William N. Celmaster

Modern Fortran Revived as the Language of Scientific Parallel Computing

New features of Fortran are changing the way in which scientists are writing and maintaining large analytic codes. Further, a number of these new features make it easier for compilers to generate highly optimized architecture-specific codes. Among the most exciting kinds of architecture-specific optimizations are those having to do with parallelism. This paper describes Fortran 90 and the standardized language extensions for both shared-memory and distributed-memory parallelism. In particular, three case studies are examined, showing how the distributed-memory extensions (High Performance Fortran) are used both for data parallel algorithms and for single-program–multiple-data algorithms.

A Brief History of Fortran

The Fortran (FORmula TRANslating) computer language was the result of a project begun by John Backus at IBM in 1954. The goal of this project was to provide a way for programmers to express mathematical formulas through a formalism that computers could translate into machine instructions. Initially there was a great deal of skepticism about the efficacy of such a scheme. "How," the scientists asked, "would anyone be able to tolerate the inefficiencies that would result from compiled code?" But, as it turned out, the first compilers were surprisingly good, and programmers were able, for the first time, to express mathematics in a high-level computer language.

Fortran has evolved continually over the years in response to the needs of users, particularly in the areas of mathematical expressivity, program maintainability, hardware control (such as I/O), and, of course, code optimizations. In the meantime, other languages such as C and C++ have been designed to better meet the nonmathematical aspects of software design, such as graphical interfaces and complex logical layouts. These languages have caught on and have gradually begun to erode the scientific/engineering Fortran code base.

By the 1980s, pronouncements of the "death of Fortran" prompted language designers to propose extensions to Fortran that would incorporate the best features of other high-level languages and, in addition, provide new levels of mathematical expressivity popular on supercomputers such as the CYBER 205 and the CRAY systems. This language became standardized as Fortran 90 (ISO/IEC 1539: 1991; ANSI X3.198-1992). At the present time, Fortran 95, which includes many of the parallelization features of High Performance Fortran discussed later in this paper, is in the final stages of standardization. It is not yet clear whether the modernization of Fortran can, of itself, stem the C tide. However, I will demonstrate in this paper that modern Fortran is a viable mainstream language for parallelism. It is true that parallelism is not yet part of the scientific programming mainstream. However, it seems likely that, with the scientists' never-ending thirst for affordable performance, parallelism will become much more common—especially now that appropriate standards have evolved. Just as early Fortran enabled average scientists and engineers to program the computers of the 1960s, modern Fortran may enable average scientists and engineers to program parallel computers of the next decade.

An Introduction to Fortran 90

Fortran 90 introduces some important capabilities in mathematical expressivity through a wealth of natural constructs for manipulating arrays.[1] In addition, Fortran 90 incorporates modern control constructs and up-to-date features for data abstraction and data hiding. Some of these constructs, for example, DO WHILE, although not part of FORTRAN 77, are already part of the de facto Fortran standard as provided, for example, with DEC Fortran.

Among the key new features of Fortran 90 are the following:

- Inclusion of all of FORTRAN 77, so users can compile their FORTRAN 77 codes without modification
- Permissibility of free-form source code, so programmers can use long (i.e., meaningful) variable names and are not restricted to begin statements in column 7
- Modern control structures like CASE and DO WHILE, so programmers can take advantage of structured programming constructs
- Extended control of numeric precision, for architecture independence
- Array processing extensions, for more easily expressing array operations and also for expressing independence of element operations
- Pointers, for more flexible control of data placement
- Data structures, for data abstraction
- User-defined types and operators, for data abstraction
- Procedures and modules, to help programmers write reusable code
- Stream character-oriented input/output features
- New intrinsic functions

With these new features, a modern Fortran programmer can not only successfully compile and execute previous standards-compliant Fortran codes but also design better codes with

- Dramatically simplified ways of doing dynamic memory management
- Dynamic memory allocation and deallocation for memory management
- Better modularity and therefore reusability
- Better readability
- Easier program maintenance

Additionally, of course, programmers have the assurance of complete portability between platforms and architectures.

