 1) h);
result fac n}
```

After step 1) of the algorithm we get the lambda expression:

```
pfac = Y(λf λx λy. if (= x y) x
                  (if (= y (+ x 1)) (* x y)
                     (* (f x (/ (+ x y) 2))
                        (f (+ (/ (+ x y) 2) 1) y) ))).
```

Applying step 2) then yields the supercombinator version:

```
pfac = Y γ
γ f = β(α f) f
α a b c d h = if (= d h) d
                (if (= b h) (* d h)
                   (* (c (/ (+ d h) 2))
                      (a (+ (/ (+ d h) 2) 1) h)))
β p q l = p (+ l 1) (q l) l.
```

Note, however, that α contains concurrent subexpressions. By applying step 3) we thus generate the new serial combinator η:

```
pfac = Y γ
γ f = β(α f) f
α a b c d h = if (= d h) d
                (if (= b h) (* d h)
                   (* (c (/ (+ d h) 2)) (η a d h)))
β p q l = p (+ l 1) (q l) l
η j k h = j (+ (/ (+ k h) 2) 1) h.
```

The algorithm given so far generates "simple" serial combinators and is fairly straightforward. However, there are several critical optimizations that can be made which have significant effects on performance. They are described below.

C. Refinement 1: Strictness Analysis

Note in step 3a) of the basic algorithm that concurrent subexpressions are found by analyzing occurrences of strict primitive operators. However, we can generally do much better than this. In particular, it is possible that certain serial combinators are strict in some of their arguments as well. This can be inferred even for mutually recursive functions, and techniques for doing so have been discussed in detail elsewhere [15], [20]. Given such strictness information, the refinement to step 3a) is obvious: serial combinators are also generated for the strict arguments to other combinators, and instructions are included in each serial combinator's compiled code for the purpose of spawning strict arguments in parallel.

D. Refinement 2: Complexity Measures

That there is inherent concurrency in an expression does not mean that distributing its evaluation among several processors is the most advantageous thing to do. There may be cases where the cost of distributing concurrent subexpressions far outweighs the cost of computing the whole expression locally. We would like to avoid these cases whenever possible.

Consider an expression **exp** of the form **op** ***e*1** ***e*2**. The total time $T(\mathbf{exp})$ to compute **exp** on a single processor is roughly $T(\boldsymbol{e1}) + T(\boldsymbol{e2}) + C(\mathbf{op})$ where $C(\mathbf{op})$ is the cost of executing the primitive operation **op**. If an extra processor is available, the total time is roughly $\max(T(\boldsymbol{e1}), T(\boldsymbol{e2})$ + distribution costs) + $C(\mathbf{op})$, assuming that ***e*2** was evaluated on the extra processor. Clearly, if $T(\boldsymbol{e2})$ + distribution costs $< T(\boldsymbol{e1}) + T(\boldsymbol{e2})$, then distributing the evaluation of ***e*2** is a win; otherwise, we are slowing down the overall execution, even though there is "apparent" parallelism!

Thus, we refine step 3b) by first performing a simple complexity analysis on the concurrent subexpressions. From the previous paragraph it should be obvious that we should retain the *most* complex subexpression for inclusion in the current serial combinator, making the other subexpression available for possible parallel execution. In the above example, that would mean $T(\boldsymbol{e1}) > T(\boldsymbol{e2})$, guaranteeing a minimal value for $\max(T(\boldsymbol{e1}), T(\boldsymbol{e2})$ + distribution costs). If the analysis then shows that the other subexpression is worth distributing, it is compiled as a separate combinator, otherwise not. As an example, note in **pfac**, presented earlier, that the concurrent subexpression + ***d h*** occurs in the definition of α, but our compiler decides that distributing this expression would not be worthwhile.

Determining accurately the time needed to evaluate an expression is, of course, impossible in the general case. We use an heuristic in which expressions are weighted by the complexity of the primitive operations. These heuristics may be "fine-tuned" to the performance of a particular machine, but the details are not important to this paper.

E. Refinement 3: Common Subexpression Elimination

We would also like to find and eliminate redundant occurrences of subexpressions in order to avoid their reevaluation. The first step in doing this involves using standard compiler techniques to find common subexpressions (cse's) within each supercombinator. We do this *before* having converted them to serial combinators because once the latter is done it will be difficult to detect cse's that involve the bound variables of two or more serial combinators.

