

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

TITLE: HARNESSING COMPUTATIONAL POWER: DISTRIBUTED
COMBINATOR REDUCTION

LA-UR--85-205

DE85 005967

AUTHOR(S): Randy E. Michelsen, C-10
Joseph H. Fasel, C-10

SUBMITTED TO: SIAM publication of Proceedings of USARO Workshop
on Novel Computing Environments. Meeting held
at Stanford University, Stanford, CA, November 7 - 9, 1984

DISCLAIMER

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

MASTER

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

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

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Harnessing Computational Power: Distributed Combinator Evaluation

Randy E. Michelsen* and Joseph H. Fasel*

Abstract. Much interest has been generated by the notion of combinator graph reduction as a mechanism for the implementation of functional languages. An effort is currently in progress at Los Alamos National Laboratory, in collaboration with Paul Hudak of Yale University, to develop a testbed distributed implementation of a functional language based on this concept. Our overall goal is to facilitate the exploitation of implicit parallelism; within this framework, we intend to use this testbed to evaluate the utility of strategies for combinator graph reduction in a distributed computing system.

1. Introduction. The demands for enhanced computing performance through concurrency and for increased programming productivity through very high level languages have fueled a reevaluation of the traditional, imperative model of computation, which is based on the von Neumann computer. The inherent limitations of this model, coupled with the expectation of decreasing gains in performance to be obtained from improvements in hardware technology, have led to much interest in alternative models of computation. Among these, the *functional* (or *applicative*) model appears particularly promising for parallel computation. This model is based solely on the application of functions to their arguments, or put another way, on the evaluation of pure expressions, and is thus inherently parallel. By contrast, the imperative model is concerned with modifiable storage cells (which appear as so called *variables* in imperative programming languages), and consequently, with sequential control of the process of altering and examining them. Functional languages also afford a more expressive medium for programming, by freeing the programmer from concern for storage allocation, sequencing, or the explicit control of parallelism,¹ and by the use of higher order functions to generalize common programming paradigms.

Functional programming languages may be considered syntactic variants of the lambda calculus [2]; thus, a natural operational model of functional program evaluation is lambda conversion, or any other *reduction* process. In particular, graph reduction is advantageous in either a uniprocessor or multiprocessor context, because it facilitates the sharing of values. There has been much interest recently in reduction

*University of California, Los Alamos National Laboratory, Los Alamos, NM 87545 Work performed under the auspices of the U.S. Department of Energy

¹Detection of implicit parallelism in imperative programs [8] is also possible (and desirable), but is likely to be much more difficult, and is fundamentally limited by the sequential nature of the underlying

schemes based on combinator calculi [2], rather than the lambda calculus, as there is some evidence to suggest the former are more efficient [5,9]. The case for combinator reduction is particularly compelling for distributed computation.

We here describe a project underway at Los Alamos National Laboratory to evaluate the utility of the functional model of computation when realized in a distributed, combinator-based reduction architecture. In Section 2, we discuss some of the technical underpinnings of this effort, while Section 3 provides an overview of our implementation. In the final section, we summarize and present possible avenues of future research.

2. Technical Background. The subject of functional programming languages can hardly be considered a fledgling research topic [1,6]. Similarly, concurrency issues have long been of interest to operating system and programming language designers. The referential transparency of functional languages (*i.e.*, the meaning of a function is independent of the context in which it is used) guarantees that programs behave consistently, whether executed serially or in parallel.

Reduction as an operational model provides an amenable setting for the implementation of functional languages. This model, traditionally based on the conversion rules of the lambda calculus [2], is characterized by the lack of any sequencing constraint other than that imposed by the demand for constituent values during evaluation. Also absent from the model is the concept of alterable storage, the hallmark of the traditional control flow model, and a contributor to many of its unattractive characteristics. In addition, when the expressions to be reduced are represented graphically, rather than linearly, a *fully lazy* evaluation scheme is possible, by which each subexpression is evaluated at most once. This lends the graphs a "self-optimizing" property [9], in which the equivalents of some code movement optimizations for traditional programming languages, such as moving invariant expressions out of loops, occur automatically at run time.

In the purest form of the reduction operational model, parallelism is completely implicit; it arises from the application of *strict* functions, those that require values for all their arguments to yield a result. When a value is demanded from such a function, the demand may propagate to each of its arguments simultaneously. Where, to improve efficiency, it is desirable to involve the programmer in the control of parallelism, we may consider annotating an expression with suggestions about how subexpressions should be allocated to processors, which subexpressions are not worth evaluating in parallel, or conversely, which applications of nonstrict functions might profitably be evaluated *eagerly* (as if they were strict). The advantage of this scheme over explicit parallelism in imperative languages with facilities for process creation and synchronization is that by annotating a program, we may change its performance characteristics, but not its result. Functional programs are determinate. (Determinacy, after all, is the nature of functions!)

