NASA proposal

October 9, 1997

1 Identification and Significance of the Innovation

The informal standards HPF (High Performance Fortran) and MPI (Message Passing Interface) attempted to capture a community consensus view on how parallel programs could or should be written. Unfortunately the SIMD-inspired data parallel model of HPF and the SPMD model embodied in MPI were treated as incompatible extremes. As a result, realistic programs forced into the strict mold of HPF are often clumsy and difficult to understand. Despite the restrictions of the language, efficient compilers for HPF have proven very difficult to construct. The more pragmatic MPI model provides no global view of distributed data structures, and only the most basic abstractions for collective operations on such structures. This makes it hard to build libraries of software to perform common operations on distributed arrays. We propose a translation system for a minimally extended Fortran dialect that incorporates an HPF-like idea of distributed arrays into an *essentially SPMD* program, and facilitates processing of such arrays in an SPMD style. Ordinary message-passing can be freely interleaved with collective data parallel operations and library calls. The extended language can be compiled using a relatively simple source-to-source translator. This translator will be accompanied by a powerful and extensible library of collective operations.

It is hoped that eventually the HPF language definition and the associated compilers will mature to the point where it is the language of choice for scientific application programmers who need the performance of parallel machines. Some four years after definition of the language, it is not clear that we are close to seeing this goal achieved. As an interim measure, at least, it is time to consider other approaches that capture the some of the most important concepts and technologies associated with HPF without relying on very advanced compilers, which have been slow to materialize. In the proposed system we will take software technology that has emerged from HPF implementation efforts, combining these components to create a smaller and more practical programming language which is both more flexible than HPF and avoids many of its implementation problems. The three main inputs will be

- Underpinned by the HPF idea of distributed arrays, a *run-time library* of functions to parametrize these arrays, convert subscripts between global and local frames, and (most importantly) to perform complex collective transformations and communication operations on the arrays. An existing library developed by the *Parallel Compiler Runtime Consortium* project will be adopted.

- A language model, already investigated in the context of C++, which facilitates SPMD manipulation of distributed data.

- An infrastructure of existing software for building Fortran compilers.

The proposed programming model is a flexible hybrid of the data parallel and SPMD approaches. It provides HPF-like distributed arrays as language primitives, and new *distributed control* constructs to facilitate access to the local elements of these arrays. In the SPMD mold, the model allows processors the freedom to independently execute complex procedures on local elements: it is not constricted by SIMD-style array syntax.

All access to *non-local* array elements must go through library functions—typically collective communication operations. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer. On the other hand, by providing distributed arrays as language primitives we are able to relieve the programmer of error-prone tasks such as converting between local and global array subscripts and determining which processor holds a particular element. In the proposed model the programmer can (if desired) avoid dealing explicitly with local array subscripts. As in HPF, programs can be then written at a natural level of abstraction where the meaning is insensitive to the detailed mapping of elements.

In general, adding new syntax to a programming language (as opposed to adding library calls, or adding directives) is considered a final resort. We have been led to the new language model by the following chain of reasoning. SPMD (Single Program, Multiple Data) programming with collective communications is considered to be one of the most promising paradigms to have emerged in parallel computing. But, without a coherent view of distributed arrays, collective library functions need a confusing number of parameters to describe data layout. To avoid this problem distributed arrays must be viewed as self-contained entities or objects (through Distributed Array Descriptors, or DADs) to make library interfaces tractable. In object-oriented language like C++ there is a obvious path towards achieving this goal—implement distributed arrays as container classes in the STL style. But we find that even in C++ we cannot implement access to elements of these containers—especially the distributed DO-loops of the data parallel program—both cleanly *and* efficiently. We are forced to conclude that to fully exploit the potential of this paradigm some sort of language extension—or at least some sort of intelligent preprocessing of the program—is indicated. Once the need for new semantics is accepted, the language options open up. The same constructs that have been explored as class library syntax in C++ can be pulled back into Fortran as simple language extensions. Into the bargain the performance-critical distributed loops—suitably transformed by a specialized translator that can expose the underlying simplicity of the data traversals—are made highly efficient.

1.1 Outline of the extended Fortran dialect

The basic programming model is SPMD programming. HPF-style distributed arrays will be added as new language primitives. Control flow is SPMD in style, but distributed arrays are normally accessed through *global* subscripts. A new control construct will be added to express a distributed parallel loop.

