HPF Library, Language and Compiler Support for Shadow Edges in Data Parallel Irregular Computations

Thomas Brandes
Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
e-mail: brandes@gmd.de

Abstract

On distributed memory architectures data parallel compilers emulate the global address space by distributing the data onto the processors according to the mapping directives of the user and by generating explicit inter-processor communication automatically. A shadow is additionally allocated local memory to keep on one processor also non-local values of the data that is accessed or defined by this processor. While shadow edges are already well studied for structured grids, this paper focuses on its use for applications with unstructured grids where updates on the shadow edges involve unstructured communication with complex communication schedules.

The use of shadow edges is considered for High Performance Fortran (HPF) as the de facto standard language for writing data parallel programs in Fortran. A library with a HPF binding provides the explicit control of unstructured shadows and their communication schedules, also called halos. This halo library allows writing HPF programs with a performance close to hand-coded message-passing versions but where the user is freed of the burden to calculate shadow sizes and communication schedules and to do the exchanging of data with explicit message passing commands.

In many situations, the HPF compiler can create and use halos automatically. This paper outlines how the HPF compiler has to be extended to use the functionality of the halo library and shows the advantages and also the limits of this approach. The halo library and an automatic support of halos have been implemented within the ADAPTOR HPF compilation system. Example programs of typical simulation codes show the use of the concepts and the results verify the effectiveness of the chosen approach.

1 Introduction

High Performance Fortran (HPF) [9] is a data parallel, high level programming language for parallel computing that is expected to be more convenient than explicit message passing and to allow higher productivity in software development. Especially, HPF can reduce the manual effort required to convert sequential programs into a useful, scalable parallel form. Yet, current HPF compilers suffer from low performance of the generated code, especially for application codes that use indirect addressing of distributed data, e.g. in computational mechanics programs on unstructured meshes (computation fluid dynamics, finite element, etc).

In this paper, we address the issue of shadow edges that have been proven to be one kind of optimization that is absolutely necessary to make HPF competitive. While shadow edges are already a well known optimization technique for structured applications [8] and are considered within HPF 2.0 [9], they are equally important for unstructured applications. Beside the performance, shadow edges are also very helpful to avoid memory allocation for temporary data. As many scientific applications are already very keen on memory, this optimization will allow to run bigger problem sizes on the same machine configuration.

The idea of unstructured shadow edges with corresponding communication schedules, also called halos, has already been followed within the HPF+ project [2] and implemented in the Vienna Fortran Compiler [4]. Their approach is based on additional language features and gives for typical examples nearly the same performance as the hand-written counterpart using message-passing [3].

This paper describes a library approach that allows the user to create and to use halos explicitly and therefore gives more flexibility. As the use of the halo library reduces portability and requires a certain understanding of the user, the HPF compiler can be extended in such a way that halos are used automatically.

Section 2 explains the current manual practice used in message passing programs for the parallelization of applications with unstructured grids. Section 3 explains the benefits of shadow edges for HPF compilation. Section 4 defines halos as a data structure for unstructured shadow edges together with their communication schedules. The functionality of the halo library is described in Section 5. A typical example program that benefits from using halos is presented for a FEM kernel in Section 6. The support of halos by additional HPF language features instead of a halo library is discussed in Section 7. Section 8 shows how far it is possible that the HPF compiler uses halos completely automatically.

The library and the automatic techniques presented in this report have been implemented and evaluated within the ADAPTOR HPF compilation system. This implementation and performance results are outlined in Section 9. Some basic performance results give insight when it is useful to use unstructured shadows and how important the reuse of halo schedules is. The results for an FEM kernel demonstrate how the performance can be dramatically increased by using halos. Conclusions are given in Section 10.

2 Shadow Edges in Message Passing Programs

SPMD programs (single program, multiple data) based on message passing use geometric domain decomposition techniques for mapping computational mechanics programs onto parallel machines. Decomposition of the problem geometry into P pieces or subdomains, one for each of P processors, allows concurrent calculation for each subdomain. Data dependences between subdomains are resolved through communication of the necessary data between the processors owning the corresponding subdomains.

For many applications, it is useful to extend each subdomain with a shadow area to overlap with its neighbors. This shadow area will contain the non-local values of neighbored subdomains. By this way, no alteration of the source code is necessary to access the non-local data.

Figure 1 shows a typical mesh consisting of two mesh entity types: grid points and elements. By a mesh partitioning tool (e.g. JOSTLE [12]), the unstructured mesh is partitioned for a given number of processors in such a way that a good load balancing and a high data locality is achieved. The latter condition requires to

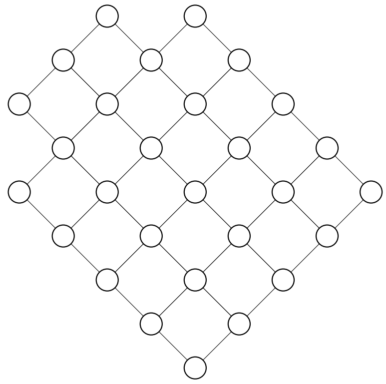


Figure 1. Typical mesh structure in computational mechanic programs.

minimize the number of graph edges between different subdomains. The higher the data locality is, the less memory size is needed for the overlapping (shadow). Figure 2 shows the partitioning of the grid in Figure 1 for three processors.

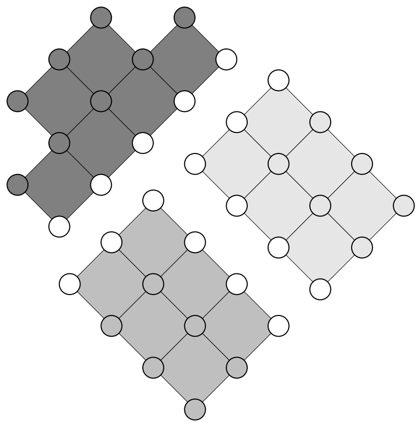


Figure 2. Mesh partitioning for three processors with overlapping points.

Parallel execution performs the concurrent computation on the values of the data structures assigned to each processor. Typically, this calculation may need the values of non-local data. E.g. for calculating the forces of the elements, the position values of some node points are needed that are not local. If these values are available in an updated shadow area, the local computation is exactly like the global computation restricted to the local parts of the data structures.