The following code fragment illustrates the simplicity of dynamic memory allocation with Fortran 90. It also includes some of the new syntax for declaring variables, some examples of array manipulations, and an example of how to use the new intrinsic matrix multiplication function. In addition, the exclamation mark, which is used to begin comment statements, is a new Fortran 90 feature that was widely used in the past as an extension to FORTRAN 77.

```
 REAL, DIMENSION(:,:,:),      ! NEW DECLARATION SYNTAX
&  ALLOCATABLE :: GRID        ! DYNAMIC STORAGE
 REAL*8 A(4,4),B(4,4),C(4,4)  ! OLD DECLARATION SYNTAX
 READ *, N                    ! READ IN THE DIMENSION
 ALLOCATE(GRID(N+2,N+2,2))    ! ALLOCATE THE STORAGE
 GRID(:,:,1) = 1.0            ! ASSIGN PART OF ARRAY
 GRID(:,:,2) = 2.0            ! ASSIGN REST OF ARRAY
 A = GRID(1:4,1:4,1)          ! ASSIGNMENT
 B = GRID(2:5,1:4,2)          ! ASSIGNMENT
 C = MATMUL(A,B)              ! MATRIX MULTIPLICATION
```

Some of the new features of Fortran 90 were introduced not only for simplified programming but also to permit better hardware-specific optimizations. For example, in Fortran 90, one can write the array assignment

```
A = B + C
```

which in FORTRAN 77 would be written as

```
       DO 100 J = 1,N
         DO 200 I = 1,M
               A(I,J) = B(I,J) + C(I,J)
200      END DO
100    END DO
```

The Fortran 90 array assignment not only is more elegant but also permits the compiler to easily recognize that the individual element assignments are independent of one another. If the compiler were targeting a vector or parallel computer, it could generate code that exploits the architecture by taking advantage of this independence between iterations.

Of course, the particular DO loop shown above is simple enough that many compilers would recognize the independence of iterations and could therefore perform the architecture-specific optimizations without the aid of Fortran 90's new array constructs. But in general, many of the new features of Fortran 90 help compilers to perform architecture-specific optimizations. More important, these features help programmers express basic numerical algorithms in ways inherently more amenable to optimizations that take advantage of multiple arithmetic units.

A Brief History of Parallel Fortran: PCF and HPF

During the past ten years, two significant efforts have been undertaken to standardize parallel extensions to Fortran. The first of these was under the auspices of the Parallel Computing Forum (PCF) and targeted global-shared-memory architectures. The PCF effort was directed to control parallelism, with little attention to language features for managing data locality. The 1991 PCF standard established an approach to shared-memory extensions of Fortran and also established an interim syntax. These extensions were later somewhat modified and incorporated in the standard extensions now known as ANSI X3H5.

At about the time the ANSI X3H5 standard was adopted, another standardization committee began work on extending Fortran 90 for distributed-memory architectures, with the goal of providing a language suitable for scalable computing. This committee became known as the High Performance Fortran Forum and produced in 1993 the High Performance Fortran (HPF) language specification.[2] The HPF programming-model target was data parallelism, and many data placement directives are provided for the programmer to optimize data locality. In addition, HPF includes ways to specify a more general style of single-program–multiple-data (SPMD) execution in which separate processors can independently work on different parts of the code. This SPMD specification is formalized in such a way as to make the resulting code far more maintainable than previous message-passing-library ways of specifying SPMD distributed parallelism.

