Per Brinch Hansen's Concerns about High Performance Fortran

Carl D. Offner
High Performance Fortran Compiler Development
Digital Equipment Corporation, 129 Parker Street, Maynard, MA 01754
offner@hpc.pko.dec.com

Abstract

Per Brinch Hansen's criticisms of High Performance Fortran in the March issue of this journal are examined and shown to be without foundation.

Keywords: programming languages, parallel programming; data parallelism, High Performance Fortran, message passing

In the March issue of this journal, Per Brinch Hansen devoted an article [1] to an evaluation of High Performance Fortran (HPF). The article is quite negative: Brinch Hansen concludes that HPF is not a suitable language for parallel programming. It is truly unfortunate, however, that every complaint in the article is based on easily avoidable ignorance and misunderstanding of HPF and of the compiler he used. Here I am going to itemize the complaints Brinch Hansen makes in his article, and address each of them.

1.
- *A forall statement cannot call a user procedure. The user is forced to inline procedure calls.*
- *A forall statement cannot have local variables.*

Response: While both these statements are true, neither is an obstacle to what Brinch Hansen wants to do.

The reason both statements are true is that a FORALL does not subsume the most general parallel construct; rather, it is simply a somewhat more general array assignment.

The reason that neither statement is an obstacle to what Brinch Hansen wants to do is that in the case he is considering, it is much more appropriate to use an INDEPENDENT do loop instead of a FORALL construct. Such a loop *can* have local variables. (Such variables are tagged as NEW in the INDEPENDENT directive, and they are local to each loop iteration.) Such loops also can contain procedure calls, making the need for procedure inlining unnecessary.

Specifically, here is how it works: Figure 1 shows Brinch Hansen's original code.

Brinch Hansen then quite correctly adds three lines to distribute the array `a`:

```
integer, parameter :: p = number_of_processors()
!hpf$ processors nodes(p)
!hpf$ distribute a*(*,cyclic) onto nodes
```

and then wants to execute the inner do loop in parallel. It is in fact an independent do loop, and the procedure `transform()` only references its own arguments. So in particular, it is RESIDENT (in HPF terms), and so we can write the code indicated in Figure 2, which does exactly what Brinch Hansen wants. The ON HOME directive tells the compiler that the called routine `transform` only accesses data that is available locally, which is true in this case.

Incidentally, Brinch Hansen did finally wind up with some code that gave him good performance. The much simpler code indicated here would have done equally well.

```
subroutine reduce(a)

  ...

  real*8, intent(inout) :: a(n, n+1)
  real*8 :: vi(n)
  integer :: i, j

  do i = 1, n-1, 1
     call eliminate(i, a(1:n, i), vi)
     do j = i+1, n+1, 1
        call transform(i, a(1:n,j), vi)
     end do
  end do
  return
end subroutine reduce
```

Figure 1: Sequential Householder Reduction; original code

```
subroutine reduce(a)

  ...

  real*8, intent(inout) :: a(n, n+1)
  real*8 :: vi(n)
  integer :: i, j

  integer, parameter :: p = number_of_processors()
  !hpf$ processors nodes(p)
  !hpf$ distribute a*(*,cyclic) onto nodes

  do i = 1, n-1, 1
     call eliminate(i, a(1:n, i), vi)
     !hpf$ independent
     do j = i+1, n+1, 1
        !hpf$ on home (a(:,j)), resident
        call transform(i, a(1:n,j), vi)
     end do
  end do
  return
end subroutine reduce
```

Figure 2: Sequential Householder Reduction: HPF code

2.
- *There is inadequate documentation.*
- *The compiler hides the details of inter-processor communication.*

"It took us two months to make this program run faster in parallel. Our frustrations were caused by the poor documentation of the HPF language and the lack of information about the behavior of running programs, which more often than not reduced our 'design' effort to an endless series of wasted 'experiments' based on guesses about what might possibly work."

"The hidden details of processor communication make performance prediction of HPF programs impractical both in theory and practice."

