Optimizing Irregular HPF Applications Using Halos

Siegfried Benkner

C&C Research Laboratories
NEC Europe Ltd.
Rathausallee 10, D-53757 St. Augustin, Germany

Abstract. This paper presents language features for High Performance Fortran (HPF) to specify non-local access patterns of distributed arrays, called halos, and to control the communication associated with these non-local accesses. Using these features crucial optimization techniques required for an efficient parallelization of irregular applications may be applied. The information provided by halos is utilized by the compiler and runtime system to optimize the management of distributed arrays and the computation of communication schedules. High-level communication primitives for halos enable the programmer to avoid redundant communication, to reuse communication schedules, and to hide communication overheads by overlapping communication with computation. Performance results of a kernel from a crash simulation code on the NEC Cenju-4, the IBM SP2, and on the NEC SX-4 show that by using the proposed extensions a performance close to hand-coded message-passing codes can be achieved for irregular problems.

1 Introduction

Many technical and scientific applications utilize irregular grids or unstructured meshes in order to model properties of the real world. Implementing these models in a programming language like Fortran results in codes where the main computations are performed within loops based on indirect (vector-subscripted) array accesses. An efficient parallelization of such codes with a high-level data-parallel paradigm like High Performance Fortran (HPF) is a challenging task. Since the access patterns of distributed arrays are hidden within dynamically defined indirection arrays and thus cannot be analyzed at compile time, the major tasks of HPF compilers, which include work distribution, local memory allocation, local address generation, and communication generation have to be deferred to the runtime. Most of the runtime parallelization techniques are based on the inspector/executor strategy [12, 13]. With this approach the compiler generates for each parallelizable loop preprocessing code, called inspector, which at runtime analyzes the access patterns of distributed arrays, and, besides a number of other tasks, derives the required communication schedules for implementing non-local data accesses by means of message-passing. The executor usually comprises a communication phase, based on the derived schedules, followed by a local computation phase executing a transformed version of the original loop.

The inspector phase may be very time-consuming and, in the case of a complex loop, may by far exceed the sequential execution time of the loop. Therefore, such schemes may be applied with reasonable efficiency only in those cases where the information obtained by the inspector is reused over many subsequent executions of a loop as long as the access patterns and distributions of arrays do not change. Since the detection of invariant inspector phases by means of compile-time or runtime analysis techniques [8, 12] is restricted, recently language extensions [2, 4] have been proposed that allow the user to assert that communication schedules associated with irregular loops are invariant, either for the whole program run or within certain computational phases, and thus may be reused.

In this paper we present language extensions for HPF which allow the user to specify non-local access patterns of distributed arrays, called *halos*, and to control the communication associated with these non-local accesses. The provision of halos enables the compiler to apply crucial compile-time and/or runtime optimization techniques required for an efficient parallelization of irregular applications. In particular, halos alleviate the local management of distributed data, the local address generation and the computation and reuse of communication schedules. Moreover, the presented features enable the programmer to avoid redundant communication, to aggregate communication, and to hide communication overheads by overlapping communication with computation.

This paper is structured as follows: In Section 2 we introduce the concept of halos and describe how halos may be specified in an HPF program. In Section 3 we present high-level communication statements for halos. In Section 4 a brief discussion of implementation aspects and performance results on three parallel machines are presented. Finally, Section 5 summarizes the paper and discusses some related work. We assume that the reader is familiar with the basic concepts of HPF [7].

2 Halos

The (super) set of non-local elements of a distributed array that may be accessed during the execution of the program by a particular processor may be specified by the user by defining a *halo*. A halo is an irregular, possibly non-contiguous area describing, based on global indices, the non-local accesses to distributed arrays. A halo for a distributed array `A` is specified by means of the *halo-directive* which has the following form:

`!HPF+ HALO (` *halo-spec-list* `) ON` *halo-proc-ref* `:: A`

A *halo-spec* is a one-dimensional integer array expression that contains exactly one *halo-dummy variable* which ranges from 1 to the number of processors in the corresponding processor array dimension. This *halo-dummy variable* must occur in exactly one dimension of the associated processor array reference *halo-proc-ref*. By evaluating the *halo-spec* for each index of the corresponding processor array dimension, the set of indices of array `A` that may result in non-local data

accesses during execution of the program is obtained. If a dimension of an array is not distributed, then the corresponding *halo-spec* must be a "*".

