HPF+
High Performance Fortran for Advanced Industrial Applications*

Siegfried Benkner

Institute for Software Technology and Parallel Systems
University of Vienna
Liechtenstein Strasse 22, A-1090 Vienna, Austria
E-Mail: sigi@par.univie.ac.at

Abstract. This paper presents HPF+, an optimized version of High Performance Fortran (HPF) for advanced industrial applications. HPF+ provides data distribution features applicable in the context of irregular, dynamically changing data structures and access patterns. Additional mechanisms are introduced to influence the mapping of computations to processors and to reduce the runtime preprocessing and communication overheads by specifying reusable communication schedules and non-local data accesses. Without sacrificing required functionality HPF+ adopts a data distribution model based on a one-level mapping of data to processors, reducing the complexity of the language, while alleviating the compiler's task of producing an efficient parallel program.

1 Introduction

High Performance Fortran (HPF) defines a set of language extensions to Fortran 90/95 (mainly in the form of directives) to facilitate efficient parallel programming on a wide range of parallel architectures. The initial version of HPF, HPF-1 [7], focussed mainly on features for specifying regular data distributions and expressing data parallel execution. Whereas these features provided reasonable support for regular codes, advanced algorithms based on highly irregular data structures could not be handled efficiently. HPF-2 [8], the current version of HPF, defines a set of "approved extensions" which address some requirements of irregular applications by providing features to distribute data in an irregular manner and to influence load balancing. However, the study of advanced industrial codes from applications areas such as weather forecasting, crashworthiness simulation, and computational fluid dynamics and the implementation of representative kernels thereof within the research compilers VFCS [3] and VFC [1] revealed that even with the "approved extensions" certain information that may decisively influence a program's performance cannot be expressed in HPF. This comprises data locality assertions, non-local access specifications, and the possibility of reusing runtime-generated communication schedules.

In this paper we describe HPF+, an improved version of HPF that addresses these requirements. HPF+ has been developed in close cooperation with application designers from industry in the context of the ESPRIT Long Term Research Project "HPF+". Besides introducing features that are crucial for an efficient handling of irregular applications, the data mapping model of HPF has been

* The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission.

simplified without sacrificing required functionality. Despite these changes the deviation from the de-facto standard HPF-2 has been kept minimal.

2 Data Distribution

The data mapping model underlying HPF is a *two-level mapping*. First arrays have to be aligned with a template, and second, the template has to be distributed onto a processor array. The requirement that each array be aligned with a template (either explicitly or implicitly) significantly complicates the semantics of data mapping directives, in particular, when distributed arrays are transferred across procedure boundaries. Additional problems are encountered in the context of allocatable arrays since templates must not be dynamically allocated. Moreover, experience gained from porting large industrial applications [11] to HPF indicates that templates and complex alignments are of very limited use when specifying data distributions. Therefore, without loss of required functionality, HPF+ simplifies the mapping model by eliminating templates.

2.1 Data Distribution Model

The fundamental data distribution model of HPF+ is a *one-level mapping* where arrays are distributed directly to virtual processor arrays or processor array sections. The number of distributed dimensions of an array must be equal to or less than the rank[2] of the corresponding processor array. In the latter case, data is replicated over the remaining processor array dimensions. Alignment may be used to specify groups of arrays that are distributed in the same way. In order to alleviate the distribution of arrays of different rank to the same processors, HPF+ introduces the concept of *processor views* to establish a relationship between processor arrays of different shape.

2.2 Processor Views

Whereas in HPF the user may define multiple processor arrays, the language does not provide any mechanisms for defining an explicit relationship between processor arrays of different shape. Hence programmers are forced to use templates and alignments whenever arrays of different rank[2] should be mapped to the same set of processors. But even with templates and alignments a number of quite simple distributions cannot be specified in HPF as shown in Figure 1. To address these limitations, HPF+ enables the programmer to impose different *views* on the set of available processors by establishing an explicit relationship between processor arrays according to the Fortran *array element order*. This is achieved by an extension of the processors directive that allows constructing processor arrays from sections of previously defined ones. The Fortran intrinsic function `RESHAPE` may be used in this context to specify arbitrary reshapes and to transpose the dimensions of multi-dimensional processor arrays.