An update of the shadow area requires unstructured communication between the processors. Every owning processor has to send the actual values to all the processors that have non-local copies in their shadow area. All the send and receive information will be hold in a communication schedule that does not change as long as the mesh structure remains unchanged. Many applications need also a reduce operation on the shadow area where all non-local copies will be reduced to a single value. But this operation can use the same communication schedule where only send and receiving is switched.

In a message passing program, the user itself is responsible for the calculation of shadow sizes and for the communication schedule. This work is tedious and error prone and requires a lot of additional coding.

3 The Importance of Shadow Edges for HPF Compilation

When compiling independent computations for distributed memory architectures, the HPF compiler determines at first the home of the computations. Then the compiler has to look at non resident data accesses and to extract the necessary communication out of the independent computations.

```
      integer L1(N), L2(N), L3(N)
      real X1(N), X2(N), X3(N)
      real A(NA), B(NB), C(NC)
!hpf$ distribute(block) :: X1, A, B, C
!hpf$ align with X1 :: L1, L2, L3, X2, X3
      ...
!hpf$ independent, on home (X1(I)), reduction (C)
      do I = 1, N
         X1(I) = A(L1(I)) ! non local read of A
         B(L2(I)) = X2(I) ! non local write of B
         ! non local accumulation of C
         C(L3(I)) = C(L3(I)) + X3(I)
      end do
```

Figure 3. HPF code with unstructured communication.

Figure 3 shows a typical HPF code with indirect addressing that needs unstructured communication. If we assume that one iteration `I` of the `INDEPENDENT` loop will be executed by the owner of `X1(I)`, the read access to the array `A`, the write access to `B` and the accumulate operation for `C` require communication. All other accesses are local, especially the read accesses to the index values `L1(I)`, `L2(I)`, and `L3(I)`.

The usual strategy of HPF compilers is to extract the communication by creating temporary arrays and generating pre-fetch and post-store communication. By this way, all data accesses in the `INDEPENDENT` loop become resident. The HPF compiler handles the example code as if the user had written it like it is shown in Figure 4.

```
      real TMP_A(N), TMP_B(N), TMP_C(N)
!hpf$ align with X1 :: TMP_A, TMP_B, TMP_C
      ...
      ! pre-fetch communication
      forall (I=1:N) TMP_A (I) = A(L1(I))
      ! initialization
      forall (I=1:N) TMP_C (I) = 0.0
      ! independent loop has only local access
!hpf$ independent, on home (X1(I)), resident
      do I = 1, N
         X1(I)    = TMP_A (I)
         TMP_B(I) = X2(I)
         TMP_C(I) = TMP_C(I) + X3(I)
      end do
      ! copy scattering
      forall (I=1:N) B(L2(I)) = TMP_B(I)
      ! sum scattering
      C = sum_scatter (TMP_C, C, L3)
```

Figure 4. Extraction of communication with temporaries.

The unstructured communication before and after the loop is realized by an inspector/executor scheme [10]. In the inspector phase the processors exchange the information about non-local accesses and build up a communication schedule that drives the data exchange in the executor phase.

In case of nearest-neighbor codes, e.g. in discretization of partial differential equations or implementing convolutions, shadow edges are a very useful optimization technique to avoid temporary arrays and local data movements. Due to the block distribution, only neighbored elements are exchanged and temporary arrays would contain mainly local values. The use of shadow edges avoids the temporary arrays. The communication is done by updating the shadow edges before the parallel loop. The computations within the loop are local and do not require any communication.

Also in unstructured applications with indirect addressing, most of the indexes might be local if there is a high data locality. The use of temporaries would imply a lot of additional memory and a lot of local memory transfer. The use of shadow edges would give the same benefits as for structured applications. But this kind of shadow edges involves some additional problems:

- The size of the shadow edges depends on the values of the indirection array and will only be known at runtime.

- The update of the shadow edges requires unstructured communication and the calculation of a communication schedule at runtime like in the inspector/executor approach. Reusing this communication schedule as long as the indirection array does not change is a very important optimization.

- The transformation of the global index values in the indirection array to local index values is not as trivial as in the regular case.

4 Halos

This section describes the data structures of a halo in more detail. A halo does not only specify the shadow edges but also its related communication schedule and the translation from global indexes to shadow indexes.

4.1 The Principle of a Halo

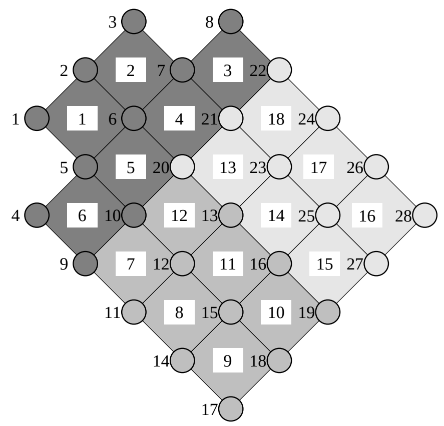


Figure 5. Mesh: distribution of elements and nodes.

```
      real, dimension (N)       :: X
      real, dimension (M)       :: Y
      integer, dimension (4,M)  :: L
!hpf$ distribute (block) :: X, Y
!hpf$ align L(*,I) with Y(I)
      ...
!hpf$ independent, on home (Y(J))
      do J = 1, M
         Y(J) = f(X(L(1,J)),X(L(2,J)),
     +               X(L(3,J)),X(L(4,J)))
      end do
```

Figure 6. Example code for using halos.

Figure 6 shows a simple HPF code where the use of halos might be useful. `X` is a distributed array that is indirectly addressed with an integer array `L`, also distributed among the processors. We call the array `X` the *halo array* and the array `L` the *halo indexes.* The code can also be applied to the example mesh of Figure 5 where `X` stands for data of the nodal points and `Y` for data of the elements. The array `L` is the connectivity matrix holding the relation between elements and nodes.

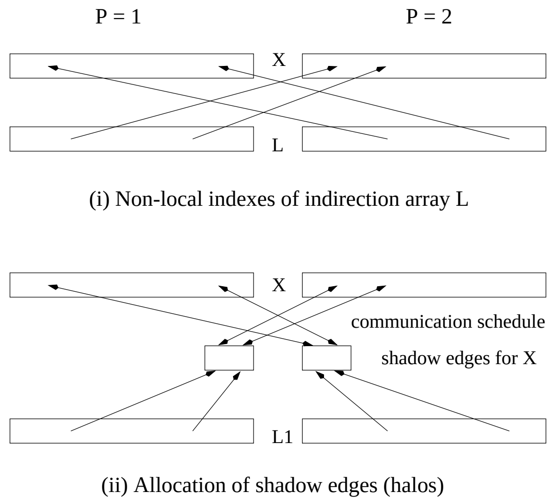


Figure 7. Shadow area and data exchange.