Can HPF and PCF extensions be used together in the same Fortran 90 code? Sure. But the PCF specification has lots of "user-beware" warnings about the correct usage of the PARALLEL REGION construct, and the HPF specification has lots of warnings about the correct usage of the EXTRINSIC(HPF_LOCAL) construct. So as you can see, there are times when a programmer had better be very knowledgeable if she or he wants to write a mixed HPF/PCF code. Digital's products support both the PCF and HPF extensions. The HPF extensions are supported as part of the DEC Fortran 90 compiler, and the PCF extensions are supported through Digital's KAP Fortran optimizer.[3,4]

Shared Memory Fortran Parallelism

The traditional discussions of parallel computing focus rather heavily on what is known as *control parallelism.* Namely, the application is analyzed in terms of the opportunities for parallel execution of various threads of control. The canonical example is a DO loop in which the individual iterations operate on independent data. Each iteration could, in principle, be executed simultaneously (provided of course that the hardware allows simultaneous access to instructions and data). Technology has evolved to the point at which compilers are often able to detect these kinds of parallelization opportunities and automatically decompose codes. Even when the compiler is not able to make this analysis, the programmer often is able to do so, perhaps after performing a few algorithmic modifications. It is then relatively easy to provide language constructs that the user can add to the program as parallelization hints to the compiler.

This kind of analysis is all well and good, provided that data can be accessed democratically and quickly by all processors. With modern hardware clocked at about 300 megahertz, this amounts to saying that memory latencies are lower than 100 nanoseconds, and memory bandwidths are greater than 100 megabytes per second. This characterizes today's single and symmetric multiprocessing (SMP) computers such as Digital's AlphaServer 8400 system, which comes with twelve 600-megaflop processors on a backplane with a bandwidth of close to 2 gigabytes per second.

In summary, the beauty of shared-memory parallelism is that the programmer does not need to worry too much about where the data is and can concentrate instead on the easier problem of control parallelism. In the simplest cases, the compiler can automatically decompose the problem without requiring any code modifications. For example, automatic decomposition for SMP systems of a code called, for example, cfd.f, can be done trivially with Digital's KAP optimizer by using the command line

```
kf90 -fkapargs='-conc' cfd.f -o cfd.exe
```

As an example of guided automatic decomposition, the following shows how a KAP parallelization assertion can be included in the code. (Actually, the code segment below is so simple that the compiler can automatically detect the parallelism without the help of the assertion.)

```
C*$*   ASSERT DO (CONCURRENT)
       DO 100 I = 4,N
          A(I) = B(I) + C(I)
       END DO
```

For explicit control of the parallelism, PCF directives can be used. In the example that follows, the KAP preprocessor form of the PCF directives are used to parallelize a loop.

```
C*KAP*PARALLEL REGION
C*KAP*&SHARED(A,B,C) LOCAL(I)
C*KAP*PARALLEL DO
      DO 10 I = 1,N
         A(I) = B(I) + C(I)
10    CONTINUE
C*KAP*END PARALLEL REGION
```

Cluster Fortran Parallelism

High Performance Fortran V1.1 is currently the only language standard for distributed-memory parallel computing. The most significant way in which HPF extends Fortran 90 is through a rich family of data placement directives. There are also library routines and some extensions for control parallelism. HPF is the simplest way of parallelizing data-parallel applications on clusters (also known as "farms") of workstations and servers. Other methods of cluster parallelism, such as message passing, require more bookkeeping and are therefore less easy to express and less easy to maintain. In addition, during the past year, HPF has become widely available and is supported on the platforms of all major vendors.

HPF is often considered to be a *data parallel* language. That is, it facilitates parallelization of array-based algorithms in which the instruction stream can be described as a sequence of array manipulations, each of which is inherently parallel. What is less well known is that HPF also provides a powerful way of expressing the more general SPMD parallelism mentioned earlier. This kind of parallelism, often expressed with message-passing libraries such as MPI,[5] is one in which individual processors can operate simultaneously on independent instruction streams and generally exchange data either by explicitly sharing memory or by exchanging messages. Three case studies follow which illustrate the data parallel and the SPMD styles of programming.

A One-dimensional Finite-difference Algorithm

Consider a simple one-dimensional grid problem—the most mind-bogglingly simple illustration of HPF in action—in which each grid value is updated as a linear combination of its (previous) nearest neighbors.