2.3 Data Distribution Formats for Irregular Codes

For irregular applications based on unstructured meshes, the basic HPF distributions, `BLOCK(m)` and `CYCLIC(m)`, may introduce unacceptable communication

[2] In HPF the number of distributed dimensions of an array must match the rank of the processor array to which the distribution refers.

```
!HPF$ PROCESSORS R(4)
!HPF+ PROCESSORS R2(2,2) = RESHAPE(R)
      DIMENSION :: A(2,2), B(4)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO R2 :: A
!HPF$ DISTRIBUTE(BLOCK)       OnTO R  :: B
```

Fig. 1. This example shows a simple HPF+ distribution which is not expressible in HPF since the user cannot assume any relationship between processor arrays of different shape. In HPF+ the second processors declaration specifies a relationship between processor arrays `R2` and `R`, and thus the above directives specify the following distributions: $A(1,1)$ and $B(1)$ to processor $R(1)$, $A(2,1)$ and $B(2)$ to processor $R(2)$, $A(1,2)$ and $B(3)$ to processor $R(3)$, and $A(2,2)$ and $B(4)$ to processor $R(4)$.

overheads or an inadequate balance of the workload across the processors of the target machine. In addition, the use of multiple grids to model the underlying problem domain usually exhibits different levels of parallelism; inter-grid parallelism if the computation on each grid can be performed independently of other grids, and intra-grid parallelism, which is characterized by the usual loosely-synchronous data parallelism of structured grid codes. In order to exploit both levels of parallelism while providing the opportunity to balance the workload, it is necessary to distribute the grids to suitably sized subsets of processors. To address the requirements of such applications HPF+ extends the basic data distribution mechanisms with *distribution to processor subsets*, *general block distributions*, *multi-block distributions*, and *indirect distributions*. Figure 2 shows examples of general block, indirect, and multi-block distribution formats.

2.3.1 Distribution to Subsets of Processors In HPF, distribution directives must refer to whole processor arrays. Limited forms of distributions to subsets of processors can be achieved in HPF only by using templates and alignments. However, this is a rather inflexible method, which requires to know the shapes of the data arrays, templates and processor arrays, and thus it cannot be applied in the context of dynamically allocated data objects. In HPF+, distributions to subsets of processors may be specified either by using a processor array that has been constructed from a section of another processor array, or, by directly specifying a processor array section as the target of the distribution.

2.3.2 General Block Distributions For many applications it is crucial to partition an array into contiguous blocks of different size due to load balancing requirements. This is supported in HPF+ by the *general block distribution*, `GEN_BLOCK`, which requires a one-dimensional integer array (expression) as an argument to define the size of each individual block. The size of this argument must be equal to number of processors in the corresponding processor array (dimension), and thus exactly one block is mapped to each processor. General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of a simple unstructured mesh are partitioned prior to execution and then appropriately renumbered, the resulting distribution can be described in this manner.

```
!HPF$ PROCESSORS R(NR_OF_PROCS)

REAL, DIMENSION(:)              :: A, B, C

INTEGER, DIMENSION(NR_OF_PROCS)  :: GBS   ! block-sizes of GEN_BLOCK
INTEGER, DIMENSION(SIZE(B))      :: IMAP  ! mapping array of INDIRECT
INTEGER, DIMENSION(NR_OF_BLOCKS) :: MBS   ! block-sizes of MULTI_BLOCK
INTEGER, DIMENSION(NR_OF_BLOCKS) :: MMAP  ! mapping array of MULTI_BLOCK

!HPF$ DISTRIBUTE(GEN_BLOCK(GBS))        :: A
!HPF$ DISTRIBUTE(INDIRECT(IMAP))        :: B
!HPF+ DISTRIBUTE(MULTI_BLOCK(MBS,MMAP)) :: C
```

Fig. 2. HPF+ distribution formats for irregular codes.

2.3.3 Indirect Distributions Indirect distributions offer the ability to express totally unstructured or irregular distributions that do not involve replication. Such distributions may be defined by means of the `INDIRECT` distribution format with a mapping array as an argument. The *mapping array* must be an integer array (expression) that is conformable with the array (dimension) to be distributed and specifies for each array element the processor to which it is mapped. A mapping array itself may be distributed in the usual way. Note that the flexibility offered by indirect distributions requires a different compilation strategy compared to regular distributions and sophisticated runtime analysis and optimization techniques in order to achieve reasonable efficiency.