For each cse we then determine the smallest expression that contains all instances of that cse (in a graph-theoretic sense this amounts to finding the *least common ancestor* of all occurrences). From this expression we then abstract the cse, creating a new combinator which takes a single instance of the cse as an argument. For example, **pfac** will now become

$$\begin{aligned}
&\text{pfac} = Y\,\gamma\\
&\gamma f = \beta\ (\alpha f)\ f\\
&\alpha\ a\ b\ c\ d\ h = \text{if } (=\ d\ h)\ d\\
&\qquad\qquad (\text{if } (=\ b\ h)\ (*\ d\ h)\\
&\qquad\qquad (\delta\ c\ a\ h\ (/\ (+\ d\ h)\ 2)))\\
&\beta\ p\ q\ l = p\ (+\ l\ 1)\ (q\ l)\ l\\
&\delta\ x\ y\ h\ z = *\ (x\ z)\ (\eta\ y\ z\ h)\\
&\eta\ j\ k\ h = j\ (+\ k\ 1)\ h.
\end{aligned}$$

Note that the formal parameter z in the definition of δ corresponds to the cse $(/\ (+\ d\ h)\ 2)$ in the definition of α, and will only be computed once.

Unfortunately, the cost of accessing the possibly nonlocal value of a cse may be greater than recomputing its value locally! To incorporate this observation we resort again to a complexity analysis to determine if recomputation of a cse is worthwhile. The total time to evaluate n instances of a cse **exp** is

1) $n * T(\mathbf{exp})$, if we recompute the subexpression each time;

2) $T(\mathbf{exp}) + (n *$ cost of getting its value), if we abstract the subexpression and compute it once.

Thus, if the average cost of getting its value is less than the cost of recomputing it, the abstraction is a win. If the value was computed locally, then the cost of getting its value is effectively zero; otherwise, it is roughly twice the communication time to the appropriate processor.

Of course, it is impossible to statically determine the exact cost of access since the distribution of the computation is generally not available until run-time. Thus, our compiler again resorts to an heuristic strategy for cse's. Specifically, an instance of a cse is *duplicated* (meaning it will get recomputed) if and only if the following conditions hold.

1) It refers only to strict variables (variables whose values we know have already been computed).

2) It is concurrent with at least one other instance of the cse.

3) Its complexity is lower than the access cost between two processors.

It remains to be seen whether this set of heuristics is appropriate for a broad range of programs.

F. Refinement 4: "Uncurrying" and Partial Evaluation

In our previous work on distributed combinator reduction [11] we discussed a compile-time technique for partially performing some of the graph reduction in order to "uncurry" the combinators. That is, combinators are generally defined as curried higher-order functions, as in $\alpha = \lambda \mathbf{x}.\ \lambda \mathbf{y}.\mathbf{exp}$. Uncurrying them yields definitions such as $\alpha = \lambda(\mathbf{x},\mathbf{y}).\mathbf{exp}$, which can be implemented in graph reduction more efficiently, especially in a parallel system where granularity is once again a concern.

Such partial evaluation has a surprising additional effect: many applications of the serial combinators can be collapsed into one! This is because a major (and worthwhile) goal of the serial combinator analysis is to produce combinators that are "fully lazy," yet most partial applications of serial combinators are not shared, so there is no reason to preserve this property in all cases. We can also eliminate occurrences of the Y combinator by making the serial combinators directly recursive—this also is a form of partial evaluation.

As an example, if we perform partial evaluation on the serial combinator definition of **pfac** given earlier, we arrive at

$$
\begin{aligned}
&\text{pfac} = \alpha\\
&\alpha\ l\ h = \text{if}\ (=\ l\ h)\ l\\
&\qquad\qquad \text{if}\ (=\ (+\ l\ 1)\ h)\ (*\ l\ h)\\
&\qquad\qquad\quad (\beta\ (/\ (+\ l\ h)\ 2)\ l\ h)\\
&\beta\ c\ l\ h = *\ (\alpha\ l\ c)(\alpha\ (+\ 1\ c)\ h).
\end{aligned}
$$

Although the resulting serial combinators in this case bear a strong resemblance to the original program, in general they may be quite different.

