Parallel Computation, Functional Programming, and Fortran 8x

REX L. PAGE* AND LINDA S. BARASCH*

Abstract. The American National Standards Institute's committee on Fortran is developing an extension to Fortran 77 that is expected to be ready for public review in 1986. The extended language, which is informally referred to as Fortran 8x, apparently will include structure-valued functions and recursion. These new features will make it possible to design programs in a functional style. Furthermore, the semantics of Fortran 8x, following the precedent of previous standardized versions of Fortran, will require the value of an expression to be independent of the order in which its subexpressions are evaluated. For these reasons, Fortran 8x could provide a relatively painless migration path for scientific codes that need to take advantage of computer systems that implement parallelism via concurrent evaluation of subexpressions. This paper describes some of the new features of Fortran 8x, illustrates their use in expressing programs in a functional style, and analyses the potential parallel performance of such programs under realistic assumptions about currently available hardware.

1. Functional programming. Functional programming has been widely discussed as a basis for software development in parallel processing environments [Kahn74, GrPa77, Back78, GrPa80, BuSi81, GPKL82, KeLi84]. The primary attraction is that nothing inhibits parallel computation in functional programs except a requirement by one subcomputation of a value produced by another. That is, functional programs require sequential computation only at points where inherent data dependencies would impede parallelism regardless of the notation used to describe the algorithms. Compilers can sometimes discover parallelism in conventional, sequential programs by analyzing data dependencies [PaKL80], but functional programs greatly simplify this analysis and avoid the inadvertent introduction of unnecessary dependencies (e.g., through reuse of variables for independent intermediate results). We expect that parallelism designed into the program by the programmer (and functional programming is a tool for expressing such designs) will be more profitable in the long run than parallelism discovered by a compiler. Both have their place: functional programming serves as a tool for describing parallel algorithms; a good compiler can help make up for language deficiencies and can salvage some parallelism from programs.

* Computing Research Division, Amoco Production Company P.O. Box 3385, Tulsa, OK 74102
designed to meet other criteria.

An important opportunity for parallelism in functional programs appears in the evaluation of expressions through concurrent computation of the values of independent subexpressions. When an expression is viewed as a composition of functions (or, equivalently, operations), concurrent evaluation of subexpressions means concurrent evaluation of arguments in function invocations. We call this *operand concurrency*, viewing function arguments and operator operands in the same light. Functional programs also admit *component concurrency*, in which computations delivering individual components of data structures (e.g., array components) can proceed simultaneously, and *stream concurrency*, in which the consumers of data (higher level functions in a composition of functions) can operate concurrently with producers (lower level functions in a compositions). We discuss these modes of concurrency in more detail in Section 3.

An implicit requirement that this parallel evaluation strategy places on a programming language is that the semantics of the language must not impose a sequential discipline on the evaluation of procedure arguments. In particular, programmers must be willing to tolerate the result of an invocation regardless of the order in which its arguments are computed, even if the argument computations are overlapped in an arbitrary fashion. When entire expressions are treated as single expressions, as they are in functional programs, these expressions often contain a great deal of exploitable, coarse-grained parallelism.

To support the functional approach in a reasonable fashion, a programming language must provide, as a minimum, structure-valued functions and recursion. Structure-valued functions make it possible for a single expression to deliver the complete result of a computation, and recursion supplies the ability to describe unbounded computations with a finite number of symbols. Some control over parameter evaluation, such as demand-driven versus eager or sequential versus parallel [Gri81, PaCa81, Bur84], is also useful, but much can be done without it.

A committee of the American National Standards Institute (ANSI) is developing a proposal specifying an extension of Fortran that includes both structure-valued functions and recursion, the two essential elements required to support functional programming. With this extended language, referred to informally as Fortran 8x, there emerges the possibility that functional programming techniques may work their way into the mainstream of scientific computation. We envision this happening in a subversive way: multiprocessor systems may extract their greatest measure of parallelism from programs exhibiting the functional approach, and programmers, upon discovering that such programs perform better than those using the conventional approach, would migrate toward a functional style. This plot is merely conjectural, of course. Furthermore, we envision potential benefits to software development beyond computational performance (the advantages of declarative versus procedural descriptions of algorithms [Smol84]), but these issues are not the subject of this paper. Our purpose is to explain some ways in which Fortran 8x may be useful and to consider some ramifications of mapping Fortran 8x programs onto multiprocessor systems.