2.3.4 Multi-Block Distributions Multi-block distributions are a combination of general block an indirect distributions. These distributions offer the flexibility of indirect distributions while exploiting the structural regularity of general block distributions. Opposed to the general block distribution, which maps contiguous blocks of an array (dimension) based on a fixed one-to-one relationship to the processors, multi-block distributions allow to assign contiguous blocks according to a user-defined, many-to-one relationship to the processors, and thus multiple blocks may be mapped to each processor. The `MULTI_BLOCK` distribution format requires two one-dimensional integer arrays (or array expressions) as arguments; the first argument specifies the size of each block, and the second argument specifies for each block the processor (index) to which it is mapped.

2.4 Dynamic Data Distribution

In order to reflect the requirements of adaptive grid codes to change the distribution of data according to different phases of the algorithm, the concept of dynamic data distribution is provided. Opposed to statically distributed arrays which have a fixed distribution within the scope of their declaration, the distribution of arrays with the `DYNAMIC` attribute may be changed at runtime, either by executing a `REDISTRIBUTE` statement or as the result of a procedure call. In order to define or change the distribution of an array argument inside a procedure and to make the new distribution visible in the calling routine after

```
!HPF$ DISTRIBUTE *(BLOCK) ONTO *R :: X   ! descriptive mapping in HPF
!HPF+ INHERIT,RANGE(BLOCK) ONTO R :: X   ! descriptive mapping in HPF+
```

Fig. 3. Descriptive Mappings – HPF versus HPF+. A descriptive mapping in HPF constitutes an assertion that the distribution of the actual argument matches the distribution specified for the dummy argument. Thus no redistribution will be required. In HPF+ this can be specified by means of the `INHERIT` attribute together with the `RANGE` attribute specifying a single distribution.

return from the procedure, both the actual and formal argument must have the `DYNAMIC` attribute.

While dynamic data distribution is a very flexible feature, its unrestricted use may have a severe impact on the performance of the generated parallel program. For example, certain important optimizations may not be performed if multiple distributions may *reach* a particular array reference. Since interprocedural reaching distribution analysis [10, 12] and similar techniques are restricted, language support is needed, in order to provide the compiler with information about the set of different distributions a dynamically distributed array may be associated with at runtime. For this purpose the `RANGE` attribute can be used. Unlike the HPF-2 approved extensions, in HPF+ not only the possible distribution formats but also the corresponding processor arrays may be specified within the range attribute. Moreover, the `RANGE` attribute may be specified for dummy arguments with an *inherited distribution* as described in Section 2.5.

2.5 Distribution of Dummy Arrays

HPF provides four different mechanisms for specifying the mapping of dummy arguments; *prescriptive, descriptive*, and *transcriptive* mappings and the `INHERIT` attribute. In particular, transcriptive mappings and the associated problems caused by the inheritance of templates from the caller significantly complicate the semantics of the argument transfer. HPF+ simplifies the semantics of procedure calls without sacrificing required functionality; the distribution of a dummy array may be specified either *explicitly*, or, it may be *inherited* from the corresponding actual argument.

An explicit distribution of a dummy array is specified in the same way as for non-formal objects and is always enforced on entry to the procedure. Consequently, if the actual argument does not have a matching distribution, an implicit redistribution will be performed by the compiler.

An *inherited* distribution is specified by means of the `INHERIT` attribute. In this case a dummy argument adopts the distribution of the corresponding actual argument without the need for a redistribution. The `INHERIT` directive provides a suitable means for writing procedures that are independent of the distribution of the arguments. This flexibility, however, may restrict certain optimizations since, in the worst case, the compiler must parameterize the procedure to work with arbitrary incoming distributions. Thus, by providing the `RANGE` attribute the user may restrict the number of distributions that may be associated with a dummy array that has the `INHERIT` attribute. Note that specifying the `RANGE` attribute with a *single* distribution constitutes an assertion to the compiler, that the actual argument will have this distribution in all incarnations of the procedure. Therefore, this feature can be used to achieve the functionality of *descriptive* and *transcriptive mappings* of HPF without introducing new syntax as shown in Example 3.