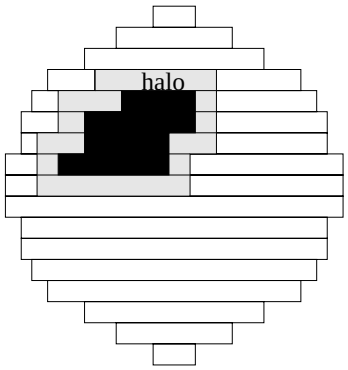
The *halo-directive* is a specification directive and thus all expressions appearing in a *halo-spec* must be *specification expressions*. However, in most irregular applications the distribution of arrays is dependent on runtime information and thus has to be determined dynamically. In this case halos may be specified by means of the `HALO` attribute within the `REDISTRIBUTE` directive. This feature allows recomputing or changing the halo of an array whenever its distribution is changed.

Figure 1, which is an excerpt from an HPF+[4] kernel of the IFS weather prediction code [1], illustrates the use of a halo for a dynamically distributed array. In this example we assume that both the mapping array used as argument to the `INDIRECT` distribution format and the halo array are generated by a partitioner and read in at runtime.

An alternative method to specify halos is based on using array valued functions. With this approach the halo is computed by means of an `EXTRINSIC HPF_LOCAL` function, which returns the halo indices in a one-dimensional integer array. In order to compute the halo of a distributed array by means of a function, information about the distribution of the array is usually required. This can be achieved by passing the array that is being distributed as argument to the halo-function and using the distribution inquiry functions of the HPF Local Library, like for example `LOCAL_LINDEX` and `LOCAL_UINDEX`, which return the lower-bound and upper-bound of the local section an a particular processor (see [7] page 242).

Specifying a halo for a distributed array constitutes an assertion that all non-local accesses are restricted to the halo. Halos do not change the semantics of data distribution directives and the concept of *ownership*. Array elements that are stored in the halo on a particular processor are not *owned* by that processor. Consequently, the semantics of the `ON HOME` clause, which may be used to specify the distribution of the iteration space of irregular loops is not influenced by halos.

Usually HPF compilers extend the local section of a distributed array that has to be allocated on a particular processor in order to store copies of non-local data elements moved in from other processors. This additional storage is referred to as *extension area*. The main advantage of such an approach, compared to a method where non-local data is stored in separate buffers, is that a uniform local addressing scheme for accessing local and non-local data can be utilized. For regular codes the size of the required extension area can usually be determined by the compiler by analyzing the access patterns of distributed arrays. In the case of indirect array accesses, however, the amount of storage required for non-local data cannot be determined statically. Therefore, many compilers determine the size of the extension area based on simple heuristics, for example, as some fraction of the local section. If it turns out at runtime that the extension area is too small, the array has to be reallocated with a larger extension area and the local addressing scheme has to be modified accordingly, which may cause severe overheads.




```
      TYPE AOA
         INTEGER, POINTER, DIMENSION(:) :: IDX
      END TYPE AOA
      TYPE (AOA), ALLOCATABLE :: SLHALO(:)

!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS())

      REAL, DIMENSION(NGP,NFL)         :: ZSL
!HPF+ DYNAMIC, RANGE(INDIRECT,*), HALO :: ZSL
      ...

      ALLOCATE(SLHALO(NUMBER_OF_PROCESSORS()))
      READ(...) MAP                ! mapping array for grid points

      DO I = 1, NUMBER_OF_PROCESSORS()
        READ(...) N                ! number of non-local elements on R(I)
        ALLOCATE(SLHALO(I)%IDX(N))
        READ(...) SLHALO(I)%IDX(:) ! indices of non-local elements on R(I)
      END DO

!HPF+ REDISTRIBUTE(INDIRECT(MAP),*), HALO(SLHALO(I)%IDX,*) ON R(I) :: ZSL
```