Figure 7 gives the idea of a halo and how it is computed. Every processor looks at its local part of the array `L` and determines which of these indexes are non-local for the array `X`. The number of the non-local indexes gives the size of the shadow area that is needed for the array `X`. For every non-local index there will be a corresponding index pointing to the shadow area. Every element in the shadow area corresponds to an element owned by a remote processor and the communication schedule reflects this relation .

Figure 8 shows for the example mesh of Figure 5 where shadow nodes are needed when each processor allocates its local part of the mesh.

4.2 Allocation of Shadow Edges

The number of non-local elements corresponds to the size needed for the shadow edges. In many applications, this is not necessarily the number of non-local halo indexes as a non-local element might be accessed twice or even more. The necessary size of shadow edges might be different for the processors onto which the array is distributed.

Shadow memory for the non-local elements must be allocated for the halo array X. This memory is usually put at the end of the local part of the array. So the array will be allocated with the local size of the distributed dimension plus the shadow size. If the array has already been allocated before the size of the shadow is known, reallocation is necessary.

Shadow edges might also be allocated completely separately as own arrays or on the heap. But this complicates the partitioning of the computations as the compiler has always to distinguish between local accesses and accesses to the shadow area.

4.3 Global and Local Addressing

For the independent computation, the global addresses must be translated to local addresses pointing into the shadow area. Usually a temporary integer array `L1` will be created. It will be initialized with the values of `L` and then the non-local indexes will be replaced with the new ones pointing into the shadow area. In many situations, these indexes can be translated in place in the old array `L` avoiding additional memory.

Figure 8 shows for the example mesh of Figure 5 where the global indexes are replaced with local indexes. It is assumed that the shadow node points are allocated behind the local node points.

4.4 The Halo Communication Schedule

With the creation of the halo, a communication schedule is set up that contains the communication pattern to actualize non-local values in the shadows or to write or reduce non-local values. It reflects the relation between local values and non-local copies in the shadow edges on other processors.

The communication schedule as part of the halo data structure is built up in exactly the same way as it is done for the well known inspector-executor scheme. The building of the schedule corresponds to the inspector phase, updates of the shadow edges are done by an executor step.

It should be mentioned that the halo data structure is a distributed data structure. E.g. processor P_1 does not have to know the size of shadow edges on other processors and the communication between processors P_i and P_j ($i \neq 1$, $j \neq 1$).

4.5 Localized Computations

With the use of shadow edges, the example code becomes now a code where all accesses in the

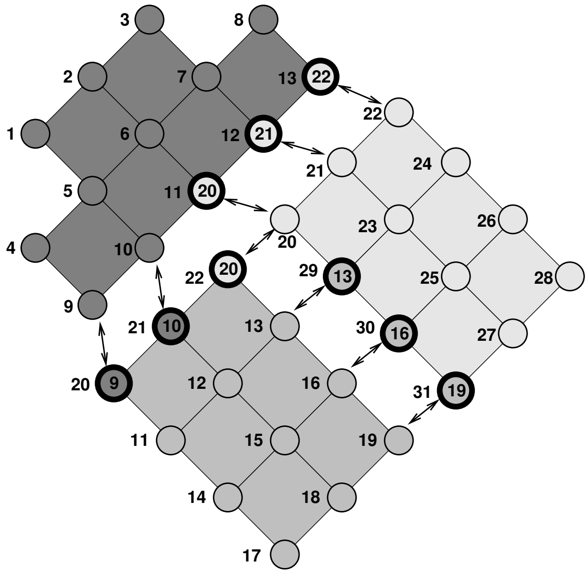


Figure 8. Mesh with shadow nodes and halo schedule.

`INDEPENDENT` loop are local and the computation can be partitioned directly onto the processors (see Figure 9).

```
      integer, dimension (4,M)  :: L1
!HPF$ align L1 with L
      ...
      L1 = L
      ! create shadow edges for array X
      ! update non-local indexes of L1
      ! update values of X in the shadow edges
!hpf$ independent, on home (Y(J)), resident(X)
      do J = 1, M
         Y(J) = f(X(L1(1,J)),X(L1(2,J)),
     +            X(L1(3,J)),X(L4(4,J)))
      end do
```

Figure 9. Localized computations.

4.6 Reusing Halos

In many applications, the arrays used for indirect addressing do not change. It might also be the case, that the same integer array is used in the same way for different arrays with the same shape and with the same distribution. In these cases, the halo structure can be reused. As the computation of shadow edges and the corresponding communication schedule is rather expensive, this optimization is very important.

There is a new runtime approach for reusing communication schedules in unstructured applications [6] that has been proven as very effective. This approach can be used in the same way for halos and their schedules. A halo structure is stored in a data base if the related halo index array has the TRACE attribute. This attribute guarantees that modifications of the array are tracked and it can be decided at runtime whether it is safe to reuse the halo schedule.

5 The Halo Library

The HPF halo library allows the explicit use of halos for distributed arrays that are indirectly accessed. It provides the highest flexibility as the user has not to rely on the capabilities of the compiler to use halos automatically.

5.1 Design Issues

Halos can only be defined for halo arrays that are distributed along one dimension. This is not a serious restriction as this is the case for all unstructured applications with indirect addressing considered so far.

The halo structure is completely hidden and will only be identified by an integer value. A halo is not directly related to the array for which it has been defined. So it can also be used for all arrays that have the same shape and the same mapping in the distributed dimension.

From the users point of view, the following topics are very important for a successful use of the halo library:

- The indirectly accessed array must have sufficient shadow. This shadow can be either defined by the SHADOW directive, or the array is dynamic in the sense of HPF to allow reallocation.
- The user takes responsibility where global and where localized shadow indexes are used. He must use the local indexes in the independent computations where the halo indexes are used. If the indirection array is used in other operations, the old global indexes must be restored.
- In the independent computation, the access to the indirectly accessed array must be defined as local via the RESIDENT clause. Otherwise the compiler would treat the shadow indexes like global indexes.
- Halos and their related schedules can be reused as long as the halo is defined. The halo must be redefined when the integer array with the halo indexes is modified. This is the case when the geometry of the mesh changes.

