High-Performance Functional Computing: Guest Editors' Introduction

R. L. PAGE[1] AND R. R. OLDEHOEFT[2]

[1]*School of Computer Science, University of Oklahoma, Norman, OK 73019*
[2]*Computer Science Department, Colorado State University, Fort Collins, CO 80523*

The 1995 High Performance Functional Computing Conference was held on April 9–11, in Denver, Colorado. The meeting, sponsored by Lawrence Livermore National Laboratory and Colorado State University, included a keynote address by Guy Steele on "What Good is Functional Programming?," 19 contributed papers* (from 38 submissions), and a lively panel discussion led by Jack Dennis, titled "Nondeterminacy in Functional Programming: An Essential Feature or a Programmer's Nightmare?" Among these articles, we found several that we believe will interest readers of this journal. This is not surprising, since scientific applications dominate high-performance computing, and functional language researchers increasingly use these applications to test their ideas. We hope readers will agree that the ideas presented in this special issue illustrate many benefits of functional languages in scientific programming.

WHY FUNCTIONAL LANGUAGES?

A functional language enables a computation to be specified entirely in the realm of formulas and equations, rather than the step-by-step procedures of conventional programs. One uses built-in operations and function definitions to describe a computation as a formula that relates values to results. Equations associated with the formula define needed functions and data values.

Referential transparency is the important characteristic distinguishing the functional approach from the familiar imperative paradigm. The two expressions

$$f(x, y, z) + f(x, y, z)$$

$$\textbf{let } a = f(x, y, z) \textbf{ in } a + a$$

have the same value because a and $f(x, y, z)$ may substitute for each other freely. Definitions in a functional program associate permanent values with names (rather than temporary values subject to change at a later time), which means that expressions that appear to be equal (because they combine values in the same way) are, in fact, equal.

As a consequence, functions produce values but introduce *no side effects*. We may abbreviate an expression by associating a name to be used in its place, and the name retains this value throughout its existence—this is *single assignment*. The name is bound to a value, not to a memory location, so there is *no aliasing* to cause coherence problems.

In the imperative paradigm, computation proceeds by altering values stored in memory locations, following a determinate sequence. In the functional approach, a formula denotes a value within the context of a collection of equations. Recursion, which can establish values in new contexts, replaces the notion of an iterative sequence of states.

Since functions are central in the functional approach, some languages treat them on par with other values; *higher-order functions* take functions as parameters and yield functions as results. Some languages support *currying*, i.e., partial

Received and revised April 1995

* All conference papers, in Postscript format, are available via anonymous ftp from sisasl.llnl.gov, in /pub/hpfc/papers95/.

Scientific Programming, Vol. 5, pp. 93–95 (1996)
CCC 1058-9244/96/020093-03

evaluation of functions. A function, presented with some of its arguments, defines a new function whose inputs are the unspecified arguments. This curried form is a more specialized and potentially more efficient function.

Another common feature of functional languages is *type inference.* Instead of requiring the programmer to provide the type of each name explicitly, it is possible to infer the types of some values from the types of input values and the transformations done by the operations, then build on such inferences to determine the types of all (or nearly all) of the expressions in a program without reference to explicit type declarations. When such languages contain intrinsic operations that can deal with multiple types of data, defined functions can inherit this *polymorphic* character, which can greatly extend the notion of reusable code.

These language features yield important benefits. First, programs may be reasoned about in a straightforward way because function definitions are mathematically sound, and side effects and aliasing do not get in the way. The long-sought goal of proving nontrivial programs to be correct is achievable with functional languages.

Second, programs are more concise because of their formulas often refer to large data aggregates rather than individual units. Higher-order functions, type inference, and polymorphism also contribute to conciseness. This is important because smaller programs are easier to write and maintain than larger ones.

Third, compilers can automatically parallelize programs to exploit multiple processor machines. The lack of imperative sequencing, side effects, and aliasing problems leaves data dependency as the only mechanism constraining the order of operations. No complex analysis is required to uncover potential parallelism.

THE SELECTED PAPERS

The growing relevance of the functional approach in hiding machine dependencies and providing portability, while successfully exploiting parallel systems, has energized research in the functional approach to high performance on parallel machines. We hope some of this energy, as seen in the HPFC Conference, will stimulate the imagination of readers of this special issue of *Scientific Programming.*

"Integrating Imperative and Functional Programming in Real-World Applications," by Tom DeBoni, John Feo, and Doug Peters, discusses parallelizing a legacy code for molecular dynamics. In the original Fortran 77 program, a kernel contained 30% of the source lines but accounted for 90% of the execution time. They rewrote the kernel in Sisal, and integrated the new, highly parallel version with the rest of program through Sisal's foreign language interface. Results show useful speedup with modest effort.

"Functional Implementations of the Jacobi Eigensolver," by Wim Böhm and Bob Hiromoto, is a description of the development of versions of this algorithm in the Id language, and their execution in the Monsoon dataflow machine simulation environment. They show that this algorithm is efficiently expressible in Id's functional paradigm, and that a version using fewer features (in Sisal) is also expressible and efficiently executable. Of note is their improvement in the sweep of Sameh's technique for groupwise parallel rotations from $O(n^4)$ to $O(n^3)$.

"Static Mapping of Functional Programs: An Example in Signal Processing," by Jack Dennis, views signal-processing programs as modules exchanging streams of data. Composed modules can be mapped statically onto multiprocessors to exploit the inherent parallelism in these applications. Each module becomes a set of threads to run on a group of processing elements. The article discusses performance for conventional DSP hardware and for a parallel architecture.

In "Fast Digit-Index Permutations," by Dorothy Bollman, Jaime Seguel, and John Feo, a tensor sum approach provides a unifying framework for digit-reversal algorithms within fast Fourier transform (FFT) computations. The authors obtain a new algorithm for the general class of digit-index permutations whose parallel-time complexity is $O(\log \log n)$. This, along with their earlier work in using a tensor product formulation for the combining phase in FFT algorithms, yields a complete set of tools for designing, modifying, and implementing high-performance FFTs in functional languages.

"Update-in-place Analysis for True Multidimensional Arrays," by Steven Fitzgerald and Rod Oldehoeft, introduces a new, more general method for update-in-place analysis that is applicable to multidimensional arrays as well as the vector-of-vectors storage layout used in the current Sisal. This analysis avoids excessive copying in functional languages with single-assignment semantics. Multidimensional arrays have several expres-

sivity and performance advantages. The new method also works for individual functions as well as whole programs.

"Compiler-Enforced Cache Coherence Using a Functional Language," by Rich Wolski and David Cann, discusses support for correct, determinate execution on a parallel system with no hardware support for cache coherence. Strict language semantics helps elimination of stale cache data, and automated data management allows alignment and partitioning to avoid false sharing. This computer hardware was so difficult to deal with that efforts to produce compilers for imperative languages were never successful; the analysis of potentially interfering execution sequences was simply too difficult. On the other hand, efforts to produce compilers for functional languages succeeded in short order, indicating that the functional approach can yield time-to-market benefits, in addition to the high-performance benefits and program correctness benefits discussed in other articles in this issue.