A debate: Retire FORTRAN?

New applicative languages, based on mathematical functions rather than FORTRAN-like statements, will enhance our ability to decompose problems for concurrent processing.

James R. McGraw

What role can programming languages play in our continuing efforts for high-speed computation, and what role *should* they play? The overwhelming majority of high-speed computing is now done in FORTRAN on large vector machines such as the Cray 1 and the CDC Cyber 205. While the exact nature of future supercomputers is unclear, by almost all indications they will be multiprocessors. Conservative plans would link on the order of 64 processors, while more aggressive approaches, such as the "data flow" strategy, would link several thousand processors. In either case, the goal is to arrange each program to use as many processors as needed to complete execution in the shortest amount of time. To what extent can languages and their associated compilers assist us in reaching this goal?

Three different options have received the most attention to date.

▶ We can stay with FORTRAN and invest all energy in better compilation and optimization techniques for mapping programs onto the new multiprocessors.

▶ We can extend FORTRAN or some similar language by adding explicit features for describing concurrency and controlling interactions. Concurrency here means any form of simultaneous program activity.

▶ We can define a new language having its features designed around the concept of providing concurrency in clear and "safe" ways.

David Kuck and his colleagues at the University of Illinois seem to favor[1] the first option. From an organizational and political standpoint this option is quite attractive because of the current investment in software written in FORTRAN. The second option removes some of the burden for finding concurrency from the compiler and gives it to the programmers to manage. Unfortunately, almost every proposed extension would also allow programmers to write programs that introduce time-dependent errors, which are often insidiously difficult to eliminate. Those who do research in "applicative" languages favor the third option. Applicative languages express calculations as mathematical functions. In FORTRAN, "N = N + 1" is a well-defined statement, even though it is not meant to be interpreted as a mathematical equation. Restricting programs to contain only mathematical functions enhances our ability to identify and exploit concurrency.

Our basic position can be summarized as follows:

▶ Programs for multiprocessors should clearly express concurrency and at the same time ensure determinate execution, that is, execution that gives the same results every time, even though the order of execution may vary. Compilers should be responsible for identifying concurrency appropriate to a particular architecture.

▶ Continued use of FORTRAN by programmers will limit their capacity to express algorithms that contain large amounts of concurrency. Also, FORTRAN compilers have significant limitations in their ability to identify concurrency.

▶ Applicative languages[2,3] allow programmers to express concurrency without introducing synchronization problems that could otherwise lead to indeterminate behavior. In addition, compilers for these languages have far better information on potential concurrency.

FORTRAN and other memory-based languages fail to assist programmers in expressing concurrency. Kuck has shown that it is possible to analyze programs not originally intended for parallel processors and find concurrency. However, the analysis can only approximate the concurrency available in a basically sequential algorithm.

In contrast, applicative languages suggest a frame of mind for allowing programmers to reason with and ex-

Continued on page 68

James McGraw is in the computing research group at Lawrence Livermore National Laboratory, in Livermore, California.

A debate: Retire FORTRAN?

press concurrency. These languages have a model of execution that emphasizes concurrency and introduces sequencing when it is necessary to satisfy data dependencies. Programmers can write and test programs with confidence that the results are repeatable. And most important, compilers have more precise information on what data dependencies exist in each program. Applicative languages present algorithms at a higher level of abstraction, which allows compilers more flexibility to optimize performance for a variety of computer architectures.

In the discussion that follows, I will lay out my criteria for evaluating the various language options for high-speed computing. Then I will examine the approach advocated by Kuck and highlight the major drawbacks to staying with FORTRAN. Finally, I will examine the work in applicative languages, presenting the primary advantages of this option and detailing areas where continued research is necessary. Part of my discussion refers to a 1982 paper[4] by David Gajski, David Padua, David Kuck and Robert Kuhn. That paper examines, among other topics, research in applicative languages, and it is quite thoughtfully done. The paper does, however, contain several inaccuracies, which I will attempt to identify and correct.