Fig. 1. Specifying Halos. In this example, the first dimension of array `ZSL` represents the reduced grid used in the semi-Lagrangian interpolation method of the IFS weather prediction code [1], and the second dimension stores physical quantities at each grid point. In order to store the reduced grid without wasting memory it is linearized along the latitudes. As a result of this linearization, an `INDIRECT` distribution is required. The figure sketches the local section of a particular processor (black) and the required halo (gray). Both the mapping array `MAP` and the halo array `SLHALO`, which describes on each processor the non-local indices with respect to the first dimension of `ZSL`, are read before the array is distributed by means of the `REDISTRIBUTE` statement. The array `SLHALO` is an array of arrays defined by means of derived types, where `SLHALO(I)%IDX` contains the non-local indices that need to be accessed on processor `R(I)`.

The information provided by halos is utilized to allocate memory for non-local data by extending the local section on each processor according to the size of the halo. Moreover, the communication schedule for gathering the non-local data described by a halo from their owner processors can be determined at the time the array is allocated. This schedule is then stored in the distribution descriptor of the array and can be applied whenever it is necessary to update the halo. The main advantage of such a scheme is that the same schedule can be used in different parts of a program instead of determining a communication schedule by means of an inspector at each individual loop. Since the halo must represent the super-set of all non-local data accesses, it can never happen that the array has to be reallocated with a larger extension area.

Halos can be considered as a generalization of *shadows*, which have been included in the Approved Extensions of HPF-2. By means of the `SHADOW` directive the user may specify how much storage will be required for storing copies of non-local data accessed in nearest neighbor stencil operations. The shadow directive has the form `SHADOW(ls:hs)` where `ls` and `hs` (which must be scalar integer specification expressions) specify the widths of the shadow edges at the low end and high end of a distributed array dimension. Shadows are usually specified only in the context of `BLOCK` distributions, whereas halos may be specified for any distribution, including `GEN_BLOCK` and `INDIRECT` distributions. As currently defined in HPF-2, shadows are not applicable in the context of dynamically distributed arrays.

3 High-level Communication Primitives for Halos

In order to enable the user to control the communication associated with halos without having to deal with the details of message-passing, high-level communication primitives are provided.

By means of the `UPDATE_HALO` directive the user may enforce that the halo of all processors is updated by receiving the actual values from the owner processor. Although updating of halos could be managed automatically, it may be impossible for the compiler to avoid redundant updates of halos. If, for example, an array with a halo is accessed in subsequent loops without writing to elements that are contained in the halo of some processor, it is not necessary to update the halo at each individual loop. In the case of indirect data accesses or complicated control structures, such an optimization can not be performed automatically and thus needs to be triggered by the user.

The halo of a distributed array may not only be used to read copies of non-local data elements but also to perform non-local writes of array elements. This may be the case if particular assignments are not processed according to the owner computes rule, for example, if the iteration space of a loop is distributed explicitly by the user by means of the `ON HOME` clause. If processors write to non-local array elements in their halo in the context of array reduction operations, then the results of these operations have to be sent to the owners of the corresponding array elements and accumulated. For this purpose the `REDUCE_HALO`

directive is provided. In a REDUCE_HALO operation the flow of messages is in the opposite direction of an UPDATE_HALO operation. A processor which is the *owner* of an array element that is contained in the halo of other processors, receives all the corresponding halo elements from these processors and accumulates them according to the reduction operation specified in the REDUCE_HALO directive.

Similar to asynchronous I/O as defined in the HPF-2 Approved Extensions, the UPDATE_HALO and REDUCE_HALO directives may be combined with the ASYNCHRONOUS attribute and the WAIT statement in order to hide the communication overheads by overlapping the update or reduction of the halo with computation. Updating the halo of an array asynchronously and utilizing halos in the context of reductions is shown in Figure 2.

