Modern Fortran as the Port of Entry for Scientific Parallel Computing

Bill Celmaster

Digital Equipment Corporation, 129 Parker Street, Maynard, MA 01754, USA

Abstract

New features of Fortran are changing the way in which scientists are writing, maintaining and parallelizing large analytic codes. Among the most exciting kinds of language developments 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. Several case-studies are examined showing how the distributed-memory extensions (High Performance Fortran) are used both for data parallel and MIMD (multiple instruction multiple data) algorithms.

1 A Brief History of Fortran

Fortran (*FORmula TRANslating*) 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. Fortran has evolved continuously over the years in response to the needs of users. Areas of evolution have addressed mathematical expressivity, program maintainability, hardware control (such as 1/0) and, of course, code optimizations. In the meantime, other languages such as C and C++ have been designed to better meet the non-mathematical aspects of software design. By the 1980's, pronouncements of the 'death of Fortran' prompted language designers to propose extensions to Fortran which incorporated the best features of these other high-level languages and, in addition, provided new levels of mathematical expressivity that had become 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).

Although it isn't clear at this time whether the modernization of Fortran can, of itself, stem the C tide, I will try to demonstrate in this paper that modern Fortran is a viable mainstream language for parallelism. Although parallelism is not yet part of the scientific programming mainstream, it seems likely that parallelism *will* become much more common now that appropriate standards have evolved. Just as early Fortran enabled average scientists and engineers to program computers of the 1960's, modern Fortran may enable average scientists and engineers to program parallel computers by the beginning of the next millenia.

2 An Introduction to Fortran 90

Fortran 90 has added some important capabilities in the area of mathematical expressivity by introducing a wealth of natural constructs for manipulating

arrays. In addition, Fortran 90 has incorporated modern control constructs and up-to-date features for data abstraction and data hiding.

The following code fragment illustrates the simplicity of dynamic memory allocation with Fortran 90. It also illustrates some of the new syntax for declaring data types, some examples of array manipulations, and an example of how to use the new intrinsic matrix multiplication function.

```
      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
```

In general, many of the new features of Fortran 90 help compilers to perform architecture-targetted optimizations. More importantly, these features help programmers express basic numerical algorithms inways (suchas using the intrinsic function MATMUL above), which are inherently more amenable to optimizations that take advantage of multiple arithmetic units.

3 A brief history of parallel Fortran: PCF and HPF

Over the past IO 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 targetted global-shared-memory architectures. The PCF design center was *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. In addition, compiler technologies have evolved to the point that compilers are often able to detect shared-memory parallelization opportunities and automatically decompose codes.

This kind of parallelism is all well and good provided that data can be accessed democratically and quickly by all processors. With modern hardware, this amounts to saying that memory latencies are lower than 100 nanoseconds, and memory bandwidths are greater than 100 MB/s. Those kinds of parameters characterize many of today's shared-memory servers, but have not characterized any massively parallel or network parallel systems. As a result, at about the time of adoption of the ANSI X3H5 standard, another standardization committee began work on extending Fortran 90 for distributed-memor, 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. The HPF design center is *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 MIMD (mul-

tiple instruction multiple data) execution, in which separate processors can independently work on different parts of the code. This MIMD specification is formalized in such a way as to make the resulting code far more maintainable than previous message-library ways of specifying MIMD distributed parallelism.

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 the Digital KAP™ Fortran optimizer.

4 Cluster Fortran parallelism

High Performance Fortran V1.1 is the only language standard, today, 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. Without question, HPF is the simplest way of decreasing turnaround time via parallelism, on clusters (a.k.a. farms) of workstations or servers. Furthermore, HPF has over the past year 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 know is the fact that HPF also provides a very powerful way of expressing the more general parallelism known as Multiple Instruction Multiple Data (MIMD) parallelism. This kind of parallelism is one in which individual processors can operate simultaneously on independent instruction streams, and generally exchange data either by explicitly sharing memory or exchanging messages. Several case-studies follow which illustrate both the data parallel and the MIMD style of programming.

4.1 Finite-difference algorithms

As the most mind-bogglingly simple illustration of HPF in action, consider a simple one-dimensional grid problem 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*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 equally sized 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 take the Fortran 90 code and add to it 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 in a way which aligns elements to the same processors as the corresponding elements of the X array. The resultant code, for arrays with** 1000 **elements, is:**

```
!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 irresponsible for generating all of the boundary-element communication code, and for determining the most even distribution of arrays. Of course, in this example, the time to communicate boundary element data between processors is generally far greater than the time to perform floating point operations. If latency is too large, the problem isn't worth parallelizing.

One example of a low-latency network is Digital's Memory Channel^{TM} cluster interconnect with HPF latencies on the order of 10 microseconds. With such low latency, the above example may be worth parallelizing for, say, 4 processors.

This one-dimension example can be generalized to various two- and three-dimensional examples that typically arise through the solutions of partial differential equations. The distribution directives generalize to allow various kinds of rectangular partitioning of the arrays. Just as in the 1-D case, the compiler takes care of boundary communications. For large problems, the communications times are governed less by the latency, and more by the bandwidth. Digital's Fortran 90 compiler and runtime libraries performs several optimizations of those communications, including so-called message-vectorization and shadow-edge replication.

4.2 Communications *and MIMD 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 the context of distributed parallel programming. The following code fragment illustrates how this is done.

```
!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
```

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. Notice that the programmer needn't be explicit about the parallelism. In fact, scientists and engineers rarely want to express parallelism. If you examine typical message-passing programs, the messages often express communication of vector and array information.

Having said this, it turns out that despite the fervent hopes of program-

mers, there are times when a parallel algorithm can most simply be expressed as a collection of individual instruction streams operating on local data. This MIMD style of programming can be expressed in HPF with the EXTRINSIC (HPF_LOCAL) declaration, as illustrated by continuing the above code segment as follows:

```
      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
```

Because the subroutine CFDis declaredto be EXTRINSIC(HPF_LOCAL), the HPF compiler executes that routine independently oneach 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 2 processors, the first one operates on the first column of V and the second one operates on the second column ofV.

4.3 Clusters of SMP systems

During these last few years of the second millenium, we are witnessing the emergence of systems that consist of clusters of shared-memory computers. This exciting development is a natural evolution of the exponential increase in performance of mid-priced ($1OOK – $1OOOK) systems.

There are two natural ways of writing parallel Fortran programs for such systems. The easiest way is to use HPF, and to target the totaf number of processors. So, for example, if there were two SMP systems each with 4 processors, one would compile the HPF program for 8 processors (more generally, for 8 peers). If the program contained, for instance, block-distribution directives, the affected arrays would be split up into 8 chunks of contiguous array sections.

The second way of writing parallel Fortran programs for clustered-SMP systems is to use HPF to target the totaf 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. This technique will generally be more complex than the method described above, and at this time it is unclear whether it would have any advantages.

References

1. **J. Adams, W. Brainerd, J. Martin, B. Smith and J. Wagener, *Fortran 90 Handbook*, McGraw-Hill, Inc., NY, 1992**