3 Parallel Execution and Computation Mapping

The base language of HPF+, Fortran 95, provides several constructs such as array assignments and `FORALL` to specify data parallel operations on arrays. Although these constructs do not define the order in which individual array elements are assigned, they exhibit a *fetch-before-store* semantics. In order to perform important optimizations, the compiler must ensure that the semantics of an array assignment is preserved without enforcing the evaluation of the right hand side prior to defining the left hand side. Similarly, many parallelizing loop transformations require that the iterations may be performed in any order. However, in many cases, and particularly if arrays are accessed indirectly, as in most irregular codes, this information cannot be derived by the compiler. Thus the user may provide this information by means of the `INDEPENDENT` directive which may be specified in HPF+ not only for `DO` loops and `FORALL` statements, but also for array assignment statements.[3]

In order to parallelize loops, forall statements and array assignments efficiently, the iteration space or index space of these constructs has to be distributed to the available processors with the goal of minimizing the communication while attempting to balance the load. This task, referred to as work distribution, decisively influences a program's performance. Usually work distribution is determined by the compiler according to the *owner-computes* rule. If data distributions or access patterns are irregular or dependent on runtime data, work distribution based on compile-time information may result in a poor data locality or bad balance of work load. Therefore, HPF+ enables the user to specify the work distribution. The features provided for an efficient parallelization of irregular loops, forall statements and array assignments are described next.

3.1 `DO` Loops

As in HPF, the `INDEPENDENT` directive allows the programmer to assert that no data object written by one iteration is read or written by another, and thus the iterations of a `DO` loop may be executed in any order. The `NEW` clause may be used to introduce *private variables*. The `REDUCTION` clause and *reduction statements* support the specification of reduction operations without introducing dependences. Unlike HPF, the distribution of loop iterations to processors may be specified either directly or according to the distribution of arrays by appending an `ON` clause or `ON HOME` clause to the `INDEPENDENT` directive.

Moreover, HPF+ provides an extension of `INDEPENDENT` directives to be used for perfectly nested loops without forcing the user to specify a separate directive in front of each loop. This is achieved by including a list of loop iteration variables in the `INDEPENDENT` directive of the outermost loop relieving the programmer from the task of specifying the `NEW` clause for iteration variables of inner loops and alleviating the specification of work distribution.

[3] In HPF the `INDEPENDENT` directive cannot be specified for array assignments.

```
!HPF+ INDEPENDENT(I,J), ON HOME(A(I,J))  ! nested INDEPENDENT loop
      DO I=1,N
         DO J=1,N
      ...
!HPF+ INDEPENDENT, NEW(J), ON HOME(A(I,:))
      DO I=1,N
!HPF+    INDEPENDENT, ON HOME(A(:,J))
         DO J=1,N
      ...
!HPF$ ON HOME(B(Y))
      A(X) = B(Y) + C(Y)                 ! irregular array assignment
```

Fig. 4. In HPF+ a single `INDEPENDENT` directive followed by a list of loop variables may be used for perfectly nested loops. The two loop nests shown in this example are semantically equivalent. The `ON HOME` clauses specify that iteration $[i, j]$ is to be executed on the processor owning array element `A(I,J)`. The `ON HOME` directive specifies that the work distribution for the subsequent array assignment statement should be executed according to the distribution of the array section `B(Y)`. In case `B` and `C` have the same distribution, less communication than by a work distribution based on the owner-computes rule may be required.

3.2 `FORALL` Statement and Construct

An `INDEPENDENT` directive in the context of a `FORALL` statement or construct asserts that no combination of the `FORALL` indices assigns to the same array element repeatedly, or to an array element read by any other combination. In order to partition the index space of `FORALL` statements or constructs, the `ON` or `ON HOME` clause may be used, either in connection with an `INDEPENDENT` directive or stand-alone, to specify the processors responsible for assigning to individual array elements.