Response: The two itemized statements are both false. Brinch Hansen's frustrations are due to his failure to read readily available material. For instance, the documentation [3] that comes with the Digital Fortran/HPF compiler (which is the main compiler Brinch Hansen says he used) includes:

- An HPF tutorial, built around some common kinds of coding examples, giving many hints as to how to use HPF effectively to write easily maintainable and efficient code.
- A more detailed chapter, titled *HPF Essentials*, which contains, among many other things, an extensive discussion of different kinds of data mappings.
- A chapter titled *Optimizing HPF Programs*, which contains a lot of information about compiler switches, directives, and programming practices that can be used to good effect. This chapter also contains advice on converting old-style Fortran 77 programs to modern standard Fortran and HPF.
- An installed UNIX man page for the compiler, with a complete listing of all available switches. Among these are some HPF-specific switches (the main one being "`-show hpf_all`") that can be used to have the compiler report what message-passing code it is generating, what lines in the source this code corresponds to, and why the compiler thinks that message-passing is needed at that point. In practice, this gives a lot of information very quickly and easily, and is always the first thing to look at.

In addition, there is a profiler that comes with the HPF Parallel Software Environment, which can be used to give a more quantitative measure of the time actually taken by message-passing code.

None of this material is hard to find. It's the basic documentation and toolkit, and it is delivered in the same manner as with any software product.

Brinch Hansen seems to have relied pretty much on the HPF Handbook [6] for his knowledge of the language. The Handbook was written to document HPF Version 1.0, and has been out of date for over a year. HPF Version 2.0 was published in January, 1997, and is what our compiler supports. The official language report [4] is available on-line.

There are some differences between Version 1 and Version 2 of HPF, and in particular, the subroutine interface details have been greatly simplified in version 2.

3. *The HPF compiler did not support independent do loops.*

Response: The Digital Fortran/HPF compiler has supported INDEPENDENT do loops since Digital Fortran version 4.1, which first shipped in January, 1997. Brinch Hansen was evidently using an out-of-date compiler.

4. *Independent do loops constitute an insecure form of parallelism in which the programmer is completely responsible for the absence of race conditions.*

   Response: This is true, but only in the sense that an INDEPENDENT directive is an assertion that the loop iterations are truly independent of each other. If they are not, the program is non-conforming. If the programmer's assertion is correct, however, the compiler is responsible for handling any necessary synchronization.

   I don't see that this puts any more strain on the programmer than would be there if the program were written in an explicit SPMD style (i.e., with explicit message-passing), which is what Brinch Hansen seems to prefer. In such a style, the programmer has to be continually aware of a multitude of low-level details, in a program that is of necessity much larger.

5. *Conforming implementations of HPF do not have to support the HPF_LOCAL extrinsic subroutine model.*

   Response: Technically this is true, but every implementation does support HPF_LOCAL, and always will, because it is such an important way to escape to a lower-level programming model in certain circumstances. So there is no loss of portability in using this model.

6. *Function calls cannot be used in parallel assignment statments.*

   Response: Actually, they can. Function calls can occur in FORALL constructs provided they are PURE. The problem with the example Brinch Hansen uses is that the function he calls accesses non-local data. (It is not RESIDENT, to use the HPF term.) Therefore, although the construct is legal, the compiler will not generate parallel code for it. This is a case where inlining the function body (which is only 1 line long) is in fact a good thing to do, as it enables the compiler to optimize the communication. The inlining can be encapsulated by a macro, as we will see below; Brinch Hansen also likes to do this in his own code.

The benchmark that Brinch Hansen was trying to code that led to this last complaint was a simple red-black relaxation code. Now actually, one of the first test cases we tried when we were developing our HPF compiler was a red-black relaxation—in 3 dimensions. Our results, which showed optimal scaling, were published in [5] in 1995.

Our code, which runs as well or slightly better than comparable explicit message-passing code, has been available on the net since at least 1995, and is pointed to by our high performance computing web page (`http://www.digital.com/info/hpc/fortran/optimizations.html`), as well as by the paper mentioned above.