The translator for the extended language *does not* attempt to automatically insert the communication functions needed to fetch non-local data. Instead, a rich library of collective operations on distributed arrays is made directly available to the programmer. These include the collective communication functions that *ought* to be utilized by highly optimizing HPF compilers, but are often under-exploited by the compromise translation systems available today. The library also provides collective arithmetic operations directly analogous to the array intrinsics of Fortran 90 and HPF.

A handful of language extensions provide distributed arrays as a new family of data objects, independent of ordinary Fortran arrays. Needless to say, ordinary Fortran arrays are still available to the programmer, but they are processed sequentially. A distinguishing property of the proposed system, compared to HPF, is that it includes ordinary Fortran as a strict subset, *and ordinary Fortran constructs are unchanged by the translator*. In contrast, the distributed arrays and variables of HPF are syntactically similar to the sequential data in ordinary Fortran. In some contexts—especially inside procedures—it is difficult to tell a priori whether a particular variable is "global" or "local". This can confuse both the programmer and the compiler, and make translation of general HPF programs much more complex than it needs to be. It may lead to drastically inefficient code if the programmer has overlooked the fact that a particular variable was not, in fact, locally resident (or has overlooked that, while the variable *is* in fact locally resident, this is not been made sufficiently evident to the compiler; the compiler may assume the worst and generate code for a non-local fetch anyway!)

Our proposal insists that a segment of ordinary Fortran code is "translated" by directly copying the source code through to the generated code. This strategy is applied universally. Our system will *not* attempt to exploit parallelism even in "explicitly parallel" constructs such as the array syntax of Fortran 90 or the FORALL statement of Fortran 95. This policy drastically simplifies the translator, and gives the programmer much finer control over the generated code.

1.2 Example syntax

The examples in the following sections use certain syntax extensions. The syntax is illustrative, only. An early phase of the project will define a detailed concrete syntax. Where appropriate we should copy the syntax of HPF, because that syntax is well "debugged" and widely understood. The comment-prefix of HPF directives will be omitted—our extensions are bona fide language extensions, not directives.

The distributed data *model* of HPF will be subsumed wholesale. Although our presentation of distributed arrays differs in some obvious ways, we explicitly or implicitly support all the distribution and alignment options of HPF.

Processor arrangements will be declared as in HPF, through a `PROCESSORS` statement. The explicit `TEMPLATE` concept of HPF is abandoned in favour a *distributed range* concept. (We need the idea of a distributed range to parametrize the crucial distributed loop. Supporting *both* templates and distributed ranges is an unnecesary complication.) A distributed range is a generalization of the range type of, for example, Pascal *[check this]*. It represents a range of subscript values, and incorporates a mapping of that range into a dimension of a processor arrangement. The mapping options will be identical to HPF: block, cyclic, block-cyclic, etc. A distributed array declaration will be distinguished from a sequential array declaration. For illustration we will do this by using a different kind of brackets for distributed arrays:

```
PROCESSORS P(4, 4)

RANGE X(N), Y(M)
DISTRIBUTE X ONTO P(BLOCK, *)
DISTRIBUTE Y ONTO P(*, CYCLIC)

REAL A [X, Y]
INTEGER B [Y]
INTEGER C [X, Y, 10]
REAL D [X(1 : N / 2), Y(::2)]
```

Ranges `X` and `Y` are each distributed over a single dimensions of the processor arrangement. The array `A` is an `N` by `M` distributed array, distributed blockwise in its first dimension and cyclically in its second. In general the brackets in a distributed array declaration contain a list of *orthogonal* ranges. `B` is a one dimensional array distributed cyclically in the second dimension of `P` and implicitly replicated over the first. `C` is a three dimensional array with two distributed and one *collapsed* range. The declaration of D illustrates how HPF-like non-trivial alignment relations can be introduced by using subranges in array declarations. `D` is an `N / 2` by `N / 2` distributed array.

Provision of distributed arrays as language primitives provides a very clean, systematic interface to libraries of collective operations. A standard library will parallel (and parallelize) the Fortran 90 transformational intrinsics on ordinary sequential arrays. Examples of standard collective operations are

```
CALL DA_CSHIFT(DST, SRC, DIM, SHIFT)
RES = DA_SUM(SRC)
CALL DA_REMAP(DST, SRC)
```

Here `DST` and `SRC` are distributed arrays. The subroutine `DA_CSHIFT` is closely analogous to the Fortran 90 `CSHIFT` intrinsic. The arguments should be *aligned* arrrays. The function `DA_SUM` sums all elements of the argument and broadcasts the result value. The subroutine `DA_REMAP` takes two distributed arrays of the same shape *which may have any, unrelated mapping* and copies the elements of one to the other.