Role of the language and compiler

Simply stated, a language and its associated compiler are the interface between the user of a computer system and the underlying architecture of the machine. Assuming the architecture is based on some form of multiprocessor, what characteristics make for a good language interface? I suggest two key factors:

- the ease with which a user can safely express an algorithm's concurrency
- the ease with which a compiler can exploit the available concurrency on a machine.

Effective use of multiprocessors depends heavily on a user's ability to program algorithms that contain large amounts of concurrency. No amount of program analysis will transform a basically sequential algorithm into a different, parallel one. An inadequate programming language may interfere with a user's ability to express the most appropriate algorithms. As a simple example, recursion provides a mechanism for implementing divide-and-conquer algorithms, many of which are highly parallel and extremely fast. Will programmers take advantage of divide-and-conquer concurrency, if the language they are given does not support recursion? I think not.

Within the goal of expressing concurrency is the issue of determinism, or "program safety." Should a language definition guarantee that a program's behavior is repeatable? Most current language implementations for applications programming try to ensure repeatability, which is fundamental to many of the debugging schemes applied to erroneous programs. It is generally accepted that testing a program with different sets of data is insufficient to convince people that a program is correct. However, indeterminacy adds a new dimension to the problem because rerunning a program with the same data may produce different results. Hence, determinate behavior should be a high priority in a language designed for concurrency.

Once a program has been written, all agree that a language's compiler must fully exploit the types of concurrency available in the architecture. The difficulty of this problem depends heavily on the nature of the target machine. Compiling for a 64-processor system with global shared memory and compiling for a data-flow system with 1000 processors are totally different problems. In a data-flow computer, the machine represents a program as a graph. Nodes represent operations, and arcs represent data. Each piece of data in the program carries with it the addresses of all operations that need it. The execution rule for each node is simply to execute the operation on any processor after all inputs for the operation are available. Communication costs, exploitable levels of concurrency, available forms of synchronization—all of these machine factors contribute to the tradeoffs that a compiler must make.

This article cannot thoroughly treat all compilation strategies, but a few key observations deserve mention. Before a compiler can map concurrency onto a system, it must first identify that concurrency in the program—it cannot exploit concurrency that it cannot find. Also, experience has shown that exploiting concurrency almost always involves making tradeoffs with other objectives, such as minimizing the total number of instructions executed, or minimizing memory usage. In particular, simultaneous program activity appears to make significantly heavier demands for memory than does sequential program activity. Gajski and his colleagues correctly point out[4] that data-flow systems and functional language semantics "deny the programmer direct control of memory allocation." And yet almost all of the transformations that their paper mentions—scalar renaming, scalar expansion and node splitting, for example—introduce extra memory that the programmers cannot see or control. Such loss of control is likely to be present in any optimizing system.

Drawbacks of FORTRAN

Because FORTRAN provides no language support for expressing concur-

A debate: Retire FORTRAN?

rency, the user has few options for representing it in a program. Programmers need more control of concurrency. Employing FORTRAN for concurrent processing puts all of the emphasis on translation; success depends entirely on the ability of the compiler to identify and exploit concurrency. In this regard, Kuck and his colleagues are to be commended for the degree to which they have succeeded.[1] These results notwithstanding, the general approach still suffers from some serious problems.

The major problem is simply detecting parallelism in FORTRAN programs. Two operations can proceed simultaneously if and only if their results do not depend upon each other either directly or indirectly. If the compiler cannot be absolutely certain, it must be conservative and sequence the operations in the order specified in the program. The difficulty in being certain is inherent in FORTRAN and all other languages based on explicit assignment of values to memory cells.

A very simple example illustrates the problem. Consider the following segment of FORTRAN code:

```
A(I) = G(X)
A(J) = H(Y)
```

Can the two function calls G and H proceed simultaneously? You cannot answer this question without looking at the procedure bodies for G and H. These two functions may share many variables through the FORTRAN statement COMMON. If either function call modifies one of the common variables, then the functions almost certainly need to be sequenced.

Even if the two functions do not share any COMMON blocks, problems may still arise through the parameters. Assume the function calls G and H both modify their own input parameters X and Y. That does not look like a problem because X and Y are different names. But do they represent different objects? They could be formal parameters in the current environment, in which case we need to determine whether any calls to this environment bind the same objects to X and Y. If X and Y can be "aliases" for the same object, we have a problem. For that matter, if any combination of X, Y, I, J and A can ever be aliases for the same object, we have a problem. The algorithm for determining all aliases within a given program requires use of "global flow analysis," an extremely expensive calculation.