IV. A Distributed Virtual Graph Reducer

With this explication the reader should understand the motivation for our graph reduction strategy using serial combinators as the primitive graph operations. In this section we will describe a virtual graph reduction engine suitable for implementation on a multiprocessor.

A. Abstract Model

As a starting point we will first describe an abstract version of the reducer that makes no reference to multiple processors. In this abstract model there is a single global address space representing a free list of available cells from which the program graph is constructed (think of it as one large heap, in the Lisp sense). Each node in the program graph is labeled with either a primitive or serial combinator, and contains pointers to other nodes whose values are arguments to that combinator (if the argument is already known, its value may be encoded directly). For example, the initial graph for the program **pfac 1 10** will simply be an α node with the arguments 1 and 10. In general, a node may be in one of three states:

1) *dormant,* waiting to be evaluated;
2) *active,* in the process of being evaluated; or
3) *terminal,* completely evaluated.

Evaluation takes place as messages are passed between nodes in the program graph. There are two types of such messages: *get-val* and *return-val,* whose behavior depends on the state of the target node, according to the following rules.

1) *get-val.* State of target node:
   a) dormant—the node is being "demanded" for the first time, so: the target node is made active, a pointer to the "requesting" node is saved, and the combinator code is executed;
   b) active—evaluation of the node has already begun, so it is only necessary to save a pointer to the requesting node;
   c) terminal—the target node has already been evaluated, so its value is sent via return-val message to the requesting node.
2) *return-val.* State of target node:
   a) dormant—error;
   b) active—the return value is an argument to the combinator represented by the target node, so it is saved and the compiled combinator code is resumed if appropriate;
   c) terminal—error.

The execution of the combinator code in rule 1a) may cause further graph mutations and state transitions. For example, it may send get-val messages in pursuit of its arguments, or it may compute a final value for the node, which is stored in the graph and sent to all requesters via a return-val message. In the latter case the node then becomes terminal.

B. Multiprocessor Model

On a multiprocessor without shared memory, the above scenario only requires a few changes since the message-based strategy serves well as an interprocessor communications protocol. Each processor is made responsible for one contiguous portion of the global address space, and thus parallelism comes to bear when concurrent redexes reside in the address spaces of different processors. A *task queue* is maintained on each processor that essentially contains get-val or return-val messages, which act as pointers to available redexes, and which are processed one at a time. Initially, all processors are given copies of the compiled serial combinators (this corresponds to the "pure code" described in [11]). Program execution occurs by mapping the initial program graph onto the global address space and sending a get-val message to the root node. Eventually, a return-val message is sent from the root node to the "system," indicating program termination.

As the evaluation unfolds, expanding portions of the graph are distributed to neighboring processors for increased parallelism. This distribution process is controlled by a dynamic load-balancing mechanism that we refer to as *diffusion scheduling,* which may take into account such factors as processor load, memory utilization, and direction of global references. The intent is for tasks to be "pushed away" from busy processors and "drawn toward" those to which they have global references (thus maintaining locality). In this way work "diffuses" through the network in the direction of least resistance.

As an example of one such diffusion strategy, one may choose to evaluate a dormant node n on the neighboring

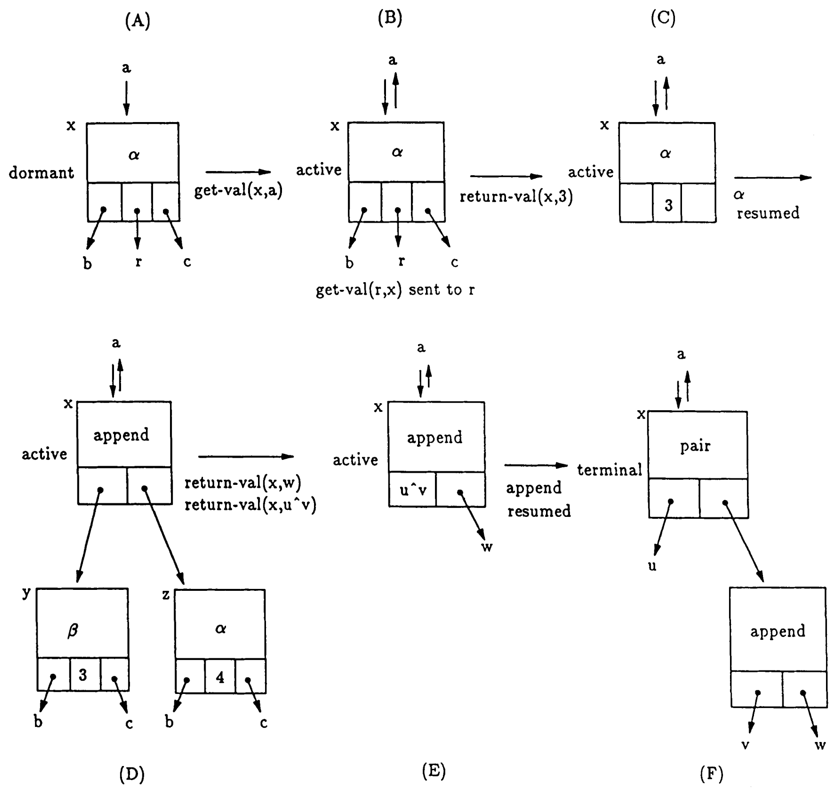

Fig. 1. Example of serial combinator reduction.