The syntax for declaration of *distributed data* is complemented by syntax extensions for *distributed control*. The simplest example is the `AT` construct. A distributed array is subscripted with a global subscript. Subscripting expressions are only meaningful on the processors that hold the elements selected. To make sure a statement using a distributed array element is only executed on a processor that holds copies of of the element, a stylized form of conditional called the `AT` construct is provided. The code fragment below illustrates a pair of nested *at* constructs.

```
AT(X(n))
  AT(Y(17))
    A [n, 17] = B [17] + 23 ;
  ENDAT
ENDAT
```

This construct is an abstraction of the *if* statement in a low-level SPMD program that tests whether the local processor contains a desired array element.

A more important and powerful distributed control construct is the `OVERALL` distributed loop. This construct superficially resembles the Fortran 95 `FORALL` construct. It differs from `FORALL` in several respects

- The index ranges are distributed ranges, not triplets.
- There are no restriction on what kind of statements can appear in the body of the construct. Any executable statements are allowed. This reflects the explicit SPMD emphasis of the language, in constrast to the SIMD heritage of HPF.
- Each "iteration" of the construct is localized to a well-defined processor (or group of processors) through the use of distributed ranges for the loop indices. An iteration should only access array elements held locally on a processor executing the iteration. This restriction sounds inconvenient: in practise it can usually be accomodated quite painlessly by limiting access to arrays aligned with the loop ranges.

In this simple example of an `OVERALL` construct is used to initialize the array `A` with some expression involving the global subscripts

```
OVERALL(I = X, J = Y)
  A [I, J] = I + J
ENDOVERALL
```

In a slightly more complex example we add together elements from two arrays

```
OVERALL(I = X, J = Y)
  A [I, J] = B [J] + C [I, J, 17]
ENDOVERALL
```

This operation is legal due to the alignment relation between the `A`, `B` and `C` arrays. As discussed above, all elements on the left hand side of an assignment in an `OVERALL` must be co-resident with the elements on the right hand side. The examples in the following subsections illustrate more general `OVERALL` constructs.

1.3 Example 1: Cholesky decomposition

Figure 1 gives a parallel implementation of Cholesky decomposition in the new language. In the declaration of `A` the first dimension is specified with an integer range rather than a distributed range. This means that this dimension is on-processor—not distributed (*collapsed* in HPF terminology). In general the collective operation `DA_REMAP` copies the elements of one distributed array or section to another of the same shape. In the current example it is used to implement a variant of the the operation called `GATHER-TO-ALL` in MPI. In spite of the fact that we have demanded that the programmer explicitly specify which processor performs every operation, and exactly how communications are to be inserted, we claim that this implementation is at least as simple as any that could be achieved in HPF (and clearer than anything that could be written in MPI).

Other features illustrated by this example are construction of sections of distributed arrays (directly analogous to construction of sections of Fortran 90 arrays) and use of a *subrange* to parametrize an `OVERALL` loop.

1.4 Example 2: Red-Black relaxation

Figure 2 gives a parallel implementation of red-black relaxation in the new language. Following the proposals of HPF 2.0 standard, *ghost regions* are allowed for arrays. In the example the width of these regions is defined by specifying the `SHADOW` attribute (in our language this attribute is specified for ranges rather than arrays). The ghost regions are explicitly brought up to date using the subroutine `DA_WRITE_HALO` from the standard library. The arguments following the array itself define the mode of updating ghost regions at the extremes

```
INTEGER, PARAMETER :: N = 100

PROCESSORS P(NP)
RANGE, DISTRIBUTE ONTO P(CYCLIC) :: X(N)

REAL A [N, X]

REAL B [N]          ! used as a buffer

! initialize the array;

AT(X (1))
  A [1, 1] = SQRT(A [1, 1])
ENDAT

DO K = 1, SIZE - 1
  AT(X (K))
    DO S = K + 1, SIZE
      A [S, K] = A [S, K] / A [K, K]
    ENDDO
  ENDAT

  CALL DA_REMAP(B [K + 1 :], A [K, K + 1 :])

  OVERALL(I = X (K + 1 :))
    DO J = I, SIZE
      A [J, I] = A [J, I] + B [J] * B [I]
    ENDDO
  ENDOVERALL

  AT(X (K + 1))
    A [K + 1, I] = SQRT(A [K + 1, I])
  ENDAT
ENDDO
```