To date, Kuck's efforts have focused on finding concurrency inside loops. His system understands several very important forms of concurrent loop processing and recognizes programs that can use these forms of concurrency. The two primary forms are fully parallel loops, in which all passes can proceed simultaneously, and wavefront-parallel loops, in which all loop passes may execute simultaneously, as long as each pass remains some fixed set of instructions behind its "predecessor" pass in the original program. One can exploit these forms of concurrency when a given loop has certain structural properties. To enhance possibilities for having these properties, Kuck defines various program transformations that can reduce possible conflicts between aliased names. Although this strategy can work on many programs that have the "right" structure, it cannot resolve the alias problems mentioned above.

Ultimately, then, the effectiveness of FORTRAN analysis comes down to one key question: How well will analysis do on the existing practical programs that users will want to execute on multiprocessors? Kuck has worked hard to collect a very large number of sample programs from various sources. On these programs his approach apparently works quite well. No one can speak for all potential users, but I can share some insight about the nature of computing at one important site, Lawrence Livermore National Laboratory. Here, most programs are written in an "extended" FORTRAN. These programs routinely use array subscripts that access words of memory beyond the declared scope of the array! In this context it is often impossible to determine when two different array accesses refer to the same object because a subscript can be used as a pointer to access words anywhere in memory. Two heavily used extensions at Livermore are absolute memory addressing and in-line assembly language programming. When either of these options is used, almost all hope for successful analysis is lost. These extensions permeate programs at Livermore, and without extensive rewriting, FORTRAN analysis will fail. Any programs that "abuse" FORTRAN, by going beyond array bounds, for example, are likely to suffer a similar fate. This type of difficulty weakens a primary motive for staying with FORTRAN, namely, being able to use existing programs without rewriting.

Applicative languages

If programmers are ever to give up FORTRAN, the successor must offer some overwhelming advantages. No one in the applicative-language or data-flow community is suggesting that *the* replacement language has been found. Twenty-five years of intense effort on FORTRAN cannot be casually brushed aside. Computer scientists have only begun to give significant attention to applicative languages. The five or six years of research has been exploratory and diverse; even data-flow experts are divided on the best ways to proceed. It is, however, fair to ask what the research has accomplished so far, and what basis there is for our belief in what this approach can accomplish in the future.

One criterion for evaluating languages concerns their expressive ability. In this regard, I believe applicative languages make three major contributions. First, they make concurrency the general "rule" of execution, and introduce sequencing only when necessary to satisfy data dependencies. Second, they allow us to exploit easily some powerful language design options that can reduce programming time and program size. Third, they ensure that all interactions between concurrent program components are expressed determinately, or, if some indeterminacy is allowed, that it is confined to very-well-identified modules. Because even erroneous programs are determinate, programmers have the opportunity to find and cor-

A debate: Retire FORTRAN?

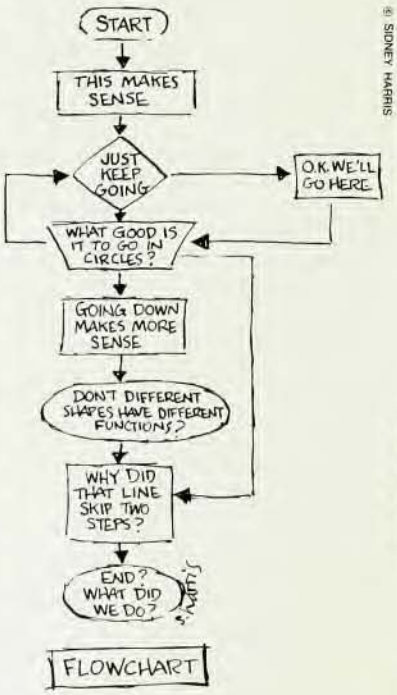
rect their mistakes.