4 Implementation and Performance Results

In this section we give a brief overview of the implementation of the described features in the Vienna Fortran Compiler (VFC) [3], a source-to-source compilation system that transforms HPF+ [4] programs into explicitly parallel Fortran 90 SPMD message-passing programs with embedded calls to an MPI based runtime library. VFC supports block, cyclic, generalized block, indirect distributions, basic alignments, and dynamic data distributions. For the parallelization of irregular loops the INDEPENDENT directive, the ON HOME clause, and the REUSE clause are provided. The REUSE clause [2] allows the user to assert that the access patterns of distributed arrays are invariant and thus the corresponding communication schedules may be reused once they have been determined by means of an inspector or based on the HALO attribute.

VFC transforms all distributed arrays such that they are dynamically allocated in the target program. For each distributed array calls to runtime library procedures are generated that take as arguments the shape of the array and the distribution and, if specified, the halo information. These functions evaluate at runtime the distribution and compute the local section that has to be allocated on the processor that called the function. If an array has a halo, the local section is extended accordingly. In this case, the communication schedules required for updating the halo are computed and stored in the distribution descriptor of the array.

The parallelization of INDEPENDENT loops with indirect accesses to distributed arrays is based on the inspector/executor paradigm [12, 13]. The required runtime support is based on an extended version of the CHAOS/PARTI library [6]. The first part of the inspector determines a distribution of the loop iteration space, usually by evaluating the ON HOME clause. The remainder of the inspector code consists of a series of loops that trace the accesses to distributed arrays on a dimension-by-dimension basis. The resulting traces are passed to runtime procedures that determine the required communication schedules, the size of the extension area for storing copies of non-local data, and a corresponding local addressing scheme.

```
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
      REAL           :: X(3,NN), F(6,NN)
      INTEGER        :: IX(4,NE)
!HPF+ DYNAMIC, HALO :: X, F
!HPF$ DYNAMIC       :: IX
!HPF$ ALIGN F(*,I) WITH X(*,I)

      INTERFACE
         EXTRINSIC(HPF_LOCAL) FUNCTION halo_function(I, BS, IX)
             INTEGER, POINTER :: halo_function(:)
             INTEGER          :: I, BS(:), IX(:,:)
             ...
         END FUNCTION halo_function
      END INTERFACE
      ...

!HPF+ REDISTRIBUTE(*,GEN_BLOCK(BS)),                      &
!HPF+ HALO (halo_function(I, BS, IX)) ON R(I) :: X
      ...
!HPF+ UPDATE_HALO, ASYNCHRONOUS(ID=I1) :: X
      ...    ! some computation that does not involve X
      F = 0.
!HPF+ WAIT (ID=I1)
      ...

!HPF$ INDEPENDENT, ON HOME (IX(1,I)), REDUCTION(F)
      DO I = 1, NE
        F(:,IX(:,I)) = F(:,IX(:,I)) + ...
      END DO
      ...
!HPF+ REDUCE_HALO(+) :: F
```

Fig. 2. Update of halos. This example is an excerpt of a crash simulation kernel [5]. The code employes a finite element mesh based on 4-node shell elements as sketched in Figure 3. In the code the mesh is presented by means of the arrays `X` and `F` which store the coordinates and forces of each node, and the array `IX` which stores the 4 node numbers of each element. At the time the `REDISTRIBUTE` statement is executed, the halo of `X` is determined on each processor by evaluating the array valued function `halo_function`. Since `F` is aligned with `X` it also gets redistributed with the same halo as `X`. The `UPDATE_HALO` operation ensures that all copies of a particular element of X that are in the halo of some processor have the same value as on the owner processor. The communication to update the halo of `X` is done asynchronously and overlapped with some computation that does not involve `X`. The `INDEPENDENT` loop performs a sum-scatter reduction operation to add the elemental forces back to the forces stored at the nodes. The loop can be executed without communication due to the halo of `F`. In Figure 3, the local part of this operation is indicated by the solid arrows. The execution of the `REDUCE_HALO` directive accumulates all halo elements of `F` on the corresponding owner processors. For example, processor 1 receives the local values of `F` at node `n3` from processor 2 and processor 3, and adds them to its own value of `F` at node n3.