processor p having the minimum value of $C(n,p)$ where

$$C(n,p) = \text{load}(p) + (k * \text{ref-cost}(n,p)).$$

Ref-cost(n,p) is a measure of the cost of n's references to nodes residing on processors other than p, load(p) is simply the number of tasks on p's task queue, and the constant k is a weighting factor indicating the relative importance of the two parameters—the lower the value of k, the more likely it is that work simply diffuses towards the least-loaded processors.

An important optimization to this overall strategy is to avoid the message-passing protocol for local communications, bypassing the task queue entirely whenever possible. This, coupled with the fact that the combinator definitions are represented as *conventional compiled straight-line code*, means that we can take advantage of the efficient sequential features of the von Neumann processors, using message passing only when necessary.

C. A Simple Example

It is helpful to proceed through a short example showing a few reduction steps for a typical serial combinator. Consider the following definition of the serial combinator α, taken from the "six queens" program given in Section V-D:

```
α bd row col = if (= row 6) [ ]
                  (append (β bd row col)
                          (α bd (+ row 1) col)).
```

Suppose we are about to evaluate the application of α to three arguments, shown graphically in Fig. 1(a) as the dormant node x. Fig. 1(b)–(f) then shows five reduction steps.

Step 1) The message get-val(x,a) represents a request from node a for x's value. Node x then becomes active, and a pointer to a is saved. The code for α is then executed, which causes a get-val message to be sent in pursuit of r's value.

Step 2) Eventually, a return-val message produces a value (in this case 3) for r, which is saved in the graph.

Step 3) Since x was only awaiting the value of r, the activity of α is resumed. Following now the definition of α, we note that $r \neq 6$—thus x is mutated into an append node, and two new nodes, y and z, are created (possibly on some other processor due to the work of the diffusion scheduler). The code for append is then invoked, which causes two get-val messages to be spawned in pursuit of the values of y and z.

Step 4) Eventually, values also return for y and z (note that the value of a list is really a pair of values, rather than a pointer to the pair—this is yet another optimization aimed at reducing communication costs).

Step 5) The code for append is then resumed, which causes a transformation of x into a pair. At this point x's value has been completely determined; therefore, it becomes terminal, and a return-val message is sent to node a.

V. Simulation Results

Our intent is that the virtual machine described in the last section may be implemented on a variety of multiprocessor architectures, and thus the details may vary significantly from machine to machine. Previous work by the authors and others [11], [19] has demonstrated via simulation the viability of the overall strategy, and we are currently involved in an implementation on a 128-node Intel iPSC hypercube. In this section we present new simulation results using serial combinators.

Our simulator allows one to easily change parameters such as the number of processors, network topology, and diffusion heuristics. In all of the simulation results given below, the diffusion heuristic is as described in Section IV-B with $k = 0$. Other parameters are adjusted depending on the experiment. We measure execution time by taking as a unit time step the execution of a single memory-to-memory CPU instruction, such as a move or addition. Other more complex operations are weighted accordingly—for example, we use a message delay of ten time steps between adjacent processors.

A principal result that we compute for each simulation is the *speedup factor,* i.e., the ratio of time taken to compute the result on one processor to that taken on the whole system. This number is somewhat "introspective" since we are comparing results within our own evaluation model, but it is nevertheless a useful tool.