For each interior grid index *i,* the update algorithm is

$$Y(i) = X(i-1) + X(i+1) - 2 \times X(i)$$

In Fortran 90, the resulting DO loop can be expressed as a single array assignment. How would this be parallelized? The simplest way to imagine parallelization would be to partition the X and Y arrays into equal-size chunks, with one chunk on each processor. Each iteration could proceed simultaneously, and at the chunk boundaries, some communication would occur between processors. The HPF implementation of this idea is simply to add the Fortran 90 code to two data placement statements. One of these declares that the X array should be distributed into chunks, or blocks. The other declares that the Y array should be distributed such that the elements align to the same processors as the corresponding elements of the X array. The resultant code for arrays with 1,000 elements is as follows:

```
!HPF$ DISTRIBUTE X(BLOCK)
!HPF$ ALIGN Y WITH X
      REAL*8 X(1000), Y(1000)

      <initialize x>

      Y(2:999) = X(1:998) + X(3:1000) - 2 * X(2:999)

      <check the answer>
      END
```

The HPF compiler is responsible for generating all of the boundary-element communication code. The compiler is also responsible for determining the most even distribution of arrays. (If, for example, there were 13 processors, some chunks would be bigger than others.)

This simple example is useful not only as an illustration of the power of HPF but also as a way of pointing to one of the hazards of parallel algorithm development. Each of the element-updates involves three floating-point operations—an addition, a subtraction, and a multiplication. So, as an example, on a four-processor system, each processor would operate on 250 elements with 750 floating-point operations. In addition, each processor would be required to communicate one word of data for each of the two chunk boundaries. The time that each of these communications takes is known as the communications latency. Typical transmission control protocol/internet protocol (TCP/IP) network latencies are twenty thousand times (or more) longer than the time it typically takes a high-performance system to perform a floating-point operation. Thus even 750 floating-point operations are negligible compared with the time taken to communicate. In the above example, network parallelism would be a net loss, since the total execution time would be totally swamped by the network latency.

Of course, some communication mechanisms are of lower latency than TCP/IP networks. As an example, Digital's implementation of MEMORY CHANNEL cluster interconnect reduces the latency to less than 1000 floating-point operations (relative to the performance of, say, Digital's AlphaStation 600 5/300 system). For SMP, the latency is even smaller. In both cases, there may be a benefit to parallelism.

A Three-dimensional Red–Black Poisson Equation Solver

The example of a one-dimensional algorithm in the previous section can be easily generalized to a more realistic three-dimensional algorithm for solving the Poisson equation using a relaxation technique commonly known as the red–black method. The grid is partitioned into two colors, following a two-dimensional checkerboard arrangement. Each red grid element is updated based on the values of neighboring black elements. A similar array assignment can be written as in the previous example or, as shown in the partial code segment below, alternatively can use the HPF FORALL construct to express the assignments in a style similar to that for serial DO loops.

```
!HPF$ DISTRIBUTE(*,BLOCK,BLOCK) :: U,V
      <other stuff>
      FORALL (I=2:NX-1,J=2:NY-1:2,K=2:NZ-1:2)
         U(I,J,K) = FACTOR*(HSQ*F(I,J,K) +    &
                U(I-1,J,K) + U(I+1,J,K)   +    &
```

The distribution directive lays out the array so that the first dimension is completely contained within a processor, with the other two dimensions block-distributed across processors in rectangular chunks. The red–black checkerboarding is performed along the second and third dimensions. Note also the Fortran 90 free-form syntax employed here, in which the ampersand is used as an end-of- line continuation statement.