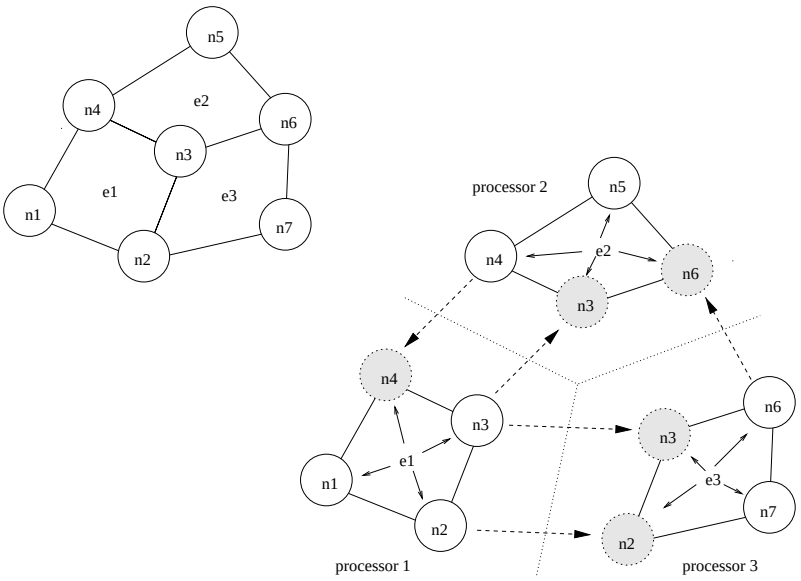


Fig. 3. Finite Element Mesh. This figure sketches the finite element mesh utilized by the crash simulation kernel. The mesh is based on 4-node shell elements In the figure three elements are shown, each of which is mapped to a processor as depicted in the right-hand side. The nodes at the boundary between the processors, have to be accessed on several processors. For example, node `n3` is owned by processor 1 but needs to be accessed also on the other processors. Thus `n3` is contained in the halo of processor 2 and 3. The local (owned) nodes L(p) and the halos H(p) on processor p are as follows: L(1) = {n1, n2, n3}, H(1) = {n4}; L(2) = {n4, n5}, H(2) = {n3, n6}; L(3) = {n6, n7}, H(3) ={n2, n3}. The dotted arrows indicate the flow of messages in the case of an `UPDATE_HALO` operation. The solid arrows indicate the local part of the sum-scatter reduction operation to add the elemental forces back to the forces stored at the nodes. The communication required during the `REDUCE_HALO` operation is in the oposite direction of the dotted arrows.

If a distributed array has a halo, then the computation of communication schedules is not performed within the inspector, since in this case a schedule has already been determined at the time the array has been allocated. Moreover, if the compiler can determine that the access patterns of a distributed array are invariant, or if the user asserts this by means of the `REUSE` clause, then the inspector code is guarded such that it is executed only when the loop is reached for the first time.

In Table 1 performance results of a crash simulation kernel [5] (see Figure 2 and 3) and an equivalent hand-coded MPI/F77 version on a NEC Cenju-4 distributed memory machine with 32 VR10000 processors, a NEC SX-4 shared memory vector supercomputer with 8 processors, and on an IBM SP2L with 32 RS/6000 processors are presented.

	NEC Cenju-4			IBM SP2			NEC SX-4		
	MPI	HPF+	HPF-2	MPI	HPF+	HPF-2	MPI	HPF+	HPF-2
1 processor	524	592	8721	904	1320	16953	33	42	4414
2 processors	263	293	4318	457	676	8578	18	23	2235
4 processors	129	144	2217	235	344	4512	8.7	4.4	1125
8 processors	63	70	1143	128	192	2938	4.4	8.2	579
16 processors	31	36	619	79	106	1462	-		
32 processors	16	19	369	48	73	990	-		

Table 1. Elapsed times (secs) of main computational loop of crash simulation kernel.