3.3 Array Assignments

For array assignment statements, including `WHERE` statements, the `INDEPENDENT` directive may be used to assert that neither are array elements defined several times, nor is any array element both defined and read. The `ON HOME` clause may be used to specify that the work distribution should be derived depending on the ownership of a particular array or array section accessed.

4 Schedule Reuse

Schedules are internal communication objects that have to be constructed by the runtime system when a parallel loop with irregular array accesses is entered. This is done in a preprocessing phase called inspector[13]. Informally, a schedule may be described as the set of non-local elements of a distributed array that are accessed during execution of a parallel loop.

Building a schedule may be very expensive, in some cases dominating the execution time of the program. Often, this cost can be amortized across subsequent executions of a parallel loop (for example, if the parallel loop is enclosed in a sequential time-step loop) or across different loops with similar access patterns.

```
      DO t = 1, max_time_steps
         ...
!HPF+    INDEPENDENT, ON HOME(b(y(i))), REUSE(MODULO(t,10) /= 1)
         DO i = 1, n
            a(x(i)) = ... b(y(i)) ...
         END DO
         ...
      END DO
```

Fig. 5. The `REUSE` Clause. The optional condition associated with the `REUSE` clause implies that the communication schedules required for non-local accesses to `A` and `B` will be (re)computed in iterations $1, 11, 21, ...$, of the outer loop. In all other iterations the communication schedules will be reused. Note that the `REUSE` clause could also be used if the inner loop were located in a procedure called from within the time-step loop.

However, existing analysis techniques[6] are not powerful enough to detect all instances of redundant schedule computations; runtime analysis techniques[13] are similarly restricted. Thus, language support is required for this problem.

4.1 The `REUSE` Clause

The redundancy of a schedule computation in the context of an `INDEPENDENT` loop can be expressed by specifying the `REUSE` clause. This constitutes an assertion to the compiler that the communication schedules of all arrays accessed are invariant for *all* executions of the loop and thus have to be computed only once, when the loop is executed the first time. The invariance of the communication schedules of a loop implies that all distributed arrays referenced within the loop must have the same distribution and all indirection arrays used in subscripts of distributed arrays must have the same contents whenever the loop is executed. Note that this also applies to arrays referenced in the `ON HOME` clause, which must be present if the `REUSE` clause is specified. In order to handle conditional schedule reuse, a condition in the form of a scalar logical expression may be associated with the `REUSE` clause. If a `REUSE` clause contains a condition, then the schedules are reused if and only if the condition evaluates to *true*. This feature can be used to permit the reuse of communication schedules within certain different phases of a program, while forcing recomputation of schedules between these phases, as for example in adaptive problems. Optionally, a reuse-clause may be followed by a list of distributed arrays. In this case the reuse of communication schedules is restricted to the arrays specified in the list.

4.2 Schedule Variables

The concept of *schedule variables* [2] is introduced in order to provide the user with explicit control over application and reuse of schedules. In addition to the functionality provided by the `REUSE` clause, schedule variables may be used for applying a single communication schedule to different arrays or for using the same schedules in different parts of the program.

A schedule variable `S` is declared in the specification part of a program by means of the `SCHEDULE` directive of the form:

```
!HPF+ SCHEDULE :: S
```

```
!HPF+ SCHEDULE :: S

        DO k = max_time_steps
           ...
!HPF+      INDEPENDENT, ON HOME(a(x(i))), GATHER(b,c::S)
           DO i = 1, n
             a(x(i)) = b(x(i)) + c(x(i))
           END DO
           ...
           IF (reconfigure(...)) THEN
              CALL recompute(x)
!HPF+         RESET S
           END IF
           ...
        END DO
```

Fig. 6. Schedule Variables. At the time execution reaches the inner loop first, the schedule variable `S` is undefined. The `GATHER` clause results in the execution of the inspector for the loop and the resulting schedule is assigned to `S`. On subsequent iterations of the outer loop, the schedule bound to `S` is reused as long as the logical function `reconfigure` does not evaluate to *true*. If `reconfigure` evaluates to `.TRUE.`, the `RESET` directive sets the schedule variable `S` to *undefined* and the inspector is executed again in the next iteration of the outer loop.

