F^{--} : Parallel Extensions to Fortran 90

Robert W. Numrich and *Jon L. Steidel*, Cray Research, Inc., 655 Lone Oak Dr., Eagan MN 55121

A distributed memory parallel computer with a global address space can be programmed very efficiently using a one-sided message passing programming model [6]. To support this programming model using a Fortran-like syntax, we previously proposed a parallel extension to Fortran 77 [5]. The programmer added processor indices to an array reference whenever the array was located in another processor's memory. Processors lived on a grid in the same way that data lived on a grid. The programmer added processor grid coordinates only when needed to point to remote memory. Otherwise all memory references were local.

In this paper we present similar extensions to Fortran 90. Processor coordinates are enclosed in square brackets just as data coordinates are enclosed in parentheses. For example, the statement

```
x(:) = y(:)[remote_pe]
```

copies the array y from the memory of the remote processor to the array x in the local processor. We have extended the syntax to include the Fortran 90 pointer and the Fortran 90 derived type. These extensions allow the programmer to handle dynamically allocated, irregular arrays on different processors and to point to them in a natural way. The problem of irregular data structures is still a research topic for data parallel extensions to Fortran such as High Performance Fortran (HPF) [3] or Vienna Fortran [7].

1 Scalars and Arrays

F^{--} adds a single notational extension to Fortran 90. It borrows square brackets from the C language to enclose global processor information into a coordinate grid in the same way that it uses parentheses to enclose local data information into a coordinate grid. For example, the Fortran 90 declaration

```
real, dimension[npes] :: s
```

tells the compiler to allocate a real scalar s on each processor. The declarations

```
real, dimension(n)  [npes] :: y
real, dimension(n,m)[npes] :: z
```

tell the compiler to allocate a real array y of length n and a two dimensional real array z of size $n \times m$ on each processor. Processor coordinates may be multi-dimensional, for example,

```
real, dimension(n,m)[np,nq] :: z
```

The compiler computes the processor number from a linear rule in the same way that it computes the address of a variable from a linear rule. For example, processor $[r, s]$ in grid $[np, nq]$ is linearized to the logical processor number

$$pe(r, s) = \text{base}(pe) + (s - 1) * np + r - 1 \tag{1}$$

relative to the base processor

$$\text{base}(pe) = pe(1, 1). \tag{2}$$

If the programmer wishes to number processors from a base other than one, the dimension statement can be modified to indicate the lower bound of the processor coordinate. For example, the declaration

```
real, dimension(n,m)[0:np-1,0:nq-1] :: z
```

defines a processor grid with zero as lower bound.

The meaning of F^{--} notation is quite different from that of the HPF [2],[4] or Vienna Fortran [1] extensions to Fortran 90. In HPF, the dimension of a distributed array is its global dimension. The global dimension is distributed to processors according to some distribution rule defined by HPF compiler directives. The dimension of an F^{--} array is the local dimension on each processor. HPF may coerce global data to local data through a directive or across a subroutine boundary. F^{--} coerces local data to global data through a set of processor coordinates. The two points of view are entirely opposite. HPF adopts an implicit model; F^{--} adopts an explicit model.

Processor coordinate syntax is necessary only at those points in the code where explicit communication is taking place. In the absence of processor coordinate syntax, the processor number defaults to the local processor. For example, the executable statement

```
t = s
```

copies the local value of s into the local scalar t. On the other hand, the statement

```
t = s[q]
```

copies the scalar s from processor q into the local scalar t. A statement like this one is an assertion to the compiler to execute the copy. Since the other processors may be modifying data in ways unknown to the local processor, it may not be safe for the compiler to optimize around these assertions. In the same way, the statement

```
x(:) = y(:)[remote_pe]
```

tells the compiler to copy the entire array y from a remote processor to the local array x; the statement

```
x(:) = y(row,:)[remote_pe]
```

copies a row of a matrix; the statement

```
x(index(:)) = y(jndex(:))[remote_pe]
```

gathers data from a remote processor and scatters it locally; the statement

```
x(:) = y[index(:)]
```

gathers data from a list of remote processors. If one processor has been designated master, it may broadcast variables to other processors with the simple statement

```
if(me.eq.master) x[:] = y
```

The programmer is responsible for inserting any necessary synchronization around such statements.

2 Pointers

F^{--} syntax for Fortran 90 pointers adds processor dimensions to the pointer declaration, for example,

```
real, dimension(:)[np,*], pointer :: ptr
```

The processor grid tells the compiler to allocate the pointer itself at the same address in each processor's memory. Data dimensions may be deferred for pointers but processor dimensions may not be deferred. When a pointer is assigned to a target, the pointer retains its own processor dimensions, which may be different from those of the target. If the array y is a target array

```
real, dimension(n)[*], target :: y
```

then the pointer assignment

```
ptr => y
```

creates an alias for the variable y with a two dimensional processor grid rather than a one dimensional grid. The statement

```
x(:) = ptr(:)[p,q]
```

generates a copy of the array y from processor $[p, q]$ in the grid. In other words, the programmer can freely change the processor grid anywhere in the program where it makes sense to do so.