2. Fortran 8x. Fortran 8x, as described in the current draft of the ANSI committee on Fortran [X3J88], provides an adequate notation for expressing functional programs. We will use this notation to specify algorithms, and we will analyze the performance of these programs under a hypothetical Fortran 8x compiler that would map them, using the computation strategy outlined in Section 3, onto a multiprocessor system with partitioned memory (e.g., the Cosmic Cube [Seitz83]). The specific interconnection structure of the multiprocessor system and the question of partitioned versus shared memory play only a minor role in our considerations. Thus, our conclusions apply to any multiprocessor organization with sufficient communication bandwidth among processors.

Some of the facilities of Fortran 8x are summarized below. We have not attempted a complete description of the language, but have concentrated on the features that will be used in the algorithm specifications which follow. For example, we have not described the Pascal-like data structure facilities of Fortran 8x but have discussed many of the array facilities because the examples make heavy use of arrays.

Array-valued expressions, assignment. An array may be used as a primary in an expression. When an array is used in a binary operation, the other operand must be either a scalar (which is broadcast across the whole array to produce the result array) or another array of the same 'shape'. (The shape of an array is a vector of integers giving the number of array components in each dimension of the array.) For example, if \( A(10), B(10), C(10) \), then \( C = 2^A(A+B) \) sets \( C(i) = 2^A(A(i)+B(i)) \) for each \( i \) from 1 to 10. An array assignment requires that the destination (left-hand-side) be the same shape as the source (right-hand-side), unless the source is a scalar, in which case it is broadcast to all of the elements of the destination change re-shaping.

Array literals: \([3, 2, 9, 1, 4, 5, 12, 0], [0:n, 2] \), etc. Square brackets are used to delimit constructed vector values. This notation is for one-dimensional arrays only; arrays of higher rank can be constructed by applying reshaping operations to vectors. Elements of the vector are separated by commas. A start/stop/stride notation can be used. Thus \([4, 9, 0:n, 2] \) denotes the vector (i.e., one-dimensional array) whose components are \( 4, 9, 0, 2, 4, \ldots, 2^{(n/2)} \), respectively.

Array intrinsic: generic functions, replicate, sum, etc. Intrinsic functions apply to arrays as well as scalars: \( C = \exp(A) \) sets \( C(i) = \exp(A(i)) \) for each \( i \), when \( A \) and \( C \) are arrays. Mathematical functions like \( \exp \) operate on COMPLEX entities as well as REAL, as in Fortran 77. The intrinsic function \( \text{sum}(x) \) computes the sum of the elements of \( x \). The intrinsic inquiry \( \text{shape}(x) \) delivers the one-dimensional array that specifies the shape of the array \( x \). (For example, under the declaration REAL \( A(0:9, 1:n) \), \( \text{shape}(A) \) is \( [10,n] \)). The intrinsic inquiry size(\( x \)) delivers the total number of array elements of \( x \) (size(A) is \( 10^n \), under the preceding declaration of \( A \)). The intrinsic array constructor replicate(\( x, \text{dim}, \text{copies} \)) builds its array result by appending several copies of the array \( x \) along the dimension specified by \( \text{dim} \).

Array-valued functions, internal functions, recursion. Programmers may define functions that deliver arrays as results. (This is in contrast to Fortran 77, in which the result of a numeric function must be a single
A function with an array result may be so designated by declaring its result parameter to be an array. (The result parameter is either the function-name itself, or another name included in a result specification in the function header.) A procedure (either a function or a subroutine) may have private internal procedures. Such an internal procedure is placed after the last statement in the parent's body, but before the parent's end statement. Procedures may invoke themselves recursively if they are designated as recursive functions in the header statement.