Initially, at the time the scoping unit of a schedule directive is entered, schedule variables are set to *undefined*.[4] Schedule variables may be associated with communication schedules of arrays in the context of independent directives by means of the `GATHER` or `SCATTER` clause. If a schedule variable occurring in such a context is *undefined*, then a value for the variable will be defined by executing an inspector. Otherwise, if the schedule is already defined, the schedule computation will be suppressed and the communication schedule to which the variable is bound will be applied.

More precisely, whenever a loop with a clause of the form `GATHER(A::S)`, where `A` denotes a distributed array and `S` a schedule variable, is executed, either of the following actions is performed depending on whether `S` is *undefined* or *defined*. If the schedule variable is *undefined*, an inspector for `A` is executed, the hereby determined schedule is associated with `S`, and on each processor all non-local elements of `A` read in the loop are gathered according to `S`. If, on the other hand, the schedule variable is already *defined*, then the associated gather communication is performed as prescribed by `S` without executing an inspector. Similarly, a `SCATTER` clause is utilized to define and/or use schedules for arrays that are *written* in a loop.

By means of the `RESET` directive the user may explicitly *undefine* a schedule variable. Thus, the reuse of schedules may be controlled by the user depending on runtime-conditions. This is particularly important for applications that exhibit different communication patterns in different program phases.

If a schedule variable declared in a procedure has the `SAVE` attribute, its value

[4] This may not be the case if a schedule variable has the `SAVE` attribute.

is retained. Thus, the communication schedules used inside procedures may be reused between subsequent incarnations. This feature, for example, may be used whenever a procedure containing parallel loops with indirect array accesses is called from within a time-step loop located in a different procedure. Another way to use a schedule variable in different program units is to declare it in a module and make it accessible via use-association.

5 Non-Local Access Specifications

A major task of an HPF compiler, when transforming executable constructs that reference distributed arrays, is to determine non-local data accesses and to generate the necessary communication. In order to detect non-local data references, the access patterns, the ownership of the accessed data, and the processors that perform the accesses must be known. If any of this information cannot be derived statically by the compiler, for example if arrays are accessed indirectly, communication generation has to be deferred to the runtime. Despite a potentially large preprocessing overhead, communication generation based on runtime techniques may inhibit a number of crucial optimizations with respect to the management of non-local data and the related index conversion techniques.

HPF+, therefore, allows the user to provide the compiler or the runtime system with high level information about non-local accesses without requiring to deal with the details of message passing. The (super) set of non-local elements of a distributed array that may be accessed during the execution of the program may be specified by the user by defining *shadow regions* or *halo regions*.

5.1 Shadow Regions

A shadow region is a rectilinear, contiguous area around the local section of an array used for storing non-local data accessed during nearest-neighbor or stencil computations. Shadow regions, also referred to as *overlap areas* [5], may be specified by means of the `SHADOW` directive by providing offsets with respect to the lower and upper bound of distributed dimensions. The information provided by shadow areas simplifies the organization of communication as well as the local addressing significantly.

5.2 Halo Regions

A halo region is an irregular, non-contiguous area describing the non-local accesses to a distributed arrays. Whereas shadow regions are usually specified in the context of `BLOCK` or `GEN_BLOCK` distributions, halo regions may be specified for *any* distribution. A halo region for a distributed array is specified by means of the *halo-directive* which has the following syntax:

`HALO` (*halo-spec-list*) `ON` *processor-array* (*halo-dummy-list*)

A *halo-spec* is a one-dimensional integer array expression (usually a function reference) that contains exactly one *halo-dummy variable*[5] which ranges from 1 to the number of processors in the corresponding processor array dimension. By evaluating the *halo-spec* for each processor, the set of non-local indices of

[5] For *halo-dummy* variables the same conventions as for *alignment-dummy* variables hold.

```
!HPF$ PROCESSORS, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
!HPF$ DISTRIBUTE (INDIRECT(MAP))      :: A
!HPF$ HALO (halo_function(I)) ON R(I) :: A
```

Fig. 7. The Halo Directive. The set of non-local indices of `A` that may be accessed during execution of the program is obtained by evaluating the array-valued function `halo_function` on each processor with the corresponding processor index `I`.

the corresponding array that may be accessed during execution of the program is obtained. This information is utilized to organize the allocation of temporary storage for non-local data and to compute the associated communication schedules. Example 7 shows the use of halos.