Figure 1: Implementation of Cholesky decomposition in mooted syntax.

```
PROCESSORS P(NP, NP)

RANGE X(N), Y(N)
DISTRIBUTE X ONTO P(BLOCK, *)
DISTRIBUTE Y ONTO P(BLOCK, *)
SHADOW (1) X, Y

REAL A [X, Y]

! initialize the array;

OVERALL(I = X, J = Y)
  A [I, J] = ...
ENDOVERALL

DO ITER = 1, NITER
  DO ICOLOUR = 0, 1
    CALL DA_WRITE_HALO(A, (/CYCL, CYCL/), (/1, 1/))

    OVERALL(I = X)
      OVERALL(J = Y(MOD(ICOLOUR + I, 2) : : 2))
        A [I, J] = (A [I, J - 1] + A [I, J + 1] +     &
   &                A [I - 1, J] + A [I + 1, J]) / 4
      ENDOVERALL
    ENDOVERALL
  ENDDO
ENDDO
```

Figure 2: Implementation red-black relaxation in mooted syntax.

```
...
CALL DA_WRITE_HALO(A, (/CYCL, CYCL/), (/1, 1/))

OVERALL(I = X)
  OVERALL(J = Y(MOD(ICOLOUR + I, 2) : : 2))
    OLD_VAL = A [I, J]
    NEW_VAL = MOD(A [I, J] + FLOOR(RAND() * Q), Q)

    DIFF = 0
    IF(NEW_VAL .EQ. A [I - 1, J]) DIFF = DIFF + CUP
    IF(NEW_VAL .EQ. A [I + 1, J]) DIFF = DIFF + CUP
    IF(NEW_VAL .EQ. A [I, J - 1]) DIFF = DIFF + CUP
    IF(NEW_VAL .EQ. A [I, J + 1]) DIFF = DIFF + CUP
    IF(OLD_VAL .EQ. A [I - 1, J]) DIFF = DIFF - CUP
    IF(OLD_VAL .EQ. A [I + 1, J]) DIFF = DIFF - CUP
    IF(OLD_VAL .EQ. A [I, J - 1]) DIFF = DIFF - CUP
    IF(OLD_VAL .EQ. A [I, J + 1]) DIFF = DIFF - CUP

    IF(RAND() < EXP(DIFF)) A [I, J] = NEW_VAL
  ENDOVERALL
ENDOVERALL
...
```

Figure 3: Simulation 2d Potts model in mooted syntax.

of the array (cyclic wraparound in this case) and the actual width of the halo to be written. Subsequently the `OVERALL` construct can access elements of the array displaced by up to one place from the alignment of the parametric range.

Note the ease with which sites of a particular colour are selected by using nested `OVERALL` constructs, parametrizing the inner one with a subrange. In HPF this could be expressed using array syntax and a `WHERE` construct, but that would be relatively inefficient. Alternatively it could be expressed using nested `INDEPENDENT DO` loops or `FORALL`s, but nesting these constructs makes it difficult for a compiler to analyse them and emit efficient parallel code. In our scheme no special analysis is necessary. A simple efficient scheme is applied universally to translate subscripts in `OVERALL` constructs.

1.5 Example 3: Potts model simulation

Simulation of the 2-D Q-State Potts model by the classic Metropolis heatbath algorithm follows a pattern very similar red-black relaxation. In figure 3 we merely display the innermost loops. This example illustrates the power and utility of allowing arbitrary sequential control flow inside the `OVERALL` loop.

1.6 Other features of the language.

This rapid tour of the proposed language necessarily omits many details. These details are quite well understood—the language model described has been investigated over the last two years in the context of C++ class libraries. We are even investigating similar extensions for the Java language.

2 Related R/R&D

2.1 PCRC

The *Parallel Compiler Runtime Consortium* (PCRC) project, funded by DARPA, produced a body of software to support compilation and exection of HPF-like languages. This body includes the *NPAC runtime kernel.* The run-time kernel is a C++ class library. It is descended from the run-time library of an early research implementation of HPF, with influences from the Fortran 90D run-time and the CHAOS/PARTI libraries. The kernel is currently implemented on top of MPI.