The kernel is based on an explicit time-marching scheme realized as a sequential loop from within procedures to calculate the shell-element forces and to update the velocities and displacements are called. In our evaluation a mesh with 102720 nodes and 102400 elements was used and 1000 time-steps were performed.

Table 1 shows the elapsed times of the hand-coded MPI kernel, the times obtained with the HPF+ kernel that utilized the described `HALO` features (except asychronous updates) and the `REUSE` clause, and the times for a variant using HPF-2 features only. The HPF kernels were parallelized with VFC 1.5 and the generated SPMD programs were compiled with the Fortran 90 compiler of the target machine.

The huge gap between the two HPF variants is due to the fact that in the HPF-2 kernel communication schedules are computed with an inspector in every iteration, wheres in the HPF+ kernel the communication schedules required for `X` and `F` are computed once, based on the `HALO` attribute, at the time the arrays are allocated/distributed and reused over all time-steps.

5 Conclusions

In this paper we presented language extensions for HPF that allow the user to specify non-local access patterns for distributed arrays and to control the communication associated with these non-local accesses. The concept of halos allows the user to specify in addition to the distribution of arrays a description of the non-local data elements that may have to be accessed at runtime. This information allows computing the communication schedules for implementing non-local data accesses at the time the distribution of an array is evaluated and reusing these schedules in different parts of a program. By means of high-level communication primitives the user can control and optimize the communication associated with halos without having to deal with the details of message-passing. The performance results show that by using these extensions a performance close to hand-coded message-passing codes can be achieved for irregular codes.

Similar extensions have been proposed in HPF/JA [11], including *explicit shadows*, where the user may indicate that computation should be performed

on the local section of a distributed array and on the allocated shadow edges by means of an extended on-clause. The `REFLECT` directive is provided to control the update of shadow edges. HPF/JA includes also language features for communication schedule reuse.

In contrast to the static concept of shadows as provided by the Approved Extensions of HPF-2, halos may also be used in the context of dynamically distributed arrays. The halo of an array may be changed whenever the array has to be redistributed.

References

1. S. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. The IFS model: A parallel production weather code. Parallel Computing 21, 1995.
2. S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-Like Languages. ACM Proceedings International Conference on Supercomputing, Melbourne, Australia, July 1998.
3. S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. Proceedings International Conference on High Performance Computing and Networking (HPCN'98), Amsterdam, April 1998, Lecture Notes in Computer Science, Vol. 1401, pp. 816-827, Springer Verlag.
4. S. Benkner. HPF+: High Performance Fortran for Advanced Industrial Applications. Proceedings International Conference on High Performance Computing and Networking (HPCN'98), Amsterdam, April 1998, Lecture Notes in Computer Science, Vol. 1401, pp. 797-808, Springer Verlag.
5. J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, M. Holzner. Performance Issues of the Parallel PAM-CRASH Code. The International Journal of Supercomputing Applications and High-Performance Computing, Vol.11, No.1, pp.3-11, Spring 1997.
6. R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI Library to Irregular Problems in Computational Chemistry and Computational Aerodynamics, Proceedings of the 1993 Scalable Parallel Libraries Conference, pp. 45-56, IEEE Computer Society Press, October 1993.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification 2.0*, Rice University, January, 1997.
8. 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.
9. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Vers. 1.1, June 1995.
10. 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.
11. JAHPF Homepage: http://www.tokyo.rist.or.jp/~shunchan/index-e.html
12. R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Proceedings Supercomputing '93, pp. 361-370, 1993.
13. J. Saltz and K. Crowley and R. Mirchandaney and H. Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. Journal of Parallel and Distributed Computing, 8(4), pp. 303-312, April 1990.