In applicative languages, much of the expressible concurrency derives from one source—simultaneous evaluation of all inputs to a function. In an applicative language *every* executable action is a mathematical function, and many functions simultaneously can be in the process of evaluating their inputs. There are two different ways a language can take advantage of this basic source of concurrency: the data-driven scheme[5] and the demand-driven scheme.[6] In a data-driven scheme, each function "waits" for all of its inputs to be calculated. When all inputs are available, a processor computes the function and sends the results to those functions needing them. In a demand-driven scheme, each function waits for some later function to request its results. When such a request arrives, a processor requests all of the function's inputs, thus initiating more simultaneous work. After the inputs arrive, a processor again computes and passes on the results. Both of these schemes are carried out in a way that does not require processors to wait idly for the function's inputs to arrive. The processors are always available to do useful work elsewhere in the program.

Furthermore, both schemes stress concurrency as the default mode of execution and impose sequencing only when required to do so by dependencies among the data. Unlike FORTRAN, applicative languages have no concept of step-by-step execution. Each function operates autonomously and executes when all inputs have been computed. Therefore, opportunities for concurrency arise naturally at all levels of program structure—from highest-level function calls to independent iterations of a loop to primitive operations such as addition. In the case of sequencing, which occurs only when one function requires the results produced by another function, the data dependency between the two functions will cause the consumer to wait for the producer to finish. But this sequencing is exactly the sequencing imposed by the algorithm.

A second basic source of concurrency allows applicative languages to reduce even the sequencing normally created by data dependencies. Consider a simple dependency situation where a producer function must build an array to be used by a consumer. Although the array is described as one logical unit, we may be able to treat each element of the array as a separate data dependency. The consumer may then begin to use elements of the array before the entire array has been produced. This form of concurrency is called nonstrictness.

In a short article, it is difficult to illustrate why applicative languages allow the expression of such general forms of concurrency. The key factor in languages is not in what programmers can do, but in what they cannot do. Every operation in an applicative language must be expressed as a pure function. The function must identify all necessary inputs and not modify them. The only effect that a function can have is to produce some result value or values. These results must be determinate. This restricted programming environment is the principal "price" extracted from users. Undoubtedly, for many traditional programmers this price would appear to be quite high. However, our experience shows that it is not that difficult to teach this new style. Furthermore, a programmer does not lose the ability to express algorithms by using an applicative language.



In a recent paper,[7] David Turner illustrated the power and "semantic elegance" of applicative languages. He used recursion and set abstraction to enumerate without repetition all possible paraffin molecules, in increasing order of size. Determining when two seemingly different molecules were actually different orientations of the same molecule was a major portion of the problem. Turner concluded that using an applicative language significantly reduced the time needed to produce a correct solution, and by using a few optimizations he arrived at a relatively efficient implementation.

Expressiveness also includes safety. How much should a language try to protect a programmer from time-dependent execution behavior? An example from Gajski's group highlights this issue. Consider these two doubly-nested FORTRAN loops:

```
      DO 11 I = 1,N                    (loop 1)
      DO 11 J = 1,N
11    A(I,J) = A(I - 1,J) + A(I,J - 1)

      DO 22 I = 1,N                    (loop 2)
      DO 22 J = 1,N
22    A(I,J) = A(I - W(K),J)
             + A(I,J - W(K))
```

Loop 1 can be executed using wavefront parallelism, a fact that a compiler can recognize. However, in loop 2 a compiler will be unable to determine whether or not the wavefront parallelism can be exploited:

> Parallel execution is possible in cases like the program in [loop 2], but only through explicit parallelism. The programmer might know that W(K) is always less than some small value and therefore know

A debate: Retire FORTRAN?

that the wavefront algorithm can be applied successfully.[4]

These two program segments are significantly different in that in the latter case wavefront processing is safe only because the "programmer might know" a condition that cannot be verified. If the programmer asserts that wavefront parallelism is safe, he may do so incorrectly. Compilation may produce an indeterminate program. The original programmer may be right, but a later program change might alter the crucial property. What happens then? The view taken in applicative languages is that all parallelism should be expressed in a manner that clearly shows the lack of data dependencies to both programmer and compiler. That is why loop 2 in the above example would not be programmable as wavefront concurrency in any applicative language. Gajski had thought that some data-flow languages did allow this possibility. Hence, designers of applicative languages have made the decision to forego concurrency that only a programmer can see in favor of retaining determinate behavior of all programs, correct or erroneous.