5.2 Definition of Halos

The following subroutine sets up internal data structures for a halo where ARRAY is the indirectly accessed array and INDEXES is an array of global indexes used for the distributed dimension. The halo data structures are not explicitly available but are referred to by an integer value.

```
subroutine HALO_DEFINE (HALO_ID, ARRAY, INDEXES
                        [,MASK] [,LB] [,UB])
```

- HALO_ID is a scalar INTENT(OUT) argument of type default integer.
- ARRAY is an INTENT(IN) argument. It may be of any type, but it must be a distributed array that is exactly distributed along one dimension. All other dimensions must be serial.
- INDEXES is an INTENT(IN) argument. It must be of type default integer.
- MASK is an optional INTENT(IN) argument. It must be of type default logical. MASK and INDEXES must be conform and have the same distribution.
- LB and UB are optional INTENT(IN) arguments. They must be scalar of default integer. They specify the boundaries of the ARRAY argument in the distributed dimension. If the ARRAY is allocated, the boundaries must be exact. If the array is not allocated yet, the arguments are mandatory and specify the future size of the array in that dimension.

The HALO_DEFINE routine has not any side effects on the ARRAY and INDEXES arguments. The argument ARRAY gives the routine only the information how to find out which of the indexes are global or local. Shadow edges must not be available at this time. If the array is distributed but not allocated yet, the future dimensions can be specified. The distribution information of the ARRAY and the boundary values will be used to establish an internal template reflecting the halo arrays.

The halo can be used later for all arrays that have the same shape and the same mapping in the distributed dimension. But updates and reductions on the halo require that the corresponding arrays have sufficient shadow area.

In all of the following operations, HALO_ID is an INTENT(IN) argument of type default integer. Its value must refer to a defined halo structure.

5.3 Releasing Halos

The internal data structure for the halo can be freed with the HALO_FREE routine.

```
subroutine HALO_FREE (HALO_ID)
```

A halo should be freed if it is no more used or if the integer array with the halo indexes has been modified. If the indirection array used for the definition of the halo has the TRACE attribute, the internal structures built for the halo will only be freed when the indirection array changes. This allows the reuse of halos in case of an automatic approach.

5.4 Defining Halo Shadows

The following subroutine returns the maximal size of shadow needed on one processor when using the halo for a distributed array. Its value is like the whole halo structure dependent on the number of active processors. When the size of the shadow is known, the user might define new arrays with the appropriate shadow.

```
subroutine HALO_MAX_SIZE (HALO_ID, SIZE)
```

- SIZE is an INTENT(OUT) argument of type default integer.

The following subroutine can be used to verify if there is sufficient shadow available for the ARRAY. This routine might be useful when the user has defined a shadow via the SHADOW directive of HPF.

```
subroutine HALO_VERIFY_SHADOW (HALO_ID, ARRAY, OKAY)
```

- ARRAY is an INTENT(IN) array variable. It must be distributed along one dimension, where the distribution of this dimension conforms to the distribution of the array used for the definition of the halo structure.
- OKAY is an INTENT(OUT) argument of type default logical.

If ARRAY is dynamic and not sufficient shadow is available, the following routine guarantees a reallocation of the array that will have sufficient shadow size. This subroutine returns immediately if sufficient shadow is available for ARRAY.

```
subroutine HALO_CREATE_SHADOW (HALO_ID, ARRAY)
```

- ARRAY is an INTENT(INOUT) array argument. It must be distributed along one dimension, where the distribution of this dimension conforms to the distribution of the array used for the definition of the halo structure. It should be dynamic when not sufficient shadow is available.

5.5 Localizing Indirection Arrays

The following routines transform global indexes to local indexes pointing to the shadow area and vice versa.

```
subroutine HALO_LOCAL_INDEXES (HALO_ID, INDEXES)
subroutine HALO_GLOBAL_INDEXES (HALO_ID, INDEXES)
```

- INDEXES is an INTENT(INOUT) argument. It must be of type default integer.

Attention: It is absolutely left to the user to know whether the indirection array contains global indexes or shadow indexes. If the localization of indexes is called twice, shadow indexes might be considered as global indexes resulting in completely undefined indexes.

Instead of localizing indexes in place, it is also possible to use a new temporary array with localized indexes. This might avoid confusion between global and shadow indexes, but it also becomes necessary if the global indirection array is not only used for the halo array. The routine HALO_NEW_LOCAL_INDEXES transforms all global indexes to a new array with local indexes pointing to the shadow area.

```
subroutine HALO_NEW_LOCAL_INDEXES
            (HALO_ID, G_INDEXES, L_INDEXES)
subroutine HALO_FREE_LOCAL_INDEXES (HALO_ID, L_INDEXES)
```

- G_INDEXES is an INTENT(IN) array argument. It must be of type default integer.
- L_INDEXES is an INTENT(INOUT) argument. It must be of type default integer and have the pointer attribute.

The temporary array will be allocated within the routine. Therefore the argument L_INDEXES must have the pointer attribute. The temporary array with the local indexes is part of the halo structure and is also reused when the halo is reused. This makes the routine very effective for the automatic use of halos (see Section 8).

5.6 Shortcuts

The routine HALO_ALL_DEFINE can be used as an abbreviation for the usual definition of a halo structure with creating the shadow area for the halo array and translating the global indexes to local indexes.

```
subroutine HALO_ALL_DEFINE (HALO_ID, ARRAY,
                  INDEXES [, MASK] [, HEAP_FLAG])
! implements
call HALO_DEFINE (HALO_ID, ARRAY, INDEXES [, MASK])
call HALO_CREATE_SHADOW (HALO_ID, ARRAY [, HEAP_FLAG])
call HALO_LOCAL_INDEXES (HALO_ID, INDEXES)
```

The routine HALO_AUTO_DEFINE is nearly the same but uses a temporary array for the localized indexes.

```
subroutine HALO_AUTO_DEFINE (HALO_ID, ARRAY, INDEXES,
                             TMP_INDEXES [,MASK])
! implements
call HALO_DEFINE (HALO_ID, ARRAY, INDEXES [, MASK])
call HALO_CREATE_SHADOW (HALO_ID, ARRAY)
call HALO_NEW_LOCAL_INDEXES (HALO_ID, INDEXES,
                             TMP_INDEXES)
```

The routine HALO_AUTO_FREE does not only free the halo data structure but also the temporary indexes.

```
subroutine HALO_AUTO_FREE (HALO_ID, TMP_INDEXES)
! implements
call HALO_FREE_LOCAL_INDEXES (HALO_ID, TMP_INDEXES)
call HALO_FREE (HALO_ID)
```

5.7 Communication for Halos

The update routine makes sure that all non-local copies in the shadows on other processors are updated with the actual value of the owning processor.

```
subroutine HALO_UPDATE (HALO_ID, ARRAY)
```

- ARRAY is an INTENT(INOUT) array argument whose shadow must be sufficient enough for the halo structure.