The library consists of several compiler-specific interfaces to a common *run-time kernel.* The interfaces include two different Fortran interfaces (used by different HPF compilers), a user-level C++ interface called ad++[1], and a proposed Java interface. The development of several top-level interfaces has produced a robust kernel interface, on which other language- and compiler- specific interfaces can be constructed straightforwardly.

The largest part of the kernel is concerned with global communication and arithmetic operations on distributed arrays. The communication operations supported include HPF/F90 array intrinsic operations such as `CSHIFT`, the function `pcrc_write_halo`, which updates ghost areas of a distributed array, the function `pcrc_remap`, which is equivalent to a Fortran 90 array assignment between a conforming pair of sections of two arbitrarily distributed HPF arrays, and various gather- and scatter- type operations allowing irregular patterns of data access. Arithmetic operations supported include all F95 array reduction and matrix arithmetic operations, and HPF combining scatter. A complete set of HPF standard library functions is under development.

At the time of writing the kernel is fully functional and quite mature, two of the four interfaces described are complete, and others are in progress.

In the new system proposed here, the kernel runtime software will supply the collective communication operations refered to in the examples. The ad++ interface mentioned above is the prototype for the "language model" we propose. The translator for the extended Fortran language will be adapted from a subset HPF translator also developed in PCRC.

2.2 Global Arrays

Global Arrays[2] is a successful application-oriented library for SPMD programming in Fortran (and other languages). It includes a model of distributed

[1] ad++ is currently implemented as a set of header files defining distributed arrays as type-secure container class templates, function templates for collective array operations, and macros for distributed control constructs.

[2] J. Nieplocha, R. Harrison and R. Littlefield, "Global Arrays: A portable 'shared memory' programming model for distributed memory computers," Supercomputing 1994, pp340-349, IEEE Computer Society Press (1994).

arrays similar to HPF (and therefore similar to our proposal). As in our proposal—unlike in HPF—non-local elements of distributed arrays must be accessed through library calls.

Global Arrays differs from our proposal in two important respects. Firstly it does not assume language extensions—the distributed arrays are represented as integer handles manipulated through library functions[3]. Secondly the model for access to remote data is through simulated shared memory operations rather than collective communication operations. However, there is no fundamental incompatibility between our model and the GAs model. Since the distributed array models have much in common and our proposed language can support multiple communication libraries, it is quite feasible that we could (in a later phase) import GAs-like communication and synchronization into our library, or simply provide a more elegant language-binding to the standard GAs library.

2.3 CHAOS/PARTI

CHAOS[4] is a run-time library of collective communications to support (especially) applications with *irregular* patterns of access to distributed data. The PCRC run-time kernel described above has some support for irregular patterns of array access, but this is not its main emphasis. As with GAs, we see no fundamental problem in making CHAOS callable from our system, although additional interfacing code may be necessary for subscripting HPF-like arrays.

2.4 ZPL

ZPL[5] is a new programming language for scientific computations. Like Fortran 90, it is an array language. It has an idea of performing computations over a *region*, or set of indices. Within a compound statement prefixed by a *region specifier*, aligned elements of arrays distributed over the same region can be accessed. This idea has certain similarities to our `OVERALL` construct.

In ZPL, parallelism and communication are more implicit than in our proposed languge. The connection between ZPL programming and SPMD programming is not explicit. While there are certainly attractions to the more abstract point of view, the language we are proposing deliberately provides lower-level access to the parallel machine.

[3] GAs manages without language extensions because it emphasizes the task-parallel approach rather than data parallel loops.

[4] R. Das, M. Uysal, J.H. Salz and Y.-S. Hwang, "Communication Optimization for Irregular Scientfic Computations on Distributed Memory Architectures," Journal of Parallel and Distributed Computing, 22(3):462-479 (1994).

[5] `http://www.cs.washington.edu/research/projects/zpl/`

2.5 F- -

F- -[6] is an extended Fortran dialect for SPMD programming. The approach is very different to the one proposed here. In F- -, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In F- - the logical model of communication is built into the language—remote memory access with intrinsics for synchronization. In our proposal there are no communication primitives in the language itself. We follow the MPI philosophy of providing communication through separate libraries. While F- - and our approach share an underlying programming model, we believe that our approach offers greater opportunities for exploiting established software technologies.

[6] R.W. Numrich and J.L. Steidel, "F–: A simple parallel extension to Fortran 90," SIAM News, 30 (1997).