Array sections: $X(a:b,c:;d)$. A rectangular portion of an array can be denoted by specifying a starting component, an ending component and (optionally) a stride in each dimension, separating the start, end, and stride by colons. For example, if $A$ and $B$ are one-dimensional arrays with 10 components each, then $B(3:10) = A(1:8)*2$ puts the squares of the first eight components of $A$ into the last eight components of $B$; the other components of $B$ are not affected. If $X$ is a 5-by-10 two-dimensional array, then $B(2:6) = X(3:4,1:10)$ puts every fourth component in row 3 of $X$ into $B(2), B(4)$ and $B(6)$, respectively. When the stride is omitted, it defaults to 1. When the starting index is omitted, it defaults to the array's lower bound in that dimension, and when the ending index is omitted, it defaults to the upper bound. Thus, $X(:,3)$ denotes the entire third column of $X$, $A(5)$ denotes the first 5 elements of $A$, $X(3,:)$ denotes a 3-by-3 square matrix in the bottom right corner of $X$, and $A(3:2)$ denotes every other element of $A$, starting with the third.

Procedure interfaces: generic, optional, and keyword parameters. Intrinsic operators were generic even in Fortran II (e.g., `+` has always been able to denote addition of either INTEGER or REAL numbers, depending on its operands). Fortran 77 introduced the notion of generic intrinsic functions (e.g., SQRT can operate on REAL, DOUBLE PRECISION, or COMPLEX numbers). Fortran 8x proposes to extend these convenience to functions defined by programmers. It will be permissible to define several procedures of the same name, but with differing types of arguments. Upon encountering a reference to a procedure, the system will select the appropriate procedure, depending on the datatypes of the arguments in its invocation. For example, the name FFT may be used to invoke a Fourier transform of either COMPLEX or REAL data if the programmer has defined two functions named FFT, one with a COMPLEX-valued argument and another with a REAL-valued argument.

Dummy arguments' legacy: A(0), size, precision, etc. A dummy argument inherits many properties from the actual argument. When the dummy argument in a procedure is an assumed-shape array (that is, its declaration has omitted upper bounds in every dimension, as in REAL A(0:0,:)), the procedure inherits its size and shape (but not its rank) from the calling program and may inquire about them via the SIZE and SHAPE functions. (The rank of an array is the number of subscripts and is explicit in the declaration, although it may also be deduced from the length of the shape vector.) Numeric variables, either REAL or COMPLEX, may inherit a precision attribute from the calling program. For example, the argument of a function may be single precision in one invocation and double precision in another; it is expected that the system will automatically invoke different versions of the procedure in these cases, even though the programmer describes the computation in only one procedure.

Dynamic array allocation: REAL Z(n,n), allocate(R(0:size(Z)*2)), etc. Declarations of local arrays in procedures may specify runtime values for dimension bounds. There is also a mechanism for direct, runtime allocation of arrays.

Miscellaneous: block do, repeat, exit, 'comment, and continuation. The block do is a loop construct along the lines of the Block IF of Fortran 77. It may contain an exit statement at any point within the loop. An exclamation point indicates that the remainder of a line is commentary. A line is continued on the following line if its last nonblank character (prior to any commentary) is an ampersand.

As this summary indicates, Fortran 8x incorporates many of the facilities found to be important in modern studies of programming languages. We have not attempted to describe the entire language, but to discuss only those notations that we will use in our presentation of algorithms, and only where they augment the facilities of Fortran 77.

3. Parallelism. Functional programs connote a great deal of potential parallelism because they place few constraints on computational sequence. We will discuss three ways in which a multiprocessor could take advantage of this freedom: operand concurrency, component concurrency, and stream concurrency.

A multiprocessor system can make provisions for concurrently evaluating the arguments of functions (or operands of operators). The performance gains that can be realized through this sort of operand concurrency depend on the amount of parallelism exposed and on the granularity of the parallel tasks (i.e., the amount of computation in the tasks compared to the cost of initiating and terminating them). Assuming no dearth of potentially parallel activity, coarse granularity will lead better utilization of parallel hardware than will fine granularity.