A. Simple Comparison of Serial to Standard Combinators

In our earlier experiments [11] we used a standard fixed set of combinators, and it is helpful to compare their efficiency to that of serial combinators. One such comparison is shown in Fig. 2 for **pfac 1 32** run on a torus network of various sizes. Since the unit time steps were slightly different in the two cases, it is probably dangerous to draw concrete conclusions from these numbers; nevertheless, we believe the results demonstrate that serial combinators provide a remarkable time saving over standard combinators when used in this model of computation. The reason for the large disparity, which appears considerably larger than comparisons between standard and supercombinators [17], is that our serial combinators are executed as conventional compiled machine code, consistent with the target multiprocessor architecture. On the other hand, note that the speedup factor has a maximum value of about 8 for the standard combinators, compared to only 2.3 for the serial combinators. This is because the higher overhead in using a fixed set of combinators is itself being distributed for parallel execution! The introspective speedup factor thus shows "apparent" parallelism that is in fact wasteful. Note, by the way, that the speedup for serial combinators decreases after 8 processors—this reflects the computation being spread "too thin."

number of processors	1	2	4	8	16	32
serial combinators	491	309	230	216	233	233
standard combinators	12206	6196	3335	2252	1625	1519

Fig. 2. Total cycles consumed, serial versus standard combinators.

number of processors	1	2	4	8	16
with cse elimination	16363	8437	4433	2559	1698
speedup	1.0	1.9	3.7	6.4	9.6
without cse elimination	21473	10843	5649	3018	1843
speedup	1.0	2.0	3.8	7.1	11.7

Fig. 3. Comparison of serial combinator execution time, with and without cse elimination.

B. The Effect of CSE Elimination

We also conducted a series of experiments to test the effectiveness of common subexpression elimination in the **pfac** program. The results, shown in Fig. 3, are for **pfac 1 1024** on a torus, and demonstrate that even though the speedup exhibited without cse elimination was always greater than the speedup with cse elimination, the latter consistently required less time to execute.

C. Variations in Program Size

Fig. 4 shows a large number of data points for the **pfac** program on a torus, showing variations in program size and number of processors. The results demonstrate that the degree of parallelism scales well with program size, as well as with number of processors (up to the point where the number of processors exceeds the maximum available parallelism in the program).

D. A Larger Example: Six Queens

Consider the following ALFL program to generate all solutions to the "queens" problem on a six-by-six board:

```
{n = 6;
 Result allboards [ ] 0;
 allboards board col =
   {Result col=n → [board], findboards 0;
    findboards row =
      {Result row=n → [ ], newbds ^^findboards (row + 1);
       newbds = safe board (col − 1) →
                   allboards (row ^ board) (col + 1), [ ];
       safe  [ ]  coll = true;
       safe (r ^ bd) coll = r≠row & coll≠col &
                             abs(r − row)≠abs(coll − col) &
                             safe bd (coll − 1)
      }
   }
}
```

The symbols " ^ " and " ^^ " are infix operators for cons and append, respectively, and **hd** and **tl** are like car and cdr, respectively, in Lisp. A list may also be constructed using brackets, as in $[a, b, c]$ (which is equivalent to a^b^c^[ ]). We will assume that the append operator in this example eagerly evaluates both of its arguments. In practice this can be ac-

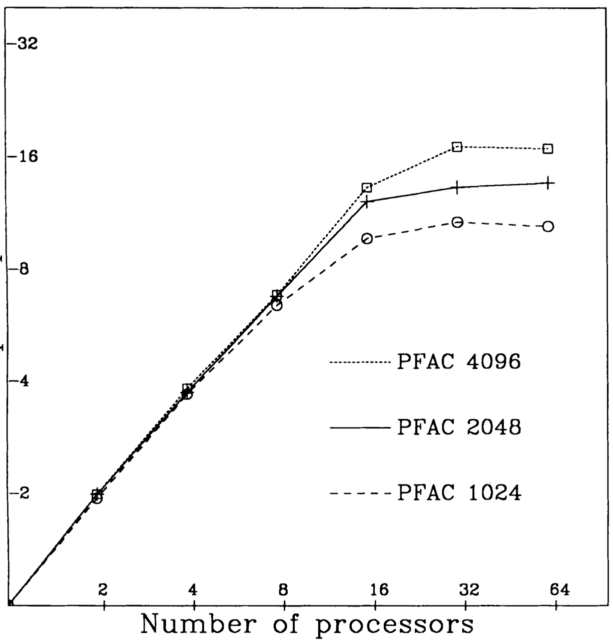

Fig. 4. Parallelism scales with program size.