A reduction for a halo reduces all non-local copies in the shadows on other processors to a single value on the owning processor. Like the XXX_SCATTER operations of the HPF library, there are corresponding HALO_REDUCE_XXX operations for the different reduction operations in the halo library.

```
subroutine HALO_REDUCE_COPY    (HALO_ID, ARRAY)
subroutine HALO_REDUCE_SUM     (HALO_ID, ARRAY)
subroutine HALO_REDUCE_PRODUCT (HALO_ID, ARRAY)
subroutine HALO_REDUCE_MINVAL  (HALO_ID, ARRAY)
subroutine HALO_REDUCE_MAXVAL  (HALO_ID, ARRAY)
subroutine HALO_REDUCE_ALL     (HALO_ID, ARRAY)
subroutine HALO_REDUCE_ANY     (HALO_ID, ARRAY)
subroutine HALO_REDUCE_PARITY  (HALO_ID, ARRAY)
subroutine HALO_REDUCE_IALL    (HALO_ID, ARRAY)
subroutine HALO_REDUCE_IANY    (HALO_ID, ARRAY)
subroutine HALO_REDUCE_IPARITY (HALO_ID, ARRAY)
```

- ARRAY is an INTENT(INOUT) array argument whose shadow must be sufficient enough for the halo structure.
- ARRAY must have a valid type for the corresponding reduction operations.

5.8 Initialization of Shadow Edges

Reductions need an initialization of the reduction variable with the zero element. The unstructured reduction needs also an initialization of the elements in the shadow area. This can be done by the following routine that involves no communication:

```
subroutine HALO_INIT (HALO_ID, ARRAY, VALUE)
```

- ARRAY is an INTENT(INOUT) array argument whose shadow must be sufficient enough for the halo structure.
- VALUE is an INTENT(IN) scalar argument. It must be of the same type as the ARRAY argument.

6 Using Halos in a FEM Kernel

This section presents an example program that contains specific, important features of an industrial code used for crash simulations, in a very simplified form. It is a kernel of an explicit time-marching Finite Element program. A similar kernel program is used within the HPF+ project as a benchmark program [11].

6.1 Description of the Grid

The program works on an unstructured grid that consists of 4-node shell elements. It has NP nodal points, for each point the coordinates X, the force F and the velocity V are given. For each of the NEL elements the values IX(1,IEL), .., IX(4,IEL) specify the points belonging to the IEL-th element.

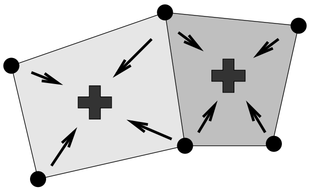
```
integer, parameter:: NEL = 25600  ! number of elements
integer, parameter:: NP  = 25760  ! number of nodes

real    F(6,NP)       ! forces
real    X(3,NP)       ! coordinates
real    V(3,NP)       ! velocity
integer IX(4,NEL)     ! points in one element
```

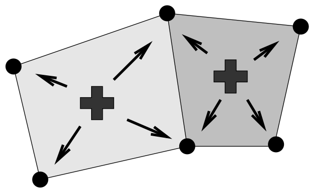
Two templates define the problem space, one for the elements, one for the points. The arrays of the programs are aligned to these two templates. A block distribution of the templates maps the arrays of the program to the processors. It should be noted that the values of the points of one element are not necessarily on the same processor as the element.

```
!hpf$ template, dimension (NEL) :: ELEMENT
!hpf$ template, dimension (NP)  :: POINT
!hpf$ distribute (block)  :: ELEMENT, POINT

!hpf$ align (*,I) with POINT(I)   :: F, V, X
!hpf$ align (*,I) with ELEMENT(I) :: IX
```



(i) Elemental stresses calculated from nodal coordinates



(ii) Nodal forces calculated from elemental stresses

Figure 10. Schematic view of the kernel.

6.2 Description of the Algorithm

The program performs stress-strain calculations on the grid. This is shown in Figure 10: for every element of the unstructured mesh the computation takes the latest values of coordinates of the nodal points, calculates "forces" on the element, adds the elemental forces back to the forces stored at the nodes. The computation of the forces represents the most important contribution of the overall computation cost. With the new values of the forces in the nodal points, the moving equations are solved to compute new positions of the points. All these calculations are repeated for a certain number of iterations.

Figure 11 shows the implementation of the FEM kernel in HPF. We assume that for the first loop an iteration is executed by the owner of the element, for the second loop an iteration is executed by the owner of the nodal point. The routines `CALC_FORCE` and `UPDATE` are `PURE` routines and can be executed independently and without any communication.

Reading the values of `X` for one element requires unstructured communication as the values of the integer index array `IX` are only known at compile time. In the same way as the read operation of the array `X`, the reduction operation on the forces `F` requires unstructured communication.

The program makes use of a run-time generated mesh corresponding to a box-beam. The ordering of nodes and elements in the generated mesh is such that the chosen block distribution corresponds to horizontal slicing of the mesh. The interface lengths are approximately constant with increasing process sizes. Therefore the distribution of the grid provides a high locality,

```
      real FORCE (6,4), XN(3,4)
      ...
      do IT = 1, ITTOT
        F = 0.0
!hpf$ independent, new(XN, FORCE), reduction (F),
!hpf$+on home (ELEMENT(IEL))
        do IEL = 1, NEL
          ! collect nodal points of element IEL
          do I = 1, 4
            do K = 1, 3
              XN(K,I) = X(K,IX(I,IEL))
            end do
          end do
          call CALC_FORCE (XN, FORCE)
          do I = 1, 4
            do K = 1, 6
              F(K,IX(I,IEL)) =
     +          F(K,IX(I,IEL)) + FORCE(K,I)
            end do
          end do
        end do ! independent loop over elements
!hpf$ independent, on home (POINT(IP))
        do IP = 1, NP
!          solve moving equation for one point
           call UPDATE (F(:,IP),V(:,IP),X(:,IP))
        end do
      end do ! iteration loop
```

Figure 11. HPF code for the FEM kernel.

most of the node points are local for each element.