In this example, the parallelism is similar to that of the one-dimensional finite-difference example. However, communication now occurs along the two-dimensional boundaries between blocks. The HPF compiler is responsible for these communications. Digital's Fortran 90 compiler performs several optimizations of those communications. First, it packages up all of the data that must be communicated into long vectors so that the start-up latency is effectively hidden. Second, the compiler creates so-called shadow edges (processor-local copies of nonlocal boundary edges) for the local arrays so as to minimize the effect of buffering of neighbor values. These kinds of optimizations can be extremely tedious to message-passing programmers, and one of the virtues of a high-level language like HPF is that the compiler can take care of the bookkeeping. Also, since the compiler can reliably do buffer-management bookkeeping (for example, ensuring that communication buffers do not overflow), the communications runtime library can be optimized to a far greater extent than one would normally expect from a user-safe message library. Indeed, Digital's HPF communications are performed using a proprietary optimized communications library, Digital's Parallel Software Environment.[6]

Communications and SPMD Programming with HPF

Since HPF can be used to place data, it stands to reason that communication can be forced between processors. The beauty of HPF is that all of this can be done in the context of mathematics rather than in the context of distributed parallel programming. The code fragment in Figure 1 illustrates how this is done.

On two processors, the two columns of the U and V arrays are each on different processors; thus the array assignment causes one of those columns to be moved to the other processor. This kind of an operation begins to provide programmers with explicit ways to control data communication and therefore to more explicitly manage the association of data and operations to processors. Notice that the programmer need not be explicit about the parallelism. In fact, scientists and engineers rarely want to express parallelism. In typical message-passing programs, the messages often express communication of vector and array information.

However, despite the fervent hopes of programmers, there are times when a parallel algorithm can be expressed most simply as a collection of individual instruction streams operating on local data. This SPMD style of programming can be expressed in HPF with the EXTRINSIC(HPF_LOCAL) declaration, as illustrated by continuing the above code segment as shown in Figure 2.

Because the subroutine CFD is declared to be EXTRINSIC(HPF_LOCAL), the HPF compiler executes that routine independently on each processor (or more generally, the execution is done once per *peer* process), operating on routine-local data. As for the array argument, V, which is passed to the CFD routine, each processor operates only on its local slice of that array. In the specific example above on two processors, the first one operates on the first column of V and the second one operates on the second column of V.

It is important to mention here that, although HPF permits—and even encourages—SPMD programming, the more popular method at this time is the message-passing technique embodied in, for example, the PVM[7] and MPI[5] libraries. These libraries can be invoked from Fortran, and can also be used in conjunction with EXTRINSIC(HPF_LOCAL) subroutines.

```
!HPF$ DISTRIBUTE(*,BLOCK) :: U
!HPF$ ALIGN V WITH U
      REAL*8 U(N,2),V(N,2)
      <initialize arrays>
      V(:,1) = U(:,2)       ! MOVE A VECTOR BETWEEN PROCESSORS
```

Figure 1
Code Example Showing Control of Data Communication without Expression of Parallelism

```
                    CALL CFD(V)       ! DO LOCAL WORK ON THE LOCAL PART OF V
                    <finish the main program>

                    EXTRINSIC(HPF_LOCAL) SUBROUTINE CFD(VLOCAL)
                    REAL*8, DIMENSION(:,:)  :: VLOCAL
              !HPF$ DISTRIBUTE *(*,BLOCK)    :: VLOCAL
                    <do arbitrarily complex work with vlocal>
                    END
```

Figure 2
Code Example of Parallel Algorithm Expressed as Collection of Instruction Streams

Clusters of SMP Systems

During these last few years of the second millennium, we are witnessing the emergence of systems that consist of clusters of shared-memory SMP computers. This exciting development is the logical result of the exponential increase in performance of mid-priced ($100K to $1000K) systems.

There are two natural ways of writing parallel Fortran programs for clusters of SMP systems. The easiest way is to use HPF and to target the total number of processors. So, for example, if there were two SMP systems, each with four processors, one would compile the HPF program for eight processors (more generally, for eight peers). If the program contained, for instance, block-distribution directives, the affected arrays would be split up into eight chunks of contiguous array sections.