complished either by providing such a strict operator as a primitive, or by annotating the source program. For example, one could write

··· **newbds ^^ #(findboards (row + 1))** ···

where the # sign indicates that the second argument is to be evaluated in parallel. Such annotations are beyond the scope of this paper, but are discussed thoroughly in [16].

In supercombinators the queens program becomes

```
Result C1 (Y C12)
C12 allbds = C11 (C10 (C8 allbds))
C11 v29 bd = v29 (^ bd [ ]) (C4 bd) bd
C10 v26 v25 v27 v28 col =
  C2 (= col 6) v25 (Y (v26 (v27 (− col 1)) v28 (+ col 1) (C6 (C5 col))))
C8 v20 v18 v21 v22 v23 v19 row =
  v18 (C3 (= row 6) (v19 (+ row 1))) (v20 (^ row v21) v22) (Y (v23 row))
C6 v14 v15 v16 bd =
  v14 (= bd [ ]) (≠ (hd bd) v15) (abs (− (hd bd) v15)) (v16 (tl bd))
C5 v11 v9 v10 v12 v13 col1 =
  if v9 true (& v10 (& (≠ col1 v11)
                    (& (≠ v12 (abs (− col1 v11))) (v13 (− col1 1)))))
C4 v6 v7 v5 v8 safe = v5 (if (safe v6 v7) v8 [ ])
C3 v3 v4 newbds = if v3 [ ] (^^ newbds v4)
C2 v1 v2 findbds = if v1 v2 (findbds 0)
C1 allbds = allbds [ ] 0.
```

And finally in serial combinators:

```
Result δ [ ] 0
δ bd col = if (= col 6) (^ bd [ ]) (α bd col 0)
α bd col row = if (= row 6) [ ] (^^ (β bd col row)
                                    (α bd col (+ row 1)))
β bd col row = if (γ col row bd (− col 1))
                  (δ (^ row bd) (+ col 1)) [ ]
γ col row bd col 1 = if (= bd [ ]) true
                       (η col row (hd bd) (tl bd) col 1)
η col row hdbd tlbd col 1 =
  (& (≠ hdbd row) (& (≠ col 1 col) (& (≠ (abs (− hdbd row))
                                         (abs (− col 1 col)))
                                    (γ col row tlbd (− col1 1))))).
```

Fig. 5 shows this "six queens" program run on various numbers of processors, using both a torus and hypercube topology. It is interesting to note that the results for the torus and hypercube are for the most part identical! It remains to be seen whether other programs and diffusion heuristics will exhibit different behavior, such that the lower diameter of the cube will become significant.

VI. Related and Future Work

Turner reports reasonably good performance in using a combinator reducer on a conventional machine [29], especially when compared to a normal-order reducer implemented using a version of Landin's SECD machine [8]. More extensive results have been reported by Peyton Jones [21]. Combinators have also been used as the intermediate code of a compiler targeted for a conventional machine [13].

A fair amount of work has been done in the area of reduction and data flow machines (the list is too long to include here, but see the summary in [27]), but little has been done in the way of building machines specifically for combinator reduction. Two important exceptions to this are the SKIM machine [3] and Burroughs' NORMA [23], although they are both strictly sequential machines and use a fixed set of combinators.