Consider the computation of the definite integral of a function. The number of potentially parallel tasks will be large because computations of the values of the function at the sample points within the interval may proceed in parallel. That is, there are as many parallel tasks available as there are sample points in the interval. These parallel tasks are exposed through operand concurrency in the Fortran 8x program of Figure 1.
recursive function integrate(f,a,b,h)
real integrate
    ! sum f(x): x = a, a+h, ..., b-h
    ! function to be integrated
real a, h, b
    ! endpoints of interval of integration
real hi
    ! stepsizes
if (a+h .gt. b) then
    integrate = 0.0
else
    integrate = f(a) + integrate(f,a+h,b,h)
end if
end recursive function integrate

FIG. 1. Definite Integral

The two operands, f(a) and integrate(f,a+h,b,h), in the primary expression of the function of Figure 1 could be computed as separate, independent tasks. Evaluation of the second operand in the same manner would lead to separate tasks for each f-value to be computed. Separate tasks could be performed on separate processors, leading to simultaneous computation of as many f-values as there are processors available. If each f-value requires a substantial computation (coarse granularity), the cost of initiating a separate task to compute an f-value will be only a small percentage of the cost of computing the value. This will result in nearly linear speed-up compared to sequential computation of the function-values on a single processor (linear speed-up using n processors means that the computation runs n times faster than it would using only one processor).

On the other hand, if the time involved in task initiation is on the same order as the time for task computation, separate tasks on multiple processors could slow the computation down rather than speed it up, which is why a computing system will probably need help from the programmer to decide whether to take advantage of operand concurrency [Bur84]. The potential for coarse granularity is increased if the primary expression splits the interval in half rather than moving along one step at a time.

integrate = integrate(f,a,(a+b)/2,h)+integrate((a+b)/2,b,h),
but in either case the decision concerning operand concurrency must be faced.

We will analyze the trade-off between operand concurrency and sequential computation with respect to an algorithm for computing discrete Fourier transforms via the Runge/Good/Cooley-Tukey formula [CoT65,AhH74]. This algorithm, like the interval-halving version of the integration function discussed above, falls into the divide-and-conquer category. That is, it segments its input data, operates on the segments independently, and puts the segment results together in an appropriate way.

function FFT(c) result(f)
    ! caveat: size(c)=2**ln
complex(actual_prec(c),actual_exp_range(c)) f(0:)
& ! inherited
    ! f(0:2**(size(c)-1)) ! precision
real char(actual_prec(c),actual_exp_range(c)) P
& ! scientific
integer n
& ! notation
n = size(c)
& ! symbol: P
f = FFT(c,exp(cmplx(0,0),(P,-*atan(1.0707)/n)*(2**(n-1))))

recursive function FFT(a,w) result(b)
complex(actual_prec(a),actual_exp_range(a)) a(0:),
& ! input vector
w(0:),
& ! roots of unity
b(0:2**(n-1)) ! result vector
if (size(a) .eq. 1) then
    b = a
else
    b = replicate(FFT(a(0:2),w(0:2)),dim=1,ncopies=2) +
    w*replicate(FFT(a(1:2),w(0:2)),dim=1,ncopies=2)
endif
end recursive function FFT
end function FFT

FIG. 2. Fourier Transform

The Fortran 8x function in Figure 2 specifies the Fourier transform in the radix-2 case using a direct application of a recursive formula for the transform [Ham71] rather than the iterative form of the algorithm developed by Cooley and Tukey. (The general case, in which the length, n, of the input vector can be any number rather than being restricted to a power of two, is similar, but involves a factorization of n.) The primary opportunity for operand parallelism in this program occurs in the else-branch of the internal function FFTs. One operand of the vector addition in this formula is the Fourier transform of the even-numbered components of the input vector; the other operand is the result of multiplying the Fourier transform of the odd components by the roots of unity (component-wise vector multiplication).

A parallel evaluation system using operand parallelism will spawn two subtasks from the task responsible for computing this expression, one for each Fourier transform. When the transform for the odd components is delivered to the spawning task, multiplication by the roots of unity can proceed. When that is completed and when the transform of the even components has been delivered, the vector addition can proceed. Both the vector addition and the vector multiplication will occur within the spawning task (operand concurrency is not needed for the vector multiplication because the roots of unity are available as input and need not be computed).