6.3 Explicit Use of Halos

As the chosen distribution implies a lot of locality, unstructured shadow areas for the arrays `X` and `F` should be utilized. The same halo structure can be used for both arrays. The arrays should be defined dynamic to allow a reallocation with sufficient shadow size. As the geometry of the grid does not change, the halo structure can be used for all iterations of the outer loop. Figure 12 shows the HPF code where the unstructured communication is implemented explicitly via the halo library.

7 HPF Language Features for Halos

This section summarizes the approach followed in the HPF+ project [2] where new HPF language features have been introduced for using halos. This approach is compared with the library approach.

7.1 Definition of Halos

In the HPF+ language, a halo is defined as an attribute for an array.

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

```
      use HALO_LIBRARY
      ...
!hpf$ dynamic :: F, X
      integer :: HALO_ID
      ...
      call HALO_DEFINE (HALO_ID, X, IX)
      call HALO_CREATE_SHADOW (HALO_ID, X)
      call HALO_CREATE_SHADOW (HALO_ID, F)
      call HALO_LOCAL_INDEXES (IX)
      ...
      do IT = 1, ITTOT
        ....
        F = 0.0
        call HALO_UPDATE (HALO_ID, X)
        call HALO_INIT (HALO_ID, F, 0.0)
!hpf$ independent, new(XN, FORCE),
!hpff+on home (ELEMENT(IEL)), resident (X, F)
        do IEL = 1, NEL
           ...
        end do ! independent loop over elements
        call HALO_REDUCE_SUM (HALO_ID, F)
!hpf$ independent, on home (POINT(IP)), resident
        do IP = 1, NP
           call UPDATE (F(:,IP),V(:,IP),X(:,IP))
        end do
      end do ! iteration loop
      ...
      call HALO_GLOBAL_INDEXES (HALO_ID, IX)
      call HALO_FREE (HALO_ID)
```

Figure 12. FEM kernel using the halo library.

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*. The following example shows a typical use of the halo directive.

```
      type AOA
         integer, pointer, dimension(:) :: IDX
      end type AOA
      type (AOA), allocatable :: SLHALO(:)

!hpf$ processors :: R(number_of_processors ())

!hpf+ halo (SLHALO(I)%IDX) on R(I) :: A
```

The integer values specify for each processor the global elements that will be indirectly accessed. An internal data structure is set up that contains the related communication schedule and that is used for translating global addresses to addresses into the shadow.

A HALO is strongly connected to one array and cannot be used for arrays with the same shape and the same distribution. There is the implicit assumption that the halo is also available for all aligned arrays.

7.2 Global and Local Addressing of Halo Arrays

In the HPF+ language approach, the user is no more responsible for the translation of the indirection array to local addresses. Only the HPF compiler translates global addresses to local and shadow addresses.

The compiler has to create a copy of the indirection array where global addresses are replaced with shadow addresses. This new array might be reused if the compiler can verify that the indirection array has not changed.

The HPF compiler assumes that for every non-local access to the halo array there is an entry in the shadow. So it is not possible to use a non-local index that has not been specified in the halo clause before.

7.3 Communications for Halos

The update and reduction operations for halos are exactly like they are specified in the halo library. A halo identifier must not be specified as the halo is directly related to the indirectly accessed array. The language embedding allows the specification of reduction operations like +, *, `MIN` and so on.

```
!hpf+ update_halo :: A
!hpf+ reduce_halo(+) :: A
```

As there are no more different halo structures for different halo index arrays, these halo operations will update the whole shadow area even if it is only necessary for a certain part.

7.4 Comparison of the Language Extensions with the Halo Library

With the library approach, the user is completely responsible for the management of halos, arguments for the halo identifier must be global or passed to subroutines if necessary. The program is no more appropriate for serial compilation with a usual Fortran compiler where no halo library is available. Nevertheless, all routines of the halo library could be implemented straightforwardly by doing nothing at all and by returning default values. But the main problem is certainly the management of the localized indexes. Accesses with these indexes must be considered as resident and treated correctly. This problem does not exist with a language approach.

In the language approach, halos are directly connected with one halo array. They can be reused for arrays of the same shape only if these arrays are directly aligned to the halo array. The definition of a halo implies that every indirect access to the halo array contains only global addresses that have been used for the definition of the halo. The use of the halo library provides more freedom in the use of halos with the penalty of losing portability.

8 Towards the Automatic Use of Halos

Neither the halo library nor the HPF+ extensions are standardized. Their use requires a deep understanding of halos and an incorrect use might result in serious bugs difficult to fix. This section describes an approach for the HPF compiler to use halos automatically.

8.1 Automatic Generation of Halo Library Calls

We demonstrate the automatic generation of calls to the halo library for the FEM kernel of Figure 11. Figure 13 shows the code that is generated internally by the HPF compiler.

```
      integer :: ID1, ID2
      integer, dimension (:,:),
     +         pointer :: TMP1, TMP2
!hpf$ align with IX :: TMP1, TMP2
      ...
      do IT = 1, ITTOT
        ....
        F = 0.0
        call HALO_AUTO_DEFINE (ID1, X, IX, TMP1)
        call HALO_AUTO_DEFINE (ID2, F, IX, TMP2)
        call HALO_UPDATE (ID1, X)
        call HALO_INIT (ID2, F, 0.0)
!hpf$ independent, new(XN, FORCE),
!hpff+on home (ELEMENT(IEL)), resident (X, F)
        do IEL = 1, NEL
           ...
             XN(K,I) = X(K,TMP1(I,IEL))
           ...
             F(K,TMP2(I,IEL)) =
     +         F(K,TMP2(I,IEL)) +  FORCE(K,I)
           ...
        end do ! independent loop over elements
        call HALO_REDUCE_SUM (ID2, F)
        call HALO_AUTO_FREE (ID2, TMP2)
        call HALO_AUTO_FREE (ID1, TMP1)
        ...
      end do ! iteration loop
      ...
```

Figure 13. Automatic use of halo edges within a HPF compiler.

The compiler uses a temporary array for the local index values that needs the same amount of memory as the index array. This temporary array is allocated and defined only once as it becomes part of the internal halo structure. When the halo is reused, the temporary array is also reused. For applications that will reuse the halo, this approach is more efficient than changing the index values in place as the global index values must be restored after every use. The compiler has to make the worst case assumption that the indirection array is also used for other purposes.