To our knowledge the first use of a "diffusion-style" task scheduler was by Halstead [6], [7], although his work was based on the actor model of computation rather than graph reduction (cf. [9], [10]). A more recent effort is Keller's Rediflow multiprocessor, a blend of data flow and reduction

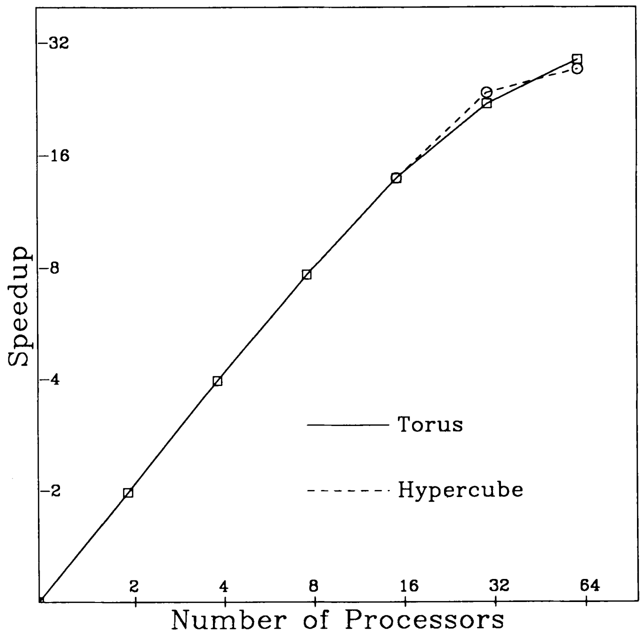

Fig. 5. Six queens on torus and hypercube.

ideas [19], but using a lambda calculus reducer instead of combinators.

An interesting open research problem for serial combinators is determining strategies for *migrating code*. Instead of duplicating the serial combinator definitions among the processors, one might have them migrate dynamically by demand. This introduces a "meta-garbage-collection" problem, in that it becomes necessary to keep track of what portion of the pure code each processor has and where the nearest source of that code resides. ("Reference trees" are one solution to this problem [7].)

From a language viewpoint we have also been exploring ways for experienced programmers to provide more precise control over their parallel programs. In particular, it is possible to map a user's program to a particular multiprocessor topology by annotating the source (see [16] for details). This is an important capability for programmers who know precisely the optimal way that their programs should be distributed for parallel evaluation.

In many ways the research that we have reported only scratches the surface—few people have any useful experience with this style of computing and the architectures that it implies. Ultimately, a machine might be specially tailored for distributed serial combinator reduction, but we currently do not have sufficient empirical data to convince us of the right design choices. It has been our hope that with an abstract model and flexible simulation tools, these parameters will become more obvious as our research progresses. Our pending implementation on a 128-node Intel iPSC will be a useful testbed for many of our ideas.

Acknowledgment

The authors gratefully acknowledge the support of the Burroughs Austin Research Center (in particular C. Pixley and L. Thomas), whose imaginative research efforts make academic/industrial research collaborations a pleasure.

References

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 4, pp. 613–641, Aug. 1978.
[2] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. Amsterdam, The Netherlands: North-Holland, 1984.
[3] T. Clarke, P. Gladstone, C. MacLean, and A. Norman, "SKIM—The *S,K,I* reduction machine," in *Proc. 1980 LISP Conf.*, Stanford Univ., Stanford, CA, Aug. 1980, pp. 128–135.
[4] H. K. Curry and R. Feys, *Combinatory Logic*. Amsterdam, The Netherlands: North-Holland, 1958.
[5] J. Darlington, P. Henderson, and D. A. Turner, Eds., *Functional Programming and Its Applications*. Cambridge, England: Cambridge Univ. Press, 1982.
[6] R. H. Halstead, Jr., "Multiple-processor implementations of message-passing systems," Lab. Comput. Sci., Mass. Inst. Technol., Cambridge, Tech. Rep. MIT/LCS/TR-198, Jan. 1978.
[7] ——, "Reference tree networks: Virtual machine and implementation," Lab. Comput. Sci., Mass. Inst. Technol., Cambridge, Tech. Rep. MIT/LCS/TR-22, 1979.
[8] P. Henderson, *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
[9] C. Hewitt, "Viewing control structures as patterns of passing messages," Mass. Inst. Technol., Cambridge, Working Paper 92, Apr. 1976.
[10] ——, "Design of the APIARY for actor systems," in *Proc. 1980 LISP Conf.*, Stanford Univ., Stanford, CA, Aug. 1980, pp. 107–118.
[11] P. Hudak and B. Goldberg, "Experiments in diffused combinator reduction," in *Proc. ACM Symp. Lisp Functional Programming*, Aug. 1984, pp. 167–176.
[12] P. Hudak, *ALFL Reference Manual and Programmers Guide*, 2nd ed., Yale Univ., New Haven, CT, Tech. Rep. YALEU/DCS/TR-322, Oct. 1984.