The code is much simpler than the message-passing code that Brinch Hansen writes in his companion article on MPI [2], and is also much easer to maintain. To see this, look first at the computational kernel of Brinch Hansen's message-passing code:

```
void relax(int qi, int qj, int up, int down,
           int left, int right, subgrid u)
{ int b, i, j, k;
  for (b = 0; b <= 1; b++)
    { exchange(qi, qj, 1 - b, up, down,
        left, right, u);
      for (i = 1; i <= m; i++)
        { k = (i + b)%2;
          j = 2 - k;
          while (j <= m - k)
            { nextstate(u, i, j);
              j = j + 2
            }
        };
    }
}
```

We want to compare this with the equivalent HPF code. We take the HPF code from our red-black problem (cut down to 2 dimensions, as in the Brinch Hansen example). Aside from some insignificant changes in variable names and bounds, this performs the same computation. First, we define a macro to encapsulate the stencil computation. The body of the FORALL superficially performs the computation in Brinch Hansen's `nextstate()` function. As explained below, however, it really performs much more—it also takes care of all the communication in the companion `exchange()` routine.

```
#define rb(i_init, j_init) \
  forall (i = i_init:m-1:2, j = j_init:m-1:2) \
    u(i, j) = factor*(hsq*u(i, j) + \
                        u(i-1, j) + u(i+1, j) + \
                        u(i, j-1) + u(i, j+1))
```

Using this macro, the computational kernel appears as follows:

```
! red-black relaxation: the red points

rb(2,2)
rb(3,3)

! red-black relaxation: the black points

rb(3,2)
rb(2,3)
```

That is all there is to it. In this form, the bounds of the FORALL are clearly exposed, and do not have to be deciphered from the modulus computation `(i + b)%2`. For this and other reasons, I think it is clear that the HPF code is more understandable and cleanly written.

But the difference is even more striking than just cleanliness. In order to set up for his kernel, Brinch Hansen devotes two full pages in his MPI article to explicit message-passing code to manage his shadow edges. In contrast, *all this code is generated automatically by the HPF compiler.* That is, the compiler handles the introduction of shadow edges, and the movement of data into those edges automatically, and does it as well as a programmer could do by hand using explicit message passing. This is a prime example of a case in which the HPF language, together with a good compiler, makes unnecessary the writing of a lot of error-prone low-level code—code that does not directly solve the actual problem—at no cost in efficiency.

And efficiency *is* important—we certainly agree with Brinch Hansen on that point. Figure 3 shows the latest figures on speedup for our 3-dimensional red-black relaxation code, using a $128 \times 128 \times 128$ array:

	number of processors			
	1	2	4	8
time (sec)	251.1	115.7	50.6	15.8
speedup	1.0	2.2	5.0	15.9

Figure 3: Speedup for 3-dimensional red-black relaxation on a $128 \times 128 \times 128$ array, implemented in HPF

The important thing, of course, is that the speedup is at least linear. Superlinear speedup, as we see here, is due to cache effects—with more processors, there is more available cache. And by the time we get to 8 processors, evidently a large chunk of the problem fits completely in cache, which explains the remarkably large speedup for that configuration.

This, I believe, is a fair response to all of Brinch Hansen's concerns about HPF. Using what I have written here, he should be able to get off to a much faster start the next time he writes a program in HPF. And finally, everything I have written about here is publicly available, and has been for a very long time.

References

[1] Per Brinch Hansen. An evaluation of High Performance Fortran. *ACM Sigplan Notices*, 33(3):57–64, March 1998.

[2] Per Brinch Hansen. An evaluation of the Message-Passing Interface. *ACM Sigplan Notices*, 33(3):65–72, March 1998.

[3] Digital Equipment Corporation. *Digital High Performance Fortran 90 HPF and PSE Manual.* Maynard, Massachusetts, 1998. (First edition was published in 1995).

[4] High Performance Fortran Forum. High Performance Fortran language specification, Version 2.0, January 1997. Available from the Center for Research on Parallel Computation, Rice University, Houston, TX, and on-line as `ftp://softlib.rice.edu/pub/HPF/hpf_v20.ps.gz`

[5] Jonathan Harris, John Bircsak, M. Regina Bolduc, Jill Ann Diewald, Israel Gale, Neil W. Johnson, Shin Lee, C. Alexander Nelson, and Carl D. Offner. Compiling High-Performance Fortran for distributed-memory systems. *Digital Technical Journal*, 7(3):5–23, 1995. Available on the web from `http://www.digital.com/info/DTJJ00`.

[6] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Jr. Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, 1994.