The second way of writing parallel Fortran programs for clustered SMP systems is to use HPF to target the total number of SMP machines and then to use PCF (or more generally, shared-memory extensions) to achieve parallelism locally on each of the SMP machines. For example, one might write

```
!HPF$ DISTRIBUTE (*,BLOCK) :: V
      <stuff>
      EXTRINSIC(HPF_LOCAL) SUBROUTINE CFD(V)
      <stuff>
C*KAP*PARALLEL REGION
```

If the target system consisted of two SMP systems, each with four processors, and the above program was compiled for two peers, then the V array would be distributed into two chunks of columns—one chunk per SMP system. Then the routine, CFD, would be executed once per SMP system; and the PCF directives would, on each system, cause parallelism on four threads of execution.

It is unclear at this time whether there would ever be a practical reason for using a mix of HPF and PCF extensions. It might be tempting to think that there would be performance advantages associated with the local use of shared-memory parallelism. However, experience has shown that program performance tends to be restricted by the weakest link in the performance chain (an observation that has been enshrined as "Amdahl's Law"). In the case of clustered SMP systems, the weak link would be the inter-SMP communication and not the intra-SMP (shared-memory) communication. This casts some doubt on the worth of local communications optimizations. Experimentation will be necessary.

Whatever else one might say about parallelism, one thing is certain: The future will not be boring.

Summary

Fortran was developed and has continued to evolve as a computer language that is particularly suited to expressing mathematical formulas. Among the recent extensions to Fortran are a variety of constructs for the high-level manipulation of arrays. These constructs are especially amenable to parallel optimization. In addition, there are extensions (PCF) for explicit shared-memory parallelization and also data-parallel extensions (HPF) for cluster parallelism. The Digital Fortran compiler performs many interesting optimizations of codes written using HPF. These HPF codes are able to hide—without sacrificing performance—much of the tedium that otherwise accompanies cluster programming. Today, the most exciting frontier for Fortran is that of SMP clusters and other nonuniform-memory-access (NUMA) systems.

References

1. J. Adams et al., *Fortran 90 Handbook* (New York: McGraw-Hill, Inc., 1992).

2. "High Performance Fortran language specification," *Scientific Programming,* vol. 2: 1–170 (New York: John Wiley and Sons, Inc., 1993), and C. Koelbel et al., *The High Performance Fortran Handbook* (Boston: MIT Press, 1994).

3. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal,* vol. 7, no. 3 (1995): 5–23.

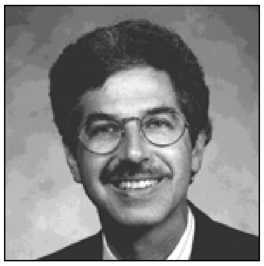
4. R. Kuhn, B. Leasure, and S. Shah, "The KAP Parallelizer for DEC Fortran and DEC C Programs," *Digital Technical Journal,* vol. 6, no. 3 (1994): 57–70.

5. For example see *Proceedings of Supercomputing '93* (IEEE, November 1993): 878–883, and W. Gropp, E. Lusk, and A. Skjellum, *Using MPI* (Boston: MIT Press, 1994).

6. E. Benson et al., "Design of Digital's Parallel Software Environment," *Digital Technical Journal,* vol. 7, no. 3 (1995): 24–38.

7. For example see A. Geist et al., *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Network Parallel Computing* (Boston: MIT Press, 1994).

Biography



William N. Celmaster
Bill Celmaster has long been involved with high-performance computing, both as a scientist and as a computing consultant. Joining Digital from BBN in 1991, Bill managed the porting of major scientific and engineering applications to the DECmpp 12000 system. Now a member of Digital's High Performance Computing Expertise Center, he is responsible for parallel software demonstrations and performance characterization of Digital's high-performance systems. He has published numerous papers on parallel computing methods, as well as on topics in the field of physics. Bill received a bachelor of science degree in mathematics and physics from the University of British Columbia and a Ph.D. in physics from Harvard University.