Consider such a spawning task running on a processor. The processor performing the task sets up the two subtasks, sends the necessary portions of the input vector, the roots of unity, and the task to be performed to a neighboring processor, and then goes on to perform some other task on its list while waiting for results from the subtasks. When the results are delivered
it performs the vector multiplication, then the vector addition, and finally sends its result to the task that requested the transform. Of course, the top level task in this hierarchy omits the last step (sending result), and the bottom level tasks omit the arithmetic steps. Assuming that (1) a processor performing a spawning task can transmit to both subtask processors simultaneously (e.g., via multiple DMA channels), (2) both the sending and the receiving processors are tied up during communication (slightly pessimistic in view of the possibilities for DMA channels), (3) the system switches to on-board operation (no task spawning) after m levels of recursion (the programmer probably would have to plan for this by annotating the program in some way; the idea would be to avoid granularity that is too fine to be profitable), and (4) the system has at least \(2^m\) processors, the following formula gives the expected speedup, \(s\), of a system using operand concurrence to compute the Fourier transform of an n-component vector compared to a system following the Cooley–Tukey iterative form of the algorithm. The number \(\xi\) represents the ratio of the system’s communication performance to its arithmetic performance (i.e., the time to send one number to a neighboring processor divided by the time required for an arithmetic operation). The formula is slightly pessimistic because it assumes that the Cooley–Tukey algorithm involves no overhead at all, only arithmetic operations; in particular, bit-reversed permutation is ignored. On the other hand, the formula is optimistic for values of \(m\) that are near \(\log_2(n)\) because near the maximal depth of recursion, overhead involved in initiating and closing communication messages would effectively increase the value of \(\xi\). Therefore, the formula is realistic only when \(m\) is bounded away from \(\log_2(n)\).

\[
s(n,m,\xi) = \log_2(n)[(3/4)\xi(1-2^{-m}) + (\log_2(n)-m-2)/2^{m} + 2]
\]

where \(n=\) size of input vector
\(m=\) depth of recursion
and \(\xi=\) commun/arith ratio

The efficiency of processor utilization (i.e., the ratio of speedup to the average number of processors involved in the computation over time) is given below as a function of the same parameters.

\[
e(n,m,\xi) = 1/[1 + (9/8)\xi m/\log_2(n)]
\]

These formulas show that the speedup, \(s(n,n(n),\xi)\), increases without bound (but very slowly) as the size of the problem and the number of processors increases \(n(n)\) is any unbounded function bounded away from \(\log_2(n)\) from below). Similarly, the efficiency, \(e(n,n(n),\xi)\), increases to 100 percent if \(n(n)/\log(n)\) converges to zero (\(\xi(n)=\log_2(\log_2(n))\) would satisfy both criteria).

Convergence is so slow, however, that it is of theoretical interest only. For vector sizes of practical interest (e.g., 256 to 4096 components) and for communication/arithmatic ratios consistent with current microprocessor technology (e.g., \(\xi=2\) for a processor capable of half a million floating point

operations per second and with DMA channels operating at a million bytes per second), speedup will be in the range of 2 to 3 at efficiency levels of 60 to 75 percent, utilizing 2 to 4 processors on the average. This may seem disappointing. However, the Fourier transform is an \(\log(n)\) problem, and since reading and writing the data is linear, speedup proportional to \(\log(n)\) is all that is available from any sort of parallelism. (Fox and Newton observed substantially greater speedup for FFTs by ignoring the \(\log\) involved in distributing the input data from a single source to the computational elements and collecting the result data at the conclusion of the computation [Fox&82, Newt&82]. Efficiency remains the same in their scheme as in ours, except that processors are dedicated to a single task in contrast to our model in which processors waiting for partial results switch to other tasks until the awaited results are available.)

In any case, Fourier transforms are an important part of any library intended for scientific computing that they should be carefully coded to match the hardware available. Functional programming techniques can simplify the code substantially (compare, for example, the FFT in Figure 2 to an iterative FFT) without sacrificing potential parallelism. This makes functional techniques ideal for programming at the applications level, even if they are unsuitable at the library level.