An alternative to the lambda calculus is a *combinator* calculus, in which the bound variables of lambda expressions have been eliminated by the introduction of a small set of functions called combinators [3]. Just as functional programming languages may be regarded as syntactic variants of the lambda calculus, a combinator calculus can be considered a kind of machine code for these languages; thus Turner [9] has suggested *compiling* functional programs into graphical representations of combinator expressions. An evaluator for combinator expressions can be significantly faster than an evaluator for lambda expressions, such as the SECD machine [6], because in the absence of bound variables, there is no need for an *environment* in which to record the bindings of variables to values. A good deal of overhead in the creation and retrieval of bindings is thus eliminated, the environmental information having been compiled into the graph. For distributed computation, the gain in efficiency is even more significant. A centralized environment can be a serial bottleneck, whereas in a combinator graph, the environmental information is *distributed*, each subexpression being supplied just the values it needs. In the analogy of a combinator calculus to a von Neumann instruction set, combinators are like instructions such as loads, stores, and jumps, in that they provide the mechanism that links the outputs of primitive operations (such as addition) to the inputs of other operations. Theoretically, two combinators suffice [3]; practically, we may desire a larger, though perhaps still modest, collection of combinators that capture several common patterns of communication.

Hughes [4] has suggested a variation on Turner's combinator reduction scheme, in which the set of combinators is not fixed for all programs but is optimally chosen for the expression at hand. These *supercombinators* represent the largest subcomputations, derived from the original lambda expression, that preserve full laziness. If Curry-Turner combinators are like the fixed instruction set of a general-purpose computer, supercombinators are like specialized microcoded instructions, made to order for a particular problem. Hudak [7] has suggested a further variation on supercombinators for parallel processing. His *serial combinators* are refinements of supercombinators that have no concurrent substructure. They thus embody the smallest reasonable granularity for concurrency in a computation; we may designate a serial combinator as a schedulable unit of computation, or we may choose to combine several of them, but there is no point in decomposition below this level.

3. Current Implementation. The system presently under development uses a collection of Symbolics Lisp machines connected by an Ethernet and is intended to provide a testbed for further research. The system is based on a message-passing paradigm, with communication, scheduling, and reduction embodied in separate processes. The major functional units of the system are now under code development.

In a typical scenario, a source program in the functional programming language ALFL [7] is compiled into a serial combinator graph. The serial combinators themselves are compiled into Lisp, and from Lisp, into executable code. Code for all serial combinators, as well as that for the supporting scheduling and communication processes, is distributed to each machine in the network. Execution begins on one machine; as the graph unfolds during the process of reduction, it diffuses through the

network. It is reduced by cooperating elements of the processor network, eventually to normal form. The diffusion of the combinator graph is controlled by a decentralized, load-balancing scheduling algorithm [7]. This algorithm attempts, through distribution of spawned tasks to neighboring processors with available capacity, to disperse work through the network while preserving locality.

4. Conclusion and Directions for Future Research. This paper has presented the rationale for, and an overview of, a project at Los Alamos National Laboratory to develop and analyze a distributed implementation of a functional language. The operational model employed is serial combinator reduction.

Completion of this implementation will provide us a testbed to evaluate combinator-based graph reduction schemes in an actual multiprocessor environment, and to refine our techniques in compilation, scheduling, load balancing, and communication. It can also give us the means for experimenting with source language annotations to aid in the control of concurrency. In addition, the implementation will help us to evaluate the applicability of functional languages to Laboratory problems in science, engineering, and artificial intelligence.

We hope to apply the experience gained from this work to other varieties of computer architecture to be found among the Laboratory's computing resources. In particular, we believe that graph reduction is appropriate for tightly coupled, common memory multiprocessor configurations, as well as for distributed ones. In fact, common memory should enhance the value sharing aspect of graph reduction, while simplifying the problems of load balancing and communication. Of course, one cannot indefinitely add processors to a common memory machine without degrading performance; thus, distribution of processing power becomes necessary. On the basis of increases in computing power needed for some Los Alamos problems, we can envision large-scale processing networks with each node a fairly powerful shared memory multiprocessor, perhaps on the order of a Cray X-MP.

REFERENCES

- [1] J. BACKUS, *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Comm. ACM 21 (8), pp. 613-641.
- [2] A. CHURCH, *The calculi of lambda conversion*, Annals of Mathematical Studies, Princeton University Press, Princeton, NJ, 1941.
- [3] H. K. CURRY and R. FEYS, *Combinatory Logic, Volume I*, North-Holland Publishing, Amsterdam, 1958.
- [4] R. J. M. HUGHES, *Super-combinators: A new implementation technique for applicative languages*, Proc. 1982 ACM Symp. on Lisp and Functional Programming, pp. 1-9.
- [5] S. L. P. JONES, *An investigation of the relative efficiencies of combinators and lambda expressions*, Proc. 1982 ACM Symp. on Lisp and Functional

Programming, pp. 150-158.

- [6] P. J. LANDIN, *The mechanical evaluation of expressions*, Computer Journal, 6, 4 (1963), pp. 308-320.
- [7] P. HUDAK and B. GOLDBERG, *Experiments in diffused combinator reduction*, Proc. 1984 Symp. on Lisp and Functional Programming, pp. 167-176.
- [8] K. OTTENSTEIN, *A brief survey of implicit parallelism detection*, in J. S. KOWALIK (Ed.), *MIMD Computation: The HEP Supercomputer and its Applications*, MIT Press, Cambridge, Mass., to appear 1985.
- [9] D. A. TURNER, *A new implementation technique for applicative languages*, Software-Practice and Experience, 9 (1979), pp. 31-49.