Pointers provide a mechanism for one processor to tell another processor about irregular or dynamically changing data structures. For example, suppose processor $[p, q]$ allocates an array $x(n)$ on its local heap and then assigns a pointer to it with the statements

```
n = some_function(me)
allocate(x(n))
your_ptr => x
```

Another processor may then assign a second pointer to the first pointer on processor $[p, q]$ with the assignment statement

```
my_ptr => your_ptr[p,q]
```

The second pointer now contains information about the remote array x so that with the following statements,

```
allocate(y(size(my_ptr)))
y(:) = my_ptr(:)[p,q]
```

the local processor allocates an array y of the appropriate size and copies the array x from the remote processor.

3 Derived Types

The Fortran 90 derived type is similar to a structure in the C language. For example, the derived type

```
type z
    real, dimension(n) :: a,b
end type z
```

contains two component arrays of length n. To declare a variable of this type on each processor, F^{--} syntax adds a processor grid to the variable declaration, for example,

```
type(z), dimension[np,*] :: y
```

Then the executable statement

```
y%b(:) = y%a(:)[p,q]
```

copies the array component a of variable y from processor $[p, q]$ into local array component b of the local variable y. The processor grid is attached to a variable of the type, not to the type itself nor to its components. Processor grids are illegal inside a derived type definition.

4 Dynamic Memory Allocation

Fortran 90 allows dynamic allocation of memory through allocatable variables. If an allocatable array will be used to communicate between processors, a processor grid must be added to the declaration, for example,

```
real, dimension(:,:)[np,*] allocatable :: a
```

At an allocation statement,

```
allocate(a(n,m))
```

the compiler generates implicit barriers before and after the statement. The programmer is responsible to see that all processors reach the barriers and that each processor asks for the same size array. Deallocation also requires implicit barriers generated by the compiler.

5 Stacks, Heaps and All That

A data declaration statement with square bracket notation signals the compiler that it must know how to locate the variable in the memory of each processor. Static memory allocation is the easiest way to satisfy this requirement. If the base address of the array variable $z(n, m)$ is the same on each processor,

$$\text{base}(z) = \text{loc}(z(1,1)) = \text{constant} \ , \tag{3}$$

the address computation for element $z(i, j)$ in any processor's memory can be linearized in the normal way

$$\text{loc}(z(i,j)) = \text{base}(z) + (j-1) * n + i - 1 \ . \tag{4}$$

Since the base is the same on all processors, knowing the location of the array on one processor is sufficient to know it on all processors. This convention makes global address computation simple and efficient.

Other rules may be used in place of the simple but restrictive one given by equation (3). For example, on a shared memory machine the array $z(n, m)[npes]$ might be allocated to a contiguous block of memory of size $n \times m \times npes$ with each processor owning its own section of size $n \times m$. In that case the base depends on the processor number

$$\text{base}(z, pe) = \text{loc}(z(1,1)[1]) + (pe-1) * n * m \tag{5}$$

On a virtual memory machine, the array might be located on different pages on different processors so that the rule for finding the base address requires a global virtual memory map such that

$$\text{base}(z, pe) = \text{vm\_map}(z, pe) \tag{6}$$

On a cluster of workstations finding the base address of a variable might require a trip through the operating system while on a network of machines it might require an internet address such as a Uniform Resource Locator (URL), for example,

```
my_screen = applet%screen_display["http://www.java.server"]
```

The compiler probably requires a switch that tells it how to interpret the meaning of the processor grid information so that it can generate the correct code at compile time.

6 Summary

Simplicity is the greatest attribute of the F^{--} programming model. Its syntax can be explained in just a few pages of text that is easily understood by any Fortran programmer. We picked the processor grid notation deliberately so that it looks and behaves very much like normal Fortran array syntax. F^{--} is simple not only for a programmer to understand but also for a compiler developer to implement. The compiler always knows whether data is local or remote. No analysis is necessary; either the square brackets are there or they aren't. If not there, the reference is local; if there, computing the processor number is simple and efficient.

References

[1] B. CHAPMAN, P. MEHROTRA, AND H. ZIMA, *Programming in Vienna Fortran*, Scientific Programming, 1 (1992), pp. 31–50.

[2] HIGH PERFORMANCE FORTRAN FORUM, *High Performance Fortran/Journal of Development*, Scientific Programming, 2 (1993).

[3] ——, *HPF-2 scope of activities and motivating applications*, Tech. Rep. CRPC-TR94492, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1994.

[4] C. KOELBEL, D. LOVEMAN, R. SCHREIBER, G. STEELE, AND M. ZOSEL, *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA, 1994.

[5] R. W. NUMRICH, *F^{--}: A Parallel Extension to Cray Fortran.* To be published in *Scientific Programming*, 1996.

[6] A. SAWDEY, M. O'KEEFE, R. BLECK, AND R. W. NUMRICH, *The design, implementation, and performance of a parallel ocean circulation model.* Proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, Reading, England, November 1994. Published by Wold Scientific Publishers.

[7] M. UJALDON, E. L. ZAPATA, B. M. CHAPMAN, AND H. P. ZIMA, *Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation*, Tech. Rep. TR 95-5, Institute for Software Technology and Parallel Systems, University of Vienna, Vienna, Austria, October 1995.