[13] P. Hudak and D. Kranz, "A combinator-based compiler for a functional language," in *Proc. 11th ACM Symp. Principles Programming Lang.*, Jan. 1984, pp. 121–132.
[14] P. Hudak and B. Goldberg, "Serial combinators: "Optimal" grains of parallelism," presented at 1985 Conf. Functional Programming Comput. Architecture, Sept. 1985.
[15] P. Hudak and J. Young, "A set-theoretic characterization of function strictness in the lambda calculus," Yale Univ., New Haven, CT, Tech. Rep. YALEU/DCS/TR-391, Jan. 1985.
[16] P. Hudak and L. Smith, "Para-functional programming: A paradigm for programming multiprocessor systems," Yale Univ., New Haven, CT, Tech. Rep. YALEU/DCS/TR-390, Jan. 1985.
[17] R. J. M. Hughes, "Super-combinators: A new implementation method for applicative languages," in *Proc. ACM Symp. Lisp Functional Programming*, Aug. 1982, pp. 1–10.
[18] R. M. Keller, "FEL programmer's guide," Univ. Utah, Salt Lake City, Tech. Rep. AMPS TR 7, Mar. 1982.
[19] R. M. Keller and F. C. H. Lin, "Simulated performance of a reduction-based multiprocessor," *IEEE Comput.*, vol. 17, pp. 70–82, July 1984.
[20] A. Mycroft, "The theory and practice of transforming call-by-need into call-by-value," in *Proc. Int. Symp. Programming*, vol. 83. New York: Springer-Verlag, 1980, pp. 269–281.
[21] S. L. Peyton Jones, "An investigation of the relative efficiencies of combinators and lambda expressions," in *Proc. ACM Symp. Lisp Functional Programming*, Aug. 1982, pp. 150–158.
[22] J. A. Rees and N. I. Adams, "T: A dialect of LISP or, Lambda: The ultimate software tool," in *Proc. ACM Symp. Lisp Functional Programming*, Aug. 1982, pp. 114–122.
[23] H. Richards, Jr., "An overview of the Burroughs NORMA," Burroughs Austin Res. Cen., Jan. 1985.
[24] M. Schonfinkel, "Uber die bausteine der mathematischen logik," *Math. Ann.*, vol. 92, p. 305, 1924.
[25] G. L. Steele and G. J. Sussman, "The revised report on scheme," Mass. Inst. Technol., Cambridge, Tech. Rep. AI 452, Jan. 1978.
[26] J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. Cambridge, MA: M.I.T. Press, 1977.
[27] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architectures," *Comput. Surv.*, vol. 14, no. 1, pp. 93–143, Mar. 1982.

[28] D. A. Turner, *SASL Language Manual,* Univ. St. Andrews, 1976.
[29] ——, "A new implementation technique for applicative languages," *Software-Practice Experience,* vol. 9, pp. 31–49, 1979.

Paul Hudak (S'79–M'82) was born in Baltimore, MD, on July 15, 1952. He received the B.S. degree in electrical engineering from Vanderbilt University, Nashville, TN, in 1973, the M.S. degree in computer science from the Massachusetts Institute of Technology, Cambridge, in 1974, and the Ph.D. degree in computer science from the University of Utah, Salt Lake City, in 1982.

From 1974 to 1979 he was a Member of the Technical Staff at Watkins-Johnson Company, Gaithersburg, MD. He is currently an Assistant Professor in the Programming Languages and Systems Group, Department of Computer Science, Yale University, New Haven, CT, a position he has held since 1982. His primary research interests are functional and logic programming, fifth-generation computer architecture, and semantic program analysis.

Dr. Hudak is a member of the ACM (SIGPLAN, SIGARCH, SIGOPS, SIGACT), Eta Kappa Nu, and Sigma Xi. He is also a recipient of an IBM Faculty Development Award (1984) and a Presidential Young Investigator Award (1985).

Benjamin Goldberg was born in Las Cruces, NM, on January 31, 1961. He received the B.A. degree in mathematical sciences from Williams College, Williamstown, MA, in 1982 and the M.S. degree in computer science from Yale University, New Haven, CT, in 1984. He is currently a fourth-year doctoral student in the Programming Languages and Systems Group at Yale where he is designing a system called ALFALFA, a multiprocessor implementation of serial combinators.

His general research interests include parallel computer architecture, programming language semantics, and functional programming.