8.2 Reuse of Halos

The definition of halos creates a certain overhead. But a halo and its schedule can be reused in many applications so that this overhead is amortized over multiple reuses. For the automatic reuse of halos, the main problem is the verification that the indirection array has not changed.

There is a strategy to implement efficiently an inspector-executor paradigm where communication schedules can be reused [6]. It does not require any data-flow and inter-procedural analysis at compile time. It will also reuse communication schedules in situations where even best compile time analysis might fail. The same strategy can be used for the reuse of halos. This method requires only the `TRACE` directive that tells the compiler to track modifications of arrays that have this attribute. Setting this attribute for the indirection array, the halo is reused as long as the geometry of the mesh does not change.

In the example of Figure 13 the second halo definition will return the same halo that has been created during the first definition. The arrays `X` and `F` have the same distribution and the integer array `IX` has not been modified. The temporary array `TMP2` points to the same array as `TMP1`. Though the halos are freed explicitly for every iteration of the outer loop, the internal data structures are kept in an internal data base and will be reused as long as the indirection array `IX` has not been modified.

8.3 Requirements for Using Halos

The HPF compiler should verify the following conditions before using a halo:

- The indirectly accessed array must be distributed along one dimension. It should have been defined with an explicit shadow or it should have the `DYNAMIC` attribute.

- The index value in the distributed dimension contains an integer array that is not changed within the independent loop.

The definition of a halo needs a whole integer array for the halo indexes. This might cause problems in the following situations:

- Only a part of the integer array is used within the loop. If the whole array is used for the definition of the halo, the shadows might become unnecessarily large. Integer values that are out-of-range should be ignored in any case.
- In certain situations, the compiler might be able to create a mask array and to use the `MASK` argument of the `HALO_AUTO_DEFINE` routine. But the automatic generation of such a mask can be very complicated and it restricts the possibility of reusing the halo as modifications of the masks must also be tracked.
- The index in the distributed dimension of the halo array can be an integer expression like `IND(I)+1`. In this case the compiler must generate another temporary array resulting in additional memory overhead and additional problems for automatic reuse of halos. So this should be avoided by the user.

9 Performance Results

9.1 Implementation within the ADAPTOR HPF Compilation System

ADAPTOR (Automatic Data Parallelism Translator) is a public domain HPF compilation system developed at GMD for compiling data parallel HPF programs to equivalent message passing programs [1]. ADAPTOR translates the data parallel program to an equivalent SPMD program (single program, multiple data) that is executed by all processors in their local address space. Every processor only allocates that portion of a distributed array that is owned by it. Computation partitioning is based on the owner-computes rule. All accesses to non-local data are implemented by means of message passing.

The generated SPMD program is linked with the ADAPTOR specific HPF runtime system DALIB (distributed array library) and the message passing library (see Fig. 14).

ADAPTOR supports nearly the full HPF 2.0 base language and most of the approved extensions. The source files of ADAPTOR, documentation files in PostScript and a number of example programs are available via `WWW` [1]. A basic automatic support of halos is available since Version 6.0 (June 1998), the explicit use of halos via the HPF Halo library by the user

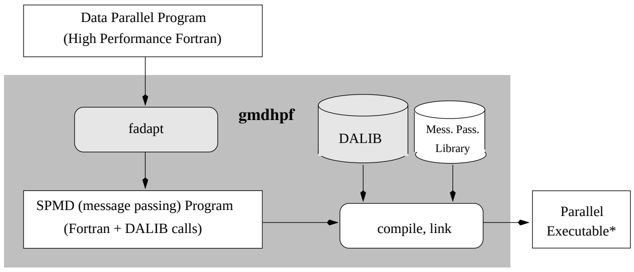


Figure 14. Overview of ADAPTOR.

and by the compiler is supported with Version 7.0 (Jan. 2000).

The halo library has been implemented in a similar way like the HPF library. Arguments will be checked at compile-time as far as possible. The implementation uses the internal array descriptors that are available for each distributed array at runtime.

ADAPTOR generates halo and corresponding calls of the halo library automatically as outlined in Section 8. Reusing of halos and related communication schedules is possible via the `TRACE` directive that generates additional code to decide at runtime whether the geometry has changed or not.

9.2 Basic Performance Results

Figure 15 shows the code that has been used to measure the benefits and penalties of halos. It contains indirect addressing of an array `A` that requires unstructured read communication.

```
      real, dimension (N) :: A, B
      integer, dimension (N) :: L
!hpf$ distribute (block) :: A, B, L
      ...
      do IT = 1, 100   ! iteration loop
!hpf$ independent, on home (B(I))
         do I = 1, N
            B(I) = B(I) + A(L(I))
         end do
      end do
```

Figure 15. Basic kernel with unstructured communication.

The code was executed on an IBM SP 2 with four processors. The times are average times for one iteration of the inner loop. Four versions of the code have been measured depending on whether a temporary array or shadow edges are used and whether the schedule/halo is reused or not.

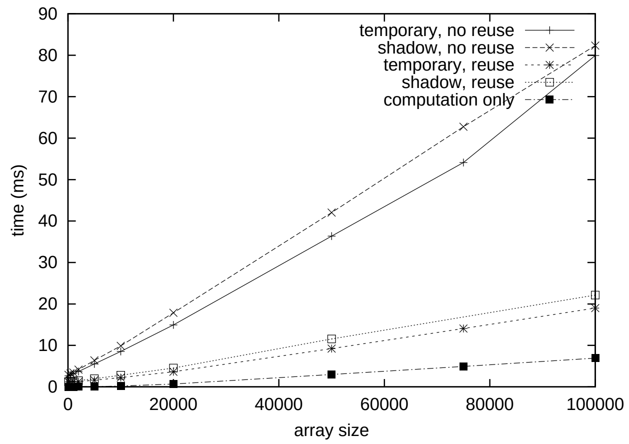


Figure 16. Execution times for completely irregular communication patterns.

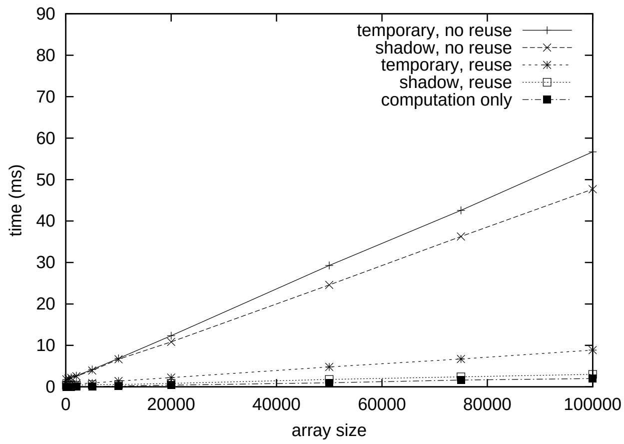


Figure 17. Execution times for communication patterns with high locality.