![recursive function quadrature](c(a,b,h) result(q))
real integrate
! sum f(x)=a+h,....b-h
real f; elemental f
! function to be integrated
real a,b
! endpoints of interval of integration
real h
! stepsize
real q
! result
integer n; n = int((b-a)/h)
q = h*sum(f(a+h*[0*n-1]))
end function quadrature

FIG.3. Quadrature

Besides operand concurrence, Fortran 8x provides an opportunity for component concurrence. Consider the function specified in Figure 3. The program calls for the application of a function to a mesh of points, each point being designated by a component in an array. A computing system designed to take advantage of operand concurrence would partition the mesh into disjoint sets and assign to computation of the function values on different portions of the partition to different processors. The size of a subset on which it would be profitable to introduce concurrence would vary with the complexity of the function being evaluated and the characteristics of the computing system performing the computation. Annotations in the definition of the function suggesting an appropriate cutoff below which the values should all be computed on a single processor

function f(x)
    elemental f
    ! inboard unless size(x) > c


would help the system take best advantage of its hardware. Effective handling of operand concurrency in complex expressions in which the same component may be needed in more than one partition calls for dataflow analysis [PaKL80] to determine an effective mapping of partitions onto processors. This makes component concurrency more difficult to implement than operand concurrency on multiprocessor computing systems.

A third mode of concurrency available in Fortran 8x programs involves initiating the computation of a function before the computation of its arguments has been completed. Suppose an expression contains a function composition, $f(g(x))$, in which the outer function has an array argument and the inner function delivers an array result (Figure 4). If the inner function produces its array of results in sequence and the outer function carries out its computation incrementally, based on initial segments of its argument array, then it may be profitable to overlap the computation of both functions. Results from the inner function would be streamed to the outer one as they became available, setting up a form of pipeline computation that generalizes the hardware pipelines found on vector processors. Again, some type of annotation in the program indicating that the pipeline could be advantageous would help a multiprocessor system generate an efficient process to carry out the computation. Further annotations to suggest appropriate buffer sizes between consumer and producer processes could further enhance performance [Burk85].

```fortran
function f(x)
  real x ! stream array input
  ...
end function f

function g(x)
  real x ! stream array output
  ...
end function g

FIG. 4. Streaming f(g(x))
```

4. Conclusion. Fortran 8x appears to be well suited to the functional style of programming. It could stand improvement in some areas (such as control over calling sequences — call by need versus call by value, for example, or annotations to suggest profitable parallelism), but all of the necessary functionality is present in the language in some form.

The semantics of Fortran 77 (and Fortran 66 before it) permits concurrent evaluation of operands and function-arguments in expressions, and compilers for computers with multiple arithmetic/logic units have long taken advantage of this type of concurrency. However, without the expressive power of recursion and structure-valued functions, this type of concurrency is limited to a small class of fortuitous applications. The addition of this expressive power to Fortran 8x is what makes it suitable for general purpose applications exploiting parallelism via operand concurrency.

Fortran 8x admits other kinds of parallel computation too. One is the concurrent evaluation of individual array components in array-valued expressions. A third type of parallelism permitted by the semantics of Fortran 8x involves concurrently initiating all functions in a composition of functions within a complex expression, permitting outer invocations to consume results as they are produced by inner invocations. This effects a stream or pipeline type of concurrency. Component-based concurrency will be important in parallel computer architectures that support single instructions that operate concurrently on large conglomerates of data. Operand and stream concurrency will be important for multiprocessors capable of supporting heterogeneous collections of simultaneous tasks.

We have demonstrated the use of a functional coding style in Fortran 8x that makes it profitable to take advantage of operand, component, and pipeline concurrency, and we have suggested a way in which a compiler could adapt such code to appropriate underlying hardware. The problem of adjusting the granularity of the parallelism to a particular computing system can be partially solved through sequential execution of individual Fortran statements and through parameterization of parts of the code to cut off parallel decomposition in an algorithm at an appropriate level.

We plan to develop a Fortran 8x compiler targeting a commercially available multiprocessor with a hypercube interconnection structure [Foo84, Sei85]. This compiler will implement operand concurrency and some aspects of component concurrency. It will be used to test experimentally the efficacy of Fortran 8x and functional programming techniques to express parallel algorithms that use multiprocessor resources in an efficient way.

It appears that Fortran 8x has great potential as a programming language for massively parallel processors. This is a particularly happy circumstance because the new language is upward compatible with Fortran 77, in which so many important scientific applications have been developed. Thus, Fortran 8x provides an attractive, minimally disruptive, migration path from sequential to parallel coding in a field (scientific computing) that has traditionally been the first to use the latest, fastest computing technology.
REFERENCES