Architectures and translation. The process of translating an applicative-language program for a particular architecture depends heavily on the nature of that architecture. If the architecture of the target machine is organized around a central memory structure, then translation of an applicative-language program will be more difficult because that is not the model of the language. Translation to a data-flow machine, on the other hand, is likely to be much easier because the models match very closely.

The first step in the translation, regardless of target, is to transform the program into a dependence graph. Here, applicative languages have a significant advantage because absolutely no analysis is needed to produce a dependence graph with complete and precise information. The restricted language definition has already required programmers to show all dependencies clearly. A simple, inexpensive algorithm then generates a graph with just the dependencies imposed by the program. Algorithms for analyzing FORTRAN must spend significant time in global flow analysis routines to produce these same types of graphs, but even then the resulting graphs may not show all of the same concurrency. Hence, applicative languages start translation from a better vantage point.

From here, one can apply standard[8] machine-independent optimizations. These optimizations are more likely to produce significant improvements because they are working with complete information. Further optimizations will be needed to reduce the extreme demands that applicative languages currently make on memory space. We expect that analysis similar to that of Kuck's work will allow us to reduce these demands considerably.

For translation to a data-flow machine, it is interesting to note that some problems that normally require compiler analysis do not arise. In the ID language[2] there is no distinction between parallel loops and sequential ones. The model of execution simply states that loops will proceed as fast as possible, limited only by the data dependencies. When translating to a data-flow machine, the compiler never needs to determine which loops are parallel and which are not. Because the machine does its execution using the same data-dependency model as the language, it will handle all loops correctly. Furthermore, if the machine supports nonstrict data structures, then one can similarly exploit wavefront parallelism with no compiler analysis at all. For any multiprocessor target other than a data-flow machine, a compiler would still need to perform the analysis, probably using Kuck's techniques. However, starting from an applicative language will give the system better information with which to work.

* * *

This work was performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory, under contract number W-7405-ENG-48, and through DOE's Office of Basic Energy Sciences, Applied Mathematical Sciences Program, grant AK-01-04. Thanks go to Steve Skedzielewski and Robert Keller for their constructive comments.

References

1. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe in *Proc. 8th ACM Symp. Principles of Programming Languages*, Association for Computing Machinery, New York (1981), page 207.
2. Arvind, K. P. Gostelow, W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Tech. Report TR114a, Dept. of Information and Computer Science, University of California, Irvine (December 1978).
3. J. R. McGraw, ACM Trans. Prog. Lang. Syst. **4**, 44 (1982); W. B. Ackerman, Computer **15**, 15 (1982).
4. D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, Computer **15**, 58 (1982).
5. Arvind, K. P. Gostelow, Computer **15**, 42 (1982).
6. A. L. Davis, R. M. Keller, Computer **15**, 26 (1982).
7. D. A. Turner in *Proc. of the 1981 ACM Conf. on Functional Programming Languages and Computer Architectures*, Association for Computing Machinery, New York (1981), page 85.
8. A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice Hall, Englewood Cliffs, New Jersey (1973). □

Response to Kuck and Wolfe

I see two major differences in approach reflected in our articles. First, we look for concurrency in different places within programs. Second, we make different demands on programmers. Applicative languages stress concurrency at all granularities—primitive operations, loops and function calls. FORTRAN analysis focuses on loops. Applicative languages require programmers to rethink and rewrite their programs in terms of pure mathematical functions. FORTRAN analysis allows programmers to continue to use their old dusty decks and make the best of a difficult situation. In the short-term, FORTRAN analysis can significantly improve performance, especially on vector machines. In the long-term, applicative languages will allow users to exploit far more varied forms of concurrency. However, if you insist on staying with FORTRAN through its slow evolution, I have suggestions for FORTRAN 9x—and they are all along the lines of applicative languages. —JRMcG