Figure 16 shows execution times for different array sizes `N` where the indirection array `L` is completely random (all-to-all communication). Without reusing the schedule/halo, the use of shadow edges causes a small performance loss (around 10 %) due to the computation of the localized indexes. Reusing schedules/halos reduces the average iteration time dramatically. As the time needed for the creation and definition of the local indexes is also amortized, shadow edges now show a small performance gain as no temporary for the values `A(L(I))` is needed.

Figure 17 shows execution times for different array sizes `N` where the indirection array `L` has a high locality. Without schedule/halo reusing the benefits of shadow edges are only very small. With a reuse of the internal inspector data, the shadow edges show a very high performance gain against the use of temporaries as the locality implies only very few data movements. The time of one iteration is very close the pure computational time.

The results show that reusing communication/halos is the most important issue and that shadow areas are only advantageous when the indirection arrays have a high locality.

9.3 Performance Results for a FEM Kernel

This section presents results for the FEM kernel described in Section 6. Like for the basic kernel, four different versions (with or without shadow, with or without schedule/halo reuse) have been considered. An additional version with explicit use of halos has also been measured where the global indexes have been replaced with the local indexes directly in the indirection array and thus avoiding the temporary array.

Table 1 shows the average execution times (in milliseconds) for one of 100 iterations on the IBM SP2 with up to 32 processors. Due to the high locality, the use of shadow edges gives a speed-up of nearly 2. The explicit use of halos results only in a very small performance gain when compared with the automatic use.

IBM SP2	1	2	4	8	16	32
temporary	885	451	238	122	68	51
temp., reuse	544	279	144	72	40	23
shadow	373	193	100	61	32	23
shadow, reuse	264	136	71	39	22	14
explicit	261	133	69	38	21	13

Table 1. Execution times for one iteration (ms) of the FEM kernel on the IBM SP2.

10 Conclusions

There is no doubt about the benefits of using halos in a data parallel language like High Performance Fortran (HPF). The results presented in this paper show how dramatically the performance can be improved by shadow edges for unstructured computations when some kind of locality is given. The performance is nearly the same as for hand written message passing programs where the same strategy is used. The use of halos has also been verified for a large computational fluid dynamics code [7] and for a code dealing with electromagnetic problems [5].

But the use of halos remains a problem. The automatic use in an HPF compiler works only in certain situations. Especially the reuse of the communication schedule associated with a halo is difficult as the compiler has to make worst case assumptions if it is not able to track modifications of the indirection arrays.

A library approach gives explicit control to the user but requires the same knowledge about the data decomposition and its halos as it is necessary in a message passing version. But the library frees the user from writing difficult and error prone code to calculate shadow sizes and communication schedules. Providing a similar support in the HPF language directly is possible but has the same quality. Both, the language and the library approach lack from a standardization.

The public domain HPF compilation system ADAPTOR provides now library support for using halos and it is also able to use halos automatically. For the automatic use, the user only has to add the `DYNAMIC` directive for the halo array to allow the reallocation with shadow edges and the `TRACE` directive for the halo index array to help the compiler to track modifications at runtime and so to give the possibility of reusing the halo communication schedule.

Future work will consider a more convenient use of the halo library and improvements for the automatic use of halos.

Acknowledgements

I thank Falk Zimmermann (NEC Europe) and Guy Lonsdale (NEC Europe) for the many discussions about the FEM kernel. The very fruitful discussions with Siegfried Benkner (University of Vienna) gave insights about the implementation and problems of halos in the Vienna Fortran Compiler.

References

[1] ADAPTOR. High Performance Fortran Compilation System. WWW documentation, Institute for Algorithms and Scientific Computing (SCAI, GMD), 1999. `http://www.gmd.de/SCAI/lab/adaptor`.

[2] S. Benkner. HPF+: High Performance Fortran for advanced scientific and engineering applications. *Future Generation Computer Systems*, 15:381–391, 1999.

[3] S. Benkner. Optimizing Irregular HPF Applications Using Halos. In J. e. a. Rolim, editor, *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 1015–1024, San Juan, Puerto Rico, USA, Apr. 1999. Springer.

[4] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7:67–81, 1999.

[5] T. Brandes, E. Cagniot, J. Dekeyser, P. Boulet, Y. Le Menach, G. Marques, S. Clénet, and F. Piriou. Parallelization of a Fortran 90 Program for Electomagnetic Problems with High Performance Fortran. In *Proceedings of 3rd HPF Users Group Meeting, Redondo Beach, CA*, Aug. 1999.

[6] T. Brandes and C. Germain. A Tracing Protocol for Optimizing Data Parallel Irregular Computations. In D. Pritchard and J. Reeve (Eds.), editor, *Euro-Par'98 Parallel Processing, Southampton*, pages 629–638. Lecture Notes in Computer Science (1470), Sept. 1998.

[7] T. Brandes, F. Zimmermann, C. Borel, and M. Brédif. Evaluation of High Performance Fortran for an Industrial Computational Fluid Dynamics Code. In *Proceedings of VECPAR 98, Porto, Portugal*, pages 467–480. Faculdade de Engenharia da Universidade do Porto, June 1998.

[8] H. Gerndt and H. Zima. Optimizing Communication in Superb. In H. Burkhart, editor, *CONPAR 90 - VAPP IV*, Lecture Notes in Computer Science, pages 300–311. Springer-Verlag, 1990.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, Jan. 1997.

[10] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[11] The HPF+ Constortium. Report on Evaluation Results with Benchmark Programs Vers. 1. HPF+ Deliverable D1.2a, AVL, ECMWF, ESI, NAS, University of Pavia, University of Vienna, Dec. 1996.

[12] C. Walshaw, M. Cross, S. Johnson, and M. Everett. JOSTLE: Partitioning of Unstructured Meshes for Massively Parallel Machines. In N. S. et al., editor, *Parallel Computational Fluid Dynamics: New Algorithms and Applications*. Elsevier, Amsterdam, 1995.