6 Locality Assertions

HPF+ provides the user with language features to assert that the parallelization of certain language constructs may be accomplished without the need to generate inter-processor communication. This includes the `PUREST` attribute and the `RESIDENT` attribute which are described next.

6.1 Purest Procedures

In Fortran, procedures that are called in the context of data parallel constructs must have the `PURE` attribute, which enforces that they are free of side-effects. However, for a parallelizing compiler the `PURE` attribute is of limited use, since a pure procedure may, for example, read distributed global data objects. Calling a procedure that accesses distributed global data objects from an `INDEPENDENT` loop may result in a serialization of the loop because communication may be required to reference the data.

HPF+ extends the concept of pure procedures by providing *purest procedures*. A purest procedure is a pure procedure with the additional constraint that its invocation does not require communication. More precisely, the invocation of a purest procedure must not result in communication on entry, during execution, and upon exit from the procedure. Whenever a purest procedure is called all arguments are assumed to be already local on the processors executing the call. This information can be exploited by the compiler for an efficient parallelization of `INDEPENDENT` loops that contain procedure calls. To specify that a procedure is purest, the `PUREST` directive, which must occur in the specification part of a program unit, is provided.

6.2 The `RESIDENT` Clause

The `RESIDENT` clause may be used in combination with the `ON HOME` clause to specify that the data accessed in the corresponding loop, forall, or array assignment is local to the executing processors. If the resident-clause includes a list of arrays, then the locality assertion refers only to the specified arrays. This information, the compiler may not be able to derive automatically, is utilized to avoid the generation of unnecessary communication.

7 Conclusion

In this paper we described HPF+, an optimized version of HPF, providing new features for an efficient handling of irregular codes. This includes processor views, multi-block distributions, purest procedures, halo regions, and mechanisms for controlling the reuse of communication schedules. The one-level mapping model of HPF+ simplifies the semantics of data distributions, particularly in the context of procedures, without sacrificing required functionality. A number of HPF+ features, initially proposed in Vienna Fortran [4, 14], have already been adopted by the Approved Extensions of HPF-2. This includes generalized block and indirect distributions, distribution to processor subsets, distribution ranges, and the possibility to redistribute arrays as the result of a procedure call.

HPF+ has been implemented in the Vienna HPF+ Compiler (VFC)[1], a new source-to-source compilation system which transforms HPF+ programs to Fortran 90 message passing programs with calls to an MPI-based runtime library.

References

1. S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. Proceedings HPCN'98, Amsterdam, April 1998.
2. S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-Like Languages. NASA Contractor Report 201740, ICASE, Hampton, VA, September 1997.
3. S. Benkner, et.al. Vienna Fortran Compilation System. Version 1.2. User's Guide, University of Vienna, Institute for Software Technology and Parallel Systems, February 1996.
4. S. Benkner. Vienna Fortran 90 and its Compilation. Ph.D. Thesis. TR 94-8, University of Vienna, Institute for Software Technology and Parallel Systems, 1994.
5. H. M. Gerndt. Updating Distributed Variables in SUPERB. Concurrency: Practice and Experience, Vol.2, Sept. 1990.
6. R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. Proceedings of the 5*th* Workshop on Languages and Compilers for Parallel Computing, New Haven, August 1992.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.1* TR, Rice University, November 10, 1994.
8. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 2.0* TR, Rice University, January 31, 1997.
9. ISO. Fortran 90 Standard, May 1991, ISO/IEC 1539 :1991 (E)
10. M. W. Hall, S. Hirandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Proceedings of Supercomputing (SC92), Minneapolis, November, 1992.
11. J. J. Nucciarone, Y. Ozyoruk, and L. N. Long. New Life in Dusty Decks: Results of Porting a CM Fortran-based Aeroacoustic Model to High Performance Fortran. Proceedings SC97, San Jose, CA, November 1997.
12. D.J. Palermo, E.W. Hodges, and P. Banerjee. Interprocedural array redistribution data-flow analysis. Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing, San Jose, CA, August, 1996.
13. R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Technical Report, UMIACS-TR-93-32, University of Maryland, April 1993.
14. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.